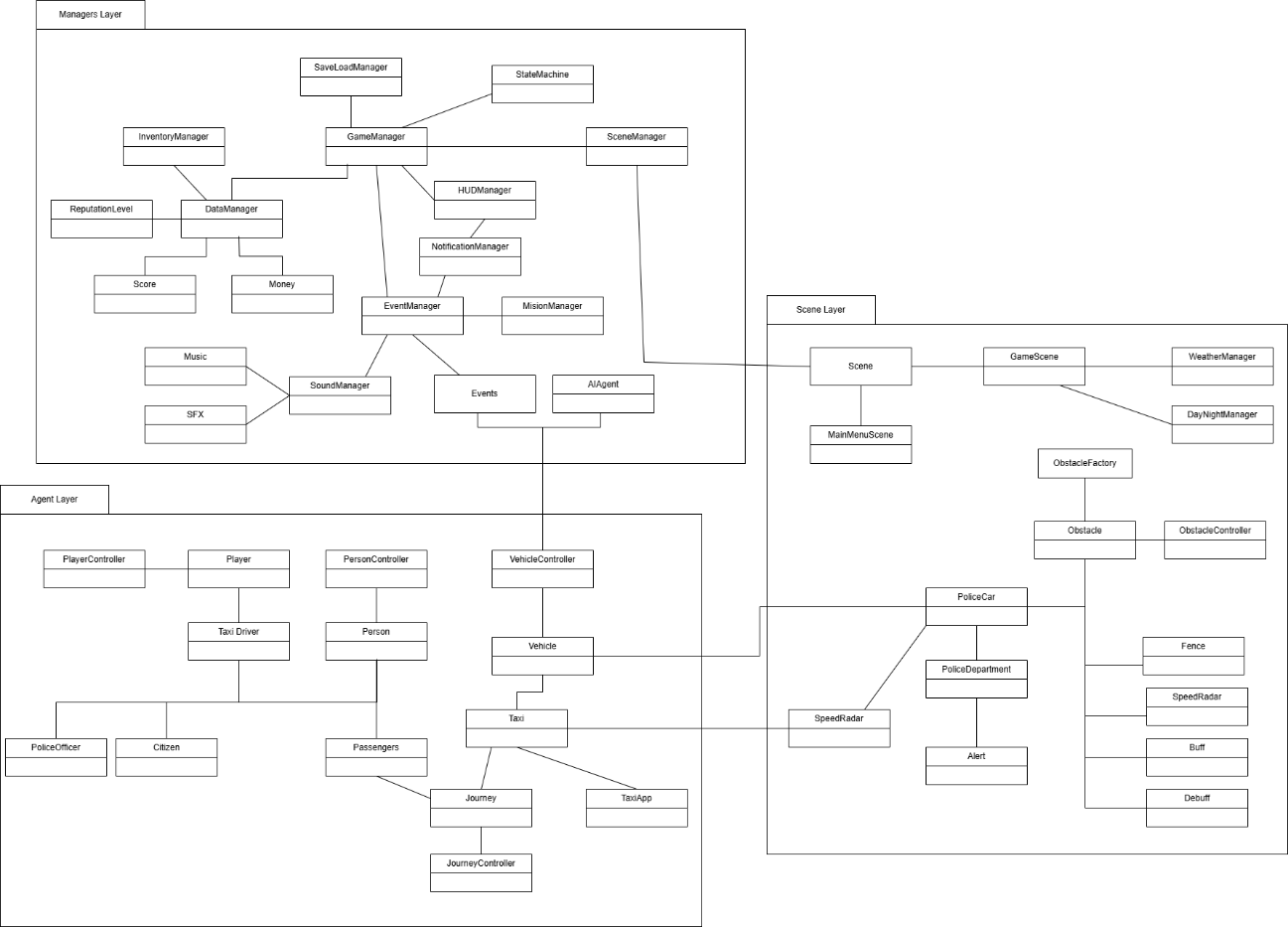
**Proyecto Final – Paradigmas y Técnicas de Programación**

Javier Ahumada y Rodrigo Covas

# Informe Previo:

**Diseño de Arquitectura**



**Arquitectura**

La arquitectura elegida para el programa sigue el patrón de Modelo-Vista-Controlador (MVC), que divide la lógica del programa en tres componentes principales.

El modelo está formado por las clases y scripts que representan el estado del juego y su lógica, como el taxi, los obstáculos o los datos del juego (DataManager y SaveLoadManager).

La vista está formada por las interfaces de usuario (HUDManager y NotificationManager) y los gráficos y animaciones.

