



UNIVERSITAT_{DE}
BARCELONA

Trabajo de final de grado

GRADO EN INGENIERÍA INFORMÁTICA

Facultat de Matemàtiques e Informàtica
Universitat de Barcelona

Refactorización de un Juego Serio en VR para Desarrollar el Pensamiento Computacional

Autor: Oscar Muñoz Soldado

Directoras:

Dra. Inmaculada Rodríguez Santiago

Dra. Anna Puig Puig

Realizado en:

Departamento de Matemàtiques e Informàtica

Barcelona, 10 de Junio de 2024

Abstract

This final degree project focuses on refactoring the VR serious game "Cooking Code" to enhance its modularity, maintainability, and portability across various VR headsets. "Cooking Code" uses the metaphor of cooking to teach programming concepts, leveraging the immersive capabilities of VR to improve learning outcomes. The project involves updating the Unity version, refactoring the legacy code, and incorporating new features that utilize the latest technological advancements. The result is a codebase designed to follow software design principles, facilitating future development and enhancements.

Resumen

Este proyecto de fin de grado se centra en la refactorización del juego serio en VR "Cooking Code" para mejorar su modularidad, mantenibilidad y portabilidad en varios visores de VR. "Cooking Code" utiliza la metáfora de la cocina para enseñar conceptos de programación, aprovechando las capacidades inmersivas de la VR para mejorar los resultados del aprendizaje. Este proyecto implica actualizar la versión de Unity, refactorizar el código legacy e incorporar nuevas funcionalidades que aprovechen los últimos avances tecnológicos. El resultado es una base de código diseñada para seguir principios de diseño de software, facilitando el desarrollo y mejoras futuras.

Resum

Aquest projecte de fi de grau se centra en la refactorització del joc seriós en VR "Cooking Code" per millorar la seva modularitat, mantenibilitat i portabilitat en diversos visors de VR. "Cooking Code" utilitza la metàfora de la cuina per ensenyar conceptes de programació, aprofitant les capacitats immersives de la VR per millorar els resultats de l'aprenentatge. El projecte implica actualitzar la versió de Unity, refactoritzar el codi legacy i incorporar noves funcionalitats que aprofitin els últims avenços tecnològics. El resultat és una base de codi dissenyada per seguir els principis de disseny de programari, facilitant el desenvolupament i les millores futures.

Índice

1	Introducción	4
1.1	Contexto y Motivación	4
1.1.1	Contexto	4
1.1.2	Motivación	4
1.2	Objetivo principal	5
1.3	Objetivos específicos	5
1.4	Planificación	5
1.5	Estructura de la memoria	5
2	Antecedentes	7
2.1	Serious Games para el pensamiento computacional	7
2.1.1	Job simulator	7
2.1.2	The Farmer Was Replaced	8
2.1.3	Lightbot	8
2.2	Proyectos precedentes	10
2.2.1	Modo cocinero	10
2.2.2	Modo camarero	10
3	Análisis	13
3.1	Requisitos funcionales	13
3.1.1	Modo de Juego Chef	15
3.1.2	Modo de Juego Waiter	16
3.2	Requisitos no funcionales	18
3.3	Análisis de Usuarios	18
3.3.1	Usuarios sin experiencia en programación	18
3.3.2	Usuarios con experiencia en programación	19
4	Diseño	20
4.1	Diseño del juego	20
4.1.1	Modo cocinero	20
4.1.2	Modo camarero	23
4.2	Diseño del software	25
4.2.1	Arquitectura del código Legacy	25
4.2.2	Diagrama de clases del código Legacy	25
4.2.3	Exploración del código Legacy	29
4.2.4	Arquitectura del sistema	31
4.2.5	Backend: Modelo de datos	32
4.3	Refactorización y Patrones de Diseño utilizados	32
4.3.1	Bloques de Programación	32

4.3.2	Tipos de bloque	33
4.3.3	Ingredientes	35
4.3.4	Pizarra de Programación	35
4.3.5	Generación aleatoria de pedidos	37
4.3.6	Generación de pedidos en el modo cocinero	37
4.3.7	Generación de pedidos en el modo camarero	40
4.3.8	Implementación en el código	41
4.3.9	Generación del feedback	41
4.3.10	Comunicación con la API	43
4.3.11	Mención al patrón Observer	44
4.4	Diagrama de Clases	45
4.4.1	Explicación de Clases Específicas	45
4.4.2	Otras Clases Específicas de los Modos de Juego	46
5	Implementación y resultados	53
5.1	Entorno de desarrollo	53
5.1.1	Actualización de unity	53
5.1.2	Herramientas y librerías utilizadas	53
5.2	Desarrollo del modo cocinero	55
5.2.1	Funcionalidades Refactorizadas e Incorporadas	55
5.2.2	Funcionalidades No Incorporadas	55
5.3	Desarrollo del modo camarero	56
5.3.1	Funcionalidades Refactorizadas e Incorporadas	56
5.3.2	Funcionalidades No Incorporadas	56
6	Conclusiones	58
6.1	Resumen y contribuciones al proyecto	58
6.2	Trabajo Futuro	59
A	Apéndice 1 - Instalación del proyecto	60
A.1	Instalación del proyecto de Unity	60
A.2	Instalación del backend	61
A.3	Despliegue de la aplicación web en Azure y vinculación con GitHub	63
B	Apéndice 2 - Implementación del Sistema de Logging	64
B.1	Formato de los Logs	64
B.2	Nombre del Fichero	64
B.3	Ruta del Archivo de Log	64
B.4	Creación del Logger	65
B.5	Eventos Registrados	65

Chapter 1

Introducción

En el presente trabajo de fin de grado, se aborda la refactorización de un juego serio en realidad virtual (VR) denominado "Cooking Code". Este juego utiliza la metáfora de la cocina para enseñar conceptos de programación, aprovechando las capacidades inmersivas de la VR para mejorar los resultados de aprendizaje. La necesidad de mantener el código actualizado y aprovechar las nuevas funcionalidades de las últimas versiones de Unity y librerías de VR, motiva este proyecto. A continuación, se detallan el contexto, la motivación, los objetivos y la planificación del proyecto.

1.1 Contexto y Motivación

1.1.1 Contexto

Los serious games, o juegos serios, son diseñados con un propósito educativo más allá del entretenimiento. En particular, los juegos serios para el desarrollo del pensamiento computacional son herramientas efectivas para enseñar conceptos de programación y resolución de problemas de manera práctica y divertida. En particular, aquellos juegos que utilizan metáforas facilitan la comprensión de conceptos abstractos al compararlos con elementos familiares. Cooking Code, el juego en el que se basa este proyecto, utiliza la metáfora de cocinero para enseñar conceptos de programación. De hecho, la Realidad Virtual (VR) proporciona realismo e inmersividad, mejorando la retención del conocimiento y ofreciendo una experiencia de aprendizaje más efectiva.

1.1.2 Motivación

Un desafío en el desarrollo con Unity es la necesidad de mantener el código actualizado. Unity y las librerías de realidad virtual cambian frecuentemente, y mantener un código legacy¹ puede limitar el aprovechamiento de nuevas funcionalidades. Esto motiva la necesidad de refactoring [7] del código y una actualización de versión de Unity.

Aunque requiere tiempo y esfuerzo, el refactoring ofrece beneficios significativos en términos de claridad, mantenibilidad, flexibilidad y rendimiento. Refactorizar implica

¹El código legacy se refiere a código fuente antiguo que presenta varios problemas debido a su antigüedad, uso de versiones obsoletas de tecnologías y bibliotecas, y una estructura deficiente derivada del desarrollo en caliente. En el ámbito de los videojuegos, esto significa que se han ido añadiendo funcionalidades que no se tenían pensadas inicialmente, lo que resulta en un código difícil de mantener y expandir.

mejorar la estructura del código sin alterar su funcionalidad, resultando en un sistema más adaptable a futuros cambios. Al realizar el refactoring, el proyecto se prepara para integrarse con nuevas tecnologías, manteniendo su relevancia y eficiencia.

1.2 Objetivo principal

El objetivo principal de este proyecto es extender y hacer más portable el juego. Esto implicará hacer un refactoring completo del código del juego VR Cooking Code, también conocido como “remake”. Esto consiste en analizar las funcionalidades presentes en el código Legacy y replicarlas de forma modular y ampliable en la nueva versión.

Otro punto crucial es actualizar Unity de la versión 2019.4.22f1 a la versión 2022.3.26f1, manteniendo así el software al día con las librerías y tecnologías actuales. En esta nueva versión tenemos Open XR, una API que nos permite programar videojuegos en VR para que sean compatibles con todo tipo de dispositivos o visores.

1.3 Objetivos específicos

Los objetivos específicos en los que se desglosan el objetivo general son los siguientes:

- Aprender a usar Unity VR y C#.
- Analizar y depurar las principales funcionalidades del proyecto Legacy.
 - Instalar el proyecto en la versión Unity 2019.4.22f1.
 - Corregir errores en la versión Legacy.
 - Instalar la versión actual de Unity 2022.3.26f1.
 - Instalar tecnologías de VR y configurar el proyecto base.
- Analizar, rediseñar el código usando técnicas de refactoring.
- Validar el juego
 - Implementar componentes básicos.
 - Implementar lógica del juego.

1.4 Planificación

En la figura 1.1 se muestra la planificación del proyecto que se ha realizado por Sprints una metodología SCRUM. El diagrama está dividido en las diferentes objetivos, pudiendo ver que áreas se le ha dedicado un mayor tiempo.

1.5 Estructura de la memoria

En el siguiente capítulo, se presentarán ejemplos de otros juegos serios, analizando sus ventajas y desventajas. También se comentarán brevemente los proyectos precedentes a este trabajo. Posteriormente, se explicará el diseño de la nueva versión del juego,

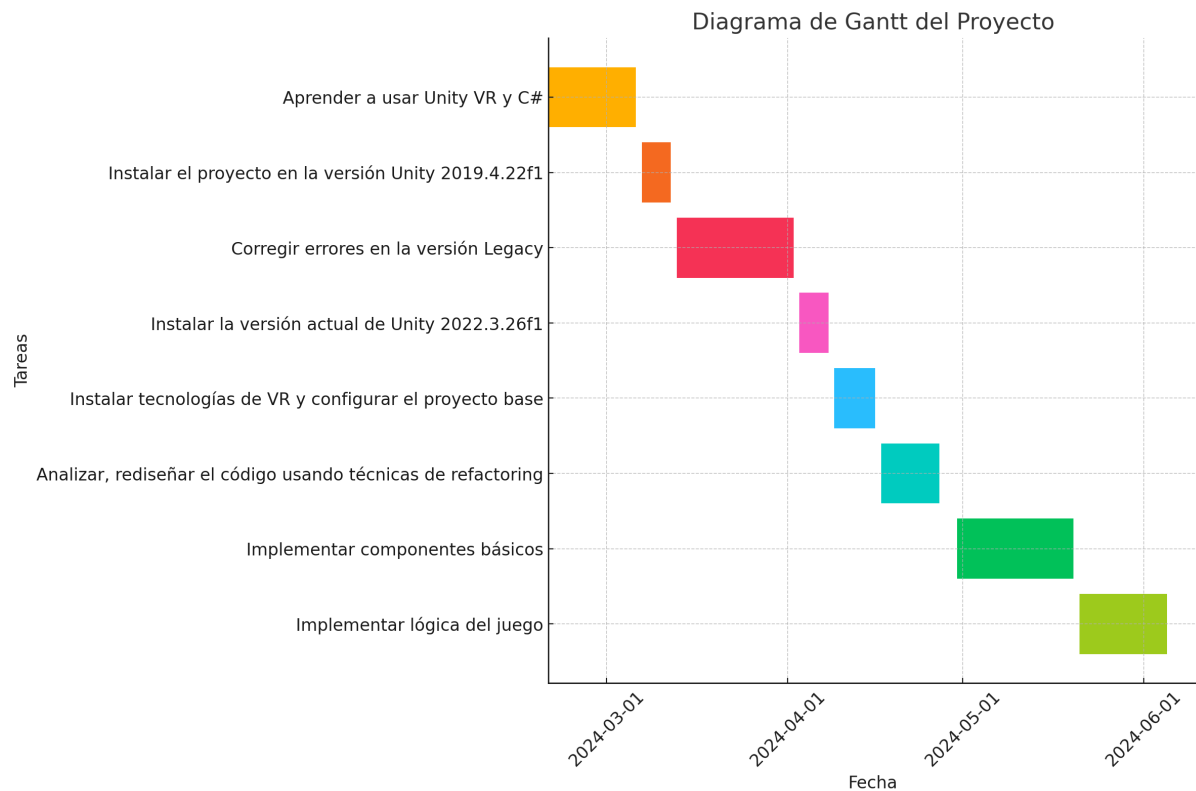


Figure 1.1: Diagrama de Gantt (Fuente propia).

destacando las ventajas del refactoring. Se explorará el código Legacy para identificar las áreas que requieren cambios o rediseños. Luego, se presentará el nuevo diseño de los componentes y los patrones utilizados.

Con el diseño ya realizado, se detallará la implementación de los nuevos componentes, incluyendo las tecnologías y herramientas empleadas. Finalmente, se mostrará la nueva versión del juego y se presentarán las conclusiones.

Chapter 2

Antecedentes

Este capítulo proporciona una comprensión fundamental de los serious games y su aplicación en el desarrollo del pensamiento computacional. Además, se explorarán proyectos anteriores relevantes que sirven de base para la refactorización y extensión del juego VR Cooking Code. Se analizarán sus características, ventajas y limitaciones, lo cual ayudará a contextualizar.

Este análisis de antecedentes permitirá apreciar las mejoras y cambios implementados en la nueva versión del juego, subrayando la importancia de estos proyectos previos como punto de partida para el desarrollo actual.

2.1 Serious Games para el pensamiento computacional

En esta sección se explora el papel de los serious games en el desarrollo del pensamiento computacional. Estos juegos, diseñados con un propósito educativo, proporcionan una plataforma interactiva para aprender habilidades clave en programación y resolución de problemas [1].

2.1.1 Job simulator

Job Simulator (ver figura 2.1) es un juego de realidad virtual ambientado en un futuro donde los robots han reemplazado a los humanos en sus trabajos. Los jugadores simulan diversos entornos laborales, como una oficina, una tienda o una cocina. Este juego es especialmente útil para el desarrollo del pensamiento computacional, ya que los jugadores deben seguir instrucciones detalladas, adaptarse a cambios y solucionar problemas inesperados. Estas actividades fomentan habilidades de secuenciación lógica, resolución de problemas y pensamiento crítico, que son fundamentales en la programación.

Comparación

En nuestro caso, profundizamos más en lo que es programación en sí. Utilizamos bloques de programación para generar y recibir pedidos, forzando así la comprensión de estos conceptos de manera directa y aplicable.



Figure 2.1: Job Simulator (Fuente: Steam).

2.1.2 The Farmer Was Replaced

The Farmer Was Replaced (ver figura 2.2) es un serious game que simula la gestión de una granja automatizada. Los jugadores asumen el rol de un granjero que programa y controla robots para realizar tareas agrícolas. Este juego introduce a los jugadores a conceptos de automatización y programación, ya que deben crear algoritmos eficientes para optimizar la producción. Enfrentarse a desafíos que requieren un pensamiento lógico y estructurado ayuda a desarrollar habilidades cruciales para la programación, como la planificación, la depuración y la mejora continua de los procesos.

Comparación

The Farmer Was Replaced es un ejemplo mucho más complejo que nuestro caso. Está orientado a un público más familiarizado con la programación, ya que se trabaja directamente con código y se permite realizar tareas más avanzadas.

2.1.3 Lightbot

Lightbot (ver figura 2.3) es un juego de puzzle diseñado para enseñar conceptos básicos de programación a través de la resolución de problemas. Los jugadores deben guiar a un robot a través de una serie de laberintos utilizando comandos simples, como mover, saltar y encender luces. Este juego introduce a los jugadores a los conceptos de secuenciación, loops y condiciones.

Comparación

Lightbot comparte la simplicidad y la enseñanza de conceptos básicos de programación con nuestro juego. Ambos utilizan bloques de programación para resolver tareas, pero mientras que Lightbot lo hace en un entorno 2D, nuestro juego lo lleva a la realidad virtual,

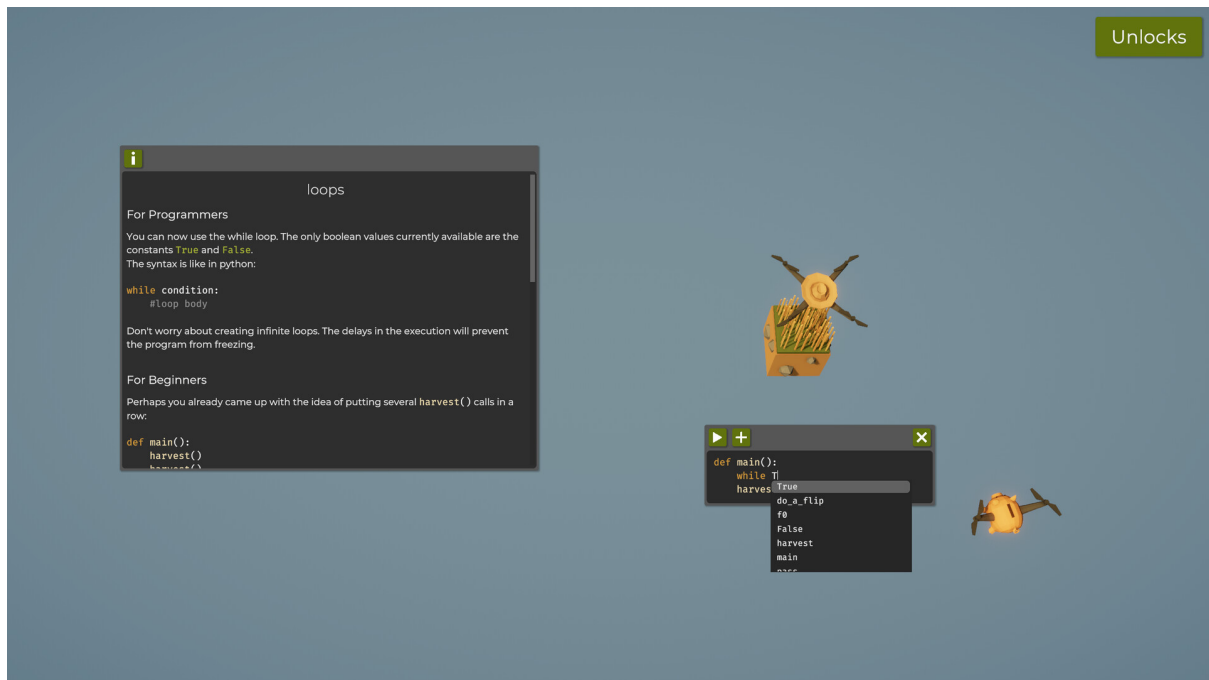


Figure 2.2: The Farmer Was Replaced (Fuente: Steam).

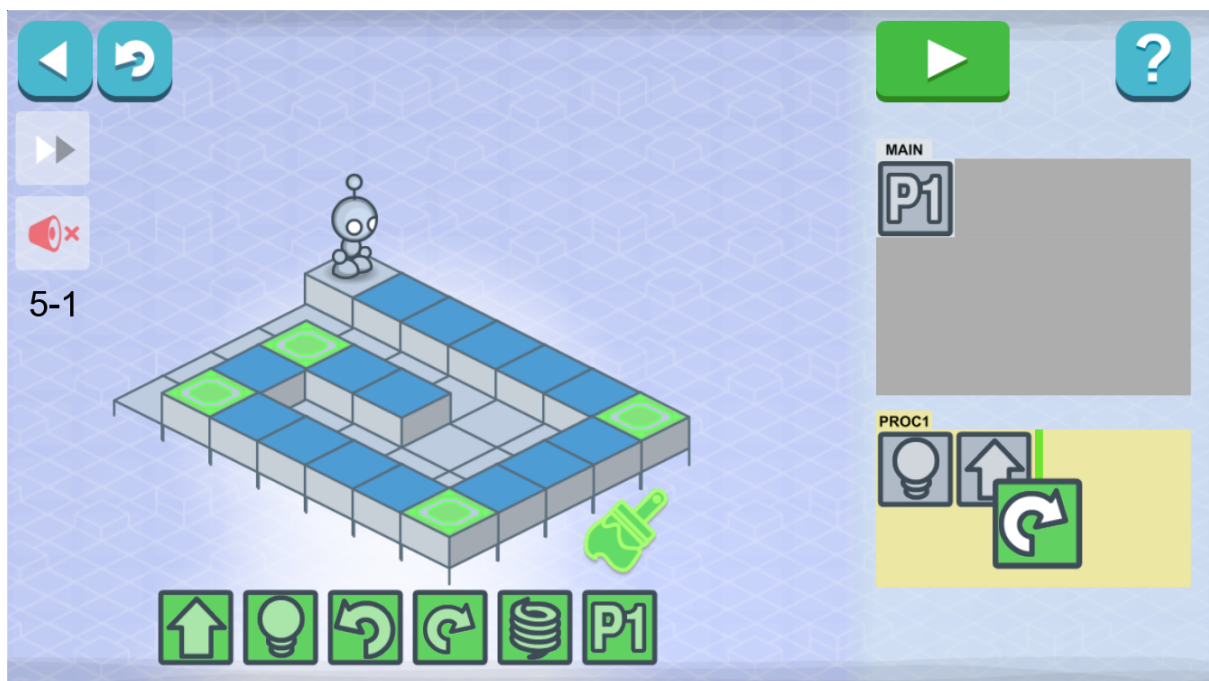


Figure 2.3: Lightbot (Fuente: Lightbot).

permitiendo una interacción más directa e inmersiva. Esta diferencia de entorno permite una experiencia más realista y práctica en nuestro juego, potenciando el aprendizaje a través de la inmersión total en un entorno tridimensional.

