TECNOLÓGICO NACIONAL DE MÉXICO
INSTITUTO TECNOLÓGICO DE TIJUANA

SUBDIRECCIÓN ACADÉMICA
DEPARTAMENTO DE SISTEMAS Y COMPUTACIÓN

SEMESTRE:
Agosto - Diciembre 2025

CARRERA:
Ingeniería en Sistemas Computacionales

MATERIA:
Patrones de Diseño

TÍTULO ACTIVIDAD:
Examen unidad 4 y 5

Integrantes:

Antunez Partida Jorge Octavio - 21211910

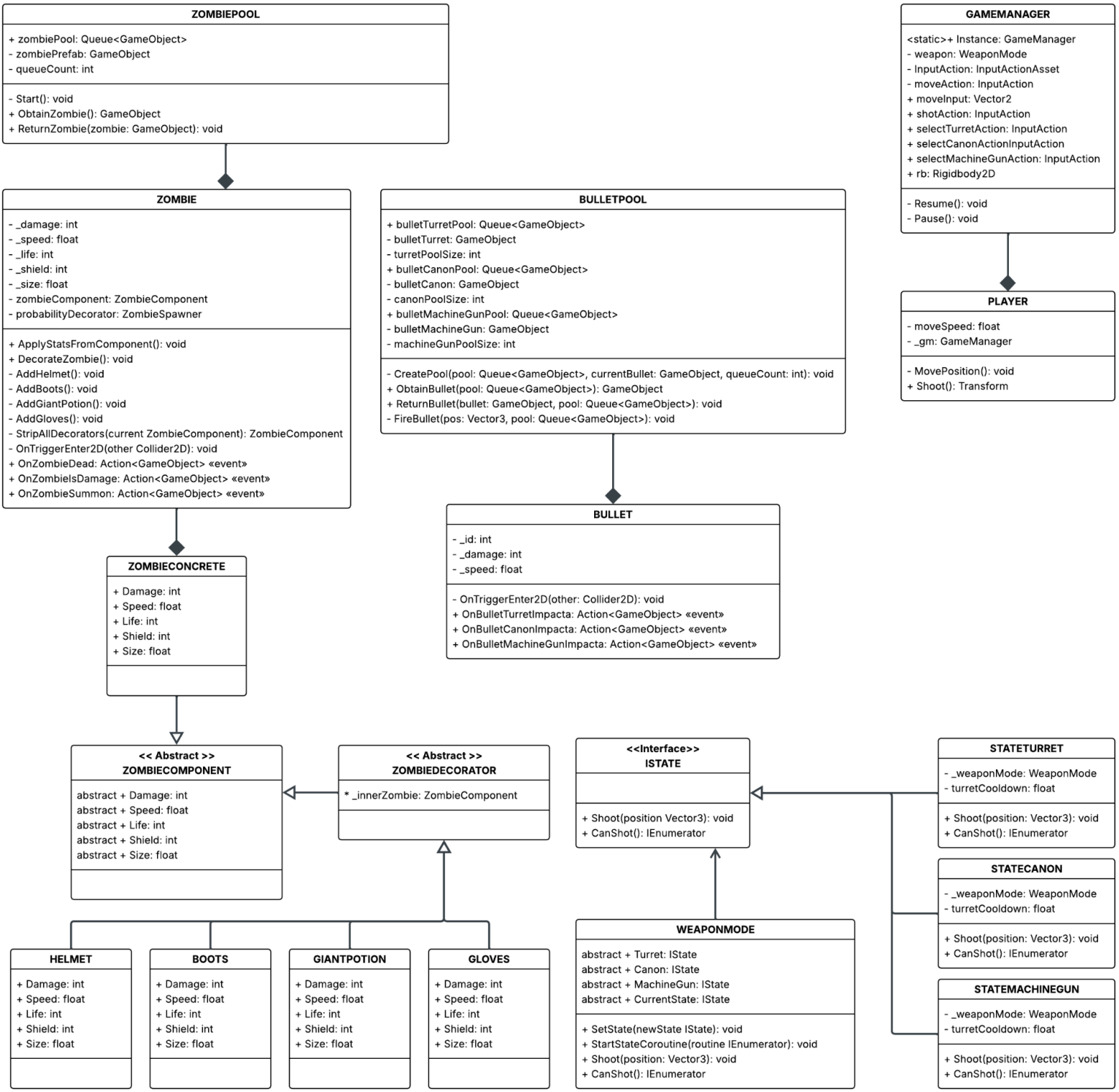
NOMBRE DEL MAESTRO(A):

