Chapter 3

Bad Smells in Code

by Kent Beck and Martin Fowler

"If it stinks, change it."
— Grandma Beck, discussing child-rearing 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 it.

Now comes the dilemma. It is easy to explain 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. Instead of appealing to some vague notion of programming aesthetics (which, frankly, is what we consultants usually do), I wanted something a bit more solid.

When I was writing the first edition of this book, I was mulling over this issue as I visited Kent Beck in Zurich. Perhaps he was under the influence of the odors of his newborn daughter at the time, but he had come up with the notion of 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 have looked at lots of code, written for projects that span the gamut from wildly successful to nearly dead. In doing so, we have learned to look for certain structures in the code that suggest—sometimes, 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 give you is precise criteria for when a refactoring is overdue. In our experience, no set of metrics rivals informed human intuition. What we will do is give you indications that there is trouble that can be solved by a refactoring. You will have to develop your own sense of how many instance variables or how many lines of code in a method are too many.

Use this chapter and the table on the inside back cover as a way to give you inspiration when you're not sure what refactorings to do. Read the chapter (or skim the table) and try to identify what it is you're smelling, then go to the refactorings we suggest to see whether they will help you. You may not find the exact smell you can detect, but hopefully it should point you in the right direction.

Mysterious Name

Puzzling over some text to understand what's going on is a great thing if you're reading a detective novel, but not when you're reading code. We may fantasize about being International Men of Mystery, but our code needs to be mundane and clear. One of the most important parts of clear code is good names, so we put a lot of thought into naming functions, modules, variables, classes, so they clearly communicate what they do and how to use them.

Sadly, however, naming is one of the two hard things [mf-2h] in programming. So, perhaps the most common refactorings we do are the renames: *Change Function Declaration (124)* (to rename a function), *Rename Variable (137)*, and *Rename Field (244)*. People are often afraid to rename things, thinking it's not worth the trouble, but a good name can save hours of puzzled incomprehension in the future.

Renaming is not just an exercise in changing names. When you can't think of a good name for something, it's often a sign of a deeper design malaise. Puzzling over a tricky name has often led us to significant simplifications to our code.

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. Duplication means that every time you read these copies, you need to read them carefully to see if there's any difference. If you need to change the duplicated code, you have to find and catch each duplication.

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 Function (106)* and invoke the code from both places. If you have code that's similar, but not quite identical, see if you can use *Slide Statements (223)* to arrange the code so the similar items are all together for easy extraction. If the duplicate fragments are in subclasses of a common base class, you can use *Pull Up Method (350)* to avoid calling one from another.

Long Function

In our experience, the programs that live best and longest are those with short functions. Programmers new to such a code base often feel that no computation ever takes place—that the program is an endless sequence of delegation. When you have lived with such a program for a few years, however, you learn just how valuable all those little functions are. All of the payoffs of indirection—explanation, sharing, and choosing—are supported by small functions.

Since the early days of programming, people have realized that the longer a function is, the more difficult it is to understand. Older languages carried an overhead in subroutine calls, which deterred people from small functions. Modern languages have pretty much eliminated that overhead for in-process calls. There is still overhead for the reader of the code because you have to switch context to see what the function does. Development environments that allow you to quickly jump between a function call and its declaration, or to see both functions at once, help eliminate this step, but the real key to making it easy to understand small functions is good naming. If you have a good name for a function, you mostly don't need to look at its body.

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

Ninety-nine percent of the time, all you have to do to shorten a function is *Extract Function (106)*. Find parts of the function that seem to go nicely together and make a new one.

If you have a function with lots of parameters and temporary variables, they get in the way of extracting. If you try to use *Extract Function (106)*, you end up passing so many parameters to the extracted method that the result is scarcely more readable than the original. You can often use *Replace Temp with Query (178)* to eliminate the temps. Long lists of parameters can be slimmed down with *Introduce Parameter Object (140)* and *Preserve Whole Object (319)*.

If you've tried that and you still have too many temps and parameters, it's time to get out the heavy artillery: *Replace Function with Command (337)*.

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 (260)* to deal with conditional expressions. A big switch statement should have its legs turned into single function calls with *Extract Function (106)*. If there's more than one switch statement switching on the same condition, you should apply *Replace Conditional with Polymorphism (272)*.

With loops, extract the loop and the code within the loop into its own method. If you find it hard to give an extracted loop a name, that may be because it's doing two different things—in which case don't be afraid to use *Split Loop (227)* to break out the separate tasks.

Long Parameter List

In our early programming days, we were taught to pass in as parameters everything needed by a function. This was understandable because the alternative was global data, and global data quickly becomes evil. But long parameter lists are often confusing in their own right.

