Trabajo de Fin de Grado

Entorno de desarrollo reutilizable para videojuegos basado en patrones de diseño

Reusable development environment for video games based on design patterns

David Rodríguez Báez

La Laguna, 26 de junio de 2016

D. **Francisco Almeida Rodríguez**, con N.I.F. 4283571-M profesor Titular de Universidad adscrito al Departamento de Nombre del Departamento de la Universidad de La Laguna, como tutor

C E R T I F I C A (N)

Que la presente memoria titulada:

"Entorno de desarrollo reutilizable para videojuegos basado en patrones de diseño" ha sido realizada bajo su dirección por D. **David Rodríguez Báez**, con N.I.F. 54052962-H.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 26 de junio de 2016.

Agradecimientos

A mis padres, por enseñarme que la perseverancia y la constancia marcan la diferencia entre el triunfo y el fracaso. Por un apoyo que nunca faltó, porque cuando tienes unos referentes tan maravillosos solo intentas estar a su altura y que estén orgullosos de ti.

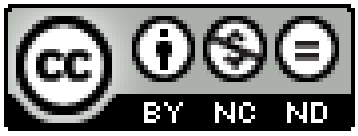
A mi gran hermano, porque me marcó un camino a seguir, estuviste desde el principio (y mira que era pesadito con las preguntas que te hacía xD) y hasta el día de hoy nunca me han faltado tus consejos.

A mí novia y compañera en la vida, por el cariño y apoyo que en muchos momentos difíciles siempre he recibido de ti. Porque cada día que pasa tengo más claro que juntos compartiremos grandes sueños y adoro que seamos grandes soñadores.

A mis compañeros de facultad, porque nunca me vi solo en una carrera que muchas veces puede ser bastante dura. Porque poco a poco se forjó una segunda familia, que ha sido crucial en los éxitos que hemos conseguido en los últimos años y muchos que aún nos quedan por conseguir.

A mi director de proyecto, porque más allá de un tutor se ha convertido en un mentor que siempre ha dicho las cosas claras y que sin él no hubiera sido posible este proyecto.

Licencia



Resumen

El objetivo de este trabajo ha sido desarrollar una plataforma de videojuegos, que a su vez será propiamente un videojuego, para Smartphone y Tablet Android, basado en patrones y con un modelo reutilizable.

Al comienzo y durante el transcurso del videojuego, el usuario puede modificar (de manera trivial), el entorno del mismo. Asimismo, el cambio en el contexto que envuelve al personaje, tendrá una repercusión en el modelo inteligente que es el motor de la aplicación.

Los escenarios y características del personaje pueden ser exportados e importados. Además, incluir nuevas estructuras de objetos, así como otros "minijuegos", dentro de la plataforma se postula como una tarea sencilla.

Palabras clave: Android, Juego, Videojuego, Patrón de Diseño, Reutilización, Exportación y Estructura.

Abstract

The objective of this final degree project has been to develop a gaming platform, which also can be used like a game, for smartphones and tablets with Android OS. This platform is based on design patterns and has a reusable model. The user can modify, at the beginning and during the game, the environment of the game itself. Changes in the context of the environment that covers the character will have consequences in the intelligent model, which is the motor of the application.

The scenarios and characteristics of the character can be exported and imported. In addition, the platform allow to include new objects structures, as well as other mini-games, inside the platform itself.

Keywords: Android, Game, Design Patterns, Gaming, Reusable, Export, Structure.

Índice General

Índice de figuras

Índice de tablas

Capítulo 1. Análisis

1.1 Introducción

En este capítulo recogeremos información relevante sobre algunas aplicaciones similares que podemos encontrar en el mercado. Esto nos ayudará a tener un punto de vista más amplio, que nos aporte nuevas ideas y nos ayude a concretar las que ya teníamos. Además, estableceremos los requisitos, funcionales y no funcionales, que nuestro proyecto deberá cumplir.

1.2 Antecedentes

En el mercado se pueden encontrar videojuegos de diferente índole. Desde simuladores, en los cuales suelen existir estrategias complejas y cambiantes, basadas en una serie de conocimientos específicos, hasta juegos de arcade, en los que la estrategia suele pasar a un segundo plano.

Realizando una búsqueda de videojuegos o aplicaciones similares a la que se desarrollará en este proyecto, en la red se aprecian los siguientes videojuegos:

- **Brick-force** [25]**:** Desde el punto de vista creacional del entorno del juego, se podría decir que la idea que plantea brick-force se asemeja a la que se pretende desarrollar en este proyecto. En dicho juego se podrá modificar el entorno del personaje, cumpliendo una serie de reglas si se desea jugar con otros usuarios.
- **Black & White** [27]**:** En dicho juego Peter Molyneux propone la difícil tarea de ser Dios, en el transcurso del juego se moldeará a una criatura que podrá ser buena, mala o neutral, dependiendo de las acciones llevadas a cabo por el usuario.
- **RPG Maker** [4]**:** Herramienta para crear un videojuego con una historia propia, dentro de unas limitaciones. Es bastante sencillo de utilizar. Se dispondrá de un panel donde encontrar los elementos disponibles para añadir al entorno de creación. También se podrá añadir distintos personajes y especificar como luchan entre ellos.
- **Far Cry** [26]: Juego para PC donde se permitirá seleccionar las habilidades del personaje y, según las habilidades que se seleccionen, la interacción con el entorno será diferente.
- **Minecraft** [28]: Desarrolla la idea de un mundo abierto en el que se moldeará a placer el mapa por el que se mueve el personaje. Además se

dispondrá de la posibilidad de recolectar recursos y crear objetos con distintas utilidades.

1.3 Requisitos

Dividiremos los requisitos del proyecto en funcionales y no funcionales.

1.3.1 Requisitos Funcionales

- Importación de escenarios.
- Importación de personajes.
- Exportación de escenarios.
- Exportación de personajes.
- Configuración de personajes a través de los ficheros de exportación.
- Configuración de escenarios.
- Modificación de personajes.
- Modificación de escenarios.
- Pausado de juego.
- Guardado de juego.
- Videojuego en 3D.
- Juego en primera persona.
- Mostrar características del personaje.
- Capacidad de incorporar minijuegos.
- Distintos tipos de escenarios.

1.3.2 Requisitos no funcionales

- Usabilidad: El videojuego se desarrollará teniendo en cuenta que la franja de edad media de los futuros consumidores oscilará entre 12 y 30.
- Disponibilidad en Android:
    - Smartphone.
    - Tablet.
- Portabilidad: Se permitirá que el usuario pueda exportar su personaje y el escenario que ha diseñado.
- Escalabilidad: La arquitectura basada en patrones permitirá la adición de nuevos objetos, escenas y minijuegos.

Capítulo 2.
Unity versus otros motores de desarrollo de videojuegos

2.1 Introducción

En este capítulo haremos una breve comparativa de los principales motores de desarrollo que se encuentran actualmente en el mercado. Esta información y otros factores nos ayudaron a seleccionar el motor en el que nos hemos basado en este proyecto.

2.2 Motores de desarrollo de videojuegos

Para poder seleccionar correctamente la plataforma de desarrollo del videojuego, se han analizado las siguientes:

- **Ogre 3D** [28]**:** Es un motor de renderizado 3D, orientado a escenas y de software libre. Fue diseñado para que a los desarrolladores les resultase más fácil e intuitiva la producción de aplicaciones que utilizasen gráficos 3D acelerados por hardware.
- **Blender** [29]**:** La mayor ventaja de este motor es que, siendo gratuito, es muy completo, incluye: iluminación, renderizado, animación, creación de gráficos tridimensionales, etc.
- **Unity** [23]**:** Es un motor de videojuego multiplataforma creado por Unity Technologies. Cuenta con iluminación en tiempo real, texturas basadas en materiales del mundo real y reflejos HDR.

Después de analizar diversas plataformas, se optó por la versión gratuita de Unity, debido a que se adaptaba mejor a los requerimientos del proyecto, y además contaba con numerosos tutoriales de diferente índole.

Está disponible como plataforma de desarrollo para Microsoft Windows y OS X. Permite la exportación del juego para las siguientes plataformas: iOS, Android, Windows 8, BlackBerry 10, Linux, PS3, PS4, XBOX One, etc.

Unity permite que se desarrolle código en: UnityScript (variación de Javascript), C# y Boo (derivado de Python).

Desde el sitio web oficial se pueden descargar dos versiones: “Personal Edition” (utilizada en este proyecto) y “Professional Edition”.

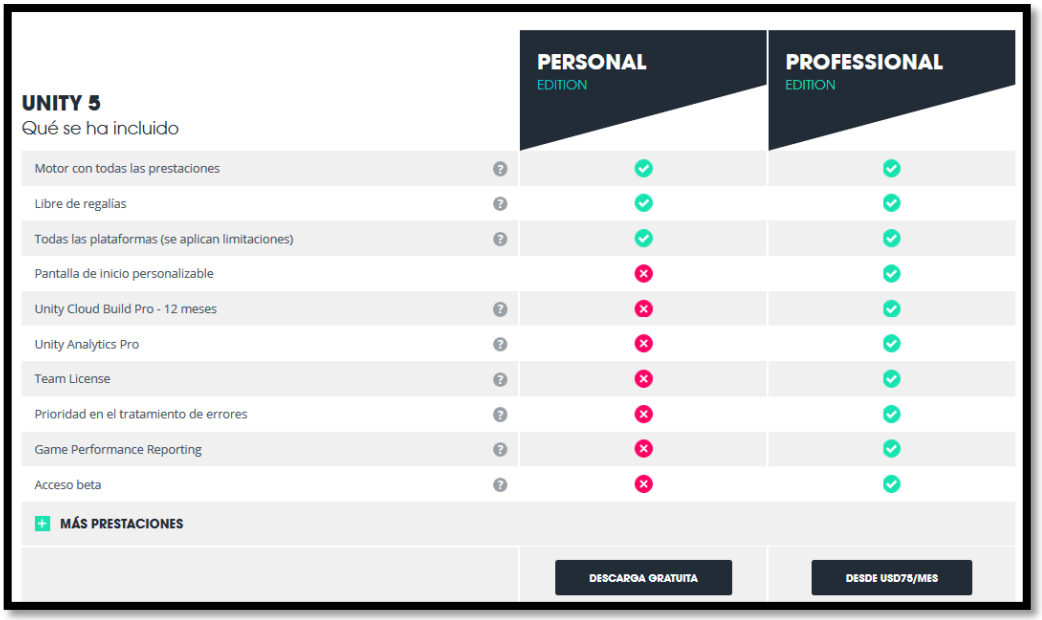


Ilustración 1. Comparativa versiones de Unity

El lenguaje de desarrollo que se ha utilizado ha sido C#, ya que es un lenguaje de programación orientado a objetos. Esta ideología encaja perfectamente con la estructura que se pretende desarrollar.

