Refactoring

Vladyslav Sharapov

20/11/25

- Make your code gooder

Read code much harder that write it

Technical debt

- Is "a concept in programming that reflects the extra development work that arises when code that is easy to implement in the short run is used instead of applying the best overall solution".
- Aka design debt or code debt
- If technical debt is not repaid, it can accumulate 'interest', making it harder to implement changes later.
- Technical debt is not necessarily a bad thing, and sometimes (e.g., as a proof-of-concept) technical debt is required to move projects forward.

Any fool can write code that a computer can understand. Good programmers wrote code that humans can understand

When to re-write your code

- If it doesn’t work
- After you get the functionality working
- When you realise there’s a better way to do it
- When you have finished the basic requirements
- Just before it’s graded

Full rewrite

- Sometimes after a prototype (proof of concept) a full re-write is a good idea
- Normally a partial re-write is better
- Refactoring is also an option
- Makes less sense with disposable code (aka assignments that don’t satisfy a real marketplace need)

Refactoring

- *is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behaviour.*
- *is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behaviour.*
- **Advantages include**
    - improved code readability
    - reduced complexity
    - source-code maintainability
    - more expressive internal
    - architecture
    - Improved object model

Code Smells

- Once code has an object-oriented design, focus on improving its design one can
- If the design is procedural, begin this you cannot do
- Refactoring mentions code smells
- “If it stinks, change it”

To change or not to change

- Good programming practices
- It’s not always straightforward that a bad code smell must lead to a refactoring.
- Have to use judgement.
- Still, as new designers, bad code smells likely mean you should change your code.

Code Smells: Duplicated Code

- The same, or very similar code, appears in many places (code clones)
- **Problem**
    - A bug fix in one code clone may not be propagated to all
    - Makes code larger that it needs to be
- **Fix**: extract method refactoring
    - Create new method that encapsulates duplicated code
    - Replace code clones with method call

Commented Code

- A block of code that is surrounded by a comment block.
- Problem
    - Only the person that commented the code will know why it's commented and not deleted, it may have variables and methods no longer present or worse rely on logic that has changed.
- **Fix:** delete the code, you can always go back in the source control (Git) to an earlier version if you want to re-instate it.

Code smells: Long Method

- A method that has too many lines of code
- How long is too long? Depends.
- Over 20 is usually a bad sign.
  Under 10 lines is typically
  good.
- No hard and fast rules.
- **Problem**
    - The longer a method, the harder it is to understand, change, and reuse
- **Fix**: extract method
- Take chunks of code from inside long method, and make a new method
- Call new method inside the now-not-so-long method.

Code smells: Feature Envy

- A method in one class uses primarily data and methods from another class to perform its work
- Problem:
    - Indicates abstraction fault.
    - Ideally want data, and actions on that data, to live in the same class.
    - Feature Envy indicates the method was incorrectly placed in the wrong class
- Fix:
    - Move the method with feature envy to the class containing the most frequently used methods and data items

Code smells: Large class

- A class is trying to do too much
- Many instance variables
- Many methods
- **Problem**:
    - Indicates abstraction fault
    - There is likely more than one concern embedded in the code
    - Or some methods belong on other classes
    - Associated with duplicated code
- **Fix:**
    - Extract class refactoring
    - Take a subset of the instance variables and methods and create a new class with them
    - This makes the initial (long) class shorter
    - Move method refactoring
    - Move one or more methods to other classes

Code smells: switch statements

- The cases in a switch statement contain logic for different types of instances of the same class
- In object-oriented code, this indicates new subclasses should be created
- **Problem**
    - The same switch/case structure appears in many places
- **Fix**
    - Create new subclasses
    - Extract method to move case block logic into methods on the new subclasses

Code smells: Data class

- A class that has only class variables, getter/setter methods/properties, and nothing else
- Is just acting as a data holder
- **Problem**
    - Typically, other classes have methods with feature envy
    - That is, there are usually other methods that primarily manipulate data in the data class

Code smells: Data class

- **Problem**
    - Indicates these methods should really be on the data class
    - Can indicate the design is really procedural
- **Fix**

    - Examine methods that use datain the data class, and use move method refactoring to shift methods

Student Aromas

- Multiple Classes per file.
    - A single pair of files per class .cpp + .h
    - Smaller classes enums and namespace constants can be combined into a utility/common file
- Old code in comments
    - This can only lead to confusion an alternate path that you decided not follow should be removed when discarded.
- Code never used
    - Remove extra methods not used anymore.
- Code never written
    - Remove method declarations with no definitions.
- Variables never used
    - Remove variables declared but never used.
- Variable duplicated
    - Two variables storing the same data. Reduce and use only one.
- Variables over used.
    - Using the same variable for two different purposes. Each variable should have a distinct use.
- Confusing names
    - Ambiguous , anonymous, contradictory and generic. Each variable should have a properly qualified meaningful name.
- Lack of comments
    - Expect reader to arrive at correct understanding by reverse engineering. Comments should be greater than code.
- Simple comments
    - Each line of code commented by itself with no regard to the algorithm
- Magic Numbers
    - Constants entered directly into code.
    - Only use zero, one and two
- Automatic conversations
    - Use static_cast and have matching data types
-