

---

# Práctica 2: Space Invaders refactored. Completa

---

**Fecha de entrega:** 5 de Diciembre de 2019, 9:00

**Objetivo:** Herencia, polimorfismo, clases abstractas, interfaces, excepciones y gestión de ficheros

## 1. Introducción

El objetivo de esta práctica consiste, fundamentalmente, en aplicar los mecanismos que ofrece la POO para mejorar el código desarrollado hasta ahora. En particular, en esta versión de la práctica incluiremos las siguientes mejoras:

- Primero, como se explica en la sección 2, refactorizamos<sup>1</sup> el código de la práctica anterior. Para ello, modificaremos parte del controlador distribuyendo su funcionalidad entre un conjunto de clases.
- Vamos a hacer uso de la herencia para reorganizar los objetos de juego. Hemos visto que hay mucho código repetido en los distintos tipos de aliens o entre el misil y las bombas. Por ello, vamos a crear una estructura de clases que nos permita extender fácilmente la funcionalidad del juego.
- Una vez refactorizada la práctica, vamos a añadir nuevos objetos al juego.
- La herencia también nos va a permitir redefinir cómo almacenamos la información del estado del juego. En la práctica anterior, al no usar herencia, debíamos tener una lista para cada conjunto de objetos. Sin embargo, en esta versión de la práctica, podremos usar una sola estructura de datos para todos los objetos de juego.
- Vamos a incluir la posibilidad de modificar cómo se representa el tablero. Para ello, crearemos otro *printer* con el que podamos ver el juego de forma *serializada*.

---

<sup>1</sup>Refactorizar consiste en cambiar la estructura del código (se supone que para mejorarlo) sin cambiar su funcionalidad.

- Utilizaremos la serialización poder grabar y cargar partidas en disco. Para ello, ampliaremos el juego para gestionar ficheros de entrada y salida.
- También vamos a incluir el manejo y tratamiento de excepciones. En ocasiones, existen estados en la ejecución del programa que deben ser tratados convenientemente. Además, cada estado debe proporcionar al usuario información relevante como, por ejemplo, errores producidos al procesar un determinado comando. En este caso, el objetivo es dotar al programa de mayor robustez, así como mejorar la interoperabilidad con el usuario.

Todos los cambios comentados anteriormente se llevarán a cabo de forma progresiva. El objetivo principal es extender la práctica de una manera robusta, preservando la funcionalidad en cada paso que hagamos y modificando el mínimo código para ampliarla.

**Importante:** Hemos dividido el enunciado en varias partes que iremos publicando progresivamente, en esta primera parte nos centraremos en el primer ítem de la lista. Solo habrá una entrega para toda la práctica completa.

## 2. Refactorización de la solución de la práctica anterior

Hay una regla no escrita en programación que dice *Fat models and skinny controllers*. Lo que viene a decir es que el código de los controladores debe ser mínimo y, para ello, deberemos llevar la mayor parte de la funcionalidad a los modelos. Una manera de adelgazar el controlador es utilizando el patrón *Command* que, como veremos, permite encapsular acciones de manera uniforme y extender el sistema con nuevas acciones sin modificar el controlador.

El cuerpo del método `run` del controlador va a tener - más o menos - este aspecto. Tu código no tiene que ser exactamente igual, pero lo importante es que veas que se asemeja a esta propuesta.

```
while (!game.isFinished()){
    System.out.println(PROMPT);
    String[] words = in.nextLine().toLowerCase().trim().split ("\\s+");

    Command command = CommandGenerator.parse(words);
    if (command != null) {
        if (command.execute(game))
            System.out.println(game);
    }
    else {
        System.out.format(unknownCommandMsg);
    }
}
```

Básicamente, mientras el juego no termina, leemos un comando de la consola, lo parseamos, lo ejecutamos y, si la ejecución es satisfactoria y ha cambiado el estado del juego, lo repintamos. Este mismo controlador nos valdría para diferentes versiones del juego o incluso para diferentes juegos. En la próxima sección vamos a ver cómo funciona el patrón *Command*.

### 2.1. Patrón Command

El patrón *command* es un patrón de diseño<sup>2</sup> muy conocido. En esta práctica no necesitas conocer de este patrón más de lo que se explica aquí. Para aplicarlo, cada comando del juego se representa por una clase diferente, que llamamos *MoveCommand*, *ShootCommand*, *UpdateCommand*, *ResetCommand*, *HelpCommand*,... y que heredan de una clase abstracta *Command*. Las clases concretas invocan métodos de la clase *Game* para ejecutar los comandos respectivos.

En la práctica anterior, para saber qué comando se ejecutaba, el método *run* del controlador contenía un switch - o una serie de if's anidados - cuyas opciones correspondían a los diferentes comandos. Con la aplicación del patrón *command*, para saber qué comando ejecutar, el método *run* del controlador divide en palabras el texto proporcionado por el usuario (*input*) a través de la consola, para a continuación invocar un método de la *clase utilidad*<sup>3</sup> *CommandGenerator*, al que se le pasa el *input* como parámetro. Este método le pasa, a su vez, el *input* a un objeto *comando* de cada una de las clases concretas (que, como decimos, son subclases de *Command*) para averiguar cuál de ellos lo acepta como correcto. De esta forma, cada subclase de *Command* busca en el *input* el texto del comando que la subclase representa.

