Refactoring—a Shot in the Dark?

Marko Leppänen, Tampere University of Technology
Simo Mäkinen, University of Helsinki
Samuel Lahtinen and Outi Sieve-Korte, Tampere University of Technology
Antti-Pekka Tuovinen and Tomi Männistö, University of Helsinki.

The article appears in the November-December 2015 refactoring special issue of IEEE Software. This is the author's post-print version of the article which is in content identical to the publisher's final version. See below for the link to the publisher's final version.

Publisher's version available at http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7310989&isnumber=7310982. The digital object identifier (DOI) of the article is 10.1109/MS.2015.132

The article should be referred to in the following manner:

> Leppänen, Marko; Mäkinen, Simo; Lahtinen, Samuel; Sievi-Korte, Outi; Tuovinen, Antti-Pekka; Männisto, Tomi, "Refactoring-a Shot in the Dark?," in *Software, IEEE*, vol.32, no.6, pp.62-70, Nov.-Dec. 2015.

Refactoring—a Shot in the Dark?

Marko Leppänen
Tampere University of Technology
Department of Pervasive Computing
P.O.Box 527, FI-33101 TAMPERE, Finland

Simo Mäkinen
University of Helsinki
Department of Computer Science
P.O. Box 68 (Gustaf Hällströmin katu 2b)
FI-00014 UNIVERSITY OF HELSINKI, Finland

Samuel Lahtinen
Outi Sieve-Korte
Tampere University of Technology
Department of Pervasive Computing
P.O.Box 527, FI-33101 TAMPERE, Finland

Antti-Pekka Tuovinen
Tomi Männistö
University of Helsinki
Department of Computer Science
P.O. Box 68 (Gustaf Hällströmin katu 2b)
FI-00014 UNIVERSITY OF HELSINKI, Finland

Abstract

A study performed semistructured interviews of 12 seasoned software architects and developers at nine Finnish companies. Its main goals were to find out how the practitioners viewed the role and importance of refactoring, and how and when they refactored. Another goal was to see whether shortened cycle times and, especially, continuous-deployment practices affected how and when refactoring was done. The results paint a multifaceted picture with some common patterns. The respondents considered refactoring to be valuable but had difficulty explaining and justifying it to management and customers. Refactoring often occurred in conjunction with the development of new features because it seemed to require a clear business need. The respondents didn't use measurements to quantify the need for or impact of refactoring.

1. Introduction

Refactoring is considered an implicit part of agile software development. When working on code, developers are expected to constantly improve its quality—for example, maintainability. However, developers are often pressured to use their work time to add features rather than refactor the code. The increasing pace of business requires quickly delivering a steady stream of new functionalities.

To study the state of refactoring in the industry, we interviewed software architects and developers at prominent Finish software companies. We aimed to find out what they considered to be refactoring, what benefits and challenges they saw in it, and how they incorporated refactoring into their development workflows. We also wanted to learn whether the companies' decision to employ refactoring was based on facts or just a shot in the dark.

2. Background

Refactoring is basically a straightforward technique. However, researchers have introduced additional concepts that paint a more complex picture of refactoring in development workflows.

Martin Fowler defined refactoring as making behavior-preserving design improvements right after adding and testing new functionality [1]. But developers can also refactor code opportunistically—recurrent "upkeep refactoring" has been called "floss refactoring," "litter-pickup refactoring," or "comprehension refactoring" [2], [3], [4]. Adding new functionality might trigger "preparatory refactoring" that makes the changes easier [4].

Without floss refactoring, developers might need to apply "root canal refactoring" or "planned refactoring"—that is, change large parts of the code, which requires specific backlog items [3], [4]. Fowler considered a recurring need for planned refactoring as a bad smell [4]. He suggested that architectural changes could be done as "long-term refactoring," in which developers gradually and knowingly migrate the system to a new architecture by applying opportunistic changes. Michael Stal recommended recurrent refactoring of the architectural design directly [5]. Although everyday work can easily include upkeep refactoring, large-scale or architectural refactoring is difficult to justify to stakeholders, especially customers, who might not understand the nature of software [2], [4], [5].

