Lecture 13
Design Patterns

Overview
- Design as trade-offs
- Holder Pattern
- Proxy Pattern
- Delegation

Design in a Nutshell (1/3)
- Up to now, focused on how to program
  - be appropriately lazy: re-use code and ideas
- Design Patterns are proven solutions to common problems
  - used successfully before by others, refined by experience
  - generally language-independent - learn once, apply everywhere

Design in a Nutshell (2/3)
- Increasingly we learn about good design
  - some designs are better than others
  - "better" means, for example:
    - more efficient in space or time required (traditional criteria)
    - more robust, maintainable/extensible
    - more reusable, understandable
  - these are central concerns of Software Engineering
    - discussed in detail in CS32 (CSCI0320)
Design in a Nutshell (3/3)

● There are trade-offs to make everywhere
  o architect balances aesthetics, functionality, cost
  o mechanical engineer balances manufacturability, strength, maintainability, cost
● Need to defend your trade-offs
  o no perfect solution, no exact rules
  o up to now, designs have been rather straight-forward

Designing Takes Experience (1/3)

● Experiences are gained by:
  o doing, using, seeing examples (good and bad)
● Rarely find the final design on first try
  o like writing: you always find a “better” way
  o we changed some design styles for this year’s CS15
  o no one ever stops learning
    ▪ you never stop practicing music, sports, ...
    ▪ Malcolm Gladwell’s pop-psych 10,000 hour “rule” in “Outliers” (but a Princeton study disagrees…)

Designing Takes Experience (2/3)

● CS15 provides you with supervised practice
  o typically, TAs have more experience designing than you
  o but they are still learning from CS15 and other classes
  o guide you towards “best” design, but there are many others – as problems get larger, the number of possible designs increase

Designing Takes Experience (3/3)

● But why is experience useful?
  o know what to do in certain circumstances
  o recognize problems you’ve solved before
  o can remember previous solutions and reuse them
    ▪ i.e., recognizing programming patterns
● As a jump-start, why not make a catalogue of good designs you could read and learn from?
Designing Pattern Bibles

The two “bibles” of Design Patterns:

- The Timeless Way of Building by Christopher Alexander (1979)
  - design patterns in architecture
  - Alexander’s patterns in architecture initiated the study of design patterns in software
- Design Patterns by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (1994) (“gang of four”)

Reuse Designs Rather Than Redesign (1/3)

- Libraries are predefined classes you can reuse
  - components, like architect’s windows
  - examples: cs015.prj, javafx.scene, Demos.Cars
  - like components, no indication on how to use them in a program

Reuse Design Rather Than Redesign (2/3)

- Patterns are more general than libraries
  - specify some relationships between classes
  - one pattern may represent many interacting classes
  - general, so they must be applied to specific problem
  - no actual code re-use
- Progression in abstraction/generality vs. code re-use, from concrete classes to abstract classes, to interfaces (no code-reuse, just contract for methods) to patterns (idea re-use)
- Pattern name, abstract, and key aspects of a design pattern’s structure

Reuse Design Rather Than Redesign (3/3)

Example Pattern
- Name: Colonial Revival
- Abstract: example of 18th-century architecture
- Key Aspects: typically white or gray wood, facade shows symmetrically balanced windows centered on door, columns on a portico, windows have double-hung sashes and panes,...
You’ve Already Seen Simple Patterns (1/4)

- Constructors
  - **Pattern Name:** initialization
  - **Abstract:** way to ensure objects have proper internal state before use
  - **Key Aspects:** method that first calls `super()` (if the class has a parent) and then sets up instance’s own instance variables

You’ve Already Seen Simple Patterns (2/4)

- Accessor/Mutator methods
  - **Pattern Name:** encapsulation
  - **Abstract:** keeps other objects from changing an object’s internal state improperly
  - **Key Aspects:** use `public` accessor and mutator methods as appropriate to interact with `private` instance variables

You’ve Already Seen Simple Patterns (3/4)