Aquel objeto *comando* que acepte el input como correcto devuelve al *CommandGenerator* otro objeto *comando* de la misma clase que él. El parseador pasará, a su vez, el objeto *comando* recibido al controlador. Los objetos *comando* para los cuales el input no es correcto devuelven el valor null. Si ninguna de las subclases concretas de comando acepta el input como correcto, es decir, si todas ellas devuelven null, el controlador informa al usuario de que el texto introducido no corresponde a ningún comando conocido. De esta forma, si el texto proporcionado por el usuario corresponde a un comando del sistema, el controlador obtiene del parseador *CommandGenerator* un objeto de la subclase que representa a ese comando y que puede, a su vez, ejecutar el comando.

### Implementación

El código de la clase abstracta *Command* es el siguiente:

```
public abstract class Command {

    protected final String name;
    protected final String shortcut;
    private final String details ;
    private final String help;

    protected static final String incorrectNumArgsMsg = "Incorrect number of arguments";
    protected static final String incorrectArgsMsg = "Incorrect argument format";

    public Command(String name, String shortcut, String details, String help){
        this.name = name;
        this.shortCut = shortCut;
        this.details = details;
        this.help = help;
    }
}
```

<sup>2</sup>Los patrones de diseño de software en general, y el patrón *command* en particular, se estudian en la asignatura Ingeniería del Software.

<sup>3</sup>Una clase utilidad es aquella en la que todos los métodos son estáticos.

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

public abstract Command parse(String[] commandWords);

protected boolean matchCommandName(String name) {
    return this.shortcut.equalsIgnoreCase(name) ||
           this.name.equalsIgnoreCase(name);
}

public String helpText(){
    return details + " : " + help + "\n";
}
}
```

De los métodos abstractos anteriores, `execute` se implementa invocando algún método con el objeto `game` pasado como parámetro y ejecutando alguna acción más. El método `parse` se implementa con un método que parsea el texto de su primer argumento (que es el texto proporcionado por el usuario por consola, dividido en palabras) y devuelve:

- o bien un objeto de una subclase de `Command`, si el texto que ha dado lugar al primer argumento se corresponde con el texto asociado a esa subclase,
- o el valor `null`, en otro caso.

La clase `CommandGenerator` contiene la siguiente declaración e inicialización de atributo:

```
private static Command[] availableCommands = {
    new ListCommand(),
    new HelpCommand(),
    new ResetCommand(),
    new ExitCommand(),
    new ListCommand(),
    new UpdateCommand(),
    new MoveCommand(),
    new ShockwaveCommand()
};
```

Este atributo se usa en los dos siguientes métodos de `CommandGenerator`:

- `public static Command parseCommand(String[] commandWords)`, que, a su vez, invoca el método `parse` de cada subclase de `Command`, tal y como se ha explicado anteriormente,
- `public static String commandHelp()`, que tiene una estructura similar al método anterior, pero invocando el método `helpText()` de cada subclase de `Command`. Este método es invocado por el método `execute` de la clase `HelpCommand`.

Una de las ventajas de usar el patrón *Command* es que es muy fácil y natural implementar el “Ctrl-Z” o `undo`. Al encapsular los comandos como objetos los podemos apilar y desapilar, yendo para adelante y para atrás en la historia del juego. No lo vamos a implementar en esta práctica pero seguro que durante tu carrera lo utilizas más de una vez.

## 2.2. Herencia y polimorfismo

Quizás la parte más frustrante y mayor fuente de errores de la primera práctica es tener que replicar código en los objetos del juego y en las listas de objetos. Esto lo vamos a resolver usando la herramienta básica de la programación orientada a objetos: la *herencia*. Para ello, se creará una jerarquía de clases que heredan de `GameObject`.

- La clase abstracta `GameObject` tendrá los atributos y métodos básicos para controlar la posición en el tablero y una referencia a la clase `Game`.
- De `GameObject` heredarán `Ship` y `Weapon`. La primera clase representará las naves del juego, mientras que la segunda representará los objetos que causan daño.
- La clase `EnemyShip`, que hereda de la clase `Ship`, representará todas las naves enemigas, incluyendo el Ovni. Entre otros elementos, esta clase deberá gestionar el movimiento de las naves y los puntos que se obtienen al ser destruidas por el jugador.
- De forma similar, la clase `UCMSHIP` también hereda de `Ship` y representará la nave que controla el jugador.
- La clase `AlienShip` es probablemente una de las clases más complicadas de la práctica. En particular, esta clase – junto con la clase `Ovni` – hereda de `EnemyShip` y representará las naves del juego que se mueven con un comportamiento grupal, es decir, todas en la misma dirección. Su movimiento seguirá las mismas reglas que fueron definidas en la práctica 1. Para representar este movimiento, se utilizará un atributo estático que represente el número de naves que deben realizar un determinado movimiento. Así, todos los objetos de esta clase compartirán el mismo atributo.
- Intuitivamente, las clases `RegularAlien` y `DestroyerAlien` heredarán de `AlienShip`, implementando su comportamiento.
- Finalmente, las clases `Bomb` y `UCMMissile` heredan de `Weapon`, representando el movimiento de los disparos realizados por las naves `DestroyerAlien` y `UCMSHIP`, respectivamente.
- La clase `Shockwave` también heredarán de `Weapon`; aunque no se mueva y tenga una forma de atacar diferente también lo podemos considerar como un arma. Si más adelante tuviéramos otros *power-ups* refinaremos la jerarquía de clases para tener armas con diferentes comportamientos.

Es importante remarcar que en esta versión de la práctica, la mayor parte de la lógica estará distribuida en los propios elementos del juego, quedando la clase `Game` liberada de esta tarea. De esta forma se consigue un diseño escalable, permitiendo añadir nuevos elementos en el juego de forma sencilla.

Adicionalmente, podemos utilizar la herencia para refactorizar el código de las listas. En esta práctica, utilizaremos una clase – llamada `GameObjectBoard` – que será la encargada de gestionar la lista de elementos de tipo `GameObject`. La clase `Game` solo tendrá un atributo de tipo `GameObjectBoard` y otro de tipo `UCMSHIP`. Con estos dos atributos se gestionarán todos los elementos del juego, incluidos el Ovni, los disparos del jugador y las bombas lanzadas por las naves enemigas.

El hecho de emplear la palabra *board* no quiere decir que ahora vayamos a utilizar como tablero una matriz para gestionar los objetos; seguiremos usando una lista donde estarán todos los objetos de juego.

### 2.3. Detalles de implementación

Hay varios aspectos que van a cambiar radicalmente la estructura del código:

- Sólo tenemos un contenedor para todos los objetos.
- Desde el **Game** y el **board** sólo manejamos abstracciones de los objetos, por lo que no podemos distinguir quién es quién.
- Toda la lógica del juego estará en los objetos de juego. Cada clase concreta sabe sus detalles acerca de cómo se mueve, como ataca, como recibe ataque y cuando realiza *computer actions*.

Estas refactorizaciones son complejas y os vamos a dar ayuda; empecemos por la clase **Game**, la cual se encarga de demasiadas tareas en la práctica 1. En esta nueva versión, al igual que hemos hecho con el controlador, vamos a delegar su comportamiento a las demás clases. Veamos un fragmento de código.

