We are DOOMED: Guía de trabajo en grupo

Luis

23 de abril de 2025

Resumen

Como bien conocemos de experiencia, los trabajos en grupo son grandes oportunidades de intercambio y también grandes oportunidades para conocer lo que es el caos. En este documento expongo los problemas con los que me he ido topando mientras trabajaba en grupo y propongo una serie de soluciones. Estas soluciones no están escritas en piedra, si tenéis una propuesta mejor, pues como dice su nombre, proponedla (porfa).

1. Introducción

Esta guía tiene como objetivo hacer que nuestro trabajo en grupo sea lo más cómodo y estructurado posible, aunque de primeras pueda pareceros lo contrario. Todo lo que propongo aquí es una **inversión**, si todos seguimos una guía común, **nos ahorraremos futuras calentadas de cabeza** para comprender el trabajo de nuestros compañeros, que es lo que ocurriría si cada uno trabaja como le parece. Trabajar como a uno le parece, de primeras, parece cómodo, pero luego te das cuenta de que cada uno de tus compañeros sigue su propia estructura de sistema, llama a las cosas de una forma, guarda sus archivos en tal sitio, sus variables son en snake_case mientras que las tuyas son en camelCase y muchas cosas más, un pifostio.

2. Mentalidad de trabajo

Antes de ponernos técnicos, una buena alineación mental del equipo es clave para el desarrollo conjunto, puesto que beneficia a la coherencia y el entendimiento del trabajo de nuestros compañeros. Esto lo traduzco como "Si todos los demás piensan como yo, como yo sé como yo haría algo, entonces sé como ese algo que han hecho los demás está hecho". Obviamente nadie va a pensar exactamente igual que tú, pero cuanto más parecido pensemos mejor.

2.1. Mentalidad profesional

Como en todos los trabajos, intento que todo lo aprendido sea lo más útil posible en mi futura carrera profesional. Qué mejor modo de hacerlo, que hacerlo todo como alguien que se dedique a ello.

Cuando trabajemos, sea cual sea la parte que hayamos elegido, ya existen formas estándares de llevarla a cabo. Todo esto, claro, son modelos conceptuales y estructurales de como diseñar nuestros módulos, si no, nuestro trabajo ya estaría hecho y solo habría que copiar y pegar. A lo que quiero llegar con todo esto, es que **antes de diseñar o implementar algo, busquemos si existe una forma común de hacerlo.** Yo soy de los que les gusta hacer las cosas por sí mismo e inventarse sus propios métodos de resolver problemas, pero creedme que aunque sigáis todo esto, aún tenéis mucho margen para pensar y crear cosas por vosotros mismos, no os preocupéis. Bueno, y si se da el caso de que vuestra propia idea es la mejor candidata, pues adelante.

Uno de los grandes beneficios de que todos llevemos a cabo cada tarea de la forma común, es que nuestros módulos encajarán más fácilmente con el de nuestros compañeros. ¿Por qué ocurre esto? Esto ocurre porque cuando implementamos un módulo de esta manera, la forma de interactuar con los demás módulos ya ha sido pensada previamente. Esto sucede porque todos esos estándares son producto de la experiencia de gente que lleva años desarrollando juegos y estructurando proyectos, aprovechémosla entonces.

2.2. Top-down y Bottom-Up

Para el **diseño** de nuestro juego propongo el uso del análisis **Top-down**, en este tipo de análisis se parte desde una visión del sistema como un todo y luego se van concretando cada una de sus partes hasta llegar a elementos atómicos. Veámoslo así, si miramos la figura 1, es como si comenzásemos desde arriba, lo más genérico, y fuésemos descomponiendo hacia abajo, lo más concreto. Esta figura es solo un ejemplo, puede que la descomposición sea diferente.

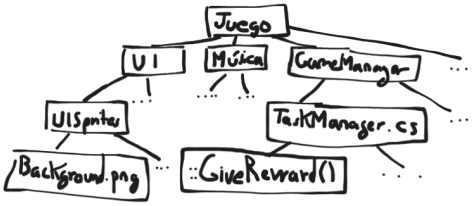

Figura 1: Descomposición del sistema

Para poder comprender el sistema en su totalidad queda un paso intermedio que es el establecimiento de relaciones entre cada uno de los módulos.

