Preface

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 quicker and easier both to add new classes to the hierarchy and to use the classes in the rest of the system.

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"?

If you thought the consultant was right then you will enjoy this book. 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 then you desperately need this book. The book will show you that cleaning up code can have surprisingly deep effects in a programming project, and will actually improve productivity. It is not a technique used by lazy programmers to avoid adding features to a system (though it can be), but a technique used by master programmers to build flexible, highly productive systems.

- - Ralph Johnson

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

Kent Beck was brought in to restart the project, an exercise that involved rewriting almost the whole system from scratch. He did several things differently, but one of the most important was to insist on continuous cleaning up of the code using refactoring. 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 a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure. It is a disciplined way to clean up code that mini-

mizes the chances of introducing bugs. 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.

With refactoring you find the balance of work changes. Instead of design occurring all up front, you find that design occurs continuously during development. The design learns from the implementation and the resulting interaction leads to a program whose design stays good as the development continues.

What’s in this Book

This book is a guide to refactoring, written for a professional programmer. My aim is to show you how to do refactoring in a controlled and efficient manner. In such a way that you don’t introduce bugs into the code, but instead methodically improve the structure. I use three kinds of discussion to show you that.

- **Principles**: in a few chapters I’ll talk about basic principles of refactoring, why you should do it, limitations, the importance of unit tests, tools, and the like.
- **“Over the shoulder” examples:** here I’ll walk you step by step through a bout of refactoring. We’ll start with some poorly factored code, and I’ll show you how I go about refactoring it. This

should give you a sense of the process of refactoring. How often you proceed without a clear goal, until you clean up the code enough to see where you need to go. How you combine different kinds of refactoring in different places.

- **Catalog of refactorings:** the heart of this book is a catalog of refactorings. This is by no means a comprehensive catalog. It is the beginning of such a catalog. It includes the refactorings that I have used in this book and come across in other places. It acts as a reference for me. When I want to do something, like *Replace Switch with Polymorphism (147)*, it reminds how to do it in a safe, step-by-step manner. I hope this is the section of the book you'll often come back to.

Structure of the Book

It's traditional to start books with an introduction. While I agree with that principle, I don't find it's easy to introduce Refactoring with a generalized discussion or definitions. So I start with an example. Chapter 1 takes a small program with some common design flaws, and refactors into a more acceptable object-oriented program. Along the way we'll see both the process of refactoring and the application of several useful refactorings.

In Chapter 2 I'll cover more of the general principles of refactoring, some definitions, the role of testing and performance tuning, and I'll outline some of the problems with refactoring. For Chapter 3 I have Kent Beck to help me describe how to find bad smells in your code, and how to deodorize them with refactorings. Testing plays a very important role in refactoring, so Chapter 4 looks at how to build tests into the code using a simple open source Java testing framework.

Now we come to the catalog of refactorings, which stretches from Chapter 6 to Chapter 12. As I've said this is not a comprehensive catalog, but it will give you a good start by describing a core set of refactorings. With each refactoring I discuss why it is a good idea to do it, give a step by step description of how to do it, and add an example to help make it clearer.

In writing this book I'm describing the fruit of a lot of research done by others. Chapter 13 and Chapter 14 are guest chapters by the original

researchers which describe the issues in adopting refactoring and taste of what refactoring tools will look like.

Chapter 15 is a final, longer example of how you can bring several similar classes together into an inheritance hierarchy. Some readers find the length of the example is too much to wade through. Others feel that an example of this length is essential to understand how refactoring works in a more real world problem. So I leave it up to you, try it and see how you go.

Who Should Read this Book?

This book is aimed at a professional programmer, someone who writes software for a living. The examples and discussion include a lot of code to read and understand. The examples are all in Java. I chose Java because it is increasingly a well-known language that can be easily understood by anyone with a C background. It is also an object-oriented language, and object-oriented mechanisms are a great help in refactoring.

I should stress that while refactoring is focused on the code, it has a large impact on the design of system. So I believe it is vital for senior designers and architects to understand the principles of refactoring and to use them in their projects. Refactoring is best introduced by a respected and experienced developer who can best understand the principles behind it and adapt those principles to the specific workplace. This is particularly true when you are using a language other than Java, as you have to adapt the examples I've given to other languages.

Building on the Foundations Laid by Others

I need to say right now, at the beginning, that I owe a big debt with this book. A debt to those whose work over the last decade has developed the field of refactoring. Ideally one of them should have written this book, but I ended up being the one with the time and energy.

Two of the biggest proponents of refactoring are **Ward Cunningham** and **Kent Beck**. They used it as a central part of their development process in the early days, and have adapted their development processes to take advantage of it. In particular it was my collaboration with Kent that really showed me the importance of refactoring, an inspiration that led directly to this book.

Ralph Johnson leads a department at the University of Illinois at Urbana-Champaign that is notable for it's practical contributions to object technology. Ralph has long been a champion of refactoring, and several of his students have worked on the topic. **Bill Opdyke** developed the first detailed written work on refactoring in his Ph.D. thesis. **John Brant** and **Don Roberts** have gone beyond writing words, into writing a tool, *Refactory*, for refactoring Smalltalk programs.

I also want to thank the team who developed the Chrysler Comprehensive Compensation system (C3). Working with them and Kent cemented the principles and benefits of refactoring into me on a first hand basis.

tbd: Finish acknowledgment list.

Martin Fowler
Melrose, Massachusetts
mailto:fowler@acm.org
http://ourworld.compuserve.com/homepages/martin_fowler