```
public class Game implements IPlayerController{
    ....

    GameObjectBoard board;
    private UCMSHIP player;
    ....

    public Game (Level level, Random random){
        this.rand = random;
        this.level = level;
        initializer = new BoardInitializer();
        initGame();
    }

    public void initGame () {
        currentCycle = 0;
        board = initializer .initialize (this, level );
        player = new UCMSHIP(this, DIM_X / 2, DIM_Y - 1);
        board.add(player);
    }
    ....

    public boolean aliensWin() {
        return !player.isAlive () || AlienShip.haveLanded();
    }

    private boolean playerWin () {
        return AlienShip.allDead();
    }

    public void update() {
        board.computerAction();
        board.update();
        currentCycle += 1;
    }
    ...
}
```

El nuevo **game** mantiene una referencia al **player** y al **board** donde se almacenan los objetos de juego. Cuando tiene que hacer alguna acción, la delega a la clase correspondiente. Podríamos decir que el **Game** no hace absolutamente nada salvo delegar. De hecho

el código de la clase `Game` es tan pequeño que os lo vamos a dar en los anexos casi completamente. Igual tu código difiere en alguna parte o en el nombre de algún método o atributo, puedes modificar el nuestro, o el tuyo, como te sea más cómodo; lo importante es que entiendas bien su función.

En el `Game` usamos una clase auxiliar para inicializar el juego. El `boardInitializer` se encarga de añadir los objetos de juego en el juego dependiendo del nivel.

Si nos fijamos en la declaración del `Game` vemos que implementa un interfaz `IPlayerController`. Esta interfaz no es 100 % necesaria, pero nos ayuda a abstraer qué métodos son los necesarios para tratar la comunicación con el jugador. En realidad es lo que se conoce como *mixin* es una forma de incluir métodos de una clase en otra, sin que exista relación de herencia entre ellas.

```
public interface IPlayerController {  
  
    // Player actions  
    public boolean move (int numCells);  
    public boolean shootMissile();  
    public boolean shockWave();  
  
    // Callbacks  
    public void receivePoints(int points);  
    public void enableShockWave();  
    public void enableMissile();  
}
```

Vemos que hay dos tipos de métodos:

- Las acciones que puede hacer el jugador. Estas llamadas le llegan a través de los comandos del controlador.
- Los *callbacks* que son las acciones de retorno. Por ejemplo cuando se habilita el *shockWave*, después de matar al *Ovni*. *Callback* es un término muy común para denotar este tipo de funciones, que se llaman al final de unas determinadas acciones.

El `Game` debe implementar estos métodos del interfaz, aunque recuerda que no debe hacer nada más que delegar al objeto que le corresponda.

Vamos a usar otro interfaz para encapsular los métodos relacionados con los *ataques*. La clase `GameObject` implementa el interfaz `IAttack`. La idea es que todos los objetos del juego deben tener la posibilidad de atacar o ser atacados. Por ello, tenemos que implementar la posibilidad de recibir ataque por cada uno de los proyectiles.

