UNIVERSIDAD
COMPLUTENSE
MADRID

Tecnología de la Programación

Presentación de la Práctica 3

(Basado en la práctica de Alberto Núñez Covarrubias y Simon Pickin)

Ana M. González de Miguel (ISIA, UCM)

Índice

1. Introducción

- ✓ Los objetivos cubiertos en esta práctica son: **excepciones** y **entrada/salida**.
- ✓ Fecha de entrega: **10 de Diciembre** a las 9:00.
- ✓ Con esta práctica mejoramos y extendemos la funcionalidad de la anterior:
  - Primero, incluimos el **manejo y tratamiento de excepciones**. Conseguimos un programa mas robusto y una mejor interoperabilidad con el usuario. Tratamos estados del programa de manera conveniente proporcionando información relevante como, por ejemplo, errores producidos al procesar comandos.
  - Segundo, gestionamos ficheros para poder **grabar y cargar partidas en disco**. Se incluyen los comandos necesarios para poder realizar la escritura en disco y la carga en memoria de una partida. Introducir estos comandos es sencillo gracias al patrón *Command* de la práctica anterior.

2. Manejo de Excepciones

- ✓ El tratamiento de excepciones es útil para controlar determinadas situaciones del juego en **tiempo de ejecución** como, por ejemplo, mostrar información relevante al jugador sobre la utilización de un comando.
- ✓ En la práctica anterior, cada comando invocaba al método correspondiente (de la clase *Controller* o de la clase *Game*) para realizar las operaciones.
- ✓ Por ejemplo, para incluir una planta en el tablero, la clase *Game* debía comprobar si la casilla indicada por el usuario estaba libre o si se disponían de suficientes *suncoins*. Y en caso de no poder incluir la planta se devolvía un valor *booleano* a *false* controlando que no se añadiera la planta. Pero no se indicaba el motivo exacto.

- ✓ En esta práctica, cada clase podrá **lanzar y procesar** determinadas excepciones para tratar determinadas situaciones del juego. En ocasiones esta gestión consistirá únicamente en dar un mensaje al usuario y en otras el tratamiento será mas complejo.
- ✓ Las excepciones relativas a los ficheros son explicadas en la siguiente sección.
- ✓ Inicialmente, deben manejarse **excepciones lanzadas por el sistema**. Es decir, aquellas que no son creadas ni lanzadas por el usuario. Al menos, debe tratarse la excepción *NumberFormatException*, lanzándola cuando se intente parsear un número, en formato *String*, que no pueda ser transformado al formato indicado.

- ✓ Además, deben crearse **dos nuevas excepciones**: *CommandParserException* y *CommandExecuteException*.
- ✓ *CommandParserException* debe tratar los errores producidos al parsear un comando. Es decir, aquellos producidos durante la ejecución del método *parse(),* tales como comando desconocido o parámetros incorrectos.
- ✓ *ComandExecuteException* debe utilizarse para tratar las situaciones de error al ejecutar el método *execute()* de un comando. Por ejemplo, que una planta no pueda ser incluida en el tablero.

- ✓ Con la **implementación** de estas excepciones debe eliminarse la comunicación directa entre los comandos y el controlador.
- ✓ Ya no será necesario que el comando indique al controlador cuando se ha producido un error o cuando debe actualizarse el tablero. Basta con controlar las excepciones que pueden producirse durante la ejecución de los comandos.
- ✓ Además, como ahora se tratan situaciones de error tanto en el parseo como en la ejecución de los comandos, los mensajes de error son mucho más descriptivos.

- ✓ Los **cambios** a realizar son los siguientes:
  - Modificar la signatura del método *execute()* de *Command* para que devuelva un valor de tipo *boolean* en lugar de *void*. Así, si el valor devuelto es *true*, el controlador podrá actualizar el tablero. En otro caso, el controlador no imprimirá el tablero.
  - Eliminar de la lista de parámetros de los métodos *parse()* y *execute()* la referencia al objeto de la clase *Controller*. Eliminarlo también del método *parseCommand()* de la clase *CommandParser*.
  - El controlador deberá poder capturar las excepciones lanzadas por los métodos *parse()* y *execute()* de la clase *Command*.
  - Incluir el tratamiento de excepciones del sistema en las clases correspondientes para que puedan ser capturadas por el controlador imprimiendo por pantalla los mensajes correspondientes.

