Lasers and Mirrors
Programación Orientada a Objetos

Trabajo Práctico

Federico Bond
Legajo 52247

Kevin J. Hanna
Legajo 52024

Fecha de Entrega: 4 de Noviembre, 2011

Índice

1. Introducción

El objetivo de este trabajo práctico consistió en implementar una variante del juego Lasers & Mirrors en el lenguaje Java. El trabajo nos permitió reforzar varios conceptos de diseño e implementación de objetos aprendidos en la materia.

2. Explicación de la jerarquía

El proyecto sigue a grandes rasgos una arquitectura MVC (Modelo-Vista-Controlador). La elección de este patrón nos permitió separar bien los distintos componentes del programa, principalmente el frontend del backend, de manera prolija y elegante.

El modelo se encuentra en el paquete `game`, específicamente en la clase `Game`. En este mismo paquete se encuentran también las interfaces `Cell`, `Beam` y `Drawable` que permiten exportar objetos al frontend para su posterior visualización. La clase `Ray` modela un rayo láser que se puede propagar por el tablero.

En el paquete `frontend` se encuentran todas las clases relacionadas con elementos visuales, en particular las interfaces `View` y `ViewContainer` con sus respectivas implementaciones, y la implementación del controlador.

En el paquete `tiles` se encuentran las implementaciones de los distintos elementos que se pueden incluir en el tablero. Dentro de `tiles`, los paquetes `direction` y `propagation` contienen componentes reutilizables que definen el comportamiento de cada elemento.

En los paquetes `parser` e `iogame` se encuentran las clases que se encargan del manejo de archivos, tanto para cargar archivos de tablero como para guardar y recuperar juegos.

Por último, en el paquete `tests`, como su nombre lo indica, se encuentra el conjunto de tests unitarios diseñados.

3. Organización del trabajo

3.1. Herramientas utilizadas

Ambos miembros del equipo teníamos experiencia trabajando con el sistema de control de versiones Git, por lo que decidimos desde el primer momento alojar el proyecto en el servicio GitHub[1] para poder facilitar la colaboración. Esto nos permitió iterar muy rápido, haciendo cambios sustanciales en el diseño del programa sin perder tiempo combinando manualmente el código escrito por cada uno.

[1] GitHub es un servicio que ofrece alojaminento de repositorios Git y herramientas de colaboración a través de una interfaz muy usable http://github.com/

4. Problemas encontrados

4.1. Observer y Observable

El primer problema que tuvimos que encarar fue la notificación al frontend de cambios en el tablero. Nuestra primera implementación hacía uso de las clases provistas por la librería de Java, en particular `java.util.Observer` `java.util.Observable`. Sin embargo, éstas clases resultaron bastante incómodas para la funcionalidad que queríamos utilizar y terminamos implementando una versión más sencilla de este patrón con un sólo observador y tres callbacks bastante específicos. Estos callbacks deben ser implementados por cualquier clase que actúe de `Observer`.

4.2. Uso de Componentes

Otro desafío de diseño que se nos presentó fue el manejo de los distintos tipos de comportamiento que tenían los objetos del tablero (en adelante, Tiles) frente a la forma en que propagaban rayos y la forma en que rotaban. En un primer momento probamos usar interfaces para definir este comportamiento y casteos explícitos para acceder al mismo. Esta solución no nos permitía reusar la lógica, que debía estar implementada en cada clase. El código resultante bastante incómodo de manejar. Por sugerencia de Pablo Costesich, alumno también del ITBA, investigamos el uso de componentes para compartir comportamiento común de forma reusable y elegante.

Se implementaron dos tipos de componentes, unos de Dirección y otros de Propagación. El código resultante permitió encapsular bien la funcionalidad, logrando también una alta cohesión. Los componentes de propagación resultaron los más difíciles de implementar porque necesitaban acceso al estado del tile, por lo que caímos en el problema bastante conocido de compartir información entre componentes. La solución sencilla fué incluir una referencia al tile desde la construcción del componente, pero encapsular todo el acceso a los datos del mismo a través de métodos wrappers protegidos en la clase abstracta `PropagationComponent`.

4.3. Interfaces para frontend

Desde el principio estabamos decididos a compartir desde el backend sólo lo justo y necesario con el frontend, lo que nos planteó varios desafíos a la hora de definir las interfaces a usar para ejercer este ocultamiento. En particular, al ejecutar el callback de actualización de celda, queríamos evitar pasar un objeto tile completo, sino algo que sólo permitiera acceder a los métodos necesarios para dibujarlo. Nuestra primera aproximación al problema fue definir una sóla interfaz `Drawable`. Esta interfaz nos permitía ocultar tiles frente al frontend, pero perdía su valor semántico si a un Drawable le pedíamos sus rayos, para poder tambien dibujarlos (los rayos tendrían

que tambien implementar Drawable, pero no tenía sentido pedirles a su vez sus rayos). Luego de varios días de discusión nos dimos cuenta que lo que teníamos era un problema de nomenclatura y no de diseño, y fue entonces que la interfaz Drawable pasó a ser base para otras dos, con mayor valor semántico: `Cell`, que oculta un `Tile`, y `Beam` que oculta un `Ray`. Esto nos permitió además ganar valor semántico en el frontend y hacer el código más lejible.

