Objectives

- Design Patterns
  - Composition, Delegation
  - Strategy
  - MVC

1

Review

1. What is common to how we address most code smells?
2. What is the design-by-contract/Liskov Substitution Principle?
   - How does it relate to the Roulette code base?
3. What are design patterns? How are they used?
4. What was the solution to the Audobon Society's bird modeling problem?

2

Review: 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
  - Preconditions can only be *weakened*
  - Postconditions can only be *strengthened*
- Derived types can then be switched out without breaking the code

Review: 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

Towards a Solution: 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?

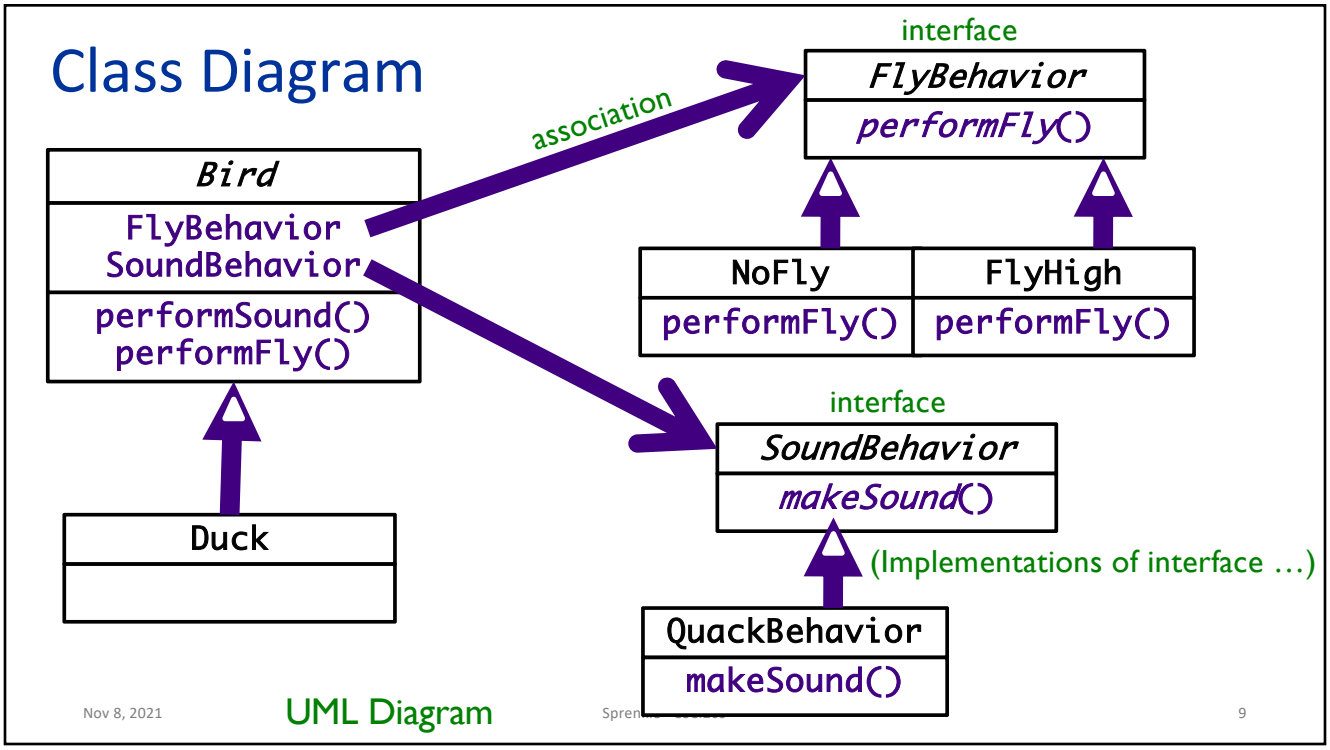
Designing Flexible Behaviors

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

        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?

