Segundo Proyecto de Programación

Curso 2024

GWENT: Fase 2

Después de pasar la ardua parte de aprender Unity y programar en 12 semanas, llega la parte de aprender a hacerlo correctamente. ¿Y qué mejor forma de hacerlo que con una expansión al GWENT?

La expansión contará con 2 partes. Sólo los aventureros fuertes y capaces llegarán a la segunda.

La primera es la siguiente. Vamos a ser realistas, no nos gusta que ustedes sean los únicos capaces de crear mazos, nosotros también queremos, suena divertido. Pero el juego lo ponen ustedes, así que lleguemos a un acuerdo. Queremos un simple botón dentro del juego que al tocarlo se abra una simple ventana con un simple editor de texto en el cual nosotros vamos a poner un simple mazo. Ahora viene la parte no tan simple:

Ustedes se preguntarán cómo será eso posible, pues para ello hemos ingeniado un **Lenguaje de Propósito Específico (DSL)** en el cual todas las cartas se escribirán y se necesitarán importar al sistema de la misma forma. El rol de ustedes es crear un mini compilador para ese DSL. El compilador es una caja oscura en esta gran carrera que son las Ciencias de la Computación, pero por suerte ustedes no tienen que implementarlo completo (aún). El compilador es un software encargado de leer un archivo de texto de cualquier programa y llevarlo a un lenguaje muy simple tal que la computadora lo pueda entender (o no). Esto suena abrumador, así que vayamos por partes.

Primero vayamos con el DSL. En este DSL ustedes pueden escribir declaraciones de cartas y efectos. Miren aquí un ejemplo:

```
effect {
    Name: "Damage",
    Params: {
        Amount: Number
    },
    Action: (targets, context) => {
        for target in targets {
            i = 0;
            while (i++ < Amount)
                target.Power -= 1;
        };
    }
}

effect {
    Name: "Draw",
    Action: (targets, context) => {
        topCard = context.Deck.Pop();
        context.Hand.Add(topCard);
        context.Hand.Shuffle();
    }
}

effect {
    Name: "ReturnToDeck",
    Action: (targets, context) => {
        for target in targets {
            owner = target.Owner;
            deck = context.DeckOfPlayer(owner);
```

```
            deck.Push(target);
            deck.Shuffle();
            context.Board.Remove(target);
        };
    }
}

card {
    Type: "Oro",
    Name: "Beluga",
    Faction: "Northern Realms",
    Power: 10,
    Range: ["Melee", "Ranged"],
    OnActivation: [
        {
            Effect: {
                Name: "Damage", // este efecto tiene que estar previamente definido.
                Amount: 5, // ... y tener estos parámetros.
            },
            Selector: {
                Source: "board", // o "hand", "otherHand", "deck", "otherDeck", "field", "otherField",
"parent".
                Single: false, // aunque por defecto es false.
                Predicate: (unit) => unit.Faction == "Northern" @@ "Realms"
            },
            PostAction: {
                Type: "ReturnToDeck",
                Selector: { // opcional dentro de PostAction, en cuyo caso no se vuelve a seleccionar
sino que se usa los del padre.
```

```
                    Source: "parent",
                    Single: false,
                    Predicate: (unit) => unit.Power < 1
                },
            }
        },
        {
            Effect: "Draw" // si se pone un string directo es equivalente a { Name: "Draw" }
        }
    ]
}
```

Vamos por partes. Primero los efectos (`effect`):

- **Name**: Nombre del efecto, requerido.
- **Params**: Parámetros que recibe el efecto junto con el tipo asociado (`Number` , `String` o `Bool`), opcionales.
- **Action**: Es la función que hace el efecto posible. Los `Params` del efecto son accesibles dentro del cuerpo de la función. Se definen explícitamente los siguentes dos parámetros.
    - `targets` es la lista de objetivos. Todos los efectos tienen uno o varios objetivos declarados por su selector cuando este efecto es invocado *(se verá luego al explicar las cartas)*.
    - `Context` es el contexto del juego, que contiene información sobre el estado del juego. Tiene las siguinetes propiedades.
        - `TriggerPlayer` : devuelve el identificador (*id*) del jugador que desencadenó el efecto.
        - `Board` : devuelve una lista de todas las cartas del tablero.
        - `HandOfPlayer(player)` , `FieldOfPlayer(player)` , `GraveyardOfPlayer(player)` y `DeckOfPlayer(player)` : Cada una de estas propiedades devuelve la lista correspondiente del jugador pasado como parámetro. Como azúcar sintáctica, `context.Hand` es un diminutivo de `context.HandOfPlayer(context.TriggerPlayer)` . De forma equivalente se tiene lo mismo para `Deck` , `Field` y `Graveyard` . Adicionalmente, cada carta tiene la propiedad `card.Owner` que devuelve el *id* del jugador dueño de la carta, o sea, del jugador de cuyo deck esta carta formó parte inicialmente.

A continuación se listan los métodos que tienen cada una de las listas de cartas accesibles desde el contexto.

- `Find(predicate)` : Devuelve todas las cartas que cumplen con el predicado, el predicado es una función que recibe una carta y devuelve un booleano, ej: " `context.hand.find((card) => card.Power == 5)` " devuelve todas las cartas en la mano del usuario que tengan poder 5.
- `Push(card)` : Agrega una carta al tope de la lista.
- `SendBottom(card)` : Agrega una carta al fondo de la lista.
- `Pop()` : Quita la carta que está al tope de la lista y la devuelve.
- `Remove(card)` : Remueve una carta de la lista.
- `Shuffle()` : Mezcla la lista.

