Lecture 11

Architecture Patterns
Architecture: The Big Picture

Game Engine
- Physics Engine
- AI Engine (e.g. Pathfinding)

Game Content
- Character Scripts
- Character Data
- UI Elements
- Models and Textures
- Sounds

Player
- GUI
- Rendering Engine
- Audio Engine

Compiler

Data Management Layer
Architecture: The Big Picture

How do we get started?
Utilizing Software Patterns

- **Pattern**: reusable solution to a problem
  - Typically a template, not a code library
  - Tells you how to design your code
  - Made by someone who ran into problem first

- In many cases, pattern gives you the **interface**
  - List of headers for non-hidden methods
  - Specification for non-hidden methods
  - Only thing missing is the implementation

2110 all over again
Example: Singletons

- **Goal**: Want to limit class to a single instance
  - Do not want to allow users to construct new objects
  - But do want them to access the single object

- **Application**: Writing to the console/terminal
  - Want a unique output stream to write to console
  - Many output streams would conflict w/ each other
  - Given by a unique object in Java (`System.out`)
  - A class with static methods in C# (not a singleton)
Creating a Singleton in Java

```java
public class Singleton {

    public static final Singleton instance = new Singleton();

    private Singleton() {
        // Initialize all fields for instance
    }

    public static Singleton getInstance() {
        return instance;
    }

}
```
Creating a Singleton in Java

```java
public class Singleton {

    public static final Singleton instance = new Singleton();

    private Singleton() {
        // Initialize all fields for instance
    }

    public static Singleton getInstance() {
        return instance;
    }
}
```
Essentially same idea as software pattern
- Template showing how to organize code
- But does not contain any code itself

Only difference is scope
- Software pattern: simple functionality
- Architecture pattern: complete program

Classic pattern: Model-View-Controller (MVC)
- Most popular pattern in single client applications
Model-View-Controller Pattern

**Model**
- Defines/manages the program data
- Responds to the controller requests

**Controller**
- Updates model in response to events
- Updates view with model changes

**View**
- Displays model to the user/player
- Provides interface for the controller

Calls the methods of
Example: Temperature Converter

**Model:** (TemperatureModel.java)
- Stores one value: fahrenheit
- But the methods present two values

**View:** (TemperatureView.java)
- Constructor creates GUI components
- Receives user input but does not “do anything”

**Controller:** (TemperatureConverter.java)
- **Main class:** instantiates all of the objects
- “Communicates” between model and view
TemperatureConverter Example

**View**

**Controller**

**Model**

```
TemperatureConverter

View

Controller

Model

TemperatureModel

@105dc

fahrenheit 32.0 double

gxFahrenheit() setFahrenheit(double)
gxCentigrade() setCentigrade(double)
```

Architecture Patterns
The Game Loop and MVC

- **Model**: The game state
  - Value of game resources
  - Location of game objects

- **View**: The draw phase
  - Focus of upcoming lectures

- **Controller**: The update phase
  - Alters the game state
  - Topic of previous lecture
# Model-Controller Separation (Standard)

<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  
  • Implements **object logic**  
  • Complex actions on model  
  • May affect multiple models  
  • **Example**: attack, collide | • Process **user input**  
  • Determine action for input  
  • **Example**: mouse, gamepad  
  • Call action in the model |

**Traditional controllers are “lightweight”**
Classic Software Problem: Extensibility

- **Given**: Class with some base functionality
  - Might be provided in the language API
  - Might be provided in 3rd party software

- **Goal**: Object with *additional* functionality
  - Classic solution is to subclass original class first
  - **Example**: Extending GUI widgets (e.g. Swing)

- But subclassing does not always work…
  - How do you extend a *Singleton* object?
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
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
Model-Controller Separation (Standard)

**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

---

**Architecture Patterns**

**Redundant Behavior**
Model-Controller Separation (Alternate)

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

**Controller**
- Process **game actions**
  - Determine from input or AI
  - Find *all* objects affected
  - Apply action to objects
- Process **interactions**
  - Look at current game state
  - Look for “triggering” event
  - Apply interaction outcome

In this case, models are lightweight
Does Not Completely Solve Problem

Can I flee?

• Code **correctness** a concern
  • Methods have specifications
  • Must use according to spec

• Check correctness via **typing**
  • Find methods in object class
  • **Example**: orc.flee()
  • Check type of parameters
  • **Example**: force_to_flee(orc)

• **Logical** association with type
  • Even if not part of class
Issues with the OO Paradigm

- Object-oriented programming is very **noun-centric**
  - All code must be organized into classes
  - Polymorphism determines capability via type

- OO became popular with **traditional MVC pattern**
  - Widget libraries are nouns implementing view
  - Data structures (e.g. CS 2110) are all nouns
  - Controllers are not necessarily nouns, but lightweight

- Games, interactive media break this paradigm
  - View is animation (process) oriented, not widget oriented
  - Actions/capabilities only loosely connected to entities
Programming and Parts of Speech

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

- **Request**: Original Object
- **Decorator Object**: New Functionality
- **Original Object**: Original Functionality
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

Original Object

Reference to delegate

Delegate Object 2

Request

Forward Request

Inversion of the Decorator Pattern
Example: Sort Algorithms

```java
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); }
    }
}

public interface Sorter {
    public void sort(Object[] list);
}
```

Architecture Patterns
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</td>
</tr>
<tr>
<td>• “Layer” for more methods</td>
<td>delegate implementation</td>
</tr>
<tr>
<td>(e.g. Java I/O classes)</td>
<td>• Like higher-order functions</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

Redundant Behavior

Delegates?

NPC

Human

Orc

Human Warrior

Human Archer

Orc Warrior

Orc Archer

Slot

Slot

Slot

Human

Warrior

Archer
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

<table>
<thead>
<tr>
<th>Model</th>
<th>Controller</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Store/retrieve <strong>object data</strong></td>
<td>• Process <strong>interactions</strong></td>
</tr>
<tr>
<td>• Preserve any invariants</td>
<td>• Look at current game state</td>
</tr>
<tr>
<td>• Data may include delegates</td>
<td>• Look for “triggering” event</td>
</tr>
<tr>
<td>• Determines <strong>is-a</strong> properties</td>
<td>• Apply interaction outcome</td>
</tr>
</tbody>
</table>

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
}
```
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
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