Proyecto de Programación II

Pedro Pablo Álvarez Portelles
Abel Llerena Domingo.

Introducción

1 Estructura del Proyecto

El proyecto se encuentra en una solución de *C#*, dividida al mismo tiempo en varios subproyectos. Estos tienen funciones específicas dentro del mismo. Los subproyectos utilizados son los siguientes:

1. *jsonManager*: se encarga de la lectura de ficheros *.json*, ademas de su deserialización(todo add ref)

2. *cardEngine*: es la lógica central del juego, conteniendo todo lo necesario para el funcionamiento de las cartas, jugadores, partidas, etc.

3. *interpreterMLC*: un intérprete sencillo de un lenguaje de programación propio, creado con la única finalidad de ser utilizado en este proyecto.

4. *CardWEB*: una interfaz web hecha en *Microsoft .NET Razor* y que permite consumir la lógica creada para este juego.

A parte de estos proyectos de C#, se creó un directorio llamado *cardsDB* el cual guarda los archivos *.json* con las cartas que se han creado.

En lo adelante este informe se dedicará a explicar el diseño interno de dichos subproyectos, así como las estructuras y jerarquías de clases utilizadas, patrones de diseño y la razón de su utilización y otras cuestiones relacionadas con el funcionamiento de este juego.

2 El subproyecto *cardEngine*

En este subproyecto se establece la lógica principal del juego, es decir, todo el código que se encargará de gestionar la partida, jugadores, cartas, etc. Para conseguir esto, se crearon un conjunto de clases, siguiendo siempre los principios *SOLID*, en cuya estructura quedan plasmadas las reglas del juego ya sea mas implícita o explícitamente.

2.1 La clase abstracta `Card`

Esta clase es una abstracción de los objetos tipo `carta`: todas las cartas tienen algunas características en común como un nombre, una descripción, una imagen...

Sin embargo, la abstracción utilizada trae consigo que algunas funcionalidades sean comunes a todas las cartas como un constructor, un método de clonación y un método de conversión a `string`. Por esta razón, resulta más conveniente una clase abstracta en lugar de una interfaz.

2.1.1 La clase `MonsterCard`

Esta clase hereda de la clase abstracta `Card`. Su objetivo es servir como las cartas principales del juego, esto es, las encargadas de realizar ataques, utilizar poderes, etc...

Como propiedades específicas, añade las siguientes:

```
(...)
public int AttackPoints{get;private set;}
public int HP{get; private set;}
public bool IsAlive{get;private set;}
public bool IsDead{get;private set;}
public Power?[] Powers{get;private set;}
(...)
```

Con estas propiedades, se obtiene una carta capaz de infligir un daño así como de recibirlo, además de poder guardar un conjunto de poderes (clase `Power` la cual será explicada más adelante).

Además, añade algunos métodos propios tales como:

```
(...)
void Attack(MonsterCard target);
```

```
    void AssignPower(Power power);
    (...)
```

Con estos métodos, se completa la funcionalidad de una carta tipo Monstruo.

Recalcar además que esta clase tiene un constructor nuevo, que recibe un objeto de tipo `MonsterCardJsonItem`, cuyo propósito será explicado más adelante.

2.1.2 La clase `EffectCard`

Esta contiene un poder y su único propósito es ser equipada en algún Monstruo. Al igual que `MonsterCard`, hereda de la clase abstracta `Card` y añade las siguientes propiedades y métodos:

```
    (...)
    public Power power{get;private set};

    public void UseCard(MonsterCard target);
    (...)
```

Mientras la propiedad `power` es un objeto de tipo `Power`, el método `UseCard()` se encarga de llamar al método `AssignPower()` ya visto anteriormente, el cual a su vez se encarga de asignar el poder `power` a la carta especificada (`target` en este caso).

Esta clase al igual que `MonsterCard`, incorpora un nuevo constructor que recibe un objeto de tipo `EffectCardJsonItem`.

2.1.3 Algunas precisiones

Estas clases tienen una funcionalidad muy reducida a primera vista, sin embargo, esto es una decisión de diseño, ya que su responsabilidad se reduce a ser cartas: no necesitan información sobre si pertenecen o no a algún jugador, si están en el campo, mano o baraja, etc(*Single Responsability*).