The views on refactoring have been mixed. Some people see refactoring as beneficial, even a success factor, whereas others still strongly advocate the "if it ain't broke, don't fix it" mentality [6]. Developers feel that refactorings' benefits are difficult to measure and sell to management [7]. Developers don't care to make quality measurements; they want to tackle

TABLE 1. THE COMPANIES IN THE STUDY AND THEIR REFACTORING PRACTICES

Company	Domain	Product	Stairway to Heaven level	Test automation	Refactoring development ratio estimate	Refactoring frequency	Deployment frequency
A	Machine control	Internal, outsourced	A	Some (some builds could be broken for weeks)	50/50	Weekly	Monthly
B	Medical monitoring	Contracted	A	Yes	95/5	Almost daily	Every two years
C	Web service	Contracted	B	No	(10–15)/85	Daily	Once per project
D	Weather information	External	C	Yes	(20–30)/70	Daily	Weekly to biweekly
E	Web services	Contracted	C	Yes	33/67	Daily	Once a month
F	Web services	Contracted	C	Yes	(5–15)/85	Incorporated in feature development	Once a year
G	Transportion service on demand	External	C	Yes	15/85	Daily	Bimonthly
H	Industrial automation	Internal	D	Yes	(10–20)/80	Daily to weekly	Several years
I	Content management system product	External	D	Yes	20/80	Daily	10 times a day
J	Web services	Contracted	E	Yes	20/80	Every other day	Daily

the concrete problems in code, rather than trying to improve internal code quality metrics [8]. There are conflicting results if refactoring actually improves code metrics [9].

Refactoring poses risks, and experienced architects might even aim to minimize it [10]. Risks and difficulties increase as the system size increases [9], [11] and careless refactoring might introduce a significant number of faults [12].

3. Study Description

We looked for information-rich cases of software in companies in varied industry domains and with varied process maturity. We conducted semistructured, themed interviews in nine companies with 12 senior software architects or developers in 10 sessions. The interviewees had a minimum of 10 years' experience.

We carried out the interviews in Finnish in 2015, with one or two interviewers and interviewees present. Audio was recorded and transcribed. We analyzed the data qualitatively following Per Runeson and Martin Höst's case study guidelines for identifying common themes from the transcripts and annotating the respective sections of text with specific keywords [13].

Table 1 presents the companies' characteristics. The industry domains ranged from digital-services production in application areas such as weather information, public transportation, and Web services, to embedded industrial automation and machine control. The companies' sizes differed, but the interviewees generally worked in small development teams of fewer than 10.

Software development teams performed product development internally, performed it externally for customers and users, or contracted work to other parties, which can have implications for accepted refactoring practices. Regarding development practice maturity, we roughly mapped the involved companies according to the Stairway to Heaven model [14]. The model conceptualizes organizational software development capabilities as five steps: A—traditional development, B—agile R&D organization, C—continuous integration, D—continuous deployment, and E—R&D as an innovation system.

We used the Stairway to Heaven level to get a more representative sample of the companies regarding the adoption of development practices supporting continuous delivery and deployment. Seven companies were at step C or higher.

Most companies employed test automation, which provides the essential safety net for refactoring. Although internal development cycles could be fast, production releases were far rarer, so that the deployment frequency could be up to several

years, depending on the domain. Also, Table 1 presents the interviewees' impressions regarding the refactoring effort in their recent projects. Refactoring was common; typically, the interviewees spent a fifth of their working time on it.

Our research protocol helped establish a solid chain of evidence for the conclusions made in this article. Although we tried to gather rich experience-based data from a variety of companies with divergent domains and process maturity, the sampling of companies was based mostly on convenience sampling and the availability of cases identified in the network of familiar companies. In the companies, we looked for cases with a suitable history in terms of refactoring and individuals who had been involved in software development for a long time with field experience in refactoring. We acknowledge that single case studies, such as this one, are limited to describing the situation in the cases, whereas theory creation or confirmation of hypotheses requires more studies. So, we concentrate on reporting insights from the cases and avoid quantitative aggregates because of their lack of clear representativeness.

4. Understanding Refactoring and Its Causes

The interviewees generally understood refactoring as changing the code's structure without altering the program's perceived behavior. This resonates with the idea originally presented by Martin Fowler [1]. However, the perceptions of the magnitude and nature of structural changes representing refactoring were equivocal. Interviewees usually reserved the word "refactoring" for restructuring and redesigning the system, as in preparatory or planned refactoring [4]. They didn't consider daily small changes—for example, method renaming and moving the code around a bit—as refactoring. As example cases, the interviewees described tasks whose work amount would equal several days. So, the practical use of the term didn't fully align with Fowler's original meaning. Thus, you should be aware that the use of the term "refactoring" in this article reflects our interviewees' understanding of it.

