TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM
**TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG**
**KHOA CÔNG NGHỆ THÔNG TIN**

**ĐỒ ÁN CUỐI KÌ MÔN**

**MẪU THIẾT KẾ**

# ÁP DỤNG CÁC MẪU THIẾT KẾ VÀO GAME

*Người hướng dẫn*: **THẦY Vũ Đình Hồng**

*Người thực hiện*: **PHẠM THÀNH PHƯƠNG – 51603247**

Khoá : **20**

**THÀNH PHỐ HỒ CHÍ MINH, NĂM 2021**

TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM
**TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG**
**KHOA CÔNG NGHỆ THÔNG TIN**

**ĐỒ ÁN CUỐI KÌ MÔN**

**MẪU THIẾT KẾ**

# ÁP DỤNG CÁC MẪU THIẾT KẾ VÀO GAME

*Người hướng dẫn*: **THẦY Vũ Đình Hồng**

*Người thực hiện*: **PHẠM THÀNH PHƯƠNG – 51603247**

Khoá : **20**

**THÀNH PHỐ HỒ CHÍ MINH, NĂM 2021**

# LỜI CẢM ƠN

Để đồ án này được hoàn thành đúng tiến độ và đạt kết quả tốt đẹp, ngoài sự nỗ lực của em, còn có sự giúp đỡ, đóng góp ý kiến và hướng dẫn nhiệt tình của thầy cô trong khoa Công nghệ thông tin giúp em có thêm động lực để hoàn thành đồ án này.

Em xin gửi lời cảm ơn chân thành đến thầy Vũ Đình Hồng đã tận tình hướng dẫn em vượt qua khó khăn, vướng mắc trong suốt thời gian làm đồ án.

Em cũng xin chân thành cảm ơn toàn thể giảng viên, công nhân viên chức trường Đại học Tôn Đức Thắng đã tạo điều kiện, môi trường giáo dục tốt đẹp, cung cấp tài liệu và kiến thức đầy đủ giúp em có cơ sở lý thuyết vững vàng và tạo điều kiện giúp đỡ em trong quá trình học tập.

Với trình độ chuyên môn còn hạn chế, đồ án có bị thiếu sót và sai lầm ở một số chỗ mà em không tìm ra. Em rất mong nhận được sự chỉ dẫn, đóng góp ý kiến của thầy và bạn bè để em có thể bổ sung, cập nhật những sai lầm sau khi đồ án kết thúc, qua đó nâng cao ý thức của em, phục vụ tốt cho các dự án thực tế sau này.

Em xin chân thành cảm ơn.

*Tp. HCM, ngày 7 tháng 05 năm 2021*

*Tác giả*

*(Ký và ghi rõ họ tên)*

*Phạm Thành Phương*

# ĐỒ ÁN ĐƯỢC HOÀN THÀNH
# TẠI TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG

Chúng em xin cam đoan đồ án cuối kì **"Mẫu thiết kế"** là công trình nghiên cứu của sinh viên Phạm Thành Phương (51603247).Những phần sử dụng tài liệu tham khảo trong đồ án đã được nêu rõ trong phần tài liệu tham khảo. Các số liệu, kết quả trình bày trong đồ án là hoàn toàn trung thực, nếu sai phạm em xin chịu hoàn toàn trách nhiệm và chịu mọi ký luật của bộ môn và nhà trường.

**Nếu phát hiện có bất kỳ sự gian lận nào em xin hoàn toàn chịu trách nhiệm về nội dung đồ án của em.** Trường đại học Tôn Đức Thắng không liên quan đến những vi phạm tác quyền, bản quyền do tôi gây ra trong quá trình thực hiện (nếu có).

*TP. Hồ Chí Minh, ngày 07 tháng 05 năm 2021*

*Tác giả*

*(ký tên và ghi rõ họ tên)*

*Phạm Thành Phương*

# PHẦN XÁC NHẬN VÀ ĐÁNH GIÁ CỦA GIẢNG VIÊN

**Phần xác nhận của GV hướng dẫn**

_____

_____

_____

_____

_____

_____

_____

Tp. Hồ Chí Minh, ngày  tháng  năm
(kí và ghi họ tên)

**Phần đánh giá của GV chấm bài**

_____

_____

_____

_____

_____

_____

_____

Tp. Hồ Chí Minh, ngày  tháng  năm
(kí và ghi họ tên)

# TÓM TẮT

Design Pattern là một kỹ thuật trong lập trình hướng đối tượng, nó khá quan trọng và mọi lập trình viên muốn giỏi đều phải biết. Được sử dụng thường xuyên trong các ngôn ngữ OOP. Nó sẽ cung cấp cho bạn các "mẫu thiết kế", giải pháp để giải quyết các vấn đề chung, thường gặp trong lập trình. Các vấn đề mà bạn gặp phải có thể bạn sẽ tự nghĩ ra cách giải quyết nhưng có thể nó chưa phải là tối ưu. Design Pattern giúp bạn giải quyết vấn đề một cách tối ưu nhất, cung cấp cho bạn các giải pháp trong lập trình OOP.

Design Patterns không phải là ngôn ngữ cụ thể nào cả. Nó có thể thực hiện được ở phần lớn các ngôn ngữ lập trình, chẳng hạn như Java, C#, thậm chí là Javascript hay bất kỳ ngôn ngữ lập trình nào khác.

Đặc biệt trong việc thiết kế game là một trong những nơi cần phải sử dụng các mẫu thiết kế cực nhiều. Lý do rất đơn giản là bởi game chạy luôn luôn cần được tối ưu, tối ưu hơn nữa, và việc nâng cấp hay chỉnh sửa sẽ được diễn ra thường xuyên nên nếu sử dụng các mẫu thiết kế sẽ cho ra được các tựa game được tối tưu và dễ dàng nâng cấp hơn nữa.

# MỤC LỤC

# CHƯƠNG 1 – GIỚI THIỆU VỀ ĐỀ TÀI

Đề tài cuối kỳ môn mẫu thiết kế làm về thiết kế một tựa game bắn súng 2d đi cảnh. Qua đó sử dụng các mẫu thiết kế để áp dụng vào trong game. Tựa game hiện tại đã sử dụng các mẫu thiết kế sau :

- Singleton Pattern
- Observer Pattern
- Command pattern
- State Pattern
- Object Pool
- Factory Pattern
- Memento Pattern
- Strategy Pattern
- Adapter Pattern
- Flyweight Pattern

Link video demo game : https://youtu.be/yCNU-ibLcnw

Link github của đồ án đã public tại : https://github.com/PhuongPham1203/Mau-thiet-ke-doan.git

Đồ án sử dụng nền tảng Unity3d để làm game. Yêu cầu Unity phiên bản 2019.4.15f1 hoặc cao hơn để chạy.

Các sơ đồ class được làm trong bài được vẽ trên https://app.diagrams.net/ được lưu tại file : .\documents\Mau thiet ke.drawio

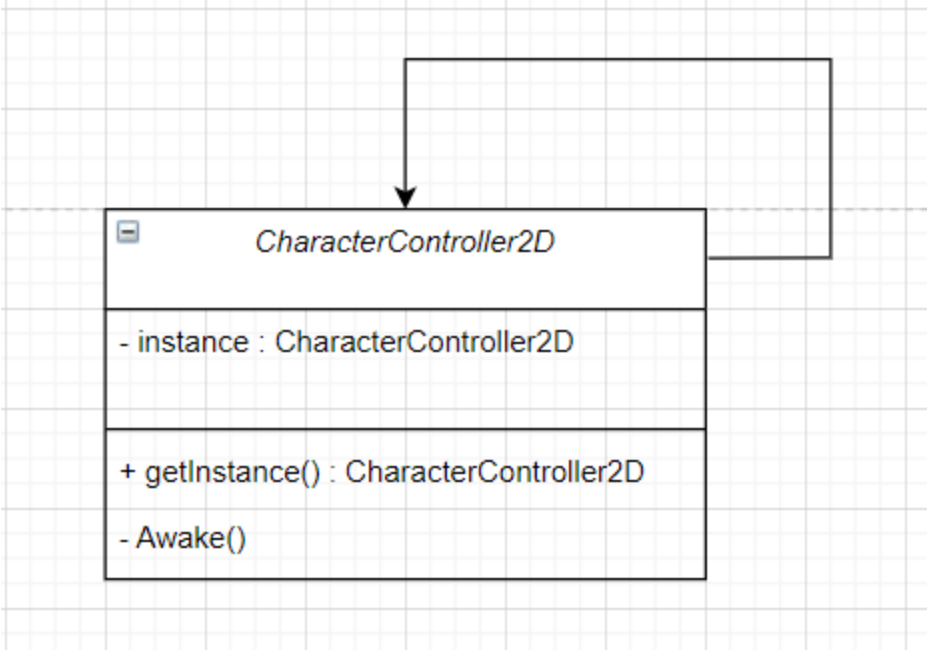
# CHƯƠNG 2 - SINGLETON PATTERN

## 2.1 Tại sao lại dùng Singleton Pattern

Trong một tựa game chúng ta sẽ có rất nhiều các đối tượng cần tồn tại duy nhất và có thể truy suất ở bất cứ nơi nào. Ví dụ : UI thanh máu của nhân vật chính ( hay player ) chỉ có một, nhân vật người chơi điều khiển chỉ có một, menu tùy chọn … những đối tượng này xuyên suốt cả màn chơi chỉ có một đối tượng duy nhất nên ta sẽ áp dụng Singleton Pattern.

Bên dưới sẽ mô tả việc thiết lập một Singleton Pattern cho điều khiển nhân vật chính (hay player).

## 2.2 Sơ đồ lớp




**Tại sao lại có hàm Awake() ở đây ?**

Hàm Awake() là một phương thức có sẵn trong unity và sẽ gọi ngay khi bắt đầu Scene game và sẽ khởi tạo Singleton.

## 2.3 Code áp dụng

Code được áp dụng tại : Assets/Scripts/CharacterController2D.cs

```
        3 references
43      private static CharacterController2D instance;
        14 references
44      public static CharacterController2D getInstance()
45      {
46          return instance;
47      }
48
        0 references
49      private void Awake()
50      {
51          if (instance == null)
52          {
53              instance = this;
54          }
55          else
56          {
57              Destroy(gameObject);
58          }
59
60          m_Rigidbody2D = GetComponent<Rigidbody2D>();
61
62          if (OnLandEvent == null)
63              OnLandEvent = new UnityEvent();
64
65          if (OnCrouchEvent == null)
66              OnCrouchEvent = new BoolEvent();
67
68
69      }
```

Trong Unity3D hàm Awake() sẽ khởi tạo một CharacterController2D và sẽ kiểm tra luôn nếu nó đã tồn tại ta có thể xóa luôn gameObject đó để tránh việc có nhiều CharacterController2D.

Để lấy CharacterController2D ta chỉ việc gọi hàm getInstance() là lấy được.

Ví dụ tại khi đến điểm lưu tạm trong game chúng ta sẽ gọi đến CharacterController2D để lưu tại tạm thời trạng thái của nhân vật chính.


Code tại file : Assets/Scripts/MementoPattern/Checkpoint.cs

```
0 references
public class Checkpoint : MonoBehaviour
{
    0 references
    private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.gameObject.layer == 8)
        {

            CharacterController2D controller = CharacterController2D.getInstance();

            // * Luu tam

            controller.careTaker.LevelMarker = controller.CreateMarker(controller.currenthp, controller.money, controller.score, controller.transform.position);


        }
    }
}

```

# CHƯƠNG 3 - OBSERVER PATTERN

## 3.1 Tại sao lại dùng Observer Pattern

Các đối tượng trong game luôn luôn có sự tương tác với nhau. Như viên đạn va vào kẻ địch, nhân vật chính nhận nhiệm vụ mới, nhân vật chính bị thương và trừ máu, giết kẻ địch sẽ được cộng tiền,… Trong trường hợp khi một số đối tượng nhất định cần được thông báo thường xuyên về những thay đổi xảy ra trong các đối tượng khác. Để có một thiết kế tốt có nghĩa là tách rời càng nhiều càng tốt và giảm sự phụ thuộc. Mẫu thiết kế Observer (quan sát) có thể được sử dụng bất cứ khi nào mà một đối tượng có sự thay đổi trạng thái, tất các thành phần phụ thuộc của nó sẽ được thông báo và cập nhật một cách tự động.

