- Práctica 3: Excepciones y ficheros
- Introducción
- Manejo de excepciones
  - Excepciones en los comandos y controlador
  - Excepciones en `GameModel`
- Carga de la *configuración de juego* de un fichero
  - Formato del fichero de carga
  - Factoría de objetos del juego
  - *Configuración del juego*: GameConfiguration
  - Clase encargada de leer la configuración del fichero: FileGameConfiguration
  - Modificación de la clase Game
  - Comando de carga: LoadCommand
  - Errores durante la carga del fichero
  - Ajustando el método `reset` de Game
  - Guardado en el fichero del juego actual (opcional)
  - Sacando las configuraciones iniciales de la clase Game (opcional) # Práctica 3: Excepciones y ficheros

Entrega: semana del 2 de diciembre

Objetivos: Manejo de excepciones y tratamiento de ficheros

Preguntas Frecuentes: Como es habitual que tengáis dudas (es normal) las iremos recopilando en este documento de preguntas frecuentes. Para saber los últimos cambios que se han introducido puedes consultar la historia del documento.

Introducción

En esta práctica se ampliará la funcionalidad del juego en dos aspectos:

- Incluiremos la definición y el tratamiento de excepciones. Durante la ejecución del juego pueden presentarse estados excepcionales que deben ser tratados de forma particular. Por ahora, muchos de estos estados excepcionales los hemos tratados con la devolución del valor `null` pero seguro que ya has sufrido muchos errores del tipo `NullPointerException` y entiendes el motivo por el que no es adecuado el manejo de objetos nulos en Java. Además, al lanzar las excepciones allí donde se producen, los mensajes de error pueden ser más descriptivos y proporcionar al usuario información relevante de por qué se ha llegado a ellos (por ejemplo, errores producidos al procesar un determinado comando). El objetivo último es dotar al programa de mayor robustez, así como mejorar la interoperabilidad con el usuario.

- Cargaremos de ficheros las configuraciones iniciales del tablero. De esta forma las configuraciones iniciales no serán parte del código del programa ni estarán limitadas a unas pocas configuraciones fijas.

Manejo de excepciones

El tratamiento de excepciones en un lenguaje como Java resulta muy útil para controlar determinadas situaciones que se producen durante la ejecución del juego.

En esta sección se enumerarán las excepciones que deben tratarse durante el juego, se explicará la forma de implementarlas y se mostrarán ejemplos de ejecución. Hemos dividido la sección en dos grandes bloques, uno para las excepciones que se producen en los comandos y controlador, y otro para las excepciones que se producen en el modelo. Las excepciones relativas a los ficheros serán explicadas en la sección siguiente.

Para simplificar se recomienda incluir todas las excepciones en el paquete `tp1.exceptions`.

Excepciones en los comandos y controlador

Una de las principales modificaciones que realizaremos a la hora de incluir el manejo de excepciones en el juego consistirá en ampliar la comunicación entre los comandos y el controlador. Para ello cambiaremos algunos métodos para que en una situación de error, en vez de devolver `null`, lancen una excepción.

Básicamente, los cambios que se deben realizar son los siguientes:

- Se debe definir una excepción general `CommandException` y como subclases dos excepciones concretas:
  - `CommandParseException`: excepción para errores que tienen lugar al "parsear" un comando, es decir, aquellos producidos durante la ejecución del método `parse()`, tales como comando desconocido, número de parámetros incorrecto o tipo de parámetros no válido.
  - `CommandExecuteException`: excepción para representar situaciones de error que se pueden dar al ejecutar el método `execute()` de un comando; por ejemplo, solicitar cambiar el rol a un lemming que no existe.
- La cabecera del método `parse(String[] commandWords)` de la clase `Command` pasa a poder lanzar excepciones de tipo `CommandParseException`:

  ```
  public abstract Command parse(String[] parameter) throws CommandParseException;
  ```

  Por ejemplo, el método `parse()` de la clase `NoParamsCommand` que se proporcionó en la práctica anterior pasa a ser de la siguiente forma:

  ```
  public Command parse(String[] commandWords) throws CommandParseException {
      if (commandWords.length < 1 || !matchCommandName(commandWords[0]))
          return null;

      if (commandWords.length == 1 && matchCommandName(commandWords[0]))
          return this;
  ```

```
        throw new CommandParseException(Messages.COMMAND_INCORRECT_PARAMETER_NUMBER);
    }
```

Fíjate en que los comandos lanzan excepciones solo cuando la entrada concuerda con el nombre (y por lo tanto deberían ser capaces de parsear) pero existe algún fallo en los argumentos pasados a dicho comando (y se da un error). Es decir, la situación en la que el comando simplemente no puede parsear la entrada porque no concuerda con el nombre no se trata de un error (en cuyo caso no se lanza la excepción sino que se devuelve `null`).

