STEVENS
INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

SSW 322: Software Engineering Design VI

Software Code Smell
2020 Spring

Prof. Lu Xiao
lxiao6@stevens.edu
Office Hour: Monday/Wednesday 2 to 4 pm
https://stevens.zoom.us/j/632866976
Software Engineering
School of Systems and Enterprises

No Final Exam!

Questions Responses 21

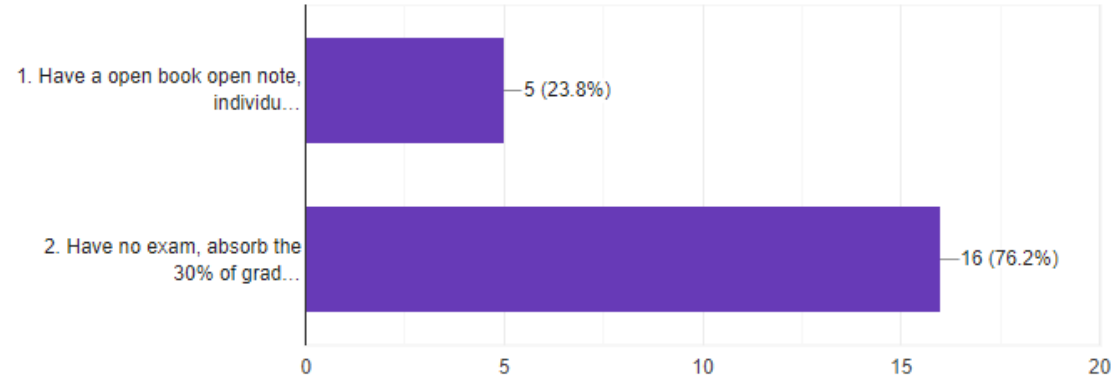
21 responses

Accepting responses

Summary Question Individual

What is your preference

21 responses



Today's Topic – Code Smell

- What is code smell? Why do we care?
- What are the different types of code smells?
- How to get rid of code smells?

– What? How can code "smell"??
– Well it doesn't have a nose... but it definitely can stink!

What is code smell, and why do we care?

- According to Martin Fowler, "a code smell is a surface indication that usually corresponds to a deeper problem in the system". Another way to look at smells is with respect to principles and quality:
  - ***"smells are certain structures in the code that indicate violation of fundamental design principles and negatively impact design quality".***
- Code smells are usually ***not*** bugs—they are not technically incorrect and do not currently prevent the program from functioning.
- Code smells indicate weaknesses in design that may be slowing down development or increasing the risk of bugs or failures in the future.

- What are the different types of code smells?
  - Bloaters
  - OO Abusers
  - Change preventers
  - Dispensables
  - Couplers

Code Smells

- **Bloaters**: Code, methods and classes that have increased to such gargantuan proportions that they are hard to work with.
- **OO Abusers**: All these smells are incomplete or incorrect application of OO programming principles.
- **Change Preventers**: If you need to change something in one place in your code, you have to make many changes in other places too.
- **Dispensables:** Something pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand.
- **Couplers:** Excessive coupling between classes.

Bloaters

- **Long Method:** A method contains too many lines of code.
- **Large Class:** A class contains many fields/ methods/lines of code.
- **Primitive Obsession:** Use of primitives instead of small objects for simple tasks
- **Long Parameter List:** More than three or four parameters for a method.
- **Data Clumps:** different parts of the code contain identical groups of variables. These clumps should be turned into their own classes.

Bloaters

- 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).
- Bloaters make code difficult to understand and maintain. For example:
  - Long methods and large classes are difficult to understand, test, and debug
  - Primitive types have low cohesion: related primitive types are scattered around in the code.

