# Tutorial: Making a Flyweight Game

## Introduction

This tutorial is designed to familiarise you with the flyweight software design pattern by guiding you through the flyweight design pattern is implemented. The game is set in a 3D perspective and is to be developed using the Unity game engine and the C# programming language.

## The Scenario

You control a rolling ball on the game screen. Navigate around rectangular cuboid and spherical obstacles that are generated using the flyweight design pattern to collect the yellow cubes. Yellow cubes increase your score count.  Try and get a high score!

## Prerequisites:

- Basic understanding of Unity
- General knowledge of the C# programming language and your IDE of choice
- Have read documentation on Flyweight design pattern
- Have completed Flyweight Quiz

## Development Environment Requirements

Unity: At least 5.5.2 or above
Microsoft Visual Studio 2013 or greater or Monodevelop (bundled with Unity)

## Creating Your Flyweight Game

1.Open up Unity and on the “New Project” screen make sure you name your project appropriately and select 3D as the project type. Selecting 3D will set up some Unity defaults e.g. Camera arrangements and default imports - you don’t need to worry about these details.

2. Create a script within the unity assets window called “EnemyShip”. This class will be where the intrinsic state of your objects (the shared properties) are kept. From the code snippet below you can paste this into your IDE after double clicking on the newly created script.

```
using System;
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public abstract class Enemyship : MonoBehaviour {

    private string shipName;

    public string getShipName() {
        return shipName;
    }

    public void setShipName(string input) {
        this.shipName = input;
    }

    public void shipAction() {
        Console.WriteLine(getShipName() + " is performing an action");
    }

      public void getShip(int rSx, int rSz, float wallPos, String t) {

            if(t.Equals("UFO")){
            GameObject ship =
GameObject.CreatePrimitive(PrimitiveType.Sphere);
                  Rigidbody rigidbody = ship.AddComponent<Rigidbody>
();
                  ship.transform.position = new Vector3(rSx , 0 , rSz);
            }
            else if(t.Equals("MOTHERSHIP")){
            GameObject ship2 =
GameObject.CreatePrimitive(PrimitiveType.Cube);
                  ship2.transform.localScale += new Vector3(5.0F, 0,
0);
                  Rigidbody rigidbody2 = ship2.AddComponent<Rigidbody>
();
                  ship2.transform.position = new Vector3(rSx , 0 ,
rSz);
            }
    }
}
```

In the above sample we can see quite a lot happening. What we're doing is defining some basic methods for our each of our flyweights to have, a name and object type and then assigning them to that entity depending on the object that is required to be instantiated. Additionally we're also randomly assigning a unity transform property (which randomly spawns the object on the map).

This class acts as our parent class for flyweight objects to inherit from. The states defined here will also be our shared properties that all object will have known as their intrinsic state.

3. Now that we have out intrinsic object state set up that our objects will inherit from we can begin adding our "objects" known as a "flyweights". For this we have to create another script called "MotherShip". This will be where we store the Extrinsic states of the object (the non-shared object properties) these will be determined by the program in this scenario these states are immutable and created when the object is created..

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Mothership : Enemyship {

object
    public void MotherShipEnemy() {
        setShipName("Mothership");
    }
}
```

Unlike the EnemyShip class our flyweights are very lightweight and only non-shared properties that the flyweights do not share are kept here. These can be randomly defined from a set of predefined variables or randomly generated themselves.. You may also have checks which ensures no two objects are created the same or that additional values such as HP/ATK power are defined here to spawn hidden special super monsters in your game!

4. One object simply isn't enough so let's add a basic UFO object too.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class Ufo : Enemyship {

    public void UFOEnemy() {
        setShipName("UFO");
    }
}
```

Just like MotherShip, our Ufo objects are very simple and only assign a simple extrinsic property.

5. Now that we have our objects set up we can create our factory class which will be used to create objects within unity. There a number of ways you can implement the flyweight design pattern, so you may not see other examples online use this method.