- La cabecera del método estático `parse(String[] commandWords)` de `CommandGenerator` pasa a poder lanzar excepciones de tipo `CommandParseException`:

  ```
  public static Command parse(String[] commandWords) throws CommandParseException
  ```

  de forma que el método lanza una excepción del tipo

  ```
  throw new CommandParseException(Messages.UNKNOWN_COMMAND.formatted(commandWords[0]));
  ```

  si se encuentra con un comando desconocido, en lugar de devolver `null` y esperar a que `Controller` trate el caso mediante un simple *if-then-else*.

- La excepcion `NumberFormatException` (que se lanza cuando se produce un error al convertir un `String` a un `int`) se capturará para lanzar en su lugar una `CommandParseException`.

  Una buena práctica en el tratamiento de excepciones consiste en recoger una excepción de bajo nivel para, a continuación, lanzar una de alto nivel que contiene otro mensaje que, aunque será necesariamente menos específico que el de la de bajo nivel, es también de utilidad. Por ejemplo, en el método `parse` del comando `SetRoleCommand` podemos hacer lo siguiente:

  ```
  } catch (NumberFormatException e) {
      throw new CommandParseException(Messages.INVALID_POSITION.formatted
          (Messages.POSITION.formatted(row, col));
  }
  ```

- La cabecera del método abstracto `execute()` también se modifica indicando que puede lanzar excepciones de tipo `CommandExecuteException`:

  ```
  public abstract void execute(GameModel game, GameView view) throws CommandExecuteExcept
  ```

- El controlador debe poder capturar, dentro del método `run()`, las excepciones lanzadas por los dos métodos anteriores

  ```
  public void run() {
      ...
      while (!game.isFinished()) {
          ...
          try { ... }
  ```

```
            catch (CommandException e) {
                view.showError(e.getMessage());
                Throwable cause = e.getCause();
                if (cause != null)
                    view.showError(cause.getMessage());
            }
        }
        ...
    }
```

Así, todos los mensajes de error se pasan a la vista desde el bucle del método `run()` del controlador.

Uno de los comandos que más excepciones lanzará será el comando `setRole`, que explicaremos en la sección siguiente, pues la gran mayoría de ellas surgen en el modelo. En la **etapa de análisis** (*parseo*), este comando deberá analizar el rol y convertir las filas y columnas a enteros. En todos esos procesos se deberán capturar las excepciones que se produzcan para generar una excepción del tipo `CommandParseException`. En la **ejecución** del comando puede ocurrir que la posición solicitada se encuentre fuera del tablero o que no haya ningún objeto en ella al que aplicar el rol. En la práctica anterior estos dos errores eran indistinguibles pero en esta podremos distinguirlos, lanzando una excepción del tipo `CommandExecuteException` con un mensaje de error adecuado.

Pasamos a explicar cómo se deben tratar las excepciones en el modelo.

Excepciones en `GameModel`

Como hemos comentado anteriormente, los errores al ejecutar los comandos surgen de la lógica del juego. En la práctica anterior los métodos invocados por los comandos susceptibles de generar errores devolvían un `boolean` para indicar si su invocación había tenido éxito. Por ejemplo, el método `setRole(Position pos, LemmingRole role)` de `GameModel` devolvía `false` cuando la posición se encontraba fuera del tablero o cuando se intentaba establecer el role en una posición en la que no había ningún lemming al que aplicarlo. Sin embargo, al agrupar ámbos comporatamientos el mensaje de error que podíamos generar se limitaba a informar de que el rol no había podido aplicarse en esa posición, sin concretarse si la posición se encontraba fuera del tablero o no había ningún objeto en dicha posición al que se pudiera aplicar.

