A Field Study of Refactoring Challenges and Benefits

Miryung Kim * miryung@ece.utexas.edu
Thomas Zimmermann + tzimmer@microsoft.com
Nachiappan Nagappan + nachin@microsoft.com

* The University of Texas at Austin, TX, USA
+ Microsoft Research, Redmond, WA, USA

ABSTRACT

It is widely believed that refactoring improves software quality and developer productivity. However, few empirical studies quantitatively assess refactoring benefits or investigate developers' perception towards these benefits. This paper presents a field study of refactoring benefits and challenges at Microsoft through three complementary study methods: a survey, semi-structured interviews with professional software engineers, and quantitative analysis of version history data. Our survey finds that the refactoring definition in practice is not confined to a rigorous definition of *semantics-preserving code transformations* and that developers perceive that refactoring involves substantial cost and risks. We also report on interviews with a designated refactoring team that has led a multi-year, centralized effort on refactoring Windows. The quantitative analysis of Windows 7 version history finds that the binary modules refactored by this team experienced significant reduction in the number of inter-module dependencies and post-release defects, indicating a visible benefit of refactoring.

Categories and Subject Descriptors:
D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement—*restructuring*

General Terms: Measurement, Experimentation

Keywords: Refactoring; empirical study; software evolution; component dependencies; defects; churn.

1. INTRODUCTION

It is widely believed that refactoring improves software quality and developer productivity by making it easier to maintain and understand software systems [13]. Many believe that a lack of refactoring incurs technical debt to be repaid in the form of increased maintenance cost [5]. For example, eXtreme Programming claims that refactoring saves development cost [4] and advocates the rule of *refactor mercilessly* throughout the entire project life cycles. On the other hand, there exists a conventional wisdom that software engineers often avoid refactoring, when they are constrained by a lack of resources (e.g., right before major software releases). Some also believe that refactoring does not provide immediate benefit unlike new features or bug fixes.

Recent empirical studies show contradicting evidence on the benefit of refactoring as well. Ratzinger et al. [29] found that, if the number of refactoring edits increases in the preceding time period, the number of defects decreases. On the other hand, Weißgerber and Diehl found that a high ratio of refactoring edits is often followed by an increasing ratio of bug reports [34, 35] and that incomplete or incorrect refactorings cause bugs [14]. In our previous study, we found similar evidence that refactoring edits have a strong temporal and spatial correlation with bug fixes [18].

These contradicting findings motivated us to conduct a field study of refactoring definition, benefits, and challenges in a large software development organization and investigate whether there is a visible benefit of refactoring a large system. In this paper, we address the following research questions: (1) What is the definition of refactoring from developers' perspectives? By refactoring, do developers indeed mean behavior-preserving code transformations or changes to a program structure [23, 13]? (2) What is the developers' perception about refactoring benefits and risks, and in which contexts do developers refactor code? (3) As claimed in the literature, are there visible refactoring benefits such as reduction in the number of bugs, reduction in the average size of code changes after refactoring, and reduction in the number of component dependencies?

To investigate the definition of refactoring in practice and the value perception toward refactoring, we conducted a survey with over three hundred engineers whose check-in comments included a keyword "*refactor**" in the last two years. From our survey participants, we also came to know about a multi-year refactoring effort on Windows. Because Windows is one of the largest, long-surviving software systems within Microsoft and a designated team led an intentional effort of system-wide refactoring, we focused on the case study of Windows. We interviewed the refactoring team and then assessed the impact of the team's refactoring on reduction of inter-module dependencies and post-release defects using Windows 7 version history.

Our field study found the following results:

- The refactoring definition in practice seems to differ from a rigorous academic definition of *behavior-preserving program transformations*. Our survey participants perceived that refactoring involves substantial cost and risks, and they needed various types of tool support beyond automated refactoring within IDEs.

- The interviews with a designated Windows refactoring team provide insights into how system-wide refactoring was carried out in a large organization. The team led a centralized refactoring effort by conducting an analysis of a de-facto dependency structure and by developing custom refactoring support tools and processes.

- The binary modules refactored by the refactoring team had significant reduction in the number of inter-module dependencies and the number of post-release defects. When comparing the top 25% of most frequently refactored binaries against the top 25% of most frequently modified binaries, there is a statistically significant difference in changes to the total number of dependencies. Further, the top 25% of frequently refactored binaries reduced the number of post-release defects 12.2% more than other modified binaries on average.

While there are many anecdotes about the benefit of refactoring, few empirical studies quantitatively assess refactoring benefit. To the best of our knowledge, our study is the first to quantitatively assess the impact of multi-year, system-wide refactoring on inter-module dependencies and post-release defects in a large organization. Our study provides evidence that *refactoring change is likely to be more safe and reliable than regular change in a large system.* Our survey is also the first large-scale investigation into the refactoring definition in practice and the value perception about refactoring from developers' perspectives. Based on our study, we propose future research directions on refactoring—we need to provide various types of tool support beyond automated refactorings in IDEs, such as refactoring-aware code reviews, refactoring cost and benefit estimation, and automated validation of program correctness after refactoring edits.

2. RELATED WORK

Refactoring Definition. While refactoring is defined as behavior-preserving code transformations in the academic literature [23], the de-facto definition of refactoring in practice seems to be very different from such rigorous definition. Fowler catalogs 72 types of structural changes in object oriented programs but these transformations do not necessarily guarantee behavior preservation [13]. In fact, Fowler recommends developers to write test code first before, since these refactorings may change a program's behavior. Murphy-Hill et al. analyzed refactoring logs and found that developers often interleave refactorings with other behavior-modifying transformations [26], indicating that pure refactoring revisions are rare. Johnson's refactoring definition is aligned with these findings—*refactoring improves behavior in some aspects but does not necessarily preserve behavior in all aspects* [16]. Our survey in Section 3 also finds that refactoring is not confined to low-level, semantics-preserving transformations from developers' perspectives.

Quantitative Assessment of Refactoring Benefits. While several prior research efforts have conceptually advanced our understanding of the benefit of refactoring through metaphors, few empirical studies assess refactoring benefits quantitatively. Sullivan et al. first linked software modularity with option theories [32]. A module provides an option to substitute it with a better one without symmetric obligations, and investing in refactoring activities can be seen as purchasing *options* for future adaptability, which will produce benefits when changes happen and the module can be replaced easily. Baldwin and Clark [2] argued that the modularization of a system can generate tremendous value in an industry, given that this strategy creates valuable options for module improvement. Ward Cunningham drew the comparison between debt and a lack of refactoring: a quick and dirty implementation leaves *technical debt* that incur *penalties* in terms of increased maintenance costs [8]. While these projects advanced conceptual understanding of refactoring impact, they do not quantify the benefits of refactoring.

