## **Refactoring: Improving the Design of Existing Code**

## **Chapter 3: Bad Smells in Code**

***If it stinks, change it.*** If the code smells bad, refactor it.

1. The top stinker is **duplicated code**. To fix this, simply use Extract Method and call the code from both methods in the same class.
2. **Long methods and parameters** stink. Extracting the method can make it shorter, more maintainable, and easier to understand. Comments often signal the code clumps that need to be extracted.
3. Another code that stinks are those code having **large class**. It shows too many instance variables which makes the code messy and difficult to understand. The solution for this one is to eliminate redundancy in the class itself.
4. With **divergent change,** it occurs when once class is commonly changed in different ways for different reasons. To clean this up, you use Extract Class to put them all together.
5. **Shotgun surgery** is when making one change requires many small changes in different classes. Move Method and Move Field can simplify this by consolidating changes in a single class. Envy often centers on data.
6. With this **feature envy**, you can use Move Method, or Extract Method.
7. **Data items** tends to be like children. Bunches of data that hang around together really ought to be made into their own subject.
8. **Primitive types** are your building blocks. One of the valuable things about objects is that they blur or even break the line between primitive and larger classes. You can solve this issue by using Replace Data Value with Object or Relace Type Code with Class.
9. The problem with **switch statements** is duplication, which may be scattered throughout a program. The challenge is to decide where the polymorphism should take place. One solution is to use Extract Method to remove the switch statement, and then Move Method to place it in the class that requires polymorphism.
10. 10. Now, **Parallel inheritance hierarchies** is really a special case of shotgun surgery. Every time you make a subclass of one class, you also have to make a subclass of another. The general strategy for eliminating the duplication is to make sure that instances of one hierarchy refer to instances of the other. If you use Move Method and Move Field, the hierarch on the referring class disappears.
11. A **(lazy) class** that isn’t' doing enough to pay for itself should be eliminated by using Collapse Hierarchy.
12. For this kind of stink, a **speculative generality** often results to difficulty of understanding and maintaining. The machinery just gets in the way, so get rid of it. This can be spotted when the only users of a method or class are test cases.
13. A common case of **temporary field** occurs when a complicated algorithm needs several variables which makes it plain confusing. With this you can use Extract Class with the variables, and the methods that require them.
14. **Message chains** occur when a client repeatedly requests objects from different objects in a chain. Any change to the intermediate relationships causes the client to have change. The move to use here is Hide Delegate or Extract Method then Move Method.
15. One of the prime features of objects is encapsulation—hiding internal details from the rest of the world. Encapsulation often comes with delegation. However, this can go too far. You can find half the methods are delegating to this other class. After a while it is time to use Remove **Middle Man** and talk to the object that really knows what's going on.
16. **Overintimate** classes need to be broken up as lovers were in ancient days. Use Move Method and Move Field to separate the pieces to reduce the intimacy. Inheritance often can lead to overintimacy.
17. **Data classes** are simply containers of fields and their corresponding getters and setters. However, if these classes are manipulated extensively by other classes, it is recommended to apply encapsulation.
18. **Refused bequest** is strong when a subclass is reusing behavior but doesn't want to support the interface of the superclass. In such cases, it's recommended to use Replace Inheritance with Delegation to eliminate the hierarchy.
19. Sometimes, **comments** lead us to bad code which makes it stink. If you need a comment to explain what a block of code does, try Extract Method. If other issues arise, you can use other method such as Rename Method, and Introduce Assertion. A good time to use a comment is when you don't know what to do.

## **Chapter 4: Building Tests**

The essential precondition for refactoring is to have solid tests. Ensure that all tests are fully automated and check their own results. A suite of tests is a powerful bug detector that reduces the time it takes to find bugs. Writing tests requires extra code, but when tests are automated, they can be quite fun to write, unlike manual tests that can be quite boring. Before adding a feature to a system, begin by writing tests to determine what needs to be done to add the function. It is useful to differentiate between failures and errors because they tend to manifest differently, and the debugging process is different.

The **JUnit Testing Framework**, an open-source testing framework that allows you to perform all the essential tasks needed for testing, is used for **unit tests**. Each test class works within a single package and tests the interfaces to other packages. On the other hand, **functional** tests are written to ensure that the software works as a whole, providing quality assurance to the customer without regard for programmer productivity. When you receive a bug report, start by writing a unit test that exposes the bug to ensure that a similar bug doesn't bypass your unit tests again. The JUnit framework is designed for writing unit tests, while functional tests are often performed with other tools.

We should add more tests, testing each one for any conditions that might cause the class to fail. We are testing to find bugs. It is better to write and run incomplete tests than not to run complete tests. A key trick with tests is to look for boundary conditions where things might go wrong and focus your tests there. Your tests will not find every bug, but as you refactor, you will understand the program better and thus find more bugs.