Gathering Refactoring Data: a Comparison of Four Methods

Emerson Murphy-Hill
and Andrew P. Black

Portland State University
emerson,black@cs.pdx.edu

Danny Dig

Massachusetts Institute of
Technology
dannydig@csail.mit.edu

Chris Parnin

Georgia Institute of Technology
chris.parnin@gatech.edu

Abstract

Those of us who seek to build better refactoring tools need empirical data collected from real refactoring sessions. The literature reports on different methods for capturing this data, but little is known about how the method of data capture affects the quality of the results. This paper describes 4 methods for drawing conclusions about how programmers refactor, characterizes the assumptions made by each, and presents a family of experiments to test those assumptions. We hope that the results of the experiments will help future researchers choose a data-collection method appropriate to the question that they want to investigate.

Categories and Subject Descriptors D.2.0 [*Software Engineering*]: General

General Terms Design, Experimentation, Measurement

Keywords refactoring, methodology, research, empirical software engineering

1. Introduction

Researchers seeking to improve refactoring tools frequently collect empirical data about refactoring sessions to motivate and evaluate these improvements. For example, a researcher may wish to find evidence that some refactorings are performed frequently, or that programmers prefer one tool's user interface over another, or that programmers would benefit from some new kind of refactoring. Empirical data can be a powerful motivator for new refactoring tools, and is vital for their evaluation.

At least four sources have been used for such data: commit logs, code history, programmers, and refactoring tool usage logs. Corresponding to each source is a mechanism for extracting the information from the raw data; we call the

combination of source and mechanism a *research method*. In the next section, we discuss the assumptions, benefits and disadvantages of each research method.

2. The Four Methods and their Assumptions

2.1 Mining the Commit Log

A fairly straightforward way of identifying refactorings is to look in the commit logs of versioned repositories for mention of the word "refactor." Most commonly maintained by *cvs* or *subversion*, commit logs are updated when a programmer commits a change to the repository. By searching for the word "refactor," and possibly for related words such as "rename" or "extract," a researcher can find periods in the development history when a programmer noted that refactoring took place. This method is appealing because commit logs for a large number of open source projects are available.

Several researchers have used commit logs as indicators for refactoring. Stroggylos and Spinellis [18] used this method to conclude that classes become less coherent as a result of refactoring. Ratzinger and colleagues [16] used 13 keywords to find refactorings, and concluded that an increase in refactoring tends to be followed by a decrease in software defects. Hindle and colleagues [9] classified source-code commits based on their log messages, concluding that refactorings are common in commits that affect many files.

One of the authors (Parnin) has previously presented preliminary evidence that commit logs are incomplete indicators of refactoring activity, noting that in one project, only 7 revisions were logged as refactoring even though 55 revisions contained evidence of refactoring [14, §7.1.3]. This is perhaps unsurprising: there are several reasons why commit log messages might be poor indicators of refactoring. Programmers may simply be careless when creating log messages; it is also possible that commits that contain refactorings mixed with other changes may not be tagged as refactorings. Moreover, log messages that do contain the word "refactor" may be biased toward certain kinds of refactorings, possibly sweeping refactorings like RENAME CLASS. The limitations of this data-collection method color the inferences that can be drawn from the data. For example, Ratzinger and colleagues' [16] results may indicate that

large refactorings like RENAME CLASS correlate inversely with defects, but the data may not speak to the correlation between smaller refactorings like EXTRACT METHOD and defects. In general, we see that the method chosen to collect refactoring data influences the generality of the conclusions.

The mining commit of logs assumes that the programmer both *recalls* performing the refactoring and *describes* the refactoring accurately.

2.2 Analyzing Code Histories

Another method for uncovering past refactorings is by analyzing a sequence of versions of the source code. Researchers have taken a window of two (not necessarily adjacent) versions of the code from a source repository and inferred where refactorings had been performed either by manual comparison or by automating the comparison using a software tool.