```
public interface IAttack {  
    default boolean performAttack(GameObject other) {return false;};  
  
    default boolean receiveMissileAttack(int damage) {return false;};  
    default boolean receiveBombAttack(int damage) {return false;};  
    default boolean receiveShockWaveAttack(int damage) {return false;};  
}
```

Por defecto `performAttack` devuelve `false` lo que quiere decir que el objeto de juego no hace daño (sólo hacen daño el misil, las bombas y el shockwave). Los *destroyers* no hace daño, lo que hacen es lanzar bombas que son las que realmente hacen daño. Si un objeto de juego puede atacar, tendrá que sobrescribir el método e implementar la lógica de su ataque.

De igual manera todos los objetos pueden recibir daño de otro objeto. Por defecto no lo reciben. Pero, por ejemplo, los aliens y las bombas sí reciben daño del misil, o el UCMSShip y el misil sí reciben ataques de la bomba.

En el anexo además de parte de la implementación del `GameObject` también encontrarás las cabeceras de la clase `GameObjectBoard`, donde se encapsula el comportamiento de la lista de objetos de juego. Deberás rellenar esos métodos, al igual que en el `Game` esos métodos tienen muy poca lógica, toda la funcionalidad se delega a los objetos de juego.

Los objetos de juego que hacen acciones aleatorias, como tirar bombas, deberán implementar otro interfaz `IExecuteRandomActions` donde se encapsulan como métodos estáticos los métodos para saber si ejecuta o no la acción. Este código también te lo damos para que lo uses donde sea necesario.

Uno de los objetos que implementa `IExecuteRandomActions` es el `Ovni`. Recomendamos siempre tener un ovni en el tablero, con atributo de estado para saber si está `enable` o `disable`. Así podemos gestionar la aparición de un nuevo ovni desde el mismo objeto ya que consistirá simplemente en habilitarlo, en lugar de añadir un nuevo objeto.

### 3. Extensión del juego

#### 3.1. Incorporación de nuevos objetos de juego

En esta versión de la práctica vamos a hacer una pequeña ampliación:

- **Nave explosiva.** Las naves de tipo *común* pueden transformarse en una nave *explosiva* durante la partida. Si eso ocurre – esta nave – tras ser destruida, quita un punto de vida a todos los objetos que hay en una casilla adyacente (horizontal, vertical y diagonalmente). La probabilidad de que una nave *común* se transforme en nave *explosiva* es del 5% y se calculará en la acción `computer action`. El movimiento, los puntos de daño y los puntos obtenidos al ser destruida, serán los mismos que los de la nave común. Sin embargo, se representará en pantalla con la letra `E`.
- **SuperMisil.** El jugador puede “comprar” un *supermisil* por 20 puntos. Para ello se deberá incluir el comando adicional `comprar supermisil`, el cual recibe un único argumento cuyo valor tiene que ser `supermisil`. En la interfaz de usuario se deberá mostrar el número de supermisiles que éste posee durante la partida, comenzando con cero supermisiles al inicio del juego. Además, se deberá modificar el comando `shoot` para que acepte un parámetro nuevo, en este caso *supermisil*, de forma que la nave pueda lanzar uno de los supermisiles comprados. En el caso de que el jugador ejecute este comando y no disponga de supermisiles, se mostrará por pantalla el correspondiente mensaje de error, de igual forma que si al comprar el supermisil, no se dispone de los puntos necesarios. El comportamiento del disparo normal será el mismo que el de la práctica 1. El supermisil actúa de forma similar al misil, salvo que, en este caso, causa dos puntos de daño a las naves enemigas. Una vez impacte con un proyectil enemigo o una nave, aunque ésta tenga un único punto de vida, el supermisil desaparece.

Si has hecho bien la parte I y II de la práctica esta extensión te tiene que resultar sencilla.

#### 3.2. Serializar el juego

En esta versión de la práctica vamos a permitir dos modos de pintado: `formatted` y `serialized`. El modo `formatted` es el que vimos en la práctica anterior, donde el juego se



muestra como un tablero, mientras que en el modo `serialized` mostramos más información como texto plano sin formato. Esta nueva forma de pintado nos sirve para hacer *debug* del juego y, como veremos en la siguiente sección, para poder salvar el juego en un fichero.

También crearemos un nuevo comando `serialize` que nos muestre el juego serializado. El formato de serialización será el siguiente:

- La primera línea será siempre — Space Invaders v2.0 —
- La segunda línea dará información sobre el Game: `G;totalCycles`
- La tercera línea dará información sobre el Player: `P;x,y;live;points;superpower;missiles`
- Después, en cada línea, daremos información sobre los objetos de juego que estén en el tablero:
  - Ovni: `O;x,y;live`
  - Regular alien: `R;x,y;live;cyclesNextAlienMove;dir`
  - Destroyer alien: `D;x,y;live;cyclesNextAlienMove;dir`
  - Explosive alien: `E;x,y;live;cyclesNextAlienMove;dir`
  - Laser: `L;x,y`
  - Supermissile: `X;x,y`
  - Bomb: `B;x,y`

Por ejemplo esto sería una traza de salida:

