Toward Predicting Architectural Significance of Implementation Issues

Arman Shahbazian, Daye Nam, Nenad Medvidovic

University of Southern California, Los Angeles, CA, USA
{armansha, dayenam, neno}@usc.edu

ABSTRACT

In a software system's development lifecycle, engineers make numerous design decisions that subsequently cause architectural change in the system. Previous studies have shown that, more often than not, these architectural changes are unintentional by-products of continual software maintenance tasks. The result of inadvertent architectural changes is accumulation of technical debt and deterioration of software quality. Despite their important implications, there is a relative shortage of techniques, tools, and empirical studies pertaining to architectural design decisions. In this paper, we take a step toward addressing that scarcity by using the information in the issue and code repositories of open-source software systems to investigate the cause and frequency of such architectural design decisions. Furthermore, building on these results, we develop a predictive model that is able to identify the architectural significance of newly submitted issues, thereby helping engineers to prevent the adverse effects of architectural decay. The results of this study are based on the analysis of 21,062 issues affecting 301 versions of 5 large open-source systems for which the code changes and issues were publicly accessible.

ACM Reference Format:

Arman Shahbazian, Daye Nam, Nenad Medvidovic . 2018. Toward Predicting Architectural Significance of Implementation Issues. In *MSR '18: MSR '18: 15th International Conference on Mining Software Repositories , May 28–29, 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, Article 4, 5 pages. https://doi.org/10.1145/3196398.3196440

1 INTRODUCTION

In a software development project, numerous design decisions are made that change the architecture of the system and directly affect the system's quality [33]. Prior work has shown that these architectural changes are frequently unintentional by-products of continual software maintenance tasks [40]. The result of inadvertent architectural changes is accumulation of technical debt and deterioration of software quality [16, 41]. A large body of work has focused on reducing the amount of technical debt and ameliorating the downsides of this phenomenon [24]. However, architectural decay is still evident during the evolution of many, if not most, software systems [5, 17]. In some situations, engineers deliberately *decide* to use solutions that are suboptimal to meet certain objectives, such as shorter time to market or other business requirements [7]. While the reasons for intentional accumulation of technical debt and architectural decay are still open for investigation, our focus in this work is developing techniques that identify and predict architectural changes, thus reducing the adverse effects of architectural decay.

In this paper, we aim (1) to automatically identify the issues leading to *architectural design decisions* in existing systems and (2) to predict the probable impact of implementation-level issues and resulting system changes on those decisions. Despite their far-reaching implications [33], there is a relative shortage of techniques, tools, and empirical studies pertaining to the role that architectural design decisions play in long-lived systems. Our work takes a step toward addressing that scarcity. We present an approach that uses the readily available information in the issue and code repositories of software systems and enables engineers to investigate the causes and frequencies of making new and modifying existing architectural design decisions.

To that end, we mine the issue and code repositories of five large open-source software systems to extract their issues and pertinent code changes. In order to detect the architectural changes in a system, we use a state-of-the-art software architecture analysis workbench [6]. For each issue, we recover the architecture of the system before and after its resolution. We use *a2a* [6], a metric specifically designed for measuring architectural change, to identify the issues causing architectural changes. We call these issues *architecturally significant*. We have shown previously that, *a2a* can be used to accurately recover architectural design decisions [30]. Unlike existing techniques that typically focus on undoing the side-effects of architectural decay ex post facto (e.g., [31, 37]), building on our results, we develop a predictive model that is able to identify the architectural significance of newly submitted issues. Doing so enables engineers to prevent architectural decay before the offending code changes are committed and merged with the system's code-base.

The main contributions of this paper are as follows:

- A technique for automatically detecting existing issues that are architecturally significant.
- A reusable dataset of 21,062 issues identified across five large open-source software systems that are labeled by their architectural significance.
- A classifier for predicting the architectural significance of newly submitted issues.

To support researchers in evaluating related approaches and replicating our experiments, we make our dataset available online.[1]

MSR '18, May 28–29, 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5716-6/18/05…$15.00
https://doi.org/10.1145/3196398.3196440

[1] The supplementary website can be reached at: https://softarch.usc.edu/predictar

The remainder of this paper is organized as follows. Section 2 overviews the foundational work enabling this research. Section 3 describes our experimental setup and results for identifying and predicting architecturally significant issues. Sections 4 and 5 discuss the related work and conclude the paper.