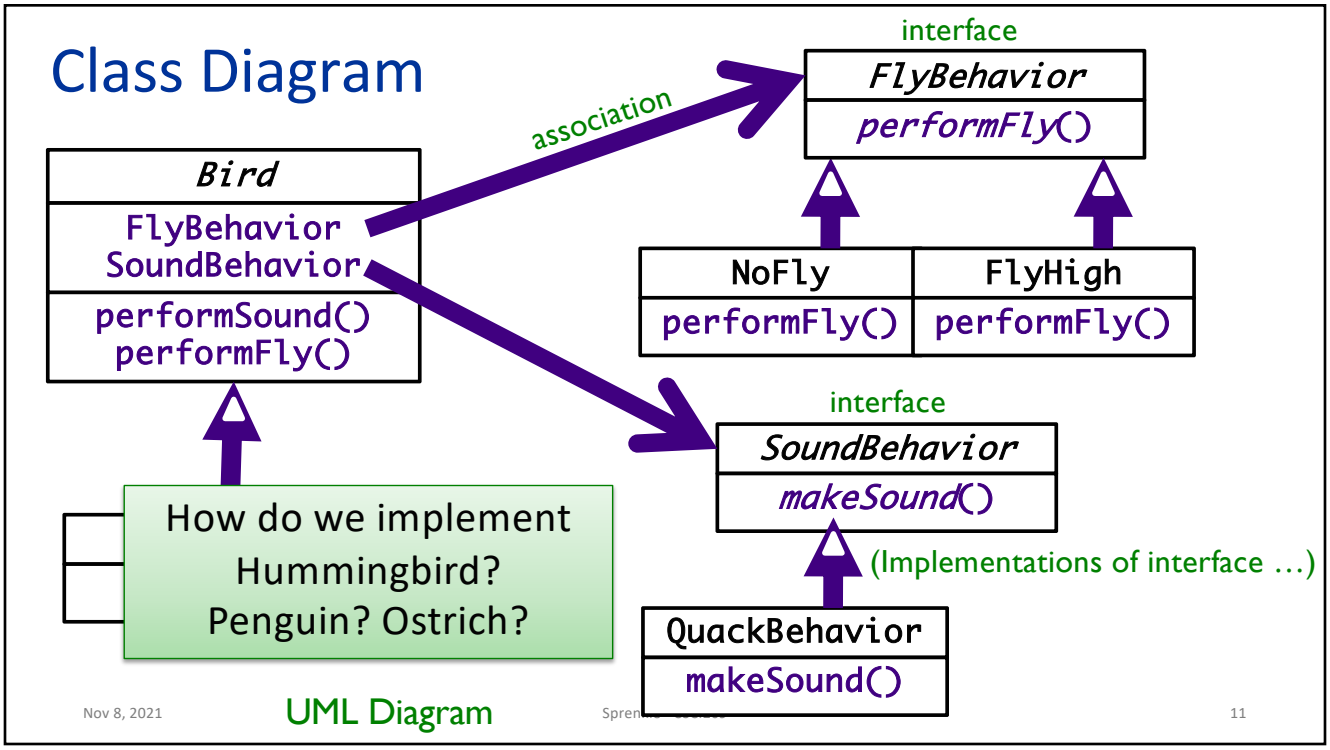


9

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

10



11

Design Principle:
Favor Composition Over Inheritance

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

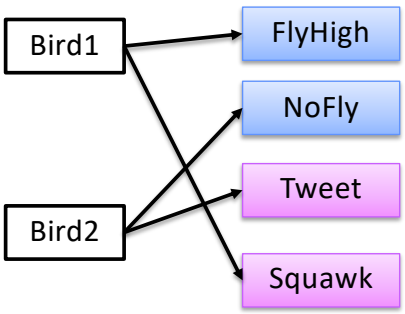
Why is composition preferred over inheritance?

