Objectives

- Design Patterns

Review

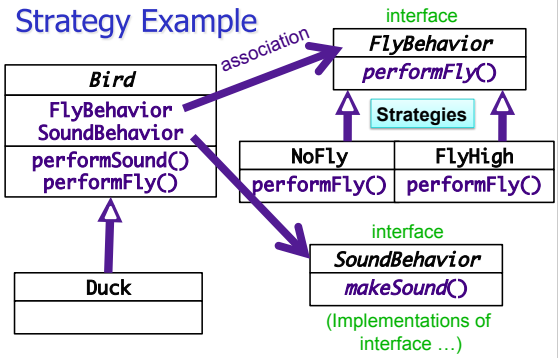
- What is a design pattern?
- What design patterns did we discuss?
  - What design principle does it follow?
- Why do we prefer composition over inheritance?
- What design pattern is used in the screen savers code?

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

Strategy Example



What Are the Benefits of the Strategy Pattern?

What Are the Benefits of the Strategy Pattern?

- Uses **delegation** ← Pattern in its own right
  - Reduces Bird's responsibilities
    - Delegated 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

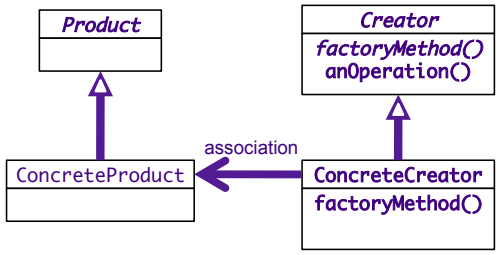
Discussion: Applying Design Patterns

- 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 Flying and Sounds 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 responsbility
- 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"

Review: Factory Method Pattern



UML Class Diagram

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

Notes

- Compiler's names of classes
  - Anonymous class names
    - `ClassName$#.class`
  - Look inside `<workspace_dir>/ScreenSavers/bin/screensaver/nomodify`
- Don't *need* to know design pattern to understand code
  - Helps to know the **terminology** to understand the naming

Dependency Inversion Principle

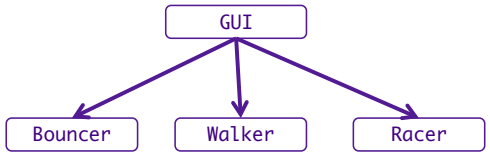
Depend upon abstractions.
Do not depend upon concrete classes.

- High-level components should not depend on low-level components
  - Both should depend on abstractions
- Abstractions should not depend upon details. Details should depend upon abstractions
- "Inversion" from the way you think
- Other techniques besides Factory Method for adhering to principle

Dependency Inversion Principle

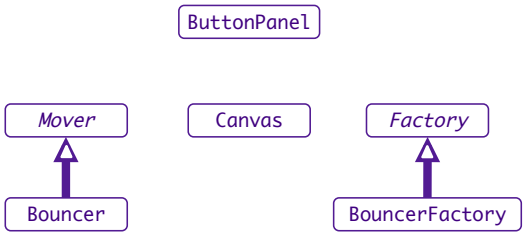
- How would we build/design the screen saver application?
  - Know we need to view/display a screen saver
    - Buttons, slider, objects that move
    - Top-down
  - Know we need to create a bunch of types of screen savers
    - Abstraction
    - Bottom-up

One Option for Screen Saver Dependencies



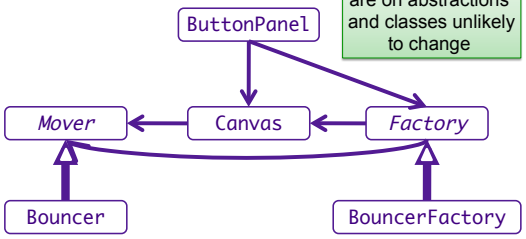
High-level component is dependent on concrete classes.
If implementations change, GUI may have to change

Our Screen Saver Dependencies



Screen Saver Dependencies

Note: dependencies are on abstractions and classes unlikely to change



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 subclasses

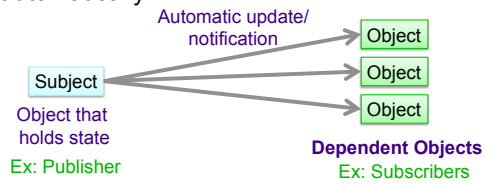
What's the problem with following all of these guidelines?

