Objectives

- Event handling
- Design Patterns
  - Composition
  - Strategy

Review

- What are the main packages of GUIs in Java?
- What are some of the components of Java?
- What are layout managers?
- How do we make our GUIs handle events?

Review: An Anonymous Class Listener

```
void makeButton(String label, final Color bgColor) {
  JButton button = new JButton(label);
  getContentPane().add(button);

  button.addActionListener( new ActionListener() {
         public void actionPerformed(ActionEvent evt) {
            getContentPane().setBackground(bgColor);
         }
      } );
}
```

Anonymous Inner Classes

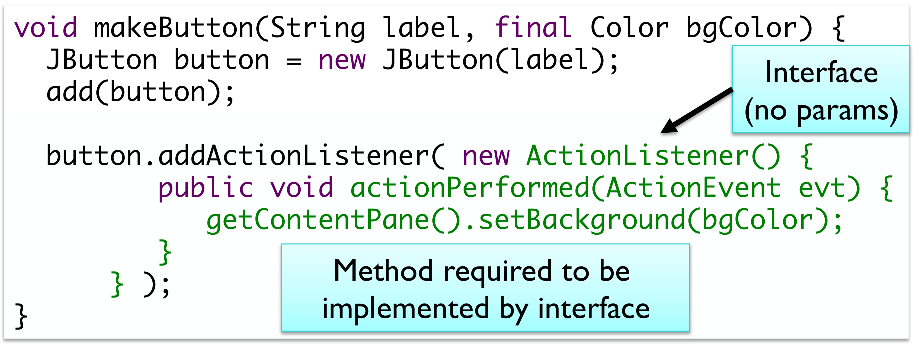
- Confusing syntax!
- Create a new class that implements `ActionListener` interface
  - Define required method, `actionPerformed`, inside braces
- Any needed parameters are inside the parentheses, following the supertype name:

```
new SuperType(construction parameters) {
      inner class methods and data
}
```

Anonymous Inner Classes

- Supertype can be an *interface* or a *class*
  - If an **interface**, inner class implements the interface and required methods
  - If a **class**, the inner class extends that class
- Anonymous inner classes do **not** have constructors
  - Parameters are passed to ***superclass's*** constructor
  - If inner class implements an interface, **no** construction parameters

An Anonymous Class Listener



```
void makeButton(String label, final Color bgColor) {
  JButton button = new JButton(label);
  add(button);

  button.addActionListener( new ActionListener() {
         public void actionPerformed(ActionEvent evt) {
            getContentPane().setBackground(bgColor);
         }
      } );
}
```

Interface (no params)

Method required to be implemented by interface

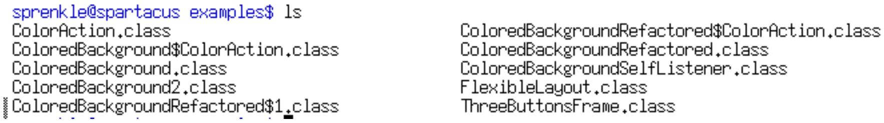
Anonymous Inner Classes

- Differentiate between
  - Construction of a new object of a class
  - Construction of an object of an anonymous inner class that extends that class…

```
// this is a Person object
Person queen = new Person("Mary");

// this is an object of an anonymous
// inner class extending the Person class
Person count = new Person("Dracula") {. . .};
```

Compiler's Names of Classes

- Contents of Eclipse project's `bin` directory examples:



```
sprenkle@spartacus examples$ ls
ColorAction.class                              ColoredBackgroundRefactored$ColorAction.class
ColoredBackground$ColorAction.class            ColoredBackgroundRefactored.class
ColoredBackground.class                        ColoredBackgroundSelfListener.class
ColoredBackground2.class                       FlexibleLayout.class
ColoredBackgroundRefactored$1.class            ThreeButtonsFrame.class
```

Some unusual names. Why?

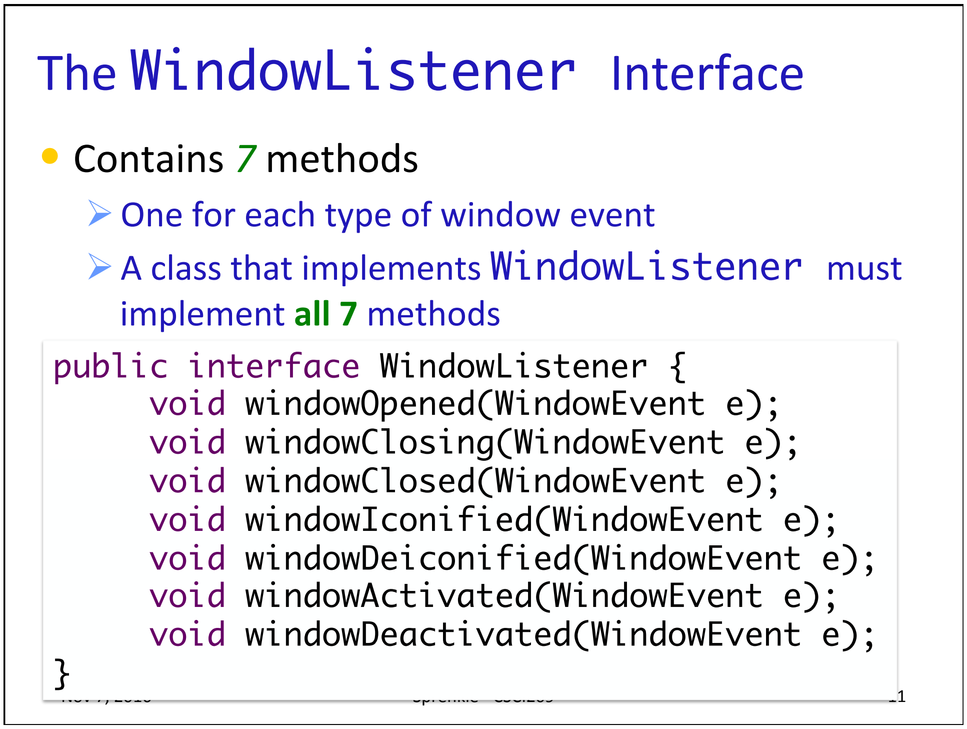
Read Others’ GUI Code

- `CardLayoutDemo`
- `CardLayoutExample`

- Lots of example code and tutorials available online
  - Find something similar to what you want and adapt