En general, los mensajes de error son más descriptivos si se crean allá donde se ha producido el error. Por ello, vamos a considerar que hay un error cuando se solicita el cambio en una posición externa al tablero y mantendrémos el `booleano` para indicar que el game no ha sido modificiado por dicho cambio, al no tener ningún lemming al que aplicar dicho rol en dicha posición. Esto nos permitirá mostrar los siguientes mensajes de error:

- Mensaje al intentar establecer un rol en una posición sin lemmings:

  `[DEBUG] Executing: setRole Walker A 2`

```
[ERROR] Error: Command execute problem
[ERROR] Error: No lemming in position (3,2) admits role Walker
```

- Mensaje al intentar establecer el rol en una posición fuera del tablero:

```
[DEBUG] Executing: setRole Walker A 25

[ERROR] Error: Command execute problem
[ERROR] Error: Position (0,24) off the board
```

Por lo tanto, la cabecera del método `setRole` de `GameModel` quedará así:

```
public boolean setRole(LemmingRole role, Position pos) throws OffBoardException;
```

Además, consideraremos las siguientes excepciones lanzadas por los métodos de `GameModel`. Todas ellas heredarán de una clase `GameModelException`.

- `OffBoardException`: excepción que se produce cuando se intenta acceder de manera indebida a una posición fuera del tablero (por ejemplo, al tratar de colocar un rol a un lemming fuera del tablero).
- `GameParseException`: excepción que se produce al parsear un objeto del juego. Como subclases de esta clase se considerarán las siguientes excepciones:
  - `RoleParseException`: excepción lanzada por la clase `LemmingRoleFactory` al tratar de generar un rol con un string que no corresponda a ninguno de los que existen.
  - `ObjectParseException`: excepción que se produce al tratar de parsear un objeto en formato texto y no poder convertirlo al objeto correspondiente, pues no sigue el formato establecido. La veremos en la sección de los ficheros.

Volviendo a los errores de ejecución de los comandos, el método `execute` de cada comando invocará a un método de `GameModel` susceptible de lanzar las excepciones listadas más arriba. En cada caso deberemos construir la nueva excepción `CommandExecuteException` de manera que *recubra* o *decore* a la excepción capturada. Por ejemplo:

```
} catch (OffBoardException obe) {
    throw new CommandExecuteException(Messages.ERROR_COMMAND_EXECUTE, obe);
}
```

De esta forma la causa última del error (la excepción `obe`) no se pierde y, en particular, se puede recuperar el mensaje de la excepción recubierta en el controlador (fíjate en el `cause.getMessage()` que aparece en el método `run` del controlador).

Carga de la *configuración de juego* de un fichero

En este apartado nuestro objetivo será conseguir *cargar una configuración del juego* contenida en un fichero de texto. Para realizar esta tarea utilizaremos varias técnicas de programación aprendidas anteriormente.

El resultado final será el comando `load` del juego, cuya ayuda se mostrará de la siguiente forma:

```
[DEBUG] Executing: help

Available commands:
.........
   [l]oad <fileName>: load the game configuration from text file <fileName>
.........
```

Para conseguirlo, dividiremos la tarea en otras tareas más pequeñas.

- Empezaremos explicando el formato del fichero de texto que queremos cargar.
- Seguiremos creando una factoría de objetos del juego `GameObjectFactory`, pues es necesaria para poder cargar la representación textual de los objetos contenida en el fichero.
- Tras ello, nos plantearemos crear una clase encargada de la lectura de la configuración del fichero de texto. Llamaremos a esta clase `FileGameConfiguration`, que delegará parte de sus tareas en la factoría de objetos.
- A partir de aquí, explicaremos los cambios necesarios en `Game` y ya por último podremos crear el comando `LoadCommand`.

Al finalizar estas modificaciones será necesario comprobar los mensajes de errores que se producirán en la carga del fichero y el comportamiento adecuado del comando `reset` que indirectamente se verá afectado.

Se recomienda en una primera etapa realizar la carga de fichero considerando que el formato del fichero es correcto y posteriormente ajustar el código para que sea capaz de detectar errores en el fichero.

Formato del fichero de carga.

Para poder cargar una configuración de un `game` desde un fichero de texto es necesario establecer un formato sobre los datos contenidos en el fichero. En este caso explicaremos el formato a través de un ejemplo simple.

El siguiente ejemplo representa un fichero de `game` con un lemming, dos paredes y una puerta de salida:

```
0 1 0 0 2
(3,2) Lemming RIGHT 1 Walker
(4,2) Wall
(4,3) MetalWall
```

`(5,4) ExitDoor`

En la primera línea `0 1 0 0 2` se encuentra el estado del game. Como se puede observar el estado está formado por 5 números enteros, por orden de aparición el *número del ciclo* en el que se encuentra, el *número de lemmings* en el tablero, el *número de lemmings muertos*, el*número de lemmings que han salido con éxito* y el *número de lemmings que es necesario que salgan para ganar el juego*. Tras esta primera línea aparecen, en cualquier orden, los objetos del juego.