2.3 Unity, conceptos básicos

Como paso previo para poder explicar cómo hacer uso de los patrones en Unity, se procederá a explicar algunos de los conceptos básicos de esta plataforma:

- **Assets**: Objetos básicos de cualquier proyecto que componen las escenas del videojuego. Estos pueden ser una imagen, un script, un *prefab*, un modelo o un sonido.
- **Scenes**: Son las áreas de contenido del videojuego. Por ejemplo, una escena es el menú principal y otra es la configuración del juego. Cada escena está compuesta por *assets*, y todas las escenas forman el videojuego.
- **GameObjects**: Cuando utilizamos un *asset* en una escena, este pasa a convertirse en un *GameObject* (Objeto de juego).
- **Components**: Pueden ser de distintos tipos. Sirven para crear un comportamiento, definir la apariencia o influenciar en otros aspectos a un objeto de nuestro juego. Por ejemplo: partículas, luces, cámaras, scripts, etc.
- **Script**: Es un componente esencial en nuestro videojuego. Son la base para crear nuestra aplicación, con ellos definiremos patrones de diseño, inteligencia artificial de nuestro juego, etc.

- **Prefabs**: Un *Prefab* (prefabricado) es un tipo de *GameObject* reutilizable, es decir, se puede insertar en cualquier escena el número de veces que se quiera. Cuando se agrega un *Prefab* a una escena, se crea una instancia del mismo. Todas las instancias del *Prefab* están vinculadas con el original y son clones de esté. Al realizar cualquier cambio en el Prefab original los cambios se aplican a todas las instancias.

Desde el punto de vista de la interfaz, Unity está compuesto por dos vistas principales "Hierarchy" e "Inspector". En la primera podemos añadir los objetos de la escena que estemos modificando. Con estas escenas tendremos nuestro videojuego.

Dichos objetos tienen características en función del tipo que sean (*Cube*, *Spheres*, *Point Ligth*, etc.). Estos atributos los podemos configurar en la vista de "Inspector".

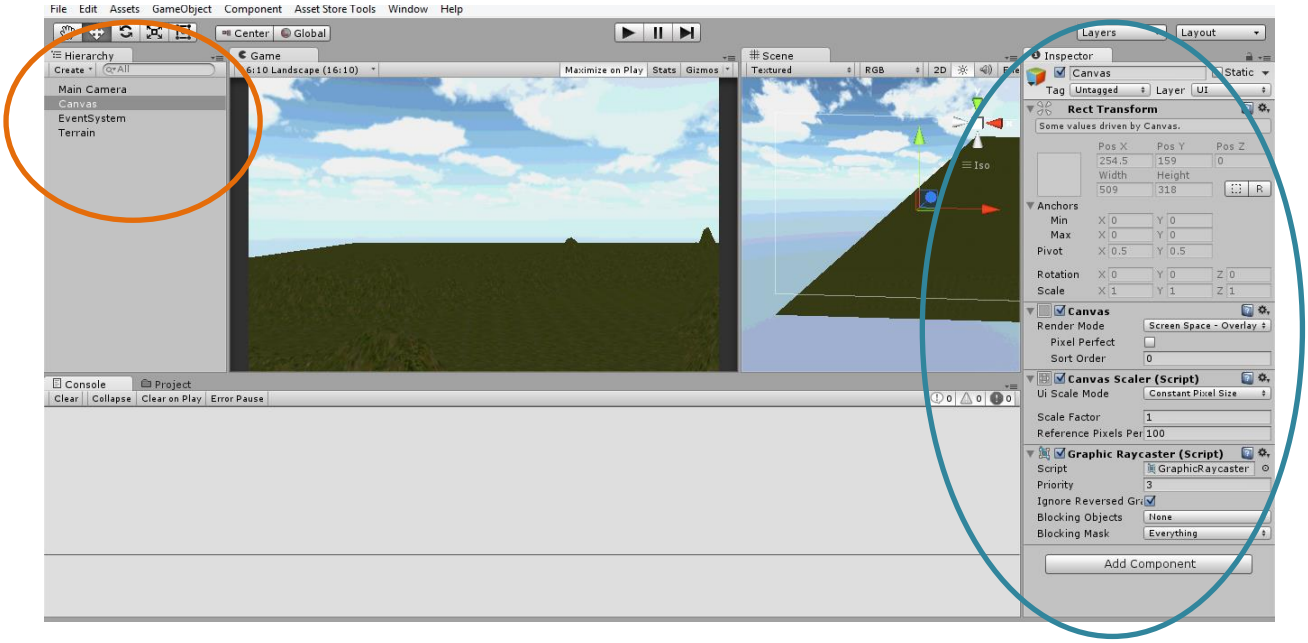


Ilustración 1. Unity. Naranja: vista "Hierarchy". Azul: vista "Inspector"

Capítulo 3. Patrones de diseño

3.1 Introducción

Los patrones de diseño [3] y la inteligencia artificial se han fusionado con el objetivo de proporcionar una estructura reutilizable. En ella se podrá ir incorporando nuevos objetos, sin necesidad de conocer todo el contexto, que afecten al modelo inteligente del juego. Esta miscelánea entre ambas materias proporcionará un modelo altamente escalable, cambiante y responsivo.

3.2 Patrones utilizados

Pasaremos a describir los patrones de diseño utilizados, los problemas asociados a los mismos, las ventajas que estos tienen en nuestro videojuego, y cómo se deberían implementar en el caso de querer aprovechar al máximo sus funcionalidades.

3.2.1 Factoría Abstracta:

El objetivo de este patrón es poder crear nuevas familias de objetos que tengan cualidades en común. Cada familia definirá de forma precisa, y posiblemente diferente, esas características que tienen en común.

Para este videojuego se han creado tres tipos de escenas haciendo uso del patrón Factoría Abstracta.

Ilustración 2.Patrón Factoría Abstracta. Escenas del proyecto

I.3.2.1.1 Ventajas:

- Posibilidad de añadir nuevas familias de objetos.
- Facilidad para cambiar las características de una familia.
- Abstracción, por parte del cliente, de las familias que pueden existir, puesto que este trabaja con una clase abstracta y no con una familia concreta.

I.3.2.1.2 Diagrama de clases:

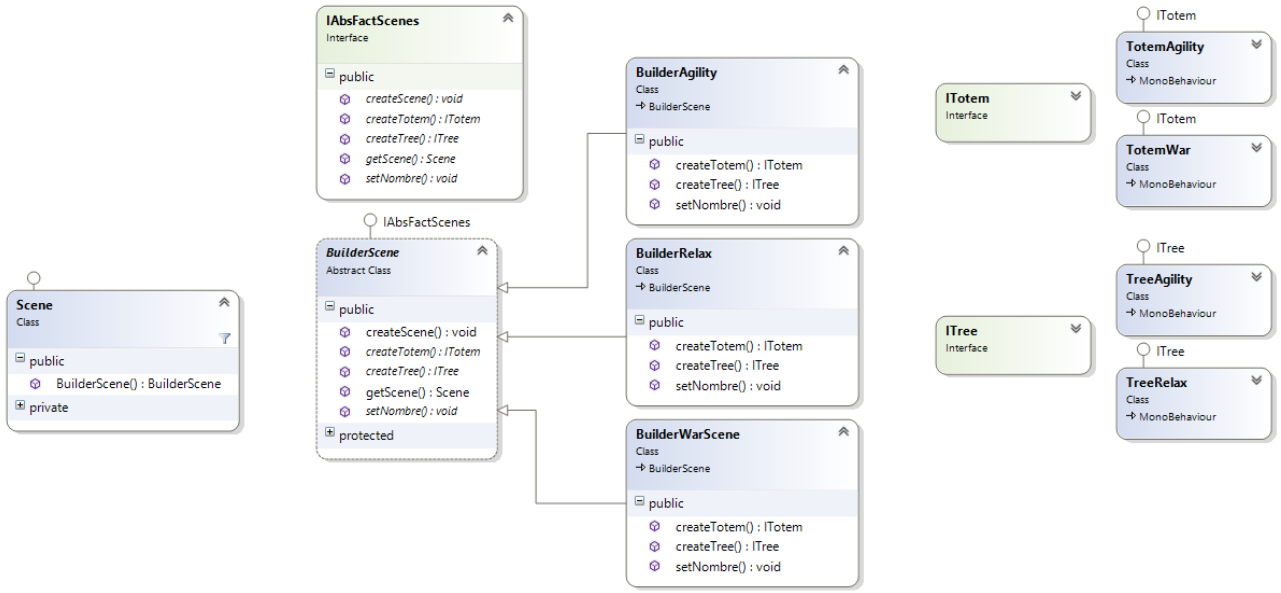


Ilustración 3. Diagrama de Clases Patrón Factoría Abstracta

Las componentes del patrón son los siguientes:

- **Factoría Abstracta:** Define una interfaz para la creación de objetos abstractos. En este proyecto está representada por los siguientes elementos:
  - IAbsFactScenes: Esta interfaz representa la creación de escenas abstractas del videojuego. Define los métodos que deben implementar las factorías de las escenas concretas.
  - BuilderScene: Esta clase abstracta implementa la interfaz IAbsFactScenes y define el método "createScene()" ya que se quiere que la responsabilidad de mantener la referencia a la escena de Unity recaiga sobre una escena abstracta y no sobre las escenas concretas.
- **Factoría Concreta:** Implementa las operaciones necesarias para la creación de objetos concretos. En el proyecto, las factorías concretas quedan definidas por las siguientes clases:
  - BuilderAgility: Define los métodos de "IAbsFactScenes", de manera específica, para crear escenas de estrategia.

    - BuilderRelax: Esta clase implementa los métodos de la interfaz, de manera que se creen escenas de relajación.
    - BuilderWarScene: Define los métodos, de una forma concreta, para permitir la creación de escenas de guerra.
- **Producto Abstracto:** Declara una interfaz que represente los objetos de un tipo de producto. En el proyecto, se corresponde con las siguientes interfaces:
    - ITotem: Establece los métodos necesarios para definir los distintos tipos de tótem.
    - ITree: Define una interfaz que representa los distintos tipos de árbol que estarán ligados a cada escena.
- **Producto Concreto:** Implementa la interfaz del producto abstracto correspondiente. Especifica un producto concreto que la debida fábrica concreta deberá crear. En el proyecto, quedan establecidos por las siguientes clases:
    - TotemAgility: Producto concreto representativo de las escenas de estrategia. Visualmente se corresponde con un libro y está vinculado al minijuego de estrategia.
    - TotemWar: Producto concreto representativo de las escenas de guerra. Visualmente se corresponde con una pistola y está vinculado al minijuego de guerra.
    - TreeAgility: Producto concreto que representa un árbol relativo a las escenas de estrategia.
    - TreeRelax: Producto concreto que representa un árbol relativo a las escenas de relajación.
- **Cliente:** Utiliza la factoría para crear las distintas familias. En el proyecto queda representado por la siguiente clase:
    - Scene: Cliente que utiliza la clase abstracta "BuilderScene" para crear las distintas escenas (estrategia, guerra y relajación). Se utiliza esta clase abstracta, en vez de la interfaz "IAbsFactScenes", ya que se necesita mantener la referencia a la escena de Unity.

