Práctica 2 - Parte II: Lemmings Extended

Entrega: Semana del 18 de noviembre

Objetivos: Herencia, polimorfismo, clases abstractas e interfaces.

Preguntas Frecuentes: Como es habitual que tengáis dudas (es normal) 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/2425-Lemmings/commits/main/enunciados/faq.md).

En esta práctica vamos a extender el código con nuevas funcionalidades. El principal objetivo de esta práctica será conseguir aplicar a los lemmings otros roles, de modo que puedan interactuar con su entorno de otras formas.

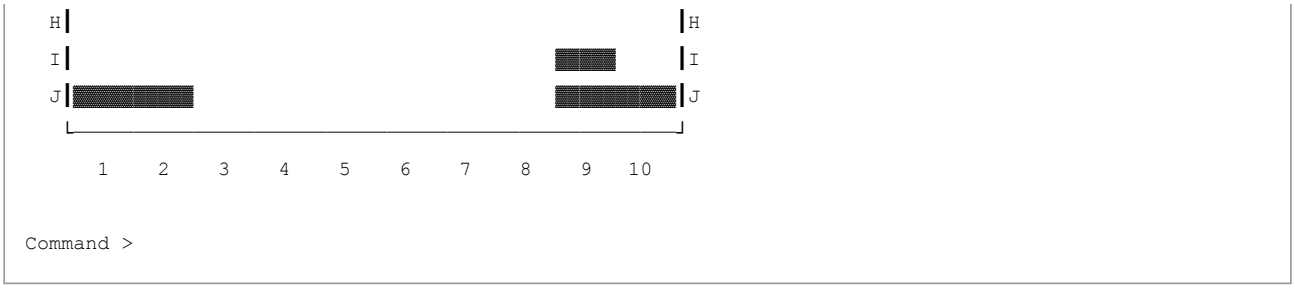
Antes de comenzar, tened en cuenta la **advertencia**:

> *La falta de encapsulación, el uso de métodos que devuelvan listas, y el uso de `instanceof` o `getClass()` tiene como consecuencia un **suspenso directo** en la práctica. Es incluso peor implementar un `instanceof` casero, por ejemplo así: cada subclase de la clase `GameObject` contiene un conjunto de métodos `esX`, uno por cada subclase X de `GameObject`; el método `esX` de la clase X devuelve `true` y los demás métodos `esX` de la clase X devuelven `false`.*

Nuevo comando y nuevo rol: SetRoleCommand y Parachuter

El objetivo que se persigue en este apartado es crear un nuevo comando para poder cambiar en tiempo de ejecución el rol de los lemmings. Para que tenga sentido dicho objetivo es necesario que creemos algún rol extra. Empezaremos añadiendo el rol de paracaidista (Parachuter). El resultado final, tras haber completado todos los subapartados, en ejecución será el siguiente:

```
Command > h
[DEBUG] Executing: h

Available commands:
   [s]et[R]ole ROLE ROW COL: sets the lemming in position (ROW,COL) to role ROLE
      [P]arachuter: Lemming falls with a parachute
      [W]alker: Lemming that walks
   [n]one | "": user does not perform any action
   [r]eset: reset the game to initial configuration
   [h]elp: print this help message
   [e]xit: exits the game

Command >
[DEBUG] Executing:

Number of cycles: 3
Lemmings in board: 4
Dead lemmings: 0
Lemmings exit door: 0 |2

      1    2    3    4    5    6    7    8    9   10
   ┌──────────────────────────────────────────────────┐
 A|                                   □              |A
 B|                                                  |B
 C|                                                  |C
 D|                         B                        |D
 E|          ▓▓▓▓▓▓▓▓▓▓▓▓▓  B                        |E
 F|                    □                ▓▓▓▓▓       |F
 G|                  ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓|G
 H|                                                  |H
 I|                                        ▓▓▓▓▓     |I
 J|▓▓▓▓▓▓▓▓▓▓  B                           ▓▓▓▓▓▓▓▓▓▓|J
   └──────────────────────────────────────────────────┘
      1    2    3    4    5    6    7    8    9   10

Command > sr Parachuter A 8
[DEBUG] Executing: sr Parachuter A 8

Number of cycles: 4
Lemmings in board: 3
Dead lemmings: 1
Lemmings exit door: 0 |2

      1    2    3    4    5    6    7    8    9   10
   ┌──────────────────────────────────────────────────┐
 A|                                                  |A
 B|                                   □  ▓▓▓▓▓▓▓▓▓▓  |B
 C|                                                  |C
 D|                                                  |D
 E|          ▓▓▓▓▓▓▓▓▓▓▓▓▓  B                        |E
 F|                    □     B          ▓▓▓▓▓       |F
 G|                  ▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓|G
```



Os proponemos una secuencia de tareas que nos llevarán a conseguir dicho objetivo.

Interfaz de roles: `LemmingRole`

Empezaremos creando un interfaz para que podamos cambiar el rol del lemming.

```
public interface LemmingRole {

    public void start( Lemming lemming );
    public void play( Lemming lemming );
    public String getIcon( Lemming lemming );
        // ...
}
```

De momento, un rol consiste en la implementación de las dos funciones `play` y `getIcon` descritas en el interfaz. La función `start` tendrá el código que se debe ejecutar al asignar el rol al lemming, si es que hiciera falta.
Una vez definido el interfaz tenemos que indicar que nuestro `WalkerRole` implementa dicho interfaz.

