Tecnología de la Programación

Implementación de la Práctica 3

(Basado en la práctica de Simon J. Pickin)

Ana M. González de Miguel (ISIA, UCM)

Índice

1. Introducción

- ✓ La práctica 3 extiende y mejora la aplicación 2048 implementada con la práctica 2.
- ✓ Nuevos objetivos cubiertos en la asignatura: manejo de excepciones y entrada-salida.
- ✓ Modificaciones sobre la práctica 2
  - Introducir excepciones para hacer a la aplicación 2048 más robusta y mejorar su estructura.
  - Sumar dos nuevos comandos a la aplicación 2048 para:
    - Salvar el estado del juego en un fichero.
    - Cargar desde un fichero el estado del juego previamente salvado.
  - Utilizar el patrón *command* para añadir los dos nuevos comandos.
  - Manejar las excepciones necesarias con entrada-salida.

2. Sumando Manejo de Excepciones

- ✓ Hacer refactoring de la aplicación 2048 desarrollada en la práctica 2.
- ✓ Paquete principal de clases:

```
package tp.pr3;
```

- ✓ La nueva aplicación 2048 debe definir, lanzar y capturar todas las excepciones que sean necesarias así como aquellas que se consideren apropiadas. A continuación se detallan **cinco pasos** para conseguir esta funcionalidad.
- ✓ **Primero**, cazar cualquier excepción que el sistema pueda lanzar al leer de *input* y realizar una acción apropiada. No hay que declarar explícitamente el lanzamiento de estas primeras excepciones. Por ejemplo, *Game2048* debería poder cazar excepciones *NumberFormatException*. Y hay más entradas…

- ✓ **Segundo**, declarar explícitamente el lanzamiento de las siguientes excepciones
  - errores que puedan ocurrir en los métodos de la clase utilizada para almacenar los estados del juego para los comandos *undo* y *redo*,
  - errores que puedan ocurrir en los métodos de la clase *ArrayAsList*.
- ✓ Los métodos de la clase *GameState* no necesitan lanzar ninguna excepción. No así, el método *pop()* de la clase *GameStateStack* debería poder lanzar *EmptyStackException*.
- ✓ Para lanzar excepciones desde los métodos de la clase *ArrayAsList* tener en cuenta la funcionalidad equivalente implementada en la clase *ArrayList*.
- ✓ Tratar dichas excepciones de la clase *ArrayAsList* como errores fatales que pueden suceder durante la ejecución de la aplicación.

- ✓ **Tercero**, crear las siguientes excepciones de programador
  - errores que puedan suceder en el *parsing* de cualquiera de los comandos,
  - errores que puedan suceder en la ejecución de cualquiera de los comandos.
- ✓ Crear un directorio para estas excepciones:

```
package tp.pr3.exceptions;
```

- ✓ Crear una constructora con un parámetro *message* para el mensaje de error que se quiere mostrar con cada una de estas excepciones.
- ✓ Declarar explícitamente el lanzamiento de estas excepciones, capturarlas y realizar acciones apropiadas (al menos mostrar mensajes) cuando se capturen.

- ✓ Las excepciones de ejecución de los comandos pueden ser re-lanzadas considerando como excepciones originales las implementadas en el segundo paso (con declaración explícita de lanzamiento).
- ✓ El uso de estas excepciones facilita la impresión de mensajes de tipo notificación desde el metodo *run()* del controlador.
- ✓ Ya no es necesario que las clases de comandos se comuniquen con el controlador para decirle que no imprima el tablero en el caso en que suceda algun error, o no imprimir el mensaje "*unknown command*".
- ✓ Cambiar el método *execute()* de la clase *Command* para que devuelva *boolean* en lugar de *void*. Este *boolean* puede ser utilizado por los comandos para comunicar al controlador que no es necesario hacer *printing*.

- Con estos cambios tampoco es necesario pasar el controlador como parámetro de los métodos *parse()* y *execute()* de la clase *Command*, ni en el método *parseCommand()* de la clase *CommandParser*.
- Además, puede prescindirse del método *setNoPrintGameState()* y su atributo *boolean* utilizados en el controlador de la práctica 2. Así, los atributos del controlador pueden ser:

```
public static final String commandErrorMsg = "Unknown command. Use
'help' to see the available commands“;
private Game game;
private Scanner scanner;
```

- Es útil pasar el *scanner* como argumento del método *parse()* de la clase *Command* para ser utilizado en las clases *PlayCommand* y *SaveCommand*. Esto evita declarar un metodo *getScanner()* en la clase *Controller*.