2 BACKGROUND

In our study, we rely on a software system's architecture and the means to recover it, as well as tracking of implementation issues. We overview each of these topics briefly.

2.1 Software Architecture

The structure of a software system's architecture is a graph whose vertices represent system components, while the edges denote their interconnections (e.g., call dependencies and logical couplings). Each component encompasses implementation entities such as classes, interfaces, methods, libraries, etc. The architecture of a software system is rarely accurately documented or kept up-to-date during the system's lifecycle. We use recovery techniques to obtain a system's architecture from its implementation.

Researchers have developed a range of architecture recovery techniques. In this work, we use *Algorithm for Comprehension-Driven Clustering* (ACDC) [35] and *Architecture Recovery using Concerns* (ARC) [14], as they have been shown to exhibit better scalability and accuracy than competing techniques [13]. Our study could be repeated using other recovery techniques. ACDC is oriented toward program comprehension and is based on subsystem patterns. ARC, on the other hand, produces components that are semantically related and share similar system level concerns (e.g., a component whose main concern is handling network communications). In order to detect architectural differences across different points in the development history of a software system, we use *a2a*, a similarity metric calculated based on the cost of transforming one architecture to another (e.g., by adding or removing components) [6].

2.2 Issue Tracking

All of our subject systems in this study use Jira [2] as their issue repository. A similar approach can be applied to systems using different repositories. When reporting implementation issues, engineers typically categorize them into different types: *bug*, *new feature*, *feature improvement*, *task* to be performed, etc. Engineers also assign a priority value to an issue, to denotes their perception of the issue's importance (e.g., critical or minor). Each issue has a status that indicates the position of the issue in its lifecycle [3]: issues start as "open", progress to "resolved", and finally to "closed".

In our study, we focus on issues that have been "resolved" or "closed". Issues can be resolved in different ways. We ignore the issues that fall under the "won't fix", or "cannot reproduce" categories and only focus on issues that have been "fixed". The reason is that these issues have been verified and addressed by developers, so that any effects caused by them would presumably appear in certain system versions and disappear once the issue is addressed. Additionally, a fixed issue contains information that is useful for our study: (1) *description*, (2) *affected versions* in which the issue has been found, (3) *type* of issue, (4) *priority*, and (5) *fixing commits*, i.e., the changes applied to the system to resolve the issue. Finding fixing commits is not always easy since there is no standard method for engineers to keep track of this information in issues trackers. In Jira, we found three popular methods: (1) directly mapping to fixing commits, (2) using pull requests, and (3) using patch files. Our approach supports all three methods.

System	Domain	Versions	Issues	Avg. LOC
Hadoop	Data Proc. Framework	68	7374	1.96M
Nutch	Web Crawler	21	1524	118K
Wicket	Web App Framework	72	4637	332K
CXF	Service Framework	120	5852	915K
OpenJPA	Java Persistence API	20	1675	511K

Table 1: Subject systems analyzed in our study.

3 STUDY SETUP

The goal of this work is to detect the implementation issues that lead to—possibly unintentional—design decisions and subsequently change the system's architecture. Furthermore, we want to expand these results by predicting whether resolving a submitted issue would require architecturally significant changes. This information could prove valuable to engineers and help them deliver higher quality code [25]. In the remainder of this section, we will summarize our subject systems (Section 3.1), and describe the devised workflow for determining the architectural significance of issues followed by a discussion of the results (Section 3.2). In Section 3.3, we will introduce a machine learning approach for predicting architectural significance of issues. Finally, Section 3.4 will overview the threats to the validity of our results.

3.1 Subject Systems

We report the empirical results involving five Apache [1] open-source projects. Apache was chosen because it is one of the largest open-source organizations in the world and has produced a number of impactful systems. Furthermore, Apache systems have well-maintained code repositories, release notes, and issue trackers. Table 1 lists our subject systems. We analyzed the largest available Apache systems that rely on Jira [2] for tracking issues and satisfy the following criteria:

(1) The systems belong to different software domains, to ensure broad applicability of our results.
(2) The issues and their fixing commits are tracked. Specifically, we analyze "resolved" and "closed" issues because they have complete sets of fixing commits.
(3) The systems have large numbers of resolved and closed issues to give us sufficient data points for our analysis and machine learning models.