Mientras tanto, para la **implementación, verificación, y testing** propongo el análisis **Bottom-Up**. Que tomando la figura 1 sería comenzando de lo más concreto, desde abajo, hacia arriba, lo más genérico. Con esta metodología vamos asegurándonos de que a medida que vamos uniendo elementos para formar nuestro sistema, todos sus componentes hayan sido previamente verificados de forma aislada. Es decir, verificando una nueva agrupación, si nos surgen fallos, los fallos **son a causa de las relaciones entre los módulos** y **no de fallos internos de módulos individuales** (porque de eso ya nos hemos asegurado previamente).
Ej.

1. Primero implementamos, verificamos y testeamos las funciones/métodos/etc (elementos atómicos, indivisibles).
2. Luego verificamos y testeamos el módulo (clase).
3. Luego verificamos y testeamos las relaciones con otros módulos, para fomar un sistema.
4. Y luego sería cosa de ir agrupando los sistemas, que a su vez hacen sistemas más grandes y estos otros más grandes, etc; verificandolos y testeándolos, hasta tener el todo, el juego.

2.2.1. Resumen

Top-Down, general a concreto, lo usamos para **diseño**.
Bottom-Up, concreto a general, lo usamos para **implementación, verificación y testing**.

Cuando digo verificación y testing, no tiene que ser un test explícito (ej. unit test), puede bastar con asegurarnos de forma manual que eso que hemos hecho opera como queremos. Vamos; ejectuar y ver que eso funciona.

2.3. Programar y Documentar, amigos inseparables

Señores, documentemos, principalmente por el bien del equipo y también por nosotros mismos, nunca sabremos si seremos capaces de acordarnos de lo que hicimos hace 1 mes. Documentar también se aplica a más tareas aparte de programar, por ejemplo: también hay que documentar la estructuración del proyecto, esto entre muchas más cosas. Centrándonos en la documentación de programación, cuando digo documentación, me refiero tanto a documentación interna (en el código) como externa (word, LaTeX, etc).

2.4. Evitar las "ansias de optimización"

No intentemos optimizar antes de tiempo, primero centrémonos en tener un código legible y modular. Cuando tengamos problemas de rendimiento y haya que optimizar, haremos uso del perfilador de Unity, para identificar qué funciones son las que toman más tiempo de CPU y ya identificadas esas partes críticas, las optimizaremos.

Tratar de optimizar todas las funciones que vamos escribiendo puede ser contraproducente en cuanto a nuestro tiempo y a legibilidad del código. Muchas veces la causa del bajo rendimiento son solo un par de funciones concretas que se ejecutan de forma recurrente, ahí es donde debemos de actuar.

3. Nuestro entorno común

Cada uno usa su propio sistema operativo, su propio editor de código, sus etc y, tranquilos, lo seguirá usando. Pero existen elementos comunes en nuestro proyecto, ese es nuestro entorno común, en el que debemos de organizarnos y operar de una forma definida para lograr un proyecto en el que todos sus elementos estén escritos en el mismo "idioma", de forma que podamos entendernos entre nosotros.

3.1. Estructura de directorios

En nuestro proyecto de Unity tenemos gran variedad de archivos, pero podemos agruparlos por su finalidad o tipo. En clase hemos creado algunas carpetas como Scripts y Prefabs. Hay más tipos de archivos, por lo tanto, se han creado nuevas carpetas y esta es la finalidad de cada una:

- Animations: Archivos de animación (.anim) y controladores (.controller)
- Audio: Música y efectos de sonido (.mp3, .wav, .ogg)
- Materials: Materiales y shaders
- Models: Modelos 3D (.fbx, .obj) y sus texturas asociadas
- Prefabs: Prefabs reutilizables (personajes, ítems, UI)
- Resources: Archivos cargados dinámicamente con 'Resources.Load()'
- Scenes: Escenas del juego (.unity)
- Scripts: Código en C# (.cs)
- Sprites: Imágenes 2D (.png, .jpg)
- UI: Elementos de interfaz gráfica (botones, paneles, etc.)
- Plugins: Librerías externas (DLLs, paquetes)
- StreamingAssets: Archivos que deben permanecer sin compresión (videos, configuraciones)
- Editor: Scripts exclusivos para el Editor de Unity
- Tests: Pruebas unitarias y de integración
- Shaders: Sombras y efectos gráficos personalizados
- ThirdParty: Assets de terceros (Asset Store, paquetes externos)

Esta estructura de carpetas es producto de una consulta a ChatGPT preguntando cual es la estructura de carpetas más común para proyectos en Unity.

