An Empirical Study of Refactoring Challenges and Benefits at Microsoft

Miryung Kim, *Member, IEEE,* Thomas Zimmermann, *Member, IEEE,* Nachiappan Nagappan, *Member, IEEE*

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 the top 5% of preferentially refactored modules experience higher reduction in the number of inter-module dependencies and several complexity measures but increase size more than the bottom 95%. This indicates that measuring the impact of refactoring requires multi-dimensional assessment.

Index Terms—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 [1]. Many believe that a lack of refactoring incurs technical debt to be repaid in the form of increased maintenance cost [2]. For example, eXtreme Programming claims that refactoring saves development cost and advocates the rule of *refactor mercilessly* throughout the entire project life cycles [3]. 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 [4].

Recent empirical studies show contradicting evidence on the benefit of refactoring as well. Ratzinger et al. [5] found that, if the number of refactorings 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 is often followed by an increasing ratio of bug reports [6], [7] and that incomplete or incorrect refactorings cause bugs [8]. We also found similar evidence that there exists a strong correlation between the location and timing of API-level refactorings and bug fixes [9].

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 that modify a program structure [1], [10]? (2) What is the developers' perception about refactoring benefits and risks, and in which contexts do developers refactor code? (3) 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 answer these questions, we conducted a survey with 328 professional software engineers whose check-in comments included a keyword "*refactor**". 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 interviewed the refactoring team of Windows. Using the version history, we then assessed the impact of refactoring on various software metrics such as defects, inter-module dependencies, size and locality of code changes, complexity, test coverage, and people and organization related metrics.

To distinguish the impact of refactoring vs. regular changes, we define the degree of *preferential refactoring*—applying refactorings more frequently to a module, relative to the frequency of regular changes. For example, if a module is ranked at the 5th in terms

- *M. Kim is with the Department of Electrical and Computer Engineering at the University of Texas at Austin.*
- *T. Zimmermann and N. Nagappan are with Microsoft Research at Redmond.*

of regular commits but ranked the 3rd in terms of refactoring commits, the rank difference is 2. This positive number indicates that, refactoring is preferentially applied to the module relative to regular commits. We use the rank difference measure specified in Section 4.4 instead of the proportion of refactoring commits out of all commits per module, because the preferential refactoring measure is less sensitive to the total number of commits made in each module. We then investigate the relationship between preferential refactoring and software metrics. In terms of a threshold, we contrast the top 5% preferentially refactored modules against the bottom 95%, because the top 5% modules cover most refactoring commits—over 90%. We use both bivariate correlation analysis and multivariate regression analysis to investigate how much different development factors may impact the decision to apply refactoring and how these factors contribute to reduction of inter-module dependencies and defects [11].

Our field study makes the following contributions:

- 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.
- To collect refactoring data from version histories, we explore two separate methods. First, we isolate refactoring commits based on branches relevant to refactoring tasks. Second, we analyze check-in comments from version histories based on the refactoring related keywords identified from our survey.
- The top 5% of preferentially refactored modules decrease the number of dependencies by a factor of 0.85, while the rest increases it by a factor of 1.10 compared to the average number of dependency changes per modules.
- The top 5% of preferentially refactored modules decrease post-release defects by 7% less than the rest, indicating that defect reduction cannot be contributed to the refactoring changes alone. It is more likely that the defect reduction in Windows 7 is enabled by both refactoring and non-refactoring changes in the modified modules.
- We also collect data and conduct statistical analysis to measure the impact of refactoring on the size and locality of code changes, test coverage metrics, and people and organization related metrics, etc. The study results indicate that the refactoring effort was preferentially focused on the modules that have relatively low churn measures, have higher test block coverage and relatively few developers worked on in Windows Vista. Top 5% of preferentially refactored modules experience a greater rate of reduction in certain complexity measures, but increases LOC and crosscutting changes more than other modules.[1]

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 various software metrics in a large organization. Consistent with our interview study, refactoring was preferentially applied to modules with a large number of dependencies and preferential refactoring is correlated with reduction in the number of dependencies. Preferentially refactored modules have higher test adequacy and they experience defect reduction; however, this defect reduction cannot be attributed to the role of refactoring changes alone according to our regression analysis. Preferentially refactored modules experience higher reduction in several complexity measures but increase size more than the bottom 95%. This indicates that measuring the impact of refactoring requires multi-dimensional assessment.

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. As the benefit of refactoring is multi-dimensional and not consistent across various metrics, we believe that managers and developers can benefit from automated tool support for assessing the impact of refactoring on various software metrics.

2 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 [12]. We analyzed the survey responses by identifying the topics

1. A module unit in Windows is a `.dll` or `.exe` component and its definition appears in Section 4.

and keywords and by tagging individual responses with the identified topics. The first author did a two pass analysis by first identifying emerging categories and next by tagging individual answers using the categories. Suppose that a developer answered the following to the question, *"Based on your own experience, what are the risks involved in refactoring?"* We then tagged the answer with categories, *regression bugs and build breaks*, *merge conflicts*, and *time taken from other tasks*, which emerged from the participant's answer.

> *"Depending on the scope of the refactoring, it can be easy to unintentionally introduce subtle bugs if you aren't careful, especially if you are deliberately changing the behavior of the code at the same time. Other risks include making it difficult to merge changes from others (especially troublesome because larger refactoring typically takes a significant amount of time during which others are likely to make changes to the same code), and making it difficult for others to merge with you (effectively spreading out the cost of the merge to everyone else who made changes to the same code)."*

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#.

2.1 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 [10]. 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 [13] 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.[2]

> *"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."*

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 developers want to make.

2.2 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 [14]. 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 [15]. 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.

2. In the following, each italicized, indented paragraph corresponds to a quote from answers to our survey (Section 2) or interviews (Section 3).

TABLE 1
Summary of Survey Questions (The full list is available as a technical report [12].)

Background	What is your role in your team (i.e., developer, tester, program manager, team lead, dev manager, etc.)? (multiple choice) Which best describes your primary work area? (multiple choice) How many years have you worked in software industry? (simple answer) Which programming languages are you familiar with? (multiple choice)
Definition	How do you define *refactoring*? (open answer, max characters: 2000) Which keywords do you use or have you seen being used to mark refactoring activities in change commit messages? (open answer, max characters: 2000) How does the abstraction level of Fowler's refactorings such as "Extract Method" and "Use Base Type Whenever Possible" match the kinds of refactorings that you often perform? (open answer, max characters: 2000)
Context	How many hours per month roughly do you spend on refactoring? (min number 0 to max number 160) How often do you perform refactoring? (multiple choice: daily, weekly, monthly, yearly, seldom, never) In which situations do you perform refactorings? (open answer, max characters: 2000)
Value-Perception	What benefits have you observed from refactoring? (open answer, max characters: 2000) What are the challenges associated with performing refactorings? (open answer, max characters: 2000) Based on your own experience, what are the risks involved in refactoring? (open answer, max characters: 2000) How strongly do you agree or disagree with each of the following statements? (scale: strongly agree, agree, neither agree or disagree, disagree, strongly disagree, no response) • *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? (open answer: max characters: 2000) What percentage of your refactoring is done manually as opposed to using automated refactoring tools? (min number 0 to max number 100) The following lists some of the types of refactorings. Please indicate whether you know these refactorings or used them before. [multiple choice: (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? (scale: strongly agree, agree, neither agree or disagree, disagree, strongly disagree, no response) • *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.
	If you would like to be informed about the results of this research, please enter your alias in the following box. (max characters: 256) If you would be willing to participate in a follow-up interview (15 minutes) to share your perspective and anecdotes on refactoring at Microsoft, please enter your alias in the following box. (max characters: 256) If you have any other comments on this survey, please write them in the following text box. (max characters: 2000)

"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 developers 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. [16]. 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."