```
Command > serialize
--- Space Invaders v2.0 ---
G:13;3
P;4;7;2;0;false;0
O;6;0;1
L;4;6
R;3;2;2;2;left
D;3;3;1;2;left
D;4;3;1;2;left
E;4;2;2;2;left
E;2;2;2;2;left
E;5;2;2;2;left
B;2;6
```

Como la dirección y el número de ciclos para el siguiente movimiento de los aliens es el mismo, lo podemos serializar a nivel de juego o a nivel de alien, aunque en este último caso habrá duplicidad de información. Este comando no cambia el estado del juego, simplemente vuelca la información en la consola. Para implementar esta funcionalidad se recomienda hacer los siguientes cambios.

- Vamos a crear un interfaz `GamePrinter` con los siguientes métodos:

```
public interface GamePrinter {
    String toString(Game game);
    public GamePrinter parse(String name);
    public String helpText();
}
```

- Renombraremos el **GamePrinter** de la práctica anterior a **FormattedPrinter** y haremos que implemente el interfaz anterior. Posiblemente tendrás que hacer pequeños ajustes para implementar el interfaz.
- Programamos la nueva clase **Serializer** que también implementará **GamePrinter**.
- Para desacoplar la vista de la lógica de la aplicación haremos que el **Game** no dependa del **GamePrinter** y así poder cambiar la manera que se pinta el juego sin hacer cambios, así que eliminaremos cualquier referencia a **GamePrinter**.
- El controlador y los comandos sí podrán hacer uso del **GamePrinter**. Así, en cada bucle del juego, cuando haya que repintar el controlador ejecutará la instrucción:

```
System.out.println(printer.toString(game));
```

- **printer** es un objeto de una clase que implemente **GamePrinter**, normalmente, el controlador utilizará el **FormattedPrinter** como en la primera práctica.
- Finalmente, crearemos un nuevo comando para serializar **SerializeCommand** que añadiremos a la lista de comandos disponibles.

Para gestionar los diferentes printers, al igual que hicimos con los comandos, vamos a crear una clase generadora **PrinterGenerator** donde se subscribe los diferentes *printers*. Esta clase, así como el interfaz **GamePrinter**, se proporciona en los anexos:

```
public class PrinterGenerator {
    private static GamePrinter[] availablePrinters = {
        new BoardPrinter(Game.DIM_X, Game.DIM_Y),
        new Serializer(),
    };

    public static GamePrinter parse(String name) {
        ...
    }

    public static String printerHelp() {
        ...
    }
}
```

Esta clase se usará en el controlador y en el comando **Serialize** para imprimir con el formato correspondiente. Aprovechando que cada uno de los printers tiene un método *printerHelp*, vamos a añadir un nuevo comando **ListPrintersCommand** que nos liste los printers disponibles, de la siguiente manera:

```
Command > listPrinters
```

```
boardPrinter : prints the game formatted as a board of dimension: 8x9
serializer   : prints the game as plain text
```

## 4. Manejo de excepciones

El tratamiento de excepciones en Java resulta muy útil para controlar determinadas situaciones del juego en tiempo de ejecución como, por ejemplo, mostrar información relevante al usuario sobre la ejecución de un comando. En la práctica anterior, cada comando invocaba el método correspondiente – de la clase *Controller* o de la clase *Game* – para poder llevar a cabo las operaciones necesarias. Por ejemplo, en el caso de no poder disparar el misil, se devolvía un valor de tipo `boolean` con valor `false`. De esta forma, el juego podía controlar que el misil no se había añadido, pero no se indicaba el motivo exacto.

En esta práctica vamos a trabajar con el manejo de excepciones, de forma que cada clase pueda lanzar y procesar determinadas excepciones para tratar determinadas situaciones durante el juego. En algunos casos, estas situaciones consistirán únicamente en proporcionar un mensaje al usuario, mientras que en otras su tratamiento será más complejo. Cabe destacar que en esta sección no se van a tratar las excepciones relativas a los ficheros, las cuales serán explicadas en detalle en la sección siguiente.

Inicialmente se van a manejar las excepciones lanzadas por el sistema, es decir, aquellas que no son creadas ni lanzadas por el usuario. Al menos, debe tratarse las siguiente excepción:

- `NumberFormatException`, que será lanzada cuando ocurra un error al transformar un número de formato `String` a formato `int`.

Además, se deberán crear dos excepciones: *CommandParseException* y *CommandExecuteException*. La primera de ellas tratará los errores producidos al parsear un comando, es decir, aquéllos producidos durante la ejecución del método *parse*, tales como “comando desconocido” o “parámetros incorrectos”. La segunda se utilizará para tratar las situaciones de error al ejecutar el método *execute* de un comando como, por ejemplo, que la nave del usuario no puede desplazarse en la dirección introducida por el usuario.

### 4.1. Implementación

Una de las principales modificaciones que realizaremos al incluir el manejo de excepciones en el juego consistirá en eliminar la comunicación directa entre los comandos y el controlador. De esta forma, no será necesario que cada comando indique al controlador cuándo se ha producido un error o cuándo se debe actualizar el tablero, sino que bastará con controlar las excepciones que puedan producirse durante la ejecución del comando. 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 serán mucho más descriptivos que en la práctica anterior. Básicamente, los cambios a realizar serán los siguientes:

1. El controlador deberá poder capturar las excepciones lanzadas por los métodos *execute* y *parse* de la clase *Command*.
2. Incluir el tratamiento de excepciones del sistema en las clases correspondientes para que puedan ser capturadas por el controlador e imprima por pantalla el mensaje correspondiente.

Ahora todos los mensajes se imprimen desde el método *run* del controlador<sup>4</sup> cuyo cuerpo se asemejará al código siguiente:

---

<sup>4</sup>Salvo los mensajes que indican que se ha guardado / cargado el juego. (Ver la siguiente sección)

