DOI:10.1145/3131873

Developers know refactoring improves their software, but many find themselves unable to do so when they want to.

BY EWAN TEMPERO, TONY GORSCHEK, AND LEFTERIS ANGELIS

Barriers to Refactoring

REFACTORING[6] IS SOMETHING software developers like to do. They refactor a lot. But do they refactor as much as they would like? Are there barriers that prevent them from doing so? Refactoring is an important tool for improving quality. Many development methodologies rely on refactoring, especially for agile methodologies but also in more plan-driven organizations. If barriers exist, they would undermine the effectiveness of many product-development organizations. We conducted a large-scale survey in 2009 of 3,785 practitioners' use of object-oriented concepts,[7] including questions as to whether they would refactor to deal with certain design problems. We expected either that practitioners would tell us our choice of design principles was inappropriate for basing a refactoring decision or that refactoring is the right decision to take when designs were believed to have quality problems. However, we were told the decision of whether or not to refactor was due to non-design considerations.

It is now eight years since the survey, but little has changed in integrated development environment (IDE) support for refactoring, and what has changed has done little to address the barriers we identified. We hope that presenting what we have learned will encourage improvement in refactoring support.

Reasons to Not Refactor

The reasons practitioners gave us for not refactoring a design believed to have quality problems can be categorized as follows:

Resources. Concern over the resources required was a frequently cited reason for not refactoring. The resource mentioned most often was time, as in "Deadlines often don't allow refactorings"; "Sometimes there is just no time"; and "No time no time."

Risk. Also frequently cited was the risk involved in making a change, in particular introducing new faults or other problems, as in "That kind of refactoring is time consuming and there is a large risk of introducing bugs" and "If it's working I leave it alone."

Difficulty. Another concern was the difficulty in making the change, as in "Inheritance is tricky to refactor correctly" and "This kind of refactoring is usually difficult."

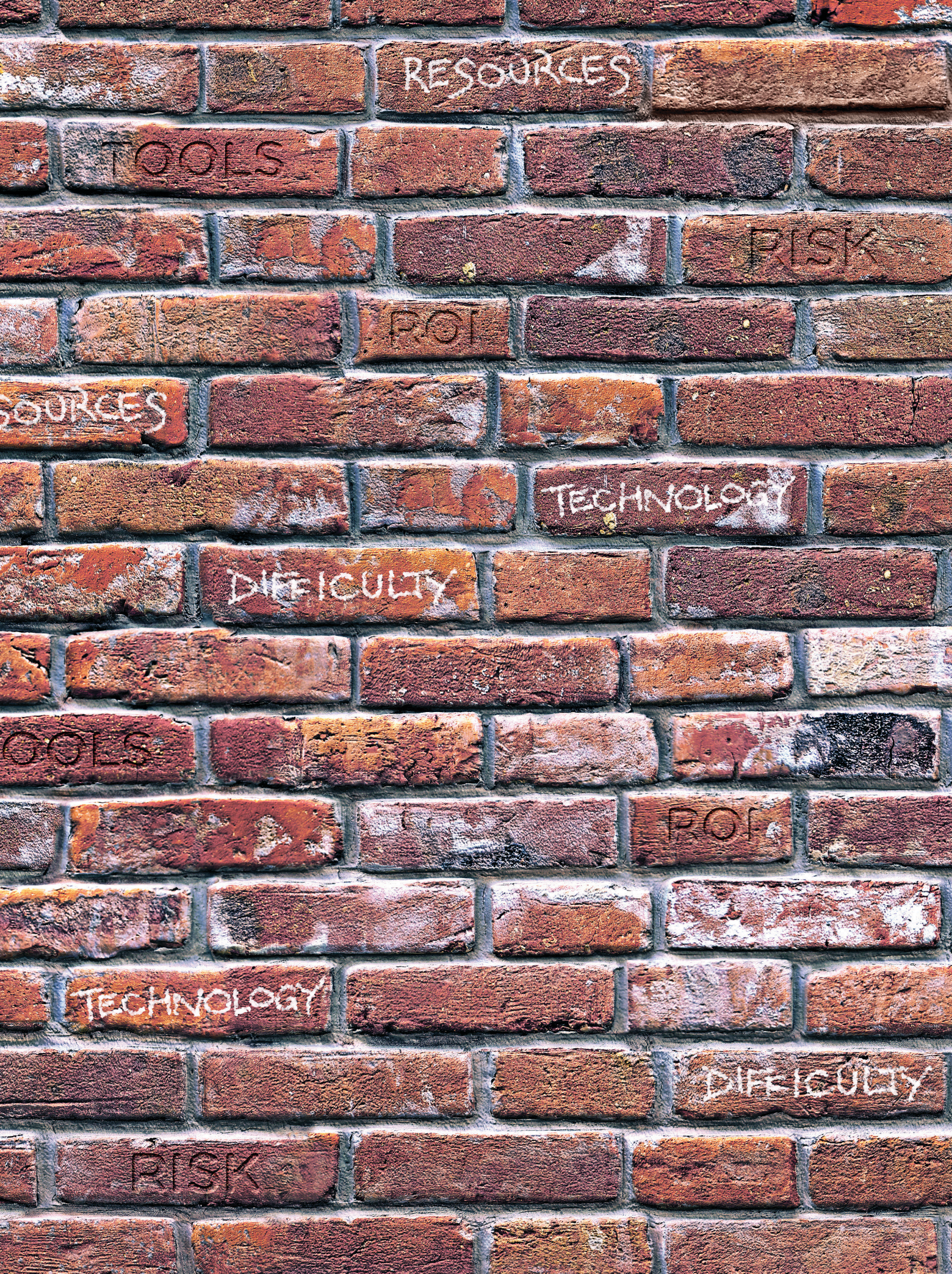
ROI. Participants acknowledged that while there may be benefits from refactoring, there are also costs, and the return on investment, or ROI, has to be considered, as in "Again, I have to weigh the costs and benefits. The benefits have to be clear before taking on the costs of refactoring, retesting, etc."