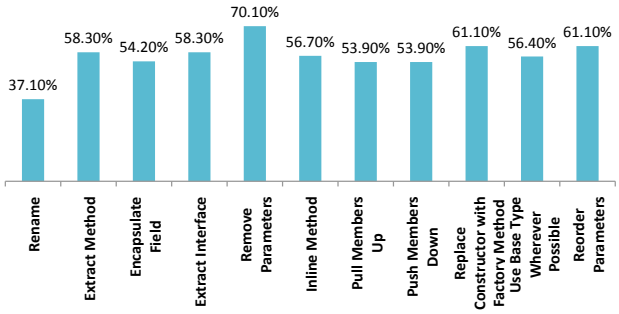

Fig. 1. The percentage of survey participants who know individual refactoring types but do those refactorings manually

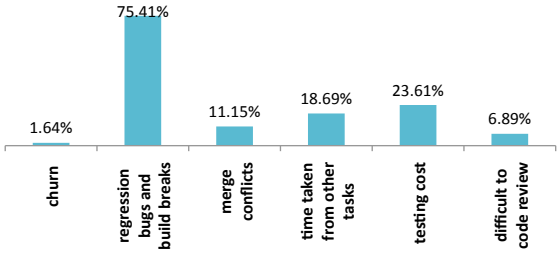

Fig. 2. The risk factors associated with refactoring

"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."

2.3 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. 76% of the participants consider that refactoring comes with a risk of introducing subtle bugs and functionality regression; 11% 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."

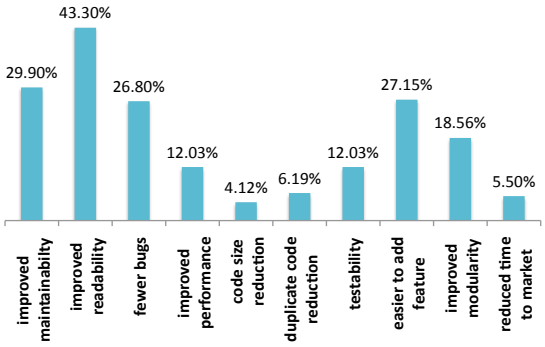

Fig. 3. Various types of refactoring benefits that developers experienced

When we asked, "what benefits have you observed from refactoring?", developers reported improved maintainability, improved readability, fewer bugs, improved performance, reduction of code size, reduction of duplicate code, improved testability, improved extensibility & easier to add new feature, improved modularity, reduced time to market, 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' modules; 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 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.

3 Interviews with the Windows Refactoring Team

In order to examine how the survey respondents' perception matches reality in terms of refactoring and to

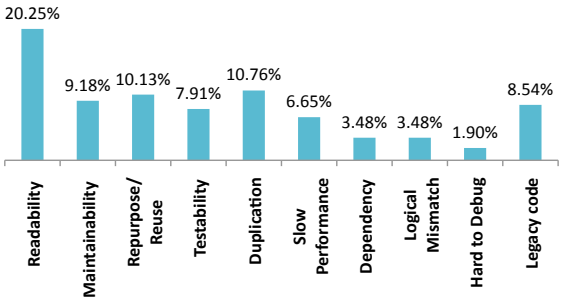

Fig. 4. The symptoms of code that help developers initiate refactoring

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.

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. The first author spent two weeks to categorize the interview transcripts in terms of refactoring motivation, intended benefits, process, and tool support. She then discussed the findings with the sixth subject, (a researcher who is familiar with the Windows refactoring project and collaborated with the refactoring team) to check her interpretation.

- 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 module level could be reduced and optimized to make modular reasoning of the system more efficient, to maximize parallel development efficiency, to avoid unwanted parallel change interference, and to selectively rebuild and retest subsystems effectively.

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

"How did you carry out system-wide refactorings on a very large system?" The refactoring team analyzed the de-facto module level dependency structure before making refactoring decisions. After the initial analysis of module level dependencies, the team came up with a layered architecture, where individual modules were assigned with layer numbers, so that the partial ordering dependency relationships among modules could be documented and enforced. To help with the analysis of de-facto dependency structure, the team used a new tool called MaX [17]. MaX not only computes module 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. 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 modules with a layer number, we could enforce a partial ordering—that's what we call it, the layer map."*

The team introduced *quality gate checks*, which prevented developers from committing code changes that violate the layer architecture constraints to the version control system. The refactoring team then refactored the existing system by splitting existing modules into subcomponent modules or by replacing existing modules with new modules.

They created two custom tools to ease migration of existing modules to new 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 module for each logical group of API family, which can be later bound to a concrete module implementation depending on the system configuration. Then a customized loader loads an appropriate target module implementation instead of the empty header module during the module loading time. This separates API contracts from API implementations, thus avoiding inclusion of unnecessary modules in a different execution environment, where only a minimal functionality instead of a full functionality is desired. 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 team's refactoring decisions were made after substantial analysis of a de-facto dependency struc-

ture.

- 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 Quantitative Analysis of Windows 7 Version History

To examine whether the refactoring done by this team had a visible benefit, we analyze Windows 7 version history data. We first generate hypotheses based on our survey and interview study findings. We then conduct statistical analysis using the software metrics data collected from version histories.

4.1 Study Hypotheses

As software functionality varies for different projects and the expertise level of developers who work on the projects varies across different organizations, our study goal is to contrast the impact and characteristics of refactoring changes against that of non-refactorings in the same organization and the same project. In other words, we compare the characteristics of refactoring vs. non-refactoring vs. all changes, noted as *refactoring churn*, *non-refactoring churn*, and *regular churn* (i.e., the union of refactorings and non-refactorings).

We generate study hypotheses based on our qualitative study findings. These hypotheses are motivated by our survey and interviews, as well as the refactoring literature. The hypotheses are described in Table 2 and the following subsections discuss our data collection and analysis method and corresponding findings.

- *H1 (Dependency)*: We investigate the relationship between refactoring and dependency because our interview study indicates that the primary goal of Windows refactoring is to reduce undesirable inter-module dependencies (i.e. the number of neighbor modules connected via dependencies).
- *H2 (Defect)*: We investigate the relationship between refactoring and defect because many of our survey participants perceive that refactoring comes with a risk of introducing defects and regression bugs.
- *H3 (Complexity)*: The hypotheses on complexity are motivated by prior studies on technical debt [18]–[23].
- *H4 (Size, Churn and Locality)*: The hypotheses on size, churn, and locality are motivated by the fact that developers often initiate refactoring to improve changeability and maintainability [24] and that crosscutting concerns pose challenges in evolving software systems [25]–[28].
- *H5 (Developer and Organization)*: The hypotheses on organizational characteristics are motivated by the fact that the more people who touch the code, the higher the chance of code decay and the higher need of coordination among the engineers, calling for refactoring of the relevant modules [29].
- *H6 (Test Coverage)*: We investigate the hypothesis on test adequacy because our survey respondents said, *"If there are extensive unit tests, then (it's) great. If there are no tests or there is insufficient documentation for test scenarios, refactoring should not be done."*
- *H7 (Layer)*: We investigate the hypothesis on a layered architecture to confirm our interview findings that the refactoring team split core modules and moved reusable functionality from upper layers to lower layers to repurpose Windows for different execution environments.

