Objectives

- Design Patterns
  - Strategy
  - MVC
  - Factory

1

Review

1. What is common to how we address 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?
   a) What is an example of a design pattern?
4. What was the solution to the Audobon Society's bird modeling problem?
   a) Why is composition preferred over inheritance?

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
- Derived types can then be switched out without consequence

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

Design Principle: Favor Composition Over Inheritance

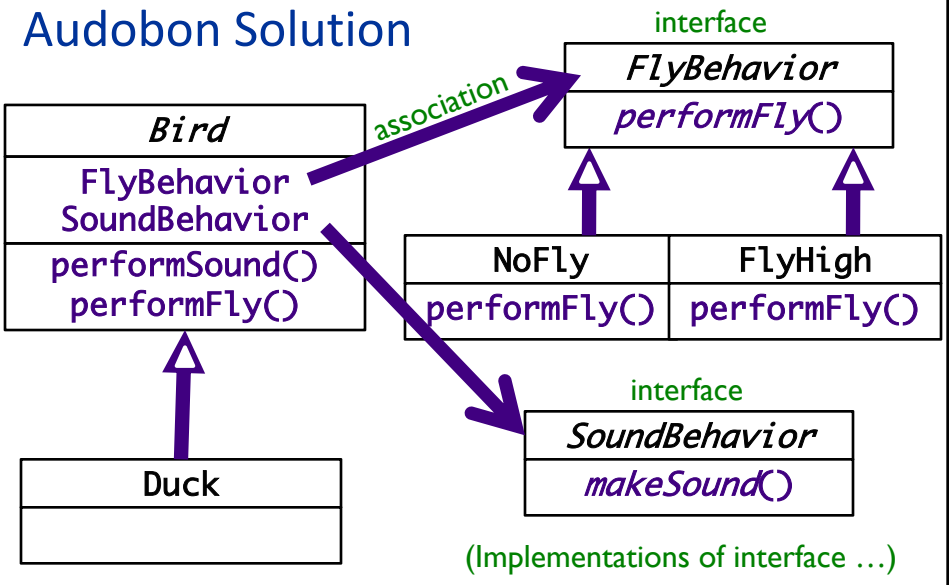
- Design Pattern: Composition
  - Using other objects in your class
  - "Delegate" responsibilities to this object

Why is composition preferred over inheritance?

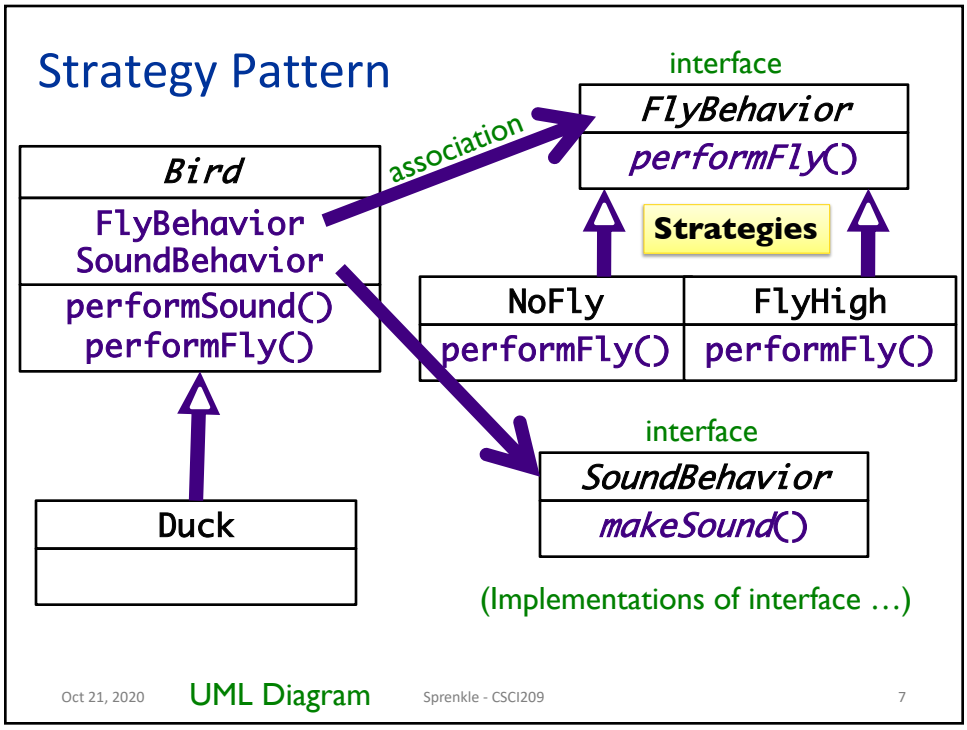
- Inheritance → dependence on parent class
  - Only want to depend on things you know won't change (higher stability)
- Composition: Provide different behaviors for your class by plugging in new object