```

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");
    }
}

```

## 4.2. Ejemplos de ejecución

En esta sección se muestran algunos ejemplos que tratan las excepciones anteriormente descritas:

La ejecución con un número incorrecto de parámetros (0 o más que 2) produce el mensaje siguiente:

```
Usage: Main <EASY|HARD|INSANE> [seed]
```

La ejecución con el parámetro *difficult* produce el mensaje siguiente:

```
Usage: Main <EASY|HARD|INSANE> [seed]: level must be one of: EASY, HARD, INSANE
```

La ejecución con los parámetros *easy XL* produce el mensaje siguiente (después de capturar una *RuntimeException*):

```
"Usage: Main <EASY|HARD|INSANE> [seed]: the seed must be a number"
```

Ahora mostramos una ejecución con parámetros de programa válidos (tales como *insane* o *easy -1* o *hard 0*):

## 5. Ficheros

Seguidamente, añadiremos una nueva funcionalidad que consistirá en poder guardar y cargar partidas almacenadas en ficheros de texto. Para permitir guardar el estado de una partida en un fichero, así como cargar el estado de una partida previamente guardada, se van a crear dos comandos nuevos, *SaveCommand* y *LoadCommand*, tal que:

- `save fileName` guardará el estado actual de la partida en el fichero *fileName*. Hay que tener en cuenta que *fileName* es 'case-sensitive' y, por lo tanto, distingue entre mayúsculas y minúsculas.
- `load fileName` cargará la partida almacenada en el fichero *fileName*.

Se debe tener en cuenta que durante una misma partida se pueden guardar varios estados del juego, es decir, que se debe poder permitir al usuario guardar varios estados de la partida durante el transcurso de la misma utilizando, o no, ficheros diferentes. En el caso de que ya exista un fichero con el mismo nombre, se sobrescribirá el mismo.

### 5.1. Implementación

Vamos a utilizar la funcionalidad de *serializar* que hemos descrito en el apartado anterior para guardar la información en un fichero.

El manejo de ficheros implica el uso de nuevos tipos de excepciones, tanto del sistema, como las definidas por el usuario. Por ello, se deberá crear una nueva excepción `FileContentsException` y lanzarla en caso de detectar un problema en el contenido del fichero, como por ejemplo, leer un *level* desconocido.

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, 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`.

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

- El método `execute()` de la clase `SaveCommand` debe:
  - Comprobar que el nombre del fichero proporcionado por el usuario es un nombre válido, donde esta noción depende del sistema operativo.
  - Intentar abrir el fichero con este nombre en modo escritura. Para simplificar la tarea, si el fichero ya existe, se sobrescribirá.
  - La extensión del fichero, que será siempre `‘.dat’`, la añadirá automáticamente el programa. Así, el usuario únicamente deberá **proporcionar el nombre sin extensión**.
- A continuación, el método `execute()` de la clase `SaveCommand`, utilizará la funcionalidad de *serializar* que hemos programado en la sección anterior para generar un `String` con los datos del juego y los volcará en el fichero.
- 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 poder cargar el estado de una partida, deberán tenerse en cuenta las siguientes consideraciones:

- 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 en el sistema de ficheros.
  - Intentar abrir el fichero con este nombre en modo lectura.
  - Leer la cabecera, que consta del mensaje `“— Space Invaders v2.0 —”`. En un contexto real, probablemente también aceptaría cabeceras de algunas versiones anteriores.
- A continuación, el método `execute()` de la clase `LoadCommand` invocará a un método llamado `load()` de la clase `Game`, que a su vez llamará – por ejemplo – a un método `load()` de la clase `GameObjectBoard`.

- Cada objeto de juego debe ser responsable de *parsearse* a sí mismo. Así que cuando recorramos las entradas del fichero desde la clase **Board**, crearemos los objetos necesarios y les pasaremos la información para que inicialicen sus atributos.
- 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 re-empaquetarla en un *CommandExecuteException*.

Hay circunstancias en las que es buena práctica recoger excepciones de bajo nivel para a continuación lanzar una excepción de alto nivel<sup>5</sup>.

- 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>".

En el apéndice proporcionamos el código de un método para comprobar que una cadena de caracteres constituye un nombre de fichero válido, así como un método para comprobar que un fichero con este nombre existe en el sistema de ficheros y un método para comprobar que un fichero con este nombre puede abrirse en lectura.

## 5.2. Mejorar la calidad del código

Para mejorar la calidad del código, se deben tener en cuenta las siguientes observaciones:

Al cargar un fichero con datos inválidos, la aplicación no debería parar su ejecución (o bloquearse) y tampoco debería quedarse en un estado incoherente. Para evitar que se produzcan estados incoherentes, el método `load` de *Game* debería almacenar los datos del game que se quieren cargar en una copia de seguridad (en memoria) antes de empezar a cargarlos. De este modo, cuando se produzca una excepción debido a que el contenido del fichero sea incorrecto, en el método `load` de *Game*, se podrá recoger esta excepción, reponer el estado anterior del game a partir de la copia de seguridad y a continuación relanzar la misma excepción (para que se recoja más arriba en la pila de llamadas).

---

<sup>5</sup>Tal vez habrás visto ejemplos de mala práctica a este respecto, en forma de aplicaciones de comercio electrónico en web que imprimen para el usuario un mensaje de tipo "SQL error..."; al usuario no le importa este nivel de detalle de implementación.

## 6. Anexos a la Práctica 2

