Predictable, Flexible or Correct: Trading off Refactoring Design Choices

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. Such tools can improve the speed and accuracy of software development, yet developers frequently eschew automation in favor of manual refactoring. Developers report distrust and lack of predictability as reasons for not using automated tools, but there are no comprehensive explanations of trust and predictability nor guidelines for how to improve these aspects of tools. In this position paper we explore choices and tradeoffs in refactoring tool design.

CCS CONCEPTS

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

KEYWORDS

refactoring, tool usability, productivity

1 INTRODUCTION

Refactoring is prevalent during software development and evolution but is tedious and error-prone to perform manually. Therefore, automating such program changes have a high potential impact on developer productivity. However, despite most mainstream IDEs providing extensive support for automated refactorings, these tools are *underused* [14, 22, 31, 39] and have been for decades.

Underuse of refactoring tools has been researched for a long time. Through empirical studies and interviews, previous work has found underuse to be tied to reasons ranging from unawareness of refactorings [30, 31, 39], issues in user-interface [10, 21], bugs [14], distrust [22, 31, 39] and reluctance to rely on unpredictable automation [14, 31, 39]. Many of these areas have been addressed by researchers, such as improving refactoring correctness [2, 6, 27], detecting overly strong preconditions [18] and bugs [17], together with various improvements to the user interface of tools [5, 7, 10, 15, 20, 21].

An area that has been mostly unexplored is improving *predictability* and avoiding *"unintended code changes"* [22, 39]. In an effort to make tools 'smarter', they may be designed to handle precondition violations or other problems by introducing additional code changes. For example, IntelliJ's `Rename` will rename comments and references in xml-files as well, and `Extract Method` may generate additional code to make programs compilable when preconditions are violated. In this way, an otherwise rejected refactoring can be made to correctly succeed. However, these code changes may alter code in ways that developers do not expect or want. They are nonetheless *correct* with respect to program semantics. In comparison, mainstream IDEs offer so-called unsafe [4] refactorings which trade correctness for usefulness by allowing the developer to continue a refactoring invocation after being alerted of violated preconditions. In this case, a developer may experience the tool as unpredictable when it rejects invocations of refactorings that are *incorrect* because she expect the refactoring to proceed even if it breaks the code and manually clean up compilation errors afterward [4, 39].

Refactoring tools have been extensively studied as program transformations between well-formed programs of a single language, under the assumption that user's goal is "improving code quality without changing the behavior of the program", including breaking its well-formed property. In practice, a refactoring operation is not necessarily applied with the expectation of increasing code quality.

Developers apply refactoring tools to programs that are outside the domain of well-formed, single-language, closed-world programs, and do so with the intent of reaching a source code state they have envisioned.

As long as tool designers consider the latter a misuse of the tools, rather than part of their specification, developers are left with either overly restrictive tools that require too much setup to use, or unpredictable ad-hoc solutions to use cases that are part of their everyday work. As a field, we need to consider new approaches to tool use that has previously been dubbed incorrect.

How does the design of refactoring transformations impact tool predictability? A design is usually predictable when it shares similarities with other designs that the user has encountered before, or somehow conforms to the user's mental model. Refactoring tools may be made more predictable either by improving the developers' ability to make predictions presumably through education, or by improving the tool's predictability through a coherent and consistent design. Such a design should allow users to trust that they can rely on, and generalize from, previous experiences by minimizing variation or contraction.

In contrast, the design of refactoring operations typically vary across IDEs and even within the same IDE [24]. In this position paper, we discuss the notions of predictability and trust in refactoring

ICSEW'20, May 23–29, 2020, Seoul, Republic of Korea
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7963-2/20/05…$15.00
https://doi.org/10.1145/3387940.3392185

tools and how they may be influenced by tradeoffs in the design of refactoring workflows.

