Trabajo práctico especial

**Abstract**

Para esta ocasión, la cátedra nos solicitó implementar una variante del juego Sokoban. Dicho juego debió ser escrito en el lenguaje Java, y la interfaz gráfica se logró con el uso de la herramienta Swing.

**Diseño y jerarquía de clases**

Lo primero de lo que se ocupó nuestro equipo fue de realizar un diseño robusto y lo más escalable posible para la aplicación. Esta tarea se llevó a cabo durante los primero tres días luego de la entrega del enunciado.

Cuando todos logramos estar de acuerdo con el diseño final, se procedió a trasladar todas las ideas a código de Java. Nuestro diseño cuenta con varias ramas jerárquicas: los padres principales de dichas jerarquías son las clases “Cell” y “Piece”. En principio decidimos que no todo sea una celda en el tablero, y que haya piezas sobre celdas, para que sea más escalable.

Pensamos que Cell debía una clase abstracta, de la que heredarían “Floor” y “Wall”: lo pensamos así porque todas las celdas conocen su posición en el tablero, conocen al tablero. Además, todas las celdas responden a los mismos mensajes, de manera distinta: entre ellos están canReceive(Piece p), react(Piece p) o hasPiece() por ejemplo. Floor es la clase que representa a una celda vacía: esta clase es capaz de almacenar una pieza, y en caso de estar vacía, guarda una instancia de “Empty”, que es una clase que modela el comportamiento del vacío. De Floor extienden “Hole” y “Destination”, que representan al agujero y a la celda destino respectivamente. Ambos heredan de piso porque pueden almacenar piezas igual que el piso, pero se comportan de manera distinta cuando una pieza está sobre ellos. Luego Wall es la clase que modela las paredes, e impide que cualquier pieza avance sobre ella.

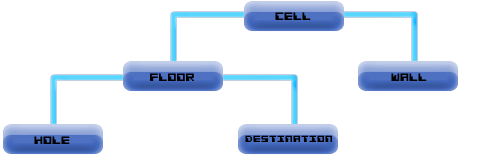
Por otro lado está Piece, que modela una pieza del tablero. Al igual que Cell, es una clase abstracta: hay ciertos comportamientos que toda pieza tiene, pero que no son suficientes como para poder convertir a Piece en una clase concreta. Por ejemplo, toda pieza conoce la celda sobre la que está posicionada. Además, como las piezas pueden moverse por el tablero, podrían caer por un agujero, por lo que deberían saber responder a los mensajes move(Direction d) y fall();

De Piece extienden directamente “Hero”, que modela al jugador, “Box”, que modela a la caja, y “Empty”, que modela al vacío. Luego, de Box hereda “BombBox”, que representa a la caja bomba: esta herencia fue hecha pensando que la caja bomba en particular es una caja, y tiene las mismas propiedades que una caja convencional (un color por ejemplo), pero además explota luego de cierta cantidad de movimientos.

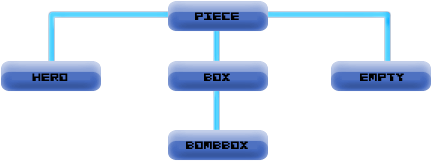
Además contamos con algunas interfaces: “fallable()” es una de ellas. Luego de haber entregado el trabajo práctico, el grupo se dio cuenta de que no todas las piezas implementan fallable(), aunque todas responden al mensaje fall(). Esto debería haberse corregido, haciendo que Piece implements fallable(); pero la idea principal era obligar a que todas las piezas, por ser movibles, implementen dicha interfaz.

Tuvimos que modelar otros objetos que no pertenecen directamente a ninguna de las jerarquías antes mencionadas. Por ejemplo, la clase “Board”: esta clase representa a un tablero. Fue pensada como una matriz de objetos de tipo Cell, ya que todo en el tablero es una celda, con o sin pieza. Dicho tablero cuenta con métodos de acceso a celdas y de colocación de celdas: este último método, llamado put() valida que se pueda colocar dicha pieza en el mismo.

A continuación, una ilustración de los árboles de jerarquías:



**Imagen1:** Jerarquía de la clase Cell.

  
**Imagen2:** Jerarquía de la clase Piece.

**Problemas y soluciones**

Durante el desarrollo de la aplicación, varias opiniones fueron tenidas en cuenta a la hora de determinar cuál sería el diseño más correcto mientras que otras fueron descartadas.