5

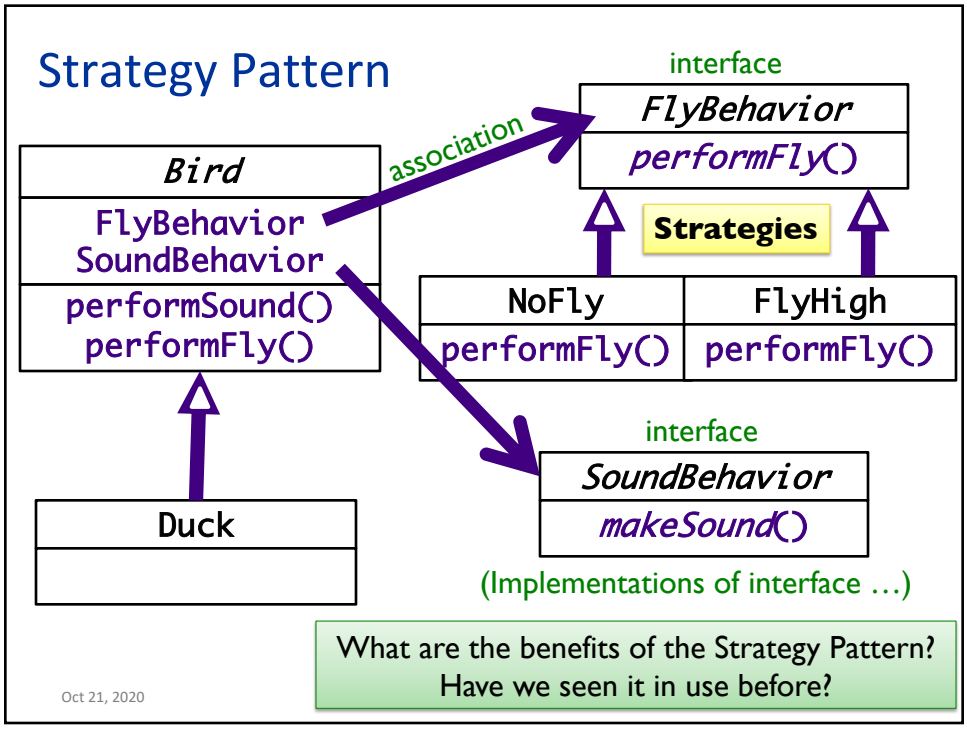
Audobon Solution



6


Strategy Pattern
interface
FlyBehavior
performFly()
association
Bird
FlyBehavior
SoundBehavior
performSound()
performFly()
Strategies
NoFly
performFly()
FlyHigh
performFly()
interface
SoundBehavior
makeSound()
Duck
(Implementations of interface …)
Oct 21, 2020
UML Diagram
Sprenkle - CSCI209
7

7


Strategy Pattern
interface
FlyBehavior
performFly()
association
Bird
FlyBehavior
SoundBehavior
performSound()
performFly()
Strategies
NoFly
performFly()
FlyHigh
performFly()
interface
SoundBehavior
makeSound()
Duck
(Implementations of interface …)
What are the benefits of the Strategy Pattern?
Have we seen it in use before?
Oct 21, 2020

