Refactoring Tools Are Trustworthy Enough

John Brant

Refactoring tools don't have to guarantee correctness to be useful. Sometimes imperfect tools can be particularly helpful.

A COMMON DEFINITION of refactoring is "a behavior-preserving transformation that improves the overall code quality." Code quality is subjective, and a particular refactoring in a sequence of refactorings often might temporarily make the code worse. So, the code-quality-improvement part of the definition is often omitted, which leaves that refactorings are simply behavior-preserving transformations.

From that definition, the most important part of tool-supported refactorings appears to be correctness in behavior preservation. However, from a developer's viewpoint, the most important part is the refactoring's usefulness: can it help developers get their job done better and faster? Although absolute correctness is a great feature to have, it's neither a necessary nor sufficient condition for developers to use an automated refactoring tool.

Consider an imperfect refactoring tool. If a developer needs to perform a refactoring that the tool provides, he or she has two options. The developer can either use the tool and fix the bugs it introduced or perform manual refactoring and fix the bugs the manual changes introduced. If the time spent using the tool and fixing the bugs is less than the time doing it manually, the tool is useful. Furthermore, if the tool supports preview and undo, it can be more useful. With previewing, the developer can double-check that the changes look correct before they're saved; with undo, the developer can quickly revert the changes if they introduced any bugs.

Often, even a buggy refactoring tool is more useful than an automated refactoring tool that never introduces bugs. For example, automated tools often can't check all the preconditions for a refactoring. The preconditions might be undecidable, or no efficient algorithm exists for checking them. In this case, the buggy tool might check as much as it can and proceed with the refactoring, whereas the correct version sees that it can't check everything it needs and aborts the refactoring, leaving the developer to perform it manually. Depending on the buggy tool's defect rate and the developer's abilities, the buggy tool might introduce fewer errors than the correct tool paired with manual refactoring.

Even when a refactoring can be implemented without bugs, it can be beneficial to relax some preconditions to allow non-behavior-preserving transformations. For example, after implementing Extract Method in the Smalltalk Refactoring Browser, my colleagues and I received an email requesting that we allow the extracted method to override

continued on page 82

 0740-7459/15/$31.00 © 2015 IEEE

Trust Must Be Earned

Friedrich Steimann

Creating bug-free refactoring tools is a real challenge. However, tool developers will have to meet this challenge for their tools to be truly accepted.

WHEN I ASK people about the progress of their programming projects, I often get answers like "I got it to work—now I need to do some refactoring!" What they mean is that they managed to tweak their code so that it appears to do what it's supposed to do, but knowing the process, they realize all too well that its result won't pass even the lightest code review. In the following refactoring phase, whether it's manual or tool supported, minor or even larger behavior changes go unnoticed, are tolerated, or are even welcomed (because refactoring the code has revealed logical errors). I assume that this conception of refactoring is by far the most common, and I have no objections to it (other than, perhaps, that I would question such a software process per se).

Now imagine a scenario in which code has undergone extensive (and expensive) certification. If this code is touched in multiple locations, chances are that the entire certification must be repeated. Pervasive changes typically become necessary if the functional requirements change and the code's current design can't accommodate the new requirements in a form that would allow isolated certification of the changed code. If, however, we had refactoring tools that have been certified to preserve behavior, we might be able to refactor the code so that the necessary functional changes remain local and don't require global recertification of the software. Unfortunately, we don't have such tools.

There's also a third perspective—the one I care about most. As an engineer, and even more so as a researcher, I want to do things that are state-of-the-art. Where the state-of-the-art leaves something to be desired, I want to push it further. If that's impossible, I want to know why, and I want people to understand why so that they can adjust their expectations. Refactoring-tool users will more easily accept limitations if these limitations are inherent in the nature of the matter and aren't engineering shortcomings.

What we have today is the common sentiment that "if only the tool people had enough resources, they would fix the refactoring bugs," suggesting that no fundamental obstacles to fixing them exist. This of course has the corollary that the bugs aren't troubling enough to be fixed (because otherwise, the necessary resources would be made available). For this corollary, two explanations are common: "Hardly anyone uses refactoring tools anyway, so who cares about the bugs?" and "The bugs aren't a real problem; my compiler and test suite will catch them as I go." I reject both expla-

continued on page 82

0740-7459/15/$31.00 © 2015 IEEE

Point continued from page 80

an inherited method. Although the person requesting the change knew that this wouldn't preserve behavior, he also knew that the Extract Method transformation could be much quicker and more reliable with the tool than by hand. So, we promised to warn the developer about the issue but perform the transformation anyhow if the developer agreed.

Refactoring tools can also become more useful by relaxing the definition of "behavior." Under a strict definition, a program that executes more slowly than the original has changed its behavior. For some programs, such a change would be unacceptable. However, for most programs, executing a few milliseconds slower is acceptable. So, most refactoring tools omit execution time from the preserved behaviors.

Reflection is another area in which the behavior preservation requirement is usually relaxed. For example, if you employ reflection to use strings to find classes by name, any Rename Class refactoring will break your program. By allowing refactoring tools to ignore certain behaviors, we can build more useful tools. Consider replacing a set of radio buttons with a drop-down list. Such a change obviously isn't behavior preserving because the user will interact with the application differently. However, if we look at the behavior on the basis of what's saved to the database or what other widgets get enabled or disabled when users make a selection, it could be considered a refactoring.