Obviamente esto tampoco está escrito en piedra, también podemos crear nuestras propias carpetas así como también crearemos subcarpetas dentro de las carpetas principales.

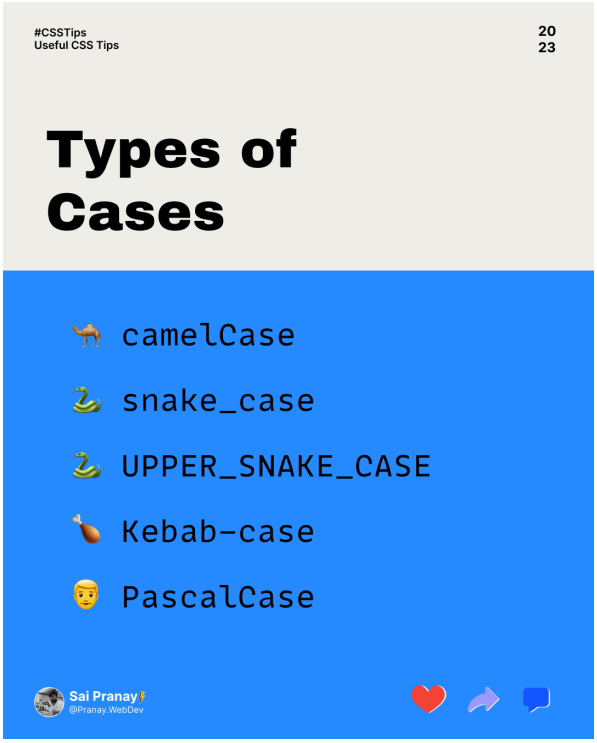

Figura 2: Formatos de escritura @Pranay.WebDev

3.2. Estilo de programación

También mantengamos la coherencia a la hora de programar, para ello en esta página tenéis una referencia de programación de C#, concretamente para Unity. De todas formas os dejo aquí un resumen:

Estilo :

- Las variables es escriben en camelCase.
- Las clases se escriben en PascalCase.
- Los atributos privados se escriben en _camelCase, nótese el _ al principio.
- Los atributos públicos se escriben en PascalCase, hay que definir sus getters y setters.
- Los atributos estáticos se escriben en PascalCase.
- Los métodos se escriben en PascalCase.
- Los parámetros (de métodos, eventos, acciones) se escriben en camelCase.
- Los namespaces se escriben en PascalCase.
- Los delegates se escriben en PascalCaseCallback, esto es PascalCase con el sufijo Callback.
- Los eventos y acciones se escriben en OnPascalCase, esto es PascalCase con el prefijo On delante. Para los que no sepais que es esto, esto no es un tipo de método al que etiquetamos como acción o evento, es otra cosa.
- Las llaves se abren en una nueva línea.
- Véase la figura 3 para el estilo de los switches.

```
switch (someExpression)
{

   case 0:
      {
         DoSomething();
      }
      break;

   case 1:
      {
         DoSomethingElse();
      }
      break;

   case 2:
      {
         int n = 1;
         DoAnotherThing(n);
      }
      break;
}
```

Figura 3: Estilo de switches en C#

Seguramente al leer esta lista habéis visto términos como delegates, eventos y acciones. Os recomiendo que investiguéis acerca de esto, porque son recursos muy útiles, sobre todo los eventos. Y si no, podéis consultármelo personalmente y os lo aclaro.

Por cierto, tengo decir que ese no es mi estilo, este es el estándar, yo soy de programar en C con snake_case y llaves empezando en la definición de la función (que es el estándar de C). Entonces, al igual los demás, tendré que adaptarme.