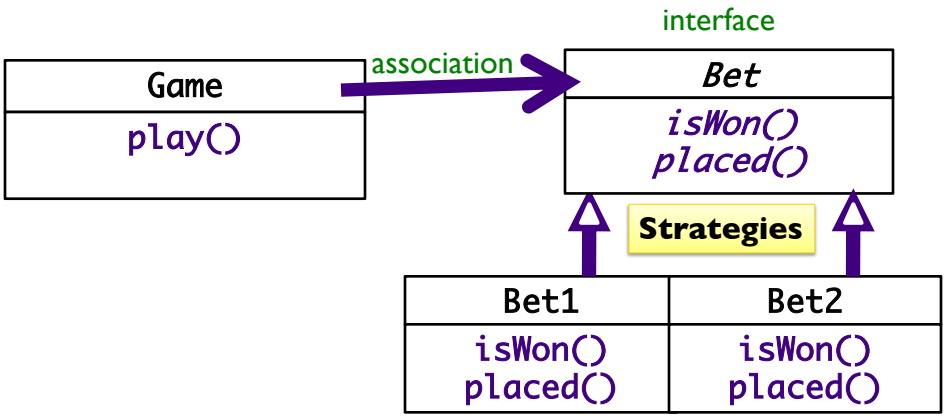
8

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



Benefits of Audobon Solution

- 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: Audobon 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 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 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**

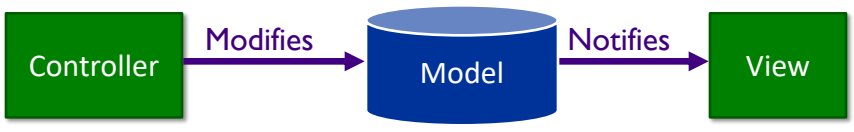
Goal: loosely coupled designs between objects that interact

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

17

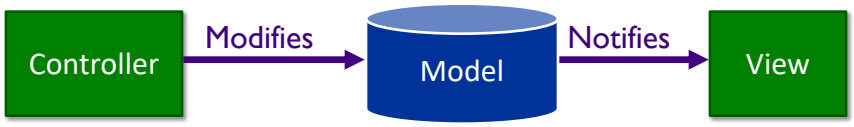
Model - Viewer - Controller (MVC)

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



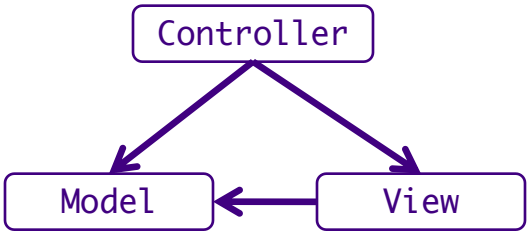
18

Model-Viewer-Controller



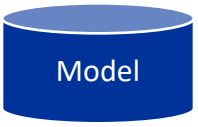
- Can have multiple viewers and controllers
- Goal: modify one component without affecting others

Direct associations



19

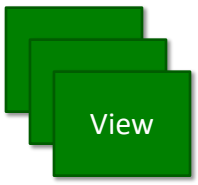
Model



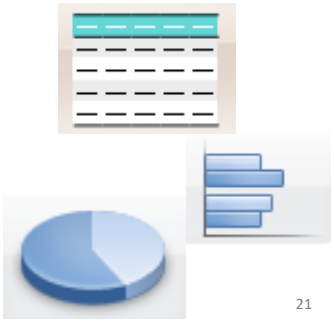
- Represents application state
- Responsible for managing application state
- Purely **functional**
  - Nothing about how view presented to user