3.2 Recovering Significant Issues

Figure 1 depicts our framework for identifying a system's architecturally significant issues. The process begins by mining the set of issues from our subject systems' issue repositories and filtering the ones not conforming to our criteria (recall Section 2.2). On average, about 35% of the issues are discarded at this stage. For each issue, our framework automatically extracts its pertinent commit information. The commit information is used to identify the system version at which the issue has been merged with the code base and

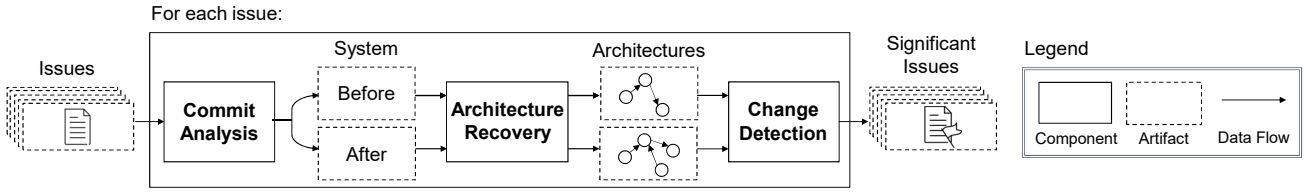

Figure 1: Framework for the identification of architecturally significant issues in a software project.

	Overall							ARC							ACDC						
System ↓	**#**	**B %**	**F/I %**	**O %**	**C %**	**Mj %**	**Mn %**	**#**	**B %**	**F/I %**	**O %**	**C %**	**Mj %**	**Mn %**	**#**	**B %**	**F/I %**	**O %**	**C %**	**Mj %**	**Mn %**
Hadoop	7374	56.6	32.2	11.2	15.2	61.4	23.4	1066	61.5	36.6	1.9	26.6	60.2	13.2	633	58.9	38.6	2.5	27.8	59.1	13.1
Nutch	1524	45.7	43.8	10.6	7.0	52.2	40.8	89	42.7	57.3	0.0	4.5	64.0	31.5	60	38.3	61.7	0.0	1.7	66.7	31.7
Wicket	4637	59.7	33.5	6.8	2.4	59.5	38.1	1362	57.9	37.0	5.1	1.5	59.1	39.4	930	60.3	35.8	3.9	2.0	62.8	35.2
Cxf	5852	62.3	26.0	11.7	3.4	77.7	18.9	2441	64.0	31.7	4.3	3.7	76.5	19.8	1403	57.0	37.1	5.9	3.4	77.7	19.0
OpenJPA	1675	62.2	21.6	16.2	5.3	71.1	23.6	580	58.6	25.5	15.9	4.7	77.8	17.6	453	56.3	26.3	17.4	4.2	79.7	16.1

Table 2: Overview of the results of our architectural significance analysis. Column *Overall* shows the issues' general distribution. *ARC* and *ADCD* columns depict the distribution of architecturally significant issues under each recovery technique. Issue types are bug (*B*), feature/improvement (*F/I*), or other (*O*). Issue priority is critical (*C*), major (*Mj*), or minor (*Mn*)

the version immediately preceding it. We then extract the system's source-code at these two snapshots. By applying the two selected architecture recovery techniques (ACDC [35] and ARC [14]), we recover two architectural views of the system at each snapshot. Finally, the *a2a* [6] similarity metric is used, with the highest sensitivity, to identify any architectural discrepancies stemming from the issue and its extracted commits. Issues whose resolution has caused architectural change, as indicated by *a2a*, are labeled as significant.

Table 2 displays the results of running the above analysis on our subject systems. The *Overall* column contains the general distribution of issues in each system. The other top level columns contain information about the architecturally significant issues detected using ACDC and ARC. The data is further subdivided based on the issue type and priority. Issue type can be bug (*B*), feature or improvement (*F/I*), or other types such as test and task (*O*). Issue priority is either critical (*C*), major (*Mj*), or minor (*Mn*).