2.2 Proyectos precedentes

Cooking Code es un juego serio en realidad virtual (VR) desarrollado en la Facultad de Matemáticas e Informática de la Universidad de Barcelona a través de diversos trabajos de final de grado. Desde su inicio en 2020, el juego ha evolucionado significativamente con la incorporación de nuevos modos de juego y tecnologías. En 2020, se introdujo el primer modo de juego, el modo cocinero. En 2021, se añadió el modo camarero, seguido en 2022 por el desarrollo de un agente conversacional para evaluar la experiencia de juego. Finalmente, en 2023, se integró este agente conversacional al juego y se perfeccionaron conceptos y mecánicas.

En los proyectos anteriores, se han desarrollado dos modos de juego principales que sirven para enseñar a los jugadores la interpretación y escritura de pseudocódigo en un entorno de cocina virtual. Ambos modos están diseñados para ser tanto educativos como entretenidos, y se estructuran en jornadas laborales que actúan como niveles en el juego. Al finalizar cada jornada, los jugadores reciben un resumen de su desempeño, incluyendo los puntos de experiencia obtenidos, que son cruciales para desbloquear nuevos niveles y mecánicas más avanzadas, como el uso de sentencias condicionales en el pseudocódigo y la incorporación de nuevos ingredientes.

2.2.1 Modo cocinero

En este modo de juego, el jugador asume el rol de un cocinero en una cocina virtual que podemos ver en la figura 2.4. El objetivo principal es leer pseudocódigo para producir hamburguesas. Este modo de juego está diseñado para ayudar a los jugadores a familiarizarse con la interpretación del pseudocódigo y a entender cómo las instrucciones secuenciales se traducen en acciones específicas dentro del juego. El jugador debe seguir las instrucciones paso a paso para ensamblar los ingredientes correctamente y crear las hamburguesas solicitadas. Los pasos específicos que debe seguir el jugador son:

- Entender el pseudocódigo que recibe: Interpretar las instrucciones escritas en pseudocódigo para saber qué ingredientes y procesos son necesarios.
- Cocinar la carne: Seguir las instrucciones para cocinar la carne al punto indicado en el pseudocódigo.
- Montar la hamburguesa: Ensamblar los ingredientes en el orden correcto, siguiendo las instrucciones secuenciales proporcionadas.
- Enviar la hamburguesa: Una vez que la hamburguesa esté correctamente montada, enviarla al cliente virtual para completar el pedido.

2.2.2 Modo camarero

En el modo camarero (ver figura 2.5), el jugador asume el rol de un camarero en un restaurante virtual. Este modo de juego es más avanzado y requiere que el jugador escriba pseudocódigo para atender las peticiones de los clientes del restaurante, gracias a los bloques que podemos ver en la figura 2.6. El proceso implica varias etapas:

- Recogida del Pedido: El jugador debe moverse por el restaurante para recoger los pedidos de los clientes, que están expresados en lenguaje natural.



Figure 2.4: Código Legacy: Cocina (Fuente: Cooking Code[13]).



Figure 2.5: Código Legacy: Restaurante (Fuente: Waiter Code[12]).

- Traducción a Pseudocódigo: El camarero debe traducir el pedido de los clientes a pseudocódigo, ya que el cocinero solo entiende este tipo de instrucciones.
- Envío a Cocina: Una vez traducido, el pedido se envía a la cocina donde será preparado por el cocinero virtual.
- Entrega del Pedido: Finalmente, el jugador debe recoger el pedido preparado y entregarlo a la mesa correspondiente.

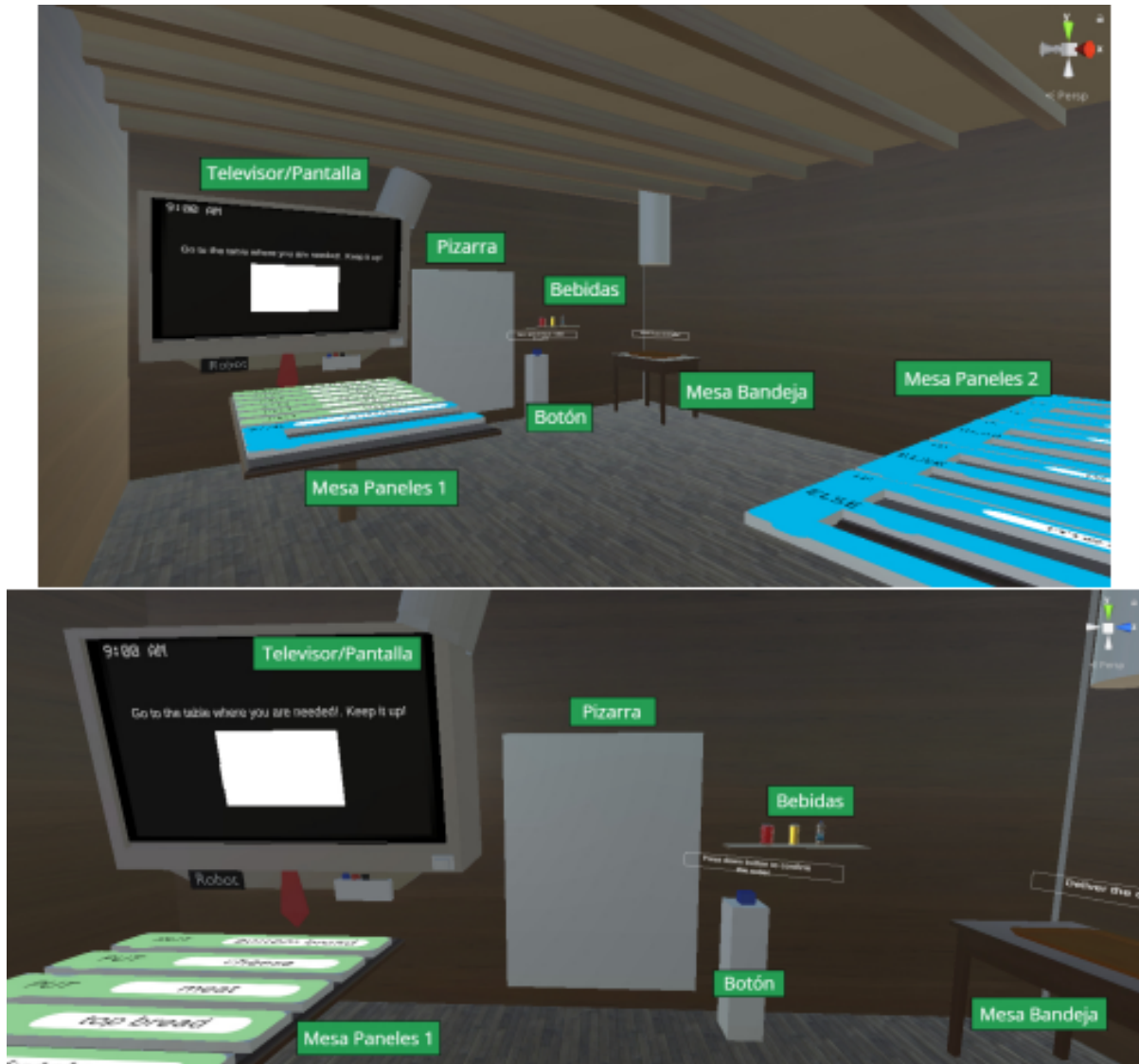


Figure 2.6: Código Legacy: Backstage del restaurante donde se preparan los pedidos (Fuente: Waiter Code[12]).

Chapter 3

Análisis

En este capítulo, se realiza un estudio detallado de los requisitos del proyecto para definir todas las funcionalidades y características del videojuego necesarias. Se analizan tanto los requisitos funcionales como los no funcionales, abarcando aspectos clave como el rendimiento, la usabilidad y la portabilidad. Además, se presenta un análisis del perfil de los usuarios a los que está dirigido el videojuego, dividiéndolos en subgrupos según su experiencia en programación para adaptar mejor el diseño y la funcionalidad del juego a sus necesidades específicas.

3.1 Requisitos funcionales

En esta sección se presentan los casos de uso modificados (ver Figura 3.1) para el proyecto de juego serio en realidad virtual. Para una versión más completa de los casos de uso, se pueden consultar los proyectos anteriores: WaiterCode [12], RestaurantCode[14] y CookingCode[13].

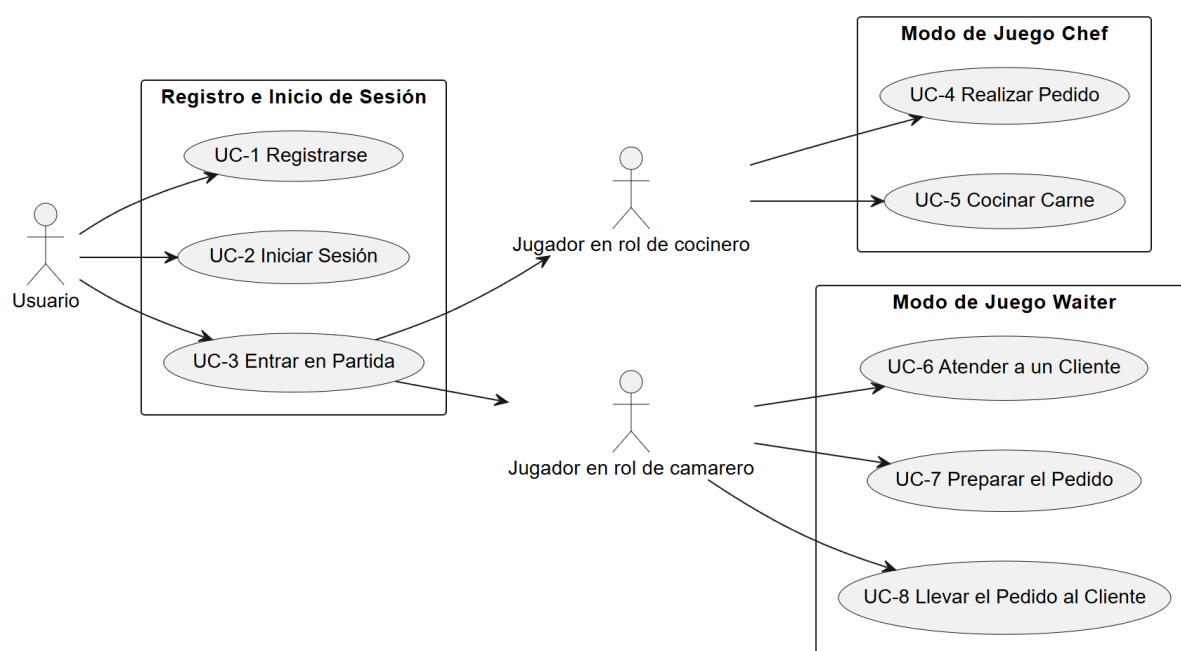


Figure 3.1: Casos de uso (Fuente propia).

Los principales modos de juego y casos de uso modificados son los siguientes:

- **Registro e inicio de sesión**
 - UC-1: Registrarse
 - UC-2: Iniciar Sesión
 - UC-3: Entrar en Partida
- **Modo de Juego Chef**
 - UC-4: Realizar pedido
 - UC-5: Cocinar carne
- **Modo de Juego Waiter**
 - UC-6: Atender a un cliente
 - UC-7: Preparar el pedido
 - UC-8: Llevar el pedido al cliente

Registro e inicio de sesión

UC-1 Registrarse

- **Actor Principal:** Usuario.
- **Flujo Principal:**
 1. El usuario selecciona la opción de registrarse.
 2. El usuario introduce un nombre de usuario y una contraseña.
 3. El sistema valida las credenciales.
 4. El sistema proporciona feedback positivo y registra al usuario.
- **Flujo Alternativo:**
 - En caso de que el nombre de usuario ya exista, el sistema proporciona feedback negativo.

UC-2 Iniciar Sesión

- **Actor Principal:** Usuario.
- **Flujo Principal:**
 1. El usuario selecciona la opción de iniciar sesión.
 2. El usuario introduce su nombre de usuario y contraseña.
 3. El sistema valida las credenciales.
 4. Si las credenciales son correctas, el usuario accede al menú principal y puede entrar en partida.
- **Flujo Alternativo:**
 - Si las credenciales son incorrectas, el sistema proporciona feedback negativo.

UC-3 Entrar en Partida

- **Precondiciones:** El usuario debe haber iniciado sesión.
- **Actor Principal:** Usuario.
- **Flujo Principal:**
 1. El usuario selecciona la opción de entrar en partida.
 2. El usuario elige entre el modo cocinero o el modo camarero.
 3. El usuario hace clic en el modo deseado y entra en la escena correspondiente.

3.1.1 Modo de Juego Chef

UC-4: Realizar Pedido

- **Actor Principal:** Jugador en el rol de cocinero.
- **Descripción:** El jugador debe preparar un pedido basado en instrucciones dadas en forma de pseudocódigo. El proceso implica seleccionar y organizar ingredientes de acuerdo a las instrucciones para completar el pedido correctamente.
- **Precondiciones:**
 - El jugador debe haber iniciado sesión y seleccionado el modo cocinero.
- **Flujo Principal:**
 1. El sistema genera y muestra un pedido en forma de pseudocódigo.
 2. El jugador interpreta el pseudocódigo para identificar los ingredientes y acciones necesarias.
 3. El jugador selecciona y prepara los ingredientes conforme a las instrucciones.
 4. El jugador organiza los ingredientes para montar el plato según las especificaciones del pseudocódigo.
 5. El jugador envía el pedido completado para su evaluación.
 6. El sistema evalúa el pedido.
 7. El sistema proporciona feedback sobre el pedido y otorga puntos de experiencia si el pedido es correcto.
- **Postcondiciones:**
 - El pedido ha sido completado y evaluado.
 - El jugador recibe feedback y, si es correcto, puntos de experiencia.
- **Flujo Alternativo:**
 - Si el pedido es incorrecto, el sistema notifica al jugador del error y genera un nuevo pedido.

UC-5: Cocinar Carne

- **Actor Principal:** Jugador en el rol de cocinero.
- **Descripción:** El jugador debe cocinar la carne adecuadamente como parte de un pedido, siguiendo indicaciones visuales y sonoras que reflejan el estado de cocción.
- **Precondiciones:**
 - El jugador debe haber iniciado sesión y estar en el proceso de completar un pedido que incluya carne.
- **Flujo Principal:**
 1. El jugador coloca la carne en la plancha.
 2. El sistema ofrece feedback visual (cambio de color de la carne, humo) y sonoro (sonido de cocinándose) para indicar el proceso de cocción.
 3. El jugador observa los cambios y decide cuándo la carne está adecuadamente cocida.
 4. El jugador retira la carne de la plancha y la utiliza en el pedido correspondiente.
- **Postcondiciones:**
 - La carne ha sido cocinada correctamente y está lista para ser usada en un plato.
- **Flujo Alternativo:**
 - Si la carne no se cocina suficientemente y queda cruda, al entregar el pedido, se reciben menos puntos de experiencia debido a la carne cruda.

3.1.2 Modo de Juego Waiter

UC-6: Atender a un Cliente

- **Actor:** Jugador en el rol de camarero.
- **Descripción:** El jugador debe interactuar con un cliente para recoger su pedido.
- **Precondiciones:**
 - El jugador debe haber iniciado sesión y seleccionado el modo camarero.
- **Flujo Principal:**
 1. El jugador se acerca a la mesa del cliente para obtener el pedido en lenguaje natural.
 2. El sistema recuerda la descripción del pedido.
- **Postcondiciones:**
 - El pedido del cliente ha sido correctamente atendido y está listo para ser programado.

UC-7: Preparar el Pedido

- **Actor:** Jugador en el rol de camarero.
- **Descripción:** El jugador debe programar el pedido en la pizarra de programación y generar el pedido.
- **Precondiciones:**
 - El pedido debe haber sido correctamente recibido y entendido del cliente.
- **Flujo Principal:**
 1. El jugador se dirige a la pizarra de programación.
 2. Utilizando bloques de programación, el jugador traduce el pedido a pseudocódigo.
 3. El jugador pulsa el botón para generar el pedido.
 4. El sistema proporciona feedback del pedido y si es correcto, genera el pedido encima de una bandeja.
- **Flujo Alternativo:**
 - Si el pedido es incorrecto al pulsar el botón, el sistema indica que es incorrecto y recuerda la descripción del pedido.
- **Postcondiciones:**
 - El pedido está correctamente encima de una bandeja.

UC-8: Llevar el Pedido al Cliente

- **Actor:** Jugador en el rol de camarero.
- **Descripción:** Una vez que el pedido está preparado, el jugador debe llevar el pedido al cliente usando la bandeja.
- **Precondiciones:**
 - El pedido debe haber sido preparado y listo para ser servido.
- **Flujo Principal:**
 1. El jugador recoge la bandeja preparada de la mesa.
 2. El jugador entrega la bandeja a la mesa del cliente.
 3. El sistema proporciona feedback correcto del pedido.
- **Flujo Alternativo:**
 - Al jugador se le cae el pedido de la bandeja.
 - El sistema proporciona feedback negativo y genera un nuevo pedido.
- **Postcondiciones:**
 - El cliente ha recibido su pedido correctamente.

3.2 Requisitos no funcionales

Los requisitos no funcionales son aquellos que definen criterios de calidad y restricciones del sistema, en lugar de describir funciones específicas. Estos requisitos se centran en aspectos como el rendimiento, la seguridad, la estabilidad, la extensibilidad, el mantenimiento y la portabilidad del software, asegurando que el producto final no solo cumpla con su propósito funcional, sino que también ofrezca una experiencia de usuario óptima y sostenible a largo plazo.

- **Rendimiento:** El videojuego debe ofrecer un buen rendimiento en visores VR como Oculus y Pico, garantizando una experiencia de juego fluida y sin interrupciones. Esto incluye mantener altas tasas de fotogramas por segundo (FPS) y minimizar la latencia para evitar mareos y mejorar la inmersión del usuario.
- **Seguridad:** La seguridad de los datos de los usuarios es una prioridad. Por lo tanto, se implementará cifrado de contraseñas en el backend para proteger la información personal y garantizar que los datos de acceso sean inaccesibles para terceros no autorizados.
- **Estabilidad:** El juego debe ser mínimamente robusto y estar libre de errores críticos que impidan jugar. Esto implica realizar pruebas exhaustivas y continuas para identificar y corregir bugs, asegurando que los jugadores puedan disfrutar de una experiencia sin interrupciones.
- **Extensibilidad y Mantenimiento:** El proyecto está diseñado para ser fácilmente extensible y mantenible a través de un proceso continuo de refactoring. Este enfoque permite la incorporación de nuevas funcionalidades y mejoras de manera eficiente, facilitando el crecimiento y la evolución del juego.
- **Portabilidad:** Gracias al refactoring y al uso de OpenXR, el videojuego será portable a múltiples visores VR como Oculus Quest y Pico. Esto asegura que el juego pueda ser disfrutado por una audiencia más amplia, independientemente del hardware VR que posean.

3.3 Análisis de Usuarios

El grupo de usuarios al que va enfocado este videojuego está formado por estudiantes de entre 12 y 18 años. Dentro de este rango de edad, se pueden identificar dos subgrupos: aquellos que no han tenido contacto con el mundo de la programación y aquellos que sí han tenido y ya tienen experiencia.

3.3.1 Usuarios sin experiencia en programación

Por un lado, los adolescentes que no tienen nada de experiencia con el mundo de la programación y nunca han mostrado interés por este mundo, ya sea porque no saben que existe o porque lo que se les ha mostrado no ha captado suficientemente su interés. No obstante, en general, tienen un conocimiento respetable respecto a la tecnología junto a la gran capacidad que tienen de adaptarse a esta. Comparten, por otra parte, una atracción muy grande a nuevas formas de entretenimiento, encontrándose abiertos a cualquier novedad.

3.3.2 Usuarios con experiencia en programación

Por otro lado, los que ya están iniciados en el mundo de la programación son jóvenes que están altamente interesados en el ámbito tecnológico, especialmente en el ámbito de los videojuegos y su desarrollo. Este grupo es propenso a mostrar un gran entusiasmo por juegos como Cooking Code. Además, a menudo suelen estar bien informados respecto a las últimas tecnologías, como podrían ser las gafas de Realidad Virtual. Han tenido algunas experiencias previas con la programación, ya sea por el simple conocimiento de su existencia o incluso por haber probado algunos ejercicios relacionados. Además, también es un grupo con bastante experiencia en el mundo de los videojuegos, lo que hace que tengan una gran facilidad para adaptarse rápidamente a nuevas mecánicas y formas de interactuar.

Chapter 4

Diseño

En este capítulo veremos el diseño general del juego, como funciona y mecánicas. Luego pasaremos a hablar del diseño de software, explorando el código Legacy para compararlo con la versión actual. Veremos toda la refactorización que se ha aplicado junto a los patrones de diseño utilizados. Finalmente tendremos el diagrama de clases.

4.1 Diseño del juego

Cooking Code es un "serious game" del género RPG (Role-Playing Game) dirigido a niños de entre 12 y 16 años. El objetivo principal es familiarizar a los jugadores con la programación y ofrecerles una forma divertida de aprender mediante la simulación de un restaurante en un entorno de realidad virtual (VR). Los participantes asumen roles específicos utilizando la metáfora de cocinar hamburguesas y servir pedidos. Durante el juego, realizarán acciones como agarrar o soltar objetos, tocar objetos para activar algún tipo de comportamiento, y moverse por la escena. Estas interacciones clave permiten experimentar de manera práctica y educativa un entorno de restaurante realista y dinámico.

