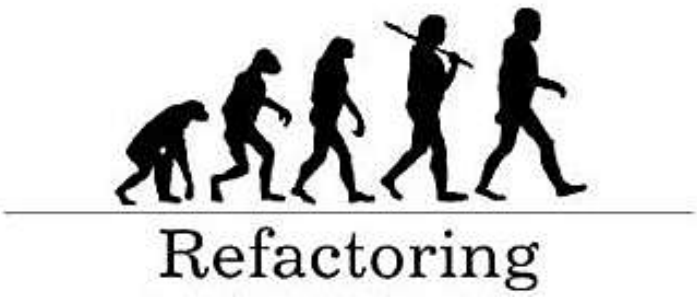
Code Smells and Refactoring



Technical Debt

A. "Ward" Cunningham (creador de la primera Wiki y uno de los pioneros en Design Pattern y Extreme Programming) encontró conexiones entre el software y las finanzas.

¿Puedes encontrar esas conexiones? :

Software	**Finanzas**
Dejar o aplazar el diseño. (Technical Debt)	Pagar los intereses.
Mejorar el diseño, preservando el comportamiento de la aplicación. (Refactoring)	Tomar dinero prestado.
Desarrollo lento debido a problemas en el diseño. (Code Smells)	Amortizar la deuda original.

B. Hay buenas razones para tomar una deuda financiera.
Nombra al menos 2 buenas razones o circunstancias para tomar una deuda técnica.

C. Si utilizas una tarjeta de crédito. ¿Cuál es la mejor manera de pagar la deuda?
   a) Pagar todo el balance en el mes.
   b) Pagar cada mes una parte considerable hasta completar toda la deuda.
   c) Esperar varios meses hasta pagar.

D. Acumular deuda financiera lleva a la bancarrota.
Nombra al menos 2 problemas que trae acumular deuda técnica.

Code Smells

Long Method	Los métodos cortos son más fáciles de leer, entender y atacar. Refactoriza métodos largos en varios más cortos de ser posible.
Dead Code	Eliminar sin piedad el código que no está siendo utilizado. Es por eso que tenemos Source Control Systems.
Long Parameter List	La mayor cantidad de parámetros que tiene un método origina una mayor complejidad. Limita el número de parámetros que necesitas en un método, o usa un objeto que combine los parámetros.
Duplicated code	Código duplicado es la perdición en el desarrollo de software. Remueve el código duplicado apenas sea posible. También deberías estar atento a casos sutiles de duplicación.
Data Clumps	Si siempre ves los mismos datos dando vueltas juntos, quizás deberían pertenecer a una sola unidad. Considera juntar datos relacionados en una clase.
Primitive Obsession	No utilices variables de tipos de datos primitivos como un pobre sustituto de una clase. Si tu tipo de dato es lo suficientemente complejo, escribe una clase que lo represente.
Refused Bequest	Si heredas de una clase, pero nunca usas ninguna de las funcionalidades heredadas, ¿en realidad deberías estar usando herencia?
Switch Statements	Este smell existe cuando el mismo switch (o cadena de "if…else if…else if") está duplicado a lo largo del sistema. Esta duplicación revela la falta de orientación a objetos y pierde la oportunidad de confiar en la elegancia del polimorfismo.
Shotgun Surgery	Si un cambio en una clase requiere varios cambios en cascada en clases relacionadas, considere refactorizar de manera que el cambio este limitado a una sola clase.
Lazy Class	Las clases deben tener su propio peso. Cada clase adicional incrementa la complejidad del proyecto. Si tienes una clase que no está haciendo lo suficiente para sustentarse por ella misma, ¿puede ser absorbida o combinada en otra clase?
Indecent Exposure	Ten cuidado con las clases que innecesariamente exponen su interior. Refactoriza agresivamente las clases para minimizar la parte pública. Debes tener una razón para cada item que haces público. Sino lo tienes, escóndelo.
Feature Envy	Métodos que hacen uso intensivo de otra clase quizás pertenezcan a otra clase. Considera mover este método a la clase a la cual le tiene envidia.
Speculative Generality	Escribe código para resolver los problemas que conoces hoy y preocúpate de los problemas de mañana cuando se materialicen. Todo el mundo se pierde en el "what if". "You Aren't Gonna Need It" (Principio YAGNI)
Divergent Change	Si, a través del tiempo, realizas cambios a una clase que toca completamente diferentes partes de la clase, quizás contenga mucha funcionalidad no relacionada. Considera aislar las partes que cambian en otra clase.
Comments	Hay una delgada línea entre los comentarios que iluminan y los que oscurecen. ¿Los comentarios son innecesarios? ¿Explican el "por qué" y no el "qué"? ¿Puedes refactorizar el código de forma que los comentarios no sean requeridos? Y recuerda, estás escribiendo comentarios para personas y no máquinas.