Maribel Guerrero Luis

Fecha

11/12/2025

UML


ZOMBIEPOOL
+ zombiePool: Queue<GameObject>
- zombiePrefab: GameObject
- queueCount: int
- Start(): void
+ ObtainZombie(): GameObject
+ ReturnZombie(zombie: GameObject): void
GAMEMANAGER
<static>+ Instance: GameManager
- weapon: WeaponMode
- InputAction: InputActionAsset
- moveAction: InputAction
+ moveInput: Vector2
+ shotAction: InputAction
+ selectTurretAction: InputAction
+ selectCanonActionInputAction
+ selectMachineGunAction: InputAction
+ rb: Rigidbody2D
- Resume(): void
- Pause(): void
ZOMBIE
- _damage: int
- _speed: float
- _life: int
- _shield: int
- _size: float
- zombieComponent: ZombieComponent
- probabilityDecorator: ZombieSpawner
+ ApplyStatsFromComponent(): void
+ DecorateZombie(): void
- AddHelmet(): void
- AddBoots(): void
- AddGiantPotion(): void
- AddGloves(): void
- StripAllDecorators(current ZombieComponent): ZombieComponent
- OnTriggerEnter2D(other Collider2D): void
+ OnZombieDead: Action<GameObject> «event»
+ OnZombieIsDamage: Action<GameObject> «event»
+ OnZombieSummon: Action<GameObject> «event»
BULLETPOOL
+ bulletTurretPool: Queue<GameObject>
- bulletTurret: GameObject
- turretPoolSize: int
+ bulletCanonPool: Queue<GameObject>
- bulletCanon: GameObject
- canonPoolSize: int
+ bulletMachineGunPool: Queue<GameObject>
- bulletMachineGun: GameObject
- machineGunPoolSize: int
- CreatePool(pool: Queue<GameObject>, currentBullet: GameObject, queueCount: int): void
+ ObtainBullet(pool: Queue<GameObject>): GameObject
+ ReturnBullet(bullet: GameObject, pool: Queue<GameObject>): void
- FireBullet(pos: Vector3, pool: Queue<GameObject>): void
PLAYER
- moveSpeed: float
- _gm: GameManager
- MovePosition(): void
+ Shoot(): Transform
BULLET
- _id: int
- _damage: int
- _speed: float
- OnTriggerEnter2D(other: Collider2D): void
+ OnBulletTurretImpacta: Action<GameObject> «event»
+ OnBulletCanonImpacta: Action<GameObject> «event»
+ OnBulletMachineGunImpacta: Action<GameObject> «event»
ZOMBIECONCRETE
+ Damage: int
+ Speed: float
+ Life: int
+ Shield: int
+ Size: float
<< Abstract >>
ZOMBIECOMPONENT
abstract + Damage: int
abstract + Speed: float
abstract + Life: int
abstract + Shield: int
abstract + Size: float
<< Abstract >>
ZOMBIEDECORATOR
* _innerZombie: ZombieComponent
<<Interface>>
ISTATE
+ Shoot(position Vector3): void
+ CanShot(): IEnumerator
STATETURRET
- _weaponMode: WeaponMode
- turretCooldown: float
+ Shoot(position: Vector3): void
+ CanShot(): IEnumerator
STATECANON
- _weaponMode: WeaponMode
- turretCooldown: float
+ Shoot(position: Vector3): void
+ CanShot(): IEnumerator
STATEMACHINEGUN
- _weaponMode: WeaponMode
- turretCooldown: float
+ Shoot(position: Vector3): void
+ CanShot(): IEnumerator
WEAPONMODE
abstract + Turret: IState
abstract + Canon: IState
abstract + MachineGun: IState
abstract + CurrentState: IState
+ SetState(newState IState): void
+ StartStateCoroutine(routine IEnumerator): void
+ Shoot(position: Vector3): void
+ CanShot(): IEnumerator
HELMET
+ Damage: int
+ Speed: float
+ Life: int
+ Shield: int
+ Size: float
BOOTS
+ Damage: int
+ Speed: float
+ Life: int
+ Shield: int
+ Size: float
GIANTPOTION
+ Damage: int
+ Speed: float
+ Life: int
+ Shield: int
+ Size: float
GLOVES
+ Damage: int
+ Speed: float
+ Life: int
+ Shield: int
+ Size: float

Código

Clase GameManager que utiliza el patrón singleton

Utilice el patrón singleton para manejar únicamente los inputs del jugador, este singleton lo maneja como el componente “Controller” del patrón **MVC**.