Al iniciar el juego, los usuarios deben hacer login con una cuenta existente o crear una nueva. Una vez dentro, pueden elegir entre dos modos de juego: modo cocinero y modo camarero. En ambos modos de juego, el día empieza a las 9 y termina a las 17, con una duración de 5 minutos en tiempo real.

4.1.1 Modo cocinero

En este modo, los jugadores asumen el rol de un cocinero cuya tarea es preparar hamburguesas. Un robot en forma de pantalla muestra una secuencia de bloques de instrucciones que el jugador debe interpretar para completar los pedidos. La jugabilidad incluye la interacción manual con los controladores para añadir los ingredientes correspondientes a cada hamburguesa. Existen contenedores proveedores con un límite de ingredientes por pedido y suministros ilimitados como el ketchup y los platos. Podemos ver estos elementos en la figura 4.1 donde vemos la zona de trabajo del cocinero.

Al comienzo de cada día, se muestra un pedido que el jugador debe completar, como podemos ver en la figura 4.1. Si el pedido se completa con éxito, el jugador recibe puntos de experiencia. La cantidad de experiencia obtenida y el feedback recibido dependen de la precisión del pedido, con comentarios como "La carne está cruda" o "Hay ingredientes extra". Si el pedido es correcto el jugador gana 30 puntos de experiencia, si el pedido tiene



Figure 4.1: Zona de trabajo del cocinero diseñada en la nueva versión del juego(Fuente propia).

más o menos ingredientes, cuenta como incorrecto y proporciona 0 puntos de experiencia. Si el pedido es correcto pero tiene la carne cruda se le restan -15 puntos de experiencia. La generación del feedback se explica más a profundidad en la sección 4.3.9.

A medida que el jugador gana experiencia y sube de nivel, la dificultad aumenta. Esto se logra añadiendo nuevos bloques de programación o incrementando la cantidad de bloques. Los bloques de programación actuales¹ son:

- Bloque Put (ver figura 4.2): Añadir ingrediente.
- Bloque If (ver figura 4.3): Ejecutar una acción si se cumple una condición.
- Bloque For (ver figura 4.4): Repetir una acción un número determinado de veces.



Figure 4.2: Bloque Put (Fuente propia).

El bloque If evalúa sus condiciones basándose en la cantidad inicial de ingredientes disponibles al comenzar el pedido. Por ejemplo, si en el Bloque If vemos que pone que la carne sea mayor o igual a 2, tomará como valor de carne el que tenemos en el contenedor

¹Como se puede observar en las figuras, se han añadido desplegables para seleccionar diferentes opciones. Esto es un añadido en comparación con el código Legacy, donde no existían estas opciones.

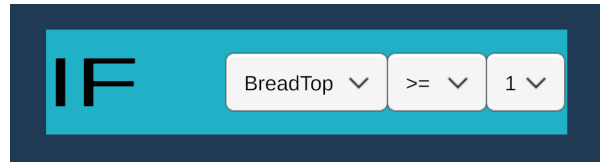


Figure 4.3: Bloque If (Fuente propia).



Figure 4.4: Bloque For (Fuente propia).

proveedor de carne. Un ingrediente clave es la carne, que debe cocinarse en cada pedido. La plancha proporciona soporte visual (la carne cambia de color y aparecen partículas de humo) y auditivo (sonido de cocción) para indicar el estado de cocción, como podemos ver en la figura 4.5.

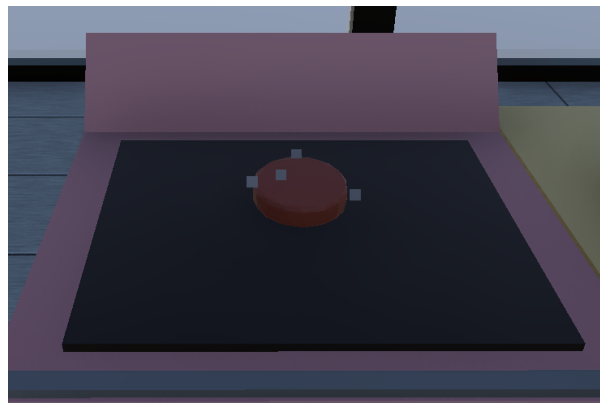


Figure 4.5: Carne cocinándose en la plancha (Fuente propia).

La progresión del juego se estructura en niveles, con cambios en la mecánica de los pedidos:

- Nivel 0 y 1: Aumenta la cantidad de ingredientes.
- Nivel 2: Se introducen pedidos con el bloque If.
- Nivel 3: Se introducen pedidos con el bloque For.
- Nivel 4: Se combinan pedidos con bloques If y For.

Al inicio de cada día, todos los ingredientes se reponen a cinco unidades. Después de cada pedido, se limpia la zona de trabajo y se restablecen los ingredientes agotados a cinco unidades.

La extensibilidad del modo cocinero se ve reforzada por la facilidad para añadir nuevos ingredientes, niveles y feedback. Actualmente, hay 4 ingredientes disponibles en el juego, pero se pueden agregar fácilmente más ingredientes a medida que se desarrollen nuevas

recetas o se introduzcan nuevos conceptos de pseudocódigo (Bloques de Programación). Esto permite una expansión continua del contenido y mantiene el interés del jugador.

Los niveles, que actualmente son 4, también se pueden extender al añadir nuevos ingredientes y recetas.

La generación de pedidos es aleatoria y proporciona pseudocódigo basado en bloques de programación mediante un algoritmo que se explicará en detalle en la sección 4.3.6. Podemos ver un ejemplo de este en la figura 4.1. Este enfoque asegura que los jugadores no encuentren tareas repetidas.

La interfaz donde se muestra el pseudocódigo es una pizarra de programación modificada para mostrar una lista de bloques de programación. Una explicación más a detalle sobre esta pizarra se encuentra en la sección 4.3.4.

En cuanto al feedback, la extensibilidad afecta a este elemento del juego de manera significativa. Al finalizar cada jornada laboral, el jugador recibe un resumen detallado de su desempeño. Este resumen incluye los puntos de experiencia obtenidos y los errores cometidos. Existe la posibilidad de añadir más tipos de feedback, ampliando la información que recibe el jugador.

Con esta estructura extensible, el modo cocinero no solo puede expandirse fácilmente con nuevos ingredientes y niveles, sino que también puede ofrecer un feedback más personalizado, mejorando la experiencia de aprendizaje de los jugadores.

4.1.2 Modo camarero

Este modo comparte muchos elementos con el modo cocinero, pero con adaptaciones específicas para simular un entorno de servicio al cliente en un restaurante. En este modo, los jugadores asumen el rol de un camarero. Al comenzar el día, deben atender a un cliente que les indicará mediante lenguaje natural la hamburguesa que desea con sus ingredientes, por ejemplo "Quiero una hamburguesa con carne, lechuga y queso.", como podemos ver en la figura 4.6. A continuación, el jugador debe dirigirse a una sala para montar la secuencia de instrucciones en una Pizarra de Programación. El jugador debe soltar bloques de programación para añadirlos, o agarrar los existentes para eliminarlos o moverlos.

Este modo comparte elementos con el modo cocinero, como los Bloques de Programación, la Pizarra de Programación y el Robot Pantalla. Sin embargo, introduce elementos nuevos y específicos del entorno de servicio, como la pizarra interactiva, que en el modo cocinero es estática y solo se lee, así como la bandeja para entregar los pedidos y las mesas donde los clientes esperan sus órdenes. Podemos ver los diferentes elementos en la figura 4.7

Para obtener los Bloques de Programación, el jugador dispone de una mesa con diferentes tipos de bloques para usar cuantas veces quiera, a diferencia de los ingredientes. Los Bloques de Programación contienen elementos interactivos, por ejemplo:

- El bloque Put (ver figura 4.2) contiene un desplegable para indicar el tipo de ingrediente.
- El bloque For (ver figura 4.4) tiene un desplegable para indicar la cantidad de iteraciones.

El jugador puede interactuar con los Bloques de Programación utilizando una "línea

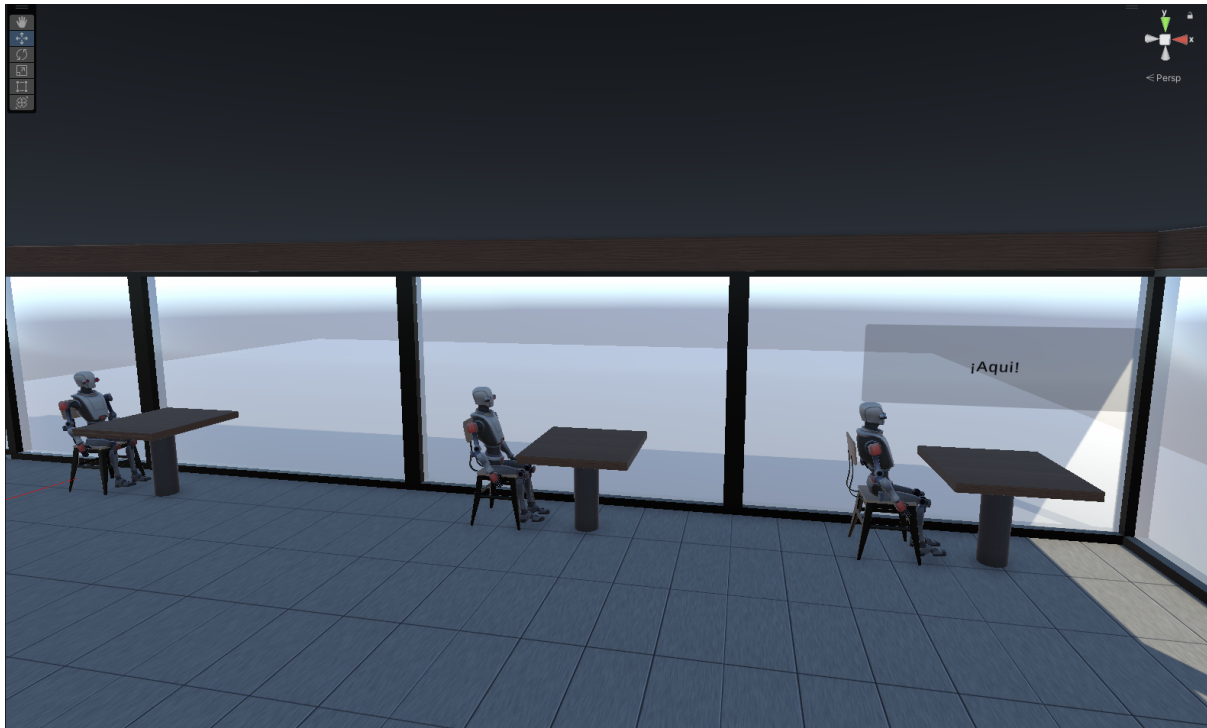


Figure 4.6: Zona de restaurante del camarero diseñada en la nueva versión del juego (Fuente propia).

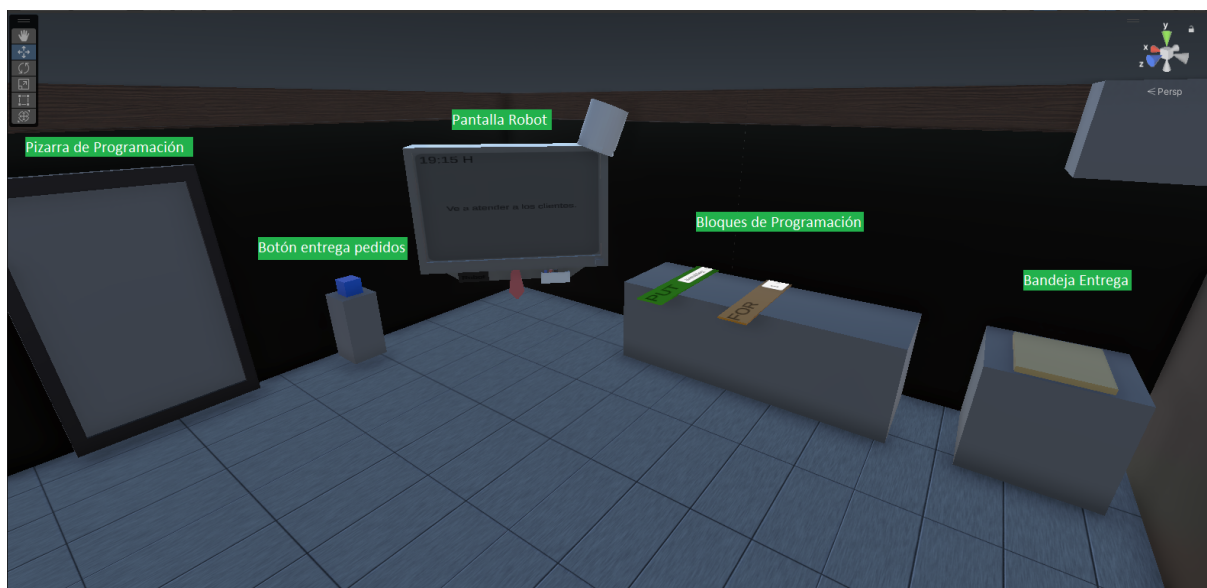


Figure 4.7: Zona de trabajo del camarero diseñada en la nueva versión del juego (Fuente propia).

de interacción”² que sale del controlador para manipularlos a distancia. Una vez que el pedido está ”programado”, el jugador puede pulsar un botón para que aparezca la hamburguesa montada en una bandeja. Si el pedido es incorrecto, la hamburguesa no se creará y se mostrará un feedback similar al del modo cocinero. Una vez que la hambur-

²En los juegos de realidad virtual, los controladores suelen emitir ”rayos láser” o ”líneas de interacción” que permiten al usuario apuntar y seleccionar objetos dentro del entorno virtual.

guesa está lista, el jugador debe entregarla a la mesa correspondiente, evitando que se caiga.

Después de entregar la hamburguesa, el jugador verá la experiencia ganada en el robot pantalla. Este robot también indica si hay un pedido pendiente, y muestra el pedido del cliente si está en curso, o la experiencia ganada si se ha completado el pedido.

En el modo camarero, la extensibilidad se destaca por la capacidad de añadir más mesas, montar hamburguesas con cualquier tipo de ingrediente, y la posibilidad de aplicar pedidos aleatorios gracias a poder traducir un pedido en forma de lista de ingredientes a un pedido en lenguaje natural que simule a un cliente pidiendo su hamburguesa.

4.2 Diseño del software

4.2.1 Arquitectura del código Legacy

El cambio más notable que veremos en la arquitectura de la nueva versión será la sustitución de la sección de Backend por otras tecnologías y endpoints. En el código Legacy se utiliza un Backend de Node.js con una base de datos MongoDB.

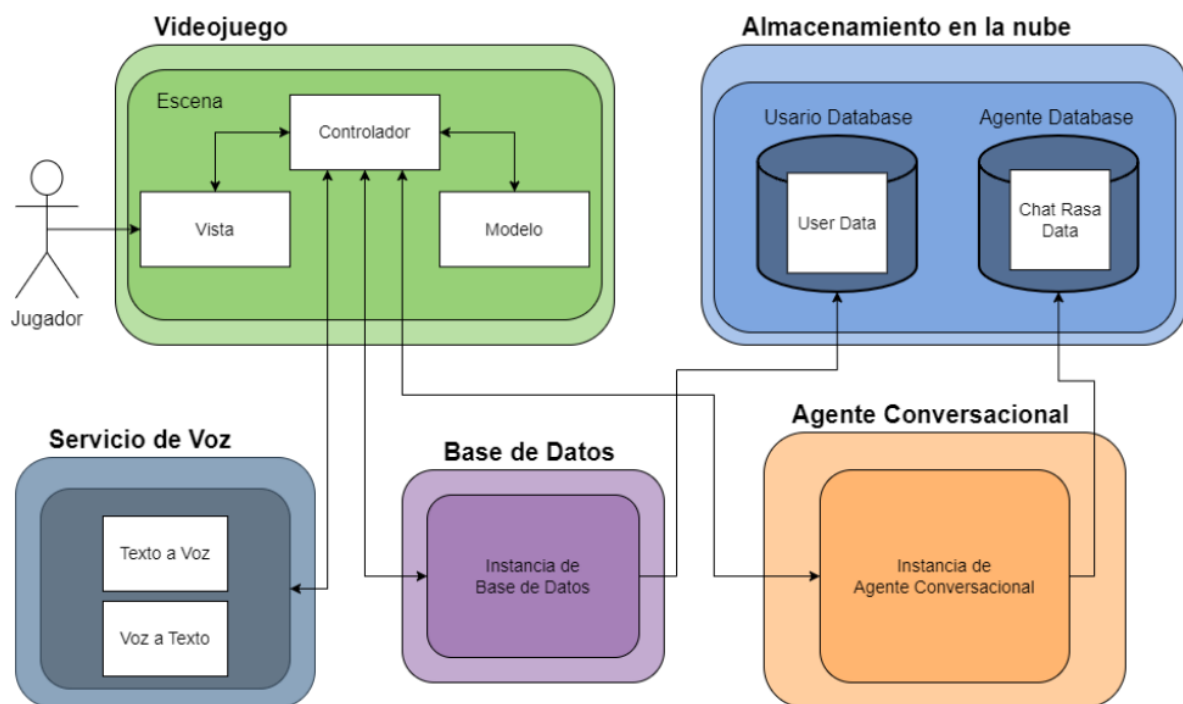


Figure 4.8: Arquitectura del código legacy (Fuente: Waiter Code[12]).

4.2.2 Diagrama de clases del código Legacy

Antes de empezar a explorar el código Legacy, comentaré las clases que hace falta refactorizar completamente (en color rojo) y las clases que se tienen que adaptar para tener una estructura modular compartida por ambos modos (en color amarillo). Aunque tengamos clases con el mismo nombre, cocinero y camarero no comparten ninguna clase. Son totalmente independientes y las clases tienen funciones diferentes.

Modo cocinero

Como podemos ver en la figura 4.9, al ser el código base de la primera versión no tenemos muchos problemas y generalmente tenemos un código que no está mal estructurado. Cabe recalcar que el código utiliza funciones de la SDK obsoleta de Oculus por lo cual hay clases y funciones que no veremos en la versión final. Como por ejemplo `grabberIndicator`.

La primera parte que tocaremos es la sección de Usuarios y información del nivel. Esta podemos verla en clases como `DataLevel`, `UserJSON`, `LogrosJSON` y `UserController`. Unificaremos esta parte junto a la del modo camarero, añadiendo la capa de permanencia de la base de datos.

La siguiente es la parte de creación de ingredientes. Podemos verla en la implementación de `FactoryIngredients`, `IngredientsCreacio` y sus derivados. Se creará una clase base `Supply`, una clase hija `Ingredient` y clases hijas para cada ingrediente junto una clase `IngredientFactory` como la que tenemos. Esto nos permite aplicar lógica a distintos niveles de jerarquía, por ejemplo, podemos tener una lógica general en `Supply` que sea heredada y especializada en `Ingredient` y sus clases hijas específicas.

Otra parte importante es la generación de pedidos, actualmente los pedidos que se generan son estáticos, es decir, no se generan aleatoriamente. Esto se comenta en la sección 4.2.3.

Finalmente, adaptaremos la generación del feedback de la clase `FeedbackGenerator` como se explica en la sección 4.3.9.

Modo camarero

Al ser más complejo, en este modo tenemos más problemas como podemos ver en la figura 4.10.

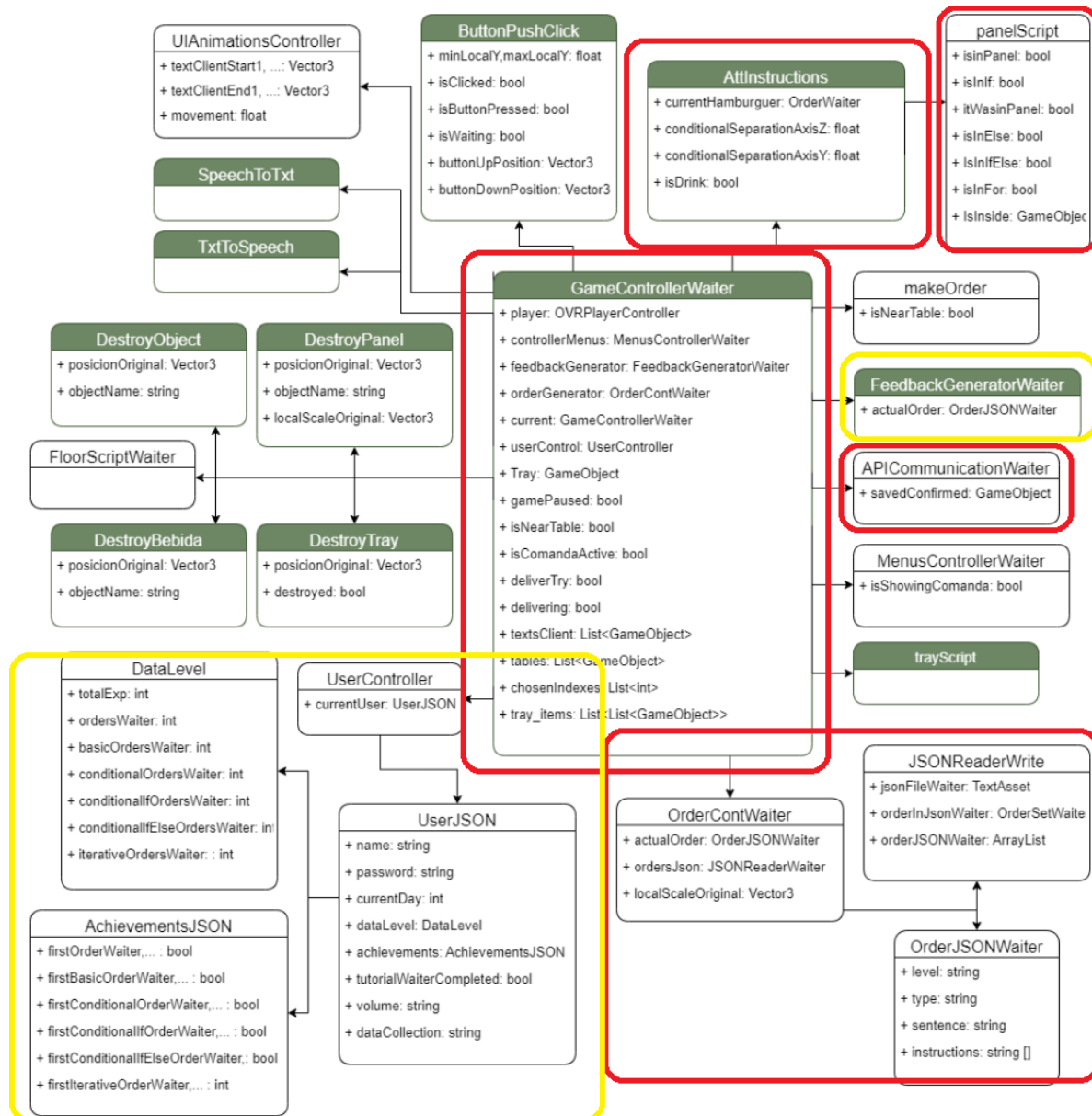


Figure 4.10: Diagrama de clases del modo camarero (Fuente: Waiter Code[12]).

