Towards a Taxonomy of Programming-related Difficulties during Maintenance

Aiko Yamashita
Mesan AS & Simula Research Laboratory
Oslo, Norway
Email: aiko@simula.no

Leon Moonen
Simula Research Laboratory
Oslo, Norway
Email: leon.moonen@computer.org

***Abstract*—Empirical studies that investigate the relationship between source code characteristics and maintenance outcomes rarely use causal models to *explain* the relations between the code characteristics and the outcomes. We conjecture that the lack of a comprehensive catalogue of programming-related difficulties and their effects on different maintenance outcomes is one of the reasons behind this. This paper takes the first step in addressing this situation based on empirical evidence collected in a longitudinal maintenance study on four systems. Professional developers were hired to implement a number of changes in each of the systems. These activities were observed in detail over a period of 7 weeks, during which we recorded on a daily basis *what specific problems* they faced. The collected data was transcribed and analyzed using open and axial coding. Based on an analysis of these results, we propose a preliminary taxonomy to describe the programming-related difficulties that developers face during maintenance. Our intention is not to replace the existing categorizations/taxonomies, but to take the first steps towards an integrated, comprehensive catalogue by aligning our empirical observations and the earlier literature.**

***Keywords*-maintainability, maintenance difficulties, maintenance problems, program comprehension, empirical study.**

I. Introduction

The assessment of how well a software system can be maintained is one of the greater and long-lasting challenges of software engineering. There is a substantial body of work that investigates if certain source code characteristics affect given maintenance outcomes (such as effort needed or defect proneness). When assessing software maintainability, it is not only important to determine *if* different code characteristics have an effect on maintenance outcomes, but we also need causal models to better understand *how* those characteristics affect the outcomes. These models are needed, for example, to uncover and reason about the interaction effects between different factors that may play a role.

One approach to get a step further in the development of causal models in software maintenance research is closely studying the difficulties that developers actually experience while solving change requests during maintenance. The study of maintenance difficulties is relevant for developing such causal models because these experiences reflect and potentially explain different outcomes of a maintenance project, such as performance, product quality or developers' motivation levels.

To this date, the software engineering literature does not provide an *integrated* compendium of programming-related difficulties occurring during programming specific activities during maintenance. A good starting point would be to look at *incremental change* work-flow described by Rajlich & Gosavi [1]. An integrated, well-defined taxonomy of difficulties during *each* of the steps in an working flow such as the *incremental change* flow can potentially support better understanding of the factors that affect maintainability.

By lack of *integration* we mean for instance, work on *program comprehension* only covers the initial steps (e.g., *concept extraction*, *concept location* [1]) of a maintenance task, leaving out code modification activities. Maintenance *challenges* have been depicted and categorized at project management level, and maintenance *activities* have been classified at the goal level, but these taxonomies are too coarse-grained to really address the "programming" aspects of maintenance and evolution at the level we see in [1].

In this paper, we present a proposal for a taxonomy on programming-related maintenance difficulties that is based on detailed observations of a maintenance project involving four Java web applications and six software professionals over a period of 7 weeks in total.

The remainder of this paper is structured as follows: Section 2 presents the related work and the knowledge gap. The approach for developing the taxonomy is discussed in Section 3. Section 4 presents the taxonomy and Section 5 argues for its relevance based on existing theories. We conclude and present plans for future work in Section 6.

II. Related Work and Knowledge Gap

Several studies have addressed and described different maintenance-related problems or difficulties, as well as different factors causing difficulties during maintenance.

Lientz and Swanson [2], and Dekleva [3] describe maintenance problems from a managerial perspective. For instance, they reported: *user knowledge*, *programmer effectiveness*, *product quality*, *programmer time availability*, *machine requirements*, and *system reliability* as the sources for major problems during maintenance. Similarly, Dekleva describes maintenance problems elicited from experienced software developers, and identified four major categories: *maintenance management*, *organizational environment*, *personnel factors*, and *system characteristics*. Palvia et al. [4] elaborate further on the initial set of problems/issues in software maintenance reported by Lientz and Swanson. Chapin et al. [5] provide a classification for types of software maintenance and evolution, and describe how their characteristics can

1063-6773/13 $26.00 © 2013 IEEE
DOI 10.1109/ICSM.2013.63

impact different aspects of the product and the business. Hall et al. [6] and Chen et al. [7] describe maintenance issues and potential factors behind them from a Software Process Improvement perspective, and Reedy et al. [8] propose a catalogue of maintenance issues within the context of software configuration management.

Karahasanovic et al., [9] reported a catalogue of program comprehension difficulties resulting from an experiment designed to investigate whether following a *systematic* strategy for program comprehension is realistic or not in large programs. They also compared which difficulties were associated to which type of comprehension strategy (i.e., between "as-needed" and "systematic" strategies).

