Chapter 4. Code Smells

> When you have learned to look at your words with critical detachment, you will find that rereading a piece five or six times in a row will each time bring to light fresh spots of trouble. [Barzun, 229]

Refactoring, or improving the design of existing code, requires that you know what code needs improvement. Catalogs of refactorings help you gain this knowledge, yet your situation may be different from what you see in a catalog. It's therefore necessary to learn common design problems so you can recognize them in your own code.

The most common design problems result from code that

- Is duplicated
- Is unclear
- Is complicated

These criteria can certainly help you discover places in code that need improvement. On the other hand, many programmers find this list to be too vague; they don't know how to spot duplication in code that isn't outwardly the same, they aren't sure how to tell when code is clearly communicating its intent, and they don't know how to distinguish simple code from complicated code.

In their chapter "Bad Smells in Code" in *Refactoring* [F], Martin Fowler and Kent Beck provide additional guidance for identifying design problems. They liken design problems to smells and explain which refactorings, or combinations of refactorings, work best to eliminate odors.

Fowler and Beck's code smells target problems that occur everywhere: in methods, classes, hierarchies, packages (namespaces, modules), and entire systems. The names of their smells, such as Feature Envy, Primitive Obsession, and Speculative Generality, provide a rich and colorful vocabulary with which programmers may rapidly communicate about design problems.

I decided it would be useful to discover which of Fowler and Beck's 22 code smells are addressed by the refactorings I present in this book. While completing this task, I discovered 5 new code smells that suggest the need for pattern-directed refactorings. In all, the refactorings in this book address 12 code smells.

Table 4.1 lists the 12 smells and some refactorings to consider when you want to remove the smells. Deodorizing such smells is best done by considering the associated refactorings. The sections in this chapter discuss each of the 12 smells in turn and provide guidance for when to use the different refactorings.

Table 4.1

Smell[a]	Refactoring
Duplicated Code (39) [F]	*Form Template Method (205)* *Introduce Polymorphic Creation with Factory Method (88)* *Chain Constructors (340)* *Replace One/Many Distinctions with Composite (224)* *Extract Composite (214)* *Unify Interfaces with Adapter (247)* *Introduce Null Object (301)*
Long Method (40) [F]	*Compose Method (123)* *Move Accumulation to Collecting Parameter (313)* *Replace Conditional Dispatcher with Command (191)* *Move Accumulation to Visitor (320)* *Replace Conditional Logic with Strategy (129)*
Conditional Complexity (41)	*Replace Conditional Logic with Strategy (129)* *Move Embellishment to Decorator (144)* *Replace State-Altering Conditionals with State (166)* *Introduce Null Object (301)*
Primitive Obsession (41) [F]	*Replace Type Code with Class (286)* *Replace State-Altering Conditionals with State (166)* *Replace Conditional Logic with Strategy (129)* *Replace Implicit Tree with Composite (178)* *Replace Implicit Language with Interpreter (269)* *Move Embellishment to Decorator (144)* *Encapsulate Composite with Builder (96)*
Indecent Exposure (42)	*Encapsulate Classes with Factory (80)*
Solution Sprawl (43)	*Move Creation Knowledge to Factory (68)*
Alternative Classes with Different Interfaces (43) [F]	*Unify Interfaces with Adapter (247)*

Smell[a]	Refactoring
Lazy Class (43) [F]	*Inline Singleton (114)*
Large Class (44) [F]	*Replace Conditional Dispatcher with Command (191)* *Replace State-Altering Conditionals with State (166)* *Replace Implicit Language with Interpreter (269)*
Switch Statements (44) [F]	*Replace Conditional Dispatcher with Command (191)* *Move Accumulation to Visitor (320)*
Combinatorial Explosion (45)	*Replace Implicit Language with Interpreter (269)*
Oddball Solution (45)	*Unify Interfaces with Adapter (247)*

a. Page numbers refer to the page of the current book where the smell is discussed further. [F] indicates that the smell is discussed in Fowler and Beck's chapter "Bad Smells in Code" in *Refactoring* [F].

Duplicated Code

Duplicated code is the most pervasive and pungent smell in software. It tends to be either explicit or subtle. Explicit duplication exists in identical code, while subtle duplication exists in structures or processing steps that are outwardly different yet essentially the same.

You can often remove explicit and/or subtle duplication in subclasses of a hierarchy by applying *Form Template Method (205)*. If a method in the subclasses is implemented similarly, except for an object creation step, applying *Introduce Polymorphic Creation with Factory Method (88)* will pave the way for removing more duplication by means of a Template Method.

If the constructors of a class contain duplicated code, you can often eliminate the duplication by applying *Chain Constructors (340)*.

If you have separate code for processing a single object or a collection of objects, you may be able to remove duplication by applying *Replace One/Many Distinctions with Composite (224)*.

If subclasses of a hierarchy each implement their own Composite, the implementations may be identical, in which case you can use *Extract Composite (214)*.

