# Refactoring Operations Grounded in Manual Code Changes

Anna Maria Eilertsen
Department of Informatics
University of Bergen
Norway
anna.eilertsen@uib.no

## ABSTRACT

Refactoring tools automate tedious and error-prone source code changes. The prevalence and difficulty of refactorings in software development makes this a high-impact area for successful automation of manual operations. Automated refactorings tools can improve the speed and accuracy of software development and are easily accessible in many programming environments. Even so, developers frequently eschew automation in favor of manual refactoring and cite reasons like lack of support for real usage scenarios and unpredictable tools. In this paper, we propose to redesign refactoring operations into transformations that are useful and applicable in real software evolution scenarios with the help of repository mining and user studies.

## CCS CONCEPTS

• **Software and its engineering** → General programming languages; **Software maintenance tools**; *Software notations and tools.*

## KEYWORDS

refactoring, refactoring tools, productivity

**ACM Reference Format:**
Anna Maria Eilertsen. 2020. Refactoring Operations Grounded in Manual Code Changes. In *42nd International Conference on Software Engineering Companion (ICSE '20 Companion), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3377812.3381395

## 1 INTRODUCTION

Refactoring is a technique for changing the internal structure of a program while preserving its external behavior. Refactoring operations can be performed either as a set of manual source code changes concerted by a developer or with an automated refactoring tool. Such tools improve the accuracy and speed with which developers can perform routine tasks, and most mainstream IDEs include extensive automated support for many refactoring operations. Despite their availability and benefits, developers frequently avoid using them and perform a majority of their refactorings manually [11, 14, 25, 33].

*ICSE '20 Companion, May 23–29, 2020, Seoul, Republic of Korea*
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7122-3/20/05…$15.00
https://doi.org/10.1145/3377812.3381395

We refer to the manual application of a refactoring when a tool is available as *disuse* of the tool. Disuse of refactoring tools is problematic because manually orchestrating such program transformations require error-prone, repetitive changes distributed across programs. To do this manually, a developer must mentally reconstruct program binding information and complex relationships between program elements according to the language syntax and semantics. These activities are time-consuming and may induce disorientation [2] and errors.

In this work we propose to improve refactoring tools by altering the source code transformations they implement such that they are more effectively and frequently useful for developers. Through surveying the literature for reasons that motivate disuse, we find that developers find the tools unpredictable, have difficulties invoking them correctly and do not trust them to not mess up their code [11, 24, 25, 33]. Furthermore, developers report being unaware that they are, in fact, performing a refactoring, and that tools do not implement real usage scenarios [11, 14, 25, 33]. Previous efforts at improving these factors has been directed mainly at either user-interface or transformation correctness. We ask whether tools can better support real refactoring activities if the refactorings operations themselves are altered.

Refactorings are inherently complex; developers manage this complexity manually by making incremental and localized changes. For example, when moving a method from one class to another, a typical manual strategy is to cut and paste the method, which (hopefully) induces compilation errors at all references, and then navigate between the resulting compiler errors to update calls and inheritance hierarchies. In this way, the developer only ever forms a lightweight and flexible *plan* of the source code change, and rely on the compiler and IDE support to guide her through a sequence of local code changes [33]. In contrast, refactoring tools automates the code change as a single source code transformation, sometimes comprising many different types of code changes. This atomic design require the developer to invoke and configure the tool correctly up-front, thereby forcing her to formulate the entire plan at once. Going back to our move-example, the developer must select both the target class (which must exist) and, for all non-static methods, the class object through it will be accessed before she can execute the refactoring. This premature commitment [7] to configurations options increases the developer's effort and subtracts from the tool's benefits.

Our hypothesis is: **Refactoring tools are more effective and useful during software evolution and maintenance if they automate source code changes that more closely align with the order and *chunking* of developers' manual code changes.** To achieve this, refactoring operations may need to be decomposed into smaller operations that potentially do not preserve behavior individually, but can be combined into behavior-preserving source code changes. We speculate whether trading some safety for simplicity and consistency can lead developers to find the tools more predictable and, in fact, trust them more.

To validate my hypothesis, we formulate the following research questions:

- R1: Why do developers disuse refactoring tools during software evolution tasks?
- R2: How do developers' expectations from refactoring tools differ from the actual design of refactoring transformations?
- R3: How do developers conceptually *chunk* their manual code changes during manual refactoring operations?
- R4: Which manual code changes do developers consider repetitive and tedious during manual refactoring?
- R5: Can information about developers manual refactorings, expectations and chunking be used to design refactoring transformations that are more useful during software evolution?

