Lecture 7

Game Architecture Revisited
Recall: The Game Loop

60 times/s = 16.7 ms
The Game Loop

- Almost everything is in loop
  - Except asynchronous actions
  - Is enough for simple games

- How do we organize this loop?
  - Do not want spaghetti code
  - Distribute over programmers
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

Controller calls the methods of...
The Game Loop and MVC

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

- **View**: The draw phase
  - Rendering commands only
  - Major computation in update

- **Controller**: The update phase
  - Alters the game state
  - Vast majority of your code
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

**Controller**
- 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

![Diagram of NPC hierarchy with redundant behavior](image.png)
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

No Help

Redundant Behavior
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>
<tbody>
<tr>
<td>• Store/retrieve <strong>object data</strong></td>
<td>• Process <strong>game actions</strong></td>
</tr>
<tr>
<td>• Limit access (getter/setter)</td>
<td>• Determine from input or AI</td>
</tr>
<tr>
<td>• Preserve any invariants</td>
<td>• Find <em>all</em> objects effected</td>
</tr>
<tr>
<td>• Only affects this object</td>
<td>• Apply action to objects</td>
</tr>
</tbody>
</table>

**In this case, models are lightweight**

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

<table>
<thead>
<tr>
<th>Classes/Types are Nouns</th>
<th>Actions are Verbs</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Methods have verb names</td>
<td>• Capability of a game object</td>
</tr>
<tr>
<td>• Method calls are sentences</td>
<td>• Often just a simple function</td>
</tr>
<tr>
<td>• subject.verb(object)</td>
<td>• damage(object)</td>
</tr>
<tr>
<td>• subject.verb()</td>
<td>• collide(object1,object1)</td>
</tr>
<tr>
<td>• Classes related by <em>is-a</em></td>
<td>• Relates to objects via <em>can-it</em></td>
</tr>
<tr>
<td>• Indicates class a subclass of</td>
<td>• <strong>Example</strong>: Orc <em>can-it</em> attack</td>
</tr>
<tr>
<td>• <strong>Example</strong>: String <em>is-a</em> Object</td>
<td>• Not necessarily tied to class</td>
</tr>
<tr>
<td>• Objects are class <em>instances</em></td>
<td>• <strong>Example</strong>: swapping items</td>
</tr>
</tbody>
</table>
Duck Typing: Reaction to This Issue

- “Type” determined by its
  - Names of its methods
  - Names of its properties
  - If it “quacks like a duck”
- Python has this capability
  - `hasattr(<object>,<string>)`
  - True if object has attribute or method of that name
- This has many **problems**
  - Correctness is a **nightmare**

**Java:**
```java
public boolean equals(Object h) {
    if (!(h instanceof Person)) {
        return false;
    }
    Person ob = (Person)h;
    return name.equals(ob.name);
}
```

**Python:**
```python
def __eq__(self,ob):
    if (not (hasattr(ob,'name')))
        return False
    return (self.name == ob.name)
```
Duck Typing: Reaction to This Issue

- “Type” determined by its
  - Names of its methods
  - Names of its properties
- If it “quacks like a duck”
  - Python has this capability
    - `hasattr(<object>, <string>)`
      - True if object has attribute or method of that name
- This has many problems
  - Correctness is a nightmare

Java:

```java
public boolean equals(Object h) {
  if (!(h instanceof Person)) {
    return false;
  }
  Person ob = (Person)h;
  return name.equals(ob.name);
}
```

Python:

```python
def __eq__(self, ob):
  if (not (hasattr(ob, 'name'))):
    return False
  return (self.name == ob.name)
```

- What do we really want?
  - Capabilities over properties
  - Extend capabilities without necessarily changing type
  - Without using new languages
  - Again, use *software patterns*
Possible Solution: Decorator Pattern

Possible Solution: Decorator Pattern

Request → Decorator Object → Original Object

New Functionality

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

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
public class SortableArray extends ArrayList {

    private Sorter sorter = new MergeSorter();
    new QuickSorter();

    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);
}
Comparison of Approaches

Decoration

• Pattern applies to *decorator*
  • Given the original object
  • Requests through decorator

• Monolithic solution
  • Decorator has all methods
  • “Layer” for more methods (e.g. Java I/O classes)

• Works on *any* object/class

Delegation

• Applies to *original object*
  • You designed object class
  • All requests through object

• Modular solution
  • Each method can have own delegate implementation
  • Like higher-order functions

• Limited to classes you make
The Subclass Problem Revisited

NPC

Human

Human Warrior

Human Archer

Orc

Orc Warrior

Orc Archer

Redundant Behavior

Delegates?

NPC

Human

Warrior

Archer

Orc

Slot

Slot

Slot
The Subclass Problem Revisited

Delegates?

NPC

Human

Orc

Warrior

Archer

Human

Warrior

Archer

Orc

Warrior

Archer

Redundant Behavior

Topic for Next Time