Supporting Software Architecture Maintenance by Providing Task-specific Recommendations

Matthias Galster
University of Canterbury
New Zealand
mgalster@ieee.org

Christoph Treude
University of Adelaide
Australia
christoph.treude@adelaide.edu.au

Kelly Blincoe
University of Auckland
New Zealand
k.blincoe@auckland.ac.nz

***Abstract*—During software maintenance, developers have different information needs (e.g., to understand what type of maintenance activity to perform, the impact of a maintenance activity and its effort). However, information to support developers may be distributed across various sources. Furthermore, information captured in formal architecture documentation may be outdated. In this paper, we put forward a late breaking idea and outline a solution to improve the productivity of developers by providing task-specific recommendations based on concrete information needs that arise during software maintenance.**

***Index Terms*—software maintenance, software architecture, natural language processing, text classification**

I. Introduction

Expertise and skills to maintain high quality software systems grow with experience. It is generally accepted that *experienced* developers are *better* developers [1]. However, skills and expertise required of those who maintain software constantly evolve [2], e.g., the half-life of technology-related software engineering knowledge is around five years [3]. Also, those who develop and maintain software often have only limited training in software development [4]. In this paper, we put forward a *late breaking idea* to develop new methods (and tools) to improve the productivity of software developers by providing task-specific recommendations based on concrete information needs during software maintenance. In particular, we propose the use of knowledge extraction and classification approaches to mine various types of software development data and data sources.

II. Problem Description

The overall problem our work aims to address is the lack of information available to developers to make informed maintenance decisions and to perform maintenance tasks. In detail, our work aims at addressing the following problems:

- **Insufficient information about maintenance activities:** Poor understanding of the concrete tasks that are required to complete maintenance activities (including the impact of maintenance activities) and related decisions make maintenance difficult. For example, applying "extract method" refactoring may be suitable for long methods that are logically structured, but is more difficult for long and poorly structured methods which implement a very specific and unique business function. Furthermore, applying "extract method" refactoring may contradict a conscious design decision made by the "original" developers (e.g., to reduce coupling at the cost of low modularity). Developers often learn these trade-offs through experience as software maintenance is not typically taught in software engineering degrees.
- **Poor understanding of architecture change:** Uninformed maintenance activities lead to increasing architecture complexity, accumulate architecture debt and decay, and affect quality attributes such as understandability, reliability, etc. Therefore, software engineers need to understand *where* major architecture changes occur (at different levels of abstraction, such as component and/or system level). Also, software engineers need to understand *how much* (i.e., to which extent and when) architecture changes happen over time at both levels.

Some practical consequences of the above problems include difficulties in budgeting and resource allocation (e.g., based on the maintenance effort), staffing (e.g., based on required expertise) and planning (e.g., when prioritizing maintenance activities and when handling maintenance activities that are performed across iterations in agile environments).

III. Motivating Example

Consider a developer who is working on adding a new preference to the open source reference management software JabRef,[1] e.g., for configuring default file names. Since preferences in JabRef can affect the behaviour of several components, such a task is architecture-relevant, and information on how to address it exists in different sources and formats:

- A **GitHub issue** outlining the desired architecture of JabRef[2] contains a diagram showing the overall architecture and *informal guidelines* on how preferences should be implemented: "If only one or two methods are touched, I would pass the parameter, but if this would touch a lot more methods, we should use the observer pattern instead." Natural language processing can be used to extract this relevant information.

[1] https://github.com/JabRef/jabref
[2] https://github.com/JabRef/jabref/issues/1579

- The existing JabRef **source code** contains previously implemented preferences. Applying static or dynamic analysis [5] would reveal how these have been implemented, and, in particular, whether they followed the observer pattern.
- Architectural knowledge, such as how to properly implement design patterns, has been described in formal and informal documentation. In the case of the observer pattern, **textbook** knowledge such as the Gang-of-Four book [6] would be relevant.
- At the same time, many **informal documentation** sources, such as blog posts [7], discuss *opinions and experiences* related to architectural designs. For the observer pattern, Neill Morgan's blog post on pros and cons of the observer pattern[3] could serve as a source for developer recommendations.

It would take considerable time to find all of the relevant information when performing a maintenance task like this one and some relevant information is likely to be missed.

IV. Proposed Solution

We propose the use of automated tools that can assist developers during software maintenance activities. Our envisioned solution uses *natural language processing techniques* to filter and combine architecture-related information across multiple sources, some of which are *general* sources of information (e.g., online discussions or version control/issue tracker data of publicly available open source systems), while others are *specific* to the system under maintenance (e.g., version control data and documentation of a particular system). The novelty of our proposed work is as follows:

- We will focus on *architecture-relevant* information. This is because maintenance at the code level has been studied in the past, e.g., in the context of removing technical debt and code smells [8]. Maintenance of systems at the architecture level on the other hand is less understood, yet places significant challenges in practice (see for example the growing interest in *architectural* technical debt [9]). As argued by Martini and Bosch, sub-optimal architecture decisions can lead to immature architectural artifacts and compromised quality attributes [10]. Also, architectural information is often recorded by more informal means (e.g., technical or personal notes, emails, wikis, minutes) [11], and, even though high-level abstractions often remain useful even when the implementation details change, gets easily outdated due to its complexity [12] and no longer represents the implemented architecture.
- We will utilize *existing* software development data, such as software repositories and software development tools that developers commonly use (e.g., version control systems, bug trackers, online/public sites and communities). This removes the need to create and maintain separate documentation, and, therefore, reduces workload and outdated data. Prior research suggests that these sources do indeed provide reusable information for different activities in the context of software architecting (e.g., [13]).
- We will utilize *multiple* sources, rather than relying on one data source. Information from different sources can complement each other to provide more comprehensive information based on the information needs of developers during maintenance. Relevant information can often be found scattered throughout multiple forms of documentation and information sources. For example, we may combine commits and related information (e.g., who committed, and when), version control data and change logs, API references, wikis, project management data and discussions on Slack (e.g., how controversial a refactoring recommendation was in the past). Some sources can be general (e.g., discussions on Stack Overflow), while others may be specific to a system (e.g., an issue tracker).
- We will include *opinions and experience* of those who develop and maintain software. As research has shown, practitioners often base their decisions on the experience and opinions of others they trust rather than on empirical evidence [14], [15].