Cada línea del fichero es la descripción de un objeto del juego. En cada línea el primer dato que aparece es la posición del objeto y el resto de los datos identifican el tipo del objeto y su estado.

La primera línea del fichero

```
`(3,2) Lemming RIGHT 1 Walker`
```

es un lemming situado en la fila `3` y en la columna `2` (que se corresponden en la vista del usuario con la fila `D` y la columna `3`); el valor `RIGHT` indica que su dirección es la derecha, el `1` indica que ha caído una posición y por último aparece su rol, en este caso caminante (`Walker`). Tanto para `Walker` como para `Lemming`, `Wall`, `MetalWall` o `ExitDoor` se pueden utilizar sus variantes cortas `L`, `W`, `MW` o `ED` respectivamente. Como se puede observar el patrón es muy simple:

```
posicionDelObjeto tipoDeObjeto atributosDelObjeto
```

El lemming es el único objeto que tiene atributos. En este caso primero aparece su estado (dirección y altura de caída) y luego su rol.

Asumiremos que el número de lemmings en el tablero indicado en el estado inicial del fichero coincide con la cantidad de lemmings que hay en el fichero. De no ser así se cargaría un juego incoherente y el comportamiento probablemente también sería incoherente[1].

Factoría de objetos del juego

Como hemos visto anteriormente en cada línea del fichero se encuentra la representación textual del objeto, que debería coincidir con la representación textual que devuelve el `toString()` del objeto. Por lo tanto, un string con ese formato deberíamos poder generar un objeto del juego.

Para crear un objeto del juego a partir de un string crearemos, al igual que hicimos con los roles, una factoría de objetos del juego. Crea la clase `GameObjectFactory` y añade en la clase abstracta *GameObject* el método:

[1]Se podría prescindir en el estado del juego de ese valor, por ejemplo, con un contador estático de lemmings creados en la clase lemmings, y así a la vez que vamos incorporándolos al contenedor los podríamos contar, pero eso complicaría más la lectura del fichero, por lo que hemos decidido dejarlo así. No obstante, si quieres realizar dicho contador estático en la clase `Lemming` y detectar la incoherencia del fichero para poder avisar del error lo puedes realizar como parte opcional de la práctica. Al igual que esta es posible tener otras incoherencias en el fichero, tales como valores negativos en el estado del game, etc. que ignoraremos.

```
public GameObject parse(String line, GameWorld game) throws ObjectParserException, OffBoardE
```

Se puede observar que este método devuelve dos tipos de excepciones:

- `ObjectParserException`: excepción lanzada cuando no se puede analizar la línea porque su formato incorrecto, por ejemplo por no tener todos los datos necesarios, por tener más datos de los necesarios, por tener un nombre de objeto o de rol desconocido, por no poder convertir a enteros los datos numéricos, etc.
- `OffBoardException`: excepción lanzada cuando la posición se encuentra fuera del tablero.

Es posible que para realizar el análisis de la línea para cada *GameObject* necesites métodos auxiliares. Si te sirve de ayuda podrías considerar añadir en las clases adecuadas alguno de los métodos siguientes:

```
private static Position getPositionFrom(String line) throws ObjectParseException, OffBoardEx
private static String getObjectNameFrom(String line) throws ObjectParseException {...}
private static Direction getLemmingDirectionFrom(String line) throws ObjectParseException {
private static int getLemmingHeigthFrom(String line) throws ObjectParseException {...}
private static LemmingRole getLemmingRoleFrom(String line) throws ObjectParseException {...
```

Esta propuesta de métodos se encarga tanto de la parte sintáctica como de la comprobación de la corrección de los datos con respecto al modelo (`OffBoardException`). Si estas funciones no te ayudan impleméntalo como consideres más adecuado.

Fíjate en que, al igual que ocurría con la clase `CommandGenerator`, la factoría de objetos **nunca** devuelve el valor `null`: o bien tiene éxito al crear el objeto o bien lanza una excepción[2].

Pasaremos ahora a describir la clase encargada de leer la **configuración del juego** de un fichero de texto.

