Refactoring Chapter 1-2

**Chapter 1. Refactoring, a First Example**

* Programs should be refactored for better understanding
* Even so the program works. Is this not just an aesthetic judgment, a dislike of ugly code? It is

until we want to change the system. The compiler doesn't care whether the code is ugly or clean.

But when we change the system, there is a human involved, and humans do care. A poorly

designed system is hard to change. Hard because it is hard to figure out where the changes are

needed. If it is hard to figure out what to change, there is a strong chance that the programmer

will make a mistake and introduce bugs.

* The problem with copying and pasting

code comes when you have to change it later. If you are writing a program that you don't expect

to change, then cut and paste is fine. If the program is long lived and likely to change, then cut

and paste is a menace.

* You may be tempted to make the fewest possible changes to the program; after all, it works fine.

Remember the old engineering adage: "if it ain't broke, don't fix it." The program may not be

broken, but it does hurt. It is making your life more difficult because you find it hard to make the

changes your users want. This is where refactoring comes in.

* When you find you have to add a feature to a program, and the program's code is not

structured in a convenient way to add the feature, first refactor the program to make it

easy to add the feature, then add the feature.

**The First Step in Refactoring**

Whenever I do refactoring, the first step is always the same. I need to build a solid set of tests for

that section of code. The tests are essential because even though I follow refactorings structured

to avoid most of the opportunities for introducing bugs, I'm still human and still make mistakes.

Thus I need solid tests.

Because the statement result produces a string, I create a few customers, give each customer a

few rentals of various kinds of films, and generate the statement strings. I then do a string

comparison between the new string and some reference strings that I have hand checked. I set

up all of these tests so I can run them from one Java command on the command line. The tests

take only a few seconds to run, and as you will see, I run them often.

An important part of the tests is the way they report their results. They either say "OK," meaning

that all the strings are identical to the reference strings, or they print a list of failures: lines that

turned out differently. The tests are thus self-checking. It is vital to make tests self-checking. If

you don't, you end up spending time hand checking some numbers from the test against some

numbers of a desk pad, and that slows you down.

* Before you start refactoring, check that you have a solid suite of tests. These tests

must be self-checking.

**Decomposing and Redistributing the Statement Method**

The obvious first target of my attention is the overly long statement method. When I look at a long

method like that, I am looking to decompose the method into smaller pieces. Smaller pieces of

code tend to make things more manageable. They are easier to work with and move around.

Any fool can write code that a computer can understand. Good programmers write

code that humans can understand.

Code that communicates its purpose is very important. I often refactor just when I'm reading

some code. That way as I gain understanding about the program, I embed that understanding

into the code for later so I don't forget what I learned.

Get rid of temporary variables such as this as much as possible. Temps are often a

problem in that they cause a lot of parameters to be passed around when they don't have to be.

You can easily lose track of what they are there for. They are particularly insidious in long

methods. Of course there is a performance price to pay; here the charge is now calculated twice.

But it is easy to optimize that in the rental class, and you can optimize much more effectively

when the code is properly factored.

**Removing Temps -** Temporary variables can be a problem. They are useful only within their

own routine, and thus they encourage long, complex routines.

By extracting the calculations I can create the htmlStatement method and reuse all of the

calculation code that was in the original statement method. I didn't copy and paste, so if the

calculation rules change I have only one place in the code to go to. Any other kind of statement

will be really quick and easy to prepare. The refactoring did not take long. I spent most of the time

figuring out what the code did, and I would have had to do that anyway.

Some code is copied from the ASCII version, mainly due to setting up the loop. Further

refactoring could clean that up. Extracting methods for header, footer, and detail line are one

route I could take. You can see how to do this in the example for Form Template Method. But

now the users are clamoring again. They are getting ready to change the classification of the

movies in the store. It's still not clear what changes they want to make, but it sounds like new

classifications will be introduced, and the existing ones could well be changed. The charges and

frequent renter point allocations for these classifications are to be decided. At the moment,

making these kind of changes is awkward. I have to get into the charge and frequent renter point

methods and alter the conditional code to make changes to film classifications. Back on with the

refactoring hat.

