Como se puede apreciar en el diagrama en nuestro proyecto todo parte de tres nodos principales que son Cards, Cards Engine y ConsoleApp que explicaremos detalladamente a continuación:

1. **Cards**

Esta es la súper clase de la que heredan los dos tipos de carta protagonistas de nuestro juego (*MonsterCards* y *PowerCards*) las cartas de monstruos y las cartas de poder que no son más que habilidades que tendrán esos monstruos.

Cada carta tiene las siguientes propiedades:

**-Nombre**: No es más que el nombre de la carta  
**-Descripción** pública: Es la descripción del efecto de la carta que se mostrara en el tablero (en el caso de las *PowerCards* debe ser lo más explícita y certera posible) o simplemente una descripción para argumentar la historia en el juego (en el caso de los *MonsterCards*).

Las cartas de monstruo además de los campos anteriores incluyen:

**-Tipo:** Explicar bonificación de ataque, adjuntar diagrama circulo  
**-Estado:** es el estado del monstruo (puede ser *Normal*, *Muerto* o *Envenenado*)  
**-Puntos de ataque:  
-Puntos de vida:**

Las cartas de poder contarán con las siguientes propiedades:

**-Energía de activación:** el usuario debe contar con una cantidad igual o superior de energía para poder usar dicha carta.  
**-Código:** es el código que deberá escribir el creador de la carta. Para su correcta definición debe ser escrito de la siguiente manera:

Condition{  
 Condiciones que se deben cumplir para que la carta se pueda ejecutar

}  
Actions{  
 Acciones a realizar si se cumplen las condiciones

}

El funcionamiento del lenguaje esta mejor explicado en la sección 4. Lenguaje

La clase *Card* incluye además dos *enum* uno para los tipos y otro para los estados

Los *MonsterCard* tienen dos métodos: *UpdateLife()* que modifica la vida del monstruo pero no permite que esta sea negativa y *WeaknessValue()* que dado el tipo del monstruo que la ataca devuelve si es débil, fuerte o neutral con respecto a este ataque.

Para el correcto funcionamiento del programa fue necesario implementar un método *Clone()* para estos objetos. Mas adelante explicaremos su uso.

Estado envenenado le quita el 3% de la vida al monstruo envenenado (mientras este porciento sea mayor que 1 sino le quita 1 un punto) al inicio del turno

1. **CardsEngine**

En este conjunto de clases es donde se desarrolla toda la lógica del juego:

**Deck:**

El objeto *Deck* no es más que la colección de *MonsterCards*(monstruos) y *PowerCards*(habilidades) que usted elije para jugar. Para este juego son 3 monstruos y 4 habilidades para cada uno (12 en total). Las habilidades están asociadas a los monstruos a través del diccionario *associations* que establece la relación de cada habilidad con el índice del monstruo en el array *monsters* del *Deck*.

**Board:**

El objeto *Board* se compone de dos propiedades: *hands*(Cartas en la mano) y *monsters*(Monstruos en el campo).

-*Monstruos en el campo*: por definición de nuestro juego cada jugador contara con tres monstruos a lo sumo en su campo.  
-*Cartas en la mano*: cada jugador comenzara el juego con una cantidad de 5 cartas en la mano y en cada turno robara dos cartas más de su mazo de habilidades. Cada jugador podrá tener como máximo 7 cartas en la mano.

**Game**:

Esta clase es la que controla el flujo del juego, todo el contexto es almacenado en ella, por ello explicaremos detalladamente sus propiedades y funcionalidades:

ADJUNTAR SCREEN DE CODIGODEL CONSTRUCTOR DE GAME

Propiedades:

-players: Esta propiedad está definida por un array booleano de tamaño (cantidad de jugadores) que sindica que el jugador i es real, si es true, en cambio, si es false, es un NPC. Esto es necesario para llevar el control y saber si el juego necesita interactuar con un jugador real o si simplemente ejecutar el código del NPC.  
-decks: Obviamente el juego no puede comenzar si no están definidos todos los decks de los participantes que son necesarios para construir el tablero.  
-turn: lleva el numero de turno en el que se esta.  
-currentPlayer: es el jugador que esta jugando.  
-energyPonits: lleva la cantidad de puntos de energía que tiene cada jugar actualmente, necesarios para poder activar habilidades.  
-losers: esta propiedad es un array booleano que indica que le jugador *i* perdió si está en true. Cuando quede solo uno en *false* se acabó el juego y este gana el juego.  
-board: Aquí se almacena el objeto tablero relacionado a este juego  
-npcs: en una lista guardamos las instancias de todos los *npcs* del juego.