Dependency Inversion Principle

Depend upon abstractions

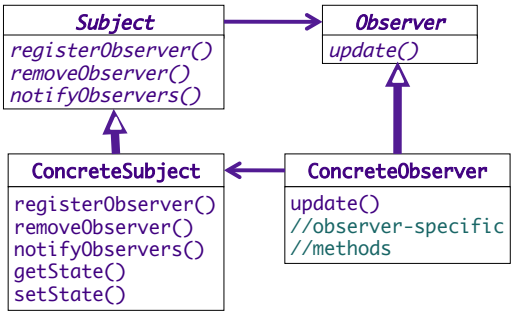
Design Pattern: Observer

- Defines a 1-to-many dependency between objects
- When one object changes state, all of its dependents are notified and updated automatically



Observer Pattern

Have we seen this pattern?



Design Principle: **Loose Coupling**

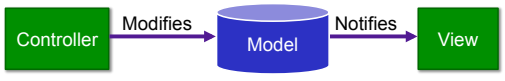
- A principle behind Observer pattern

Strive for loosely coupled designs between objects that interact

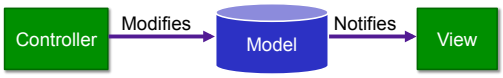
- Loosely coupled objects can interact but have very little knowledge of each other
  - Minimize dependency between objects
  - More flexible systems
  - Handle change

Model - Viewer - Controller (MVC)

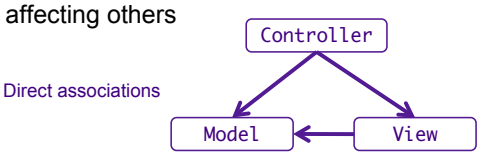
- A common **design pattern** for GUIs
- Separate
  - Model: application data
  - View: graphical representation
  - Controller: input processing



Model-Viewer-Controller



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

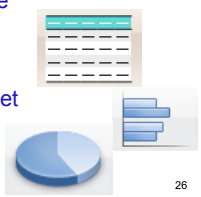


Model

- Code that carries out some task
- Nothing about how view presented to user
- Purely **functional**
- Must be able to register views and notify views of changes

Multiple Views

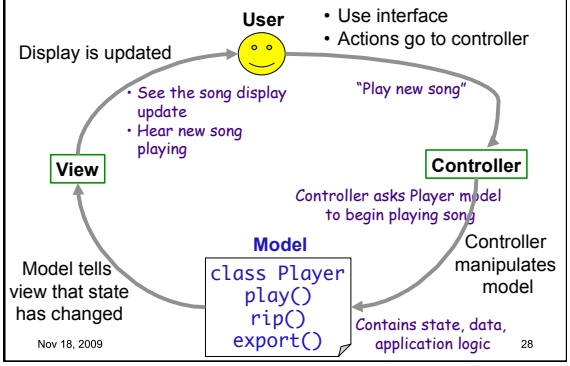
- Provides GUI interface components of model
  - Look & Feel of the application
- User manipulates view
  - Informs **controller** of change
- Example of multiple views: spreadsheet data
  - Rows/columns in spreadsheet
  - Pie chart, bar chart, …



Controller(s)

- Takes user input and figures out what it means to the model
  - Makes decisions about behavior of model based on UI
- Update **model** as user interacts with **view**
  - Calls model's mutator methods
- Views are associated with controllers

Example: Music Player



MVC: Combination of Design Patterns

MVC: Combination of Design Patterns

- Observer
  - Views, Controller notified of Model's state changes
- Strategy
  - View can plug in different controllers
  - View does not know how model gets updated
- Composite
  - View is a composite of GUI components
  - Top-level component learns about update, updates components

Code Analysis

- Consider GUIs we've seen
  - Which use the MVC pattern?
    - Identify M, V, and C in applicable GUIs

Exam Feedback

Grade	Score
A	74-83
B	66-73
C	58-65

- Good:
  - JUnit properties
  - Inner classes
  - Layout Managers
  - Comparing Java and Python
- Not so good:
  - Change → Abstraction
  - Code smells → poor *design*
  - Collection framework → interfaces, implementations, algorithms