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.