Objectives

- Animation
- Design Patterns

Discussion of Roulette Assignment

- How easy/difficult to refactor for extensibility?
- Was it easier to add to your refactored code?
  - What would your refactored classes have looked like if I hadn't told you that you were going to add the three other bets?
- How easy/difficult was it to test your classes?

Animation Review

- What object do we use to "draw" in Java?
  - What are some things we can do?

Understanding Code

Import project:
`/home/courses/cs209/handouts/screensavers.tar`

- Bouncers (package `bouncers`)
  - What does each class do?
  - How does it draw?
  - How does it animate?

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 JUnit framework

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

Motivating Example

- 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?

Designing Flexible Behaviors

- Include behaviors in abstract `Bird` class
  - `FlyBehavior` object has `performFly()` method
  - `SoundBehavior` object has `makeSound()` method
- Could have setter methods in `Bird` class to change these
  - Example: bird gets wings 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 {
      //Recall: protected FlyBehavior flyB;
      //Recall: protected SoundBehavior soundB;

      public Duck() {

      }
      …
}
```

What do we need to do in here?

Designing Flexible Behaviors

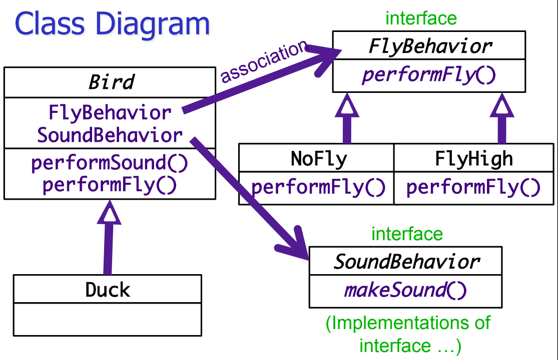
```
public class Duck {

     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



UML Diagram

Design Principle:
Favor Composition Over Inheritance

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

Why is composition preferred over inheritance?

- Composition: Provide different behaviors for your class by plugging in new object
- Inheritance → dependence on parent class
  - Only want to depend on things you know won't change (higher stability)

Another Solution: Interface

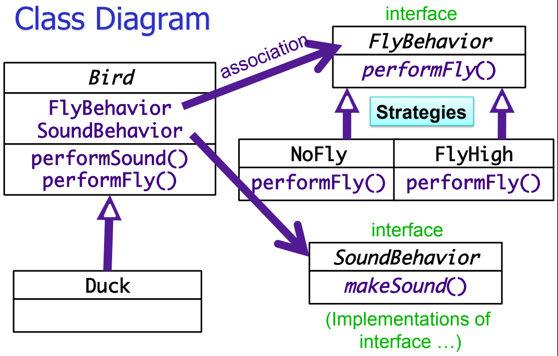
- We could have a `Flyable` with a `performFly() method` and a `Chirpable` interface with a `chirp()` method

Pros and cons of this solution?

Pros and Cons of Interface Solution

- We could have a `Flyable` with a `performFly() method` and a `Chirpable` interface with a `chirp()` method
- Pros: Using an interface → more flexible
  - Depending on interface instead of implementation
- Con: Duplicated code, implement in each class

Class Diagram



Design Pattern: **Strategy**

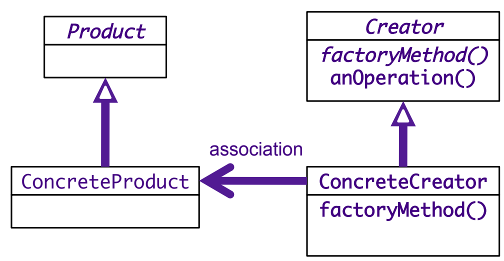
- Defines a family of algorithms, encapsulates each one, and makes them interchangeable
- Lets algorithm/behavior vary independently from clients that use it
  - Allows behavior changes at runtime
- Design Principle:

Favor ***composition*** over inheritance

Design Pattern: **Factory Methods**

- Allows you to create objects without specifying exact (concrete) class of created object
- How it works:
  1. Define a method for creating objects
  2. Child classes override method to specify the derived type of product that will be created
- Often used to refer to any method whose main purpose is creating objects

Factory Method Pattern



UML Class Diagram

CODE REVIEW

Understanding Code: Screen Savers

- In Eclipse, import an existing project:
  `/home/courses/cs209/handouts/screensavers.tar`
- Run `Main` class
- Answer questions about code
  - What represents an object in the screen saver?
  - How generates screen saver objects?
  - How handles animation?
  - How handles events?

Mapping Factory Design Pattern to Screen Savers

- How does the screen saver application use factory methods?
- What would be the alternative solution?
- What problems are the factories addressing?

Mapping Factory Design Pattern to Screen Savers

- How does the screen saver application use factory methods?
- What would be the alternative solution?
- What problems are the factories addressing?
  - Delegate creation of concrete `Movers`
    - Likely to change
    - Encapsulate change in factory
  - Using abstraction instead of specifying concrete classes
    - Reduces dependencies to concrete classes

Compiler's Names of Classes

- Anonymous class names
  - `ClassName$#.class`
- Look inside <workspace_dir>/ScreenSavers/bin/screensaver/nomodify

Assignment 12

- Complete screen savers for
  - Racers
  - Random Walkers
  - "Interesting" circles
- Due Friday