20

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

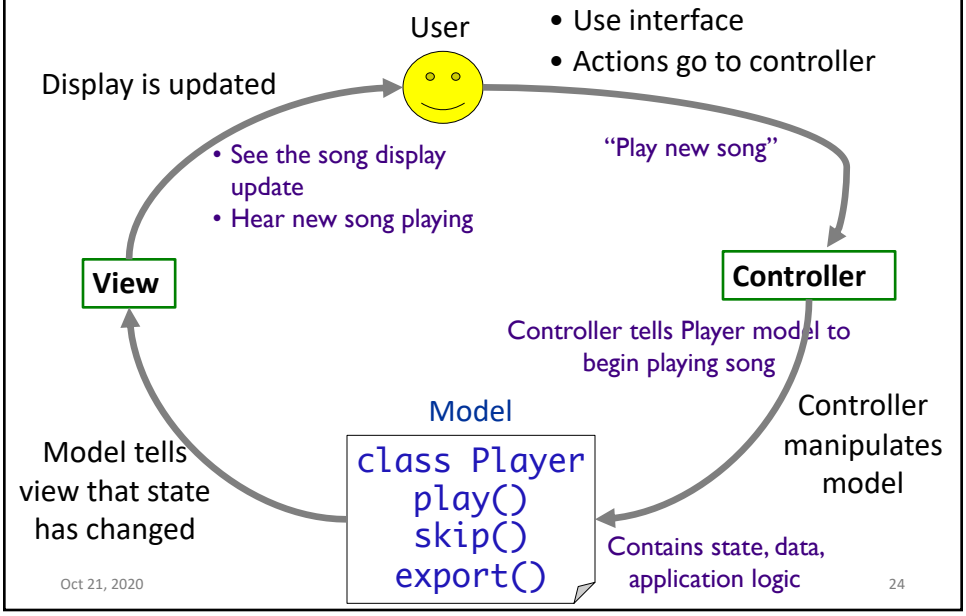
Example: Goblin Game

- Model: GamePiece and child classes
- View-Controller: Game
  - View: displaying locations of model
  - Implemented KeyListener
    - Key strokes made changes to (controlled) the Human

```
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);
        …
```


23

Example: Music Player



24

MVC: Combination of Design Patterns

- Observer
  - Views, Controller notified of Model's state changes
- Strategy
  - View can plug in different controllers
  - Different views of the same model
- Composite
  - View is a composite of GUI components
    - Top-level component learns about model update, updates components
    - A container computes its preferred size by combining all the preferred sizes of its components

Summary: Model View Controller (MVC)

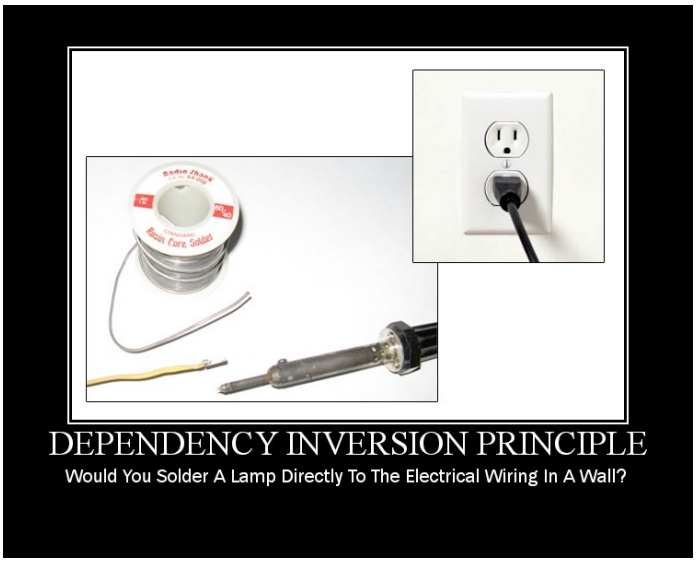
- Common design pattern
  - Used in GUIs, Web Applications
  - Helpful to understand how GUIs are designed
- Combination of design patterns
- Design principles applied
  - Loosely coupled
    - Components are aware of each other but not *too* integrated
  - Depend on abstractions

Dependency Inversion Principle

Depend upon abstractions

"Inversion" from the way you think

27



28

Dependency Inversion Principle

Depend upon abstractions.
Do not depend upon concrete classes.

- High-level components should not depend on low-level components
  - Both should depend on abstractions
  - High-level: more user-facing
  - Low-level: work horses – doing the work/processing
- Abstractions should not depend upon details. Details should depend upon abstractions
- "Inversion" from the way you think

