Approaches to Software Maintainability and Software Maintainability Research

Literature and Method Review of Software Maintainability

I. Introduction

Software Maintenance (SM) is one of the most costly activities in software development, constituting 45-60% of lifecycle costs [1]. To reduce the cost of SM, software should be created in a way that facilitates SM tasks as much as possible. The ease with which software tasks are performed to modify, update or improve a system is referred to as maintainability [2]. By researching maintainability approaches, improved techniques may emerge that improve the lifecycle of software, reduce maintenance costs for organizations and ultimately lead to the creation of more reliable software.

The aim of this literature review is to identify how software can be designed in a way that supports maintainability and to investigate research methods that have been employed to study software maintainability. Relevant literature has been found using online databases and two maintainability paradigms have been identified and examined in detail. Five texts have been chosen to facilitate this investigation and will be reviewed in terms of their findings and research methods. These texts will be compared and possible areas for future research in the field of SM will be suggested.

II. Review Method

Literature was obtained through articles found on the Scopus database. The search terms were "Software Quality" and "Software Maintainability" and the results were limited to "Computer Science", "Articles", "Reviews" and "Conference Papers". Of the resulting articles, the abstracts were read and an affinity diagram was created and resulted in the identification of four key topic areas, *Approaches to Maintainability*, *Defining Maintainability*, *Predicting Maintenance Needs* and *Project Management*. Of these four topic areas the first, *Approaches to Maintainability,* was deemed to contain the most relevant articles to the review topic. The 14 articles within this category were read and two major approaches were identified, *Architectural Paradigms* and *Heuristic Paradigms*. Of these 14 articles, five have been selected for comprehensive analysis of study results and research methods.

III. Literature and Method Review

A. Architectural Paradigms

Architectural Paradigms for the sake of this review are approaches to software maintainability that involve high-level system-wide strategies. Topics in this category that will be discussed are *Model Driven Development* (MDD) and *Domain Specific Languages* (DSLs). MDD is an approach to software development that provides developers with a graphical user interface with which to arrange prefabricated software components [3]. These components can be used to generate programs quickly and have apparent benefits to maintainability due to its simplicity in terms of software design [4]. DSLs are simplified programming languages created for use by domain experts that allow them to easily understand and maintain programs without the assistance of developers. Additionally, DSLs also easily facilitate communication between developers should they be required [5, p. 81].

To test the maintainability of MD and DSL approaches against conventional Code-Centric (CC) methods, Atalag, Yang, Tempero and Warren [4] created a replica of an existing clinical application using a hybrid MD and DSL approach and compared the maintainability in terms of time spent on maintenance and resultant lines of code changed over a series of change requests. They found that the MDD/DSL paradigm required 1/22nd of the effort to maintain in terms of additions, deletions and modifications and half the time to complete change requests compared to the existing system performing the same maintenance tasks. This experiment used a quantitative Control-Group Time-Series design [7, p. 208] to capture observations across a control-group, in this case the existing CC system, and compared it to the experimental MD/DSL approach that was being maintained in parallel. A limitation of the study was the use of only one developer being assigned to each task [4, p. 857] which meant that the developers themselves couldn't be ruled out as potential variables. To remove the developers as variables, the researchers could have randomly assigned developers to the maintenance of each program. Another possible variable was that the existing CC system had been created fifteen years earlier and been subject to maintenance prior to the study resulting in what was possibly an already eroded system, a phenomenon in which a system becomes more complex and difficult to maintain as more maintenance is conducted [6]. To address this possibility and improve the internal validity of this study, the future work could include the development of a third application using conventional CC software development approaches to confirm that improvements in maintainability were indeed due to the MD/DSL approach as opposed to the existing system being inherently difficult to maintain. Their findings that MD/DSL approaches lead to more maintainable systems do however seem to be consistent with other research into MD approaches.