2.2 La clase `Deck`

La responsabilidad de esta clase se acota a representar el mazo de cartas de un jugador. Este está formado por cartas de los tipos concretos ya expuestos,

por lo que para su representación interna resulta conveniente definir:

```
private Card[] DeckCards { get; set; }
```

Con lo cual, por el principio de *Sustitución de Liskov*, es posible guardar objetos derivados de la clase abstracta `Card`.

Esta clase además de un constructor, incluye el método `Draw()` el cual retorna una carta del mazo y la elimina del mismo, simulando el proceso de "robar" una carta. Con esto, se evita el tener que remover aleatoriamente el mazo, pues el tomar cartas aliatorias es un proceso equivalente.

Destacar brevemente que esta clase no tiene una manera concreta de construirse, solo que se le provea un *array* de cartas, separando así la lógica encargada de manejar el mazo de la lógica encargada de decidir que cartas poner en el mismo.

2.3 La clase `CardFactory`

En esta clase se implementa una variación del patrón de diseño *factory*. Su función es gestionar una "tienda" de mazos de cartas, de la cual se pueden obtener o registrar cartas.

Las cartas registradas serán guardadas en disco de forma persistente utilizando las funcionalidades que provee el subproyecto *jsonManager* (el cual se explicará mas adelante), y cuando se instancia una nueva `CardFactory`, esta leerá del disco las cartas anteriormente cradas, manteniendo así una consistencia con las acciones pasadas.

El patrón de diseño utilizado es una variación del patrón *factory*. Con esto existen distintas estrategias para la creación de un mazo, basadas en distintos parámetros de entrada, y que mediante polimorfismo se decide de manera automática cual es la estrategia pedida por un usuario. Las estrategias implementadas son las siguientes:

1. Obtener un `Deck` basado en el nombre de las cartas que se desean.
2. Obtener un `Deck` basado en los objetos carta que se especifiquen.
3. Obtener un `Deck` completamente aleatorio, basándose en la probabilidad de aparecer de cada carta.

Su interfaz sería aproximadamente así:

```
//para obtener un Deck random
public Deck GetDeck();
//para obtener un Deck por los nombres de las cartas pedidas
```

```
public Deck GetDeck(string[] names);
//para obtener un Deck con cartas especificas
public Deck GetDeck(Card[] requestedCards);
```

En el caso de los dos últimos métodos, en caso de proveer más cartas de las que se permiten en un mazo, se tomarán algunas de las pedidas hasta rellenar los espacios; en el caso contrario de que se especifiquen muy pocas cartas, se completará automáticamente con cartas aleatorias que concuerden con las probabilidades de aparecer de cada carta.

Por último, destacar el siguiente método:

```
public void CreateCard<T>(T args) where T : Card;
```

El cual se encarga de escribir en disco una carta nueva. Se debe especificar el tipo de la carta que será guardada, pues para distintos tipos de cartas existen distintas propiedades y con la configuración de *.NET* para archivos *.json*, no es posible guardarlos todos en un mismo archivo.

2.3.1 La clase `Power`

Esta clase representa un poder. Un poder no es más que un `string` que contiene un segmento de código interpretable por el subproyecto *interpreterMLC*, unido a un nombre que identifica a dicho poder.

Esta clase contiene el método `UsePower()` que básicamente recolecta información sobre la partida, la carta atacante y la carta víctima del ataque, y la utiliza para llamar al intérprete con el código conteniendo en el poder.

2.3.2 Las clases `Player` y `VirtualPlayer`

Estas son las encargadas de definir las acciones que puede realizar un jugador. Así, estas guardan la mano y el mazo del jugador, así como una sección de la mesa donde este puede colocar cartas. Al asociar a un jugador su porción de la mesa, se evita que otros jugadores puedan colocar cartas ahí.

El jugador también lleva cuenta de si está jugando (`bool IsPlaying`), y de cuales monstuos ha usado en este turno y cuales no. También tiene un método `BeginTurn()` y un método `EndTurn()`, los cuales gestionan que empiece o termine un turno.

Las acciones relacionadas con las cartas como jugar, atacar, equipar poderes, etc, tienen aquí también métodos que gestionan su lógica a nivel

de jugador.