Technical. Participants reported a variety of constraints due to characteristics of the project that restricted

» key insights

- **Developers understand the value of refactoring but are often prevented from doing it due to factors beyond their control.**
- **Refactoring has benefits, but also costs and risks, and developers' inability to quantify them inhibits their use of refactoring.**
- **The decision to refactor is ultimately a business decision, and support is needed to allow developers to make the business case regarding whether or not to refactor.**

ILLUSTRATION BY JUSTIN METZ


RESOURCES
TOOLS
RISK
ROI
SOURCES
TECHNOLOGY
DIFFICULTY
OOLS
ROI
TECHNOLOGY
DIFFICULTY
RISK

their ability to refactor. Examples included having to implement a third-party interface that exceeds a limit, concern about the impact of any potential required changes on other parts of the system, the degree of familiarity with the code, dealing with legacy code, and lack of other support (such as test suites), as in "A large legacy codebase makes such refactoring tough"; "And with no tests, if I try to change things, they *will* break"; and "I will always refactor code I am creating; however, if that is not the case which is the most usual case then I don't have time budgeted for refactoring."

Management. Participants observed they did not always control what they spend their time on. Their managers or clients also had a say, as in "I want to, but the boss doesn't like it"; "deadlines and pointy-haired bosses"; and "I'd like to more, but the client isn't paying for it [sic]."

Tools. Inadequate tool support was mentioned as a reason for not refactoring; however, the tools referred to were not those that do the refactoring: "Source control makes this particular refactoring quite painful, that is, a lack of tool support."

Perhaps the most surprising theme is the complete lack of concern about the existence and quality of tool support for refactoring.

