Programación Orientada a Objetos

Trabajo Práctico Especial

Silversphere

Autores:

Daniel Lobo - 51171

Agustín Scigliano - 51277

Felipe Martinez - 51224

Resumen

El objetivo de este trabajo práctico consistió en implementar una variante del juego *Silversphere* 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 noviembre de 2012

Índice

1. Introducción

En este juego, se dispone de un tablero con un jugador, cajas, cajas dentro del agua, un cubo de hielo, un interruptor, un destino que inicialmente no aparece en el tablero, agua y árboles.

La idea del juego es mover el cubo de hielo y ponerlo sobre el interruptor para que aparezca en el tablero el destino al que luego el jugador tiene que llegar. El nivel está completado una vez que el jugador se posiciona sobre el destino. A su vez, existe agua en el tablero. El jugador puede mover cajas hacia una posicion donde hay agua para luego usar esa posicion para caminar sobre ella y utilizarla como "puente" en algunas ocasiones. Si el jugador llega a caerse al agua, se pierde la partida.

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, interruptores o simplemente pisos, o agua. Además se puede tener una caja en agua llamada `WaterBox` que puede tener contenido también. La segunda capa contiene paredes y objetos móviles tales como cajas, el cubo de hielo, y el personaje, estos últimos heredan representativamente de la clase `Content`. Paralelamente, al ser seteado el tablero se ubican los árboles que no poseen contenido.

2.2. Condición de ganado

Para alcanzar la condición de ganado, es necesario ubicar la pieza `Ice` sobre un `Interruptor`. Esto hará que aparezca un `Target` en el tablero y luego se debo posicionar al `Player` en dicho `Target`.

2.3. Consideraciones

Cuando un usuario llega a la condición de ganado o perdido de una partida, se le presenta un mensaje de diálogo comentándole lo ocurrido y, acto siguiente, se lo retorna al menú principal del juego en ambos casos.

3. Problemas y decisiones

3.1. Estados de juegos y comunicación con el frontend

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

Se decidió implementar un `enum` en el backend que cambia de estado y elije entre 3 estados diferentes: `Win`, `Playing`, `Lose`. Este `enum` es el encargado de establecer una "conexión" entre backend y frontend

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 `Cell`, 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 los `Tree`.

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 (o una `WaterBox`) 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 un cubo de hielo. 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 y, en el caso del `WaterBox`, desplazarlo todos los casilleros posibles hacia la dirección de desplazamiento hasta encontrarse con un `Water`, un `Tree` o una `Box`. Vale aclarar que en el caso de desplazar una `Box` el movimiento será posible si en ese otro casillero a ocupar, no hay un `Tree` u otra `Box`.

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 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 frontend 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.