If you process objects differently merely because they have different interfaces, applying *Unify Interfaces with Adapter (247)* will pave the way for removing duplicated processing logic.

If you have conditional logic to deal with an object when it is null and the same null logic is duplicated throughout your system, applying *Introduce Null Object (301)* will eliminate the duplication and simplify the system.

Long Method

In their description of this smell, Fowler and Beck [F] explain several good reasons why short methods are superior to long methods. A principal reason involves the sharing of logic. Two long methods may very well contain duplicated code. Yet if you break those methods into smaller ones, you can often find ways for them to share logic.

Fowler and Beck also describe how small methods help explain code. If you don't understand what a chunk of code does and you extract that code to a small, well-named method, it will be easier to understand the original code. Systems that have a majority of small methods tend to be easier to extend and maintain because they're easier to understand and contain less duplication.

What is the preferred size of small methods? I would say ten lines of code or fewer, with the majority of your methods using one to five lines of code. If you make the vast majority of a system's methods small, you can have a few methods that are larger, as long as they are simple to understand and don't contain duplication.

Some programmers choose not to write small methods because they fear the performance costs associated with chaining calls to many small methods. This is an unfortunate choice for several reasons. First, good designers don't prematurely optimize code. Second, chaining together small method calls often costs very little in performance—a fact you can confirm by using a profiler. Third, if you do happen to experience performance problems, you can often refactor to improve performance without having to give up your small methods.

When I'm faced with a long method, one of my first impulses is to break it down into a Composed Method [Beck, SBPP] by applying the refactoring *Compose Method (123)*. This work usually involves applying *Extract Method* [F]. If the code you're transforming into a Composed Method accumulates information to a common variable, consider applying *Move Accumulation to Collecting Parameter (313)*.

If your method is long because it contains a large switch statement for dispatching and handling requests, you can shrink the method by using *Replace Conditional Dispatcher with Command (191)*.

If you use a switch statement to gather data from numerous classes with different interfaces, you can shrink the size of the method by applying *Move Accumulation to Visitor (320)*.

If a method is long because it contains numerous versions of an algorithm and conditional logic to choose which version to use at runtime, you can shrink the size of the method by applying *Replace Conditional Logic with Strategy (129)*.

Conditional Complexity

Conditional logic is innocent in its infancy, when it is simple to understand and contained within a few lines of code. Unfortunately, it rarely ages well. For example, you implement several new features and suddenly your conditional logic becomes complicated and expansive. Several refactorings in *Refactoring* [F] and this catalog address such problems.

If conditional logic controls which of several variants of a calculation to execute, consider applying *Replace Conditional Logic with Strategy (129)*.

If conditional logic controls which of several pieces of special-case behavior must be executed in addition to the class's core behavior, you may want to use *Move Embellishment to Decorator (144)*.

If the conditional expressions that control an object's state transitions are complex, consider simplifying the logic by applying *Replace State-Altering Conditionals with State (166)*.

Dealing with null cases often leads to the creation of conditional logic. If the same null conditional logic is duplicated throughout your system, you can clean it up by using *Introduce Null Object (301)*.

Primitive Obsession

Primitives, which include integers, strings, doubles, arrays, and other low-level language elements, are generic because many people use them. Classes, on the other hand, may be as specific as you need them to be because you create them for specific purposes. In many cases, classes provide a simpler and more natural way to model things than primitives. In addition, once you create a class, you'll often discover that other code in a system belongs in that class.

Fowler and Beck [F] explain how Primitive Obsession manifests itself when code

relies too much on primitives. This typically occurs when you haven't yet seen how a higher-level abstraction can clarify or simplify your code. Fowler's refactorings include many of the most common solutions for addressing this problem. This book builds on those solutions and offers more.

If a primitive value controls logic in a class and the primitive value isn't type-safe (i.e., clients can assign it to an unsafe or incorrect value), consider applying *Replace Type Code with Class (286)*. The result will be code that is type-safe and capable of being extended by new behavior (something you can't do with a primitive).

If an object's state transitions are controlled by complex conditional logic that uses primitive values, you can use *Replace State-Altering Conditionals with State (166)*. The result will be numerous classes to represent each state and simplified state transition logic.

If complicated conditional logic controls which algorithm to run and that logic relies on primitive values, consider applying *Replace Conditional Logic with Strategy (129)*.

If you implicitly create a tree structure using a primitive representation, such as a string, your code may be difficult to work with, prone to errors, and/or filled with duplication. Applying *Replace Implicit Tree with Composite (178)* will reduce these problems.

If many methods of a class exist to support numerous combinations of primitive values, you may have an implicit language. If so, consider applying *Replace Implicit Language with Interpreter (269)*.

If primitive values exist in a class only to provide embellishments to the class's core responsibility, you may want to use *Move Embellishment to Decorator (144)*.

Finally, even if you have a class, it may still be too primitive to make life easy for clients. This may be the case if you have a Composite [DP] implementation that is tricky to work with. You can simplify how clients build the Composite by applying *Encapsulate Composite with Builder (96)*.