- Composite Objects
  - **Pattern Name:** containment
  - **Abstract:** models objects that are composed of other objects
  - **Key Aspect:** store components as instance variables, initialize them in the constructor through instantiation or association (via parameter passing) and provide access protection through encapsulation

  ** Note: Containment pattern uses initialization and encapsulation patterns

You’ve Already Seen Simple Patterns (4/4)

- Delegation
  - **Pattern Name:** delegation
  - **Abstract:** have a complex object perform an action by telling its components what to do (e.g., moving themselves)
  - **Key Aspect:** writing all of the detailed code for each component would clutter the move method of a complex object. Instead, delegate particulars of moving to individual components
Example: Colorized App

- Specification:
  - App with one rectangle, a row of three buttons, and a quit button. One button should be labeled “Red”, the second “Green”, and the third “Blue”. Clicking on a button should set the corresponding color to be the current color for drawing the rectangle: any time the user clicks on the rectangle after clicking a color button, the color of the rectangle should be set to the specified current color. The rectangle should be white by default.

Analyzing the New Specification

- Some of this we already know how to do:
  - creating the scene, colors, rectangle, buttons
  - labelling buttons
- What don’t we know?
  - how to model, or hold onto, the “current color”
  - what objects we must create ourselves vs. what we can take directly from JavaFX

More Graphical Containment

- Let’s create a new graphical containment tree!
- Note that graphical and logical containment can be quite different. See next slide for comparison

Graphical vs. Logical Containment

Graphical

Logical
What Type of Buttons?

- We’ll start with buttons - this time for Red, Green and Blue colors.
- Should we use Button or RadioButton?
  - Buttons are used to perform one-time action.
  - RadioButtons are used to set semi-permanent choice (choice stays same until explicitly changed by pressing different RadioButton); also enforce only one choice selected at a time via a ToggleGroup.
  - We only want one color at a time, so we will use RadioButtons.

Modeling “Specified” Color (1/3)

- How do we model the concept of “specified” color?
  - i.e., how do we keep track of the last color clicked?
- Challenge: Colors are immutable. Once created, a Color instance (not instance variable!) models the same color forever.
- RadioButtons need to refer to an object instance which is mutable.

Modeling “Specified” Color (2/3)

- Colorable is an interface for objects which have mutable color (cars, bouncing balls,...)
  - two methods: getColor() and setColor()
- Remember when you had to think about a very similar problem for LiteBrite?

Modeling “Specified” Color (3/3)

- Solution: Define a mutable object (called ColorHolder) which implements Colorable and models concept of current color
  - more generic than in LiteBrite, where Palette had its own instance variable for the current color
  - allows us to “set” and “get” color in one place (our ColorHolder), instead of everywhere that has a color reference.
**ColorHolder**

- **ColorHandler**s for each **RadioButton** will record instances of their own immutable **Color**, and call `setColor(...)` on the associated **ColorHolder** when clicked
- **ClickableRect** will set its current color by calling `getColor()` on instance of **ColorHolder** every time the rectangle is clicked on

**ClickableRect**

- Because we want a rectangle that will support mouse clicks to change color, let’s make a **ClickableRect** similar to the one in the [JavaFX lab](#)
- **ClickableRect** contains an instance variable of **Rectangle** to serve as the actual rectangle
- **ClickableRect** will be associated with **ColorHolder**—this way, it can find out from **ColorHolder** what color to set the **Rectangle** to when clicked

**Syntax: The **ColorHolder** Class**

- Think of **ColorHolder** as a standalone instance variable with set/get
- Each **ColorHandler** tells **ColorHolder** to change its color reference when activated
- The rectangle gets current color from **ColorHolder**
- So, whenever the rectangle is redrawn, it has current color

**Syntax: The **ClickableRect** Class**

- The **ClickableRect** has an instance variable for **Rectangle** and for **ColorHolder**
- The **Rectangle** is associated with a **ClickHandler**
- There is a `getNode` method just like in the JavaFX lab!
- The elided inner **ClickHandler** will update **Rectangle**’s color when **Rectangle** is clicked
**ColorHolder and ColorHandler (1/2)**

