

"Año de la recuperación y consolidación de la economía peruana"



Facultad de Ingeniería de Sistemas E Informática

"Guía Práctica N°13"

ASIGNATURA: DESARROLLO DE VIDEOJUEGOS

DOCENTE: FERNANDEZ RIVERA, DIEGO ALEJANDRO

NRC: 62092

ESTUDIANTE:

Apellidos y Nombres	CODIGO
BELITO RAMIREZ MORI OCTAVIO	74902137
CORONEL BURGOS JAVIER DANIEL	71997263
QUISPE UBALDO ALFREDO	71438344

HUANCAYO - 2025

TITULO:

Roles: Ingeniero de Pruebas e Ingeniero de Rendimiento

Proyecto: Práctica 13

Fecha: 13/11/2025

Objetivo de las pruebas

Se realiza **Tip de IA Co-piloto y Dev Kata: El Desafío de la Regresión**

Tip de IA Co-piloto**Pruebas GameLogic**

Prompt de la IA:

“Dada esta clase en C#, genera un conjunto de pruebas NUnit para ella en Unity. Incluye pruebas para casos extremos, como inicializar con cero o con un número negativo de objetivos.”

Código generado GameLogicEdgeCaseTests.cs

```
using NUnit.Framework;

public class GameLogicEdgeCaseTests
{
    // =====
    // PRUEBAS DE INICIALIZACIÓN
    // =====

    [Test]
    public void Constructor_SetsMinimumObjectives_WhenInitializedWithZero()
    {
        // ARRANGE & ACT
        var gameLogic = new GameLogic(0);

        // ASSERT
        Assert.AreEqual(1, gameLogic.ObjectivesToWin,
            "Si se inicializa con 0, el valor debe forzarse a 1.");
        Assert.AreEqual(0, gameLogic.ObjectivesCompleted);
        Assert.IsFalse(gameLogic.IsVictoryConditionMet);
    }

    [Test]
    public void Constructor_SetsMinimumObjectives_WhenInitializedWithNegativeValue()
    {
        // ARRANGE & ACT
        var gameLogic = new GameLogic(-10);

        // ASSERT
        Assert.AreEqual(1, gameLogic.ObjectivesToWin,
            "Los valores negativos deben convertirse en 1.");
        Assert.AreEqual(0, gameLogic.ObjectivesCompleted);
    }
}
```

```
        Assert.IsFalse(gameLogic.IsVictoryConditionMet);
    }

// =====
// PRUEBAS DE PROGRESO
// =====

[Test]
public void CompleteObjective_DoesNotExceed_ObjectivesToWin()
{
    // ARRANGE
    var gameLogic = new GameLogic(1);

    // ACT
    gameLogic.CompleteObjective(); // 1/1
    gameLogic.CompleteObjective(); // intento extra
    gameLogic.CompleteObjective(); // intento extra

    // ASSERT
    Assert.AreEqual(1, gameLogic.ObjectivesCompleted,
        "Nunca debe superar la cantidad de objetivos requeridos.");
    Assert.IsTrue(gameLogic.IsVictoryConditionMet);
}

[Test]
public void CompletingObjectives_ReachesVictory_WhenCountMatches()
{
    // ARRANGE
    var gameLogic = new GameLogic(3);

    // ACT
    gameLogic.CompleteObjective(); // 1
    gameLogic.CompleteObjective(); // 2
    gameLogic.CompleteObjective(); // 3

    // ASSERT
    Assert.IsTrue(gameLogic.IsVictoryConditionMet,
        "Debe activar victoria cuando los objetivos completados alcanzan el total.");
}

[Test]
public void CompleteObjective_DoesNothing_AfterVictoryConditionMet()
{
    // ARRANGE
    var gameLogic = new GameLogic(2);

    // ACT
    gameLogic.CompleteObjective(); // 1
    gameLogic.CompleteObjective(); // 2 → victoria
    gameLogic.CompleteObjective(); // intento extra

    // ASSERT
    Assert.AreEqual(2, gameLogic.ObjectivesCompleted,
        "No debe incrementar después de cumplirse la victoria.");
}

// =====
// PRUEBAS DE CONSISTENCIA
// =====

[Test]
public void IsVictoryConditionMet_IsFalse_WhenNotEnoughObjectives()
{
```

