Summary of bad smells (from *Refactoring* by Martin Fowler)

- **Duplicated Code:** The same code structure in two or more places is a good sign that the code needs to be refactored: if you need to make a change in one place, you'll probably need to change the other one as well, but you might miss it
- **Long Method:** Long methods should be decomposed for clarity and ease of maintenance
- **Large Class:** Classes that are trying to do too much often have large numbers of instance variables. Sometimes groups of variables can be clumped together. Sometimes they are only used occasionally. Over-large classes can also suffer from code duplication.
- **Long Parameter List:** Long parameter lists are hard to understand. You don't need to pass in everything a method needs, just enough so it can find all it needs.
- **Divergent Change:** Software should be structured for ease of change. If one class is changed in different ways for different reasons, it may be worth splitting the class in two so each one relates to a particular kind of change.
- **Shotgun Surgery:** If a type of program change requires lots of little code changes in various different classes, it may be hard to find all the right places that do need changing. Maybe the p[laces that are affected should all be brought together into one class.
- **Feature Envy:** This is where a method on one class seems more interested in the attributes (usually data) of another class than in its own class. Maybe the method would be happier in the other class.
- **Data Clumps:** Sometimes you see the same bunch of data items together in various places: fields in a couple of classes, parameters to methods, local data. Maybe they should be grouped together into a little class.
- **Primitive Obsession:** Sometimes it's worth turning a primitive data type into a lightweight class to make it clear what it is for and what sort of operations are allowed on it (eg creating a date class rather than using a couple of integers)
- **Switch Statements:** Switch statements tend to cause duplication. You often find similar switch statements scattered through the program in several places. If a new data value is added to the range, you have to check all the various switch statements. Maybe classes and polymorphism would be more appropriate.
- **Parallel Inheritance Hierarchies:** In this case, whenever you make a subclass of one class, you have to make a subclass of another one to match.
- **Lazy Class:** Classes that are not doing much useful work should be eliminated
- **Speculative Generality: Often** methods or classes are designed to do things that in fact are not required. The dead-wood should probably be removed.
- **Temporary Field:** It can be confusing when some of the member variables in a class are only used occasionally
- **Message Chains:** A client asks one object for another object, which is then asked for another object, which is then asked for another, etc. This ties the code to a particular class structure.
- **Middle Man:** Delegation is often useful, but sometimes it can go too far. If a class is acting as a delegate, but is performing no useful extra work, it may be possible to remove it from the hierarchy.
- **Inappropriate Intimacy:** This is where classes seem to spend too much time delving into each other's private parts. Time to throw a bucket of cold water over them!
- **Alternative classes with different interfaces:** Classes that do similar things, but have different names, should be modified to share a common protocol
- **Incomplete Library Class:** It's bad form to modify the code in a library, but sometimes they don't do all they should do
- **Data Class:** Classes that just have data fields, and access methods, but no real behaviour. If the data is public, make it private!
- **Refused Bequest:** If a subclass doesn't want or need all of the behaviour of its base class, maybe the class hierarchy is wrong.
- **Comments:** If the comments are present in the code because the code is bad, improve the code.

Code Smell Cheat Sheet

SYMPTOMS	CODE SMELL	NOTES
• Duplicated codes • Same code structure or expression in more than one place	**Duplicated Code**	
• A long method	**Long Method**	- Long methods are bad because long procedures are hard to understand. - Name a small method after the intention of the code, not implementation details. Small methods should have good names that reveal the intention of the code. - "The key here is not method length but the semantic distance between what the method does and how it does it."
• A large class • A class that's trying to do too much • A class with too many instance variables	**Large Class**	
• A long parameter list	**Long Parameter List**	- Long parameter lists are bad because they are hard to understand and use and can easily become inconsistent.
• A class is commonly changed in different ways for different reasons • A class suffers many kinds of changes.	**Divergent Code**	What we want are: - "When we make a change we want to be able to jump to a single clear point in the system and make the change." - Each object is changed only as a result of one kind of change. - Ideally, have a one-to-one link between common changes and classes.

Code Smell Cheat Sheet

SYMPTOMS	CODE SMELL	NOTES
• A change requires alerting many classes • When you want to make a kind of change, you need to make a lot of little changes to a lot of different classes.	**Shotgun Surgery**	"When the changes are all over the place, they are hard to find, and it's easy to miss an important change."
• A method seems more interested in another class than the one it actually is in. • A method does not leverage data or methods from the class it belongs to. Instead, it requires lots of data or methods from a different class.	**Feature Envy**	
• Three or four data items clump together in lots of places such as fields in a couple of classes or parameters in many method signatures.	**Data Clumps**	- "Bunches of data that hang around together really ought to be made into their own object."
• Using multiple primitive data types to represent a concept such as using three integers to represent a date	**Primitive Obsession**	- Don't be afraid to use small objects for small tasks such as money classes that combine number and currency
• A switch statement that is duplicated in multiple, different places. If you add a new clause to the switch, you have to painstakingly find each scattered switch statement and change it.	**Switch Statements**	- "One of the most obvious symptoms of object-oriented code is its comparative lack of switch (or case) statements." - Consider polymorphism when you see a switch statement.

Code Smell Cheat Sheet

SYMPTOMS	CODE SMELL	NOTES
• Parallel inheritance hierarchies • Every time you make a subclass of one class, you also have to make a subclass of another. • Prefixes of the class names in one hierarchy are the same as the prefixes in another hierarchy.	**Parallel Inheritance Hierarchies**	- "Parallel inheritance hierarchies is really a special case of shotgun surgery."
• A class that isn't doing enough to pay for itself.	**Lazy Class**	- "Each class you create costs money to maintain and understand."
• The only users of a method or class are test cases.	**Speculative Generality**	- This happens when people thought they need a method or class for a future requirement but it turned out they didn't really need it.
• An instance variable is set only in certain circumstances.	**Temporary Field**	- "Such code is difficult to understand, because you expect an object to need all of its variables. Trying to understand why a variable is there when it doesn't seem to be used can drive you nuts."
• A method calling a different method which calls a different method which calls a different method ...	**Message Chains**	- A message chain couples a client of the method to the structure of the navigation. Any change to the intermediate relationships requires the client to have to change.

Code Smell Cheat Sheet

SYMPTOMS	CODE SMELL	NOTES
• A class with lots of methods delegated to this other class	**Middle Man**	
• Classes delving in each others' private parts too much	**Inappropriate Intimacy**	
• Methods that do the same thing but have different signatures for what they do	**Alternative Classes with Different Interfaces**	
• Trying to modify a library class to do something you'd like it to do	**Incomplete Library Class**	
• Classes have nothing but fields and getters and setters for these fields. • Classes act as dumb data holders and are manipulated in far too much detail by other classes.	**Data Class**	- "Data classes are like children. They are okay as a starting point, but to participate as a grownup object, they need to take some responsibility."

Code Smell Cheat Sheet

SYMPTOMS	CODE SMELL	NOTES
• A subclass only uses a few methods or data given by the superclass (Unless it's causing confusion and problems, this smell is too faint to be worth cleaning.) • A subclass does not want to support the interface of the superclass.	**Refused Bequest**	
• Using comments to explain what a block of code does	**Comments**	- Use comments to indicate areas you are not sure and to say why you did something. - "When you feel the need to write a comment, first try to refactor the code so that any comment becomes superfluous."

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] 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]
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.