29

FACTORY DESIGN PATTERN

30

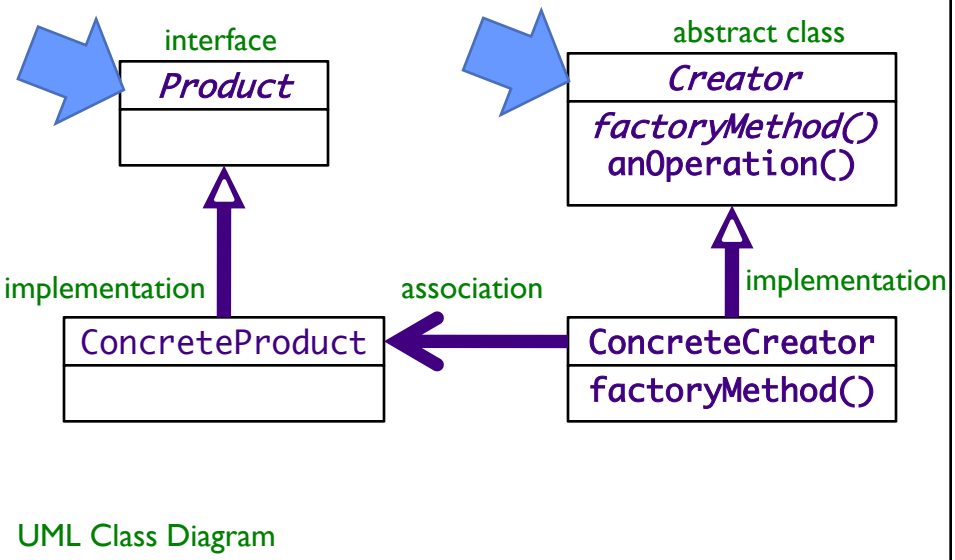
Design Pattern: **Factory Methods**

- Allows creating objects without specifying exact (concrete) class of created object
- Often used to refer to any method whose main purpose is creating objects

- 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

31

Factory Method Pattern



UML Class Diagram

32

Guidelines to Follow DIP

- No variable should hold a reference to a concrete class
  - Using new → holding reference to concrete class
  - Use factory instead
- No class should derive from a concrete class
  - Why? Depends on a concrete class
  - Derive from an interface or abstract class instead
- No method should override an implemented method of its base class
  - Base class wasn't an abstraction
  - Those methods are meant to be shared by child classes

What's a problem with following all of these guidelines?

33

Dependency Inversion Principle

Depend upon abstractions

34

Discussion of Abstraction

- What does abstraction allow?
- Are there any limitations to abstraction?

35

Abstraction Discussion

- Making code abstract makes code easier/more resilient to change
- Examples:
  - Magic number → Constant
    - Change constant (once) → changes value everywhere it is used
  - Long method → Extract method(s)
    - Method call is an abstraction of the concrete statements
      - Can change the implementation of the method without breaking the calling code
  - Large class→Extract class(es)
    - Class encapsulates state/functionality
    - Can change implementation of class and not break the code that uses the class

36

Abstract Discussion

- Abstraction makes it (a little) harder to understand code
  - Examples:
    - Need to look up the value of the defined constant
    - Need to read a called method's API or go to its source to understand what it does
- However, those are relatively low costs and will get cheaper as you get better at coding

Summary of Designing for Change

Use ***abstraction*** for code that is *likely to change*

- Can depend on code that is *stable* and unlikely to change
  - Example of stable code: `System.out`

Design patterns in practice

SCREENSAVERS

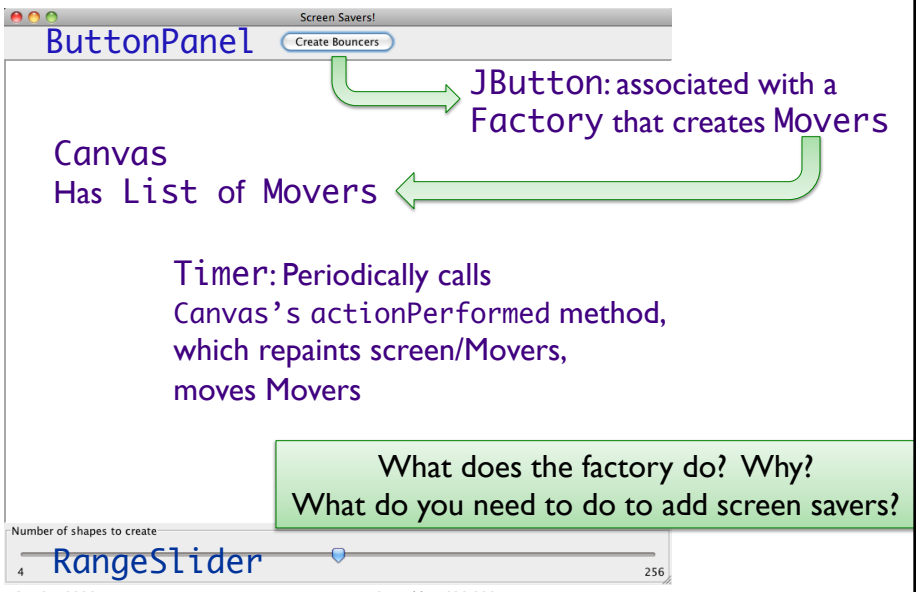
39

Understanding ScreenSavers Code

- How do you run the code?
- What represents an object in the screen saver?
- How are screen saver objects generated?
- How is animation handled?
- How are events handled?

