Chapter 3: Bad Smells in Code

"If it stinks, change it."
— Grandma Beck, discussing child raising philosophy

By now you have a good idea of how refactoring works. But just because you know how doesn't mean you know when. Deciding when to start refactoring (and when to stop) is just as important to refactoring as knowing how to operate the mechanics of a refactoring.

Now comes the dilemma. It is easy to explain to you how to delete an instance variable or create a hierarchy. These are simple matters. Trying to explain when you should do these things is not so cut-and-dried. Rather than appealing to some vague notion of programming aesthetics (which frankly is what we consultants usually do) I wanted something a bit more solid.

I was mulling over this tricky issue when I visited Kent in Zurich. Perhaps he was under the influence of the odors of his new born daughter at the time, but he had come up with the notion describing the "when" of refactoring in terms of smells. "Smells," you say, "and that is supposed to be better than vague aesthetics?" Well, yes. We look at lots of code, written by projects that span the gamut from wildly successful to nearly dead. And in doing so, we have learned to look for certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring. (We are switching over to "we" in this chapter to reflect the fact that Kent and I wrote this chapter jointly. You can tell the difference because the funny jokes are mine and the others are his.)

One thing we won't try to do here is give you precise criteria for when a refactoring is overdue. In our experience there is no set of metrics that rival informed human intuition. What we will do is give you indications that there is trouble that could be solved by a refactoring. You will have to develop your own sense of how many instance variables is too many instance variables and how many lines of code in a method is too many lines.

Duplicated code

Number one on 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.

The simplest duplicated code problem is when you have the same expression in two methods of the same class. Then all you have to do is *Extract Method (114)* and invoke the code from both places.

Another common duplication problem is when you have the same expression in two sibling subclasses. You can eliminate this duplication by using *Extract Method (114)* in both classes, then *Pull Up Method (271)*. If the code is similar but not the same, then you need to use *Extract Method (114)* to separate the similar bits from the different bits. You may then find you can use *Form Template Method (289)*. If the methods do the same thing using a different algorithm, you can choose the clearer of the two algorithms and use *Substitute Algorithm (132)*.

If you have duplicated code in two unrelated classes, consider using *Extract Component (166)* 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.

Long method

The object programs that live best and longest are those with short methods. Programmers new to objects often get the feeling that no computation ever takes place, that object programs are endless sequences of delegation. When you have lived with a program for a few years, however, you learn just how valuable all those little methods are. All of the pay-offs of indirection- explanation, sharing, and choosing- are supported by little methods.

Since the early days of programming people have realized that the longer a procedure is, the more difficult it is to understand. Older languages carried an overhead in subroutine calls, which deterred people from small method, modern OO languages have pretty much eliminated that overhead. There is still an overhead to the reader of the code as you have to switch context to look and see what the sub-procedure does. Development environments that allow you to see two methods at once help to eliminate this; but the real key to making it easy to understand small methods is good naming. If you have a good name for a method you don't need to look at the body.

The net effect of this is that you should be much more aggressive about decomposing methods. A heuristic we follow is whenever we feel the need to comment something, we instead write a method. Such a method contains the code that was commented, but is named after what the intention of the code is, rather than how it does it. We may do this on a group of lines, or on as small as a single line of code. We will do this even if the method call is longer than the code it replaces, providing the method name explains the purpose of the code. The key here is not the method length, but the semantic distance between what the method does and how it does it.

99% of the time, all you have to shorten a method is *Extract Method (114)*. Find a part of the method that seems to go nicely together and make a new method.

If you have a method with lots of parameters and temporary variables, they 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 *Inline Temp (121)* to get rid of the temps. Long lists of parameters can be slimmed down with *Introduce Parameter Object (247)* and *Preserve Whole Object (241)*.

If you've tried that, and you still have too many temps and parameters, then it's time to get out the heavy artillery: *Replace Method with Method Object (130)*.

How do you identify the clumps of code to extract? A good technique is to look for comments. They often signal this kind of semantic distance. A block of code with a comment that tells you what it is doing can be replaced by a method whose name is based on the comment. Even a single line is worth extracting if it needs explanation.

Conditionals and loops also give signs for extractions. Use *Decompose Conditional (136)* to deal with conditional expressions. With loops extract the loop and the code within the loop into its own method.

Large Class

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 Component (166)* to bundle up a number of the variables. Choose variables to go together in the component that makes sense with each other. 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 (277)* often is easier. However you can only do this if the subset of instance variables that are used do not change during an object’s lifetime.

