Lasers and Mirrors
Programación Orientada a Objetos

Trabajo Práctico

Federico Bond
ndelegajo

Kevin J. Hanna
ndelegajo

Fecha de Entrega: 4 de Noviembre, 2011

Índice

1. Introducción

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

2. Diseño

2.1. Explicación de la jerarquía

El proyecto sigue a grandes rasgos una arquitectura MVC (Modelo-Vista-Controlador).

El modelo se encuentra en el paquete `game`, específicamente en la clase `Game`. Las interfaces `Cell`, `Beam` y Drawable sirven para exportar objetos al frontend. La clase `Ray` modela un rayo láser que se puede propagar por el tablero.

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 trajo también otras ventajas Esto nos permitió iterar muy rápido, haciendo cambios sustanciales en el diseño del programa sin perder tiempo intentando combinar manualmente el código escrito por cada uno.

4. Problemas encontrados

4.1. Observer

El primer problema que encontramos consistió en resolver la notificación al frontend de cambios en el tablero. La primera implementación hacía uso de las clases provistas por la librería de Java, en particular Observer (`java.util.Observer`) y Observable (`java.util.Observable`). Sin embargo, éstas resultaron bastante incómodas para los eventos que queríamos utilizar y terminamos implementando una versión más sencilla con un sólo observador y tres callbacks bastante específicos, que deben estar implementados por cualquier clase que implemente la interfaz `Observer`.

[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/

4.2. Uso de Componentes

Otro desafío de diseño que se nos presentó fue cómo manejar los multiples tipos de comportamiento que tenían los distinto objetos del tablero (en adelante, Tiles) frente a la forma en que propagaban rayos y la forma en que rotaban. En un primer momento se probó con iterfaces, pero el código resultaba bastante incómodo de manejar y carecía de elegancia. 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 quedo, a nuestro criterio, muy elegante y flexible, con un correcto encapsulamiento y alta cohesión. Los componente de propagación resultaron más difícil de implementar porque necesitaban acceso al estado del tile, por lo que caímos un el problema bastante conocido de compartir información entre componentes. La solución sencilla fué incluir una referencia al tile desde la construcción del componente, pero manejar todo el acceso a los datos del mismo a través de 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 un par de desafíos para 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 días de discusión, nos dimos cuenta que teníamos más un problema de nombres que de diseño, entonces 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 ademas ganar valor semántico en el frontend y hacer el código más elegante.

4.4. Forma imperativa del parser

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, pero si intentamos hacer uso de conceptos aprendidos como expresiones regulares para simplificar el código.

5. Decisiones de diseño

5.1. Arquitectura general

El diseño general del juego responde a la arquitectura MVC (Modelo, Vista, Controlador), lo que permitió una muy buena separación de responsabilidades entre los componente del programa. Creemos importante destacar que este diseño no surgió desde el principio sino que se decidió implementar en respuesta al crecimiento orgánico que sufrió la implementación del frontend. Nos resultó provechoso haber ya implementado parte del frontend en Swing para conocer cómo se manejaba esta librería en la práctica y tomar una mejor desición 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. De esta forma, el controlador, cuya inter La implementación del juego Lasers and Mirrors intenta recrear una arquitectura MVC (Model, View y Controller). En el modelo (`Game.java`), se encuentra toda la lógica de negocios del juego. Las vistas, en nuestro caso las clases que implementan View, sirven para interactuar con el usuario. El controlador (`GameController.java`) es un intermediario entre las vistas y el modelo. Este sigue siendo de frontend y se encarga de llamar al modelo que está en el backend.

5.2. Game

La lógica del juego en sí no presentó problemas muy grandes a la hora de implementar, 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. Game vs. Board

Una de las decisiones que tomamos fue hacer un tablero genérico (con celdas), independiente del juego para el que se use. Es decir, las constantes de tamaño máximo y el método para validarlo se encuentran en la clase Game y no en la clase Board.

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, se podría extender a otras formas.[2]. De la misma forma se llegó a la conclusión de separar el parser de Game.

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 imcorporó 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 prestar atención a eso en cada subclase.

[2]Habría que armar un controlador con los métodos save y load, pero al ser static es imposible en esta versión de Java