I.3.2.1.3 Utilización:

- Si se quisiera añadir un nuevo escenario, el desarrollador debería crear un objeto que implemente la interfaz asociada a las características básicas que debe tener una escena.

3.2.2 Constructor:

Este patrón posibilita el crear un objeto completo a partir de otros más “simples”. Centraliza el proceso de creación en un único punto, permitiendo que el objeto sea creado de diferentes formas, según las partes que lo compongan.

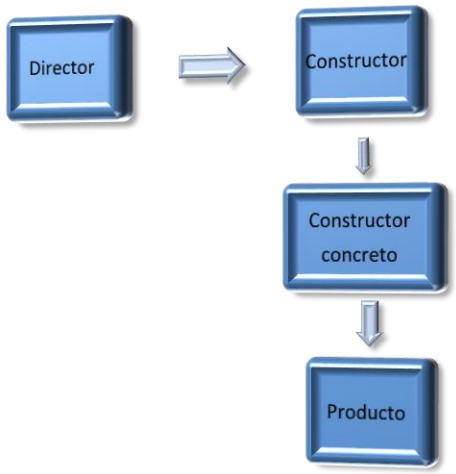


Ilustración 4. Patrón Constructor

I.3.2.2.1 Ventajas:

- Permite añadir nuevas partes que compongan un objeto.
- Ampliación de la funcionalidad de un objeto.
- Permite cambiar el proceso de creación de una manera sencilla.

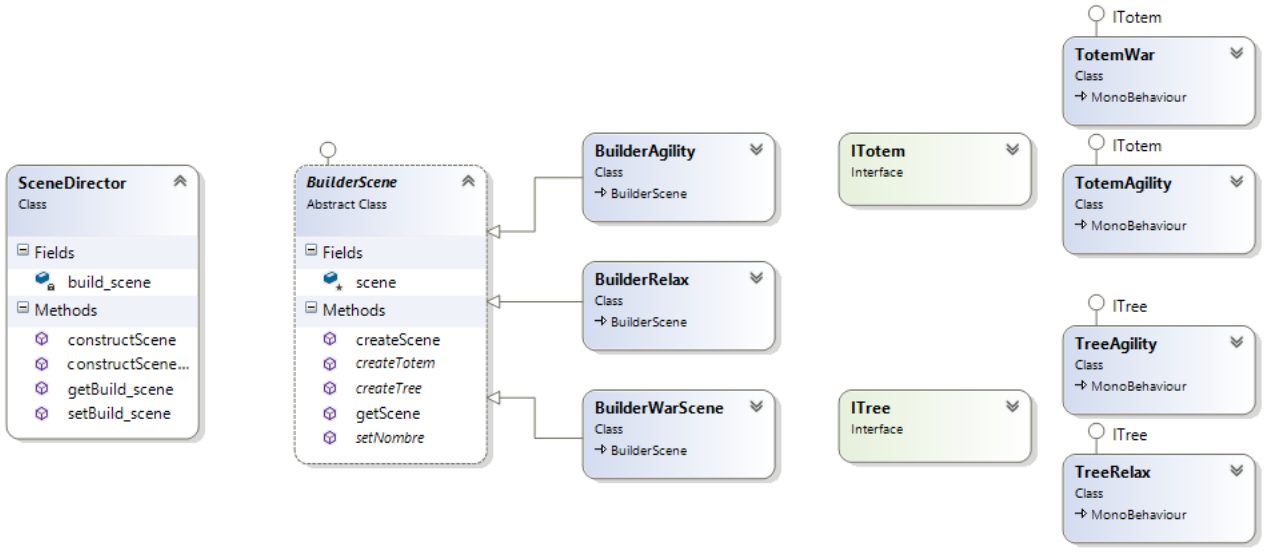


Ilustración 5. Diagrama de Clases Patrón Constructor

Las componentes del patrón son los siguientes:

- **Producto Abstracto:** Declara una interfaz que represente los objetos de un tipo de producto. Véase el Patrón Factoría Abstracta.
- **Producto Concreto:** Define el objeto complejo que se quiere construir. Véase el Patrón Factoría Abstracta.
- **Constructor:** Declara las operaciones necesarias, clase abstracta, para la creación de las partes del Producto. Véase el Patrón Factoría Abstracta.
- **Constructor Concreto**: Implementa Constructor y acopla las partes que constituyen el objeto complejo. Véase el Patrón Factoría Abstracta.
- **Director:** Construye un objeto haciendo referencia a la interfaz Constructor. En el proyecto queda representado por la siguiente clase:
    - SceneDirector: Clase que hace referencia al Constructor, sin conocer el tipo de constructor concreto (BuilderAgility, BuilderRelax e BuilderWarScene) que se llegará a utilizar.

I.3.2.2.2 Utilización:

- Si se deseara añadir un nuevo constructor, el desarrollador debería crear una clase que extienda de la clase abstracta “Constructor”. En el caso de querer añadir un producto concreto, este debe implementar la interfaz del producto abstracto (ITotem ó ITree).

3.2.3 Singleton:

Restringe la creación de objetos pertenecientes a una clase. Garantiza que una clase únicamente tenga una instancia y que las demás clases puedan acceder a ella.

Este patrón permite solucionar los problemas asociados a la liberación de memoria que Unity realiza al cambiar de escenas, por ejemplo cuando pasamos del menú principal y empezamos a jugar. Dicha liberación de memoria borra los datos que se tengan hasta el momento, es por esto que necesitamos utilizar este patrón para guardar el estado del juego así como los otros componentes esenciales.

Además, debemos tener en cuenta que Unity no permite que creemos muchas instancias de objetos que están vinculados con partes gráficas, es decir, que implementen clases de Unity. Debido a esto ha sido necesario controlar la creación de estas instancias.

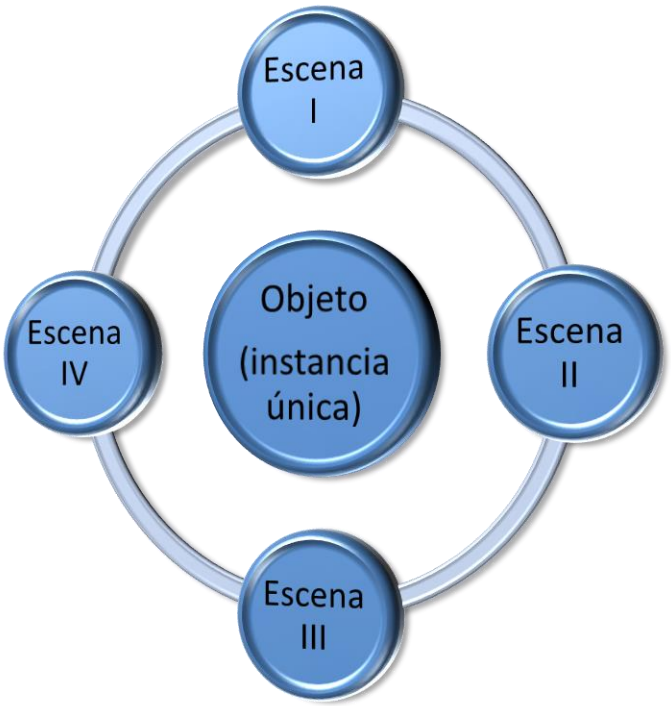


Ilustración 6. Patrón Singleton

I.3.2.3.1 Ventajas:

- Única creación de un objeto complejo.
- Instancia accesible desde los demás objetos de la aplicación.

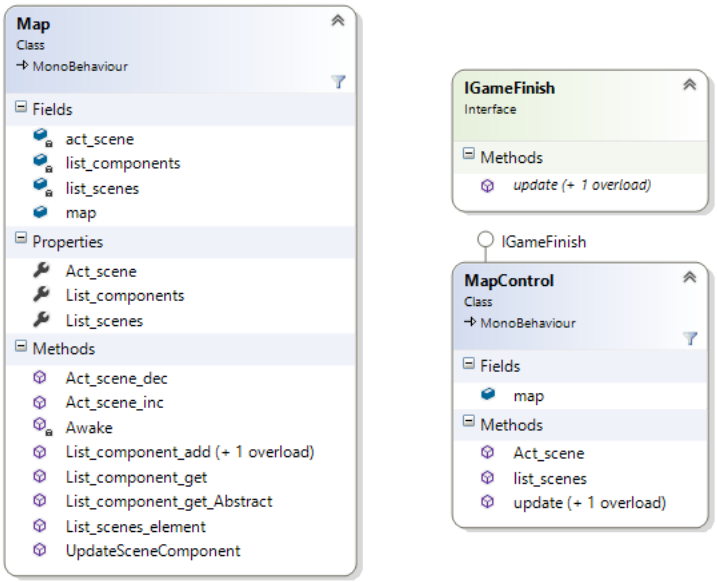


Ilustración 7. Diagrama de Clases Patrón Singleton

Las componentes del patrón son los siguientes:

- **Clase Singleton:** Define una única instancia para que los clientes puedan acceder a ella. En el proyecto queda representado por la siguiente clase:
  - Map: Clase con una única referencia, que almacena información crucial del videojuego.
- **Clase de Control:** Concreta una clase que es la encargada de recuperar y controlar la instancia única de la clase Singleton. En el proyecto queda representado por la siguiente clase:
  - MapControl: Clase encarga de controlar la liberación de memoria, que Unity lleva a cabo cuando se realizan desarrollos sobre la parte gráfica de dicho motor.

I.3.2.3.2 Utilización:

- Los clientes accederán a la instancia única de "Map", siendo transparente para ellos el trabajo que realiza "MapControl".

3.2.4 Decorador:

Permite añadir responsabilidades a objetos concretos de forma dinámica. El patrón decorador se ha utilizado para la creación visual de los objetos que componen los escenarios. Permitiendo que esta creación cambie según el tipo de escena que debamos dibujar.

