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.

Refactoring is the opposite of this practice. With refactoring you can take a bad design, chaos even, and rework it into well-designed code.

The problem with copying and pasting code comes when you have to change it later.

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.

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

As we do the refactoring, we will lean on the tests. I'm going to be relying on the tests to tell me whether I introduce a bug.

My first step is to find a logical clump of code and use Extract Method. An obvious piece here is the switch statement. This looks like it would make a good chunk to extract into its own method.

First I need to look in the fragment for any variables that are local in scope to the method we are looking at, the local variables and parameters. This segment of code uses two: each and thisAmount. Of these each is not modified by the code but thisAmount is modified. Any nonmodified variable I can pass in as a parameter. Modified variables need more care. If there is only one, I can return it. The temp is initialized to 0 each time around the loop and is not altered until the switch gets to it. So I can just assign the result.

As I look at amountFor, I can see that it uses information from the rental, but does not use information from the customer.

This immediately raises my suspicions that the method is on the wrong object. In most cases a method should be on the object whose data it uses, thus the method should be moved to the rental.

I like to 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.

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.

getMovie().getPriceCode() -> getPriceCode()

We have several types of movie that have different ways of answering the same question. This sounds like a job for subclasses.

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

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);

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.

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.

Number one in the stink parade is duplicated code. If you see the same code structure in more than one place, you can be sure that your program will be better if you find a way to unify them.

Another common duplication problem is when you have the same expression in two sibling´subclasses. You can eliminate this duplication by using Extract Method in both classes then Pull Up Field. If the code is similar but not the same, you need to use Extract Method to separate the similar bits from the different bits. You may then find you can use Form Template 64 Method. If the methods do the same thing with a different algorithm, you can choose the clearerof the two algorithms and use Substitute Algorithm.

If you have duplicated code in two unrelated classes, consider using Extract Class in one class and then use the new component in the other. Another possibility is that the method really belongs only in one of the classes and should be invoked by the other class or that the method belongs in a third class that should be referred to by both of the original classes. You have to decide where the method makes sense and ensure it is there and nowhere else.

whenever we feel the need to comment something, we write a method instead. Such a method contains the code that was commented but is named after the intention of the code rather than how it does it. provided the method name explains the purpose of the code.

Ninety-nine percent of the time, all you have to do to shorten a method is Extract Method. Find parts of the method that seem to go nicely together and make a new method.

If you have a method with lots of parameters and temporary variables, these elements get in the way of extracting methods. If you try to use Extract Method, you end up passing so many of the parameters and temporary variables as parameters to the extracted method that the result is scarcely more readable than the original. You can often use Replace Temp with Query to eliminate the temps. Long lists of parameters can be slimmed down with Introduce Parameter Object and Preserve Whole Object. If you've tried that, and you still have too many temps and parameters, it's time to get out the heavy artillery: Replace Method with Method Object.

Conditionals and loops also give signs for extractions. Use Decompose Conditional to deal with conditional expressions. With loops, extract the loop and the code within the loop into its own method.

When a class is trying to do too much, it often shows up as too many instance variables. When a class has too many instance variables, duplicated code cannot be far behind. You can Extract Class to bundle a number of the variables. Choose variables to go together in the component that makes sense for each. For example, "depositAmount" and "depositCurrency" are likely to belong together in a component. More generally, common prefixes or suffixes for some subset of the variables in a class suggest the opportunity for a component. If the component makes sense as a subclass, you'll find Extract Subclass often is easier. Sometimes a class does not use all of its instance variables all of the time. If so, you may be able to Extract Class or Extract Subclass many times.

As with a class with a huge wad of variables, the usual solution for a class with too much code is either to Extract Class or Extract Subclass. A useful trick is to determine how clients use the class and to use Extract Interface for each of these uses

Use Replace Parameter with Method when you can get the data in one parameter by making a request of an object you already know about, this object might be a field or might be another parameter.

when every time you make a kind of change, you have to make a lot of little changes to a lot of different classes. When the changes are all over the place, they are hard to find, and it's easy to miss an important change. In this case you want to use Move Method and Move Field to put all the changes into a single class. If no current class looks like a good candidate, create one.

The class should have only one reason to change

A classic smell is a method that seems more interested in a class other than the one it actually is in.

Most times you see a switch statement you should consider polymorphism. The issue is where the polymorphism should occur. Often the switch statement switches on a type code. You want the method or class that hosts the type code value. So use Extract Method to extract the switch statement and then Move Method to get it onto the class where the polymorphism is needed

If you only have a few cases that affect a single method, and you don't expect them to change, then polymorphism is overkill. In this case Replace Parameter with Explicit Methods is a good option. If one of your conditional cases is a null, try Introduce Null Object.

If the classes do have common interests, use Extract Class to put the commonality in a safe place and make honest classes of them.

I like Replace Temp with Queryto get rid of any temporary variables that I can remove. If the temp is used for many things, I use Split Temporary Variable first to make the temp easier to replace.

Sometimes, however, the temporary variables are just too tangled to replace. I need Replace Method with Method Object. This allows me to break up even the most tangled method, at the cost of introducing a new class for the job.