2 TRADING FLEXIBILITY FOR PREDICTABILITY

Underuse or disuse of refactoring tools refers to developers performing refactoring operations manually rather than using tools. Previous work found that more than half of refactorings are done manually: Kim et al. finds that developers at Microsoft self-report doing 86% manually [14]; Murphy-Hill et al. found 90% to be manual [22]; 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 [23, 31]. Vakilian found that developers prefer automating simple code changes that they can predict the outcome of and manually composing them into more complex code changes, rather than relying on complex automated tools [38, 39].

Reasons for disuse arise from both technical and human-oriented aspects of refactoring tools. Developers report avoiding tools due to lack of trust and predictability [14, 31], finding them awkward to invoke and incorporate into their workflow [39], and lacking support for refactorings performed in the wild [14, 31]. State-of-the-practice tools[1] implement diverse versions of the same refactorings and similar refactorings appear under different names, thereby making it challenging for users to predict their outcomes accurately [24].

Current refactoring operations are often ambiguous. Most tools implement refactorings from Opdyke's [25] or Fowler's [8] catalogues, potentially extended using new techniques and encompassing new language features. The IDE interactions are still largely similar to the Smalltalk refactoring browser [26]. This results in ambiguous interpretations of refactorings that are more complex than `Rename`. A recent survey illustrates the range by which developers' expectations differ for even simple refactorings [24], while in more complex refactorings, differences are likely to diverge further. Fowler's refactoring catalogue lists high-level descriptions of refactorings, including steps like "adjust the method's body to its new home" after moving it to another location.

As additional techniques or algorithms make tools more powerful and flexible, they also become increasingly difficult to predict. Some overly strong preconditions can be amended by making additional code changes [28, 36]. For example, when invoking `Extract Method` in IntelliJ on a selection with incomplete return branching, the tool automatically generates a boolean flag as a workaround [36] rather than rejecting the refactoring[2] like Eclipse would do. We do not know how such additional changes impact the predictability of the tool. Constraint-based refactoring tools can, in theory, generate any program equivalent to the original [27, 32, 34] but also face difficulties with how to capture refactoring intents. Previous works suggest composing smaller refactorings [19, 38], yet there is no guide for how to do such a decomposition. Schäfer et al. decomposed `Extract Method` into components that eased implementation, but without addressing the impact on use [29].

Technical areas of refactoring tools regularly receive incremental improvements. Such areas include incorrectness and bugs [17, 18], long processing time and limited applicability of refactoring operations. Researchers effort at improving tools' speed [12] and correctness [2, 6, 9, 13, 33] or language extensions [29] across increasingly varied programs [1, 28, 36, 37]. However, predictability and trust are challenging both to define and improve [4, 11], and have traditionally received little focus.

Alternative perspectives on the refactoring process invite different design priorities for tools. The traditional model [16] describes the process as driven by the intention to refactor code, and consists of the following steps:

(1) Identify where the software should be refactored;
(2) Determine which refactoring(s) should be applied to the identified places;
(3) Guarantee that the applied refactoring preserves behavior;
(4) Apply the refactoring;
(5) Assess the effect of the refactoring on quality characteristics of the software (e.g., complexity, understandability, maintainability) or the process (e.g., productivity, cost, effort);
(6) Maintain the consistency between the refactored program code and other software artifacts (such as documentation, design documents, requirements specifications, tests and so on).

In practice, developers are rarely given time to perform smell-removal or "root-canal" [22] refactoring during software development and evolution. Most refactorings occur as part of a larger, high-level goal [14, 31]. Such a goal may not be overall behavior preserving: for example, introducing a feature, fixing a bug or changing an API may include refactoring operations that IDEs can automate. Since IDEs usually implement these code transformations as atomic refactorings like `Move Method` or `Extract Class`, the developer has to be able to *map* her high-level goal onto the available refactoring operations. This may include composing several operations interleaved with manual code changes. During this process we may assume that she assess the tools not on their contribution to code quality per se, but on their likeliness to bring her closer to her high-level goal than the corresponding manual change would [3, 7]. The second perspective illustrates the importance of predictability and why an unsafe tool may in fact be more useful than a restrictive and correct one.

