Object-Oriented Software Analysis and Design

School of Computer Science
University of Windsor

Anti-Patterns & Code Smells

- No matter how well you design your code, there will still be changes that need to be made.
- **Refactoring** helps manage this.
  - It is the process of making changes to your code so that the external behaviors of the code are not changed, but the internal structure is improved.
- This is done by making small, incremental changes to the code structure and testing frequently to make sure these changes have not altered the behavior of the code.

Anti-Patterns & Code Smells

- Ideally, refactoring changes are made when features are added, and not when the code is complete. This saves time, and makes adding features easier.
- Changes are needed in code when bad code emerges. Just like patterns emerge in design, bad code can emerge as patterns as well. These are known as **anti-patterns** or **code smells**.
- Code smells help "sniff out" what is bad in the code.

Anti-Patterns & Code Smells (contd.)

- Comments
- Duplicate Code
- Long Method
- Large Class
- Data Class
- Data Clumps
- Long Parameter List
- Divergent Class
- Shotgun Surgery
- Feature Envy
- Inappropriate Intimacy
- Message Chains
- Primitive Obsession
- Switch Statements
- Speculative Generality
- Refused Request

Comments

- One of the most common examples of bad code is comments. This code smell can occur between two extremes.
    - If **no comments** are provided in the code, it can be hard for someone else, or even the original developer returning to the code after some time away, to understand what the code is doing or should be doing.
    - On the other hand, if there are **too many comments**, they might get out of sync as the code changes. Comments can be a "deodorant" for bad smelling code. The use of a lot of comments to explain complicated design can indicate that bad design is being covered up.

Comments (contd.)

- There are other ways comments can indicate bad code, though. If the comments take on a “**reminder**” nature, so they indicate something that needs to be done, or that if a change is made to one section in the code, it needs to be updated in another method, this indicates bad code.

- Comments are very useful for documenting application programmer interfaces (APIs) in the system, for documenting the rationale for a particular choice of data structure or algorithm, and make it easier for others to understand and use the code.

Duplicate Code

- Duplicated code occurs **when blocks of code exist in the design that are similar, but have slight differences**. These blocks of code appear in multiple places in the software.

- This can be a problem, because if something needs to change, then the code needs to be updated in multiple places. This applies to adding functionalities, updating an algorithm, or fixing a bug.

- Instead, if the code only needed to be updated in one location, it is easier to implement the change. It also reduces the chance that a block of code was missed in an update or change.

- This anti-pattern relates to the D.R.Y. principle, or “Don’t Repeat Yourself”, which suggests that programs should be written so that they can perform the same tasks but with less code.

Long Method

- The long method anti-pattern suggests that code should not have long methods. Long methods can indicate that the method is more complex or has more occurring within it than it should.
- Determining if a code is too long can be difficult. There is even some debate if length of code is even a good measure of code complexity.
- Some methods, such as setting up a user interface, can be naturally long, even if focused on a specific task. Sometimes a long method is appropriate.

Long Method (contd.)

- This anti-pattern may also depend on the programming language for the system.
- Some developers suggest that having an entire method visible at once on the screen is a good guideline, with no more than around 50 lines of code.
- However, some studies show that programmers can handle methods of a couple hundred lines before they introduce bugs! Determining “how long is too long” for a method isn’t always a straightforward process!

Large Class

- The large class anti-pattern suggests that classes should not be too large.
- Large classes are commonly referred to as **God classes**, **Blob classes**, or **Black Hole classes**. They are classes that continue to grow and grow, although they typically start out as regular-sized.
- Large classes occur when more responsibilities are needed, and these classes seem like the appropriate place to put the responsibilities.
- This growth will require extensive comments to document where in the code of the class certain functionalities exist.
- Classes should have an explicit purpose to keep the class cohesive, so it does one thing well. If a functionality is not specific to the class' responsibility, it may be better to place it elsewhere.

Data Class

- The data class anti-pattern is on the opposite end of the spectrum of the large class. It occurs when there is too small of a class. These are referred to as **data class**es.

- Data classes are classes that **contain only data and no real functionality**. These classes usually have only getter and setter methods, but not much else. This indicates that it may not be a good abstraction or a necessary class.

Data Clumps

- **Data clump**s are groups of data appearing together in the instance variables of a class, or parameters to methods.
- Consider this code smell through an example. Imagine a method with integer variables x, y, and z.

```java
public void doSomething (int x, int y, int z) {
    //...
}
```

- If there are many methods in the system that perform various manipulations on the variables, it is better to have an object as the parameter, instead of using the variables as parameters over and over again. That object can be used in their place as a parameter.

Data Clumps (contd.)

```
public class Point3D {
    private int x;
    private int y;
    private int z;
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
    public int getZ() {
        return z;
    }
```

```
//contd.
    public void setX(int
         newX) {
        x = newX;
    }
    public void setY(int
         newY) {
        y = newY;
    }
    public void setZ(int
         newZ) {
        z = newZ;
    }
}
```

- Be careful not to just create data classes, however. The classes should do more than just store data.
- The original `doSomething()` method, or a useful part it, might be added to the Point3D class to avoid this.