- La declaración de los métodos de la clase *Command* queda de la siguiente manera:

```
// execute returns a boolean to tell the controller whether or not to
print the game
public abstract boolean execute(Game game);
public abstract Command parse(String[] commandWords, Scanner in);
public String helpText(){
    return " " + commandText + ": " + helpText;
}
```

- No obstante, faltan por añadir las clausulas *throw* que sean necesarias.

- ✓ **Cuarto**, decidir si es necesario crear, lanzar y capturar mas excepciones en la aplicación 2048.
- ✓ **Quinto**, implementar manejo de excepciones para las situaciones en que se gana o se pierde el juego.
- ✓ Estas situaciones no son errores. Pero la gestión de tales excepciones es especialmente útil cuando el controlador, sabiendo por invocacion de método que el juego ha terminado, debe imprimir un mensaje u otro según se haya ganado o perdido el juego.
- ✓ Recomendación: implementar una *GameOverException* que contemple ambas situaciones.

3. Sumando Funcionalidad de Entrada-Salida

- ✓ En esta fase sumamos dos nuevos comandos a la aplicación 2048: *save <filename>* y *load <filename>.* El primero sirve para salvar en un fichero el estado completo del juego. El segundo permite cargar desde un fichero el estado del juego previamente salvado.
- ✓ Observaciones:
  - El manejo del parámetro fichero *<filename>* de ambos comandos debe ser *case-sensitive*.
  - El juego que se carga con el comando *load* puede ser de distinto tipo y tener un tablero de distinto tamaño al que el usuario esté jugando en el momento de ejecutar el comando.
- ✓ Seguir los siguientes **cinco pasos** para la implementación de esta fase.

- ✓ **Primero**, sumar los dos nuevos comandos a la aplicación 2048. Para ello, seguir las instrucciones dadas en la práctica 2 para los comandos *undo* y *redo*.
- ✓ **Segundo**, implementar el método *parse()* de las clases *SaveCommand* y *LoadCommand*.
- ✓ En la clase *SaveCommand*, este método debe validar que el *filename* contiene caracteres válidos y que dicho fichero no existe. Si contiene caracteres inválidos, lanzar una excepción apropiada de programador. Y si el fichero ya existe, preguntar al jugador si desea sobreescribirlo, pidiendo otro *filename* en caso negativo (ver el apéndice al final de la presentación).
- ✓ El método *parse()* de *LoadCommand* también debe validar *filename* y que el fichero existe.

- Para la implementación del método *parse()* de la clase *SaveCommand* puede encontrar útil es siguiente código:

```
Command command = null;
// tu código
String filename = confirmFileNameStringForWrite(filenameString, in);
if (filename != null)
    command = new SaveCommand(filename);
```

- De manera similar, la implementación del método parse() en la clase LoadCommand puede necesitar el siguiente código:

```
Command command = null;
// tu código
String filename = confirmFileNameStringForRead(filenameString, in);
if (filename != null)
    command = new LoadCommand(commandWords[1]);
```

- ✓ **Tercero**, revisar el tipo de juego utilizado en la clase *Game*.
- ✓ En la práctica 2, la clase *Game* contenía un atributo con información de la aplicación que se está jugando. Y este atributo podía ser del tipo *GameRules* o *GameType*.
- ✓ Ahora, en la práctica 3, el comando *save* debe escribir esta información en un fichero y resulta mucho más sencillo manejar un *string* del tipo de juego que un *string* de las reglas del juego. Por ello el atributo de la clase *Game* debe ser del tipo *GameType*. Aunque esto puede implicar algunos cambios en otras clases.
- ✓ Al ejecutar el comando *load*, el tipo de juego debe ser recuperado de dicha representación *string* y el objeto *GameRules* correspondiente (uno de las clases que implementan la interfaz *GameRules*) generado.

