**Design Patterns**

****

**Let’s Understand first**

Suppose you are building a house yourself without any plan. You have a rough idea of what you want or need for the house and you start building the first wall, then the second wall and so on. Sooner you find out that this is not going to work and it can be more problematic if you are building the house with multiple people. How will they know what to do without a plan? The same thing goes for software.

Being a software engineer your purpose should not be only writing a bunch of code but also to reduce complexity and taking care of code reusability as well. You will be expected to write a clean, elegant, bug-free and manageable code so that other developers can understand and later if it requires any kind of change then it can be done with minimal effect on your entire project and here design pattern plays a crucial role here.

**Definition**

In software engineering, a design pattern is a general repeatable solution to a commonly occurring problem in software design.

A design pattern isn't a finished design that can be transformed directly into code.

It is a description or template for how to solve a problem that can be used in many different situations.

**Importance of Design Patterns**

* It makes code reusable, bug-free, and clean.
* Speed up development process.
* Changes or modifications become easier.
* Reduce common problems of developers they face during the development process.
* Improve object oriented skill.
* Easy to understand the flow of code.
* Less code is so easy to maintain.

**Types of Design Patterns**

Design Patterns are divided into 3 categories:

**1)Creational Pattern:** It provides object creation mechanisms that increase flexibility and reusability of existing code.

**2)Structural Pattern:** Explains how to assemble objects and classes into larger structures while keeping the structures flexible and efficient.

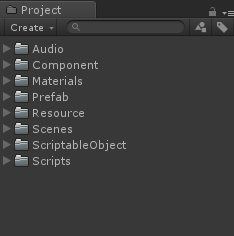
**3)Behavioral Pattern:** These patterns describe how a group of objects cooperate to perform a task.

**Let’s Practice with Unity**

**Required software:** Unity3D version LTS 2017.4.6f1

Visual Studio 2019

**Folder Structure:**

****

**Scenes:** Here all the scenes lie with their respective design pattern names and the main scene is named as “DesignPattern” we have to start from this scene.

**Scripts:** Here all the scripts lie with their respective design pattern names.

**Materials:** Here all the materials lie.

**Prefab:** To save pre-fabricated objects.

**Resource:** Here all the prefabs lie that we want to separate from others.

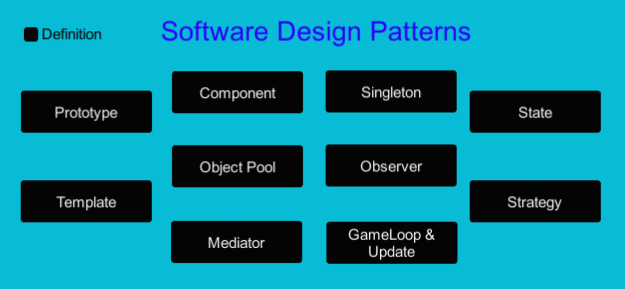
**Audio:** Here all the audio files will be saved(if used).

**Component:** Here we will save a prefab which can be used to perform some common tasks in other scenes.

**ScriptableObjects:** To save [scriptable objects](https://docs.unity3d.com/Manual/class-ScriptableObject.html).

**How to navigate?**

**Step1:** Open Main Scene named as → **DesignPattern**

****

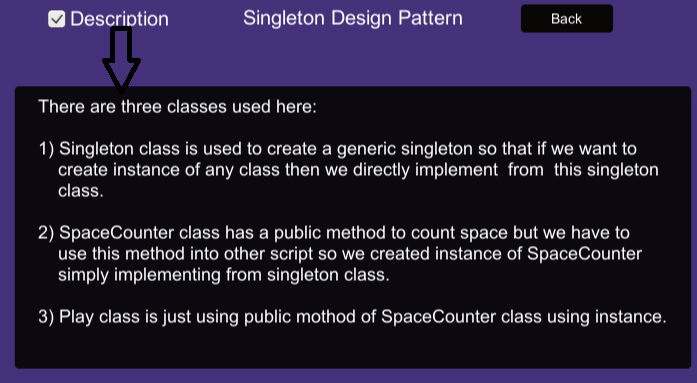
**Step2:** Click on Definition to study design patterns.