The interviewees nominated the constant rush as the leading cause of refactoring because there wasn't enough time to create good code. Developers reported knowingly taking shortcuts and acknowledged that the choices might come back to haunt them.

Refactoring needs also arise because people learn. While engaged in daily software development, developers gain new insights and see existing code and structures in a new light. The interviewed developers felt that initially, they lacked the technical skills or understanding about the domain necessary to make good decisions. In addition, as a developer at a start-up mentioned, refactoring lets you try to learn whether a certain way of coding would be more appropriate than the current implementation, providing the opportunity for personal growth.

Architectural constraints might still be vague at the outset, leading to refactoring. Several interviewees replied that sometimes they had a less than perfect picture of what was to be built because the information between developers and customers didn't flow smoothly. The interviewee from company A, whose development is heavily outsourced, said that distributed teams, possibly separated by a language barrier, contributed to the miscommunications. The design constraints also changed because the world evolved, resulting in subsequent changes in requirements.

In contrast with what we just discussed, refactoring can be a philosophy of developing software. A developer mentioned that he promoted and practiced a culture in which developers gradually constructed the code better and better. Refactoring in this sense became a part of the normal development flow.

5. Are Refactoring Decision Rational?

A tug of war seemed to exist between feature development and refactoring. New features generate revenue, whereas refactoring is an investment in the developers' experience. However, the interviewees clearly stated that time spent on refactoring usually paid itself back in future development. New features can be rapidly implemented if the code quality is already good. However, the need for refactoring seemed to arise only from the developers' tacit knowledge, and no good measures for quantifying the return on investment existed. Making purely rational decisions was difficult.

Overall, the interviewees had difficulty estimating the workload needed for refactoring when developing new features. If the code quality was bad, unexpected things could bog down the development even in tasks perceived as easy and small. In company B, a relatively small change, adding a battery charge level to a display, took three and a half weeks instead of the expected few hours. In relation to this, a few companies tracked the team velocity as a proxy metric for code quality because velocity tends to decrease quickly when quality problems occur in the code.

Table 1 presents the ratio of the time spent on refactoring to the time spent developing new features. This could serve to roughly estimate code quality because the time spent on refactoring was more pronounced in systems in which developers felt the overall quality was low. Of course, this ratio also reflects what counted as refactoring. If all code polishing was counted, this figure could be significantly higher, as several interviewees affirmed. The developers were usually satisfied with the code quality if refactoring took less than 20 to 30 percent of their time. Strikingly, in company B's project, refactoring took 95 percent of the time. Consequently, that company's interviewee mused that it would have been more efficient had the project been scrapped and restarted from scratch.

The interviewees primarily considered refactoring as an internal issue of the development team. Most interviewees included refactoring in the effort estimate of a task. However, it was a different story when the need for a larger restructuring or redesign arose. In those cases, most interviewees asked management for permission. Such a task was usually perceived to take more than one day.

In customer projects, refactoring didn't "bring cash to the house," so convincing customers to pay for it was sometimes difficult. We speculate that if you have to communicate the need for refactoring to the customer, it becomes a trust issue. Even though agile methodologies build on mutual trust, they also promote visibility with frequent demos. Because refactoring is by definition something that doesn't change the code's behavior, its benefits are difficult to show to customers.

However, several interviewees said that the more technical background the customer's representative had, the easier it was to convince him or her to take quality issues into account. One interviewee said that refactoring had to be demystified because the concept wasn't self-evident to customers. In committing stakeholders to their way of working, developers can introduce refactoring as a significant tool for continuous quality.

If the development was for in-house purposes only or there wasn't so much hurry to produce new features, refactoring was more common. However, even in those cases, it didn't pay off to change something that remained static for the foreseeable future. So, refactoring was usually paired with an actual functional change. One attribute affecting the decision making was that if the software's expected development lifecycle was short, it didn't pay to refactor the code. It seemed that monolithic software underwent more refactoring than limited-life microservices, which tended to eventually become static.

6. Why Bother?

Refactoring doesn't bring visible changes for users in terms of added functionality, so what does refactoring improve? Our interviewees pointed out several benefits.

Regardless of the results for users, the code level changes are visible to the developers, who have to read, understand, and modify the code. Almost all interviewees agreed that by refactoring, they were trying to make future development easier—not only improving the code's maintainability but also easing the addition of features. A software architect at company J said,

> *Then another thing is to make future development possible altogether, so that we don't end up in a situation in which maintenance or further development would be impossible.*

Refactoring is obviously done by developers for developers. Most interviewees saw that refactoring can improve how developers read and understand code. This is comprehension refactoring as Martin Fowler defined it [4]. One developer mentioned that you write code not only for yourself or the compiler but also for other developers who might need to take care of the code later and adopt the style the code was written in. For example, company D's interviewee stated that code was more readable and easier to understand when the method names were clear and development followed the single-responsibility principle.

