Lecture 12

Component Design
Recall: Problem with Subclassing

- Games have *lots* of classes
  - Each game entity is different
  - Needs its own functionality (e.g. object methods)
- Want to avoid **redundancies**
  - Makes code hard to change
  - Common source of bugs
- Might be tempted to **subclass**
  - Common behavior in parents
  - Specific behavior in children

Recall: Problem with Subclassing

Human

- Human Warrior
- Human Archer

Orc

- Orc Warrior
- Orc Archer

NPC

Redundant Behavior
Recall: Problem with Subclassing

- Games have *lots* of classes
  - Each game entity is different
  - Needs its own functionality (e.g. object methods)

- Want to avoid *redundancies*
  - Makes code hard to change
  - Common source of bugs

- Might be tempted to *subclass*
  - Common behavior in parents
  - Specific behavior in children

![Diagram showing subclassing issues]

**Redundant Behavior**

**No Help**
Model

- Store/retrieve **object data**
  - Limit access (getter/setter)
  - Preserve any invariants
  - Only affects this object

- Implements **object logic**
  - Complex actions on model
  - May affect multiple models
  - **Example**: attack, collide

Redundant Behavior
## Model-Controller Separation (Alternate)

<table>
<thead>
<tr>
<th>Model</th>
<th>Controller</th>
</tr>
</thead>
</table>
| • Store/retrieve **object data**  
  • Limit access (getter/setter)  
  • Preserve any invariants  
  • Only affects this object | • Process **game actions**  
  • Determine from input or AI  
  • Find *all* objects effected  
  • Apply action to objects |
| | • Process **interactions**  
  • Look at current game state  
  • Look for “triggering” event  
  • Apply interaction outcome |

In this case, models are lightweight
### Classes/Types are Nouns

- Methods have verb names
- Method calls are sentences
  - `subject.verb(object)`
  - `subject.verb()`
- Classes related by *is-a*
  - Indicates class a subclass of
  - **Example**: String is-a Object
- Objects are class *instances*

### Actions are Verbs

- Capability of a game object
- Often just a simple function
  - `damage(object)`
  - `collide(object1, object1)`
- Relates to objects via *can-it*
  - **Example**: Orc can-it flee
- Not necessarily tied to class
  - **Example**: swapping items
Possible Solution: Decorator Pattern

Possible Solution: Decorator Pattern

Possible Solution: Decorator Pattern

Possible Solution: Decorator Pattern

Possible Solution: Decorator Pattern

Possible Solution: Decorator Pattern

Possible Solution: Decorator Pattern

Possible Solution: Decorator Pattern

Possible Solution: Decorator Pattern

Possible Solution: Decorator Pattern

Possible Solution: Decorator Pattern

Possible Solution: Decorator Pattern

Possible Solution: Decorator Pattern

Possible Solution: Decorator Pattern

Possible Solution: Decorator Pattern

Possible Solution: Decorator Pattern

Possible Solution: Decorator Pattern

Possible Solution: Decorator Pattern

Possible Solution: Decorator Pattern

Possible Solution: Decorator Pattern

Possible Solution: Decorator Pattern

Possible Solution: Decorator Pattern

Possible Solution: Decorator Pattern

Possible Solution: Decorator Pattern

Possible Solution: Decorator Pattern

Possible Solution: Decorator Pattern

Possible Solution: Decorator Pattern

Possible Solution: Decorator Pattern

Possible Solution: Decorator Pattern

Possible Solution: Decorator Pattern

Possible Solution: Decorator Pattern

Possible Solution: Decorator Pattern
Java I/O Example

```java
InputStream input = System.in;

Reader reader = new InputStreamReader(input);

BufferedReader buffer = new BufferedReader(reader);
```

- **Built-in console input**
- **Make characters easy to read**
- **Read whole line at a time**

Most of `java.io` works this way
Alternate Solution: Delegation Pattern

Inversion of the Decorator Pattern
Alternate Solution: Delegation Pattern

Inversion of the Decorator Pattern
**Example: Sort Algorithms**

```java
public class SortableArray extends ArrayList {
    private Sorter sorter = new MergeSorter();
    public void setSorter(Sorter s) { sorter = s; }
    public void sort() {
        Object[] list = toArray();
        sorter.sort(list);
        clear();
        for (o: list) { add(o); }
    }
}
```

```java
public interface Sorter {
    public void sort(Object[] list);
}
```
### Comparison of Approaches