What Researchers Know

The results of our survey are still significant because they indicate missing support for refactoring. Although there has been much research on refactoring, including into developers' use and factors that may affect their use, such studies have focused mainly on what developers do when they have decided to refactor, rather than whether or not they will refactor. For example, a 2006 study by Xing and Stroulia[18] suggested refactoring was a common activity and raised questions about the quality of existing tool support. Murphy-Hill and Black[11] proposed five principles for supporting what they called "floss" refactoring: frequent refactoring intermingled with other kinds of program changes.[11] They also observed that existing tools rarely meet all principles and suggested this might explain the underutilization of tools they had observed. Consequently, tool-usability issues and their underuse is a recognized research focus.[12,17]

We know that developers refactor frequently but also that different developer groups have different refactoring behavior. Simple refactoring (such as the simple "rename" operation) is the most frequently applied. Generally, the more complex the refactoring operation is, the less frequently it is used.[12,17]

Kim et al.[10] studied the use of refactoring on Microsoft Windows, surveying more than 300 engineers who had carried out some form of refactoring of Windows, interviewing a subset. They identified such challenges performing refactoring as managing inter-component and inter-branch dependencies and maintaining backward compatibility.

More recently, Chen et al.[3] surveyed 105 practitioners using agile software-development processes to understand how they planned and practiced refactoring, finding that while 98.1% of those surveyed did careful planning when creating new features, only 40% addressed refactoring.

Existing research has studied the behavior and issues faced by developers who have made the decision to refactor. Here, we are interested in those developers who have made the decision *not* to refactor.

Survey Design

We used a survey to increase the number of developers we could reach, looking to determine when and if refactoring is used. But instead of asking about this outright we used realistic scenarios as a basis for the investigation. The survey participants were doing development in some object-oriented language. We asked what they consider to be good design principles and later in the survey if these good principles were broken, would they use refactoring to fix the problem (or not). We asked them to indicate what they thought was a good design, focusing on a reasonable level of class depth and size.

The use of a survey limits the kind of questions we could ask, so we chose two simple design heuristics as our focus. We asked about the basis for two of the metrics from the well-known set of object oriented metrics proposed by Chidamber and Kemerer,[5] namely Weighted Meth-

ods per Class, or WMC, and Depth of Inheritance Tree, or DIT. WMC is a form of size measure for classes, and we used its simplest form, a count of the number of methods in the class. DIT, which captures a characteristic of the inheritance hierarchy, is the length of the longest path from a class to a root of the hierarchy.

In the software engineering research community, these metrics are regularly presented as indicating possible design problems; the higher the values of WMC or DIT the more likely there are problems. Researchers have sought to establish thresholds for these and other metrics, including Chidamber et al.[4] and Shatnawi.[15] However, measurements of software systems indicate there are classes that have many methods or are very deep in the hierarchy.[2] The disconnect between theory and practice could be due to practitioners not being aware of the theory or knowing the theory but believing it to be wrong, perhaps because the theory really *is* wrong. We previously reported results for part of our survey indicating that 12.0% (452) of participants had a preference for a limit on the number of methods, whereas 25.2% (952) indicated a preference for a limit on the depth of a class; for details see Gorschek et al.[7] These results suggest a significant proportion of developers believe the theory.

We asked survey participants whether they thought there should be a limit to the number of methods or depth of classes and, if so, what the threshold should be. It is important to note we let participants give their own views of what is good or bad, then, and only then, asked whether or not they would refactor designs in which the number of methods or depth exceeded the threshold they themselves found to be relevant. We first asked them to indicate what they believed was a good threshold for size—WMC—and depth—DIT. We then asked if they would use refactoring to fix classes in light of their chosen limits. We also gave them the opportunity to provide free-text commentary on their chosen response. This commentary was revealing as to what factors affected participants' decision to refactor (or not).

Survey Questions

We asked our survey participants about practices relating to commonly taught design principles for object-oriented programming; the full survey and raw data is available on our website http://sefolklore.com. We designed the questions relevant to this article as two pairs of questions for two design metrics, one pair (Q14, Q16) relating to class size as measured by number of methods and the second pair (Q18, Q20) relating to depth of a class in an inheritance hierarchy. We also invited participants to provide a free-text comment to Q16 and Q20 relating to their response. The labels for the possible responses to the questions were used to report the results.

Q14. What do you think is the largest number of methods a class should have? (Choose one of the alternatives, and add a number in the textbox that replaces "N" in the text of the alternative you chose.)

