Práctica 3: Road Fighter v 3.0

Fecha de entrega: 13 Diciembre 2021, 9:00

Objetivo: Manejo de excepciones y tratamiento de ficheros

1. Introducción

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

- Incluir la definición y el tratamiento de excepciones. Durante la ejecución del juego pueden presentarse estados excepcionales que deben ser tratados de forma particular. Además, cada uno de estos estados debe 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 de mejorar la interoperabilidad con el usuario.
- Gestionar ficheros para poder grabar y, si fuera el caso, cargar partidas en memoria. Estas tareas requerirían incluir los comandos necesarios para poder realizar tanto la escritura del estado de una partida en un fichero (save) como, llegado el caso, su carga de memoria. La grabación del juego se realizará pasando por un comando que genere el estado de la partida (serialize).
- Ampliaremos también la práctica con la funcionalidad de guardar y cargar el Record.

2. Manejo de excepciones

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.

2.1. Descripción

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

para mostrar información relevante al usuario sobre lo ocurrido durante el parseo o la ejecución de un comando. En las prácticas anteriores, cada comando invocaba su método correspondiente - generalmente de la clase Game - para poder llevar a cabo operaciones sobre el juego. Así, si se quería añadir una granada a la carretera, se debía comprobar si la casilla indicada por el usuario estaba libre y si el jugador disponía de suficientes Coins para añadirla. Si alguna de estas dos condiciones no se daba, la ejecución del comando devolvía false y se mostraba un mensaje de error. De esta forma, con un *booleano* el juego podía saber si la granada no se había llegado a añadir, pero a costa de incluir en los métodos mensajes de escritura que indicaban el motivo exacto de lo que había ocurrido.

En esta práctica vamos a contar con el manejo de excepciones para realizar esta tarea, y ciertos métodos van a poder lanzar y procesar determinadas excepciones para tratar determinadas situaciones durante el juego. En algunos casos, este tratamiento consistirá únicamente en proporcionar un mensaje al usuario, mientras que en otros el tratamiento será más complejo. En esta sección no vamos a ocuparnos de tratar las excepciones relativas a los ficheros, que serán explicadas en la sección siguiente.

En una primera aproximación vamos a tratar dos tipos de excepciones. Por una parte, se va a definir un nuevo tipo de excepciones llamado GameException que será una superclase que recoge dos nuevos tipos de excepciones: CommandParseException y CommandExecuteException. La primera de estas dos sirve para tratar los errores que ocurren al parsear un comando, es decir, aquellos producidos durante la ejecución del método parse(), tales como comando desconocido, número de parámetros incorrecto y tipo de parámetros no válido. La segunda se utiliza para tratar las situaciones de error que se pueden dar al ejecutar el método execute() de un comando; por ejemplo, no tener suficientes Coins para ejecutar un comando o que la casilla donde se quiere añadir un elemento del juego está ocupada. Por otra parte, nos ocuparemos de alguna excepción lanzada por el sistema, es decir, no creada ni lanzada por nosotros. En el juego esto ocurre con la excepción NumberFormatException, que ya se usó en la práctica anterior y que se lanza cuando se produce un error al tratar de transformar un número entero que se encuentra en formato String a su formato habitual int.

2.2. Aspectos generales de la implementación

Una de las principales modificaciones que realizaremos al incluir el manejo de excepciones en el juego consistirá en ampliar la comunicación entre los comandos y el controlador. Continuará existiendo la previa, a través del booleano de retorno del método execute() que indica si se ha actualizado el tablero y, por tanto, este debe ser mostrado. Pero también se contemplará la posibilidad de que se haya producido un error de forma que, controlando el flujo de las excepciones que puedan producirse durante el parseo o la ejecución de un comando, se podrá informar del error al usuario. Además, puesto que ahora se van a tratar las situaciones de error tanto en el procesamiento como en la ejecución de los comandos, los mensajes de error mostrados al usuario podrán ser mucho más descriptivos que en la práctica anterior.