4.2 Data Collection: Identifying Refactoring Commits

To identify refactoring events, we use two separate methods respectively. First, we identify refactoring-related branches and isolate changes from those branches. Second, we identify refactoring related keywords and mine refactoring commits from Windows 7 version history by searching for these keywords in the commit logs [30].

Refactoring Branch Identification. In many development organizations, changes are made to specific branches and later merged to the main trunk. For example, the Windows refactoring team created certain branches to apply refactoring exclusively. So we asked the team to classify each branch as a refactoring vs. non-refactoring branch. We believe that our method of identifying refactorings is reliable because the 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 are changes made to the refactoring branches owned by the refactoring team; 98.73% of changes are made to non-refactoring branches. The number of committers who worked on the refactoring branches is 2.04%, while the number of committers on non-refactoring branches is 99.84%. Please 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 modules are affected by at least one change from the refactoring branches, and 99.05% of modules are affected by at least one change from non-refactoring branches. In our study, refactored modules are modules 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 module into three modules in Windows 7, we call the three modules as *refactored modules* in Windows 7.

Mining Commit Logs. To identify refactoring from version histories, we look for commit messages with

TABLE 2
A summary of study hypotheses and results

Modularity	H1.A: Refactoring was preferentially applied to the modules with a large number of inter-module dependencies.	Confirmed
	H1.B: Preferential refactoring is correlated to changes in the number of inter-module dependencies.	Confirmed
Defect	H2.A: Refactoring was *not* preferentially applied to the modules with a large number of post-release defects.	Confirmed
	H2.B: Preferential refactoring is correlated to reduction in the number of defects.	Rejected
Complexity	H3.A: Refactoring was preferentially made to the modules with high complexity.	Rejected
	H3.B: Preferential refactoring is correlated with reduction in complexity.	Rejected
Size	H4.A: Refactoring was preferentially applied to the modules with large size and preferential refactoring is correlated with size reduction.	Rejected
Churn	H4.B: Refactoring was preferentially applied to the modules where a large number of edits or commits, and preferential refactoring is correlated with the decrease in churn measures.	Rejected
Locality	H4.C: Refactoring was preferentially applied to the modules where logical changes tend to be crosscutting and scattered, and preferential refactoring is correlated with the decrease in the number of crosscutting changes.	Rejected
Developer and Organization	H5.A: Refactoring was preferentially applied to the modules touched by a large number of developers.	Rejected
	H5.B: Refactoring was preferentially applied to the modules that are not cohesive in terms of organizational contributions.	Confirmed
	H5.C: Refactoring was preferentially applied to the modules that are diffused in terms of organizations and developer contribution.	Confirmed
Test Coverage	H6: Refactoring was preferentially applied to the modules with high test adequacy.	Confirmed
Layer	H7: Preferential refactoring is correlated with reduction in the layer number.	Rejected

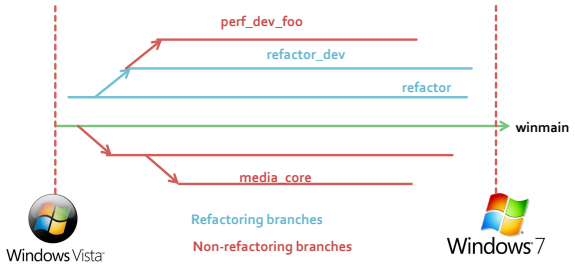

Fig. 5. Categorization of all Windows 7 commits into refactoring vs. non-refactorings based on branches.

certain keywords. In our survey, we asked the survey participant, *"which keywords do you use or have you seen being used to mark refactoring activities in change commit messages?* Based on the responses, we identify a list of top ten keywords that developers use or have seen being used to mark refactoring events: *refactor (254), clean-up (42), rewrite (22), restructure (15), redesign (15), move (15), extract (11), improve (9), split (7), reorganize (7), rename(7) out of 328 responses*. By matching the keywords against the commit messages, we detected refactoring commits from version histories.

According to this method, 5.76% of commits are refactoring, while 94.29% of commits are non-refactoring. The number of committers for the refactoring changes is 50.74%, while the number is 98.56% for non-refactoring. 95.07% of modules are affected by at least one refactoring commit, and 99.92% of modules are affected by at least one non-refactoring.

4.3 Data Collection: Software Metrics

This section discusses our data collection method for defect, dependency, and developers metrics. For other categories, Table 3 clarifies how the data are collected.

Dependencies. For our study, we analyzed dependencies at a module level. Here, a module refers to an executable file (COM, EXE, etc.) or a dynamic-link library file (DLL) shipped with Windows. Modules 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. This module unit is typically used for program analysis within Microsoft and the smallest units to which defects are accurately mapped. 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 [17] 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 modules. MaX is used for change impact analysis and for integration testing [17]. For our analysis, we generated a system-wide dependency graph with MaX at the function level. Since modules are the lowest level of granularity to which defects can be accurately mapped back to, we lifted this graph up to the module level in a separate post-processing step.

Defects. 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 [33]) 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 module level between Windows Vista and Windows 7.

Developers and Organization. We use committer information extracted from version control systems. Based on the Windows product organization's structure and

TABLE 3
Software metrics to be used for data collection and statistical analysis

Size	lines of code (LOC), # of classes, # of parameters, # of local variables, # of functions, # of blocks
Churn	total churn—added, deleted, and modified LOC frequency—# edits or # check-ins that account for churn relative churn as normalized values obtained during the development process # of changed files
Complexity	fan in, fan out, cyclomatic complexity [31], inheritance depth, nested block depth coupling—coupling through member variables, function parameters, classes defined locally in class member function bodies, immediate base classes, and return type
Organization and People	
(people)	engineers (NOE)—The number of engineers who wrote/contributed code to a module. ex-engineers (NOEE)—The number of engineers who no longer work in the organization.
(cohesiveness of ownership)	depth of master ownership (DMO)—The level in the organization structure of an organization at which the ownership of a module is determined/attributed to a particular engineer. percentage of organization contributing to development (PO) —the ratio of the number of people reporting at the DMO level owner relative to the master owner organization size. level of organizational code ownership (OCO)—the percent of edits from the organization that contains the module owner of if there is no owner, then the organization that made the majority of the edits to that module
(diffusion of contribution)	overall organization ownership (OOW)—the ratio of the percentage of people at the DMO level making edits to a module relative to total engineers editing the module organization intersection factor (OIF)—a measure of the number of different organizations that contribute to the grater than 10% of edits The seven organization and people measures are from Nagappan et al.'s study on the relationship between the organizational structure and software quality [32].
Test Coverage	block coverage—A basic block is a set of contiguous instructions in the physical layout of a module that has exactly one entry point and one exit point. Calls, jumps, and branches mark the end of a block. A block typically consists of multiple machine-code instructions. The number of blocks covered during testing constitutes the block coverage measure. arc (branch) coverage—Arcs between blocks represent the transfer of control between basic blocks due to conditional and unconditional jumps, as well as due to control falling through from one block to another. Similar to block coverage the proportion of arcs covered in a module constitute the arc coverage.
Defect	post-release failures—the count of the number of fixes that were mapped back to components after the products were released for a time period of the first six months.

committer information, we measure seven metrics that represent the number of contributors, the cohesiveness of ownership, and the diffusion of contribution used by Nagappan et al. These metrics are summarized in Table 3 and the detailed description and data collection method of these measures are available elsewhere [32].

4.4 Analysis Method: Preferential Refactoring

To distinguish the role of refactoring vs. regular changes, we define the degree of *preferential refactoring*—applying refactorings more frequently to a module, relative to the frequency of regular changes.