Xing and Stroulia found that 70% of structural changes in Eclipse's evolution history are due to refactorings and existing IDEs lack support for complex refactorings [36]. Dig et al. studied the role of refactorings in API evolution, and found that 80% of the changes that break client applications are API-level refactorings [10]. While these studies focused on the frequency and types of refactorings, they did not focus on how refactoring impacts inter-module dependencies and defects. MacCormack et al. [22] defined modularity metrics and used these metrics to study evolution of Mozilla and Linux. They found that the redesign of Mozilla resulted in an architecture that was significantly more modular than that of its predecessor. However, unlike our study on Windows, their study merely monitored design structure changes in terms of modularity metrics without identifying the modules where refactoring changes are made.

Conflicting Evidence on Refactoring Benefit. Kataoka et al. [17] proposed a refactoring evaluation method that compares software before and after refactoring in terms of coupling metrics. Kolb et al. [20] performed a case study on the design and implementation of existing software and found that refactoring improves software with respect to maintainability and reusability. Moser et al. [24] conducted a case study in an industrial, agile environment and found that refactoring enhances quality and reusability related metrics. Carriere et al.'s case study found the average time taken to resolve tickets decreases after re-architecting the system [7]. Ratzinger et al. developed defect prediction models based on software evolution attributes and found that refactoring related features and defects have an inverse correlation [29]—if the number of refactoring edits increases in the preceding time period, the number of defects decreases. These studies indicated that refactoring positively affects productivity or quality measurements.

On the other hand, several research efforts found contradicting evidence that refactoring may affect software quality negatively. Weißgerber and Diehl found that refactoring edits often occur together with other types of changes and that refactoring edits are followed by an increasing number of bugs [34]. Kim et al. found that the number of bug fixes increases after API refactorings [18]. Nagappan and Ball found that code churn—the number of added, deleted, and modified lines of code—is correlated with defect density [27]— since refactoring often introduces a large amount of structural changes to the system, some question the benefit of refactoring. Görg and Weißgerber detected errors caused by incomplete refactorings by relating API-level refactorings to the corresponding class hierarchy [34].

Because manual refactoring is often tedious and error-prone, modern IDEs provide features that automate the application of refactorings [15, 30]. However, recent research found several limitations of tool-assisted refactorings as well. Daniel et al. found dozens of bugs in the refactoring tools in popular IDEs [9]. Murphy-Hill et al. found that refactoring tools do a poor job of communicating errors and programmers do not leverage them as effectively as they could [26]. Vakilian et al. [33] and Murphy et al. [25] found that programmers do not use some automated refactorings despite their awareness of the availability of automated refactorings.

These contradicting findings on refactoring benefits motivate our survey on the value perception about refactoring. They also motivate our analysis on the relationship between refactoring and inter-module dependencies and defects.

Refactoring Change Identification. A number of existing techniques address the problem of automatically inferring refactorings from two program versions. These techniques compare code elements in terms of their name [36] and structure similarity to identify move and rename refactorings [11]. Prete et al. encode Fowler's refactoring types in template logic rules and use a logic query approach to automatically find complex refactorings from two program versions [28]. A survey of existing refactoring reconstruction techniques is described elsewhere [28]. Kim et al. use the results of API-level refactoring reconstruction to study the correlation between API-level refactorings and bug fixes [18]. While it is certainly possible to identify refactorings using refactoring reconstruction techniques, in our Windows 7 analysis, we identify the branches that a designated refactoring team created to apply and maintain refactorings exclusively and isolate changes from those branches. We believe that our method of identifying refactorings is reliable as a designated team confirmed all refactoring branches manually and reached a consensus about the role of those refactoring branches within the team.

Empirical Studies on Windows. Prior studies on Windows focused on primarily defect prediction. Nagappan and Ball investigated the impact of code churn on defect density and found that relative code churn measures were very effective indicators of code quality [27]. Zimmermann and Nagappan built a system wide dependency graph of Windows Server 2003. By computing network centrality measures, they observed that network measures based on dependency structure were 10% more effective in defect prediction, compared to complexity metrics [37]. More recently, Bird et al. observed that socio-technical network measures combined with dependency measures were stronger indicators of failures than dependency measures alone [6]. Our current study is significantly different from these prior studies by distinguishing *refactoring changes* from *non-refactoring changes* and by focusing on the impact of refactoring on inter-module dependencies and defects.

3. A SURVEY OF REFACTORING PRACTICES

In order to understand refactoring practices at Microsoft, we sent a survey to 1290 engineers whose change comments included the keyword *"refactor*"* in the last 2 years in five Microsoft products: Windows Phone, Exchange, Windows, Office Communication and Services (OCS), and Office. We purposely targeted the engineers who are already familiar with the terms, *refactor, refactoring, refactored, etc.*, because our goal is to understand their own refactoring definition and their perception about the value of refactoring. The survey consisted of 22 multiple choice and free-form questions, which were designed to understand the participant's own refactoring definition, when and how they refactor code, including refactoring tool usage, developers' perception toward the benefits, risks, and challenges of refactoring. Table 1 shows a summary of the survey questions; the full list is available as a technical report [19]. We analyzed the survey responses by identifying the topics and keywords and by tagging individual responses with the identified topics. In total, 328 engineers participated in the survey. 83% of them were developers, 16% of them were test engineers, 0.9% of them were build engineers, and 0.3% of them were program managers. The participants had 6.35 years of experience at Microsoft and 9.74 years of experience in software industry on average with a familiarity with C++, C, and C#.

3.1 What is a Refactoring Definition in Practice?

When we asked, "how do you define refactoring?", we found that developers do not necessarily consider that refactoring is confined to behavior preserving transformations [23]. 78% define refactoring as code transformation that improves some aspects of program behavior such as readability, maintainability, or performance. 46% of developers did not mention preservation of behavior, semantics, or functionality in their refactoring definition at all. This observation is consistent with Johnson's argument [16] that, while refactoring preserves some behavior, it does not preserve behavior in all aspects. The following shows a few examples of refactoring definitions by developers.[1]

"Rewriting code to make it better in some way."

"Changing code to make it easier to maintain. Strictly speaking, refactoring means that behavior does not change, but realistically speaking, it usually is done while adding features or fixing bugs."