***Never.* There should never be more than about N methods.**

***Try.* I try to avoid having more than about N methods in a class but allow exceptions in extreme circumstances.**

***Prefer.* I prefer to avoid having more than N methods in a class but am not fanatical about it.**

***Don't.* I don't really think about how many methods there are in a class but prefer to avoid having classes with more than N methods.**

***None.* I don't think there should be any limit on the number of methods in a class.**

Q16. What is the likelihood you will refactor a class (to create classes with fewer methods) if it has more methods than the limit you chose in question Q14?

***Always.* Always.**

***Most.* Most of the time.**

***Obvious.* Only if it is obvious how to refactor the class.**

***Nothing.* Only if I have nothing else to do.**

***Forces.* Only if someone forces me to.**

***Hardly.* Hardly ever.**

Q18. What do you think the maximum depth of a class should be in the inheritance hierarchy? (Choose one of the alternatives and add a number in the textbox that replaces "N" in the text of the alternative you chose.)

Similar responses, as with Q14, but replacing "number of methods" with "depth of class."

Q20. What is the likelihood you will refactor the class hierarchy (to reduce the maximum depth of classes) if there are classes deeper than the limit you chose in question Q18?

Same responses as with Q16.

We analyzed the data using content analysis.[13] Specifically, we coded the free-text answers using provisional coding[14] to map statements to two main categories of general motivation for "why" or "why not" refactoring was done. Subsequently, we coded the overall categories using simultaneous coding[14] to further refine types of motivation, as with, say, more detailed reasoning in terms of time, risk, or economic factors.

Survey Summary

We had 3,785 participants complete the compulsory questions of the survey (see the sidebar "Survey Questions"). The responses of greatest interest to us were from participants who indicated strong agreement with design principles that limit the number of methods or depth of a class. This is the category of developers who should be most interested in making sure that size and depth are not too large and deep, respectively, and so would presumably consider refactoring. We used the responses "Never" or "Try" to Q14 (for methods) or Q18 (for depth) as indicating such strong agreement. We know (as reported in Gorschek et al.[7]) that 452 (12.0%) participants indicated a strong preference for establishing a limit on the number of methods, whereas 952 (25.2%) indicated a strong preference for there being a limit on the depth of a class. The median choice for the limit on number of methods was 10 and for depth 3.

Table 1 reports the distribution of responses from all participants to the questions on the likelihood of refactoring a class that exceeds a specified number of methods (Q16) or depth (Q20). And Table 2 reports the distribution of responses to Q16 and Q20 for just the "believers" who indicated strong agreement with the design principles. The columns "Comments for Q16" and "Comments for Q20" report, for those who would impose a limit, how many provided a free-text comment.

We classified participants responding "Always" or "Most" to questions Q16 and Q20 as an indication that they would refactor classes they perceived as being "bad." Considerably more (952) agreed with having a limit on the depth of a class than on the number of methods (452). For those participants inclined to limit the number of methods, 265/452 (58.6%) would refactor classes that exceed their choice of limit (in bold). For those inclined to limit the depth of a class, 364/952 (38.2%) would refactor if the choice of depth was exceeded.

So developers are more inclined to limit the depth of a class than the number of methods but less inclined to do anything when that limit is exceeded. In both cases a significant proportion would not refactor their designs even when, by their own assessment, a particular design had problems. We wanted to know why this was so.

One possibility is there is something about the participants' backgrounds that influences their decision to refactor or not. For example, project managers may be more concerned about resources and risk, and inexperienced developers may view refactoring as too difficult. We collected demographic information for all participants, including amount of experience, programming language, operating system, role, type of development, and highest qualification. This information indicated our sample provided a broad representation of developers. Half of the 3,785 participants reported more than four years of experience, half had a bachelor's, and another 30% had master's or higher academic credentials. Within the role category, almost all reported having a programmer role (95.6%), but all other roles also had good representation; for example, architects made up 64.1% of the total. Participants could report more than one role. The highest reported language was C# (55.7%), though Java (49.4%) and C++ (45%) were also well represented; full details are available in Gorschek et al.[7]

Table 1. Likelihood of refactoring a class if limit of number of methods or class depth is exceeded (all participants).