Este solo se encarga de manejar la entrada de señales por parte del usuario, manejandolas como si fueran peticiones y este ya se comunica con el “Model” que en este caso son los datos y funciones del juego.

```
Script de Unity (1 referencia de recurso) | 5 referencias
public class GameManager : MonoBehaviour
{
    public static GameManager Instance;
    [SerializeField] private WeaponMode weapon;
    [SerializeField] public Transform shotPoint;
    public GameObject pauseMenuUI;
    private bool isPaused = false;

    Mensaje de Unity | 0 referencias
    void Awake()
    {
        InitializeInputSystem();

        if (Instance != null && Instance != this)
        {
            Destroy(gameObject);
            return;
        }

        Instance = this;
        DontDestroyOnLoad(gameObject);
    }

    #region Player
    #region Attributes
    [SerializeField] private InputActionAsset InputAction;
    private InputAction moveAction;
    public Vector2 moveInput;

    public InputAction shotAction;

    public InputAction selectTurretAction;
    public InputAction selectCanonAction;
    public InputAction selectMachineGunAction;

    [SerializeField] public Rigidbody2D rb;
    [SerializeField] public Animator anim;
    #endregion
```

```
1 referencia
void InitializeInputSystem()
{
    moveAction = InputAction.FindActionMap("Player").FindAction("Move");
    shotAction = InputAction.FindActionMap("Player").FindAction("Jump");

    selectTurretAction = InputAction.FindActionMap("Player").FindAction("Turret");
    selectCanonAction = InputAction.FindActionMap("Player").FindAction("Canon");
    selectMachineGunAction = InputAction.FindActionMap("Player").FindAction("MachineGun");
}

Mensaje de Unity | 0 referencias
void Update()
{
    moveInput = moveAction.ReadValue<Vector2>();

    if (shotAction.IsPressed())
    {
        weapon.CurrentState.Shoot(shotPoint.position);
    }

    if (selectTurretAction.WasPressedThisFrame())
    {
        weapon.SetState(weapon.Turret);
    }
    if (selectCanonAction.WasPressedThisFrame())
    {
        weapon.SetState(weapon.Canon);
    }
    if (selectMachineGunAction.WasPressedThisFrame())
    {
        weapon.SetState(weapon.MachineGun);
    }
    if (Keyboard.current.tabKey.wasPressedThisFrame)
    {
        if (isPaused)
        {
            Resume();
        }
        else
        {
            Pause();
        }
    }
}
#endregion
```

Clase BulletPool que maneja la lógica del patrón ObjectPool de las balas

El patrón de piscina de objetos se implementó para poder generar balas constantemente sin la necesidad de instanciar las miles de veces, gracias a este patrón solo generamos una cantidad suficiente de balas al comienzo del juego y estas mismas serán las que se están reutilizando constantemente sin generar basura en la memoria. Gracias a que solo estamos usando los mismos objetos todo el tiempo.

```
Script de Unity (2 referencias de recurso) | 2 referencias
public class BulletPool : MonoBehaviour
{
    #region Pools
    public Queue<GameObject> bulletTurretPool = new Queue<GameObject>();
    [SerializeField] private GameObject bulletTurret;
    [SerializeField] private int turretPoolSize;

    public Queue<GameObject> bulletCanonPool = new Queue<GameObject>();
    [SerializeField] private GameObject bulletCanon;
    [SerializeField] private int canonPoolSize;

    public Queue<GameObject> bulletMachineGunPool = new Queue<GameObject>();
    [SerializeField] private GameObject bulletMachineGun;
    [SerializeField] private int machineGunPoolSize;
    #endregion

    Mensaje de Unity | 0 referencias
    void Start()
    {
        CreatePool(bulletTurretPool, bulletTurret, turretPoolSize);
        CreatePool(bulletCanonPool, bulletCanon, canonPoolSize);
        CreatePool(bulletMachineGunPool, bulletMachineGun, machineGunPoolSize);
    }

    3 referencias
    void CreatePool(Queue<GameObject> pool, GameObject currentBullet, int queueCount)
    {
        for (int i = 0; i < queueCount; i++)
        {
            GameObject bullet = Instantiate(currentBullet);
            Bullet bulletScript = bullet.GetComponent<Bullet>();

            bullet.transform.parent = transform;
            // Temporary position off-screen

            if (bulletScript != null)
            {
                bulletScript.Id = i;
            }

            pool.Enqueue(bullet);
            bullet.SetActive(false);
        }
    }
```

```
1 referencia
public GameObject ObtainBullet(Queue<GameObject> pool)
{
    if (pool.Count > 0)
    {
        GameObject bullet = pool.Dequeue();

        bullet.SetActive(true);
        return bullet;
    }
    return null;
}

3 referencias
public void ReturnBullet(GameObject bullet, Queue<GameObject> pool)
{
    bullet.transform.position = transform.position; // Temporary position off-screen
    bullet.SetActive(false);
    pool.Enqueue(bullet);
}
3 referencias
private void FireBullet(Vector3 pos, Queue<GameObject> pool)
{
    if (pool.Count == 0)
    {
        Debug.LogWarning("No hay balas en la piscina!");
        SuccessfulShot(false);
        return;
    }

    GameObject bullet = ObtainBullet(pool);
    bullet.transform.position = pos;

    Rigidbody2D rb_;
    rb_ = bullet.GetComponent<Rigidbody2D>();
    rb_.linearVelocity = transform.right * bullet.GetComponent<Bullet>().Speed;
    SuccessfulShot(true);
}


#region Eventos Propios
public static event Action<bool> OnSuccessfulShot;

2 referencias
public static void SuccessfulShot(bool successfulShot)
{
    OnSuccessfulShot?.Invoke(successfulShot);
}
#endregion
```