4.4. Forma imperativa del parser

La implementación del parser no nos permitió hacer un buen uso del paradigma orientado a objetos dada la naturaleza imperativa de los archivos que hubo que parsear, pero si intentamos hacer uso de conceptos aprendidos como expresiones regulares para simplificar el código. Además, separamos el parseo en sí de los valores posibles (provistos por la cátedra) para la construcción de cada tile (en `TileValues`). Esta información está en cierta forma repetida en cada clase Tile, pero extraerla programáticamente hubiera aumentado la complejidad del código a un nivel que no se justificaba dado el tamaño del proyecto.

5. Decisiones de diseño

5.1. Arquitectura general

Como ya lo hemos mencionado, el proyecto sigue una arquitectura MVC. Creemos conveniente destacar que este diseño no surgió desde el principio sino que se decidió implementar en respuesta al crecimiento orgánico que sufrió el frontend. Nos resultó provechoso ya haber implementado parte del frontend en Swing para conocer el manejo de ésta librería en la práctica y tomar una mejor decisión acerca de cómo organizar el código.

En las primeras iteraciones del programa, la ventana, los paneles y el menú estaban muy acoplados y nos resultaba difícil lograr una buena separación sin plantear la figura del controlador, que iba a responder por las vistas y manejar el flujo principal del programa. Usando un controlador, pudimos manejar la comunicación entre el frontend y el backend de forma abstracta.

5.2. Game

La lógica del juego en sí no presentó problemas muy grandes a la hora de implementarla, pero sí perdimos bastante tiempo al principio al encontrar inconsistencias en los métodos que trabajaban con coordenadas, donde los parámetros estában en el órden equivocado. Esto motivó el uso de la nomenclatura `row` y `column` en lugar de `x` e `y` para facilitar la comprensión y

la búsqueda de errores.

5.3. Separación del tablero

Se separó desde un principio la lógica de juego del manejo del tablero, para facilitar el testeo y evitar incluir código de validaciones genéricas de tablero en la implementación del juego.

5.4. Método de guardar y cargar juego

Inicialmente, la clase Game contaba con métodos para cargar y guardar una partida. Decidimos armar un paquete `iogame` para organizar todas las opciones de guardado y cargado. Aunque en nuestro caso solo es posible mediante serialización (`IOSerialize`), se podrían proveer otras clases que implementen `IOHandler`.

5.5. Propagación de rayos

La propagación de rayos fue una de las partes más complicadas de diseñar. La separación en componentes de propagación permitiría que los rayos se propaguen a través de los tiles como si estos fueran una caja negra, moviendo efectivamente la lógica de routeo a cada componente, pero todavía quedaba definir cómo manejar las bifurcaciones. El primer diseño, implementado de forma recursiva, permitía un manejo sencillo de rayos, porque el componente pasaba el mensaje al rayo con la siguiente acción a ejecutar. La desventaja era que los objetos `Ray` eran bastante pesados, dado que contaban con referencias tanto a su posición como al tablero, que eran copiadas junto con el estado del rayo para persistir cada segmento del mismo (su trazo). La reimplementación de la propagación de forma recursiva permitió eliminar estas referencias porque son inyectadas por el juego sólo al momento de iniciar la propagación. Con este diseño, se incorporó una pila de bifurcaciones al que cada componente puede apilar tantas bifurcaciones como quiera. La última ventaja del diseño iterativo es que el rayo sólo continúa su curso de propagación una vez que retornó del componente y se propagaron sus bifurcaciones, lo que evita problemas relacionados con rayos que retornan a un tile luego de haber seguido su curso, lo que podría provocar comportamiento inadecuado si uno no es muy cuidadoso.

5.5.1. Prevención de ciclos

Para evitar que los rayos queden atrapados en un ciclo infinito de espejos, si el `PropagationComponent` detecta que está persistiendo el trazo de un rayo del mismo color, lo detiene en ese momento, con lo cual no hace falta tener en cuenta esto en cada subclase.

5.6. View y ViewContainer

Dada la naturaleza de las aplicaciones de escritorio, buscamos separar desde el principio la `View` propiamente dicha, que maneja los elemento visuales del juego, del contenedor en el que se encuentra. En este caso, el contenedor es un `JFrame` que expone una interfaz para realizar acciones incluso cuando un juego no ha sido cargado o que exceden la responsabilidad del `View`.

5.7. Paneles

Para simplificar el manejo de ventanas, decidimos implementar la vista con un sólo JFrame cuyos paneles pueden ser reemplazados en tiempo de ejecución cuando se carga un juego. Consideramos que está decisión también mejoró la usabilidad final del programa, al no tener el usuario preocuparse por el manejo de las ventanas.