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.