Sometimes a class will not be using all of its instance variables all of the time. If so, you may be able to *Extract Component (166)* or *Extract Subclass (277)* many times.

As with a class with too many instance variables, a class with too much code is prime breeding ground for duplicated code, chaos, and death.

The simplest solution (have we mentioned that we like simple solutions?) is if you can eliminate redundancy in the class itself. If you have five hundred line methods with lots of code in common, you may be able to turn them into five ten line methods with another ten two line methods extracted from the original.

As with a class with a huge wad of variables, the usual solution for a class with too much code is either to *Extract Component (166)* or *Extract Subclass (277)*.

Long Parameter List

In our early programming days we were taught to pass everything a routine needed in as parameters. This was understandable because the alternative was global data, and global data is evil and usually painful. Objects change this situation because if you don't have something you need, you can always ask another object to get it for you. Thus with objects you don't pass in everything the method needs, instead you pass enough so that the method can get to everything it needs. A lot of what a method needs is available on method's host class. Thus in object-oriented programs parameter lists tend to be much smaller than on traditional programs.

This is good because long parameter lists are hard to understand, they get inconsistent and difficult to use, and because you are forever changing them as you need more data. Most changes are removed by passing objects because you are much more likely to just need to make a couple of requests to get at a new piece of data.

Use *Replace Parameter with Method (245)* 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 it might be another parameter. Use *Preserve Whole Object (241)* to take a bunch of data gleaned from an object and replace it with the object itself. If you have several data items with no logical object, then *Introduce Parameter Object (247)*.

There is one important exception to doing these changes. This is when you explicitly do not wish to create a dependency from the called object to the larger object. In those cases unpacking data and sending it along as parameters is reasonable, but pay attention to the pain involved. If the parameter list is too long or changes too often you will need to rethink your dependency structure.

Divergent Change

We structure our software to make change easier, after all software is meant to be soft. So when we make a change we want to be able to jump to a single clear point in the system and make the change.

If you look at a class and say, “Well, I will have to change these three methods every time I get a new database. I have to change these four methods every time there is a new financial instrument.” you likely have a situation where two objects are better than one. Of course you often discover this only after you've added a few databases or financial instruments. Any change to handle a variation should change a single class and all the typing in the new class should be expressing the variation. So for this you identify those things you are changing and use *Extract Component (166)* to put them all together.

Shotgun Surgery

You whiff this when every time you add some variation, 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 (160)* and *Move Field (164)* to get all the changes into a single class. If no current class looks like a good candidate, then create one. Often you can use *Inline Component (170)* to bring a whole bunch of behavior together. You then get a small dose of Divergent Change, but you can easily deal with that.

Feature Envy

The whole point of objects is that they are a packaging technique that packages data together with the processes upon that data. A classic smell is thus a method that seems more interested in another class than the one it's actually in. The most common focus of the envy is the data. We've lost count of the times we've seen a method that invokes half-a-dozen getting methods on another object in order to calculate some value. Fortunately the cure is obvious, the method clearly wants to be elsewhere so you use *Move Method (160)* to get it there. Sometimes only part of the method suffers from envy, in that case use *Extract Method (114)* on the jealous bit and then *Move Method (160)* to give it a dream home.

Of course not all cases are so cut and dried. Often a method uses features of several classes, so which one should it live with? The heuristic we use is to which class has most of the data and put the method with that data. This is often made easier by using *Extract Method (114)* to break into methods that go into different places.

Of course there several sophisticated patterns that break this rule. From the [Gang of Four] Strategy and Visitor immediately leap to mind. Kent's Self Delegation is another. You do these to combat the Divergent Change smell. The fundamental rule of thumb is to put things together that change together. Data and the behavior that references that data usually change together, but there are exceptions. When those exceptions occur we move the behavior to keep changes in one place. Strategy and Visitor allow you to change behavior easily because it isolates the small amount of behavior that needs to be overridden, at the cost of further indirection.

Data Clumps

Data items tend to be like children — they enjoy hanging around in groups together. Often you'll see the same three or four data items together in lots of places: fields in a couple of classes, parameters in many method signatures. Bunches of data that hang around together really ought to be made into their own object. The first step is to look for where they appear as fields. Use *Extract Component (166)* on the fields to turn them into an object. Then turn your attention to method signatures, using *Introduce Parameter Object (247)* or *Preserve Whole Object (241)* to slim them down. The immediate benefit here is that you can shrink a lot of parameter lists and make method calling a lot simpler. Don't worry about data clumps that only use some of the fields of the new object. As long as you are replacing a couple or more fields with the new object, you'll come out ahead.