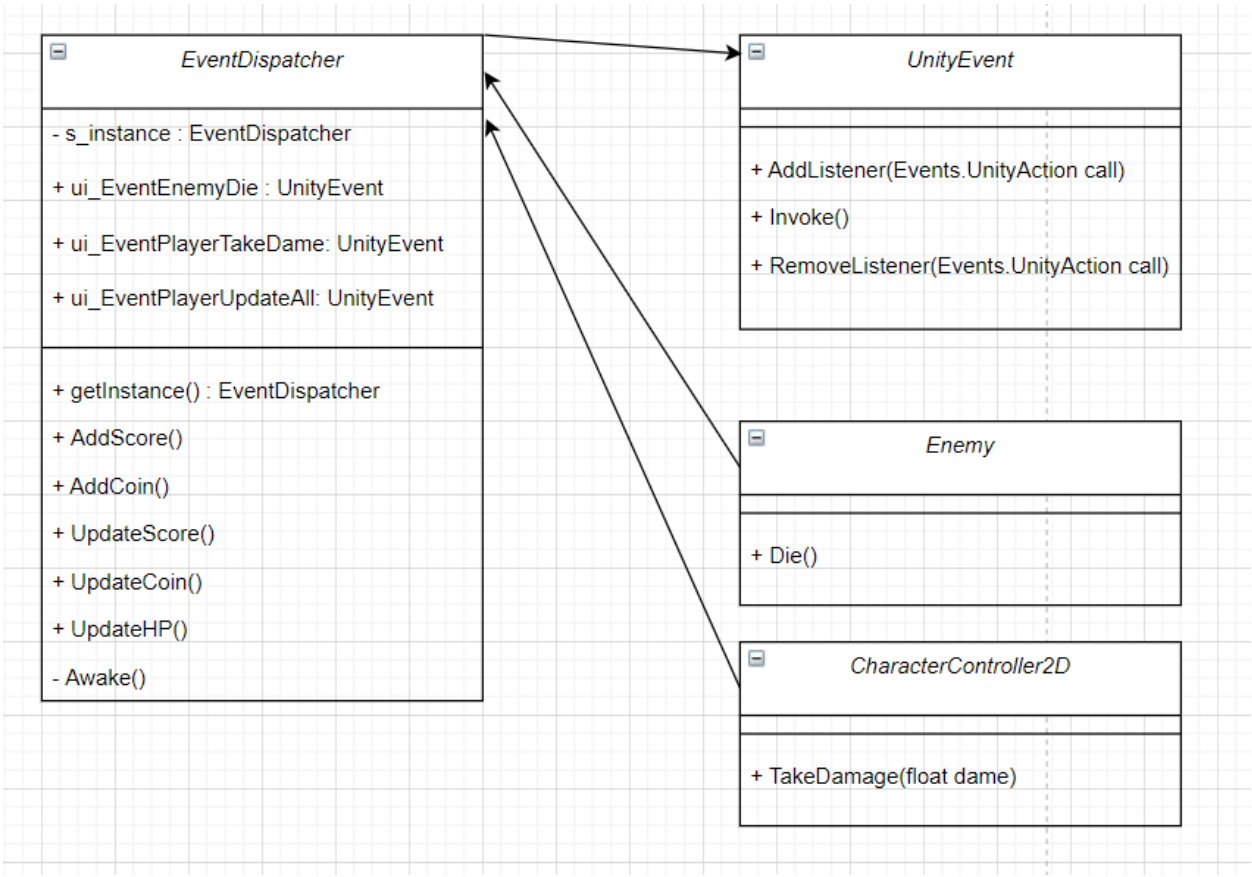
Một ví dụ cho thấy cần sử dụng Observer Pattern : mỗi khi chúng ta giết một quân định chúng ta cần gọi đến : nhiệm vụ, UI điểm số, UI tiền, UI thông báo thành tích… và mỗi khi xảy ra ta lại code toàn bộ thứ trên để thông báo , điều này gây thừa mứa code và lãng phí thời gian. Thế nên ta cần sử dụng Observer Pattern để kết nối 1-n các Object lại với nhau đê thông báo.

Trong C# chúng ta có Event để sử dụng để xây dựng Observer Pattern và Unity3d cũng có cơ chế tương tự là UnityEvents. Và ở bài này chúng ta sẽ sử dụng nó để áp dụng.

Áp dụng vào bài chúng ta có các sự kiện mỗi khi có kẻ địch chết sẽ cần cộng thêm điểm và tiền cho người chơi và cập nhật trên UI hiển thị. Và mỗi khi người chơi bị thương sẽ cần cập nhật thanh máu.


70
20

## 3.2 Sơ đồ lớp


EventDispatcher
- s_instance : EventDispatcher
+ ui_EventEnemyDie : UnityEvent
+ ui_EventPlayerTakeDame: UnityEvent
+ ui_EventPlayerUpdateAll: UnityEvent
+ getInstance() : EventDispatcher
+ AddScore()
+ AddCoin()
+ UpdateScore()
+ UpdateCoin()
+ UpdateHP()
- Awake()
UnityEvent
+ AddListener(Events.UnityAction call)
+ Invoke()
+ RemoveListener(Events.UnityAction call)
Enemy
+ Die()
CharacterController2D
+ TakeDamage(float dame)

## 3.3 Code áp dụng

Code được áp dụng tại :

Assets/Scripts/ObserverPattern/EventDispatcher.cs

Assets/Scripts/CharacterController2D.cs

Assets/Scripts/Enemy.cs

```
6 references
public class EventDispatcher : MonoBehaviour
{
    2 references
    public Text score;
    2 references
    public Text coin;
    1 reference
    public Image hp;

    3 references
    public UnityEvent ui_EventEnemyDie = new UnityEvent();
    2 references
    public UnityEvent ui_EventPlayerTakeDame = new UnityEvent();
    5 references
    public UnityEvent ui_EventPlayerUpdateAll = new UnityEvent();



    #region Singleton
    // * su dung singleton
    3 references
    private static EventDispatcher s_instance;
    4 references
    public static EventDispatcher GetInstance()
    {
        return s_instance;
    }

```

```
29      void Awake()
30      {
31          if (s_instance == null)
32          {
33              s_instance = this;
34
35          }
36          else
37          {
38              Destroy(gameObject);
39          }
40
41          this.ui_EventEnemyDie.AddListener(AddScore);
42          this.ui_EventEnemyDie.AddListener(AddCoin);
43          this.ui_EventPlayerTakeDame.AddListener(UpdateHP);
44
45          this.ui_EventPlayerUpdateAll.AddListener(UpdateCoin);
46          this.ui_EventPlayerUpdateAll.AddListener(UpdateScore);
47          this.ui_EventPlayerUpdateAll.AddListener(UpdateHP);
48      }
49
50
51
52      #endregion
53
            1 reference
54      public void AddScore()
55      {
56
57          CharacterController2D.getInstance().score+=10;
58          this.score.text = CharacterController2D.getInstance().score.ToString();
59      }
60
            1 reference
61      public void AddCoin()
62      {
63          CharacterController2D.getInstance().money+=10;
64          this.coin.text = CharacterController2D.getInstance().money.ToString();
65      }
66
```

Trong UnityEvent chúng ta có 3 phương thức :

AddListener() – Thêm event

RemoveListener() – xóa event

Invoke() – gọi đến event

Mỗi khi chúng ta gọi hàm Invoke() các event đã được thêm vào từ trước sẽ chạy.

Bên trên chúng ta đã khơi tạo và thêm các event cần lắng nghe.

Tiếp theo chúng ta sẽ gọi Invoke() để gọi khi có sự kiện xảy ra.

```
195     // * CharacterController2D.cs
        2 references
196     public void TakeDamage(float dame)
197     {
198         this.currenthp -= dame;
199         if (this.currenthp < 0)
200         {
201             this.currenthp = 0;
202         }
203         EventDispatcher.GetInstance().ui_EventPlayerTakeDame.Invoke();
204
205         if (this.currenthp <= 0)
206         {
207             Debug.Log("Player Die");
208             //SceneManager.LoadScene(SceneManager.GetActiveScene().name);
209
210             this.RestoreLevel(this.careTaker.LevelMarker);
211         }
212
213     }
214
```

```
57      // * Enemy.cs
        2 references
58      public virtual void Die()
59      {
60          EventDispatcher.GetInstance().ui_EventEnemyDie.Invoke();
61          Instantiate(deathEffect, transform.position, Quaternion.identity);
62          Destroy(gameObject);
63      }
64
```

# CHƯƠNG 4 - COMMAND PATTERN

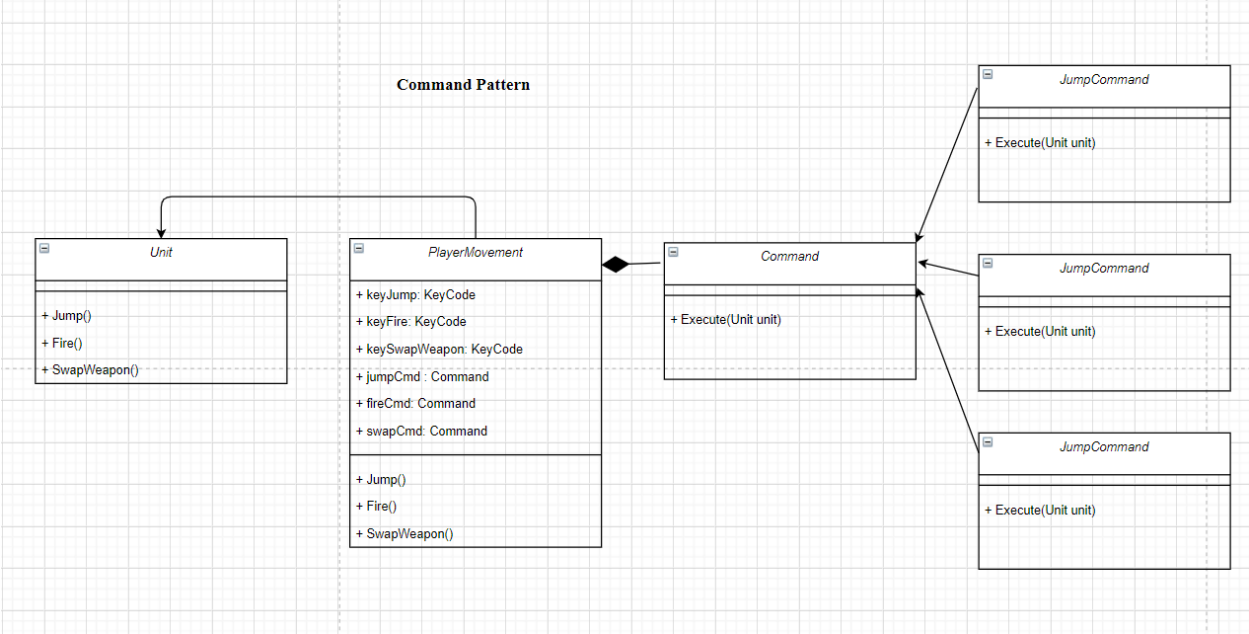
## 3.1 Tại sao lại dùng Command Pattern

Trong một  tựa game mỗi người chơi sẽ có thói quen hay sở thích khác nhau về cách điều khiển. Như người thì thích dùng phím AWSD để di chuyển, có người thuận thay phải

lại thích dùng phím mũi tên để di chuyển, các phím chức năng cũng như thế. Việc thay đổi nút để điều khiển nhân vật chính là cần thiết. Chưa kể đến việc có người còn thích sử dụng tay cầm để chơi game nữa. Từ đó việc code chay để gán phím sử dụng là rất bất tiện.

Chúng ta sẽ cần sử dụng Command pattern để dễ dàng cho việc User muốn thay gì thì thay.

## 3.2 Sơ đồ lớp



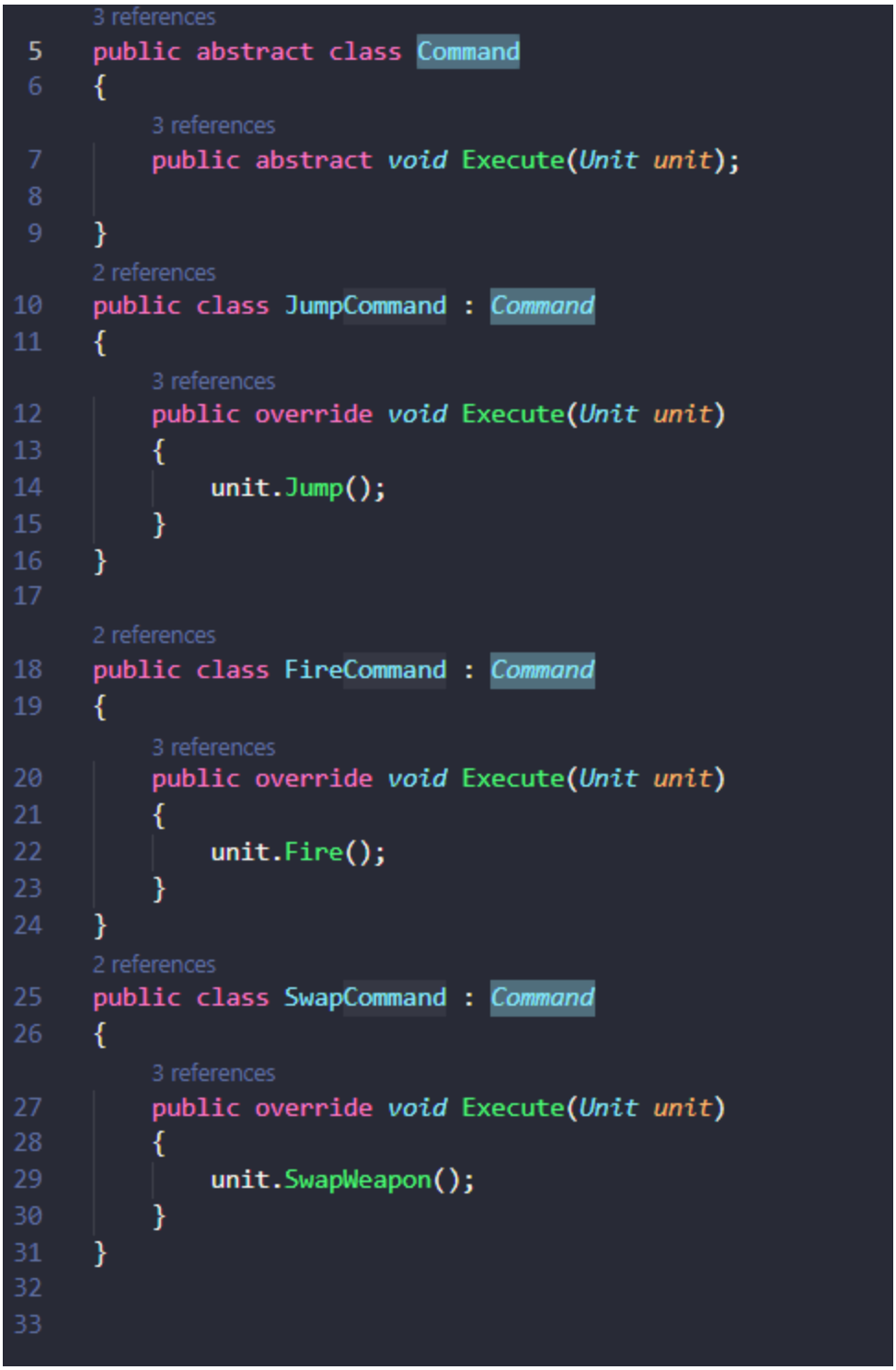

## 3.3 Code áp dụng

Code được áp dụng tại :

Assets/Scripts/CommandPattern/Command.cs

Assets/Scripts/CommandPattern/Unit.cs

Assets/Scripts/PlayerMovement.cs



```
3 references
public abstract class Command
{
    3 references
    public abstract void Execute(Unit unit);

}
2 references
public class JumpCommand : Command
{
    3 references
    public override void Execute(Unit unit)
    {
        unit.Jump();
    }
}

2 references
public class FireCommand : Command
{
    3 references
    public override void Execute(Unit unit)
    {
        unit.Fire();
    }
}
2 references
public class SwapCommand : Command
{
    3 references
    public override void Execute(Unit unit)
    {
        unit.SwapWeapon();
    }
}
```

Ta sẽ định nghĩa Command. Sau đó Kế thừa class Command để định nghĩa ra class JumpCommand, SwapCommand và class FireCommand.