Other types of events

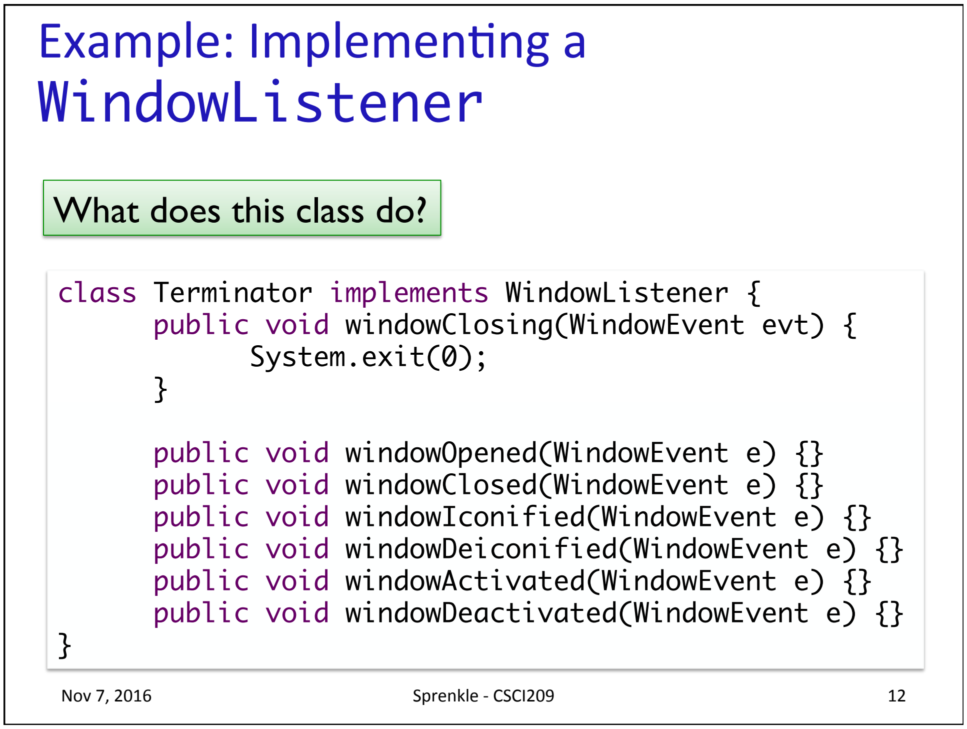
EVENT HANDLING



The WindowListener Interface

- Contains 7 methods
  - One for each type of window event
  - A class that implements WindowListener must implement **all 7** methods

```
public interface WindowListener {
     void windowOpened(WindowEvent e);
     void windowClosing(WindowEvent e);
     void windowClosed(WindowEvent e);
     void windowIconified(WindowEvent e);
     void windowDeiconified(WindowEvent e);
     void windowActivated(WindowEvent e);
     void windowDeactivated(WindowEvent e);
}
```



Example: Implementing a WindowListener

What does this class do?

```
class Terminator implements WindowListener {
      public void windowClosing(WindowEvent evt) {
            System.exit(0);
      }

      public void windowOpened(WindowEvent e) {}
      public void windowClosed(WindowEvent e) {}
      public void windowIconified(WindowEvent e) {}
      public void windowDeiconified(WindowEvent e) {}
      public void windowActivated(WindowEvent e) {}
      public void windowDeactivated(WindowEvent e) {}
}
```

Example: Implementing a `WindowListener`

- Listens for window events on a frame and ends the program when the frame is closing

```
class Terminator implements WindowListener {
      public void windowClosing(WindowEvent evt) {
            System.exit(0);
      }

      public void windowOpened(WindowEvent e) {}
      public void windowClosed(WindowEvent e) {}
      public void windowIconified(WindowEvent e) {}
      public void windowDeiconified(WindowEvent e) {}
      public void windowActivated(WindowEvent e) {}
      public void windowDeactivated(WindowEvent e) {}
}
```

For JFrames use `setDefaultClosedOperation`

Adapter Classes

- Most AWT listener interfaces have a corresponding ***adapter class***
  - Implements each of interface's methods but does nothing inside each
  - No adapter classes for AWT interfaces with only one method (such as `ActionListener`)

Adapter Classes

- If you want a `WindowListener` class that does nothing with most window events
    - Create a new class that **extends** `WindowAdapter` and override relevant method(s)

- When is extending a class a problem?
    - How big of a concern is that for this specific case/ type of class?

Extending an Adapter Class

- Redefine `Terminator` in much less code…

```
class Terminator extends WindowAdapter {
      public void windowClosing(WindowEvent evt) {
            System.exit(0);
      }
      // all other methods are the same as in
      // WindowAdapter–all do nothing.
}
```

Registering a WindowListener

- Register Terminator to listen for window events
  - Assuming that our “main” window frame is named frame
- Result: if frame is closed, the program should exit

```
WindowListener listener = new Terminator();
frame.addWindowListener(listener);
```

Alternative: Registering a WindowListener

```
frame.addWindowListener( new
      WindowAdapter() {
            public void windowClosing(WindowEvent evt) {
                  System.exit(0);
            }
      } );
```

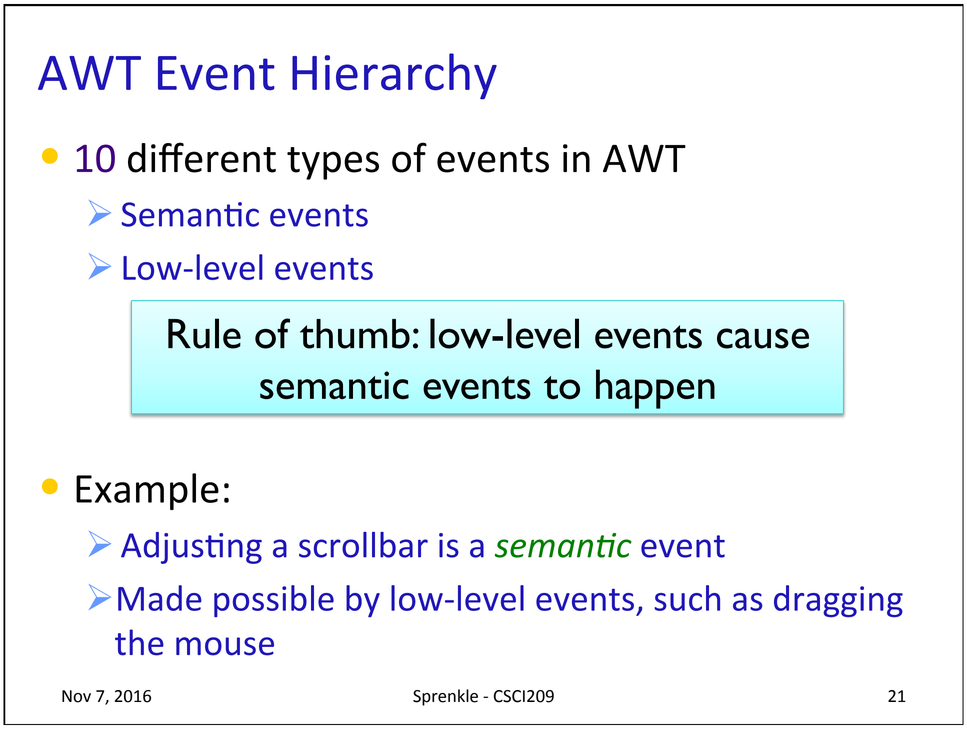
What is going on in this code?