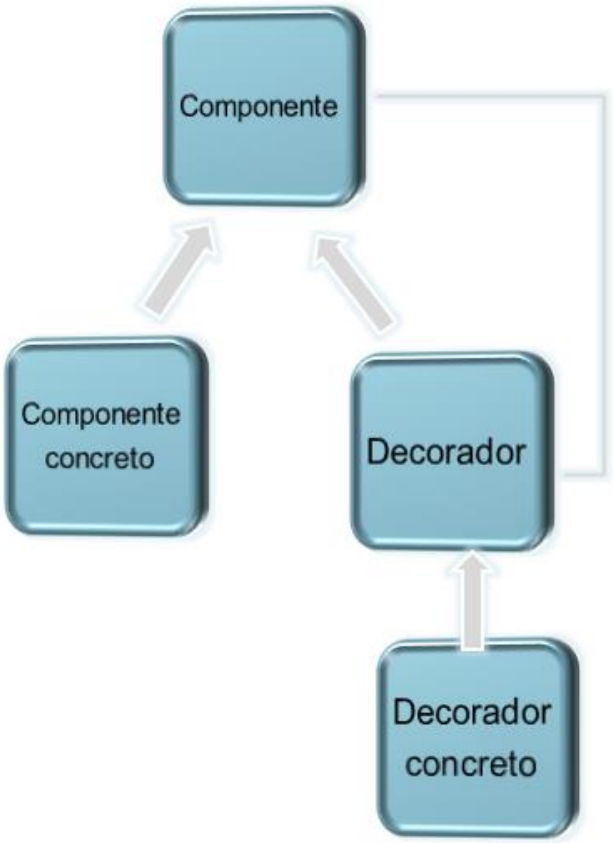


Ilustración 8. Patrón Decorador

I.3.2.4.1 Ventajas:

- Añadir responsabilidades de forma progresiva y dinámica.
- Más flexible que la herencia.

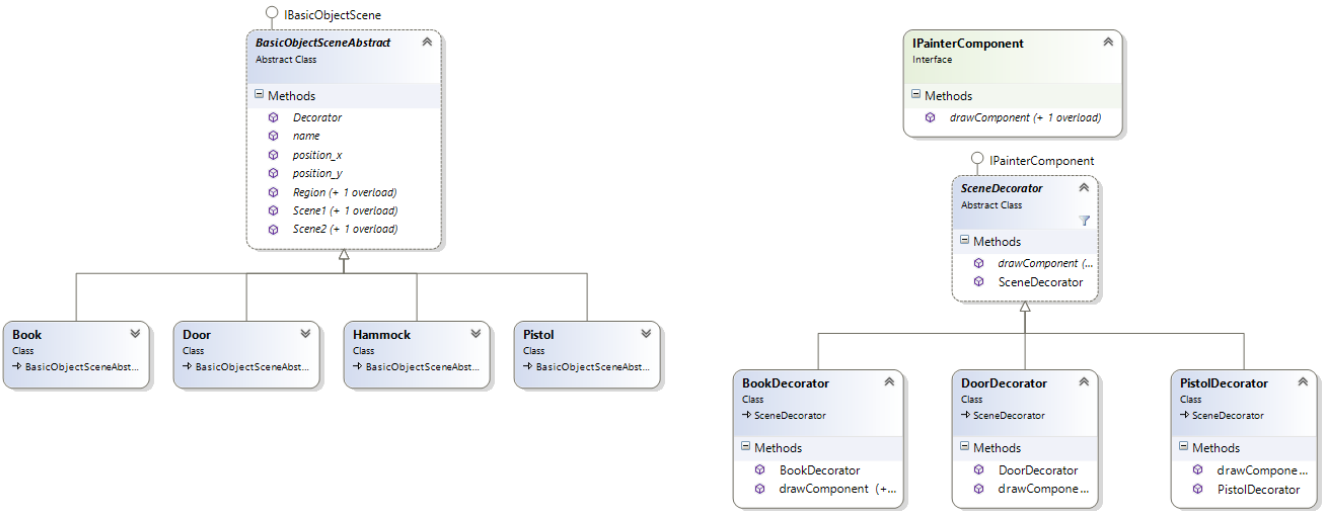


Ilustración 9. Diagrama de Clases Patrón Decorador

Las componentes del patrón son los siguientes:

- **Componente:** Define la interfaz de los objetos a los que se les pueden adicionar responsabilidades dinámicamente. En el proyecto queda representado por la siguiente clase:
    - BasicObjectSceneAbstract: Esta interfaz representa la creación de objetos abstractos del videojuego. Define los métodos que deben implementar los componentes concretos.
- **Componente Concreto:** Define el objeto al que se le puede agregar una responsabilidad. En el proyecto, se corresponde con las siguientes componentes:
    - Book: Componente concreto asociado a la escena de estrategia. Trasladará a un minijuego de estrategia.
    - Door: Componente concreto utilizado en todas las escenas. Llevará a las distintas escenas del videojuego.
    - Pistol: Componente concreto asociado a la escena de guerra. Cargará un minijuego de guerra.
    - Hammock: Componente concreto asociado a la escena de relajación.
- **Decorador:** Define una interfaz para la creación de decoradores concretos. Mantiene una referencia al objeto Componente. En el proyecto queda representado por las siguientes clases:
    - IPainterComponent: Esta interfaz representa la creación de decoradores concretos. Define los métodos que deben implementar los decoradores de las escenas concretas.
    - SceneDecorator: Esta clase abstracta implementa la interfaz "IPainterComponent" y mantiene la referencia al Objeto Componente.
- **Decorador Concreto:** Añade la responsabilidad al Componente. En el proyecto se corresponde con las siguientes clases:
    - BookDecorator: Esta clase se encarga de agregar responsabilidades dinámicamente al componente "Book".
    - DoorDecorator: Esta clase se encarga de agregar responsabilidades dinámicamente al componente "Door".
    - PistolDecorator: Esta clase se encarga de agregar responsabilidades dinámicamente al componente "Pistol".

I.3.2.4.2 Utilización:

- Si se deseara añadir un nuevo decorador, el desarrollador debería crear una clase que extienda de la clase abstracta "SceneDecorator". Esta nueva clase puede extender la responsabilidad de componentes que previamente existan o de nuevos componentes.

3.2.5 Observador:

Patrón de diseño que permite a ciertas clases, llamadas observadores, reaccionar ante un evento determinado. Define una dependencia del tipo uno-a-muchos entre objetos. Cuando uno de los objetos observados cambia su estado, notifica este cambio a todos los objetos dependientes de esta información. El patrón observador se ha utilizado para notificar a la plataforma el resultado de los minijuegos y que esta se reaccione ante dichos resultados.

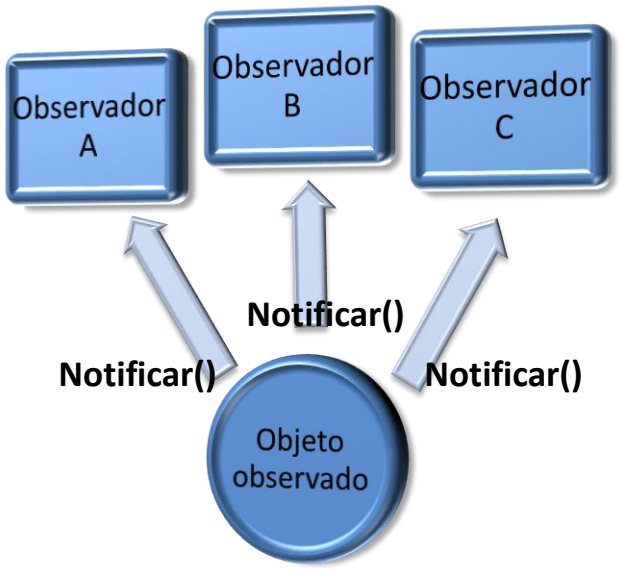


Ilustración 10. Patrón Observador

I.3.2.5.1 Ventajas:

- No hay necesidad de una acoplación fuerte entre clases. Un objeto tiene una lista de observadores que recibirán notificaciones de los cambios del primer objeto.
- Un objeto no conoce a los observadores, solo conoce que es un objeto que puede ser observado.

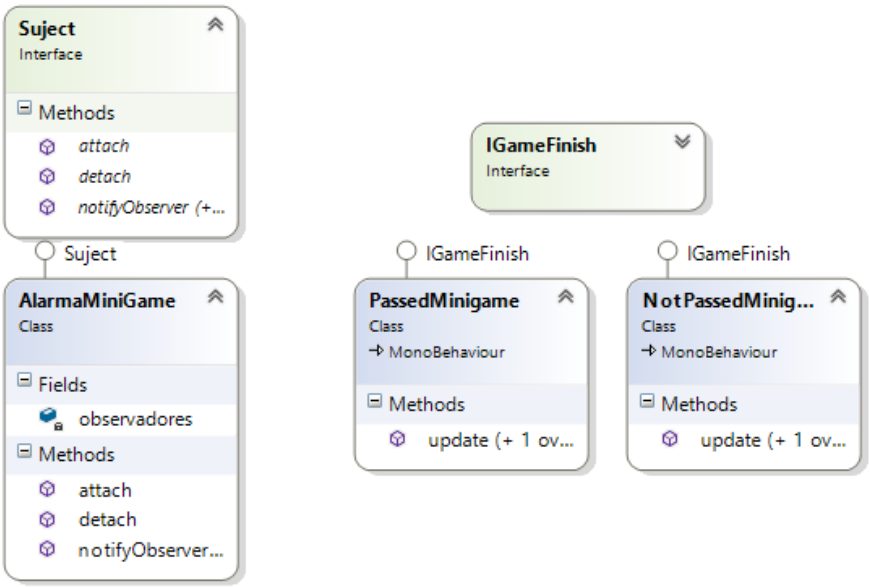


Ilustración 11. Diagrama de Clases Patrón Observador

Las componentes del patrón son los siguientes:

- **Sujeto:** Objeto que ofrece la posibilidad de añadir y eliminar observadores. En el proyecto queda representado por la siguiente clase:
    - Subject: Posee un método llamado attach() y otro detach() que sirven para agregar o excluir observadores.
- **Sujeto Concreto:** Notifica a sus observadores cuando su estado cambia. En el proyecto se corresponde con los siguientes clases:
    - AlarmaMiniGame: Clase que extiende de la interfaz Sujeto, notifica cambios de su estado a los observadores.
- **Observador:** Define la interfaz que sirve para notificar a los observadores los cambios realizados en el Sujeto. En el proyecto queda representado por la siguiente clase:
    - IGameFinish: Esta interfaz representa la creación de objetos observadores. Define los métodos que deben implementar los observadores concretos.
- **Observador Concreto:** Implementa la interfaz de Observador. Mantiene una referencia a un Sujeto Concreto, almacenando información de su estado.

