Refactoring

To the reader

This is a very early draft of a potential book project on refactoring. I have become convinced that refactoring is a very important part of effective software development, and yet is not talked about enough. I would like to refer people to a book, but nobody seems to be working on one. Refactoring is something I have done for a long time, but never really paid much attention to. The book project is very much inspired by working with Kent Beck on a major project, where I noticed the attention he paid to refactoring and the effect it had on the team. Inspiration also came from conversations with Ward Cunningham, Ralph Johnson, and the University of Illinois Refactorers.

This book comes in three parts. The first part gives you an overview of what refactoring is, and why you should do it. It also gives tips on the project management issues of refactoring, how to make it work, and how it affects the rest of the process. If you are a project manager or team leader this bit is for you. If you are a developer this is the bit to show your boss.

The second part takes a number of poorly factored programs and gives a blow by blow description of how I set about refactoring them. This is the best way I can think of to show you how refactoring works in practice. Usually I refactor a little at a time throughout the project, so the code does not get into the state that I start with in these examples. However the examples do show how refactoring can work with an awkward program, even a purely procedural program.

The last part is a catalog of refactorings. Each refactoring is described in a set form (I'm still wondering whether to use the P word). Each refactoring is given a brief name (in verb form). Then there is a brief statement of the problem the refactoring deals with and what the refactoring does as a solution. Next comes a general discussion of the issues around using the refactoring. Lastly I give a series of steps to go through when you do that refactoring. At some point I will also add some examples or refer to examples from the second section.

At the moment this book is still in an in-progress state. I am particularly interested in comments as to how you feel this can be improved. Various things to do are listed in the list with 'TK'.

The introduction needs more material, the TKs indicate the things I can think of to put in it.

So far I have three chapters in the second part. One is a simple 'first example' that shows some simple classes for describing charges for renting videos. One shows taking several Java classes that ought to be linked by inheritance and showing how they can be combined into a hierarchy. The other takes a procedural program (a master-transaction update program) and refactors it in Smalltalk. The next one I intend to write is one that starts with a program where there is a series of dumb data objects being manipulated by an over complex GUI. I am also intending to write a chapter that shows how to take a program that uses arrays a lot and refactor it to use collections.

I have pretty much decided to do all the book in Java, in this case I will probably change the master-transaction chapter to use Java.

The catalog is also in an early state. For many refactorings I have only the problem/solution pair, others add just the steps. Examples of entries in a more full form are Create Foreign Method, Decompose a Method, Generalize Method, and Preserve Whole Object. I haven't decided yet whether to put examples into the catalog, or to point the reader to examples in the second part. I will have a better idea when I have done more of the second part chapters.

Introduction

Once upon a time a consultant made a visit to a development project. The consultant looked at some of the code that had been written, it was a class hierarchy at the center of the system. As he wandered through the hierarchy he saw that it was rather messy. The higher level classes made certain assumptions about how the classes would work, assumptions that were embodied in inherited code. That code didn't suit all the subclasses, however, and were overridden quite heavily. If the superclass had been modified a little, then much less overriding would have been necessary. In other places some of the intention of the superclass had not been properly understood, and behavior that was there in the superclass was duplicated. In yet other places several subclasses did the same thing with code that could clearly be moved up the hierarchy.

The consultant recommended to the project management that the code be looked at and cleaned up, but the project management didn't seem too enthusiastic. The code seemed to work and there were considerable schedule pressures. They said they would get around to it at some later point.

The consultant had also shown the programmers who had worked on the hierarchy what was going on. The programmers were keen and saw the problem. They knew that it wasn't really their fault, sometimes these things need a new pair of eyes to spot the problem. So they spent a day or two cleaning up the hierarchy. When they had finished they had removed half the code in the hierarchy, without reducing its functionality. They were pleased with this, and found that now it was easier to add new classes to the hierarchy.

The project management was not so pleased. Schedules were tight and there was a lot of work to do. These two programmers had spent two days doing work that had done nothing to add the many features that the system needed to deliver in a few months time. The old code had worked just fine. So the design was a bit more 'pure' a bit more 'clean'. The project needed to ship code that worked, not code that would please an academic. The consultant suggested that this cleaning up be done with the rest of the system. Such an activity would halt the project for a week or two. All to make the code look better, not to make it do anything that it didn't already do.