	Q16 Refactor Methods		Q20 Refactor Depth	
Always	125	3.3%	120	3.2%
Most	845	22.3%	536	14.2%
Obvious	1,725	45.6%	1,660	43.9%
Nothing	415	11.0%	366	9.7%
Forces	128	3.4%	224	5.9%
Hardly	547	14.5%	879	23.2%
Total	**3,785**		**3,785**	

Table 2. Likelihood of refactoring by those who limit number of methods or class depth.

Response to Q16/Q20	Q14 Limit Methods		Comments for Q16		Q18 Limit Depth		Comments for Q20	
Always	38	**8.4%**	14	8.6%	76	**8.0%**	29	8.4%
Most	227	**50.2%**	84	51.9%	288	**30.2%**	102	29.7%
Obvious	145	32.1%	50	30.9%	415	43.6%	138	40.1%
Nothing	28	6.2%	7	4.3%	77	8.1%	21	6.1%
Forces	4	0.9%	2	1.2%	26	2.7%	12	3.5%
Hardly	10	2.2%	5	3.1%	70	7.4%	42	12.2%
Total	**452**		**162**		**952**		**344**	

Table 3. Comment category for those who would not refactor ("No") and those who would refactor ("Yes"); participants may be in more than one category.

Category	Limit Methods			Limit Depth		
	No		Yes	No		Yes
Resources	23	47.9%	27	49	36.0%	8
Risk	19	39.6%	6	43	31.6%	1
Technical	11	22.9%	11	25	18.4%	17
Difficulty	3	6.2%	5	13	9.6%	6
ROI	0	0%	0	13	9.6%	3
Management	1	2.1%	0	11	8.1%	3
Tools	1	2.1%	1	0	0%	0
Participants	**48**		**46**	**136**		**35**

We examined just those participants who agreed with having a limit, as they were responding to whether they would refactor to deal with a design issue. Our statistical analysis, which was based on logistic regression, showed two groups—architects and experienced C# programmers—were more inclined to refactor if the number of methods exceeded the participants' chosen limit. If the class depth exceeded the limit, C# developers and architects were again more inclined to refactor. Those in the programmer role were less inclined to refactor. Given that almost all participants indicated having a programmer role, this result may simply reflect the overall response.

These results are interesting, although none of the variables are good predictors, as they only weakly indicate whether or not developers will refactor. Neither participant role nor background gave a clear answer as to why they chose not to refactor even if their class design contradicted their view of good design. We had to dig deeper into the motivation offered by the respondents.

Participant Comments

To better understand what influences a decision to refactor, we examined the comments provided by those participants who indicated there should be a limit but would not refactor if the limit was exceeded. As outlined in Table 2, 162 participants who would limit the number

of methods provided comments ("Comments for Q16"), as did 344 who would limit the depth of classes ("Comments for Q20").

We identified the barriers (described earlier under the subhead "Reasons to Not Refactor") through analysis of free-text comments. Not all comments explained participants' reasons for not refactoring. The distribution of those who did is outlined in Table 3 ("No" columns). Some participants' comments fall into more than one category; for example, we classified the participant comment "Unless a class is ridiculously huge or easy to refactor, it's probably not worth the time or risk" as both "resources" and "risk."

While we were most interested in participants who would not refactor, we note that those who *would* refactor made similar comments. The numbers in each category are also reported in Table 3 ("Yes" columns).

Interpreting the Results

As with any survey data, especially qualitative data based on free text, we urge caution in its interpretation. Our categories are not necessarily orthogonal; a management decision to not refactor might be due to perceived cost, risk, or ROI calculation. Some technical concerns may be about risk, as when the reason given for not refactoring is that there are insufficient tests, the real concern may be due to the risks. Some management restrictions could be due to managers' concern over ROI, while concern regarding difficulty could be about the risk of introducing faults. Where we could, we chose the most specific category. Nevertheless, while the exact numbers may be uncertain, the themes are quite clear.

