Documento de Diseño: Sentinel Editor Engine

Sistema de Testing de UI mediante MCP (Editor-Side Persistence)

1. Arquitectura de Inicialización (Editor Lifecycle)

En lugar de depender de una escena, utilizamos `[InitializeOnLoad]` y `EditorApplication.update` para asegurar que el servidor MCP y los servicios estén listos desde que abres el proyecto.

SentinelEditorProvider (El Punto de Entrada)

C#

```
using UnityEditor;
using UnityEngine;
using Sentinel.Editor.Services;

namespace Sentinel.Editor
{
    [InitializeOnLoad]
    public static class SentinelEditorProvider
    {
        private static readonly UIInspectorService _inspector = new UIInspectorService();
        private static readonly UIInteractionService _interaction = new UIInteractionService();

        static SentinelEditorProvider()
        {
            // Se ejecuta automáticamente al cargar el Editor
            InitiazeSentinel();
        }

        private static void InitiazeSentinel()
        {
            Debug.Log("[Sentinel] Editor Service iniciado y listo para recibir llamadas MCP.");
        }

        public static UIInspectorService Inspector { get { return _inspector; } }
        public static UIInteractionService Interaction { get { return _interaction; } }
    }
}
```

---

2. Herramientas MCP Especializadas (Estructura de Comandos)

Para que el modelo de lenguaje (OpenAI) pueda "testear de A a B", necesita herramientas con responsabilidades únicas. Aquí definimos las 4 herramientas clave:

A. Herramienta: `query_ui_structure`

Propósito: Mapear los elementos actuales.

- **Input:** `rootName` (opcional).
- **Output:** JSON con `name`, `type`, `worldBound` y `pickingMode`.

B. Herramienta: `get_element_detail`

Propósito: Inspección profunda de un elemento sospechoso de fallo.

- **Input:** `elementId`.
- **Output:** Clases USS aplicadas, estilos calculados (visibilidad, opacidad) y si tiene callbacks registrados.

C. Herramienta: `perform_ui_gesture`

Propósito: Ejecutar interacciones físicas.

- **Input:** `elementId`, `gestureType` (Click, DoubleClick, LongPress, DragTo).

D. Herramienta: `wait_for_ui_state`

Propósito: Testing asíncrono.

- **Input:** `elementId`, `condition` (IsVisible, IsEnabled, TextEquals), `timeout`.

---

3. Implementación de Servicios Especializados

UI Inspector (Especializado en Percepción)

C#
```
using UnityEngine;
using UnityEngine.UIElements;
using UnityEditor;
using System.Collections.Generic;

namespace Sentinel.Editor.Services
```

```
{
    public sealed class UIInspectorService
    {
        public List<VisualElement> GetAllInteractableElements(VisualElement root)
        {
            List<VisualElement> interactables = new List<VisualElement>();
            // Buscamos solo elementos que bloqueen el paso del ratón o sean botones/inputs
            root.Query<VisualElement>().ForEach((VisualElement element) =>
            {
                if (element.pickingMode == PickingMode.Position)
                {
                    interactables.Add(element);
                }
            });
            return interactables;
        }
    }
}
```

UI Interaction (Especializado en Acción vía Framework)

Este servicio utiliza la técnica del vídeo de la Unite: simular eventos a través del sistema de dispatch de Unity Editor.

C#

```
using UnityEditor;
using UnityEngine.UIElements;
using Unity.UI.TestFramework;
using System.Threading.Tasks;

namespace Sentinel.Editor.Services
{
    public sealed class UIInteractionService : EditorWindow
    {
        public async Task<string> ExecuteClick(string elementName)
        {
            // Localizamos la ventana del Editor que contiene la UI a testear
            VisualElement root = GetActiveEditorUI();
            VisualElement target = root.Q<VisualElement>(elementName);

            if (target == null)
            {
                return $"Error: No se encuentra '{elementName}'";
            }

            // Usamos el Framework para un click real de bajo nivel
            using (Pointer pointer = new Pointer(PointerId.mouseDevice))
```

```
        {
            pointer.Click(target);
        }

        await Task.Delay(100); // Pequeña espera para el reflow de la UI
        return "Click ejecutado con éxito.";
    }

    private VisualElement GetActiveEditorUI()
    {
        // Lógica para encontrar la ventana de juego o el panel de UI Toolkit activo
        return null; // Implementación específica según la ventana objetivo
    }
  }
}
```

---

4. El "Bridge" de Comunicación (MCP Wrapper)

Este es el código que el Chatbot llamará cuando OpenAI decida usar una herramienta. No hay `MonoBehaviour`, es pura lógica de C# en Editor.

C#

```
namespace Sentinel.Editor.Bridge
{
  public static class MCPToolDispatcher
  {
    public static async Task<string> DispatchCommand(string toolName, string
jsonArguments)
    {
        switch (toolName)
        {
            case "perform_ui_gesture":
                // Parsear argumentos y llamar a SentinelEditorProvider.Interaction
                return await SentinelEditorProvider.Interaction.ExecuteClick("btn_login");

            case "query_ui_structure":
                // Retornar la jerarquía procesada por SentinelEditorProvider.Inspector
                return "{}";

            default:
                return "Herramienta no reconocida.";
        }
    }
  }
}
```

---

5. Reflexión sobre el Flujo de A a B

Con esta estructura de **Herramientas Especializadas**, el modelo de lenguaje puede realizar razonamientos de este tipo:

1. **Llamada 1 (`query_ui_structure`):** "Veo un `Button` llamado `start_quest_btn` pero está fuera de los límites de la pantalla."
2. **Llamada 2 (`get_element_detail`):** "Confirmado, el elemento tiene un estilo `display: none` aplicado por la clase USS `.hidden`."
3. **Llamada 3 (Acción previa):** "Debo primero pulsar el botón `open_journal_btn` para que el otro aparezca."
4. **Llamada 4 (`perform_ui_gesture`):** "Hago click en `open_journal_btn`."
5. **Llamada 5 (`wait_for_ui_state`):** "Espero a que `start_quest_btn` sea visible."

Ventajas de este enfoque

- **Persistencia:** Al estar en el espacio de `UnityEditor`, la IA puede resetear el Play Mode, recargar escenas o modificar el UXML, y el sistema de testeo seguirá "vivo".
- **Granularidad:** Al tener muchas herramientas pequeñas, OpenAI consume menos tokens por respuesta y comete menos errores de sintaxis en los argumentos.
- **Sin Basura en Escena:** Tu proyecto de juego permanece limpio; no hay GameObjects de "testing" ensuciando las builds de producción.