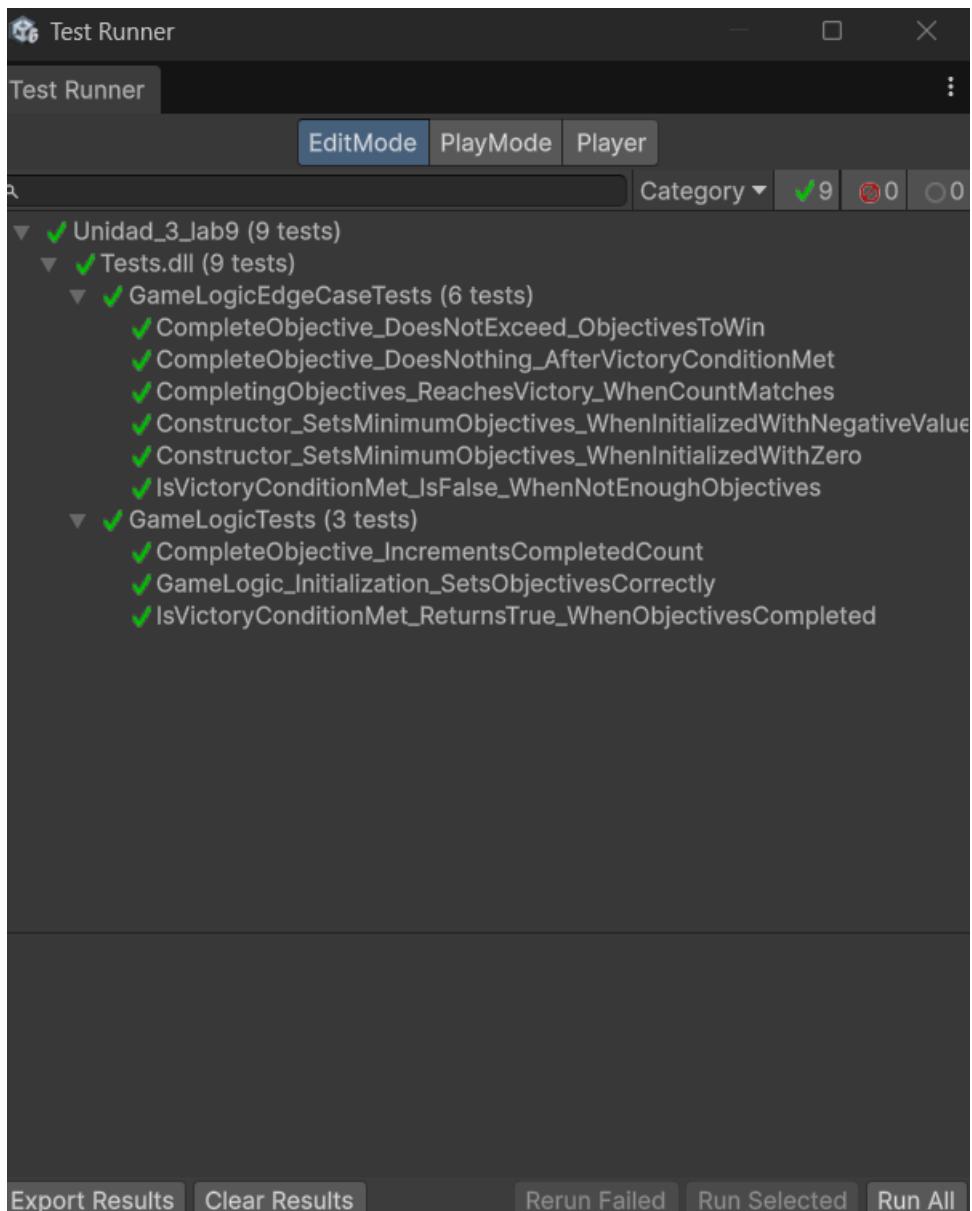
```
var gameLogic = new GameLogic(5);

gameLogic.CompleteObjective(); // 1

Assert.IsFalse(gameLogic.IsVictoryConditionMet);
}

}
```

Resultados:



Comparación:

La IA generó un conjunto adicional de pruebas no consideradas inicialmente por mí, especialmente relacionadas con casos límite: inicialización con cero, valores negativos, e intentos de completar objetivos después de haber alcanzado la victoria.

También reforzó el patrón 'Arrange – Act – Assert', lo cual me ayudó a mantener una estructura clara y profesional en mis pruebas.

No tomé sus resultados como reemplazo, sino como complemento para mejorar mi cobertura y verificar que la clase GameLogic funcionara correctamente bajo escenarios inesperados.