Como en el modo cocinero, unificaremos la parte de usuarios para poder ser utilizada en ambos modos. También rediseñaremos la forma de comunicarse con la base de datos de la clase `APICommunicationWaiter` en la sección 4.2.3.

Las clases `OrderContWaiter`, `JSONReaderWrite` y `OrderJSONWaiter`, responsables de la generación de pedidos también se rediseñarán como en el cocinero. También se adaptará el feedback que recibimos de estos en `FeedbackGenerator`.

Los dos principales problemas del modo camarero son `AttInstructions` y `panelScript` que se explican en las secciones 4.2.3 y 4.2.3.

Finalmente, la clase `GameControllerWaiter` será diferente debido a los cambios en

las otras clases.

4.2.3 Exploración del código Legacy

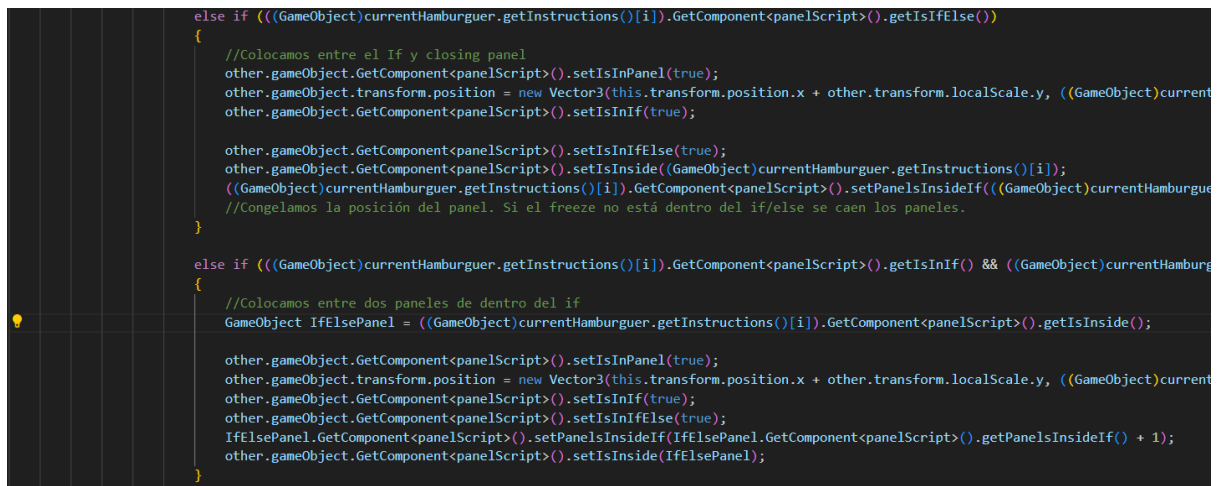
El código Legacy de este proyecto ha pasado por diversas fases de desarrollo y ha sido modificado por diferentes programadores, lo que ha generado varios problemas. A continuación, se detallan estos problemas.

Primero, estamos trabajando con una versión obsoleta de Unity, y Unity recomienda utilizar versiones más actuales. Además, la SDK de Oculus también se considera obsoleta, dado que ha sido reemplazada por la API OpenXR. Esto causa errores al configurar el proyecto y dificulta el mantenimiento del código, ya que es dependiente de una versión antigua de Unity V.2019.4.22f1 y una SDK V.57.0.2-deprecated descontinuada.

En un análisis más detallado, encontramos que algunas partes del código violan los principios SOLID, que son fundamentales para mantener un código legible, flexible, extensible y fácil de mantener.

Pizarra de programación

Esta clase viola el Principio Open/Closed, que establece que el software debe estar abierto a la extensión, pero cerrado a la modificación. Actualmente, la clase tiene muchos condicionales anidados, como podemos ver en la figura 4.11, para detectar qué tipo de bloque está colisionando con la pizarra, lo que dificulta su ampliación y requiere modificar el código existente repetidamente al añadir un nuevo bloque. Este enfoque también provoca la generación de código repetido (alrededor de 564 líneas) ya que cada vez que se cumple una condición, se ejecuta el código para colocar el bloque correspondiente, que es muy similar en muchos casos. Esta lógica podría mejorarse dividiendo las responsabilidades de manera más eficiente, como se explica en la sección 4.3.4.



```

else if (((GameObject)currentHamburger.getInstructions()[i]).GetComponent<panelScript>().getIsIfElse())
{
    //Colocamos entre el If y closing panel
    other.gameObject.GetComponent<panelScript>().setIsInPanel(true);
    other.gameObject.transform.position = new Vector3(this.transform.position.x + other.transform.localScale.y, ((GameObject)currentHamburger.getInstructions()[i]).GetComponent<panelScript>().getIsIfElse(), 0);
    other.gameObject.GetComponent<panelScript>().setIsInIf(true);

    other.gameObject.GetComponent<panelScript>().setIsInIfElse(true);
    other.gameObject.GetComponent<panelScript>().setIsInside(((GameObject)currentHamburger.getInstructions()[i]).GetComponent<panelScript>().getIsIfElse());
    ((GameObject)currentHamburger.getInstructions()[i]).GetComponent<panelScript>().setPanelsInsideIf(((GameObject)currentHamburger.getInstructions()[i]).GetComponent<panelScript>().getIsIfElse());
    //Congelamos la posición del panel. Si el freeze no está dentro del if/else se caen los paneles.
}

else if (((GameObject)currentHamburger.getInstructions()[i]).GetComponent<panelScript>().getIsInIf() && ((GameObject)currentHamburger.getInstructions()[i]).GetComponent<panelScript>().getIsIfElse())
{
    //Colocamos entre dos paneles de dentro del if
    GameObject IfElsePanel = ((GameObject)currentHamburger.getInstructions()[i]).GetComponent<panelScript>().getIsInside();

    other.gameObject.GetComponent<panelScript>().setIsInPanel(true);
    other.gameObject.transform.position = new Vector3(this.transform.position.x + other.transform.localScale.y, ((GameObject)currentHamburger.getInstructions()[i]).GetComponent<panelScript>().getIsIfElse(), 0);
    other.gameObject.GetComponent<panelScript>().setIsInIf(true);
    other.gameObject.GetComponent<panelScript>().setIsInIfElse(true);
    IfElsePanel.GetComponent<panelScript>().setPanelsInsideIf(IfElsePanel.GetComponent<panelScript>().getPanelsInsideIf() + 1);
    other.gameObject.GetComponent<panelScript>().setIsInside(IfElsePanel);
}

```

Figure 4.11: Ejemplo de condicionales anidados (Fuente propia).

Otro problema es la escalabilidad espacial del objeto pizarra. Al añadir un bloque, se posiciona en un sitio fijo y mantiene su tamaño inicial. Sería ideal que la pizarra pudiera hacerse más grande sin perder la correcta colocación de los Bloques de Programación, como se puede ver en la sección 4.3.4.

Bloques de programación

Los bloques de programación y la pizarra no simulan correctamente la “ejecución” del programa. Actualmente, cuando un bloque colisiona con la pizarra, se “añade” el ingrediente a la lista interna de la clase `AttInstructions` para generar la hamburguesa, lo que hace que la pizarra maneje tanto la lógica de ejecución de los bloques como el control de su visualización, violando el Principio de Responsabilidad Única (**Single Responsibility Principle**). La solución a este problema es que la pizarra solo muestre los bloques y que otra clase se encargue de ejecutar estos bloques para obtener los ingredientes. Esto lo podemos ver con más detalle en las secciones 4.3.4 y 4.3.2 respectivamente.

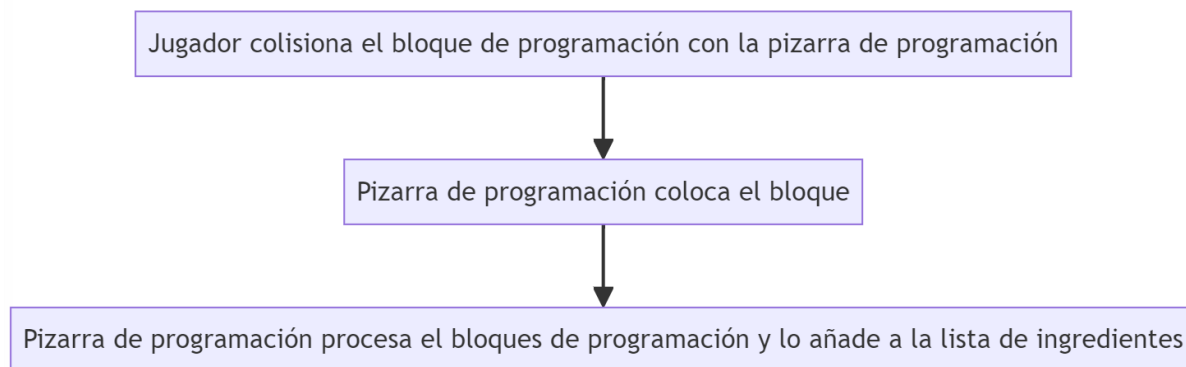


Figure 4.12: Pasos para añadir un bloque de programación en la versión Legacy (Fuente propia).

Cabe destacar en este punto que si quisiéramos añadir otro bloque, tendríamos que modificar cómo interactúa este bloque con la pizarra en lugar de modificar el bloque en sí, debido a cómo están repartidas las responsabilidades entre clases. Además, los bloques tienen un tamaño fijo, y si son más grandes o más pequeños, se visualizan incorrectamente, dificultando el ajuste de su posición y tamaño en la pizarra.

Un aspecto importante a mencionar es que, en la versión Legacy, solo existe una clase de Bloques de Programación llamada `panelScript`, aunque hay varios tipos. Desde el punto de vista del diseño de software, sería más flexible y mantenible utilizar el concepto de herencia y polimorfismo para crear una clase base o madre de la cual hereden los diferentes tipos de bloques. Este enfoque cumple el Principio de **Sustitución de Liskov**, uno de los principios SOLID, que establece que los objetos de una clase base deben poder ser reemplazados por objetos de una clase derivada sin alterar el correcto funcionamiento del programa. Además, facilitaría la extensión del sistema al permitir añadir nuevos tipos de bloques sin necesidad de modificar la clase base, respetando así el Principio **Open/Closed**.

Generación de pedidos

En ambos modos de juego, la generación de pedidos presenta problemas que afectan la jugabilidad y la flexibilidad del sistema. Actualmente, los pedidos son estáticos y no se generan de manera aleatoria, lo que reduce la variedad y la experiencia del jugador. En el modo cocinero, los bloques de programación son solo imágenes, lo que limita la implementación de un sistema de generación de pedidos aleatoria. Con la incorporación

del modo camarero, donde se colocan bloques de programación en una pizarra, sería beneficioso utilizar los mismos componentes para ambos modos.

Cada modo de juego cuenta con sus propios assets fijos, lo que limita la flexibilidad y la reutilización de estos elementos. La idea es hacer que los assets sean variables dinámicamente, permitiendo su reutilización en ambos modos. Esto no solo mejoraría la jugabilidad al introducir mayor variedad, sino que también optimizaría el desarrollo al reducir la necesidad de crear assets específicos para cada modo.

Comunicación con la API

Como podemos ver en la figura 4.10, existe una clase llamada `APICommunicationWaiter` que nos permite comunicarnos con el Backend para acceder a la base de datos. También existe un clase `APICommunication` aunque no la podemos ver en ninguno de los diagramas de los proyectos anteriores. Esto no deja nada claro que clase es la funcional o si son ambas. Tener dos clases que cumplen la misma función para cada modo no es óptimo, ya que tenemos código repetido y hace que la estructura sea más complicada. Además, en el caso que queramos crear un nuevo modo de juego tendríamos que crear otra clase en vez de utilizar una clase común. Generalmente si los datos que tenemos que guardar en la base de datos son los mismos, no deberían tener dos clases diferentes.

Otro problema, que he detectado al analizar el código es que cuando se realizan llamadas a la API se especifica con una variable la URL del Backend con cada llamada cuando podría estar en una constante para evitar código repetido.

También tenemos una clase que realiza diversas responsabilidades, esto viola el Principio de **Single Responsibility**. Este principio dice que cada clase tiene que tener una única responsabilidad para evitar tener clases que den servicios a clases diferentes, incrementando entonces el acoplamiento. En esta caso, la clase `APICommunication` gestiona elementos de la vista y gestiona la comunicación de la API.

```
IEnumerator putRequest()
{
    string jsonData = JsonUtility.ToJson(GameControllerWaiter.current.current_user().getCurrentUser());
    //var jsonString = JsonUtility.ToJson(jsonData) ?? "";

    //byte[] myData = System.Text.Encoding.UTF8.GetBytes();
    UnityWebRequest www = UnityWebRequest.Put("http://shrouded-sands-87010.herokuapp.com/sendUser", jsonData);
    www.SetRequestHeader("Content-Type", "application/json");

    yield return www.SendWebRequest();
    savedConfirmed.GetComponent<TextMeshProUGUI>().SetText("SAVED!");
}
```

Figure 4.13: Gestión de vista y comunicación de la API (Fuente propia).

4.2.4 Arquitectura del sistema

La arquitectura del sistema, como se muestra en la figura 4.14, no ha cambiado significativamente; mantenemos los mismos módulos. La única diferencia notable está en el Backend. Dado que la interacción con la API se va a modificar, he decidido simplificar las llamadas limitando las funciones del Backend a la autenticación de usuarios y el almacenamiento de propiedades. Además, como requisito no funcional del proyecto, se ha implementado el cifrado de contraseñas para garantizar la seguridad de los datos.

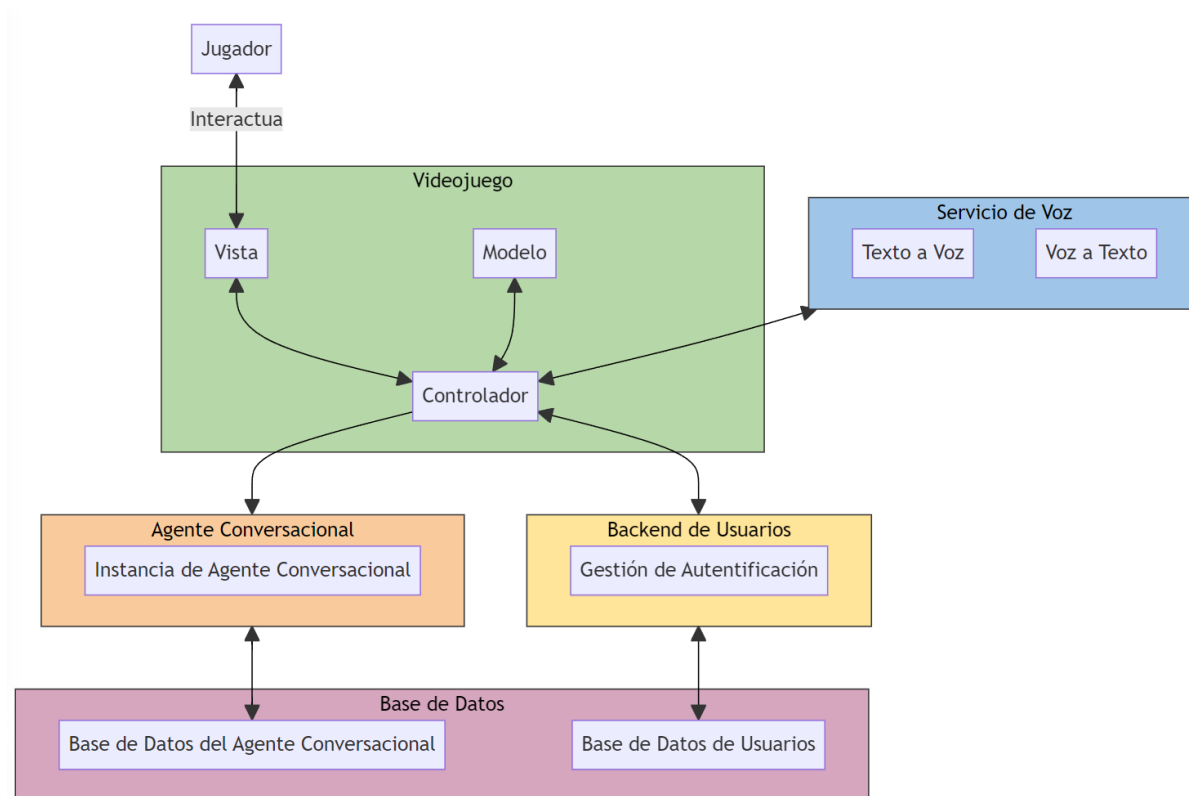


Figure 4.14: Arquitectura del sistema (Fuente propia).

4.2.5 Backend: Modelo de datos

Las funciones del Backend están limitadas a la autenticación de usuarios y al almacenamiento de propiedades, donde se guarda la experiencia del jugador y la cantidad de pedidos realizados. En la figura 4.15, se presenta el modelo de datos que ilustra qué información se guarda, utilizando un modelo entidad-relación para proporcionar una visión clara y estructurada de los datos manejados por el sistema. Cabe destacar que el backend anterior no cifraba las contraseñas, lo que representaba una vulnerabilidad. Con la implementación del cifrado, se refuerza la integridad y la seguridad del sistema, cumpliendo con los requisitos no funcionales del proyecto.

4.3 Refactorización y Patrones de Diseño utilizados

En el desarrollo del proyecto, hemos aplicado varios patrones de diseño que nos ayudan a organizar y estructurar el código de manera eficiente y mantenible. A continuación, se detallan las refactorizaciones realizadas y los patrones de diseño utilizados, explicando el problema enfrentado y cómo cada patrón nos ayudó a resolverlo.

4.3.1 Bloques de Programación

Implementación de una nueva clase base

En el código Legacy teníamos una clase `panelScript` que gestionaba todos los Bloques de Programación (anteriormente llamados paneles). En esta nueva implementación, he optado por tener una clase base `ProgrammingBlock` y las clases hijas correspondientes

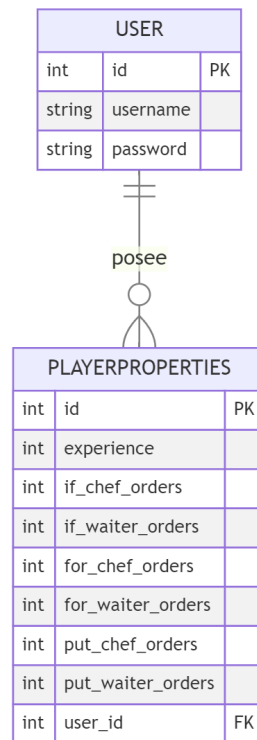


Figure 4.15: Modelo Entidad-Relación (Fuente propia).

PutBlock, IfBlock y ForBlock. Respetando el Principio de Sustitución de Liskov y el Principio Open/Close como explicaba en la sección 4.2.3.

Tipos de bloque

4.3.2 Tipos de bloque

Los Bloques de Programación tienen una variable de tipo `ProgrammingBlockType` para conocer en todo momento con qué tipo estamos tratando. Además, se introducen los enums `ConditionalOperatorType` e `IngredientType` para una mejor gestión de los tipos de operadores y tipos de ingredientes respectivamente. A continuación, se muestran los posibles valores de estos enums:

- `ConditionalOperatorType`:
 - `GreaterThan`
 - `LessThan`
 - `GreaterThanOrEqual`
 - `LessThanOrEqual`
- `IngredientType`:
 - `BreadTop`
 - `BreadBottom`
 - `Meat`

- Cheese
- Lettuce
- Ketchup

Fábrica de bloques

También existe una clase `ProgrammingBlockFactory` que utiliza el patrón Factory. El Factory Pattern es utilizado para crear objetos sin especificar la clase exacta del objeto que será creado. Este patrón nos permite encapsular la lógica de creación de objetos, facilitando la adición de nuevas clases de objetos sin modificar el código existente.

Esta `ProgrammingBlockFactory` es responsable de instanciar los `ProgrammingBlock` y configurarlo con los valores pasados por parámetro. Al centralizar la lógica de creación, podemos fácilmente extender el juego añadiendo nuevos tipos de entidades sin alterar significativamente el código base.

