# CS 383 Team Lead 3
# Quality Assurance and Patterns

- Patterns

Tosin Bangudu, Kyle Hash, Liam Mathews

# Isomorphic Design Demo

# What Are Design Patterns

- Design patterns are general solutions to problems in programming.
- Great implementation flexibility.
- Encapsulation of objects.
- Can prevent future issues.
- Can improve the readability of your code.

Three Primary Kinds:

1) Creational
2) Structural
3) Behavioral

# 2 Patterns Needed for Oral Exam

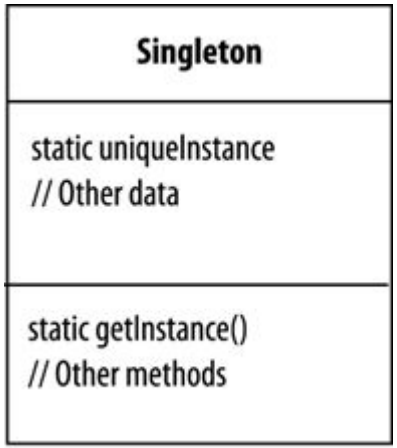
# Creational

Focused primarily on the instantiation of a class and/or objects, according to your needs.

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

# Singleton

- Make sure a class has only one instance, and make it globally accessible.
- Useful for when a feature is used by an application in only one way.
- Makes connecting scripts in your project easier.
- Allows for “lazy instantiation.”



# EX: Singleton Design

- Audio Manager
- Score Manager
- Health Manager

# Singleton

- Declare a private static instance of the class within itself.
- Declare a public static instance of the initializer.
- Static allows for the class instance to be called from other scripts without the need for a reference.
- All variables and references within the Singleton class can now be accessed anywhere.

```
public class Singleton :
MonoBehaviour
{
        public static Singleton instance;
}
```

# Singleton

- Global access makes it easy to cause unwanted changes, be sure to protect and privatize variables and references.
- Can make implementing changes more difficult as the project continues. Changes will need to be made in all associated scripts.
- Be sure to know what you will need to do in advance!

# Singleton

- Sealed keyword prevents creation of subclasses. (Not the bad part.)
- Make sure that no more than one thread can be created by singleton class, this code can fail.
- Unity is thread safe, so we will not worry about it for now.
- Thread safety can be achieved in a variety of ways.

```
// Bad code! Do not use!
public sealed class Singleton
{
   private static Singleton instance=null;

   private Singleton() { }

   public static Singleton Instance
   {
     get {
       if (instance==null) {
         instance = new Singleton();
       }
       return instance;
     }
   }
}
```

# Singleton

- Thread safe example.
- Lazy instantiation, as it is started by the first reference to the contents of Nested().
- Because the Nested class is marked private, the contents must contain the keyword internal to be visible to the rest of our Singleton class.

```
// Thread safe and fully lazy
public sealed class Singleton
 {
   private Singleton() { }
   public static Singleton Instance {
     get {
       return Nested.instance;
     }
   }

   private class Nested {
     // Explicit static constructor to tell C# compiler
     // not to mark type as beforefieldinit
     static Nested() { }
     internal static readonly Singleton instance = new Singleton();
   }
}
```

# Singleton

- Another way to make a singleton thread safe.
- If you use this method, be prepared to explain why in the oral exam.

```
public class Singleton :
MonoBehaviour
{
        public static Singleton
Instance { get; private set; }
}
```

# Structural

Focused on the composition of classes and objects to form large structures.