When we asked, "how does the abstraction level of Martin Fowler's refactorings or refactoring types supported by Visual Studio match the kinds of refactoring that you perform?", 71% said these basic refactorings are often a part of *larger, higher-level* effort to improve existing software. 46% of developers agree that refactorings supported by automated tools differ from the kind of refactorings they perform manually. In particular, one developer said, the refactorings listed in Table 1 form the minimum granular unit of any refactoring effort, but none are worthy of being called refactoring in and of themselves. The refactorings she performs are larger efforts aimed at interfaces and contracts to reduce software complexity, which may utilize any of the listed low-level refactoring types, but have a larger idea behind them. As another example, a participant said,

"These (Fowler's refactoring types or refactoring types supported by Visual Studio) are the small code transformation tasks often performed, but they are unlikely to be performed alone. There's usually a bigger architectural change behind them."

[1]In the following, each italicized, indented paragraph corresponds to a quote from answers to our survey (Section 3) or interviews (Section 4).

Table 1: Summary of Survey Questions (the full list is available as a technical report [19])

Background	What is your role in your team (i.e., developer, tester, program manager, team lead, dev manager, etc.)? Which best describes your primary work area? How many years have you worked in software industry? Which programming languages are you familiar with?
Definition	How do you define *refactoring*? Which keywords do you use or have you seen being used to mark refactoring activities in change commit messages? How does the abstraction level of Fowler's refactorings such as "Extract Method" match the kinds of refactorings that you often perform?
Context	How many hours per month roughly do you spend on refactoring? How often do you perform refactoring? In which situations do you perform refactorings?
Value-Perception	What benefits have you observed from refactoring? What are the challenges associated with performing refactorings? Based on your own experience, what are the risks involved in refactoring? How strongly do you agree or disagree with each of the following statements? • *Refactoring improves program readability* • *Refactoring introduces subtle bugs* • *Refactoring breaks other people's code* • *Refactoring improves performance* • *Refactoring makes it easier to fix bugs...*
Tools	What tools do you use during refactoring? What percentage of your refactoring is done manually as opposed to using automated refactoring tools? The following lists some of the types of refactorings. Please indicate whether you know these refactorings or used them before. [Multiple choices: (1) usually do this both manually and using automated tools (2) usually do this manually, (3) usually do this using automated tools, (4) know this refactoring type but don't use it, (5) don't know this refactoring type.] • *Rename*, *Extract Method*, *Encapsulate Field*, *Extract Interface*, *Remove Parameters*, ... These refactoring types were selected from Fowler's catalog. How strongly do you agree or disagree with each of the following statements? • *I interleave refactorings with other types of changes that modify external program behavior.* • *Refactorings supported by a tool differ from the kind of refactorings I perform manually.* • *Refactorings that I apply are higher level changes than the ones supported by tools.* • *How do you ensure program correctness after refactoring? ...* Only a few statements are shown in this paper for presentation purposes.

These remarks indicate that the scope and types of code transformations supported by refactoring engines are often too low-level and do not directly match the kinds of refactoring that developer want to make.

3.2 What Are the Challenges Associated with Refactoring?

When we asked developers, "what are the challenges associated with doing refactorings at Microsoft?", 28% of developers pointed out inherent challenges such as working on large code bases, a large amount of inter-component dependencies, the needs for coordination with other developers and teams, and the difficulty of ensuring program correctness after refactoring. 29% of developers also mentioned a lack of tool support for refactoring change integration, code review tools targeting refactoring edits, and sophisticated refactoring engines in which a user can easily define new refactoring types. The difficulty of merging and integration after refactoring often discourages people from doing refactoring. Version control systems that they use are sensitive to rename and move refactoring, and it makes it hard for developers to understand code change history after refactorings. The following quotes describe the challenges of refactoring change integration and code reviews after refactoring:

"Cross-branch integration was the biggest problem. We have this sort of problem every time we fix any bug or refactor anything, although in this case it was particularly painful because refactoring moved files, which prevented cross-branch integration patches from being applicable."

"It (refactoring) typically increases the number of lines/files involved in a check-in. That burdens code reviewers and increases the odds that your change will collide with someone else's change."

Many participants also mentioned that, when a regression test suite is inadequate, there is no safety net for checking the correctness of refactoring. Thus, it often prevents from developers to initiate refactoring effort.

"If there are extensive unit tests, then (it's) great, (one) would need to refactor the unit tests and run them, and do some sanity testing on scenarios as well. If there are no tests, then (one) need to go from known scenarios and make sure they all work. If there is insufficient documentation for scenarios, refactoring should not be done."

In addition to these inherent and technical challenges of refactoring reported by the participants, maintaining backward compatibility often discourages them from initiating refactoring effort.

According to self-reported data, developers do most refactoring manually and they do not use refactoring tools despite their awareness of refactoring types supported by the tools. When we asked, "what percentage of your refactoring is done manually as opposed to using automated refactoring tools?", developers said they do 86% of refactoring manually on average. Surprisingly 51% of developers do all 100% of their refactoring manually. Figure 1 shows the percentages of developers who usually apply individual refactoring types manually despite the awareness and availability of automated refactoring tool support. Considering that 55% of these de-

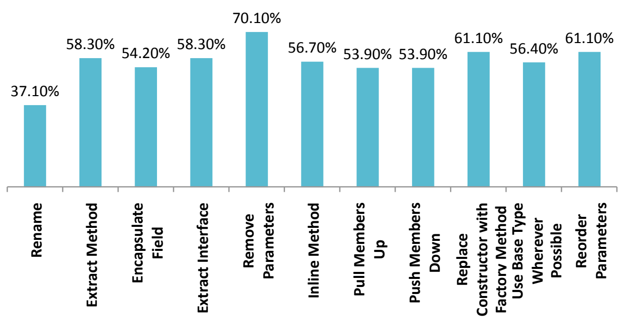

Figure 1: The percentage of survey participants who know individual refactoring types but do those refactorings manually

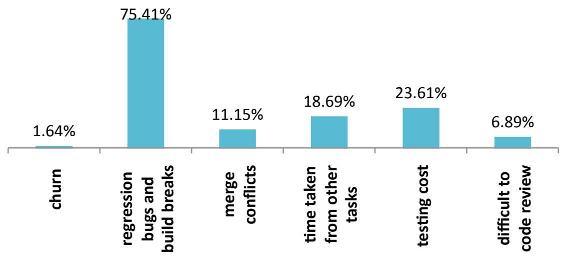

Figure 2: The risk factors associated with refactoring

velopers reported that they have automated refactoring engines available in their development environments, this lack of usage of automated refactoring engines is very surprising. With an exception of rename refactoring, more than a half of the participants said that they apply those refactorings manually, despite their awareness of the refactoring types and availability of automated tool support. This result is aligned with Vakilian et al. [33]. Our survey responses indicate that the investment in tool support for refactoring must go beyond automated code transformation, for example, tool support for change integration, code reviews after refactoring, validation of program correctness, estimation of refactoring cost and benefit, etc.