**Replacing the Conditional Logic on Price Code with Polymorphism**

The first part of this problem is that switch statement. It is a bad idea to do a switch based on an

attribute of another object. If you must use a switch statement, it should be on your own data, not

on someone else's.

**Chapter 2. Principles in Refactoring**

The preceding example should have given you a good feel for what refactoring is all about. Now

it's time to step back and look at the key principles of refactoring and at some of the issues you

need to think about in using refactoring.

**Defining Refactoring**

**Refactoring** (noun): a change made to the internal structure of software to make it

easier to understand and cheaper to modify without changing its observable behavior.

So you might spend a few hours refactoring, during which you might apply a couple of dozen

individual refactorings.

I've been asked, "Is refactoring just cleaning up code?" In a way the answer is yes, but I think

refactoring goes further because it provides a technique for cleaning up code in a more efficient

and controlled manner. Since I've been using refactoring, I've noticed that I clean code far more

effectively than I did before. This is because I know which refactorings to use, I know how to use

them in a manner that minimizes bugs, and I test at every possible opportunity.

I should amplify a couple of points in my definitions. First, the purpose of refactoring is to make

the software easier to understand and modify. You can make many changes in software that

make little or no change in the observable behavior. Only changes made to make the software

easier to understand are refactorings. A good contrast is performance optimization. Like

refactoring, performance optimization does not usually change the behavior of a component

(other than its speed); it only alters the internal structure. However, the purpose is different.

Performance optimization often makes code harder to understand, but you need to do it to get the

performance you need.

The second thing I want to highlight is that refactoring does not change the observable behavior

of the software. The software still carries out the same function that it did before. Any user,

whether an end user or another programmer, cannot tell that things have changed.

**The Two Hats**

This second point leads to Kent Beck's metaphor of two hats. When you use refactoring to

develop software, you divide your time between two distinct activities: adding function and

refactoring. When you add function, you shouldn't be changing existing code; you are just adding

new capabilities. You can measure your progress by adding tests and getting the tests to work.

When you refactor, you make a point of not adding function; you only restructure the code. You

don't add any tests (unless you find a case you missed earlier); you only restructure the code.

You don't add any tests (unless you find a case you missed earlier); you only change tests when

you absolutely need to in order to cope with a change in an interface.

As you develop software, you probably find yourself swapping hats frequently. You start by trying

to add a new function, and you realize this would be much easier if the code were structured

differently. So you swap hats and refactor for a while. Once the code is better structured, you

swap hats and add the new function. Once you get the new function working, you realize you

coded it in a way that's awkward to understand, so you swap hats again and refactor. All this

might take only ten minutes, but during this time you should always be aware of which hat you're

wearing.

**Why Should You Refactor?**

I don't want to proclaim refactoring as the cure for all 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. Refactoring is a

tool that can, and should, be used for several purposes.

**Refactoring Improves the Design of Software**

Without refactoring, the design of the program will decay. As people change code—changes to

realize short-term goals or changes made without a full comprehension of the design of the

code—the code loses its structure. It becomes harder to see the design by reading the code.

Refactoring is rather like tidying up the code. Work is done to remove bits that aren't really in the

right place. Loss of the structure of 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 code 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. Thus an important aspect of improving

design is to eliminate 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, because the effect on

the footprint of the programs rarely is significant. Reducing the amount of code does, however,

make a big difference in 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 and only once, which is the essence of good design.

**Refactoring Makes Software Easier to Understand**

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 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? It does matter if it takes a programmer a week to make a

change that would have taken only an hour if she had understood your code.

The trouble is that when you are trying to get the program to work, you are not thinking about that

future developer. It takes a change of rhythm to make 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. Programming in this mode is all about saying exactly what you

mean.