Khai báo và sử dụng tại PlayerMovement.cs

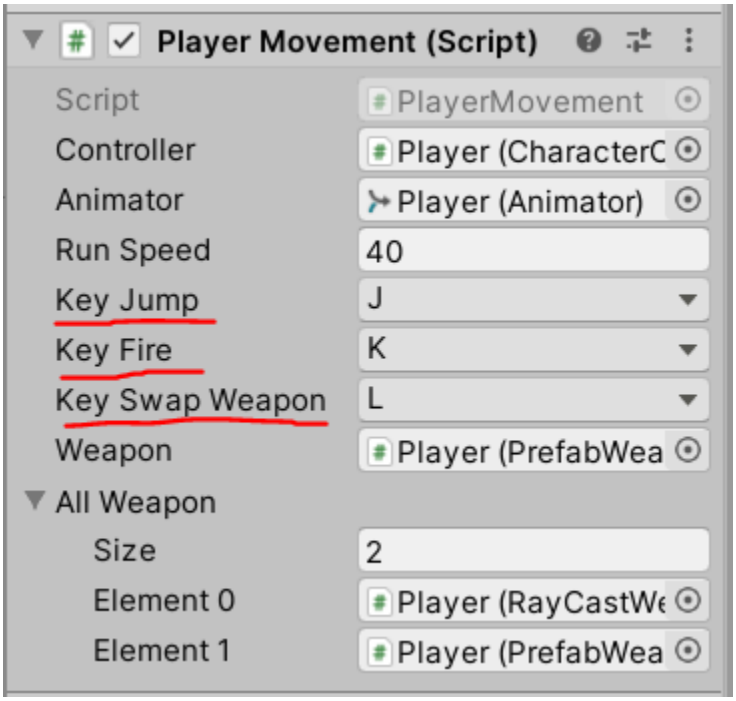
```
17      // * PlayerMovement.cs
18      // * Input Controller - Command Pattern
19      //public Unit unitControl;
            1 reference
20      public KeyCode keyJump;
            1 reference
21      public KeyCode keyFire;
            1 reference
22      public KeyCode keySwapWeapon;
23
            1 reference
24      private Command jumpCmd = new JumpCommand();
            1 reference
25      private Command fireCmd = new FireCommand();
            1 reference
26      private Command swapCmd = new SwapCommand();
27
```

```
        1 reference
86      public override void Jump()
87      {
88          jump = true;
89          animator.SetBool("IsJumping", true);
90      }
        1 reference
91      public override void SwapWeapon()
92      {
93          if (this.weapon.GetInstanceID() == this.allWeapon[this.allWeapon.Count -
94          {
95              this.weapon.enabled = false;
96              this.weapon = this.allWeapon[0];
97              this.weapon.enabled = true;
98
99              return;
100         }
101         //base.SwapWeapon();
102         for (int i = 0; i < this.allWeapon.Count; i++)
103         {
104
105             if (this.allWeapon[i].GetInstanceID() == this.weapon.GetInstanceID())
106             {
107                 this.weapon.enabled = false;
108                 this.weapon = this.allWeapon[i + 1];
109                 this.weapon.enabled = true;
110
111                 break;
112             }
113         }
114     }
        1 reference
115     public override void Fire()
116     {
117         //Debug.Log("fire");
118         this.weapon.Shoot();
119     }
120
```

Sử dụng các Command bên trên tại Update().

```
37      // Update is called once per frame
        0 references
38      void Update()
39      {
40
41          horizontalMove = Input.GetAxisRaw("Horizontal") * runSpeed;
42
43          animator.SetFloat("Speed", Mathf.Abs(horizontalMove));
44
45
46          if (Input.GetKeyDown(keyJump))
47          {
48              jumpCmd.Execute(this);
49          }
50
51          if (Input.GetButtonDown("Crouch"))
52          {
53              crouch = true;
54          }
55          else if (Input.GetButtonUp("Crouch"))
56          {
57              crouch = false;
58          }
59
60
61          // Swap
62          if (Input.GetKeyDown(keySwapWeapon))
63          {
64              swapCmd.Execute(this);
65          }
66
67          if (Input.GetKeyDown(keyFire))
68          {
69              fireCmd.Execute(this);
70          }
71
```

Cài đặt code đã xong. Giờ chúng ta có thể đối nút cho người chơi điều khiển .



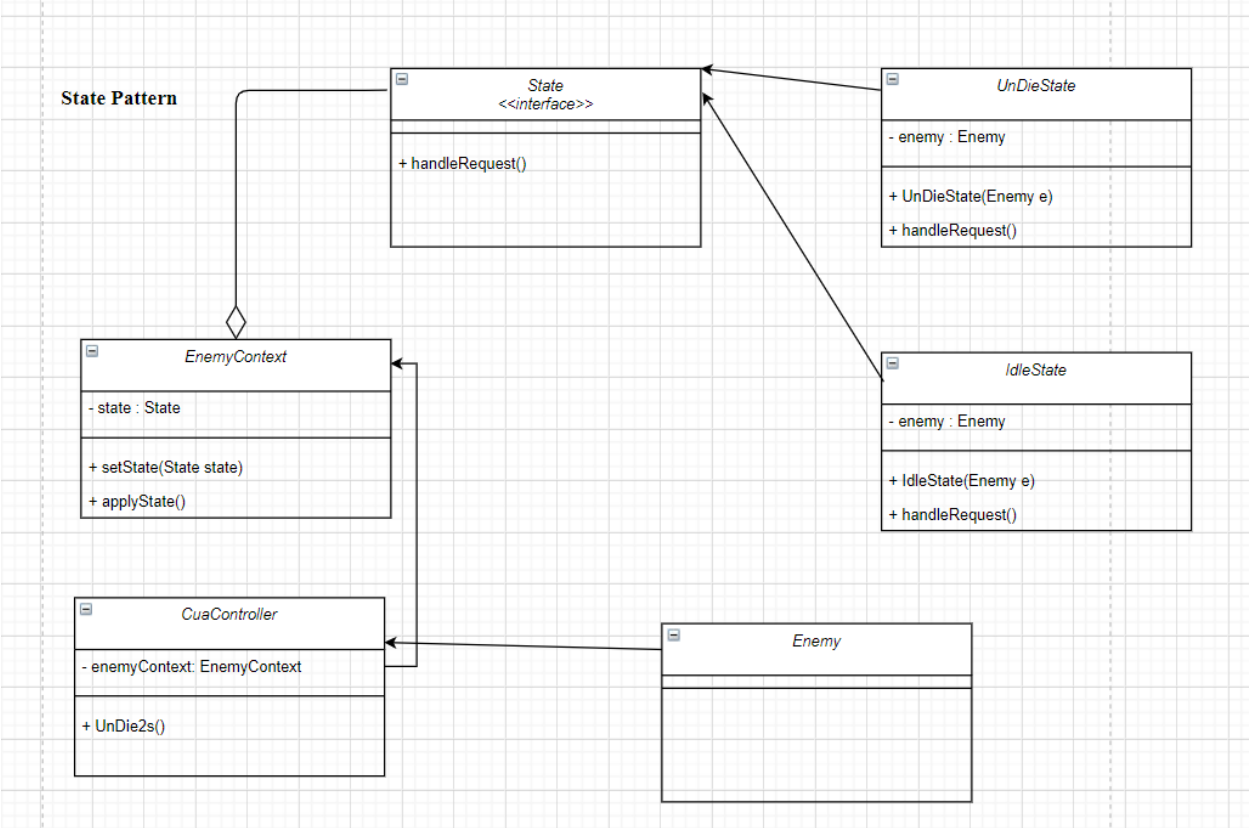

# CHƯƠNG 5 - STATE PATTERN

## 3.1 Tại sao lại dùng State Pattern

Trong game các đối tượng tại một thời điểm sẽ có các trạng thái khác nhau. Và các đối tượng đó sẽ thay đổi hành vi, hành động của mình dựa trên trạng thái đó. Ví dụ kẻ địch sẽ đứng yên nếu người chơi chưa đi vào trong vùng tấn công của kẻ địch, trạng thái này sẽ là Idle ( không làm gì, rảnh rỗi ). Khi kẻ địch sang trạng thái tấn công, sẽ có những hành vi khác để chống lại người chơi.

Trong ví dụ này kẻ địch là con cua sẽ có 2 trạng thái là Idle – đứng yên và UnDie – không thể bị thương.

## 3.2 Sơ đồ lớp




## 3.3 Code áp dụng

Code được áp dụng tại :

Assets/Scripts/StatePattern/State.cs

Assets/Scripts/StatePattern/IdleState.cs

Assets/Scripts/StatePattern/UnDieState.cs

Assets/Scripts/StatePattern/EnemyContext.cs

Assets/Scripts/StatePattern/CuaController.cs

Đầu tiên chúng ta tạo một interface State.

```
4 references
public interface State
{
    1 reference
    void handleRequest();
}
```

Tiếp theo chúng ta tạo ra các class implement interface bên trên

```
1 reference
public class UnDieState : State
{
    3 references
    private Enemy enemy;

    1 reference
    public UnDieState(Enemy e){
        this.enemy = e;
    }
    1 reference
    public void handleRequest(){
        this.enemy.stateEnemy = StateEnemy.UnDie;
        this.enemy.GetComponent<SpriteRenderer>().material.color = Color.red;
        //Debug.Log(this.enemy.name);
        //Debug.Log(this.enemy.stateEnemy);
    }
}

```

```
1 reference
public class IdleState : State
{
    3 references
    private Enemy enemy;

    1 reference
    public IdleState(Enemy e)
    {
        this.enemy = e;
    }
    1 reference
    public void handleRequest()
    {
        this.enemy.stateEnemy = StateEnemy.Idle;
        this.enemy.GetComponent<SpriteRenderer>().material.color = Color.white;

    }
}

```

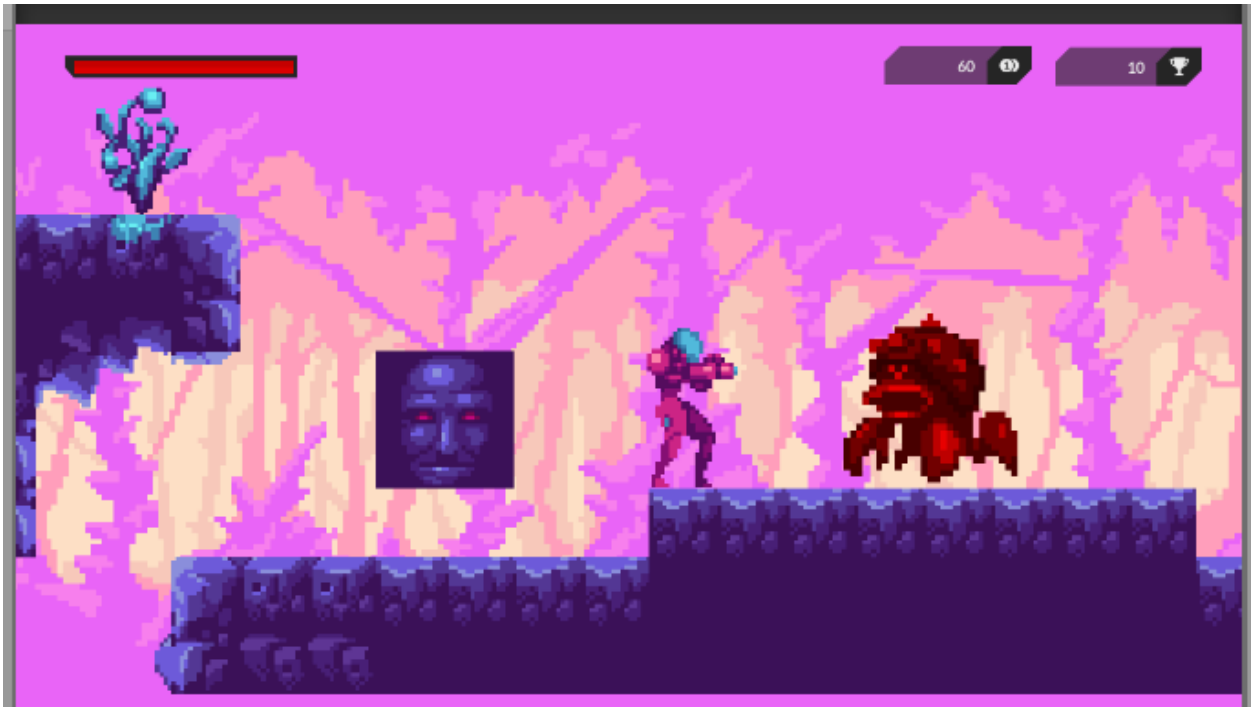
Cuối cùng, chúng ta tạo một class Context. Class này chứa thông tin State hiện tại và nhận yêu cầu xử lý trực tiếp từ CuaController.

```
2 references
public class EnemyContext
{
    2 references
    private State state;

    2 references
    public void setState(State state) {
        this.state = state;
    }

    2 references
    public void applyState() {
        this.state.handleRequest();
    }
}
```