Although the developers—similarly to a large survey study at Microsoft [7]—wanted to keep the code maintainable and readable through refactoring, they didn't seem to have a good way to tell whether they had succeeded. The interviewees didn't rate code metrics high in terms of utility. So, actions were driven more by intuition than metrics. Readability and understandability could also be subjective; for one architect, that meant keeping things simple by reducing the amount of code.

Several developers said that increasing generalization improved code reusability. They saved effort because they needed less added code for each new case. One developer explained that template functions in C++ were a good example of this. According to another developer, splitting classes made the code more modular, enabling more flexible work distribution among teams and work sites, compared with one gigantic class.

Algorithmic performance could trigger refactoring, too. For company H in the industrial-automation domain, optimizing program execution speed was a key driver for refactoring. In addition, the company wanted refactoring to increase reliability and robustness. The refactoring needs were thus potentially domain- and application-specific, and the desired quality attributes weren't always the same.

One architect stated that refactoring was required when developers felt uneasy about extending the code. The code's perceived internal complexity could increase gradually over time, and refactoring could help fight it. Software decay emerged as an issue because it was challenging to foresee in which direction the software would evolve and how to program the code properly when starting development.

Refactoring could also keep the developers' spirits high. Several interviewees noted that giving developers the chance to refactor increased their motivation and satisfaction. Continuous improvement could create an atmosphere in which developers felt generally happier.

7. What Could Possibly Go Wrong?

The interviewees' primary fear was that refactoring might break the code somewhere else. Despite good intentions, refactoring-related changes might introduce defects that go unnoticed no matter how complete the test suites are; a concern that had been shared earlier at Microsoft [7].

An architect we interviewed mentioned that modifying interfaces and method signatures could be particularly challenging when external services depended on the interfaces. Refactoring such code might break the other end and cause harm for the external provider. So, developers should think twice before refactoring; this motivates the need for clear interface deprecation policies that are communicated to all parties. This is in line with previous arguments that underlined the difficulty of changing core services' fundamental characteristics [15].

Several interviewees acknowledged that success wasn't always guaranteed; code structures might not have improved at all after refactoring, or the perceived internal code quality might have actually degraded. An architect from company J stated,

> *One thing could be starting a huge refactoring, if we feel there's a lot of technical debt here. We then start refactoring, which is done for weeks and months, only to eventually notice that it's never going to get fixed, and we have to give up at some point. We realize that we just did a month's work for nothing because we weren't able to fix anything sensible.*

Although restricted to structural and other behavior-preserving changes, refactoring is still programming. Code is added, modified, and deleted, which all takes time. According to the interviewees, it wasn't easy to see beforehand whether this effort would later provide added value in some form. A software architect from company F stated fittingly,

> *It's like a wish that when I refactor, I'll save some time in the future. It's sort of an investment in the future that might never realize.*

The challenge is to improve the code and not spend time on changes from which few people or no one will benefit. When making the code extendable and easy to read, understand, and maintain, developers should somehow be able to predict future needs. This might be why several companies tried to tie refactoring with a functional change.

8. Refactoring Tools and Metrics

We also asked about tool use. We anticipated the tools to fall into two categories: ones that help refactoring and others that indicate the need for refactoring. However, most of the companies had no tools for the latter category.

Several interviewees said that the essential toolset for refactoring was a good version control system and a good set of automated tests on a continuous-integration server. The version control system ensured that refactoring-related changes didn't interfere with the rest of the team. Good automated tests gave some certainty that the changes didn't affect the code's functionality. So, step C (continuous improvement) seemed to be the watershed level. However, some interviewees weren't satisfied with the tests' quality or coverage.

Although continuous-improvement practices support effective refactoring, interviewees were slightly concerned about fitting bigger changes into the practice of committing often to a working codebase. A large nonincremental restructuring of the code could break down the system. However, branching in version control helped achieve such changes. Still, developers usually postponed huge restructurings until the software had stabilized so that no more significant changes were required. At this point, a stable version of the system could be available all the time, and refactoring could be done so that no new versions of the system were published.