Método Execute

Para representar los bloques internos, hemos usado una lista de `ProgrammingBlock`, la cual se itera para obtener los resultados de estos bloques. Por ejemplo, un bloque de bucle o condición tiene dentro otros bloques de programación, formando así una estructura de árbol.

Para ello, usamos el patrón Composite. El **Composite Pattern** (Patrón de Composición) permite tratar objetos individuales y compuestos de manera uniforme. Este patrón se utiliza para representar jerarquías de objetos, como los árboles de componentes o elementos de la interfaz de usuario. Esto nos permite ejecutar el mismo método en todos los bloques, haciendo que los bloques “padres” sean responsables de ejecutar el método de los “hijos”. Así, solo es necesario ejecutar un bloque para obtener el resultado de todos los bloques inferiores.

La clase padre `ProgrammingBlock` tiene un método obligatorio a sobrescribir llamado `Execute()`. Este método sirve para obtener el resultado de ejecución de los bloques. El resultado es una lista de ingredientes, ya que los bloques `For` y `If` pueden contener otros bloques dentro de ellos, como se muestra en la figura 4.16.

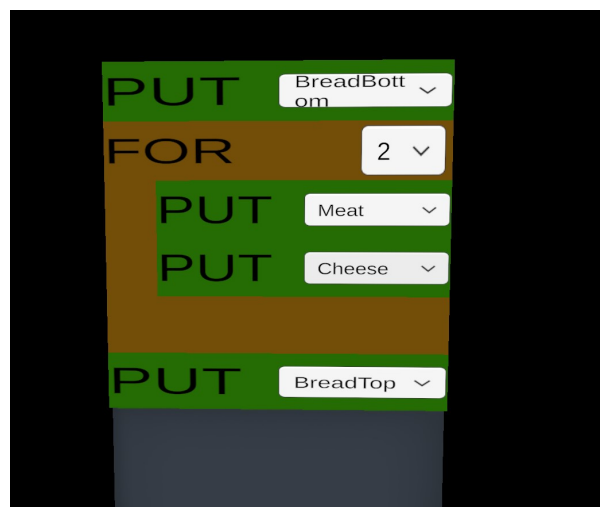


Figure 4.16: Bloque For con bloques en su interior (Fuente propia).

Bloque If

El bloque **If** tiene las siguientes características:

- Una variable que indica si es **true** o **false**.
- Una variable del tipo **ConditionalOperatorType** (un enum) para saber el operador de la condición.
- Un tipo de variable **IngredientType** (también un enum) que define el tipo de ingrediente.
- Un valor que indica la cantidad de ingredientes.

Todo esto es necesario para determinar si la condición es **true** o **false**. Tenemos una función **Evaluate** que nos permite evaluar la condición con un valor y así fijar si es **true** o **false**. Si la condición es verdadera, el método **Execute** se ejecuta.

Bloque For

El bloque **For** tiene como variable un número de iteraciones y, al hacer **Execute**, repite el array que tiene dentro las veces indicadas por el número de iteraciones.

Consideraciones adicionales

Extensibilidad: La nueva estructura con la clase base (**ProgrammingBlock**) y patrones de diseño permite una fácil extensibilidad para añadir nuevos tipos de bloques en el futuro.

Mantenibilidad: La separación de responsabilidades y el uso de patrones de diseño facilitan el mantenimiento y la comprensión del código.

Reutilización de código: La implementación de métodos y atributos comunes en la clase base **ProgrammingBlock** permite la reutilización de código y evita la duplicación.

4.3.3 Ingredientes

En el código Legacy teníamos una clase para cada ingrediente y todos implementaban una interfaz **CreacioIngredient**. En esta versión he optado por tener una clase base **Ingredient** que hereda de otra clase base **Supply**. Como en los Bloques de Programación de la sección 4.3.1, esto nos permite dividir las responsabilidades de cada clase.

También tenemos una clase **IngredientFactory** que nos permite delegar la responsabilidad de crear ingredientes a esta. Podemos encontrar la explicación del **Factory Pattern** en la sección 4.3.2.

4.3.4 Pizarra de Programación

Cambio de nombre y problemas anteriores

En el código Legacy, la clase que se encargaba de gestionar la Pizarra de Programación en el modo camarero se llamaba **AttInstructions**, ahora se llama **ProgrammingBoard**. Mantener nombres coherentes nos ayuda a entender y estructurar el código.

Teníamos una clase con muchos condicionales anidados para detectar qué tipo de bloque estábamos interactuando. Otro problema era la repetición de código para colocar los bloques visualmente.

Responsabilidad de la nueva clase

Para dejar de violar el principio de responsabilidad única, hemos hecho que la clase `ProgrammingBoard` solo gestione la colocación de los bloques en la pizarra.

Estructura de `ProgrammingBoard`

La `ProgrammingBoard` está formada por una lista de `Line`. `Line` es una clase creada para gestionar y guardar información de las líneas de la pizarra. Guarda el número de línea, la posición de la línea (para colocar el bloque), el `ProgrammingBlock` que ocupa esta línea y la `parentLine` de tipo `Line`. Esta última se utiliza para indicar que un bloque está dentro de un bloque de tipo `IfBlock` o `ForBlock`.

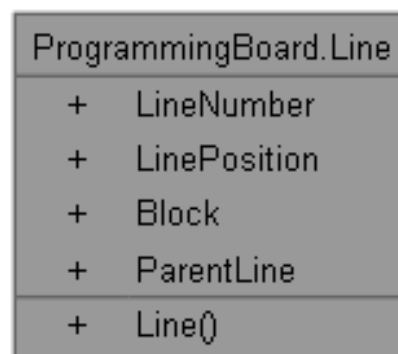


Figure 4.17: Clase interna `Line` (Fuente propia).

Funcionamiento de `ProgrammingBoard`

Primero de todo, tenemos cuatro formas de dibujar los bloques dentro de una línea:

- `PlaceBlockInline`: Permite colocar un bloque sin indentación.
- `PlaceBlockTabbed`: Permite colocar un bloque con una indentación que podemos especificar (en este caso de `If` o `For`, esto influye en el color y se pasa por parámetro un `gameObject` fijo para ambos casos). Podemos ver un ejemplo de esto en la figura 4.16.
- `PlaceEmptyTabbed`: Permite colocar una indentación sin un bloque.
- `PlaceEndBlockTabbed`: Permite cerrar las indentaciones.

Construcción de la pizarra

A partir de una lista de `ProgrammingBlock`, podemos comprobar el tipo de instrucción y construir una forma u otra a partir de un método específico como `PlaceIfBlock` en el caso de los `If`, `PlaceForBlock` en el caso de `For` y `PlaceBlockInLine` en el caso del `Put`. Esto nos permite pasarle a `ProgrammingBoard` una lista de Bloques de Programación y mostrarlos en la pizarra.

Modo cocinero

Poder pasarle a la `ProgrammingBoard` una lista de Bloques de Programación es útil en el modo cocinero, ya que nos permite mostrar el pseudocódigo que queramos y no solo una imagen como pasaba en el código Legacy. También tenemos un método para poder limpiar la Pizarra de los Bloques de Programación.

Adición y eliminación de bloques

Para añadir bloques y modificar la pizarra, utilizamos una función para conocer en qué línea se quiere añadir y comprobar, a través de `parentLine`, si necesitamos añadir el bloque a otro bloque como un `For` o `If` o a la propia lista. Después de esto, limpiamos la pizarra y volvemos a mostrar los bloques, esta vez con el bloque añadido. Lo mismo pasa para eliminar un bloque: detecta que el bloque agarrado sale de la pizarra, borra y carga otra vez.

Cantidad de líneas y escala

También es importante señalar que podemos especificar la cantidad de líneas que tiene la pizarra en el Inspector de Unity³ para que después `ProgrammingBoard` pueda calcular dónde colocar el bloque y la escala de este para que quede bien en la pizarra.

4.3.5 Generación aleatoria de pedidos

La generación aleatoria de pedidos es un componente esencial para aumentar la jugabilidad y variabilidad del juego. En versiones anteriores, la generación de pedidos era fija, limitando así la diversidad de los desafíos presentados al jugador. Gracias a las nuevas implementaciones en los bloques de programación y en la pizarra, es posible incorporar un sistema de generación de pedidos genérico y fácilmente ampliable para ambos modos de juego.

4.3.6 Generación de pedidos en el modo cocinero

En el modo cocinero, se genera una lista de bloques de programación que se muestra en la pizarra. Este enfoque flexible y dinámico permite una mayor personalización y complejidad en los pedidos, mejorando significativamente la experiencia del jugador.

Algoritmo de generación de pedidos

Para entender cómo funciona este sistema, es fundamental conocer los algoritmos utilizados en la generación aleatoria de pedidos.

La figura 4.18 muestra el proceso completo de generación de un pedido en el modo cocinero. Este proceso comienza con la adición de una base que depende del nivel del jugador, principalmente compuesta por el pan inferior, una carne y ingredientes adicionales. Luego, se añaden bloques adicionales, como condicionales o bucles, y se generan los ingredientes del interior de manera aleatoria, según el nivel de dificultad y los ingredientes

³El Inspector de Unity es una ventana en el editor de Unity que permite a los desarrolladores ver y editar las propiedades de los objetos seleccionados. Es una herramienta esencial para modificar y ajustar componentes y valores en tiempo real durante el desarrollo del juego.

disponibles. Este algoritmo garantiza que los ingredientes seleccionados estén disponibles en el inventario y que la lista de instrucciones no sobrepase la capacidad de la pizarra.

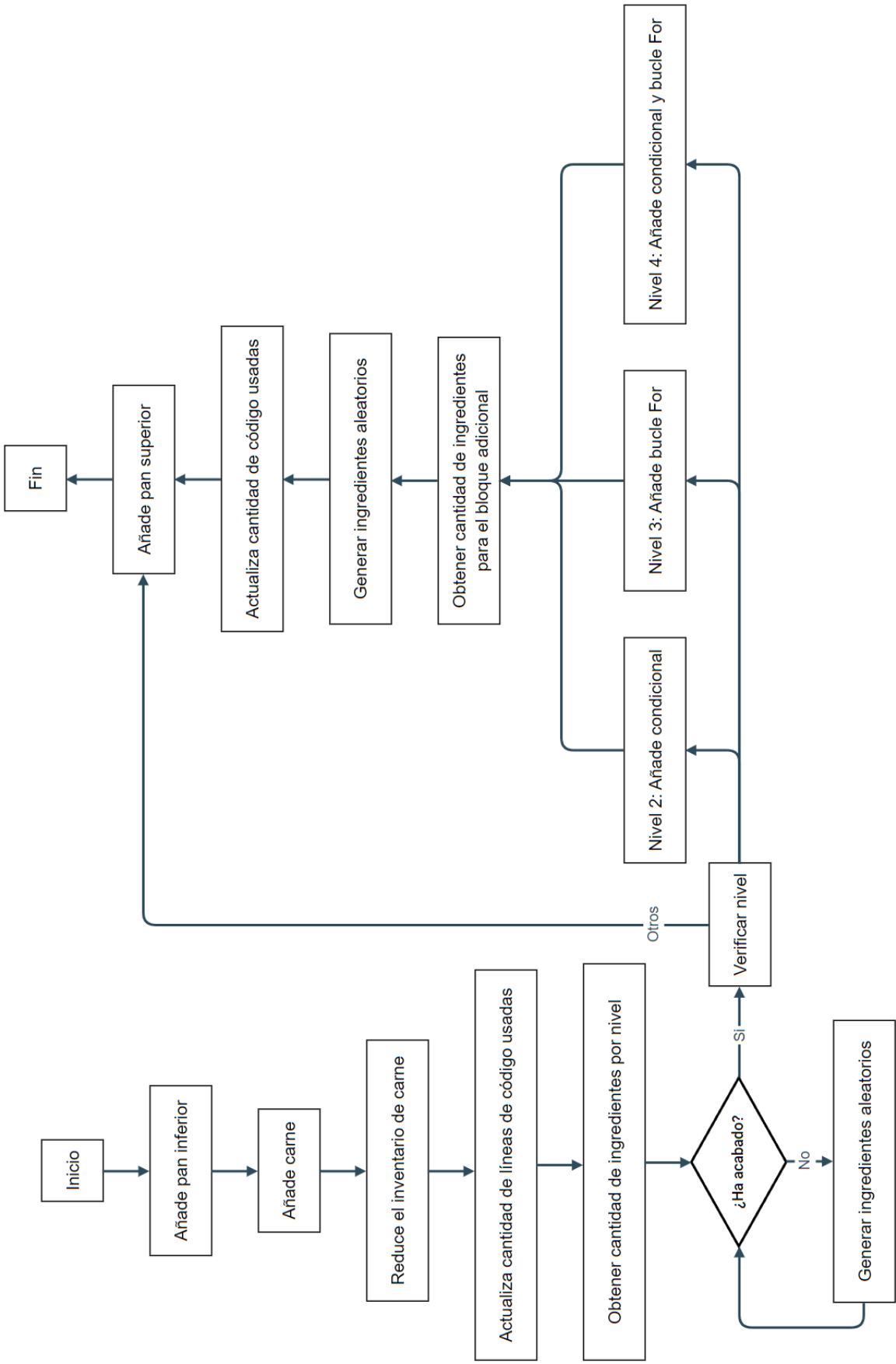


Figure 4.18: Generación de pedidos en el modo cocinero (Fuente propia).

Selección de ingredientes aleatorios

En la figura 4.19, se detalla el algoritmo para la selección de ingredientes aleatorios en el modo cocinero. Este algoritmo verifica la disponibilidad de los ingredientes en el inventario antes de añadirlos al pedido, asegurando que siempre se puedan cumplir los pedidos generados.

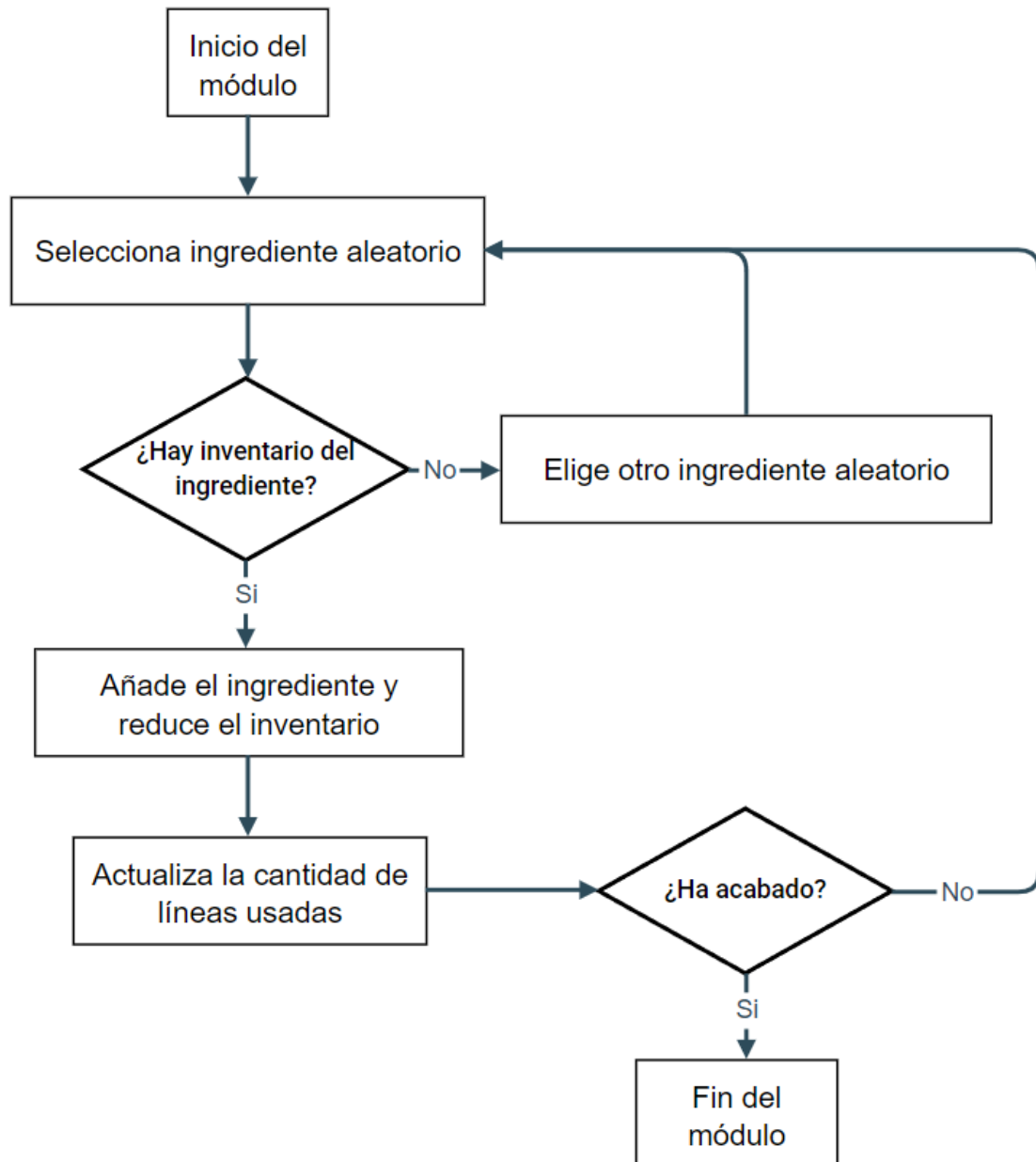


Figure 4.19: Generación de ingredientes aleatorios (Fuente propia).

4.3.7 Generación de pedidos en el modo camarero

A diferencia del modo cocinero, en el modo camarero nos interesa generar una lista de ingredientes que el cliente pide. En este caso, los pedidos son más básicos debido a que

solo disponemos del bloque **Put** y del bloque **For**, y no tenemos límite de suministro. Por ello, generamos una lista de ingredientes dependiendo del nivel, que comprueba si es posible debido a las líneas de la pizarra.

Algoritmo de generación de pedidos

Para fomentar el uso del bloque **For**, a partir de niveles más altos, aparecerán patrones repetitivos como "carne, queso, carne, queso, carne, queso". Este tipo de pedidos incentiva el uso del bloque **For** para optimizar la programación en la pizarra. Por ejemplo, si tenemos un pedido que requiere 6 combinaciones de "carne, queso", no podremos "programarlo" con bloques **Put** debido al tamaño limitado de la Pizarra de Programación, que puede tener, por ejemplo, solo 10 líneas.

Traducción a lenguaje natural

Una vez generado el pedido, deberemos traducir la lista de ingredientes a lenguaje natural para que los jugadores puedan comprenderla. Los ingredientes se traducen a lenguaje natural para simular que es un cliente pidiendo su hamburguesa.

4.3.8 Implementación en el código

La generación de pedidos para ambos modos está implementada en la clase **OrderGenerator**, que se encarga de manejar la lógica de generación de pedidos, tanto para el modo cocinero como para el modo camarero.

La traducción de los pedidos generados se maneja en la clase **OrderUtils**, esta también permite la traducción de bloques de programación a ingredientes. Necesaria para poder obtener una lista de ingredientes a partir de los bloques de programación en la pizarra de programación, haciendo uso del método **Execute** explicado en 4.3.2.

La división de responsabilidades entre **OrderGenerator** y **OrderUtils** en este caso, asegura un código limpio y modular, permitiendo mejoras y cambios de manera eficiente.

4.3.9 Generación del feedback

Esta parte no ha cambiado tanto como las demás. En la versión Legacy, teníamos una serie de comprobaciones para obtener el feedback. Por ejemplo, si el pedido tenía más ingredientes, se avisaba al jugador que había puesto ingredientes extras o si el pedido era correcto. La clase **FeedbackGenerator** se encarga de hacer eso mismo, pero con algunos cambios ligeros. Tenemos un enum llamado **FeedbackType** para almacenar los tipos de feedback que podemos tener, lo que ayuda a mantener una misma consistencia a través de las clases. Esta clase se centra en comparar una lista de ingredientes con una lista de ingredientes objetivo para identificar los errores cometidos y transmitir una lista de feedback.

Los tipos de feedback que existen actualmente son:

- **Correct**: El pedido es correcto.
- **RawMeat**: El pedido contiene carne cruda.
- **InsufficientIngredients**: Faltan ingredientes en el pedido.