Opdyke [13] manually compared 12 versions of the *Choices* file system, and concluded that refactoring resulted in increasing abstraction of interfaces over time. Weißgerber and Diehl [20] automatically processed CVS data for refactorings, concluding that programmers did not spend time just refactoring. Xing and Stroulia [21] processed the Eclipse code base and concluded that reducing the visibility of class members is a frequently-performed refactoring that development environments do not currently support. Demeyer and colleagues [4] use a set of object-oriented change metrics and heuristics to detect refactorings such as MERGE CLASSES or SPLIT CLASS) that will serve as markers for the reverse engineer. Antoniol and colleagues [1] use a technique inspired by Information Retrieval to detect discontinuities (e.g., SPLIT CLASS) in classes. Godfrey and Zou [8] implemented a tool for detecting refactorings in procedural code. Another of the authors (Dig) developed RefactoringCrawler [5], which combines static and semantic analysis to detect API refactorings in frameworks and libraries. The follow-up tool, RefacLib [19], combines syntactic analysis with a set of change-metrics to detect API refactorings in libraries. They concluded that automated detection of API refactorings is both practical and accurate for the purpose of automated migration of client applications toward newer library APIs. Dig and Johnson have used manual comparison of versions to find API changes [6]. They guided their inspection using the change documents (e.g., version release notes) that usually accompany major software versions, and concluded that more than 80% of API-breaking changes are caused by refactorings.

Code history analysis can be problematic as a method for finding refactorings. Examining non-adjacent versions can speed up the process, but fine-grained refactorings can be lost in the process. For example, if a programmer adds a new method in an existing class and then performs an EXTRACT LOCAL VARIABLE refactoring inside the new method, a comparison of the first and final versions will fail to produce any evidence of the EXTRACT LOCAL VARIABLE (since the first version does not contain the method enclosing the extracted variable).

Manual history inspection is slow, and researchers may inadvertently miss some refactorings when comparing versions. Automatic history inspection is faster, but refactoring detectors are typically limited in the kinds of refactorings that they can detect. For example, the tools described in this subsection focus on detecting high-level, architectural refactorings (e.g., MERGE CLASSES) or medium-level refactorings (e.g., API refactorings like RENAME METHOD). None of the current detection tools focuses on detecting low-level refactorings like REPLACE TEMP WITH QUERY. To extend the set of refactorings automatically detected, it would be necessary to implement new detection strategies. Two of the most serious challenges for automatic detection tools are *scalability*, and noise introduced by several *overlapping refactorings* (e.g., a method was renamed and the class that defines the method was also renamed). To cope with such challenges, current tools are heuristic; however, the parameters of the heuristics can dramatically affect the accuracy of the results. Thus, using these tools requires some experimentation.

Xing and Stroulia's [21] UMLDiff tool helped them find high-level refactorings like MOVE CLASS, but was not capable of finding lower level refactorings like REPLACE TEMP WITH QUERY. Thus, while their conclusion that a particular refactoring is performed frequently is certainly valid, they cannot address the question of whether other, undetected, refactorings are performed even more frequently. This is an important question to be able to ask if one's goal is to direct one's tool-bulding efforts where they will have most benefit.

Code history analysis assumes that there is adequate *granularity* in the history, that an appropriate *window* of observation is chosen, and that any automatic refactoring detection heuristics are appropriately parameterized[1].

2.3 Observing Programmers

A third method of collecting information about refactoring is for researchers to observe developers working on a software project, and to characterize their refactoring behavior. Such observation may take the form of direct observation (a controlled experiment or ethnography) or indirect observation (a survey or a project post-mortem). This method has the advantage that observations can be directed towards the specific aspects of the refactoring task that is of interest.

Using direct observation, Boshernitsan and colleagues [2] observed 5 programmers use their refactoring tool in a laboratory setting, and concluded that the tool is intuitive. Murphy-Hill and Black observed programmers perform EXTRACT METHOD in a laboratory [11], concluding that programmers had difficulty in selecting code and understanding error messages. Using indirect observation, Pizka [15] described a case study in which a project was refactored for 5

---

[1] Note that choosing the "appropriate" tool settings may be project-specific.

months; they concluded that refactoring without a concrete goal is not useful. Bourqun and Keller [3] described a 6-month refactoring case study; they observed that the average number of lines per class decreased by about 10%.