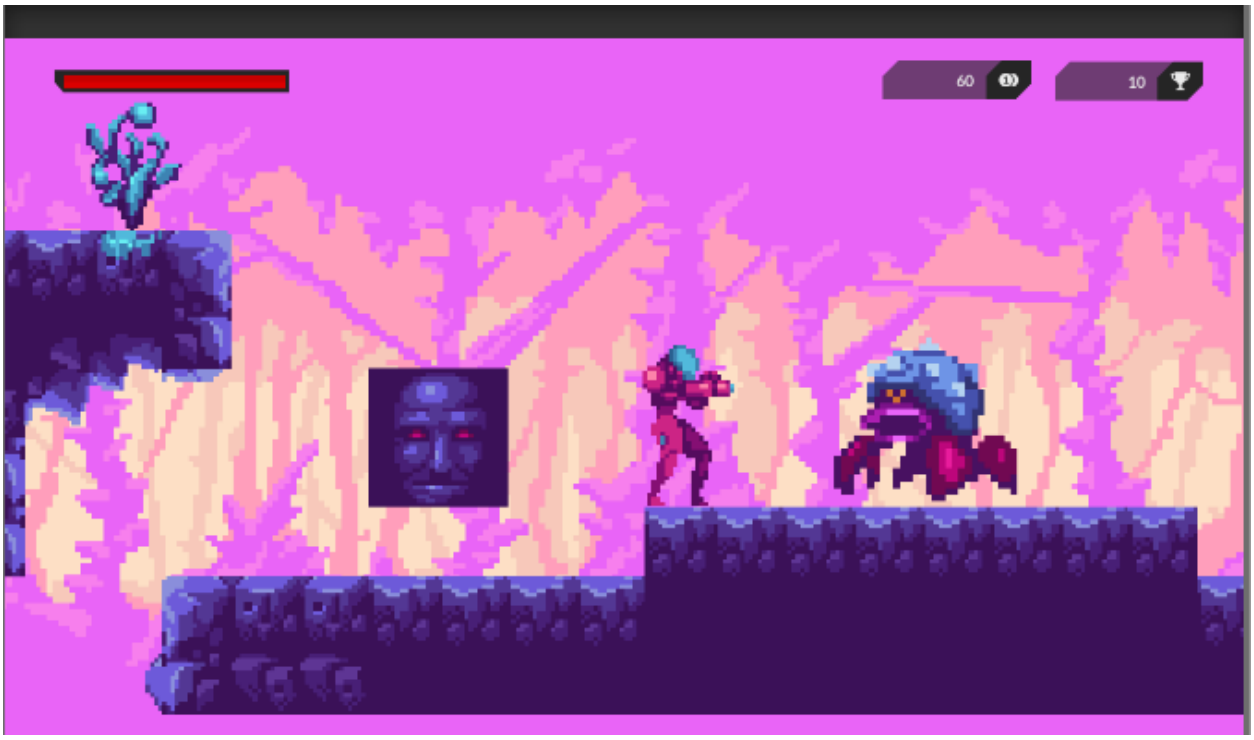
Và đây là điều khiển con cua này mỗi 2 giây sẽ bất tử và 2 giây sau thì bình thường.

```
0 references
public class CuaController : Enemy
{
    5 references
    private EnemyContext enemyContext;
    // Start is called before the first frame update
    0 references
    void Start()
    {
        this.enemyContext = new EnemyContext();


        InvokeRepeating("UnDie2s", 1.0f, 2f);

    }


    0 references
    public void UnDie2s()
    {

        if (this.stateEnemy == StateEnemy.Idle)
        {
            //Debug.Log("run idle");
            enemyContext.setState(new UnDieState(this));
            enemyContext.applyState();
        }
        else if (this.stateEnemy == StateEnemy.UnDie)
        {
            //Debug.Log("run undie");

            enemyContext.setState(new IdleState(this));
            enemyContext.applyState();

        }
    }

}

```

Lúc con cua màu đỏ là bất tử :




Và lúc bình thường :

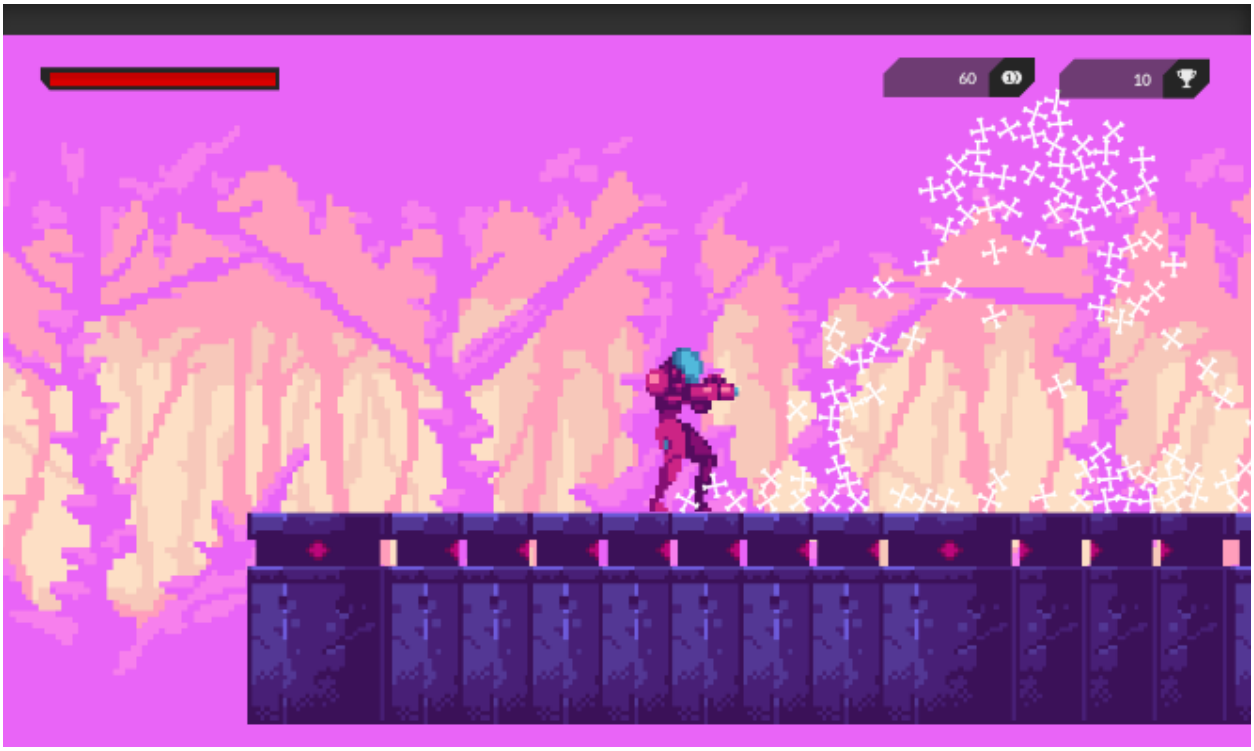


# CHƯƠNG 6 - OBJECT POOL

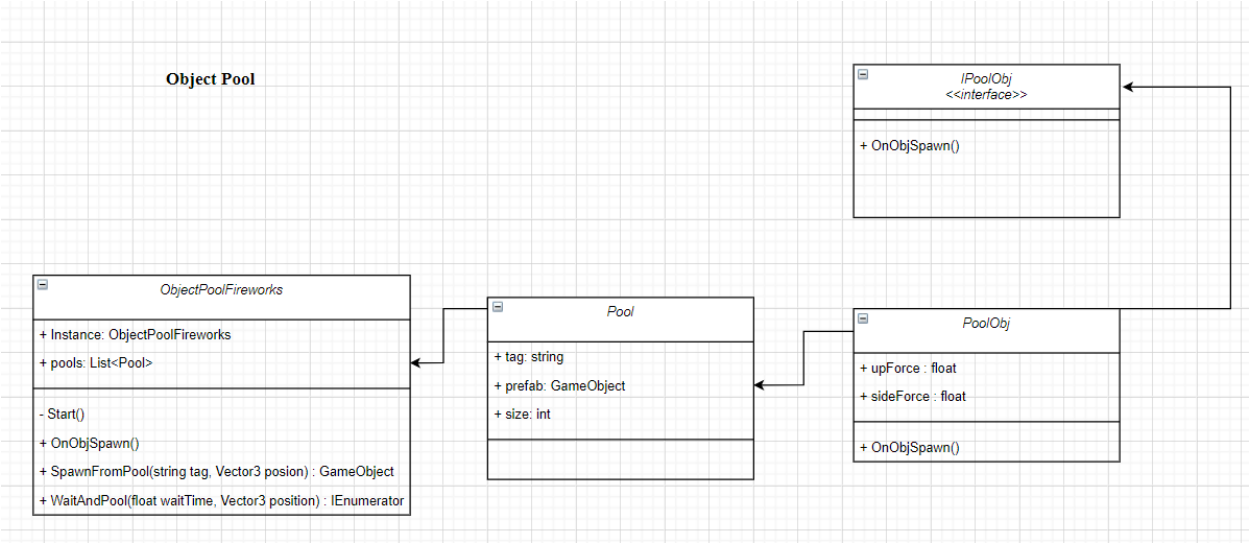
## 3.1 Tại sao lại dùng Object Pool

Trong các tựa game, ví dụ dễ thấy nhất là trong game bắn gà, chúng ta có thể thấy số lượng đạn bắn ra bay kín màn hình. Mỗi lần bắn là một lần khởi tạo viên đạn. Điều này sẽ gây ra chi phí khởi tạo đối tượng rất là lớn. Nếu số lượng nhỏ thì không sao những khi đạn lên đến cả trăm cả ngàn thì lúc đó game chạy sẽ rất chậm tốn chi phí cpu nhiều. Đặc biệt không tốt cho trải nghiệm của game thủ. Vì thế chúng ta có thể sử dụng Object Pool để giải quyết vấn đề này. Số lượng đạn sẽ được tạo ra sẵn ngay từ khi bắt đầu trò chơi. Khi nào cần đến chúng ta sẽ lấy ra dùng và dùng xong thì trả lại vào pool.

Tại tựa game này Object Pool được sử dụng vào việc tạo hiệu ứng khi game thủ đánh bại con boss. Khi con boss chết, xương của con boss sẽ bung ra rất nhiều. Và khi con boss tiếp theo chết, ta cũng sử dụng chính những cái xương đó để bung ra trên màn hình.



## 3.2 Sơ đồ lớp




## 3.3 Code áp dụng

Code áp dụng tại :

Assets/Scripts/ObjectPool/IPoolObj.cs

Assets/Scripts/ObjectPool/ObjectPoolFireworks.cs

Assets/Scripts/ObjectPool/PoolObj.cs

Đầu tiên chúng ta tạo interface :

```
    3 references
5   public interface IPoolObj
6   {
        1 reference
7       void OnObjSpawn();
8   }
9
```

Tiếp đến tạo gameobject kế thừa nó

```
0 references
public class PoolObj : MonoBehaviour,IPoolObj
{
    2 references
    public float upForce = 1f;
    2 references
    public float sideForce = 0.1f;

    1 reference
    public void OnObjSpawn(){
        float xForce = Random.Range(-sideForce,sideForce);
        float yForce = Random.Range(-upForce/2f,upForce);

        Vector2 force = new Vector2(xForce,yForce);

        this.GetComponent<Rigidbody2D>().velocity = force;
    }
}
```

```
[System.Serializable]
2 references
public class Pool
{
    1 reference
    public string tag;
    1 reference
    public GameObject prefab;
    1 reference
    public int size;
}
```

Class Pool dùng để định nghĩa, gán tag cho gameobject bên trên và xác định số lượng gameobject trong pool.

Tiếp theo chúng ta khởi tạo các gameobject đó khi bắt đầu scene game.