Para controlar el flujo del juego desde fuera de la clase Game tenemos siguientes métodos:

-UpdateTurn() : Pasa al siguiente turno y reinicia el contador del jugador actual.  
-NextPlayer() : Pasa al siguiente jugador.  
-SetPlayer(): Pone el contador de jugador actual en un jugador en específico.

-UpdateEnergy() : Actualiza la energía, es usado tanto cuando un jugador juega una carta como para actualizar la energía al comienzo del turno.  
-UpdateLosers() : Establece que el jugador dado por parámetros perdió.

Otros métodos de la clase son:

-CanPlay() : este método compara la energía necesaria para jugar una carta con la energía del jugador y si la primera es menor o igual que la segunda entonces se puede jugar la carta.  
-PlayCard() : Este método se ejecuta cada vez que un jugador decide que carta usar y contra quien. El monstruo objetivo se decide según la formación del jugador objetivo, el primer monstruo vivo será seleccionado. Posteriormente se intenta ejecutar el efecto de la carta, si se cumplieron las condiciones para ejecutar las acciones de la carta entonces se ejecutan (este proceso está más detallado en la sección 4. Lenguaje), se verifica si el monstruo objetivo murió en caso de que si se cambia el estado del mismo y se eliminan las cartas asociadas a ese monstruos de la mano del jugador objetivo, se actualiza la energía del jugador quitando el costo energético de la carta jugada y se remueve la carta usada de la mano.

**NPC**:

Los objetos de tipo NPC solo contarán con un índice para identificarlos.  
PlayTurn() : es el método que se invoca cuando le toca jugar al NPC.  
La jugada del NPC se procesa con los métodos:  
Potencia() : este método recibe la mano del NPC como parámetro y crea el conjunto potencia de las cartas, podando por el requerimiento de energía. Posteriormente es llamado el método…  
Permutation() : este método permuta cada una de los conjuntos potencia y con cada una de las permutaciones llama al método...  
(Ambos metodos Potencia() y Permutation() fueron implementados con BackTracking)  
EvaluateBetterPlay() : aquí se clona la instancia de Game clonando cada una de sus propiedades y  
se juegan todas las cartas de la permutacion que se recibió, se evalúa el resultado y finalmente se  
escogerá la combinación de cartas que mas daño haya causado al jugador objetivo.

**Engine**:

Esta clase es estática y fue necesaria para implementar métodos auxiliares.

-LoadCards() : esta función carga todas las cartas desde archivos tipo JSON y las almacena en dos listas, definidas también en esta clase.  
-SaveMonsterCard() y SavePowerCard() son utilizados para guardar permanentemente las cartas creadas por cualquier jugador.  
-GetInitialHand() : Solo es utilizada para repartir la mano inicial de 5 cartas a cada jugador, completamente en aleatorio.  
-Draw() : Simplemente toma una carta *random* del *Deck* cuando es el inicio del juego, en los siguientes turnos solo toma una carta del *Deck*, si el monstruo asociado a esta, no ha muerto aun.  
-PlayerWins() : Verifica si un jugador gano, esto ocurre cuando solo queda un jugador sin perder. Se ejecuta al final de cada turno.  
-PlayerLose() : Determina si un jugador perdió, esto sucede cuando todos sus monstruos han muerto. Se ejecuta después de cada ataque.  
-MonsterDied() : determina si la vida de un monstruo llegó a cero. Se ejecuta después de cada ataque.  
-DeleteDeadMonsterPowerCards(): Elimina de la mano las cartas asociadas a un monstruo que murió.

-UpdateMonsterState(): Este método al inicio del turno realiza las acciones referentes a los estados de cada monstruo (hasta el momento solo se realizan acciones para los monstruos envenenados).

-ActionDraw(): este es el método utilizado para robar cartas del *Deck*, tanto por la acción *Draw* del lenguaje, como al inicio del turno de cada jugador.