Section 2 describes related work and how this work fits into it. Section 3 describes the creation of software evolution tasks and interview setup used to investigate RQ1-4. Section 4 describes how findings will be analyzed and how they contribute to RQ5. Section 5 describes the expected contributions from this work, and Section 6 proposes the timeline.

## 2 RELATED WORK

Previous work consistently report that more than half of refactorings are done manually. Kim et al. finds that developers at Microsoft self-report doing 86% manually [11]; Murphy-Hill et al. found 90% to be manually [14]; both Negara et al. and Silva et al. report more than half to be performed manually, and that developers are more likely to avoid automation as the complexity of refactorings increase [15, 16, 25]. Vakilian also found that developers prefer automating simple code changes [32] and conclude that refactoring tools should implement smaller code changes that can be composed to form more complex ones. Field studies of refactoring have provided quantitative data about refactoring use and disuse, but without providing actionable findings on disuse [14–16, 33]. Our work is motivated by these findings and improve on them by grounding terminology in code changes or tool behavior during software evolution scenarios.

Developers report avoiding tools due to lack of trust and predictability [11, 25], finding them awkward to invoke and incorporate into their workflow [33], and lacking support for refactorings performed in the wild [11]. There is little observational data on tool disuse. Most findings are self-reported in surveys and interviews without concrete examples or definitions. A recent survey illustrates how developers' expectations differ for even simple refactorings, as do the implementations of state-of-practice tools [17], but contains simple abstract examples. Most lab studies use explicit refactoring goals which are not representative for the majority of *floss-refactoring* [14] during software evolution. Kataoka et al. studied benefits of refactoring in software evolution, and reported some interesting expectations from refactoring tools that violated their specifications, like using `Rename` to redirect method calls [9]. Our work is motivated by these findings and compliment them in terms of study design and data collection.

Technical reasons for disuse include incorrectness [1, 27], long processing time [10], bugs [13] and limited applicability of refactoring operations [13, 22]. Researchers improve tools' speed [10] and correctness [1, 3, 6], create new techniques [22, 30, 31] or language extensions [23] to increase the scope of tools. Most of these areas are orthogonal to our approach. We may make, and evaluate, tradeoffs with technical aspects.

In order to improve trust, researchers attempt to ensure that tool produces only correct programs. For example, constraint-based refactoring tools can, in theory, generate any program equivalent to the original [21, 26] but also face difficulties with capturing intent and determine acceptable additional changes [28]. In the program metamorphosis system [20], the user themselves navigates the space of possible programs through a structural editor while the a model-checker ensures that behavior is preserved at the end. Synthesis-based refactorings [19] lets the user invoke a refactoring by introducing a few changes in the code and automatically synthesize a sequence of refactoring that can complete the operation. These approaches complement ours: the search-based ones may benefit from a dataset to develop heuristics for delimiting the search space, and all of them may implement the transformations that we develop. In contrast, we prioritize predictability and transparency over behavior-preservation and correctness.

There are no comprehensive, modern lists of refactoring specifications. Fowler's refactoring catalogue [5] lists high-level, ambiguous descriptions of refactorings with steps like "adjust the method's body to its new home". The most detailed catalogue [18] is 30 years old, based on closed-world, C++ programs. Many preconditions and assumptions about the programs are violated or impossible to check in modern software. We speculate if it is more useful to provide a different set of refactorings to modern developers. Vakilian suggested decomposed refactorings and performed a case study on `Extract Class` [33]. Schäfer et al. decomposed `Extract Method` into components that eased implementation, not use. Our work is motivated by these findings and aims to propose and evaluate a set of decomposed code changes.

## 3 STUDYING REFACTORING DISUSE AND MANUAL CHANGES

We investigate R1-4 in a lab study and interview setup designed to capture refactoring tool disuse during software evolution tasks. The task set is designed by mining repositories for refactorings with RefactorMiner [29] and investigating refactorings from a preexisting dataset [25]. By manually investigating the mined code changes we pick severals potential commits that have strong conceptual motivation as a software evolution task, typically described in the commit message or related documentation artifact. To ensure that the study investigate refactoring tool disuse in software evolution scenarios, and not usability issues in refactoring invocation scenarios, we take care to motivate the changes without using code smells or refactoring terminology. This conceptual motivation is used as a task description for our subjects, and when prompted for more information, we present them with additional context collected from the project's pull request discussion or other documents.

The tasks are selected such that their goals are not refactoring code, but they can be partially solved using refactoring tools. For example, one task instructs the subject to Remove two methods. In order to do so, she needs to inline, or otherwise move, the method's bodies into their callers. Another task instructs the developers to remove a functionality from a class. This functionality is specified by callers through a boolean flag argument in method calls to this class. In order to solve the task, subjects may use repeated invocations of `Remove Parameter` or other refactorings that can solve their high-level goal. This is a realistic task – and was praised as such by several subjects – that lets us both observe their workflow in a realistic, complex refactoring task, and provides a grounding foundation for a subsequent interview about their previous experiences.