This understandability works another way, too. I use refactoring to help me understand unfamiliar code. When I look at unfamiliar code, I have to try to understand what it does. I look at a couple of lines and say to myself, oh yes, that's what this bit of code is doing. With refactoring I don't stop at the mental note. I actually change the code to better reflect my understanding, and then I test that understanding by rerunning the code to see if it still works.

Early on I do refactoring like this on little details. As the code gets clearer, I find I can see things about the design that I could not see before. Had I not changed the code, I probably never would have seen these things, because I'm just not clever enough to visualize all this in my head. Ralph Johnson describes these early refactorings as wiping the dirt off a window so you can see beyond. When I'm studying code I find refactoring leads me to higher levels of understanding that otherwise I would miss.

**Refactoring Helps you Find Bugs**

Help in understanding the code also helps me spot bugs. I admit I'm not terribly good at finding bugs. Some people can read a lump of code and see bugs, I cannot. However, I find that if I refactor code, I work deeply on understanding what the code does, and I put that new understanding right back into the code. By clarifying the structure of the program, I clarify certain assumptions I've made, to the point at which even I can't avoid spotting the bugs.

It reminds me of a statement Kent Beck often makes about himself, "I'm not a great programmer; I'm just a good programmer with great habits." Refactoring helps me be much more effective at writing robust code.

**Refactoring Helps you Program Faster**

In the end, all the earlier points come down to this: Refactoring helps you develop code more quickly.

This sounds counterintuitive. When I talk about refactoring, people can easily see that it improves quality. Improving design, improving readability, reducing bugs, all these improve quality. But doesn't all this reduce the speed of development?

I strongly believe that a good design is essential for rapid software development. Indeed, the whole point of having a good design is to allow rapid development. Without a good design, you can progress quickly for a while, but soon the poor design starts to slow you down. You spend time finding and fixing bugs instead of adding new function. Changes take longer as you try to understand the system and find the duplicate code. New features need more coding as you patch over a patch that patches a patch on the original code base.

A good design is essential to maintaining speed in software development. Refactoring helps you develop software more rapidly, because it stops the design of the system from decaying. It can even improve a design.

**When Should you Refactor?**

When I talk about refactoring, I'm often asked about how it should be scheduled. Should we allocate two weeks every couple of months to refactoring?

In almost all cases, I'm opposed to setting aside time for refactoring. In my view refactoring is not an activity you set aside time to do. Refactoring is something you do all the time in little bursts. You don't decide to refactor, you refactor because you want to do something else, and refactoring helps you do that other thing.

**The Rule of Three**

Here's a guideline Don Roberts gave me: The first time you do something, you just do it. The second time you do something similar, you wince at the duplication, but you do the duplicate thing anyway. The third time you do something similar, you refactor.

**Tip**

Three strikes and you refactor.

**Refactor When You Add Function**

The most common time to refactor is when I want to add a new feature to some software. Often the first reason to refactor here is to help me understand some code I need to modify. This code may have been written by someone else, or I may have written it. Whenever I have to think to 50 understand what the code is doing, I ask myself if I can refactor the code to make that understanding more immediately apparent. Then I refactor it. This is partly for the next time I pass by here, but mostly it's because I can understand more things if I clarify the code as I'm going along.

The other driver of refactoring here is a design that does not help me add a feature easily. I look at the design and say to myself, "If only I'd designed the code this way, adding this feature would be easy." In this case I don't fret over my past misdeeds—I fix them by refactoring. I do this partly to make future enhancements easy, but mostly I do it because I've found it's the fastest way. Refactoring is a quick and smooth process. Once I've refactored, adding the feature can go much more quickly and smoothly.

**Refactor When You Need to Fix a Bug**

In fixing bugs much of the use of refactoring comes from making code more understandable. As I look at the code trying to understand it, I refactor to help improve my understanding. Often I find that this active process of working with the code helps in finding the bug. One way to look at it is that if you do get a bug report, it's a sign you need refactoring, because the code was not clear enough for you to see there was a bug.

