Resumen

Implementación de principios S.O.L.I.D. en aplicación de ejercicio cognitivo.

Contexto

Unity es un motor gráfico, que, haciendo uso de el lenguaje de programación C#, habilita la implementación de comportamientos a los objetos en escena. C# por su parte es un lenguaje enfocado desde su inicio a la *POO* (de hecho, cada programa es en sí una clase, con atributos y métodos que definen su comportamiento).

Las características fundamentales de la *POO*, frente a la convencional programación procedural, son:

- Abstracción
- Encapsulamiento
- Herencia
- Polimorfismo

Descripción

En el ámbito en el que se desarrolla este proyecto, se presupone la certeza de que se trata de un proyecto de largo desarrollo. Ello implica la obligatoriedad de desarrollar código atendiendo a las características resaltadas a continuación:

- Reutilización de código
- Extensibilidad
- Mantenimiento seguro

Análisis

Un enfoque para solucionar este problema podría basarse en la aplicación de una serie de principios. Estos principios se denominan S.O.L.I.D. por sus siglas, y se popularizó a principios de los 2000.

- ***Single Responsability Principle:*** principio de responsabilidad única. El enfoque de este principio es el encapsulamiento y la división de programas en entidades miniaturizadas. De esta manera, los programas desarrollados pasan a tener una estructura más clara y depurable.

- ***Open/Closed Principle:*** este principio establece que un programa, tras haber sido testeado y validado, no debe estar abierto a la modificación, pero sí a la ampliación. Esto tiene sentido, pues al modificar un programa corremos el riesgo de modificar le comportamiento ya testeado previamente.

- ***Liskovs Substitution Principle:*** este principio establece que, en caso de emplear *herencia*, se debe habilitar la sustitución de un tipo derivado por el tipo básico, garantizando el funcionamiento del programa. Este principio no ha sido empleado de momento, pues se considera mejor una relación *tiene- un* a *es-un.* La herencia conlleva desventajas como el *coupling*, gran desventaja al depender estrechamente de una *superclase*.

- ***Interface Segregation:*** este principio establece que, en caso de emplear *interfaces* e implementarlas en *subclases*, cada interfaz debe implementar lo que precise la subclase. Por tanto, en caso de existir una gran *interfaz* que es implementada por múltiples *subclases* que implementan algunos métodos, es preferible dividir el *interfaz* en varias *interfaces*.

- ***Dependency Inversion Principle:*** este principio es consecuencia de la necesidad de implementar una jerarquía coherente con lo primitiva que es cada clase que se emplea. De esta forma, la *superclase* debe ser siempre capaz de contener todas las posibilidades que deriven sus respectivas *subclases*.

Aplicación

Atendiendo a lo descrito anteriormente, se procede a la aplicación de dichos principios, al enfoque rígido y interdependiente previamente desarrollado. Los cambios han sido aplicados en múltiples facetas.

Así pues, las categorías de los programas que fueron adaptados son:

- Inputs: acelerómetro, drag.
- UI: colores de botones, Modificador de Paletas, Renderizador de sprites.
- De propósito general: Administrador de paneles, administrador de pantallas, administrador de puntuaciones.
- Multimedia: ratio de Pantalla, Cámara, Explorador de Archivos.
- Juegos: Detector de colisiones.
- Web: Firebase.

Se muestra a continuación el diagrama UML que define el alcance de la implementación. Se puede observar cómo se han respetado los mencionados principios, garantizando un código robusto, genérico y mantenible.

Diagrama UML (General)