Programmer observation also has limitations. If the researcher uses indirect observation, the accuracy of the results depends on the programmer's ability to accurately recall their refactorings. When researchers observe refactoring over long-term programming projects, results may be affected by uncontrolled, external factors. For example, a project that refactors frequently may be successful because of the refactoring, or for some other reason, such as experienced management. Researchers who watch programmers in the wild may observe very little refactoring over the span of a few hours. Controlled experiments are expensive, as they require time commitments both on the part of the researchers and programmers. Controlled experiments also may not be valid outside the context of the experiment (external validity), as the tasks that researchers assign to programmers may not be representative of common programming tasks. For example, while Murphy-Hill and Black observed that programmers had difficulty in selecting code for EXTRACT METHOD [11], programmers may have little difficulty in selecting code for other refactorings.

This research method relies on the assumption that a programmer can accurately *recall* refactorings, the assumption of *frequent* refactoring activity, and the assumption that a controlled experiment is *externally valid*.

2.4 Logging Refactoring Tool Use

Some development environments automatically record programmer activity in a log file. Such an environment can record general programming events, or can be specialized to collect just refactoring tool events. A chief advantage of this approach is that it can be highly accurate, because it records every use of a refactoring tool.

Murphy and colleagues observed 99 users of the Eclipse IDE using a logging tool [10] and noted that about twice as many people used the RENAME tool as used the EXTRACT METHOD tool. Dig and colleagues recorded refactorings from two programmers using the Eclipse IDE [7], and note that their MolhadoRef tool reduces refactoring-induced merge conflicts compared to CVS. Robbes observed two programmers in the Squeak environment using the Spyware logging tool [17] and notes that programmers did not use any refactoring tool during most programming sessions.

Refactoring tool logs are limited in two ways. First, logged refactorings may not contain sufficient detail to recreate what happened during a refactoring; Murphy and colleagues' data [10], for instance, do not tell us if most RENAMES were of variables, methods, or classes. Second, refactoring tool usage logs *always* omit manual refactorings. As a consequence, for instance, while Robbes' study [17] provides evidence that programmers use refactoring tools infrequently, we cannot infer that refactoring itself was in-

	context	fidelity
explicit	commit log mining	tool usage logs
implicit	programmer observation	analyzing histories

Table 1. The research methods vary both in how they identify refactorings and the detail they capture about that event. When deciding which method to use, a researcher can choose a property from the top and property from the left to decide which method is most appropriate for the hypothesis.

frequent: it may be that programmers preferred to refactor manually.

Finally, we observe that this research method inherently assumes that a programmer *uses tools* for refactoring.

2.5 Summary of Methods

Each of the research methods described above has its own strengths and weaknesses (Table 1). The methods that rely on explicit identification of refactorings (commit logs and tool logs) **precisely** identify refactoring activity, but can be **inaccurate** because they ignore implicit refactorings (those that are not tagged or that are performed without a tool). In contrast, the implicit methods may be more accurate because they discover manual refactorings and refactorings that the programmer considered incidental to other changes, but may lack precision because of the variance between observers and the efficacy of analysis techniques. Commit Logs and Programmer Observation both provide more context explaining a refactoring event (**context**), whereas history analysis and tool logs may more faithfully capture the details of the refactoring event (**fidelity**).

Finally, researchers can conduct experiments to complement the strengths of research methods. For example, methods allowing collection of rich context may be more appropriate for exploratory analysis, striving to discover the reasons and motivations of programmers during refactoring activity. On the other hand, a research study wishing to gather reliable data on the occurrence of refactoring may want to choose methods with higher fidelity. Moreover, it is important to weigh the results of implicit and explicit methods, which may capture only specific types of refactoring.

3. Hypotheses and Experiments

As we emphasized when we described them, each method for finding refactorings relies on assumptions that the research community has not yet validated. In Table 2, we propose several hypotheses related to these assumptions, and outline experiments for testing those hypotheses. For each hypothesis, we state *why* we suspect that the hypothesis is true, briefly describe *how* we plan to conduct an experiment to test the hypothesis, and note *what effect* the confirmation of the hypothesis will have on future research.

