Gwent++

Guía de desarrollo

Antes de leer este documento se recomienda leer la Guía de Desarrollo inicial y la Guía de juego, que se encuentra en este mismo directorio, para entender mejor las reglas sobre las cuales el funcionamiento de este juego está construido.

Introducción

Gwent-pro es una juego de cartas digitales para dos jugadores que enfrenta a dos ejércitos en un campo de batalla. Es una aplicación, desarrollada con tecnología .NET Core 8.0,usando Unity como framework para la interfaz gráfica, y en el lenguaje C#.

En esta expansión se le otorga al usuario la posibilidad de crear sus propias cartas y efectos dentro del juego. Está posibilidad está dada gracias a un Lenguaje de Propósito Específico (DSL) mediante el cual el usuario podrá compilar sus ideas materializándolas como cartas o efectos completamente funcionales dentro del Gwent-pro que ya conocemos.

Descripción general de la Expansión Gwent++

El desarrollo de la expansión vino estrechamente ligado a la implementación de algún tipo de intérprete/compilador que permitiera la comunicación entre el usuario y el juego original Gwent-pro con el propósito de personalizar sus cartas y efectos. En esta guía se espera que el lector esté familiarizado con términos de compilación, ya que se mencionarán a continuación sin dar muchas definiciones o explicaciones extensas al respecto.

La integración de esta expansión también trajo pequeños cambios a la interfaz visual del juego original y ligeras adaptaciones en la comunicación entre el back-End (esta vez ampliado) y el front-End (previamente desarrollado) que se mantiene basado en el patrón de Diseño Observer.

La expansión le quita un poco de libertad al usuario a la hora de desarrollar sus propias cartas, brindándole la posibilidad (implícitamente) de solo crear personajes basados en Morty Smith. Esta decisión fue tomada para preservar la temática del Gwent-Pro original y la integración visual de los nuevos elementos al juego, asegurando una mejor experiencia.

Gwent DSL

El lenguaje creado para introducir cartas y efectos al juego está compuesto por declaraciones de cartas y efectos, estos a su vez, están compuestos por declaraciones de sus respectivos campos.

Una declaración de carta contiene los campos: Type, Name, Faction, Power y Range (opcionales para cartas de tipo “Líder”, en caso de declararse serán ignorados), un campo de OnActivation que representa los efectos que ejecutará la carta con sus respectivos parámetros, y adicionalmente (pero obligatorio) está la declaración de CharacterDescription, EffectDescription y Quote.

Solo es posible declarar cartas de unidad y líderes. Una facción solo será tomada en cuenta para elegirla como mazo, una vez tenga líder.

Una declaración de efecto contiene los campos: Name, Params(opcional) y Action. El Action es lo que se define como DSLFunction, y además de lo definido como aceptado en el lenguaje, nuestro DSL también acepta operadores extra como: ! (negación lógica), != (distinto de), -- (post-decremento), *= (asignación multiplicativa), /= (asignación con división); y la estructura if/else como estructura de control de flujo de ejecución.

Implementación de GwentCompiler

El funcionamiento del compilador se implementó en una biblioteca de clases que se encuentra en la ruta “Assets/GwentCompiler”.

GwentCompiler implementa clases para el procesamiento de la información introducida por el usuario hasta crear el elemento del juego deseado o, en caso de que el usuario cometiera un error, informarle del mismo y guiarlo de vuelta al camino por el que deseaba transitar.

El Compilador

GwentCompiler es la clase principal de nuestro compilador y la que se encarga, de transportar la cadena de texto que recibe como entrada por todo el proceso que veremos a continuación.

Su único método Compile() recibe como parámetro la entrada de código fuente del usuario. Durante el juego esta información puede venir de dos maneras: mediante un archivo de texto plano almacenado en la ruta “Assets/CardsCollection/TestFile”, cuyo contenido es leído y procesado automáticamente al presionar el botón Jugar del menú principal desde el juego Gwent-pro. Posteriormente es recomendable mirar la consola del editor, esta informará si la compilación ha ocurrido sin problemas o informará del error en el caso contrario.

En el cuerpo del método Compile() el texto de entrada es analizado léxica y sintácticamente, al finalizar ambos procesos nuestro compilador conoce qué tipo de objeto está creando (carta o efecto) y cuáles son sus componentes. En este punto el compilador interpreta dichos componentes por separado y crea la carta o efecto compilado correspondiente. Finalmente, esta carta o efecto compilado es guardado mediante un proceso de serialización binaria y pasa a ser parte del juego original. En caso de ocurrir un error durante alguno de estos procesos será debidamente informado al usuario.

Detección, generación y reporte de errores

Cada componente del compilador es capaz de detectar errores por su cuenta según el proceso que realice, esto se extiende a lo largo de toda la implementación del Compilador. Sin embargo, la capacidad de generarlos y reportarlos está encapsulada en la interface IErrorReporter implementada por los componentes del compilador (Lexer, Parser, Interpreter y ObjectCompiler, que veremos a continuación).