Aquí podemos observar que el manejo de los objetos para regresar a la piscina es por medio de eventos, esto es parte de la **Arquitectura orientada a eventos**, este patrón resulta muy útil para trabajar con este tipo de proyectos, permite una comunicación directa entre los elementos que interactúan directamente entre si y ase mucho más fácil comunicar métodos dentro del programa.

```
#region Eventos Susbcritos
Mensaje de Unity | 0 referencias
private void OnEnable()
{
    StateTurret.OnShootTurret += FireBulletTurret;
    StateCanon.OnShootCanon += FireBulletCanon;
    StateMachineGun.OnShootMachineGun += FireBulletMachineGun;

    Bullet.OnBulletTurretImpacta += ReturnBulletTurret;
    Bullet.OnBulletCanonImpacta += ReturnBulletCanon;
    Bullet.OnBulletMachineGunImpacta += ReturnBulletMachineGun;
}

Mensaje de Unity | 0 referencias
private void OnDisable()
{
    StateTurret.OnShootTurret -= FireBulletTurret;
    StateCanon.OnShootCanon -= FireBulletCanon;
    StateMachineGun.OnShootMachineGun -= FireBulletMachineGun;

    Bullet.OnBulletTurretImpacta -= ReturnBulletTurret;
    Bullet.OnBulletCanonImpacta -= ReturnBulletCanon;
    Bullet.OnBulletMachineGunImpacta -= ReturnBulletMachineGun;
}

2 referencias
void FireBulletTurret(Vector3 pos)
{
    FireBullet(pos, bulletTurretPool);
}

2 referencias
void FireBulletCanon(Vector3 pos)
{
    FireBullet(pos, bulletCanonPool);
}

2 referencias
void FireBulletMachineGun(Vector3 pos)
{
    FireBullet(pos, bulletMachineGunPool);
}

2 referencias
void ReturnBulletTurret(GameObject bullet)
{
    ReturnBullet(bullet, bulletTurretPool);
}
2 referencias
void ReturnBulletCanon(GameObject bullet)
{
    ReturnBullet(bullet, bulletCanonPool);
}
2 referencias
void ReturnBulletMachineGun(GameObject bullet)
{
    ReturnBullet(bullet, bulletMachineGunPool);
}
#endregion
```