Anonymous Inner Class

```
frame.addWindowListener( new
      WindowAdapter() {
            public void windowClosing(WindowEvent evt) {
                  System.exit(0);
            }
      } );
```

- Defines a new anonymous class that extends `WindowAdapter` class
- Adds `windowClosing` method to anonymous class
- Inherits other 6 methods from `WindowAdapter`
- Creates an object of this new class
  - Object also does not have a name
- Passes new no-name object to `addWindowListener` method of `frame`

TYPES OF EVENTS

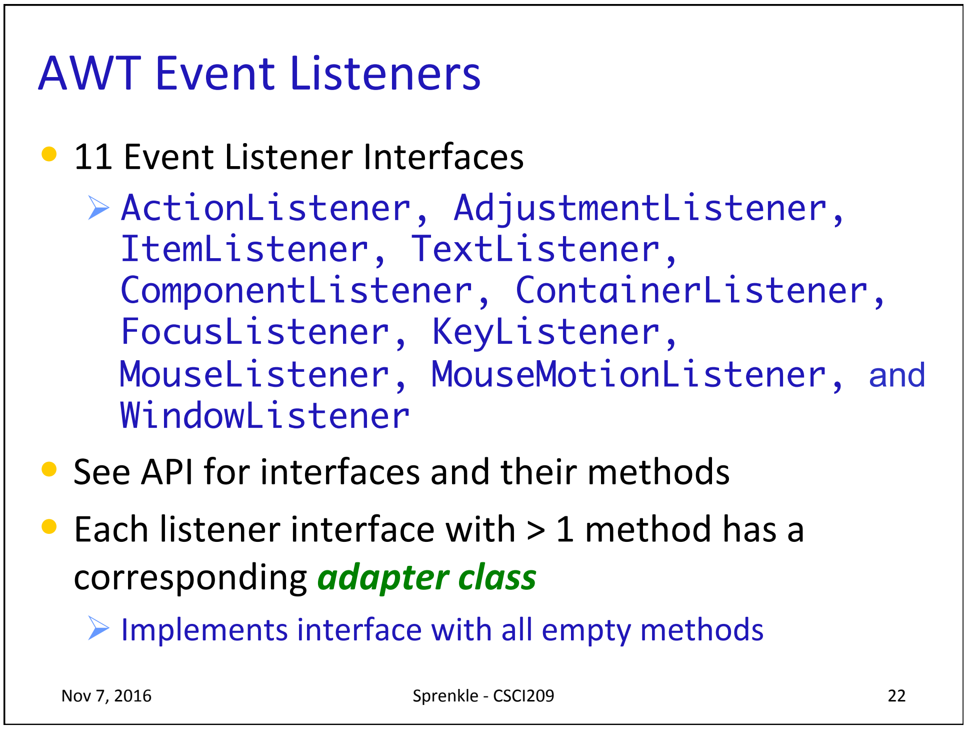


AWT Event Hierarchy

- 10 different types of events in AWT
  - Semantic events
  - Low-level events

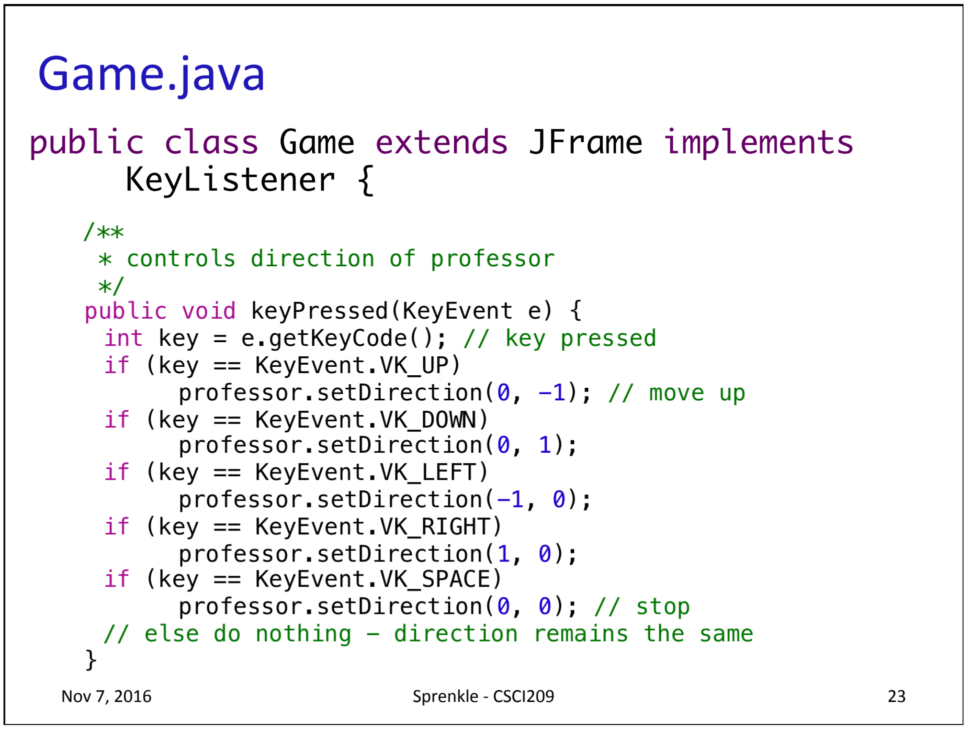
Rule of thumb: low-level events cause semantic events to happen

- Example:
  - Adjusting a scrollbar is a *semantic* event
  - Made possible by low-level events, such as dragging the mouse



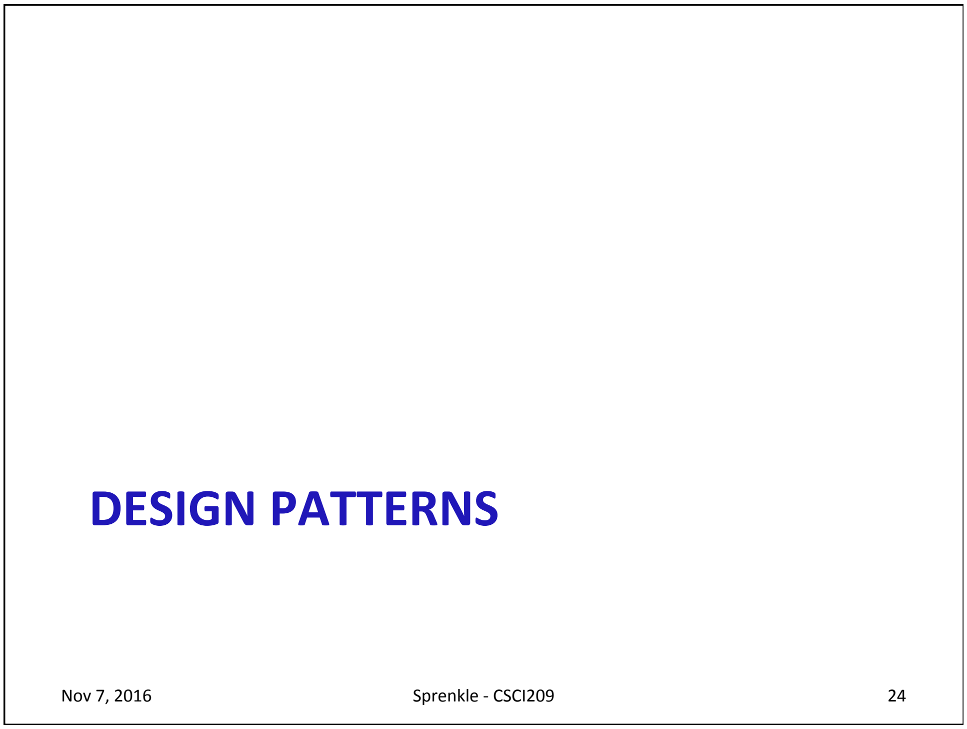
AWT Event Listeners

- 11 Event Listener Interfaces
  - `ActionListener`, `AdjustmentListener`, `ItemListener`, `TextListener`, `ComponentListener`, `ContainerListener`, `FocusListener`, `KeyListener`, `MouseListener`, `MouseMotionListener`, and `WindowListener`
- See API for interfaces and their methods
- Each listener interface with > 1 method has a corresponding ***adapter class***
  - Implements interface with all empty methods