Webster et al. [10] propose a taxonomy of risks in software maintenance, which describes potential factors negatively affecting software maintenance. Examples include: antiquated system and technology, program code complexity, difficulties in understanding programming "tricks", large number of lines of code affected by change request, and large number of principal software functions affected by the change.

From the maintenance problems illustrated within the identified literature, focus has been laid on -process and organizational factors, following more a managerial perspective. Technical or programming perspectives for maintenance issues seem to have a lower profile, with the work by Webster et al. and Karahasanovic et al. being the closest or most relevant to the focus of our work.

Most of the above-mentioned categorizations are not optimal for studying the relation between code characteristics and maintenance, since they: (1) do not cover the whole span of programming activities during maintenance, or (2) lack of enough level of detail (concreteness) to be of any use at the programming level. For instance, Karahasanovic et al. only focuses on comprehension-related difficulties, while comprehension only accounts for one portion of the maintenance activities, leaving out other tasks such *problem solving*, *impact analysis* and *debugging*. The work by Lientz and Swanson, Dekleva, Palvia, and Chen all focuses on managerial or process improvement perspectives. They cover the whole spectrum of the software maintenance process, and mention difficulties related to code quality very superficially.

III. Methodology

In this section, we first describe the context of the maintenance project from which the observations where derived. Subsequently, we provide a description of the qualitative techniques used in order to develop our taxonomy.

Maintenance Context: In 2008, Simula's Software Engineering department commissioned a maintenance project that involved adapting four existing Java applications to a new Content Management System (CMS). The project was conducted between September and December by six professional developers from two companies. The tasks covered the main maintenance classes (corrective, adaptive, perfective) and included adapting the systems to retrieve information from the new CMS, replacing the existing authentication mechanisms by web services and adding new reporting functionality. The project was designed as an ethnographic study in which the developers' activities were observed in detail (with consent). None of the maintainers, nor their companies, were involved in the original development, so they had no prior knowledge about of the systems. For more details on the study context we refer to [11].

Table I
EXAMPLE CODES FOR THE THREE PERSPECTIVES

Activity	*Difficulty*	*Factor*
Add source code	Confusion	Code Size
Modify source code	Error propagation	Bad naming
Look up information	Manual search	Logic entanglement
Configure library	Inconsistent change	Duplication
Debug	Scattered change	Rule violation

Data Collection: Three main data sources were used in order to build the taxonomy. First, the researcher present at the study kept an *observation logbook* where the most important aspects of the study were annotated in a daily basis. Second, interviews or progress meetings (20-30 minutes) were conducted daily for all developers to keep track of the progress, and register difficulties encountered during the project (e.g., dev4: *"It took me 3 hours to understand this method because of strangely named variables"*). Third, recorded think-aloud sessions (ca. 30 minutes) were conducted every second day to observe the developers in their daily activities. Subsequently, the recorded material from the interview and the think-aloud sessions were transcribed, annotated and summarized.

Data Analysis: *Grounded theory* was used to analyze the collected data. To this end, the transcripts from the interviews were coded using *open* and *axial* coding techniques [12]. *Open coding* is a form of content analysis in which the statements from the developers were annotated using more abstract labels (codes) that were (initially) derived from the *observation logbook*. For generating the coding schema, three perspectives were used: *Activity*, *Difficulty* and *Factor*. Activity refers to the maintenance context in which the difficulty manifests. Difficulty refers to the concrete hindrance to the activity being performed at the time, and Factor is the potential cause of that hindrance. For example, the following statement: *"...while searching of the place to make the change, I spent many hours because I had to examine many classes..."* would be coded as *Activity:* concept location [1], *Difficulty:* time consuming/manual search, and *Factor:* complexity. The coding schema was revised in an iterative fashion, during the annotation process. An example of various codes is shown in Table I. Subsequently, a technique called *axial coding* was used to group the annotated statements according to the most similar concepts, using all three perspectives described previously, thereby making connections between the initial categories and creating higher level abstractions.

IV. The Taxonomy

Based on the analysis of the data, and alignment of the terminology and concepts with the existing literature (discussed in Section II), we identified the following five types of maintenance difficulties:

(1) Introduction of faults as result of changes: comprises the introduction of faults to the system as result of changes performed in the code. Faults normally manifest in the form of *failures* identified by the developers during maintenance or detected during acceptance testing.

(2) Confusion and erroneous hypothesis generation during program comprehension: relates to developers generating an initial hypothesis on the behavior of the system or one of its components, and subsequently finding contradictory or inconsistent evidence. This category also included situations where the developer generates an erroneous hypothesis from "confounding" evidence in the code (which in some situations lead to the introduction of defects).