3.3. Modo de trabajo en git

No voy a establecer una metodología de trabajo en git, pero quiero que cada característica (feature) se desarrolle en una rama aparte del main. También quiero que según vayais terminando partes de vuestro trabajo y hayáis comprobado que funcionan bien, vayáis haciendo merges al main, **Solo si habeis comprobado que funciona bien**. No os olvidéis de ir haciendo merges continuamente porque si tardamos mucho en hacer merges, esto puede resultar en merge conflicts interminables.

3.3.1. Commits

A la hora de realizar un commit, estos deben de ser lo más atómico posibles, prácticamente commit por función. También su mensaje de commit tiene que ser descriptivo. Un estándar en la industria son los conventional commits, que os lo resumo aquí abajo:

Los mensajes de commit tienen esta forma:
<tipo>[(ámbito *opcional*)]: <descripción>

Como podemos ver, el ámbito es opcional, el ámbito puede ser un archivo, un módulo, un conjunto de módulos, una escena, etc...

Algunos de los posibles **tipos** son:

- feat: nueva característica (feature)
- perf: mejora de rendimiento (performance)
- docs: documentación (tanto interna como externa)

- refactor: refactorización de código
- style: cambios que afectan al estilo del código (indentación, casetypes...)
- chore: cambios que no afectan la lógica del código o la funcionalidad

Aquí dejo algunos **ejemplos de commits reales** (algunos con ámbito y otros sin):

- feat(MissionManager.cs): Implementada misión de matar goblins
- perf(map generation): reducido el tiempo de generación de mapa mediante el algoritmo x
- docs(UIManager.cs): especificación de funciones
- refactor(Player): Separada la lógica del jugador en varias funciones
- style(PlayerMove.cs): ajusta la indentación en el archivo PlayerMove.cs
- chore: limpiar archivos de configuración no utilizados

De todas formas, si no sabéis cómo etiquetar un commit, podéis explicarle a chatGPT lo que habéis hecho y que os lo etiquete conforme a Conventional Commits, para ir aprendiendo.

3.3.2. Merges

Importante, antes de hacer un merge al main hay que hacer un merge del main a nuestra rama. ¿Por qué? Principalmente, al hacer un merge del main a nuestra rama integraremos las nuevas características y podremos probar si todo funciona bien **en nuestra rama**. Por lo tanto, si surgen problemas, en vez de surgir en la rama main que es donde todos trabajamos, estarán en nuestra rama y ahí los arreglaremos.

3.4. Documentación

Para la documentación en código de nuestro proyecto se hará uso de la documentación en formato XML. Que son de la forma que muestra la figura 4. Si se os hace muy cuesta arriba, siempre podéis pedirle a Copilot o ChatGPT que os la genere, pero no os olvidéis de revisarla.

Luego, para la documentación externa haremos uso de una carpeta de drive donde a cada módulo le corresponderá un documento detallando cómo ha sido estructurado, las consideraciones de los autores, además de otra información que se considere relevante a la hora de trabajar con él. Cómo este proyecto va a tener un ciclo de vida bastante breve, no creo que sea necesario elaborar documentos extensos, como los que se hacen de forma habitual. Por lo tanto, ofrezco total libertad de estructuración y escritura del documento. De todas formas os dejo aquí el resultado de mi consulta con ChatGPT sobre cómo se suelen elaborar estos documentos (podéis saltaros esta parte e ir a la siguiente sección):

1. **Nombre del módulo**
   Ejemplo: *Módulo de Inventario*
2. **Propósito / Descripción general**
   Breve explicación de qué hace este módulo y por qué existe.
   Ejemplo: Este módulo gestiona los objetos recolectables del jugador, permitiendo almacenar, usar y eliminar ítems desde un menú de inventario.
3. **Clases principales**
   Lista de scripts relevantes con su descripción.
   - `InventoryManager`: Clase principal que maneja la lógica del inventario.
   - `Item`: Representa un objeto que puede estar en el inventario.
   - `InventoryUI`: Renderiza el inventario en pantalla.