The interviewees agreed that no tool could actually help in the refactoring activity itself. Even though developers found refactoring tools in certain IDEs, such as Eclipse, useful, they didn't find them necessary for development. They saw the tools at best as a way to save some time and avoid errors introduced by manual work, but they had little trust in automation.

No good tools for quantifying refactoring needs seemed to exist; the interviewees usually deemed metrics unessential because they appeared to measure nothing useful. One interviewee mentioned that he and his colleagues tried to find suitable metrics but usually ended up in a worse situation because "nothing else mattered than the one number the tool produced." However, the companies involved in safety-related software (A, B, and H) used static code analysis to find problematic parts of the code. Nevertheless, this didn't give them any metrics, just a list of things to fix as soon as possible, like "a continuation of the compiler." This resonates with Gábor Szöke and his colleagues' finding that developers tend to target concrete problems, not improve metrics or remove antipatterns [8].

However, some interviewees felt that tools or metrics for measuring technical debt and the need for refactoring could be useful. In customer projects, metrics for internal quality could help prove the need for refactoring to a reluctant customer. Internal projects might benefit from a mechanism to grade the code and to avoid trivial errors in code quality.

One developer mentioned that the Code Climate tool had been useful. Another interviewee told us he used a code grader for his hobby project and perceived no significant quality increase after refactoring the code from a low grade to the highest one. Still, no interviewee said that his or her current project used some method to grade code quality.

Benefits	Risks
• Easier future development	• Breaking Something
• Understandability	• Causing externally visible changes
• Reuse	• Worsening the code quality
• Improving quality attributes such as performance	• Wasting time and effort
• Boosting morale and motivation	

Figure 1. Refactoring's benefits and risks, according to the interviewees.

9. Tips and Tricks Learned

The interviewees found the following practices and techniques useful in their everyday work.

Establish a safety net. Sensible version control practices and a meaningful test suite are essential to comfortably make changes to working code.

Use revision control to log refactoring operations. This will reveal your refactoring steps, and you can track the rationale of refactoring. If a new feature requires refactoring, do refactoring first, commit it separately, and add the feature later.

Check and revise unit tests before refactoring. Refactoring often invalidates unit tests, hindering testing.

Use code reviews for major refactoring operations.

Break it and fix it. It's often easier to write a new implementation instead of modifying an existing one. Remove the implementation fragments you're going to refactor and write new ones. You need a reliable test suite to test that the functionality matches the previous implementation.

Do small refactoring tasks whenever needed. You can perform minor refactoring on the fly while fixing bugs or adding features. This helps maintain the code base's quality, decreases the slow deterioration of the code, and even improves the code base in long run. Include small refactoring tasks in your task estimations. This lets you maintain the code base's quality and decreases technical debt.

Schedule your refactoring. Don't perform major refactoring tasks right before a release date. If you're using continuous deployment, don't do larger refactoring tasks at the end of the week, to avoid emergency bug fixes on weekends.

Have a dedicated refactoring day each week. Especially during early development, exact requirements are rare, and the optimal solutions for the final product are unknown. Use the first sensible solution, "code with the flow," and move forward, instead of trying to figure out the optimal solution. Spend your refactoring day to go through the code, tidy it up, and refactor the parts that turn out to be poor.

Involve technology-oriented people from the customer organization. This makes it easier to communicate your refactoring needs.

Have small teams working closely together so that communication is easy and refactoring can occur without a major hassle. Exclaiming "Don't touch this component for a couple of hours; I'm doing a major rewrite on it" is sufficient.

To measure refactoring needs, consult the programmers. They live with the code every day; they know its good and bad parts.

10. Conclusions

The interviewees seemed to be well aware of refactoring. However, as we mentioned before, they seemed to have reserved the word for quite large restructurings, and they saw opportunistic refactoring as an integral part of development, not a separate programming mode.

The companies were committed to delivering features, and refactoring was a second-class citizen. The cost of work and pressing deadlines often required postponing or reducing refactoring, much to the developers' dismay. One contributing factor was the difficulty of selling refactoring, especially to business-oriented customer representatives. One interviewee declared that "It is a part of the customer representative's, product owner's, or product manager's competence to buy refactoring." To make matters worse, there was no easy way to give refactoring a return-on-investment metric.

As a whole, the developers didn't use maintainability metrics because they felt they knew the refactoring needs without measurements. They usually knew how to write decent code, so checking coding conventions automatically was unnecessary. If used, code reviews helped catch the most obvious problems.