Clase ZombiePool que maneja la lógica del patrón ObjectPool de los zombies

```
Script de Unity (2 referencias de recurso) | 1 referencia
public class ZombiePool : MonoBehaviour
{
    public Queue<GameObject> zombiePool = new Queue<GameObject>();
    [SerializeField] private GameObject zombiePrefab;
    [SerializeField] private int queueCount;

    0 referencias
    public int CurrentSizePool
    {
        get { return zombiePool.Count; }
    }

    Mensaje de Unity | 0 referencias
    void Start()
    {
        for (int i = 0; i < queueCount; i++)
        {
            GameObject zombie = Instantiate(zombiePrefab);
            Zombie zombieScript = zombie.GetComponent<Zombie>();

            zombie.transform.parent = transform;

            if (zombieScript != null)
            {
                zombieScript.Id = i;
            }

            zombiePool.Enqueue(zombie);
            zombie.SetActive(false);
        }
    }
```

```
1 referencia
public GameObject ObtainZombie()
{
    if (zombiePool.Count > 0)
    {
        GameObject zombie = zombiePool.Dequeue();

        zombie.SetActive(true);
        return zombie;
    }
    return null;
}

2 referencias
public void ReturnZombie(GameObject zombie)
{
    zombie.transform.position = transform.position;
    zombie.SetActive(false);
    zombiePool.Enqueue(zombie);
}

#region Eventos Susbcritos
Mensaje de Unity | 0 referencias
private void OnEnable()
{
    Zombie.OnZombieDead += ReturnZombie;
}

Mensaje de Unity | 0 referencias
private void OnDisable()
{
    Zombie.OnZombieDead -= ReturnZombie;
}
#endregion
```

```
Script de Unity (1 referencia de recurso) | 13 referencias
public class Zombie : MonoBehaviour
{
    #region Attributes
    [SerializeField] public int _id;
    [SerializeField] private int _damage;
    [SerializeField] private float _speed;
    [SerializeField] private int _life;
    [SerializeField] private int _shield;
    [SerializeField] private float _size;

    private ZombieComponent zombieComponent;
    1 referencia
    public ZombieComponent ZombieComponent => zombieComponent;

    1 referencia
    public int Id
    {
        get => _id;
        set => _id = value;
    }

    0 referencias
    public int Life
    {
        get => _life;
        set => _life = value;
    }
    2 referencias
    public int Shield
    {
        get => _shield;
        set => _shield = value;
    }
    0 referencias
    public int Damage => _damage;
    0 referencias
    public float Speed => _speed;
    0 referencias
    public float Size => _size;
    #endregion

    #region Variables
    public Vector2 direction = Vector2.right;

    [SerializeField] private ZombieSpawner probabilityDecorator;
    #endregion

    #region Eventos Propios
    public static event Action<GameObject> OnZombieDead;
    public static event Action<GameObject> OnZombieIsDamage;
    public static event Action OnZombieSummon;
    #endregion
```

En esta parte del código del zombie añadimos decoradores de forma aleatoria al momento de activar el zombie, todos estos decoradores se eliminan del objeto concreto al momento de ser desactivado y devuelto a la pool.

```
5 referencias
void ApplyStatsFromComponent()
{
    _damage = zombieComponent.Damage;
    _speed = zombieComponent.Speed;
    _life = zombieComponent.Life;
    _shield = zombieComponent.Shield;
    _size = zombieComponent.Size;
    transform.localScale = Vector3.one * _size;
}

Mensaje de Unity | 0 referencias
void Awake()
{
    zombieComponent = new ZombieConcrete();
    probabilityDecorator = FindFirstObjectByType<ZombieSpawner>();
}

Mensaje de Unity | 0 referencias
private void OnEnable()
{
    zombieComponent = StripAllDecorators(zombieComponent);
    ApplyStatsFromComponent();
}

1 referencia
public void DecorateZombie()
{
    float addHelmetProbability = URandom.value;
    if (addHelmetProbability <= probabilityDecorator.helmet) AddHelmet();

    float addBootsProbability = URandom.value;
    if (addBootsProbability <= probabilityDecorator.boots) AddBoots();

    float addGiantPotionProbability = URandom.value;
    if (addGiantPotionProbability <= probabilityDecorator.giantPotion) AddGiantPotion();

    float addGlovesProbability = URandom.value;
    if (addGlovesProbability <= probabilityDecorator.gloves) AddGloves();
}
```

```
1 referencia
void AddHelmet()
{
    zombieComponent = new Helmet(zombieComponent);
    ApplyStatsFromComponent();
    OnZombieSummon?.Invoke();
}

1 referencia
void AddBoots()
{
    zombieComponent = new Boots(zombieComponent);
    ApplyStatsFromComponent();
    OnZombieSummon?.Invoke();
}

1 referencia
void AddGiantPotion()
{
    zombieComponent = new GiantPotion(zombieComponent);
    ApplyStatsFromComponent();
    OnZombieSummon?.Invoke();
}

1 referencia
void AddGloves()
{
    zombieComponent = new Gloves(zombieComponent);
    ApplyStatsFromComponent();
    OnZombieSummon?.Invoke();
}

1 referencia
ZombieComponent StripAllDecorators(ZombieComponent current)
{
    while (current is ZombieDecorator decorator)
    {
        current = decorator.Inner;
    }

    return current;
}
```