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

1. La cabecera del método abstracto parse() de la clase Command pasa a poder lanzar excepciones de tipo CommandParseException:

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

2. La cabecera del método abstracto execute() de la clase Command pasa a poder lanzar excepciones de tipo CommandExecuteException:

   ```
   public abstract boolean execute(Game game) throws CommandExecuteException;
   ```

3. La cabecera del método estático getCommmand() de Command pasa a poder lanzar excepciones de tipo CommandParseException:

   public static Command getCommmand(String[] commandWords) throws CommandParseException;

   de forma que el método lanza una excepción del tipo

   throw new CommandParseException(String.format("[ERROR]: %s", UNKNOWN_COMMAND_MSG));

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

4. El controlador debe poder capturar, dentro del método run(), las excepciones lanzadas por los dos métodos anteriores, que son subclase de una nueva clase de excepciones GameException

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

5. Se deben definir nuevas clases (GameException, CommandParseException, CommandExecuteException y algunas más) y lanzar excepciones de estos tipos y tratarlas adecuadamente de forma que el controlador pueda comunicar al usuario los problemas que ocurran. Por ejemplo, el método parse() de la clase Command que se proporcionó en la práctica anterior pasa a ser de la siguiente forma:

```
protected Command parse(String[] words) throws CommandParseException {
    if (matchCommandName(words[0])) {
        if (words.length != 1) {
            throw new CommandParseException(String.format("[ERROR]:
                Command %s: %s", name,
                INCORRECT_NUMBER_OF_ARGS_MSG));
        } else {
            return this;
        }
    }
    return null;
}
```

Haciendo esto así, todos los mensajes se imprimen desde el bucle del método run() del controlador cuyo cuerpo se parecerá al código siguiente:

```
while (!game.isFinished()) {
        if (refreshDisplay) {
            printGame();
        }
        refreshDisplay = false;

        System.out.println(PROMPT);
```

```
            String s = scanner.nextLine();

            String[] parameters = s.toLowerCase().trim().split(" ");
            System.out.println("[DEBUG] Executing: " + s);
            try {
                Command command = Command.getCommand(parameters);
                refreshDisplay = command.execute(game);
            }
            catch (GameException ex) {
                System.out.format(ex.getMessage() + "%n%n");
            }
        }
```

2.3. Nuevos tipos de excepciones

En el desarrollo de esta práctica debes definir (en algún caso solo incluir y manejar), lanzar y capturar excepciones de, al menos, los siguientes tipos (en la sección siguiente aparecerá alguno más):

- GameException: es la superclase de las excepciones que se deben definir y de la que heredan las subclases CommandParseException y CommandExecuteException.
- CommandParseException: nuevo tipo de excepción lanzada por algún error detectado en el parseo de un comando.
    - NumberFormatException: excepción del sistema lanzada cuando un elemento proporcionado por el jugador debería ser un dato numérico y no lo es.
- CommandExecuteException: nuevo tipo de excepción lanzada por algún error detectado en la ejecución de un comando. Son subclases de esta las siguientes:
    - InvalidPositionException: nuevo tipo de excepción lanzada cuando una posición de la carretera proporcionada por el usuario está ocupada o no pertenece a la carretera.
    - NotEnoughCoinsException: nuevo tipo de excepción lanzada cuando no es posible realizar alguna acción pedida por el usuario al no tener el jugador suficiente saldo para llevarla a cabo.
    - InputOutputRecordException: nuevo tipo de excepción lanzada cuando hay problemas en la lectura o escritura del récord.