Por último, la clase `VirtualPlayer`, es derivada de `Player`, y representa un jugador virtual, cuya lógica se ejecuta internamente luego de iniciar su turno, es decir, una llamada al método `BeginTurn()`, internamente llama a la lógica que realiza un conjunto de jugadas y finalmente, temina el turno actual.

2.3.3 La clase `Match`

Esta es la clase que integra todas las demás reglas del juego. Se encarga de gestionar la partida general, terminar turnos, asignar mazos, determinar ganadores, etc.

Los métodos y propiedades de esta clases son:

```
(...)
public Player player { get; private set; }
public Player enemy { get; private set; }
public CardFactory Factory { get; private set; }
public int TurnCounter { get; private set; }
public bool isPlaying { get; private set; }

public void SetDeck(Deck deck, bool player);
public void SetDeck(string[] names, bool player);
public void SetDeck(bool player);

public void BeginGame();

public void EndTurn();
public void Play(int card, MonsterCard? target = null);
public bool Attack(int monsterIndex, MonsterCard target);
public bool UsePower(int monsterIndex, int powerIndex,
    MonsterCard target);
public void DropCard(int cardIndex);

public Player? Winner();
(...)
```

Contiene dos objetos de tipo `Player` (opcionalmente uno de ellos puede ser `VirtualPlayer`), los cuales se llaman respectivamente *player* y *enemy*.

Antes de empezar una partida se deben establecer los mazos. Esto se

hace mediante uno de los métodos polimórficos `SetDeck()`, los cuales internamente llaman a una instancia de `CardFactory` la cual ya tiene implementa la lógica necesaria para crear un mazo de cartas.

Luego de esto, es necesario llamar al método `BeginGame()`, el cual permite que los jugadores empiecen a hacer movimientos. Cada vez que se realiza una acción que hace que ya no sea posible atacar o invocar otro monstruo el turno termina de forma automática, llamando a un método privado llamado `AutoEndTurn()`; sin embargo, es posible terminar el turno de manera "manual" llamando a `EndTurn()`. Al finalizar un turno, los jugadores intercambian posiciones, es decir, *player* pasaría a ser *enemy* y viceversa. Con esto se facilitan muchas implementaciones, puesto que ahora siempre las acciones serán realizadas por *player*, independientemente de cuantos turnos hallan transcurrido.

Fuera de esto, la clase `Match` solo establece las reglas para el juego, asegurando que:

1. Solo se ataca después de finalizado el primer turno
2. Que no se realicen jugadas antes de iniciar la partida
3. Que los monstuos que mueran en cada ataque sean retirados de la mesa (no se pueden recuperar)
4. Que luego de que se decida un ganador se detenga la partida

Es decir, aunque las responsabilidades de esta clase parecen ser muy amplias, no existe una manera de separarlas en subclases mas simples, ya que este es el objeto mas sencillo capaz de manejar las reglas del juego.

2.3.4 El namespace `CardJsonItems`

Este espacio de nombres contiene tres clases:

1. `CardJsonItem`
2. `MonsterCardJsonItem`
3. `EffectCardJsonItem`.

Estas clases heredan de la interfaz `IJsonItem`, la cual viene definida en el subproyecto *jsonManager*. Con ellas se pueden construir archivos *.json* que contienen la información necesaria para la creación de objetos del tipo que representan.

Estas son utilizadas como parámetros de tipo en la creación de objetos de tipo `JsonRW` (se explicará más adelante), y cuya responsabilidad será leer y escribir de disco archivos *.json* con todas las cartas creadas.

3 El subproyecto *jsonManager*

Este subproyecto contiene clases e interfaces utilizadas para el manejo de archivos *.json*. Solo contiene las siguientes definiciones:

1. `JsonRW`
2. `IJsonItem`

La interfaz `IJsonItem`, se utiliza como objeto base para la lectura de archivos *.json*: la clase `JsonRW` solo acepta para la lectura clases derivadas de dicha interfaz.

Por otro lado, la clase `JsonRW`, recibe un tipo derivado de `IJsonItem` además de un `string` que contiene la dirección de el fichero que controlará la instancia creada de esta clase. Finalmente, utilizando el método `Add()`, se pueden añadir nuevos *items* al fichero los cuales serán guardados automáticamente.