Decorador de los Zombies

Los decoradores se usan para añadir elementos a un objeto sin modificar la clase, además que es bastante escalable y modular lo que hace muy fácil trabajar con estos decoradores, y crear variantes de un mismo objeto sin modificar ninguna clase ya existente.

```
public abstract class ZombieComponent
{
    10 referencias
    public abstract int Damage { get; }
    10 referencias
    public abstract float Speed { get; }
    10 referencias
    public abstract int Life { get; }
    10 referencias
    public abstract int Shield { get; }
    10 referencias
    public abstract float Size { get; }
    5 referencias
    public abstract void AddFunction();
}
```

```
public class ZombieConcrete : ZombieComponent
{
    6 referencias
    public override int Damage => 3;
    6 referencias
    public override float Speed => 2.2f;
    6 referencias
    public override int Life => 20;
    6 referencias
    public override int Shield => 0;
    6 referencias
    public override float Size => 1;
    1 referencia
    public override void AddFunction() { }
}
```

```
public abstract class ZombieDecorator : ZombieComponent
{
    protected ZombieComponent _innerZombie;

    4 referencias
    public ZombieDecorator(ZombieComponent zombie)
    {
        _innerZombie = zombie;
    }

    public ZombieComponent Inner => _innerZombie;

}
```

Decoradores

```
public class Helmet : ZombieDecorator
{
    1 referencia
    public Helmet(ZombieComponent zombie) : base(zombie) { }
    6 referencias
    public override int Damage => _innerZombie.Damage;
    6 referencias
    public override float Speed => _innerZombie.Speed;
    6 referencias
    public override int Life => _innerZombie.Life;
    6 referencias
    public override int Shield => _innerZombie.Shield + 20;
    6 referencias
    public override float Size => _innerZombie.Size;
    1 referencia
    public override void AddFunction() { }
}
```

```
public class Boots : ZombieDecorator
{
    1 referencia
    public Boots(ZombieComponent zombie) : base(zombie) { }
    6 referencias
    public override int Damage => _innerZombie.Damage;
    6 referencias
    public override float Speed => _innerZombie.Speed + 1;
    6 referencias
    public override int Life => _innerZombie.Life;
    6 referencias
    public override int Shield => _innerZombie.Shield;
    6 referencias
    public override float Size => _innerZombie.Size;
    1 referencia
    public override void AddFunction() { }
}
```

```
public class GiantPotion : ZombieDecorator
{
    1 referencia
    public GiantPotion(ZombieComponent zombie) : base(zombie) { }
    6 referencias
    public override int Damage => _innerZombie.Damage;
    6 referencias
    public override float Speed => _innerZombie.Speed - 0.75f;
    6 referencias
    public override int Life => _innerZombie.Life + 20;
    6 referencias
    public override int Shield => _innerZombie.Shield;
    6 referencias
    public override float Size => _innerZombie.Size + 0.4f;
    1 referencia
    public override void AddFunction() { }
}
```

```
public class Gloves : ZombieDecorator
{
    1 referencia
    public Gloves(ZombieComponent zombie) : base(zombie) { }
    6 referencias
    public override int Damage => _innerZombie.Damage + 4;
    6 referencias
    public override float Speed => _innerZombie.Speed;
    6 referencias
    public override int Life => _innerZombie.Life;
    6 referencias
    public override int Shield => _innerZombie.Shield;
    6 referencias
    public override float Size => _innerZombie.Size;
    1 referencia
    public override void AddFunction() { }
}
```

Patrón de Estados para distintos modos del arma

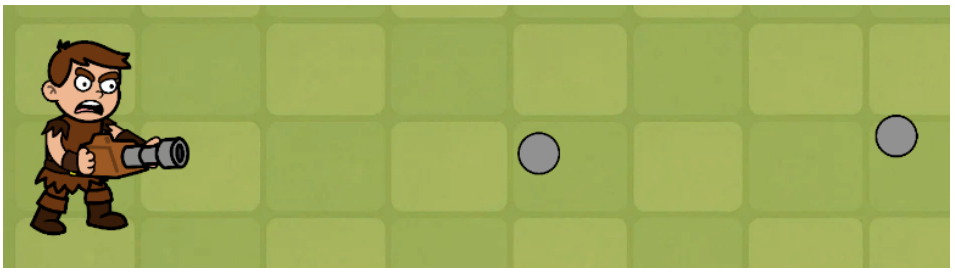
Con este patrón podemos tener distintos modos de funcionamiento para el arma, únicamente cambiando el estado de esta, lo que cambia la manera en la que funciona esta, para este juego hace que cada estado dispare su propio tipo de proyectil con distintas cadencias de disparo.