Our subjects are professional developers who are at least somewhat familiar with refactorings and have at least a year of experience in Java. They are instructed to solve software evolution tasks in Java using IntelliJ IDE. We take care to select tasks such that the motivation and the code change are possible to understand without deep domain knowledge, and are possible to solve within two hours; a number of pilot studies verified this. We create a scoped-down clone of an existing software project, Apache Commons Lang, that is well-written with tests and documentation and understandable without domain knowledge. We find two suitable tasks from the projects' history that mirror similar scenarios that we frequently observe. These two commits can be used out of the box without any mapping. One other task is mapped from another software project onto a class in this project, such that the core transformations are similar. We aggregate the project versions where the different commits reside and create a standalone version where all refactoring tasks can be executed in succession. This result in a collection of tasks that more accurately represents refactoring in the wild than is common in previous lab study tasks. To our knowledge this is a novel way of creating realistic refactoring tasks.

During the experiment, participants' source code changes and workflow are recorded using screen capture and notes. Subjects are prompted using a think-aloud protocol, to report their own terminology, descriptions and motivations for their changes, and to *chunk* their source code changes into tedious or repetitive sets. After they finish, the interviewer ask in-depth about motivating reasons for using or disusing refactoring tools during the tasks based on their observations. They are also prompted using what-if questions. We expect that many participants will not use refactoring tools even though it could save them significant time. None of our pilot participants have, and were reluctant when we suggested it because of the conceptual difference between the tasks and the refactoring names.

The results from this study form two complementing datasets. One is the transcribed dialogues containing concepts, beliefs, terminology and expectations. The other is a set of observed source code changes applied to solve the software evolution tasks. Through combined analysis of the two we expect to ground terms like predictable, trust, tedious and repetitive, in actual source code changes or other relevant observations. This analysis will conclude my investigation of RQ1-4. We will report our qualitative findings and publish the task set. Both may benefit tool designers and researchers of workflows, productivity and refactorings. In the time of writing, the user study is concluded and is being analyzed.

## 4 DESIGNING PREDICTABLE REFACTORINGS

The datasets from Section 3 are used to investigate RQ5. We observe that developers could have used refactorings like `Inline Method`, `Move Method` and `Remove Parameter` during the study, yet frequently chose not to. We investigate whether their manual changes differ from the design of these tools. We explore different source code transformations that align with the chunking and terminology used by developers to see if they can be made more predictable than the existing refactorings.

For an example of what such a decomposition could be like, consider that find-and-replace seems to be an intuitive decomposition of a textual change. If we find that developers chunk their manual changes similarly (which was the case with several pilot participants), the following is an example of how `Inline Method` and `Move Method` could be decomposed into a similar pattern: `Inline Method` *finds* calls to a method and *replaces* them with the method body. Then one can consider there to be a clean-up-step where indirection through parameters and return values are removed. `Move Method` first moves a method to another class, then *finds* calls to the original method and *replaces* them with calls to the new method. If the method is accessed through an object, replacing the call includes replacing the invocation argument with a reference to an object from the new class.

The proposed operations are evaluated in terms of expressivity, feasibility, composability and usefulness. To evaluate whether they combine to express common refactoring operations we manually analyze how to combine them into some common refactorings, and validate them by using them to replay real refactorings from our mined software repositories. We previously investigated how many instances of `Inline Method` from Silva et al.'s dataset was supported by Eclipse, and found that 24% were not. A decomposed version should perform at least comparably. To evaluate the feasibility of these refactorings, we implement a prototype tool that automates them and can be invoked from an IDE. The prototype's usefulness is evaluated by repeating my user study using this tool. We use the same evolution tasks and compare developers' experience with my tool to investigate the potential impact on developers' real software evolution tasks. The decomposition will impact which program transformation frameworks is preferred. Eclipse and IntelliJ are both candidates due to their tight IDE integration. IntelliJ would be ideal for repeating the study in the same IDE. Regardless of IDE, the backend may be implemented using other rewriting tools. JastAdd [4] provides easy language extensions that are useful for refactorings and has been used in related work [21–23]. Another option is the Rascal metaprogramming language [8, 12].

## 5 CONTRIBUTION

The expected contributions of this work are:

(1) A reusable set of realistic software evolution tasks that can be solved partially using refactoring tools.
(2) A dataset of source code changes and developers' expectations and reasons for disuse grounded in code changes.