> To measure the degree of ***preferential refactoring*** for each module m, we use the following rank difference measure, defined as:
>
> $$all\_commit\_rank(m) - ref\_commit\_rank(m) \quad (1)$$
>
> where $all\_commit\_rank(m)$ is the rank of module m among all modified modules in terms of the commit count and $ref\_commit\_rank(m)$ is the rank of module m among all modified modules in terms of the refactoring commit count.

This notion of ***preferential refactoring*** is used throughout the later subsections to distinguish the impact of refactoring vs. regular changes. For example, if a module is ranked at the 5th in terms of regular commits but ranked the 3rd in terms of refactoring commits, the rank difference is 2. This positive number indicates that, refactoring is preferentially applied to the module relative to regular commits. We use the rank difference measure Eq(1), instead of the proportion of refactoring commits out of all commits per module, defined as $\frac{refactoring\_commit\_count(m)}{all\_commit\_count(m)}$, because this proportion measure is very sensitive to the total number of commits made to each module and because the modules with very few changes pose a significant noise.

We then sort the modules based on the rank difference in descending order and contrast the characteristics of the top 5% group against the characteristics of the bottom 95% group. The reason why we choose a particular threshold of top 5% instead of top n% is that the top 5% modules with most refactoring commits account for 90% of all refactoring commits—in other words, the top 5% group represents concentrated refactoring effort.

4.5 Hypothesis H1. Dependencies

To investigate H1.A, for each module in Vista, we measure *Neighbors*, the number of neighbor modules connected via dependencies. We then contrast the average inter-module dependencies of the top 5% preferentially refactored modules vs. that of bottom 95% of preferentially refactored modules in Vista. These results are summarized in Table 4. The second column and the third column report the relative ratio of a software metric with respect to the average metric value in Vista for the top 5% and bottom 95% groups respectively. The fourth column reports the p-value of the Wilcoxon test. Statistically significant test results (p-value ≤ 0.05) are marked in yellow background. Our results indicate that the top 5% most preferentially refactored modules have

7% more inter-module dependencies on average than the average modified module in Vista, while the bottom 95% have almost the same number of inter-module dependencies as the average modified module. In other words, developers did *preferentially applied refactorings* to the modules with a higher number of inter-module dependencies in Vista. A two-sided, unpaired Wilcoxon (Mann Whitney) test indicates that the top 5% group is statistically different from the bottom 95% in terms of its inter-module dependency coverage (p-value: 0.0026).

To investigate H1.B, we contrast the changes in the metric value for the top 5% group vs. the bottom 95% group. The changes in a metric are then normalized with respect to the *absolute* average delta of the software metric among all modified modules. Suppose that the top 5% of most preferentially refactored modules decreased the value of a software metric by 5 on average, while the bottom 95% increased the metric value by 10 on average. On average, a modified module has increased the metric value by 9.25. We then normalize the decrease in the top 5% group (-5) and the increase in the bottom 95% group (+10) with respect to the absolute value of the average change (9.25) average of all modified modules, resulting in -0.54 and +1.08 respectively.

Using this method, we measure the impact of preferential refactoring on the changes in inter-module dependencies. The result indicates that the top 5% of preferentially refactored modules decreased the number of inter-module dependencies by a factor of 0.85, while the bottom 95% of preferentially refactored modules increased the number of inter-module dependencies by a factor of 1.10 with respect to the average change of all modified modules. The Wilcoxon test indicates that the trend is statistically significant (p-value: 0.000022).

We conclude that, preferential refactoring is correlated with the decrease of inter-module dependencies. This is consistent with our interviews with the Windows refactoring team that *their goal is to reduce undesirable inter-module dependencies.*

4.6 Hypothesis H2. Defect

Similar to the analysis of contrasting the top 5% group vs. the bottom 95% group in Section 4.5, we investigate the relationship between preferential refactoring treatment and defects. The results in Table 4 indicate that the top 5% group has 9% fewer post-release defects than the rest in Vista, confirming that refactoring was not necessarily preferentially applied to the modules with a high number of defects.

Further, the top 5% group decreased post-release defects by a factor of 0.93, while the bottom 95% group decreased by a factor of 1.00 with respect to the average change in defect counts. In other words, the top 5% group is correlated with the reduction of post-release defects, but less so, compared to the rest. This result indicates that the cause of defect reduction cannot be attributed to refactoring changes alone. It is possible that the defect reduction is enabled by other events in those refactored modules as well.

4.7 Hypothesis H3. Complexity

We use the same method of contrasting the top 5% vs. bottom 95%. Table 4 summarizes the results. Top 5% of most preferentially refactored modules tend to have lower complexity measures in Vista compared to the bottom 95%. However, this distinction is not statistically significant, according to the Wilcoxon test, rejecting the hypothesis H3.A.

Top 5% group reduces fan-in, fan-out, and cyclomatic complexity measures more than the rest. For example, if we assume that an average modified module has 100 fan-ins in Windows Vista, the top 5% group covers 67, while the bottom 95% covers 102 fan-ins. The member read based coupling (C5.1) increases much less for the top 5%, compared to the rest. The results are statistically significant (p-value $\leq$ 0.05). However, for other complexity metrics, the same trend does not hold or is not statistically significant. In summary, we found that, the Windows refactoring effort did not reduce various complexity measures consistently.

4.8 Hypothesis H4. Size, Churn, and Locality

We investigate the relationship between refactoring and size, churn, and locality respectively. The hypotheses H4.A, H4.B and H4.C are motivated by the fact that developers often initiate refactoring to improve changeability and maintainability [24] and that crosscutting concerns pose challenges in evolving software systems [25]–[28]. We use the same study method of contrasting the top 5% vs. the bottom 95% described in Section 4.5.

Regarding H4.A, the top 5% has smaller size metrics than the rest in Vista. This indicates that refactoring changes were *not* preferentially applied to the modules with large size. Furthermore, the top 5% increased size more in terms of LOC, while the rest decreased their module size. In other words, preferential refactoring did not play any role in decreasing various size measures, rejecting H4.A.

Regarding H4.B, in contrast to our original hypothesis, the top 5% group consistently has lower churn measures in Vista, compared to the rest. In other words, the less frequent changes are in Vista, the more likely they are to receive preferential refactoring treatment during Windows 7 development. One possible explanation is that developers might have determined that the modules to be refactored are problematic and that making too many changes to them could impede the stability of the system. Thus, to preserve stability, they may have applied fewer changes intentionally. The churn measures in the top 5% group decrease less, compared to the rest. The less frequent refactorings are, the greater the decrease in churn measurements, rejecting H4.B.

Regarding H4.C, to measure the degree of crosscutting changes, we measure the average number of modified

TABLE 4
The relationship between Windows 7 refactoring and various software metrics. Statistically significant test results (p-value $\leq$ 0.05) are marked in yellow background.