**Refactor As You Do a Code Review**

Some organizations do regular code reviews; those that don't would do better if they did. Code reviews help spread knowledge through a development team. Reviews help more experienced developers pass knowledge to less experienced people. They help more people understand more aspects of a large software system. They are also very important in writing clear code. My code may look clear to me but not to my team. That's inevitable—it's very hard for people to put themselves in the shoes of someone unfamiliar with the things they are working on. Reviews also give the opportunity for more people to suggest useful ideas. I can only think of so many good ideas in a week. Having other people contribute makes my life easier, so I always look for many reviews.

I've found that refactoring helps me review someone else's code. Before I started using refactoring, I could read the code, understand some degree of it, and make suggestions. Now when I come up with ideas, I consider whether they can be easily implemented then and there with refactoring. If so, I refactor. When I do it a few times, I can see more clearly what the code looks like with the suggestions in place. I don't have to imagine what it would be like, I can see what it is like. As a result, I can come up with a second level of ideas that I would never have realized had I not refactored.

Refactoring also helps the code review have more concrete results. Not only are there suggestions, but also many suggestions are implemented there and then. You end up with much more of a sense of accomplishment from the exercise.

To make this process work, you have to have small review groups. My experience suggests having one reviewer and the original author work on the code together. The reviewer suggests changes, and they both decide whether the changes can be easily refactored in. If so, they make the changes.

With larger design reviews it is often better to obtain several opinions in a larger group. Showing code often is not the best device for this. I prefer UML diagrams and walking through scenarios with CRC cards. So I do design reviews with groups and code reviews with individual reviewers.

**Problems with Refactoring**

When you learn a new technique that greatly improves your productivity, it is hard to see when it does not apply. Usually you learn it within a specific context, often just a single project. It is hard to see what causes the technique to be less effective, even harmful. Ten years ago it was like that with objects. If someone asked me when not to use objects, it was hard to answer. It wasn't that I didn't think objects had limitations—I'm too cynical for that. It was just that I didn't know what those limitations were, although I knew what the benefits were.

Refactoring is like that now. We know the benefits of refactoring. We know they can make a palpable difference to our work. But we haven't had broad enough experience to see where the limitations apply.

This section is shorter than I would like and is more tentative. As more people learn about refactoring, we will learn more. For you this means that while I certainly believe you should try refactoring for the real gains it can provide, you should also monitor its progress. Look out for problems that refactoring may be introducing. Let us know about these problems. As we learn more about refactoring, we will come up with more solutions to problems and learn about what problems are difficult to solve.

**Databases**

One problem area for refactoring is databases. Most business applications are tightly coupled to the database schema that supports them. That's one reason that the database is difficult to change. Another reason is data migration. Even if you have carefully layered your system to minimize the dependencies between the database schema and the object model, changing the database schema forces you to migrate the data, which can be a long and fraught task.

With nonobject databases a way to deal with this problem is to place a separate layer of software between your object model and your database model. That way you can isolate the changes to the two different models. As you update one model, you don't need to update the other. You just update the intermediate layer. Such a layer adds complexity but gives you a lot of flexibility. Even without refactoring it is very important in situations in which you have multiple databases or a complex database model that you don't have control over.

**When Shouldn't You Refactor?**

There are times when you should not refactor at all. The principle example is when you should rewrite from scratch instead. There are times when the existing code is such a mess that although you could refactor it, it would be easier to start from the beginning. This decision is not an easy one to make, and I admit that I don't really have good guidelines for it.

A clear sign of the need to rewrite is when the current code just does not work. You may discover this only by trying to test it and discovering that the code is so full of bugs that you cannot stabilize it. Remember, code has to work mostly correctly before you refactor.

A compromise route is to refactor a large piece of software into components with strong encapsulation. Then you can make a refactor-versus-rebuild decision for one component at a time. This is a promising approach, but I don't have enough data to write good rules for doing that. With a key legacy system, this would certainly be an appealing direction to take.