Ahora las cartas (`card`).

- **Type**: El tipo de la carta (puede ser "*Oro*", "*Plata*", "*Clima*", "*Aumento*", "*Líder*" y cualquier otra que ustedes hayan personalizado).
- **Name**: Nombre de la carta.
- **Faction**: La facción de la carta, idealmente todas las cartas de un mismo mazo tienen la misma facción.
- **Power**: El poder o los puntos que tienen las cartas. *Clima*, *Aumento* y *Líder* deberían tener `Power = 0` o su Power debería ser ignorado.
- **Range**: Un array de posibles rangos para la carta. Se acepta `"Melee"` , `"Ranged"` o `"Siege"` .
- **OnActivation**: Una lista de efectos que se van a ejecutar en secuencia cuando la carta se coloque en el campo. En la invocación de un efecto hay varias propiedades según se describe a continuación.
  - **Effect**: Un objeto que lleva dentro el nombre del efecto a usar (debe haber sido declarado anteriormente con `effect`) y los parámetros que recibe, cada uno como propiedad diferente. Como azúcar sintáctica, si se pone un string directo es equivalente a `Effect: { Name: "NombreDelEfecto" }` (eg. `{ Effect : "Draw" }`).
  - **Selector**: Un filtro de las cartas a las cuales se les va a aplicar el efecto (que en la definición del efecto serán los `targets`). El selector se construye a partir de las siguientes propiedades.
    - **Source**: es la fuente de donde se sacan las cartas, fuentes aceptables son: `"hand"` , `"otherHand"` , `"deck"` , `"otherDeck"` , `"field"` , `"otherField"` o `"parent"` . En particular, la fuente `"parent"` solo puede ser especificada como fuente en una *PostAction (más delante se explica lo que es)*, y significa que su fuente será la lista de `targets` construida por el efecto del cual ella es *PostAction*.
    - **Single**: es un booleano que dice si solo se va a tomar el primer valor de la búsqueda (igual será una lista pero con un solo elemento) o si se seleccionaran todas las cartas que cumplan el predicado.

        - **Predicate**: es el filtro en sí, es una función que recibe una carta y devuelve un booleano que indica si se debe tomar o no.
    - **PostAction**: Otra declaración de efecto, opcional, a ejecutarse después que el primer efecto se haga ver (el cual a su vez puede tener otro *PostAction*). En un *PostAction*, la propiedad *Selector* es opcional y en caso de ser omitida se usará como `targets` la misma lista que su efecto "padre". En caso de querer filtrar un subconjunto de los `targets` del padre, entonces se definiría un *Selector* con `Source: "parent"` y un *Predicate* a conveniencia. Un ojo capaz se da cuenta que es virtualmente es lo mismo poner `n` efectos en *OnActivation* que poner uno de detrás de otro como *PostAction*. Ponerlo como *PostAction* permite sin embargo representar comportamientos anidados que dependan de los objetivos del padre.

El lenguaje de una función en el DSL es simple:

- Se aceptan operadores aritméticos (`+` , `-` , `*` , `/` , `^` , `++`), lógicos (`&&` , `||`), de comparación (`<` , `>` , `==` , `>=` , `<=`), concatenación de cadenas (`@` , `@@`) o de asignación (`=`).

> *El operador `@@` incluye un espacio entre las cadenas que se concatenan.*

- Se acepta declaraciones de constantes y variable, (i.e. `temp = 5`)
- Se aceptan accesos a propiedades tanto del contexto (`context.Hand`) como de una carta (`card.Power`)
- Se acepta el indexado en listas (i.e. `context.Hand.Find((card) => true)[0]`)
- Se aceptan ciclos en listas (`for` y `while`) (i.e.: `for target in targets` , `while (i < count)`).
- Las funciones y el cuerpo de los ciclos pueden ser tanto una expresión como un bloque de expresiones. Un bloque de expresiones es declarar varias expresiones entre llaves y terminadas en `;` (como se puede apreciar en los ejemplos de `Action` de los efectos). Una sola expresión se ve en el ejemplo del `Predicate` del selector en la carta.

GWENT: Fase 2++ *[Opcional]*

Para los aventureros que lograron llegar tan lejos y se piensan que no han tenido desafío suficiente les sugiero lo siguiente. Uno se cansa de jugar con uno mismo o incluso somos antisociales y no tenemos amigos con los que jugar. Créennos un amigo que juegue con nosotros. O sea, creen un jugador de GWENT capaz de jugar cualquier partida, no importa el mazo que tenga. Para ello les vamos

a dar una ayuda. Sólo tienen que implementar la siguiente interfaz. `IContext` no tienen que implementarla. Asuman que `Card` tiene todas las propiedades descritas en el DSL con accesores públicos.

```
public interface IPlayer {
    public Card Play(IContext context);
}

public interface IContext {
    public List<Card> Graveyard {get; set;}
    public List<Card> Hand {get; set;}
    public List<Card> Board {get; set;}
}
```

Como bonus, los que decidan implementarla les daremos el beneficio de participar en un gran torneo de toda la facultad. Que la mejor IA gane.

PD: Cualquier mejora al DSL se le considerará también como aventurero de GWENT Phase 2 Pro.