3 DISCUSSION

We have argued the importance of improving non-technical aspects of tools like predictability and trust. How might we as a community approach these areas of refactoring tool design?

Perhaps refactoring operations can be amended to be more consistent or even decomposed into simpler operations that can express a similar set of transformations? Data mined from repositories provide valuable information about the context in which such operations are applied, and datasets of IDE interactions can teach us about how developers use refactoring tools to explore, or sketch, changes that they never apply, yet learn from. However, while mining provides knowledge about the refactoring operations that developers do eventually keep (manually or using tools), it is equally valuable to learn about what transformations developers do *not* want refactoring tools to automate. Through surveys or lab studies grounded in realistic software evolution scenarios, we may

[1]IDEs like Eclipse, IntelliJ, NetBeans, and Visual Studio
[2]https://youtrack.jetbrains.com/issue/DOC-6231

learn more about which "unintended" changes developers consider unacceptable.

A natural question to ask is what impacts developer's development of trust in tools. Perhaps more importantly, is what impacts developers *loss* of trust in refactoring tools. This may vary across skill-level (or rather, the accuracy of their mental models): For example, an expert user who expects a refactoring tool to succeed but instead encounter an uncommon error message and a tool failure, may be able to correctly associate the error with limitations of an underlying analysis tool and adjust her expectation of the refactoring tool appropriately. On the contrary, a novice experiencing the same scenario may be unable to correctly interpret the error and instead conclude broadly that the refactoring tool is flawed and untrustworthy in general. Is there any way refactoring tools can meet the needs of both novice and expert users?

Studies may also provide more information about what non-language factors impact developer predictions. While developers have self-reported issues with trust and predictability in surveys and interviews, previous lab studies have emphasized more technical or UI-based usability problems that shows up when the code changes are motivated with a refactoring goal. Consider that is quite different to measure how efficiently developers use a tool like `Extract Method` to achieve a prescribed goal of extracting methods, in comparison to how efficiently they can use it to achieve software evolution goals [35]. Indeed, in a study of refactoring benefits in software evolution, developers were found to attempt to use refactoring tools in ways that were outside of the specifications, yet nonetheless made "intuitive sense" to the participants [35]. This aligns with the claim that unsafe tools may be more useful than strictly safe ones, and developers seem satisfied with unsafe tools that can speed up code changes as it is clear what has changed [4, 22, 39]. There exist little observational information about developers' use of refactoring tools during software evolution, nor are self-reported experiences usually grounded in examples or concrete code changes. It could be interesting to explore how software context like function names, test-code, API-borders, polyglot projects or code ownership interact with refactoring tools and developers' expectations from them.

By considering how trust and predictability of refactoring tools are impacted by the design of tools and transformations, we may better provide developers with useful automation that they are not afraid to use.

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.

REFERENCES

[1] Aharon Abadi, R Ettinger, and YA Feldman. 2009. Fine slicing for advanced method extraction. In *3rd workshop on refactoring tools*, Vol. 21. https://doi.org/10.1007/978-3-642-28872-2_32

[2] 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

[3] Alan F Blackwell. 2002. First steps in programming: A rationale for attention investment models. In *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*. IEEE, 2–10. https://doi.org/10.1109/HCC.2002.1046334

[4] J. Brant and F. Steimann. 2015. Refactoring Tools are Trustworthy Enough and Trust Must be Earned. *IEEE Software* 32, 6 (Nov 2015), 80–83. https://doi.org/10.1109/MS.2015.145

[5] Dustin Campbell and Mark Miller. 2008. Designing Refactoring Tools for Developers. In *Proceedings of the 2nd Workshop on Refactoring Tools (WRT '08)*. ACM, New York, NY, USA, Article 9, 2 pages. https://doi.org/10.1145/1636642.1636651

[6] 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

[7] S. R. Foster, W. G. Griswold, and S. Lerner. 2012. WitchDoctor: IDE support for real-time auto-completion of refactorings. In *2012 34th International Conference on Software Engineering (ICSE)*. 222–232. https://doi.org/10.1109/ICSE.2012.6227191

[8] Martin Fowler. 1999. *Refactoring: improving the design of existing code*. Addison-Wesley.

