# **Post-read: Programming Fundamentals**

# **Key points covered in post read**

The key points covered in the session are:

* Design pattern
* need for design pattern
* Singleton Design pattern
* Object pooling design pattern

# **Introduction -**

# Design patterns in gaming refer to recurring solutions or approaches that game developers use to solve common problems in game design. These patterns are often tried and tested solutions that have been used successfully in other games, and they help developers save time and effort in designing games.

# Some common design patterns in gaming include:

# **Singleton Pattern**: This pattern is used to ensure that only one instance of a class is created and that this instance is globally accessible. It is often used to manage resources such as audio, graphics, and input.

# **State Pattern**: This pattern is used to manage the state of an object or a system. It is often used to manage the state of a game, such as whether the player is in a menu, playing the game, or in a cutscene.eg used in unity Animator

# **Observer Pattern**: This pattern is used to notify objects of changes to other objects. It is often used in game events such as scoring, level completion, or enemy defeat.eg Used in achievement system

# **Object Pooling :** Object pool pattern is a software creational design pattern which is used in situations where the cost of initializing a class instance is very high

# **Factory Pattern :** This type of design pattern comes under a creational pattern as this pattern provides one of the best ways to create an object.In the Factory pattern, we create objects without exposing the creation logic to the client and refer to newly created objects using a common interface.

# **Command Pattern :** In object-oriented programming, the command pattern is a behavioral design pattern in which an object is used to encapsulate all information needed to perform an action or trigger an event at a later time. This information includes the method name, the object that owns the method and values for the method parameters. In simple words it lets you create a command by a means of combination of different classes and methods into a single executable entity

# **Flyweight Pattern :** The Flyweight Pattern is a structural design pattern that aims to minimize memory usage by sharing data between multiple objects. The basic idea is to separate intrinsic data (data that is common to all objects of a certain type) from extrinsic data (data that is specific to each object). The intrinsic data is then shared between objects, while the extrinsic data is passed in as a parameter when needed. These are just a few examples of design patterns used in gaming. They help game developers create more efficient, scalable, and maintainable games, which ultimately result in better player experiences.

There are more patterns like strategy and factory that can be learned later as you become a better game Dev.

# **Component Pattern :** Unity Itself uses a Component pattern . Here all code is made to be standalone module that gives a specific behavior to the added object . Here the code is not written in the same class as a monolith but broken down into many classes talking with each other .

### Looking at some Design patterns in Depth -

### Singleton Pattern -

The Singleton Pattern ensures that only one instance of a class is created and that this instance is globally accessible. This is useful for managing resources that should have only one instance throughout the game, such as audio, graphics, and input.

It allows a global state for a class to be accessed from anywhere

Here's an example implementation of the Singleton Pattern in Unity:

public class SingletonExample : MonoBehaviour

{

private static SingletonExample instance;

private void Awake()

{

if (instance == null)

{

instance = this;

DontDestroyOnLoad(gameObject);

}

else

{

Destroy(gameObject);

}

}

}  
  
use it to define a global state for any class so that one can access it from anywhere

**Code Explanation**  :  
 In this example, we have a class called SingletonExample that inherits from the MonoBehaviour class. We declare a private static instance variable of type SingletonExample.

In the Awake method, we check if the instance is null. If it is null, we assign the current object (this) to the instance variable, and we use DontDestroyOnLoad to ensure that the object is not destroyed when a new scene is loaded.

If the instance is not null, it means that another instance of the class already exists, so we destroy the current object to ensure that there is only one instance of the class at any given time.

That's it! With this simple code, we have implemented the Singleton Pattern in Unity, ensuring that there is only one instance of the SingletonExample class throughout the game.

### Object pooling -

Object Pooling -

Object pool pattern is a software creational design pattern which is used in situations where the cost of initializing a class instance is very high  
 Example in a game when there is a need to instantiate a lot of bullets then instantiating / creating bullets adds a lot of time complexity for the whole code and hence slows down the game so one can already instantiate the bullet at the start of the game & not Destroy it but enable and disable it .

Basically, an Object pool is a container which contains some amount of objects. So, when an object is taken from the pool, it is not available in the pool until it is put back.