Configuración del juego: GameConfiguration Antes de empezar la tarea es conveniente que nos planteemos la siguiente cuestión:

`¿Qué consideramos que es una `configuración del juego'?`

Una posible respuesta es que una configuración consiste en unos cuantos valores enteros indicando el número de ciclo, número de lemmings en el tablero, etc. y un contenedor inicializado. Por lo que podríamos considerar que una *configuración del juego* es un objeto que es capaz de suministrar esos valores:

```
// game status
public int getCycle();
public int numLemmingsInBoard();
public int numLemmingsDead();
public int numLemingsExit();
```

[2]Seguimos la misma técnica que el método Integer.valueOf.

```
public int numLemmingToWin();
// game objects
public GameObjectContainer getGameObjects();
```

Tu primera tarea consiste en crear dicho interfaz.

Clase encargada de leer la configuración del fichero: FileGameConfiguration

A continuación, crearemos una clase encargada de leer la configuración del fichero, la clase `FileGameConfiguration` que se situará en el paquete `tp1.logic`. Esta clase tendrá un constructor con dos parámetros: el nombre del fichero y el `game`, para poder *enganchar* los objetos del fichero con el game en ejecución:

```
public FileGameConfiguration(String fileName, GameWorld game) throws GameLoadException;
```

Como puede observarse en su cabecera puede lanzar una única excepción: - `GameLoadException`: lanzada tanto en caso de que el fichero no exista `FileNotFoundException` (excepción estándar de Java), en el caso de que haya algún problema con la lectura o con el formato del fichero o en el caso de que alguna de las posiciones se encuentren fuera del tablero.

Modificación de la clase Game

Añadiremos al modelo la opción de cargar configuraciones desde un fichero. Para ello sólo nos hará falta crear en `Game` un método público para poder cargar el juego a través del fichero y añadirlo en el interfaz adecuado:

```
public void load(String fileName) throws GameLoadException {...}
```

Como se puede observar, este método puede lanzar la misma excepción que el constructor de la clase `FileGameConfiguration` y además en los mismos casos.

Comando de carga: LoadCommand Ya estamos en situación de poder crear el comando nuevo de carga `LoadCommand`. Su tarea consistirá en cargar una configuración de game desde un fichero.

La ayuda general tras la implementación de dicho comando será la siguiente:

```
Command > help
[DEBUG] Executing: help

Available commands:
   [s]et[R]ole ROLE ROW COL: sets the lemming in position (ROW,COL) to role ROLE
      [D]own [C]aver: Lemming caves downwards
      [P]arachuter: Lemming falls with a parachute
      [W]alker: Lemming that walks
   [n]one | "": user does not perform any action
   [r]eset [numLevel]: reset the game to initial configuration if not numLevel else load the
   [l]oad <fileName>: load the game configuration from text file <fileName>
   [h]elp: print this help message
   [e]xit: exits the game
```

```
Command >
```

Errores durante la carga del fichero

Durante la carga de la configuración del juego del fichero se pueden producir muchas excepciones ya que es necesario comprobar que tanto el estado del juego almacenado como cada descripción de objeto (i.e., cada línea) es correcta. Es posible que estos errores los tengas que analizar en el método `parse` de `GameObject` o de alguna de sus subclases.

- Tiene que existir el fichero:

  ```
  Command > load conf
  [DEBUG] Executing: load conf

  [ERROR] Error: Invalid file "conf" configuration
  [ERROR] Error: File not found: "conf"
  ```

- La línea de estado del game mantiene el formato adecuado.

  ```
  Command > load conf_2
  [DEBUG] Executing: load conf_2

  [ERROR] Error: Invalid file "conf_2" configuration
  [ERROR] Error: Incorrect game status "2 0 3 5"
  ```

- Los objetos que aparecen en el fichero son de tipo conocido:

  ```
  Command > load conf_3
  [DEBUG] Executing: r conf_3

  [ERROR] Error: Invalid file "conf_3" configuration
  [ERROR] Error: Unknown game object: "(3,2) Potato RIGHT 10 Walker"
  ```

- La posición de cada objeto dado está dentro del tablero:

  ```
  Command > load conf_4
  [DEBUG] Executing: load conf_4

  [ERROR] Error: Invalid file "conf_4" configuration
  [ERROR] Error: Object position is off board: "(3,18) Lemming RIGHT 10 Walker"
  ```

- Las direcciones de los lemmings son valores conocidos:

  ```
  Command > load conf_5
  [DEBUG] Executing: r conf_5

  [ERROR] Error: Invalid file "conf_5" configuration
  [ERROR] Error: Unknown object direction: "(3,2) Lemming NORTH 10 Walker"
  ```