- ✓ **Cuarto**, implementar el método *execute()* de las clases *SaveCommand* y *LoadCommand*.
- ✓ Implementar flujos de caracteres en lugar de flujos de bytes. En concreto, utilizar *BufferedWriter* para escribir en fichero y *BufferedReader* para leer desde fichero.
- ✓ Utilizar bloques *try-with-resources* capturando *IOException* en cada uno de los métodos *execute()* (en *SaveCommand* y en *LoadCommand*). El siguiente código muestra un ejemplo:

```
try(BufferedWriter out = new BufferedWriter(new FileWriter(fileName)))
{
 // tu código
} catch (IOException ioe) {
 // tu código
}
```

- El método *execute()* de *SaveCommad* debe invocar un método *store()* de *Game* que a su vez invoca el método *store()* de *Board*. De manera análoga, el método *execute()* de *LoadCommand* invoca un método *load()* de *Game* que a su vez invoca el método *load()* de *Board*.
- Cada uno de estos métodos devuelve *void* a excepción del método *load()* de *Game*. Este debe devolver el tipo de juego para que el método *execute()* de *LoadCommand* pueda incluir dicho tipo de juego en el mensaje de "successful load" que debe ser mostrado en la salida antes de devolver el control al método *run()* de *Controller*.
- Cada clase *Board*, *Game*, *SaveCommand* y *LoadCommand* almacenan y cargan sus propios datos.

- ✓ **Quinto**, implementar el formato de fichero en los métodos vistos anteriormente.
- ✓ Los juegos deben almacenarse en fichero de texto utilizando el siguiente formato:

```
This file stores a saved 2048 game
4 0 0 0
0 0 2 0
0 0 0 0
2 0 0 0
2 4 original
```

- ✓ La primera línea contiene un mensaje que identifica que el fichero de texto ha sido salvado por la aplicación 2048. Y esta línea debe ser validada por el comando *load*. Si la primera línea que se carga no contiene este mensaje, hay que mostrar una excepción.

- A continuación, el formato contiene una línea en blanco seguida de una representación del tablero que consiste en líneas de *number-strings* separados por *tabs.*
- Para finalizar, el fichero contiene otra línea en blanco y una última línea con dos *number-strings* y una palabra separados por *tabs*. Estos tres elementos, de izquierda a derecha, contienen la siguiente información:
  - El número de celdas iniciales no nulas utilizadas en el juego,
  - El número actual de puntos del juego,
  - Una representación string del tipo de juego.
- Para ayudar en la generación del último elemento, el apéndice de esta presentación contiene código para la clase *GameType*. La generación de representaciones de objetos diseñada para facilitar almacenamiento o transmisión se conoce como *externalisation*.

- ✓ Observaciones:
  - El tamaño del tablero almacenado no tiene por qué coincidir con el tamaño del tablero del juego al tiempo de ejecutar el comando *load*. Por ello, hay que deducir el tamaño del tablero al leer la primera línea de datos del tablero en el fichero.
  - La última línea del tablero almacenado no es la última línea del fichero. Estoy hay que tenerlo en cuenta a la hora de implementar el bucle del cuerpo del método *load()* de la clase *Board*.

4. Mas Refactoring

- ✓ El método *lose()* de la interfaz *GameRules* necesita invocar un método *canMerge()* de *Board* para validar si es posible realizar alguna fusión en el tablero. Y este último método necesita, de forma repetida, evaluar si dos celdas vecinas pueden ser fusionadas. Sin embargo, esta condición no es la misma para todos los juegos.
- ✓ Para mejorar la interfaz *GameRules*, incluir un nuevo *default method* que evalúe dicha condición

```
default boolean canMergeNeighbours(Cell cell1, Cell cell2) {
    // tu código.
}
```

- ✓ Este método se incluye como *default* (y no como método normal de la interfaz) porque la condición es la misma para dos de los juegos.

- El método *lose()* debe pasar la referencia *this* como parámetro de la llamada a *canMerge()*. De esta manera se garantiza que una referencia del objeto correcto de *GameRules* sea pasado al método *canMerge().*

```
default boolean lose(Board board) {
    return board.isFull() && !board.canMerge(this);
}
```

- El método *merge()* de cada una de las clases que implementan la interfaz *GameRules* pueden usar también una llamada al nuevo método *canMergeNeighbours().*

```
if (canMergeNeighbours(self, other)) {
    // tu código.
}
```

