
THE UNIVERSITY OF
WAIKATO
*Te Whare Wānanga o Waikato*

DEPARTMENT OF
COMPUTER SCIENCE
*Te Tari Rorohiko*

# 2024

Contributors

J. Turner

V. Moxham-Bettridge

J. Bowen

# DETECTING CODE SMELLS

In the last session we introduced data structures as one of the key concepts in computer science theory. This week we move beyond data structures to another key concept, software quality. One way of detecting quality is to avoid "code smells" which define common issues in low quality programs.

## Why is Code Quality important?

When we talk about code quality, we are often referring to software architecture. Software architecture refers to the **structure** of a system, as well as the **methods** for creating the structures and systems. Essentially, it is the organisation of how things in a large complex piece of software fit together. Software architecture is one of the fundamental building blocks of Software Engineering.

Alongside good software architecture, comes code quality. There are a few basics to ensure that you follow the best standards, these include: code structure, naming conventions, comments and documentation, all of which will be covered today.

Often when discussing software architecture and code quality, students ask "but why?". One of the main reasons behind this is a lack of understanding of the complexity of managing large software projects. When we write code the two biggest lies we tell ourselves is that we will remember what it does and that we will fix it later. However, neither of these are true!

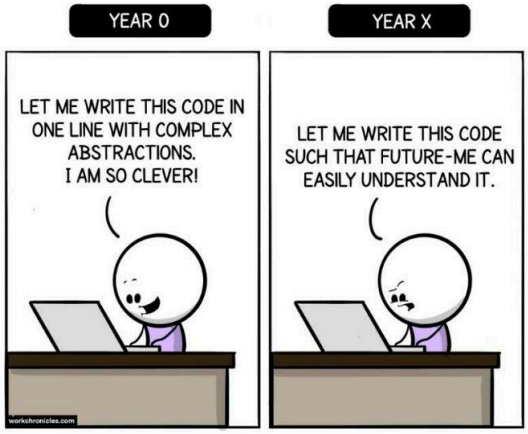


*Figure 45: New vs. Experienced Developers*

At university, a student writing an average 3rd year web assignment will write about 500 lines of code in total, usually across 3-4 files. However, in the real world programs have much larger

code bases. For example, the software to control a Boeing 787 has 6.5 million lines of code behind its avionics and online support systems. Similarly, the Large Hadron Collider uses 50 million lines of code. If we combined all Google services we would have 2 billion lines of code!

Hopefully it is obvious that not all of these code bases are maintained by one person, rather the commercial development is done by teams. This means that other people have to understand what your code does. Furthermore, what you write will involve using external libraries and APIs, so you need to be able to understand what other people's code does. This is where structure, naming conventions, comments and documentation become hugely important, as they help to reduce the likelihood of these problems occurring.

## Code Quality

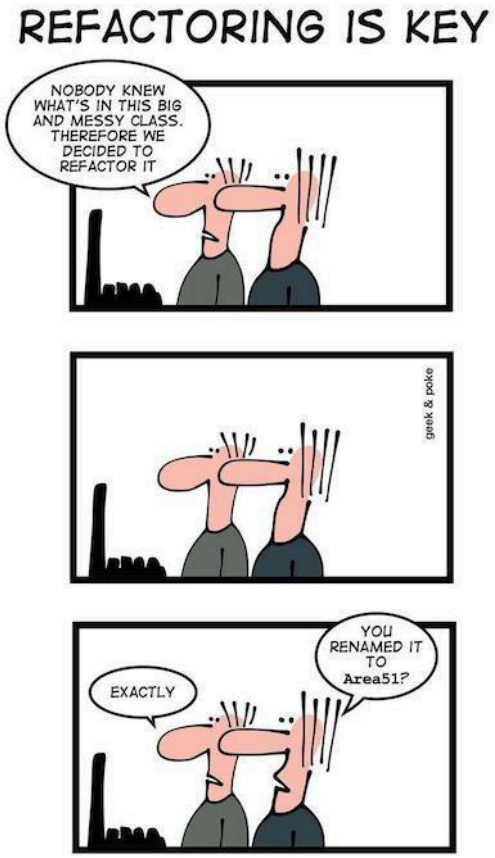


*Figure 46: Refactoring*

Good structure is defined by following best programming practices for your chosen programming language. For example, if we consider the Java programming language our code should be object-oriented with well constructed class hierarchies such that code reusability is prioritised. Modularity is important to ensure that the code is broken down into the appropriate

structures. We must follow a consistent indentation format and refactor (update or modify the code) regularly to improve code quality as modifications are made.