- ✓ Ahora todos los mensajes se imprimen desde el método *run()* de *Controller* cuyo **bucle** debe asemejarse al siguiente código:

```
while (!game.isFinished()){
    System.out.print(prompt);
    String[] words = in.nextLine().trim(). split ("\\s+");
    try {
        Command command = CommandGenerator.parse(words);
        if (command != null) {
            if (command.execute(game))
                printGame();
        } else
            System.out.println(unknownCommandMsg);
    } catch (CommandParseException | CommandExecuteException ex) {
        System.out.format(ex.getMessage() + "%n%n");
    }
}
```

3. Ficheros

- ✓ Básicamente, la nueva funcionalidad con ficheros consiste en poder **guardar y cargar** partidas almacenadas en **ficheros de texto**.
- ✓ Para guardar el estado de una partida en un fichero y, para cargar el estado de una partida previamente guardada, se van a crear los comandos *SaveCommand* y *LoadCommand*, respectivamente.
  - *save fileName* guardará el estado actual de la partida en el fichero *fileName* (teniendo en cuenta que *fileName* es "case sensitive").
  - *load fileName* cargará la partida almacenada en el fichero *fileName*.

- ✓ Durante una misma partida será posible guardar varios estados del juego utilizando o no ficheros diferentes.
- ✓ En el caso de que ya exista un fichero con el mismo nombre, se sobreescribirá el mismo.
- ✓ El **formato del fichero de texto** que contiene el estado de una partida es el siguiente:

```
cycle: yy
sunCoins: ss
level: ll
remZombies: zz
plantList: P1, P2, ..., Pn
zombieList: Z1, Z2, ..., Zm
```

- ✓ donde
  - *yy* es el número del ciclo actual,
  - *ss* es el número de *sunCoins* que tiene el jugador,
  - *ll* es el nivel del juego,
  - *zz* es el número de zombies que quedan por salir,
  - *plantList* es la lista de plantas que están en el tablero y
  - *zombieList* es la lista de los zombies que están en el tablero.
- ✓ Las listas *plantList* y *zombieList* contienen todas las instancias de plantas y zombies que están en tablero, separadas por comas. Esto es, *P1, P2, ..., Pn* para las plantas y *Z1, Z2, ..., Zm* para los zombies. Para cada elemento de estas listas se guarda la siguiente información:

```
symbol:lr:x:y:t
```

- ✓ donde
  - *symbol* es el símbolo del elemento,
  - *lr* es la vida restante,
  - *x* es el número de fila,
  - *y* es el número de columna y
  - *t* es el número de ciclos que faltan hasta que el objeto de juego tenga que realizar la siguiente acción.
- ✓ El manejo de ficheros implica el uso de nuevos tipos de **excepciones**, tanto del sistema como definidas por el usuario.
- ✓ Se debe crear una nueva excepción *FileContentsException* y lanzarla en caso de detectar un problema en el contenido del fichero del tipo que no puede ser detectado por el sistema, p.ej. un *level* desconocido.

- ✓ Para la **implementación** de esta funcionalidad usaremos **flujos de caracteres** en lugar de flujos de bytes.
- ✓ En particular, recomendamos el uso de *BufferedWriter* y *FileWriter* para escribir en un fichero, y *BufferedReader* y *FileReader* para leer de un fichero.
- ✓ Además, se recomienda el uso de bloques **try-with-resources** para el código donde se abre el fichero, capturando *IOException* en cada uno de los métodos *execute()* de las clases *LoadCommand* y *SaveCommand*. Cada uno de estos métodos devuelve *void*.