The data shows that, in general, there are more *bug*s submitted to issue repositories than *features* or *improvements*. Interestingly, although the number of architecturally significant issues is only a small fraction of all submitted issues, their distribution in terms of priority is not very different from the original issue distribution. While this finding deserves a closer inspection, it appears that engineers are typically unable to isolate architecturally significant issues, and consequently do not consider them any more (or less) important than "regular" issues. The distribution of bugs, features, and improvements does not show drastic change between significant and regular issues either. This is a finding we have not seen in literature previously. Overall, our results suggest that architectural significance is an overlooked facet of implementation issues, and cannot be easily inferred from the existing tags applied to issues.

3.3 Predicting Significant Issues

The main objective of our work is to enable classifying issues based on their architectural significance. Recent studies have shown that developers who explicitly consider the impact of their code-level changes on their system's architecture deliver higher quality code [25]. This suggests that notifying engineers of the likely architectural importance of issues at the time they are submitted can result in better-informed implementation decisions. To enable such notification, we use the information that is readily available for newly submitted issues: title, description, priority, and type.

Figure 2 displays the overview of our classification-model building process. Most of the information in issues is textual. Therefore, to construct the automatic classification model, first, we need to parse and pre-process the textual sections of the retrieved issues, i.e., title and description. This step has a big impact on optimizing the classification step and data-noise removal [36]. For each issue, we remove the English stop-words [27], code snippets, and stack traces that are sometimes submitted alongside issues descriptions. We then use the Porter stemmer to reduce the inflected (or sometimes derived) words to their word stems [26]. This helps to group the words with similar basic meanings together.

To perform machine learning on text documents, we must transform the textual content into numerical feature vectors. The most intuitive way to do so is to use the *bags-of-words* representation. In this approach, we create a dictionary of all the words in the corpora. Then, for each issue we count the occurrences of the words in the dictionary. Using this approach, issues with longer descriptions will have higher average count values than shorter documents, even though they may pertain to the same topic. To address this problem, we use *Term Frequencies* (TF) [20], which normalizes the number of occurrences with respect the total number of words in the issue description and title. Finally, we append the one-hot-encoded [21] representation of issue type and priority to this feature vector.

Previous studies have shown that when training models on primarily textual data with on the order of several thousand data points, a

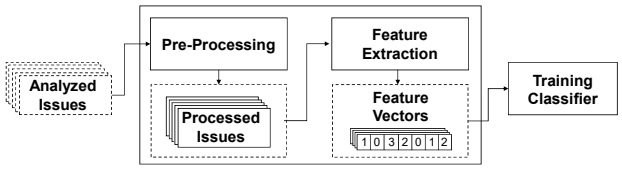

Figure 2: Workflow for building our automatic architectural significance classifier.

	ARC		ACDC	
System ↓	**Precision**	**Recall**	**Precision**	**Recall**
Hadoop	0.793	0.637	0.883	0.547
Nutch	0.941	0.276	0.951	0.217
Wicket	0.843	0.657	0.678	0.417
Cxf	0.801	0.698	0.928	0.468
OpenJpa	0.965	0.503	0.903	0.399
Cross-Project	0.816	0.592	0.806	0.573

Table 3: Precision and recall of our classifier for each system. The Cross-Project row shows the result of applying the classifier on the combined issues of all systems.

classifier with high bias performs well. Theoretical and empirical results suggest that Naive Bayes [12] does well in such circumstances [11, 22], and we adopt it for our classification.

We use precision and recall as the performance evaluation metrics. Precision shows the ratio of correctly predicted architecturally significant issues over all predicted significant issues. Recall denotes the ratio of correctly predicted architecturally significant issues over all of the actually significant issues. Table 3 shows the evaluation results obtained using the 10-fold-cross-validation setup, where each dataset is randomly partitioned into 10 equal-sized subsets. Nine of the subsets are used as training data, while the last subset is retained as testing data. The process is then repeated 10 times.

