Práctica 2 - Parte I: Lemmings Refactored

Objetivos: Herencia, polimorfismo, clases abstractas e interfaces.

Preguntas frecuentes: Como es habitual (y normal), que tengáis dudas, las iremos recopilando en este documento de preguntas frecuentes (../faq.md). Para saber los últimos cambios que se han introducido puedes consultar la historia del documento (https://github.com/informaticaucm-TPI/2324-SpaceInvaders-SOLUCION/commits/main/enunciados/faq.md).

Introducción

Esta práctica consiste, fundamentalmente, en aplicar los mecanismos que ofrece la POO para mejorar el código desarrollado hasta ahora en la Práctica 1. En particular, incluiremos las siguientes mejoras:

- En la *Parte I* de la Práctica 2 refactorizaremos[1] el código de la Práctica 1 (../practica1/practica1.md), preparándolo así para la *Parte II*. Al finalizar la refactorización, la práctica debe pasar los mismos test que se pasaron en la Práctica 1.

  - Modificaremos parte del controlador, distribuyendo su funcionalidad entre un conjunto de clases, mejor estructuradas, de cara a facilitar las extensiones posteriores.

  - Vamos a hacer uso de la herencia para reorganizar los objetos del juego. Como hemos visto, hay mucho código repetido en los distintos tipos de objetos. Por ello, vamos a crear una jerarquía de clases que nos permita extender más fácilmente la funcionalidad del 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 almacenar todos los objetos de juego.

- En la *Parte II*, una vez refactorizada la práctica, añadiremos nuevos objetos y nuevos comandos al juego, de una forma segura, ordenada y fiable, gracias a la nueva estructura del código.

Los cambios anteriores 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 demos, buscando modificar la menor cantidad de código posible cuando la ampliemos.

Refactorización de la solución de la práctica anterior

Patrón *Command*

En la práctica anterior el usuario podía realizar varias acciones: pedir ayuda, resetear el juego, actualizarlo, etc. El objetivo técnico es poder añadir nuevas acciones sin tener que modificar código ajeno a la nueva acción. Para ello, vamos a introducir el patrón de diseño *Command*[2], que es especialmente adecuado para este tipo de situaciones. La idea general es encapsular cada acción del usuario en su propia clase. Cada acción se corresponderá con un comando, de tal manera que el comportamiento de un comando quedará completamente aislado del resto.

En el patrón *Command* van a intervenir las siguientes entidades, las cuales iremos explicando en varios pasos, a medida que vayamos profundizando en los detalles:

- La clase `Command` es una clase abstracta que encapsula la funcionalidad común de todos los comandos concretos. Tiene atributos para almacenar el nombre (`name`), el atajo (`shortcut`), sus detalles (`details`) y su ayuda (`help`). Esos atributos son inicializados en su constructora y se cuenta con métodos *getters* para ellos.

- La clase `NoParamsCommand` es una clase abstracta que hereda de `Command` y que sirve de base para todos los comandos que no tienen parámetros (de momento todos). Más adelante añadiremos comandos que sí tienen parámetros y que por lo tanto no heredarán de `NoParamsCommand`.

- Los comandos concretos se corresponden con las acciones del usuario: `HelpCommand`, `ExitCommand`...

- Cada acción va a tener su propia clase y cada comando tendrá tres métodos básicos:

    - `protected boolean matchCommand(String)` comprueba si una acción introducida por teclado se corresponde con la del comando.
    - `public Command parse(String[])` recibe como parámetro un array de *strings*, que en esta práctica secorresponden con la entrada proporcionada por el usuario. En caso de que el array de *strings* encaje con el commando actual devuelve una instancia del comando; si no devolverá `null`.
    - `public void execute(Game, GameView)` ejecuta la acción del comando, modificando el juego y pidiendo a la vista que se actualice en los casos necesarios.