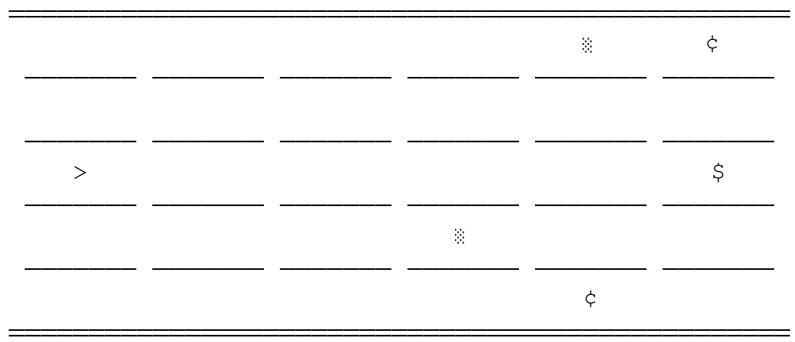
Una buena práctica en el tratamiento de excepciones consiste en recoger una excepción de bajo nivel para a continuación lanzar una de alto nivel que envuelve la anterior y que contiene otro mensaje que, aunque necesariamente menos específico que el de la de bajo nivel, es también de utilidad. Por ejemplo,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

```
catch (NotEnoughCoinsException e) {
    System.out.println(e.getMessage());
    throw new CommandExecuteException(String.format("[ERROR]: %s", FAILED_MSG)
        , e);
}
```

Es decir, se captura la excepción de más bajo nivel NotEnoughCoinsException, se muestra el mensaje de error y se relanza como excepción de más alto nivel CommandExecuteException.

Veamos un ejemplo:

```
[DEBUG] Executing: s
Distance: 100
Coins: 1
Cycle: 0
Total obstacles: 86
Total coins: 25
Supercoin is present
Ellapsed Time: 0.00 s
```



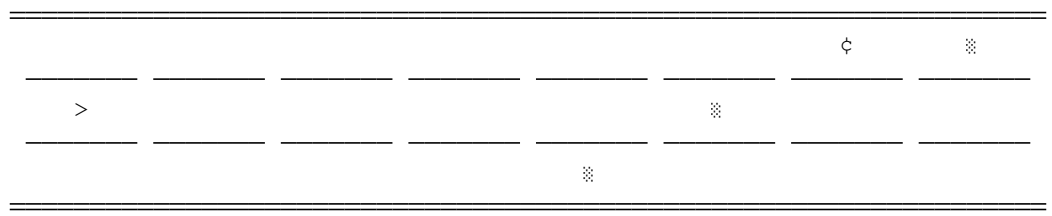
```
Command >
g 5 0
[DEBUG] Executing: g 5 0
Invalid position.
[ERROR]: Failed to add granade

Command >
g 1 0
[DEBUG] Executing: g 1 0
Not enough coins
[ERROR]: Failed to add granade
Command >
```

2.4. Serializar y guardar

En Informática, el término *serialization* (en español, serialización o secuenciación) es la conversión del estado de ejecución de un programa, o de parte de un programa, en un flujo de bytes, habitualmente con el objetivo de guardarlo en un fichero o transmitirlo por la red. El término contrario *deserialization* se refiere al proceso inverso de reconstruir el estado de ejecución de un programa, o de parte de un programa, a partir de un flujo de bytes. Aquí nos interesa producir un flujo de bytes que represente el estado actual del juego –no nos hace falta representar el estado de ejecución completo del programa– con el objetivo de escribir este estado en, y si se diera el caso leer este estado de, un fichero de texto. A menudo se utilizan los términos *stringification*/*destringification* para referirse a la serialización/deserialización cuando se trabaja con un flujo de texto.