Local variables are only in scope in that method, so when I use Extract Method, these variables cause me extra work. In some cases they even prevent me from doing the refactoring at all.

The easiest case with local variables is when the variables are read but not changed. In this case I can just pass them in as a parameter.

Replace Method with Method Object You have a long method that uses local variables in such a way that you cannot apply Extract Method.

Turn the method into its own object so that all the local variables become fields on that object. You can then decompose the method into other methods on the same object.

By extracting pieces out of a large method,you make things much more comprehensible. The difficulty in decomposing a method lies in local variables. If they are rampant, decomposition can be difficult. Using Replace Temp with Query helps to reduce this burden, but occasionally you may find you cannot break down a method that needs breaking. In this case you reach deep into the tool bag and get out your method object [Beck].

Applying Replace Method with Method Object turns all these local variables into fields on the´method object. You can then use Extract Method on this new object to create additional methods that break down the original method.

1)

Class Account

int gamma (int inputVal, int quantity, int yearToDate) {

int importantValue1 = (inputVal \* quantity) + delta();

int importantValue2 = (inputVal \* yearToDate) + 100;

if ((yearToDate - importantValue1) > 100)

importantValue2 -= 20;

int importantValue3 = importantValue2 \* 7;

// and so on.

return importantValue3 - 2 \* importantValue1;

}

2)

class Gamma...

private final Account \_account;

private int inputVal;

private int quantity;

private int yearToDate;

private int importantValue1;

private int importantValue2;

private int importantValue3;

3)

Gamma (Account source, int inputValArg, int quantityArg, int

yearToDateArg) {

\_account = source;

inputVal = inputValArg;

quantity = quantityArg;

yearToDate = yearToDateArg;

}

4)

int compute () {

importantValue1 = (inputVal \* quantity) + \_**account.delta();**

importantValue2 = (inputVal \* yearToDate) + 100;

if ((yearToDate - importantValue1) > 100)

importantValue2 -= 20;

int importantValue3 = importantValue2 \* 7;

// and so on.

return importantValue3 - 2 \* importantValue1;

}

5)

int gamma (int inputVal, int quantity, int yearToDate) {

return new Gamma(this, inputVal, quantity,

yearToDate).compute();

}

6)

int compute () {

importantValue1 = (inputVal \* quantity) + \_account.delta();

importantValue2 = (inputVal \* yearToDate) + 100;

importantThing();

int importantValue3 = importantValue2 \* 7;

// and so on.

return importantValue3 - 2 \* importantValue1;

}

void importantThing() {

if ((yearToDate - importantValue1) > 100)

importantValue2 -= 20;

}

I usually look through the methods on a class to find a method that seems to reference another object more than the object it lives on.

I consider moving a field if I see more methods on another class using the field than the class itself.

Move field to another class and on the class it was call the object of that new class which returns it

Encapsulation means that objects need to know less about other parts of the system. Then when things change, fewer objects need to be told about the change

Hide Delegate (only one on more it is better to remove the middle men)

1)

manager = john.getDepartment().getManager();

2)

public Person getManager() {

return \_department.getManager();

}

3)

manager = john.getManager();

remove middle men: manager = john.getDepartment()

Foreign method

A server class you are using needs an additional method, but you can't modify the class

Date newStart = nextDay(previousEnd);

private static Date nextDay(Date arg) {

// foreign method, should be on date

return new Date (arg.getYear(),arg.getMonth(), arg.getDate() +

1);

}

Introduce Local Extension

A server class you are using needs several additional methods, but you can't modify the class. Create a new class that contains these extra methods. Make this extension class a subclass or a wrapper of the original.

A local extension is a separate class, but it is a subtype of the class it is extending. That means it supports all the things the original can do but also adds the extra features. Instead of using the original class, you instantiate the local extension and use it.

class MfDate...

Date nextDay() {

return new Date (getYear(),getMonth(), getDate() + 1);

}

Replace Array with object

Arrays are a common structure for organizing data. However, they should be used only to contain a collection of similar objects in some order

Remove Control Flag

You have a variable that is acting as a control flag for a series of boolean expressions.

**Use a break or return instead.**

void checkSecurity(String[] people) {

boolean **found** = false;

for (int i = 0; i < people.length; i++) {

if (! **found**) {

if (people[i].equals ("Don")){

sendAlert();

**found** = true;

}

if (people[i].equals ("John")){

sendAlert();

found = true;

}

}

}

}

To

void checkSecurity(String[] people) {

for (int i = 0; i < people.length; i++) {

if (people[i].equals ("Don")){

sendAlert();

break;

}

if (people[i].equals ("John")){

sendAlert();

break;

}

}

}

double getPayAmount() {

double result;

if (\_isDead) result = deadAmount();

else {

to

double getPayAmount() {

if (\_isDead) return deadAmount();

Introduce Null Object

You have repeated checks for a null value.

Replace the null value with a null object.

The essence of polymorphism is that instead of asking an object what type it is and then invoking some behavior based on the answer, you just invoke the behavior. The object, depending on its type, does the right thing. One of the less intuitive places to do this is where you have a null value in a field.

Separate Query from Modifier

You have a method that returns a value but also changes the state of an object.

Create two methods, one for the query and one for the modification.