www.codinghorror.com/blog/2006/05/code-smells.html

Atributos de Calidad

Claridad	Cualquier persona puede escribir código que una máquina pueda entender, pero solo un buen programador escribe código que otra persona entienda.
Libre de Duplicados	Un programa debe expresar cada idea una vez y solo una vez. El código no debe ser idéntico a otro ni expresar el mismo comportamiento.
Poco Acoplamiento	Componentes poco acoplados pueden cambiarse sin impactar otros componentes.
Encapsulamiento	El código está debidamente encapsulado y no expone datos o comportamiento que debería ser invisible.
Simple	El código refleja el camino más corto a la solución y no agrega complejidad de manera innecesaria. El código simple es más fácil de mantener y evolucionar.
Alta Cohesión	Métodos hacen solo una única cosa y las clases solo tiene una única responsabilidad.

Extract Method

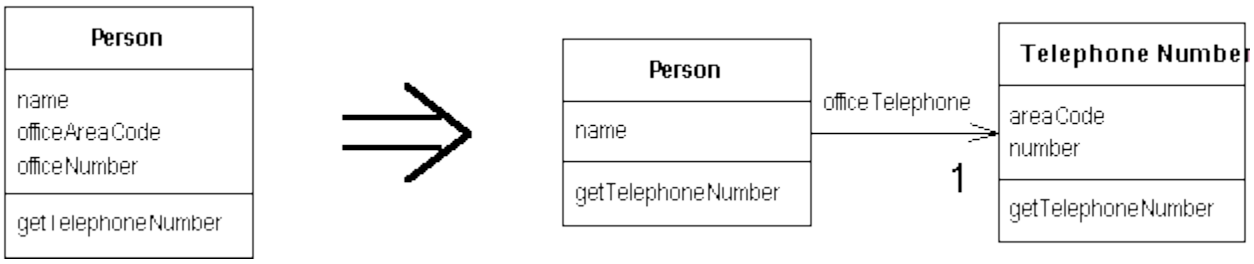
You have a code fragment that can be grouped together.
Turn the fragment into a method whose name explains the purpose of the method.

```
void printOwing() {
        printBanner();

        //print details
        System.out.println ("name:     " + _name);
        System.out.println ("amount    " + getOutstanding());
}
```

```
void printOwing() {
        printBanner();
        printDetails(getOutstanding());
}

void printDetails (double outstanding) {
        System.out.println ("name:     " + _name);
        System.out.println ("amount    " + outstanding);
}
```

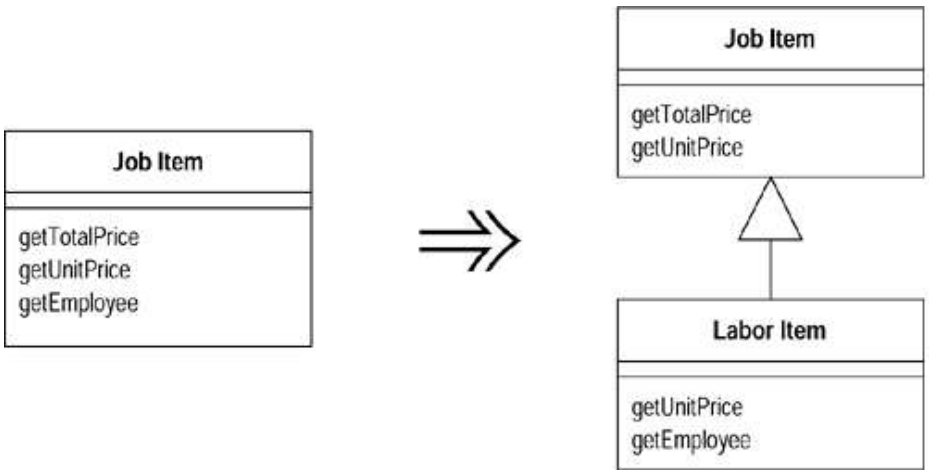
Extract Class

You have one class doing work that should be done by two.
Create a new class and move the relevant fields and methods from the old class into the new class.