"I'd love a tool that could estimate the benefits of refactoring. Also, it'd be awesome to have better tools to help figure out who knows a lot about the existing code to have somebody to talk to and how it has evolved to understand why the code was written the way it was, which helps avoid the same mistakes."

"I hope this research leads to improved code understanding tools. I don't feel a great need for automated refactoring tools, but I would like code understanding and visualization tools to help me make sure that my manual refactorings are valid."

"What we need is a better validation tool that checks correctness of refactoring, not a better refactoring tool."

3.3 What Are the Risks and Benefits of Refactoring?

When we asked developers, "based on your experience, what are the risks involved in refactorings?", they reported regression bugs, code churns, merge conflicts, time taken from other tasks, the difficulty of doing code reviews after refactoring, and the risk of over-engineering. Figure 2 summarizes the percentage of developers who mentioned each particular risk factor. Note that the total sum is over 100% as one developer could mention more than one risk factor. 77% of the participants consider that refactoring comes with a risk of introducing subtle bugs and functionality regression; 12% say that code merging is hard after refactoring; and 24% mention increased testing cost.

"The primary risk is regression, mostly from misunderstanding subtle corner cases in the original code and not accounting for them in the refactored code."

"Over-engineering—you may create an unnecessary architecture that is not needed by any feature but all code chunks have to adapt to it."

"The value of refactoring is difficult to measure. How do you measure the value of a bug that never existed, or the time saved on a later undetermined feature? How does this value bubble up to management? Because there's no way to place immediate value on the practice of refactoring, it makes it difficult to justify to management."

When we asked, "what benefits have you observed from refactoring?", developers reported improved maintainability (30%), improved readability (43%), fewer bugs (27%), improved performance (12%), reduction of code size (12%), reduction of duplicate code (18%), improved testability (12%), improved extensibility & easier to add new feature (27%), improved modularity (19%), reduced time to market (5%), etc, as shown in Figure 3.

When we asked, "in which situations do you perform refactorings?" developers reported the symptoms of code that help them decide on refactoring (see Figure 4). 22% mentioned poor readability; 11% mentioned poor maintainability; 11% mentioned the difficulty of repurposing existing code for different scenarios and anticipated features; 9% mentioned the difficulty of testing code without refactoring; 13% mentioned code duplication; 8% mentioned slow performance; 5% mentioned dependencies to other teams' binaries; and 9% mentioned old legacy code that they need to work on. 46% of developers said they do refactoring in the context of bug fixes and feature additions, and 57% of the responses indicate that refactoring is driven by immediate concrete, visible needs of changes that they must implement in a short term, rather than potentially uncertain benefits

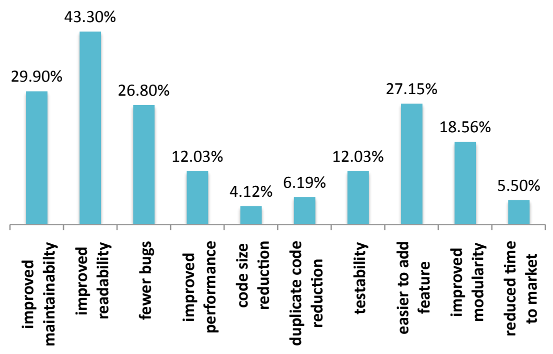

Figure 3: Various types of refactoring benefits that developers experienced

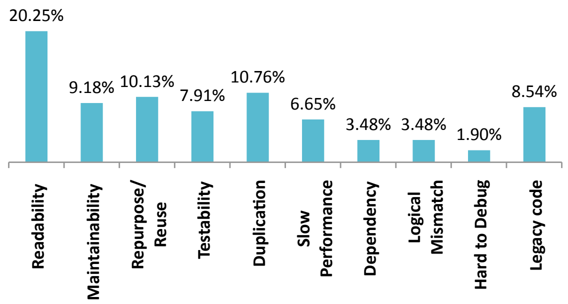

Figure 4: The symptoms of code that help developers initiate refactoring

of long-term maintainability. In addition, more than 95% of developers do refactoring across all milestones and not only in MQ milestones—a period designated to fix bugs and clean up code without the responsibility to add new features. This indicates the pervasiveness of refactoring effort. According to self-reported data, developers spend about 13 hours per month working on refactoring, which is close to 10% of their work, assuming developers work about 160 hours per month.

4. A CASE STUDY OF WINDOWS REFACTORING

In order to examine how the survey respondents' perception matches reality in terms of refactoring and to investigate whether there are visible benefits of refactoring, we decided to conduct follow-up interviews with a subset of the survey participants and to analyze the version history data. In terms of a subject program, we decided to focus on Windows, because it is the largest, long-surviving software system within Microsoft and because we learned from our survey that a designated refactoring team has led an intentional, system-wide refactoring effort for many years.

Section 4.1 describes the results of semi-structured interviews with six key members of the Windows refactoring team, and Section 4.2 describes quantitative assessment of refactoring benefits using the Windows 7 version history.

4.1 Interviews

In this section, we investigate a multi-year refactoring effort made by a designated refactoring team in Windows. To understand the goal of this refactoring project and how the refactoring project was carried out, we conducted one-on-one interviews with six key members of this team. The following describes the role of interview participants. The interviews with the participants were audio-recorded and transcribed later for analysis. The first author of this paper led all interviews.

- Architect (90 minutes)
- Architect / Development Manager (30 minutes)
- Development Team Lead (75 minutes)
- Development Team Lead (85 minutes)
- Developer (75 minutes)
- Researcher (60 minutes)

The interview study results are organized by the questions raised during the interviews.

"What motivated your team to lead this refactoring effort?" The refactoring effort was initiated by a few architects who recognized that a large number of dependencies at the binary module level could be reduced and optimized to make modular reasoning of the system more efficient. The goal of refactoring was to maximize parallel development efficiency, to avoid unwanted parallel change interference, and make it easier to selectively rebuild and retest subsystems effectively.

> *"When we started looking at the system as a whole, there were so many dependencies."*

> [Paraphrased for confidentiality] *"Two different teams working on completely different features could cause a build to break if they do not watch out dependencies during our change integration across branches"*

> *"If X percent of the binaries are at a strongly connected component and you touch one of those things and you have to retest X percent of the binaries again. . . "*

"What are the goals of Windows refactoring?" The refactoring effort was driven by foresights to repurpose the existing system to target different execution environments at a much faster pace than before. Thus, the refactoring effort had a business value of its own. This refactoring also involved breaking inter-module dependencies by moving a set of related functionality (APIs) from one binary to another binary (or new binaries).

