Objectives

- Code Smells
- Refactoring

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`

Duplicated Code

- What's the problem with duplicated code?
- Why do we like it?
  - What made us write the duplicated code?

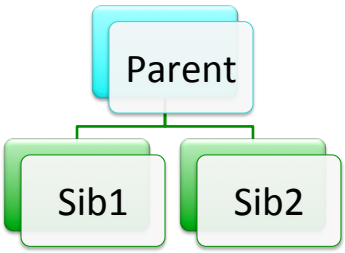
What can we do when we have duplicated code?
(How can we get rid of the duplicate code?)

Duplicated Code

- What's the problem with duplicated code?
  - If code changes, need to change in every location
  - Duplicate effort to test code to make sure it works
    - More statements for test suite to test!
  - When trying to search for code, may find a duplicate code→ not the one you're looking for

Duplicated Code

- Consider: same expression in at least one method of a class
  - Solution: Extract method
  - Call method from those two places
- Consider: duplicated code in 2 sibling child classes



Duplicated Code

- Consider: duplicated code in 2 sibling child classes
  - Extract method, put into parent class
    - Eclipse: extract method, pull up
  - If similar but not duplicate, extract the duplicate code or parameterize
- Consider: duplicated code in unrelated classes

Duplicated Code

- Consider: duplicated code in unrelated classes
  - Ask: where does method belong?
  - One solution:
    - Extract class
    - Use new class in classes
  - Another solution:
    - Keep in one class
    - Other class calls that method

Refactoring: Solution to Code Smells

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

- Example
  - Creating a single method that replaces 2 or more sections of similar code
    - Reduces redundant code
    - Makes code easier to debug, test

After refactoring your code, what should you do next?

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
  - Smaller methods have reader overhead
    - Look at code for called methods
    - But, should use descriptive names
    - In Eclipse, use F3 to jump to a method's definition
- Solutions:
  - Find lines of code that go together (may be identified by a comment) and extract method

Large Class

- What's 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

Long Parameter List

- More difficult to use (do I have everything?)
  - Example: `MediaItem`, subclass constructors
- If method signature changes, have a lot of places to change
- Solutions: Use objects
  - Instead of separate parameters for an object's data
  - Group parameters together

Eclipse: Refactor → Introduce Parameter Object
OR Refactor → Change Method Signature

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 (e.g., -1 instead of 0), only one place to change
  - Less error-prone (e.g., was I using 1 or -1?)

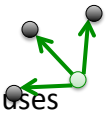
Eclipse: Refactor → Extract Constant

Divergent Change & Shotgun Surgery

Problem: when make a change, can't identify single point in code 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

Eclipse: Refactor → Extract Class

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: Check references
Refactor → Extract Method

Middle Man

- Issue: Many methods of one class are delegating to another class
- How could this happen?
  - Refactoring!
- 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 and money to maintain and 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

These comments are different from API comments

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

Set Up for in-class work

- Import Bins project
- Go to schedule page and download bins.tar
- In Eclipse, select
  - File → Import → General → Existing projects into workspace
  - Select the Archive file button
    - Choose the bins.tar file you just downloaded
  - Finish