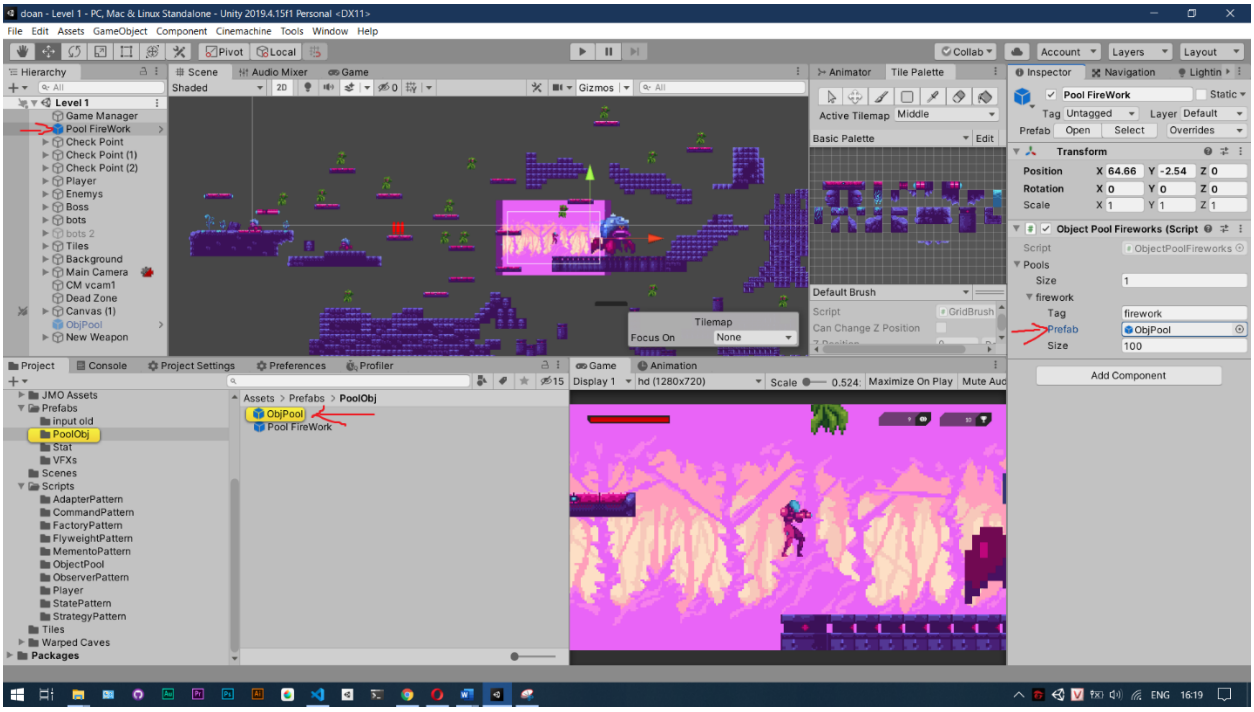
```
15      public static ObjectPoolFireworks Instance;
        0 references
16      public void Awake()
17      {
18          if (Instance == null)
19          {
20              Instance = this;
21          }
22          else
23          {
24              Destroy(gameObject);
25          }
26      }
27
        1 reference
28      public List<Pool> pools;
        5 references
29      public Dictionary<string, Queue<GameObject>> poolDic;
30      // Start is called before the first frame update
        0 references
31      void Start()
32      {
33          this.poolDic = new Dictionary<string, Queue<GameObject>>();
34
35          foreach (Pool p in this.pools)
36          {
37
38              Queue<GameObject> objectPool = new Queue<GameObject>();
39              for (int i = 0; i < p.size; i++)
40              {
41                  GameObject obj = Instantiate(p.prefab);
42
43                  obj.SetActive(false);
44                  objectPool.Enqueue(obj);
45              }
46
47              this.poolDic.Add(p.tag, objectPool);
48          }
49      }
50
```

Tiếp đến là hàm gọi ra các gameobject trong pool

```
        1 reference
58      public GameObject SpawnFromPool(string tag, Vector3 posion)
59      {
60          if (!this.poolDic.ContainsKey(tag))
61          {
62              Debug.Log("Khong co tag nay");
63              return null;
64          }
65
66
67          GameObject objToSpawn = this.poolDic[tag].Dequeue();
68
69          objToSpawn.SetActive(true);
70          objToSpawn.transform.position = posion;
71
72
73          IPoolObj pO = objToSpawn.GetComponent<IPoolObj>();
74          if (pO != null)
75          {
76              pO.OnObjSpawn();
77          }
78
79
80          this.poolDic[tag].Enqueue(objToSpawn);
81
82          return objToSpawn;
83
84
85      }
86
87
88
        1 reference
89      public IEnumerator WaitAndPool(float waitTime, Vector3 position)
90      {
91          for (int i = 0; i < 150; i++)
92          {
93              yield return new WaitForSeconds(waitTime);
94              this.SpawnFromPool("firework", position);
95
96          }
97
98          //yield return new WaitForSeconds(3f);
```

Rồi cuối cùng là khi boss bị giết chúng ta sẽ chạy lấy ra các gameobject đó :

```
0 references
public class BossController : Enemy
{

    3 references
    ObjectPoolFireworks objectPoolFireworks;
    0 references
    private void Start() {
        this.objectPoolFireworks = ObjectPoolFireworks.Instance;
    }



    2 references
    public override void Die()
    {
        base.Die();

        this.objectPoolFireworks.StartCoroutine(this.objectPoolFireworks.WaitAndPool(0.01f,this.transform.position));

    }
}

```



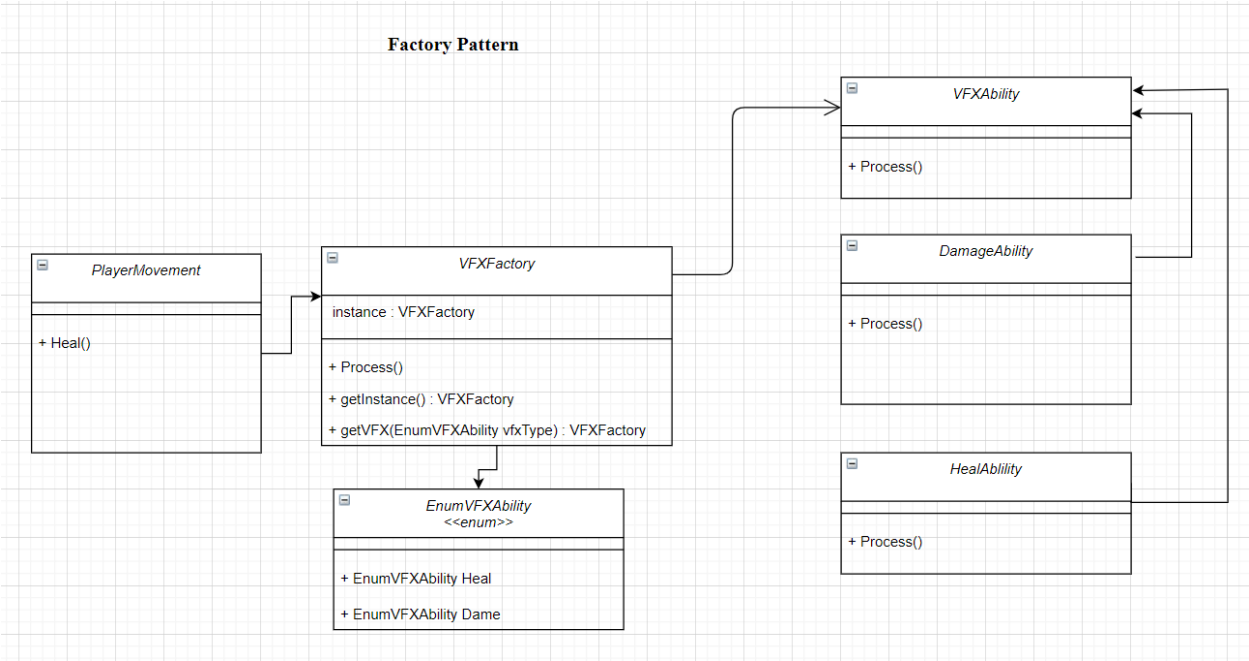
# CHƯƠNG 7 - FACTORY PATTERN

## 3.1 Tại sao lại dùng Factory Pattern

Một tựa game sẽ luôn có các đối tượng giống nhau về kiểu nhưng lại có các thành phần khác biệt. Khi sử dụng Factory Pattern chúng ta giảm sự phụ thuộc các module, chúng ta cần tạo một đối tượng, code chúng ta không bị thay đổi ở phía được gọi đến. Khi mở rộng chúng ta chỉ cần tạo thêm các sub class và implement thêm vào factory method.

Trong game này chúng ta sử dụng Factory pattern áp dụng vào việc tạo ra các Vfx và sau này có thêm chúng ta chỉ việc thêm tạo ra Vfx mới và thêm vào trong facrory method là xong.

## 3.2 Sơ đồ lớp




## 3.3 Code áp dụng

Code được áp dụng tại :

Assets/Scripts/FactoryPattern/VFXAbility.cs

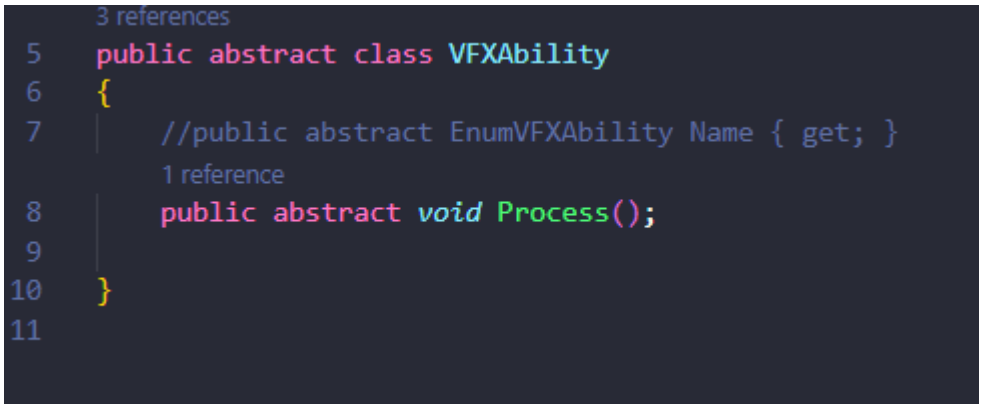
Assets/Scripts/FactoryPattern/VFXFactory.cs

Assets/Scripts/FactoryPattern/HealAblility.cs

Assets/Scripts/FactoryPattern/DamageAbility.cs

Assets/Scripts/PlayerMovement.cs

Factory Pattern được áp dụng vào việc tạo ra các hiệu ứng cho game. Cụ thể lần sử dụng này là tạo hiệu ứng hồi máu. Đầu tiên là tạo abstract class VFXAbility.






Khởi tạo sub class các hiệu ứng sẽ dùng.

Cuối cùng là tạo VFXFactory để tạo ra các vfx cần dùng đến.

```
3 references
public class VFXFactory : MonoBehaviour
{
    3 references
    private static VFXFactory instance;
    1 reference
    public static VFXFactory getInstance()
    {
        return instance;
    }

    public VFXAbility getVFX(EnumVFXAbility vfxType)
    {
        switch (vfxType)
        {
            case EnumVFXAbility.Heal:
                return new HealAblility();

            case EnumVFXAbility.Dame:
                return new HealAblility();

            default:
                Debug.Log("Not have VFX");
                return null;
        }
    }
```

Hàm getVFX gọi đến sẽ trả về hiệu ứng tương ứng được thêm vào.

Sử dụng enum để định nghĩa các kiểu vfx tương ứng.

```
4 references | 2 references | 1 reference
public enum EnumVFXAbility{Heal,Dame}
```

Mỗi khi sử dụng ta chỉ việc gọi đến nó. Trong ví dụ dưới đây là khi người chơi hồi máu.

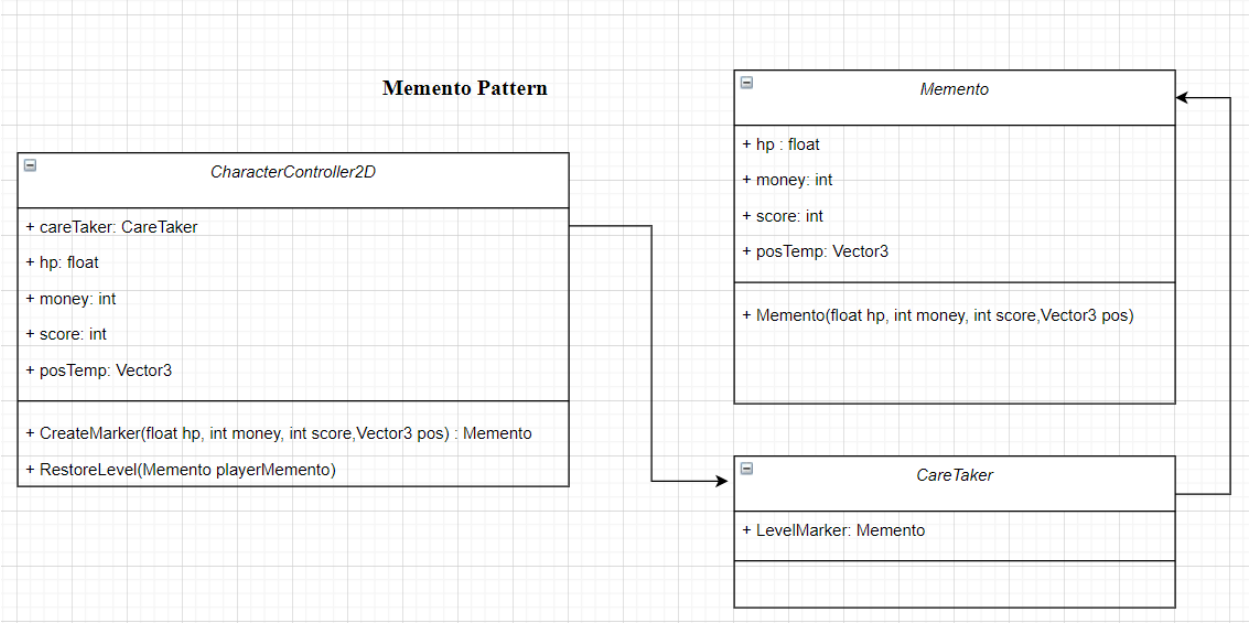
```
73          if (Input.GetKeyDown(KeyCode.Alpha1))
74          {
75              // * Heal
76              this.Heal();
77          }
78
79      }
        1 reference
80      public void Heal()
81      {
82          controller.currenthp += 10;
83          if (controller.currenthp > controller.hp)
84          {
85              controller.currenthp = controller.hp;
86          }
87          VFXFactory.getInstance().getVFX(EnumVFXAbility.Heal).Process();
88      }
89
```

# CHƯƠNG 8 - MEMENTO PATTERN

## 3.1 Tại sao lại dùng Memento Pattern

Khi chơi game chúng ta cần lưu lại trạng thái của nhân vật cũng như nhiều thứ khác nếu không mỗi lần khi người chơi chết sẽ không thể khôi phục lại trạng thái trước đó. Trong game sẽ có những điểm checkpoint tạm thời và khi người chơi chết mà vẫn còn mạng sẽ được trở lại check point tạm thời đó và khôi phục trạng thái trước. Memento Pattern là cách để giải quyết vấn đề bên trên.