**Step3:** Various design pattern listed here are clickable buttons if you click any pattern then following page displays:



**Step3:**  Press the continue button to move to Demo Scene or press Back to move to the Front page.

In the Demo Scene there are two top toggle buttons **Description** and **Back , Description** button displays about code used in this scene whereas **Back** button is used to back to the front page.



You can find all the scripts of a particular design pattern to its respective named folder.



**Let’s Study all the given Design patterns**

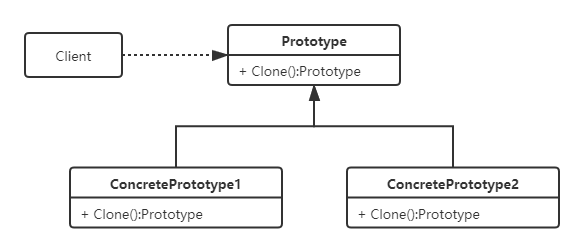
**1.Prototype Design Pattern:**

It is Fully initialized instance to be copied or cloned.

In the prototype pattern a new object is created by cloning an existing object.

Prototype is a creational design pattern.

We have demonstrated this pattern by using Scriptable Objects. By using scriptable objects we can minimize the memory usage and it also promotes reusability of an object.



**Working:**

This demo has two buttons for each shape one for **cube** and other for **cylinder**

When we click on this button then it fetches data from its respective Scriptable object.

#### **GAME OBJECT NAME**

**Shape1Btn** and **Shape2Btn**

#### **SCRIPTS**

**ShapeUnit.cs** >> Fetches the data from a scriptable object for a given shape.

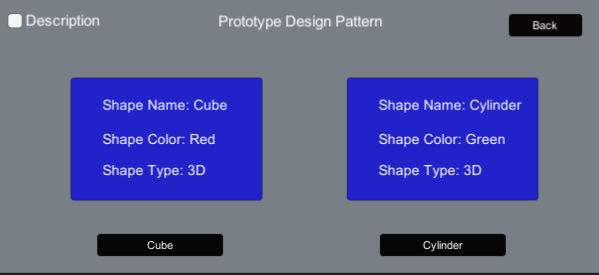
#### **FUNCTIONALITY**

1. Invokes a public method named ‘ShowData’ when we click on the button (**ShapeUnit.cs** >> **ShowData**).

So, if we need a new shape then we can just make another scriptable object by changing the values without any modification in the scripts.

We use this pattern for the following reasons

1. When the process of making an object is expensive or takes a long time.
2. When the client application must be unaware of the creation of an object.
3. When you need to keep the number of classes in your application to a minimum level.



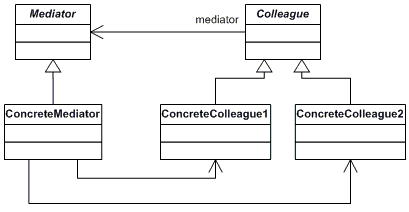
By clicking on Description you can check the function of each script.

**2.Mediator Design Pattern:**

Mediator pattern is used to reduce communication complexity between multiple objects or classes.

This pattern provides a mediator class which normally handles all the communications between different classes and supports easy maintainability of the code by loose coupling.

This pattern comes under the Behavior Design Pattern.



**Working:**

We are Implementing mediator pattern by example of a simple chat room where multiple users can send messages to the chatroom and it is the responsibility of the chatroom to show the messages to all users.

You can check the functions of each script by clicking on the **Description** button.

# 

#### **GAME OBJECT NAME**

**ChatRoom**,**InputManager** and **User**

#### **SCRIPTS**

**InputClass.cs** >> **InputManager** >> Fetches the data from input text fields from any user.

**User.cs** >> **User** >> Holds reference of current user.

**ChatRoomMediater.cs** >> **ChatRoom** >> It shows a message.

#### **FUNCTIONALITY**

1. Invokes a public method named ‘FirstUserMsg' and ‘SecondUserMsg’ when we click on the send button (**InputClass.cs** >> **FirstUserMsg ,InputClass.cs** >> **SecondUserMsg**).
2. Invokes a public method named ‘SendMessage’ for current user with its message.(**User.cs** >> **User**)
3. Invokes a public method named ‘ShowMessage’ for current user with its message to show the message(**ChatRoomMediater.cs** >> **ChatRoom** )

So we can clearly see that objects are not communicating directly with each other, they are communicating via a mediator.