I.3.2.5.2 Utilización:

- Si se deseara añadir un nuevo observador concreto, el desarrollador debería crear un objeto que extienda de la interfaz "IGameFinish". Además se deberá extender una clase de "AlarmaMiniGame", de esta manera los observadores podrán ser notificados del resultado de los minijuegos.

3.3 Patrones de Diseño en Unity

Este apartado explica algunas consideraciones que se han de tener si se quiere utilizar patrones de diseño en Unity, usando el lenguaje C#.

Para poder explicar estas consideraciones, voy a tomar como ejemplo a uno de los patrones implementados en el proyecto, Abstract Factory.

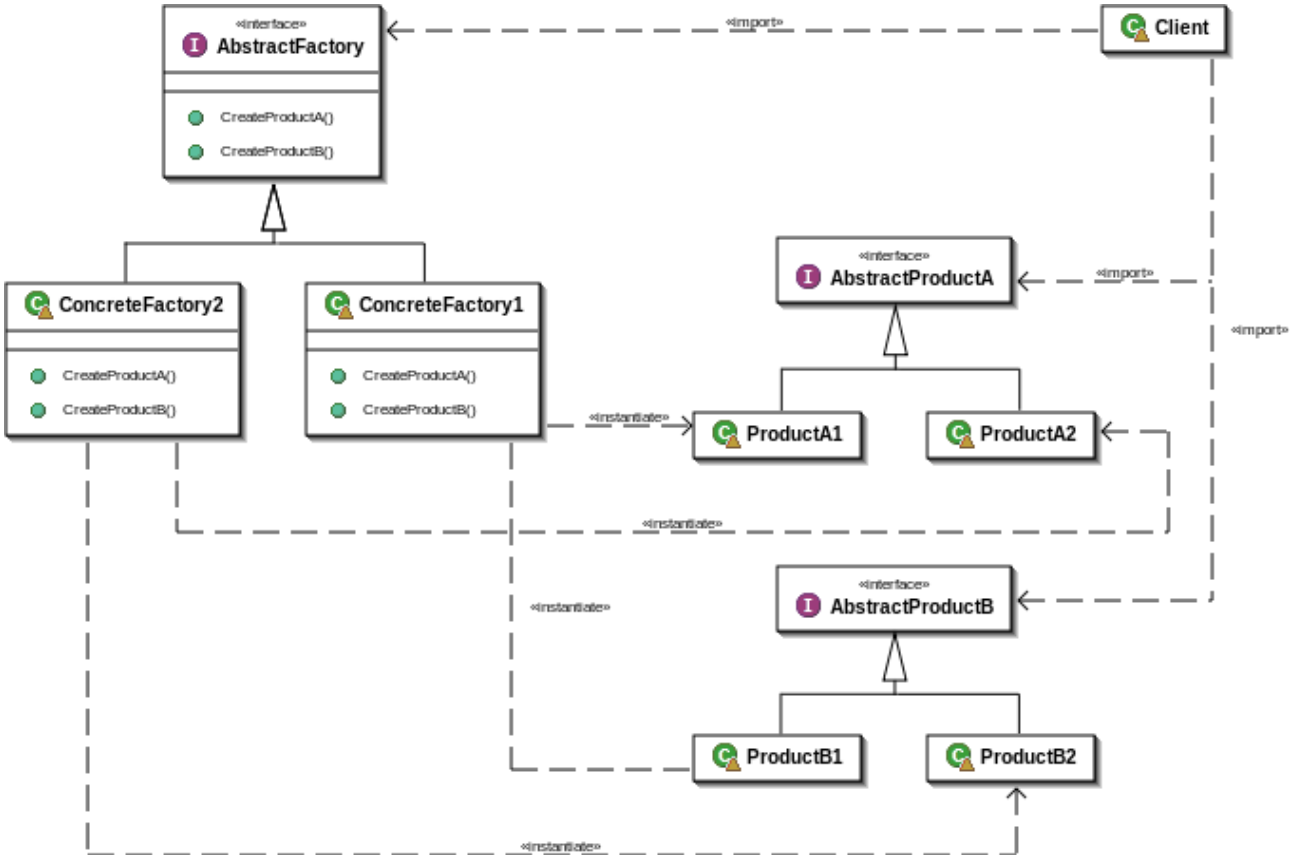


Ilustración 12. Diagrama del Patrón Abstract Factory. [5]

Como podemos observar en la imagen anterior "ProductA1", "ProductA2", "ProductB1" y "ProductB2" heredan de la factoría concreta. Dicha factoría viene a representar en nuestro proyecto a las distintas familias ("Guerra", "Estrategia" y "Relajación"). Estas familias han sido creadas, entre otras cosas, para asociar diferentes objetos a cada una de ellas.

Por ejemplo, la escena de Guerra contará como tótem principal una pistola, mientras que las otras dos familias no. Es decir que cada familia tiene vinculado un producto y las características del mismo.

Nuestro proyecto permite que el usuario modifique el mapa y, por consiguiente, los objetos o productos con los que se puede encontrar el personaje. Para conseguir esto se desarrolló una funcionalidad que, de forma dinámica, crea nuevos objetos dentro de la interfaz gráfica del videojuego, teniendo en cuenta el tipo de familia con la que se trabaje.

Ilustración 13. Representación de objetos dinámicos. En naranja los objetos creados dinámicamente.

Para poder realizarlo, la clase "ProductoB1", por ejemplo, ha de heredar de la clase "MonoBehaviour" (clase base de Unity). El problema es que C# no permite la multiherencia, es decir "ProductB1" no podrá heredar de "ConcreteFactory1" y de "MonoBehaviour". Para solventarlo "ConcreteFactory1" deberá ser una interfaz, solucionando los problemas de pérdida de información que puedan surgir de convertir una clase abstracta en una interfaz.

Capítulo 4. Carga de datos

4.1 Introducción

Este capítulo describe el sistema de importación y exportación de datos que hemos desarrollado para permitir la importación y exportación de los elementos del entorno: los escenarios y personajes del videojuego.

Para llevar a cabo este almacenamiento en la memoria del dispositivo (Smartphone, Tablet o Computadora) se ha utilizado un proceso de serialización de algunos de los objetos de nuestro proyecto.

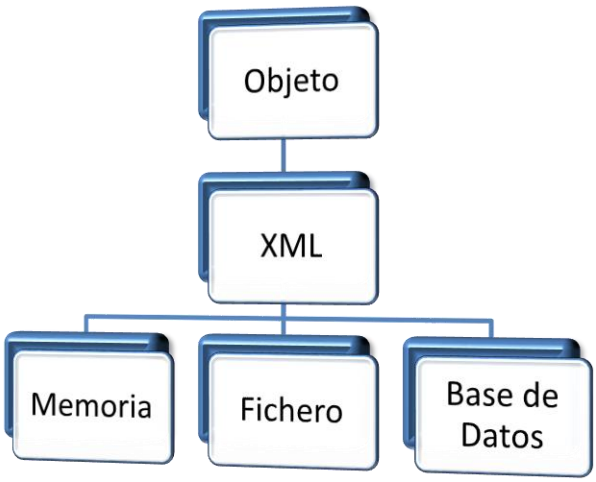


Ilustración 14. Proceso de serialización

El proceso de serialización que se ha desarrollado, permite almacenar el estado de un mapa/personaje y volver a crearlo cuando sea necesario.

En este proyecto, se ha optado por almacenar los objetos necesarios, para modificar el mapa y el personaje, en ficheros xml. Esta decisión posibilitaría, al usuario del videojuego, la modificación de estos ficheros.

```
<?xml version="1.0" encoding="UTF-8"?>
- <MapaInterpreter xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    <Region_num>3</Region_num>
    <Region>20</Region>
    <Act_scene>2</Act_scene>
  - <List_scenes>
    - <Scene>
        <Nombre>War Scene</Nombre>
        <Id_scene>0</Id_scene>
      </Scene>
    - <Scene>
        <Nombre>Relax Scene</Nombre>
        <Id_scene>0</Id_scene>
      </Scene>
    - <Scene>
        <Nombre>Agility Scene</Nombre>
        <Id_scene>0</Id_scene>
      </Scene>
  </List_scenes>
  - <List_components>
    - <BasicObjectSceneAbstract xsi:type="Door">
        <scene1>0</scene1>
        <scene2>1</scene2>
        <region>1</region>
      </BasicObjectSceneAbstract>
    - <BasicObjectSceneAbstract xsi:type="Door">
        <scene1>0</scene1>
        <scene2>1</scene2>
        <region>1</region>
      </BasicObjectSceneAbstract>
```

Ilustración 15. Ejemplo de fichero exportado

En la imagen anterior podemos observar un fichero que contiene la información de un escenario complejo con sus numerosos objetos. Como se puede apreciar, modificarlo no resulta una tarea tediosa y crea un aliciente para que el usuario se anime a crear/modificar nuevos escenarios desde cualquier dispositivo.

Para realizar la importación de escenarios o personajes, se realiza un proceso de deserialización. Consiste en recuperar el estado de uno o varios objetos y además debe completar y crear nuevos. Todo esto a partir de la información obtenida de los ficheros mostrados previamente.

Capítulo 5. Diseño del videojuego

5.1 Prototipo inicial

Este apartado está dedicado a contar en qué fase se encontraba el videojuego inicial, empezado en la asignatura "Diseño arquitectónicos y patrones". Este prototipo fue desarrollado en Java utilizando la librería AndEngine.