- ✓ Para **guardar** el estado de una partida:
  - El método *execute()* de la clase *SaveCommand* debe:
    - Comprobar que el nombre del fichero proporcionado por el usuario es un nombre válido de fichero, donde esta noción depende del sistema operativo.
    - Intentar abrir el fichero con este nombre en escritura. Para simplificar la tarea, si el fichero ya existe, se sobreescribirá.
    - Escribir la cabecera, que consta del mensaje *"Plants Vs Zombies v3.0",* en el fichero seguido por una línea en blanco. La extensión del fichero, que será siempre *.dat*, la añadirá automáticamente el programa. Así, el usuario únicamente deberá proporcionar el nombre sin extensión.
    - Invocar a un método llamado *store()* de la clase *Game*, que a su vez llamará a un método *store()* de la clase *Board*, si existe esta última clase.

- El método *store()* de *Game* (o de *Board*, en su caso):
  - o bien puede llamar a un método *store()* de cada una de las dos listas (o lista única), que a su vez llaman a un método *store()* de cada elemento de cada lista, de modo que cada clase se ocupe de escribir sus propios datos en fichero,
  - o bien puede llamar a un método *externalise()* de cada una de las dos listas (o lista única), que a su vez llaman a un método *externalise()* de cada elemento de cada lista, donde los métodos *externalise()* devuelven un objeto *String*, de manera parecida al funcionamiento del método *toString()*.
- En caso de haber podido guardar el estado del juego en fichero con éxito, se debe imprimir el mensaje:

```
"Game successfully saved in file
<nombre_proporcionado_por_el_usuario>.dat. Use
the load command to reload it".
```

- ✓ Para **cargar** el estado de una partida:
  - El método *execute()* de la clase *LoadCommand* debe:
    - Comprobar que el nombre del fichero proporcionado por el usuario es un nombre válido y que existe.
    - Intentar abrir el fichero con este nombre en lectura.
    - Leer la cabecera, que consta del mensaje *"Plants Vs Zombies v3.0"*, seguido por una línea en blanco.
    - Invocar a un método llamado *load()* de la clase *Game*, que a su vez llamará a un método *load()* de la clase *Board*, si existe esta última clase. No hace falta que el método *load()* de *Game* (o de *Board* en su caso) invoque a un método *load()* de cada una de las dos listas (o lista única), que a su vez llamen a un método *load()* de cada elemento de cada lista, ya que la lectura de ficheros se hace más fácilmente por líneas enteras. En vez de ello, el método *load()* de *Game* puede crear una nueva lista de plantas y una nueva lista de zombies para que las nuevas listas puedan poblarse con los datos leídos de fichero.

- En caso de ocurrir un problema a la hora de cargar un fichero, el método *execute()* de la clase *LoadCommand* debe recoger la excepción lanzada y reempaquetarla en una *CommandExcecuteException*. El método *load()* de *Game*, así como los métodos llamados desde este método, pueden comprobar la coherencia de los datos leídos y, en caso de haber un problema, lanzar un *FileContentsException* que será recogida por el método *execute()* de la clase *LoadCommand*.
- En caso de haber podido cargar el estado del juego de fichero con éxito, se debe imprimir el mensaje:

```
"Game successfully loaded from file
<nombre_proporcionado_por_el_usuario>".
```

- Tras cargar una partida, se utiliza el pintado del tablero en modo *Release*.

4. Mejora de la Calidad del código

- ✓ **Robustez**. Evitar los estados incoherentes al cargar un fichero de contenido inválido guardando los datos de *game* en una copia de seguridad (en memoria).
- ✓ Generar la copia antes de empezar a cargar los datos del fichero.
- ✓ Cuando se produzca una excepción debida a que el contenido del fichero no es correcto, recoger la excepción desde el método *load()* de *Game* y reponer el estado anterior de *game* a partir de la copia. A continuación relanzar la misma excepción para que se recoja mas arriba en la pila de llamadas.