Pruebas código ineficiente

Prompt:

Analiza este código C# de Unity en busca de problemas de rendimiento y sugiere optimizaciones.

```
using UnityEngine;

public class ChaseState : AIState
{
    public ChaseState(AIController controller) : base(controller) { }

    public override void OnEnter()
    {
        Debug.Log("Entrando en estado de Persecución.");
        m_agent.speed = m_controller.chaseSpeed;
    }

    public override void UpdateState()
    {
        // 1. Condición de transición: ¿hemos perdido al jugador?
        if (Vector3.Distance(m_controller.transform.position, m_playerTransform.position) > m_controller.loseSightRadius)
        {
            m_controller.ChangeState(new PatrolState(m_controller));
            return;
        }

        // 2. Lógica del estado: perseguir al jugador.
        // m_agent.destination = m_playerTransform.position;

        //CÓDIGO INEFICIENTE
        m_agent.destination = GameObject.FindGameObjectWithTag("Player").transform.position;
    }

    public override void OnExit() {}
}
```

Resultado:

```
using UnityEngine;

public class ChaseState : AIState
{
    // 🔥 Nuevo: Cacheamos la referencia al jugador una sola vez
    private Transform _cachedPlayerTransform;

    public ChaseState(AIController controller) : base(controller) { }

    public override void OnEnter()
    {
        Debug.Log("Entrando en estado de Persecución.");
        m_agent.speed = m_controller.chaseSpeed;

        // Cacheo eficiente de la referencia del jugador
        // -----
        // Solo se ejecuta al entrar al estado, NO cada frame.
        var playerObj = GameObject.FindGameObjectWithTag("Player");
        if (playerObj != null)
            _cachedPlayerTransform = playerObj.transform;
    }

    public override void UpdateState()
    {
        // Si no encontramos al jugador, no podemos perseguir
        if (_cachedPlayerTransform == null)
            return;

        // 1. Condición de transición: ¿hemos perdido al jugador?
        if (Vector3.Distance(m_controller.transform.position, _cachedPlayerTransform.position)
            > m_controller.loseSightRadius)
        {
            m_controller.ChangeState(new PatrolState(m_controller));
            return;
        }

        // 2. Lógica del estado: perseguir al jugador usando referencia cacheada
        m_agent.destination = _cachedPlayerTransform.position;

        // -----
        // ✗ CÓDIGO INEFICIENTE — COMENTADO
        // Esto buscaba al jugador en TODA la escena cada frame.
        // m_agent.destination = GameObject.FindGameObjectWithTag("Player").transform.position;
        // -----
    }

    public override void OnExit() { }
}
```

Comparación:

La IA identificó correctamente el principal problema de rendimiento en mi código: llamar `GameObject.FindGameObjectWithTag` dentro de `UpdateState`, lo cual busca en toda la escena cada frame y es muy costoso. La optimización sugerida fue cachear la referencia al jugador en `OnEnter`, evitando búsquedas repetidas. Además, la IA explicó claramente

la diferencia de rendimiento: Find es la operación más lenta porque recorre toda la escena, GetComponent es más rápido porque solo busca dentro de un objeto, y acceder a una referencia cacheada es prácticamente instantáneo porque solo lee un valor en memoria. Esto me permitió entender no solo qué optimizar, sino por qué.

