Martin Fowler
fowler@acm.org

Refactoring: Doing Design

After the Program Runs

If you have worked with objects for any length of time, you should have heard about the notion of iterative development. The idea of iteration is that you cannot get a design right the first time, you need to refine it as you build the software. By iterating over the design several times, you get a better design. Even throwing away code helps; indeed it is the sign of a good project that it does regularly throw away code. To not do so indicates a lack of learning—and keeping bad ideas around.

Iteration is a great principle to discuss, but it has some problems in practice. If you make a change to existing code, does that not carry the risk of introducing bugs? While people have come up with various techniques and methods for design in advance, there is not much discussion of how to apply them in an iterative process, or how to do the iteration in a controlled and efficient manner.

Refactoring is the first technique I've come across that is explicitly about doing iterative development in a controlled manner. It starts with software that currently works but is not well suited to an enhancement you wish to make. Refactoring is the controlled process of altering the existing software so it's design is the way you want it now, rather than the way you wanted it then. It does this by applying a series of particular code transformations, each of which are called *refactorings*. Each refactoring helps change the code in a way that both is rapid and does not introduce bugs. (Yes, I know that means the word refactoring means two different things—I guess overloading is just ingrained in this industry!)

How does refactoring do this magic? Essentially, there are two routes. The first is manual; the second relies on tools. Although using tools is less common at the moment, I'll start with that.

Refactoring With Tools

The essence of the tools-based approach is the notion of the *semantics-preserving transformation*. This is a transformation that you can prove will not change the execution of the program. An example of this is a refactoring called *extract method*. If you have a long section of procedural code, you can make it easier to read by taking a suitable chunk of the procedure and turning it into a separate method. To do this with a tool, you select the code you wish to extract, and the tool analyzes this code, looking for temporary variables and parameters. Depending on what the selected code does with these local variables, you may have to pass in parameters, return a value, or even not be able to do the extraction. The tool does this analysis, prompts for the new method name, asks you to name any parameters, and creates the new method with a call from the old one. The program works exactly the same as it did before but is now a little easier for a human to read. This is important: Any damn fool can write code that a computer can understand, the trick is to write code that humans can understand.

The tool builder has to prove that the refactoring is semantics preserving, then figure out how to provably carry out the transformation in the code. It's hairy stuff and requires a lot of knowledge of compiler technology, but it can be done. The benefit is immediate. Once the transformation is encoded, you can use it with complete confidence. A semantics-preserving transformation is never going to add a bug to your program.

Tools like this are not science fiction. Bill Opdyke, while a student at the University of Illinois, proved several of these refactorings. Since then, two other graduate students, John Brant and Don Roberts, have produced a refactoring browser for Smalltalk that implements these refactorings, and a few more. If you are a Smalltalker, you should download it from www.cs.uiuc.edu/users/droberts/Refactory.html.

The refactoring browser is an awesome tool. With it I can safely reorganize some pretty ugly code (and yes even Smalltalk can get ugly). But what if you are working outside of Smalltalk? Is refactoring still applicable? The answer is yes, although not with tool support.

Refactoring Without Tools

Although manual refactoring is not as easy, it is still possible—and useful. It boils down to two principles: take small steps and test frequently.

Humans can use the same semantics-preserving transformation tools use, but in method extraction you have to look at the local variables yourself. It takes a little longer, but it isn't too difficult, because you're looking at only a small section of code. You then move the code over, put in the call, and recompile.

If you did it correctly, you won't get any bugs. Of course, that's a big if, so this is where tests come in. With manual refactoring you need to have a battery of tests that exercise the code sufficiently to give you confidence that you won't introduce new bugs. If you build self-testing code, then you will already have those tests. If not, you have to build them yourself, but of course tests are useful anyway, since they make it easier to add new function as well as refactor.

Martin Fowler is an independent consultant based in Boston, MA.

56
MindQ Ad

Refactoring to Understand Code When you follow a rhythm of small change, test, small change, test, you can make some remarkably large changes to a design. I've gone into some pretty nasty lumps of code, and after a few hours found class structures that radically improve the software design. I don't usually have the design in mind when I start. I just go into the code and refactor initially just to understand how the code works. Gradually, as I simplify the code, I begin to see what a better design might be and alter the code in that direction.

Refactoring to understand code is an important part of the activity. This is obviously true if you are working with someone else's code, but it is often true with your own code as well. I've often gone back to my own programming and not fully understood what I was doing from the code, or gained a better understanding by comparing it to later work. Of course, you can do a similar thing by commenting, but I've found it better to try to refactor the code so its intention is clear from the code, and it does not need so many comments. Some refactorers go so far to claim that most comments are thus rendered unnecessary. I don't go that far, but I do prefer to reach clarity through good factoring if I can.

The Value of Refactoring Refactoring has become a central part of my development process, and of the process I teach through my seminars and consulting work. It's a design technique that is a great complement to the up-front design techniques advocated by the UML and various methods.

Its great strength is that it works on existing software. I'm rarely faced with a green field. Often I have to work with some form of existing code base. Refactoring provides a way to manipulate and improve the code base without getting trapped in bugs.

When working with new code, it gives developers the ability to change designs and make the iterative development process much more controlled. I've noticed that it makes a significant improvement to development speed. When trying to add new function to software, several projects have seen how it is quicker to first refactor the existing software to make the change easier. Thus I don't recommend setting aside time to refactoring. It should be something you do because you need to add a feature or fix a bug. That also makes it easier to justify to skeptical managers.

The biggest problem with refactoring at the moment is finding out more about how to do it. I'm in the process of writing about what I, and various others, have learned about the process, but that is still a way from publication. You can find some more information, including references and an introductory example of refactoring, at ourworld.compuserve.com/homepages/Martin_Fowler/.

I hope I've stimulated you to find out more about refactoring. I believe it is going to be one of crucial new developments of the next few years, and tools that include refactorings will be very important to software developers. I'd be interested to know what you think; drop me a note at fowler@acm.org.