Following naming conventions includes using informative class, method, and variable names. We must follow the conventions of the language and be consistent to make code easier to read for others. For example, in Java all classes start with an uppercase letter while methods start with a lowercase letter. If a class started with a lowercase letter the other programmers on your team would be confused by your code as they would expect this to be a method not a class, thus making your code less readable and understandable.

Documentation can take two forms, comments directly in the code as well as reports and other documents that define the code. The comments you include in the code should be informative, that is, do not comment for the sake of it as we only use comments to communicate useful information. Other documentation includes architectural design, technical and user documentation, however, these concepts are outside the scope of this session.

## What are Code Smells?


*Figure 47: Code Smells*

Kent Beck popularised the term **Code Smells** in the 1990s as a part of the extreme programming process. It became more commonly used when Martin Fowler popularised the practice of code refactoring. Code Smells themselves are not bugs or errors, rather these occur when we do not follow the best programming practices. To quote Fowler: *"Any fool can write code that a computer can understand, good programmers write code that every human can understand"*.

What Fowler is getting at here is that code smells do not mean that the software does not work, in fact it may even work correctly, the problem is that the code smells may slow down the computation, increase the risk of failure or make the program vulnerable to bugs in the future. The good news is that code smells, like the name implies, can be "sniffed out" earlier in the

development process, thus reducing the cost required to fix them and improving quality. In fact most code smells have common solutions that improve the overall quality of the program. However, much like with data structures, knowing when and how to apply these is a matter of practice.

## Types of Code Smells

Code smells can loosely be divided into five categories. Next we explore each of these categories with one example for each (the full definitions can be found at https://refactoring.guru/refactoring/smells).

### Bloaters

**Bloaters** are code, methods and classes that have increased to such gargantuan (massive) proportions that they are hard to work with. Usually these smells do not crop up right away, rather they accumulate over time as the program evolves (and especially when nobody makes an effort to eradicate them).

For example, consider the **Large Class** code smell, this is when a class performs so many different functions it's unclear what its original objective is. This makes it difficult to update or modify the functionality of the class.

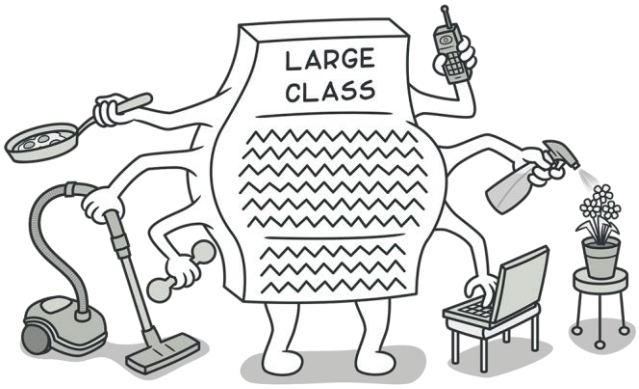


*Figure 48: Large Class Code Smell*

We fix this code smell by breaking down the functionality into smaller classes, subclasses and/or interfaces. This means that each of the desired functionality is better contained within each of these classes.

### Object-Orientation Abusers

These code smells do exactly what the name suggests, they break the fundamental principles of object-oriented programming. That is, they do not adhere to the basics of abstraction (hiding

details), inheritance (relationships between objects), encapsulation (grouping together common elements) and polymorphism (objects can be treated as the same even when they're not!).

An example of an object-oriented abuser is the **Switch Statements** code smell. This is when you have one massive switch statement with so many different cases it is difficult to follow, and they often give very different results. There are multiple ways to fix this but usually it involves applying polymorphism and inheritance correctly.