12

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

Composition in the Bird Classes

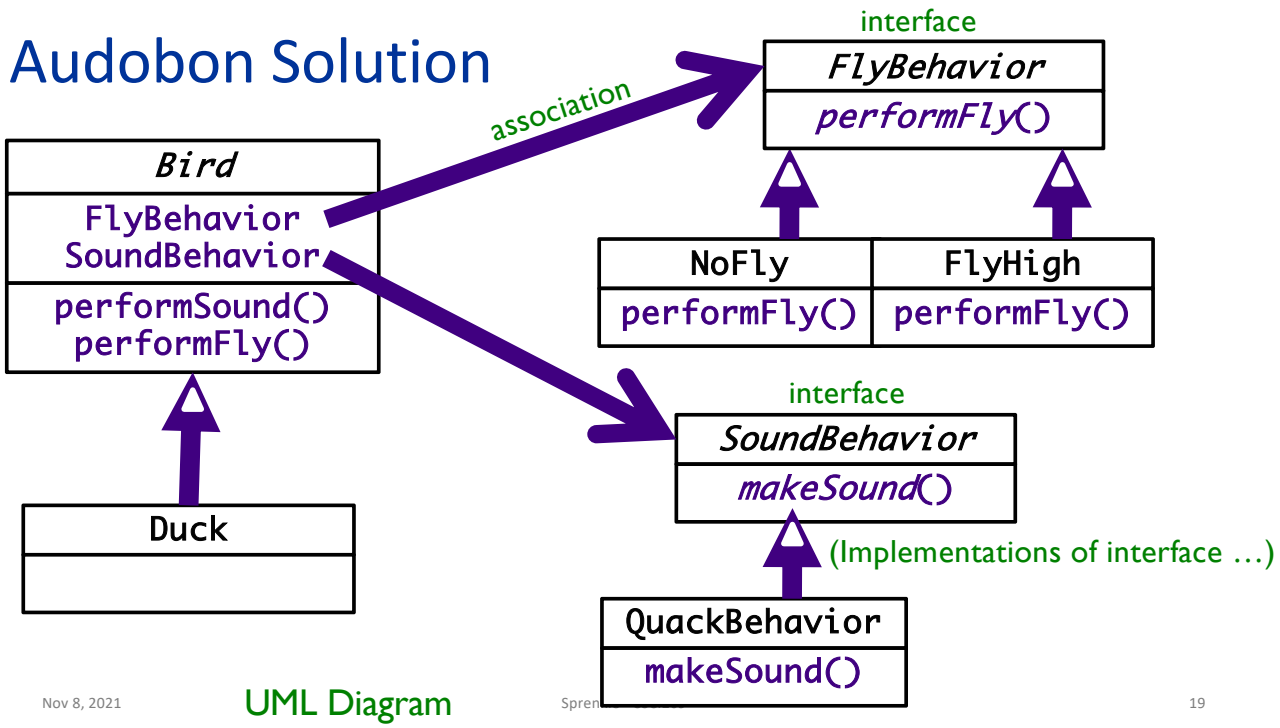


- Bird class is *composed* of these behaviors
  - Flight
  - Sound
- Can be easily switched out
- One place to change implementation