Game.java

```
public class Game extends JFrame implements
     KeyListener {

    /**
     * controls direction of professor
     */
    public void keyPressed(KeyEvent e) {
     int key = e.getKeyCode(); // key pressed
     if (key == KeyEvent.VK_UP)
          professor.setDirection(0, -1); // move up
     if (key == KeyEvent.VK_DOWN)
          professor.setDirection(0, 1);
     if (key == KeyEvent.VK_LEFT)
          professor.setDirection(-1, 0);
     if (key == KeyEvent.VK_RIGHT)
          professor.setDirection(1, 0);
     if (key == KeyEvent.VK_SPACE)
          professor.setDirection(0, 0); // stop
     // else do nothing - direction remains the same
    }
```



DESIGN PATTERNS

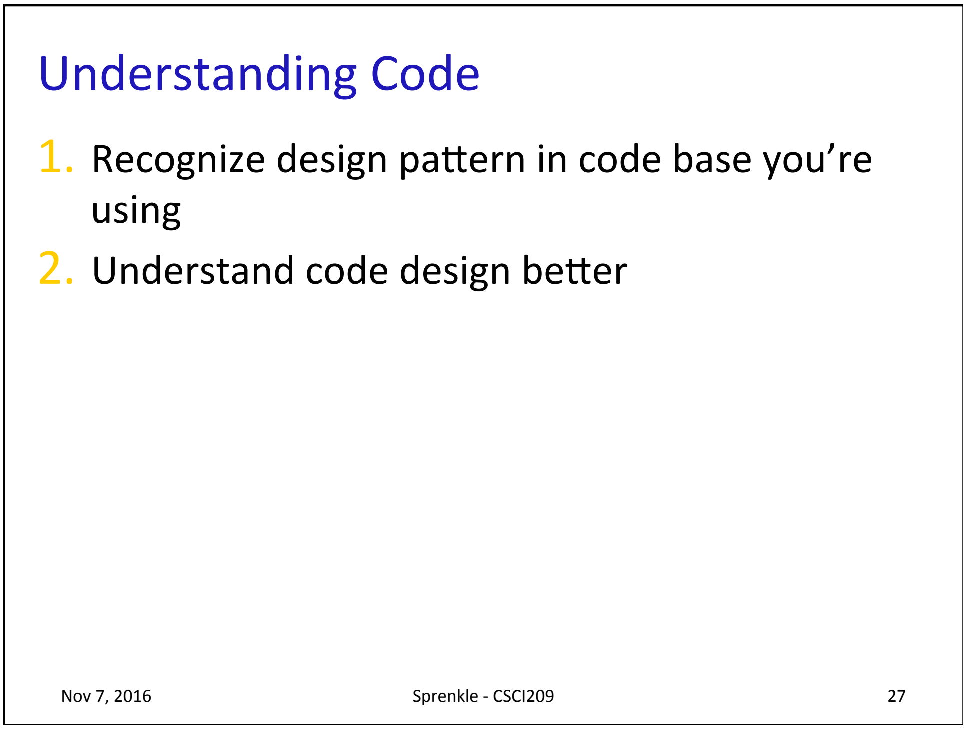
Design Pattern

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

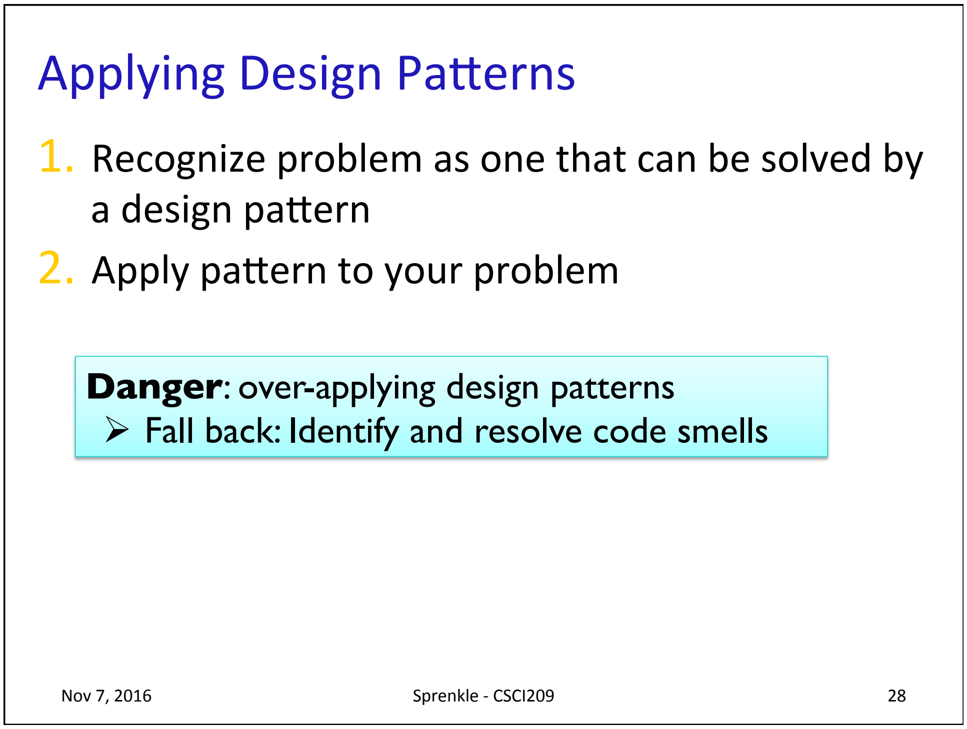
Defined Design Patterns

- 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
  - Didn't design the patterns; identified them



Understanding Code

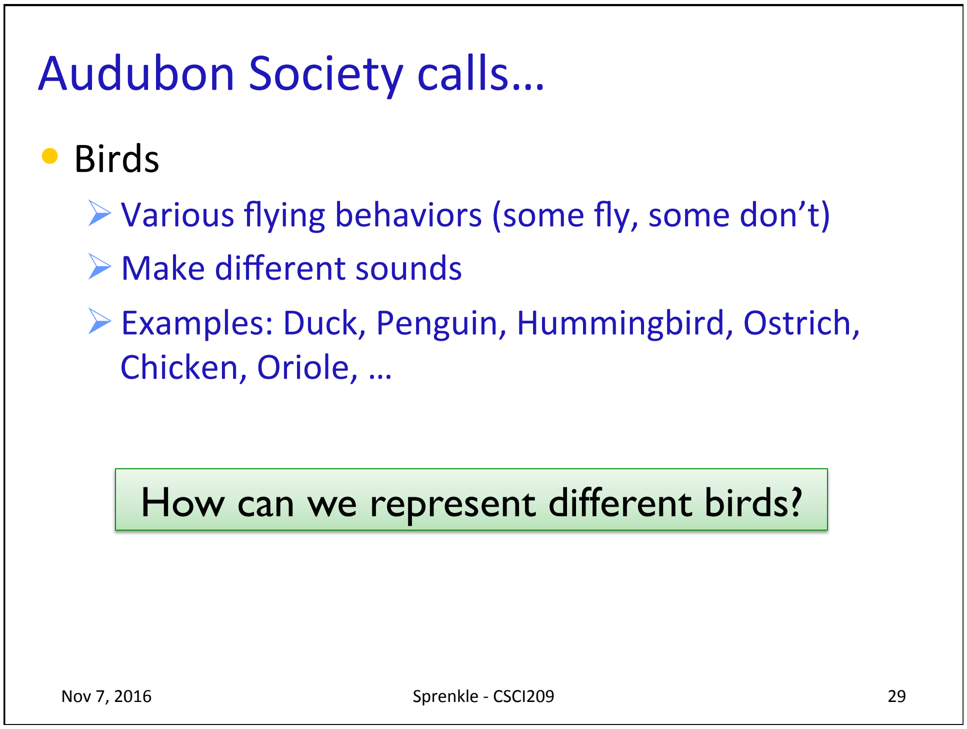
1. Recognize design pattern in code base you're using
2. Understand code design better