## 3.2 Sơ đồ lớp




## 3.3 Code áp dụng

Code được áp dụng tại :

Assets/Scripts/MementoPattern/Checkpoint.cs

Assets/Scripts/MementoPattern/Stat.cs

Assets/Scripts/CharacterController2D.cs

Đầu tiên chúng ta tạo class Memento và Caretaker  dùng để lưu trữ thông tin của người chơi khi đi qua điểm check point

```
237   //'Memento' class
      4 references
238   public class Memento
239   {
          2 references
240       public float hp;
          2 references
241       public int money;
          2 references
242       public int score;
          2 references
243       public Vector3 posTemp;
244
          1 reference
245       public Memento(float hp, int money, int score,Vector3 pos)
246       {
247           this.hp = hp;
248           this.money = money;
249           this.score = score;
250           this.posTemp = pos;
251       }
252   }
253   //'CareTaker' class
      2 references
254   public class CareTaker
255   {
256       // save check point before go to next level
          4 references
257       public Memento LevelMarker;
258   }
```

Tiếp theo là hàm tạo và khôi phục dữ liệu đã lưu :

```
215     // * Memento
        2 references
216     public Memento CreateMarker(float hp, int money, int score,Vector3 pos)
217     {
218         return new Memento(hp, money, score,pos);
219     }
220
        2 references
221     public void RestoreLevel(Memento playerMemento)
222     {
223         this.currenthp = playerMemento.hp;
224         this.money = playerMemento.money;
225         this.score = playerMemento.score;
226         this.transform.position = playerMemento.posTemp;
227
228         EventDispatcher.GetInstance().ui_EventPlayerUpdateAll.Invoke();
229
230     }
231
232
233
234
235 }
```

Nếu người chơi chết sẽ quay về điểm check point :

```
195     // * CharacterController2D.cs
        2 references
196     public void TakeDamage(float dame)
197     {
198         this.currenthp -= dame;
199         if (this.currenthp < 0)
200         {
201             this.currenthp = 0;
202         }
203         EventDispatcher.GetInstance().ui_EventPlayerTakeDame.Invoke();
204
205         if (this.currenthp <= 0)
206         {
207             Debug.Log("Player Die");
208             //SceneManager.LoadScene(SceneManager.GetActiveScene().name);
209
210             this.RestoreLevel(this.careTaker.LevelMarker);
211         }
212
213     }
214
```

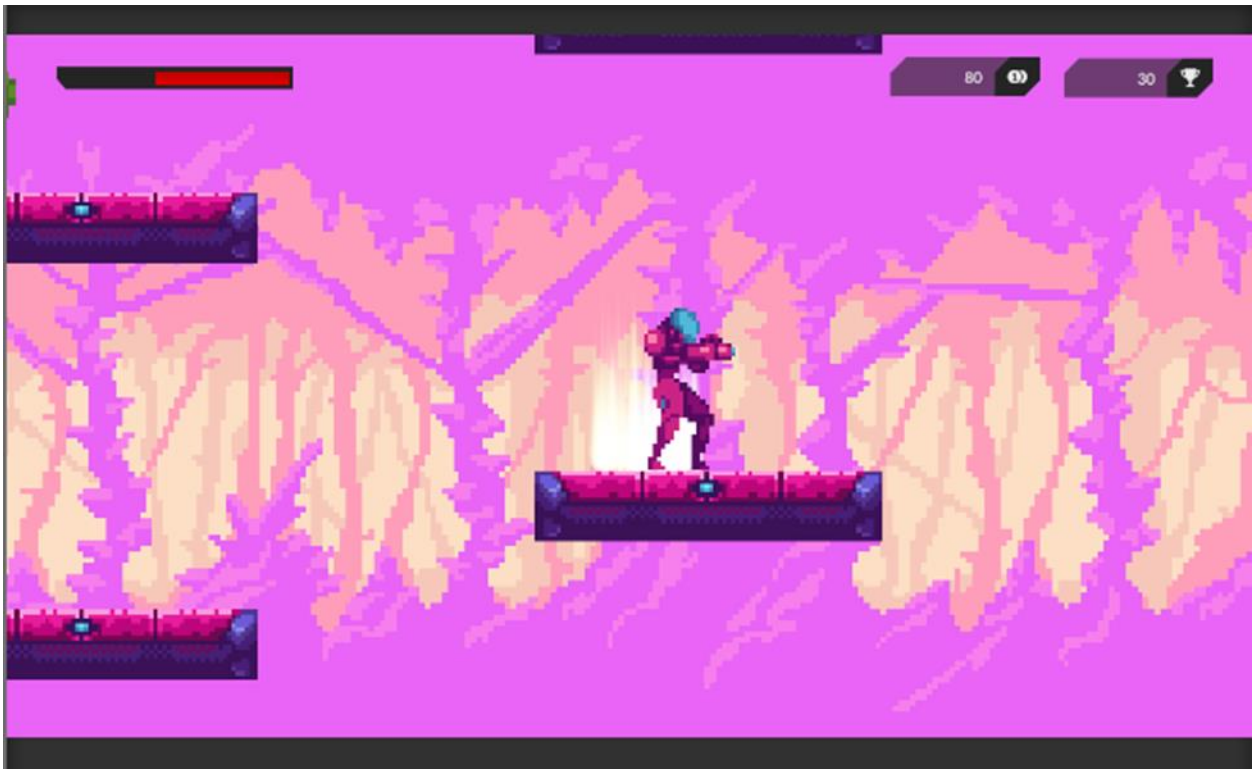
Tiếp đến là điểm check point :

```
0 references
public class Checkpoint : MonoBehaviour
{
    0 references
    private void OnTriggerEnter2D(Collider2D other)
    {
        if (other.gameObject.layer == 8)
        {

            CharacterController2D controller = CharacterController2D.getInstance();

            // * Luu tam

            controller.careTaker.LevelMarker = controller.CreateMarker(controller.currenthp, controller.money, controller.score, controller.transform.position);


        }
    }
}

```

Và đây là trạng thái trước khi đến điểm check point và tiến đến điểm này :


Người chơi đi tiếp và thay đổi trạng thái và chỉ số :

Người chơi không may mắn và chết sẽ trở lại điểm check point quay lại các chỉ số ban đầu ở điểm check point:




# CHƯƠNG 9 - STRATEGY PATTERN

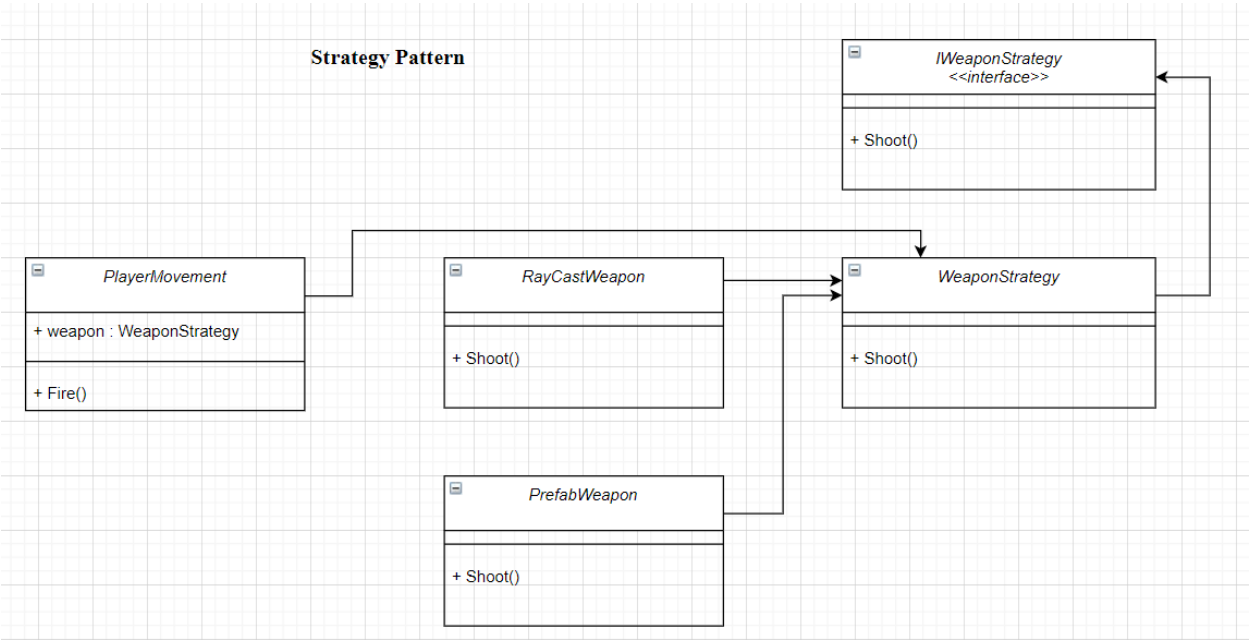
## 3.1 Tại sao lại dùng Strategy Pattern

Strategy Pattern là một trong những Pattern thuộc nhóm hành vi (Behavior Pattern). Nó cho phép định nghĩa tập hợp các thuật toán, đóng gói từng thuật toán lại, và dễ dàng thay đổi linh hoạt các thuật toán bên trong object. Strategy cho phép thuật toán biến đổi độc lập khi người dùng sử dụng chúng.

Ngay bên trên chúng ta đã thấy được lợi ích của Strategy Pattern rồi. Chúng ta có thể đóng gói các thuật toán khác nhau lại và thay đổi linh hoạt. Trong game bán súng có rất nhiều loại súng khác nhau. Mỗi loại khi bắn lại tạo ra hiệu ứng khác nhau nhưng chúng

đều chung là bắn. Khi dùng Strategy Pattern sẽ giúp thay đổi thuật toán bắn cho từng loại súng khác nhau và lựa chọn cái phù hợp cho từng loại súng.

## 3.2 Sơ đồ lớp




## 3.3 Code áp dụng

Code được áp dụng tại :

Assets/Scripts/StrategyPattern/IWeaponStrategy.cs

Assets/Scripts/StrategyPattern/WeaponStrategy.cs

Assets/Scripts/PrefabWeapon.cs

Assets/Scripts/RayCastWeapon.cs

Assets/Scripts/PlayerMovement.cs

Đầu tiên tạo interface và các class con có thuật toán Shoot() khác nhau .

```
1 reference
public interface IWeaponStrategy
{
    1 reference
    void Shoot();
}
```

```
6 references
public class WeaponStrategy : MonoBehaviour,IWeaponStrategy
{

    1 reference
    public virtual void Shoot(){

    }
}
```

```
public class RayCastWeapon : WeaponStrategy {

    6 references
    public Transform firePoint;
    1 reference
    public int damage = 40;
    1 reference
    public GameObject impactEffect;
    6 references
    public LineRenderer lineRenderer;
    0 references
    void Update () {···

    1 reference
    public override void Shoot(){
        StartCoroutine(IEnumeratorShoot());
    }

    1 reference
    IEnumerator IEnumeratorShoot ()···
}
```

```
public class PrefabWeapon : WeaponStrategy {

    2 references
    public Transform firePoint;
    1 reference
    public GameObject bulletPrefab;

    // Update is called once per frame
    0 references
    void Update () {...

    1 reference
    public override void Shoot ()
    {
        //Debug.Log("PrefabWeapon");
        Instantiate(bulletPrefab, firePoint.position, firePoint.rotation);
    }
}
```

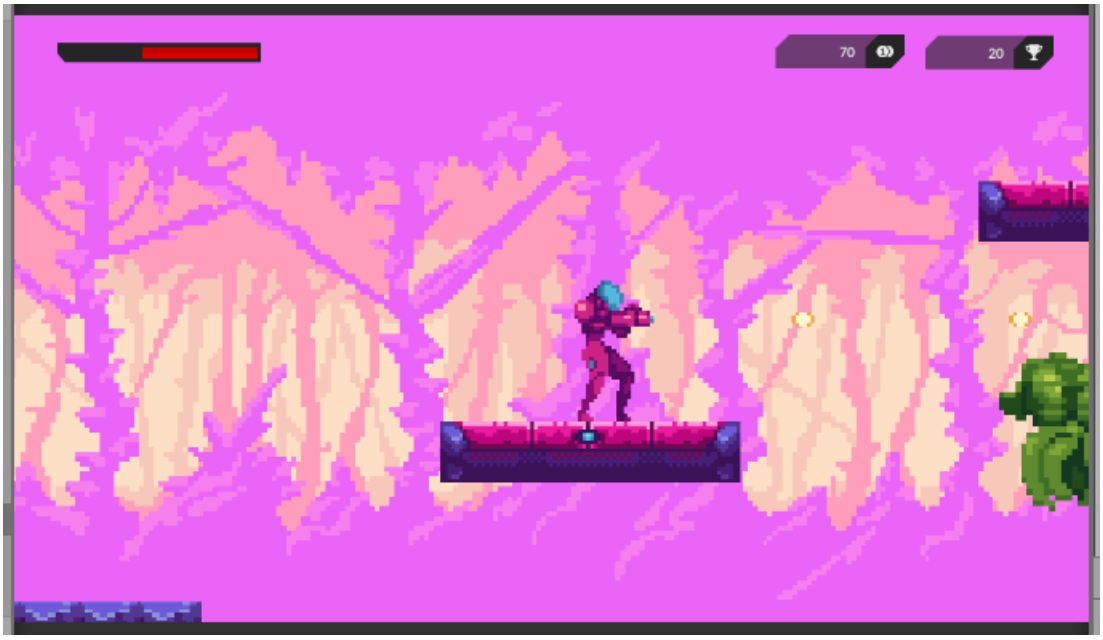
Tiếp theo là khởi tạo chúng ở PlayerMovement.cs

```
    // * PlayerMovement.cs Weapon
    9 references
    public WeaponStrategy weapon;
    10 references
    public List<WeaponStrategy> allWeapon;
```

