Objectives

• 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

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’s wings get clipped

```java
public abstract class Bird {
  protected FlyBehavior flyB;
  protected SoundBehavior soundB;

  public Bird() {
  }

  public void performSound() {
    soundB.makeSound();
  }

  public void performFly() {
    flyB.performFly();
  }
}
```

How Do We Implement...

- Hummingbird?
- Penguin?
- Ostrich?

```java
public class Duck {
  public Duck() {
    flyB = new FlyHighBehavior();
    soundB = new QuackBehavior();
  }
}
```

Class Diagram

- Bird
  - FlyBehavior
    - performFly()
  - SoundBehavior
    - performSound()
    - performFly()

- Duck
  - NoFly
  - FlyHigh
    - performFly()

(interface Implementations of interface …)
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?

- 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: Using an interface → more flexible
  - Depending on interface instead of implementation
- Con: Duplicated code, implement in each class

Strategy Pattern

Bird
  → FlyBehavior
    → performFly()

Duck
  → SoundBehavior
    → performSound()

(Implementations of interface ...)

Interface

FlyBehavior
  → performFly()

NoFly
  → performFly()

FlyHigh
  → performFly()
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**

![Strategy Pattern Diagram]

What Are the Benefits of the Strategy Pattern?

- Uses delegation
  - 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

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?
  - “Too small” classes → don’t do anything
  - Have many more strategies than necessary
  - Speculative generality
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

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?

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

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 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 Dependencies

- GUI
- Bouncer
- Walker
- Racer

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

Our Screen Saver Dependencies

- ButtonPanel
- Mover
- Canvas
- Factory
- Bouncer
- BouncerFactory

Our Screen Saver Dependencies

- ButtonPanel
- Mover
- Canvas
- Factory
- Bouncer
- BouncerFactory

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

Dependency Inversion Principle

Depend upon abstractions

What’s a problem with following all of these guidelines?
To Do

- Assign 11: Screensavers due Friday