Indecent Exposure

This smell indicates the lack of what David Parnas so famously termed "information hiding" [Parnas]. The smell occurs when methods or classes that ought not be visible to clients are publicly visible to them. Exposing such code means that clients know about code that is unimportant or only indirectly important.

This contributes to the complexity of a design.

The refactoring *Encapsulate Classes with Factory (80)* deodorizes this smell. Not every class that is useful to clients needs to be public (i.e., have a public constructor). Some classes ought to be referenced only via their common interfaces. You can make that happen if you make the class's constructors non-public and use a Factory to produce instances.

Solution Sprawl

When code and/or data used to perform a responsibility becomes sprawled across numerous classes, Solution Sprawl is in the air. This smell often results from quickly adding a feature to a system without spending enough time simplifying and consolidating the design to best accommodate the feature.

Solution Sprawl is the identical twin brother of Shotgun Surgery, a smell described by Fowler and Beck [F]. You become aware of this smell when adding or updating a system feature causes you to make changes to many different pieces of code. Solution Sprawl and Shotgun Surgery address the same problem, yet are sensed differently. We become aware of Solution Sprawl by observing it, while we become aware of Shotgun Surgery by doing it.

Move Creation Knowledge to Factory (68) is a refactoring that solves the problem of a sprawling object creation responsibility.

Alternative Classes with Different Interfaces

This Fowler and Beck [F] coding smell occurs when the interfaces of two classes are different and yet the classes are quite similar. If you can find the similarities between the two classes, you can often refactor the classes to make them share a common interface.

However, sometimes you can't directly change the interface of a class because you don't have control over the code. The typical example is when you're working with a third-party library. In that case, you can apply *Unify Interfaces with Adapter (247)* to produce a common interface for the two classes.

Lazy Class

When describing this smell, Fowler and Beck write, "A class that isn't doing enough to pay for itself should be eliminated" [F, 83]. It's not uncommon to encounter a Singleton [DP] that isn't paying for itself. In fact, the Singleton may be costing you something by making your design too dependent on what amounts to global data. *Inline Singleton (114)* explains a quick, humane procedure for eliminating a Singleton.

Large Class

Fowler and Beck [F] note that the presence of too many instance variables usually indicates that a class is trying to do too much. In general, large classes typically contain too many responsibilities. *Extract Class* [F] and *Extract Subclass* [F], which are some of the main refactorings used to address this smell, help move responsibilities to other classes. The pattern-directed refactorings in this book make use of these refactorings to reduce the size of classes.

Replace Conditional Dispatcher with Command (191) extracts behavior into Command [DP] classes, which can greatly reduce the size of a class that performs a variety of behaviors in response to different requests.

Replace State-Altering Conditionals with State (166) can reduce a large class filled with state transition code into a small class that delegates to a family of State [DP] classes.

Replace Implicit Language with Interpreter (269) can reduce a large class into a small one by transforming copious code for emulating a language into a small Interpreter [DP].

Switch Statements

Switch statements (or their equivalent, `if…elseif…elseif…` structures) aren't inherently bad. They become bad only when they make your design more complicated or rigid than it needs to be. In that case, it's best to refactor away from switch statements to a more object-based or polymorphic solution.

Replace Conditional Dispatcher with Command (191) describes how to break down a large switch statement into a collection of Command [DP] objects, each of which may be looked up and invoked without relying on conditional logic.

Move Accumulation to Visitor (320) describes an example where switch

statements are used to obtain data from instances of classes that have different interfaces. By refactoring the code to use a Visitor [DP], no conditional logic is needed and the design becomes more flexible.

Combinatorial Explosion

This smell is a subtle form of duplication. It exists when you have numerous pieces of code that do the same thing using different kinds or quantities of data or objects.

For example, say you have numerous methods on a class for performing queries. Each of these methods performs a query using specific conditions and data. The more specialized queries you need to support, the more query methods you must create. Pretty soon you have an explosion of methods to handle the many ways of performing queries. You also have an implicit query language. You can remove all of these methods and the combinatorial explosion smell by applying *Replace Implicit Language with Interpreter (269)*.

Oddball Solution

When a problem is solved one way throughout a system and the same problem is solved another way in the same system, one of the solutions is the oddball or inconsistent solution. The presence of this smell usually indicates subtly duplicated code.

To remove this duplication, first determine your preferred solution. In some cases, the solution used least often may be your preferred solution if it is better than the solution used most of the time. After determining your preferred solution, you can often apply *Substitute Algorithm* [F] to produce a consistent solution throughout your system. Given a consistent solution, you may be able to move all instances of the solution to one place, thereby removing duplication.

The Oddball Solution smell is usually present when you have a preferred way to communicate with a set of classes, yet differences in the interfaces of the classes prevent you from communicating with them in a consistent way. In that case, consider applying *Unify Interfaces with Adapter (247)* to produce a common interface by which you may communicate consistently with all of the classes. Once you do that, you can often discover ways to remove duplicated processing logic.