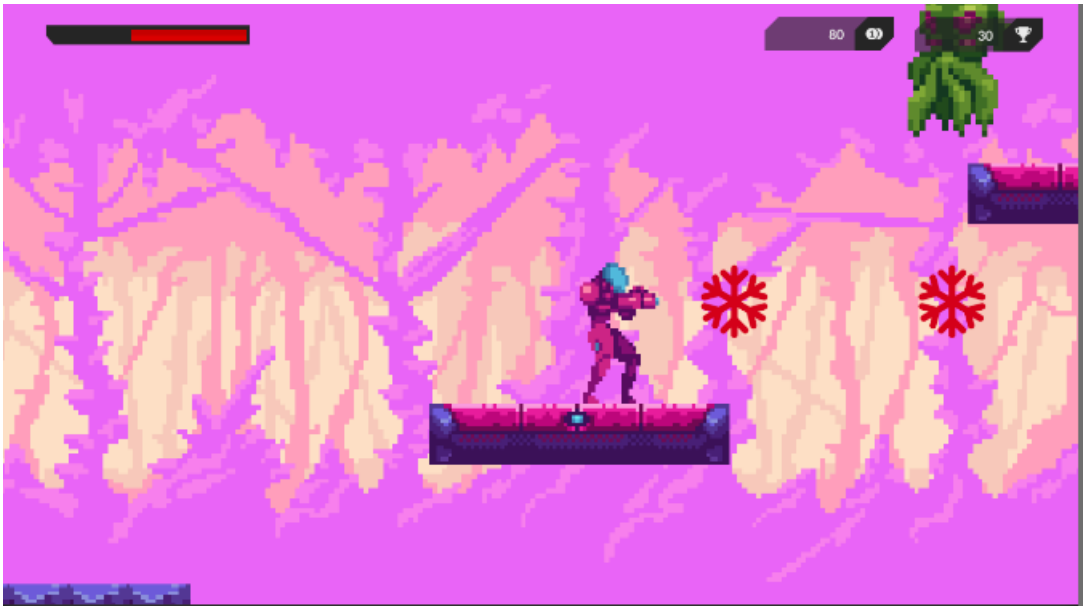
Và khi sử dụng :

```
    public override void Fire()
    {
        //Debug.Log("fire");
        this.weapon.Shoot();
    }
```

Và đây là trong game :




Đổi súng :



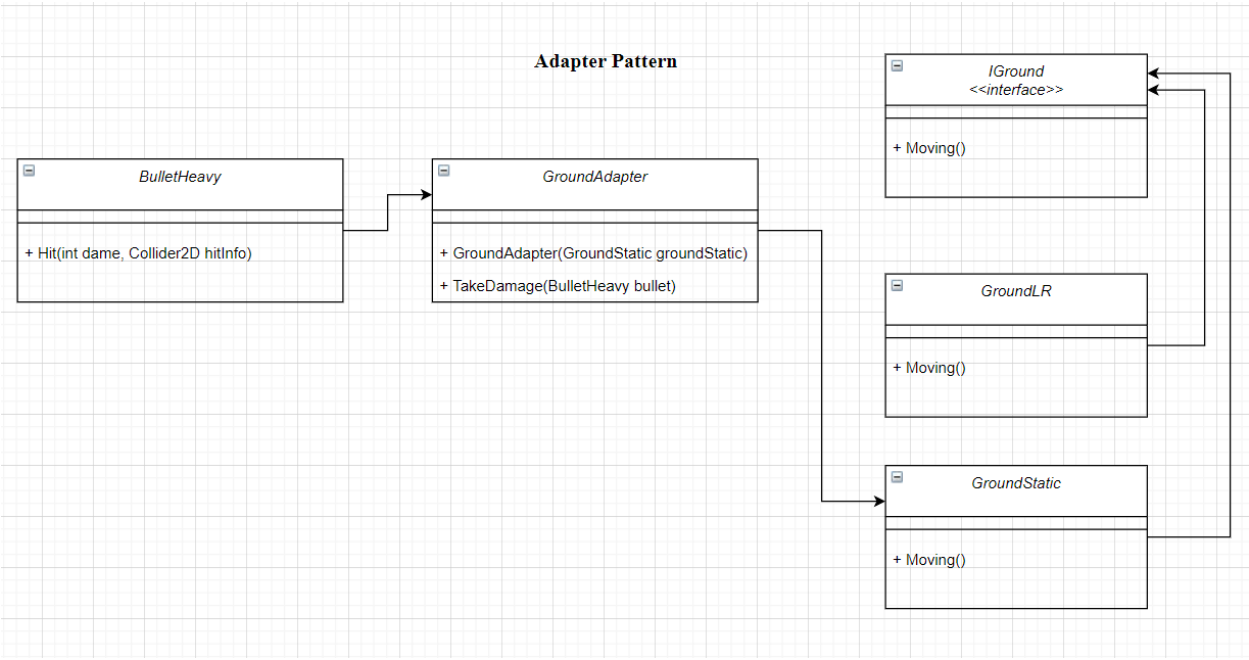
# CHƯƠNG 10 - ADAPTER PATTERN

## 3.1 Tại sao lại dùng Adapter Pattern

Khi tựa game đã hoàn thành và sau này khi nâng cấp và tạo ra thêm các loại súng mới cập nhật thêm cho người chơi. Và khi đó nhiều khi code cũ sẽ không còn khả dụng với code mới nữa.

Ở đây khi tạo ra một loại súng mới có tương tác là khi viên đạn va phải tường thì tường sẽ vỡ và biến mất. Lúc này chúng ta sẽ sử dụng đến adapter pattern để giải quyết vấn đề giao tiếp giữa hai vật thể là viên đạn và bức tường này.

## 3.2 Sơ đồ lớp




## 3.3 Code áp dụng

Code được áp dụng tại :

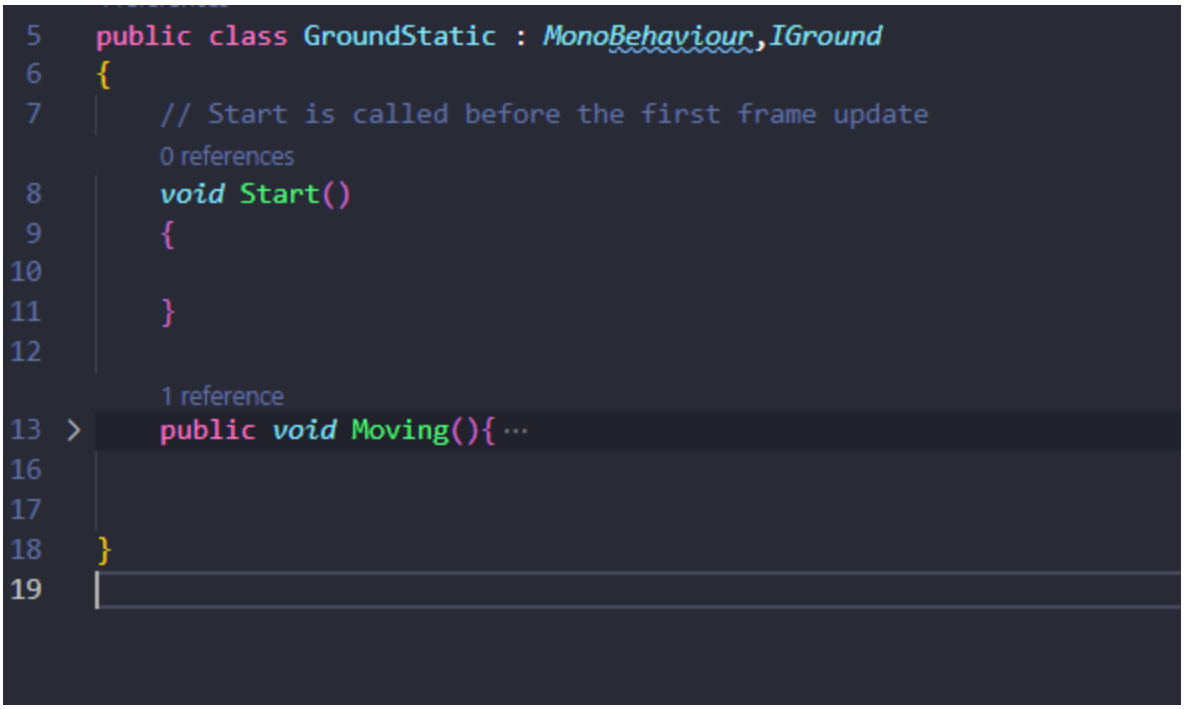
Assets/Scripts/AdapterPattern/IGround.cs

Assets/Scripts/AdapterPattern/GroundStatic.cs

Assets/Scripts/AdapterPattern/GroundLR.cs

Assets/Scripts/AdapterPattern/GroundAdapter.cs

Assets/Scripts/AdapterPattern/BulletHeavy.cs




Tại phần GroundStatic có những đoạn code cũ và sẽ không thay đổi gì trong này.

Thay vào đó sẽ sử dụng GroundAdapter để xử lý các vấn đề giữa viên đạn và bức tường/nền đất này.

```
2 references
public class GroundAdapter
{
    2 references
    public Grid grid;
    2 references
    public Tilemap tilemap;
    3 references
    public GroundStatic ground;

    1 reference
    public GroundAdapter(GroundStatic groundStatic)
    {
        this.ground = groundStatic;

        this.grid = this.ground.GetComponentInParent<Grid>();
        this.tilemap = this.ground.GetComponent<Tilemap>();
    }

    1 reference
    public void TakeDamage(BulletHeavy bullet)
    {

        Debug.Log("Detroy ground postion bullet hit");

        Vector3 posBulletHit = bullet.transform.position;
        Vector3Int position = grid.WorldToCell(posBulletHit);
        //tilemap.SetTile(position, null);

        for (int x = position.x - 1; x <= position.x + 1; x++)
        {
            for (int y = position.y - 1; y <= position.y + 1; y++)
            {
                tilemap.SetTile(new Vector3Int(x,y,position.z ), null);
            }
        }
    }
}
```

Viên đạn khi trúng đích sẽ gọi đến GroundAdapter

```
26      public void Hit(int dame, Collider2D hitInfo)
27      {
28          Enemy enemy = hitInfo.GetComponent<Enemy>();
29          if (enemy != null)
30          {
31              enemy.TakeDamage(dame);
32          }
33          else if (hitInfo.gameObject.layer == 20)
34          {// hit Ground
35
36              GroundStatic gs = hitInfo.gameObject.GetComponent<GroundStatic>();
37              if (gs != null)
38              {
39                  GroundAdapter ga = new GroundAdapter(gs);
40                  ga.TakeDamage(this);
41              }
42          }
43
44          Instantiate(impactEffect, transform.position, transform.rotation);
45
46          Destroy(gameObject);
47      }
48
49  }
```

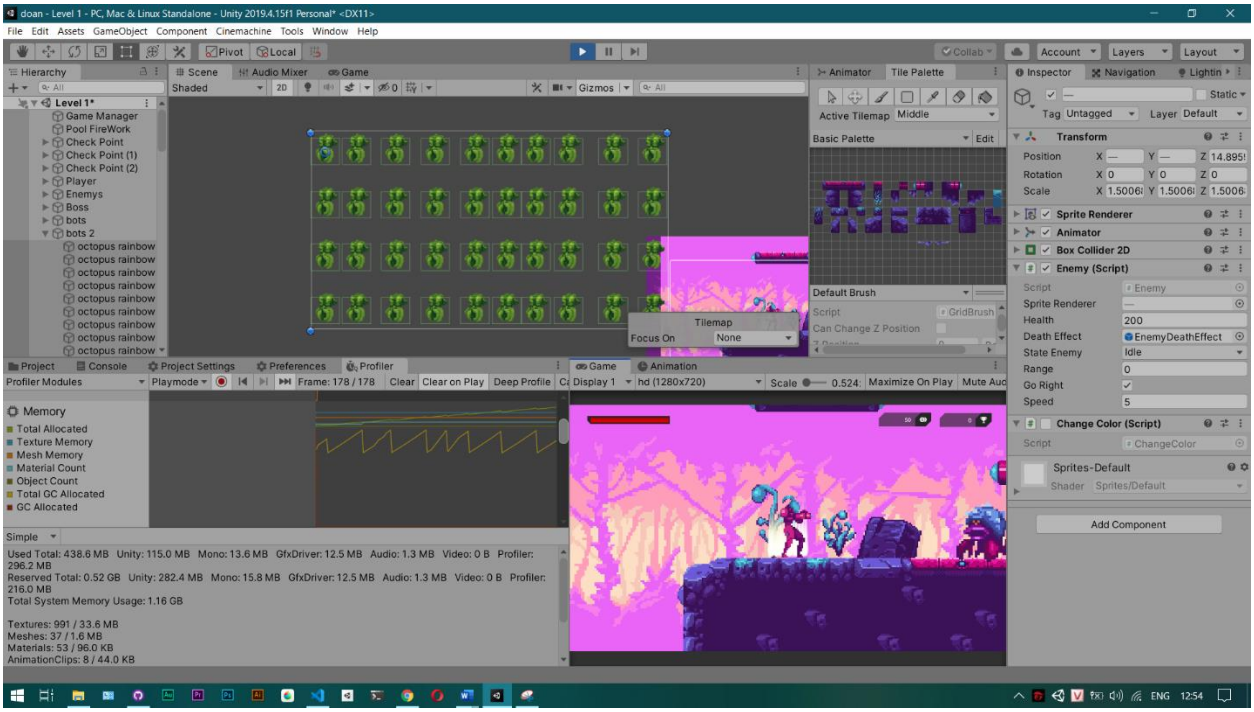
# CHƯƠNG 11 - FLYWEIGHT PATTERN

## 3.1 Tại sao lại dùng Flyweight Pattern

Đây là một pattern hay được sử dụng trong Unity3d nhưng lại ít được để ý đến . Tác dụng của nó trong ví dụ này là để tối ưu hiệu suất cho game. Trong game có các texture được sử dụng và có những cái sẽ bị tạo ra nhiều lần mặc dù nó cùng là một cái đã được tạo ra từ trước .