To test maintainability of a MD approach against conentional CC development, Martínez, Cachero, Matera, Abrahao and Luján [3] compared performance of maintenance tasks between two groups of undergraduate students. The experimental group had just finished a course unit in which they were taught an MD approach to web development and the control group had just finished a unit at a separate university in which a CC method to web development was taught. They found the students who had learned the MD approach outperformed students who had been taught CC development in a set of maintenance tasks on identical web applications that had been developed using either the CC or MD approach. This research method resembles a Static Group Comparison but with additional steps taken to ensure equivalence between the two groups in terms of age and relevant experience. The internal validity of the study could be improved by using a Pretest-Posttest Control-Group Design [7, p. 204] as well as randomizing the participants and performing pretest observations to remove uncontrolled variables among participants. The choice to employ convenience sampling by using undergraduate university students in separate universities and in different countries however, limited the researchers opportunities to implement the aforementioned control measures and threatened the external validity and generalizability of the study [3, p. 1890].

B. Heuristic Paradigms

Heuristic Paradigms are fine-grained approaches to maintainability that involve best-practices and techniques to ensure individual components of the system can be easily maintained. Approaches discussed in this category will be *Refactoring* and *Design Patterns*. Refactoring is the process of modifying a functioning program internally to improve readability, understandability and architectural quality without affecting the external functionality [6, p. 275]. Refactoring is recommended as a best practice [8, p. 110] and positive correlations were found between refactoring processes and maintainability [8..9] though there it may introduce tradeoffs in terms of overall system complexity.

Design Patterns are best-practices formed in-situ by software developers who have observed reusable solutions to common challenges in software design [6]. These solutions are adopted by other developers and used as a template to solve their own design challenges [6]. Researchers [6][9] have found that by implementing design patterns, maintainability can be increased, though, much like refactoring, they may also increase the underlying complexity of the system.

Hegedüs, Bán, Ferenc and Gyimóthy [9] carried out a study to determine the density of design patterns present in a system and the resultant maintainability of that system. By analyzing documentation changes that occurred over the development of a system and comparing its maintainability at each revision, they found a positive correlation between the presence of design patterns and maintainability. They also found, however, that while the system becomes more maintainable, the overall complexity of the system increases in parallel and reduces the understandability [9, p. 324], which would make the program less maintainable to developers who are not already familiar with the system. Their research design resembles a Simple Time-Series experiment [7, p. 208] in which changes in a single group are observed over time. Due to the nature of software development version control, this was able to be performed in an Ex Post Facto manner [7, p. 214] in which the independent variable had not been manipulated by the researchers. The inherent lack of control in the Ex Post Facto approach [7, p. 214] was compounded by the use of only a single experimental group, limiting the generalisability and making it difficult to suggest any more than an association between design patterns and maintainability. To improve this study, the researchers could have compared multiple systems in the same manner. Further research could address the impact of the increased complexity to developers by randomly allocating maintenance tasks at different thresholds of design pattern density.

Rochimah, Gautama and Akbar [6] carried out a correlational study on the impact of design patterns on the maintainability of an Academic Information System. The quantitative case study utilised a One-Group Pre-test Post-test design [7, p. 203] in which the system was assessed on a series of software quality metrics before refactoring based on design patterns. After each refactoring, the system was reassessed and a positive correlation was found between the design pattern refactoring and the maintainability of the system. The researchers identified the same tradeoff as Hegedüs et.al in terms of resulting complexity. A disadvantage of the post-test pre-test design is an inability to show cause-and-effect relationships, in this case due to only one experimental group. Therefore, this research serves as a basis for a hypothesis and future work should improve the use of multiple systems as in the next study.