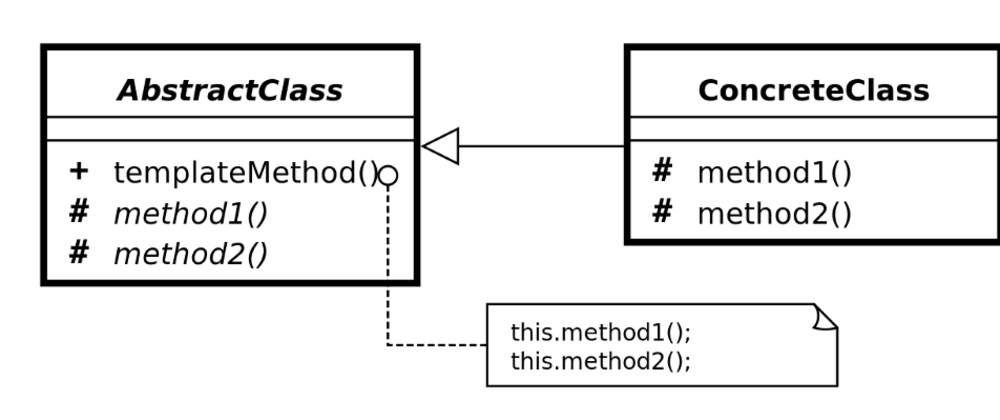
# **3.Template Method Design Pattern:**

As the name indicates Template means a Blueprint in which we can implement changes according to our requirements without changing the algorithms’s structure.

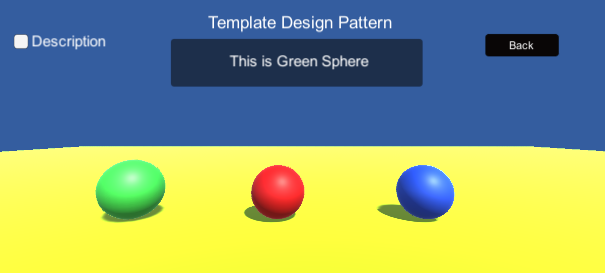
Template method design pattern is to define an algorithm as a skeleton of operations and leave the details to be implemented by the child classes.

The overall structure and sequence of the algorithm are preserved by the parent class.

This pattern comes under the Behavior Design Pattern.



**Working:**

****

This scene includes three spheres: we click any sphere then it shows three functionality: **scaling**,**jumping** and **naming**.

**Scaling** and **jumping** is common to all but **naming functionality**(shows different names in text field) is different so here we are changing one functionality and other two functionality are common.

Hence this describes the best use of templates.

#### **GAME OBJECT NAME**

**Sphere1**,**Sphere2**, and **Sphere3**

#### **SCRIPTS**

**ShapeBaseClass.cs** >> An abstract class with two default, one abstract and one template method.

**FristSphere.cs** >> **Sphere1**>> Implement Abstract method of ShapeBaseClass.

**SecondSphere.cs** >> **Sphere2**>> Implement Abstract method of ShapeBaseClass.

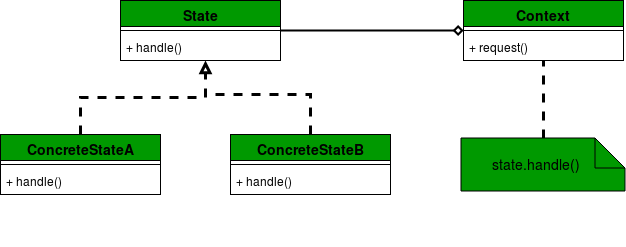
**ThirdSphere.cs** >> **Sphere3**>> IImplement Abstract method of ShapeBaseClass..