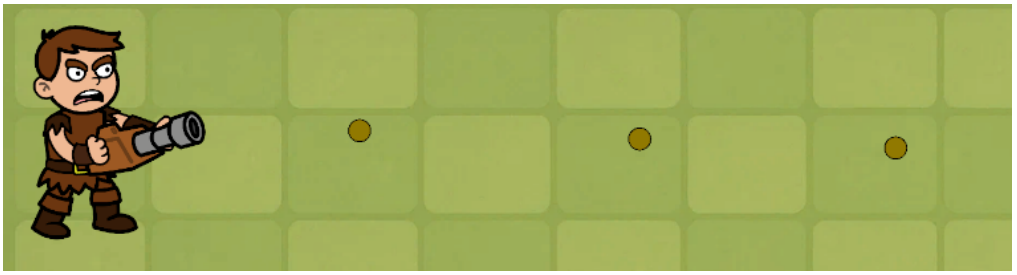
```
8 referencias
public interface IState
{
    5 referencias
    public void Shoot(Vector3 position);
    7 referencias
    IEnumerator CanShot();
}

Script de Unity (1 referencia de recurso) | 7 referencias
public class WeaponMode : MonoBehaviour
{
    3 referencias
    public IState Turret { get; set; }
    2 referencias
    public IState Canon { get; set; }
    2 referencias
    public IState MachineGun { get; set; }
    4 referencias
    public IState CurrentState { get; set; }

    Mensaje de Unity | 0 referencias
    public void Awake()
    {
        Turret = new StateTurret(this);
        Canon = new StateCanon(this);
        MachineGun = new StateMachineGun(this);
        SetState(Turret);
    }

    4 referencias
    public void SetState(IState newState)
    {
        CurrentState = newState;
    }

    3 referencias
    public void StartStateCoroutine(IEnumerator routine)
    {
        StartCoroutine(routine);
    }

    0 referencias
    public void Shoot(Vector3 position) => CurrentState.Shoot(position);
    0 referencias
    public IEnumerator CanShot() => CurrentState.CanShot();
}
```

```
public class StateTurret : IState
{
    private WeaponMode _weaponMode;
    1 referencia
    public StateTurret(WeaponMode weaponMode) => _weaponMode = weaponMode;

    [Header("Cooldowns")]
    [SerializeField] private float turretCooldown = 0.5f;
    private bool canShoot = true;

    public static event Action<Vector3> OnShootTurret;

    3 referencias
    public void Shoot(Vector3 position)
    {
        if (!canShoot) return;
        OnShootTurret?.Invoke(position);
        _weaponMode.StartStateCoroutine(CanShot());
    }

    3 referencias
    public IEnumerator CanShot()
    {
        canShoot = false;
        yield return new WaitForSeconds(turretCooldown);
        canShoot = true;
    }
}
```

```
public class StateCanon : IState
{
    private WeaponMode _weaponMode;
    1 referencia
    public StateCanon(WeaponMode weaponMode) => _weaponMode = weaponMode;

    [Header("Cooldowns")]
    [SerializeField] private float canonCooldown = 1.25f;
    private bool canShoot = true;

    public static event Action<Vector3> OnShootCanon;

    3 referencias
    public void Shoot(Vector3 position)
    {
        if (!canShoot) return;
        OnShootCanon?.Invoke(position);
        _weaponMode.StartStateCoroutine(CanShot());
    }

    3 referencias
    public IEnumerator CanShot()
    {
        canShoot = false;
        yield return new WaitForSeconds(canonCooldown);
        canShoot = true;
    }
}
```

```
4 referencias
public class StateMachineGun : IState
{
    private WeaponMode _weaponMode;
    1 referencia
    public StateMachineGun(WeaponMode weaponMode) => _weaponMode = weaponMode;

    [Header("Cooldowns")]
    [SerializeField] private float canonCooldown = 0.2f;
    private bool canShoot = true;

    public static event Action<Vector3> OnShootMachineGun;

    3 referencias
    public void Shoot(Vector3 position)
    {
        if (!canShoot) return;
        OnShootMachineGun?.Invoke(position);
        _weaponMode.StartStateCoroutine(CanShot());
    }

    3 referencias
    public IEnumerator CanShot()
    {
        canShoot = false;
        yield return new WaitForSeconds(canonCooldown);
        canShoot = true;
    }
}
```