- La clase `Controller`, correspondiente al controlador, va a quedar muy reducida pues, como veremos más adelante, su funcionalidad quedará delegada en los comandos concretos.

En la práctica anterior, para saber qué comando se ejecutaba, el **bucle de Juego**, implementado mediante el método `run()` del controlador, contenía un `switch` (o una serie de `if` anidados) cuyas opciones se correspondían con los diferentes comandos.

En la nueva versión, el método `run()` del controlador va a tener, más o menos, el siguiente aspecto. Tu código no tiene que ser exactamente igual, pero lo importante es que veas que se asemeja a esta propuesta.

```
while (!game.isFinished()) {

    String[] userWords = view.prompt();
    Command command = CommandGenerator.parse(userWords);

    if (command != null)
            command.execute(game, view);
    else
        view.showError(Messages.UNKNOWN_COMMAND.formatted(words[0]));
}
```

Mientras que la partida no finalice, en el bucle leemos una acción de la consola, la analizamos sintácticamente para obtener el comando asociado y lo ejecutamos. La ejecución del comando decidirá en particular si tiene que volver a mostrar o no el estado del juego. Además, si no se ha podido parsear la entrada dada por el usuario mostramos el mensaje de error de `Messages.UNKNOWN_COMMAND`.

En el bucle anterior, el momento clave se corresponde con la línea de código:

```
Command command = CommandGenerator.parse(userWords);
```

El controlador solo maneja comandos abstractos, por lo que no sabe qué comando concreto se ejecutará, ni qué hará exactamente el comando. Este es el mecanismo clave que nos facilitará la tarea de añadir nuevos comandos concretos.

El método `parse(String[])` de la clase `CommandGenerator` es un método estático, encargado de encontrar el comando concreto asociado a la entrada del usuario. Para ello, la clase `CommandGenerator` mantiene una lista `AVAILABLE_COMMANDS` con los comandos disponibles. Este método recorre la lista de comandos para determinar, llamando al método `parse(String[])` de cada comando, con cuál se corresponde la entrada del usuario. Cuando lo encuentra, como se ha comentado anteriormente, el `parse` del commando concreto procesa los posibles parámetros, crea una instancia de ese mismo tipo de comando y lo devuelve al controlador.

El esqueleto del código de `CommandGenerator` es:

```
public class CommandGenerator {

    private static final List<Command> AVAILABLE_COMMANDS = Arrays.asList(
        new UpdateCommand(),
        new ResetCommand(),
        new HelpCommand(),
        new ExitCommand(),
        // ...
    );
}
```

El atributo `AVAILABLE_COMMANDS` se usa en los siguientes métodos de `CommandGenerator`:

- El método

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

  invoca el método `parse` de cada subclase de `Command`, tal y como se ha explicado anteriormente.

- El método

  ```
  public static String commandHelp()
  ```

  tiene una estructura similar al método anterior, pero invoca el método `helpText()` de cada subclase de `Command`. A su vez, es invocado por el método `execute` de la clase `HelpCommand`.

Después de recibir un `Command`, el controlador simplemente lo ejecutará usando `game` y `gameView` como parámetros, pues son parte del controlador y se comunican con ambos.

Como decíamos, todos los comandos tienen una serie de información asociada, como el nombre, atajo, detalle, etc. Por ejemplo, el comando concreto `HelpCommand` define las siguientes constantes e invoca en su constructor al constructor `super` de la siguiente forma:

```
public class HelpCommand extends NoParamsCommand {

	private static final String NAME = Messages.COMMAND_HELP_NAME;
	private static final String SHORTCUT = Messages.COMMAND_HELP_SHORTCUT;
	private static final String DETAILS = Messages.COMMAND_HELP_DETAILS;
	private static final String HELP = Messages.COMMAND_HELP_HELP;

	public HelpCommand(){
		super(NAME, SHORTCUT, DETAILS, HELP);
	}

    // Implementación de execute
}
```

Como hemos indicado anteriormente, todos los comandos heredan de la clase `Command`. La clase `Command` es abstracta, por lo que son los comandos concretos los que implementan su funcionalidad:

- El método `execute` realiza la acción sobre *game* y utiliza *gameView* para volver a dibujar el estado del juego (si procede). Más adelante puede ocurrir que la ejecución de un comando no tenga éxito, en cuyo caso se puede usar *gameView* para mostrar un mensaje de error.