#### **FUNCTIONALITY**

1. An Abstract parent class named as "ShapeBaseClass" contains two default methods which remain the same, one abstract method which has to be implemented by child classes and one template method.(**ShapeBaseClass** >> **Jump**,**ShapeColor** , **ShowEffect**, **ShapeDetail**)
2. FirstSphere, SecondSphere, and ThirdSphere classes implement the abstract method of "ShapeBaseClass".(**FristSphere**>>**ShapeColor** , **SecondSphere**>> **ShapeColor**, **ThirdSphere** >>**ShapeColor** )

# **4.State Design Pattern:**

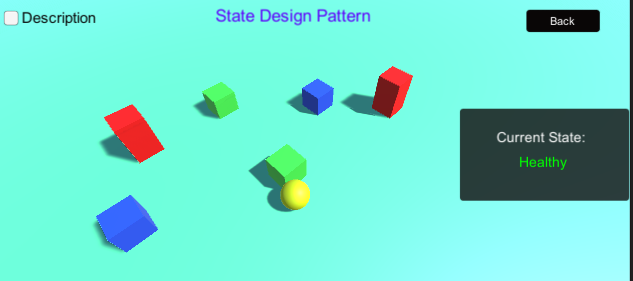
The State pattern is a behavioral software pattern that allows an object to alter its behavior when its internal state changes.



# 

# 

# **Working:**



Here Player is a yellow Sphere when it collides with different color cubes then its state changes and in that particular state we can change some behaviour as we have changed text and its color as **Healthy**(green), **Hurt**(yellow) and **Dead**(red).

We have changed only color and text but we can also use animation here.

#### **GAME OBJECT NAME**

**Player** and **State**

#### **SCRIPTS**

**PlayerController.cs** >>**Player** >> Used to move the player..

**Player.cs** >> **Player** >> Used to decide the state.

**State.cs** >> **State** >> An interface with an abstract method.

**HealthyState.cs** >> Implement Abstract method of State Interface.

**HurtState.cs** >> Implement Abstract method of State Interface.

**DeadState.cs** >> IImplement Abstract method of State Interface.

#### **FUNCTIONALITY**

1) PlayerController used to move the player(Sphere) using arrow keys and

a particular state is decided on collision of player with different color cubes.

(**PlayerController** >> **FixedUpdate, OnCollisionEnter**)

2) PlayerController decide the particular state using "BulletHit" method defined

in Player Class.

(**Player** >> **Bullethit**)

3) There is an Interface named as "State" which contains an abstract method

which is implemented by each state class.(**State** >> **ExecuteCommand**)

4) There are three State classes named as HealthyState, HurtState and

DeadState.(**FristSphere ,SecondSphere,ThirdSphere** >>**ExecuteCommand**)

# **5.Strategy Design Pattern:**

Strategy is a behavioral design pattern that lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

This pattern is mostly used when we are not sure which algorithm is best for the particular problem at compile time.

The Strategy pattern suggests that you take a class that does something specific in a lot of different ways and extract all of these algorithms into separate classes called *strategies*.

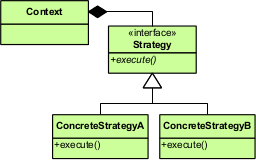
The original class, called *context*, must have a field for storing a reference to one of the strategies. The context delegates the work to a linked strategy object instead of executing it on its own.

The context isn’t responsible for selecting an appropriate algorithm for the job. Instead, the client passes the desired strategy to the context.

**State vs Strategy Pattern:**

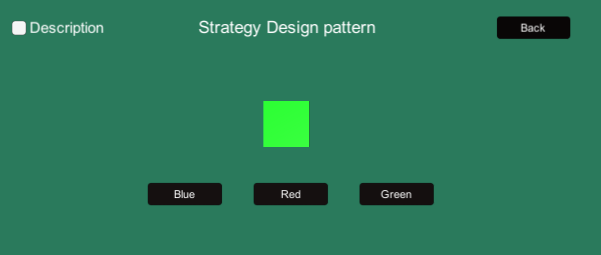
The State pattern deals with what(state or type) an object is in it .It encapsulates state-dependent behaviour.

Strategy pattern deals with how an object performs a certain task. It encapsulates an algorithm.



