Maintenance and Agile Development: Challenges, Opportunities and Future Directions

Geir K. Hanssen[(1,2)], Aiko Fallas Yamashita[(3,4)], Reidar Conradi[(1)] and Leon Moonen[(3)]
[(1)] *Norwegian University of Science and Technology, Trondheim, Norway*
[(2)] *SINTEF ICT, Trondheim, Norway*
[(3)] *Simula Research Laboratory, Oslo, Norway*
[(4)] *Department of Informatics, University of Oslo, Oslo, Norway*
ghanssen@sintef.no, aiko@simula.no, reidar.conradi@idi.ntnu.no, leon.moonen@computer.org

Abstract

Software entropy is a phenomenon where repeated changes gradually degrade the structure of the system, making it hard to understand and maintain. This phenomenon imposes challenges for organizations that have moved to agile methods from other processes, despite agile's focus on adaptability and responsiveness to change. We have investigated this issue through an industrial case study, and reviewed the literature addressing software entropy, focusing on the detection of "code smells" and their treatment by refactoring. We found that in order to remain agile despite of software entropy, developers need better support for understanding, planning and testing the impact of changes. However, it is exactly work on refactoring decision support and task complexity analysis that is lacking in literature. Based on our findings, we discuss strategies for dealing with entropy in this context and present avenues for future research.

1. Introduction

One major challenge when adopting agile software development is the interplay between an increasingly complex code base and the high-pace development demanded by the agile workflow. When this interplay is not handled properly, it decreases the productivity of the team and the product quality. Currently, there is limited understanding of the particular challenges that software entropy[1] entails when software maintenance is performed under agile methods [1]. We investigated the challenges within the aforementioned context by conducting a case study on a company that currently uses the Evo method [2]. In addition to the case study, we conducted a literature review on tools, methods and knowledge available for detection and refactoring of code smells. Space limitations prevent us from presenting the complete results of the review (which will be reported in a separate publication) but we include relevant work in the discussion below. The remainder of this paper is structured as follows: Section 2 describes the methodology and findings from the case study, Section 3 discusses the findings and potential solutions to the identified issues, and Section 4 concludes with directions for future research.

2. Case study

2.1. Context and methodology

(a) Context of study. CSoft (an anonym) is a medium-sized Norwegian software company that develops, maintains and markets a product line. The development in CSoft evolved thirteen years ago from "creative chaos" to a waterfall-inspired process and about 5 years ago they adopted Evo [2], an agile method comparable to the better-known Scrum method [3]. Currently, development is done in short iterations, open to changes in requirements and design, aims at delivering working software after each iteration, and lead users are invited to participate in the process.

(b) Case study protocol. First, we collected data by means of a workshop with CSoft and Patrick Smacchia (an external consultant) where CSoft's source code was analyzed with Smacchia's tool NDepend. Next, we conducted an in-depth interview with two architects from CSoft, and analyzed the transcripts.

(c) Technical properties of the system. The system has been under constant development for the last thirteen years, entangling solutions in ASP, COM+, VB6 and other legacy technologies. Today, most new

[1] Software entropy is known under different names such as code decay, software rot, or software erosion

code is developed in C#, distributed over approximately 160 .Net assemblies. The product has a three-tier architecture with a clean separation between the presentation and the business layer. The main problem in the software is what the architects refer to as the Blob: a very large assembly, central to the system (and aptly named Core), which consists of approximately 150K lines of code in 144 namespaces, many of them displaying cyclical dependencies.

2.2. Findings

We summarize the problems that are experienced during development by distinguishing four aspects:

(a) Analyzability and comprehensibility. The complexity of the system (specially the *Core* component) makes it difficult to understand the structure and behavior of the code. Thus, new developers joining R&D have a steep learning curve and require close follow-up over a long period of time by more experienced developers. Also, there is a lack of documentation or models that explain the structure of the system, even though this clearly would be highly beneficial, both to existing and new developers.

Comprehension issues lead to a *fear of changing the code*, both for adding new features and for refactoring. The unclear internal structure creates a cognitive overload that is in practice generally avoided by code duplication: instead of modifying or reusing existing code, developers create their own copy over which they have full control. This increases the cognitive overload for other developers, creating a vicious circle.