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

# Decorator

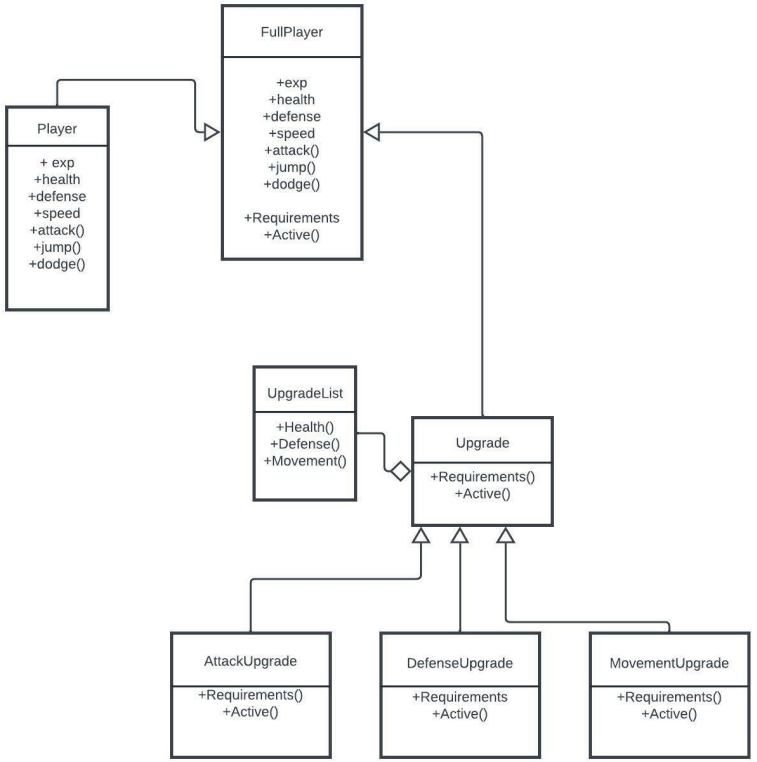
- The Decorative Pattern serves as an alternative to inheritance.
- Allows for additions to an entity to be added dynamically.
- Traits in a “core” exist in all subclasses, allowing for recursive addition to an entity.

# EX: Decorator Design

- Feature: Skill Tree.
- Decorative Pattern: Allows for dynamic changes to be made to an entity as the program runs.

.

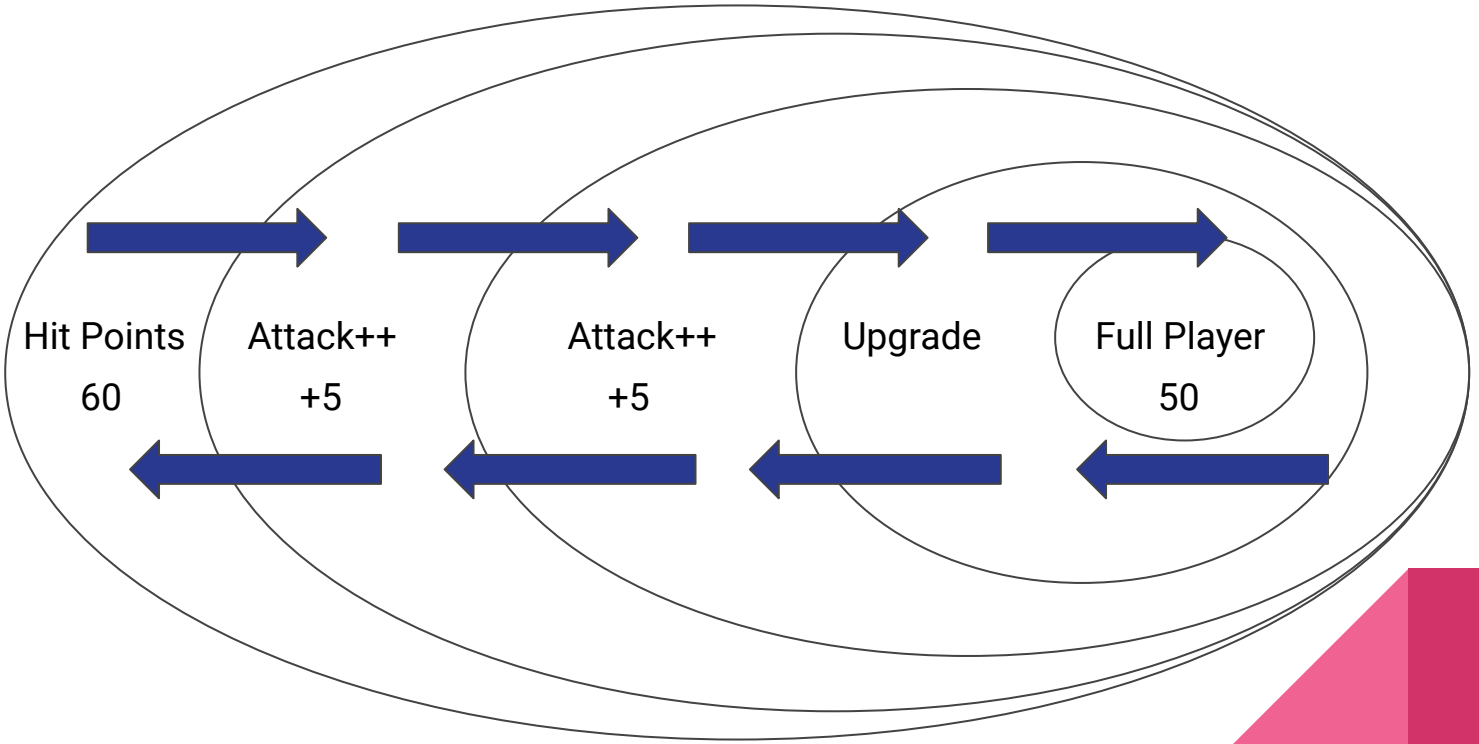
- Let’s say we want to upgrade our player’s attack value.