Dependency Inversion Principle

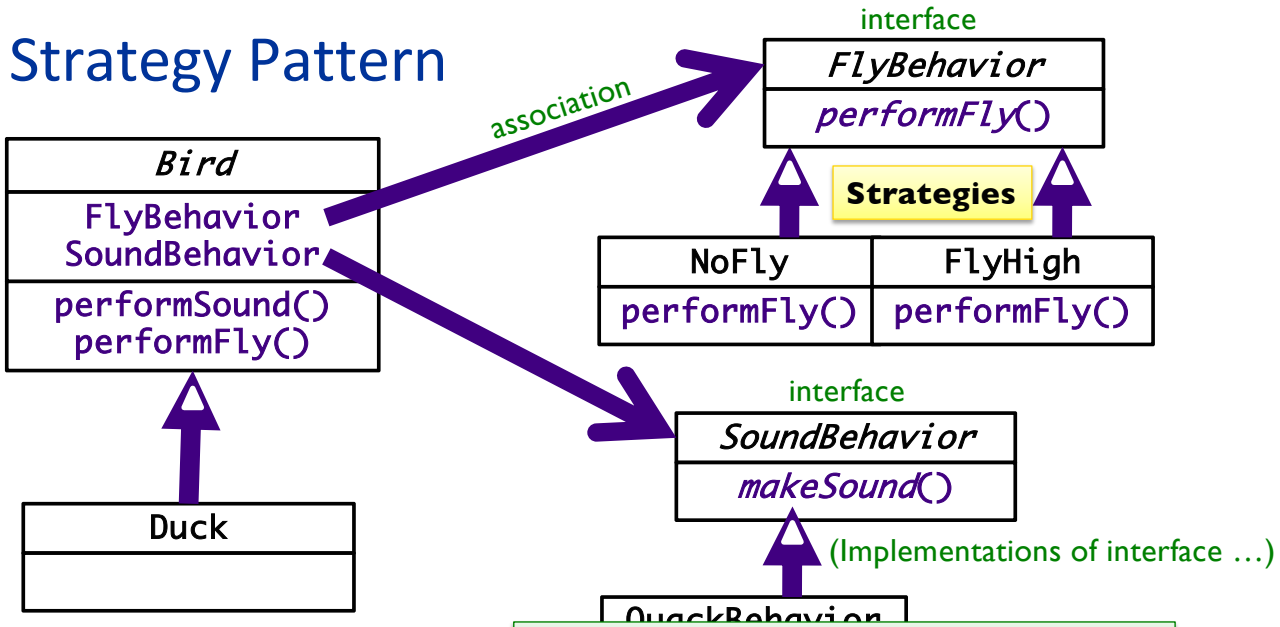
Depend upon Abstractions

Audobon Solution



 UML Diagram

Strategy Pattern



What are the benefits of the Strategy Pattern? Have we seen it in use before?

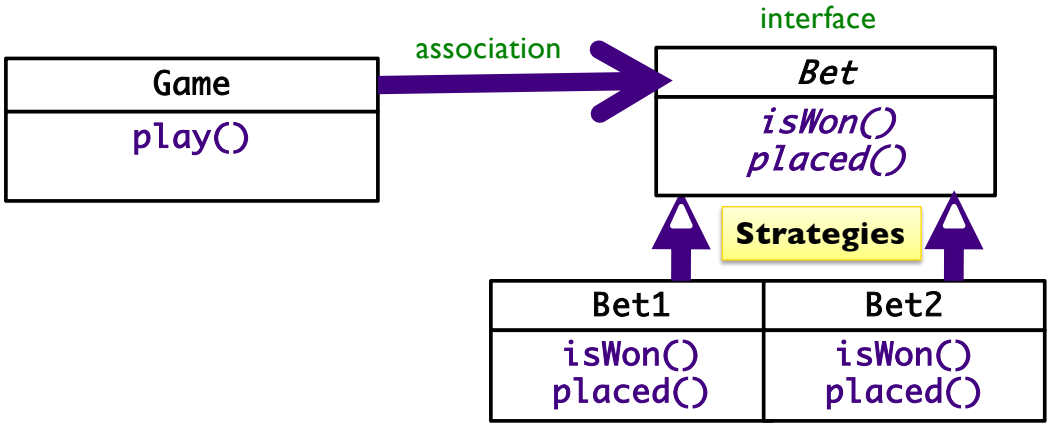
UML Diagram

Design Pattern: **Strategy**

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

Favor ***composition*** over inheritance

Roulette Sketch



Benefits of Solution to Audubon Problem

- Uses ***delegation***
  - Reduces Bird's responsibilities
  - ***Delegate*** some responsibilities to `SoundBehavior` and `FlyBehavior`
  - Reduces Bird's code
- Easy swap of different ***strategy***
  - Can easily plug in different behavior/implementation
    - Others using Bird class are coding to interface, not implementation
- Adheres to open-closed principle

Summary of Design Patterns in Audubon Solution

- Applies ***composition*** pattern
  - Uses-a or Has-a behavior rather than using inheritance
- Applies ***delegation*** pattern
  - Reduces Bird's responsibilities
  - ***Delegate*** some responsibilities to `SoundBehavior` and `FlyBehavior`
  - Reduces Bird's code
- Applies ***strategy*** pattern
  - Can easily plug in different behavior/implementation
    - Others using Bird class are coding to interface, not implementation

Discussion: Applying Design Patterns

- When should we apply the **delegation** pattern?
  - Example, if X, then we should apply the pattern.
- When should we apply the **strategy** pattern?
- When will we know we've gone too far (overapplying)?
  - What are some symptoms to look for?