Este subproyecto se utiliza en la creación y lectura de cartas como se explicó anteriormente, sin embargo no depende de las clases que realizan dichas operaciones, limitando así su responsabilidad.

4 El subproyecto *interpreterMLC*

Este proyecto, como su nombre sugiere es un intérprete simple para un lenguaje de programación creado con el propósito de crear cartas para este juego. Este lenguaje pese a ser un subconjunto de *C#*, tiene algunas características propias, como algunos detalles de la sintaxis, y el uso del caracter @ para especificar acciones como ataques... Para mas información click aquí.

4.1 La clase `Lexer`

Esta clase es un tokenizador, es decir, su función es tomar un string de entrada que lo recibe su constructor, y separarlo en tokens.

Un token es una instancia de la clase `Token`, la cual contiene el tipo del token así como su contenido. Esta clase `Token` está definida en el mismo espacio de nombres que `Lexer` ya que están muy ligados entre sí.

De este subproyecto, esta en particular es la clase más sencilla, ya que su funcionamiento se reduce a iterar sobre el string de entrada y retornar un nuevo token cada vez que lo encuentre.

4.2 La clase `Parser` y el archivo *ParserNodes.cs*

La clase `Parser`, se encarga de crear un Árbol de Sintaxis Abstracta(**AST**), que sirva como representación intermedia para la interpretación del código de entrada.

Esta utiliza una serie de gramáticas para convertir una serie de tokens proveida por la clase `Lexer`, en un árbol que pueda ser interpretado directamente recorriendo sus nodos.

En el propio espacio de nombres de esta clase, se encuentra la interfaz `AST`, la cual es la base para todos los nodos de dicho árbol. Luego, existe un conjunto de clases bastante amplio, en el que cada uno representa un posible nodo del *AST*, como operadores binarios, operadores unarios, métodos, propiedades, strings, variables y más. Estos nodos se crean de forma recursiva y al finalizar el *parsing* se obtiene un *AST* listo para ser interpretado.

4.3 La clase `Interpreter`

Esta clase se encarga de crear un `Parser` con un texto de entrada, pedirle que construya un *AST*, y finalmente interpretarlo. En el proceso de ocurrir

un error este se retorna para que el usuario decida como manejarlo, en el caso de este proyecto, normalmente se le muestra al usuario una notificación de por qué no se ejecutó correctamente su código.

4.4 Las clases `NodeVisitor` y `NodeVisitors`

Estas clases simplemente almacenan un conjunto de métodos que se utilizan para recorrer los diferentes tipos de nodos del *AST*. La clase `NodeVisitor` es la que será llamada siempre, y esta se encargará de llamar al método correspondiente de la clase `NodeVisitors` utilizando *reflection*.

Ambas clases son internas de su espacio de nombres, ya que se utilizan solo ahí, pero declararlas privadas podría provocar errores y *warnings* ya que este espacio de nombres se encuentra dividido en dos archivos para mayor claridad del código.

4.5 El espacio de nombres `Symbols`

Este espacio de nombres contiene enumeradores, clases e interfaces utilizadas para la representación de símbolos. Un símbolo es cualquier dato que ocurra en el programa, ya sea una variable o un tipo de dato por defecto.

Contiene el enumerable `SYMBOLS` el cual guarda todos los tipos de tokens que pueden ocurrir en un programa de este lenguaje de programación. Por ejemplo, `SYMBOLS.PLUS` representa un signo de adición y `SYMBOLS.STRING` un string.

La interfaz `ISymbol` es una base para la creación de símbolos, la cual establece que todos los símbolos deben tener un nombre y un tipo, que no es más que otro símbolo. En el caso de los tipos de dato por defecto, su tipo es nulo, ya que no existe un símbolo que les de orígen.

De esta interfaz se crean las clases `BuiltinDataType` y `VarSymbol`, que representan los tipos de datos por defecto y las variables respectivamente.

Utilizando estas dos últimas clases, la clase `SymbolTable` crea una tabla de símbolos que se encarga de gestionar las variables que han sido declaradas y los tipos de datos disponibles. Esta no hace la función de espacio en memoria, sino simplemente de tabla de símbolos.

Para hacer de espacio en memoria y almacenar los valores de las variables, se creó la clase `Scope`.