Perhaps the most surprising theme is the complete lack of concern about the existence and quality of tool support for refactoring. As other studies have shown,[12,17] such concerns are indeed real. We speculate they come into play only when the decision to refactor has been made, though in our survey the quality of tools did not appear to be a factor in making that decision. This suggests the emphasis within the research community on the quality of tool support is perhaps missing a significant issue.

The results indicate there is often little or no knowledge about what the actual ROI is when refactoring in a specific case.

Another theme we noted was that participants try to avoid the problem in the first place, as in "Would hope to never get to the case where a refactor of this type is necessary" and "This should be caught at design time. I am cautious to refactor this problem." One participant made this comment regarding "number of methods," and 16 participants made it for "depth of class." So there is awareness of the benefits of good design. However, even with the best of intentions, design quality can degrade, so avoidance is no solution and poor designs still need refactoring.

While our results suggest the removal of perceived design problems is not a priority, many responses indicated awareness of the benefit of refactoring, as in "Time taken to refactor the code is probably a good investment, since it will probably save developer time later." However, a common theme was that external factors (such as "deadlines" and "perceived risk") led to the decision to *not* refactor, as in "Unless a way to refactor is clear, time pressure generally prevents refactoring" and "Refactoring class hierarchies is usually really time-consuming and can have all kinds of unintended effects." Implicit in such comments is the suggestion that the benefits of refactoring do not always outweigh the cost—that refactoring is not always a good investment, as in "It sometimes takes more effort than it is worth to reduce the depth."

Whether such investment is worth it was more explicitly mentioned in other responses, as in "Again this is the time trade-off of do I see a clear opportunity to improve things and how long will it take?" In particular, the inability to determine ROI is a barrier, as in "If I can't make a reasonable estimation on how long it will take, I will leave it alone." That is, the decision to not refactor was not due to it being considered a bad idea but to uncertainty as to the benefit. The results indicate there is often little or no knowledge about what the actual ROI is when refactoring in a specific case.

As mentioned earlier, studies indicate developers prefer simple refac-

toring. For example, Murphy-Hill et al.[12] asked "Why is the 'rename' refactoring tool so much more popular than other refactoring tools?" The answer could be simply that the "rename" operation and less-complex operations are, in general, what developers need most often. However, our results suggest this is not the case. We see that developers do want to refactor complex cases and suggest that the preference for a simpler operation can be explained in ROI terms; that is, being simple, the cost is low, meaning it is almost always worth doing. More-complex operations take longer, are higher risk, and deliver benefit that is not so clear, as when one respondent said, "I will refactor the class only if it has a clear benefit."

We speculate that an unstated barrier is the difficulty translating a refactoring goal into refactoring operations.

What We Learned

We asked whether developers would refactor classes that did not meet characteristics they themselves thought were important, finding many (at least 40%, as in Table 2) would not. The reasons can be summarized as lack of resources, of information identifying consequences, of certainty regarding risk, and of support from management. This is consistent with studies on tool use. In particular, Vakilian et al.[17] reported that reasons for not using automation for refactoring involve trust, predictability, and complexity.

What we did not see was concern over lack of tool support. This was surprising, since, as Murphy-Hill and Black[11] noted, two advantages from using refactoring tools are lower error rates and less time required, so good tool support should go some way toward addressing developers' concerns.

Recent tool research has focused on primitive refactoring operations—"rename," "move," "method," and so on—whereas our question was goal-directed, asking, "Will you refactor to achieve this effect?" rather than simply, "Would you use this refactoring operation?" We speculate that an unstated barrier is the difficulty translating a refactoring goal into refactoring operations. While research has proposed refactoring operations to remove code smells (such as described by Bavota et al.[1]), Vakilian et al.'s survey[17] identified a preference for lightweight methods for invoking refactorings. We suggest providing lightweight goal-directed refactoring recommendations would be a fruitful area for future research.

