Catalog of Refactoring

Code Smells

— What? How can code "smell"??
— Well it doesn't have a nose... but it definitely can stink!

Bloaters

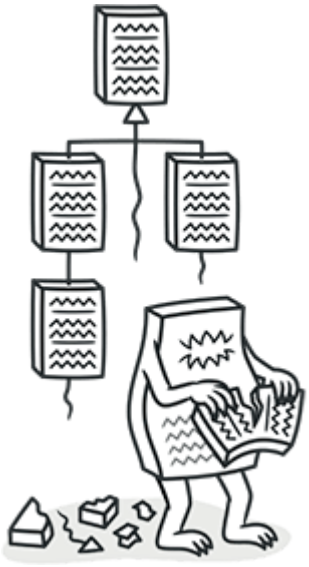
Bloaters are code, methods and classes that have increased to such gargantuan proportions that they are hard to work with. Usually these smells do not crop up right away, rather they accumulate over time as the program evolves (and especially when nobody makes an effort to eradicate them).

- **Long Method**
- **Large Class**
- **Primitive Obsession**
- **Long Parameter List**
- **Data Clumps**

Object-Orientation Abusers

All these smells are incomplete or incorrect application of object-oriented programming principles.

- **Alternative**
- **Refused Bequest**
- **Temporary Field**



- **Classes with Different Interfaces**
- **Switch Statements**

Change Preventers

These smells mean that if you need to change something in one place in your code, you have to make many changes in other places too. Program development becomes much more complicated and expensive as a result.

- **Divergent Change**
- **Parallel Inheritance Hierarchies**
- **Shotgun Surgery**

Dispensables

A dispensable is something pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand.

- **Comments**
- **Data Class**
- **Lazy Class**
- **Duplicate Code**
- **Dead Code**
- **Speculative Generality**

Couplers

All the smells in this group contribute to excessive coupling between classes or show what happens if coupling is replaced by excessive delegation.

- **Feature Envy**
- **Inappropriate Intimacy**
- **Incomplete Library Class**
- **Message Chains**
- **Middle Man**

Refactoring Techniques

Composing Methods

Much of refactoring is devoted to correctly composing methods. In most cases, excessively long methods are the root of all evil. The vagaries of code inside these methods conceal the execution logic and make the method extremely hard to understand—and even harder to change.

The refactoring techniques in this group streamline methods, remove code duplication, and pave the way for future improvements.

- **Extract Method**
- **Inline Method**
- **Extract Variable**
- **Inline Temp**
- **Replace Temp with Query**
- **Split Temporary Variable**
- **Remove Assignments to Parameters**
- **Replace Method with Method Object**
- **Substitute Algorithm**

Moving Features between Objects

Even if you have distributed functionality among different classes in a less-than-perfect way, there is still hope.

These refactoring techniques show how to safely move functionality between classes, create new classes, and hide implementation details from public access.

- **Move Method**
- **Move Field**
- **Extract Class**
- **Inline Class**
- **Hide Delegate**
- **Remove Middle Man**
- **Introduce Foreign Method**
- **Introduce Local Extension**

Organizing Data

These refactoring techniques help with data handling, replacing primitives with rich class functionality. Another important result is untangling of class associations, which makes classes more portable and reusable.

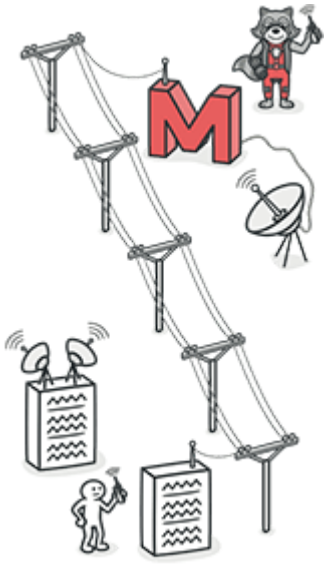
- **Change Value to Reference**
- **Change Reference to Value**
- **Duplicate Observed Data**
- **Self Encapsulate Field**
- **Replace Data Value with Object**
- **Replace Array with Object**
- **Change Unidirectional Association to Bidirectional**
- **Change Bidirectional Association to Unidirectional**
- **Encapsulate Field**
- **Encapsulate Collection**
- **Replace Magic Number with Symbolic Constant**
- **Replace Type Code with Class**
- **Replace Type Code with Subclasses**
- **Replace Type Code with State/Strategy**
- **Replace Subclass with Fields**

Simplifying Conditional Expressions

Conditionals tend to get more and more complicated in their logic over time, and there are yet more techniques to combat this as well.

- **Consolidate Conditional Expression**
- **Consolidate Duplicate Conditional Fragments**
- **Replace Conditional with Polymorphism**
- **Remove Control Flag**
- **Replace Nested Conditional with Guard Clauses**
- **Introduce Null Object**
- **Introduce Assertion**

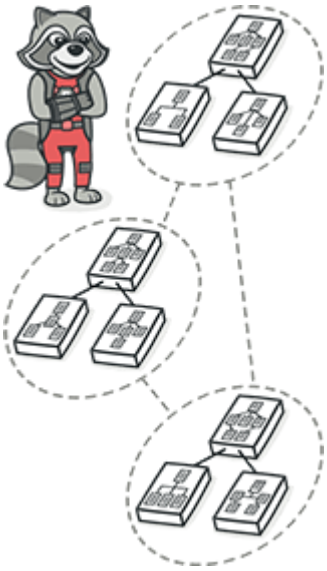
- **Decompose Conditional**



Simplifying Method Calls

These techniques make method calls simpler and easier to understand. This, in turn, simplifies the interfaces for interaction between classes.

- **Add Parameter**
- **Remove Parameter**
- **Rename Method**
- **Separate Query from Modifier**
- **Parameterize Method**
- **Introduce Parameter Object**
- **Preserve Whole Object**
- **Remove Setting Method**
- **Replace Parameter with Explicit Methods**
- **Replace Parameter with Method Call**
- **Hide Method**
- **Replace Constructor with Factory Method**
- **Replace Error Code with Exception**
- **Replace Exception with Test**



Dealing with Generalization

Abstraction has its own group of refactoring techniques, primarily associated with moving functionality along the class inheritance hierarchy, creating new classes and interfaces, and replacing inheritance with delegation and vice versa.

- **Pull Up Field**
- **Pull Up Method**
- **Pull Up Constructor Body**
- **Extract Subclass**
- **Extract Superclass**
- **Extract Interface**
- **Form Template Method**
- **Replace Inheritance with**