```
% !TEX encoding = IsoLatin9
package /* nombre de paquete */;

import /* clases importadas */

public class Game implements IPlayerController{
    public final static int DIM_X = 9;
    public final static int DIM_Y = 8;

    private int currentCycle;
    private Random rand;
    private Level level;

    GameObjectBoard board;

    private UCMSHIP player;

    private boolean doExit;
    private BoardInitializer initializer ;

    public Game (Level level, Random random){
        this.rand = random;
        this.level = level;
        initializer = new BoardInitializer();
        initGame();
    }

    public void initGame () {
        currentCycle = 0;
        board = initializer .initialize (this, level );
        player = new UCMSHIP(this, DIM_X / 2, DIM_Y - 1);
        board.add(player);
    }

    public Random getRandom() {
        return rand;
    }

    public Level getLevel() {
        return level;
    }

    public void reset() {
        initGame();
    }

    public void addObject(GameObject object) {
        board.add(object);
    }

    public String positionToString( /* coordenadas */ ) {
        return board.toString( /* coordenadas */ );
    }

    public boolean isFinished() {
        return playerWin() || aliensWin() || doExit;
    }

    public boolean aliensWin() {
```

```
        return !player.isAlive () || AlienShip.haveLanded();
    }

    private boolean playerWin () {
        return AlienShip.allDead();
    }

    public void update() {
        board.computerAction();
        board.update();
        currentCycle += 1;
    }

    public boolean isOnBoard( /* coordenadas */ ) {
        return /* condición de rango sobre las coordenadas */ ;
    }

    public void exit() {
        doExit = true;
    }

    public String infoToString() {
        return /* cadena estado—juego para imprimir junto con el tablero */;
    }

    public String getWinnerMessage () {
        if (playerWin()) return "Player win!";
        else if (aliensWin()) return "Aliens win!";
        else if (doExit) return "Player exits the game";
        else return "This should not happen";
    }

    // TODO implementar los métodos del interfaz IPlayerController
}
```



```
package /* nombre de paquete */;

import /* clases importadas */

public class BoardInitializer {

    private Level level;
    private GameObjectBoard board;
    private Game game;

    public GameObjectBoard initialize(Game game, Level level) {
        this.level = level;
        this.game = game;
        board = new GameObjectBoard(Game.DIM_X, Game.DIM_Y);

        initializeOvni ();
        initializeRegularAliens ();
        initializeDestroyerAliens ();
        return board;
    }

    private void initializeOvni () {
        // TODO implement
    }

    private void initializeRegularAliens () {
        // TODO implement
    }

    private void initializeDestroyerAliens () {
        // TODO implement
    }
}
```

```
package /* nombre de paquete */;

public interface IPlayerController {

    // PLAYER ACTIONS
    public boolean move (int numCells);
    public boolean shootLaser();
    public boolean shockWave();

    // CALLBACKS
    public void receivePoints(int points);
    public void enableShockWave();
    public void enableMissile();
}
```

```
package /* nombre de paquete */

import /* clases importadas */

public abstract class GameObject implements IAttack {
    protected /* coordenadas */;
    protected int live;
    protected Game game;

    public GameObject( Game game, /* coordenadas iniciales */ , int live) {
        /* almacenar coordenadas iniciales */
        this.game = game;
        this.live = live;
    }

    /* mÃ©todos que devuelven el valor de las coordenadas */

    public boolean isAlive() {
        return this.live > 0;
    }

    public int getLive() {
        return this.live;
    }

    public boolean isOnPosition( /* otras coordenadas */ ) {
        return /* coordenadas = otras coordenadas */ ;
    }

    public void getDamage (int damage) {
        this.live = damage >= this.live ? 0 : this.live - damage;
    }

    public boolean isOut() {
        return !game.isOnBoard( /* coordenadas */ );
    }

    public abstract void computerAction();
    public abstract void onDelete();
    public abstract void move();
    public abstract String toString();
}
```