OO Abusers

- All these smells are incomplete or incorrect application of object-oriented programming principles.
  - Switch Statements
  - Temporary Filed
  - Refused Bequest
  - Alternative Classes with Different Interfaces

Switch Statements

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

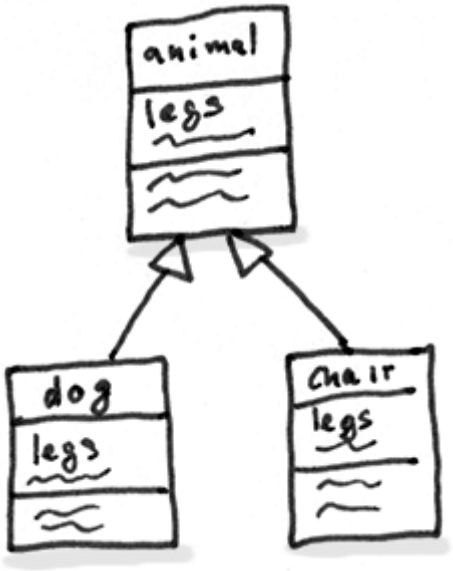
- It is not a problem when a switch operator performs simple actions. But in complicated cases, code for a single switch can be scattered in different places in the program. When a new condition is added, you have to find all the switch code and modify it.
- As a rule of thumb, when you see switch you should think of polymorphism.

Temporary Field

- Temporary fields get their values (and thus are needed by objects) only under certain circumstances. Outside of these circumstances, they are empty.
  - For example, if you let your questionnaire class aggregates the answer sheet class. The latter is null when no users has submitted responses.
- This kind of code is tough to understand. You expect to see data in object fields but for some reason they are almost always empty.

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.
- Someone was motivated to create inheritance between classes only by the desire to reuse the code in a superclass. But the superclass and subclass are completely different.



Alternative Classes with Different Interfaces

- Two classes perform identical functions but have different method names.
- The programmer who created one of the classes probably didn't know that a functionally equivalent class already existed.

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.
  - Divergent Change
  - Shotgun Surgery
  - Parallel Inheritance

Divergent Change

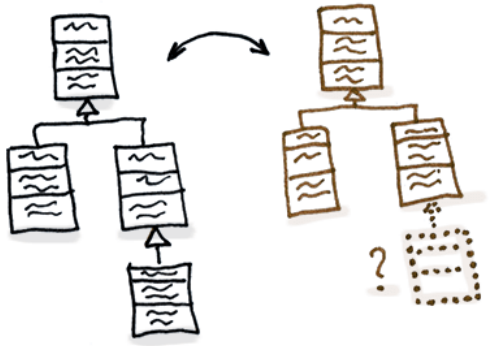
- You find yourself having to change many unrelated methods when you make changes to a class. For example, when adding a new product type you have to change the methods for finding, displaying, and ordering products.
- Often these divergent modifications are due to poor program structure or "copypasta programming”.

Shotgun Surgery

- Making any modifications requires that you make many small changes to many different classes.
- A single responsibility has been split up among a large number of classes.

Parallel Inheritance

- Whenever you create a subclass for a class, you find yourself needing to create a subclass for another class.



- All was well as long as the hierarchy stayed small. But with new classes being added, making changes has become harder and harder.

Dispensables

- A dispensable is something pointless and unneeded whose absence would make the code cleaner, more efficient and easier to understand
  - Comments
  - Duplicate code
  - Lazy class
  - Data class
  - Dead code

Dispensables

- Comments: A method is filled with explanatory comments.
- Duplicate code: Two code fragments look almost identical.
- Lazy class: A class doesn't do enough.
- Data class: a class that contains only fields and its getters/setters.
- Dead code: A variable, parameter, field, method or class is no longer used

Couplers

- All the smells in this group contribute to excessive coupling between classes or show what happens if coupling is replaced by excessive delegation.
  - Feature Envy
  - Inappropriate Intimacy
  - Message Chains
  - Middle Man

Feature Envy

- A method accesses the data of another object more than its own data.
- This smell may occur after fields are moved to a data class. If this is the case, you may want to move the operations on data to this class as well.