## 3.2 Code áp dụng

Trong trường hợp này chúng ta có 40 kẻ địch giống hết nhau. Và chúng ta muốn thay đổi màu sắc cho chúng .



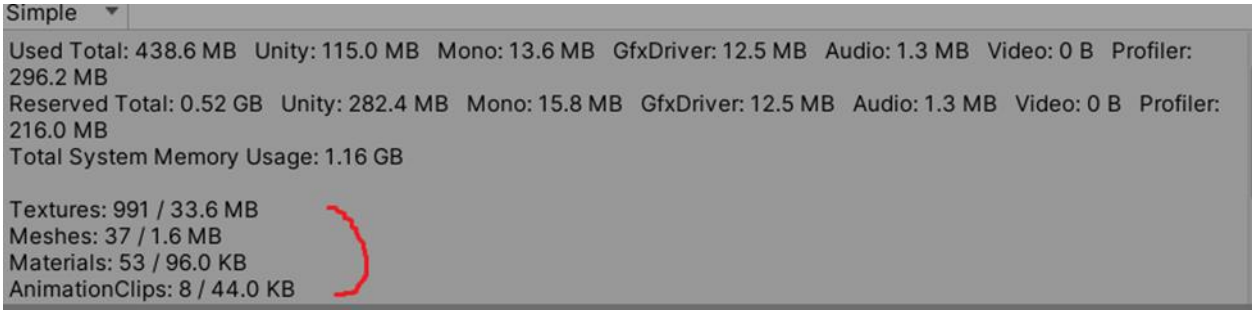

Hãy xem qua thông số nào :

Và đây là một hàm thay đổi màu sắc đơn giản :

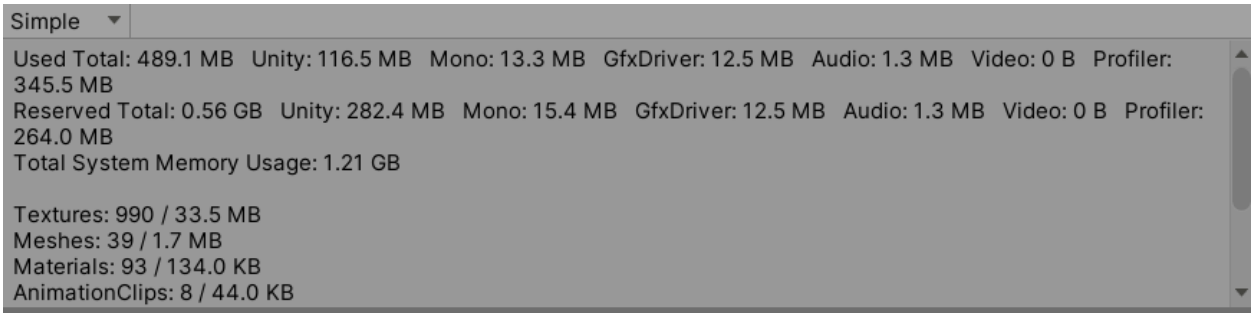
```
public class ChangeColor : MonoBehaviour
{
    2 references
    private Renderer renderer;
    0 references
    private MaterialPropertyBlock materialPropertyBlock;
    0 references
    void Awake()···
    // Start is called before the first frame update
    0 references
    void Start()···

    // Update is called once per frame
    0 references
    void Update()
    {
        ChangeColorObj();
    }

    1 reference
    public void ChangeColorObj(){
        this.renderer.material.color = GetRandomColor();
        // Debug.Log(this.renderer.material.color.ToString());
        //this.renderer.GetPropertyBlock(materialPropertyBlock);

    }

    1 reference
    private Color GetRandomColor()
    {
        return new Color(r: Random.Range(0f, 1f),
        g: Random.Range(0f, 1f),
        b: Random.Range(0f, 1f));
    }
}
```

Kết quả :


Thông số trước khi chạy random :




Thông số mới :



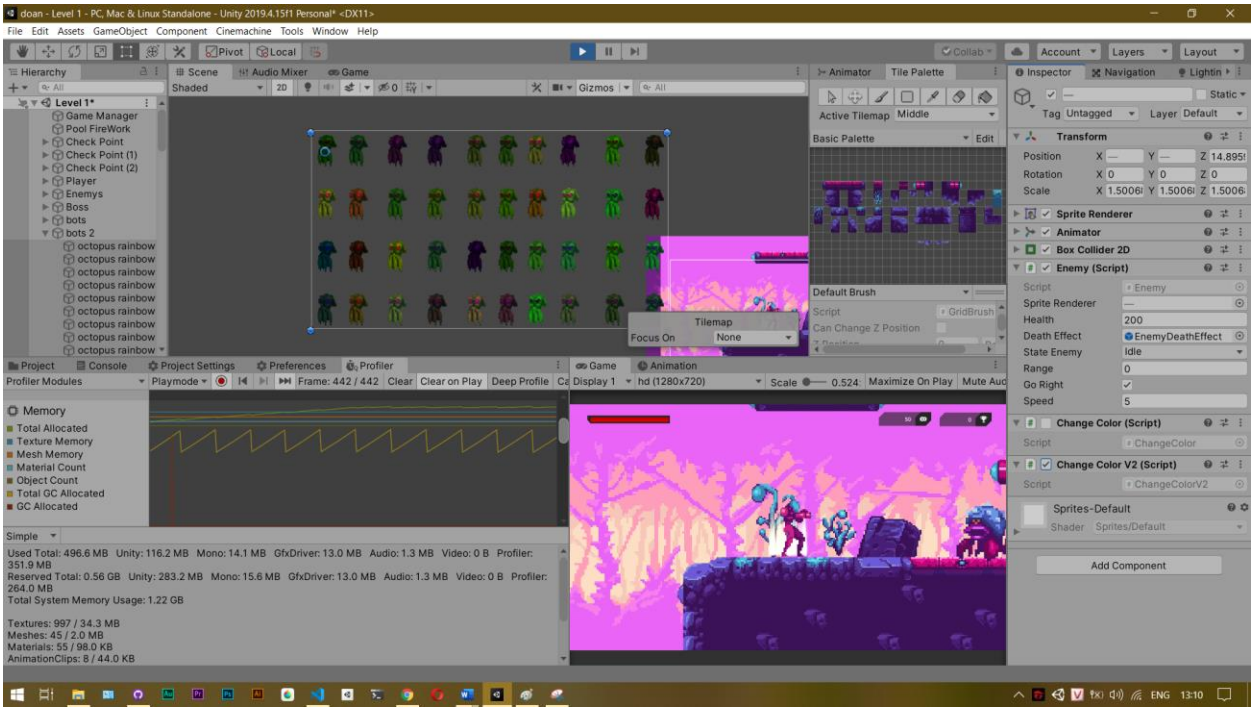

Materials từ 53/96KB tăng lên 93/134KB

Con số này nhìn có vẻ không đáng kể nhưng đây chỉ là một ví dụ nhỏ, khi dự án trở lên lớn hơn và phìn to gấp 100 hoặc 1000 lần thì đây sẽ là một rắc rối lớn.

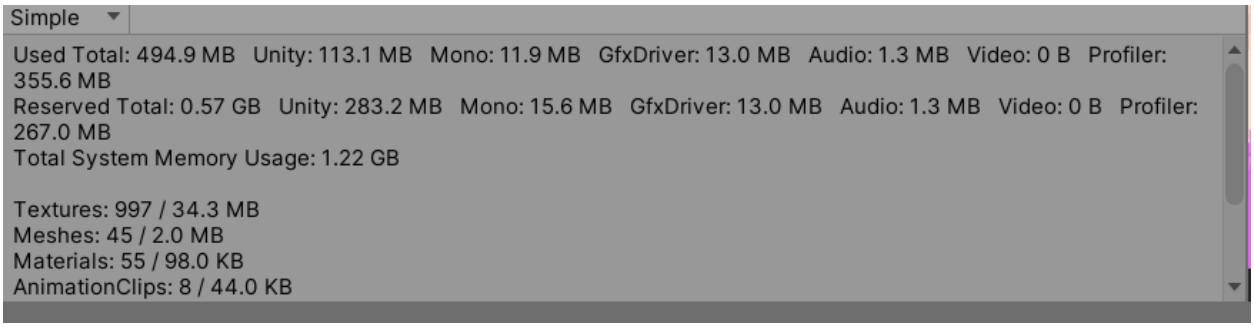
Vậy nên chúng ta sẽ sử dụng FlyWeight Pattern để giải quyết vấn đề này :

```
0 references
public class ChangeColorV2 : MonoBehaviour
{
    3 references
    private Renderer renderer;
    4 references
    private MaterialPropertyBlock materialPropertyBlock;
    0 references
    void Awake()···
    // Start is called before the first frame update
    0 references
    void Start()···

    // Update is called once per frame
    0 references
    void Update()
    {
        ChangeColorObj();
    }

    1 reference
    public void ChangeColorObj()
    {
        // * Lấy giá trị hiện tại của material property
        this.renderer.GetPropertyBlock(this.materialPropertyBlock);
        // * tạo ra giá trị mới
        this.materialPropertyBlock.SetColor("_Color",GetRandomColor());
        // * Lấy giá trị và cài vào renderer
        this.renderer.SetPropertyBlock(this.materialPropertyBlock);

    }

    1 reference
    private Color GetRandomColor()
    {
        return new Color(r: Random.Range(0f, 1f),
        g: Random.Range(0f, 1f),
        b: Random.Range(0f, 1f));
    }
}
```

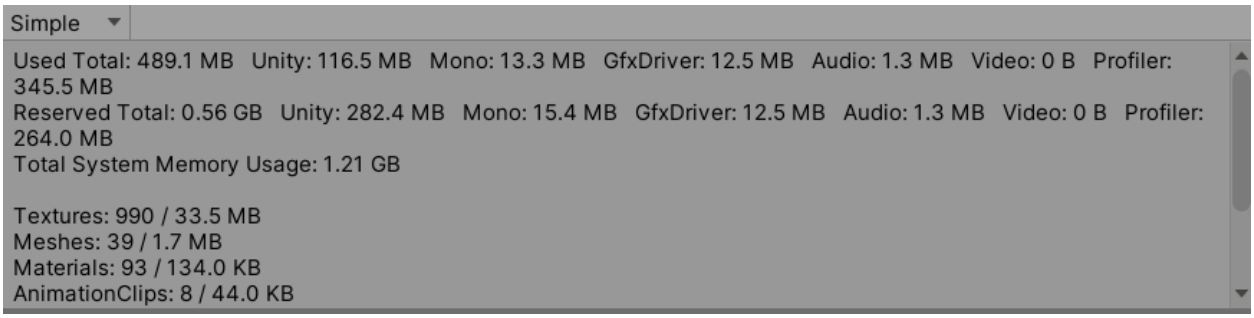
Kết quả chạy thử :



Kết quả y chang nhưng hãy xem đến các thông số sử dụng FlyWeight Pattern:




Và đây là cách cũ :



Từ bên trên chúng ta có thể dễ dàng thấy được ở cách sử dụng FlyWeight Pattern thông số materials gần  như không bị thay đổi. giảm được cực nhiều tài nguyên cần phải sử dụng ( 55/98KB < 93/134KB ).

# TÀI LIỆU THAM KHẢO


1. https://viblo.asia/p/tong-hop-cac-bai-huong-dan-ve-design-pattern-23-mau-co-ban-cua-gof-3P0lPQPG5ox
2. https://gpcoder.com/4190-huong-dan-java-design-pattern-singleton/
3. https://gpcoder.com/4747-huong-dan-java-design-pattern-observer/
4. https://viblo.asia/p/ap-dung-command-pattern-trong-unity-jvEla4o4Zkw
5. https://giniwebseo.vn/command-pattern-va-ung-dung-trong-unity-phan-1/
6. https://gpcoder.com/4626-huong-dan-java-design-pattern-flyweight/
7. https://hieuntp2.wordpress.com/van-de-tai-su-dung-doi-tuong-va-bo-nho-giam-thieu-chi-phi-cho-qua-trinh-tao-lap-va-huy-doi-tuong/
8. https://gpcoder.com/4456-huong-dan-java-design-pattern-object-pool/
9. https://gpcoder.com/4785-huong-dan-java-design-pattern-state/
10. https://www.youtube.com/watch?v=tdSmKaJvCoA&ab_channel=Brackeys
11. https://gpcoder.com/4352-huong-dan-java-design-pattern-factory-method/
12. http://gyanendushekhar.com/2017/02/01/memento-design-pattern-c/
13. https://gpcoder.com/4763-huong-dan-java-design-pattern-memento/
14. https://gpcoder.com/4796-huong-dan-java-design-pattern-strategy/
15. https://gpcoder.com/4483-huong-dan-java-design-pattern-adapter/
16. https://medium.com/@minhlee.fre.mta/l%C3%A0m-quen-v%E1%BB%9Bi-adapter-pattern-d42777c529c9