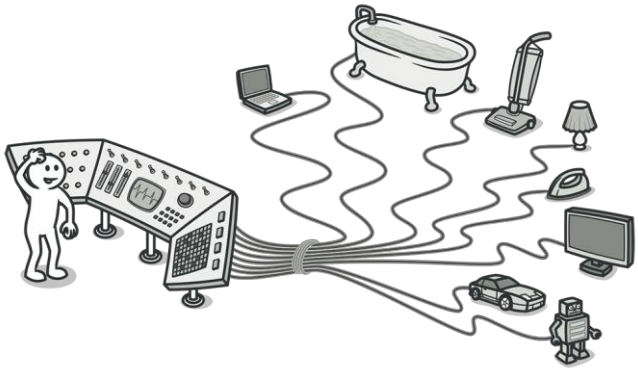

*Figure 49: Switch Statements Code Smell*

## Change Preventers

Again, not a particularly creative descriptor here for **change preventer** code smells. These group together the smells that make it difficult to update existing code because they trigger changes in other parts of the code. As a result the program becomes much more complicated and expensive.

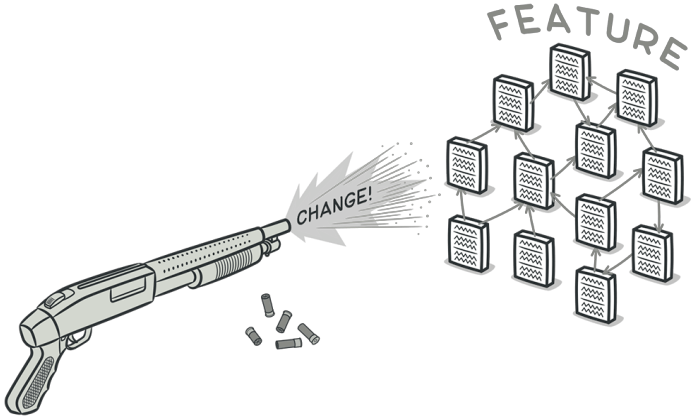


*Figure 50: Shotgun Surgery Code Smell*

An example of this code smell is **Shotgun Surgery**. This means that when you need to update or make a change to one feature in one class it requires you to make many changes to other classes. This is usually down to poor use of encapsulation and can be fixed by applying the principle correctly.

## Dispensables

**Dispensables** are code smells that are pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand. In other words, they exist but contribute nothing of value to the code base.

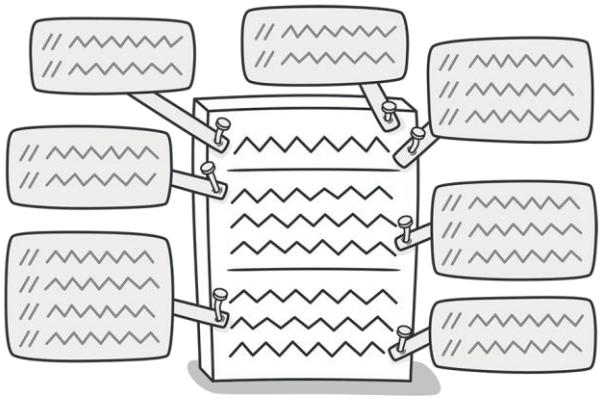

*Figure 51: Comments Code Smell*

An example of a dispensable is the **Comments** code smell. To be clear, this does not mean that we should not use comments (as specified earlier these are important for code quality!) rather, we need to use comments *appropriately*. This code smell arises when the author realises their code is unreadable or overly complex and they try to fix this by giving overly descriptive comments. To fix this issue, simply refactor the code and break down the functionality further such that you do not need this level of commenting.

## Couplers

Lastly, **Coupler** code smells contribute to the excessive coupling between classes or show what happens if coupling is replaced by excessive delegation. In other words, classes are so tightly linked that one cannot exist without the other. Much like all of the code smells here this is a clear violation of the principles of object-oriented programming.