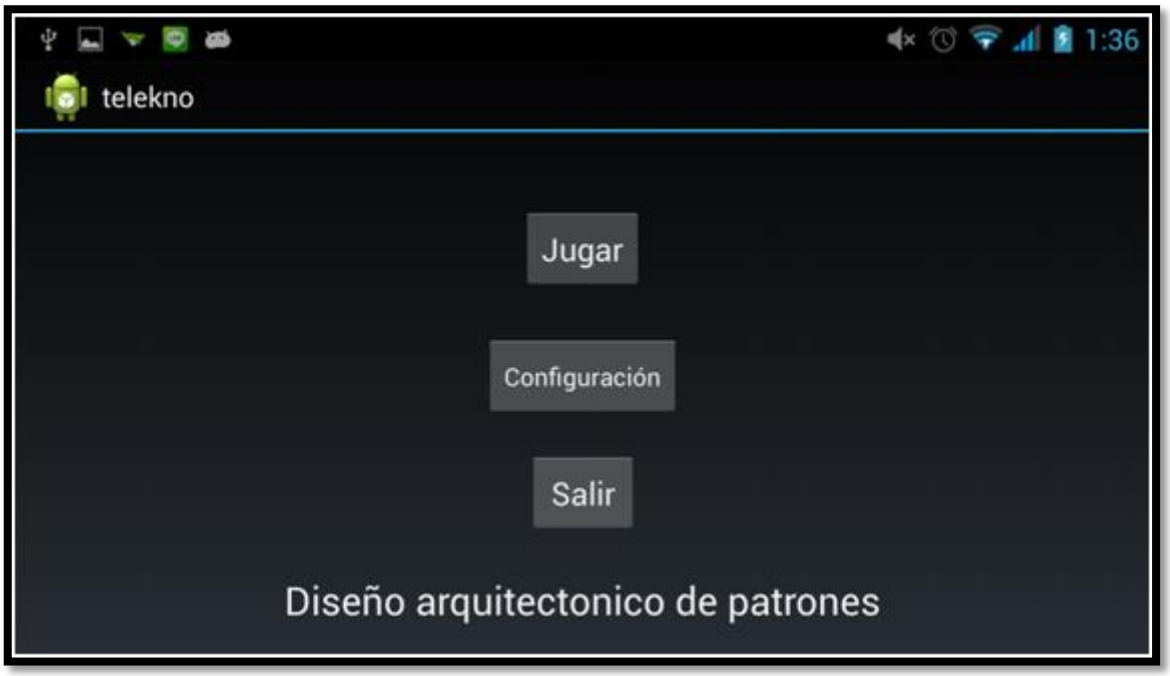


Ilustración 16. Primer Prototipo. Menú de inicio

El proyecto tenía el propósito de desarrollar un videojuego. En este el usuario debía poder modificar, inicialmente, el destino de algunas de las cuatro puertas que disponían las habitaciones que el usuario hubiera creado.

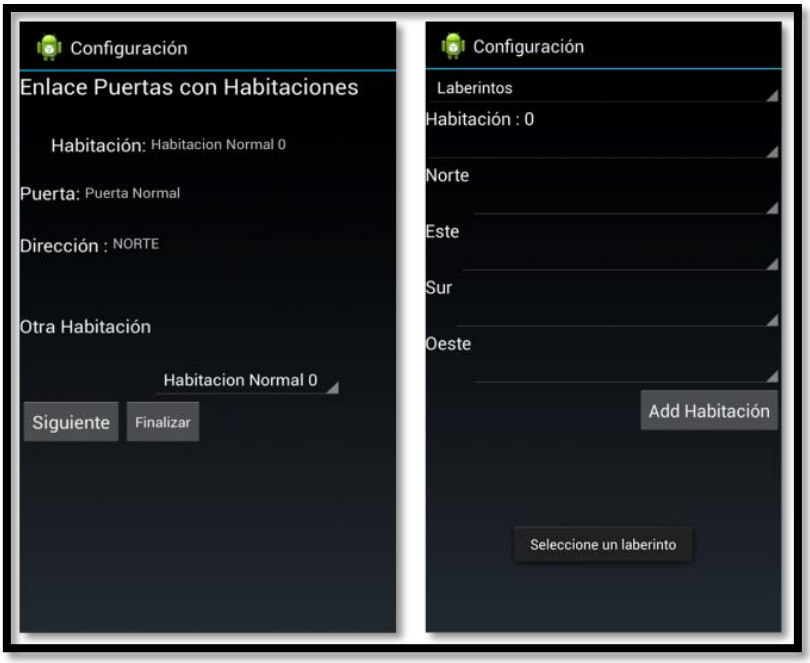


Ilustración 17. Primer prototipo. Configuración del escenario

Estas imágenes ilustran cómo el usuario configuraba las puertas de una determinada habitación, para posteriormente jugar al videojuego.

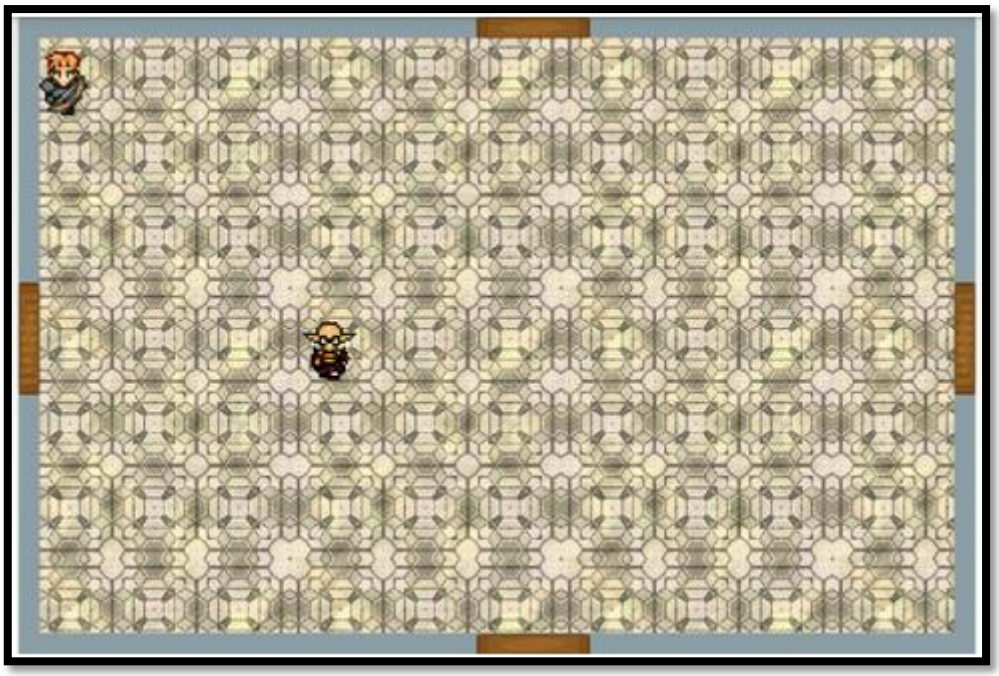


Ilustración 18. Primer prototipo. Habitación de relajación

Este primer prototipo era un juego en 2D que contaba con diferentes habitaciones. Dentro de la habitación encontrábamos un enemigo que, si decidíamos luchar contra él, el juego nos trasladaba a un minijuego.

Ilustración 19. Primer prototipo. Minijuego de suerte

Este minijuego consistía en que el usuario debía de alcanzar al enemigo (algo así como una carrera). El usuario tenía 6 lanzamientos de dados, si con la suma de esos puntos conseguía alcanzar al enemigo, entonces se pasaba el minijuego.

Además intervenía un factor suerte, que se iba incrementando a medida que el usuario ganaba batallas, y decrementando en caso contrario.

5.2 Prototipo Actual

Para el diseño del prototipo actual, se han tenido en cuenta consideraciones de jugabilidad, creación de una interfaz sencilla para la modificación de escenas, diseño llamativo, dispositivos en los que se instalará el videojuego, edad de los consumidores, etc.

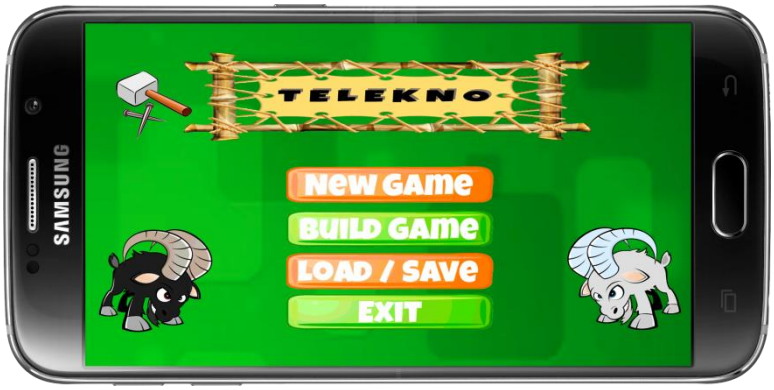


Ilustración 20. Prototipo actual. Menú de inicio. Smartphone

La imagen anterior muestra el menú de inicio en un Smartphone. A partir de ahora, las imágenes se mostrarán en una Tablet.

5.2.1 Menú

Ilustración 21. Prototipo actual. Menú de inicio. Tablet

El menú de inicio consta de cuatro opciones:

- **New Game:** Al iniciar el juego, esta opción de menú arrancará un videojuego por defecto. De esta manera se le da la posibilidad al usuario de no configurar nada y empezar a jugar directamente.
- **Build Game:** El usuario entrará en el modo construcción, lo que le permitirá configurar escenas, añadir objetos, modificar el comportamiento de los mismos, eliminar objetos, etc. Este apartado se explicará más detalladamente en los siguientes párrafos
- **Load/Save:** Esta opción permite que el usuario importe y exporte personajes y escenarios.
- **Exit**: Cerrará el videojuego.

5.2.2 Build Game

Ilustración 22. Prototipo Actual. Build Game

El usuario tiene la posibilidad de ir añadiendo nuevos escenarios al videojuego. De esta manera tiene control total sobre el escenario por donde se mueve su personaje.

Una vez le demos a “Next”, nos trasladaremos a la siguiente vista.

Ilustración 23. Prototipo Actual. Build Game II

Como podemos apreciar, en esta vista el usuario puede añadir objetos a la escena, modificarlos, situarlos en una región del mapa y eliminarlos.

Esta pantalla será accesible desde el transcurso del videojuego, permitiendo una modificación dinámica de los objetos y sus características.

Después de haber realizado la configuración, el usuario podrá jugar con el mapa que ha creado.

5.2.3 Modo Juego

Ilustración 24. Prototipo Actual. Videojuego

El usuario podrá moverse por el mapa utilizando el joystick izquierdo. Además tendrá la posibilidad de girar la cámara mediante el joystick derecho, obteniendo una visión de 360 grados del escenario.

Mediante el botón de Pause, que podemos ver en la imagen, el juego se detendrá, ofreciendo las siguientes opciones:

- **"Save Game"**: Guardar juego
- **"Main Menu"**: Volver al menú del juego, pudiendo guardarlo.
- **"Quit Game"**: Cerrar juego, sin realizar guardado del juego
- **"Close"**: Cerrar el menú de pausa.

Si queremos volver al menú de configuración de las distintas escenas, debemos utilizar el botón del martillo y los clavos. De esta manera entraremos en el modo de construcción.

También podemos apreciar, en la esquina superior izquierda, las habilidades del personaje, "shot" y "strategy" asociadas a haber ganado o perdido los minijuegos que presentaremos a continuación.

I.5.2.3.1 Minijuegos

Cada escenario dispone de un tótem especial que le transladará al minijuego asociado a ellos.