IErrorReporter se apoya en los métodos GenerateError() y su ayudante Report() para indicar que se ha producido un error en una línea determinada, y el booleano HadError para detener el paso de un componente del compilador a otro en caso de que se detectara un error.

El proceso de implementación de este compilador evidenció, como buena práctica de ingeniería de software, la idea de separa el proceso que genera errores y el que los informa, debido a la versatilidad de las formas de mostrar un error al usuario. En nuestro caso, pasamos de imprimir los errores a en una simple aplicación de consola, a posteriormente la consola de Unity y actualmente también son mostrados en la interfaz gráfica dentro de la escena destinada a compilar cartas y efectos, y realmente, la generación de errores por sí misma, nunca fue responsable de saber cómo presentarlos al usuario.

Análisis Léxico

El Lexer o Analizador Léxico es el proceso de consumir caracteres y producir Tokens. Los Tokens son algo así como fichas que tienen algún significado real para un lenguaje de programación. Nuestro Analizador Léxico recorre la entrada línea por línea e intenta identificar los tokens mediante el uso de Expresiones Regulares. Las Expresiones Regulares (también conocidas como regex o regexp) son secuencias de caracteres que conforman un patrón de búsqueda. El proceso se apoya de la clase LexicalComponents que funciona como un registro para verificar los componentes léxicos de nuestro DSL.

Una vez finalizado el proceso, se obtiene una secuencia de Tokens, donde cada uno conoce su tipo y suptipo de componente léxico, y su ubicación (línea y columna) en el código fuente.

En caso de ocurrir un error, será presentado un error léxico al usuario.

Analizador Sintáctico

El análisis sintáctico es la forma mediante la cual el compilador entiende las intenciones del usuario con el código fuente, y devuelve una estructura de datos que represente dichas intenciones de alguna manera óptima para ejecutar la orden que está recibiendo. Nuestro Analizador Sintáctico o Parser, utiliza la técnica conocida como Descenso Recursivo para llevar a cabo la primera tarea, devolviendo un Árbol de Sintaxis Abstracta para la ejecución de la segunda.

El análisis sintáctico de descenso recursivo está estrechamente ligado a las reglas de producción de gramática. Durante el análisis, se empieza por la regla gramatical más externa y se va descendiendo hacia las subexpresiones anidadas antes de llegar finalmente a lo que serían las hojas del árbol sintáctico.

El código dentro del parser es una traducción literal de las reglas de la gramática directamente a código imperativo. Cada regla se convierte en una función cuyo código es el cuerpo de la regla traducido a código.

Cada uno de estos métodos para analizar una regla gramatical produce un árbol sintáctico para esa regla y lo devuelve a quien lo llama. Cuando el cuerpo de la regla contiene un no terminal (una referencia a otra regla) llamamos al método de esa otra regla, esto es lo que produce la recursividad.

El Analizador se apoya en un grupo de métodos, para moverse por la secuencia de Tokens de la manera mas óptima, denominados en el código como Stream Methods,con funciones como Check() (que verifica si el siguiente token en la secuencia coincide con un tipo o subtipo específico), Advance() (que avanza al siguiente token de la secuencia) o Consume() (que espera un token en la siguiente posición de la secuencia y en caso de no encontrarlo, genera un error).

La detección de errores sintácticos fue posiblemente la más compleja de implementar en este compilador. Abortar después del primer error es fácil de implementar, pero es molesto para los usuarios si cada vez que corrigen lo que creen que es el único error en un archivo, aparece uno nuevo. Este compilador intenta mostrar al usuario la mayor cantidad de errores sintácticos posible.

Sin embargo, una vez que se encuentra un solo error, el analizador sintáctico ya no sabe realmente lo que está pasando, y puede informar de una serie de errores fantasma que no indican otros problemas reales en el código. Los errores en cascada pueden asustar al usuario haciéndole creer que su código está en peor estado del que está.

Los dos últimos puntos están en tensión. Queremos notificar tantos errores independientes como podamos, pero no queremos notificar los que son meros efectos secundarios de uno anterior. Aquí es dónde se introduce una función peculiar del parser: la sincronización.

La sincronización (ejecutada por el método Synchronize()), es el proceso donde una vez detectado un error sintáctico, el parser avanza hasta encontrar un "token de seguridad", un token donde es casi seguro encontrar un nuevo comienzo para continuar detectando errores y mostrarlos al usuario.

Intérprete

El intérprete es quien finalmente establece la comunicación con la computadora, es paso final en la traducción a un lenguaje que esta ya entiende, C# en nuestro caso.

El intérprete necesita recorrer el Árbol de Sintaxis Abstracta para realizar la evaluación o ejecución nodo a nodo. Nuestra forma de recorrer el árbol está basado en lo que se conoce como patrón del Visitante.

El patrón del visitante, o patrón Visitor, es una forma de separar el algoritmo de la estructura de un objeto, o más específicamente, permite definir nuevas operaciones sobre una estructura jerárquica de clases sin modificar las clases sobre las que se opera.

En este compilador, nuestro Intérprete y ObjectCompiler heredan de la clase abstracta VisitorBase que implementa las interfaces IVisitor y IErrorReporter.