Extract Sub Class

A class has features that are used only in some instances.
Create a subclass for that subset of features.



Introduce Explaining Variable

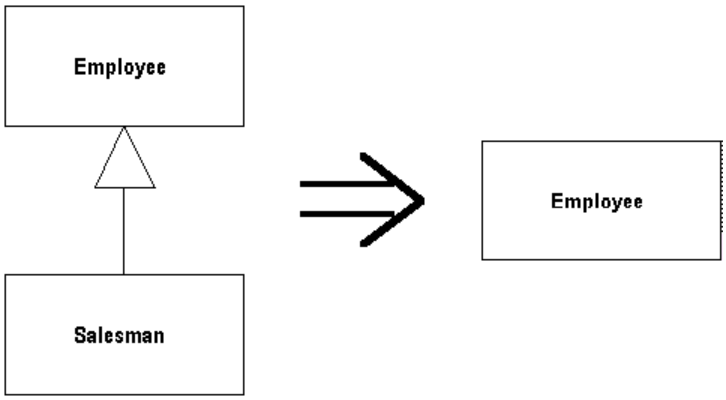
You have a complicated expression.
Put the result of the expression, or parts of the expression, in a temporary variable with a name that explains the purpose.

```
if ((platform.toUpperCase().indexOf("MAC") > -1) &&
   (browser.toUpperCase().indexOf("IE") > -1) &&
    wasInitialized() && resize > 0 ) {
         // do something
}
```

```
final boolean isMacOs     = platform.toUpperCase().indexOf("MAC") > -1;
final boolean isIEBrowser = browser.toUpperCase().indexOf("IE")  > -1;
final boolean wasResized  = resize > 0;

if (isMacOs && isIEBrowser && wasInitialized() && wasResized) {
     // do something
}
```

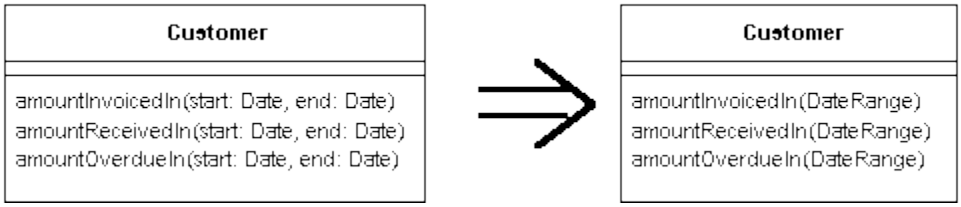
Collapse Hierarchy

A superclass and subclass are not very different.
Merge them together.



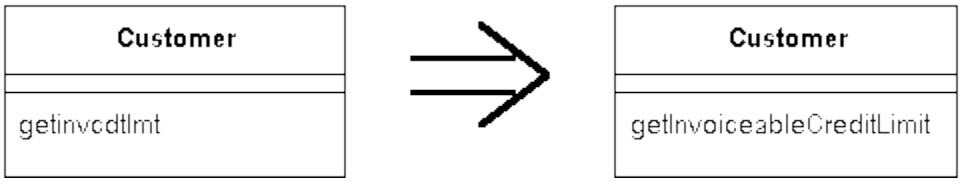
Introduce Parameter Object

You have a group of parameters that naturally go together.
Replace them with an object.