Clone(): es un método auxiliar utilizado para clonar array de cualquier tipo.

1. **ConsoleApp**
2. **Lenguaje**

Durante el proceso de creado de un PowerCard es necesario dotarla de un script que es el efecto de la carta para esto fue necesario implementar el lenguaje del cual hablaremos a continuación.

El AST del lenguaje tiene la siguiente estructura:

INSERTAR FOTO DEL AST

Una vez escrito el código que no es mas que una cadena de texto este pasa por un proceso de tokenizacion y parseo para convertir dicha cadena en información relevante y procesable por c#. Lo primero en este proceso es tokenizar la cadena para esto usamos un objeto de tipo Tokenizer que a su vez depende de otro de tipo Reader que son clases que cuentan con las herramientas necesarias para realizar este proceso. Lo segundo es el proceso de parseo llevado a cabo por un objeto de tipo Parser. Pero antes de explicar el funcionamiento hablemos de un par de conceptos necesarios para entender dichos procesos.

**Token:** un token es un objeto que representa cada unida de código que tiene importancia para nuestro lenguaje dígase palabras claves o simplemente símbolos y números. En nuestro caso almacenan la posición en la que aparecen el script (valiéndose de un objeto de tipo Position que guarda las fila y la columna en la que aparece dicho token), el pedazo de código que representa y el tipo de ese pedazo de código que puede ser: *symbol*, *number*, *keyword*, *eof*(end of file) o *unknown*(para cuando hay texto que no es válido en nuestro lenguaje).

**Error:** un error es un objeto que como su nombre indica representa la aparición de un problema tanto sintáctico como semántico. En nuestro lenguaje este almacena la posición en donde se encuentra el *error* en el *script* (con un objeto Position) y el mensaje de descripción del error.

**Position:** No es más que una estructura de datos que almacena la línea y la columna donde aparece tanto un *token* como un *error*.

Ahora si empecemos.

El proceso de tokenizacion que no es mas que tomar la cadena de caracteres e ir creando tokens a partir de lo que vamos leyendo se realiza de conjunto en las clases *Tokenizer* y *Reader*.

El objeto Tokenizer tiene tres propiedades:

-*errors*: una lista de errores.

-*tokens*: un lista de tokens.

-*reader*: un objeto de tipo Reader.

El objeto Reader tiene las siguientes propiedades:

-*symbols*: todos los posibles símbolos que pueden ser un token

-*code*: la cadena de caracteres que representa el script

-*codeByLines*: el código separado por líneas

-*line*: la línea actual

-*column*: la columna actual

-*eof*: la posición del final del archivo

-*errors*: una lista de errores

Una vez llamado el método *Tokenize()* son cargados todos los keywords y símbolos validos en nuestro lenguaje estos estaban previamente almacenados en la clase estática *TokenCodes* (solo sirve de contenedor para los keywords y los simbolos). Y mientras el reader no lea el final la cadena con el método *CheckEOF()* intentamos leer un token. Primero desechamos los espacios en blanco para lo cual no valemos del metodo IsWhiteSpace() de *reader*. Luego tratamos de leer los tokens uno a uno primero los keywords para esto por cada posible keyword llamamos al método *TryReadToken*() de *reader* que nos dice si estamos leyendo o no el código de token que le estamos pasando por parámetro en caso positivo agregamos a la lista de tokens un nuevo token en la posicion actual del código en la que estamos y con el pedazo de código que le corresponde. Si no logramos leer ningún keyword pasamos a los símbolos en caso de no poder leer ningún símbolo, intentamos con un numero usando el método TryReadNumber() de *reader* que nos devolverá un token de tipo numérico si logro leer un numero correctamente. En ultima instancia de no poder leer nada de lo anterior llamamos al metod *Read()* que devuelve un token de tipo *unknown* con el pedazo de código que no es válido para nuestro lenguaje y agrega un error de expresión desconocida a la lista. De esta manera se repite el proceso hasta el final de la cadena.

Del objeto tipo Reader solo nos faltó por mencionar un método el CheckEOL() que verifica si llegamos al final de una línea pasa ala siguiente y resetea el contador de las columnas.

El proceso de parseo es llevado a cabo por la clase Parser valiéndose de la lista de tokens generado en el proceso anterior.