40

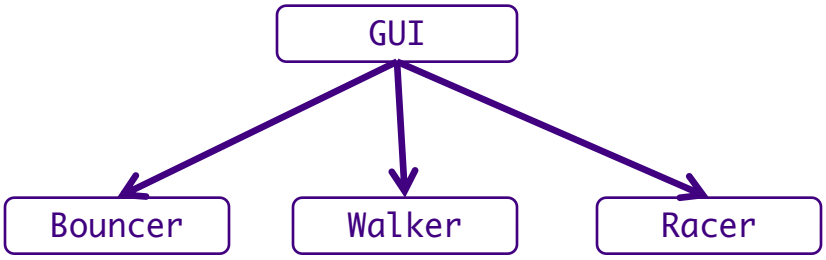
Screensavers GUI/Architecture



Dependency Inversion Principle

- How would you typically build/design the screen saver application?
  - Know we need to view/display a screen saver
    - Buttons, slider, objects that move
    - Top-down
  - Know we need to create a bunch of types of screen savers
    - Abstraction
    - Bottom-up

One Option for Screen Saver Design



Violates Dependency Inversion Principle:
High-level component is dependent on concrete classes.
If implementations change, GUI may have to change

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?

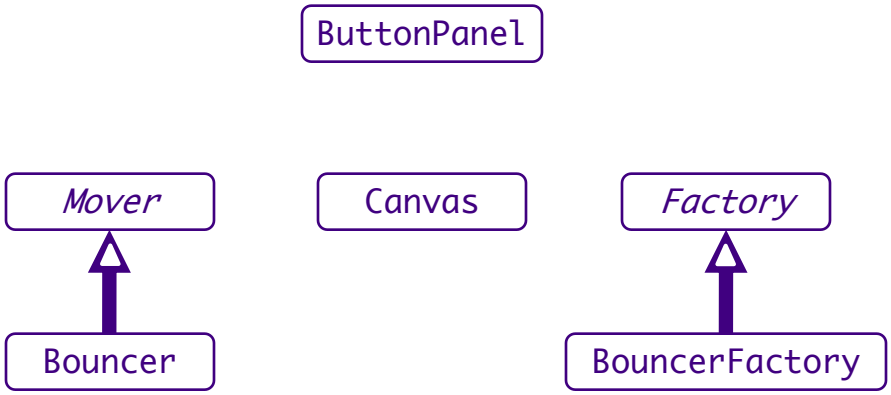
Mapping Factory Design Pattern to Screen Savers

- 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

Thoughts

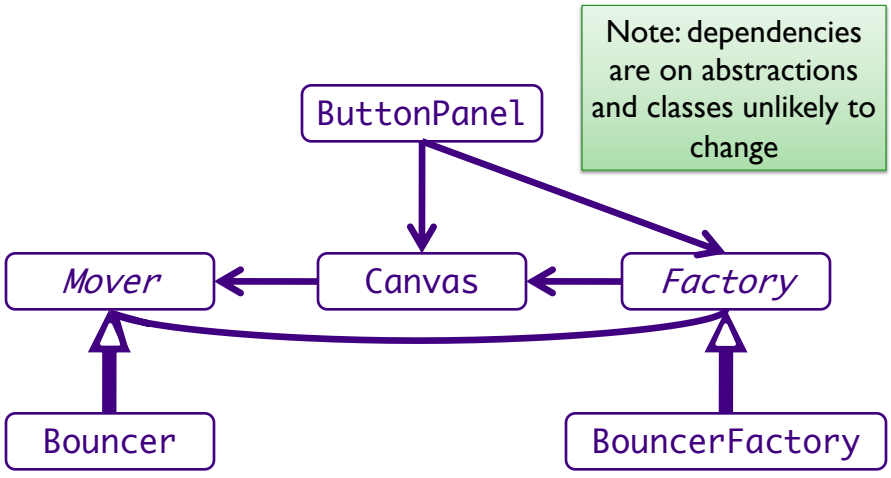
- Didn't need to know design pattern to understand code
  - Helps to know the **terminology** to understand the naming
- Design principles all come down to **where there is change, use abstraction**

Our Screen Saver Dependencies



47

Our Screen Saver Dependencies



48

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

Looking Ahead

- Assignment 8
  - Deadline extended to Thursday at 11:59 p.m.
- Exam: Fri - Sun