Malhotra and Chug [8] carried out a similar study investigating the effects of refactoring maintainability of five distinct software systems. Using the same metrics for maintainability as used in [6], they first evaluated each system before successively treating each with a refactoring process carried out by 40 experts. The experts then reviewed the resulting system using their expert opinions to assess the maintainability of each system. The study used a Multiple-group Pre-test Post-test Design [7, p. 207] to triangulate the effect of the refactoring treatment between multiple groups. The use of a control group has been omitted from this study, most likely because a control group would represent a static program and obviously not likely to change. They found that two of the five chosen refactoring techniques reduced the maintainability of each system while two have a positive correlation, with the fifth technique having no significant correlation at all. These contradictory results to [6] could be the result of researching more than one system simultaneously. The internal validity of the study is threatened, however, by the use of the same experts for both the performing of each refactoring treatment across the systems, as well as the expert reviews in which maintainability is qualitatively evaluated. Such a method may lead to reactivity according to [7, p. 104] and bias among the experts who may be expecting each refactoring task to have an obvious outcome. This could have been mitigated by use of a double blind approach in which the researchers and experts would be unaware of the treatment or intended outcomes of the treatments administered, as well as using separate groups for administering the refactoring treatments and expert reviews.

IV. Conclusion

The aim of this literature review was to identify ways in which software could be designed to support maintainability and investigate research methods that have been employed to study software maintainability. Relevant literature has been found using online databases and two maintainability paradigms, *Architectural* and *Heuristic*, have been identified and examined in detail. Five texts have been chosen to facilitate this investigation and were reviewed in terms of their findings and research methods. It was found that the Architectural approaches of MD and DSL have advantages to maintainability in terms of speed and accuracy compared to conventional CC approaches. Possible improvements to researching MD and DSL approaches involve assessing more than a single system so that results may be triangulated and avoiding convenience sampling to improve external validity. Where maintainability research involves older systems, the development of a new clone system should be considered to remove code erosion as a variable. Heuristic approaches involving design patterns do seem to increase maintainability, but research into the use of refactoring has yielded ambiguous results. Future researchers should consider blinding researchers and experts to the nature of their experiments to reduce any bias in the results. Future work would could also be undertaken to quantify the tradeoff between complexity and maintainability when performing refactoring.

2011 words

References

[1] T. Mens, Y. Guéhéneuc, J. Fernández-Ramil and M. D'Hondt, *Guest editors' introduction: software evolution.* IEEE Software vol.27, 2010, pp.22-25

[2] ISO/IEC 25010, *Systems and softwware Engineering – Systems and Software Quaity Requirements and Evaluation (SQuaRE) – System and software quality models,* 2011

[3] Y. Martínez, C. Cahero, M. Matera, S. Abrahao and S. Luján, *Impact of MDE Approaches on the Maintainability of Web Applications: An Experimental Evaluation*, Conceptual Modelling, 30th International Conference, Berlin: Springer, 2011, pp.233-246

[4] K. Atalag, H.Y Yang, E. Tempero, J.R Warren, *Evaluation of softwatre maintainability with openEHR – a comparison of architectures*, International Journal of Medical Informatics, vol.83, Ireland: Elsevier, 2014, pp.849-859

[5] S. Sinlapalun and Y. Limpiyakorn, *ARSL: A Domain Specific Language for Aircraft Minima Separation Determination*, International Conferences, ASEA and DRBC, Berlin: Springer, 2012, pp.80-87

[6] S. Rochimah, M.B. Gautama and R.J Akbar, *Refactoring the Anemic Domain Model using Patterns of Enterprise Application Architeture and its Impact on Maintainability: A Case Study*, in IAENG International Journal of Computer Science, 2019, pp. 275-290

[7] P.D. Leedy, J.E Ormond, *Practical Research and Design*, 11th Edn, England: Pearson 2018

[8] R. Malhotra and A. Chug, *An Empirical Study to Assess the Effects of Refactoring on Software Maintainability*, in Intl. Conference on Advances in Computing, Communications and Informatics, India: 2016, pp. 110-117.

[9] P. Hegedüs, D. Bán, R. Ferenc and T. Gyimóthy, *Myth or Reality? Analyzing the Effect of Design Patterns on Software Maintainability*, International Conferences, ASEA and DRBC, Berlin: Springer 2012 pp.138-145