- El método `parse(String[])` cuando tiene éxito, los argumentos corresponden con el commando, devuelve una instancia del comando concreto; si no devuelve `null`. Como cada comando procesa sus propios parámetros, este método devolverá `this` o creará una nueva instancia de la misma clase en caso de que el comando tenga atributos de estado que hagan que su comportamiento sea distinto para cada instancia de la clase.

La clase abstracta `NoParamsCommand`, que hereda de `Command`, sirve para representar a todos los comandos que no necesitan parámetros, como ***help*** o ***reset***, y de la que heredarán, por lo tanto, comandos como `HelpCommand` o `ResetCommand`. Esta clase puede implementar el método `parse` porque todos esos comandos se *parsean* igual: basta comprobar que el usuario solamente ha introducido una palabra que coincide con el nombre o la abreviatura del comando, en cuyo caso se puede devolver `this`. Obviamente, la implementación de `execute` se tiene que posponer a los comandos concretos, de modo que la clase `NoParamsCommand` tiene que ser abstracta.

Fíjate también que para los comandos con parámetros no sería correcto que su método `parse` devuelva `this`, sino que es necesario devolver un nuevo comando del tipo correspondiente.[3]

Herencia y polimorfismo

Una de las partes más frustrantes y propensa a errores de la primera práctica ha sido tener que replicar código en los objetos del juego y en las listas de objetos. Esta incomodidad la vamos a resolver utilizando el mecanismo por antonomasia de la programación orientada a objetos: la **herencia**.

Con el patrón `Command` hemos buscado poder introducir nuevos comandos sin tener que cambiar el código del controlador. Análogamente, queremos poder añadir nuevos objetos de juego sin tener que modificar el resto del código. La clave para ello es que `Game` no maneje objetos específicos, sino que maneje objetos de una entidad abstracta que vamos a llamar `GameObject`. El resto de objetos del juego heredarán de esta entidad. Como todos los elementos del juego van a ser `GameObject`, compartirán la mayoría de los atributos y métodos, y cada uno de los objetos de juego concretos será el encargado de implementar su propio comportamiento.

Todos los `GameObject` tienen al menos un atributo para almacenar su posición en el juego y otro booleano que indica si el objeto está vivo o no, y métodos auxiliares para manipular esa posición o para saber si el objeto está vivo o no. Por último, necesitaremos el siguiente método:

```
public void update()
```

para que cada objeto se actualice en función de su estado y de su contexto. Es normal que en objetos sencillos la implementación de este método sea vacía, como ocurre con la clase `Wall`.

`GameObject` será la clase base en la jerarquía de clases de los objetos del juego:

- 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 las clases `Lemming`, `Wall` y `ExitDoor`.

Contenedor de objetos del juego

Ahora que hemos conseguido que todos los objetos presentes en el tablero pertenezcan a (alguna subclase de) la clase `GameObject`, podemos ocuparnos de refactorizar el código de las listas. Al igual que en la práctica anterior, usaremos la clase `GameObjectContainer`. Sin embargo, en vez de usar varias listas, una para cada tipo de objeto, bastará que `GameObjectContainer` gestione una única lista con elementos de tipo `GameObject`. Por simplicidad, vamos a usar un `ArrayList` con elementos de tipo `GameObject`:

```
public class GameObjectContainer {

        private List<GameObject> gameObjects;

        public GameObjectContainer() {
                gameObjects = new ArrayList<>();
        }
    //...
}
```

En el contenedor manejaremos abstracciones de los objetos, por lo que no podemos (¡ni debemos!) distinguir quién es quién.

