Objectives

- Refactoring
- Liskov Substitution Principle
- Design Patterns

1

Review

1. What is guaranteed in software development?
   - This informs how we design our code
2. What is refactoring?
3. What is the process for writing maintainable code?
4. What are some code smells and how do we address them?
   - What is common to how we address code smells?
5. What is the open-closed principle?
   - How does it relate to the Roulette code base?

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

3

Review: Process to Write Maintainable Code

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

1. Identify code smell

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

4

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`

5

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

6

Speculative Generality

- Beware of too much abstraction, allowing for too much flexibility that isn't required
- Solution: Collapse classes

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

LISKOV SUBSTITUTION PRINCIPLE

9

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.*

 Liskov & Wing, 1994

10

Design by Contract

- By Bertrand Meyer (Open-Closed Principle)
- Methods of classes should 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`

Design by Contract and LSP

- Methods of classes should 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 derived/child classes
  - Preconditions can only be weakened
  - Postconditions can only be strengthened
  - ➥Derivatives must adhere to constraints for base class

Design by Contract and LSP

- Recall: Programmer interacts with interface, e.g., the base class

What if preconditions are stronger?
What if postconditions are weaker?

- For derivatives
  - Preconditions can only be weakened
  - Postconditions can only be strengthened
  - ➥ Derivatives must adhere to constraints for base class

Rectangle Class

```
public class Rectangle {
   private int myHeight;
   private int myWidth;

   public void setWidth( int w ) {
      myWidth = w;
   }

   public void setHeight( int h ) {
      myHeight = h;
   }

   // getters…
}
```

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 test method:

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

- What happens if a `Square` object is passed into method?

The Problem

- A `Square` object is ***not*** a `Rectangle` object
- Behaviors w.r.t. pre-/post-condition contract are different
- Clients depend on those behaviors

Lesson: All derivatives of class **must** have the same contract-defined ***behavior***

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

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)

There is an advanced lab machine named after her.

 Liskov & Wing, 1994

& Wing

- Jeannette Wing
  - Director of Data Science Institute at Columbia University
  - Big proponent of computational thinking as assistant director for Computer and Information Science and Engineering at the NSF from 2007 to 2010.

22

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

DESIGN PATTERNS

23

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 JUnit framework
  - Didn't design the patterns; identified them

Understanding Code w/ Design Patterns

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

26

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

27

Audubon Society calls…

- Need to represent all the different birds
  - Various flying behaviors (some fly, some don't)
  - Make different sounds
  - Examples: Duck, Penguin, Hummingbird, Ostrich, Chicken, Oriole, …

How can we represent different birds?

Solution Non-Starter: Hierarchy of Classes

- FlyingBird
  - FlyHighBird
    - Eagle
    - …
  - FlyLowBird
    - SingingFlyLowBird
    - SquawkingFlyLowBird
- FlightlessBird
  - …

Identify what is likely to change/vary
- Flying
- Sound

Designing Flexible Behaviors

- Include behaviors in abstract `Bird` class
  - `FlyBehavior` has `performFly()` method
  - `SoundBehavior` has `makeSound()` method
- Could have setter methods in `Bird` class to change these
  - Example: bird's wings get clipped

Designing Flexible Behaviors

```
public abstract class Bird {
      protected FlyBehavior flyB;
      protected SoundBehavior soundB;

      public Bird() {
            …
      }

      public void performSound() {
            soundB.makeSound();
      }

      public void performFly() {
            flyB.performFly();
      }
}
```

Designing Flexible Behaviors

```
public class Duck extends Bird {
      //Recall: protected FlyBehavior flyB;
      //Recall: protected SoundBehavior soundB;

      public Duck() {


      }
      …
}
```

What do we need to do in here?