	Hypothesis	We suspect the hypothesis is true because …	To test this hypothesis, we plan to …	A confirmed hypothesis would show that …
Mining the Commit Log	Commits labeled "refactor" do not indicate significantly more of refactoring instances than those without the label	programmers may label some varieties of refactoring when committing, but not others	randomly select between 10 and 100 versions without the "refactor" label and the same number with the label, and hand-count refactorings from code history, and compare the groups with a Wilcoxon rank-sum test	the "refactor" label is not a reliable indicator of refactoring activity
	Commits labeled "refactor" contain proportionally more global and fewer local refactorings than those without the label	programmers may be aware of refactoring when they do "big" refactorings and want to let other developers know when they do them	perform the same experiment described in the row above, except that we collect and compare the proportion of local versus global refactorings	research that finds refactorings based on labels will be biased toward global refactorings
	Commits labeled "refactor" indicate significantly fewer non-refactoring changes than those without the label	programmers may not want to mix refactoring and non-refactoring in the same commit	again compare random refactoring-labeled and non-labeled groups, but this time measure the proportion of refactorings versus non-refactorings	finding refactorings based on labels will be biased toward root canal refactoring and against floss refactoring [12]
Analyzing Code Histories	Refactorings below the method level account for the majority of refactorings	smaller changes may be needed more frequently than larger changes	manually examine between 10 and 100 versions found from the code history, identify and classify the refactorings as high- and low-level, and compute a proportion	analysis techniques that focus on high-level refactorings will not detect most refactorings
	Refactorings hidden by other changes account for a substantial proportion of refactorings	developers may perform many small refactorings in support of larger refactorings, only to invert the smaller ones before committing to the repository	observe the actual refactorings performed by several programmers, identify the refactorings checked into their code repository, and compute the proportion of total refactorings performed to refactorings checked-in	code history analysis cannot detect a substantial proportion of refactorings
Observing Programmers	Direct programmer observation yields few refactoring results per researcher hour	this is the reason that few researchers observe programmers refactoring in the wild	spend between 4 and 20 hours observing programmers and noting which refactorings are performed and at what times	observing programmers in the wild is not a cost-effective research technique
	Programmers do not recall most manual refactorings, but do recall most refactoring with tools, in retrospect	programmers may not be conscious that they are refactoring, except when they use tools	after observing programmers during coding with and without refactoring tools, ask them how much and which refactorings they performed, with and without tools, and compare the refactorings that they recalled versus the refactorings that they performed	retrospective questionnaires are not reliable indicators of manual refactoring activity, but are reliable indicators of refactoring tool usage
	Many refactorings are performed manually, without a refactoring tool, even when one is available	a previous survey suggested that programmers sometimes do not use refactoring tools [12]	collect refactoring tool logs from several programmers and compare the results to refactorings gleaned manually from the commits that the programmers made to their code repositories	use of a refactoring tool is not a reliable indicator of the quantity of refactoring performed
Logging Refactoring Tool Usage	The variety of refactorings performed with tools do not reflect the variety performed without tools	programmers may prefer to use tools for some refactorings and manual refactoring for others	perform the experiment in the row above, except that we would compare the types of refactorings with and without tools using a chi-squared statistical test	statistics about refactorings performed with a refactoring tool do not apply to refactoring in general

Table 2. Nine hypotheses about refactoring.

4. Conclusions

We encourage and applaud researchers who gather empirical data to inform and validate their work. However, this does not mean that all methods of data collection are equally appropriate; as we have seen, each method colors the data that it produces.

There are two ways to reduce the likelihood that limitations in your research method will invalidate the conclusions of your research. First, choose a method whose limitations are orthogonal to the conclusions you wish to draw. For example, if you would like to show that programmers use a PULL UP refactoring tool more than a PUSH DOWN tool, the fact that tool usage logs omit manual refactorings is irrelevant. Second, use data gathered using several different methods with orthogonal limitations. If the data all tell the same story, then the effect of the limitations is reduced. For example, if a researcher used both tool logs and history analysis to show that refactoring tools are underused, the conclusion is stronger than if only one method were used.