> *"So without taking all of [the system] stack they can take just the pieces that they want. So ideally [with] this low layer piece, we could release [a new product] at a much faster cadence, right?"*

> *"The reason we're doing the refactoring is considered as an important reason as the feature itself."*

"How did you carry out system-wide refactorings on a very large system?" The refactoring team made significant effort to analyze the de-facto binary level dependency structure before making refactoring decisions. After the initial analysis of binary level dependencies, the team came up with a layered architecture, where individual binaries were assigned with layer numbers, so that the partial ordering dependency relationships among binaries could be documented and enforced. To help with the analysis of de-facto dependency structure, the team used a new tool called MaX [31]. MaX not only computes binary level dependencies but also can distinguish benign dependency cycles within a layer from undesirable dependencies that go from low-level layers to the layers above. Furthermore, the refactoring team consulted other teams about how to decompose existing functionality into a set of logical sub-groupings (layers).

> *"Our goal was actually (A) to understand the system, and to develop a layered model of the system; and (B) to protect the model programmatically and automatically. So by developing a mathematical model of the entire system that is based on layer numbers and associating binaries with a layer number, we could enforce a partial ordering—that's what we call it, the layer map."*

The refactoring team led centralized architecting effort, while preventing architectural degradation by other teams. The team introduced new *quality gate checks*, which prevented developers from committing code changes that violate the layer architecture constraints to the version control system. Furthermore, the refactoring team educated other teams about how to use new APIs to be compliant with the layered architecture constraints. The refactoring team then refactored the existing system by splitting existing binaries into sub-component binaries or by replacing existing binaries with new binaries.

"It turns out that if you do centrally, you can do a lot of this stuff for other teams. We give them the code, and they have a test team, and their test team evaluates what they need to retest when they take the code. So we do the integration for them..."

"We have this completely automated so when you add a dependency to something, it goes through this entire quality gate process where the binaries get analyzed using whole program analysis techniques."

In addition, the refactoring team created two custom software engineering tools to ease migration of existing binary modules to new binary modules. Similar to how Java allows creation of abstract classes which later can be bound to concrete subclasses, the team created a technology that allows other teams to import an empty header binary for each logical group of API family, which can be later bound to a concrete binary implementation depending on the system configuration. Then a customized loader technology loads an appropriate target binary implementation instead of the empty header binary during the binary loading time. This technology emulates dynamic dispatching in object-oriented programming style and has two benefits. (1) It separates API contracts from API implementations, thus avoiding inclusion of unnecessary binaries in a different execution environment, where only a minimal functionality instead of a full functionality is desired. (2) It enables product-line development—variant products are built by composing different binary implementations. The above technology takes care of switching between two different API implementations during load time, but does not take care of cases where the execution of two different API implementations must be weaved carefully during runtime. To handle such cases, the team systematically inserted program changes to existing code. Such code changes followed a special coding style guideline for better readability and were partially automated by stub code generation functionality.

In summary, we found that the refactoring effort had the following distinctive characteristics:

- The refactoring effort was driven by foresights to re-purpose the existing system to target different execution environments. Thus, the refactoring had a business value of its own.
- The team's refactoring decisions were made after substantial analysis of a de-facto dependency structure.
- The refactoring effort was centralized and top down—the designated team made software changes systematically, integrated the changes to a main source tree, and educated others on how to use new APIs, while preventing architectural degradation by others.
- The refactoring was enabled and facilitated by development of custom refactoring support tools and processes such as MaX and *quality gate check*.

4.2 Quantitative Analysis of Windows 7 Version History

To examine whether the refactoring done by this team had a visible benefit, we analyzed Windows 7 version history data. We examined the impact of refactoring on two specific software measures: *dependencies* and *defects*. Because the primary goal of the Windows refactoring team was to reduce undesirable dependencies between binary modules (as mentioned in Section 4.1), we measured reduction in inter-module dependencies to check whether the refactoring team indeed achieved their goal of dependency reduction. We then focused on the relationship between refactoring and *defects*, because many of our survey participants perceived that refactoring comes with a risk of introducing defects and regression bugs.

We identified the branches that the designated refactoring team created to apply and maintain refactoring changes exclusively and isolated changes from those branches. In Windows, all changes are made to specific branches and later merged to the main trunk. We believe that our method of identifying refactorings is reliable as a designated team confirmed all refactoring branches manually and reached a consensus about the role of those refactoring branches within the team. During Windows 7 development, 1.27% of changes were changes made to the refactoring branches owned by the refactoring team; 98.73% of changes were made to non-refactoring branches. The number of committers who worked on the refactoring branches was 2.04%, while the number of committers on non-refactoring branches was 99.84%. Note that the sum of the two is greater than 100% because some committers work both on refactoring branches and non-refactoring branches. 94.64% of binaries were affected by at least one change from the refactoring branches and 99.05% of binaries were affected by at least one change from non-refactoring branches. In our study, refactored binaries are binaries where at least one change from the refactoring branches is compiled into. For example, if the refactoring team made edits on the refactoring branches to split a single Vista binary into three binaries in Windows 7, we call the three binaries as *refactored binaries* in Windows 7.

4.2.1 Data Collection

For our study we used a binary module level analysis, as this unit is typically used for program analysis within Microsoft and the smallest units to which defects are accurately mapped. Here, a binary refers to an executable file (COM, EXE, etc.) or a dynamic-link library file (DLL) shipped with Windows. Binaries are assembled from several source files and typically form a logical unit, e.g., `user32.dll` may provide programs with functionality to implement graphical user interfaces. A software dependency is a directed relation between two pieces of code such as expressions or methods. There exist different kinds of dependencies: data dependencies between the definition and use of values and call dependencies between the declaration of functions and the sites where they are called. Microsoft has an automated tool called MaX [31] that tracks dependency information at the function level, including calls, imports, exports, RPC, COM, and Registry access. MaX generates a system-wide dependency graph from both native x86 and .NET managed binaries. MaX is used for change impact analysis and for integration testing [31]. For our analysis, we generated a system-wide dependency graph with MaX at the function level. Since binaries are the lowest level of granularity to which defects can be accurately mapped back to, we lifted this graph up to the binary level in a separate post-processing step.

Microsoft records all problems that are reported for Windows in a database. In this study, we measured the changes in the number of post-release defects—defects leading to failures that occurred in the field within six months after the initial releases of Windows Vista or Windows 7. We collected all problem reports classified as non-trivial (in contrast to enhancement requests [1]) and for which the problem was fixed in a later product update. The location of the fix is used as the location of the post-release defect. To understand the impact of Windows 7 refactoring, we compared the number of dependencies and the number of post-release defects at the binary level between Windows Vista and Windows 7.