Por último, es muy importante que los detalles de la implementación del `GameObjectContainer` sean privados. Eso permite cambiar la implementación (el tipo de colección) sin tener que modificar código en el resto de la práctica.

Interfaces de `Game`

En la Práctica 1, la clase `Game` se utilizaba desde tres contextos diferentes, correspondientes a los tres tipos de referencias de `Game` que se mantenían en otros puntos del código:

- En `Controller` se mantiene una referencia a `Game` para poder enviar al juego las instrucciones del usuario (ahora a través del método `execute(Game,GameView)` de los comandos). En este contexto, se usan métodos como `update()`, `isFinished()` o `reset()`.
- En `GameView` también se mantiene una referencia a `Game` para solicitar los datos que se deben mostrar. Se utilizan métodos como `getCycle()`, `numLemmingsInBoard()` o `positionToString(int, int)`.
- Desde los objetos del juego también se mantiene una referencia a `Game`, en este caso para comunicarle al juego aquello que tiene que ver con las interacciones del objeto con su entorno. Se usan métodos como `isInAir(Position)` o `lemmingArrived()`. A menudo, a estos métodos se les llama *callbacks* (acciones de retorno).

Aunque exista esa clasificación de métodos en `Game`, nada prohíbe que, por ejemplo, un `GameObject` invoque a `reset`, o que desde `GameView` se invoque, por ejemplo, a `lemmingArrived`. Al fin y al cabo, todos ellos son métodos públicos de `Game` y pueden utilizarse desde cualquier objeto que tenga una referencia a `Game`.

Para resolver este problema podemos usar interfaces. En primer lugar, vamos a crear una interfaz distinta para cada uno de estos tres contextos. Cada interfaz va a proporcionar una ***vista parcial*** de `Game` con solo algunos de sus métodos. En concreto, vamos a definir `GameModel` para `Controller`, `GameStatus` para `GameView` y `GameWorld` para `GameObject`.

Por ejemplo:

```
public interface GameModel {

        public boolean isFinished();
        public void update();
        public void reset();
        // ...
}
```

Una vez definidos los interfaces, haremos que `Game` los implemente:

```
public class Game implements GameModel, GameStatus, GameWorld {

        // ...
        private GameObjectContainer container;
        private int nLevel;
        // ...

        // Métodos de GameModel
        // ...
        // Métodos de GameWorld
        // ...
        // Métodos de GameStatus
        // ...
        // Otros métodos
        // ...
}
```

Por último, en cada uno de los contextos en los que se usaba `Game` reemplazamos el tipo de la referencia por el interfaz correspondiente. Por ejemplo, en `Controller` tendremos una referencia a un `GameModel`. Del mismo modo, reemplazamos el tipo del parámetro de `execute` de `Command` para que admita un `GameModel` y no un `Game`:

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

Pruebas

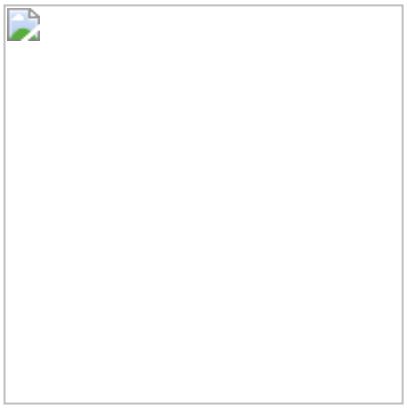
Recuerda que, una vez terminada la refactorización, la práctica debe funcionar exactamente igual que en la versión anterior y debe pasar los mismos test, aunque en la implementación tendremos muchas más clases.

Así conseguimos dejar preparada la estructura para añadir fácilmente nuevos comandos y objetos de juego en la *Parte II* de esta práctica.