SerializeCommand

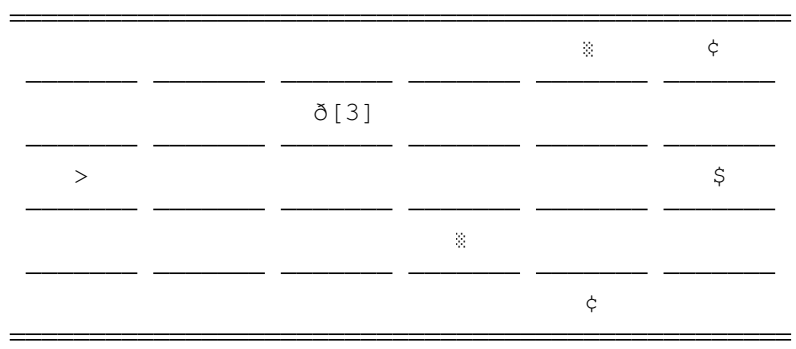
Primero vamos a crear un comand Serialize, usando la letra z, que muestra por consola la versión serializada del juego con el siguiente formato:



```
Command >
z
[DEBUG] Executing: z
---- ROAD FIGHTER SERIALIZED ----
Level: TEST
Cycles: 0
Coins: 5
EllapsedTime: 0
Game Objects:
> (0, 1)
▒ (4, 2)
▒ (5, 1)
¢ (6, 0)
▒ (7, 0)
▒ (9, 2)
```

Primero mostramos el estado general del juego y después la versión serializada de cada objeto de juego. Los objetos de juego se ordenan siguiendo orden arriba-abajo, izquierda-derecha.

En la versión serializada de los objetos mostramos su símbolo y su posición. En algunos objetos avanzados necesitaremos más información. Veamos otro ejemplo:



```
Command >
z
[DEBUG] Executing: z
---- ROAD FIGHTER SERIALIZED ----
Level: HARD
Cycles: 0
Coins: 2
EllapsedTime: 0
```

```
Game Objects:
> (0, 2)
ð[3] (2, 1) 3
▒ (3, 3)
▒ (4, 0)
¢ (4, 4)
¢ (5, 0)
$ (5, 2)
>>> (6, 4)
¢ (7, 4)
☺ (8, 0) down
...
▒ (16, 3) 1
```

Más concretamente:

- En la granada mostraremos el tiempo que falta para explotar.
- En el peatón, la dirección hacia la cual se está moviendo.
- En el muro, cuánta resistencia le queda.

Para hacer el comando serializer la idea es hacer una nueva *vista* que llamaremos GameSerializer y que será muy parecida a GamePrinter. De hecho aunque no es obligatorio podemos programar una clase abstracta View que encapsule la funcionalidad común entre las dos vistas.

Aunque no lo vamos a implementar, con esta información podríamos reconstruir el estado del juego programando un comando LoadGame.

SaveCommand

Para implementar este comando se debe definir la clase SerializeCommand. A partir de ella no es difícil definir la clase SaveCommand cuya ejecución es similar a la serialización, pero volcando el resultado en un archivo, al que se le añade un encabezamiento.

En Java existen muchos mecanismos para manejar ficheros. En esta práctica usaremos flujos de caracteres en lugar de flujos de bytes. En particular, recomendamos el uso de BufferedWriter y FileWriter para escribir en un fichero. Además, se recomienda el uso de bloques try-with-resources para el código donde se abre el fichero, capturando IOException en el método execute() de la clase SaveCommand.

Para poder guardar el estado de una partida, deberán tenerse en cuenta las siguientes consideraciones:

- El método parse() de la clase SaveCommand debe lanzar excepciones de tipo CommandParseException si no se proporciona argumento o se proporciona más de uno.
- El método execute() de la clase SaveCommand debe hacer lo siguiente:
    - La clase tiene un atributo String filename para guardar el nombre del archivo que se crea al ejecutar el comando. Para simplificar, vamos a suponer que el nombre proporcionado por el usuario es correcto en varios sentidos. Primero, porque no se va a analizar si los caracteres que componen el nombre conforman un nombre válido de archivo, algo dependiente del sistema operativo. Segundo, porque no se va a comprobar si el programa tiene permisos de escritura en el fichero. Y

tercero, porque aunque quisiéramos guardar el juego en un archivo que ya existe, el resultado de la ejecución del comando hará que ese archivo se sobrescriba.

