# Smells to Refactorings
# Quick Reference Guide


| Smell | Refactoring |
|---|---|
| **Alternative Classes with Different Interfaces:** 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 [F 85, K 43] | Unify Interfaces with Adapter [K 247] |
| | Rename Method [F 273] |
| | Move Method [F 142] |
| **Combinatorial Explosion:** A subtle form of duplication, this smell exists when numerous pieces of code do the same thing using different combinations of data or behavior. [K 45] | Replace Implicit Language with Interpreter [K 269] |
| **Comments (a.k.a. Deodorant):** When you feel like writing a comment, first try "to refactor so that the comment becomes superfluous" [F 87] | Rename Method [F 273] |
| | Extract Method [F 110] |
| | Introduce Assertion [F 267] |
| **Conditional Complexity:** Conditional logic is innocent in its infancy, when it's simple to understand and contained within a few lines of code. Unfortunately, it rarely ages well. You implement several new features and suddenly your conditional logic becomes complicated and expansive. [K 41] | Introduce Null Object [F 260, K 301] |
| | Move Embellishment to Decorator [K 144] |
| | Replace Conditional Logic with Strategy [K 129] |
| | Replace State-Altering Conditionals with State [K 166] |
| **Data Class:** Classes that have fields, getting and setting methods for the fields, and nothing else. Such classes are dumb data holders and are almost certainly being manipulated in far too much detail by other classes. [F 86] | Move Method [F 142] |
| | Encapsulate Field [F 206] |
| | Encapsulate Collection [F 208] |
| **Data Clumps:** Bunches of data that that hang around together really ought to be made into their own object. 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. [F 81] | Extract Class [F 149] |
| | Preserve Whole Object [F 288] |
| | Introduce Parameter Object [F 295] |
| **Divergent Change:** Occurs when one class is commonly changed in different ways for different reasons. Separating these divergent responsibilities decreases the chance that one change could affect another and lower maintenance costs. [F 79] | Extract Class [F 149] |
| **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. [F76, K 39] | Chain Constructors [K 340] |
| | Extract Composite [K 214] |
| | Extract Method [F 110] |
| | Extract Class [F 149] |
| | Form Template Method [F 345, K 205] |
| | Introduce Null Object [F 260, K 301] |
| | Introduce Polymorphic Creation with Factory Method [K 88] |
| | Pull Up Method [F 322] |
| | Pull Up Field [F 320] |
| | Replace One/Many Distinctions with Composite [K 224] |
| | Substitue Algorithm [F 139] |
| | Unify Interfaces with Adapter [K 247] |
| **Feature Envy:** Data and behavior that acts on that data belong together. When a method makes too many calls to other classes to obtain data or functionality, Feature Envy is in the air. [F 80] | Extract Method [F 110] |
| | Move Method [F 142] |
| | Move Field [F 146] |
| **Freeloader (a.k.a. Lazy Class):** A class that isn't doing enough to pay for itself should be eliminated. [F 83, K 43] | Collapse Hierarchy [F 344] |
| | Inline Class [F 154] |
| | Inline Singleton [K 114] |
| **Inappropriate Intimacy:** Sometimes classes become far too intimate and spend too much time delving into each others' private parts. We may not be prudes when it comes to people, but we think our classes should follow strict, puritan rules. Over-intimate classes need to be broken up as lovers were in ancient days. [F 85] | Move Method [F 142] |
| | Move Field [F 146] |
| | Change Bidirectional Association to Unidirectional Association [F 20 |
| | Extract Class [F 149] |
| | Hide Delegate [F 157] |
| | Replace Inheritance with Delegation [F 352] |
| **Incomplete Library Class:** Occurs when responsibilities emerge in our code that clearly should be moved to a library class, but we are unable or unwilling to modify the library class to accept these new responsibilities. [F 86] | Introduce Foreign Method [F 162] |
| | Introduce Local Extension [F 164] |
| **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 to 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. [K 42] | Encapsulate Classes with Factory [K 80] |
| **Large Class:** Fowler and Beck 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. [F 78, K 44] | Extract Class [F 149] |
| | Extract Subclass [F 330] |
| | Extract Interface [F 341] |
| | Replace Data Value with Object [F 175] |
| | Replace Conditional Dispatcher with Command [K 191] |
| | Replace Implicit Language with Interpreter [K 269] |
| | Replace State-Altering Conditionals with State [K 166] |

F - Fowler, Martin. Refactoring: Improving the Design of Existing Code.
K - Kerievsky, Joshua. Refactoring to Patterns.