Many people believe that the most important part of automated refactoring tools is correctness. They feel that without correctness, the tools won't be trusted, and without trust, they won't be used. However, I believe that helping developers work more efficiently is much more important than the dogma of behavior preservation. If such a tool can help developers, they'll use it, even though they can't trust that it will always be correct.

JOHN BRANT is an independent consultant and the coauthor of the Smalltalk Refactoring Browser. Contact him at brant@refactoryworkers.com.

Counterpoint continued from page 81

nations—the first because it denies refactoring the status of a relevant problem requiring tool support, the second because it implies a dependence on testing that refactoring-tool users might find unacceptable. Besides, coming up with excuses for ignoring bugs, rather than doing our best to fix them, won't increase trust in refactoring tools.

After working on refactoring tools for more than seven years, I've concluded that ridding them of their bugs is actually much harder than most tool users would believe. With respect to maintaining well-formedness (that is, the tool doesn't introduce compilation errors), static checking (as implemented by the compiler, which is basically a decision problem) must be extended to the much harder problem of computing the additional changes required to maintain a refactored program's well-formedness (basically a search problem). The idea of implementing a refactoring as a sequence of steps ("mechanics") grossly underrates the technical effort required to do this.

With respect to preserving behavior, the problems are even harder. Here, the boundaries are basically set by the precision of available static analyses. Surely, some programming languages are more amenable to such analyses than others, but I doubt whether programmers will ever adopt a programming language because of its "safe refactorability." So, we must accept that guarantees regarding behavior preservation can be given in only fairly limited cases (which might nevertheless be worthy of refactoring-tool support).

So, from my experience, in terms of reliability, current refactoring tools don't play in the same league as other programming tools, notably compilers, debuggers, or version control systems. This doesn't make them useless; having less-than-perfect refactoring tools is better than having no refactoring tools. Yet, to deserve users' trust, refactoring-tool builders can't be satisfied with the status quo but must continuously demonstrate a desire to build correct tools.

FRIEDRICH STEIMANN is full professor and chair of Programming Systems at Fernuniversität in Hagen. Contact him at steimann@acm.org.

STEIMANN RESPONDS

Software is soft because it's quickly changed. Refactoring tools make changing software even quicker. When I use a refactoring tool and the refactoring affects more than, say, a dozen distinct locations in the code, I usually look at the first couple of changes in the preview. I then find that I'll sacrifice much of the promised speedup if I try to manually check the correctness of all the scheduled changes. So, I accept all changes and wait for what the compiler has to say. If it says everything is okay, I usually don't worry about correctness and happily proceed with my work.

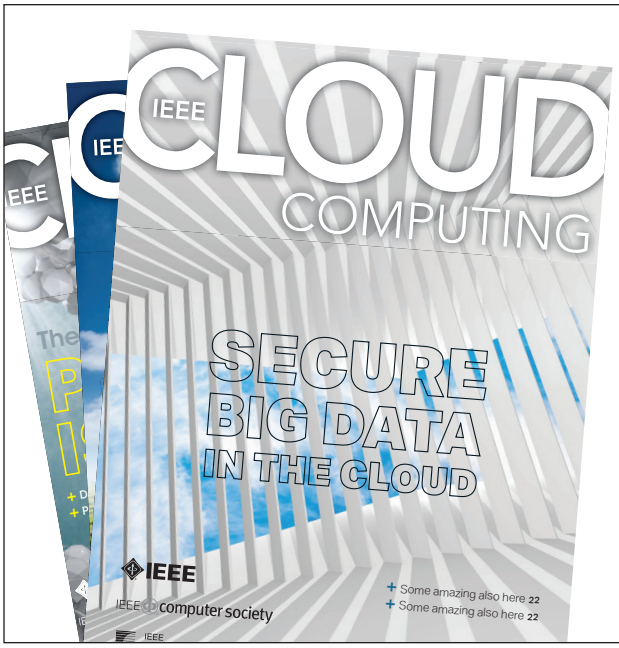
However, with this extra speed, which I certainly enjoy ("Wow, am I productive today!"), I tend to lose control of my code. If a bug pops up sometime later, I'm not sure who (me or the tool) introduced it when or where—the speed of development has overrun me. True, manual refactoring is slower and likely introduces bugs too; however, it leaves me more conscious of what I actually changed. Only if refactoring tools are correct is this consciousness never needed.

BRANT RESPONDS

First, I believe that refactoring tools and optimizing compilers do play in the same league. Both try to change code while preserving behavior. Optimizing compilers have a few more decades of research, so they're a little further along than some refactoring tools. However, they still have their issues. Many have command line switches that let users disable optimizations when they aren't working.

Second, although it's good to research what code analysis can and can't do, we can also do state-of-the-art research to determine what refactoring is needed most or create refactoring frameworks that support making new refactorings quickly.

Finally, I'm not against having refactoring tools that have been certified to preserve behavior. However, given that few compilers have such certification and that few projects get certified, I believe that time would be better spent researching other issues that affect more people.



Subscribe today!

IEEE Computer Society's newest magazine tackles the emerging technology of cloud computing.

computer.org/cloudcomputing

IEEE computer society

IEEE COMMUNICATIONS SOCIETY