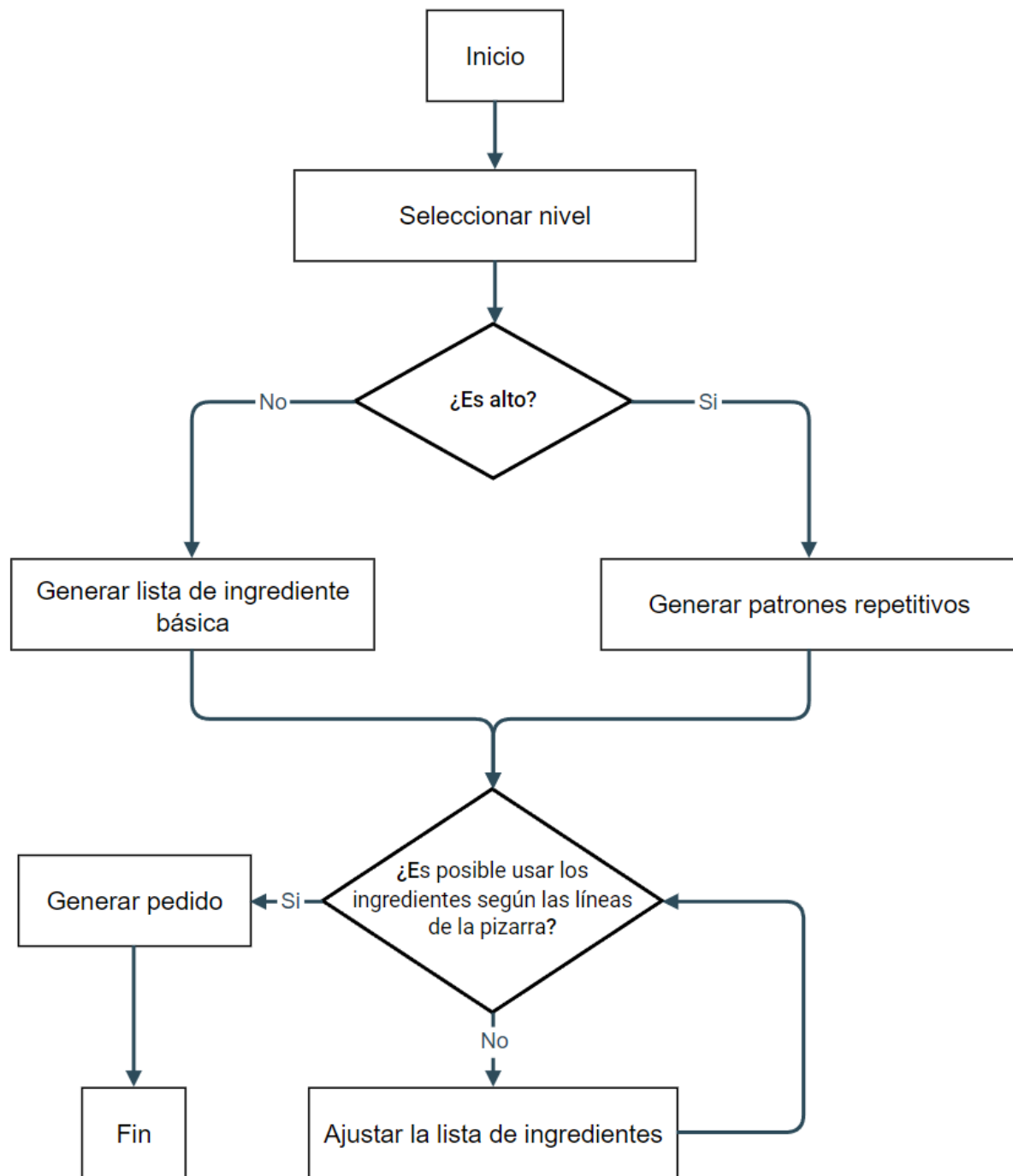


Figure 4.20: Generación de pedidos en el modo camarero (Fuente propia).

- **ExtraIngredients:** Hay ingredientes extras en el pedido.
- **Incorrect:** El pedido es incorrecto.

La principal diferencia entre la versión Legacy y la actual es que en la Legacy, la clase retornaba un string con el feedback, mientras que esta retorna el enum **FeedbackType**. Además, en la versión actualizada, podemos obtener diversos feedbacks, como se puede apreciar en la figura 4.22, ya que retorna una lista. De esta forma, cada modo de juego es responsable de mostrarle el feedback al jugador a partir de un mensaje basado en esta

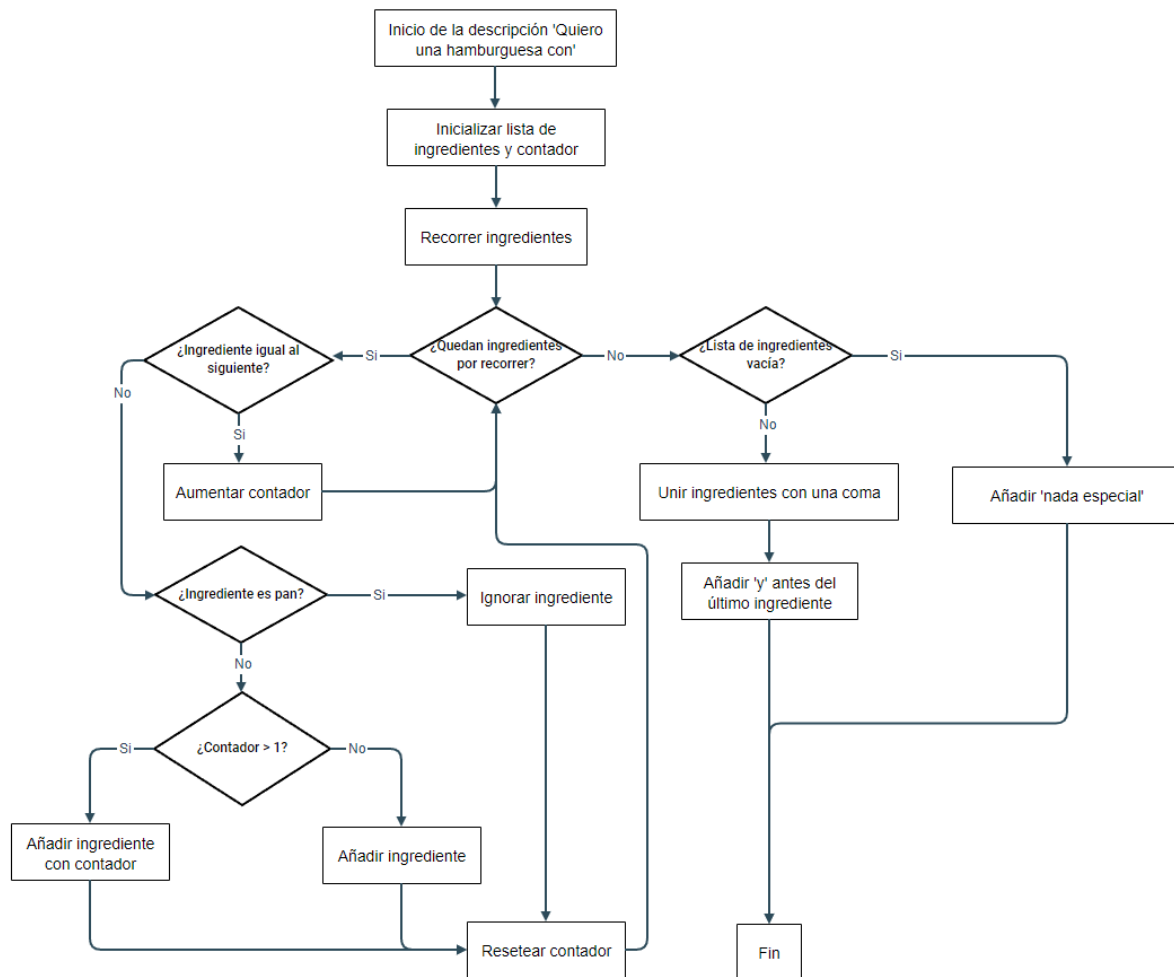


Figure 4.21: Traducción de ingredientes a lenguaje natural (Fuente propia).

lista.

4.3.10 Comunicación con la API

Refactorización de la Comunicación con la API

En la sección 4.2.3, mencionamos problemas como la falta de adherencia al **Principio de Responsabilidad Única** y la duplicación de código en varias clases. Para abordar estos problemas, hemos implementado una clase llamada **BackendConnector**.

Clase BackendConnector **BackendConnector** centraliza todas las peticiones al back-end, resolviendo la repetición de código y mejorando la responsabilidad única. Sus funciones principales incluyen:

- **Gestión de peticiones:** Maneja todas las solicitudes HTTP (GET, POST, PUT, DELETE).
- **Manejo de errores:** Gestiona y reporta errores de comunicación.
- **Autenticación:** Maneja los tokens y la autorización de las solicitudes.



Figure 4.22: Ejemplo de Feedback Múltiple (Fuente propia)

Clase GameManager Para asegurar consistencia en la sesión y datos invariables como las instancias de `BackendConnector` y `PlayerProperties`, se ha creado la clase `GameManager` utilizando el patrón Singleton.

Patrón Singleton

El patrón Singleton garantiza una única instancia de una clase, proporcionando un punto de acceso global. Esto es crucial para gestionar estados y recursos compartidos de manera eficiente.

Implementación de GameManager

- **Instancia única:** Solo existe una instancia de `GameManager`.
- **Consistencia de datos:** Mantiene referencias únicas a `BackendConnector` y `PlayerProperties`.
- **Gestión de sesión:** Permite la interacción con la misma sesión a través de diferentes escenas.

4.3.11 Mención al patrón Observer

El Observer Pattern (Patrón de Observador) define una dependencia de uno a muchos entre objetos, de manera que cuando uno de los objetos cambia su estado, todos sus dependientes son notificados y actualizados automáticamente. Este patrón es ideal para implementar sistemas de eventos y notificaciones.

En Unity, utilizamos delegates y eventos para implementar el Observer Pattern. Los delegates son punteros a funciones que permiten definir un tipo de método que puede ser referenciado y llamado, mientras que los eventos son una manera de suscribirse a estos delegates y ser notificados cuando ocurren cambios.

En nuestro proyecto, hay múltiples casos donde se aplica el Observer Pattern: al entregar el pedido en la bandeja de entrega, al pulsar el botón para generar el pedido, al entregar el pedido en la mesa, etc. Además, muchas funcionalidades de Unity funcionan con el mismo patrón, como las colisiones o la interacción con botones de la interfaz gráfica.

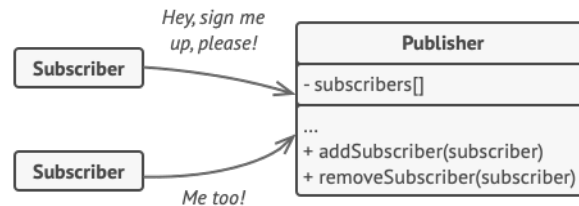


Figure 4.23: Observer Pattern Suscripción (Fuente: Refactoring Guru).

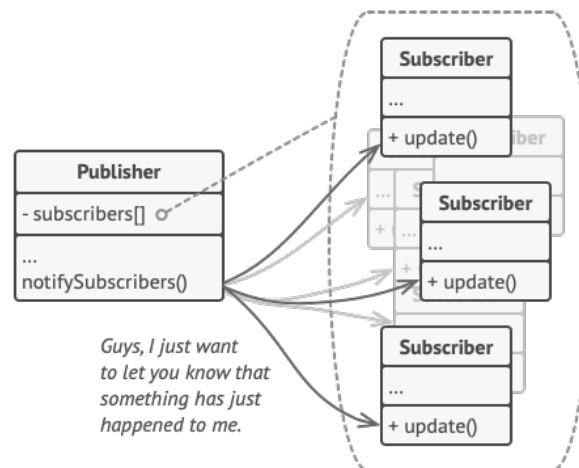


Figure 4.24: Observer Pattern Notificación (Fuente: Refactoring Guru).

4.4 Diagrama de Clases

En esta sección se presenta el diagrama de clases del proyecto y se explican algunas clases que no se han mencionado previamente. Este diagrama es fundamental para entender la estructura y las relaciones entre las diferentes clases que componen el sistema. Además, se proporcionan detalles sobre las funcionalidades y roles específicos de estas clases adicionales, enriqueciendo así la comprensión global del diseño del proyecto.

4.4.1 Explicación de Clases Específicas

Las clases *controller* se utilizan en este caso para guiar el flujo del juego. Por ejemplo, gestionan el paso de los días, la generación de pedidos y la entrega de los mismos en el

caso del cocinero. También se encargan de agrupar los diversos elementos del modelo, así como la pizarra de programación y otros elementos propios del modo de juego. Entre estos se encuentran los *RobotTV*, que se detallan a continuación.

Las clases *RobotTV* son responsables de mostrar la interfaz en ambos modos de juego. Se puede considerar que representan la vista en nuestro juego, excluyendo el entorno. Cada *RobotTV* incluye diferentes elementos de vista dependiendo del modo de juego en el que se encuentre el jugador.

4.4.2 Otras Clases Específicas de los Modos de Juego

En el modo de cocinero, tenemos la clase *DeliveryTable*, que es responsable de manejar la lógica de la mesa en la que se entregan los pedidos. También gestiona los *Suppliers*, anteriormente conocidos como contenedores proveedores, que son las clases encargadas de generar ingredientes y gestionar su suministro.

En el modo camarero, contamos con la clase *OrderButton*, que gestiona el botón que se debe presionar para generar físicamente el pedido en la bandeja. La clase *OrderBuilder* permite construir la hamburguesa sobre la bandeja para poder llevar el pedido. Finalmente, la clase *ClientTable* guarda los datos sobre las mesas de los clientes, indicando si tienen un pedido pendiente o si es la mesa seleccionada en ese momento. También gestiona la interacción con dicha mesa.

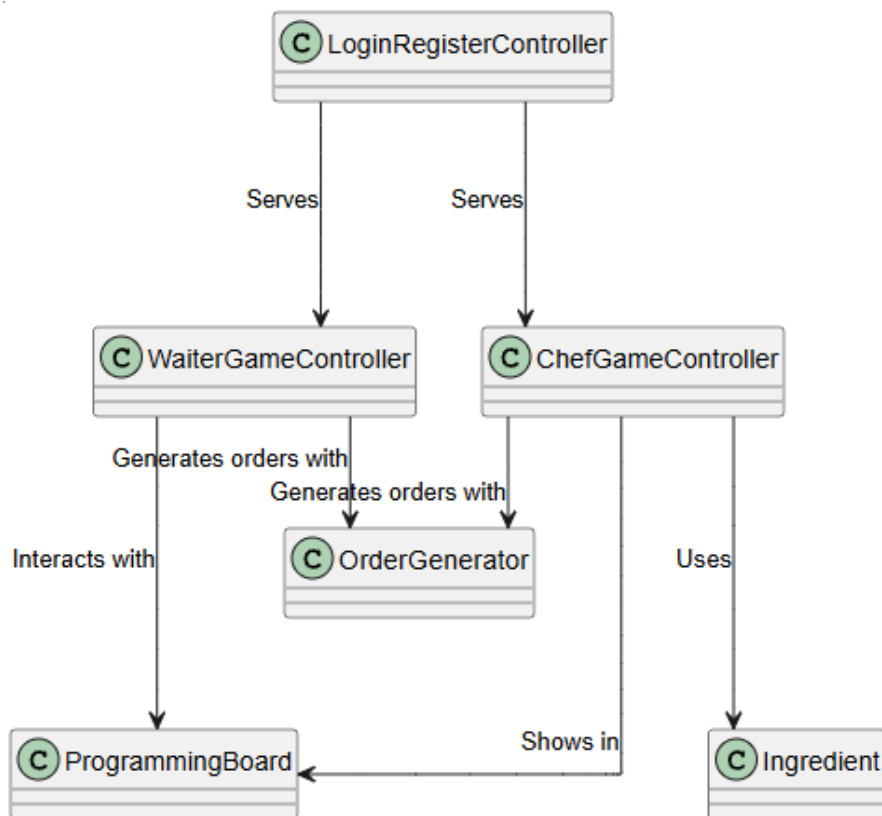


Figure 4.25: Diagram de clases general (Fuente propia).

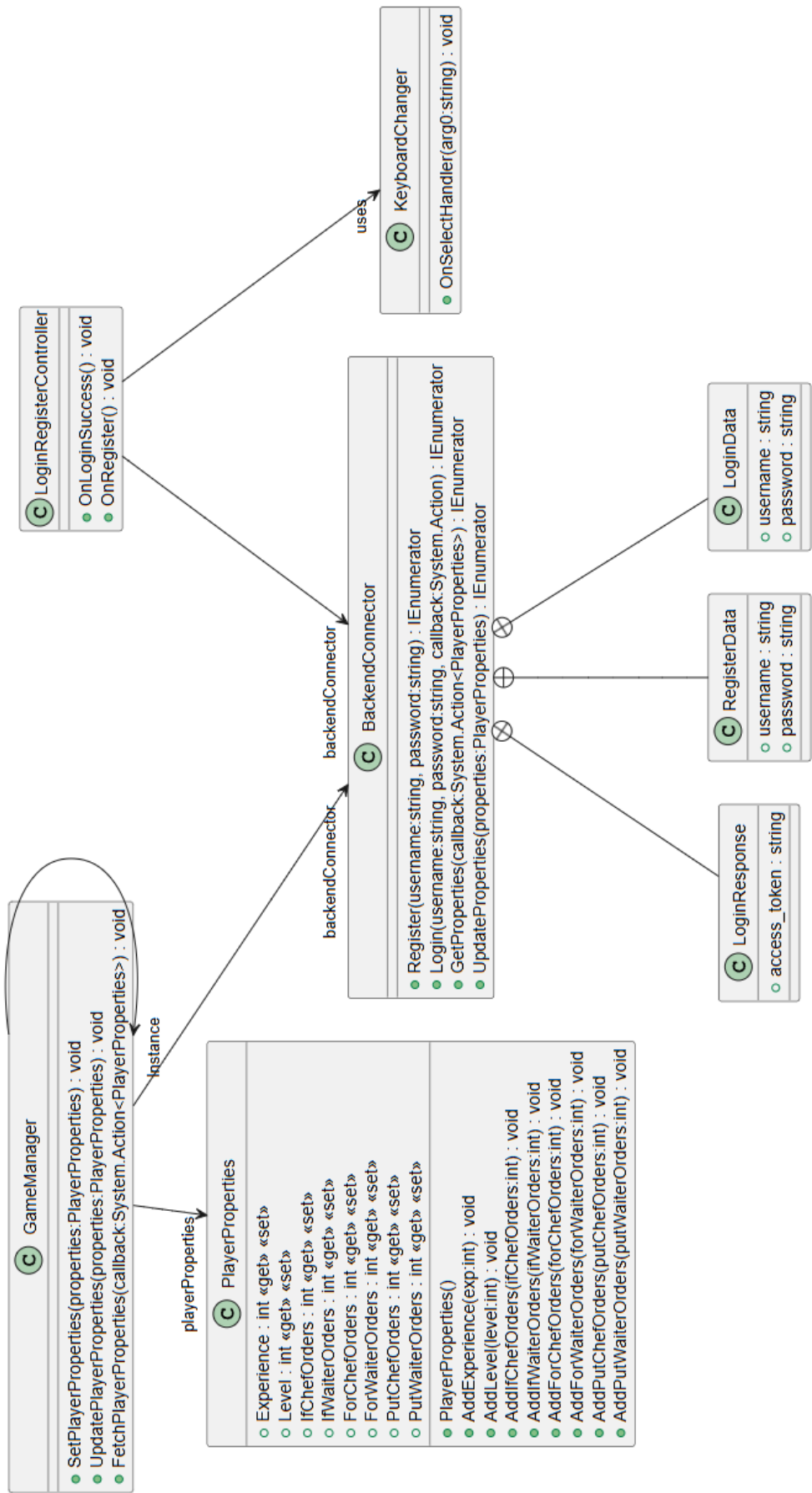


Figure 4.26: LoginRegisterController (Fuente propia).

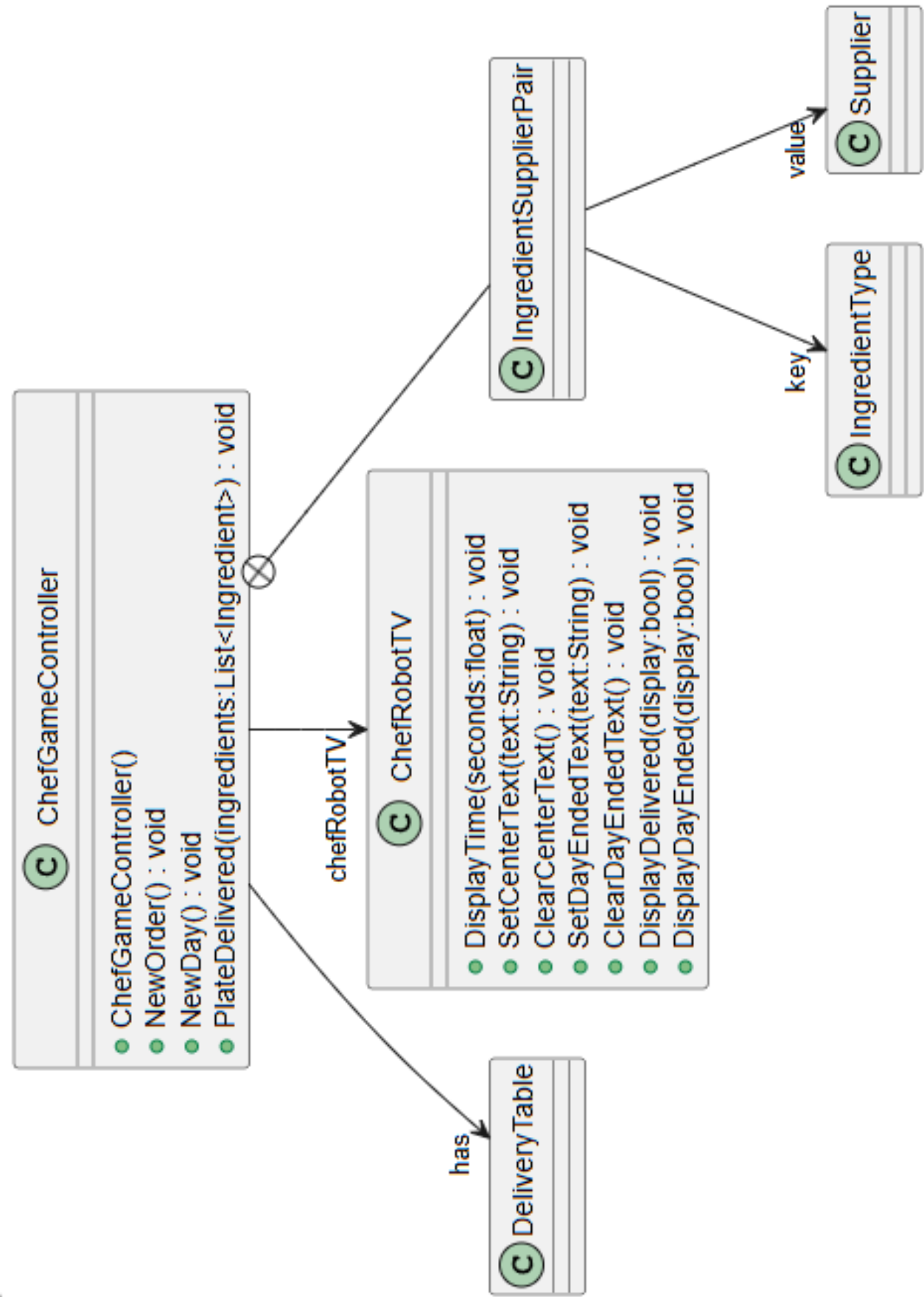


Figure 4.27: ChefGameController (Fuente propia).

