Objectives

- Code Smells
- Design Patterns
- Exam 2 Review

1

Review

1. Rumor has it that Elon Musk ranked programmers by the number of lines of code they wrote last year and fired those at the bottom of that list because they were less productive. Is that a good metric to use? Why or why not?
2. What is guaranteed in software development?
3. What are best practices in object-oriented design?
   - Provide an example of the practice (in our assignments, in our discussions, in Java, …)
4. Define *code smell*. What is an example of a code smell? How do we address that code smell?
   - What is common to how we address code smells?
5. What is the process for writing maintainable code?
   - Define terms in that process

2

Review: Designing Systems

All systems **change** during their life cycle

- Questions to consider:
  - How can we create designs that are stable in the face of change?
  - How do we know if our designs aren't maintainable?
  - What can we do if our code isn't maintainable?
- Answers will help us
  - Design our own code
  - Understand others' code

Review: Open-Closed Principle

Principle: Software entities (classes, modules, methods, etc.) should be **open** for **extension** but **closed** for **modification**

- Design modules that *never change* after completely implemented
- If requirements change, extend behavior by adding code
  - By not changing existing code → we won't create bugs!
- Closed: APIs/interfaces
- Open: add new implementations

Refactoring: Solution to Code Smells

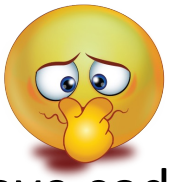
Refactoring: Updating a program to improve its design and maintainability *without changing its current functionality significantly*

After refactoring your code, what should you do next?

5

Revised Process to Write Maintainable Code

Apply the design principles, but as your code evolves, you'll see that you didn't always adhere to the principles



1. Identify code smell
2. ***Refactor*** code to remove code smell
3. ***Test*** to confirm code still works!

6

Review: Code Smells

A hint in the code that something could be designed better

- Duplicated code
- Long method
- Large class
- Long parameter list
- Very similar child classes
- Too many public variables
- Empty catch clauses
- Switch statements/long if statements
- Shotgun surgery
- Literals
- Global variables
- Side effects
- Using `instanceof`

Code Smells

- For each of the following code smells, state
  - Why these may occur in code
  - Why they are a problem in terms of maintaining code
    - Cite design principles as appropriate
  - How to fix them
- Code smells:
  1. Long methods
  2. Large class
  3. Magic numbers (e.g., -1 or 480 in code)
  4. Comments (not API/Javadoc comments)

Front two rows: 1, 3
Back two rows: 2, 4
Door side: swap order

Code Smell: Long Methods

- What's the problem with long methods?
- What made us write them?
- How can we fix them?
- What is an issue with lots of short methods?

Long Methods: Issues and Solutions

- Issues:
  - Hard to understand (see) what method does
  - Harder to change because code gets coupled
- Solution:
  - Find lines of code that go together (may be identified by a comment) and extract method
- Critique of refactored, smaller methods
  - Smaller methods have reader overhead
    - Look at code for called methods
    - But, use descriptive names for methods
    - In Eclipse, use F3 to jump to a method's definition

Code Smell: Large Class

- What could be the problem?

Large Class

- Issue: Too many instance variables → trying to do too much
  - Violates **Single Responsibility Principle**
- Solutions:
  - Bundle groups of variables together into another class
    - Look for common prefixes or suffixes
  - If includes optional instance variables (only sometimes used), create child classes
  - Look at how users use the class for ideas of how to break it up

Eclipse: Refactor → Extract Class or Extract Superclass

Literals or Magic Numbers

- If a number has a special meaning, make it a constant
- Example: Distinguish between 0 and NO_VALUE_ASSIGNED
  - If value changes (e.g., -1 instead of 0), only one place to change
  - Less error-prone (e.g., was I using 1 or -1?)

