Trabajo Práctico Especial

Programación Orientada a Objetos

Grupo 7

1. Diseño

1.1. Estructura Básica

Se modeló el juego Sokoban bajo el paradigma orientado a objetos.

Lo primero que definimos fué una clase **Game** que represente partidas de Sokoban. Un objeto de esta clase tiene atributos tales como nombre del usuario que está jugando, nombre del nivel, e información que nos permita decidir si el jugador ha ganado o no.

En un juego tenemos elementos (los cuales todos heredan de una clase **GameElement**), que los dividimos entre dos tipos: **GameTile**, una baldoza del juego (esto incluye los agujeros, paredes, y objetivos), y **GameObject**, que son los objetos que pueden estar arriba de una baldoza (el personaje del jugador y las cajas).

Dentro de un objeto **Game** debe existir un conjunto de **GameTiles**, almacenados en una matriz en nuestra implementación, representando las baldozas de un nivel de Sokoban. Debido a que un **GameObject** sólo puede estar arriba de una **GameTile**, almacenamos en cada **GameTile** información sobre quién es el **GameObject** que está encima de ella.

1.2. Diagrama UML

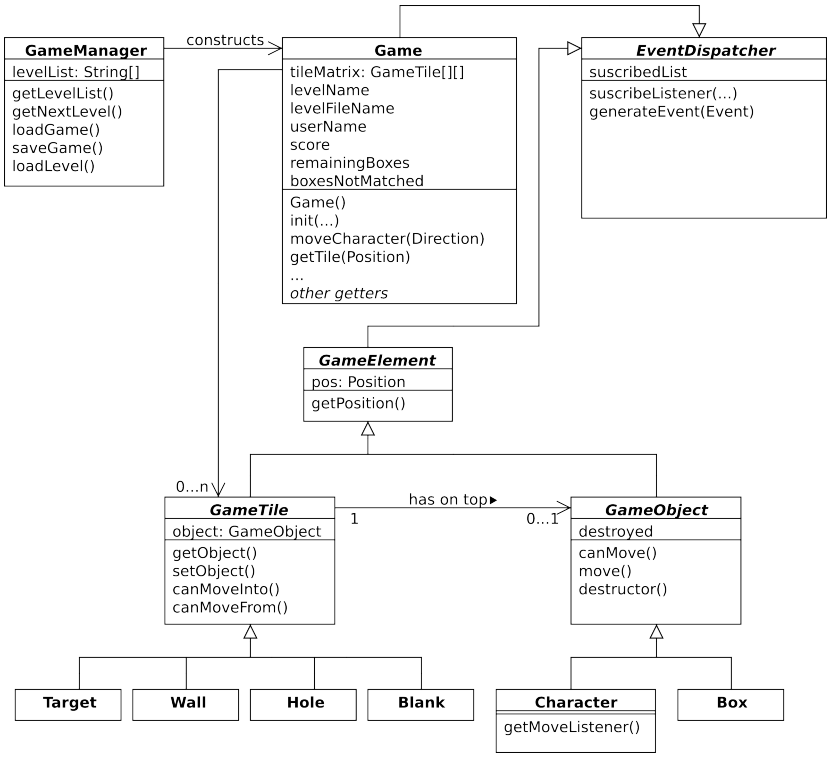

Figura 1: Diagrama de clases

1.3. Eventos

La comunicación interna entre objetos de **gamelogic** varía entre pasaje de mensajes y lanzamiento de **Events**. Este modelo es una implementación del patrón ***Observer*** en la que objetos que implementan la interfaz **EventListener** se suscriben a un **EventDispatcher**, encargado de notificar cada vez que un **Event** es lanzado.

La comunicación al exterior acerca de cambios de estado del juego se realiza puramente a través de este tipo de eventos. Movimiento de objetos, notificaciones de que el juego terminó, cambio del puntaje, son ejemplos de las notificaciones que nuestro *backend* emite. Es responsabilidad del frontend responder a estos eventos. Por otro lado, el *backend* no escucha a eventos externos, sólo responde a pasaje de mensajes. Sí escucha a eventos internos, lo que se describirá posteriormente.

1.3.1. Diagrama UML de Eventos

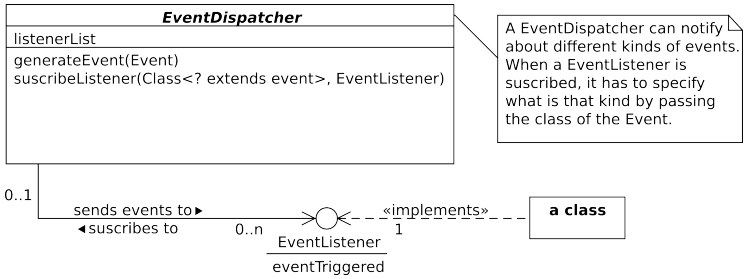

Figura 2: Diagrama de Clases de Eventos

1.4. Movimiento de GameObjects

Al recibir un evento del tipo **MoveCharacterEvent**, que representa una orden de moverse en determinada dirección, el **Character** instanciado en este juego pedirá información suficiente para saber si es posible realizar un movimiento. Este evento contiene una referencia a **Game** y dispara el método *canMove()* del **Character** y los pasos que éste realiza para responder bien son:

- Preguntar si el **GameTile** vecino tiene un objeto arriba
  - Si contiene un objeto, determinar si se puede mover en esa misma dirección llamando al método *canMove()* del **GameObject**.
- Preguntar al **GameTile** debajo suyo si puede salir en esa dirección.
- Preguntar al **GameTile** vecino si se puede mover entrando en esa dirección.