Rename

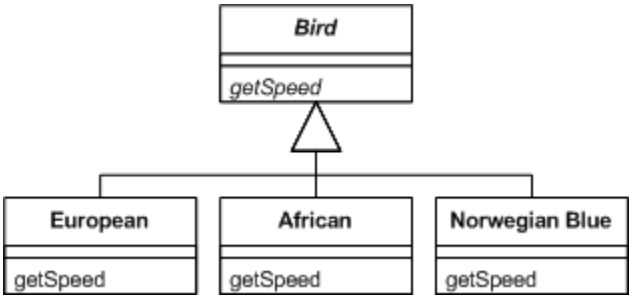
The name of a method does not reveal its purpose.
Change the name of the method.



Replace Conditional with Polymorphism

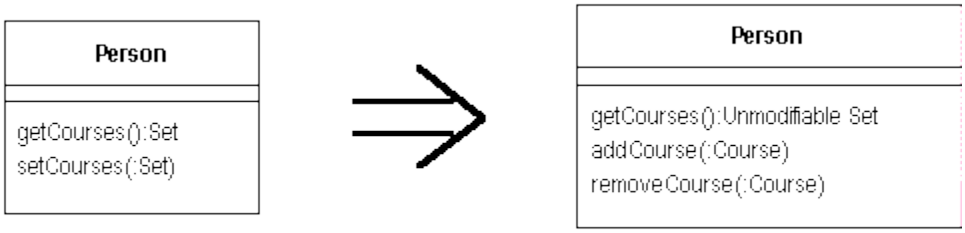
You have a conditional that chooses different behavior depending on the type of an object.
Move each leg of the conditional to an overriding method in a subclass. Make the original method abstract.

```
double getSpeed() {
  switch (_type) {
    case EUROPEAN:
      return getBaseSpeed();
    case AFRICAN:
      return getBaseSpeed() - getLoadFactor() * _numberOfCoconuts;
    case NORWEGIAN_BLUE:
      return (_isNailed) ? 0 : getBaseSpeed(_voltage);
  }
  throw new RuntimeException ("Should be unreachable");
}
```



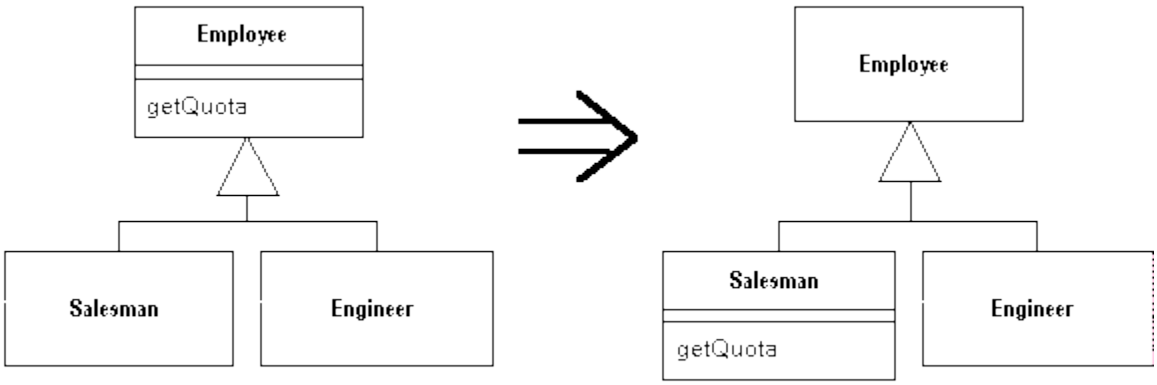
Encapsulate Collection

A method returns a collection.
Make it return a read-only view and provide add/remove methods.