Objects in the pool have a life cycle:

Creation

Usage

Destroy.

Object pooling Life cycle :

Creation

usage

Disable

Enable   
Usage

Disable

**Learning Materials -**One of the best & fun resource for learning object pooling this resource has an activity so that you can code along object pooling as well - https://www.kodeco.com/847-object-pooling-in-unity

### Observer Pattern -

The Observer Pattern is used to notify objects of changes to other objects. It is often used in game events such as scoring, level completion, or enemy defeat.

For Example : In The Witcher 3, there are various events that can happen during gameplay, such as the player completing a quest, encountering enemies, or interacting with NPCs. These events can affect different parts of the game, and it's important to keep track of them and update the game state accordingly.

To handle these events and update the game state, the Observer Pattern is used. The game uses various classes as observers, which are notified of changes to the game state and update their behavior accordingly.

For example, let's say the player completes a quest to rescue a villager. This event could trigger changes in various parts of the game, such as the player's reputation, the availability of new quests, and the behavior of NPCs in the village.

### Factory Pattern -

The basic idea behind the Factory pattern is to define an interface for creating objects, but let the subclasses decide which class to instantiate. In other words, the Factory pattern allows you to create objects without specifying the exact class of object that will be created.

Here is an example of how the Factory pattern can be used:

Suppose you are developing a game that has different types of characters, such as warriors, mages, and archers. Each character has its own unique set of attributes and abilities. Instead of creating each character object manually, you can use a factory to create the character objects for you.

First, you would define an interface called Character that specifies the methods that all characters should have. Next, you would create subclasses that implement the Character interface, such as Warrior, Mage, and Archer. Finally, you would create a factory class called CharacterFactory that has a method called createCharacter that takes a parameter specifying the type of character to create (e.g., "warrior", "mage", or "archer") and returns a new instance of the corresponding subclass.  
  
To look at a programming instance - <https://www.tutorialspoint.com/design_pattern/factory_pattern.htm>

### Command Pattern -

In a game development context, the Command Pattern can be used to handle user input. For example, if the player presses a button to jump, the button press can be encapsulated in a JumpCommand object. This object can then be passed to the game's input manager, which will store it in a list of commands to be executed.

Later, during the game update loop, the input manager can iterate through the list of commands and execute each one in turn. This allows the game to handle input in a more flexible way, since the input manager can store and execute commands at a later time, rather than processing them immediately.

Another use case for the Command Pattern in game development is to implement an undo/redo system. In this case, each command object would not only encapsulate the request to be executed, but also the information needed to undo it. This allows the game to maintain a stack of executed commands, which can be undone or redone as needed.

Overall, the Command Pattern can help to make game code more modular and flexible, since it allows different objects to communicate without being tightly coupled. This can make it easier to add new features and functionality to the game, without having to change existing code.

Read the article with more Examples - <https://www.habrador.com/tutorials/programming-patterns/1-command-pattern/>

### Flyweight Pattern -

The Flyweight Pattern is a structural design pattern that aims to minimize memory usage by sharing data between multiple objects. The basic idea is to separate intrinsic data (data that is common to all objects of a certain type) from extrinsic data (data that is specific to each object). The intrinsic data is then shared between objects, while the extrinsic data is passed in as a parameter when needed.

In game development, the Flyweight Pattern can be used to reduce the memory footprint of objects that are repeated frequently, such as game sprites or tiles. Instead of creating a new object for each instance of the sprite or tile, a single shared object can be used. The shared object contains the intrinsic data (such as the image or texture data), while the extrinsic data (such as the position and rotation of the sprite) is passed in as a parameter.

For example, imagine a game with a large number of identical trees in the background. Without the Flyweight Pattern, the game would have to create a separate tree object for each instance of the tree, each with its own image data. This would quickly consume a large amount of memory. With the Flyweight Pattern, a single shared tree object can be used, with the position and rotation of each tree passed in as a parameter.  
  
As in a game the time required to render a frame is critical hence this pattern is used to uplift performance of an existing system or to make high performing games as a system .