El controlador se encarga de las interacciones entre el modelo y la vista, manejando la interacción entre el jugador y el juego y actualizando el estado en consecuencia. A esta componente pertenecen GameManager, que coordina los estados del juego; EventManager, que se encarga de gestionar los eventos entre diferentes partes del sistema; AIAgent, que controla la lógica de la IA; y todos los controladores de vehículos, personas, trayectos o el jugador.

Mantener las distintas componentes separadas nos permiten separar las distintas partes del desarrollo del programa, de tal forma que se pueda desarrollar primero un modelo sin necesidad de vista (utilizando salidas de texto) ni controlador (utilizando una simulación ya preparada). En nuestro caso, este modelo ya ha sido desarrollado en las anteriores prácticas de la asignatura, y cualquier cambio en el funcionamiento del juego se puede desarrollar directamente sobre él. Esto nos demuestra como el modelo MVC es muy útil para modularizar las distintas partes del programa de forma que puedan ser desarrolladas y probadas por separado sin que los cambios en una componente afecten directamente a las otras

**Patrones de diseño**

En el programa hemos utilizado algunos de los patrones de diseño estudiados en la asignatura.

Por ejemplo, el patrón Singleton en el GameManager, el EventManager o el SoundManager entre otros; solo existe una instancia de cada manager de forma que son accesibles desde cualquier parte del programa, y mantienen un estado compartido para evitar incoherencias.

También utilizamos el patrón Factory utilizado en ObstacleFactory, que nos permite crear distintos tipos de obstáculo sin tener que instanciar cada uno manualmente.

El patrón Observer se utiliza por ejemplo en EventManager, que permite que distintos componentes como el HUDManager reaccionen a eventos del juego sin estar directamente acoplados entre sí.

El patrón State se utiliza en StateMachine para manejar los distintos estados del juego como “en pausa”, “jugando” o “terminado”. El AIManager, que sirve para definir las estrategias y comportamientos de los distintos elementos, como los coches de policía, es un ejemplo del uso del patrón Strategy.

**Historias de usuario**

1. Como taxista, quiero llevar a los pasajeros a sus destinos en el tiempo indicado para maximizar mis ganancias.

Necesito implementar un temporizador y un sistema para ganar y gastar el dinero.

2. Como taxista, quiero evitar los radares y a la policía para no recibir multas y poder completar mis trayectos sin problemas.

Necesito programar la IA policial, así como los radares y sus condiciones de detección.

3. Como taxista, quiero mantener mi taxi en buenas condiciones y la comodidad del pasajero alta para evitar perder el juego y ganar propinas.

Necesito crear una barra de vida y una de confort, cambiando sus valores con las colisiones.

4. Como taxista, quiero recolectar buffs para recuperar vida y recibir mejoras temporales.

Necesito implementar un sistema de ítems recolectables, así como uno de efectos del taxi.

5. Como taxista, quiero poder gastar mis ganancias en mejoras para el taxi para tener un vehículo mejor y más eficiente.

Implementar una tienda de mejoras, así como un inventario desde el que gestionar las mejoras equipo al taxi. Implementar los distintos comportamientos y aspecto del taxi según las mejoras equipadas.

# Informe Posterior:

**Enlace al Repositorio de Git:** <https://github.com/RodrigoCovas/Taxi-Drive/>

El juego final cuenta con 4 escenas: el menú principal, los ajustes, MyRide (que hace referencia a las distintas funcionalidades del taxi) y el juego en sí. A lo largo del proyecto, hemos usado una gran variedad de scripts y otros assets, algunos hechos por nosotros y otros importados desde <https://assetstore.unity.com/> . A continuación, os dejamos la estructura final de nuestro proyecto en la forma de un UML.

*(Poner UML)*

**Arquitectura**

Durante el desarrollo del juego, nos enfrentamos a 4 fases distintas en el desarrollo: el funcionamiento de los vehículos, la ciudad y cómo aparecen objetos en ella, la implementación del sistema de confort y de tiempo límite y la creación de buffs y debuffs, y el UI del juego y los menús.

*Desarrollo de los vehículos*

La fase inicial fue principalmente aprender a utilizar las físicas de Unity para implementar las distintas funcionalidades de los vehículos. Para el diseño inicial del coche utilizamos un prefab Unity Asset Store que separaba las ruedas y el chasis del coche. Empezamos por implementar diferentes funcionalidades como acelerar, frenar e implementar los colliders necesarios en las ruedas para que se moviera correctamente. Después, pasamos a añadir una funcionalidad de derrape para los vehículos. Consideramos que la fricción lateral añade una dinámica al juego que no tenía.