<table>
<thead>
<tr>
<th>Decoration</th>
<th>Delegation</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Pattern applies to <em>decorator</em></td>
<td>• Applies to <em>original object</em></td>
</tr>
<tr>
<td>• Given the original object</td>
<td>• You designed object class</td>
</tr>
<tr>
<td>• Requests through decorator</td>
<td>• All requests through object</td>
</tr>
<tr>
<td>• <strong>Monolithic</strong> solution</td>
<td>• <strong>Modular</strong> solution</td>
</tr>
<tr>
<td>• Decorator has all methods</td>
<td>• Each method can have own delegate implementation</td>
</tr>
<tr>
<td>• “Layer” for more methods</td>
<td>• Like higher-order functions</td>
</tr>
<tr>
<td>(e.g. Java I/O classes)</td>
<td></td>
</tr>
<tr>
<td>• Works on <em>any</em> object/class</td>
<td>• Limited to classes you make</td>
</tr>
</tbody>
</table>
The Subclass Problem Revisited

Delegates?

NPC

Human

Orc

Human Warrior

Human Archer

Orc Warrior

Orc Archer

Redundant Behavior
Component-Based Programming

- **Role**: Set of capabilities
  - Class with very little data
  - A collection of methods
- Add it to object as delegate
  - Object gains those methods
  - Acts as a “function pointer”
- **Can-it**: search object roles
  - Check class of each role
  - Better than duck typing
  - Possible at compile time?

Field storing a single delegate or a **set of delegates**
# Model-Controller Separation Revisited

## Model

- Store/retrieve **object data**
- Preserve any invariants
- Data may include delegates
- Determines **is-a** properties

## Controller

- Process **interactions**
- Look at current game state
- Look for “triggering” event
- Apply interaction outcome

## Components

- Process **game actions**
  - Attached to an entity (model)
  - Uses the model as context
  - Determines **can-it** properties
What about the View?

- Way too much to draw
  - Backgrounds
  - UI elements
  - Individual NPCs
  - Other moveable objects
- Cannot cram all in Draw
- Put it in game object?
  - But objects are models
  - Violates MVC again
Solution: A Drawing Canvas

- Treat display as a **container**
  - Often called a canvas
  - Cleared at start of frame
  - Objects added to container
  - Draw contents at frame end

- Canvas abstracts **rendering**
  - Hides animation details
  - Like working with widget

- Implement `draw(c)` in model
  - Classic heavyweight model
  - No problems with extension

```java
void draw(Canvas c) {
    // Specify perspective
    // Add to canvas
}
```
Solution: A Drawing Canvas

- Treat display as a **container**
  - Often called a canvas
  - Cleared at start of frame
  - Objects added to container
  - Draw contents at frame end

- Canvas abstracts **rendering**
  - Hides animation details
  - Like working with widget

- Implement `draw(c)` in model
  - Classic heavyweight model
  - No problems with extension

```java
void draw(Canvas c) {
    // Specify perspective
    // Add to canvas
}
```
Drawing and Components

- Canvas approach is not perfect
  - Needs “one size fits all” code
  - What if models are different?
- Remember first two labs
  - Needed multiple passes
  - Different drawing each pass
- How can we avoid this?
  - Pull drawing out of the canvas
  - But do not put it in the model
- Component programming!

Unity 3D does this
Problem: Canvas Methods

Model 1
\text{draw(canvas)}

Model 2
\text{draw(canvas)}

Canvas
\text{drawShape(...)}

Ideal
Problem: Canvas Methods

Model 1
\texttt{draw(canvas)}

In Practice

Canvas
\texttt{drawShape1(...)}
\texttt{drawShape2(…)}

Model 2
\texttt{draw(canvas)}
Views and Components

Model1

Component1

drawShape1(...)

Canvas

drawPrimitive(...)

Model2

Component1

drawShape1(...)
Views and Components

Model1

Complex, Shape-specific rendering

Component1
drawShape1(…)

Model2

Component1
drawShape1(…)

Canvas
drawPrimitive(…)

Primitive effects
Summary

- Games naturally fit a **specialized MVC** pattern
  - Want *lightweight* models (mainly for serialization)
  - Want *heavyweight* controllers for the game loop
  - View is specialized rendering with few widgets

- Proper design leads to unusual OO patterns
  - Subclass hierarchies are unmanageable
  - Want *component-based design* to model actions
  - Will revisit this again when we talk about AI