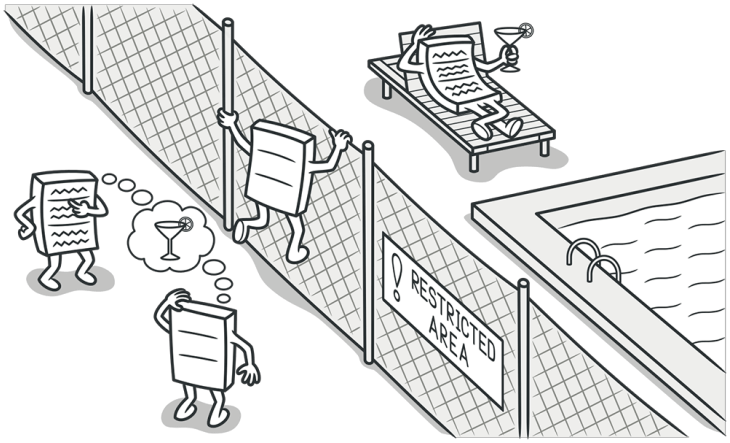

*Figure 52: Couplers Code Smell*

An example of a coupler is **Feature Envy**. This occurs when a method accesses the data of another object more than its own data. There are multiple solutions for this, but as with our other examples, they involve applying object-oriented programming correctly.

## Detecting Code Smells

As stated previously, detecting code smells can be a complicated task and requires extensive knowledge of all code smells. Before giving your own code to work with, let's work through an example. Consider the following code:

```
01   public class Example {
02       public static void main(String[] args) {
03           int[] numbers = {1, 2, 3, 4, 5};
04           for (int i = 0; i < numbers.length; i++) {
05               if (i == 2) {
06                   System.out.println("Skipping index 2");
07                   continue;
08               }
09               if (i == 4) {
10                   System.out.println("Breaking loop at
                 index 4");
11                   break;
12               }
13               System.out.println(numbers[i]);
14           }
15       }
16   }
```

There are four main code smells in the example code given above. The first is **magic numbers** on lines 5 and 9 of the code. We do not know what the numbers '2' and '4' signify or why we are skipping those indexes. There is no context and therefore they may not be understood by other programmers. To fix this we would need to declare constants with meaningful names and use them instead.

The second issue is **hard-coded values**, in that the numbers array on line 3 has specific values. This makes the code inflexible as it can only ever process those numbers. A better approach would be to take user input and run the program on that provided input.

There are also **unnecessary conditionals** in the code. The if statements inside the loop are unnecessary and add complexity to the code. In particular, lines 9-12 are going to mean that we

never print out the last value in the array which is not the intended behaviour. We simply need to remove these statements as they do not achieve anything (i.e. they are dispensable).

Lastly, there are **poor variable names** in the code, although you will see most examples of for loops begin with an "i" character; this is actually bad practice because it does not describe the purpose of the variable. A better name to use would be something like "index" or "counter".

## Today's Activity

Now you have seen an example of identifying and how to remedy code smells, you will find on the Slack channel an example of some code in the Java programming language which has multiple code smells. The challenge today is for you to identify each of the code smells present in the code (you can use the information at the following link to help you, https://refactoring.guru/refactoring/smells).

For every code smell that you identify, list what it is, why it is a problem and how you would fix the issue. You should be able to identify 8 primary issues with the code but there may be more. When you believe you have identified all 8 (or more!) issues, get a staff member to confirm your list.

## Summary

In Today's session we have given you a very quick and brief introduction to code smells. Senior developers can spot code smells quickly and efficiently, giving simple fixes to ensure the code base improves. Thus while we have given you a "taster" session and hope this will help you improve your coding skills, much more practice and knowledge is required to write really good quality code. We encourage you to explore code smells in past programs you have written and see if you can refactor your code to improve its quality.

## Advanced Exercises

If you have been able to identify the 8 core code smells, the advanced exercise for today is to refactor the code given to you in order to remove the code smells. Note that if you are not familiar with the Java programming language you are welcome to rewrite the code in a language of your choice. However, be sure to pick an object-oriented programming language!

## Useful Resources

- Code Smells: https://refactoring.guru/refactoring/smells
- 7 Code Smells You Should Know About and Avoid: https://towardsdatascience.com/7-code-smells-you-should-know-about-and-avoid-b1edf066c3a5

- Everything you need to know about Code Smells: https://www.codegrip.tech/productivity/everything-you-need-to-know-about-code-smells/
- Introduction to Java: https://www.w3schools.com/java/java_intro.asp