The numbers reported in Table 3 are the average values across the 10 repetitions. The top five rows show the results of training and running our classifier on the individual systems; the *Cross-Project* row is the result of applying the classifier on our entire corpus of issues across all systems. The precision of our classifier is very good, surpassing 90% in certain cases under both ARC and ACDC. The overall precision across the five systems is above 80% under both recovery techniques. The recall values for our classifier are lower than the precision values. In the case of ARC, with the exception of Nutch, they are all above 50% for the individual systems, and around 60% across all systems. In the case of ACDC, the recall values are lower: only the Hadoop and Cross-Project values are above 50%. The reason may lie in ACDC's dependency analysis-based architecture recovery algorithm, a hypothesis we will have to evaluate further. Nutch is again a notable outlier. OpenJpa yields the second-lowest recall values under both ARC and ACDC. These two systems have much smaller datasets of issues compared to the rest of our corpus. Moreover, Nutch has about an order of magnitude fewer architecturally significant issues than other systems, which further hampers the efficacy of our classification model.

3.4 Threats to Validity

Threats to Construct Validity: The dataset containing architecturally significant issues depends on the recovery techniques employed. To mitigate this problem, we selected two techniques that exhibit higher accuracy than their competitors [13]. Furthermore, any technique can be easily incorporated in our framework.

Threats to External Validity: Due to practical limitations, we only used open-source projects. Furthermore, all the issues in our study belong to systems implemented in Java and use the Jira issue tracker. To help mitigate this issue, we selected systems from different domains, thus expanding our study's scope (recall Section 3.1).

4 RELATED WORK

Understanding architectural decisions is important for software maintenance and comprehension. A number of studies have been conducted to justify its necessity and show its concrete benefits. Falessi et al. argued for the value of capturing and explicitly documenting design decisions [10]. Burge et al. showed that understanding architectural decision helps make better decisions [8]. Tang et al. empirically showed that knowing the architectural decisions can improve the quality of software systems [32].

Some researchers focused on the importance of having architectural awareness, i.e., an ability to understand and assess varied aspects of the software architecture and architectural decisions. Tyree et al. argued that architectural decisions are usually not documented, and this impedes software architects to understand and make architectural decisions [34]. Nowak et al. also suggested that architectural awareness can enhance the efficiency and quality of the architecture design process, and they proposed a methodology to manually capture the architectural knowledge in the design process [23].

Applying machine learning and natural language processing techniques for software comprehension and maintenance has been receiving increasing attention from the research community. Especially, issue data extracted from software repositories is widely used in that it contains important information related to bug and software quality [9]. Antoniol et al. built a classifier using machine learning techniques to classify issues into two classes: *bugs* and *non bugs* [4]. Wiese et al. used issues as contextual data to improve the co-change prediction model, i.e., a model to help developers aware of artifacts that will change together with the artifact they are working on, of software systems [39]. Weiss et al. employed a nearest neighbors technique to automatically predict the fixing efforts of issues to facilitate issue assignment and maintenance scheduling [38].

5 CONCLUSION AND FUTURE WORK

In this paper, we described a method for automatically detecting architecturally significant issues, and classifying them based on the textual and non-textual information contained in each issue. Our technique aims to raise architectural awareness thus helping engineers deliver higher quality code based on well-informed decisions. Our study was conducted on five large open-source software projects. Using our automated detection technique, we analyzed 21,062 issues and identified their architectural significance. Our results suggest that current categorizations of issues (type and priority) do not effectively encompass architectural significance. Expanding on these resulting, we built a classification model to predict the architectural significance of newly submitted issues.

For our future work, we plan to expand our study to more systems by adding the support for other issue trackers. We also plan to improve the performance of our classification model by adapting recent advances in the field of generative adversarial nets, which in theory can enable us to artificially augment the size of our dataset [42]. Finally, we aim to explore the feasibility of a similar technique to predict the non-functional effects of implementation issues (e.g., security and reliability) in existing software systems [15, 18, 19, 28, 29].

REFERENCES

[1] 2017. Apache Software Foundation, http://apache.org. (2017). http://apache.org/

[2] 2017. Jira, https://www.atlassian.com/software/jira. (2017). https://www.atlassian.com/software/jira

[3] 2017. What is an issue. (2017). https://confluence.atlassian.com/jira064/what-is-an-issue-720416138.html/

[4] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. 2008. Is It a Bug or an Enhancement?: A Text-based Approach to Classify Change Requests. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*. ACM, 23:304–23:318.