	Vista			Δ (Vista, Windows7)		
Metric	**top 5%**	**bottom 95%**	**p-value**	**top 5%**	**bottom 95%**	**p-value**
Modularity						
Neighbors	1.07	1.00	0.00	-0.85	1.10	0.00
Defect						
Post release failures	0.91	1.00	0.00	-0.93	-1.00	0.00
Complexity						
C1. Fan in	0.67	1.02	0.72	-1.18	-0.99	0.01
C2. Fan out	0.69	1.02	0.79	-1.12	-0.99	0.02
C3. Cyclomatic complexity [31]	0.77	1.01	0.75	-0.16	1.06	0.04
C4. Inheritance depth	0.57	1.02	0.23	-0.62	-1.02	0.47
C5. Coupling through						
C5.1. member reads	0.78	1.01	0.51	0.09	1.05	0.02
C5.2. member writes	0.95	1.00	0.11	0.04	1.05	0.18
C5.3. function parameters	0.58	1.02	0.12	0.72	1.01	0.13
C5.4. type declarations in local functions	0.77	1.01	0.11	0.40	1.03	0.21
C5.5. immediate base classes	0.63	1.02	0.23	-0.63	-1.02	0.90
C5.6. return type	0.54	1.02	0.30	-0.48	-1.03	0.96
Size						
S1. LOC	0.80	1.01	0.79	1.24	-0.99	0.02
S2. # of classes	0.65	1.02	0.26	-0.69	-1.02	0.52
S3. # of parameters	0.75	1.01	0.63	-1.06	-1.00	0.09
S4. # of local variables	0.78	1.01	0.66	0.61	1.02	0.86
S5. # of function	0.71	1.01	0.68	-0.77	-1.01	0.52
S6. # of blocks	0.76	1.01	0.69	0.32	1.04	0.23
Churn						
Ch1. total churn	0.12	1.05	0.00	-0.02	-1.05	0.00
Ch2. relative churn	0.40	1.03	0.00	-0.02	-1.05	0.00
Ch3. # of check-ins	0.30	1.04	0.00	-0.35	-1.03	0.03
Ch4. # changed files	0.30	1.04	0.00	-0.18	-1.04	0.04
Locality						
L1. # files per check-in	0.91	1.04	0.89	1.73	0.96	0.00
People						
O1. NOE	0.53	1.02	0.00	-0.08	-1.05	0.79
O2. NOEE	0.67	1.02	0.02	-0.79	-1.01	0.88
Cohesiveness of contribution						
O3. DMO	-0.44	1.08	0.00	-1.60	1.14	0.00
O4. PO	0.70	1.02	0.02	4.53	-1.29	0.36
O5. OCO	62%	1.02	0.00	0.05	1.05	0.06
Diffusion of contribution						
O6. OOW	0.59	1.02	0.00	0.92	1.00	0.89
O7. OIF	1.02	0.99	0.42	1.86	0.95	0.00
Test adequacy						
T1. block coverage	1.13	0.99	0.00	-1.13	-0.99	0.13
T2. arc coverage	1.15	0.99	0.00	-1.18	-0.99	0.06
Layer						
L1. layer numbers	1.01	1.00	0.83	0.35	1.03	0.09

files per check-in. Our results indicate that preferential refactoring is applied to the modules with lower degree of crosscutting changes in Vista, and the modules which received preferential refactoring treatment tend to increase crosscutting changes more than other modules in Windows 7 (173% vs. 96%), rejecting H4.C. We believe that the increase in the crosscutting changes is caused by refactorings themselves, which tend to be more scattered than regular changes (on average refactorings touched 20% more files than regular changes).

4.9 Hypothesis H5. Developer and Organization

The more people who touch the code, the higher the chance of code decay, and there could be a higher need for coordination among the engineers [29]. Such coordination needs may call for refactoring of the relevant modules. We hypothesize that refactoring could reduce the number of engineers who worked on the modules by aligning the modular structure of the system with the team structure [34]–[36]. To check these hypotheses, we study the cohesiveness and diffusion measures of ownership and contribution. The less cohesive and local the contributions are, it may call for more refactoring effort, while diffusive ownership could impede the effort of initiating refactoring.

Table 4 summarizes the results. Regarding the hypotheses H5.A, the NOE and NOEE measures are lower in the top 5% than the rest, indicating that the refactoring

effort was preferentially made to the modules where a fewer number of developers worked on. This is contrary to our initial hypothesis. One possible explanation is that the refactored modules are crucial, important modules and for these modules typically only a tight small group of developers were allowed to check-in code changes.

The cohesiveness of contribution measures (DMO, PO, and OCO) are lower in the top 5% than the rest in Vista, validating H5.B. The refactoring effort was preferentially made to the modules with lower cohesiveness in terms of contribution. However, we did not find any distinctive trends in terms of changes in those measures before and after refactoring.

Regarding the hypothesis H5.C, the diffusion measure (OOW) is lower in the top 5% than the rest in Vista. However, we did not find any distinctive trends in terms of changes in those measures before and after refactoring.

4.10 Hypothesis H6. Test Coverage

We investigate the following hypothesis about refactoring and test adequacy, because our survey indicates that, when a regression test suite is inadequate, it could prevent developers from initiating refactoring effort. Developers perceive that there is no safety net for checking the correctness of refactoring edits when the test adequacy is low. Table 4 summarizes the results. The block and arc test coverages for the top 5% of most preferentially refactored modules are indeed higher than the rest.

4.11 Hypothesis H7. Layers

The layer data in Windows is a means of quantifying the architectural constraints of the system. Simplistically speaking, the lowest level is the Windows kernel and the highest level is the UI components. The goal of assigning layer numbers to modules is to prevent reverse dependencies. In other words, modules in level 0 are more important than say level 25 for the reliability of the system as modules at level n can depend only on the modules of level $n-1$ and below only. We investigate H7 to confirm our interview findings that the refactoring team split core modules and moved reusable functionality from an upper layer to a lower layer to repurpose Windows for different execution environments.

The top 5% and the bottom 95% of most preferentially refactored modules have similar average layer numbers (1.01 and 1.00 respectively in Table 4). The difference is not statistically significant. In contrast to the interviewees' understanding, the refactoring was not preferentially applied to modules with a high layer number. Further, the layer number increase for the top 5% group is less than the increase of the bottom 95%.

4.12 Multivariate Regression Analysis

To identify how much and which factors may impact a refactoring investment decision (or selection of the modules to apply refactorings preferentially), we use multivariate regression analysis.

TABLE 5
Multivariate regression analysis for preferential refactoring. Software metrics are measured for Vista before refactoring effort.

	Estimate	Std. Error	t value
(Intercept)	193.09	47.50	4.07
# files per check-in	19.38	5.81	3.34
# of sources	0.53	0.14	3.81
OOW	-2.25	0.45	-5.01
# of edits by engineers	0.12	0.02	6.02
defects	-10.92	3.87	-2.82
NOE	-2.00	0.39	-5.07
# of check-ins	-0.22	0.04	-5.44

To select independent variables for multivariate regression, we first measure the Spearman rank correlation between the rank difference and individual software metrics before refactoring. We use the metrics from Vista, as they represent the characteristics *before* the refactoring effort was made.

$$cor(all\_commit\_rank(m) - ref\_commit\_rank(m), metric\_vista(m)) \quad (2)$$

If the correlation is a positive number close to 1, it implies that preferentially refactored modules tend to have a higher metric value before refactoring. If the correlation is near zero, it implies that there exists no correlation between preferential refactoring and the metric.

We then select all metrics with significant correlations at $p < 0.0001$, even if the correlations values do not appear to be very high. We use the lower p-value to account for multiple hypothesis testing (Bonferroni Correction) [37]. We construct an initial model where a dependent variable is a rank difference per module, and independent variables are {# of files per check-in, # of sources, # of incoming dependencies, OCO, OOW, relative churn, NOE, # of changed files, # of defects, total churn, NOEE, NOE, and # of check-ins}. Then we use a forward and backward stepwise regression to reduce the number of variables in the model.

The final model is described in Table 5. The rows represent the selected variables and the cells represent coefficient, standard errors, and t-values. The results indicate that the locality of change, the number of dependent modules (sources), the number of defects, and the number of developers who worked on the modules are significant factors for preferential refactoring. As discussed in previous sections, factors such as complexity, size, and test adequacy do not play much roles in the decision of refactoring investment.