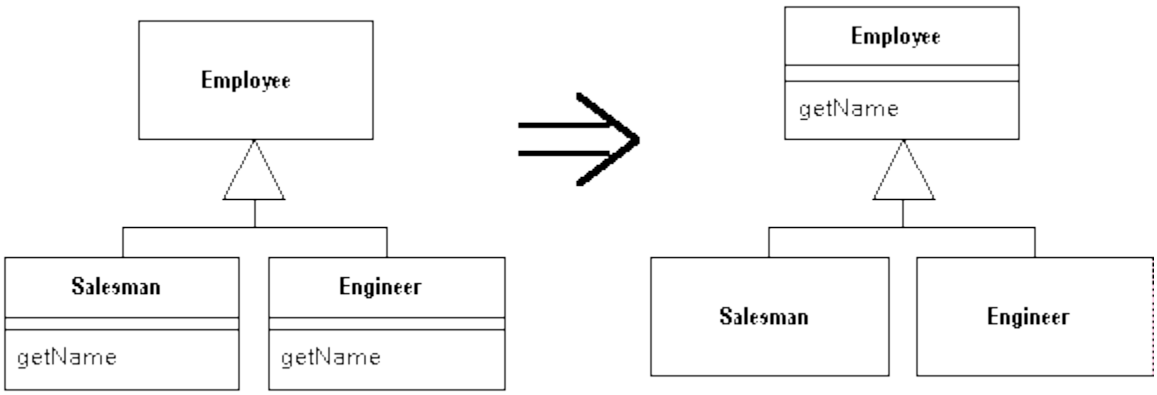
Push Down Field/Method

Behavior on a superclass is relevant only for some of its subclasses.
Move it to those subclasses.



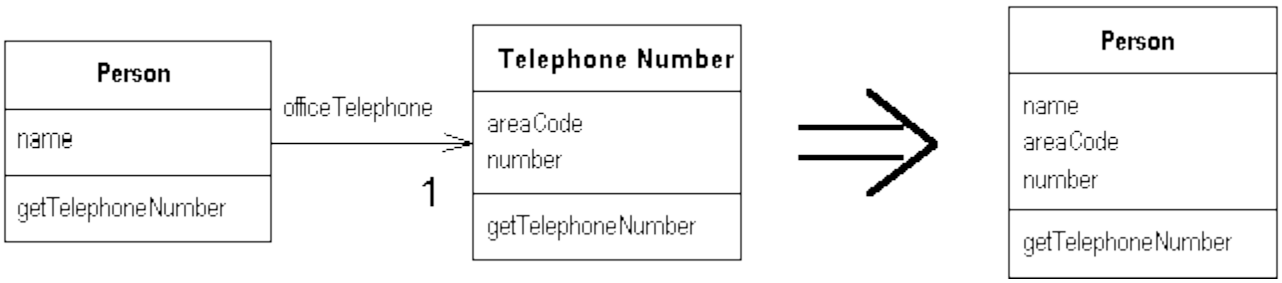
Pull Up Field/Method

You have methods with identical results on subclasses.
Move them to the superclass.



Inline Class

A class isn't doing very much.
Move all its features into another class and delete it.



Inline Method

A method's body is just as clear as its name.
Put the method's body into the body of its callers and remove the method.

```
int getRating() {
        return (moreThanFiveLateDeliveries()) ? 2 : 1;
}

boolean moreThanFiveLateDeliveries() {
        return _numberOfLateDeliveries > 5;
}
```

```
int getRating() {
        return (_numberOfLateDeliveries > 5) ? 2 : 1;
}
```

Inline Temp

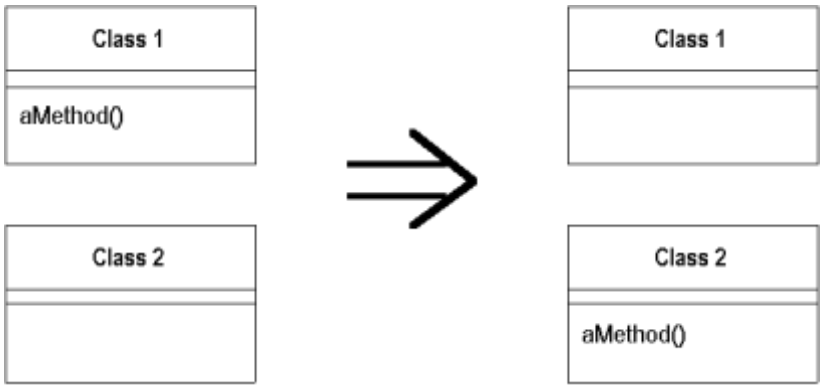
You have a temp that is assigned to once with a simple expression, and the temp is getting in the way of other refactorings.
Replace all references to that temp with the expression.

```
double basePrice = anOrder.basePrice();
return (basePrice > 1000)
```

```
return (anOrder.basePrice() > 1000)
```

Move Method

A method is, or will be, using or used by more features of another class than the class on which it is defined.
Create a new method with a similar body in the class it uses most. Either turn the old method into a simple delegation, or remove it altogether.



