# Refactoring (code smells)

From the [Wiki](http://en.wikipedia.org/wiki/Code_smell):

A code smell is any symptom in the source code of a program that possibly indicates a deeper problem. Code smells are usually not bugs — they are not technically incorrect and don't currently prevent the program from functioning. Instead, they indicate weaknesses in design that may be slowing down development or increasing the risk of bugs or failures in the future ... a code smell is a driver for refactoring.

We list a few kinds of smells here here:

**Duplicated/similar code**: see the [reading](https://open.appacademy.io/learn/swe-campus-hybrid/ruby/dry--don-t-repeat-yourself-) on DRY code.

**Long methods**: see the [reading](https://open.appacademy.io/learn/swe-campus-hybrid/ruby/method-decomposition---design) on method decomposition.

**Too many parameters**: The more parameters a method has, the greater the chance that it is too coupled to code that invokes it. Many parameters may also be a sign of an excessively complex method.

**Data clump**: if you see a group of data always being passed around together, this is usually a good candidate for refactoring out into an object. For example start\_date and end\_date might be pulled out into a date\_range object.

**Long method chains**: long method chains can often be a sign that you're violating the [Law of Demeter](http://en.wikipedia.org/wiki/Law_Of_Demeter): only talk to your "neighbors", only use one dot.

bicycle.front\_tire.rotate

bicycle.rear\_tire.rotate

# vs

bicycle.rotate\_tires

The idea is that instead of reaching into objects (like the bicycle) and calling methods on their "internals" (front\_tire and rear\_tire), we should create a method that will take care of this for us (rotate\_tires). This helps us organize our code, and prevents us from becoming too dependent on the internal structure of other objects.

LoD has disadvantages (see the wiki article); if taken too literally you end up with overly wide interfaces. However, the longer your method chains get, the more likely you should apply LoD.

**Indecent Exposure**: Classes should share the bare minimum interface with the outside world. If you don't have a compelling reason to make a method or variable public, hide it. Other classes will interact with the public surface of the class; any change to the public surface will necessitate changes to interacting classes. Minimizing exposure will better enable change.

[From the wiki page:](http://en.wikipedia.org/wiki/Coupling_(computer_programming)) "Coupling (dependency) is the degree to which each program module relies on each one of the other modules."

Indecent exposure may result in coupling that is too tight. The more extensive your classes' interface, the more tightly other classes can latch on to it.

A complicated interface can also signal a class that is doing too much. Probably there's an opportunity to break out responsibilities to other classes.

We want to minimize the amount of coupling between our classes and objects. To understand how tightly coupled your classes are, ask yourself if you changed the internals of one class, would you have to change things in the other? If you do, you've likely coupled the two classes too tightly.

**Speculative Generality**: Follow the principle of [YAGNI](http://en.wikipedia.org/wiki/You_ain't_gonna_need_it) ('You ain't gonna need it'). New devs often waste time thinking of all the ways they might "generalize" their code so that it's easy to make this change or that change, so that they can support all sorts of new features easily, etc., etc. This can result in a very complicated design, before any business need for those features even exists.

More importantly, it is very, very hard to get a speculative design like this right. Good design happens when there are clear requirements and concrete examples to think about; it's hard to make a perfect design for some hazy, poorly understood future. Don't solve abstract problems: wait until you have a concrete problem.

**God object**: A god object is one that is very tightly connected to all the other objects in the system. Good OO design results in classes that are lightly coupled. A good class delegates responsibility as necessary to other objects; it shouldn't need to know everything about what every other object is doing (omniscience), and it shouldn't micromanage how other objects manage their responsibilities. Nothing in your program usually needs to even know about the existence of everything else.

**Dead code**: don't leave commented-out (or otherwise unused) code in your code base. This is why we have version control (we'll learn about git soon!). Clean up after yourself!

## Clean Code

When push comes to shove, sometimes you need to make compromises; when deadlines hit, we all ship code we aren't 100% proud of. Still, endeavor to write clean code. Pay attention to style. Pay attention to code organization. Pay attention to code smells. Refactor often and aggressively. Bask in the beauty of clean code. Take pride in it.

## Further Reading

* Enjoy watching Ben Orenstein's classic talk: [Refactoring: From Good to Great](http://confreaks.tv/videos/aloharuby2012-refactoring-from-good-to-great).