To investigate how other factors may contribute to reduction of defects and dependencies, we use multivariate regression analysis. Using the same method described, we select all metrics that have significant correlations with preferential refactoring at $p < 0.0001$ to build an initial model. Then we use a forward and backward stepwise regression to reduce the number of variables in the model. In addition to these selected

TABLE 6
Multivariate regression analysis results for the changes in defects and dependencies

	Δdefects	**Δdependencies**
(Intercept)	-0.109	4.985
rank differences	0.001	-0.001
# of all commits	0.009	0.001
# of refactoring commits	-0.072	-0.130
refactoring ratio	0.000	0.000
# incoming dependencies	0.000	0.001
# of sources	-0.003	-0.062
OOW		-0.014
relative churn	0.000	
# of edits by engineers	0.000	
# of changed files	0.000	0.001
defects	-0.965	-0.182
total churn	0.000	0.000
NOE	-0.024	-0.024

variables, we use several independent variables: (1) preferential refactoring, as measured by the *rank differences*, (2) refactoring churn and churn, as measured by the number of refactoring commits and all commits, and (3) refactoring ratio as measured by $\frac{\#\ of\ refactoring\ commits}{\#\ of\ all\ commits}$.

In Table 6, column **Δdefects** describes a regression model for the changes in the number of defects, i.e., $defects\_windows7(m) - defects\_vista(m)$. Column **Δdependencies** describes a regression model for the changes in the number of inter-module dependencies. Each cell describes a coefficient for a selected independent variable. The resulting model for **Δdefects** indicates that one of the most important factors for defect reduction is the number of previous defects in Vista. The higher the number of defects in Vista, the higher the decrease of defects in Windows 7 (coefficient, -0.965). Another important factor is the amount of refactoring churn, indicated by **# of refactoring commits** (coefficient, -0.072). This result indicates that, among many metrics, refactoring churn is likely to play a significant role in reducing the number of defects.

Similarly, the resulting model for **Δdependencies** indicates that important factors include the number of previous defects in Vista and the amount of refactoring churn (coefficients -0.182 and -0.130 respectively).

4.13 Results of Keyword based Identification of Refactoring Commits

In addition to the designated refactoring team, other developers in the Windows also apply refactoring to the system. To understand the impact of such refactoring, we identify refactoring changes by matching refactoring keywords against commit messages described in Section 4.2.

Refactoring commits found using a branch-based method overlap with refactoring commits found using a keyword-based method. When normalizing the absolute number of commits, suppose that the total number of commits is X. The number of refactoring commits identified based on keywords is $Y = 0.058X$. The number of refactoring commits identified based on branches is $Z = 0.013X$. The overlap between refactoring commits identified using both methods is $0.004X$, which is $0.006Y$ or $0.279Z$. The absolute number of commits is normalized for presentation purposes.

Table 7 presents the same information as Table 4, and the only difference between the two is that Table 7 uses a branch-based isolation method, while Table 4 uses a keyword-based method. According to the results of Table 4, the refactoring team's effort was preferentially applied to the modules with a relative high number of inter-module dependencies, low post-release defects, low churn measures, and high organization cohesiveness metrics. The team effort is correlated with a relatively higher degree of reduction in inter-module dependencies and certain complexity measures. On the other hand, the refactoring effort indicated by the keyword method focused on the modules with a relative low number of inter-module dependencies, low post-release defects, low complexity measures, low churn measures, small sizes, and high organization cohesiveness metrics.

Since refactorings identified by a keyword method and a branch isolation method are both correlated with reduction in inter-module dependencies, complexity metrics, and churn measures, these reduction trends are unlikely to be enabled by the refactoring team's effort alone. Instead, both the refactoring made by individual developers in Windows 7 and the designated refactoring team are likely to have contributed to the reduction of these metrics.

4.14 Discussion

Table 2 summarizes the study results of refactoring commits found by the branch method.

- Refactoring was preferentially applied to the modules with a large number of dependencies. Preferential refactoring is correlated with reduction in the number of dependencies. These findings are expected and not surprising because interview participants stated the goal of system-wide refactoring is to reduce undesirable inter-module dependencies. However, we believe that there is a value in validating intended benefits using version history analysis.
- Preferential refactoring is correlated with the reduction of post-release defects, but less so, compared to the rest. This indicates that the cause of defect reduction cannot be attributed to refactoring changes alone.
- Refactoring was not preferentially applied to the modules with high complexity. Preferentially refactored modules experience a higher rate of reduction in certain complexity measures, but increase LOC and crosscutting changes more than the rest of modules. This implies that managers may need automated tool support for monitoring the impact of refactoring in a multi-dimensional way.
- Refactoring was preferentially applied to the modules with high test coverage. This is consistent with

TABLE 7
The relationship between refactoring identified by a keyword based method and various software metrics. Statistically significant test results (p-value ≤ 0.05) are marked in yellow background.

	Vista			Δ (Vista, Windows7)		
Metric	**top 5%**	**bottom 95%**	**p-value**	**top 5%**	**bottom 95%**	**p-value**
Modularity						
Neighbors	0.92	1.00	0.13	-0.46	1.08	0.01
Defect						
Post release failures	0.94	1.00	0.00	-0.96	-1.00	0.00
Complexity						
C1. Fan in	0.44	1.03	0.02	-0.73	-1.01	0.31
C2. Fan out	0.48	1.03	0.03	-0.71	-1.02	0.36
C3. Cyclomatic complexity [31]	0.51	1.03	0.02	-0.22	1.06	0.00
C4. Inheritance depth	0.36	1.03	0.01	-0.49	-1.03	0.60
C5. Coupling through						
C5.1. member reads	0.54	1.02	0.40	0.58	1.02	0.03
C5.2. member writes	0.69	1.02	0.60	-0.11	1.06	0.01
C5.3. function parameters	0.21	1.04	0.00	0.41	1.03	0.01
C5.4. type declarations in local functions	0.26	1.04	0.00	0.31	1.04	0.00
C5.5. immediate base classes	0.42	1.03	0.01	-0.52	-1.02	0.77
C5.6. return type	0.26	1.04	0.01	-0.31	-1.04	0.06
Size						
S1. LOC	0.54	1.02	0.03	-0.94	-1.00	0.14
S2. # of classes	0.42	1.03	0.00	-0.48	-1.03	0.75
S3. # of parameters	0.54	1.02	0.03	-0.91	-1.00	0.13
S4. # of local variables	0.54	1.02	0.09	0.18	1.04	0.31
S5. # of function	0.47	1.03	0.02	-0.65	-1.02	0.44
S6. # of blocks	0.50	1.03	0.01	0.10	1.05	0.00
Churn						
Ch1. total churn	0.22	1.04	0.00	-0.15	-1.04	0.01
Ch2. relative churn	0.62	1.02	0.01	0.91	-1.10	0.01
Ch3. # of check-ins	0.29	1.04	0.00	-0.27	-1.04	0.00
Ch4. # changed files	0.36	1.03	0.00	-0.33	-1.04	0.03
Locality						
L1. # files per check-in	0.55	1.02	0.00	-0.92	-1.00	0.68
People						
O1. NOE	0.57	1.02	0.00	0.58	-1.08	0.76
O2. NOEE	0.63	1.02	0.02	-0.70	-1.02	0.66
Cohesiveness of contribution						
O3. DMO	-0.16	1.06	0.20	-1.71	1.14	0.00
O4. PO	1.53	0.97	0.00	-7.28	-0.67	0.00
O5. OCO	1.01	1.00%	0.64	-1.26	1.12	0.00
Diffusion of contribution						
O6. OOW	1.12	0.99	0.33	-9.06	1.53	0.00
O7. OIF	0.94	1.00	0.02	2.49	0.92	0.00
Test adequacy						
T1. block coverage	1.05	1.00	0.07	-1.27	-0.99	0.01
T2. arc coverage	1.06	1.00	0.09	-1.29	-0.98	0.01
Layer						
L1. layer numbers	1.01	1.00	0.83	0.35	1.03	0.09

the survey participants' view that test adequacy affects developers' likelihood to initiate refactoring.