Code Smell	Donde se Encuentra (Class.Method)	Refactoring
Long Method		
Primitive Obsession		
Long Parameter List		
Dataclums		
Switch Statements		
Refused Bequest		
Divergent Change		
Lazy Class		
Duplicate Code		
Speculative Generality		
Feature Envy		
Indecent Exposure (Inappropriate Intimacy)		
Comments		

MANUAL REFACTORINGS

Extract Method – Step by Step

- Create a new method, and name it after the intention of the method (name it by what it does, not by how it does it).
  If the code you want to extract is very simple, such as a single message or function call, you should extract it if the name of the new method will reveal the intention of the code in a better way. If you can't come up with a more meaningful name, don't extract the code.

- Copy the extracted code from the source method into the new target method.

- Scan the extracted code for references to any variables that are local in scope to the source method. These are local variables and parameters to the method.

- See whether any temporary variables are used only within this extracted code. If so, declare them in the target method as temporary variables.

- Look to see whether any of these local-scope variables are modified by the extracted code. If one variable is modified, see whether you can treat the extracted code as a query and assign the result to the variable concerned.

- Pass into the target method as parameters local-scope variables that are read from the extracted code.

- Compile when you have dealt with all the locally-scoped variables.

- Replace the extracted code in the source method with a call to the target method.
  If you have moved any temporary variables over to the target method, look to see whether they were declared outside of the extracted code. If so, you can now remove the declaration.

- Compile and test.

Inverse Refactorings

A continuación se muestra una lista de refáctorings. Al lado de cada refactoring, escribe el nombre del refactoring que revierte su cambio.

Refactoring	Inverse
Collapse Hierarchy	
Extract Method	
Inline Class	
Rename Method	
Inline Temp	
Pull Up Method	

Smells to Refactorings
Quick Reference Guide

Smell	Description	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 refa ctor 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]
		Substitute 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 200]
		Extract Class [F 149]
		Hide Delegate [F 157]
		Replace Inheritance with Delegation [F 352]

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

Adapted from http://industriallogic.com/papers/smellstorefactorings.pdf

Smells to Refactorings
Quick Reference Guide

Smell	Description	Refactoring
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]
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]
		Compose Method [K 123]
		Introduce Parameter Object [F 295]
		Move Accumulation to Collecting Parameter [K 313]
		Move Accumulation to Visitor [K 320]
		Decompose Conditional [F 238]
		Preserve Whole Object [F 288]
		Replace Conditional Dispatcher with Command [K 191]
		Replace Conditional Logic with Strategy [K 129]
		Replace Method with Method Object [F 135]
		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]
		Introduce Parameter Object [F 295]
		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]
		Extract Method [F 110]
		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]
		Inline Method [F 117]
		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]
		Move Field [F 146]

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

Adapted from http://industriallogic.com/papers/smellstorefactorings.pdf

Smells to Refactorings
Quick Reference Guide

Smell	Description	Refactoring
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]
		Encapsulate Composite with Builder [K 96]
		Introduce Parameter Object [F 295]
		Extract Class [F 149]
		Move Embellishment to Decorator [K 144]
		Replace Conditional Logic with Strategy [K 129]
		Replace Implicit Language with Interpreter [K 269]
		Replace Implicit Tree with Composite [K 178]
		Replace State-Altering Conditionals with State [K 166]
		Replace Type Code with Class [F 218, K 286]
		Replace Type Code with State/Strategy [F 227]
		Replace Type Code with Subclasses [F 223]
		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]
		Push Down Method [F 322]
		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]
		Move Field [F 146]
		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]
		Rename Method [F 273]
		Remove Parameter [F 277]
		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]
		Replace Conditional Dispatcher with Command [K 191]
		Replace Conditional with Polymorphism [F 255]
		Replace Type Code with Subclasses [F 223]
		Replace Type Code with State/Strategy [F 227]
		Replace Parameter with Explicit Methods [F 285]
		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]
		Introduce Null Object [F 260, K 301]

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

Adapted from http://industriallogic.com/papers/smellstorefactorings.pdf