How do you feel about this story? Do you think the consultant was right to suggest further cleaning up? Or do you follow that old engineering adage "if it works, don't fix it"?

I must admit to some bias here. I was that consultant. Six months later the project had failed due to code that was too complex to debug or to tune to an acceptable performance.

If you thought the consultant was right, then this is a book you should read, for it will tell you how to clean up code, to keep it clean, and to treat the cleaning process as an essential part of software development. If you thought the manager was right I hope to convince you otherwise. To show you that cleaning up code can have surprisingly deep effects in a programming project. Indeed that this cleaning of code actually *improves* productivity.

That failed project was a perfect example of that. Kent Beck was brought in to restart the project, an exercise that involved rewriting almost the whole system from scratch. Amongst his diverse weapons was a determination to use refactoring at all stages of the project. The success of this project, and role refactoring played in this success, is what inspired me to write this, so that I could pass on the knowledge that Kent and others have learned in using refactoring to improve the quality of software.

What Is Refactoring?

Refactoring is the process of changing code in such a way that it does not alter the external behavior of the code, yet improves its internal structure. In essence when you refactor you are improving the design of the code after it has been written.

"Improving the design after it has been written", that's an odd turn of phrase. In our current understanding of software development we believe that we do design, and then we code. A good design comes first, and the coding comes second. Over time the code will get modified and the integrity of the system, its structure according to that design gradually fades. The code slowly sinks from engineering to hacking.

Refactoring is the opposite to this. With refactoring you can even take a bad design, chaos even, and rework it into well designed code. Each step is simple, even simplistic. You move a field from one class to another, pull some code out of a method to make into its own method, push some code up or down a hierarchy. Yet the cumulative effect of these small changes can radically improve the design. It is the exact reverse of the normal notion of software decaying.

Why Should You Refactor?

I don't want to proclaim refactoring as the cure for all of software ills. It is no 'silver bullet'. Yet it is a valuable tool, a pair of silver pliers that helps you keep a good grip on your code. A tool that can, and should, be used for several purposes.

The most obvious reason for refactoring is the one I've alluded to above: **refactoring improves the structure of existing code**. Without refactoring the program's design will decay. As people make changes to the code — changes done to realize short term goals, or changes made without a full comprehension of the design of the code — that code will lose its structure. It becomes harder to see the design by reading the code. Refactoring is rather like tidying up the code. Working to remove bits that aren't really in the right place. Loss of structure to code has a cumulative effect. The harder it is to see the design in the code, the harder it is to preserve it, and the more rapidly it decays. Regular refactoring helps it retain its shape.

Poorly designed code usually takes more code to do the same things, often because the code quite literally does the same thing in several places. **Refactoring eliminates duplicate code**. The importance of this lies in future modifications to the code. Reducing the amount of code won't make the system run any faster, its affect on the programs footprint is rarely significant, but it makes a big difference to the modification of the code. The more code there is, the harder it is to modify correctly. There's more code to understand. You change this bit of code here but the system doesn't do what you expect because you didn't change that bit over there that does much the same thing in a slightly different context. By eliminating the duplicates you ensure that the code says everything once, which is really the essence of good design. Refactoring reduces the size of your code, that's a good thing.

An old engineering adage I like is "the more there is, there more there is to go wrong". Duplicate code often leads to bugs, so refactoring remove bugs by removing unnecessary code. It goes deeper than this, however. When you refactor you are forced to look again at your own code, or concentrate on a piece of new code. The best refactoring is often done the first time you work with someone else's code. **Refactoring is a very effective way of spotting bugs**, especially when you are trying to improve its structure. Indeed there is a whole technique of software inspection, with many studies showing that inspection is more effective than testing at finding bugs. Refactoring is a very active and engaged form of inspection, and I've often found bugs when refactoring.

Programming is in many ways a conversation with a computer. You write code that tells the computer what to do, and it responds by doing exactly what you tell it. In time you close the gap between what you want it to do and what you tell it to do. Programming in this mode is all about saying exactly what you want. But there is another user of your source code. Someone who will try to read your code in a few months time to make some changes. We easily forget that extra user of the code, yet that user is actually the most important. Who cares if the computer takes a few more cycles to compile something? Yet it really matters if it takes a programmer a week to make a change that would only have taken an hour if she could have understood your code more easily.

