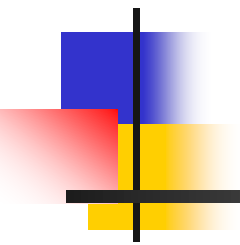


Explicaciones para la Práctica 3

La clase **Command**

- Para incluir excepciones, la cabecera de los métodos abstractos de **Command** ahora son:
  - `public abstract Command parse(String[] words, Scanner sc) throws CommandParseException;`
  - `public abstract boolean execute(Game g) throws CommandExecuteException, GameOverException;`
- Ejemplos de situaciones en las que se lanzan estas excepciones (se detallan después) son:
  - **CommandParseException:** `> move vaca`
  - **CommandExecuteException:** `> undo` **y resulta que la pila de estados estaba vacía**
  - **GameOverException:** `> move up` **y resulta que, después de esta jugada, el juego se ha acabado**

La clase **Command**

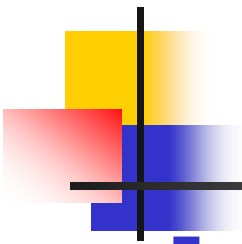
- El booleano de **`execute`** se puede usar para decirle a **`Controller`** cuándo mostrar o no el tablero, de manera que este método ya no necesita un parámetro de tipo **`Controller`**
- La cabecera de **`parseCommand`** de la clase **`CommandParser`** pasa entonces a ser:
  - `public static Command parseCommand(String[] words, Scanner sc) throws CommandParseException`

La clase **NoParamsCommand** y algunas de sus herederas

- Las cabeceras de **parse** y **execute** de las clases **NoParamsCommand**, **HelpCommand**, **ExitCommand**, y **ResetCommand** no cambian.

La clase **NoParamsCommand** y algunas de sus herederas

- **parse** de las clases **UndoCommand** y **RedoCommand** no cambia (es el de **NoParamsCommand**)
- **execute** de las clases **UndoCommand** y **RedoCommand** capturan una **EmptyStackException** y la relanzan como **CommandExecuteException**.

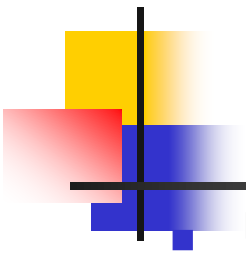


La clase **MoveCommand**

- **`execute`** lanza una **`GameOverException`**
- **`parse`** lanza una **`CommandParseException`** en estos casos:
  - `> move` //falta la dirección
  - `> move vaca` //vaca no es una dirección
  - `> move up vaca` //move no necesita tres o más items

La clase `PlayCommand`

- **`execute`** no lanza excepción porque suponemos que la petición de datos del nuevo juego se hace en **`parse`**
- **`parse`** lanza **`CommandParseException`** en las siguientes situaciones:
  - `> play` porque falta el tipo de juego
  - `> play vaca` porque `vaca` no es un tipo de juego
  - una vez parseado `play` con un tipo de juego correcto
    - se piden los datos **`size`**, **`initCells`** y **`seed`** del juego (recuerda que si no se proporcionan, se toman por defecto)
    - La petición lanza y captura **`NumberFormatException`** si **`size`**, **`initCells`** o **`seed`** no son número
    - **`parse`** lanza **`CommandParseException`** si **`initCells>size*size`**



La clase `PlayCommand`

- Una vez parseado correctamente un comando de la forma `play variante_de_juego`, los datos de ese juego se solicitan en **`parse()`**
- Cuando el usuario no da el tamaño (es decir, presiona Intro), se toma 4 como valor por defecto
- Cuando el usuario no da el número de celdas iniciales (es decir, presiona Intro), se toma 2 como valor por defecto
- Cuando el usuario no proporciona la semilla (es decir, presiona Intro), se toma **`new Random().nextInt(1000)`** como valor por defecto
- Para guardar estos datos, esta clase tiene cuatro atributos: **`gameType`**, **`size`**, **`initCells`**, **`seed`**

La clase `PlayCommand`

- La petición de un dato cualquiera **D**, de estos tres que se mencionan en la transparencia anterior, sigue esta estructura:

```
datoCorrecto=false;
while (!datoCorrecto) {
     words=scanner.nextLine().trim().split(“ +”);
     if (words.length==1)
          if (words[0] es “”) //El usuario presiona Intro
                {D=valor por defecto; datoCorrecto=true;}
           else {
                try {
                     D=Integer.parseInt(words[0]);
                     if (D cumple ...) datoCorrecto=true;
                     else “Mensaje al usuario para que rectifique”
                }
                catch (NumberFormatException ...) {...}
           }
     else “Mensaje al usuario para que introduzca un número positivo”
}
```

