Design Patterns B
Reid Holmes
GoF design patterns

- **Creational**
  - Factory Method
  - Abstract Factory
  - Builder
  - Prototype
  - Singleton

- **Structural**
  - Adaptor-class
  - Adaptor-object
  - Bridge
  - Composite
  - Decorator
  - Facade
  - Flyweight
  - Proxy

- **Behavioral**
  - Interpreter
  - Template Method
  - Chain of responsibility
  - Command
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Visitor
Pattern vocabulary

- Shared vocabulary
  - communicate qualities
  - reduce verbosity
  - focus on design
  - increase understanding
Observer

- **Intent:** Define a one-to-many relationship between objects so that when an object changes state its dependents are updated automatically.

- **Motivation:** To maintain consistency between multiple different objects without tightly coupling them.

- **Applicability:**
  - When you want to compartmentalize modifications to two dependent objects.
  - When you want to publish updates but not couple classes.
Observer

- Structure:

- Participants:
  - Subject: tracks observers and fires updates
  - Observer: subscribes/unsubscribes to subjects, receives updates
Observer

- Collaborations
  - Subjects call observer’s update method when they change
  - Subjects can forward data (push) or just send blank update notifications (pull)

- Consequences:
  - Reduce coupling between subject & observer
  - Support broadcast communication
  - Can result in expensive updates
Observer

- Implementation:
  1. Subjects track observers (abstract class helpful)
  2. Caching updates
  3. Push vs. pull

- Related to:
  - Employed by MVC & MVP.
GWT example

```java
Window.addResizeHandler(new ResizeHandler() {
    @Override
    public void onResize(ResizeEvent event) {
        if (event.getWidth() > event.getHeight()) {
            setPortrait(false);
        } else {
            setPortrait(true);
        }
    }
});
```
Decorator

- Intent: “Dynamically add additional responsibilities to structures.”

- Motivation: Sometimes we want to add new responsibilities to individual objects, not the whole class. Can enclose existing objects with another object.

- Applicability:
  - Add responsibilities dynamically and transparently.
  - Remove responsibilities dynamically.
  - When subclassing is impractical.
Decorator

- Structure

- Participants:
  - Component / concrete component
  - Decorator / concrete decorator
Decorator (code ex)

// the Window interface
interface Window {
    public void draw(); // draws the Window
    public String getDescription();
}

// implementation of a simple Window
class SimpleWindow implements Window {
    public void draw() {
        // draw window
    }
    public String getDescription() {
        return "simple window";
    }
}

// abstract decorator class
abstract class WindowDecorator implements Window {
    protected Window decoratedWindow;

    public WindowDecorator (Window decoratedWindow) {
        this.decoratedWindow = decoratedWindow;
    }
    public void draw() {
        decoratedWindow.draw();
    }
}

public class DecoratedWindowTest {
    public static void main(String[] args) {
        Window decoratedWindow = new HorizontalScrollBarDecorator(
                new VerticalScrollBarDecorator(new SimpleWindow()));
        // print the Window's description
        System.out.println(decoratedWindow.getDescription());
    }
}

// adds vertical scrollbar functionality
class VerticalScrollBarDecorator extends WindowDecorator {
    public VerticalScrollBarDecorator (Window decoratedWindow) {
        super(decoratedWindow);
    }
    public void draw() {
        drawVerticalScrollBar();
        super.draw();
    }
    private void drawVerticalScrollBar() {
        // ..
    }
    public String getDescription() {
        return decoratedWindow.getDescription() + " and vert sb";
    }
}

// adds horizontal scrollbar functionality
class HorizontalScrollBarDecorator extends WindowDecorator {
    public HorizontalScrollBarDecorator (Window decoratedWindow) {
        super(decoratedWindow);
    }
    public void draw() {
        drawHorizontalScrollBar();
        super.draw();
    }
    private void drawHorizontalScrollBar() {
        // ..
    }
    public String getDescription() {
        return decoratedWindow.getDescription() + " and horiz sb";
    }
}
Decorator

- Collaborations
  - Decorators forward requests to component object.
- Consequences:
  - More flexible.
    - (than static inheritance; arbitrary nesting possible)
  - Avoids feature-laden classes.
    - (KISS and add functionality as needed.)
  - Warn: Decorator & component are not identical.
    - (equality can be thrown off because decorator != decorated)
- Negative: Many of little objects.
  - (Lots of small, similar-looking classes differentiated by how they are connected. hard to understand and debug.)
Decorator

- Implementation:
  - 1) Interface conformance. (decorator interface required)
  - 2) Abstract decorator not needed if only one decoration is required.
  - 2) Keep component classes lightweight. (too heavyweight can overwhelm decorators
  - 3) Changing a skin instead of changing the guts. (if component is heavy, consider strategy instead)

- Related to: Decorators are a kind of single-node Composite. Decorators can change the skin, Strategy pattern can change the guts.
Singleton

- Intent: “Ensure a class has only one instance”

- Motivation: For situations when having multiple copies of an object is either unnecessary or incorrect.

- Applicability:
  - Situations when there must be only one copy of a class.
Singleton

- Structure:
  - an instance operation that retrieves the instance.
  - may be responsible for creating instance.
Singleton

- Collaborations
  - All collaboration via instance operation.

- Consequences:
  - Controlled access to instance.
  - Reduced name space.
  - Permits variable number of instances.
  - More flexible than class operation
Singleton

- Implementation:
  1. Ensure a unique instance.
  2. Provide an easy access point.

- Related to:
  - Can be used to create Abstract Factory, Builder, and Prototype.
Activity

› 5 mins:

› Right side: Develop a use for a observer pattern for your system.

› Left side: Develop a usage of a decorator pattern for your system.

› 10 mins (5 / group):

› Match up with team from other side of room. Explain your pattern and how it improves your system’s design.