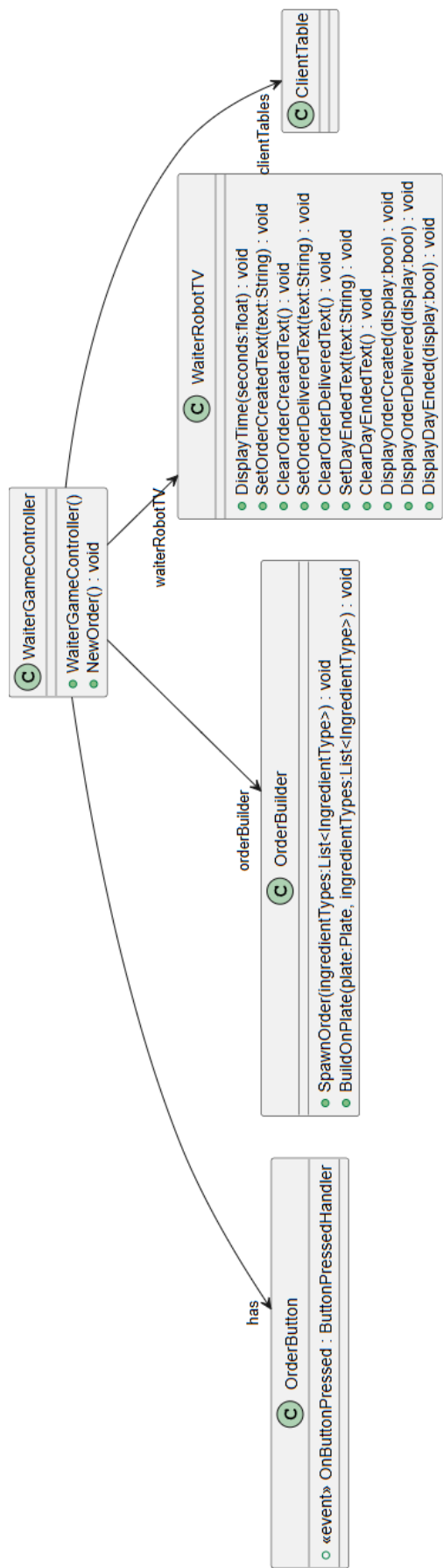


Figure 4.28: WaiterGameController (Fuente propia).

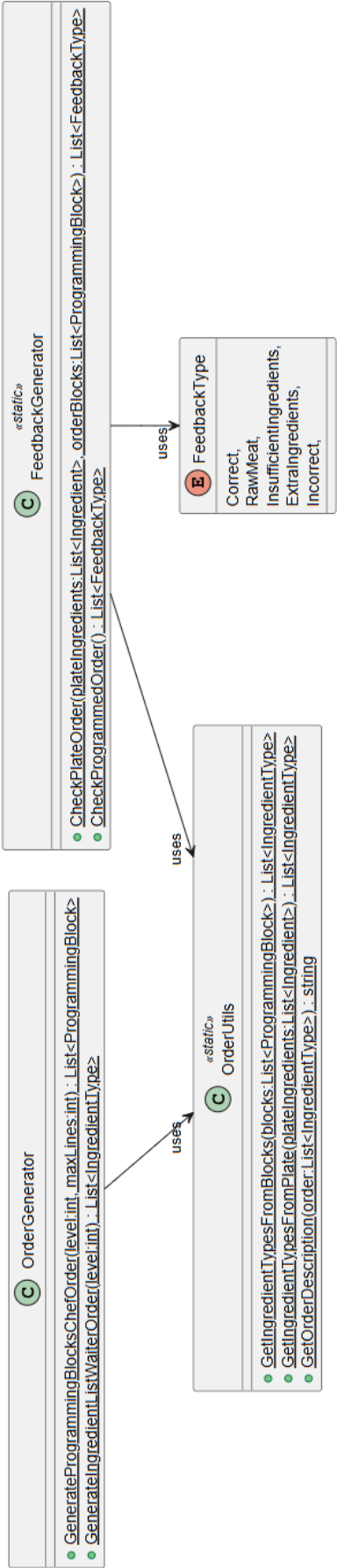


Figure 4.29: OrderGenerator (Fuente propia).

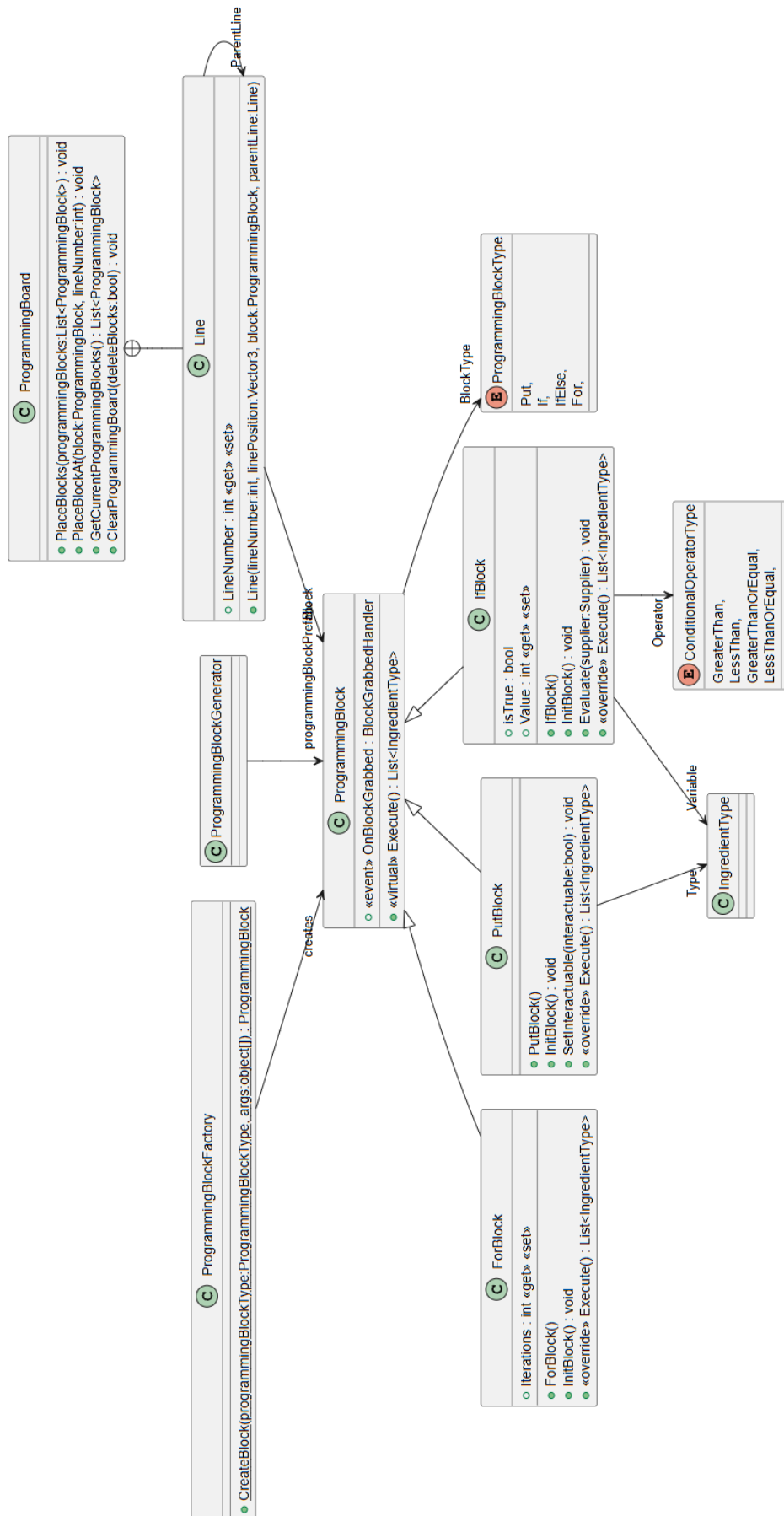


Figure 4.30: ProgrammingBlock (Fuente propia).

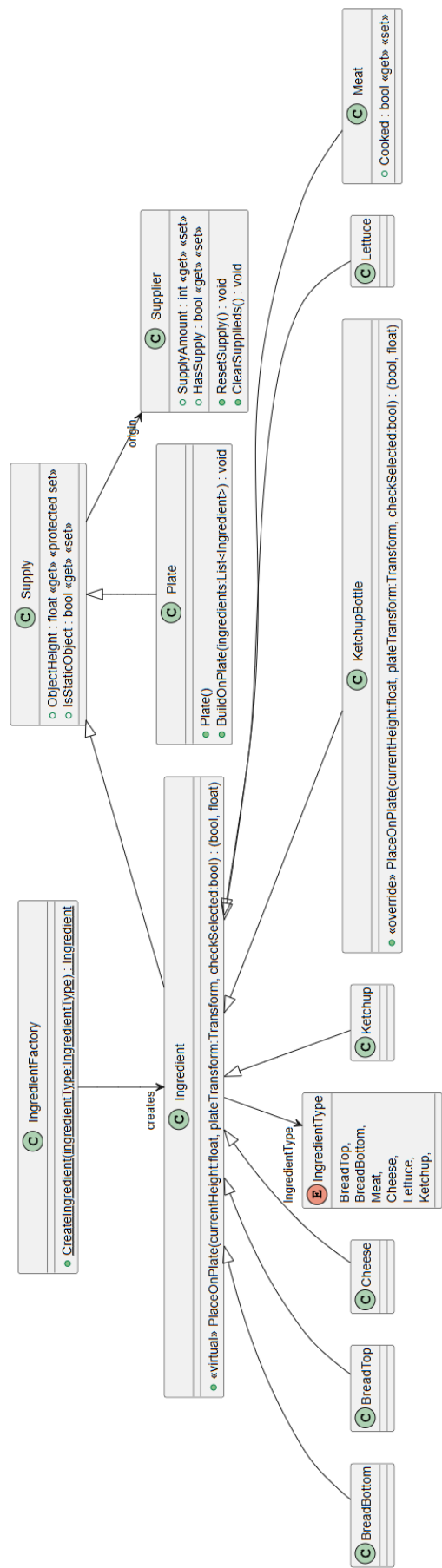


Figure 4.31: Ingredient (Fuente propia).

Chapter 5

Implementación y resultados

En este capítulo se detallan las etapas de implementación del proyecto, incluyendo la actualización del entorno de desarrollo, las herramientas y librerías utilizadas, así como el desarrollo y refactorización de las funcionalidades del videojuego. Finalmente, se presentan los resultados obtenidos durante el proceso de implementación.

5.1 Entorno de desarrollo

5.1.1 Actualización de unity

La actualización a una nueva versión de Unity trae consigo varias ventajas significativas en comparación con las versiones anteriores. Unity actualiza su motor regularmente, introduciendo mejoras en el rendimiento, nuevas funcionalidades y correcciones de errores. Por ejemplo, la actualización a Unity 2021 incluyó optimizaciones en el renderizado, mejor soporte para nuevas plataformas y herramientas avanzadas para desarrollo de realidad virtual (VR). Actualizar a una versión más reciente de Unity también asegura compatibilidad con las librerías y herramientas más recientes, como las actualizaciones del Oculus SDK y la API OpenXR. Mantener el código en una versión obsoleta de Unity puede limitar la capacidad de aprovechar estas mejoras y funcionalidades. Unity marca obsoletas las versiones anteriores, lo que significa que dejan de recibir soporte oficial y actualizaciones, aumentando el riesgo de problemas de seguridad y compatibilidad.

5.1.2 Herramientas y librerías utilizadas

En el desarrollo de nuestro juego serio en VR, hemos utilizado diversas herramientas y librerías que han sido cuidadosamente seleccionadas para maximizar la eficiencia y funcionalidad del proyecto. A continuación, se detallan algunas de las principales herramientas y librerías utilizadas, junto con la justificación de su uso.

API OpenXR

La API OpenXR es un estándar abierto para el acceso a plataformas de realidad aumentada (AR) y realidad virtual (VR). Utilizamos OpenXR en nuestro proyecto porque proporciona una interfaz unificada para el desarrollo de aplicaciones VR, asegurando compatibilidad con una amplia gama de dispositivos y plataformas.

Esta API permite desarrollar aplicaciones de VR de manera más eficiente y sostenible, ya que elimina la necesidad de escribir código específico para cada dispositivo. Además, OpenXR está respaldada por el Khronos Group, lo que garantiza su continua evolución y soporte. [5]

Flask

He elegido Flask como el framework principal para el backend debido a su simplicidad y flexibilidad. Flask permite un desarrollo rápido y eficiente, lo que facilita la creación de aplicaciones web ligeras. Su diseño minimalista proporciona la libertad de agregar extensiones según sea necesario, adaptándose perfectamente a las necesidades del proyecto. Además, Flask es bien conocido por su amplia documentación y una gran comunidad de apoyo, lo que facilita la resolución de problemas y el aprendizaje continuo.

Azure Web Application

Para el despliegue, opté por utilizar Azure Web Application. Azure ofrece un entorno de despliegue rápido y sencillo, con una integración fluida para aplicaciones construidas con Flask. La elección de Azure se debe a su capacidad para gestionar aplicaciones escalables y seguras, además de proporcionar herramientas robustas para el monitoreo y la administración de recursos. Azure Web Application soporta despliegues continuos y configuraciones automáticas, lo que agiliza el proceso de lanzamiento y actualizaciones del backend.

Gunicorn

Para manejar las solicitudes HTTP en producción, he utilizado Gunicorn, un servidor WSGI para aplicaciones Python. Gunicorn es compatible con Flask y permite manejar múltiples solicitudes simultáneamente, mejorando significativamente el rendimiento de la aplicación. Su capacidad de configuración y fácil integración con Azure Web Application lo convierten en una elección ideal para entornos de producción.

SQLAlchemy

SQLAlchemy fue seleccionado como el ORM (Object Relational Mapper) para este proyecto debido a su capacidad para simplificar la interacción con la base de datos. SQLAlchemy facilita la creación y gestión de modelos de datos, permitiendo cambiar de bases de datos sin grandes modificaciones en el código. La aplicación utiliza SQLite, pero SQLAlchemy permite una transición fluida a otras bases de datos más robustas.

SQLite

SQLite es ligera y no requiere un servidor separado, lo que facilita el desarrollo y pruebas locales. Dado que las interacciones con la base de datos son mínimas en las primeras etapas del proyecto, SQLite resulta ser una solución adecuada. Ya que permite pasarlo a producción en la misma Web Application de Azure.

5.2 Desarrollo del modo cocinero

Para entender mejor el desarrollo del modo cocinero, se han dividido las funcionalidades en dos categorías: refactorizadas e incorporadas y las no implementadas.

5.2.1 Funcionalidades Refactorizadas e Incorporadas

- **Generación nueva de pedidos:** Tal como se detalla en la sección 4.3.6, se ha refactorizado la generación de pedidos, añadiendo pedidos aleatorios.
 - Archivos implicados: `OrderGenerator.cs`, `OrderUtils.cs`.
- **Ingredientes:** Se ha aplicado una jerarquía entre los ingredientes, distribuyendo las funcionalidades.
 - Archivos implicados: `Supply.cs`, `Ingredient.cs`, clases dentro de las carpetas `Ingredient` y `Other` en `Supply`.
- **Generación de feedback:** Tal como se detalla en la sección 4.3.9, se han realizado cambios respecto al código legacy.
 - Clases implicadas: `FeedbackGenerator.cs`, `FeedbackType.cs`, `OrderUtils.cs`.
- **Contenedores proveedores:** Se ha vuelto a implementar este componente para poder servir los componentes modificados.
 - Clases implicadas: `Supply.cs`, `Supplier.cs`.
- **Plancha de la cocina:** La funcionalidad de este componente recae sobre el ingrediente `Carne`, que al interactuar con este objeto empieza a cocinarse.
 - Clases implicadas: `Meat.cs`.
- **Mostrar pseudocódigo:** Como se explica en la sección 4.3.4, se utiliza la pizarra y los bloques para mostrar el pseudocódigo.
 - Clases implicadas: Todas las clases bajo la carpeta `ProgrammingBoard`.
- **Conexión Backend:** Como se explica en la sección 4.3.10, se ha unificado la comunicación con la API.
 - Clases implicadas: `BackendConnector.cs`, `GameManager.cs`.

5.2.2 Funcionalidades No Incorporadas

- **Undo del cocinero:** Esta función no se implementó debido a priorización de otras funcionalidades. Para implementarla se deberían modificar las siguientes clases:
 - `Plate.cs`: Para gestionar los ingredientes añadidos.
 - `Supplier.cs`: Para volver a añadir el ingrediente de donde procede.
 - Se necesitaría mantener un seguimiento de los ingredientes añadidos, probablemente usando una estructura de pila.

- **Gorro cambia de color con los niveles:** Esta función no se consideró principal y no se tuvo en cuenta en el desarrollo. Para implementarla se debería:
 - Añadir los diferentes colores en `ChefRobotTV.cs`.
 - Actualizar el color del gorro en `ChefGameController.cs` durante el flujo del juego.
- **Progresión de ingredientes con los niveles:** En la versión legacy, hay ingredientes como la lechuga que se desbloquean en ciertos niveles. Para implementar esta funcionalidad se debería:
 - Modificar la forma en la que se generan los pedidos en `OrderGenerator.cs`.
 - Manejar la lógica correspondiente para ocultar el ingrediente en `Supplier.cs` o `ChefGameController.cs`.

5.3 Desarrollo del modo camarero

Al igual que en el modo cocinero, las funcionalidades del modo camarero se dividen en refactorizadas e incorporadas y no implementadas.

5.3.1 Funcionalidades Refactorizadas e Incorporadas

- **Bloques de programación:** Tal como se detalla en la sección correspondiente, se ha hecho la refactorización de los bloques de programación.
 - Clases implicadas: `ProgrammingBlock.cs`, `ProgrammingBlockType.cs`, `ProgrammingBlockF` y los bloques hijos dentro de la carpeta `Blocks` en `ProgrammingBoard`.
- **Pizarra de programación:** Tal como se detalla en la sección 4.3.4, se ha hecho la refactorización de la pizarra.
 - Clases implicadas: `ProgrammingBoard.cs`.
- **Generación de pedidos aleatorios y Generación de feedback:** Esta funcionalidad es similar a la del modo cocinero, aplicándose las mismas mejoras.

5.3.2 Funcionalidades No Incorporadas

- **If Else:** Esta funcionalidad no se implementó para simplificar el desarrollo. Para implementarla se debería:
 - Crear la clase `IfElseBlock.cs`, hacer que herede de `ProgrammingBlock` e implementar la lógica correspondiente similar a `IfBlock.cs`.
 - Crear una función para mostrar la estructura del bloque visualmente en `ProgrammingBoard.cs`.
- **Progresión de bloques con los niveles:** En la versión legacy, se desbloquea el bloque `for` después de un cierto nivel. Para implementarlo se debería:
 - Modificar la clase `OrderGenerator.cs`.
 - Ocultar el bloque `for` en `WaiterGameController.cs`.

- **Integración con el Agente Conversacional:** En la versión Legacy, tenemos un Agente Conversacional que nos ayuda a evaluar la experiencia de usuario. Para implementarlo se debería:
 - Crear una clase responsable de la gestión del agente `AgenteConversacional.cs`.
 - Añadir en la lógica del juego la aparición de este. `WaiterGameController.cs`.
- **Gorro cambia de color:** Similar al modo cocinero, pero en este caso modificando `WaiterRobotTV.cs`.

Chapter 6

Conclusiones

Este capítulo muestra las conclusiones de este proyecto y las propuestas de mejoras futuras.

6.1 Resumen y contribuciones al proyecto

Trabajar en este proyecto ha sido un reto significativo debido a la acumulación de funcionalidades de varios TFGs anteriores, lo que resultó en problemas de estructura y diseño de software. Estas dificultades hacían que entender, ampliar y hacer el código más extensible fuera muy complicado. Mi proyecto se ha centrado en realizar un refactoring exhaustivo de estos trabajos previos, lo que me llevó a la conclusión de que a veces es mejor comenzar desde cero en lugar de intentar mejorar un código obsoleto.

El refactoring ha sido una tarea increíblemente laboriosa y compleja, especialmente porque tuve que analizar y comprender el código desarrollado por varias personas, sin una estructura fija, y lidiar con errores de compilación y versiones antiguas de SDKs y Unity. Durante este proceso, instalé el proyecto en la versión Unity 2019.4.22f1, corregí errores en la versión Legacy y actualicé a la versión actual de Unity 2022.3.26f1. Además, configuré el proyecto con las tecnologías de VR necesarias, estableciendo una base sólida desde la cual trabajar.