* **IStrategy**: This is the common interface for all algorithms.
* **ConcreteStrategy**: This class implements **Istrategy** This class implements the actual algorithm.
* **Context**: This class maintains a reference of **ConcreteStrategy** Client will use this class to use the algorithms.

**Working:**

****

Here the client is the user and when the user presses the blue button then the cube turns to green. If he presses blue then the cube turns to blue this means that there are different ways to perform a task and these ways are known as strategies.

#### **GAME OBJECT NAME**

**Cube**

#### **SCRIPTS**

**ColorChangeClient.cs** >>**Player** >> Used to pass desired strategy.

**IColorStratergy.cs** >> Used to describe all the strategies.

#### **FUNCTIONALITY**

1) IColorStrategy:- This script has one Interface with an Abstract method

named as "ChangeColor" which is implemented by each strategy class.

There are three concrete strategy classes RedStrategy, BlueStrategy

and GreenStrategy. This Script has one context class which uses defined

strategies. (**RedStrategy,** **BlueStrategy, GreenStrategy**>> **ChangeColor**)

2) ColorChangeClient:- This is a client class and it decides which strategy

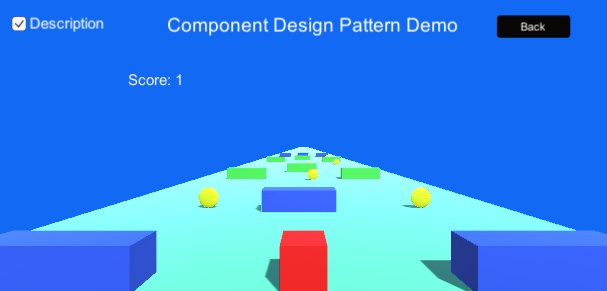
has to be used. It passes the desired strategy into the context class.

( (**ColorChangeClient** >> **RedColor, BlueColor, GreenColor**)

**6.Component Design Pattern:**

The idea behind this pattern is why Unity has different components that you can attach to a gameobject. You can add a sound component, a rigidbody component, and/or a particle system component. What if all of those were a part of the same class? Now you can modify each of the components individually without having to give a single thought to one of the other components. The components are still allowed to talk to each other.

**Working:**

****

This is a mini game but it displays best use of component design pattern in this game player is a red cube when this red cube moves and collides with coins then score increases and if we collect 5 coins then welldone panel enables and if we collides with any larger cube then game will be over.

To perform each task we had used different components so that this makes our project simple to understand.

**GAME OBJECT NAME**

**MovementController**, **CameraFollower**,**GameManager**,**ScoreManager**

#### **SCRIPTS**

**MovementController.cs** >> **MovementController** >> Used to move the player.

**CameraFollower.cs** >> **CameraFollower>>** Used to follow the player.

**ScoreManager.cs** >> **ScoreManager>>** Used to increase score.

**GameManager.cs** >> **GameManager>>** Used to get reference of panels like gameover, welldone.

**SceneManagement.cs** >> **SceneManagement>>** Used to souffle between scenes.

#### **FUNCTIONALITY**

There are five component used here:

1) The MovementController used to move the player(cube) using the arrow key.

(**MovementController.cs** >> **FixedUpdate**)

2) Camera follower is used to follow the player when it moves.

(**CameraFollower.cs** >> **LateUpdate**)

3) GameManger is used to manage games like gameover and welldone.

4) SceneManagement is used to shuffle between main scene and current

scene.(**SceneManagement.cs>> LoadMainScene**)

5) ScoreManager is used here to handle the score.