4. Apéndice de Código

- ✓ Clase enumerado *GameType*. Se recomienda ver posibles equivalencias con la clase enumerado *Direction*.

```
package tp.pr3.logic.multigames;
public enum GameType {
    ORIG("2048, original version", "original", new Rules2048()),
    FIB("2048, Fibonacci version", "fib", new RulesFib()),
    INV("2048, inverse version", "inverse", new RulesInverse());

    private String userFriendlyName;
    private String parameterName;
    private GameRules correspondingRules;

    private GameType(String friendly, String param, GameRules rules){
        userFriendlyName = friendly;
        parameterName = param;
        correspondingRules = rules;
    }
```

```
// precondition : param string contains only lower-case characters
// used in parse method PlayCommand and load method in Game
public static GameType parse(String param) {
    for (GameType gameType : GameType.values()) {
        if (gameType.parameterName.equals(param))
        return gameType;
    }
    return null;
}

// used in PlayCommand to build help message, and in parse method
exception msg
public static String externaliseAll() {
    String s = "";
    for (GameType type : GameType.values())
        s = s + " " + type.parameterName + ",";
    return s.substring(1, s. length()-1);
}

// used in Game when constructing object and when executing play
command
public GameRules getRules() {
    return correspondingRules;
}
```

```
    // used in Game in store method
    public String externalise() {
        return parameterName;
    }

    // used in PlayCommand and LoadCommand, in parse methods
    // in ack message and success message, respectively
    public String toString() {
         return userFriendlyName;
    }
}
```

- Código para ser llamado desde el método *parse()* de la clase *SaveCommand*. Este código utiliza dos métodos de la clase *MiStringUtils* que se muestran a continuación y también pueden ser llamados desde el método *parse()* de la clase *LoadCommand*.

```
/* This code supposes the following attribute declarations :
 * private boolean filename_confirmed;
 * public static final filenameInUseMsg
 * = "The file already exists ; do you want to overwrite it ? (Y/N)";
 * You may also need to add a throws clause to the declarations .
*/
```

```
private String confirmFileNameStringForWrite(String filenameString,
Scanner in) {
    String loadName = filenameString;
    filename_confirmed = false;
    while (!filename_confirmed) {
        if (MyStringUtils.validFileName(loadName)) {
            File file = new File(loadName);
            if (!file.exists())
                filename_confirmed = true;
            else {
                loadName = getLoadName(filenameString, in);
            }
        } else {
            // ADD SOME CODE HERE
        }
    }
    return loadName;
}
```

```
public String getLoadName(String filenameString, Scanner in) {
    String newFilename = null;
    boolean yesOrNo = false;
    while (!yesOrNo) {
        System.out.print(filenameInUseMsg + ": ");
        String[] responseYorN =
           in.nextLine().toLowerCase().trim().split("\\s+");
        if (responseYorN.length == 1) {
           switch (responseYorN[0]) {
               case "y“: yesOrNo = true;
                         // ADD SOME CODE HERE
               case "n": yesOrNo = true;
                        // ADD SOME CODE HERE
               default: // ADD SOME CODE HERE
           }
        } else {
           // ADD SOME CODE HERE
        }
    }
    return newFilename;
}
```

```
package tp.pr3.util;
import java.io.File;
import java.io.IOException;
import java.io.FileOutputStream;

public class MyStringUtils {

    // used in text representation of Game and Board objects
    public static final int MARGIN_SIZE = 4;

    public static String repeat(String elmnt, int numOfTimes) {
        String result = "";
        for (int i = 0; i < numOfTimes; i++) {
            result += elmnt;
        }
        return result;
    }

    public static String centre(String text, int len) {
        String out = String.format("%"+len+"s%s%"+len+"s", "",text,"");
        float mid = (out.length()/2);
        float start = mid - (len/2);
        float end = start + len;
        return out.substring((int)start, (int)end);
   }
```

```
// Used to exist method:
org.eclipse.core.internal.resources.OS.isNameValid(filename).
// This method is not completely reliable since exception could also be
thrown due to:
// incorrect permissions , no space on disk , problem accessing the
device ,...
public static boolean validFileName(String filename) {
    File file = new File(filename);
    if (file.exists()) {
        return canWriteLocal(file);
    } else {
        try {
           file.createNewFile();
           file.delete();
           return true;
        } catch (Exception e) {
           return false;
        }
   }
}
```

```
    public static boolean canWriteLocal(File file) {
        // works OK on Linux but not on Windows (apparently!)
        if (!file.canWrite()) {
            return false;
        }
        // works on Windows
        try {
            new FileOutputStream(file, true).close();
        } catch (IOException e) {
            return false;
        }
        return true;
    }
}
```