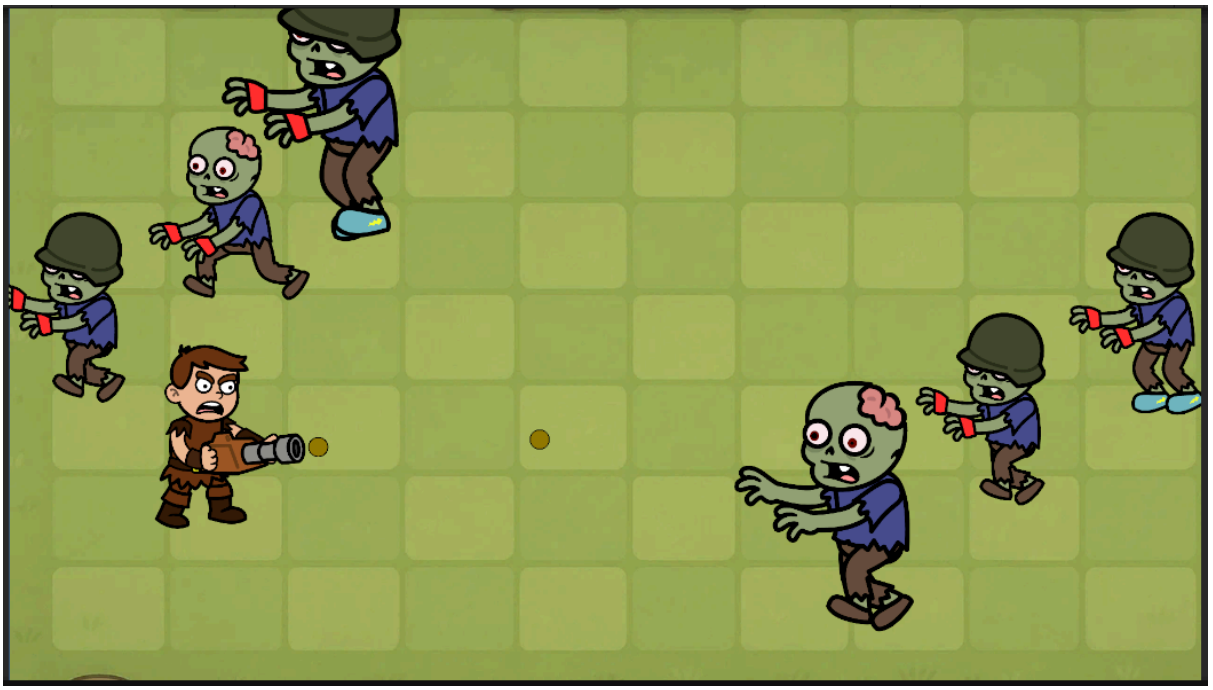




Patrones de Arquitectura: Orientada a eventos y MVC

Para este proyecto opté por combinar el patrón de diseño orientado a eventos para facilitar la comunicación entre los distintos componentes del juego de manera muy sencilla e hice uso del patrón MVC (Model, Controller y View) para la organización del proyecto.
Estos dos protones considero que funciona muy bien juntos y sobre todo para este tipo de proyectos ya que es muy útil y organizado el tener la entrada de inputs en un solo archivo y que sea este el que se encargue de comunicar la lógica (Model) con la vista (View) y gracias a los eventos son los mismo componentes los que manejan el flujo del juego.

Ejecución



Conclusión

La experiencia de desarrollo es innegablemente mejor y más eficiente al hacer uso de los patrones de diseño, sobre todo si sabemos cuándo aplicarlos y cómo aplicarlos correctamente.

Por mi experiencia personal puedo decir que investigaré más sobre los patrones y trataré de perfeccionar y aplicar todo lo aprendido a lo largo del curso.

Para terminar solo puedo recalcar que estos patrones ya están probados, son soluciones generales que siempre podemos adaptar para nuestros proyectos, por eso la importancia de realmente saber para qué sirven estos y cual es la lógica detrás de estos.

Lo que dio como resultado un desarrollo cómodo y eficiente a la hora de realizar esta actividad final.