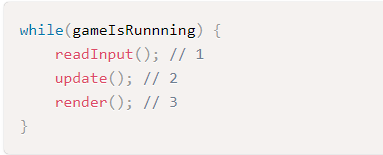
(**ScoreManager.cs** >> **OnCollisionEnter, OnTriggerEnter**)

**7.Game Loop & Update Method Design Pattern:**

Almost all games use a design pattern called **game loop**, which implementation is responsible for three essential tasks:

1. Reading user input;
2. Updating the game internal state;
3. Render the screen.

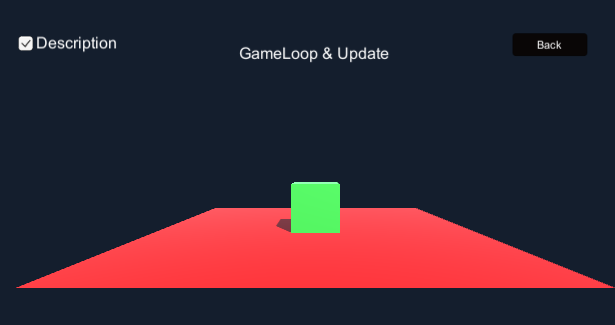
As the name implies, these three tasks run in a loop, and the number of times this loop runs in a second determines the frames per second, or FPS, of the game. Pseudo code for this implementation is shown below:



The Game Loop pattern ensures that game time progresses at equal speed in all different hardware setups.

Game Loop and Update Method are like bread and butter, they often work together.

**Working:**

****

To understand this design pattern we take a cube that will jump when we press the Up arrow key but first we have to press the spacebar. If we don’t press the spacebar then the up arrow will not be able to jump this cube.

**GAME OBJECT NAME**

**Player**

#### **SCRIPTS**

**GameLoop.cs** >> **Player**>> Used to check if the desired conditions met or not.

**MovePlayer.cs** >> **Player>>** Used to move/jump the player.

#### **FUNCTIONALITY**.

1) GameLoop script: Used to check if the desired conditions met or not for jumping of the cube.

(**GameLoop.cs** >> **Awake,Update,FixedUpdate**)

2) MovePlayer script has a method which is called by **GameLoop**.**cs**

(**MovePlayer.cs** >> **Move**)

From here we clearly observe that Unity has its own game loop and Update method built in.

We need not to give our implementation of this design pattern as this design pattern is always used when using any monobehaviour method like Awake,Start, Update, LateUpdate etc.

**8.Objectpool Design Pattern:**

Instantiating and destroying a lot of game objects during the game play can cost significant CPU time, which can be bad for performance. Object Pooling allows us to reuse already instantiated objects. For example, if you want to fire bullets during game play then object pooling will be useful.

As the name suggests, objects pooling provides you with a number of already instantiated objects. When you need an object then fetch it from the object pool. When the object is no longer required then return it back to the object pool without destroying it.

Advantage of object pooling:

1.Better game play performance

2.Re-usability of objects

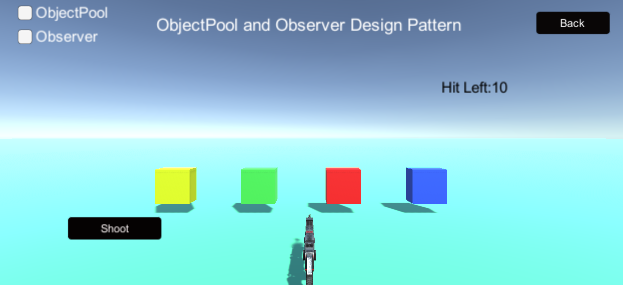
3.Better Object management

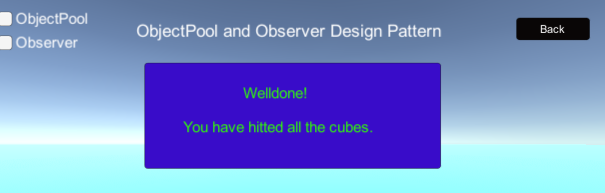
**9. Observer Design Pattern:**

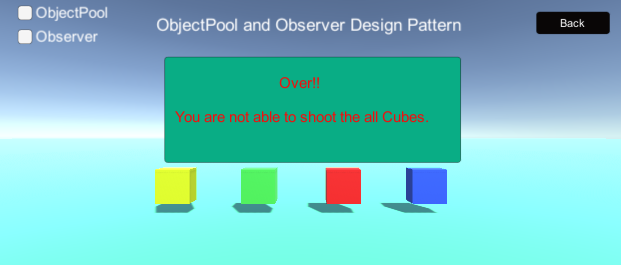
Observer design pattern includes observers (subscriber) and a Subject (Provider, Observable or Publisher). One or many observers can subscribe to the providers for any specific interest. When some specified condition occurs in the Subject object, it will send notification to all of the subscribed observers.This pattern is also helpful in any application to send push notifications to the subscribed users.