We also did not see lack of interest in refactoring. Our participants were by and large willing to consider refactoring, but barriers prevented them doing so. Tool support for performing refactoring can address some of them but not all. For example, lack of certainty about ROI is a challenging but also possibly very rewarding area of research. We are not the first to highlight the importance of using ROI (see, for example, Harun and Lichter,[8] Kazman et al.,[9] and Szöke et al.[16]), but our results suggest it is a notable barrier to improving design quality. What is needed is a decision-support system that allows practitioners to be able to quantify benefit in the long and short term. This would help inform the decision as to whether the required resources are justified or the potential risk is tolerated. It would also allow developers and managers to make informed choices as to whether or not to refactor.

The reasons participants gave for not refactoring will mostly come as no surprise, especially to practitioners. What we provide is solid data to back up everyone's suspicions. Practitioners who have been prevented from refactoring for similar reasons can at least take heart that they are not alone, or even, it may seem, in the minority. While existing research goals regarding refactoring are reasonable, we feel they do not address problems actually faced by practitioners. As researchers, we need to better understand what barriers they face and better target our research to support them.

Validity of Results

Our data is representative because of the way we elicited it. We avoided leading questions, especially for free-text responses, asking only, "Please explain your answer." This led to comments that did not provide specific reasons for refactoring (or not). A more targeted question might have yielded more and clearer results but might also have biased the responses.

None of the material we used to recruit participants mentioned refactoring or gave specifics of the concepts we would be asking about. Consequently, unlike other studies, rather than include only developers known to refactor, there should have been no obvious bias for or against the use of refactoring. We recruited through personal contacts, word of mouth, and social media, supported by our website http://sefolklore.com and a YouTube video. The demographic data we collected reflected a variety of experience, roles, languages, level of qualification, and type of development.

We set a high bar in assessing whether a participant would agree with placing a limit on number of methods or class depth or would still refactor when that limit is exceeded. Different choices would change the distribution of the frequencies in the different categories somewhat, but the same trends would be evident.

Our wording of the possible responses might have affected what responses participants chose, but the barriers we identified came from the free-text commentary, which was unlikely to be affected by such concerns. Even those who would refactor mentioned some of the same concerns (as reported in Table 3 "Yes" columns). Participants from all demographics provided comments, so there is no obvious bias due to background, as was also the case with our target group—those who would limit number of methods or depth of inheritance.

We conducted the survey more than eight years ago and IDEs have since improved, so perhaps we would get different results today. However, little has changed in refactoring support, and we are not aware of any research directly addressing the issues our survey identified. More recent surveys have identified similar issues. For example, Kim et al.[10] reported concerns regarding lack of adequate test suites. Chen et al.[3] noted 21% of participants reported lack of support from management as a reason for not refactoring and 12.4% did no planning regarding refactoring. However, these studies surveyed developers who were already committed to refactor, whereas ours included those who chose not to refactor. That these more recent surveys report findings similar to ours suggests there has been little change in barriers to refactoring since our survey .

We used two metrics to identify potential design-quality issues based on published theories. It is possible the theories are wrong. However, our conclusions are based on comments by participants who (rightly or wrongly) believed the theories.

Conclusion

There are significant barriers preventing developers from refactoring to remove software design-quality issues, no matter how they are identified. Reducing or even eliminating the barriers has the potential to significantly improve software quality. One means is to provide refactoring support that is goal-directed rather than operations-directed. Another is to provide better quantification of the benefits, thus better informing the decision as to whether or not to refactor.

Acknowledgments

We thank our survey participants, many of whom made the extra effort to provide the comments we included and discussed here. We also thank the anonymous reviewers for their valuable comments and suggestions.