- Note: *ColorHolder* implements *Colorable* and stores a reference to a specific `javafx.scene.paint.Color`.
- Each *RadioButton* will have an instance of the same *ColorHandler* that will set *ColorHolder* with a specific color.
  - *ColorHandler* has reference to its unique *Color*, as well as to the shared *ColorHolder*.
- Something to think about when designing the *ColorHandler* (which implements `javafx.event.EventHandler<ActionEvent>` - described in the next two slides):
  - What will *handle* method do?
    - Set color of *ColorHolder* to color stored in *ColorHandler*’s color, e.g., red button’s *ColorHandler* will set *ColorHolder*’s color to red.

**ColorHolder and ColorHandler (2/2)**

- *ColorHandler* will be contained (logically, not graphically – it has no graphical representation, just like other handlers) as a private inner class by *PaneOrganizer* since that is where the *ColorHolder* and the *RadioButtons* are created.

**Syntax: The ColorHandler Class**

```java
private class ColorHandler implements EventHandler<ActionEvent>
{
    private Color _color;
    public ColorHandler(Color color)
    {
        _color = color;
    }

    public void handle(ActionEvent e)
    {
        // Set holder's color (not shape's!)
        _holder.setColor(_color);
    }
}
```

**PaneOrganizer**

- *PaneOrganizer* constructs the *ClickableRect*, the color *RadioButtons* and the quit *Button*.
  - Utilizes a *BorderPane* to organize the different objects.
  - Adds the *ClickableRect* to the top, the color *RadioButtons* to the center and the quit *Button* to the bottom.
Syntax: PaneOrganizer

```java
public class PaneOrganizer {
    private BorderPane _root;
    private ColorHolder _holder;
    public PaneOrganizer() {
        _root = new BorderPane();
        _holder = new ColorHolder(Color.WHITE);
        ClickableRect rect = new ClickableRect(_holder);
        this.setUpButtons();
        // Alignment of objects elided
    }
    private void setUpButtons() {
        // Code elided
        // Inner classes elided
    }
}
```

- The `BorderPane` is used to organize the objects within the `PaneOrganizer`.
- When creating a `ClickableRect`, you need to pass in the `ColorHolder` to create the association.
- We create a `setUpButtons` method to create all the color `RadioButton`s and quit `Button`.

Setting up the Buttons (1/2)

- Want `RadioButton`s all in one row for color buttons - use `HBox`!
- Want only one `RadioButton` to be selected at any time
- Want to create a quit `Button`

Setting up the Buttons (2/2)

- Use `ToggleGroup` for color `RadioButton`s
  - Make `RadioButton`s mutually exclusive
  - still have to add individual buttons to `HBox`
- When creating `ColorHandlers`, we need reference to `ColorHolder`

Syntax: `setUpButtons` Method

```java
private void setUpButtons() {
    HBox buttonBox = new HBox();
    _root.setCenter(buttonBox);
    // Buttons differentiated by different colors and different labels
    RadioButton r1 = new RadioButton("Red");
    r1.setOnAction(new ColorHandler(Color.RED));
    RadioButton r2 = new RadioButton("Green");
    r2.setOnAction(new ColorHandler(Color.GREEN));
    RadioButton r3 = new RadioButton("Blue");
    r3.setOnAction(new ColorHandler(Color.BLUE));
    // First add buttons to `ToggleGroup` for mutual exclusion
    ToggleGroup group = new ToggleGroup();
    r1.setToggleGroup(group);
    r2.setToggleGroup(group);
    // Then add button to `ColorButtonRow`
    buttonBox.getChildren().add(r1);
    buttonBox.getChildren().add(r2);
    buttonBox.getChildren().add(r3);
    // Set up quit button
    Button quitButton = new Button("Quit");
    _root.setButton(quitButton);
    // Size setting, alignment etc. elided
}
```