Discussion: Applying Design Patterns

- When should we apply the **delegation** pattern?
  - When the requirements or implementations for a **responsibility** are likely to *change*
    - Change: Number/types of birds; types of behaviors; or lower-level implementation details
- When should we apply the **strategy** pattern?
  - When there are lots of desired behaviors for one responsibility and they could change
- When will we know we've gone too far (overapplying)? What are some symptoms to look for?
  - "Too small" classes → don't do anything
  - Have many more strategies than necessary
  - "Speculative generality"

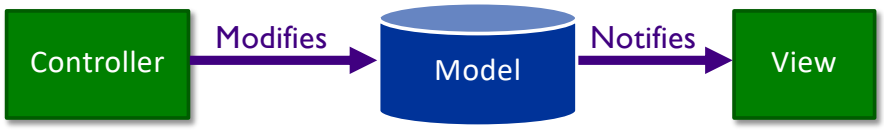
Design Principle: **Loose Coupling**

Goal: loosely coupled designs between objects that interact

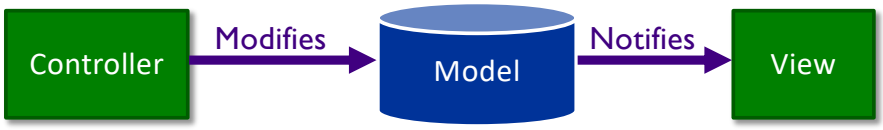
- Loosely coupled objects can interact but have very little knowledge of each other
  - Minimize dependency between objects
  - More flexible systems
  - Handle change

Model - Viewer - Controller (MVC)

- A common **design pattern** for GUIs
- Loosely coupled
  - Model: application data
  - View: graphical representation
  - Controller: input processing

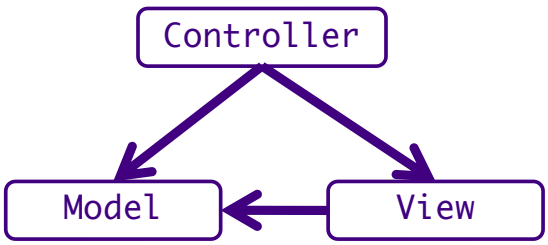


Model-Viewer-Controller

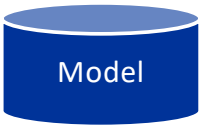


- Can have multiple viewers and controllers
- Goal: modify one component without affecting others

Direct associations

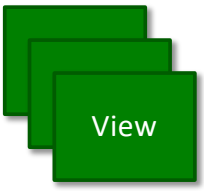


Model

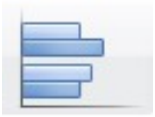


- Represents application state
- Responsible for managing application state
- Purely **functional**
  - Nothing about how view presented to user

Multiple Views

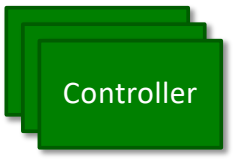


- Provides graphical components for model
  - Look & Feel of the application
- User manipulates view
  - Informs **controller** of change



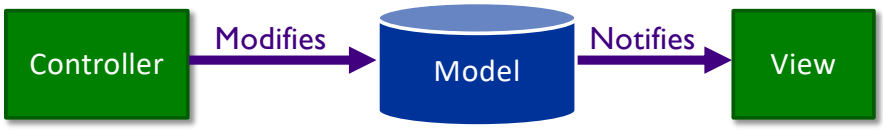
- Example of multiple views: spreadsheet data
  - Rows/columns in spreadsheet
  - Pie chart, bar chart, …

Controller(s)



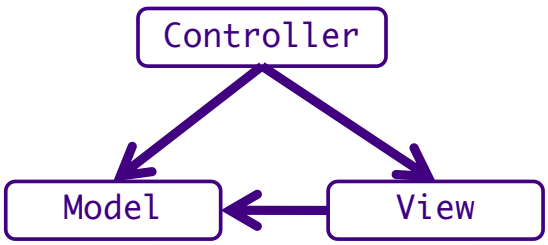
- Handles user input
- Update **model** as user interacts with **view**
  - Call model's methods (often mutators)
  - Makes decisions about behavior of model based on UI
- Views are associated with controllers

Discussion: Map MVC to Goblin Game



- Can have multiple viewers and controllers
- Goal: modify one component without affecting others

Direct associations



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 Wednesday's class
- I will hold office hours during Friday class time

Looking Ahead

- Assignment 7
  - Deadline Thursday at 11:59 p.m.
- Exam: Fri - Sun