4.2.2 Where was the refactoring effort focused on?

Figure 5 shows the cumulative ratio for three Windows Vista dependency-related measures for refactored binaries ranked in descending order by the number of refactorings: (1) the cumulative number of outgoing dependencies, i.e., Outgoing Dependencies, (2) the cumulative number of the sum of incoming and outgoing dependencies, i.e., Total Dependencies, and (3) the cumulative number of neighbor binaries connected via dependencies, i.e., Neighbors. Between two binaries, multiple dependencies could exist, but they are counted only once as neighbors. The cumulative ratio for each measure is computed as the relative cumulative sum for its values up to the x-th most **frequently refactored** binary, normalized by the total sum of the values for all N binaries (see the equation below).

$$y = \frac{\sum_{i<x}(Value)}{\sum_{i<N}(Value)} \tag{1}$$

The arrow in Figure 5 indicates that the top 25% most refactored binaries cover 53.09% of all neighbor relationships in Vista for modified binaries. Top 40% most refactored binaries cover 70.51% of all neighbor relationships in Vista for modified binaries. The percentages are similar for total number of dependencies (61.90% for top 25% and 76.06% for top 40% and outgoing dependencies (40.68% for top 25% and 58.26% for top 40%). These results indicate that the refactoring effort was focused on binaries with a large number of dependencies in Vista. This is consistent with what the refactoring team said in the interviews: their goal is to break a large number of unwanted dependencies between certain modules.

4.2.3 Did refactoring reduce binary-level dependencies?

While the total of number of binary level dependencies among modified binaries increased from Windows Vista to Windows 7, most frequently refactored components contributed to reduction of dependencies. Consider the plot in Figure 6, in which the Delta Neighbors (Refactored DLL) line shows the cumulative ratio of the differences in the number of dependency neighbors for the top X percent of most **frequently refactored** binaries. The Delta Neighbors (All Changed DLLs) line shows the cumulative ratio of the differences in the number of dependency neighbors for the top

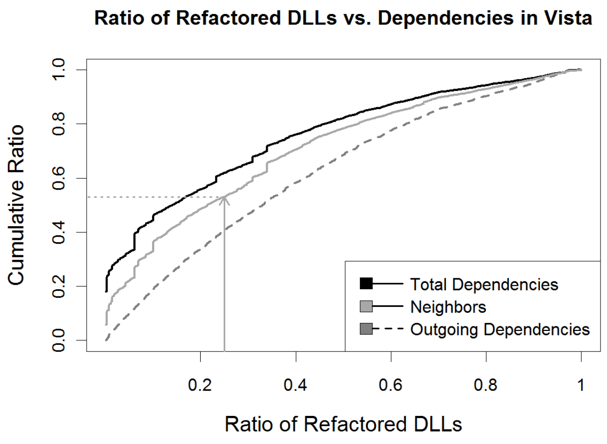

Figure 5: The cumulative ratio of Vista dependencies vs. ratio refactored binaries.

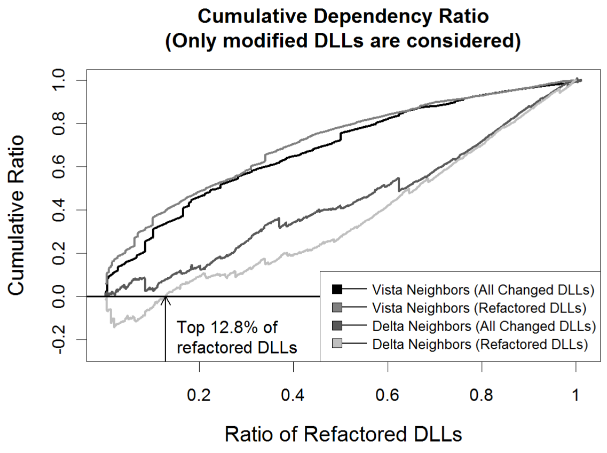

Figure 6: The cumulative ratio of the changes in the number of inter-module dependencies between Vista and Windows 7 vs. ratio refactored binaries.

X percent of most **frequently modified** binaries. Along the x-axis, the refactored binaries are ranked in descending order by the number of refactorings. In other words, the difference between the Delta Neighbors (All Changed DLLs) line and the Delta Neighbors (Refactored DLLs) line is that binaries are ordered by the number of refactoring commits as opposed to the number of regular commits along the x-axis.

The Delta Neighbors (Refactored DLLs) line remains below zero until 0.12, meaning that the top 12% of most frequently refactored binaries did not increase the total number of dependencies. Note that in the left bottom corner, the Delta Neighbors (All Changed DLLs) line stays above the Delta Neighbors (All Refactored DLLs) line, implying that the group of most frequently modified binaries increase the total number of neighbor dependencies more than the group of most refactored binaries. Figure 7 shows a bar chart that compares the changes to the total number of neighbor dependencies per binary from Windows Vista to Windows 7 for the Top 25%, Top 25% to 50%, Top 50% to 75%, and

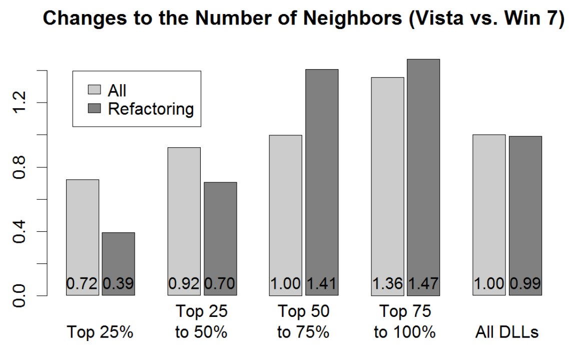

Figure 7: Changes to the total number of neighbors per binary

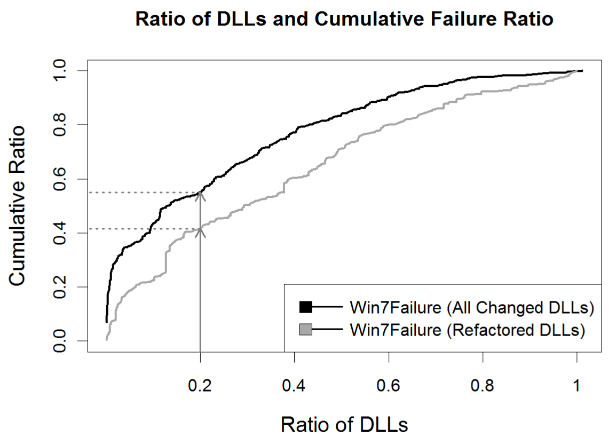

Figure 8: The cumulative ratio of defects in Windows 7 for modified binaries vs. refactored binaries

Top 75% to 100% of most refactored binaries (dark gray) and most modified binaries (light gray); the values are normalized to the total increase for all modified binaries.