(b) Modifiability and deployability. As a result of the duplication and entanglement of code, developers frequently need to perform so-called *shotgun surgery*, where even the modification of a small detail requires identifying and modifying many code segments. This problem slows down the development and increases the potential for errors, since developers may overlook changes in one or more locations. Also, deployment of the product is affected since the *Core* component aggregates features and functionality for every possible configuration of the product. This forces the release of the product as a whole, though only a fraction of the functionality is needed for a particular configuration.

(c) Testability and stability. Due to the size of the code and the amount of cross-references, there are too many paths through the code to test them all systematically. Test coverage is not considered high enough and existing tests have shown to be unstable and inconsistent (e.g., same tests running on similar systems produce different outcomes that are hard to explain). The existing tests are extremely large, thus hard to maintain and use. When a test fails, the corrective task effort is high. Although the test sets are supposed to act as a *safety net* (similar to regression tests), in practice this safeguard is not trusted. The unforeseeable effects of changes and the potentially high corrective maintenance effort increase the fear/reluctance to change existing code.

(d) Organization and process. As both the business domain and the system are highly complex, each of the development teams (4-6 developers) contains an expert, the so-called guru. This guru is technically skilled and has long experience with the code, which is vital for the team to solve the tasks. This represents a high vulnerability for the organization, as the knowledge of the system is not evenly spread.

The development process is based on two-week iterations, with a strong focus on delivering working software by the end of each iteration. However, a negative effect of this focus is that the quality of software is at times traded in for creating a working version. Each iteration ends with a review, but the high velocity typically does not give enough time to catch all issues. This results in a high workload close to a product release, when the system needs to be thoroughly tested as a whole.

The development teams are set up to have separate areas of concern, each team being responsible for a part of the total product (e.g., the reporting solution or the data storage). The rationale is to build competence around a well-defined part of the product. However, the structure of the system does not reflect this organization in practice, as functionality is spread throughout the entire code. This forces the teams to operate outside their area of concern, which has shown to negatively affect their ability to produce sufficient new and improved features of the product in their releases, leading to constant delays in their delivery.

3. Discussion

In this section we analyze the results from the case study and, drawing from our literature review, discuss strategies for addressing the issues identified:

(a) Analyzability and comprehensibility. We observe that most agile methods assume that development starts from scratch and ends with a release – post- release maintenance is not covered. In our case, the system was already very complex when Evo was adopted, and although agile methods promote communication over documentation, the lack of adequate documentation holds back the comprehension of such a system. While agile methods like to state that “the code is the documentation”, there is no guarantee that the code can serve this purpose if the system was originally developed using different methods. Van Deursen remarks that pair

programming is one of the particular agile practices that support comprehension [4]. However, in the case of CSoft, the limited number of 'experts', the high number of new coming developers, and the urgent demands on new functionality, makes that pair programming is not considered a very practical nor scalable solution for spreading knowledge. To improve program comprehension and overcome "the fear of change", we suggest introducing *semi-automatic code inspections* like those in [5], potentially extended with visualizations and analyses from [6-8]. In addition, refactorings to untangle crosscutting concerns will improve the comprehensibility [9] as they help to distinguish code for various business segments and separate business and platform code.

(b) Modifiability and deployability. According to Martin [10], dependency problems, like the ones observed in our case study, largely relate to two design smells: *rigidity* and *immobility*. Rigidity meaning that a change in the system implies a cascade of changes in other modules, and immobility refers to the inability of the system to encapsulate components that can be reused, because it implies too much effort or risk. If the smells are all over the system, high-level restructuring it is required to remove unwanted dependencies. One immediate consequence of these dependency issues is the violation of the *Interface Segregation Principle* [10], explaining most of the difficulties in the deployment stage. The analysis (and reduction) of module dependencies [11, 12] can help to revise these interfaces. In addition, specific refactorings to reduce architectural violations [13] will help to further improve modifiability and deployability.

