Programación Orientada a Objetos

Trabajo Práctico Especial

Sokoban

Autores:

Daniel Lobo - 51171

Teresa C. Di Tada - 52354

Felipe Martinez - 51224

Resumen

El objetivo de este trabajo práctico consistió en implementar una variante del juego *Sokoban* en el lenguaje Java. El fin de este informe es explicar la jerarquía diseñada usada, exponer los problemas que surgieron a lo lardo de su desarrollo y destacar las decisiones tomadas al respecto.

7 de junio de 2012

Índice

1. Introducción

En este juego, se dispone de un tablero con un jugador, cajas, cajas bomba, destinos y paredes.

La idea del juego es mover las cajas de un determinado color, hasta los destinos de dicho color. Una vez que cada destino tiene sobre él la caja correspondiente y en el caso de que en el tablero no sobren cajas, el nivel está completado. A su vez, sobre el tablero se presentan agujeros, cuyo objetivo es recibir todas las cajas extras que exista en el nivel y eliminarlas. Si el jugador llega a caerse dentro de uno de estos agujeros, se pierde la partida. Otra manera de perder la partida, es cuando el contador que posee alguna de las cajas bomba, llega al cero.

Este trabajo cuenta con la finalidad de aplicar todos los conocimientos aprendidos en la materia, aprovechando las ventajas de un diseño orientado a objetos, como son la herencia y la reusabilidad de código. Asimismo, se pone en práctica la implementación de la interfaz gráfica mediante el uso del GUI provisto por la cátedra, y el uso de la biblioteca gráfica *Swing*.

2. Jerarquía diseñada

2.1. Tipos de elementos

La primera decisión de diseño que tomamos fue la de establecer dos capas fisicas. La primera consta de pisos, pudiendo ser estos destinos o simplemente pisos, o agujeros. La segunda capa contiene paredes y objetos móviles tales como cajas, cajas bomba, y el personaje, estos últimos heredan representativamente de la clase `Movable`.

2.2. Condición de ganado

La siguiente decisión tomada fue la de definir la situación en la que se gana un nivel logicamente. Concluimos que la mejor forma de definirla era chequear que, en primer lugar cada `Target` tenga sobre si mismo una caja de su mismo color y que al ocurrir esto, se ilumine la caja. La segunda condición para ganar consiste en que la cantidad de cajas existentes sea igual a la cantidad de targets. Teniendo que eliminar las cajas cuyo color no existe en los destinos a lo largo de todo el tablero por algún agujero.

2.3. Listener

Por otro lado, escribimos un listener que se encarga de comunicar el frontend con el backend. Si bien esta implementación fue hecha muy al final del desarrollo del juego, se la consideró desde un primer momento. Más detalles sobre la misma, se puede ver en la sección de **Problemas encontrados**.

2.4. Jugador vs. Personaje

Para finalizar, decidimos tambien que el usuario del juego en si no podia ser equivalente a la ficha del jugador (usada por la clase `Smile`) ya que corresponden a dos esquemas de información y comportamiento distintos.

2.5. Consideraciones

Cuando un usuario ingresa su nombre para comenzar una nueva partida sin guardar, se le pide que el nombre no sea vacío (o sea, que el String no quede en null), que no contenga numerales (#) ni comas (,). Se asume que no son necesarios en un nombre.

3. Problemas y decisiones

3.1. Observador

3.1.1. Problema encontrado

El primer problema que encontramos consistió en resolver la notificación de cambios en el tablero desde el backend hacia el frontend con el objetivo de que este último muestre los cambios necesarios al usuario. Intentamos inicialmente algunas aproximaciones bastante imperativas y que interferían con el estilo de código y diseño que veníamos utilizando en las distintas clases del juego.

3.1.2. Decisión tomada

Finalmente, terminamos implementando una versión sencilla con una interfaz llamada `BoardListener` en el backend que se comunica con una clase llamada `InstanceBoardListener` en el frontend que implementaba dicha interfaz. Nos pareció una aproximación económica y que entraba perfectamente en la lógica del juego y que se alinea perfectamente con el estilo de desarrollo del mismo.

3.2. Forma imperativa del parser

3.2.1. Problema encontrado

La implementación del parser no nos permitió hacer un buen uso del paradigma orientado a objetos por la naturaleza imperativa de los archivos que hubo que parsear.

3.2.2. Decisión tomada

De todas manera, toda la información extraída de los archivos parseados (tanto partidas guardadas como juegos nuevos) fue cuidadosamente asignada a las distintas clases del backend que implementaban un fuerte diseño orientado a objetos.

3.3. Uso de componentes

3.3.1. Problema encontrado

Otro desafío de diseño que se nos presentó fue cómo manejar los multiples tipos de comportamiento que tenían los distintos objetos del tablero. Inicialmente se pensó llenar la matriz con vacío, que se podría decir que es la clase `Tile`, pero luego necesitabamos poner contenido sobre el vacío.

3.3.2. Decisión tomada

Por lo tanto, se optó por rellenar la matriz inicialmente con elementos del tipo `Floor` y luego reemplazar en caso de que haya un agujero y poner sobre los pisos los elementos restantes. Estos eran movibles, excepto las paredes.

3.4. Movimiento e interacción entre los elementos

3.4.1. Problema encontrado

Otro problema fue el movimiento del jugador al intentar entrar a un piso donde hay algo o no. El jugador debería poder entrar si el piso al que quiere acceder no tiene contenido o si tiene una caja (o caja bomba). En este último caso, el jugador podrá ubicarse en la posición donde estaba la caja, desplazando a la misma un casillero hacia la dirección de desplazamiento. Esto es posible si en ese otro casillero a ocupar, no hay una pared u otra caja.

3.4.2. Decisión tomada

Decidimos resolver este problema a nivel del método `move` en la clase `Board` que le pregunta al piso de al lado si es posible poner algo sobre el él. El casillero se fija si tiene contenido o no, y si tiene, intenta hacer que ese elemento movible acceda al próximo casillero en la misma dirección.

3.5. Try-catch

3.5.1. Problema encontrado

Al terminar el desarrollo del frontend, teníamos una gran cantidad de bloques try-catch que eran innecesarios y además no se enviaba al usuario un cartel o signo que indique el error cometido.

3.5.2. Decisión tomada

Para eliminar muchos de esos bloques try-catch, cambiamos las excepciones que no eran necesarias de atrapar en el backend y alguna otra en el parser a tipo `RunTimeException`. Por otro lado, a las excepciones atrapadas finalmente en el frontend se les agregó un cartel a mostrar al usuario indicando el error cometido.

4. Conclusión

Consideramos que el trabajo cumplió con su objetivo de fortalecer los conceptos adquiridos durante este cuatrimestre.

Aunque entendemos que el objetivo de la materia no es enseñar la implementación de interfaces mediante librerías gráficas, creemos que este trabajo en particular contenía un fuerte nexo entre front-end y back-end, y podríamos haber logrado un mejor resultado si se hubiesen dictado algunos conceptos más avanzados de *Swing*. De todas maneras, hemos resuelto la interfaz gráfica con las herramientas que teníamos, pero sabemos que se podría implementar de una mejor manera.

Asimismo, fue interesante afrontar este desafío sin salirse del paradigma orientado a objetos.