JDungeon: clon de Dungeon Desktop

Matías Domingues Sebastián Maio
Pablo Alejandro Costesich

June 9, 2011

Abstract

El informe detalla el desarrollo del trabajo práctico, tanto en el diseño de los objetos como en los problemas encontrados y las soluciones propuestas.

0.1 Frontend

Durante la produccion del Front-End se decidio utilizar un unico JFrame teniendo los distintos panels por separados asi al momento de realizar cambios, estos simplemente se removian/insertaban en caso de que fuera necesario.

Para ello se decidio implementar los siguientes JPanels:

MapDrawer: Esta clase se encarga de manejar el tablero llamado GamePanel provisto por la catedra. Genera las imagenes y las pone a partir de la ubicacion que se le es entregada por el Game. Asimismo esta clase contiene el MouseMotionListener el cual es el encargado de informar a el Game cuando el jugador posiciona el mouse sobre el tablero para poder modificar la informacion correspondiente en el DataPanel que se encuentra a la derecha. Dado el modo en el que se encuentra implementado nuestro listener (El cual es explicado en Game) tuvimos un problema al ingresar las imagenes donde el jugador se encontraba parado sobre sangre. Esto fue solucionado insertando los elementos en un TreeSet en donde el comparator le daba prioridad a cualquier elemento que fuera instancia de Vulnerable. GUI: Interface que interactua entre el Menu, MapDrawer y el Game.

Los metodos que tienen que ser implementados al momento de implementar la interface son: NewGame, save, load, updatePoint, quit, end, restart. La que mas vale la pena destacar es updatePoint la cual se encarga de enviar a Game por que punto fue pasado el mouse para luego ser enviado a el DataPanel.

Game: Funcion principal del FrontEnd, esta funcion extiende JFrame y cumple la funcion de contener la informacion sobre el World Durante la implementacion de esta surgieron varios problemas. En un inicio se habia decidido utilizar el observer de la API de Java, pero luego nos dimos cuenta que era necesario cambiar este metodo ya que teniamos al menos 3 clases a los cuales teniamos que subscribirnos y el problema era al momento de que entraba un evento teniamos que ver de que clase venia y la unica forma de realizarlo con dichos observers era con un instance of. Por lo tanto se decidio implementar nuestros propios "Observers" cuyos eventos eran enviados a mensajes ya implementados y especificos para dicha tarea. Otro de los problemas con los cuales nos encontramos fue al momento de iniciar una nueva partida, lo que ocurria

era que los observers no se cerraban haciendo que el jugador se mueva en la direccion solicitada la cantidad de veces en que fue realizado el Nuevo Juego, esto se arreglo poniendo el listener en el constructor de la clase Game.

DataPanel: panel que se encuentra a la derecha que muestra el nombre y apellido, no tuvimos mayores problemas con dicha implementacion, simplemente cada vez que el jugador del juego se solicita que se haga un update de los datos.

1 Parser

La idea principal del parser, dado su carácter imperativo, fue intentar utilizar objetos lo mejor posible, y hacer énfasis en un diseño modularizado y prolijo. Para este fin, e hicieron 3 clases, una encargada del manejo de archivos, llamada GameReader, otra clase encargada de crear los objetos del tablero, GameFactory y otra clase encargada de utilizar las dos anteriores y devolver el mundo de juego esperado.

Este diseño fue contemplado con el fin de que si se cambiase la forma de leer los archivos, o se cambiasen (o agregasen nuevas) todas las criaturas del mundo, no tuviese que refactorearse todo el parser, sino simplemente la clase encargada a ello.

En las primeras versiones, las clases se hicieron concretas, posteriormente se las hizo abstractas, dado que se ameritó que realmente no debería instnciarse un parser, por lo cual todos sus métodos son static.

En cuanto a los métodos saveGame y loadGame, se decidió ubicarlos en WorldAssembler por dos motivos: Se ameritó que dado que esos dos métodos debían manejar archivos, su lugar estaba dentro del parser, asi el manejo de archivos no saldría del mismo. Y en segundo lugar, se lo colocó en WorldAssembler y no en GameReader, ya que el encargado de crear el mundo de juego es GameAssembler, por ende, se decidió que debía ser su responsabilidad saber guardarlos y cargarlos, siendo, nuevamente, esta la unica clase que crea/carga mundos de juego(class World).

2 World

World contiene el comportamiento específico del juego. Se diseñó de manera tal que fuera posible desacoplar el mapa de los jugadores, siendo el árbitro entre las entidades y el frontend.

El núcleo de Word se basó en el patrón Publish/Subscribe, que recibe un objeto Listener, el cual cuenta con métodos que son llamados según eventos. Esto permitió separar el valor de retorno de las funciones tales como move(), que de otra forma deberían ser chequeadas en los puntos donde son llamadas y no por handlers específicos.

3 Board

Board se diseñó como una grilla de objetos con capacidades aumentadas para la búsqueda geoespacial y arbitraje de desplazamientos. Como tal, fue indispensable en su creación separarla del resto de los objetos, ya que de lo contrario contendría información propia de las entidades y de World.

En move() se utilizó el despacho en las entidades actuantes para resolver el movimiento. Primero se realiza un despacho a todas las entidades que son impactadas por otra, y luego a la inversa (la entidad que impacta recibe las entidades con las que impacta).

Como inconveniente particular, el hecho de almacenar sólo la información mínima indispensable para referenciar la posición creó la necesidad de emplear el operador instanceof y métodos similares para recuperar información en cada nivel. Sin embargo, cada una de estas técnicas fue contenida en clases abstractas que implementan los detalles particulares.

4 Filters

El uso del tablero, como fue especificado en el punto anterior, trajo aparejado problemas con la pérdida de información de tipos. Para un manejo más seguro y para facilitar la búsqueda en el mapa, se optó por crear un filtro basado en interfaces generativas e inspirado en los ORMs (en particular SQLAlchemy).

El filtro redujo la complejidad de unas cuantas tareas en World (tales como sincronización con la interfaz gráfica y borrado de áreas no exploradas) y otras clases del paquete core, por lo que su complejidad interna fue un sacrificio aceptable.

5 Entity y sus subclases

Entity fue pensado como el motor principal de las instancias en el mapa. Cada subclase Entity contiene los métodos principales para el movimiento.

En un principio se planteó la necesidad de una interfaz Collidable, pero finalmente se optó por delegar esta capacidad a dos métodos separados que pudieran ser sobreescritos y universales.

Vulnerable es la clase maestra de los monstruos y el jugador. Se diseñó para contener el comportamiento de pelea y de nivel común a cada instancia móvil. Si bien los métodos de experiencia están en esta clase, se decidió implementarlo de este modo para reducir el uso de instanceof. La clase Monster ignora totalmente la experiencia asignada. A su vez, Vulnerable puede ser observada para saber si la instancia muere, que es delegado a World.

Player extiende los controles de movimiento necesarios para el juego. Si bien en World éstos son públicos, es más conveniente tenerlos asociados a la instancia del jugador. Player también es observable, y contiene la lógica específica de actualización en movimiento (dispara eventos que son capturados por World y notificados a los listeners de éste como eventos de actualización).

Monster fueron diseñados conteniendo la lógica de daño y salud. Las subclases Snake, Golem y Dragon sólo implementaron los valores específicos.

Bonus, Splat y Wall fueron diseñadas como clases con comportamiento mínimo y no presentaron problemas. 1