Principles of Good Software Design

COMP3607
Object Oriented Programming II

05-Sept-2018

Code Smells

"A surface indication that usually corresponds to a deeper problem in the software system" - Martin Fowler

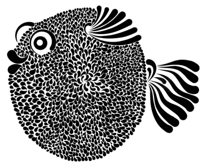
Bloaters

Bloaters are code, methods and classes that have increased to such proportions that they are hard to work with.

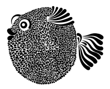
Usually these smells do not crop up right away, rather they accumulate over time as the program evolves.

For example:

- Long Method
- Large Class
- Primitive Obsession
- Long Parameter List
- Data Clumps.



Long Method



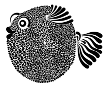
The majority of a programmer's time is spent reading code rather than writing code.

Apart from the difficulty of having to keep a lot of complex logic in mind whilst reading through a long method, it is usually a sign that the method has too many responsibilities.

Long methods make code hard to maintain and debug.

If it is not possible to view the whole method on your smartphone screen, consider breaking it up into several smaller methods, each doing one precise thing.

Large Class

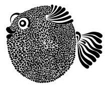


When a single class is doing too much, it often shows up as too many variables, instances and methods.

Classes usually start small but over time they get bloated as the program grows.

Programmers usually find it mentally less taxing to place a new feature in an existing class rather than create a new class for the feature.

Data Clumps



Where multiple method calls take the same set of parameters, it may be a sign that those parameters are related.

To keep the group of parameters together, it can be useful to combine them together in a class.

This can help aid organisation of code.

Object-Orientation Abusers

All these smells are incomplete or incorrect application of object-oriented programming principles.

For example:

- Switch Statements
- Temporary Field
- Refused Bequest
- Alternative Classes with Different Interfaces

Switch Statements

You have a complex `switch` operator or sequence of `if` statements.

Switch Statements (or for that matter the if-else conditions) tend to mix logic and data together, which isn’t exactly the best way to do it.

Each time the developer needs to change the logic or the data, he has a mess in his hands.

Classes should be Open for extensions, but Closed for modification (Open Closed Principle)

Temporary Field

There are times when a programmer decides to introduce fields in a class which are used only by one method.

Instead of passing it as method parameter, this is done to avoid the Long Parameter List (smell).

These fields are used only when the particular method is use under certain circumstances.

Outside of these circumstances, the fields are empty and sit idle in the class most of the time.

Refused Bequest

If a subclass uses only some of the methods and properties inherited from its parents, the hierarchy is off-kilter.

The unneeded methods may simply go unused or be redefined and give off exceptions.

The subclass ignores the functionalities of parent class (refuses to implement parent behaviour) and overrides it.

Furthermore, this will also violate the Liskov Substitution Principle as the subclass class cannot replace the parent in code without affecting the functionality.

Alternative Classes with Different Interfaces

Two different classes perform identical functions but have different interfaces (different method names, signature etc).

Such situation arises mostly when there is lack of communication between team members who end up developing similar classes, or when the developer fails to check the available classes.

Change Preventers

These smells mean that if you need to change something in one place in your code, you have to make many changes in other places too.

Program development becomes much more complicated and expensive as a result.

For example:

- Divergent Change
- Shotgun Surgery
- Parallel Inheritance Hierarchies

Shotgun Surgery

It is basically when you want to make a kind of change, you need to make a lot of little changes to a lot of different classes.

The problem is that when the changes are all over the place, they are hard to find, and it’s easy to miss an important change.

Divergent Change

It is when a class is commonly changed in different ways for different reasons and suffers many kinds of changes. So, ideally, you should have a one-to-one link between common changes and classes.

Dispensables

A dispensable is something pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand.

For example:

- Comments
- Duplicate Code
- Lazy Class
- Data Class
- Dead Code
- Speculative Generality.

Comments

Where comments are re-iterating what can be read by a developer, they may provide little value, especially when they have not been updated and no longer reflect the intent of the current code.

Rather than adding a comment to clarify a piece of code, think about whether the code can be refactored such that the comment is no longer needed.

It may be possible to provide a more descriptive name that provides the same clarity as the comment, meaning the comment can disappear, resulting in more intuitive and readable code.

Duplicate Code

When developer fixes a bug, but same symptoms are faced again later on, this can be the result of code duplication, and a bug being fixed in one occurrence of the imperfect code but not in the duplicated versions.

This poses an overhead in terms of maintenance.

When developers are not aware of the duplication, they only know to fix the occurrence they have come across.

Take care of the repeated code blocks and extract them out into a single place – don't repeat yourself!

Speculative Generality

Sometimes code is created "just in case" to support anticipated future features that never get implemented. As a result, code becomes hard to understand and support.

Couplers

All the smells in this group contribute to excessive coupling between classes or show what happens if coupling is replaced by excessive delegation.

For example

- Feature Envy
- Inappropriate Intimacy
- Message Chains
- Middle Man
- Incomplete Library Class

Feature Envy

It is when 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

Middle Man

When a class exists just to delegate to another, a developer should ask themselves what is its real purpose.

Sometimes this is the result of a refactoring task, where logic has been moved out of a class gradually, leaving an almost empty shell

```
class Pet {
  private Animal animal;

  public String getName() {
    return animal.getName();
  }

  public String getBreed() {
    return animal.getBreed();
  }

  public Owner getOwner() {
    return animal.getOwner();
  }
}
```

References

- Martin Fowler. Refactoring: Improving the Design of Existing Code