If you can obtain one parameter by asking another parameter for it, you can use *Replace Parameter with Query (324)* to remove the second parameter. Rather than pulling lots of data out of an existing data structure, you can use *Preserve Whole Object (319)* to pass the original data structure instead. If several parameters always fit together, combine them with *Introduce Parameter Object (140)*. If a parameter is used as a flag to dispatch different behavior, use *Remove Flag Argument (314)*.

Classes are a great way to reduce parameter list sizes. They are particularly useful when multiple functions share several parameter values. Then, you can use *Combine Functions into Class (144)* to capture those common values as fields. If we put on our functional programming hats, we'd say this creates a set of partially applied functions.

Global Data

Since our earliest days of writing software, we were warned of the perils of global data—how it was invented by demons from the fourth plane of hell, which is the resting place of any programmer who dares to use it. And, although we are somewhat skeptical about fire and brimstone, it's still one of the most pungent odors we are likely to run into. The problem with global data is that it can be modified from anywhere in the code base, and there's no mechanism to discover which bit of code touched it. Time and again, this leads to bugs that breed from a form of spooky action from a distance—and it's very hard to find out where the errant bit of program is. The most obvious form of global data is global variables, but we also see this problem with class variables and singletons.

Our key defense here is *Encapsulate Variable (132)*, which is always our first move when confronted with data that is open to contamination by any part of a program. At least when you have it wrapped by a function, you can start seeing where it's modified and start to control its access. Then, it's good to limit its scope as much as possible by moving it within a class or module where only that module's code can see it.

Global data is especially nasty when it's mutable. Global data that you can guarantee never changes after the program starts is relatively safe—if you have a language that can enforce that guarantee.

Global data illustrates Paracelsus's maxim: The difference between a poison and something benign is the dose. You can get away with small doses of global data, but it gets exponentially harder to deal with the more you have. Even with little bits, we like to keep it encapsulated—that's the key to coping with changes as the software evolves.

Mutable Data

Changes to data can often lead to unexpected consequences and tricky bugs. I can update some data here, not realizing that another part of the software expects something different and now fails—a failure that's particularly hard to spot if it only happens under rare conditions. For this reason, an entire school of software development—functional programming—is based on the notion that data should never change and that updating a data structure should always return a new copy of the structure with the change, leaving the old data pristine.

These kinds of languages, however, are still a relatively small part of programming; many of us work in languages that allow variables to vary. But this doesn't mean we should ignore the advantages of immutability—there are still many things we can do to limit the risks on unrestricted data updates.

You can use *Encapsulate Variable (132)* to ensure that all updates occur through narrow functions that can be easier to monitor and evolve. If a variable is being updated to store different things, use *Split Variable (240)* both to keep them separate and avoid the risky update. Try as much as possible to move logic out of code that processes the update by using *Slide Statements (223)* and *Extract Function (106)* to separate the side-effect-free code from anything that performs the update. In APIs, use *Separate Query from Modifier (306)* to ensure callers don't need to call code that has side effects unless they really need to. We like to use *Remove Setting Method (331)* as soon as we can—sometimes, just trying to find clients of a setter helps spot opportunities to reduce the scope of a variable.

Mutable data that can be calculated elsewhere is particularly pungent. It's not just a rich source of confusion, bugs, and missed dinners at home—it's also unnecessary. We spray it with a concentrated solution of vinegar and *Replace Derived Variable with Query (248)*.

Mutable data isn't a big problem when it's a variable whose scope is just a couple of lines—but its risk increases as its scope grows. Use *Combine Functions into Class (144)* or *Combine Functions into Transform (149)* to limit how much code needs to update a variable. If a variable contains some data with internal structure, it's usually better to replace the entire structure rather than modify it in place, using *Change Reference to Value (252)*.

Divergent Change

We structure our software to make change easier; after all, software is meant to be soft. When we make a change, we want to be able to jump to a single clear point in the system and make the change. When you can't do this, you are smelling one of two closely related pungencies.

Divergent change occurs when one module is often changed in different ways for different reasons. If you look at a module and say, "Well, I will have to change these three functions every time I get a new database; I have to change these four functions every time there is a new financial instrument," this is an indication of divergent change. The database interaction and financial processing problems are separate contexts, and we can make our programming life better by moving such contexts into separate modules. That way, when we have a change to one context, we only have to understand that one context and ignore the other. We always found this to be important, but now, with our brains shrinking with age, it becomes all the more imperative. Of course, you often discover this only after you've added a few databases or financial instruments; context boundaries are usually unclear in the early days of a program and continue to shift as a software system's capabilities change.