Un desafío importante fue integrar dos modos de juego (cocinero y camarero) que estaban completamente aislados, creando componentes comunes y una estructura modular. Esto implicó implementar componentes básicos y la lógica del juego, validando que las nuevas estructuras funcionaran correctamente y fueran robustas.

Estoy muy orgulloso de cómo han quedado los diferentes componentes que he desarrollado para mantener una relación coherente entre los dos modos de juego. Creo firmemente que este proyecto podrá continuarse de manera más fácil y eficiente que anteriormente. Aunque no pude incorporar todas las funcionalidades de las versiones anteriores, estoy muy satisfecho con el resultado final. He logrado establecer una base sólida y escalable que facilitará futuros desarrollos y mejoras. La incorporación de la API de OpenXR ha mejorado significativamente la portabilidad del proyecto.

Además, uno de los aspectos más gratificantes del proyecto ha sido poder aprender y trabajar con tecnologías de realidad virtual, especialmente con el hardware de Oculus Quest 2. Llevaba años siguiendo estas tecnologías, y finalmente poder desarrollarlas ha sido una experiencia muy enriquecedora.

En cuanto a validar la portabilidad con diferentes visores VR y la realización de evaluaciones con usuarios, aunque esta área requiere trabajo futuro, he sentado las bases

necesarias para que estas pruebas se puedan llevar a cabo de manera efectiva.

En resumen, este proyecto ha demostrado que, aunque el refactoring y la reestructuración del código pueden ser tareas arduas y complejas, son esenciales para garantizar la sostenibilidad y extensibilidad de los proyectos a largo plazo. La experiencia y los resultados obtenidos no solo mejoran la calidad del código actual, sino que también proporcionan una base valiosa para futuros desarrollos en el ámbito de la realidad virtual.

6.2 Trabajo Futuro

El trabajo futuro para este proyecto se centrará en varias áreas clave que necesitan desarrollo y evaluación para alcanzar su máximo potencial. A continuación, detallo los aspectos más importantes:

- **Implementación de Funcionalidades Pendientes:** Quedan varias funcionalidades por implementar que no se han podido integrar en esta versión del proyecto. Estas funcionalidades, mencionadas en el apartado de implementación, son esenciales para completar la experiencia del usuario y el juego en su totalidad.
- **Pruebas con Usuarios:** Una parte fundamental del trabajo futuro es llevar a cabo pruebas con usuarios. Estas pruebas permitirán evaluar el rendimiento del juego en entornos simulados, recopilando datos sobre la usabilidad, la eficacia educativa y la experiencia general del usuario.
- **Mejoras en los Assets del Proyecto:** Los assets actuales del proyecto son bastante básicos y pueden mejorarse.
- **Pruebas con Diferentes Visores de Realidad Virtual:** Queda pendiente realizar pruebas del juego con diferentes visores de realidad virtual para validar la portabilidad.
- **Implementación de Nuevas Funcionalidades:** Ahora que el código ha sido refactorizado y estructurado de manera eficiente, es más fácil implementar nuevas funcionalidades.
- **Incorporación de logs para monitorizar la actividad:** Añadir logs puede ser de vital importancia para poder hacer las pruebas con usuarios.

En resumen, el trabajo futuro en este proyecto se centrará en completar las funcionalidades pendientes, realizar pruebas con usuarios, mejorar los assets del proyecto, asegurar la compatibilidad con diferentes visores de realidad virtual e implementar nuevas funcionalidades.

Appendix A

Apéndice 1 - Instalación del proyecto

A.1 Instalación del proyecto de Unity

- Descargar e instalar Unity Hub y la versión de Unity 2022.3.26f1 (ver figura A.1). Puedes descargarla desde el Archivo de Unity.

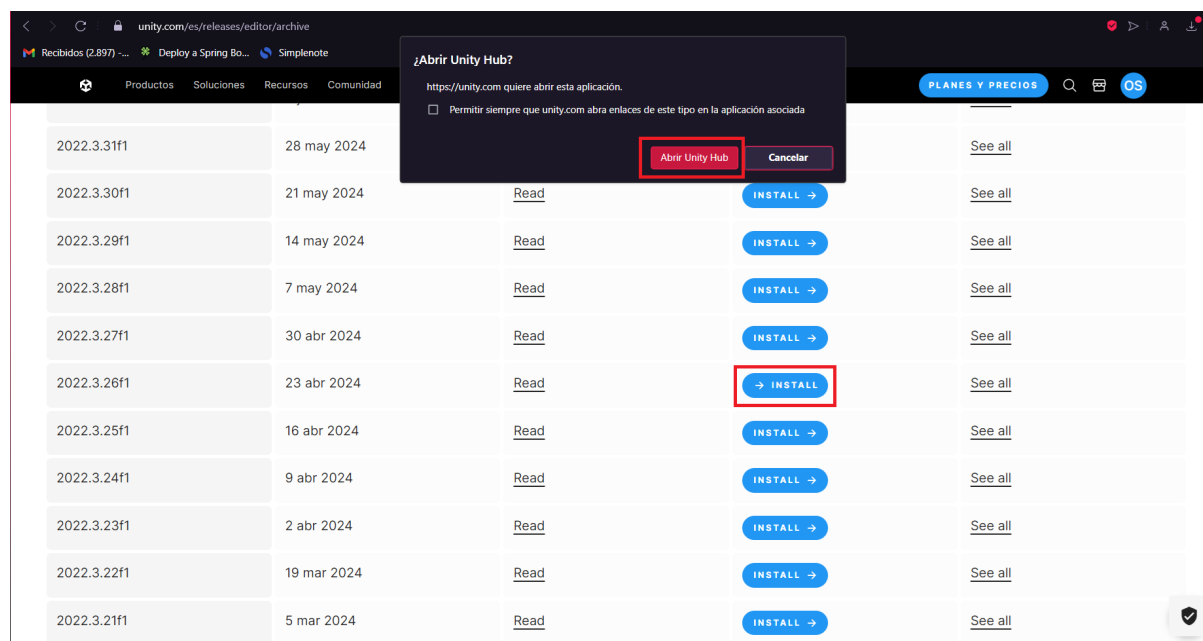


Figure A.1: Versión Unity.

- Instalar los complementos para desarrollo de aplicaciones Android (ver figura A.2), necesarios para poder realizar el build para las Oculus Quest.
- Activar el modo desarrollador en las Oculus Quest:
 1. Acceder a la aplicación de Oculus en tu teléfono móvil conectado a las gafas.
 2. Seleccionar las gafas en la aplicación y entrar en *Más configuraciones*.
 3. Activar la opción *Modo desarrollador*.
 4. Reiniciar las gafas para que el modo desarrollador se active.

Puedes encontrar más información en la página de soporte de Oculus.

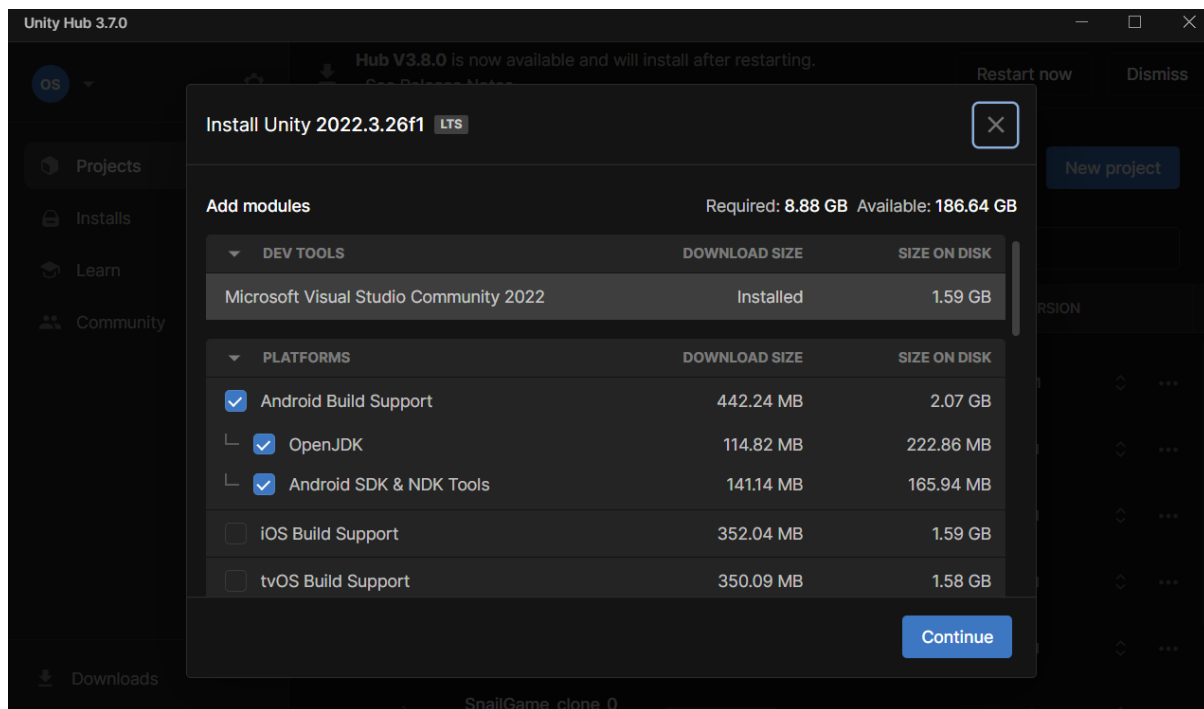


Figure A.2: Complementos Android.

- **Conectar las Oculus Quest al ordenador:**

1. Utiliza un cable USB-C para conectar las gafas al ordenador si prefieres una conexión estable.
2. Para una conexión inalámbrica, asegúrate de que tanto las gafas como el ordenador estén en la misma red Wi-Fi. Activa Air Link desde las configuraciones de las gafas utilizando la aplicación Meta Quest Link.

- **Configuración en Unity para realizar el build:**

1. Ir a *File* -> *Build Settings* (ver figura A.3).
2. Asegurarse de que la plataforma seleccionada sea *Android* y cambiarla si es necesario.
3. Pulsar en *Switch Platform* para configurar el proyecto para Android.

- **Construcción y despliegue de la aplicación:**

1. Una vez configurado, pulsar *Build* para compilar el proyecto.
2. Para instalar la aplicación en las gafas, usar SideQuest o un método similar. Conecta las gafas al ordenador, abre SideQuest y arrastra el archivo APK generado a la interfaz de SideQuest para iniciar la instalación.

A.2 Instalación del backend

Para poner en marcha el backend del proyecto, sigue estos pasos:

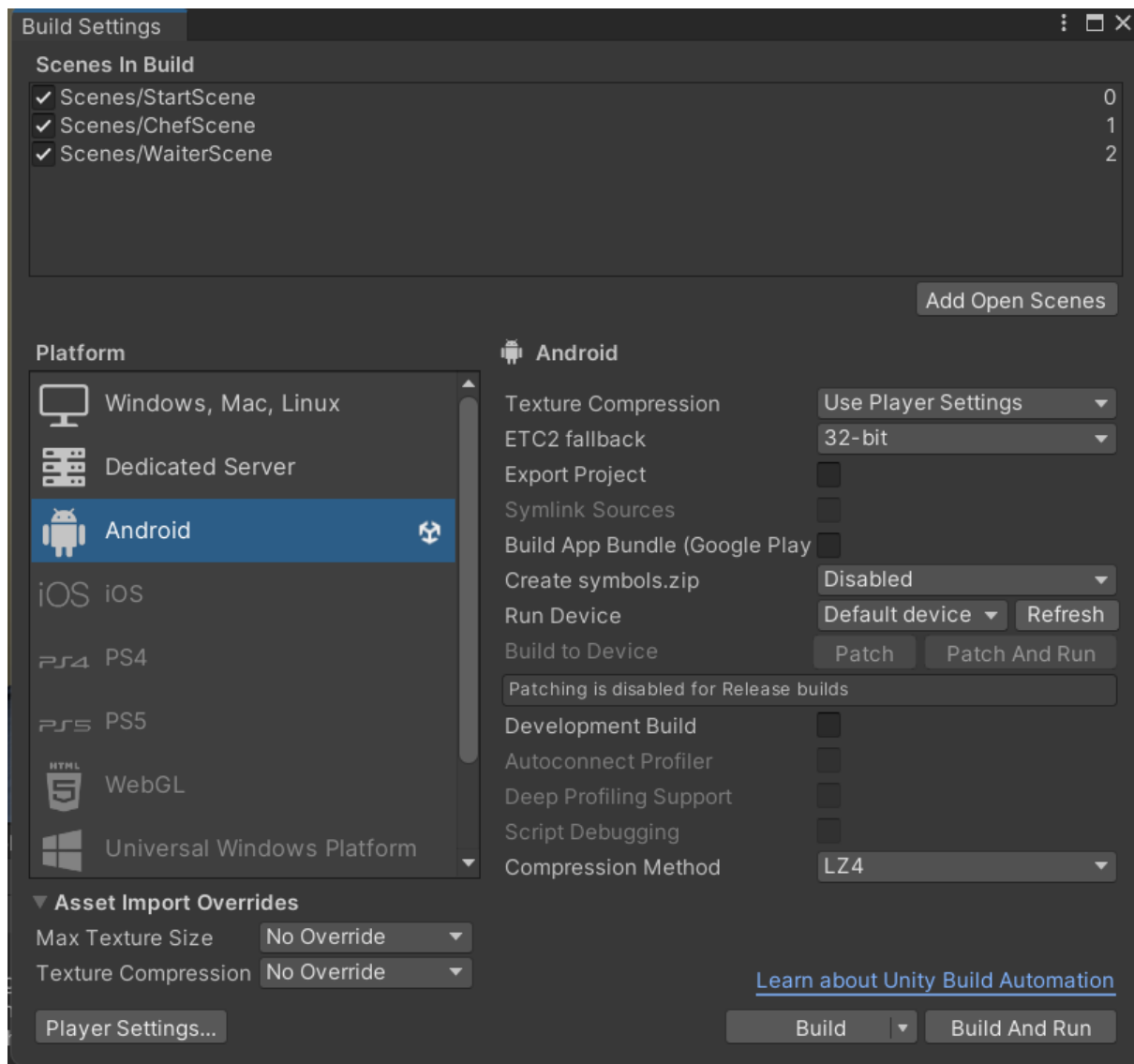


Figure A.3: Configuración de Build en Unity.

1. **Instalación de requisitos:** Ejecuta el siguiente comando para instalar todas las dependencias necesarias:

```
pip install -r requirements.txt
```

2. **Ejecución del backend:** Para iniciar el servidor, ejecuta:

```
python app.py
```

3. **Configuración en Unity:** Modifica la dirección del backend en el archivo *BackendConnector.cs* para alternar entre entornos de producción y desarrollo según necesites.

A.3 Despliegue de la aplicación web en Azure y vinculación con GitHub

Para desplegar tu aplicación web en Azure y conectarla con un repositorio de GitHub, realiza los siguientes pasos:

1. Creación de un Servicio App en Azure:

- (a) Inicia sesión en el portal de Azure.
- (b) Navega a "App Services" y haz clic en "Crear".
- (c) Elige tu suscripción y grupo de recursos.
- (d) Escribe un nombre para tu aplicación, selecciona un plan de publicación, un sistema operativo y una región.
- (e) Haz clic en "Revisar y crear" y luego en "Crear" para desplegar el servicio de aplicación.

2. Vinculación con GitHub:

- (a) Una vez que el servicio de aplicación esté operativo, ve a la sección "Deployment Center" en el menú de configuración.
- (b) Selecciona "GitHub" como fuente de tu código.
- (c) Conecta tu cuenta de GitHub y autoriza a Azure a acceder a tus repositorios.
- (d) Elige el repositorio y la rama desde donde quieres desplegar.
- (e) Configura el método de construcción y despliegue, como GitHub Actions, y sigue los pasos para configurar la acción de despliegue.

Appendix B

Apéndice 2 - Implementación del Sistema de Logging

Se ha logrado implementar como extra un sistema de logging en el proyecto de Unity para registrar eventos importantes durante el juego. Este sistema permite el seguir y supervisar al usuario en el juego, y se detalla a continuación:

B.1 Formato de los Logs

El formato de los logs sigue una estructura consistente que incluye la fecha y hora del evento, así como una categoría que indica el modo del juego. Ejemplo:

```
Log file created at 01/07/2024 13:21:09
[CHef] [01/07/2024 13:22:06]: Starting new day
[CHef] [01/07/2024 13:22:06]: Generated new order
[CHef] [01/07/2024 13:22:34]: Order delivered
[CHef] [01/07/2024 13:22:34]: Correct Order
[CHef] [01/07/2024 13:22:36]: Generated new order
```

B.2 Nombre del Fichero

El archivo de log se nombra utilizando el nombre de usuario del jugador seguido de la extensión `.txt`, lo que permite individualizar los logs por usuario. Por ejemplo, si el usuario se llama `oscar`, el archivo de log será `oscar.txt`.

B.3 Ruta del Archivo de Log

El archivo de log se guarda en la ruta persistente de datos del dispositivo utilizando `Application.persistentDataPath`. Esta ruta es específica de la aplicación y asegura que los datos sean persistentes incluso después de reiniciar la aplicación. Esta ubicación también es accesible y adecuada para almacenar datos que deben ser preservados entre sesiones.

B.4 Creación del Logger

El `Logger` se crea e inicializa en el `GameManager`, que es una instancia singleton que nunca cambia durante la vida de la aplicación. Esto asegura que solo haya una instancia del logger, proporcionando un punto centralizado para la gestión de logs.

B.5 Eventos Registrados

Algunos de los eventos importantes que se registran en los logs incluyen:

- Comienzo y final de un día en el juego.
- Generación de nuevos pedidos.
- Entregas de pedidos.
- Resultados de los pedidos (correcto o incorrecto).

Estos eventos permiten seguir el flujo de juego de una partida para supervisarla. Sería interesante añadir más eventos como la interacción del jugador con los objetos u otros, esta parte queda como trabajo a futuro para poder evaluar la eficacia del juego.

Bibliography

- [1] Gloria, A.D., Bellotti, F. and Berta, R. *Serious games for education and training*. International Journal of Serious Games. Disponible en: <https://journal.seriousgamessociety.org/index.php/IJSG/article/view/11> (Accedido el 2 de junio de 2024).
- [2] Unity. *Plataforma de desarrollo en tiempo real de Unity — Motor de VR, AR, 3D y 2D*. Disponible en: <https://unity.com/es> (Accedido el 31 de mayo de 2024).
- [3] Unity Learn. *Learn game development w/ unity: Courses & tutorials in game design, VR, AR, & real-time 3D*. Disponible en: <https://learn.unity.com/> (Accedido el 20 de febrero de 2024).
- [4] GitHub. *Let's build from here*. Disponible en: <https://github.com/> (Accedido el 10 de mayo de 2024).
- [5] Khronos Group. *OpenXR*. Disponible en: <https://www.khronos.org/openxr/> (Accedido el 25 de abril de 2024).
- [6] Jeyakeerthan. (2018). *Factory Method Design Pattern*. Medium. Disponible en: <https://medium.com/@Jeyakeerthan/factory-method-design-pattern-d099d46c72b7> (Accedido el 3 de junio de 2024).
- [7] Refactoring.Guru. *Refactoring and Design Patterns*. Disponible en: <https://refactoring.guru> (Accedido el 3 de junio de 2024).
- [8] Lightbot. *Lightbot*. Disponible en: <https://lightbot.com> (Accedido el 3 de junio de 2024).
- [9] Steam. *Job Simulator*. Disponible en: https://store.steampowered.com/app/448280/Job_Simulator/ (Accedido el 3 de junio de 2024).
- [10] Steam. *The Farmer Was Replaced*. Disponible en: https://store.steampowered.com/app/2060160/The_Farmer_Was_Replaced/ (Accedido el 3 de junio de 2024).
- [11] GeeksforGeeks. *C# Programming Language*. Disponible en: <https://www.geeksforgeeks.org/csharp-programming-language/> (Accedido el 3 de junio de 2024).
- [12] Casanova Torrequebrada, M. (2023). *Waiter Code: Videojuego en Realidad Virtual para Aprender a Programar*. Handle Proxy. Disponible en: (Accedido el 5 de marzo de 2024).

- [13] Gómez Jiménez, I. (2020). *Cooking code: Juego Para Aprender a programar con Realidad Virtual*. Handle Proxy. Disponible en: <http://hdl.handle.net/2445/172160> (Accedido el 5 de marzo de 2024).
- [14] Fuentes Raventós, A. (2021). *Restaurant code: Juego en Realidad Virtual para aprender a programar*. Handle Proxy. Disponible en: <http://hdl.handle.net/2445/182271> (Accedido el 7 de marzo de 2024).
- [15] Wang, C. (2022). *A conversational agent that evaluates user experience in a Virtual Reality game*. Handle Proxy. Disponible en: <http://hdl.handle.net/2445/198240> (Accedido el 25 de abril de 2024).
- [16] YouTube. *Fist Full of Shrimp*. Disponible en: <https://www.youtube.com/@FistFullofShrimp> (Accedido el 1 de junio de 2024).
- [17] Flask. *Flask Documentation*. Disponible en: <https://flask.palletsprojects.com/en/2.3.x/> (Accedido el 1 de junio de 2024).
- [18] SQLAlchemy. *SQLAlchemy Documentation*. Disponible en: <https://docs.sqlalchemy.org/en/14/> (Accedido el 1 de junio de 2024).
- [19] Azure. *Microsoft Azure Documentation*. Disponible en: <https://docs.microsoft.com/en-us/azure/> (Accedido el 1 de junio de 2024).
- [20] Gunicorn. *Gunicorn Documentation*. Disponible en: <https://docs.gunicorn.org/en/stable/> (Accedido el 1 de junio de 2024).