Después de ajustar los diferentes parámetros en el script VehicleController, pasamos a desarrollar dos scripts distintos dependiendo del tipo de coche: el primero es el PlayerController que se encarga de buscar diferentes inputs para la funcionalidad del taxi, y el segundo es el AIController que se encarga de seguir al jugador y es para los policías. Ahora era necesario implementar efectos audiovisuales a nuestros coches.

A ambos modelos de coche, Taxi y PoliceCar, se le añadió luces frontales y de freno. Para ello, usamos ParticleSystems para replicar el efecto que tendría la luz en la vida real. Por otro lado, ambos coches necesitaban sonido y para ello implementamos un script llamado CarSounds. Este script se encarga de ajustar el pitch de un audio del motor para dar el efecto que el coche se está moviendo. Después pasamos a añadir los efectos visuales para los derrapes. Para ello, añadimos un nuevo GameObject al Taxi: wheelEffects. Con estos nuevos GameObjects implementamos humo y marcas en el suelo utilizando TrailRenderers y ParticleSystems.

*Desarrollo de la ciudad: puntos de spawn y minimapa*

Una vez que los vehículos ya tenían una funcionalidad, empezamos a preocuparnos por el entorno. Para ahorrar tiempo, decidimos que era mejor importar una ciudad de Unity Asset Store. Con los aspectos visuales y los colliders necesarios en las carreteras, empezamos a añadir aspectos básicos y necesarios para el Game Loop: zonas de incio y fin de trayecto y un GPS para guiar al jugador.

Decidimos implementar esferas de diferentes colores para marcas ambas zonas. La funcionalidad básica de las esferas las implementamos en TimerSphereTrigger: implementamos que el jugador se tiene que mantener varios segundos dentro de la esfera para destruirla y empezar/terminar el trayecto. Después, necesitábamos spawnear las esferas en zonas aleatorias por el mapa.

Primero, creamos un script llamado RandomRoadPoint que se encarga de buscar puntos de manera aleatoria en una carretera usando el mesh collider de la carretera. Usando un NavMeshSurface fue posible determinar las zonas conducibles en el mapa. El NavMeshSurface se creó a partir del collider de las carreteras. Con esta superficie, es posible sacar el punto más cercano al punto generado aleatoriamente en RandomRoadPoint. Esto fue necesario porque muchas veces los puntos que pertenecían a los colliders no eran accesibles con los vehículos. Con esto, empezamos a generar puntos aleatoriamente por el collider. Sin embargo, la mayoría del collider de nuestra ciudad era en la autopista y muchas veces se tardaba demasiado tiempo en llegar a esos puntos. Entonces, implementamos un nuevo modo de búsqueda de coordenadas: buscar puntos cerca del jugador. Para ello, buscábamos puntos a una distancia y a un ángulo aleatorios. Posteriormente, buscábamos si existía algún punto conducible cerca.

Con los modos de spawn creados, lo implementamos para las esferas de inicio y fin de viaje en SphereSpawner. Como queríamos que se spawnearan secuencialmente, usamos un Couritine para que la esfera del final solo se spawneara una vez que la de inicio se destruyera.

Finalmente, para guiar al jugador creamos un minimapa. Este minimapa es una cámera con vista desde arriba que va siguiendo al jugador con un script Minimap. Este minimapa necesitaba un GPS para guiar al jugador a su objetivo. Para ello utiliza tags para identificar las esferas y con un line renderer dibuja el camino. Este camino es generado a partir del NavMeshSurface. Para dibujar el camino a la esfera, el line renderer utiliza shaders. El camino dibujado es solo para ser visto en el minimapa, o sea que es necesario implementar esto en un nuevo layer llamado Minimap. Este layer se hace invisible en la cámara principal cambiando el culling mask de la misma. Finalmente, queremos que el jugador sepa la distancia hasta el objetivo. Esta se calcula en un script llamado CalculatePathDistance y lo hace sumando la distancia entre todos los puntos que forman la línea del GPS.

*Desarrollo del gameloop: sistema de puntos, buffs y debuffs y aparición de eventos*

Para que el juego no se limitase a ir de un punto a otro, hacía falta un sistema de puntos que recompensase al jugador según lo bien que ha llevado al pasajero. Para ello, se implementaron un sistema de confort del pasajero y un temporizador. Según el confort y el tiempo restantes, se le dan más puntos al jugador, pudiendo llegar a no tener puntos si tanto el confort como el tiempo se han agotado.