* Back end

Durante el juego, es necesario revisar (ante situaciones determinantes) si el jugador ha ganado o ha perdido la partida. Como idea inmediata, se nos ocurrió que, luego de cada movimiento realizado por el jugador, se le pregunte al tablero por su estado, y en base a ello, determinar si la partida ha concluido o no. Sin embargo, esta solución resulta algo imperativa y repetitiva, ya que solo en algunas ocasiones sería correcto preguntar por el estado del tablero: en primera instancia, los únicos elementos en el juego que son determinantes son el jugador, y las cajas. Si un jugador cae por un agujero, la partida concluye; si todas las cajas están colocadas en sus destinos, y no sobran cajas, la partida concluye. Por lo tanto, solo en dichos casos sería correcto consultar si la partida ha terminado. Lo que se nos ocurrió, es que cada tipo de celda del tablero, es decir, un piso, un agujero o un destino, sepa reaccionar ante el tipo de pieza sobre la que tiene encima de ella. De esta manera, si una caja está posicionada sobre un agujero, se le pedirá al agujero que reaccione ante lo que tiene encima; análogamente ocurre con los destinos y los pisos. Entonces, cada vez que una pieza reaccione de manera significativa, se preguntará por el estado de la partida, determinando si es que ha concluido o no.

Como se mencionó antes, el juego se gana únicamente si todas las cajas están puestas sobre sus correspondientes destinos, y no sobra ninguna caja. Para no tener que recorrer el tablero entero cada vez que se quiera averiguar si todas las cajas están colocadas correctamente, y que el número de cajas es igual al número de destinos, lo que se nos ocurrió es que nuestra clase “Match”, que modela una partida, conozca un conjunto de cajas: dicho conjunto contiene una referencia a cada caja en el tablero. Esta solución nos pareció la más ágil y rápida para consultar el estado de cada caja.

De todas formas, a pesar de todas estas soluciones que nos salvan de preguntar con bloques “if”, en algún lado del código había que dejar registro de si la partida se había ganado o no, e ir preguntando por dicho estado. Para deshacernos de esa pregunta, lo que hicimos fue escribir una interfaz en el back end: dicha interfaz contiene únicamente dos métodos, onWin() y onLose(). Esta interfaz está implementada en el front end, de manera que cuando se le manda un onWin() o un onLose(), al usuario le aparece un mensaje avisándole que la partida ha terminado. La clase que modela a la partida conoce un objeto del tipo de la interfaz, por lo que, cuando el jugador cae por un agujero, o todas las cajas están correctamente puestas, la partida le manda uno de los dos mensajes a dicho objeto, y el juego concluye, evitando tener que preguntar por el estado de la partida.

Otra idea que tuvimos fue la de modelar el objeto vacío, o como nosotros lo llamamos, “Empty”. Cuando una celda en el tablero no contiene una pieza, en realidad lo que contiene es un objeto de tipo Empty: este objeto fue modelado para presentar cierto comportamiento, y de esta manera, evitar tener que preguntar por “null” por ejemplo, o por algún tipo en particular. Donde Empty sabe responder a las mismas preguntas que el resto de las subclases de pieza.   
  
Un problema que nos surgió al avanzar con el proyecto fue que en ciertos casos cuando queríamos recuperar una pieza había en una posición del tablero había que castear explícitamente a Floor ya que las celdas no poseen piezas. Debido a esto para ahorrarnos el casteo modelamos una interfaz "CellVisitor" la cual es implementada por una clase "CellPieceVisitor". De esta forma cada subclase de celda tenia implementado un método "visit" el cual recibía una instancia de CellPieceVisitor. El objetivo era que en cada implementación de visit se intercambiarán receptor y parámetro para que ahora como la instancia de CellPieceVisitor es el receptor pueda ya saber de antemano de que tipo es el parámetro que está recibiendo y así guardar en una variable de instancia la pieza y luego recuperarla con otro método de la clase CellPieceVisitor.  
Paralelamente a esta caso otro en el cual utilizamos esta modalidad de invertir receptor con parámetro es a la hora de cargar las imágenes en el tablero. Aquí como cada celda y pieza se dibujaba de una forma distinta utilizamos la interfaz ImgSetter implementada por Drawer setter logramos que cada pieza/celda pudiera dibujarse en su forma correcta sin tener que estar preguntando el tipo de cada objeto.

Para Finalizar Utilizamos una interfaz visual distinta, basada en el juego Portal creado por Valve ([http://www.valvesoftware.com](http://www.valvesoftware.com/)) ya que encontramos un cierto atractivo en la forma que se presentan las imágenes. Con respecto al diseño elegido estamos muy conformes y creemos que fue una buena elección ya que no encontramos problemas de gran importancia que hicieran necesaria una modificación de jerarquía.