The trouble is that when you are trying to get the program to work, you are not thinking about that future developer. It needs a change of rhythm to make those changes that make the code easier to understand. **Refactoring helps you to make your code more readable.** When refactoring you have code that works, but is not ideally structured. A little time spent refactoring can make the code better communicate its purpose.

Useful code never stays still. You always need to change it do something that was not expected when it was written. When looking at unfamiliar code you have to try to understand what it does. You look at a couple of lines and say to yourself, oh yes that's what this code is doing. With refactoring you don't stop at the mental note. You actually change the code to better reflect your understanding, and then test that understanding by re-running the code to see if it still works. In this case **refactoring helps you understand unfamiliar code**.

Now you have understood the code you need to do something new with it, something that was not envisaged by the original writer. If this is the case the code is not structured to help you, indeed you may have to do a work-round to get around this failing in the original code. So? Don't work around the original code. You can refactor the code to change the way it works so it better supports you. **Refactoring enhances code to support new requirements.** The techniques are all about not changing what it does, you won't break the existing functionality. But now the code will support your new requirements better, allowing you to make the same changes with less new code to write and maintain.

How to Reduce the Costs of Refactoring

Is that you I hear saying but, but, but? The idea of cleaning code sounds good, but it involves a cost. You do have to spend time doing it, and it is only worthwhile if the time you spend doing it is less than the time you gain from having done it. In many cases people try to clean up code, but then spend many fruitless hours trying to hunt down some bug that they introduced while cleaning up. Once this is identified, fixed, and the shins given a good kicking, it is reasonable to conclude that cleaning up code that works is just not worth the effort.

But this sort of thing need not happen. When I refactor I hardly ever spend any time debugging. Most of the time the changed code runs with no errors. When I do get an error it is usually very quick to find. If it isn't quick to find I don't spend time debugging. I get these benefits not because of any personal cleverness, but because I cultivate good habits.

The first habit is to have **comprehensive unit tests**. Before you can begin refactoring you need to have worked up a thorough set of tests that check that your code does what it is supposed to do. The tests should be black box as far as the code you are working on is concerned (after all you are going to change it) and they must be thorough. Furthermore the tests must be easily invoked and checked. Its no good using tests that take minutes of pointing and clicking to carry out, and more minutes of checking against figures to verify. The tests must be invoked from a single command and be self-verifying. That is they either say "all OK" or "here are the failures". Inevitably the tests will take a while to execute, but it should not be so long that it significantly affects your code-compile-test cycle. If tests are too much trouble to execute and check, then you won't use them, and you have to test frequently when you refactor.

We inevitably make mistakes when we code, that is part of being human. We only find those mistakes when we test. If the tests are rarely executed, after hours of work, then we are faced with a complicated debug. However if the tests are run frequently then you immediately have some idea of what went wrong. Nine times out of ten I can look at the error in the debugger and go straight to the problem. After all it must be in the last change I made.

That leads me straight to the next cost reduction feature: **take small steps**. Make each refactoring step as small as you can, then compile and test. If you want to move several fields and methods from one class to another, then move each feature one at a time, and test after each move. If your code-compile-test cycle is on the long side, then you will need to take slightly longer steps, but in general you should take the smallest step you possibly can. That way you find your mistakes quickly.

The third item to make the process easier is assumed in this book, **use an object-oriented development environment**. The great benefit of OO approaches, indeed the only benefit I really claim for objects, is that it makes it easier for you to make changes to code. This is because with objects it is easy to change the interface of a module independently from its implementation. With refactoring you need this so that you can make small changes and isolate them. A vital part of this is *Architectural Substitution*: the ability to change basic architectural assumptions in your system, even to the point of having the old architecture and the new architecture co-exisiting in your code while you change over.

TK history of the development of refactoring. Ensure the true developers of the technique get due credit.

TK refactoring and the evolutionary development process

TK refactoring and teams: pair programming.

TK tools for refactoring

TK optimizing (to allay fears about refactoring and performance)