I.5.2.3.1.1 Minijuego de Guerra

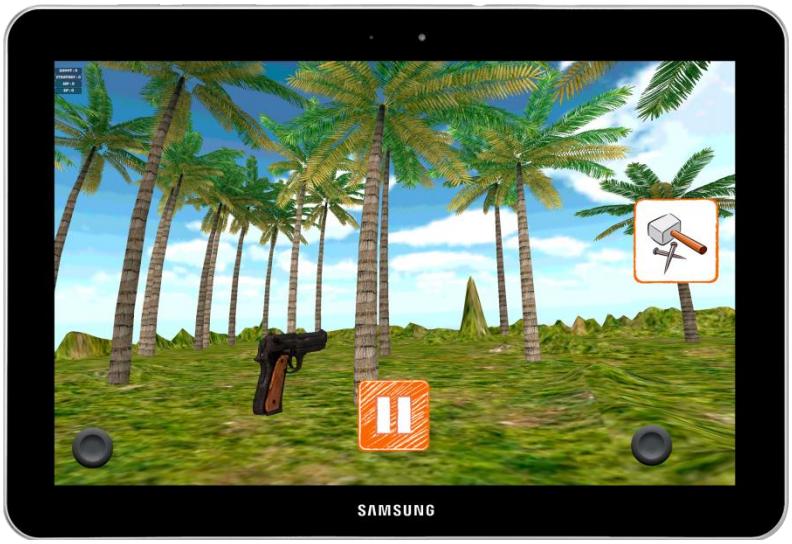


Ilustración 25. Prototipo Actual. Tótem de escenario de Guerra

En la imagen podemos apreciar el tótem del escenario de guerra. Cuando una pistola es atravesada por el personaje, el usuario deberá enfrentar un minijuego de tiro.

Ilustración 26. Prototipo Actual. Minijuego de Guerra

En este minijuego, el usuario deberá disparar a los misiles que intentan destruir el avión. Los misiles se generan en parejas y aparecerán aleatoriamente desde la derecha de la pantalla en dirección al avión. Cuando se alcance la puntuación mínima y pasarse el juego o bien cuando el avión es destruido, las clases que necesiten saber si el usuario se ha pasado el minijuego serán avisadas (patrón observer).

I.5.2.3.1.2 Minijuego de Estrategia

Ilustración 27. Prototipo Actual. Totem de escenario de Estrategia

El tótem del escenario de estrategia lo representa un libro, que transladará a un minijuego donde la habilidad y la estrategia son determinantes.

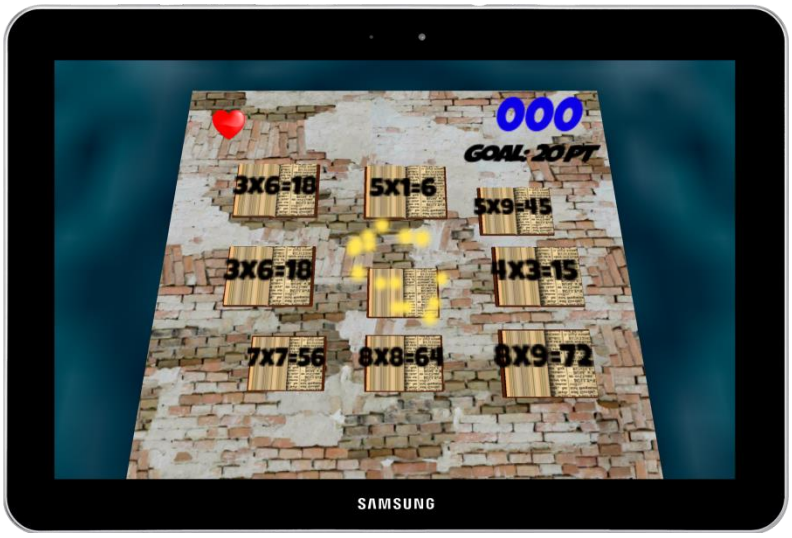


Ilustración 28. Prototipo Actual. Minijuego de Estrategia

El segundo minijuego es una adaptación del clásico juego de los topos. En esta versión el usuario tendrá que ir seleccionando aquellas operaciones que son correctas hasta alcanzar el objetivo prefijado por el juego.

El usuario contará con una serie de vidas que irá gastando a medida que este seleccione una operación incorrecta.

Una vez el juego finalice, el contexto del personaje se actualizará, y éste volverá a la escena de partida, en caso de perder o a la escena siguiente, si el usuario supera el minijuego.

5.2.4 Load /Save

Ilustración 29. Prototipo Actual. Vista de Exportación e Importación

Esta vista le presenta al usuario una manera fácil de exportar e importar el mapa y el personaje. De esta manera el usuario podrá diseñar un mapa y enviárselo a otro usuario mediante un archivo de texto.

Además, podrá ir guardando el progreso de su personaje o distintos perfiles de personaje con el fin de que en un determinado momento del videojuego le interese cargar alguno de ellos.

Esta opción resulta bastante útil puesto que la inteligencia artificial del videojuego modifica atributos de los objetos que componen las escenas, teniendo en cuenta las características que ha ido ganando el personaje.

Capítulo 6. Pruebas

6.1 Introducción

Este capítulo describe las pruebas realizadas al software implementado, para así demostrar la calidad del mismo. La cantidad de pruebas están restringidas al número de horas destinadas para ello (60 horas).

6.1.1 Pruebas de Compatibilidad

Estas pruebas están destinadas a verificar a qué nivel funciona la plataforma en distintos sistemas operativos.

I.6.1.1.1 Android

Modelos en los que se han realizado las pruebas:

- Tablet Samsung Tab S
    - Versión Android: 6.0
- BQ Aquarious
    - Versión Android: 5.0
- LG G4
    - Versión Android: 6.0
- LG G3
    - Versión Android: 5.0

Prueba	Resultados
Carga de texturas	Si
La IA responde correctamente	Si
El Videojuego es capaz de comunicarse con el dispositivo estudiado	Si
Errores de gameplay	No
Memorias de archivos corruptos	No
Funcionamiento de los Controles	Si
Carga de Textos	Si

I.6.1.1.2 Windows PC

Modelos en los que se han realizado las pruebas:

- Samsung RV511
    - SO: Windows 8.1
- Acer aspire E5-571G-56T1
    - SO: Windows 10 education

Prueba	Resultados
Carga de texturas	Si
La IA responde correctamente	Si

El Videojuego es capaz de comunicarse con el dispositivo estudiado	No, problemas al guardar la información de los mapas de los usuarios
Errores de gameplay	No
Memorias de archivos corruptos	No
Funcionamiento de los Controles	Si
Carga de Textos	Si

6.1.2 Sonar Qube

SonarQube es una plataforma para evaluar la calidad del software. Integra herramientas de medición de la calidad de código: CPD (detección de código duplicado), findbugs, PMD (análisis de métricas de calidad) y checkstyle.

El contenido del fichero de configuración sonar-project.properties es el siguiente:

```
# Required metadata
sonar.projectKey=PFG
sonar.projectName=PFG
sonar.projectVersion=1.0

# Comma-separated paths to directories with sources (required)
sonar.sources=Assets

# Language
sonar.language=cs

# Encoding of the source files
sonar.sourceEncoding=UTF-8

sonar.scm.disabled=true
```

Ilustración 30. Fichero sonar-project.properties

Podemos apreciar que el proyecto consta de 12,079 líneas de código repartidas en 180 archivos.

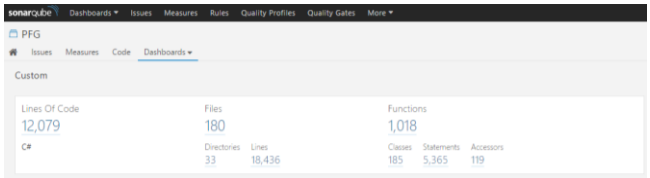


Ilustración 31. Visión general del proyecto con la herramienta Sonar

A continuación se mostrará una captura de los resultados obtenidos en Sonar al acabar la codificación del proyecto:

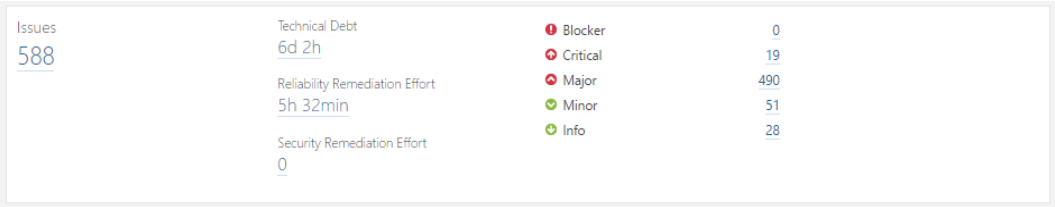


Ilustración 32. Captura de Sonar antes de corregir las evidencias

Como se puede observar en la imagen, existe una deuda técnica de 6 días y 2 horas, 19 evidencias críticas (poca probabilidad de impacto en el comportamiento del software o fallos de seguridad), 490 graves (defecto de calidad que puede tener un alto impacto en la productividad), 51 menores (leve impacto en la productividad) y 28 de información. La deuda técnica hace referencia al sobreesfuerzo que conllevaría mantener este proyecto si la calidad del código no fuera mejorado.

Se fueron resolviendo dichos evidencias, conforme su importancia. Una vez corregidos se obtuvieron los siguientes resultados:

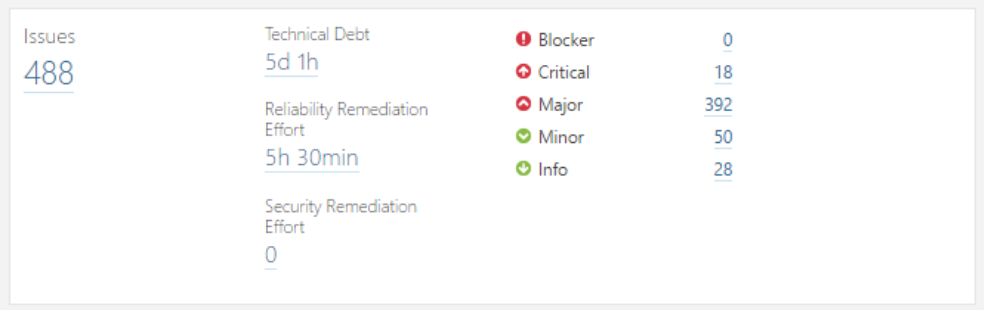


Ilustración 33. Captura de Sonar una vez corregidas algunas evidencias críticas y graves

Como se puede apreciar, la deuda técnica se ha reducido a 5 día y 1 hora, y el número de evidencias se ha visto reducida a 488.