- Una vez conocido el nombre, el atributo filename de la clase se completará con una extensión .txt. Por ejemplo, si escribimos el comando s datos, la ejecución del juego creará un archivo datos.txt.
- Se debe crear un FileWriter junto con un BufferedWriter que lo envuelva.
- Se debe utilizar el GameSerializer para generar un *String* con los datos del juego y después volcarlos en el archivo.

- En caso de haber podido guardar el estado del juego en un archivo – con éxito – se debe imprimir el mensaje:

  "Game successfully saved in file <nombre_proporcionado_por_el_usuario>.txt".

Lo ideal es que el modelo no sea consciente de que se ha hecho un Save, es decir, para este comando no debemos añadir nada en la clase Game.

DumpCommand

Para comprobar que se ha guardado correctamente la información en el fichero vamos a implementar un comando Dump, que básicamente vuelca el contenido de un fichero, línea a línea, en la consola.

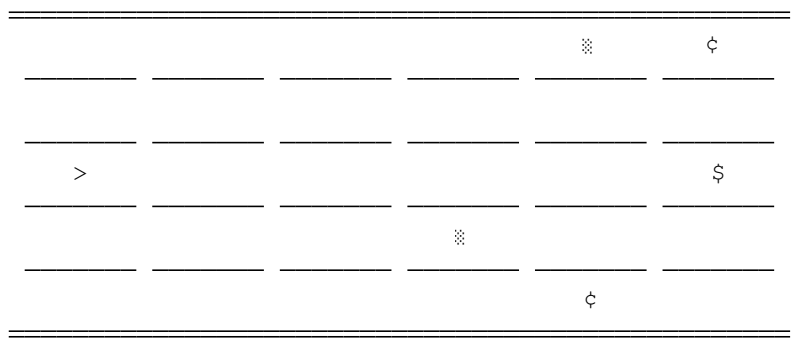
2.5. Guardar y cargar Record

La última extensión que vamos a hacer es la funcionalidad de Guardar y Cargar el record del juego. El record se cargará al principio de la partida o cuando hay reset. Veamos el formato del fichero record.txt:

```
HARD:22340
TEST:1760
EASY:6057
ADVANCED:1030
```

- Guardaremos un récord por cada nivel.
- Los records no deben almacenarse en ningún orden concreto.
- El récord se almacena en milisegundos.
- Si el fichero de records no existe o está corrupto se debe lanzar una excepción InputOutputRecordException y terminar el juego.
- Si el récord de un nivel no existe se debe crear un valor por defecto (el máximo valor de Long, por ejemplo).
- Lo ideal es crear una nueva clase Record para encapsular esta funcionalidad.

Para facilitarnos el testeo de esta funcionalidad vamos a crear un comando ShowRecordCommand, usando la letra *o*, que muestra el récord del nivel actual por pantalla (el récord se muestra formateado en segundos con dos decimales):



```
Command >
o
[DEBUG] Executing: o
HARD record is 22.34 s
```

2.6. Resumen

La lista de comandos disponibles en el help queda ahora de la siguiente manera:

```
Available commands:
[h]elp: show this help
[i]nfo: prints gameobject info
[n]one | []: update
[q]: go up
[a]: go down
[e]xit: exit game
[r]eset [<level> <seed>]: reset game
[t]est: enables test mode
[s]hoot: shoot bullet
[g]renade <x> <y>: add a grenade in position x, y
[w]ave: do wave
seriali[z]e: Serializes the board.
sa[v]e <filename>: Save the state of the game to a file.
[d]ump <filename>: Shows the content of a saved file
rec[o]rd: show level record
Clear [0]: Clears the road
Cheat [1..5]: Removes all elements of last visible column, and adds an Advanced Object
```

2.7. Casos de prueba

Los tests deben coincidir con la práctica anterior, salvo algún detalle de los mensajes de error. Se han añadido dos tests:

- 19_easy_s37.txt: Comprobar el comando save, serialize y dump.
- 20_easy_s37.txt: Prueba excepciones.