Applying Design Patterns

1. Recognize problem as one that can be solved by a design pattern
2. Apply pattern to your problem

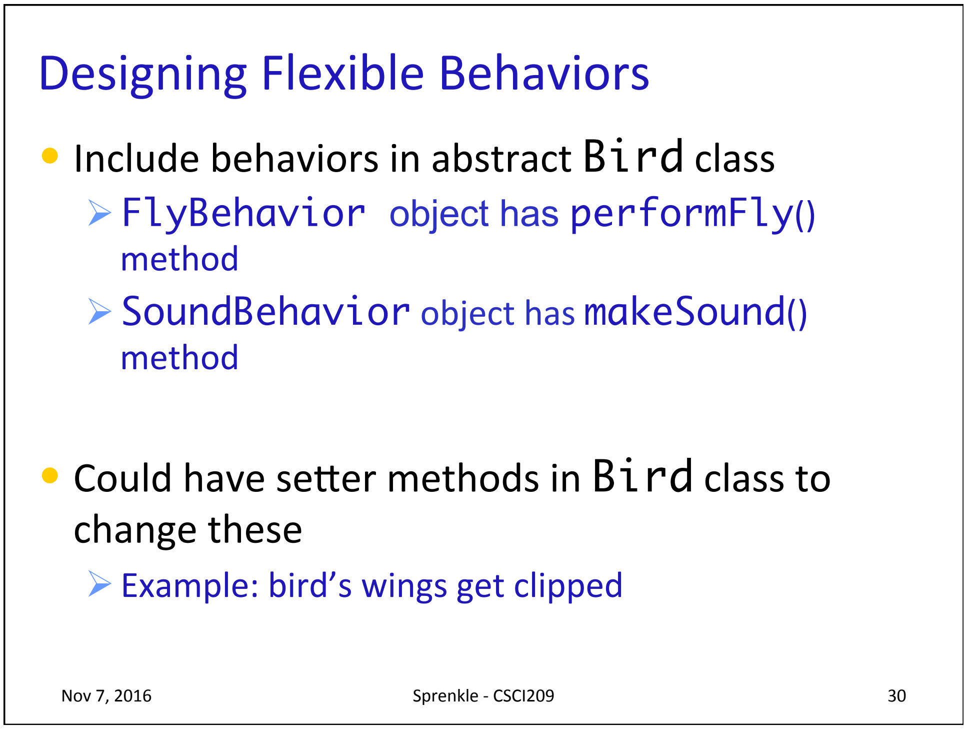
Danger: over-applying design patterns
- ➢ Fall back: Identify and resolve code smells



Audubon Society calls…

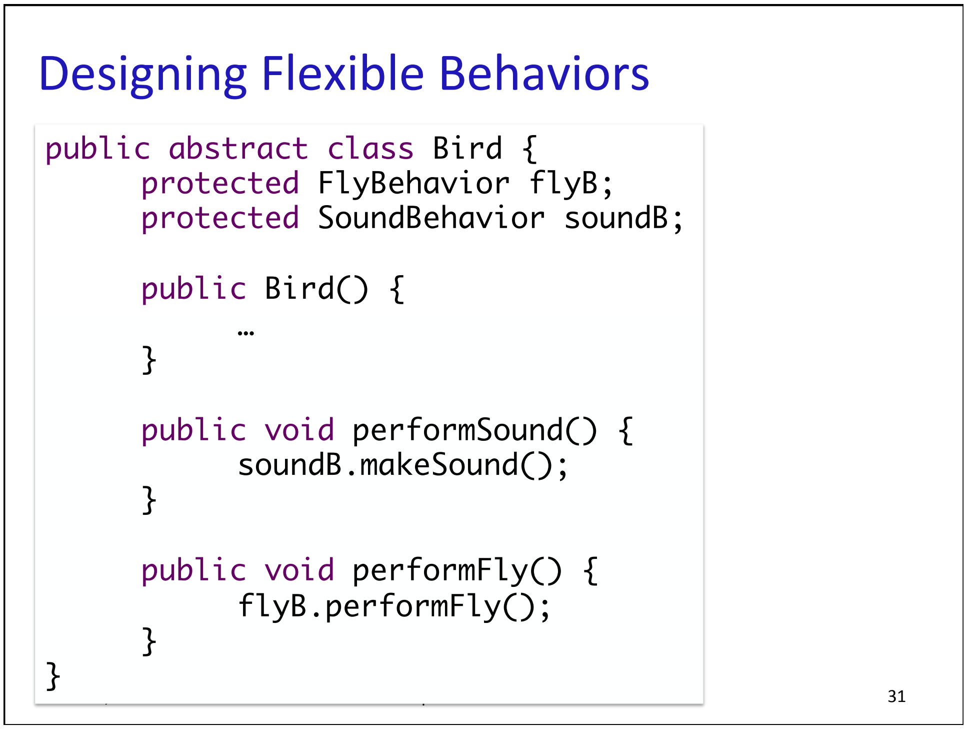
- Birds
  - Various flying behaviors (some fly, some don't)
  - Make different sounds
  - Examples: Duck, Penguin, Hummingbird, Ostrich, Chicken, Oriole, …

How can we represent different birds?