(c) Testability and stability. Unit testing is one of the important components of agile methods. In the context of our case, the sheer size combined with a large amount of dependencies in the code hinders the definition of unit tests and high levels of coverage. Due to the high pace of development, there is little room for regression, integration and system testing during the iterations and CSoft mainly relies on the customer for external quality checks. Recent work has focused on methods and techniques for improving unit test suits [14-16], alongside with empirical studies on defects prediction [17] that aid planning. However, there are still various challenges to agile testing that go beyond unit testing that are not completely understood [18, 19]. Although we consider visualization and analysis tools to be useful, we know that non-trivial refactorings are risky and time consuming due to the unstable characteristic of the system. The current lack of understanding of the effects of given code smells and refactorings makes this task very challenging [20].

(d) Organization and process. The strong focus on rapid and continuous delivery of features at CSoft has lead to the construction of teams with defined *areas of concern*. We conjecture that an important reason for delays on the incorporation of new features is due to the system not reflecting the same separation of concerns as the development tasks/teams. The resulting entanglement of *crosscutting concerns* is a common problem with software maintenance. As discussed above, refactoring towards an aspect-oriented version could help to restructure the code according to the *areas of concern* [9], but this area is relatively new and tools have only recently been presented [21, 22]. The lack of adequate information to perform the planning could be another reason for delays. Planning of iterations could be enhanced by considering additional information, such as complexity analysis of the tasks to improve on estimations obtained from planning poker. However, such complexity analysis may still have limited effect in practice, as there is not enough empirical evidence on the impact of different refactorings [20]. These uncertainties could be compensated by continuous quality monitoring, for example by combining *evolution monitoring* [23-26] and *semi-automatic code inspections* [5] to analyze metrics and code smells using a tools like NDepend. This analysis is best incorporated in the development workflow to detect and repair issues as soon as possible. In such a setting, detection of defect or performance issues in the system [27, 28] could be used as a prioritization mechanism for refactorings.

4. Concluding remarks

In this paper, we have presented some of the problems agile practitioners face when dealing with software entropy in large projects. We found that, to keep *agile responsiveness* in the presence of software entropy, better support is needed for understanding, planning and testing the impact of changes. Moreover, we found that there is relatively little empirical evidence and methods for refactoring decision support. Code smells themselves are indicators that refactoring is needed, but after analyzing code smells, one also needs guidance to drive refactoring in a cost-effective way. Detection answers the question "where are the code smells?" but more support is needed to answer questions like "which code smells should we refactor first?" and "which combination of refactorings has the best overall effect?" We underline the statement by Counsell that assessing the cost-benefits of different refactorings is still largely an open area for research [20].

Finally, our literature review indicated that most of the methodological and tool contributions were still in their development stage. More relevant case studies and better evaluations of the available tools are needed, so practitioners can evaluate the different solutions and adopt the most appropriate ones to their context. Mealy et al. [29] have suggested a set of usability requirements for refactoring tools. In addition, evaluation frameworks like the one suggested by Maletic et al. [30] are needed to assure comparable results.

5. References

[1] Rajlich, V., Changing the paradigm of software engineering, in Comm. ACM. 2006. p. 67-70.

[2] Gilb, T., Competitive Engineering: A handbook for systems engineering, requirements engineering, and software engineering using Planguage. 2005: Elsevier.

[3] Schwaber, K., Beedle, M., Agile Software Development with Scrum. 2001: Prentice Hall.

[4] van Deursen, A., Program comprehension risks and opportunities in extreme programming, in Working Conf. on Reverse Eng. 2001, IEEE. p. 176-185.

[5] van Emden, E. and L. Moonen, Java quality assurance by detecting code smells, in Working Conf. on Reverse Eng. (WCRE). 2002. p. 97-106.

[6] Parnin, C., C. Görg, and O. Nnadi, A catalogue of lightweight visualizations to support code smell inspection, in Symp. on Softw. Visualization (SOFTVIS). 2008, ACM.

[7] Trifu, A. and U. Reupke, Towards Automated Restructuring of Object Oriented Systems, in Conf. on Softw. Maintenance and Reengineering (CSMR). 2007, IEEE. p. 39-48.