Seguidamente, se ha procedido a solventar otras evidencias graves, este es el resultado obtenido al evaluar el código después de hacerlo:

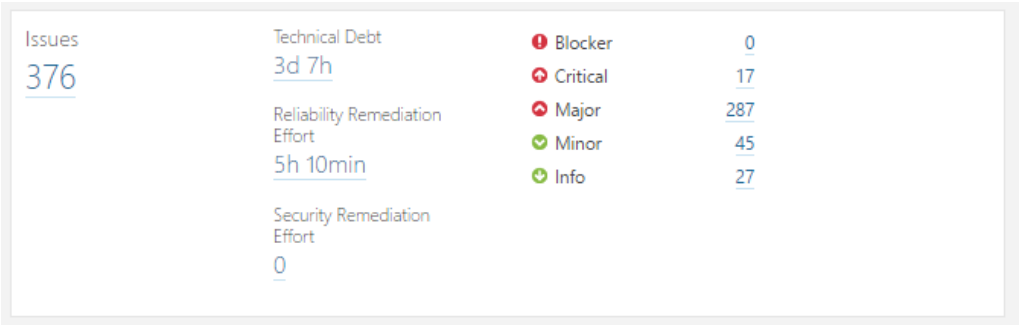


Ilustración 34. Captura de Sonar después de corregir ciertas evidencias graves

De la imagen anterior se puede deducir que, gracias a la utilización de la herramienta SonarQube, se ha mejorado la calidad del código habiendo conseguido disminuir la deuda técnica del proyecto en casi 3 días.

Capítulo 7. Guía de desarrollo

7.1 Introducción

A continuación describiremos los pasos a seguir para incorporar un nuevo videojuego dentro de la plataforma desarrollada.

1. Creación de una nueva familia:
   a. Creación de un Constructor (opcional) que deberá extender de "Builder Scene"

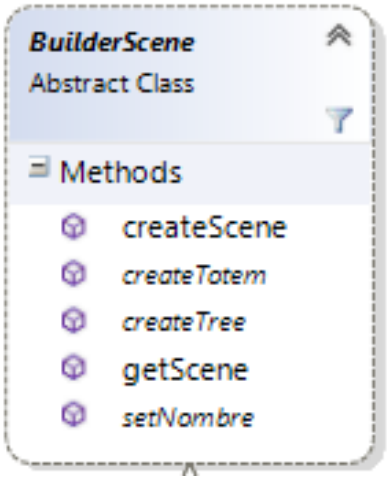


Ilustración 35. Modelo de la clase abstracta BuilderScene

2. Desarrollo de un Totem (opcional):
3. Se da la posibilidad al usuario de agregar un objeto representativo para acceder a su videojuego desde él.
4. Clase que extienda de "AlarmMiniGame" (obligatorio), proveyendo al videojuego del usuario de una clase que notificará el estado del personaje. Los observadores de la plataforma, actuarán de acorde si el minijuego se ha superado o no.
5. Clase que implemente la interfaz "IGameFinish" (opcional) para ser notificado del resultado obtenido al finalizar un minijuego. De esta manera el minijuego podrá adaptar su dificultad al nivel del jugador.

Se advierte al desarrollador que quiere incluir su videojuego dentro de la plataforma que es posible que su juego sufra alguna modificación del diseño que este tenía. Esto se debe a algunas incompatibilidades que pueden existir desde el punto de vista gráfico entre la plataforma y el videojuego que se quiere incorporar.

Capítulo 8. Conclusiones y líneas futuras

Como resultado del desarrollo realizado, podemos concluir que se ha desarrollado un modelo ampliable basado en patrones de diseño. Se han cumplido los requisitos que se planteaban para este proyecto, consiguiendo dar un paso adelante en la creación de un videojuego de mundo abierto que anima y facilita la participación del usuario.

El proyecto nos ha llevado a trabajar con el motor de desarrollo de videojuegos Unity. Motor que desde el punto de gráfico, nos permite llevar a cabo escenarios complejos y escenas sofisticadas.

El reto que planteó fusionar la "filosofía Unity" con una estructura basada en patrones de diseño fue superado con éxito. Apoyándonos en el lenguaje de desarrollo C# se han diseñado objetos ampliables y flexibles que permiten la incorporación de nuevas partes o funcionalidades.

Hemos concedido al usuario la potencialidad de construirse un escenario, añadirle las partes que desee y poder compartirlo con otros usuarios con un simple fichero.

Como trabajos futuros se plantea:

- Crear una plataforma web donde los usuarios suban minijuegos, para que estos sean incluidos en el juego.
- Creación de una aplicación de escritorio donde un usuario pueda crear nuevos mapas, así como añadir nuevos objetos a este, generando un fichero que nuestra estructura pueda procesar.
- Añadir la importación y exportación de escenarios como una parte de gamificación del juego.
- Mejoras gráficas en los distintos escenarios.
- Incorporación de algoritmos genéticos para la importación y exportación de escenarios y personajes.
- Posibilidad de incorporar nuevos tótem desde el móvil.

Capítulo 9.
Summary and Conclusions

At the end of this project, we can conclude that it has developed an expandable model based on design patterns. The requirements of the project has been achieved, getting an open world game that inspire and make easier the user participation.

The project has been developed with Unity. This development environment allows us to create complex scenarios and sophisticated scenes.

The challenge of join the “Unity philosophy” with the structure based on design patterns was overcome successfully. Working with the programing language C# it were developed expandable and flexible objects that allows the incorporation of new functionalities.

We grant the user the potentiality of construct his own scenario, adding the parts that he desire, and share it with other users with a simple file.

As future works it considers:

- Create a web platform where the users can upload their mini-games.
- Create a desktop application to allow users to create new maps, as well as add new objects.
- Add the importation and exportation of scenarios as a part of the game.
- Graphics improvement.
- Add genetic algorithms for the import and export method of scenarios and characters.
- Possibility of adding news totems from mobile.

Capítulo 10. Presupuesto

10.1 Horas dedicadas

Se estima que se han dedicado 300 horas a la elaboración del trabajo de fin de grado. Dichas horas corresponden con las exigidas en la guía docente del Trabajo de Fin de Grado. Este conjunto de horas puede ser desglosada de la siguiente manera:

Tareas	Horas	Precio por Hora	Presupuesto
Análisis	30	50	1.500 €
Diseño	70	50	3.500 €
Programación	140	36	5.040 €
Pruebas	60	32	1.920 €

Tabla 1. Presupuesto. Horas dedicadas.

10.2 Herramientas utilizadas

Precios de licencias de software empleado para diseñar y elaborar el proyecto:

Software	Descripción	Licencia	Presupuesto
Unity 4 (Free edition)	Entorno de desarrollo para estudiantes	GRATUITA	0 €
Photoshop	Editor de gráficos rasterizados	50	235,92 € / Anual
MonoDevelop v4.0.1	Entorno de desarrollo	GRATUITA	0 €

Tabla 2. Presupuesto. Herramientas utilizadas

10.3 Equipo necesario

Se han utilizado los siguientes dispositivos:

Dispositivo	Descripción	Presupuesto
Samsung Galaxy Tab S	Tableta	420 €
Aquarius 5 HD	Smartphone	210 €
LG G4	Smartphone	620 €

Tabla 3. Presupuesto. Equipo necesario

10.4 Total

El presupuesto total de fin de grado ascendería a:

Total
13.445,92 €

Tabla 4. Presupuesto. Total

Debido a que se ya se disponía de los dispositivos móviles y el software de diseño utilizado, el presupuesto real invertido no ha ascendido a dicha cantidad.

Bibliografía

[1] AwfulMedia. (s.f.). *youtube*. Obtenido de https://www.youtube.com/watch?v=qwuPiaFU37w

[2] Discom, F. (s.f.). *youtube*. Obtenido de https://www.youtube.com/watch?v=yamAzcT6Fo8

[3] Erich Gamma, R. H. (2008). *Patrones de Diseño.* Pearson.

[4] *http://www.rpgmakerweb.com/*. (s.f.).

[5] *https://en.wikipedia.org/wiki/Abstract_factory_pattern*. (s.f.).

[6] Juarez, M. (s.f.). *Factory Method*. Obtenido de http://migranitodejava.blogspot.com.es/search/label/Factory%20Method

[7] Juarez, M. (s.f.). *Patrón Abstract Factory*. Obtenido de http://migranitodejava.blogspot.com.es/search/label/Abstract%20Factory

[8] Juarez, M. (s.f.). *Patrón Builder*. Obtenido de http://migranitodejava.blogspot.com.es/search/label/Builder

[9] Juarez, M. (s.f.). *Patrón Decorador*. Obtenido de http://migranitodejava.blogspot.com.es/2011/06/decorator.html

[10] Juarez, M. (s.f.). *Patrón Observer*. Obtenido de http://migranitodejava.blogspot.com.es/2011/06/observer.html

[11] Juarez, M. (s.f.). *Patrón Singleton*. Obtenido de http://migranitodejava.blogspot.com.es/2011/05/singleton.html

[12] *msdn.microsoft.com*. (s.f.). Obtenido de https://msdn.microsoft.com/es-es/magazine/dn759441.aspx

[13] Nodician. (s.f.). *youtube*. Obtenido de https://www.youtube.com/watch?v=VS1NgiqVUu8

[14] Nodician. (s.f.). *youtube*. Obtenido de https://www.youtube.com/watch?v=8KVUAVaG7Z4

[15] Nodician. (s.f.). *youtube*. Obtenido de https://www.youtube.com/watch?v=NodIh15W9pc

[16] Nodician. (s.f.). *youtube*. Obtenido de https://www.youtube.com/watch?v=bXkKxUKdWrU

[17] Oscuridad, M. V. (s.f.). *youtube*. Obtenido de https://www.youtube.com/watch?v=49fyAxDlkxE&x-yt-ts=1421914688&x-yt-cl=84503534

[18] Pastrana, O. L. (s.f.). *youtube*. Obtenido de https://www.youtube.com/watch?v=iujMmbcfD7I

[19] Quintana, B. (s.f.). *youtube*. Obtenido de https://www.youtube.com/watch?v=C0ZWoNvkjik

[20] *taringa*. (s.f.). Obtenido de http://www.taringa.net/posts/info/17634325/Mega-tutorial-de-unity-3D-Parte1.html

[21] *tf3dm*. (s.f.). Obtenido de http://tf3dm.com/

[22] Unity. (s.f.). *unity3d*. Obtenido de https://unity3d.com/es/learn/tutorials/modules

[23] *unity3d*. (s.f.). Obtenido de https://unity3d.com/es/learn/tutorials/modules/beginner/scripting/c-sharp-vs-javascript-syntax

[24] *www.blender.org*. (s.f.).

[25] *www.brick-force.com*. (s.f.).

[26] *www.far-cry.ubisoft.com*. (s.f.).

[27] *www.lionhead.com/games/black-white*. (s.f.).

[28] *www.minecraft.net*. (s.f.).

[29] *www.ogre3d.org*. (s.f.).