(3) A set of *chunked* source code changes as repetitive, predictable or conceptually similar.
(4) A set of specifications for simple, predictable source code transformations that can help developers speed and accuracy during software evolution tasks.
(5) A prototype implementation and evaluation.

## 6 TIMELINE

The expected timeline is:

- Execute developer study during winter 2019/2020. Code and analyze collected data during spring 2020.
  Potential publication venues for the results are FSE 2020 and the Programming Journal.
- Iterative work on combined top-down / bottom-up design of appropriate refactoring operations based on findings summer-autumn 2020. Iteratively evaluate design through replicating open-source refactorings.
  Potential publication venue for the result is ICSE 2021.
- Implement operations in a prototype tool, evaluate by repeating user study winter 2020/2021.
  Potential publication venue is FSE 2021 and OOPSLA 2021.

## ACKNOWLEDGMENTS

This work is supported by the Research Council of Norway under grant number 250683 *(Co-Evo)*, and by a travel grant from the Meltzer Research Fund under grant number 18105.

## REFERENCES

[1] Fabian Bannwart and Peter Müller. 2006. Changing programs correctly: Refactoring with specifications. In *International Symposium on Formal Methods*. Springer, 492–507. https://doi.org/10.1007/11813040_33
[2] Brian De Alwis and Gail C Murphy. 2006. Using visual momentum to explain disorientation in the Eclipse IDE. In *Visual Languages and Human-Centric Computing (VL/HCC'06)*. IEEE, 51–54. https://doi.org/10.1.1.186.4464
[3] Anna Maria Eilertsen, Anya Helene Bagge, and Volker Stolz. 2016. Safer refactorings. In *International Symposium on Leveraging Applications of Formal Methods*. Springer, 517–531. https://doi.org/10.1007/978-3-319-47166-2_36
[4] Torbjörn Ekman and Görel Hedin. 2007. The jastadd extensible java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications* (Montreal, Quebec, Canada) *(OOPSLA '07)*. ACM, New York, NY, USA, 1–18. https://doi.org/10.1145/1297027.1297029
[5] Martin Fowler. 2018. *Refactoring: improving the design of existing code.* Addison-Wesley.
[6] A. Garrido and J. Meseguer. 2006. Formal Specification and Verification of Java Refactorings. In *2006 Sixth IEEE International Workshop on Source Code Analysis and Manipulation*. 165–174. https://doi.org/10.1109/SCAM.2006.16
[7] Thomas RG Green. 1989. Cognitive dimensions of notations. *People and computers V* (1989), 443–460.
[8] Mark Hills, Paul Klint, and Jurgen J Vinju. 2012. Scripting a refactoring with rascal and eclipse. In *Proceedings of the Fifth Workshop on Refactoring Tools*. ACM, 40–49.
[9] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya. 2002. A quantitative evaluation of maintainability enhancement by refactoring. In *International Conference on Software Maintenance, 2002. Proceedings.* 576–585. https://doi.org/10.1109/ICSM.2002.1167822
[10] Jongwook Kim, Don Batory, Danny Dig, and Maider Azanza. 2016. Improving refactoring speed by 10x. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 1145–1156.
[11] Miryung Kim, Thomas Zimmermann, and Nachiappan Nagappan. 2012. A Field Study of Refactoring Challenges and Benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, North Carolina) *(FSE '12)*. ACM, New York, NY, USA, Article 50, 11 pages. https://doi.org/10.1145/2393596.2393655
[12] Paul Klint, Tijs Van Der Storm, and Jurgen Vinju. 2009. Rascal: A domain specific language for source code analysis and manipulation. In *2009 Ninth IEEE International Working Conference on Source Code Analysis and Manipulation*. IEEE, 168–177. https://doi.org/10.1109/SCAM.2009.28
[13] M. Mongiovi, R. Gheyi, G. Soares, M. Ribeiro, P. Borba, and L. Teixeira. 2018. Detecting Overly Strong Preconditions in Refactoring Engines. *IEEE Transactions on Software Engineering* 44, 5 (May 2018), 429–452. https://doi.org/10.1109/TSE.2017.2693982
[14] Emerson Murphy-Hill, Chris Parnin, and Andrew P. Black. 2009. How We Refactor, and How We Know It. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 287–297. https://doi.org/10.1109/ICSE.2009.5070529