For example, if the total number of neighbors per binary increased by 1 on average for modified binaries, the total number of neighbors increased only by 0.39 per binary for the top 25% of most frequently refactored binaries, while the corresponding top 25% of most frequently modified binaries increased by 0.72 on average. The increase is statistically significant at $p<0.0001$ (paired Wilcoxon test). The bar chart indicates that the increase in the number of binary-level dependencies is relatively lower for most frequently refactored binaries than most frequently modified binaries.

4.2.4 *Are refactored binaries more likely defect-prone than non-refactored binaries?*

Figure 8 shows the cumulative ratio for post-release defects in Windows 7 for the binaries with most refactorings (gray line) and most churn (black line)—the churn is computed in terms of the number of regular commits. While the number of Windows 7 post-release defects is slightly correlated with the number of refactorings, the correlation is

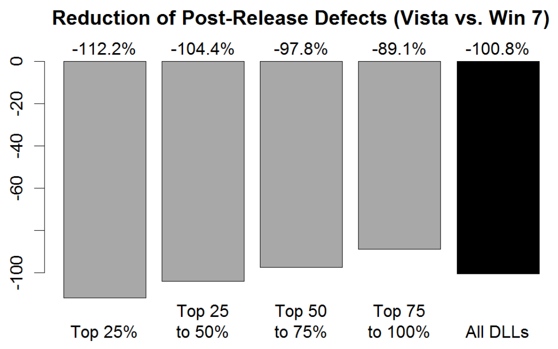

Figure 9: Changes to the total number of post-release defects per binary

weaker than for churn (Spearman correlation of 0.20 vs. 0.30; also indicated by orange line below red line). This implies that *refactoring changes are less likely to lead to post release defects than regular changes.* In other words, while a previous study of defect prediction in Windows by Nagappan and Ball [27] found code churn to be highly correlated with defects and is a good predictor of bugs, refactoring churn is likely to be relatively more safe and reliable than regular churn. The top 20% of most frequently refactored binaries are responsible for 41.6% of Windows 7 post-release defects, while the top 20% of most modified binaries are responsible for 55.0%. The top 40% of most frequently refactored binaries are responsible for 60.3% of Windows 7 post-release defects, while the top 40% of most modified binaries are responsible for 77.2%.

4.2.5 *Did refactoring reduce post release defects more?*

Most binaries that were released in Windows Vista have fewer post-release defects in Windows 7. Figure 9 shows that the top 25% of refactored binaries have 12 percent more reduction in post-release defects compared to all modified binaries. The Spearman correlation between the amount of refactoring and change in post-release defects is 0.368 and statistically significant at $p<0.0001$: the more refactoring changes, the higher the decrease in post-release defects.

We also computed the cumulative ratio of Vista defects vs. ratio of refactored binaries that are ranked in descending order by the number of refactorings. The linear increase in the cumulative number of defects shows that the modules refactored by the refactoring team did not necessarily focus on the modules with a large number of defects in Vista. Yet, looking at Figure 9, we see that these refactored modules experienced significant reduction in the number of defects in Windows 7 compared to Vista. These results indicate that Windows 7 refactoring is correlated with reduction in the number of defects.

5. THREATS TO VALIDITY

Internal validity. Our findings in Section 4 indicate *only* correlation between the refactoring effort and reduction the number of inter-module dependencies and post-release defects, *not* causation—there are other confounding factors such as the expertise level of developers that we did not examine. It is possible that the changes to the number of binary dependencies and post-release defects in Windows 7 are caused by factors other than refactoring such as the types of features added in Windows 7.

Construct validity. Construct validity issues arise when there are errors in measurement. This is negated to an extent by the fact that the entire data collection process of failures and VCS is automated. When selecting target participants for refactoring, we searched all check-ins with the keyword "refactor*" based on the assumption that people who used the word know at least approximately what it means. The definition of refactoring from developers' perspectives is broader than behavior-preserving transformations, and the granularity of refactorings also varies among the participants. For example, some survey participants refer to Fowler's refactorings, while a large number of the participants (71%) consider that refactorings are often a part of larger, higher-level effort to improve existing software. In our Windows case study, we focused on *system-wide* refactoring, because such refactoring granularity seems to be aligned with the refactoring granularity mentioned by a large number of the survey participants.

External validity. In our case, we came to know about a multi-year refactoring effort in Windows from several survey participants and to leverage this best possible scenario where intentional refactoring was performed, we focused on the case study of Windows. As opposed to formal experiments that often have a narrow focus and an emphasis on controlling context variables, case studies test theories and collect data through observation in an *unmodified* setting. While we acknowledge that our case study on Windows may not generalize to other systems, most development practices are similar to those outside of Microsoft. Furthermore, developers at Microsoft are highly representative of software developers all over the world, as they come from diverse educational and cultural backgrounds.[2] We believe that lifting the veil on the Windows refactoring process and quantifying the correlation between refactoring and defect and dependency reduction could be valuable to other development organizations. To facilitate replication our study outside Microsoft, we published the full survey questions as a technical report [19].

6. CONCLUSIONS AND FUTURE WORK

This paper presents a three-pronged view of refactoring in a large software development organization through a survey, interviews, and version history data analysis. To investigate a de-facto definition and the value perception about refactoring in practice, we conducted a survey with over three hundred professional software engineers. Then to examine whether the survey respondents' perception matches reality and whether there are visible benefits of refactoring, we interviewed a subset of engineers who led the Windows refactoring effort and analyzed Windows 7 version history data.

Our study finds the definition of refactoring in practice is broader than *behavior-preserving program transformations*. Developers perceive that refactoring involves substantial cost and risks and they need various types of refactoring support beyond automated refactoring within IDEs. Our case study of Windows shows how system-wide refactoring was carried out in a very large organization. The quantitative analysis of Windows 7 version history shows refactored modules experienced higher reduction in the number of inter-module dependencies and post-release defects than other changed modules. Our study is one of the first to show that *refactoring changes are likely to be relatively more reliable than regular changes in a large system.* Based on our study, we propose future research directions such as tool support for refactoring-aware code reviews, refactoring cost and benefit estimation, etc.