(3) Slow acquisition of overview/general understanding of the system: relates to developers spending considerable time on getting an overview of the system before they feel enough confident on the strategies they choose to solve the tasks. Also, difficulties in understanding the mechanics of subsystems fall into this category.

(4) Time-consuming information quests: relates to developers taking considerably long time before they could find the relevant information for their tasks.

(5) Wide-spread (time-consuming) changes: relates to situations where developers were forced to perform changes in extensive areas of the system in order to complete a given task. Often these changes required to manually inspect the areas in the code prior to any modifications.

A detailed description of this categorization, overall statistics and concrete examples of the difficulties identified can be found in the dissertation by Yamashita [13, pt. 2, ch. 4].

When observing the difficulties, is natural to see that some of them are related or one may constitute the cause for the other. For example, one can argue that (4) and (5) are the same type of difficulty. However, if we consider the *incremental change* steps described by Rajlich & Gosavi [1], the problems (2), (3), and (4) all relate to concept *extraction* and *location*. The difficulty (5) occurs later, when the developer performs the changes (i.e., *actualization*, *incorporation* and *propagation*), where for example the solution has to be reworked or the solution involves extensive "manual work".

Some of these difficulties relate to the work by Sillito et al., [14] who describes *challenges* developers face during change requests. For example, their challenge: "Gaining a sufficiently broad understanding" relates to our difficulty (3), while "Making and relying on false assumptions" relates to difficulty (2).

V. Taxonomy Relevance

This section discusses the relevance of this taxonomy using two perspectives: First, we discuss the importance of identifying situations where developers get confused due to misleading cues using the notion of *information beacons* [15]. Second, we discuss the notion of *cognitive overload* based on Wood's perspective on *task complexity* [16]. We show how our taxonomy helps to capture the interplay between source code properties and programming-related maintenance difficulties.

Confusion during maintenance: During the study, we could observe that two of the systems (which also happened to be smaller in LOC than the other two) displayed problems at structural or local level that confused the developers. In the first system, confounding elements were present at a structural level, where data/functionality allocation was not semantically coherent and moreover, not consistent. This resulted on early *false assumptions* of developers with respect to behavior of the code. In latter stages, developers would get confused while finding contradictory evidence on their initial assumptions. In many cases, the false assumptions would hold across the strategy selection and code modification process, leading to the *introduction of defects*. In the second system, variables where not used consistently, probably because of inconsistent *copy-paste* behavior from the programmers who initially wrote the application. In both of these cases, the challenges are closely related to confusing *beacons* or *misleading information cues*. Wiedenbeck defines beacons as: stereotypical segments of code, which serve as typical indicators of the presence of a particular programming structure or operation [15]. If we assume that software maintenance is a goal-oriented, hypothesis-driven, problem-solving process (cf. [17]), any form of confusion that leads to the generation of *false hypotheses* during program comprehension and strategy selection is an important aspect to observe when assessing the maintainability of the source code. Several studies have pointed out the criticality of beacons and information cues for program comprehension. Ko et al. observed how misleading information cues induced failed searches for task-relevant code during maintenance [18]. Wiedeneck reported that beacons which were inappropriately placed in a program led to "false comprehension" of the program's function [15]. Soloway & Erlich discussed how expert performance drops to near the level of the novice programmers due to rule violations [19].

Cognitive overload during maintenance: In our study, the two largest systems were found to be more difficult to understand (or get an overview of their behavior) and to modify than the other two smaller systems [13]. Particularly in the case involving the largest system of all four, the understanding was difficult because information cues were spread across a wide cognitive space, and developers were forced to examine many areas of the code, gathering the

"pieces of the puzzle" before they could achieve enough-good understanding of the overall system's behavior. In the same system, developers reported the usage of an extensive, third-party library, as the most time-consuming task. What they referred as "time-consuming" work was not to understand the behavior of the library, but actually to put the elements provided by the library together in order to solve the task. In contrast to the smaller systems, *consistency* was not a problem, but the *complexity*, which resulted in a different type of difficulty. No developers reported being *confused* while trying to understand the large systems, they said it was just hard to get used to the library, and then to use it. These observations are echoed by Wood's perspective on *task complexity*, in particular to the notion of *component* complexity [16]. *Component complexity* refers to the number of distinct information cues that must be processed simultaneously in the performance of a task. They also relate to Swezller's notion of *intrinsic cognitive load* [20], which states that high element interactivity in tasks imposes a high cognitive load, where elements interactivity means that they are related in a manner that requires them to be processed simultaneously. In our study, we could clearly identify situations in which *component complexity* increased the effort of *understanding*, and of *using* large libraries, because they required developers to gather distinct, wide-spread information cues.