Considering the variety of sources that can contain useful information for maintenance tasks with content ranging from facts (e.g., extracted from source code) to guidelines and opinions, our proposed approach to identify and extract architecture-relevant knowledge is more complex than existing software-related knowledge extraction approaches, e.g., for explaining API types [16].

To build a system which can automatically issue recommendations to developers who are working on architecture-relevant tasks, at least three challenges need to be addressed:

- Identify architecture-relevant tasks. We will conceptualize developers' tasks as GitHub issues, and use similar issues and the changes made to address them to determine whether a task is architecture-relevant.
- Identify architecture-relevant information. We will address this challenge using text classification approaches with a seed corpus of architecture-relevant keywords. In the example provided in Section III, a design pattern such observer would be part of a keyword catalogue.
- Collect and aggregate relevant information. To issue recommendations to developers, information from various sources needs to be aggregated, considering its credibility [17]. We will employ multi-document summarization and knowledge graph approaches [18] towards this goal.

This will result in a set of architecture-relevant recommendations for each architecture-relevant maintenance task.

V. Conclusions

In this paper we outlined some pressing problems that occur during software maintenance and in particular related to planning and understanding architecture change. We also motivated and described our solution which relies on harvesting *existing* knowledge from *multiple* sources based on the concrete information needs of developers.

[3] https://neillmorgan.wordpress.com/2010/02/07/observer-pattern-pros-cons/

REFERENCES

[1] J. Hannay, E. Arisholm, H. Engvik, and D. Sjoberg, "Effects of personality on pair programming," *IEEE Transactions on Software Engineering*, vol. 36, pp. 61–80, 2010.

[2] R. Krishnamurthy, "Breezing my way as a solution architect: A retrospective on skill development and use," *IEEE Software*, vol. 34, pp. 9–13, 2017.

[3] P. Kruchten, "Lifelong learning for lifelong employment," *IEEE Software*, vol. 32, pp. 85–87, 2015.

[4] P. Antonino, A. Morgenstern, and T. Kuhn, "Embedded-software architects: It's not only about the software," *IEEE Software*, vol. 33, pp. 56–62, 2016.

[5] B. Cornelissen, A. Zaidman, A. Van Deursen, L. Moonen, and R. Koschke, "A systematic survey of program comprehension through dynamic analysis," *IEEE Transactions on Software Engineering*, vol. 35, pp. 684–702, 2009.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[7] C. Parnin, C. Treude, and M.-A. Storey, "Blogging developer knowledge: Motivations, challenges, and future directions," in *International Conference on Program Comprehension (ICPC)*, 2013, pp. 211–214.

[8] E. Ligu, A. Chatzigeorgiou, T. Chaikalis, and N. Ygeionomakis, "Identification of refused bequest code smells," in *IEEE International Conference on Software Maintenance (ICSM)*, 2013, pp. 392–395.

[9] T. Besker, A. Martini, and J. Bosch, "Managing architectural technical debt: A unified model and systematic literature review," *Journal of Systems and Software*, vol. 135, pp. 1–16, 2018.

[10] A. Martini and J. Bosch, "The danger of architectural technical debt: Contagious debt and vicious circles," in *Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2015, pp. 1–10.

[11] A. Tang, P. Liang, and H. V. Vliet, "Software architecture documentation: The road ahead," in *Working IEEE/IFIP Conference on Software Architecture (ICSA)*, 2011, pp. 252–255.

[12] T. C. Lethbridge, J. Singer, and A. Forward, "How software engineers use documentation: The state of the practice," *IEEE software*, vol. 20, pp. 35–39, 2003.

[13] M. Soliman, M. Galster, A. Salama, and M. Riebisch, "Architectural knowledge for technology decisions in developer communities: An exploratory study with stack overflow," in *Working IEEE/IFIP Conference on Software Architecture (WICSA)*, 2016, pp. 128–133.

[14] P. Devanbu, T. Zimmermann, and C. Bird, "Belief and evidence: How software engineers form their opinions," *IEEE Software*, vol. 35, pp. 72–76, 2018.

[15] ——, "Belief and evidence in empirical software engineering," in *IEEE/ACM International Conference on Software Engineering (ICSE)*, 2016, pp. 108–119.

[16] G. Petrosyan, M. P. Robillard, and R. De Mori, "Discovering information explaining API types using text classification," in *International Conference on Software Engineering (ICSE)*, 2015, pp. 869–879.

[17] A. Williams and A. Rainer, "How do empirical software engineering researchers assess the credibility of practitioner-generated blog posts?" in *Evaluation and Assessment on Software Engineering (EASE)*, 2019, pp. 211–220.

[18] H. Li, S. Li, J. Sun, Z. Xing, X. Peng, M. Liu, and X. Zhao, "Improving API caveats accessibility by mining API caveats knowledge graph," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2018, pp. 183–193.