[8] van den Brand, M.G., et al., Using The Meta-Environment for Maintenance and Renovation, in Conf. on Softw. Maintenance and Reengineering (CSMR).2007, IEEE.

[9] Moonen, L., Dealing with Crosscutting Concerns in Existing Software, in Intl Conf. on Softw. Maintenance - Frontiers of Softw. Maintenance (ICSM/FoSM 2008).

2008, IEEE. p. 68-77.

[10] Martin, R.C., Agile Software Development, Principles, Patterns and Practice. 2002: Prentice Hall.

[11] Arevalo, G., S. Ducasse, and O. Nierstrasz, Discovering Unanticipated Dependency Schemas in Class Hierarchies, in Conf. on Softw. Maintenance and Reengineering (CSMR). 2005. p. 62-71.

[12] Leitch, R. and E. Stroulia, Assessing the Maintainability Benefits of Design Restructuring Using Dependency Analysis, in Intl Symp. on Softw. Metrics (METRICS) 2003.

[13] Bourqun, F. and R.K. Keller, High-impact Refactoring Based on Architecture Violations, in Conf. on Softw. Maintenance and Reengineering (CSMR). 2007. p. 149-158.

[14] Guerra, E.M. and C.T. Fernandes, Refactoring Test Code Safely, in Intl Conf. on Softw. Eng. Advances. 2007.

[15] van Deursen, A., et al., Refactoring test code, in extreme Programming Perspectives, M. Marchesi, et al., Editors. 2002, Addison-Wesley.

[16] van Rompaey, B., et al., On The Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test. TSE, 2007. 33(12): p. 800-817.

[17] Li, W. and R. Shatnawi, An empirical study of the bad smells and class error probability in the post-release object-oriented system evolution. JSS, 2006. 80(7): p. 1120-1128.

[18] Pettichord, B., Agile Testing Challenges, in Pacific Northwest Softw. Quality Conf. (PNSQC). 2004. p. 481.

[19] Talby, D., et al., Agile Software Testing in a Large-Scale Project. IEEE Softw., 2006. 23(4): p. 30-37.

[20] Counsell, S., et al., The Effectiveness of Refactoring, Based on a Compatibility Testing Taxonomy and a Dependency Graph, in Testing: Academic & Industrial Conf. on Practice And Research Techniques (TAIC- PART). 2006, IEEE.

[21] Binkley, D., et al., Tool-supported refactoring of existing object-oriented code into aspects. TSE, 2006. 32(9): p. 698-717.

[22] Marin, M., et al., An integrated crosscutting concern migration strategy and its semi-automated application to JHotDraw. JASE, 2009.

[23] D'Ambros, M., Supporting software evolution analysis with historical dependencies and defect information, in Intl Conf. on Softw. Maintenance (ICSM). 2008, IEEE. p. 412-415.

[24] Jermakovics, A., M. Scotto, and G. Succi, Visual identification of software evolution patterns, in Intl Ws. Principles of Softw. Evolution (IWPSE). 2007, ACM. p. 27-30.

[25] Kiefer, C., A. Bernstein, and J. Tappolet, Mining Software Repositories with iSPAROL and a Software Evolution Ontology, in Intl Ws. Mining Softw. Repositories (MSR). 2007.

[26] Xing, Z., Analyzing the Evolutionary History of the Logical Design of Object-Oriented Software. TSE, 2005. 31(10): p. 850-868.

[27] Chaabane, R., Poor Performing Patterns of Code: Analysis and Detection, in Intl Conf. on Softw. Maintenance (ICSM). 2007.

[28] Wasylkowski, A., A. Zeller, and C. Lindig, Detecting object usage anomalies, in European Softw. Eng. Conf.

& Symp. on the Foundations of Softw. Eng. (ESEM/FSE). 2007, ACM. p. 35-44.

[29] Mealy, E., Improving Usability of Software Refactoring Tools, in Australian Softw. Eng. Conf. (ASEC). 2007. p. 307-318.

[30] Maletic, J.I., A. Marcus, and M.L. Collard, A Task Oriented View of Software Visualization, in Intl Ws. Visualizing Softw. For Understanding and Analysis (VISSOFT). 2002, IEEE.