5 Threats to Validity

Internal validity. Our findings in Section 3 indicate *only* correlation between the refactoring effort and reduction of 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 module 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 transfor-

mations, 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. We do not have data on how many of the survey participants participated in large refactorings vs. small refactorings as such question was not a part of the survey. 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.

To protect confidential information, we used standard normalizations. All analyses were performed on the actual values and the normalization was done for the presentation purposes only to protect confidential information.

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.

Our study about Windows 7 refactoring may not generalize to other refactoring practices, because the Windows refactoring team had a very specific goal of reducing undesirable inter-module dependencies and therefore may not be applicable where refactoring efforts have a different or less precise goal. The hypothesis H7 assumes that the software system has a layered architecture. Thus it may not be applicable to systems without a layered architecture. Because the Windows refactoring is a large scale, system-wide refactoring, its benefit may differ from other small scale refactorings that appear in Fowler's catalog.

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.[3] We believe that lifting the veil on the Windows refactoring process and quantifying the correlation between refactoring and various metrics could be valuable to other development organizations. To facilitate replication our study outside Microsoft, we published the full survey questions as a technical report [12].

6 Related Work

Refactoring Definition. While refactoring is defined as a behavior-preserving code transformation in the academic literature [10], 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 [1]. 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 [38], indicating that pure refactoring revisions are rare. Our survey in Section 2 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 [39]. 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 [40] 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 [21]. 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 [41]. 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 [42]. 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. [43] 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.

Several research projects automatically detect the symptoms of poor software design—coined as *code smells* by Fowler [1]. Gueheneuc et al. detect inter-class design defects [44] and Marinescu identifies design flaws using

3. Global diversity and inclusion http://www.microsoft.come/about/diversity/en/us/default.aspx

software metrics [45]. Izurieta and Bieman detect accumulation of non design-pattern related code [19]. Guo et al. define domain-specific code smells [46] and investigate the consequence of technical debt [20]. Wong et al. [47] identify *modularity violations*—recurring discrepancies between which modules should change together and which modules actually change together according to version histories. While these studies correlate the symptoms of poor design with quality measurements such as the number of bugs, these studies do not directly investigate the consequence of refactoring—purposeful actions to reverse or mitigate the symptoms of poor design.

Conflicting Evidence on Refactoring Benefit. Kataoka et al. [18] proposed a refactoring evaluation method that compares software before and after refactoring in terms of coupling metrics. Kolb et al. [23] 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. [48] 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 [49]. Ratzinger et al. developed defect prediction models based on software evolution attributes and found that refactoring related features and defects have an inverse correlation [5]—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 [6]. Kim et al. found that the number of bug fixes increases after API refactorings [9]. Nagappan and Ball found that code churn—the number of added, deleted, and modified lines of code—is correlated with defect density [50]— 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 [6].

Because manual refactoring is often tedious and error-prone, modern IDEs provide features that automate the application of refactorings [51], [52]. 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 [53]. 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 [38]. Vakilian et al. [16] and Murphy et al. [54] 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 [41] and structure similarity to identify move and rename refactorings [55]. 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 [56]. This work also describes a survey of existing refactoring reconstruction techniques. Kim et al. use the results of API-level refactoring reconstruction to study the correlation between API-level refactorings and bug fixes [9]. 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 indicators of code quality [50]. 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 [57]. More recently, Bird et al. observed that socio-technical network measures combined with dependency measures were stronger indicators of failures than dependency measures alone [58]. 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. In addition, this paper uses a multivariate regression analysis to investigate the factors beyond refactoring churn that could have affected the changes in the defects and inter-module dependencies.

7 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 328 professional software engineers. Then to examine whether the survey respondents' perception matches reality and whether there are visible benefits of refactoring, we interviewed 6 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 interview study shows how system-wide refactoring was carried out in Windows. The quantitative analysis of Windows 7 version history shows that refactoring effort was focused on the modules with a high number of inter-module dependencies and high test adequacy. Consistent with the refactoring goal stated by the interview participants, preferentially refactored modules indeed experienced higher reduction in the number of inter-module dependencies than other changed modules. While preferentially refactored modules experience a higher rate of reduction in certain complexity measures, they increase LOC and crosscutting changes more than the rest. As the benefit of refactoring is multi-dimensional and not consistent across various metrics, we believe that managers and developers can benefit from automated tool support for monitoring the impact of refactoring on various software metrics.

ACKNOWLEDGMENTS

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 a part of this work, while working 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 Microsoft SEIF award.

REFERENCES

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

[2] L. A. Belady and M. Lehman, "A Model of Large Program Development," *IBM Systems Journal*, vol. 15, no. 3, pp. 225–252, 1976.

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

[4] W. Opdyke, "Refactoring, reuse & reality," copyright 1999, Lucent Technologies.

[5] 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*. New York, NY, USA: ACM, 2008, pp. 35–38.

[6] 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*. ACM, 2006, pp. 112–118.

[7] ——, "Identifying refactorings from source-code changes," in *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 231–240.

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

[9] 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.

[10] T. Mens and T. Tourwe, "A survey of software refactoring," *IEEE Transactions on Software Engineering*, vol. 30, no. 2, pp. 126–139, 2004.

[11] T. Hastie, R. Tibshirani, and J. H. Friedman, *The elements of statistical learning: data mining, inference, and prediction: with 200 full-color illustrations*. New York: Springer-Verlag, 2001.

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

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

[14] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Proactive detection of collaboration conflicts," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 168–178. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025139

[15] C. Bird and T. Zimmermann, "Assessing the value of branches with what-if analysis," 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. 45:1–45:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393648

[16] 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*, june 2012, pp. 233 –243.

[17] A. Srivastava, J. Thiagarajan, and C. Schertz, "Efficient Integration Testing using Dependency Analysis," Microsoft Research, Tech. Rep. MSR-TR-2005-94, 2005.

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

[19] C. Izurieta and J. M. Bieman, "How software designs decay: A pilot study of pattern evolution," in *ESEM*. IEEE Computer Society, 2007, pp. 449–451.

[20] Y. Guo, C. Seaman, R. Gomes, A. Cavalcanti, G. Tonin, F. Q. B. Da Silva, A. L. M. Santos, and C. Siebra, "Tracking technical debt - an exploratory case study," in *Software Maintenance (ICSM), 2011 27th IEEE International Conference on*, September 2011, pp. 528 – 531.

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

[22] N. Brown, Y. Cai, Y. Guo, R. Kazman, M. Kim, P. Kruchten, E. Lim, A. MacCormack, R. Nord, I. Ozkaya, R. Sangwan, C. Seaman, K. Sullivan, and N. Zazworka, "Managing technical debt in software-reliant systems," in *Proceedings of the FSE/SDP workshop on Future of software engineering research*, ser. FoSER '10. New York, NY, USA: ACM, 2010, pp. 47–52. [Online]. Available: http://doi.acm.org/10.1145/1882362.1882373

[23] 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.*, vol. 18, pp. 109–132, March 2006. [Online]. Available: http://dl.acm.org/citation.cfm?id=1133105.1133108

[24] 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

[25] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton, "N degrees of separation: multi-dimensional separation of concerns," in *ICSE '99: Proceedings of the 21st International Conference on*

Software Engineering. Los Alamitos, CA, USA: IEEE Computer Society Press, 1999, pp. 107–119.