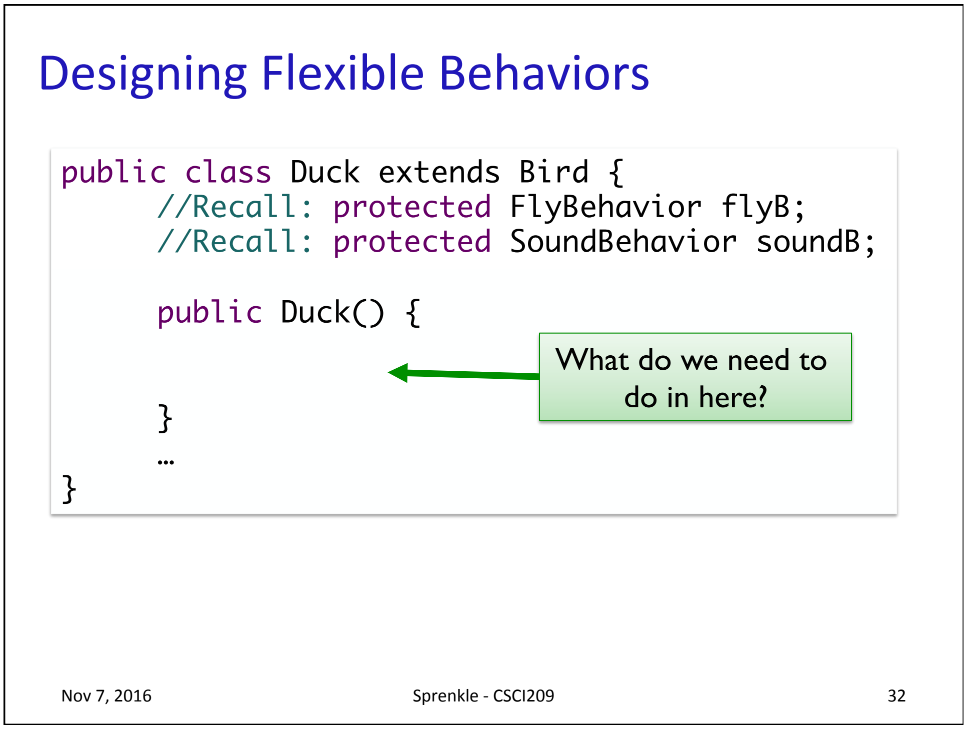
Designing Flexible Behaviors

- 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'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