If the two aspects naturally form a sequence—for example, you get data from the database and then apply your financial processing on it—then *Split Phase (154)* separates the two with a clear data structure between them. If there's more back-and-forth in the calls, then create appropriate modules and use *Move Function (198)* to divide the processing up. If functions mix the two types of processing within themselves, use *Extract Function (106)* to separate them before moving. If the modules are classes, then *Extract Class (182)* helps formalize how to do the split.

Shotgun Surgery

Shotgun surgery is similar to divergent change but is the opposite. You whiff this when, every time you make a change, you have to make a lot of little edits 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 Function (198)* and *Move Field (207)* to put all the changes into a single module. If you have a bunch of functions operating on similar data, use *Combine Functions into Class (144)*. If you have functions that are transforming or enriching a data structure, use *Combine Functions into Transform (149)*. *Split Phase (154)* is often useful here if the common functions can combine their output for a consuming phase of logic.

A useful tactic for shotgun surgery is to use inlining refactorings, such as *Inline Function (115)* or *Inline Class (186)*, to pull together poorly separated logic. You'll end up with a Long Method or a Large Class, but can then use extractions to break it up into more sensible pieces. Even though we are inordinately fond of small functions and classes in our code, we aren't afraid of creating something large as an intermediate step to reorganization.

Feature Envy

When we modularize a program, we are trying to separate the code into zones to maximize the interaction inside a zone and minimize interaction between zones. A classic case of Feature Envy occurs when a function in one module spends more time communicating with functions or data inside another module than it does within its own module. We've lost count of the times we've seen a function invoking half-a-dozen getter methods on another object to calculate some value. Fortunately, the cure for that case is obvious: The function clearly wants to be with the data, so use *Move Function (198)* to get it there. Sometimes, only a part of a function suffers from envy, in which case use *Extract Function (106)* on the jealous bit, and *Move Function (198)* to give it a dream home.

Of course not all cases are cut-and-dried. Often, a function uses features of several modules, so which one should it live with? The heuristic we use is to determine which module has most of the data and put the function with that data. This step is often made easier if you use *Extract Function (106)* to break the function into pieces that go into different places.

Of course, there are several sophisticated patterns that break this rule. From the Gang of Four [gof], Strategy and Visitor immediately leap to mind. Kent Beck's Self Delegation [Beck SBPP] is another. Use 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 the exceptions occur, we move the behavior to keep changes in one place. Strategy and Visitor allow you to change behavior easily because they isolate 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 together. Often, you'll see the same three or four data items together in lots of places: as fields in a couple of classes, as parameters in many method signatures. Bunches of data that hang around together really ought to find a home together. The first step is to look for where the clumps appear as fields. Use *Extract Class (182)* on the fields to turn the clumps into an object. Then turn your attention to method signatures using *Introduce Parameter Object (140)* or *Preserve Whole Object (319)* to slim them down. The immediate benefit is that you can shrink a lot of parameter lists and simplify method calling. Don't worry about data clumps that use only some of the fields of the new object. As long as you are replacing two 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, it's a sure sign that you have an object that's dying to be born.

You'll notice that we advocate creating a class here, not a simple record structure. We do this because using a class gives you the opportunity to make a nice perfume. You can now look for cases of feature envy, which will suggest behavior that can be moved into your new classes. We've often seen this as a powerful dynamic that creates useful classes and can remove a lot of duplication and accelerate future development, allowing the data to become productive members of society.

Primitive Obsession

Most programming environments are built on a widely used set of primitive types: integers, floating point numbers, and strings. Libraries may add additional small objects such as dates. We find many programmers are curiously reluctant to create their own fundamental types which are useful for their domain—such as money, coordinates, or ranges. We thus see calculations that treat monetary amounts as plain numbers, or calculations of physical quantities that ignore units (adding inches to millimeters), or lots of code doing `if (a < upper && a > lower)`.

Strings are particularly common petri dishes for this kind of odor: A telephone number is more than just a collection of characters. If nothing else, a proper type can often include consistent display logic for when it needs to be displayed in a user interface. Representing such types as strings is such a common stench that people call them "stringly typed" variables.

You can move out of the primitive cave into the centrally heated world of meaningful types by using *Replace Primitive with Object (174)*. If the primitive is a type code controlling conditional behavior, use *Replace Type Code with Subclasses (362)* followed by *Replace Conditional with Polymorphism (272)*.

Groups of primitives that commonly appear together are data clumps and should be civilized with *Extract Class (182)* and *Introduce Parameter Object (140)*.