- By adding the `RadioButton`s into the `HBox`, the `RadioButton`s will all be placed in a horizontal row.
- A `ToggleGroup` enforces mutual exclusion of the `RadioButton`s added to it.
- We have already seen the code for `ColorHandler`. The code for `QuitHandler` is in the next slide.
Syntax: QuitHandler

```java
private class QuitHandler implements EventHandler<ActionEvent>{
    /* This method is called by JavaFX when a click occurs on the quit button. The scene will close. */
    public void handle(ActionEvent e) {
        System.exit(0);
    }
}
```

- We also need to make an EventHandler for the Button that quits our program.
- This is another private inner class of the PaneOrganizer.
- The handle method in QuitHandler is called by JavaFX when a click occurs on the quit Button.

Syntax: PaneOrganizer

```java
public class PaneOrganizer {
    private BorderPane _root;
    private ColorHolder _holder;
    public PaneOrganizer() {
        _root = new BorderPane();
        _holder = new ColorHolder(Color.WHITE);
       ClickableRect rect = new ClickableRect(_holder);
        _root.setTop(rect.getNode());
        this.setUpButtons();
    }
    private void setUpButtons() {
        // Code elided
    }
    public BorderPane getRoot() {
        return _root;
    }
}
```

- Make sure to have a getRoot() method in order to access the PaneOrganizer's root in the App class.

Holder Pattern (1/3)

- Provide object that acts as **holder** for another object, called the **subject** (current color in our case)
  - acts as placeholder for a subject that many other objects might reference
  - holder object is "stable," the instance referenced by other objects does not change
  - holder can change what subject it is referencing, including potentially instances of a different subclass, without affecting those objects that reference the holder itself

Holder Pattern (2/3)

- **holder** object:
  - contains object for which it manages changes (can also be associated with it, like our ColorHolder is with initial color).
  - provides one level of indirection to subject instance
  - provides only accessor/mutator methods
**Holder Pattern (3/3)**

- **Advantages:**
  - easily change the object that many clients reference because those objects only refer to holder
  - provide different interface to subject
    - e.g., subject may be immutable, but holder provides mutable interface
- **Disadvantages:**
  - requires extra class, and thus, extra delegation (more on this later)

**Generic Structure of Holder (1/3)**

![Generic Structure of Holder Diagram](image)

- **Holder** represents our `ColorHolder`
  - provides mutable class to model immutable `Color` by maintaining reference to correct `Color` instance
  - other objects that must reference `Color` can only do so using `ColorHolder`
- **Subject** represents what the holder holds, i.e., a reference to a `javafx.scene.paint.Color` which can change (if we click on a new color). Changed via mutator (set method).

**Generic Structure of Holder (2/3)**

**Generic Structure of Holder (3/3)**

- **Clients** represent `ColorHandler` and `ClickableRect`
  - modify `Color`
  - delegate the changing `Color` to `ColorHolder`
    - `ColorHolder` keeps track of current `Color`

Note: Holder doesn't have to logically contain/construct the subject – our `ColorHolder` is an example of the holder receiving an association to its subject during construction.
Delegation (1/2)
- Delegation occurs when one class “hands off” responsibility to another class
  - allow run-time flexibility not possible using static class relationships like inheritance or containment
  - message can be sent to many objects, or augmented, each time it is forwarded
  - thus, provides extra flexibility and extensibility
    - e.g., button class delegates responses to its handler class
- Used in every design pattern we will see

Delegation (2/2)
- General structure of Delegation on right
- Here, Client can be delegated in two ways:
  - to ObjectA or ObjectB directly
  - to ObjectC via Intermediate

Another Example: RectangleMover
- Now we want to model movable rectangles! Here’s the spec:
  - Design a program with two rectangles and a grid of buttons. The two rectangles should be colored and placed differently to distinguish them. The buttons should be labeled “Up”, “Down”, “Left”, and “Right”. Clicking on a button moves the current rectangle in the direction denoted by the button. Clicking on a rectangle makes the rectangle the current rectangle.