Acknowledgments

References

[1] G. Antoniol, M. D. Penta, and E. Merlo. An automatic approach to identify class evolution discontinuities. In *IWPSE'04: Proceedings of International Workshop on Principles of Software Evolution*, pages 31–40, 2004.

[2] M. Boshernitsan, S. L. Graham, and M. A. Hearst. Aligning development tools with the way programmers think about code changes. In *CHI '07: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 567–576, New York, 2007. ACM.

[3] F. Bourqun and R. K. Keller. High-impact refactoring based on architecture violations. In *CSMR '07: Proceedings of the 11th European Conference on Software Maintenance and ReEngineering*, pages 149–158, Washington, DC, USA, 2007. IEEE Comp. Soc.

[4] S. Demeyer, S. Ducasse, and O. Nierstrasz. Finding refactorings via change metrics. In *OOPSLA'00: Proceedings of Object Oriented Programming, Systems, Languages, and Applications*, pages 166–177, 2000.

[5] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In D. Thomas, editor, *ECOOP*, volume 4067 of *Lecture Notes in Computer Science*, pages 404–428. Springer, 2006.

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

[7] D. Dig, K. Manzoor, R. Johnson, and T. N. Nguyen. Refactoring-aware configuration management for object-oriented programs. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 427–436, Washington, DC, USA, 2007. IEEE Comp. Soc.

[8] M. W. Godfrey and L. Zou. Using origin analysis to detect merging and splitting of source code entities. *IEEE Transactions on Software Engineering*, 31(2):166–181, 2005.

[9] A. Hindle, D. M. German, and R. Holt. What do large commits tell us?: A taxonomical study of large commits. In *MSR '08: Proceedings of the 2008 International Workshop on Mining Software Repositories*, pages 99–108, New York, 2008. ACM.

[10] G. C. Murphy, M. Kersten, and L. Findlater. How are Java software developers using the Eclipse IDE? *IEEE Software*, 23(4):76–83, 2006.

[11] E. Murphy-Hill and A. P. Black. Breaking the barriers to successful refactoring: Observations and tools for extract method. In *ICSE '08: Proceedings of the International Conference on Software Engineering*, pages 421–430, 2008.

[12] E. Murphy-Hill and A. P. Black. Refactoring tools: Fitness for purpose. *IEEE Software*, 25(5):38–44, Sept.-Oct. 2008.

[13] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, Univserity of Illinois at Urbana-Champaign, Urbana-Champaign, IL, USA, 1992.

[14] C. Parnin and C. Görg. Improving change descriptions with change contexts. In *MSR '08: Proceedings of the 2008 International Workshop on Mining Software Repositories*, pages 51–60, New York, 2008. ACM.

[15] M. Pizka. Straightening spaghetti-code with refactoring? In H. R. Arabnia and H. Reza, editors, *Software Engineering Research and Practice*, pages 846–852. CSREA Press, 2004.

[16] 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 workshop on Mining Software Repositories*, pages 35–38, New York, 2008. ACM.

[17] R. Robbes and M. Lanza. Spyware: a change-aware development toolset. In *ICSE '08: Proceedings of the International Conference on Software Engineering*, pages 847–850, New York, 2008. ACM.

[18] K. Stroggylos and D. Spinellis. Refactoring–does it improve software quality? In *WoSQ '07: Proceedings of the 5th International Workshop on Software Quality*, page 10, Washington, DC, USA, 2007. IEEE Comp. Soc.

[19] K. Taneja, D. Dig, and T. Xie. Automated detection of API refactorings in libraries. In *ASE '07: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 377–380, 2007.

[20] P. Weißgerber and S. Diehl. Are refactorings less error-prone than other changes? In *MSR '06: Proceedings of the 2006 International Workshop on Mining Software Repositories*, pages 112–118, New York, 2006. ACM.

[21] Z. Xing and E. Stroulia. Refactoring practice: How it is and how it should be supported — an Eclipse case study. In *ICSM '06: Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 458–468, Washington, DC, USA, 2006. IEEE Comp. Soc.