Repeated Switches

Talk to a true object-oriented evangelist and they'll soon get onto the evils of switch statements. They'll argue that any switch statement you see is begging for *Replace Conditional with Polymorphism (272)*. We've even heard some people argue that all conditional logic should be replaced with polymorphism, tossing most `ifs` into the dustbin of history.

Even in our more wild-eyed youth, we were never unconditionally opposed to the conditional. Indeed, the first edition of this book had a smell entitled "switch statements." The smell was there because in the late 90's we found polymorphism sadly underappreciated, and saw benefit in getting people to switch over.

These days there is more polymorphism about, and it isn't the simple red flag that it often was fifteen years ago. Furthermore, many languages support more sophisticated forms of switch statements that use more than some primitive code as their base. So we now focus on the repeated switch, where the same conditional switching logic (either in a switch/case statement or in a cascade of if/else statements) pops up in different places. The problem with such duplicate switches is that, whenever you add a clause, you have to find all the switches and update them. Against the dark forces of such repetition, polymorphism provides an elegant weapon for a more civilized codebase.

Loops

Loops have been a core part of programming since the earliest languages. But we feel they are no more relevant today than bell-bottoms and flock wallpaper. We disdained them at the time of the first edition—but Java, like most other languages at the time, didn't provide a better alternative. These days, however, first-class functions are widely supported, so we can use *Replace Loop with Pipeline (231)* to retire those anachronisms. We find that pipeline operations, such as filter and map, help us quickly see the elements that are included in the processing and what is done with them.

Lazy Element

We like using program elements to add structure—providing opportunities for variation, reuse, or just having more helpful names. But sometimes the structure isn't needed. It may be a function that's named the same as its body code reads, or a class that is essentially one simple function. Sometimes, this reflects a function that was expected to grow and be popular later, but never realized its dreams. Sometimes, it's a class that used to pay its way, but has been downsized with refactoring. Either way, such program elements need to die with dignity. Usually this means using *Inline Function (115)* or *Inline Class (186)*. With inheritance, you can use *Collapse Hierarchy (380)*.

Speculative Generality

Brian Foote suggested this name for a smell to which we are very sensitive. You get it when people say, "Oh, I think we'll need the ability to do this kind of thing someday" and thus add all sorts of hooks and special cases to handle things that aren't required. The result is often harder to understand and maintain. If all this machinery were being used, it would be worth it. But if it isn't, it isn't. The machinery just gets in the way, so get rid of it.

If you have abstract classes that aren't doing much, use *Collapse Hierarchy (380)*. Unnecessary delegation can be removed with *Inline Function (115)* and *Inline Class (186)*. Functions with unused parameters should be subject to *Change Function Declaration (124)* to remove those parameters. You should also apply *Change Function Declaration (124)* to remove any unneeded parameters, which often get tossed in for future variations that never come to pass.

Speculative generality can be spotted when the only users of a function or class are test cases. If you find such an animal, delete the test case and apply *Remove Dead Code (237)*.

Temporary Field

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

Use *Extract Class (182)* to create a home for the poor orphan variables. Use *Move Function (198)* to put all the code that concerns the fields into this new class.

You may also be able to eliminate conditional code by using *Introduce Special Case (289)* to create an alternative class for when the variables aren't valid.

Message Chains

You see message chains when a client asks one object for another object, which the client then asks for yet another object, which the client then asks for yet another another object, and so on. You may see these as a long line of `getThis` methods, or as a sequence of temps. Navigating this way means the client is coupled to the structure of the navigation. Any change to the intermediate relationships causes the client to have to change.

The move to use here is *Hide Delegate (189)*. You can do this at various points in the chain. In principle, you can do this to every object in the chain, but doing this often turns every intermediate object into a middle man. Often, a better alternative is to see what the resulting object is used for. See whether you can use *Extract Function (106)* to take a piece of the code that uses it and then *Move Function (198)* to push it down the chain. If several clients of one of the objects in the chain want to navigate the rest of the way, add a method to do that.

Some people consider any method chain to be a terrible thing. We are known for our calm, reasoned moderation. Well, at least in this case we are.

Middle Man

One of the prime features of objects is encapsulation—hiding internal details from the rest of the world. Encapsulation often comes with delegation. You ask a director whether she is free for 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.

c However, this can go too far. You look at a class's interface and find half the methods are delegating to this other class. After a while, it is time to use *Remove Middle Man (192)* and talk to the object that really knows what's going on. If only a few methods aren't doing much, use *Inline Function (115)* to inline them into the caller. If there is additional behavior, you can use *Replace Superclass with Delegate (399)* or *Replace Subclass with Delegate (381)* to fold the middle man into the real object. That allows you to extend behavior without chasing all that delegation.