Long Parameter List

- Another code smell is having **long parameter list**s. A method with a long parameter list can be difficult to use. They increase the chance of something going wrong.
- Methods with long parameter lists require extensive comments to explain what each of the parameters does and what it should be.
- If long parameter lists are not commented, however, then it may be necessary to look inside the implementation to see how they are used. This breaks encapsulation.
- The best solution for long parameter lists is to introduce parameter objects. A parameter object captures context.

Divergent Class

- Some code smells occur when making changes to the code itself. A divergent change is one such code smell. It occurs when you have to change a class in many different ways, for many different reasons.

- This relates to the large class code smell, where a large class has many different responsibilities. **Poor separation of concerns** is therefore a common cause of divergent change.

- Classes should have only one specific purpose. This reduces the number of reasons the code would need to change, and reduce the variety of changes needed to be implemented.

- Separation of concerns resolves two code smells—large class and divergent change.

Shotgun Surgery

- **Shotgun surgery** is a code smell that occurs when a change needs to be made to one requirement, and a numerous classes all over the design need to be touched to make that one change. In good design, a small change is ideally localized to one or two places (although this is not always possible).
- This is a commonly occurring code smell. It can happen if you are trying to add a feature, adjust code, fix bugs, or change algorithms.

Shotgun Surgery (contd.)

- Modular code is not always an option, however. Some changes require shotgun surgery no matter how well designed the code.
- The shotgun surgery smell is normally resolved by moving methods around.
- If a change in one place leads to changes in other places, then this indicates that the methods are related in some way. Perhaps there is a better way to organize them. If not, then you may have to deal with it as it is.

Feature Envy

- **Feature envy** is a code smell that occurs when there is a method that is more interested in the details of a class other than the one it is in.
- If two methods or classes are always talking to one another and seem as if they should be together, then chances are this is true.

Inappropriate Intimacy

- **Inappropriate intimacy** is a code smell that occurs when two classes depend too much on one another through two-way communication.
- If two classes are closely coupled, so a method in one class calls methods of the other, and vice versa, then it is likely necessary to remove this cycle.
- Methods should be factored out so both classes use another class. At the very least, this makes communication one-way, and should create looser coupling.
- Cycles are not always necessarily a bad thing. Sometimes, they are necessary. But, if there's a way to make the design simpler and easier to understand, then it is a good solution.

Message Chains

- A **message chain** occurs when a client requests another object, that object requests yet another one, and so on. These chains mean that the client is dependent on navigation along the class structure. Any changes in these relationships require modifying the client.
- It also potentially violates the Law of Demeter (or Principle of Least Knowledge is a design guideline), which specify which methods are allowed to be called.
- Long chains of calls could be appropriate, however, if they return a limited set of objects that methods are allowed to be called on. If those objects follow the Law of Demeter, then the chain is appropriate.

Primitive Obsession

- **Primitive obsession** is a code smell that occurs when you rely on the use of built-in types too much. Built-in types, or primitives, are things like `ints`, `longs`, `floats`, or `strings`. Although there will be need of them in the code, they should only exist at the lowest levels of the code.
- Overuse of primitive types occurs when abstractions are not identified, and suitable classes are not defined.
- If you are using primitive types often at a high-level, then it is a good indicator that suitable classes are not being declared, and there is a primitive obsession code smell in the system.

Switch Statements

- **Switch statements** are a code smells that occur when switch statements are scattered throughout a program. If a switch is changed, then the others must be found and updated as well.

- Although there can be a need for long if/else statements in the code, sometimes switch statements may be handled better.

  For example,

  if conditionals are checking on type codes, or the types of something, then there is a better way of handling the switch statements.

- It may be possible to reduce conditionals down to a design that uses polymorphism.

Speculative Generality

- The code smell **speculative generality** occurs when you make a superclass, interface, or code that is not needed at the time, but that may be useful someday. This practice introduces generality that may not actually help the code, but "over-engineers" it.
- In Agile development, it best to practice **Just in Time Design**. This means that there should be just enough design to take the requirements for a particular iteration to a working system. This means that all that needs to be designed for are the set of requirements chosen at the beginning of an iteration.

Speculative Generality (contd.)

- Software changes frequently. Clients can change their mind at any time, and drop requirements. So your design should stay simple, and time should not be lost on writing code that may never been used.

- If generalization is necessary, then it should be done. This change may take longer at the time, compared to if it is set up for beforehand, but it is better than writing code that may not be needed down the line.

Refused Request

- A **refused request** code smell occurs when a subclass inherits something but does not need it.
- If a superclass declares a common behavior across subclasses, and subclasses are inheriting things they do not need or use, then they may not be appropriate subclasses for the superclass.
- Instead, it may make more sense for a stand-alone class. Or perhaps the unwanted behaviors should not be defined in the superclass.
- Finally, if only some subclasses use them, then it may be better to define those behaviors in the subclasses only.

Anti-Patterns & Code Smells: Summary

- Code smells help identify bad code and bad design. It is good practice to review your code frequently for code smells to ensure that the code remains reusable, flexible, and maintainable.