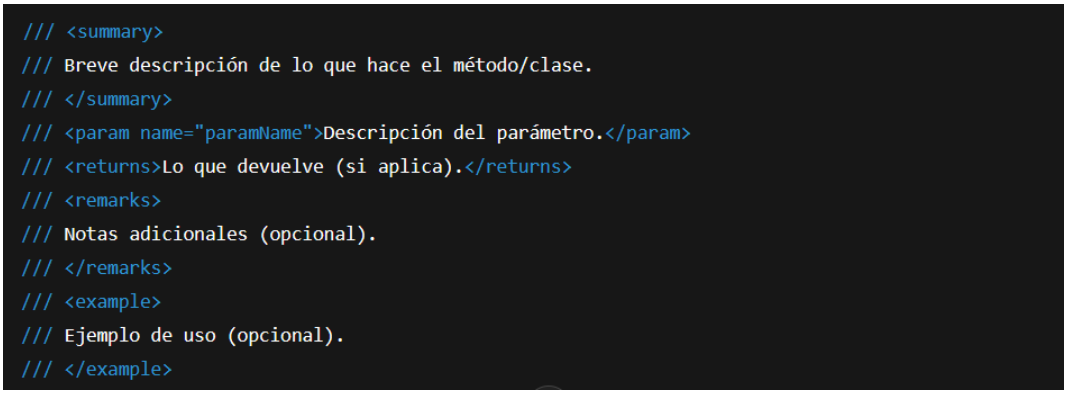

Figura 4: Documentación en formato XML (fuente: ChatGPT)

4. **Dependencias**
   Módulos o sistemas que este módulo necesita (ej. Audio, Input, SaveSystem).
5. **Estructura del sistema**
   Explicación de cómo se relacionan las clases. Puede incluir:
   - Patrón de diseño usado (Singleton, Observer, etc.)
   - Diagrama (incluir como imagen si se tiene)
6. **Flujo de uso / ciclo de vida**
   Qué ocurre desde el inicio hasta el uso completo del módulo en runtime. Ejemplo:
   - El jugador recoge un ítem → Se notifica a `InventoryManager`.
   - Se actualiza la UI mediante `InventoryUI.Refresh()`.
7. **Configuración en el editor de Unity**
   Instrucciones para preparar el módulo:
   - Prefabs necesarios
   - GameObjects que deben contener ciertos scripts
   - Componentes requeridos
8. **Eventos y mensajes**
   Eventos definidos en el módulo y su uso. Ejemplo:

   ```
   public static event Action<Item>OnItemAdded;
   ```
9. **Ejemplos de uso**
   Fragmentos de código o pseudocódigo que muestran cómo interactuar con el módulo.
10. **Errores comunes y debugging**
    Lista de problemas comunes y cómo solucionarlos. Logs útiles para depurar.
11. **Mejoras pendientes / TODOs**
    Funcionalidades por implementar, bugs conocidos o ideas de mejora.

4. Descubrir cosas nuevas

Estoy bastante seguro de que con lo que hemos dado en la carrera seremos capaces de desarrollar este proyecto sin problema. Sin embargo, os invito a descubrir nuevas técnicas, estructuras, patrones, etc; que no hemos dado en la carrera y que son capaces de resolver problemas que a primera vista parecerán complejos, pero estos son capaces de resolverlos de una forma simple y elegante. Estos son algunos de ellos:

- Patrón singleton (La implementación de singleton en Unity se hace de una forma diferente, dado que las clases que heredan de Monobehavior no construyen los objetos llamando al constructor de clase, entonces no sirve de nada hacer el constructor privado)
- Delegates, acciones, eventos y UnityEvents (Estos dos últimos no son lo mismo).
- Pools (lo dimos en clase, es lo de reutilizar objetos en vez de destruirlos y volver a construirlos)
- ScriptableObjects en Unity
- Unity new Input System (lo dimos en clase, importante evitar usar el sistema antiguo)
- C# interfaces

Y habrá muchos más, pero estos son de los que he logrado acordarme y los que me han parecido más importantes.

5. Nota final (Importante)

He elaborado este documento **no** con el propósito de que os lo aprendáis de memoria, sino para que lo tengáis abierto en segundo plano y lo vayáis consultando cuando sea necesario.