```
package /* nombre de paquete */;

public interface IAttack {
    default boolean performAttack(GameObject other) {return false;};

    default boolean receiveMissileAttack(int damage) {return false;};
    default boolean receiveBombAttack(int damage) {return false;};
    default boolean receiveShockWaveAttack(int damage) {return false;};
}
```

```
package /* nombre de paquete */;

import /* clases importadas */

public interface IExecuteRandomActions {

    static boolean canGenerateRandomOvni(Game game){
        return game.getRandom().nextDouble() < game.getLevel().getOvniFrequency();
    }

    static boolean canGenerateRandomBomb(Game game){
        return game.getRandom().nextDouble() < game.getLevel().getShootFrequency();
    }

}
```

```
package /* nombre de paquete */

import /* clases importadas */

public class GameObjectBoard {
    private GameObject[] objects;
    private int currentObjects;

    public GameObjectBoard (int width, int height) {
        // TODO implement
    }

    private int getCurrentObjects () {
        // TODO implement
    }

    public void add (GameObject object) {
        // TODO implement
    }

    private GameObject getObjectInPosition ( /* coordenadas */ ) {
        // TODO implement
    }

    private int getIndex( /* coordenadas */ ) {
        // TODO implement
    }

    private void remove (GameObject object) {
        // TODO implement
    }

    public void update() {
        // TODO implement
    }

    private void checkAttacks(GameObject object) {
        // TODO implement
    }

    public void computerAction() {
        // TODO implement
    }

    private void removeDead() {
        // TODO implement
    }

    public String toString( /* coordenadas */ ) {
        // TODO implement
    }
}
```

```
package /* nombre de paquete */ ;

public enum Level {

    EASY(4, 2, 0.2, 3, 0.5, 1),
    HARD(8, 4, 0.3, 2, 0.2, 2),
    INSANE(12, 4, 0.5, 1, 0.1, 3);

    private int numRegularAliens;
    private int numDestroyerAliens;
    private int numCyclesToMoveOneCell;
    private double ovniFrequency;
    private double shootFrequency;
    private int numRowsOfRegularAliens;
    private int turnExplodeFrequency = 0,05; // actualmente no depende del nivel

    private Level(
        int numRegularAliens,
        int numDestroyerAliens,
        double shootFrequency,
        int numCyclesToMoveOneCell,
        double ovniFrequency,
        int numRowsOfRegularAliens)
    {
        this.numRegularAliens = numRegularAliens;
        this.numDestroyerAliens = numDestroyerAliens;
        this.shootFrequency = shootFrequency;
        this.numCyclesToMoveOneCell = numCyclesToMoveOneCell;
        this.ovniFrequency = ovniFrequency;
        this.numRowsOfRegularAliens = numRowsOfRegularAliens;
    }

    public int getNumRegularAliens() {
        return numRegularAliens;
    }

    public int getNumDestroyerAliens() {
        return numDestroyerAliens;
    }

    public double getShootFrequency() {
        return shootFrequency;
    }

    public int getNumCyclesToMoveOneCell() {
        return numCyclesToMoveOneCell;
    }

    public double getOvniFrequency() {
        return ovniFrequency;
    }

    public int getNumRowsOfRegularAliens() {
        return numRowsOfRegularAliens;
    }
}
```

```
}

public int getNumRegularAliensPerRow() {
    return numRegularAliens / numRowsOfRegularAliens;
}

public int getNumDestroyerAliensPerRow() {
    return getNumDestroyerAliens();
}

public static Level parse(String cadenaEntrada) {
    for (Level level : Level.values())
        if (level.name().equalsIgnoreCase(cadenaEntrada))
            return level;
    return EASY;
}

public double getTurnExplodeFrequency(){
    return turnExplodeFrequency;
}
}
```



```
package pr2.view;

import pr2.game.Game;

public interface GamePrinter {
    String toString(Game game);
    public GamePrinter parse(String name);
    public String helpText();
}
```

```
package pr2.view;

import pr2.game.Game;

public class PrinterGenerator {
    private static GamePrinter[] availablePrinters = {
        new BoardPrinter(Game.DIM_X, Game.DIM_Y),
        new Serializer(),
    };

    public static GamePrinter parse(String name) {
        GamePrinter printer = null;
        for (GamePrinter p: availablePrinters) {
            printer = p.parse(name);
            if (printer != null) break;
        }
        return printer;
    }

    public static String printerHelp() {
        StringBuilder printers = new StringBuilder();
        for (GamePrinter p: availablePrinters)
            printers.append(p.helpText());
        return printers.toString();
    }
}
```

```
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[] words;
    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 attribute
    contents
    String contentString = line.substring(prefix.length()+1).trim();

    // 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 attribute with empty contents is valid ; use a zero-length array to store its
        words = new String[0];
    }

    return words;
}
```