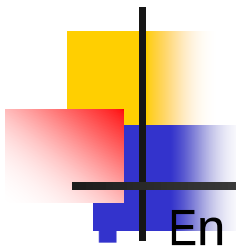
La clase **LoadCommand**

- **`execute(Game g)`** lanza **`CommandExecuteException`** si:
  - La primera línea del archivo no es **“`This file stores a saved 2048 game`”**
  - Captura excepciones **`IOException`**, **`NumberFormatException`**, **`FileContentsException`**, **`IndexOutOfBoundsException`**, … y las relanza como excepciones **`CommandExecuteException`**; estas excepciones se han podido producir en **`g.load()`** o en algún otro lugar de la carga de los datos del archivo
- **`parse`** lanza **`CommandParseException`** en las siguientes situaciones:
  - **`> load`** porque falta el nombre del archivo
  - **`> load fileName`** si el nombre del archivo no existe
  - **`> load fileName`** si **`validFileName(fileName)`** de **`MyStringUtils`** (mira el apéndice) devuelve **`false`**

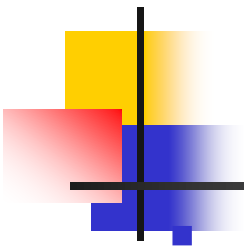
La clase **SaveCommand**

- **`execute(Game g)`** lanza **`CommandExecuteException`** si:
  - se ha lanzado una **`IOException`** al guardar la partida con **`g.store(bw)`**, capturándola y relanzándola como **`CommandExecuteException`**
- **`parse`** lanza **`CommandParseException`** en las siguientes situaciones:
  - `> save` porque falta el nombre del archivo para guardar la partida
  - `> save fileName vaca` porque el nombre del archivo no puede contener espacios

La clase **SaveCommand**



- En el parseo del nombre del archivo se usan dos métodos de **SaveCommand:**
  - **confirmFileNameStringForWrite(fileName)** (mira el apéndice) que actúa así: si **fileName** no es válido, se lanza una **CommandParseException**. Si lo es y no existe tal archivo, el atributo **filename_confirmed** se hace cierto y el parseo termina con éxito. Si lo es y existe, se pasa el control al método siguiente
  - **getLoadName(fileName**) (mira el apéndice). Este método contiene un bucle donde se le pide al usuario si desea sobrescribir el archivo **fileName**. Si contesta con **'y'**, el bucle termina, el atributo **filename_confirmed** se hace cierto y el parseo termina con éxito. Si contesta con **'n'**, se le pide un nombre y se regresa al bucle del método anterior para analizarlo. Si contesta con otra letra o con más de una palabra, lanza y captura una **YesOrNoException** y sigue en el bucle de este método

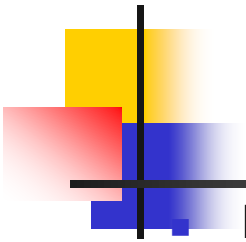


Las clases **Board** y **Game**

- En la clase **Board**:
  - El método **store()** lanza **IOException** si hay problemas al escribir en el archivo
  - El método **load()** lanza **IOException** si hay errores en la lectura del archivo
- En la clase **Game**:
  - El método **move()** lanza **GameOverException**
  - El método **load()** lanza **IOException** y **FileContentsException**. Esta última se lanza cuando los datos que se cargan no son consistentes (por ejemplo, **iniCells>size*size**)
  - El método **store()** lanza **IOException** si hay problemas al guardar una partida

La clase **Game**

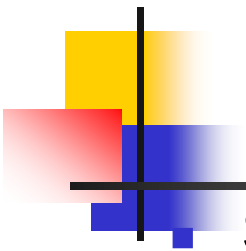
- Tiene los atributos de la Práctica 2, pero cambia el atributo **`GameRules currentRules`** por **`GameType currentGameType`**, tal como se dice en el enunciado de la práctica
- El método **`move()`** lanza **`gameOverException`** en los casos en que se gana o se pierde
- El método **`load(BufferedReader entrada)`** tiene esta estructura:
  - Guarda datos del juego actual para usarlos por defecto en caso de que la lectura de entrada falle
  - Llama a **`board.load(entrada)`**
  - Lee la línea en blanco que separa la matriz numérica de la línea final
  - Lee la última línea que contiene **`initCells points gameType`**
  - Lanza excepción de tipo **`FileContentsException`** si no hay **`gameType`**, o si **`initCells>size*size`**, por ejemplo
  - Estas excepciones u otras de lectura se capturan y se pasa entonces a cargar el juego por defecto