# Decorator

- For features such as a skill tree, inheritance is not viable, we want to make individual changes during runtime.
- We can use recursion to create “wrapping paper” around a core class, and “wrap” that wrapping paper, and so on.

# Decorator



# Decorator

- Select a class to operate as the core.
- Create subclasses which contain the elements of the core class.
- Core class and preceding subclasses will receive changes made by decorators.

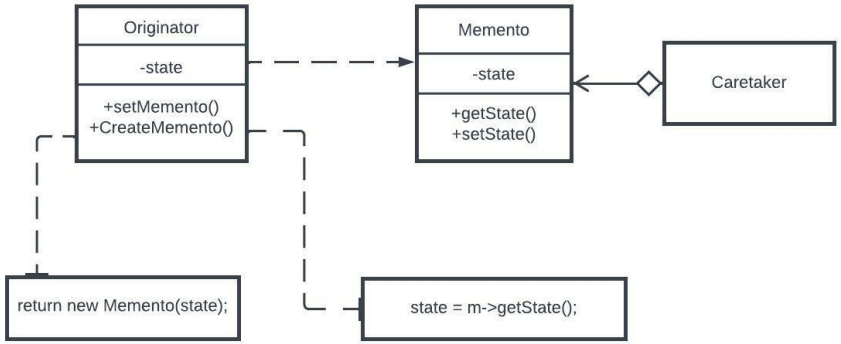
# Behavioral

Focused primarily on the communication between classes.

- Chain of Responsibility
- Command
- Mediator
- State
- Memento
- Strategy
- Visitor

# Memento

- “Capture” the current state of an object so you can return to it later if necessary.
- Does not violate encapsulation.
- A “client” requests a memento from a source object, saves the internal state of the source object to an external file, that the client can later place back into the source object.



# Memento

- Originator: the source class that is able to save itself.
- Caretaker: Decides reasoning and timing for the current state of the Originator to be saved.
- Memento: handles the saving of the data in the Originator.

# EX: Memento Design

- Save state.
- Checkpoints.
- Undo Button.
- Time Travel.

# Memento

- Decide which classes are Originator and Caretaker.
- Create a Memento class to handle transaction of data into files.
- Make sure the Originator class knows how to save itself.
- Ensure that the Caretaker knows when to save the current state, and when to apply it back into the Originator.

# Learn More

Design Patterns - Source Making

https://sourcemaking.com/design_patterns

Singletons in Unity (done right) - Game Dev Beginner

https://gamedevbeginner.com/singletons-in-unity-the-right-way/

# Questions?

# Team Deliverables

- Thursday (10/6)
  - One Test Case to Demo /2
  - Initial Test Plan to Demo /10
- Oral Exam
  - Full Test Plan /40
  - 11/14-11/18
- Patterns
  - Must implement 2 by Oral Exam /60