For a case based Example read - https://refactoring.guru/design-patterns/flyweight

### Component Pattern -

A component system is a way of designing and building things by breaking them down into smaller parts or components. Each component is designed to perform a specific function, and can be assembled with other components to create a larger, more complex system.

For example, think of a computer. It's made up of many different components, such as the motherboard, CPU, memory, hard drive, and so on. Each of these components is designed to perform a specific function, and can be combined with other components to create a functioning computer.

The advantage of a component system is that it allows for greater flexibility and scalability in design. Instead of having to create a single, monolithic system, you can create a variety of different components that can be combined in different ways to create different systems. This makes it easier to design and build complex systems, and also makes it easier to modify or upgrade existing systems by swapping out individual components.

Overall, a component system is a powerful way to design and build complex systems that are flexible, scalable, and easy to modify.  
It provides a great modularity to code and lets you adds decoupled custom behavior to any game object

Extra Readable with a gaming use case - <https://gameprogrammingpatterns.com/component.html>

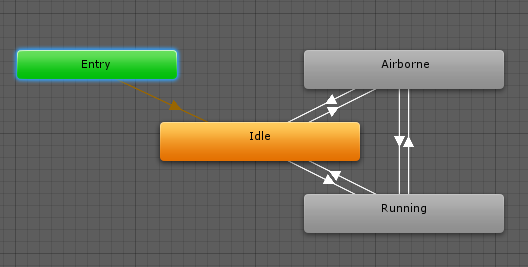
### State Pattern - The state pattern is a programming design pattern that allows an object to change its behavior based on its internal state. It is often used to model complex systems where an object's behavior can change based on external events or conditions.

### The idea behind the state pattern is to create a set of objects, each representing a different state that the system can be in. These objects contain the logic and behavior for that state, as well as the conditions under which the system should transition from one state to another.

### For example, imagine you are building a video game where a character can be in different states, such as "standing", "walking", "running", "jumping", and so on. Each of these states would be represented by a separate object, with its own set of behaviors and transitions to other states.

### Using the state pattern can make your code more modular and easier to maintain, as each state is encapsulated in its own object. It also allows for greater flexibility, as you can easily add or remove states without affecting the rest of the system.

### Overall, the state pattern is a useful tool for building complex systems that need to be able to change their behavior dynamically based on external factors. Here in unity the animator uses the state machine pattern as can be seen in the image below there is an entry state the Idle state which goes on to Airborne and Running state .

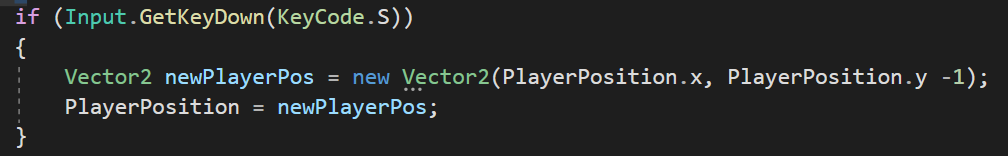


### Post-Assessment of the Live Session

Not A possible value for the Player position when the following code is run :   


1. x=10 y = 20
2. x = 0 , y = 21
3. x = 21.5 y = 2.5
4. x= -1 y = 10

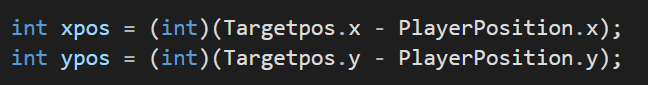
Answer : D

What happens when the following code is run ?   


1. Player moves up to the y axis by 1 unit
2. Player moves down by the y axis by 1 unit
3. Player moves left to the x axis by 1 unit
4. Player moves right towards the x axis by 1 unit

Answer : B

What are we trying to calculate in the given code ?



1. New Position of player
2. new Position of Target
3. Distance of player from the Target
4. converting the x pos and y pos of player and Target to type of integer

Answer : c

Can you convert any folder into a repository & how ?   
a) No One can only create a Repo on Github

b) Yes one can download a repo from Github

c) Yes , it can be done by going into the folder & running a command called git init

d) none of the above

Answer : c