# Smells to Refactorings
# Quick Reference Guide


| Smell | Refactoring |
|---|---|
| **Long Method:** In their description of this smell, Fowler and Beck 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 methods, you can often find ways for the two 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. [F 76, K 40] | Extract Method [F 110]<br>Compose Method [K 123]<br>Introduce Parameter Object [F 295]<br>Move Accumulation to Collecting Parameter [K 313]<br>Move Accumulation to Visitor [K 320]<br>Decompose Conditional [F 238]<br>Preserve Whole Object [F 288]<br>Replace Conditional Dispatcher with Command [K 191]<br>Replace Conditional Logic with Strategy [K 129]<br>Replace Method with Method Object [F 135]<br>Replace Temp with Query [F 120] |
| **Long Parameter List:** Long lists of parameters in a method, though common in procedural code, are difficult to understand and likely to be volatile. Consider which objects this method really needs to do its job - it's okay to make the method to do some work to track down the data it needs. [F 78] | Replace Parameter with Method [F 292]<br>Introduce Parameter Object [F 295]<br>Preserve Whole Object [F 288] |
| **Message Chains:** Occur when you see a long sequence of method calls or temporary variables to get some data. This chain makes the code dependent on the relationships between many potentially unrelated objects. [F 84] | Hide Delegate [F 157]<br>Extract Method [F 110]<br>Move Method [F 142] |
| **Middle Man:** Delegation is good, and one of the key fundamental features of objects. But too much of a good thing can lead to objects that add no value, simply passing messages on to another object. [F 85] | Remove Middle Man [F 160]<br>Inline Method [F 117]<br>Replace Delegation with Inheritance [F 355] |
| **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. [K 45] | Unify Interfaces with Adapter [K 247] |
| **Parallel Inheritance Hierarchies:** This is really a special case of Shotgun Surgery - every time you make a subclass of one class, you have to make a subclass of another. [F 83] | Move Method [F 142]<br>Move Field [F 146] |
| **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, since 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 how other code in a system belongs in that class. Fowler and Beck 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. [F 81, K 41] | Replace Data Value with Object [F 175]<br>Encapsulate Composite with Builder [K 96]<br>Introduce Parameter Object [F 295]<br>Extract Class [F 149]<br>Move Embellishment to Decorator [K 144]<br>Replace Conditional Logic with Strategy [K 129]<br>Replace Implicit Language with Interpreter [K 269]<br>Replace Implicit Tree with Composite [K 178]<br>Replace State-Altering Conditionals with State [K 166]<br>Replace Type Code with Class [F 218, K 286]<br>Replace Type Code with State/Strategy [F 227]<br>Replace Type Code with Subclasses [F 223]<br>Replace Array With Object [F 186] |
| **Refused Bequest:** This smell results from inheriting code you don't want. Instead of tolerating the inheritance, you write code to refuse the "bequest" -- which leads to ugly, confusing code, to say the least. [F 87] | Push Down Field [F 329]<br>Push Down Method [F 322]<br>Replace Inheritance with Delegation [F 352] |
| **Shotgun Surgery:** This smell is evident when you must change lots of pieces of code in different places simply to add a new or extended piece of behavior. [F 80] | Move Method [F 142]<br>Move Field [F 146]<br>Inline Class [F 154] |
| **Solution Sprawl:** When code and/or data used in performing 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. [K 43] | Move Creation Knowledge to Factory [K 68] |
| **Speculative Generality:** This odor exists when you have generic or abstract code that isn't actually needed today. Such code often exists to support future behavior, which may or may not be necessary in the future. [F 83] | Collapse Hierarchy [F 344]<br>Rename Method [F 273]<br>Remove Parameter [F 277]<br>Inline Class [F 154] |
| **Switch Statement:** This smell exists when the same switch statement (or "if…else if…else if" statement) is duplicated across a system. Such duplicated code reveals a lack of object orientation and a missed opportunity to rely on the elegance of polymorphism. [F 82, K 44] | Move Accumulation to Visitor [K 320]<br>Replace Conditional Dispatcher with Command [K 191]<br>Replace Conditional with Polymorphism [F 255]<br>Replace Type Code with Subclasses [F 223]<br>Replace Type Code with State/Strategy [F 227]<br>Replace Parameter with Explicit Methods [F 285]<br>Introduce Null Object [F 260, K 301] |
| **Temporary Field:** Objects sometimes contain fields that don't seem to be needed all the time. The rest of the time, the field is empty or contains irrelevant data, which is difficult to understand. This is often an alternative to Long Parameter List. [F 84] | Extract Class [F 149]<br>Introduce Null Object [F 260, K 301] |

F - Fowler, Martin. Refactoring: Improving the Design of Existing Code.
K - Kerievsky, Joshua. Refactoring to Patterns.