VI. Concluding Remarks and Future Work

We are aware that many of the concepts and phenomena that we have describe here, have also been reported in other studies and across different research domain areas (e.g., HCI, Program Comprehension, Software Maintenance). Our intention is not to *replace* the existing categorizations/taxonomies, but to take the first steps towards an integrated, comprehensive catalogue of difficulties/problems developers face during programming-related maintenance activities.

In the future, we would like to refine the taxonomy by improving the consistency, conciseness and descriptive richness of each type of difficulty, and by mapping the difficulties to each of the different programming-related maintenance activities. A preliminary idea is to build a catalog of programming-related maintenance activities based on the stages involved in the *Incremental Change* process, described by Rajlich & Gosavi [1]. Another idea to extend our catalog of difficulties is to "reverse-engineer" the literature on factors *affecting* maintainability (e.g., work by Nierstrasz [21]) and extract and describe in detail the *difficulties* caused by the different factors.

References

[1] V. T. Rajlich and P. Gosavi, "Incremental change in object-oriented programming," *IEEE Softw.*, vol. 21, no. 4, pp. 62–69, 2004.

[2] B. P. Lientz and E. B. Swanson, "Problems in application software maintenance," *Communications of the ACM*, vol. 24, no. 11, pp. 763–769, 1981.

[3] S. Dekleva, "Delphi study of software maintenance problems," in *IEEE Int'l Conf. Softw. Maintenance*, 1992, pp. 10–17.

[4] P. Palvia, A. Patula, and J. Nosek, "Problems and Issues in Application Software Maintenance," *J. Information Technology Management*, vol. 4, no. 3, pp. 17–28, 1995.

[5] N. Chapin, J. E. Hale, K. M. Kham, J. F. Ramil, and W.-G. Tan, "Types of software evolution and software maintenance," *J. Softw. Maintenance: Research and Practice*, vol. 13, no. 1, pp. 3–30, 2001.

[6] T. Hall, A. Rainer, N. Baddoo, and S. Beecham, "An empirical study of maintenance issues within process improvement programmes in the software industry," in *IEEE Int'l Conf. Softw. Maintenance*, 2001, pp. 422–430.

[7] J.-C. Chen and S.-J. Huang, "An empirical analysis of the impact of software development problem factors on software maintainability," *JSS*, vol. 82, no. 6, pp. 981–992, 2009.

[8] A. Reedy, D. Stephenson, E. Dudar, and F. Blumberg, "Software configuration management issues in the maintenance of Ada software systems," in *IEEE Int'l Conf. Softw. Maintenance*, 1989, pp. 234–245.

[9] A. Karahasanović, A. K. Levine, and R. Thomas, "Comprehension strategies and difficulties in maintaining object-oriented systems," *JSS*, vol. 80, no. 9, pp. 1541–1559, 2007.

[10] K. Webster, K. M. de Oliveira, and N. Anquetil, "A risk taxonomy proposal for software maintenance," in *IEEE Int'l Conf. Softw. Maintenance*, 2005, pp. 453–461.

[11] A. Yamashita, "Assessing the capability of code smells to explain maintenance problems: an empirical study combining quantitative and qualitative data," *Empirical Softw. Eng.*, pp. 1–33, 2013.

[12] A. Strauss and J. Corbin, *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. SAGE, 1998.

[13] A. Yamashita, "Assessing the Capability of Code Smells to Support Software Maintainability Assessments," Doctoral Thesis, University of Oslo, 2012.

[14] J. Sillito, K. De Volder, B. Fisher, and G. Murphy, "Managing software change tasks: an exploratory study," in *Int'l Symp. Empirical Softw. Eng.* IEEE, pp. 23–32.

[15] S. Wiedenbeck, "The initial stage of program comprehension," *J. Man-Machine Studies*, vol. 35, no. 4, pp. 517–540, 1991.

[16] R. E. Wood, "Task complexity: Definition of the construct," *Organizational Behavior and Human Decision Processes*, vol. 37, no. 1, pp. 60–82, 1986.

[17] J. Koenemann and S. P. Robertson, "Expert problem solving strategies for program comprehension," in *SIGCHI Conf. Human Factors in Computing Systems*, 1991, pp. 125–130.

[18] A. Ko, B. Myers, M. Coblenz, and H. Aung, "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks," *TSE*, vol. 32, no. 12, pp. 971–987, 2006.

[19] E. Soloway and K. Ehrlich, "Empirical Studies of Programming Knowledge," *TSE*, vol. SE-10, no. 5, pp. 595–609, 1984.

[20] J. Sweller, "Cognitive load theory, learning difficulty, and instructional design," *Learning and Instruction*, vol. 4, no. 4, pp. 295–312, 1994.

[21] O. Nierstrasz and M. Denker, "Supporting Software Change in the Programming Language," in *OOPSLA Ws. Revival of Dynamic Languages*, 2004, p. 5.