Specification Analysis
- Some things should be simple:
  - Stage, Scene, and Pane
  - Buttons
  - Rectangle
- More difficult things:
  - concept of a current rectangle
  - making a MovableRect
**Designing MovableRect**

- **MovableRect** will contain a **Rectangle**, like in the last example
  - Will also have a private class **MouseHandler**, like the previous example’s **ClickHandler**

- **moveDown()** method
  - Method should move **Rectangle** down by dy pixels
  - Move shape by getting old location and translating it down dy pixels
  - **setY** will only change y coordinate, and x coordinate will remain the same

- Other move methods are similar

**Syntax: MovableRect (1/3)**

```java
public class MovableRect {
    private Rectangle _rectangle;

    public MovableRect() {
        _rectangle = new Rectangle(Constants.WIDTH, Constants.HEIGHT);
        _rectangle.setOnMousePressed(new MouseHandler());
    }
}
```

- The **MovableRect** class contains and sets up the **Rectangle** that we manipulate from the **PaneOrganizer**, like in the previous example

**Syntax: MovableRect (2/3)**

```java
public void moveUp() {
    //remember coordinate system starts in upper left corner
    _rectangle.setY(_rectangle.getY() - Constants.DY);
}
public void moveDown() {
    _rectangle.setY(_rectangle.getY() + Constants.DY);
}
public void moveRight() {
    _rectangle.setX(_rectangle.getX() + Constants.DX);
}
public void moveLeft() {
    _rectangle.setX(_rectangle.getX() - Constants.DX);
}
public Node getNode() {
    return _rectangle;
}
//location and fill setter methods elided
```

**Current Rectangle: Simple Idea**

- First idea: maintain a **currRect** instance variable in **PaneOrganizer** and update when a **MovableRect** is clicked
  - This would work, however...

- There’s a more extensible way that allows multiple classes to access the **current rectangle** without having to update multiple references
The Proxy Pattern (1/2)

- Current rectangle sounds like current color from holder pattern
  - could use a RectangleHolder
  - rectangles would put reference to themselves in holder when clicked
  - buttons get current rectangle from holder and call messages on it
- From a design perspective, gives a lot of work to the handlers
  - would like to avoid having the handlers get a current rectangle from the holder

The Proxy Pattern (2/2)

- Alternative: use a Proxy! Proxy acts on behalf of another subject
  - has a reference to the actual object instance
  - reference can change
  - all clients know only proxy, proxy only knows subject
  - proxy has methods that match those of subject (but does not necessarily have all of the subject’s methods)
  - clients call methods on proxy, proxy forwards methods to subject
  - can control what methods can be called on underlying object that proxy models; in short, commands are fielded by the proxy interloper.

Current Rectangle: Using Proxy (1/2)

- MovableRectProxy acts on behalf of MovableRect; another example of delegation
- When the Buttons associated with the MoveHandler are clicked, it calls move methods on proxy, which in turn calls move methods on MovableRect (hooray for delegation!).
- When MovableRect is clicked, it sets proxy’s reference to itself
  - The EventHandlers don’t need to change their references
Design of the MovableRectProxy

- The MovableRectProxy class needs a method to set its MovableRect
- Needs moveUp(), moveDown(), moveLeft(), and moveRight() methods to call on its MovableRect
  - e.g., moveUp() will call MovableRect’s moveUp() method

Syntax: MovableRectProxy

```java
public class MovableRectProxy{
  private MovableRect _movingRectangle;

  public MovableRectProxy(){
    /* Constructor doesn't need to do anything because PaneOrganizer will call setRectangle(...)*/
    _movingRectangle = MovableRectProxy.createProxy();
  }

  public void setRectangle(MovableRect rect) {
    /* Sets current Rectangle */
    _movingRectangle = rect;
  }

  public void moveUp() {
    _movingRectangle.moveUp();
  }

  public void moveDown() {
    _movingRectangle.moveDown();
  }

  public void moveLeft() {
    _movingRectangle.moveLeft();
  }

  public void moveRight() {
    _movingRectangle.moveRight();
  }
}
```

