**Universidad de Costa Rica**

**Diseño de Software - CI0136**

**Prof. Alan Calderón Castro**

**II Ciclo 2020**

**Integrantes:**

- **Sebastián Alfaro L - B60210**
- **Juan Garro N - B83249**
- **Juan Manuel Torres - A96252**
- **Lester Cordero M - B62110**
- **Eddy Ruiz Soto - B76880**

---

# JUSTIFICACIÓN DISEÑO MAZE ENGINE FRAMEWORK DE BATTLE CITY

Para el diseño del MARDA, se decidió hacer uso de ciertas clases que van a formar la parte general del diseño, de las cuales van a heredar las clases específicas para el juego de Battle City. Al inicio se discutió junto con los otros grupos de proyectos, una solución que relacionaba los aspectos más generales de cada uno de los juegos. Dicha discusión fue de gran importancia para el diseño final.

Las clases que forman parte del MARDA, están hechas para que se puedan heredar clases para poder desarrollar algún juego similar. Con el uso de clases abstractas como:

**ME_Entidad**: Clase abstracta para poder heredar tipos de entidades del juego, contiene atributos y métodos base para poder implementar de manera sencilla clases específicas.

**ME_Controlador**: Clase abstracta que contiene el ciclo principal del juego, el cual es el funcionamiento que todo juego debe tener.

**ME_Laberinto**: Clase abstracta que tiene componentes base para poder generar los distintos laberintos o niveles del juego. Aquí se conforma el laberinto deseado para el juego y su composición con las distintas entidades.

**ME_Menu**: Clase abstracta para contener un estado de los distintos menús que pueda tener un juego, como su menú principal, así como los controles y reglas del juego.

**ME_Avatar**: Clase que hereda de Entidad, la cual tiene componentes generales para la creación de distintos tipos de avatares para un juego.

**ME_Avatar_Controlable**: Clase que hereda de Avatar, la cual tiene los componentes que cualquier avatar debe tener para que se pueda controlar por el jugador.

**ME_Avatar_Autonomo**: Clase que hereda de Avatar, funciona como base para cualquier avatar que tenga una inteligencia artificial, la cual permite que se herede una entidad y esta pueda moverse libremente.

**ME_Arma**: Clase que hereda de Entidad, la cual tiene componentes generales para la creación de las distintas armas que puedan utilizar las entidades creadas.

Con la implementación de estas clases, es posible construir algún juego que posea estas características; donde existan distintos tipos de laberintos, avatares controlables y no controlables por el jugador, armas y otras entidades como bloques que permitan generarse en el laberinto.

Para demostrar la funcionalidad del diseño, se implementó el juego Battle City a partir del MARDA creado.

Para justificar mejor el diseño de la estructura conformada por dichas clases nos vamos a referir a algunos principios SOLID y patrones de diseño que se aplican en él:

## Patrones de diseño

1. **Patrones creacionales**
   a. **Modelo Vista Controlador:**
      Al hablar de un juego gráfico podemos hablar de la implementación MVC. La vista es una ventana PyGame que muestra los gráficos en el monitor. El controlador es el teclado que controla el usuario, manejado por el módulo **pygame.event()** facilitado por Pygame.

2. **Patrones estructurales**
   a. **Patrón Decorador:**
      Este patrón nos permite añadir funcionalidad o características a un objeto. Y esto es exactamente lo que realizamos en nuestro diseño. Una de las principales clases del MARDA en nuestro diseño es **Entidad**, de ella se derivan múltiples clases distintas, siguiendo las distintas subclases podemos encontrar una clase **TanqueJugador**, propia del juego que se implementó. Esta última clase posee ciertos atributos o funcionalidades que necesitan ser agregados para ello de la clase **Entidad** heredan otras clases (poderes), propias del juego particular, que se encargan de decorar el tanque del jugador.

3. **Patrones de comportamiento**
   a. **Patrón Método Plantilla:**
      Define el esqueleto de un método, permitiendo que algunas subclases continúen con la tarea. En nuestro diseño podemos identificar este comportamiento por ejemplo en la clase **ME_Laberinto** en el método **draw()**, que dibuja en pantalla, pero también permite que otras clases, como por ejemplo **BattleCity_Laberinto**, dibujen sus particularidades**.**

Esta se encarga de dibujar en la pantalla pero permite que otras clases dibujen sus particularidades. Otro ejemplo de igual funcionalidad es el método además se observa en los métodos como en los métodos **update()** de la clase **BattleCity_Laberinto.**

## Principios SOLID

1. S – Principio de responsabilidad única (SRP)

   Este principio indica que una clase o componente posee una sola responsabilidad. Esto es cierto en varios componentes de nuestro diseño, la clase **Interfaz**, se encarga de únicamente de desplegar en pantalla, a un lado del laberinto, una pequeña interfaz donde se muestran aspectos importantes del juego, como son las vidas del jugador. Otro ejemplo de ello es la clase **Menú**, que se encarga del manejo y despliegue del menú. La clase **Avatar** se encarga únicamente de animar el avatar.

2. O – Principio de abierto-cerrado (OCP):

   Este principio se refiere a que nuestro código está cerrado a modificaciones, y abierto a extensiones. Precisamente, la estructura actual se basa en este principio. Por ejemplo tenemos la clase **Entidad**, que abre un gran espacio a la creación de cualquier entidad que se quiera. Otro ejemplo de ello sería la clase **Avatar**, que además es una entidad, sin modificar lo ya existente permite agregar distintos avatares dentro del juego y con más funcionalidades.

3. L – Principio de sustitución de Liskov(LSP)

   El patrón Plantilla se relaciona con este principio, que se basa en herencia, en vez de tener instancias de una clase padre, podemos tener una instancia de la clase hija, sin que se modifique el comportamiento de la clase padre. Como ejemplo de ello tenemos un laberinto del juego Batlle City que hereda de un laberinto MARDA y en vez de instanciar el laberinto MARDA se utiliza el laberinto específico del juego. Así tenemos varios otros ejemplos, como **AvatarControlable, AvatarAutonomo**.