Most interviewees said that understandability was among refactoring's top three benefits (see Figure 1), but in the end, it seemed that refactoring mostly helped maintain the interviewees' sanity. The developers acknowledged that they planned to add features to the code or maintain it anyway, so it paid off to produce code that was as good as possible in the first place.

Surprisingly, the software's application domain had little effect on refactoring practices. Deadline pressure seemed to weigh more. Even process maturity wasn't a huge determiner of refactoring. The biggest enablers for refactoring were that continuous-improvement practices and automated tests were in place.

In the end, refactoring wasn't a shot in the dark. The developers had an intuition of the code structures that needed fixing, and they appreciated the potential benefits. However, although they had a pretty good understanding about what they were aiming at, there were obstacles that could block the shot. Also, it could be difficult to tell whether the shot was a hit or a miss.

Acknowledgments

TEKES (the Finnish Funding Agency for Innovation) supported this article as part of the N4S (Need for Speed) Program of DIGILE (the Finnish Strategic Centre for Science, Technology and Innovation in the field of ICT and digital business).

References

[1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[2] J. Chen, J. Xiao, Q. Wang, L. J. Osterweil, and M. Li, "Refactoring planning and practice in agile software development: An empirical study," in *Proceedings of the 2014 International Conference on Software and System Process*, ser. ICSSP 2014. New York, NY, USA: ACM, 2014, pp. 55–64. [Online]. Available: http://doi.acm.org/10.1145/2600821.2600829

[3] E. Murphy-Hill and A. Black, "Refactoring tools: Fitness for purpose," *IEEE Software*, vol. 25, no. 5, pp. 38–44, September 2008.

[4] M. Fowler, "Workflows of Refactoring." [Online]. Available: http://martinfowler.com/articles/workflowsOfRefactoring/

[5] M. Stal, "Chapter 3 - refactoring software architectures," in *Agile Software Architecture*, M. A. Babar, A. W. Brown, and I. Mistrik, Eds. Morgan Kaufmann, 2014, pp. 63–82. [Online]. Available: http://www.sciencedirect.com/science/article/pii/B9780124077720000034

[6] M. Lindvall, D. Muthig, A. Dagnino, C. Wallin, M. Stupperich, D. Kiefer, J. May, and T. Kahkonen, "Agile software development in large organizations," *Computer*, vol. 37, no. 12, pp. 26–34, December 2004.

[7] M. Kim, T. Zimmermann, and N. Nagappan, "A field study of refactoring challenges and benefits," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, ser. FSE '12. New York, NY, USA: ACM, 2012, pp. 50:1–50:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393655

[8] G. Szoke, G. Antal, C. Nagy, R. Ferenc, and T. Gyimothy, "Bulk fixing coding issues and its effects on software quality: Is it worth refactoring?" in *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*, September 2014, pp. 95–104.

[9] R. Moser, P. Abrahamsson, W. Pedrycz, A. Sillitti, and G. Succi, "A case study on the impact of refactoring on quality and productivity in an agile team," in *Balancing Agility and Formalism in Software Engineering*, ser. Lecture Notes in Computer Science, B. Meyer, J. Nawrocki, and B. Walter, Eds. Springer Berlin Heidelberg, 2008, vol. 5082, pp. 252–266. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85279-7_20

[10] F. Buschmann, "Gardening your architecture, part 1: Refactoring," *IEEE Software*, vol. 28, no. 4, pp. 92–94, July 2011.

[11] B. Boehm, "Get ready for agile methods, with care," *Computer*, vol. 35, no. 1, pp. 64–69, January 2002.

[12] G. Bavota, B. De Carluccio, A. De Lucia, M. Di Penta, R. Oliveto, and O. Strollo, "When does a refactoring induce bugs? an empirical study," in *Source Code Analysis and Manipulation (SCAM), 2012 IEEE 12th International Working Conference on*, September 2012, pp. 104–113.

[13] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical software engineering*, vol. 14, no. 2, pp. 131–164, 2009.

[14] H. Olsson, H. Alahyari, and J. Bosch, "Climbing the "stairway to heaven" – a mulitiple-case study exploring barriers in the transition from agile development towards continuous deployment of software," in *Software Engineering and Advanced Applications (SEAA), 2012 38th EUROMICRO Conference on*, September 2012, pp. 392–399.

[15] D. Turk, R. France, and B. Rumpe, "Limitations of agile software processes," in *Proceedings of the Third International Conference on Extreme Programming and Flexible Processes in Software Engineering*, ser. XP2002, 2002, pp. 43–46.