Insider Trading

Software people like strong walls between their modules and complain bitterly about how trading data around too much increases coupling. To make things work, some trade has to occur, but we need to reduce it to a minimum and keep it all above board.

Modules that whisper to each other by the coffee machine need to be separated by using *Move Function (198)* and *Move Field (207)* to reduce the need to chat. If modules have common interests, try to create a third module to keep that commonality in a well-regulated vehicle, or use *Hide Delegate (189)* to make another module act as an intermediary.

Inheritance can often lead to collusion. Subclasses are always going to know more about their parents than their parents would like them to know. If it's time to leave home, apply *Replace Subclass with Delegate (381)* or *Replace Superclass with Delegate (399)*.

Large Class

When a class is trying to do too much, it often shows up as too many fields. When a class has too many fields, duplicated code cannot be far behind.

You can *Extract Class (182)* 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 with inheritance, you'll find *Extract Superclass (375)* or *Replace Type Code with Subclasses (362)* (which essentially is extracting a subclass) are often easier.

Sometimes a class does not use all of its fields all of the time. If so, you may be able to do these extractions many times.

As with a class with too many instance variables, a class with too much code is a prime breeding ground for duplicated code, chaos, and death. The simplest solution (have we mentioned that we like simple solutions?) is to 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.

The clients of such a class are often the best clue for splitting up the class. Look at whether clients use a subset of the features of the class. Each subset is a possible separate class. Once you've identified a useful subset, use *Extract Class (182)*, *Extract Superclass (375)*, or *Replace Type Code with Subclasses (362)* to break it out.

Alternative Classes with Different Interfaces

One of the great benefits of using classes is the support for substitution, allowing one class to swap in for another in times of need. But this only works if their interfaces are the same. Use *Change Function Declaration (124)* to make functions match up. Often, this doesn't go far enough; keep using *Move Function (198)* to move behavior into classes until the protocols match. If this leads to duplication, you may be able to use *Extract Superclass (375)* to atone.

Data Class

These are classes that have fields, getting and setting methods for the fields, and nothing else. Such classes are dumb data holders and are often being manipulated in far too much detail by other classes. In some stages, these classes may have public fields. If so, you should immediately apply *Encapsulate Record (162)* before anyone notices. Use *Remove Setting Method (331)* on any field that should not be changed.

Look for where these getting and setting methods are used by other classes. Try to use *Move Function (198)* to move behavior into the data class. If you can't move a whole function, use *Extract Function (106)* to create a function that can be moved.

Data classes are often a sign of behavior in the wrong place, which means you can make big progress by moving it from the client into the data class itself. But there are exceptions, and one of the best exceptions is a record that's being used as a result record from a distinct function invocation. A good example of this is the intermediate data structure after you've applied *Split Phase (154)*. A key characteristic of such a result record is that it's immutable (at least in practice). Immutable fields don't need to be encapsulated and information derived from immutable data can be represented as fields rather than getting methods.

Refused Bequest

Subclasses get to inherit the methods and data of their parents. But what if they don't want or need what they are given? They are given all these great gifts and pick just a few to play with.

The traditional story is that this means the hierarchy is wrong. You need to create a new sibling class and use *Push Down Method (359)* and *Push Down Field (361)* to push all the unused code to the sibling. That way the parent holds only what is common. Often, you'll hear advice that all superclasses should be abstract.

You'll guess from our snide use of "traditional" that we aren't going to advise this—at least not all the time. We do subclassing to reuse a bit of behavior all the time, and we find it a perfectly good way of doing business. There is a smell—we can't deny it—but usually it isn't a strong smell. So, we say that if the refused bequest is causing confusion and problems, follow the traditional advice. However, don't feel you have to do it all the time. Nine times out of ten this smell is too faint to be worth cleaning.

The smell of refused bequest is much stronger if the subclass is reusing behavior but does not want to support the interface of the superclass. We don't mind refusing implementations—but refusing interface gets us on our high horses. In this case, however, don't fiddle with the hierarchy; you want to gut it by applying *Replace Subclass with Delegate (381)* or *Replace Superclass with Delegate (399)*.

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 comments here is that 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.

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

If you need a comment to explain what a block of code does, try *Extract Function (106)*. If the method is already extracted but you still need a comment to explain what it does, use *Change Function Declaration (124)* to rename it. If you need to state some rules about the required state of the system, use *Introduce Assertion (302)*.

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

A good time to use a comment is when you don't know what to do. In addition to describing what is going on, comments can indicate areas in which you aren't sure. A comment can also explain why you did something. This kind of information helps future modifiers, especially forgetful ones.