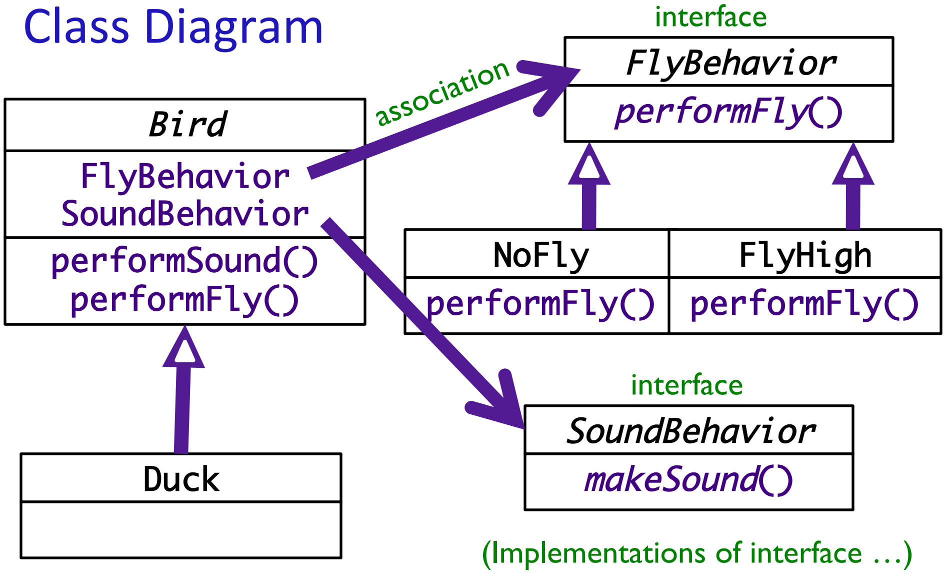
```
public class Duck extends Bird {

     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

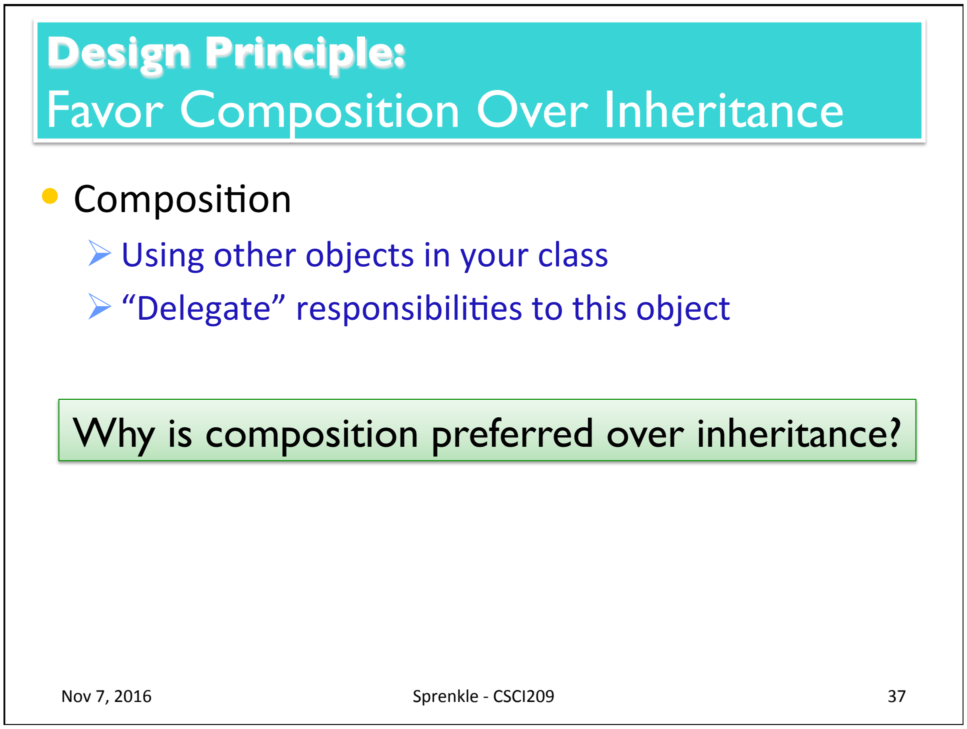


(Implementations of interface …)

UML Diagram

Unified Modeling Language (UML)

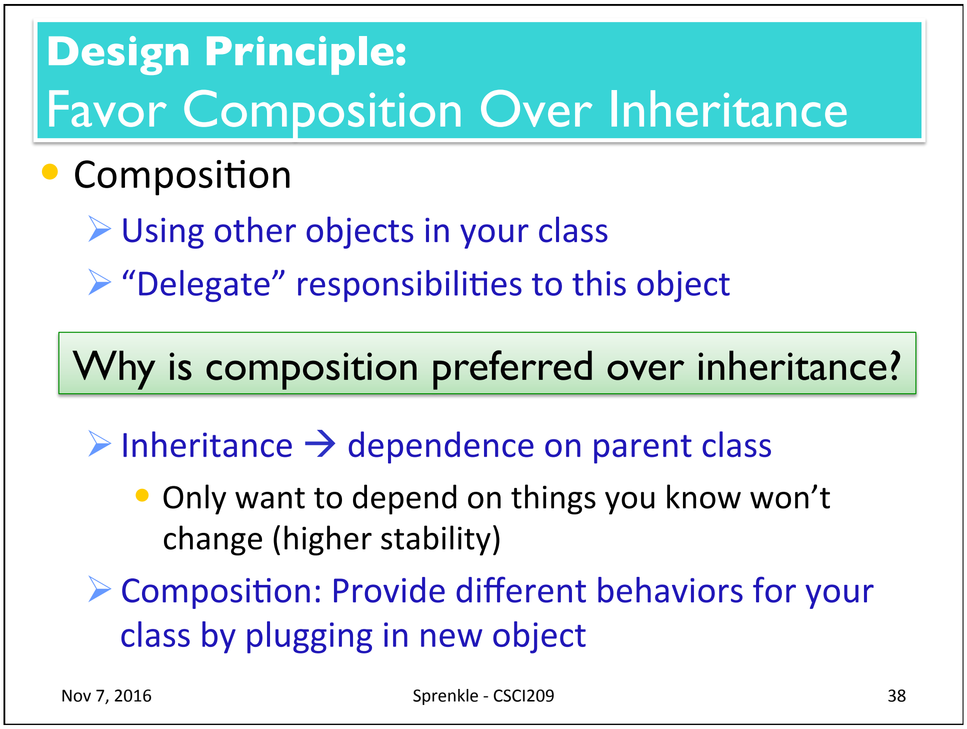
- Standardized general-purpose modeling language
    - Graphical language for visualizing, specifying and constructing the artifacts of a software system
- Includes a set of graphical notation techniques to create abstract models of specific systems
- Used in designing a large system
    - Focus on big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



Design Principle: Favor Composition Over Inheritance

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

Why is composition preferred over inheritance?

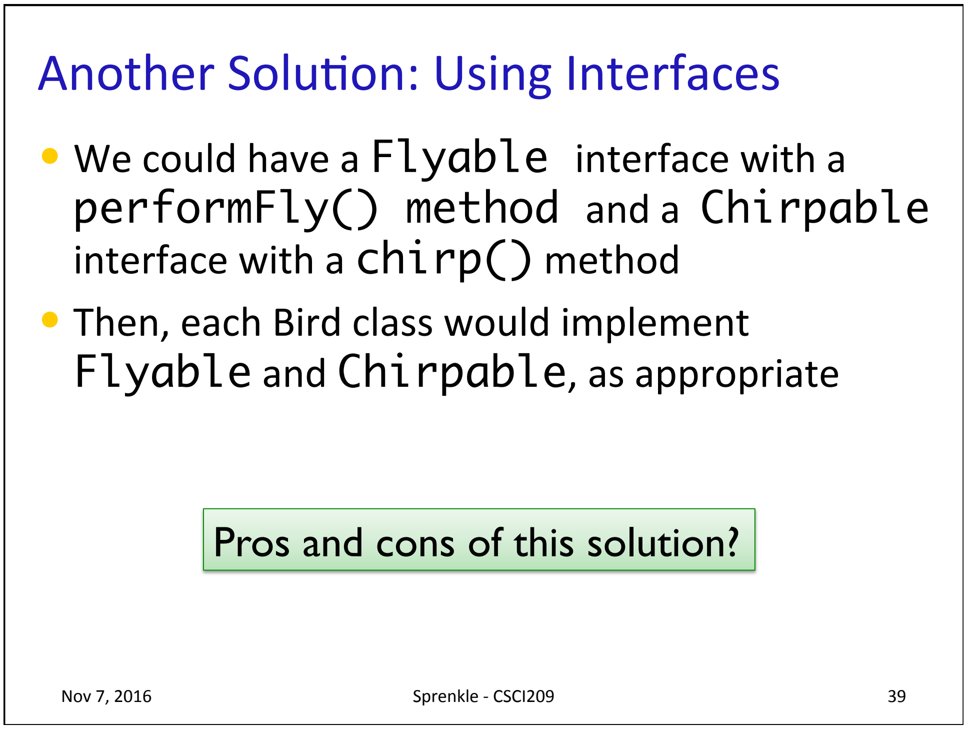


Design Principle: Favor Composition Over Inheritance

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

Why is composition preferred over inheritance?

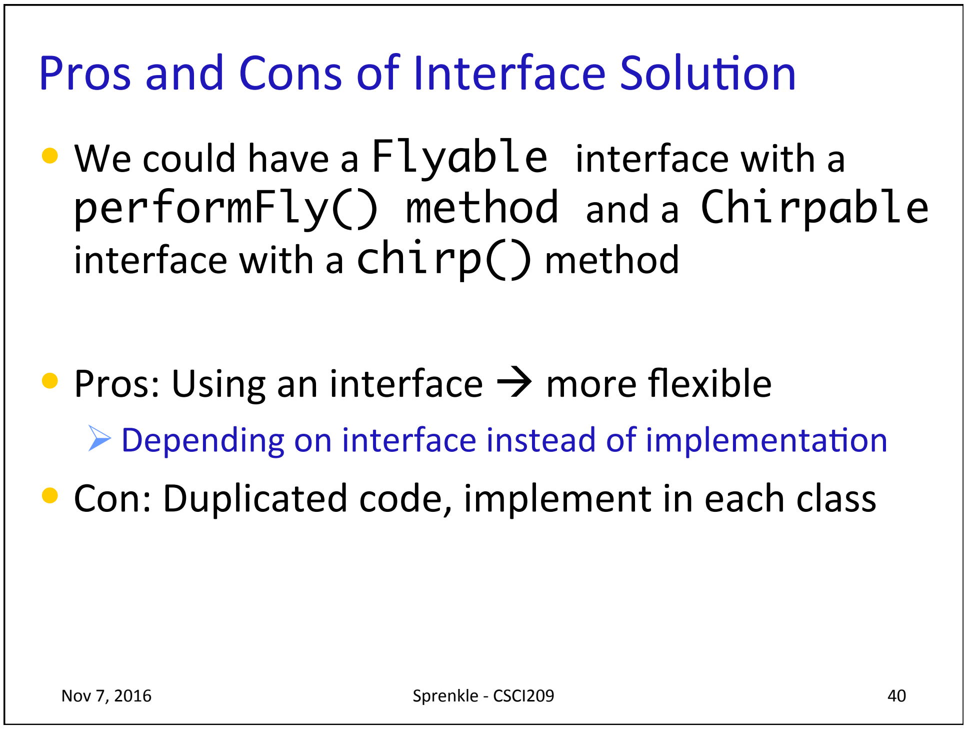
- Inheritance → dependence on parent class
  - Only want to depend on things you know won’t change (higher stability)
- Composition: Provide different behaviors for your class by plugging in new object



Another Solution: Using Interfaces

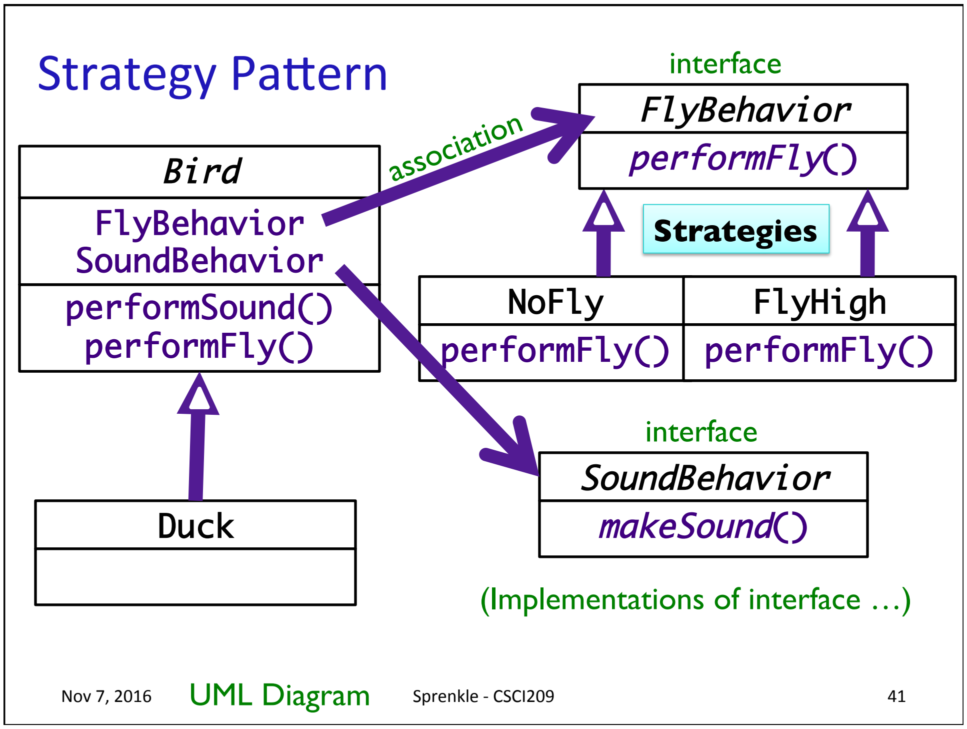
- We could have a `Flyable` interface with a `performFly() method` and a `Chirpable` interface with a `chirp()` method
- Then, each Bird class would implement `Flyable` and `Chirpable`, as appropriate

Pros and cons of this solution?



Pros and Cons of Interface Solution

- We could have a `Flyable` interface 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



Strategy Pattern

interface

FlyBehavior