- ✓ **Mas refactorización**. La aplicación sería más mantenible y flexible si las clases *PlantFactory* y *ZombieFactory* utilizaran el mismo mecanismo para la creación de plantas que la clase *CommandParser* utiliza para la creación de comandos. Es decir, que el funcionamiento de los métodos *getPlant* y *getZombie* de las primeras fuera el del método *parse* de la segunda, lo que implicaría tener un método *parse* en los objetos del juego.
- ✓ También se podría crear una clase *PrinterManager* para hacer el papel de la clase *CommandParser* y utilizar este mismo mecanismo para gestionar los distintos tipos de *printer*, lo que implicaría tener un método *parse* en los *printers*.

5. Anexo de la Práctica

- ✓ Añadir los siguientes métodos a la clase *MyStringUtils*

```
package tp.p3.util;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.nio.file.Files ;
import java.nio.file.InvalidPathException;

public class MyStringUtils {
   //…
   // returns true if string argument is a valid filename
   public static boolean isValidFilename(String filename) {
      try {
         Paths.get(filename);
         return true;
      } catch (InvalidPathException ipe) {
         return false;
      }
   }
```

```
    // returns true if file with that name exists (in which case, it may not be
    accessible )
    public static boolean fileExists(String filename) {
        try {
            Path path = Paths.get(filename);
            return Files.exists(path) && Files.isRegularFile(path);
        } catch (InvalidPathException ipe) {
            return false; // filename invalid => file cannot exist
        }
    }
    // returns true if file with that name exists and is readable
    public static boolean isReadable(String filename) {
        try {
            Path path = Paths.get(filename);
            return Files.exists(path) && Files.isRegularFile(path) &&
    Files.isReadable(path);
        } catch (InvalidPathException ipe) {
            return false; // filename invalid
        }
    }
}
```

✓ Enumerado Level

```
package tp.p3.game;
public enum Level {
    EASY(3, 0.1), HARD(5, 0.2), INSANE(10, 0.3);
    private int numberOfZombies;
    private double zombieFrequency;

    private Level(int zombieNum, double zombieFreq){
       numberOfZombies = zombieNum;
       zombieFrequency = zombieFreq;
    }

    public int getNumberOfZombies() {
       return numberOfZombies;
    }

    public double getZombieFrequency() {
       return zombieFrequency;
    }
```

```
    public static Level parse(String inputString) {
        for (Level level : Level.values())
            if (level.name().equalsIgnoreCase(inputString))
                return level;
        return null;
    }

    public static String all (String separator) {
        StringBuilder sb = new StringBuilder();
        for (Level level : Level.values())
            sb.append(level.name() + separator);
        String allLevels = sb.toString();
        return allLevels.substring(0, allLevels.length()-separator.length());
    }
}
```

✓ Método que puede llamarse desde *load()* de Game:

```
public static final String wrongPrefixMsg = "unknown game attribute: "
public static final String lineTooLongMsg = "too many words on line commencing: "
public static final String lineTooShortMsg = "missing data on line commencing: "

public String[] loadLine(BufferedReader inStream, String prefix, boolean isList)
throws IOException, FileContentsException {
    String line = inStream.readLine().trim();
    // absence of the prefix is invalid
   if (!line.startsWith(prefix + ":"))
       throw new FileContentsException(wrongPrefixMsg + prefix);
   // cut the prefix and the following colon off the line
   // then trim it to get the attribute contents
   String contentString = line.substring(prefix.length()+1).trim();
   String[] words;
   // the attribute contents are not empty
   if (!contentString.equals("")) {
      if (!isList ) {
         // split non-list attribute contents into words
         // using 1-or-more-white-spaces as separator
         words = contentString.split("\\s+");
```

```
                // a non-list attribute with contents of more than one word is invalid
                if (words.length != 1)
                    throw new FileContentsException(lineTooLongMsg + prefix);
            } else
                // split list attribute contents into words
                // using comma+0-or-more-white-spaces as separator
                words = contentString.split(",\\s*");
        // the attribute contents are empty
        } else {
            // a non-list attribute with empty contents is invalid
            if (!isList )
                throw new FileContentsException(lineTooShortMsg + prefix);
            // a list attibute with empty contents is valid;
            // use a zero-length array to store its words
            words = new String[0];
        }
        return words;
    }
```