The factory class is a factory (another design pattern) which generates objects for you according to your specifications defined in that objects class.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class FlyweightFactory : MonoBehaviour {

 public Enemyship initShip(string ship) {

        Enemyship enemyShip = null;

        if (ship.Equals("UFO")) {
            return new Ufo();
        } else if (ship.Equals("MOTHERSHIP")) {
            return new Mothership();
        } else {
            return null;
        }
    }
}
```

This is a very simple factory that creates a *new* object each time one is requested in the FlyweightTesting class which we'll cover next.

The constructor takes a simple string argument and returns an object of type EnemyShip, which we saw before.

6. Now that we have all the essential parts to define our flyweight objects and also create them we need a way to get this program started before we can see anything on screen.

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class FlyweightTesting : MonoBehaviour {

    FlyweightFactory flyweightFactory = new FlyweightFactory();

    Enemyship enemyShip = null;

    string shipType = null;

      float rN = 1;
      int rSx = 0;
      int rSz = 0;
      float wallPos = 0;
      public float spawnPeriod;
      private float nextSpawnTime;

void Start () {
            nextSpawnTime = 1;
            spawnPeriod = 0.1f;
    }

void FixedUpdate () {
            rN = Random.Range ( 0.0f, 2.0f);
            wallPos = Random.Range ( 0.0f, 2.0f);
            rSx = Random.Range (-45, 45);
            rSz = Random.Range (-25, 25);

      spawnEnemy();
        //gathers user input to define the extrinsic state of the
object being created (name/type)
    }

      void spawnEnemy(){
```

```
            float TempR = rN;
            if (TempR <=1 && Time.time > nextSpawnTime) {
                  nextSpawnTime = Time.time + spawnPeriod;
                  shipType = "UFO";
                  enemyShip = flyweightFactory.initShip(shipType);
                  enemyActions(enemyShip,"UFO");

            } else if (TempR >=1 && Time.time > nextSpawnTime) {
                  nextSpawnTime = Time.time + spawnPeriod;
                  shipType = "MOTHERSHIP";
                  enemyShip = flyweightFactory.initShip(shipType);
                  enemyActions(enemyShip,"MOTHERSHIP");

            } else { }
      }

      void playerInput(){
            if (Input.GetButtonDown("Fire1")) {
                  shipType = "UFO";

                  enemyShip = flyweightFactory.initShip(shipType);
                  enemyActions(enemyShip,"UFO");
            } else if (Input.GetButtonDown("Jump")) {
                  shipType = "MOTHERSHIP";

                  enemyShip = flyweightFactory.initShip(shipType);
                  enemyActions(enemyShip,"MOTHERSHIP");
            } else { }
      }

void enemyActions(Enemyship anEnemyShip, string t) {
        anEnemyShip.getShip(rSx, rSz, wallPos, t);
        anEnemyShip.shipAction();
    }
}
```

So this is a big class, first we start by creating a factory object for us to use and create our flyweights from. Next are some game logic values. For example we are using spawn rate and default ship transform properties.

When `(TempR <=1 && Time.time > nextSpawnTime)` is true either a mothership or ufo object is created from the factory and spawned in the game for the player to interact with.

These simple enemies act as barriers to our players objective by prohibiting player movement around the map.

# And now you're done!

7. Not really, but you do have all the requirements to get started with creating your own flyweight objects in unity! You'll see in the project folder for the flyweight if you were paying close attention to the repository that there are other scripts included in the folder. You will see "PlayerController", "CameraController" and "Rotator". These classes came bundled with the "Roll a Ball" Unity demo included with Unity. You may wish to look over these classes to learn how to move objects, control the camera movements etc, but for this tutorial we're only concerned with the Flyweight objects themselves.

Generously we have created an example already made for you to see flyweight in use with a WebGL game you can play right in your browser or download for Windows through the Github repository. You'll be amazed how well the game runs despite the number of objects we have on screen! That's the power of flyweight.

Visit our website here:
https://ac31009-team1.github.io/flyweight/
View the github repository for the game here:
https://github.com/AC31009-Team1/flyweight-game/
Download source code or a windows binary of the game here:
https://github.com/AC31009-Team1/flyweight-game/releases/tag/v1.0