Eclipse: Refactor → Extract Constant

Comments

Problem: Comments used as Febreze to cover up smells

- Describe what the code or method is doing
- Should be reserved for *why*, not what

- Solutions:
  - If need a comment for a block of code (or a long statement) → create a method with a descriptive name
  - If need a comment to describe method, rename method with more descriptive name

These [internal] comments are different from API comments

More Code Smells

- Discuss more code smells and solutions (Design Patterns) later

Software Design Rules of Thumb

- Code smells are not ***always*** bad
  - Do not always mean code is poorly designed
- Open code is not ***always*** bad
- Need to use your judgment
  - Good judgment comes from experience.
  - How do you get experience? *Bad judgment works every time*

Goal: Gain experience to improve your judgment

Refactoring Summary

- Write code and then *rewrite* code
  - Eye toward extensibility, flexibility, maintainability, and readability
  - Maintain correctness
- Reading/understanding other people's code can be difficult
  - Make your code readable, understandable
- Probably impossible to design/write "correctly" the first time
  - A lot harder to get the logic right, make sure you're not creating bugs, know/check the right answer…
  - Don't necessarily know what is likely to change

How can we create designs that are stable in the face of change?

DESIGN PATTERNS

Design Pattern

General reusable solution to a commonly occurring problem in software design

- Not a finished design that can be transformed directly into code
- Description or *template* for how to solve a problem that can be used in many different situations
  - "Experience reuse" rather than code reuse

Defined Design Patterns

- Software best practices
- Catalogued and discussed in *Design Patterns: Elements of Reusable Object-Oriented Software*
  - Written by the "**Gang of Four**": Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides
    - Erich Gamma also co-wrote original JUnit framework
  - Didn't design the patterns; identified them

Understanding Code with Design Patterns

1. Recognize design pattern in code base you're using
2. Understand code design better

Applying Design Patterns

1. Recognize problem as one that can be solved by a design pattern
2. Apply pattern to your problem

Danger: over-applying design patterns
- Fall back: Identify and resolve code smells

Design Principle: Favor Composition Over Inheritance

- Design Pattern: Composition
  - Using other objects in your class
  - "Delegate" responsibilities to this object

Why is composition preferred over inheritance?

28

Design Principle: Favor Composition Over Inheritance

- Design Pattern: Composition
  - Using other objects in your class
  - "Delegate" responsibilities to this object
- Why is composition preferred over inheritance?
  - Inheritance → dependence on parent class
    - Only want to depend on things you know won't change (higher stability)
  - Composition: Provide different behaviors for your class by plugging in new object

29

Refactoring is not just a Java Thing

Before:

```
# Check if path exists. if it doesn't, exist create directory
if not os.path.exists(config['output_dir'] +
       config['app_name'] + config['bot_dir'] + 'analysis'):
    os.makedirs(config['output_dir'] +
       config['app_name'] + config['bot_dir'] + 'analysis')
    print("Created directory for" + config['output_dir'] +
       config['app_name'] + config['bot_dir'] + 'analysis')
```


Refactoring is not just a Java Thing

Before:

```
# Check if path exists. if it doesn't, exist create directory
if not os.path.exists(config['output_dir'] +
       config['app_name'] + config['bot_dir'] + 'analysis'):
    os.makedirs(config['output_dir'] +
       config['app_name'] + config['bot_dir'] + 'analysis')
    print("Created directory for" + config['output_dir'] +
       config['app_name'] + config['bot_dir'] + 'analysis')
```

After:

```
# Check if path exists. if it doesn't, exist create directory
outputDir = config['output_dir'] +
       config['app_name'] + config['bot_dir'] + 'analysis'
if not os.path.exists(outputDir):
    os.makedirs(outputDir)
    print("Created directory for", outputDir)
```

Exam 2 Discussion

- Similar format to Exam 1
  - Timed (70 minutes), online
  - Open book/notes/slides **NOT** internet
  - 3 "sections" – very short answer, short answer, applied
  - Open Friday at 8:30 a.m. through Sunday at 11:59 p.m.
- Content covers through today's class
- I will hold office hours during Friday's class times
- Prep document posted