Dev Kata: El Desafío de la Regresión

Código de LootChestController.cs con bug:

```
using UnityEngine;

public class LootChestController : MonoBehaviour, IInteractable
{
    public bool isOpened = false;

    //👉 Esto es lo que los tests están buscando:
    public bool IsOpened => isOpened;

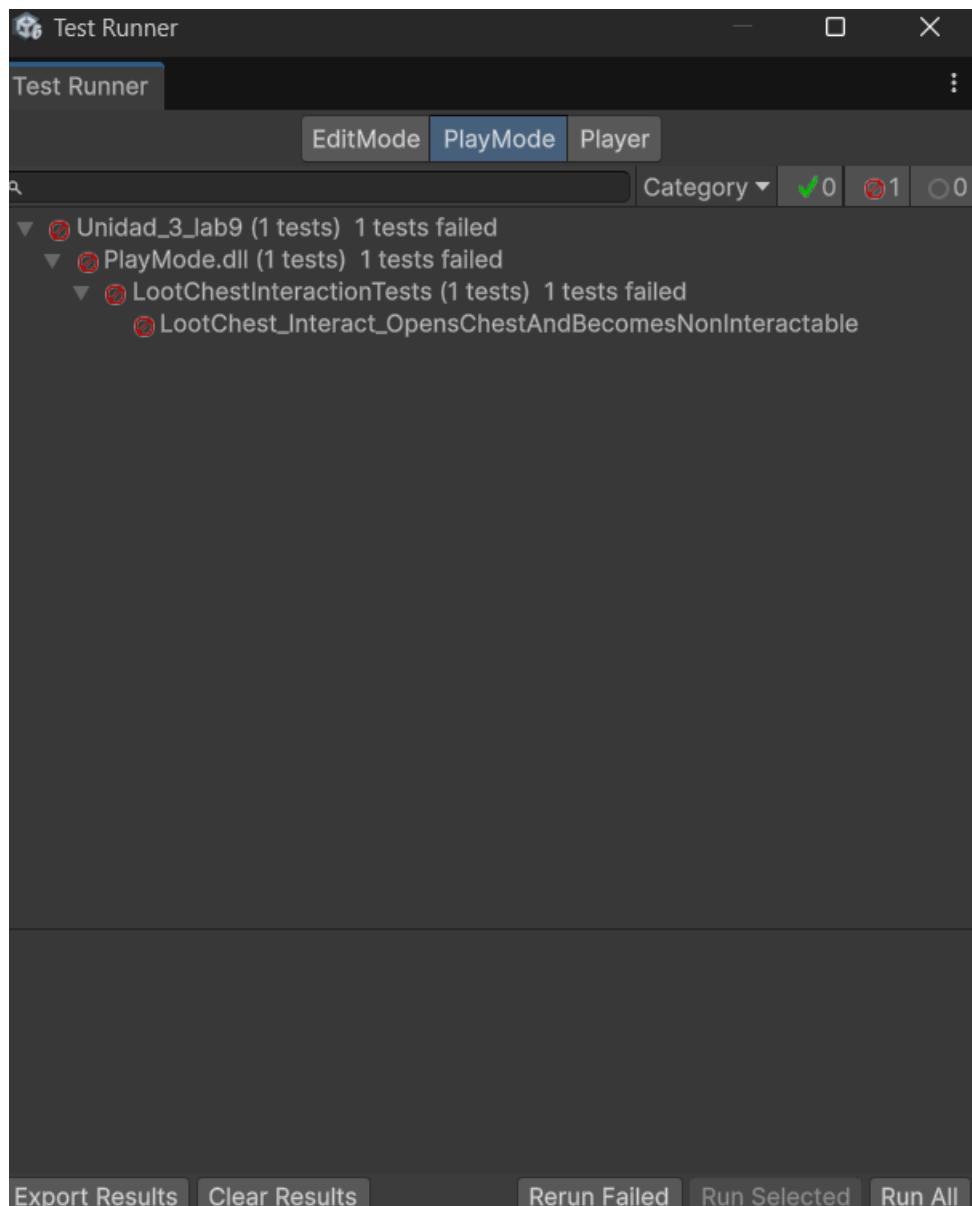
    public void Interact()
    {
        //☑ LÓGICA ORIGINAL (FUNCIONAL) — LA COMENTAMOS PARA INTRODUCIR EL BUG
        /*
        if (isOpened)
        {
            Debug.Log("Este cofre ya ha sido abierto.");
            return;
        }

        isOpened = true;
        Debug.Log("¡Has abierto el cofre y encontrado un tesoro!");
        */
    }

    //✖ BUG INTRODUCIDO (regresión)
    bool playerHasPermission = false;
    if (IsOpened || !playerHasPermission)
    {
        Debug.Log("Este cofre ya ha sido abierto o el jugador no tiene permiso.");
        return;
    }

    isOpened = true;
    Debug.Log("¡Has abierto el cofre y encontrado un tesoro (con bug)!");
}
```

Resultado:



Resultado:

Para evitar llamar funciones muy costosas innecesariamente, podríamos usar en una versión de prueba del script un booleano como `_costlyFunctionWasCalled`. La idea sería poner `costlyFunctionWasCalled = true;` dentro de `PlayComplexParticleEffect()` y, en una prueba automatizada, interactuar dos veces con el cofre y luego verificar que en la segunda interacción el estado siga abierto y que `_costlyFunctionWasCalled`

se mantenga en false. En un proyecto real, esto se haría con mocks o stubs (por ejemplo con NSubstitute), pero el concepto es el mismo: usar una bandera para asegurarnos de que la función cara no se ejecuta cuando no debería.