performFly()

association

Bird

FlyBehavior
SoundBehavior

performSound()
performFly()

Strategies

NoFly

performFly()

FlyHigh

performFly()

Duck

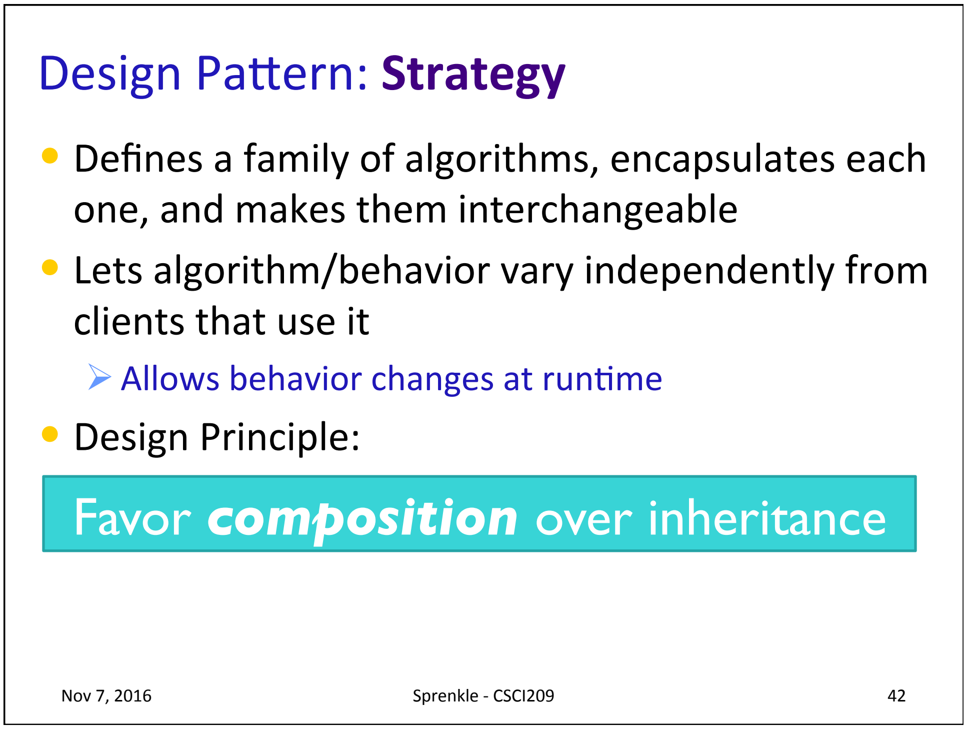
interface

SoundBehavior

makeSound()

(Implementations of interface …)

UML Diagram

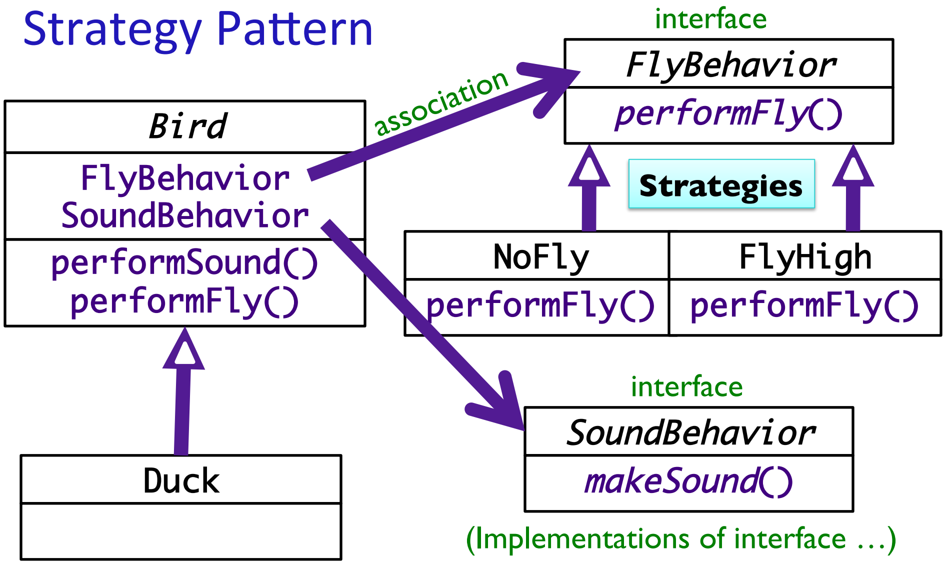


Design Pattern: **Strategy**

- 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

Strategy Pattern



interface

FlyBehavior

performFly()

association

Bird

FlyBehavior
SoundBehavior

performSound()
performFly()

Strategies

NoFly

performFly()

FlyHigh

performFly()

Duck

interface

SoundBehavior

makeSound()

(Implementations of interface …)

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

What Are the Benefits of the Strategy Pattern?

- Uses **delegation**

Pattern in its own right

  - Reduces Bird’s responsibilities
    - Delegate some responsibilities to SoundBehavior and FlyBehavior
  - Reduces Bird’s code
- Easy swap of different strategy
  - Because have **one interface**, can easily plug in different behavior/implementation
    - Coding to interface, not implementation
- Adheres to open-closed principle

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 we know that 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
- 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"

Midterm Prep

Document posted online

- Java
  - Streams
  - Comparison with Python
  - Jar files
- Software Development
  - Models
  - Testing
  - Design Principles
  - Code smells
  - Refactoring
- GUI programming
  - Event handling, inner classes

Emphasis: theory
Assignments: implementation

Little programming,
More code understanding

TODO

- Exam 2 Wednesday
- Nov 18 before class: Extra Credit Deadline