[2]Global diversity and inclusion `http://www.microsoft.come/about/diversity/en/us/default.aspx`

Acknowledgements

Thanks to Galen Hunt, Tom Ball, Chris Bird, Mike Barnett, Rob DeLine, and Andy Begel for their insightful comments. Thanks to the Microsoft Windows refactoring team for their help in understanding the data. Thanks to many managers and developers who volunteered their time to participate in our research. Miryung Kim performed this work as a visiting researcher at Microsoft Research. This work was in part supported by National Science Foundation under the grants CCF-1149391, CCF-1117902, and CCF-1043810, and by a Microsoft SEIF award.

7. REFERENCES

[1] G. Antoniol, K. Ayari, M. Di Penta, F. Khomh, and Y.-G. Guéhéneuc. Is it a bug or an enhancement?: a text-based approach to classify change requests. In *CASCON '08: Proceedings of the conference of the center for advanced studies on collaborative research: meeting of minds*, pages 304–318. ACM, 2008.

[2] C. Y. Baldwin and K. B. Clark. Design rules: The power of modularity volume 1. Cambridge, MA, USA, 1999. MIT Press.

[3] V. Basili, F. Shull, and F. Lanubile. Building knowledge through families of experiments. *IEEE Transactions on Software Engineering*, 25(4):456–473, Jul/Aug 1999.

[4] K. Beck. *extreme Programming explained, embrace change.* Addison-Wesley Professional, 2000.

[5] L. A. Belady and M. Lehman. A Model of Large Program Development. *IBM Systems Journal*, 15(3):225–252, 1976.

[6] C. Bird, N. Nagappan, H. Gall, B. Murphy, and P. Devanbu. Putting it all together: Using socio-technical networks to predict failures. In *Proceedings of the 20th International Symposium on Software Reliability Engineering*, pages 109–119. IEEE Computer Society, 2009.

[7] J. Carriere, R. Kazman, and I. Ozkaya. A cost-benefit framework for making architectural decisions in a business context. In *ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering*, pages 149–157. ACM, 2010.

[8] W. Cunningham. The wycash portfolio management system. In *OOPSLA '92: Addendum to the proceedings on Object-oriented programming systems, languages, and applications*, pages 29–30, New York, NY, USA, 1992. ACM.

[9] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *ESEC-FSE '07: Proceedings of the the 6th joint*

meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pages 185–194, New York, NY, USA, 2007. ACM.

[10] D. Dig and R. Johnson. The role of refactorings in api evolution. In *ICSM '05: Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 389–398, Washington, DC, USA, 2005. IEEE Computer Society.

[11] D. Dig and R. Johnson. Automated detection of refactorings in evolving components. In *ECOOP '06: Proceedings of European Conference on Object-Oriented Programming*, pages 404–428. Springer, 2006.

[12] B. Flyvbjerg. Five misunderstandings about case-study research. *Qualitative inquiry*, 12(2):219–245, 2006.

[13] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2000.

[14] C. Görg and P. Weißgerber. Error detection by refactoring reconstruction. In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005. ACM Press.

[15] W. G. Griswold. *Program Restructuring as an Aid to Software Maintenance*. PhD thesis, University of Washington, 1991.

[16] R. Johnson. Beyond behavior preservation. Microsoft Faculty Summit 2011, Invited Talk, July 2011.

[17] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *International Conference on Software Maintenance*, pages 576–585, 2002.

[18] M. Kim, D. Cai, and S. Kim. An empirical investigation into the role of refactorings during software evolution. In *ICSE' 11: Proceedings of the 2011 ACM and IEEE 33rd International Conference on Software Engineering*, 2011.

[19] M. Kim, T. Zimmermann, and N. Nagappan. Appendix to a field study of refactoring rationale, benefits, and challenges at microsoft. Technical Report MSR-TR-2012-4, Microsoft Research, 2012.

[20] R. Kolb, D. Muthig, T. Patzke, and K. Yamauchi. Refactoring a legacy component for reuse in a software product line: a case study: Practice articles. *J. Softw. Maint. Evol.*, 18:109–132, March 2006.

[21] A. Kuper and J. Kuper, editors. *The Social Science Encyclopedia*. Routledge, 1985.

[22] A. MacCormack, J. Rusnak, and C. Y. Baldwin. Exploring the structure of complex software designs: An empirical study of open source and proprietary code. volume 52, pages 1015–1030, 2006.

[23] T. Mens and T. Tourwe. A survey of software refactoring. *IEEE Transactions on Software Engineering*, 30(2):126–139, 2004.

[24] R. Moser, A. Sillitti, P. Abrahamsson, and G. Succi. Does refactoring improve reusability? In *ICSR*, pages 287–297, 2006.

[25] G. C. Murphy, M. Kersten, and L. Findlater. How are java software developers using the eclipse ide? volume 23, pages 76–83, Los Alamitos, CA, USA, July 2006. IEEE Computer Society Press.

[26] E. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. In *ICSE '09: Proceedings of the 31st International Conference on Software Engineering*, pages 287–297, Washington, DC, USA, 2009. IEEE Computer Society.

[27] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 284–292, New York, NY, USA, 2005. ACM.

[28] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim. Template-based reconstruction of complex refactorings. In *2010 IEEE International Conference on Software Maintenance*, pages 1 –10, 2010.

[29] J. Ratzinger, T. Sigmund, and H. C. Gall. On the relation of refactorings and software defect prediction. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 35–38, New York, NY, USA, 2008. ACM.

[30] D. Roberts, J. Brant, and R. Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, 1997.

[31] A. Srivastava, J. Thiagarajan, and C. Schertz. Efficient Integration Testing using Dependency Analysis. Technical Report MSR-TR-2005-94, Microsoft Research, 2005.

[32] K. Sullivan, P. Chalasani, and V. Sazawal. Software design as an investment activity: A real options perspective. Technical report, 1998.

[33] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. Use, disuse, and misuse of automated refactorings. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 233 –243, june 2012.

[34] P. Weißgerber and S. Diehl. Are refactorings less error-prone than other changes? In *MSR '06: Proceedings of the international workshop on Mining software repositories*, pages 112–118. ACM, 2006.

[35] P. Weißgerber and S. Diehl. Identifying refactorings from source-code changes. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pages 231–240, Washington, DC, USA, 2006. IEEE Computer Society.

[36] Z. Xing and E. Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *ASE '05: Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, pages 54–65, New York, NY, USA, 2005. ACM.

[37] T. Zimmermann and N. Nagappan. *Predicting defects using network analysis on dependency graphs*. ICSE '08. ACM, New York, NY, USA, 2008.

APPENDIX

Our complete refactoring survey questions are available as a Microsoft Research technical report, MSR-TR-2012-4:`http://http://research.microsoft.com/apps/pubs/?id=157637`.