El confort no baja siempre que el jugador se choque con algo, puesto que eso vuelve el juego más lento y aburrido. Por ello, el confort del pasajero solo va a bajar cuando un coche de policía consiga chocar con nuestro taxi. Por tanto, tenemos que detectar cuándo el jugador se choca con un vehículo de policía y no otra cosa. Para ello utilizamos las etiquetas de Unity, que nos permite asignar la etiqueta “PoliceCar” a todos los coches de policía (en este caso se puede asignar directamente al prefab) y reducir el confort sólo cuando colisionemos con un objeto con esa etiqueta.

Para hacer el juego más entretenido, creamos unos cuantos objetos consumibles intentando que sus funciones fueran lo suficientemente variadas. El BuffSpawner, que es la factoría que genera estos objetos consumibles, elige entre cada uno de los objetos de manera aleatoria cada vez que hacemos aparecer un objeto consumible, añadiendo una componente de suerte al juego. Todos ellos tienen colliders trigger, es decir, no interactúan físicamente con el entorno y solo sirven para detectar si algún objeto ha entrado en contacto con el collider. Para que el taxi identifique con qué objeto está entrando en contacto y pueda aplicar su efecto correspondiente hemos utilizado etiquetas de Unity. Todos estos objetos consumibles desaparecen al entrar en contacto con el jugador, esta lógica se encuentra en el script BuffDespawner que se ha incluido en cada uno de los prefabs.

En primer lugar, tenemos unos objetos beneficiosos que nos permiten sumar tanto tiempo (reloj) como confort (refresco), que básicamente llaman a las funciones AddTime y Heal de EventTimer y PlayerConfort cuando el jugador choca con ellos. Como objetos perjudiciales tenemos en primer lugar una cámara de seguridad que al ser recogida por el jugador llama a la función SpawnPoliceCar del PoliceSpawner tres veces, haciendo aparecer tres coches de policía en el mapa. Otro objeto perjudicial es el espejo, que al ser recogido por el jugador invierte los valores de los controles detectados por PlayerController.

Finalmente, tenemos que hacer que los coches de policía y los objetos aparezcan en el mapa. De la factoría de objetos BuffSpawner ya hemos hablado. La factoría de policías PoliceSpawner tiene algunas funcionalidades extra además de hacer aparecer coches de policía. A esta factoría se le ha impuesto un límite de coches de policía que puede haber en el mapa a la vez, para evitar que los policías saturen al jugador y este no se pueda ni mover. Además, se ha hecho una función de DespawnRandomPoliceCar que hace desaparecer a un policía aleatorio del mapa. Esta función se puede activar presionando el botón O, y su propósito es hacer el juego más fácil si el jugador lo desea o permitir salir de situaciones en las que la policía te supera. También se puede hacer aparecer más policía si se desea que el juego sea más difícil, presionando el botón P, aunque los policías aparecidos no podrán superar el límite establecido.

Los coches de policía son derrotados cuando se chocan con un muro y son incapaces de seguir moviéndose, o cuando el jugador se separa lo suficiente de ellos para evitar que le sigan siguiendo. Para evitar tener coches de policía quietos por el mapa, se ha creado un PoliceDespawner que hace desaparecer a los coches si se quedan quietos mucho tiempo y se encuentran lo suficientemente lejos del jugador.

Para que continuamente estén apareciendo objetos consumibles y coches de policía en el mapa, se ha creado el EventSpawner, que cada cierto intervalo aleatorio de tiempo (dentro de unos límites establecidos) hace aparecer un objeto consumible el 95% de las veces y un coche de policía en las ocasiones restantes.

*Desarrollo del UI y menús*

Finalmente, implementamos el UI y otras funcionalidades en los menús del juego. Para ello, empezamos por mostrar por pantalla el feedback del juego: la velocidad del coche con un script Speedometer, sprites que dicen si las luces frontales están encendidas, el minimapa con un RenderTexture y texto indicando la distancia al destino. También añadimos una barra de confort, que va disminuyendo según el confort actual del pasajero; y un temporizador que muestra el tiempo restante para completar el trayecto. Ambos elementos se activan en la escena solo cuándo se comienza un trayecto y se desactivan al terminarlo.

Era también importante hacer un sistema que permitiera mostrar mensajes en pantalla, para poder mostrar al jugador información sobre lo que está ocurriendo o lo que tiene que hacer. Para ello se creó el SystemMessages, que permite a cualquier objeto mostrar mensajes en pantalla con una duración determinada.

Por otro lado, era necesario tener distintos menús/escenas. El menú principal era desde donde se accede a todos los demás y utiliza principalmente un canvas para enseñar los diferentes botones e imágenes del UI. Las funcionalidades se crearon en un script llamado MenuButtons que se encarga de ir cambiando de escenas. Consideramos que la música es una parte importante del juego. Para ello, creamos un Audio Manager como un singleton que se encarga de poner música y efectos de sonido al juego. Para la música creamos una playlist con 10 canciones y los efectos de sonido son solamente el sonido de hacer click en un botón.