Modification: MovableRect

```java
public class MovableRect {
  private Rectangle _rectangle;
  private RectangleProxy _proxy

  public MovableRect(RectangleProxy proxy) {
    _proxy = proxy;
    _rectangle = new Rectangle(Constants.WIDTH, Constants.HEIGHT);
    _rectangle.setOnMousePressed(new MouseHandler());
  }

  //move methods elided

  private class MouseHandler implements EventHandler<MouseEvent> {
    public void handle(MouseEvent event) {
      _proxy.setRectangle(MovableRectProxy.this);
    }
  }
}
```

Design of MoveHandler

- Must write MoveHandler(s) that allow buttons to move the current MovingRectangle
- Could write 4 different MoveHandler(s) (one for each direction)
  - tedious and violates DRY (don't repeat yourself)
- In Graphics II, we used an isLeft boolean to create two different “versions” of the same Handler
- Here, we need four different “versions” – one for each direction
  - Introducing Enums!
enums (1/2)
- Enumerated types
  - Ideal for abstracting groups of constants, like directions
  - Defined within a set of constants
    ```java
    public enum Direction {
        UP, DOWN, LEFT, RIGHT;
    }
    ```
- Get specific values like using constants
  - Ex. `Direction.Down`

Enums (2/2)
- Can be used in switch statements, like ints
  ```java
  public void handle(double x, double y) {
      switch(_direction) {
          case UP:
              // code to move up
              break;
          case DOWN:
              // code to move down
              break;
          case RIGHT:
              // code to move right
              break;
          default:
              // code to move left
          }
  }
  ```

Syntax: The **MoveHandler**
```java
private class MoveHandler implements EventHandler<ActionEvent> {
    Direction _direction;
    public MoveHandler(Direction direction) {
        _direction = direction;
    }
    public void handle(ActionEvent event) {
        switch(_direction) {
            case UP:
                _proxy.moveUp();
                break;
            case DOWN:
                _proxy.moveDown();
                break;
            case RIGHT:
                _proxy.moveRight();
                break;
            default:
                _proxy.moveLeft();
                break;
        }
    }
}
```

**Syntax: PaneOrganizer (1/2)**
```java
public class PaneOrganizer {
    private BorderPane _root;
    private MovableRectProxy _proxy;
    private enum Direction{
        UP, DOWN, RIGHT, LEFT
    }
    public PaneOrganizer() {
        _root = new BorderPane();
        _proxy = new MovableRectProxy();
        this.setUpRectangles();
        this.setUpButtons();
    }
    public Pane getRoot() {
        return _root;
    }
}
```
- We use a **BorderPane** as the root **Pane**
- Same **getRoot()** as normal
private void setUpButtons() {
    GridPane buttonBox = new GridPane();
    buttonBox.setAlignment(Pos.CENTER);
    Button upButton = new Button("Up");
    GridPane.setAlignment(upButton, HPos.CENTER);
    buttonBox.add(upButton, 1, 0);
    //rest of code elided
}

• Creates four buttons and adds each to the buttonBox

• The parameters in the buttonBox's add() method are the object being added, the column number, and the row number of its location in the grid

Syntax: setUpRectangles Method

private void setUpRectangles() {
    MovableRect rect1 = new MovableRect(_proxy);
    rect1.setInitConst(Constants.RECT1X);
    rect1.setFill(Color.BISQUE);
    rect1.setFillColor(Color.CADETBLUE);
    PanPane rectPane = new Pane();
    rectPane.getChildren().addAll(rect1.getNode(), rect2.getNode());
    _root.getChildren().add(rectPane);
    _proxy.setRectangle(rect1);
}

• Again, this helper method of PaneOrganizer is private because it will only be called from PaneOrganizer's constructor

• To make the layout cleaner, the MovableRects are added to the Pane rectPane

rect1 and rect2 are instantiated and rect1 is initially passed to _proxy
Syntax: QuitHandler

```java
private class QuitHandler implements EventHandler<ActionEvent> {
    public void handle (ActionEvent event) {
        System.exit(0);
    }
}
```

- QuitHandler is handled in the same way as it was in previous example!