The other time you should avoid refactoring is when you are close to a deadline. At that point the productivity gain from refactoring would appear after the deadline and thus be too late. Ward Cunningham has a good way to think of this. He describes unfinished refactoring as going into debt. Most companies need some debt in order to function efficiently. However, with debt come interest payments, that is, the extra cost of maintenance and extension caused by overly complex code. You can bear some interest payments, but if the payments become too great, you will be overwhelmed. It is important to manage your debt, paying parts of it off by means of refactoring.

Other than when you are very close to a deadline, however, you should not put off refactoring because you haven't got time. Experience with several projects has shown that a bout of refactoring results in increased productivity. Not having enough time usually is a sign that you need to do some refactoring.

**Refactoring and Design**

Refactoring has a special role as a complement to design. When I first learned to program, I just wrote the program and muddled my way through it. In time I learned that thinking about the design in advance helped me avoid costly rework. In time I got more into this style of upfront design. Many people consider design to be the key piece and programming just mechanics. The analogy is design is an engineering drawing and code is the construction work. But software is very different from physical machines. It is much more malleable, and it is all about thinking. As Alistair Cockburn puts it, "With design I can think very fast, but my thinking is full of little holes."

One argument is that refactoring can be an alternative to upfront design. In this scenario you don't do any design at all. You just code the first approach that comes into your head, get it working, and then refactor it into shape. Actually, this approach can work. I've seen people do this and come out with a very well-designed piece of software. Those who support Extreme Programming [Beck, XP] often are portrayed as advocating this approach.

Although doing only refactoring does work, it is not the most efficient way to work. Even the extreme programmers do some design first. They will try out various ideas with CRC cards or the like until they have a plausible first solution. Only after generating a plausible first shot will they code and then refactor. The point is that refactoring changes the role of upfront design. If you don't refactor, there is a lot of pressure in getting that upfront design right. The sense is that any changes to the design later are going to be expensive. Thus you put more time and effort into the upfront design to avoid the need for such changes.

With refactoring the emphasis changes. You still do upfront design, but now you don't try to find the solution. Instead all you want is a reasonable solution. You know that as you build the solution, as you understand more about the problem, you realize that the best solution is different from the one you originally came up with. With refactoring this is not a problem, for it no longer is expensive to make the changes.

An important result of this change in emphasis is a greater movement toward simplicity of design. Before I used refactoring, I always looked for flexible solutions. With any requirement I would wonder how that requirement would change during the life of the system. Because design changes were expensive, I would look to build a design that would stand up to the changes I could foresee. The problem with building a flexible solution is that flexibility costs. Flexible solutions are more complex than simple ones. The resulting software is more difficult to maintain in general, although it is easier to flex in the direction I had in mind. Even there, however, you have to understand how to flex the design. For one or two aspects this is no big deal, but changes occur throughout the system. Building flexibility in all these places makes the overall system a lot more complex and expensive to maintain. The big frustration, of course, is that all this flexibility is not needed. Some of it is, but it's impossible to predict which pieces those are. To gain flexibility, you are forced to put in a lot more flexibility than you actually need.

With refactoring you approach the risks of change differently. You still think about potential changes, you still consider flexible solutions. But instead of implementing these flexible solutions, you ask yourself, "How difficult is it going to be to refactor a simple solution into the flexible solution?" If, as happens most of the time, the answer is "pretty easy," then you just implement the simple solution.

Refactoring can lead to simpler designs without sacrificing flexibility. This makes the design process easier and less stressful. Once you have a broad sense of things that refactor easily, you 58 don't even think of the flexible solutions. You have the confidence to refactor if the time comes. You build the simplest thing that can possibly work. As for the flexible, complex design, most of the time you aren't going to need it.

Even if you know exactly what is going on in your system, measure performance, don't speculate. You'll learn something, and nine times out of ten, it won't be that you were right!