[15] Stas Negara, Nicholas Chen, Mohsen Vakilian, Ralph E Johnson, and Danny Dig. 2013. A comparative study of manual and automated refactorings. In *European Conference on Object-Oriented Programming*. Springer, 552–576. https://doi.org/10.1007/978-3-642-39038-8_23
[16] Stas Negara, Mihai Codoban, Danny Dig, and Ralph E Johnson. 2014. Mining fine-grained code changes to detect unknown change patterns. In *Proceedings of the 36th International Conference on Software Engineering*. ACM, 803–813.
[17] Jonhnanthan Oliveira, Rohit Gheyi, Melina Mongiovi, Gustavo Soares, Márcio Ribeiro, and Alessandro Garcia. 2019. Revisiting the refactoring mechanics. *Information and Software Technology* 110 (2019), 136 – 138. https://doi.org/10.1016/j.infsof.2019.03.002
[18] William F. Opdyke. 1992. *Refactoring Object-oriented Frameworks.* Ph.D. Dissertation.
[19] Veselin Raychev, Max Schäfer, Manu Sridharan, and Martin Vechev. 2013. Refactoring with Synthesis. *SIGPLAN Not.* 48, 10 (Oct. 2013), 339–354. https://doi.org/10.1145/2544173.2509544
[20] Christoph Reichenbach, Devin Coughlin, and Amer Diwan. 2009. Program Metamorphosis. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming* (Italy) *(Genoa)*. Springer-Verlag, Berlin, Heidelberg, 394–418. https://doi.org/10.1007/978-3-642-03013-0_18
[21] Max Schaefer and Oege de Moor. 2010. Specifying and Implementing Refactorings. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications* (Reno/Tahoe, Nevada, USA) *(OOPSLA '10)*. ACM, New York, NY, USA, 286–301. https://doi.org/10.1145/1869459.1869485
[22] Max Schäfer, Torbjörn Ekman, and Oege de Moor. 2008. Sound and Extensible Renaming for Java. In *Proceedings of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications* (Nashville, TN, USA) *(OOPSLA '08)*. ACM, New York, NY, USA, 277–294. https://doi.org/10.1145/1449764.1449787
[23] Max Schäfer, Mathieu Verbaere, Torbjörn Ekman, and Oege de Moor. 2009. Stepping stones over the refactoring rubicon. In *European Conference on Object-Oriented Programming*. Springer, 369–393.
[24] T. Sharma, G. Suryanarayana, and G. Samarthyam. 2015. Challenges to and Solutions for Refactoring Adoption: An Industrial Perspective. *IEEE Software* 32, 6 (Nov 2015), 44–51. https://doi.org/10.1109/MS.2015.105
[25] Danilo Silva, Nikolaos Tsantalis, and Marco Tulio Valente. 2016. Why We Refactor? Confessions of GitHub Contributors. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA) *(FSE 2016)*. ACM, New York, NY, USA, 858–870. https://doi.org/10.1145/2950290.2950305
[26] Friedrich Steimann. 2018. Constraint-Based Refactoring. *ACM Trans. Program. Lang. Syst.* 40, 1, Article 2 (Jan. 2018), 40 pages. https://doi.org/10.1145/3156016
[27] Friedrich Steimann and Andreas Thies. 2009. From public to private to absent: Refactoring Java programs under constrained accessibility. In *European Conference on Object-Oriented Programming*. Springer, 419–443. https://doi.org/10.1007/978-3-642-03013-0_19
[28] Friedrich Steimann and Jens von Pilgrim. 2012. Refactorings Without Names. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (Essen, Germany) *(ASE 2012)*. ACM, New York, NY, USA, 290–293. https://doi.org/10.1145/2351676.2351726
[29] Nikolaos Tsantalis, Matin Mansouri, Laleh M Eshkevari, Davood Mazinanian, and Danny Dig. 2018. Accurate and efficient refactoring detection in commit history. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 483–494.
[30] N. Tsantalis, D. Mazinanian, and G. P. Krishnan. 2015. Assessing the Refactorability of Software Clones. *IEEE Transactions on Software Engineering* 41, 11 (Nov 2015), 1055–1090. https://doi.org/10.1109/TSE.2015.2448531
[31] N. Tsantalis, D. Mazinanian, and S. Rostami. 2017. Clone Refactoring with Lambda Expressions. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 60–70. https://doi.org/10.1109/ICSE.2017.14
[32] Mohsen Vakilian. 2014. *Less is sometimes more in the automation of software evolution tasks.* Ph.D. Dissertation. University of Illinois at Urbana-Champaign.
[33] M. Vakilian, N. Chen, S. Negara, B. A. Rajkumar, B. P. Bailey, and R. E. Johnson. 2012. Use, disuse, and misuse of automated refactorings. In *2012 34th International Conference on Software Engineering (ICSE)*. 233–243. https://doi.org/10.1109/ICSE.2012.6227190