Objectives
- Code Critique
  - Identifying smells
- Refactoring
- Liskov Substitution Principle

Review
- What goal are we designing to?
- What is the typical fix for code smells?
  - What is a limitation of those fixes?
- How do we address change in general?

Literals or Magic Numbers
- If a number has a special meaning, make it a constant
  - Distinguish between 0 and NO_VALUE_ASSIGNED
  - If value changes (-1 instead of 0), only one place to change

Divergent Change & Shotgun Surgery
Problem: when make a change, can’t identify single point to make change

Divergent Change
- Problem: one class commonly changed in different ways for different reasons
- Solution:
  - Identify changes for a particular cause
  - Put into a class (extract)

Shotgun Surgery
- Problem: a change causes changes in many classes
- Solution:
  - Identify class that changes should belong to

Goal: 1-to-1 mapping of common changes to classes

Data Clumps
- Problem: You have some data that always “hangs out together”
- Possible Solution: Maybe they should be an object
  - Check: if you deleted one of those pieces of data, would the others make sense?
    - If answer is no, should be an object

Message Chaining
- Dynamic coupling:
  - getOrder().getCustomer().getAddress().getState()
- Problem: client coupled to navigation structure
  - Depends on too many other classes
  - Change to intermediate class → Change in this class
- Fix: add delegate method
  - Example: add method getShippingState()
  - Can go too far if adding too many methods

Eclipse: Refactor → Extract Constant

Eclipse: Refactor → Extract Constant

Eclipse: Refactor → Extract Class

Eclipse: Check references
Refactor → Extract Method
Middle Man

• Issue:
  ➢ Many methods of one class are delegating to another class

• Possible Solutions
  ➢ Inline method into caller
  ➢ If there is additional behavior, replace delegation with inheritance to turn the middle man into a subclass of the real object

Lazy Class

• Problem
  ➢ Class in question doesn’t do much
  ➢ Classes cost time, money to maintain & understand

• How could this happen?
  ➢ Refactoring!
  ➢ Planned to be implemented but never happened

• Solution
  ➢ Get rid of class
    * Inline or collapse subclass into parent class

Speculative Generality

• Beware of too much abstraction, allowing for too much flexibility that isn’t required

• Solution: Collapse classes

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

Other Code Smells

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

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
Discussion of Abstraction
• What does abstraction allow?
• Are there any limitations to abstraction?

Liskov Substitution Principle (LSP)
• Named after Barbara Liskov
  ➢ MIT Professor of Engineering
  ➢ 2008 ACM Turing Award
  ➢ Contributions to programming languages, pervasive computing
  ➢ Trivia: first woman in the United States to receive a Ph.D. from a computer science department (Stanford, 1968)

Liskov Substitution Principle (LSP)
• The substitution principle:
  If for each object \( o_1 \) of type \( S \) there is an object \( o_2 \) of type \( T \) such that for all programs \( P \) defined in terms of \( T \), the behavior of \( P \) is unchanged when \( o_1 \) is substituted for \( o_2 \), then \( S \) is a subtype of \( T \).
• In other words...
  If a module is using a Base class, then it should be able to replace the Base class with a Derived class without affecting the functioning of the module.

Code Smell: Using instanceof
• Why isn’t this good code?
• How could we write this in a better way?

Code Smell: Using instanceof
• Previous example: had to know all of the Shape classes
  ➢ Update whenever a Shape is added or removed
• Better code: Polymorphic!

Another Example: Rectangle Class
• Previous example: had to know all of the Shape classes
  ➢ Update whenever a Shape is added or removed
• Better code: Polymorphic!
Square Class

• A square is a rectangle
  ➢ But a rectangle is not a square
• In the interest of code reuse
  
  public class Square extends Rectangle
• Any problems with this implementation?
  ➢ Inherits:

    private int myHeight;
    private int myWidth;
    public void setWidth( int w );
    public void setHeight( int h );

To Keep Square Consistent...

    public void setWidth( int w ) {
      super.setWidth(w);
      super.setHeight(w);
    }
    public void setHeight( int h ) {
      super.setWidth(h);
      super.setHeight(h);
    }

But What About Users of Classes?

• Consider the function:

    public void testMethod( Rectangle r ) {
      r.setWidth(5);
      r.setHeight(4);
      assertEquals(20, r.getWidth()*r.getHeight());
    }

• What happens if it's called with a Square?

The Problem

• A Square object is not a Rectangle object
• Behaviors are different
• Clients depend on behaviors

Lesson: All derivatives of class must have the same behavior

Design by Contract

• Methods of classes declare preconditions and postconditions
  ➢ Preconditions must be met for method to execute
  ➢ After executing, postconditions must be true
  ➢ Example for Rectangle’s setWidth:
    • myWidth == newWidth && myHeight == oldHeight
• For derivatives
  ➢ Preconditions can only be weakened
  ➢ Postconditions can only be strengthened
  ➢ Derivatives must adhere to constraints for base class

Summary of LSP

• Liskov Substitution Principle (a.k.a. design by contract) is an important feature of programs that conform to the Open-Closed Principle
  ➢ Derived types must be completely substitutable for their base types
  ➢ Derived types can then be modified without consequence
CODE CRITIQUE

Discussion of Bins Solution

- What does the code do?
  ➢ What is the purpose/responsibility of each class?
- What are the good parts of the code?
- What are some of the code smells?

Notes on Assignment 10

- No "right" answer
  ➢ Many design decisions
  ➢ Want you to defend your design decision in code critique
- No refactoring → Bins Assignment → Lots of refactoring
- Iterate

Assignment 10: Code Critique & Refactoring

- Given: a problem specification and a solution to the problem
  ➢ You refactoring your own code is emotional
  ➢ More objective with someone else’s solution
- Goals
  ➢ Read and understand someone else’s code
    ➢ Haven’t done much of this in Java
  ➢ Critique code (do you smell something?)
    ➢ Identify, articulate problems
  ➢ Refactor code to solve problems identified
  ➢ Write tests to verify the code