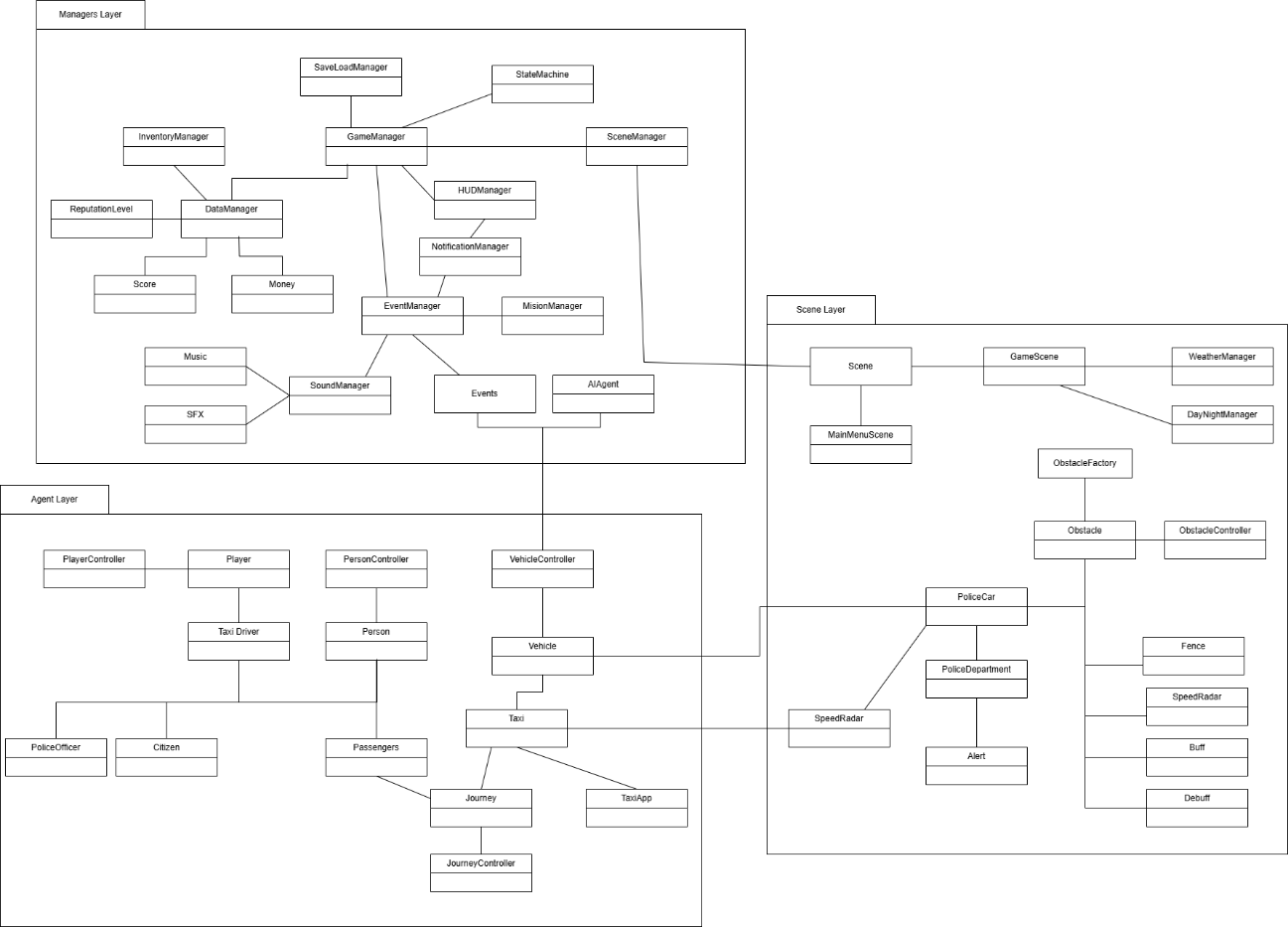
**Proyecto Final – Paradigmas y Técnicas de Programación**

Javier Ahumada y Rodrigo Covas

**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.