El método VisitBase() implementado por VisitorBase utiliza programación dinámica para realizar el recorrido por los nodos. Este enfoque evita el problema de utilizar sobrecargas para seleccionar el método a ejecutar (como en las más básicas implementaciones de Visitor), teniendo en cuenta que la sobrecarga se apoya en el tipo estático y esto probablemente cause errores en cuanto el método que se ejecutará. Desde la perspectiva de la programación dinámica, se seleccionará el método a ejecutar en base al tipo del objeto guardado en memoria.

En cuanto a la detección de errores, en nuestro Intérprete son tratados como errores en tiempo de ejecución. Los errores en tiempo de ejecución

son fallos que la semántica del lenguaje exige que detectemos y notifiquemos al usuario mientras el programa se está ejecutando. En esta fase se abandona la idea de detectar todos los errores posibles, ya que no puede predecirse el comportamiento de un programa con errores semánticos.

La clase RuntimeError que hereda de Exception, tiene el mismo comportamiento de una excepción de C#: detener la ejecución del programa. Sin embargo en lugar de usar la propia excepción de C# mostramos un mensaje más acorde a nuestro lenguaje y su respectiva ubicación. En algunos casos, como el tipo de la carta, los rangos de la carta de unidad, o el campo Source dentro del Selector de una declaración de efecto en el OnActivation de la carta, incluso se muestra una sugerencia al usuario de lo que se espera recibir.

Compilador de Objetos

Una vez finalizado el proceso de análisis sintáctico, si este ha ocurrido sin detectar errores, estaremos en presencia de una lista de declaraciones, dichas declaraciones pueden ser una declaración de carta o una declaración de efecto.

En este punto, el compilador de objetos u ObjectCompiler, es quien se encarga de interpretar cada nodo por separado y verifica si dicha interpretación es la esperada para cada componente de la carta o del efecto correspondiente.

Si el componente no coincide con lo esperado se genera y reporta un error, para informar al usuario el componente que ha sido declarado incorrectamente. En el caso contrario se crea un CompiledObject los objetos finales que serán serializados gracias a su método Save() que se apoya en la clase FileFormatter.

Es también en este punto donde se antepone la palabra “Morty” al nombre de la carta elegido por el usuario.

Integración con GwentPro

Las cartas son creadas en la clase CardsCollection la cual representa la colección de todas las cartas que existen en el juego. La colección de cartas se crea en el momento en el que se abre el juego, incluso antes de ser mostrado el menú principal, ya que necesitamos un registro de qué cartas existen para no permitir que una carta con el mismo nombre (o bien, la misma carta) sea guardada mediante el uso del compilador, de la misma manera necesitamos verificar la existencia de un efecto en caso de que una carta declare el uso de un efecto que aún no existe o cuyos parámetros no coincidan con los que declara el usuario en el código fuente. De la misma manera, no será guardado un efecto con un nombre ya

existente. La colección de cartas es actualizada en el momento donde corresponde al jugador elegir mazo, ya que a través del compilador pudo crearse una facción completamente nueva de la que no se tenga registro (al menos como facción).

Las cartas y efectos serializados son almacenados en la ruta "Assets/CardsCollection/Serialized". Dichos archivos serializados coexisten con los archivos de texto plano de los que originalmente se extraía la información para crear las cartas. Solo que ahora la colección de cartas se crea a través de la lectura de estos archivos de texto plano y del proceso de deserialización de los objetos compilados que fueron serializados a binario. Todas las cartas pasan por el método TypeClassiffier() que selecciona desde qué punto de la jerarquía de la clase Card será creada una carta, y los efectos son registrados en la clase abstracta Effect.

El proceso de activación de un efecto para una carta sigue el algoritmo original. A los efectos compilados le corresponde el número de efecto 0, y su ejecución se basa en crear una instancia de intérprete que, tras asignar en el Environment los parámetros, objetivos y contexto correspondientes al efecto, ejecuta el bloque de código (o instrucción única) declarado en el Action del efecto compilado.

Integración con Unity

La capacidad otorgada al usuario para crear sus propias cartas, entró en tensión con la forma en la que inicialmente se creaba la carta "visualmente", que era básicamente a través de imágenes prediseñadas.

Este fue uno de los principales cambios en la integración del proyecto con Unity: la implementación de un algoritmo para construir la representación visual de una carta en tiempo de ejecución a partir de las propiedades de la carta del back-end. Este algoritmo está plasmado en el método PrintCard() del script UICard que puede encontrar en el directorio del proyecto.

Este método es auxiliado por una poderosa y peculiar herramienta: la biblioteca de clases CharacterManager (ubicada en la ruta "Assets/CharacterManager"). Esta biblioteca es una implementación propia de un motor de búsqueda basado en el modelo vectorial de recuperación de la información, adaptado a este proyecto para devolver el nombre de la imagen más similar a lo que puede entenderse del nombre y descripción de la carta que está siendo representada.

El resto de los cambios para integrar el proyecto a la interfaz visual fueron mínimos y sencillos en gran medida gracias a la implementación del patrón de diseño Observer de este proyecto.