[5] P. Behnamghader, R. Alfayez, K. Srisopha, and B. Boehm. 2017. Towards Better Understanding of Software Quality Evolution through Commit-Impact Analysis. In *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. 251–262.

[6] Pooyan Behnamghader, Duc Minh Le, Joshua Garcia, Daniel Link, Arman Shahbazian, and Nenad Medvidovic. 2016. A large-scale study of architectural evolution in open-source software systems. *Empirical Software Engineering* (2016), 1–48.

[7] Nanette Brown, Yuanfang Cai, Yuepu Guo, Rick Kazman, Miryung Kim, Philippe Kruchten, Erin Lim, Alan MacCormack, Robert Nord, Ipek Ozkaya, et al. 2010. Managing technical debt in software-reliant systems. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*. ACM, 47–52.

[8] Janet E Burge. 2008. Design rationale: Researching under uncertainty. *Artificial Intelligence for Engineering Design, Analysis and Manufacturing* 22, 04 (2008), 311–324.

[9] Yguaratã Cerqueira Cavalcanti, Paulo Anselmo Mota Silveira Neto, Ivan do Carmo Machado, Tassio Ferreira Vale, Eduardo Santana Almeida, and Silvio Romero de Lemos Meira. 2014. Challenges and opportunities for software change request repositories: a systematic mapping study. *Journal of Software: Evolution and Process* 26, 7 (2014), 620–653.

[10] Davide Falessi, Lionel C Briand, Giovanni Cantone, Rafael Capilla, and Philippe Kruchten. 2013. The value of design rationale information. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22, 3 (2013), 21.

[11] George Forman and Ira Cohen. 2004. Learning from little: Comparison of classifiers given little training. In *European Conference on Principles of Data Mining and Knowledge Discovery*. Springer, 161–172.

[12] Nir Friedman, Dan Geiger, and Moises Goldszmidt. 1997. Bayesian Network Classifiers. *Mach. Learn.* 29, 2-3 (1997), 131–163.

[13] Joshua Garcia, Igor Ivkovic, and Nenad Medvidovic. 2013. A comparative analysis of software architecture recovery techniques. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*. IEEE, 486–496.

[14] Joshua Garcia, Daniel Popescu, Chris Mattmann, Nenad Medvidovic, and Yuanfang Cai. 2011. Enhancing architectural recovery using concerns. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 552–555.

[15] Michael Langhammer, Arman Shahbazian, Nenad Medvidovic, and Ralf H Reussner. 2016. Automated Extraction of Rich Software Models from Limited System Information. In *2016 13th Working IEEE/IFIP Conference on Software Architecture (WICSA)*. 99–108. https://doi.org/10.1109/WICSA.2016.35

[16] Duc Le, Daniel Link, Arman Shahbazian, and Nenad Medvidovic. 2018. An Empirical Study of Architectural Decay in Open-Source Software. In *IEEE International Conference on Software Architecture (ICSA)*. IEEE.

[17] Duc Minh Le, Pooyan Behnamghader, Joshua Garcia, Daniel Link, Arman Shahbazian, and Nenad Medvidovic. 2015. An Empirical Study of Architectural Change in Open-Source Software Systems. In *12th IEEE Working Conference on Mining Software Repositories*. 235–245.

[18] Youn Kyu Lee, Jae Young Bang, Gholamreza Safi, Arman Shahbazian, Yixue Zhao, and Nenad Medvidovic. 2017. A SEALANT for Inter-App Security Holes in Android. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 312–323. https://doi.org/10.1109/ICSE.2017.36

[19] Youn Kyu Lee, Peera Yoodee, Arman Shahbazian, Daye Nam, and Nenad Medvidovic. 2017. SEALANT: A Detection and Visualization Tool for Inter-app Security Vulnerabilities in Android. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2017)*. IEEE Press, Piscataway, NJ, USA, 883–888. http://dl.acm.org/citation.cfm?id=3155562.3155672

[20] H. P. Luhn. 1957. A Statistical Approach to Mechanized Encoding and Searching of Literary Information. *IBM Journal of Research and Development* 1, 4 (Oct 1957), 309–317.

[21] Kevin P. Murphy. 2012. *Machine Learning: A Probabilistic Perspective*. The MIT Press.

