1. Code Smells Within Classes

Comments There's a fine line between comments that illuminate and comments that obscure. Are the comments necessary? Do they explain "why" and not "what"? Can you refactor the code so the comments aren't required? And remember, you're writing comments for people, not machines.

Long Method All other things being equal, a shorter method is easier to read, easier to understand, and easier to troubleshoot. Refactor long methods into smaller methods if you can.

Long Parameter List The more parameters a method has, the more complex it is. Limit the number of parameters you need in a given method, or use an object to combine the parameters.

Duplicated code Duplicated code is the bane of software development. Stamp out duplication whenever possible. You should always be on the lookout for more subtle cases of near-duplication, too. Don't Repeat Yourself!

Conditional Complexity Watch out for large conditional logic blocks, particularly blocks that tend to grow larger or change significantly over time. Consider alternative object-oriented approaches such as decorator, strategy, or state.

Combinitorial Explosion You have lots of code that does *almost* the same thing.. but with tiny variations in data or behavior. This can be difficult to refactor-- perhaps using generics or an interpreter?

Large Class Large classes, like long methods, are difficult to read, understand, and troubleshoot. Does the class contain too many responsibilities? Can the large class be restructured or broken into smaller classes?

Type Embedded in Name Avoid placing types in method names; it's not only redundant, but it forces you to change the name if the type changes.

Uncommunicative Name Does the name of the method succinctly describe what that method does? Could you read the method's name to another developer and have them explain to you what it does? If not, rename it or rewrite it.

Inconsistent Names Pick a set of standard terminology and stick to it throughout your methods. For example, if you have Open(), you should probably have Close().

Dead Code Ruthlessly delete code that isn't being used. That's why we have source control systems!

Speculative Generality Write code to solve today's problems, and worry about tomorrow's problems when they actually materialize. Everyone loses in the "what if.." school of design. You (Probably) Aren't Gonna Need It.

Oddball Solution There should only be one way of solving the same problem in your code. If you find an oddball solution, it could be a case of poorly duplicated code-- or it could be an argument for the adapter model, if you really need multiple solutions to the same problem.

Temporary Field Watch out for objects that contain a lot of optional or unnecessary fields. If you're passing an object as a parameter to a method, make sure that you're using all of it and not cherry-picking single fields. This also applies inside a method body that has too many temp variables. You can refactor the temp variables into private class methods to clean it up.

2. Code Smells Between Classes (or Functions)

Alternative Classes with Different Interfaces If two classes are similar on the inside, but different on the outside, perhaps they can be modified to share a common interface.

Primitive Obsession Don't use a gaggle of primitive data type variables as a poor man's substitute for a class. If your data type is sufficiently complex, write a class to represent it. A common smell of this type is conditional that check for nils all over the place. Use the Null Object pattern to fix this.

Data Class Avoid classes that passively store data. Classes should contain data *and* methods to operate on that data, too.

Data Clumps If you always see the same data hanging around together, maybe it belongs together. Consider rolling the related data up into a larger class. An example is start_date and end_date. These two often go together and can be refactored into a DateRange class.

Refused Bequest If you inherit from a class, but never use any of the inherited functionality, should you really be using inheritance?

Inappropriate Intimacy Watch out for classes that spend too much time together, or classes that interface in inappropriate ways. Classes should know as little as possible about each other.

Indecent Exposure Beware of classes that unnecessarily expose their internals. Aggressively refactor classes to minimize their public surface. You should have a compelling reason for every item you make public. If you don't, hide it.

Feature Envy Methods that make extensive use of another class may belong in another class. Consider moving this method to the class it is so envious of. For example, if Report class is checking if the placed_at field of a an Order is between a date range.This functionality should be done inside the Order class, not Report. Can refactor the logic into a public function on Order called placed_between?. This code smell also applies between functions. For example, if one method is responsible for generating a key format, knowledge of that key format should only be in that

function, and not leak to other functions.

Lazy Class Classes should pull their weight. Every additional class increases the complexity of a project. If you have a class that isn't doing enough to pay for itself, can it be collapsed or combined into another class?

Message Chains Watch out for long sequences of method calls or temporary variables to get routine data. Intermediaries are dependencies in disguise.

Middle Man If a class is delegating all its work, why does it exist? Cut out the middleman. Beware classes that are merely wrappers over other classes or existing functionality in the framework.

Divergent Change If, over time, you make changes to a class that touch completely different parts of the class, it may contain too much unrelated functionality. Consider isolating the parts that changed in another class.

Shotgun Surgery If a change in one class requires cascading changes in several related classes, consider refactoring so that the changes are limited to a single class. An example is instead of using a payment gateway gem class directly from your classes, create another class called PaymentGateway, and this is the only class that uses the gem class and all other classes that want to use payments must go through PaymentGateway. That way, when you change payment providers, you only have to change one class.

Parallel Inheritance Hierarchies Every time you make a subclass of one class, you must also make a subclass of another. Consider folding the hierarchy into a single class.

Incomplete Library Class We need a method that's missing from the library, but we're unwilling or unable to change the library to include the method. The method ends up tacked on to some other class. If you can't modify the library, consider isolating the method.

Solution Sprawl If it takes five classes to do anything useful, you might have solution sprawl. Consider simplifying and consolidating your design.

Piggybank If a piece of data, or variable, is being passed through to many classes and then only used in one of the classes, its a code smell. You should move the data closer to where its being used. That is, find a way to instantiate the data in the classes where its actually being used.

3. Miscellaneous

Complicated Data If the data, either json structure or computer science data structure, is complicated or not intent revealing, the code that operates on it will also be complicated and not intent revealing. Consider refactoring the data structure to a clearer format and the code will naturally follow.