**Common Implementation of ObjectPool and Observer Design Pattern**

**Working:**







Here we had made a simple game. In this game a gun is used to fire bullets and these bullets are produced by using object pooling. When a bullet is fired then bullet count decreases by one , if all bullets are fired then gameover panel enables and if we heated all the cubes then welldone panel enables so here we have used observer pattern as well to make these events happen.

**GAME OBJECT NAME**

**Bullet(prefab)**, **Player**, **PoolManager**, **UIManager**

#### **SCRIPTS**

**Bullet.cs** >> **Bullet(prefab)** >> Used to move initialised and move the bullet.

**PoolManager.cs** >> **PoolManager>>** It creates a pool of bullets..

**PlayerScript.cs** >> **Player >>** Used to move player(gun).

**UIManager.cs** >> **UIManager>>** Used to subscribe all methods to their respective events.

#### **FUNCTIONALITY**

1) Bullet script is used to enable or disable bullet and move it to forward direction and

**Welldone** event called in this script are used to show an observer pattern.

(**Bullet.cs** >> **Update**, **OnTriggerEnter**)

2) PoolManager Script is used to create a pool of bullets it has a public function which returns

bullets.(**PoolManager.cs** >> **GetBullet**)

3) PlayerScript is used to get bullets from the PoolManager and we are shooting them with a

Gun and two events **Over** and **UpdateBullet** called.

(**PlayerScript.cs** >> **Update**, **Fire**)

4) UIManager is used to subscribe all events to their respective methods.

(**UIManager.cs** >> **UpdateBulletCount**, **Weldone**, **Wrong**)

**10. Singleton Design pattern:**

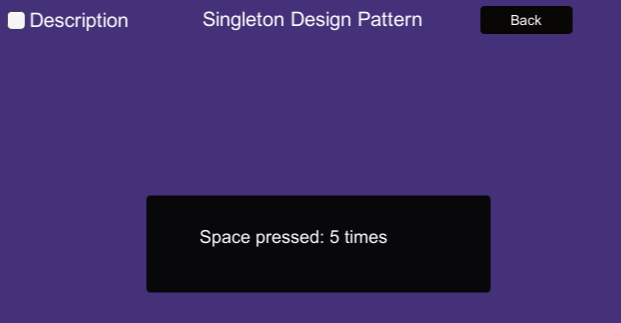
Singleton design pattern comes under the creational design pattern category. It is one of the simplest and widely used design patterns.

Singleton pattern ensures a class has only one instance and provides a global point of access to it.

A singleton class should have following features:

* Constructor of the class should be private: Since you don’t want to create multiple instances of the class, hence the constructor of the class should be private.
* The member that will be responsible for instance creation will be public and static so that members of the class can be accessed globally.

**Working:**

****

In this game when we press the spacebar then it counts shown here. We are implementing the singleton pattern as the method which counts spaces is accessible throughout the game by using its class instance which is extending a singleton class.

**GAME OBJECT NAME**

**Player**

#### **SCRIPTS**

**Singleton.cs** >> Used to create generic singleton..

**SpaceCounter.cs** >> **Player >>** It holds a method to count spacebar pressed.

**Play.cs** >> **Player >>** It is using a public method declared in SpaceCounter.cs.

#### **FUNCTIONALITY**

1) Singleton class is used to create a generic singleton so that if we want to

create instance of any class then we directly implement from this singleton

class. (**Singleton.cs** >> **Awake**)

2) SpaceCounter class has a public method to count space but we have to

use this method into other script so we created instance of SpaceCounter

simply implemented from singleton class. (**SpaceCounter.cs** >> **IncreaseScore**)

3) The Play class is just using the public method of the SpaceCounter class using an instance.

(**Play.cs** >> **Update**)

**< < < < - - - - - - - - - - - - End - - - - - - - - - - - - > > > >**