### Syntax: setUpButtons and QuitHandler

```java
private void setUpButtons() {
    GridPane buttonBox = new GridPane();
    buttonBox.setAlignment(Pos.CENTER);
    buttonBox.add(upButton, 1, 0);
    buttonBox.add(downButton, 1, 2);
    buttonBox.add(rightButton, 2, 1);
    buttonBox.add(leftButton, 0, 1);
    buttonBox.add(quitButton, 1, 3);
}
```

- Makes a new quitButton, formats it and adds QuitHandler like in previous example, and then adds buttonBox to _root

### Why Proxy pattern?

- Suppose we wanted to write a ColorChanger class that randomly changes the MovableRect's color every time it is moved.
- handle in MoveHandler would call setRandomColor every time it is executed.
- Benefit of Proxy: Don't have to update ColorChanger every time current rectangle changes.

```java
public class ColorChanger {
    private MovableRectProxy _proxy;
    public ColorChanger(MovableRectProxy proxy) {
        _proxy = proxy;
    }
    public void setRandomColor() {
        int red = (int) (Math.random() * 205);
        int green = (int) (Math.random() * 255);
        int blue = (int) (Math.random() * 255);
        _proxy.setColor(Color.rgb(red, green, blue));
    }
}
```

// In MoveHandler
```java
public void handle(ActionEvent event) {
    switch(_direction) {
        case UP: 
            _proxy.moveUp();
            break;
        case DOWN: 
            proxy.moveDown();
            break;
        case RIGHT: 
            _proxy.moveRight();
            break;
        default: 
            _proxy.moveLeft();
    }
    _colorChanger.setRandomColor();
}
```

### Design: RectangleProxy

- Analyze program and come up with a class diagram.
- All instances of MovableRect know about MovableRectProxy, but MovableRectProxy only knows about one MovableRect at a time.
Holder vs. Proxy (1/2)

- Notice the similarity between Proxy pattern and simpler Holder pattern
  - difference is in modeling
- Holder will usually contain subject that clients can access (get) and send messages to directly
  - Clients access/set the subject directly via simple accessor/mutator methods
  - This lets clients call any public method defined by subject on instance of that subject

Holder vs. Proxy (2/2)

- Proxy knows about subject that clients can call methods indirectly by way of proxy
  - Lets proxy limit the methods that can be indirectly called on the subject it models

Generic Structure of Proxy

- Proxy represents our RectangleProxy
  - provides indirect access to the Subject for the clients
  - acts on behalf of Rectangle—an example of delegation
- Subject represents the Proxy’s reference to the current Rectangle. When a Rectangle is clicked on, it sets the proxy’s reference to itself.

Clients represent Buttons

- indirectly moves the current Rectangle through the RectangleProxy
- buttons don’t need to change their references when the current Rectangle is changed.

Design Patterns...

- Serve as examples of good design
  - there are no “hard and fast” rules
  - there are concrete trade-offs to think about
  - they are tools to help you build your own designs
- Provide common vocabulary to discuss design at a more abstract level
  - give us a concise way to describe complex object interaction
  - discuss design at a higher level because we do not need to describe every object in the program
- Must be adapted to your program specification
  - may need to add extra relationships to your structure to augment a design pattern.
  - may need to create a new pattern because none exists that exactly fits your needs.
- Should be used in moderation
  - consider trade-offs carefully before using a pattern.
  - consider added complexity—-is it needed in your model?
Announcements

- Cartoon due tomorrow!
  - Early handin tonight, late handin Sunday
- DoodleJump and DoodleJumpDQs out today
- Live DoodleJump Design Check signups out today
  - Fill out signup form by Saturday
  - Design Check slots from Sunday-Wednesday