La clase **Board**

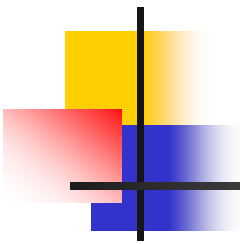
El método **`load(BufferedReader entrada)`** tiene esta estructura:

- Lee la primera línea de datos numéricos, que contiene la primera fila de la matriz
- Crea los tokens correspondientes y del número de tokens obtiene el tamaño del tablero
- Crea el tablero con ese tamaño y lo rellena de celdas, a partir de la fila leída y de la lectura del resto de las filas



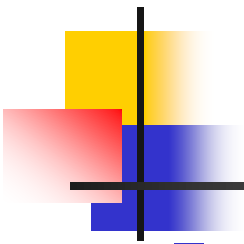
La clase **`GameStateStack`**

Solamente el método **`pop()`** (y también **`top()`**, si lo usas) lanza **`EmptyStackException`**



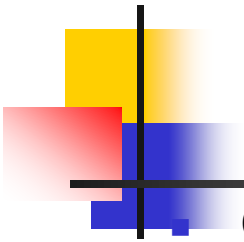
Excepciones del método main

- Trata las siguientes situaciones especiales:
  - **`args`** tiene menos de dos o más de tres argumentos: informa de que eso es erróneo
  - **`args[0]*args[0]<args[1]`**: informa de que eso es erróneo
  - **`args[i]`** no es un número: captura la excepción de tipo **`NumberFormatException`** que lanza el sistema



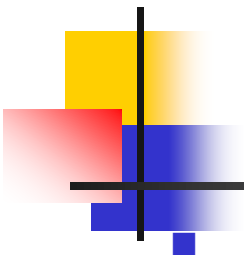
La clase **Controller**

- El método **run** de **Controller** lanza y captura excepciones de tipo **CommandParseException**, **CommandExecuteException** y **GameOverException** tal como se ha explicado en las clases ...**Command**
- Recuerda que `> vaca` se parsea en **null** y puede o no generarse una **CommandParseException**



Nuevas clases de excepciones

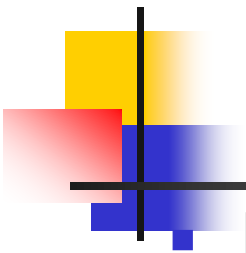
- **`GameOverException`**: excepciones de esta clase las lanza el método **`execute`** de **`MoveCommand`**, si el juego acaba después de ejecutar el movimiento
- **`CommandParseException`**: lanzadas por **`parse`** de **`MoveCommand`**, **`PlayCommand`**, **`SaveCommand`** y **`LoadCommand`** cuando el parseo encuentra errores
- **`CommandExecuteException`**: lanzadas por **`execute`** de **`Undo/RedoCommand`**, **`Load/SaveCommand`** y **`MoveCommand`**
- **`EmptyStackException`**: lanzadas por el método **`pop`** de **`GameStateStack`** cuando intenta desapilar de alguna de las pilas de estados (**`ues`** o **`res`**) y la pila está vacía
- **`YesOrNoException`**: lanzadas y capturadas por **`getLoadName(fileName`**) cuando se le pregunta al usuario si se sobreescribe **`fileName`**
- **`FileContentsException:`** lanzadas y capturadas por **`load`** de **`Game,`** cuando se intentan cargas datos no consistentes



La interfaz **`GameRules`**

- Recuerda que averiguar si el juego se ha acabado se podía hacer, en la Práctica 2, de dos formas:
  - Viendo que el tablero está lleno y ejecutando los cuatro posibles movimientos sobre un tablero auxiliar, comprobando entonces si ha habido algún cambio o no
  - O, mejor hecho, añadiendo un nuevo default method, implementado igual en **`2048`** e **`Inverse`**, que dice cuándo dos celdas son fusionables, y dejando que **`Fib`** lo sobrescriba

La interfaz `GameRules`



- En la Práctica 3 **debes añadir** un nuevo default method a esta interfaz, llamado **`canMergeNeighbours(cell1, cell2)`**
- Tal como se pide en la sección 4 **`Refactoring`**, del enunciado de esta práctica, este método averigua si las celdas de los parámetros se pueden fusionar
- La implementación por defecto que se usa es la que tienen los juegos **`2048`** e **`Inverse`**, es decir, las celdas son fusionables si tienen el mismo valor
- El juego **`Fib`** lo sobrescribe en la clase **`RulesFib`** según exige este juego, es decir, las celdas son fusionables si el valor de una celda es el siguiente, en la sucesión de Fibonacci, al valor de la otra, o las dos tienen valor 1