[26] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *European Conference on Object-oriented Programming*, vol. 1241. Lecture Notes in Computer Science 1241, 1997, pp. 220–242. [Online]. Available: citeseer.ist.psu.edu/kiczales97aspectoriented.html

[27] M. Eaddy, T. Zimmermann, K. D. Sherwood, V. Garg, G. C. Murphy, N. Nagappan, and A. V. Aho, "Do crosscutting concerns cause defects?" *IEEE Trans. Software Eng.*, vol. 34, no. 4, pp. 497–515, 2008.

[28] R. J. Walker, S. Rawal, and J. Sillito, "Do crosscutting concerns cause modularity problems?" 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. 49:1–49:11. [Online]. Available: http://doi.acm.org/10.1145/2393596.2393654

[29] F. P. Brooks, *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley Professional, August 1975. [Online]. Available: http://www.amazon.com/exec/obidos/redirect?tag=citeulike07-20&path=ASIN/0201835959

[30] E. Murphy-Hill, A. P. Black, D. Dig, and C. Parnin, "Gathering refactoring data: a comparison of four methods," in *WRT '08: Proceedings of the 2nd Workshop on Refactoring Tools*. New York, NY, USA: ACM, 2008, pp. 1–5.

[31] T. J. McCabe, "A complexity measure," *IEEE Trans. Software Eng.*, vol. 2, no. 4, pp. 308–320, 1976.

[32] N. Nagappan, B. Murphy, and V. Basili, "The influence of organizational structure on software quality: an empirical case study," in *Proceedings of the 30th international conference on Software engineering*, ser. ICSE '08. New York, NY, USA: ACM, 2008, pp. 521–530. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368160

[33] 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*. ACM, 2008, pp. 304–318.

[34] M. E. Conway, "How do committees invent?" November 16th 2012 1968.

[35] J. Herbsleb and R. Grinter, "Architectures, coordination, and distance: Conway's law and beyond," *Software, IEEE*, vol. 16, no. 5, pp. 63–70, Sep 1999.

[36] S. Bailey, S. Godbole, C. Knutson, and J. Krein, "A decade of conway's law: A literature review from 2003-2012," in *Replication in Empirical Software Engineering Research (RESER), 2013 3rd International Workshop on*, Oct 2013, pp. 1–14.

[37] P. Dalgaard, *Introductory statistics with R*, 2nd ed. New York: Springer, 2008.

[38] 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*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 287–297.

[39] K. Sullivan, P. Chalasani, S. Jha, and V. Sazawal, *Software Design as an Investment Activity: A Real Options Perspective in Real Options and Business Strategy: Applications to Decision Making*. Risk Books, November 1999.

[40] C. Y. Baldwin and K. B. Clark, *Design Rules: The Power of Modularity*. Cambridge, MA, USA: MIT Press, 1999.

[41] 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*. New York, NY, USA: ACM, 2005, pp. 54–65.

[42] 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*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 389–398.

[43] A. MacCormack, J. Rusnak, and C. Y. Baldwin, "Exploring the structure of complex software designs: An empirical study of open source and proprietary code," *Management Science*, vol. 52, no. 7, pp. 1015–1030, 2006.

[44] Y.-G. Guéhéneuc and H. Albin-Amiot, "Using design patterns and constraints to automate the detection and correction of inter-class design defects," in *Proceedings of the 39th International Conference and Exhibition on Technology of Object-Oriented Languages and Systems (TOOLS39)*, ser. TOOLS '01. Washington, DC, USA: IEEE Computer Society, 2001, pp. 296–. [Online]. Available: http://dl.acm.org/citation.cfm?id=882501.884740

[45] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proceedings of the 20th IEEE International Conference on Software Maintenance*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 350–359. [Online]. Available: http://dl.acm.org/citation.cfm?id=1018431.1021443

[46] Y. Guo, C. Seaman, N. Zazworka, and F. Shull, "Domain-specific tailoring of code smells: an empirical study," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 167–170. [Online]. Available: http://doi.acm.org/10.1145/1810295.1810321

[47] S. Wong, Y. Cai, M. Kim, and M. Dalton, "Detecting software modularity violations," in *ICSE' 11: Proceedings of the 2011 ACM and IEEE 33rd International Conference on Software Engineering*, 2011.

[48] R. Moser, A. Sillitti, P. Abrahamsson, and G. Succi, "Does refactoring improve reusability?" in *ICSR*, 2006, pp. 287–297.

[49] 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*. ACM, 2010, pp. 149–157.

[50] 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*. New York, NY, USA: ACM, 2005, pp. 284–292.

[51] W. G. Griswold, "Program restructuring as an aid to software maintenance," Ph.D. dissertation, University of Washington, 1991.

[52] D. Roberts, J. Brant, and R. Johnson, "A refactoring tool for smalltalk," *Theory and Practice of Object Systems*, vol. 3, no. 4, pp. 253–263, 1997.

[53] 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*. New York, NY, USA: ACM, 2007, pp. 185–194.

[54] G. C. Murphy, M. Kersten, and L. Findlater, "How are java software developers using the eclipse ide?" *IEEE Softw.*, vol. 23, no. 4, pp. 76–83, Jul. 2006. [Online]. Available: http://dx.doi.org/10.1109/MS.2006.105

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

[56] K. Prete, N. Rachatasumrit, N. Sudan, and M. Kim, "Template-based reconstruction of complex refactorings," in *2010 IEEE International Conference on Software Maintenance*, September 2010, pp. 1 –10.

[57] T. Zimmermann and N. Nagappan, *Predicting defects using network analysis on dependency graphs*, ser. ICSE '08. New York, NY, USA: ACM, 2008. [Online]. Available: http://doi.acm.org/10.1145/1368088.1368161

[58] 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*. IEEE Computer Society, 2009, pp. 109–119. [Online]. Available: http://dx.doi.org/10.1109/ISSRE.2009.17

Miryung Kim received a B.S. in Computer Science from Korea Advanced Institute of Science and Technology in 2001. She received a Ph.D. in Computer Science & Engineering from the University of Washington in 2008 and then joined the faculty at the University of Texas at Austin where she is currently an assistant professor. Her research interests lie in the area of software engineering, with a focus on improving developer productivity in evolving software systems. She received a Google Faculty Research Award in 2014, NSF CAREER award in 2011, a Microsoft Software Engineering Innovation Foundation Award in 2011, an IBM Jazz Innovation Award in 2009, and ACM SIGSOFT Distinguished Paper Award in 2010.

Thomas Zimmermann received his Ph.D. from Saarland University in 2008. He is a researcher in the Research in Software Engineering (RiSE) group at Microsoft Research Redmond, adjunct assistant professor at the University of Calgary, and affiliate faculty at University of Washington. His research interests include empirical software engineering, mining software repositories, computer games, recommender systems, development tools, and social networking. He is best known for his research on systematic mining of version archives and bug databases to conduct empirical studies and to build tools to support developers and managers. He received ACM SIGSOFT Distinguished Paper Awards in 2007, 2008, and 2012.

Nachiappan Nagappan received his Ph.D. from North Carolina State University in 2005. He is a principal researcher in the Research in Software Engineering (RiSE) group at Microsoft Research Redmond. His research interests are software reliability, software metrics, software testing and empirical software processes. His interdisciplinary research projects span the spectrum of software analytics ranging from intelligent software design for games, identifying data center reliability to software engineering optimization for energy in mobile devices.