[9] 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

[10] Xi Ge, Quinton L. DuBose, and Emerson Murphy-Hill. 2012. Reconciling Manual and Automatic Refactoring. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*. IEEE Press, Piscataway, NJ, USA, 211–221. http://dl.acm.org/citation.cfm?id=2337223.2337249

[11] Munawar Hafiz and Jeffrey Overbey. 2015. Refactoring myths. *IEEE Software* 32, 6 (2015), 39–43. https://doi.org/10.1109/MS.2015.130

[12] 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. https://doi.org/10.1145/2884781.2884802

[13] Jongwook Kim, Don Batory, and Milos Gligoric. 2016. Move-Instance-Method Refactorings: Experience, Issues, and Solutions. (2016).

[14] 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 (FSE '12)*. ACM, New York, NY, USA, Article 50, 11 pages. https://doi.org/10.1145/2393596.2393655

[15] Yun Young Lee, Nicholas Chen, and Ralph E. Johnson. 2013. Drag-and-drop Refactoring: Intuitive and Efficient Program Transformation. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. IEEE Press, Piscataway, NJ, USA, 23–32. http://dl.acm.org/citation.cfm?id=2486788.2486792

[16] T. Mens and T. Tourwe. 2004. A survey of software refactoring. *IEEE Transactions on Software Engineering* 30, 2 (Feb 2004), 126–139. https://doi.org/10.1109/TSE.2004.1265817

[17] M. Mongiovi. 2016. Scaling Testing of Refactoring Engines. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*. 674–676.

[18] 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

[19] Emerson Murphy-Hill. 2009. A model of refactoring tool use. *Proc. Wkshp. Refactoring Tools* (2009). https://people.engr.ncsu.edu/ermurph3/papers/wrt09.pdf

[20] E. Murphy-Hill, M. Ayazifar, and A. P. Black. 2011. Restructuring software with gestures. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 165–172. https://doi.org/10.1109/VLHCC.2011.6070394

[21] E. Murphy-Hill and A. Black. 2008. Breaking the barriers to successful refactoring. In *2008 ACM/IEEE 30th International Conference on Software Engineering*. 421–430. https://doi.org/10.1145/1368088.1368146

[22] 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

[23] 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. https://doi.org/10.1145/2568225.2568317

[24] 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

[25] William F. Opdyke. 1992. *Refactoring Object-oriented Frameworks*. Ph.D. Dissertation.

[26] Don Roberts, John Brant, and Ralph Johnson. 1997. A Refactoring Tool for Smalltalk. *Theor. Pract. Object Syst.* 3, 4 (Oct. 1997), 253–263. https://doi.org/10.1002/(SICI)1096-9942(1997)3:4<253::AID-TAPO3>3.3.CO;2-I

[27] 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 (OOPSLA '10)*. ACM, New York, NY, USA, 286–301. https://doi.org/10.1145/1869459.1869485

[28] 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 (OOPSLA '08).* ACM, New York, NY, USA, 277–294. https://doi.org/10.1145/1449764.1449787

[29] 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. https://doi.org/10.1007/978-3-642-03013-0_17

[30] 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

[31] 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 (FSE 2016).* ACM, New York, NY, USA, 858–870. https://doi.org/10.1145/2950290.2950305

[32] 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

[33] 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

[34] Friedrich Steimann and Jens von Pilgrim. 2012. Refactorings Without Names. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE 2012).* ACM, New York, NY, USA, 290–293. https://doi.org/10.1145/2351676.2351726

[35] L. Tokuda and D. Batory. 1999. Evolving object-oriented designs with refactorings. (Oct 1999), 174–181. https://doi.org/10.1109/ASE.1999.802203

[36] 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

[37] 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

[38] Mohsen Vakilian. 2014. *Less is sometimes more in the automation of software evolution tasks.* Ph.D. Dissertation. University of Illinois at Urbana-Champaign.

[39] 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