Estas comprobaciones son realizadas por pasaje de mensajes y no de eventos, debido a que todas necesitan devolver un valor y por simplicidad no agregamos funciones de *callback*.

Se utiliza un evento **moveCharacterEvent** interno para realizar esto para evitar que la instancia de **Game** mantenga información adicional acerca del **Character**. De este modo, **Game** sólo tiene una matriz de **GameTiles**. Esto fué decidido para evitar referencias desde el Game hacia el **Character**, para no dar saltos dentro del árbol de clases.

1.5. Lógica del Estado del Juego

Cada vez que el estado de un **GameElement** cambia, es disparado un evento de tipo **StateUpdatedEvent** informando que un **GameTile** dejó o pasó a tener un nuevo **GameObject** sobre él, o bien un **GameObject** actualizó su posición.

Los **GameTiles** de tipo **Target** además, disparan un evento **TargetMatchedEvent** cuando es posicionada una **Box** del mismo color sobre el tile, y un evento **TargetUnmatchedEvent** cuando una **Box** de su color deja la casilla. La clase **Game** escucha este tipo de eventos y al ser disparado uno, actualiza el contador del juego y en caso de que dispara un nuevo evento **ScoreUpdatedEvent**, avisando de un cambio en el puntaje del jugador. El orden en que ocurren estos cambios es el siguiente:

- El **GameTile** del origen ha cambiado su estado: dejó de ser anfitrión de un **GameObject**.
- El **GameTile** destino ha cambiado su estado: contiene un **GameObject**.
- **GameObject** ha cambiado su estado: cambió su posición.

Asimismo, un **Hole** al recibir un **GameObject**, le manda un mensaje para avisarle que ha sido destruído. Un objeto al ser destruído manda un evento **DestroyedEvent**. Este tipo de eventos es escuchado internamente, debido a que **Game** debe disminuír un contador de **Box**es en el nivel y enviar eventos de tipo **GameOverEvent** si el objeto que cayó es el **Character**.

En caso de que se cumpla la condición de que la *cantidad de cajas actuales en el nivel* sea igual a la *cantidad de objetivos en el nivel con una caja de su color arriba* (ambas son atributos guardados en la instancia de **Game**), el juego fué ganado, y se lanza un event de tipo **GameFinishedEvent**

1.6. Nuevas Partidas

La creación de nuevos juegos se realiza usando el patrón de diseño ***Factory***. Una clase **GameManager** es encargada de leer datos sobre cómo crear un nuevo juego a partir de un archivo e instanciar un nuevo **Game** a partir de esos datos, ya sean estos de una partida guardada o no. La clase asegura que se instanciaran todos los **GameElement** que se lean del archivo (si el archivo no está corrupto, en cuyo caso se debe tirar una **Exception**), en orden especifico: primero los **GameTiles** y luego los **GameObjects**.

Esta clase permite ser extendida y sus métodos fueron subdivididos para que puedan ser sobreescritos o sobrecargados diferenciando las subclases de **GameElement** que son instanciadas.

1.7. Elementos Auxiliares del Juego

Distintas clases del *backend* sirven a la lógica del juego:

- **Position**: objeto inmutable que representa una posición del tablero.
- **Direction**: objeto inmutable que representa una dirección en la que un **GameObject** puede moverse.
- **Color**: representación de un color por sus componentes RGB.
- **ElementType**: un enumerador para corresponder cada tipo de **GameObject** con un entero.
- **Highscores**: clase encargada de guardar los mejores puntajes dado el nombre del archivo del nivel.

2. Desafíos Encontrados

2.1. Modelado Inicial

Nos llevó varias horas ponernos de acuerdo en cómo representar los elementos del juego en una manera que se pudiera distinguir bien el *backend* del *frontend*. En un diseño original que estaba claramente fallado, el *frontend* era una herramienta fundamental que necesitaba el *backend* para generar nuevos juegos. Con esto no había independencia entre el backend y el frontend.

Heredamos de este modelado inicial el patrón ***Factory***, pero con una mejor independencia: la clase **GameManager** que construye **Game** y el resto de los **GameElements** fué dividida de manera tal que cualquier etapa de la formación del juego pueda ser sobreescrita y/o extendida por una clase hija.

2.2. Diseño del Frontend

Al adoptar el modelo MVC, nuestros modelos de *backend* debían tener clases "hermanas"que las compusieran. En un principio, estuvimos tentados por la opción ïmperativa", hacer un mapeo de los *Strings* con los nombres de las clases del *backend* hacia imágenes, con casteos para obtener más datos,

como el color de las cajas. Abandonamos esta idea por parecernos una mala práctica y nos mantuvimos dentro del paradigma orientado a objetos.

La correcta separación entre la *Vista* y el *Controlador* requirió también de un refactoreo entero de ambos, debido a que nos aceleramos a intentar "tener algo andando".

2.3. Implementación

A medida que debíamos pasar del diseño a la programación, nos encontrabamos cada día con nuevos obstáculos: debates sobre si hacemos esto de tal o cual manera fueron muy abundantes durante el proceso y encontramos que varias decisiones requirieron que volviéramos a replantear nuestro diseño en un par de ocaciones.

En momentos, cuando encontramos un *bug* importante durante la implementación, escribimos un *Unit Test* para mostrar cómo debía ser la funcionalidad y corregir el código para que estas pruebas no fallaran. Esto probó ser muy eficiente y necesitar escribir el *test* nos ayudaba a entender aún más claramente el funcionamiento del flujo de nuestro diseño.

2.4. Trabajo en Equipo

Luego del diseño inicial, fué cuestión de dividir bien el trabajo y en horas gran parte del juego estaba andando correctamente.