References

1. Bavota, G., Oliveto, R., Gethers, M., Poshyvanyk, D., and De Lucia, A. Methodbook: Recommending move method refactorings via relational topic models. *IEEE Transactions on Software Engineering 40*, 7 (July 2014), 671–694.
2. Baxter, G., Frean, M., Noble, J., Rickerby, M., Smith, H., Visser, M., Melton, H., and Tempero, E. Understanding the shape of Java software. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Oct. 2006), 397–412.
3. Chen, J., Xiao, J., Wang, Q., Osterweil, L.J., and Li, M. Perspectives on refactoring planning and practice: An empirical study. *Empirical Software Engineering 21*, 3 (June 2016), 1397–1436.
4. Chidamber, S.R., Darcy, D.P., and Kemerer, C.F. Managerial use of metrics for object-oriented software: An exploratory analysis. *IEEE Transactions on Software Engineering 24*, 8 (Aug. 1998), 629–639.
5. Chidamber, S.R. and Kemerer, C.F. A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering 20*, 6 (June 1994), 476–493.
6. Fowler, M. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, 1999.
7. Gorschek, T., Tempero, E., and Angelis, L. A large-scale empirical study of practitioners' use of object-oriented concepts. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering* (Cape Town, South Africa, 2010), 115–124.
8. Harun, M.F. and Lichter, H. Towards a technical debt-management framework based on cost-benefit analysis. In *Proceedings of the 10th International Conference on Software Engineering Advances* (Barcelona, Spain). International Academy, Research, and Industry Association, 2015.
9. Kazman, R., Cai, Y., Mo, R., Xiao, L., Feng, Q., Haziyev, S., Fedak, V., and Shapochka, A. A case study in locating the architectural roots of technical debt. In *Proceedings of the 37th International Conference on Software Engineering* (Firenze, Italy). IEEE Press, 2015.
10. Kim, M., Zimmermann, T., and Nagappan, N. A field study of refactoring challenges and benefits. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering* (Cary, NC). ACM Press, New York, 2012, article 50.
11. Murphy-Hill, E. and Black, A.P. Refactoring tools: Fitness for purpose. *IEEE Software 25*, 5 (Sept.-Oct. 2008), 38–44.
12. Murphy-Hill, E., Parnin, C., and Black, A.P. How we refactor, and how we know it. *IEEE Transactions on Software Engineering 38*, 1 (Jan. 2012), 5–18.
13. Robson, C. and McCartan, K. *Real World Research, Fourth Edition.* John Wiley & Sons, Inc., 2015.
14. Saldana, J. *The Coding Manual for Qualitative Researchers, Third Edition.* SAGE Publications Ltd., 2016.
15. Shatnawi, R., Li, W., Swain, J., and Newman, T. Finding software metrics threshold values using ROC curves. *Journal of Software Maintenance and Evolution: Research and Practice 22*, 1 (Jan. 2010), 1–16.
16. Szöke, G., Nagy, C., Ferenc, R., and Gyimóthy, T. A case study of refactoring large-scale industrial systems to efficiently improve source code. In *Proceedings of the 14th International Conference on Computational Science and Its Applications* (Guimarães, Portugal, June 30–July 3). Springer International Publishing, Cham, Switzerland, 2014, 524–540.
17. Vakilian, M., Chen, N., Negara, S., Rajkumar, B.A., Bailey, B.P., and Johnson, R.E. Use, disuse, and misuse of automated refactorings. In *Proceedings of the 34th International Conference on Software Engineering* (Zurich, Switzerland). IEEE Press, 2012, 233–243.
18. Xing, Z. and Stroulia, E. Refactoring practice: How it is and how it should be supported (an Eclipse case study). In *Proceedings of the 22nd International Conference on Software Maintenance* (Philadelphia, PA). IEEE Press, 2006, 458–468.

Ewan Tempero (e.tempero@auckland.ac.nz) is an associate professor of computer science at the University of Auckland, Auckland, New Zealand.

Tony Gorschek (tony.gorschek@gmail.com) is a professor of software engineering at the Blekinge Institute of Technology, Karlskrona, Sweden.

Lefteris Angelis (lef@csd.auth.gr) is a professor of statistics and information systems in the Department of Informatics at Aristotle University of Thessaloniki, Thessaloniki, Greece.

Watch the authors discuss their work in this exclusive *Communications* video. https://cacm.acm.org/videos/barriers-to-refactoring