[22] Andrew Y Ng and Michael I Jordan. 2002. On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes. In *Advances in neural information processing systems*. 841–848.

[23] Marcin Nowak and Cesare Pautasso. 2013. Team Situational Awareness and Architectural Decision Making with the Software Architecture Warehouse. In *Software Architecture*. Springer Berlin Heidelberg, 146–161.

[24] Matheus Paixao, Mark Harman, Yuanyuan Zhang, and Yijun Yu. 2017. An empirical study of cohesion and coupling: Balancing optimisation and disruption. *IEEE Transactions on Evolutionary Computation* (2017).

[25] Matheus Paixao, Jens Krinke, DongGyun Han, Chaiyong Ragkhitwetsagul, and Mark Harman. 2017. Are developers aware of the architectural impact of their changes?. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 95–105.

[26] M. F. Porter. 1997. Readings in Information Retrieval. Morgan Kaufmann Publishers Inc., Chapter An Algorithm for Suffix Stripping, 313–316.

[27] C. J. Van Rijsbergen. 1979. *Information Retrieval* (2nd ed.). Butterworth-Heinemann.

[28] Gholamreza Safi, Arman Shahbazian, William G. J. Halfond, and Nenad Medvidovic. 2015. Detecting Event Anomalies in Event-based Systems. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015)*. ACM, New York, NY, USA, 25–37. https://doi.org/10.1145/2786805.2786836

[29] Arman Shahbazian, George Edwards, and Nenad Medvidovic. 2016. An End-to-end Domain Specific Modeling and Analysis Platform. In *Proceedings of the 8th International Workshop on Modeling in Software Engineering (MiSE '16)*. ACM, New York, NY, USA, 8–12. https://doi.org/10.1145/2896982.2896994

[30] Arman Shahbazian, Youn Kyu Lee, Duc Le, Yuriy Brun, and Nenad Medvidovic. 2018. Recovering Architectural Design Decisions. In *IEEE International Conference on Software Architecture (ICSA)*. IEEE.

[31] L. Tahvildari, R. Gregory, and K. Kontogiannis. 1999. An approach for measuring software evolution using source code features. In *Software Engineering Conference, 1999. (APSEC '99) Proceedings. Sixth Asia Pacific*. 10–17.

[32] Antony Tang, Minh H Tran, Jun Han, and Hans Van Vliet. 2008. Design reasoning improves software design quality. In *International Conference on the Quality of Software Architectures*. Springer, 28–42.

[33] Richard N Taylor, Nenad Medvidovic, and Eric M Dashofy. 2009. Software architecture: foundations, theory, and practice. (2009).

[34] Jeff Tyree and Art Akerman. 2005. Architecture decisions: Demystifying architecture. *IEEE software* 22, 2 (2005), 19–27.

[35] Vassilios Tzerpos and Richard C Holt. 2000. ACDC: An Algorithm for Comprehension-Driven Clustering.. In *wcre*. 258–267.

[36] Alper Kursat Uysal and Serkan Gunal. 2014. The Impact of Preprocessing on Text Classification. *Inf. Process. Manage.* 50, 1 (2014), 104–112.

[37] Christopher Van der Westhuizen and André Van Der Hoek. 2002. Understanding and propagating architectural changes. In *Software Architecture*. Springer, 95–109.

[38] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. 2007. How Long Will It Take to Fix This Bug?. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*. IEEE Computer Society, 1–1.

[39] Igor Scaliante Wiese, Reginaldo R, Igor Steinmacher, Rodrigo Takashi Kuroda, Gustavo Ansaldi Oliva, Christoph Treude, and Marco Aurlio Gerosa. 2017. Using Contextual Information to Predict Co-changes. *J. Syst. Softw.* 128, C (2017), 220–235.

[40] Byron J Williams and Jeffrey C Carver. 2010. Characterizing software architecture changes: A systematic review. *Information and Software Technology* (2010), 31–51.

[41] Lu Xiao, Yuanfang Cai, Rick Kazman, Ran Mo, and Qiong Feng. 2016. Identifying and quantifying architectural debt. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 488–498.

[42] Lantao Yu, Weinan Zhang, Jun Wang, and Yong Yu. 2017. SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient.. In *AAAI*. 2852–2858.