- Las direcciones de los lemmings son sólo `RIGHT` y `LEFT`:

```
Command > load conf_6
[DEBUG] Executing: r conf_6

[ERROR] Error: Invalid file "conf_6" configuration
[ERROR] Error: Invalid lemming direction: "(3,2) Lemming UP 10 Walker"
```

- Los lemmings tienen un rol de tipo conocido:

```
Command > load conf_7
[DEBUG] Executing: r conf_7

[ERROR] Error: Invalid file "conf_7" configuration
[ERROR] Error: Invalid lemming role: "(3,2) Lemming RIGHT 10 Looser"
```

- Cada línea tiene el formato adecuado porque alguno de sus atributos no tiene el formato adecuado. Por ejemplo, los valores de las posiciones y la altura se deben poder convertir a números.

```
Command > load conf_8
[DEBUG] Executing: load conf_8

[ERROR] Error: Invalid file "conf_8" configuration
[ERROR] Error: Invalid object position: "(a,3) Lemming RIGHT 2 Walker"
```

Esas excepciones se capturarán directamente en el método `execute` de `LoadCommand`. Ten en cuenta que si la carga del fichero falla se gestiona esa excepción, informando al usuario del problema, y el juego debe poder continuar como si no se hubiese intentado la carga.

Ajustando el método `reset` de Game

Ahora que ya has completado la implementación, una de las cuestiones que te debes plantear es que el `reset` debe seguir funcionando correctamente. En particular, realizar un reset desde un juego cargado desde un fichero debería devolvernos a la configuración cargada desde este.

Para realizar esta tarea es posible que te ayude incluir en la clase `FileGameConfiguration` una constante `NONE`:

```java
public static final GameConfiguration NONE = new FileGameConfiguration();
```

Así, desde `Game` se puede usar la constante `FileGameConfiguration.NONE` como indicador de que se deber resetear el juego actual a su situación inicial. Por ejemplo, en `reset(...)` de `Game` se puede escribir:

```java
if (conf == FileGameConfiguration.NONE) {
    // inicialización estándar
} else {
    // inicialización usando conf
}
```

Revisa que funciona correctamente dicho `reset`, pues es posible que sin darte cuenta tengas algún error de encapsulamiento en la clase `FileGameConfiguration` que descubras ahora.

Comando `SaveCommand` (opcional)

Una parte opcional bastante simple consiste en guardar en un fichero el estado actual del juego. Recuerda que el método `toString()` debería devolver la representación textual del estado del game. Por lo tanto, si consigues que lo devuelva en el mismo formato que hemos estado usando para la carga desde un fichero la tarea consiste únicamente en añadir un nuevo comando `SaveCommand` y un nuevo método en `Game`:

```
public void save(String fileName) throws GameModelException {...}
```

Este método solo arrojaría excepciones en caso de que se produjera un error en la escritura del fichero. Esto podría ocurrir, por ejemplo, en casos donde el fichero estuviera bloqueado por el sistema operativo.

El resultado del comando help de la práctica con todas las extensiones será el siguiente:

```
Command > help
[DEBUG] Executing: help

Available commands:
   [s]et[R]ole ROLE ROW COL: sets the lemming in position (ROW,COL) to role ROLE
      [D]own [C]aver: Lemming caves downwards
      [P]arachuter: Lemming falls with a parachute
      [W]alker: Lemming that walks
   [n]one | "": user does not perform any action
   [r]eset [numLevel]: reset the game to initial configuration if not numLevel else load the
   [l]oad <fileName>: load the game configuration from text file <fileName>
   [s]ave <fileName>: save the actual configuration in text file <fileName>
   [h]elp: print this help message
   [e]xit: exits the game

Command >
```

Sacando las configuraciones iniciales de la clase Game (opcional)

Otra cuestión bastante simple consiste en sacar las configuraciones iniciales de la clase `Game` (las que se cargan usando `nLevel`), creando una clase que implemente `GameConfiguration` que se encargue de ellas. Esta clase sería similar a `FileGameConfiguration` y podría tener dos constructores: uno que reciba el *número del nivel*, que lanzaría una excepción en caso de que dicho nivel no sea válido, y otro sin el número de nivel, que cargaría el mapa por defecto.

Si te animas y tienes tiempo es bastante sencilla su implementación y te permitirá tener un código más limpio y ordenado.