Designing Flexible Behaviors

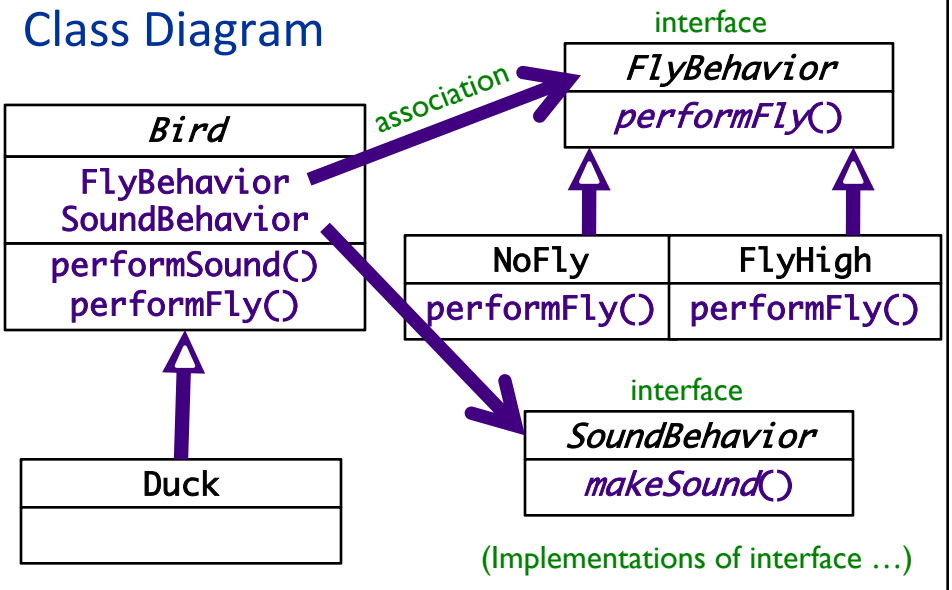
```
public class Duck extends Bird {

      public Duck() {
            flyB = new FlyHighBehavior();
            soundB = new QuackBehavior();
      }


}
```

Do we need to do anything else to *this* class, with respect to fly and sound behavior?

How Do We Implement…

- Hummingbird?
- Penguin?
- Ostrich?

Class Diagram



Unified Modeling Language (UML)

- Standardized general-purpose modeling language
  - Graphical language for visualizing, specifying and constructing the artifacts of a software system
- Includes a set of graphical notation techniques to create abstract models of specific systems
- Used in designing a large system
  - Focus on big picture, not the code

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?

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

Not covered in class

Alternative: Using Interfaces

- We could have a `Flyable` interface with a `performFly()` method and a `Vocalable` interface with a `makeSound()` method
- Then, each Bird class would *implement* `Flyable` and `Chirpable`, as appropriate, i.e.,
  - Implement `performFly` and `chirp` methods

Pros and cons of this solution?

Not covered in class

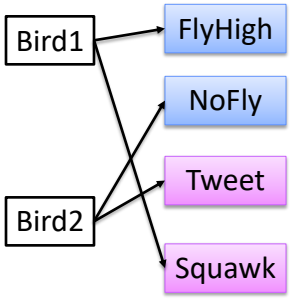
Using Interfaces: Pros and Cons

- We could have a `Flyable` interface with a `performFly()` method and a `Vocalable` interface with a `makeSound()` method
- Then, each Bird class would ***implement*** `Flyable` and `Vocalable`, as appropriate, i.e.,
  - Implement `performFly` and `makeSound` methods
- Pros: Using an interface → more flexible
  - Depending on interface instead of implementation
- Con: Duplicated code, implement in each class

Not covered in class

Comparing Approaches

Composition/Delegation



Consider what this looks like as we add more bird classes

- Bird class is *composed* of these behaviors
- Can be easily switched out
- One place to change implementation

Implement Interfaces

Bird1
- performFly
  - Fly high impl
- makeSound
  - Squawk impl

Bird2
- performFly
  - No fly impl
- makeSound
  - Tweet impl

- May need to duplicate implementing interface

Dependency Inversion Principle

Depend upon abstractions

42

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 9:30 a.m. through Sunday at 11:59 p.m.
- Content covers through Wednesday's class
- I will hold office hours during Friday class time

43

Looking Ahead

- Assignment 8
  - Deadline extended to Thursday at 11:59 p.m.
- Clone the code for Wednesday's lab
- Exam - Fri - Sun