Después, se creó el menú de ajustes que da la opción de cambiar la resolución, la calidad de los gráficos, activar el fullscreen y ajustar el volumen de la música. Para esto, creamos dropdowns, un toggle y una slider para ajustar estas distintas características. Sin embargo, cuando probamos cambiar la resolución del juego, las componentes de la UI se empezaron a mover de maneras extrañas. Entonces, modificamos el programa para que cuando la resolución cambiara, los componentes se estiraran en vez de quedarse igual. Además, tuvimos que ajustar las anclas de los componentes para que se comportaran como queríamos.

Posteriormente, se creó un menú de pausa para la escena del juego. Este menú es un canvas separado que se activa con el botón ESC y para el tiempo del juego. En este, podemos acceder cualquier otra escena o resumir la partida.

Finalmente, en la escena de MyRide, creamos una imagen con las instrucciones del juego y las diferentes funcionalidades que este ofrece para guiar al jugador y mejorar su experiencia.

**Patrones de diseño**

*Singleton*

Este patrón está presente en el AudioManager. Esto fue necesario para que cuando se creara en el menú principal, se mantuviera cuando el jugador cambie de escenas. Además, este se encarga de la música y otros SFX, que es útil en todas las escenas. Como queríamos que la música se mantuviera mientras cambias de escenas y juegas, usar un singleton fue la mejor opción.

*Factoría*

En nuestro proyecto hemos utilizado el patrón de factoría en 3 ocasiones distintas. Utilizamos una factoría para crear las esferas de inicio y final de cada viaje, para crear los coches de policía, y para crear los objetos consumibles (buffs/debuffs). El patrón de factoría nos permite hacer aparecer objetos en la escena a partir de un prefab tantas veces como queramos a lo largo del juego, sin necesidad de instanciar todos los objetos en la escena desde el principio.

En un diseño ideal, todas estas factorías heredarían de una clase abstracta Factoría que implemente la función “Random Road Point”, que es la que nos permite encontrar puntos válidos para aparecer las cosas en el mapa y utilizan todas las factorías. Sin embargo, el funcionamiento de Unity nos obligó a añadir el script de “Random Road Point” a cada una de las factorías por separado, por lo que la factoría padre dejó de tener sentido.

*Bridge*

Utilizamos el patrón Bridge al crear los vehículos del juego. En ellos, la lógica de como el vehículo y sus físicas interactúan con el juego está desacoplada de cómo se controla el vehículo. Las físicas de los vehículos las controla el script “Vehicle Controller”, las luces las controla “Car Lights” y los sonidos del coche los controla “Car Sounds”. Todos estos scripts definen una interfaz común para interactuar con los vehículos, como acelerar, frenar, derrapar o encender las luces.

Mientras, los controles del vehículo los define “Player Controller”, que recibe el input del jugador y lo convierte en instrucciones para el controlador del vehículo; y “AI Controller” en el caso de los vehículos controlados por la máquina, que calcula las mejores decisiones para el vehículo y se las manda al controlador del vehículo.

*Facade*

El patrón Facade se utiliza constantemente a lo largo del proyecto. La idea del patrón es esconder lógicas complejas detrás de funciones fáciles de utilizar.

Un ejemplo muy sencillo es la interfaz de “System Messages” que permite escribir cualquier mensaje en pantalla mediante la función WriteMessage(mensaje a mostrar, duración en pantalla) y así ocultamos los detalles de cómo y dónde se renderiza el mensaje en pantalla.

Otro ejemplo es la clase “Police Car Spawner”, que tiene la función “Spawn Police Car” que permite hacer aparecer un coche de policía en cualquier momento, sin necesidad de tener en cuenta la lógica de dónde aparece el coche o cómo se le asignan distintos componentes al objeto creado. En nuestro proyecto se utiliza por ejemplo con el objeto de la cámara de seguridad, que al ser recogido por el jugador hace aparecer a tres policías más en el mapa.

Finalmente, observamos el uso de Facade en el sistema de confort y el temporizador del viaje. El primero tiene unas funciones de “Heal” y “Take Damage” para modificar el valor de la barra de confort, y el segundo tienela función de "Add Time”. Además, ambos sistemas tienen una función de “Change State” que permite fácilmente activar y desactivar estos sistemas, como cuando se empieza y acaba un trayecto. El sistema de puntos también tiene una función “Add Points” para que cualquier evento pueda sumar puntos al jugador de forma sencilla.