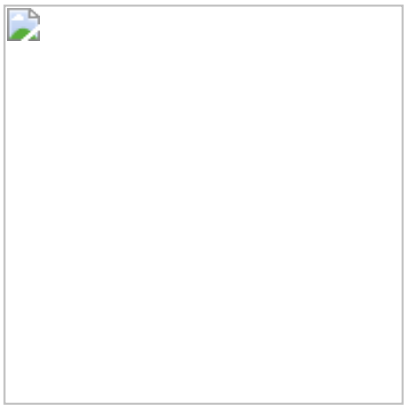
Para simplificar las pruebas, vamos a «abusar» del soporte de JUnit (https://junit.org/) dentro de Eclipse, lo cual facilitará nuestras pruebas de comparación de la salida de nuestro programa. JUnit es un *framework* para la realización de pruebas automatizadas al código Java de tu aplicación. Seguramente verás y utilizarás JUnit, o análogo, en otras asignaturas de la carrera.

Como parte de la plantilla de la práctica, se incluye la clase `tp1.Tests`, la cual es una clase de pruebas JUnit. Esta clase contiene una prueba para cada uno de los casos de prueba de la Práctica 1.

Antes de poder ejecutar las pruebas que incluye, tenemos que añadir JUnit a nuestro proyecto. Para ello, tenemos que ir a las propiedades del proyecto *Project* > *Properties*, seleccionar *Java Build Path* e ir a la pestaña *Libraries*. Allí, con *Classpath* seleccionado (no *ModulePath*), pulsamos en el botón *Add Library...*

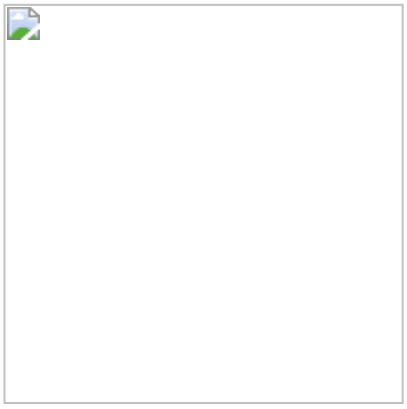


En la nueva ventana seleccionamos *JUnit* y pulsamos en el botón *Finish*



Al volver a la ventana de las propiedades del proyecto, pulsamos en el botón *Apply and Close*.

Si lo hemos configurado correctamente, al pulsar con el botón derecho del ratón sobre el fichero `Tests.java` e ir al menú *Run As*, debería aparecer la opción *JUnit Test*.



Si ejecutamos las pruebas, Eclipse mostrará una vista en la que podremos ver el resultado de las pruebas y lanzar las que hayan fallado de manera individualizada o todas a la vez. **Recuerda** que utilizamos las pruebas JUnit simplemente para comparar la salida de tu programa con la salida esperada. Si quieres ver los detalles en el caso de que no se produzca concordancia, tendrás que aplicar el mismo procedimiento que en la Práctica 1.

Fallan las pruebas JUnit

Todas las pruebas JUnit tienen éxito

---

1. Refactorizar consiste en cambiar la estructura del código sin cambiar su funcionalidad. Lo que se suele buscar con ello es mejorar el código.↩

2. Lo que vamos a ver en esta sección no es el patrón *Command* de manera rigurosa, sino una adaptación de éste a las necesidades de la práctica.↩

3. No sería correcto porque el código sería *frágil*. Devolver `this` significa devolver siempre el objeto que está almacenado en el array `AVAILABLE_COMMANDS`, cambiando previamente el valor de los atributos si se trata de un comando con parámetros. Es decir, supone que solo puede existir un objeto de esta subclase de `Command` en el juego a la vez. Si se trata de un comando con parámetros (representado por un objeto con estado) esta suposición totalmente innecesaria podría invalidarse con una pequeña modificación de la aplicación, por ejemplo, al añadir una pila de comandos ya ejecutados para implementar un comando `undo`.↩