Inappropriate Intimacy

- One class uses the internal fields and methods of another class (similar to feature envy).
- Keep a close eye on classes that spend too much time together. Good classes should know as little about each other as possible. Such classes are easier to maintain and reuse.

Message Chains

- In code you see a series of calls resembling
  - $a()->b()->c()->d()
- 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.

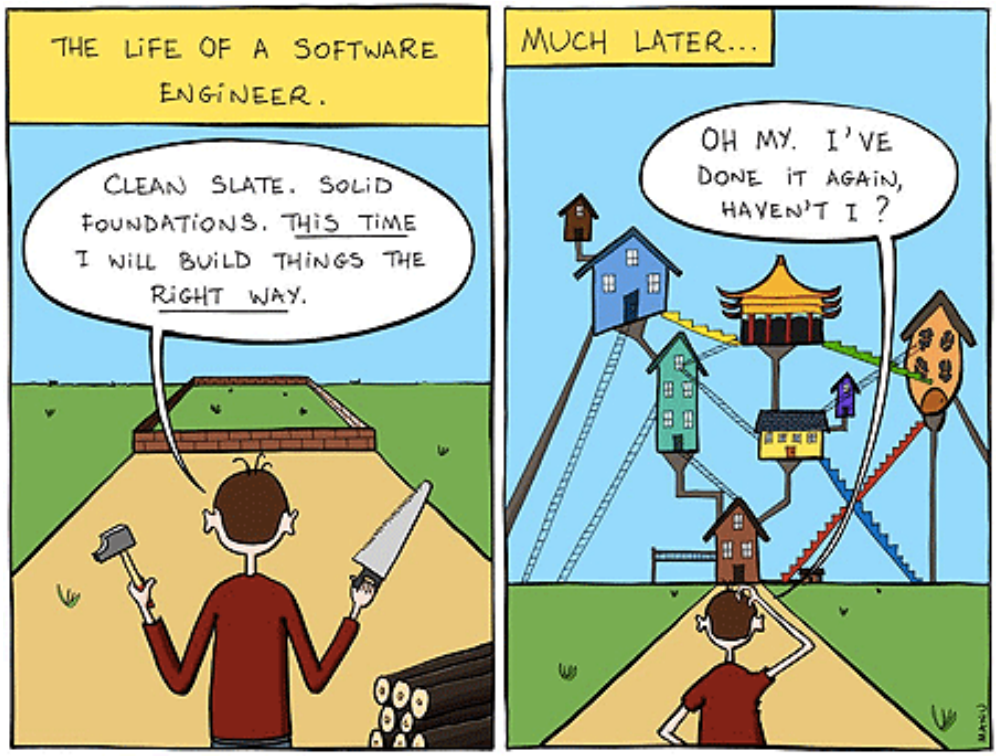
Middle Man

- If a class performs only one action, delegating work to another class, why does it exist at all?
- This smell can be the result of overzealous elimination of message chains.
- In other cases, it can be the result of the useful work of a class being gradually moved to other classes. The class remains as an empty shell that does not do anything other than delegate.

- How to remove code smells?
  - Refactoring

Refactoring

- Refactoring: Improving a piece of software's internal structure without altering its external behavior.
- Incurs a short-term time/work cost to reap long-term benefits
- A long-term investment in the overall quality of your system.



Why fix a part of your system that isn't broken?

- Each part of your system's code has 3 purposes. If the code does not do one or more of these, it is "broken.“
  1. to execute its functionality,
  2. to allow change,
  3. to communicate well to developers who read it.
- Refactoring improves software's design, code quality, and makes code easier to understand.
  - Refactoring to remove code smells

Refactoring Operations

- Extract methods, move methods, rename methods
- Extract class, extract superclass, extract subclass
- Introduce new classes, remove classes
- Replace conditional with polymorphism
- Move fields


thank you