A good test is to consider deleting one of the data values: if you did this would the others make any sense? If they don't that's a sure sign that you have an object that's dying to be born.

Reducing field lists and parameter lists will certainly remove a few bad smells, but once you have the objects you get the opportunity to make a nice perfume. You can now look for cases of Divergent Change which will suggest behavior that can be moved into your new classes. Before long these classes will be productive members of society.

Case Statements

One of the most obvious symptoms of object-oriented code is its lack of case (or switch) statements. The problem with case statements is essentially that of duplication. Often you find the same case statement scattered about a program for different purposes. If you add a type to the case you have to find all these case statements and change them. The object-oriented notion of polymorphism gives you an elegant way to deal with this problem.

So most times you see a case statement you should think about polymorphism, the issue is where should the polymorphism occur. Usually case statement will switch on a type code. You will want the method on class that hosts the type code value. So use *Extract Method (114)* to extract the switch statement and then *Move Method (160)* to get it onto the class where the polymorphism is needed. At that point you have to decide whether to *Replace Type Code with Subclasses (224)* or *Replace Type Code with State/Strategy (227)*. When you have set up the inheritance structure you can then *Replace Switch with Polymorphism (147)*.

Parallel Inheritance Hierarchies

This smell is really a special case of “the same rate off change in two objects”. In this case, every time you make a subclass of one class you also have to make a subclass of another. You can recognize this smell because the prefixes of the class names in one hierarchy are the same as the prefixes in another hierarchy.

The general strategy for eliminating the duplication is to make sure that instances of one hierarchy refer to instances of the other, then use *Move Method (160)* and *Move Field (164)* the hierarchy disappears on the referring class.

Lazy Class

Each class you create costs money. If there is a class that isn't doing enough to pay for itself, it should be eliminated.

If you have subclasses that aren't doing enough, try to use *Collapse Hierarchy (288)*. Nearly useless components should be subjected to *Inline Component (170)*.

Similar subclasses

Sometimes you will see a class with four subclasses, each of which only implements three simple methods. Often you will get a vague feeling that the class doesn't deserve subclasses, but you won't immediately be able to see how to eliminate them. This feeling can last for months or even years. Don't worry. If you keep nibbling away at the problems you can see how to solve, eventually you will find yourself looking at the subclasses again, and all the difficult issues to resolve will have disappeared.

Once you've done this, look for new opportunities to use inheritance now that you are no longer wasting it.

Temporary Field

Sometimes you will see an object where an instance variable is only set in certain circumstances. Such code is difficult to understand, because you expect an object to need all of its variables. Trying to understand why a variable is there when it doesn't seem to be used can drive you nuts.

Use *Extract Component (166)* to create a home for the poor orphan variables. Put all the code that concerns the variables in the component. You may also be able to eliminate conditional code by using *Introduce Null Object (151)* to create an alternative component for when the variables aren't valid.

Middle Man

One of the prime features of objects is encapsulation: hiding internal details from the rest of the world. Often this encapsulation comes with delegation, you ask a director if she is free a meeting, she delegates the message to her diary, and gives you an answer. All well and good, there is no need to know whether the director uses a diary, an electronic gizmo, or a secretary to keep track of her appointments.

However this can go too far (particularly with those who take the Law of Demeter too seriously). You look at a class's interface and find half the methods are delegating to this other class. After a while it's time to cut out the middle man and talk to the object that really knows what's going on. Use *Move Method (160)* and *Move Field (164)* to move features out of the middle man into the other objects until the middle man has nothing left.

Alternative classes with different interfaces

Use *Rename Method (234)* on any methods that do the same thing but have different signatures for what they do. Often this doesn't go far enough. In these cases the classes aren't doing enough yet. Keep using

Move Method (160) to move behavior to them until the protocols are the same. If you have to redundantly move code to accomplish this, you may be able to use *Extract Superclass (282)* to atone.

Comments

Don't worry, we aren't saying that people shouldn't write comments. In our olfactory analogy, comments aren't a bad smell, indeed they are a sweet smell.

The reason we mention them here is because comments are often used as a deodorant. It's surprising how often you look at thickly commented code, and notice that the comments are there because the code is bad.

Thus comments lead us to bad code that has all the rotten whiffs we've discussed in the rest of this chapter. Our first action to remove the bad smells by refactoring. When we're done we often find that the comments are now superfluous.

When you feel the need to write a comment, try first to refactor the code so that any comment would be superfluous.