Para permitir que el lemming cambie de rol en tiempo de ejecución necesitamos que su atributo `role` tenga el tipo abstracto `LemmingRole`:

```
        private LemmingRole role;
```

De esta forma podremos utilizar el polimorfismo para asignar a la variable `role` tanto un objeto de tipo `WalkerRole` como `ParachuterRole` (u otros).

Nuevo rol: Paracaidista (ParachuterRole)

Ahora crearemos un nuevo rol, el de paracaidista. Este rol se podrá aplicar a cualquier lemming que se encuentre en el tablero cuando definamos el comando `SetRoleCommand`. Si el rol lo tiene un lemming que se encuentra en el aire, este lo mantendrá y no morirá aunque caiga desde una altura muy grande. Se puede considerar que el rol reduce la fuerza con la que cae el lemming, dejándola a 0. El paracaidas se podrá colocar en cualquier lemming, pero solo tendrá efecto si el lemming se encuentra en el aire. El lemming mantendrá el paracaídas abierto hasta que aterrice y seguirá bajando de uno en uno en cada actualización. Cuando aterrice se deshabilitará el rol de paracaidista y pasará a ser otra vez un caminante en la misma dirección que llevaba antes. Además, si se activa el rol de paracaidista sobre un lemming que se encuentra en el suelo el rol se deshabilitará inmediatamente y se quedará con su rol de caminante.

Para realizar esta tarea se pide incluir una nueva clase `ParachuterRole` que implemente el interfaz `LemmingRole`. La implementación del comportamiento de un lemming con dicho rol la debemos realizar el método `play` del rol, no en el lemming. Fíjate que en ese método tendrás acceso al lemming al que tengas aplicado dicho rol y por tanto, podrás consultar el estado de dicho lemming e incluso pedirle que cambie la fuerza con la que cae y que caiga. Si has programado separando en métodos el comportamiento del lemming caminante será algo simple pedirle al lemming que caiga (método `fall`). Si no te tocará dividir su comportamiento en métodos y así evitar repetir código.

Ya solo queda permitir el cambio de rol del lemming durante la ejecución. Esta tarea la realizaremos añadiendo dos métodos en la clase `Lemming`: uno que permita cambiar de rol en ejecución y otro que deshabilite el rol actual dejándolo en su rol `WalkerRole`. Dichos métodos han de ser los siguientes:

```
        public void setRole(LemmingRole role);
        public void disableRole();
```

Como aún no dispones del comando `SetRoleCommand`, una forma de realizar las pruebas consiste en crear una configuración inicial con lemmings con paracaídas. Se recomienda crear lemmings con paracaídas en el aire y en el suelo y ver que este rol se desactiva adecuadamente al llegar al suelo, de modo que el lemming sigue vivo y caminando en la misma dirección que tenía antes de abrir el paracaídas.

Factoría de roles

Ahora añadiremos una ***factoría*** de roles a nuestra práctica. Una factoría nos permite separar la lógica de la creación de un objeto del lugar en el que se crea. Dicha factoría va a venir implementada en una clase `LemmingRoleFactory` en el paquete `lemmingsRoles`, con un método

```
        public static LemmingRole parse(String input);
```

Usando la factoría, para crear un rol cuyo tipo viene dado en un string `input`, basta hacer

```
        LemmingRole role = LemmingRoleFactory.parse(input);
```

Utiliza la misma técnica empleada en `CommandGenerator` para crear dicha factoría. Al igual que allí permite nombres cortos y largos para los roles. En este caso puedes simplificar un poco el estado de cada rol, pues no es necesario tener separada la descripción de la ayuda. En esta factoría, al igual que en el `CommandGenerator`, los roles sin estado pueden devolver `this`, pero cuando generemos algún rol con estado es necesario devolver siempre una instancia de la clase(`return new XXXRole()`).

Para crear la factoría de la forma correcta necesitarás añadir algún método a la interfaz `LemmingRole`.

Nuevo comando: SetRoleCommand

Una vez realizados todos los pasos anteriores, vamos a añadir un nuevo comando al juego que permita cambiar el rol al lemming en tiempo de ejecución. Los argumentos de este parámetro serán el rol a aplicar y la posición del tablero en la que aplicar dicho rol. Como en una posición puede haber varios lemmings a la vez, el rol se aplicará al primer lemming que se encuentre en esa posición en el container y que lo admita. Sin embargo, fíjate que en el contenedor no sabemos (ni queremos saberlo) qué objeto es un lemming (todos son `GameObject`). Por eso añadiremos a `GameObject` un método para cambiar su rol, indicando además si ha tenido éxito o no:

```
        public boolean setRole(LemmingRole role);
```

Nótese que hemos cambiado el método propuesto en el apartado **Paracaidista**, devolviendo un *booleano* en vez de `void`. El booleano indicará si ha tenido éxito o no al aplicar el rol al objeto. Solo los objetos del tipo lemming, **que no tengan ya ese rol**, realizarán la tarea de cambiar el rol, el resto indicará que no tiene éxito. De esta forma será muy sencillo programar `SetRoleCommand`. Si no existe ningún objeto al que se pueda aplicar dicho rol o la posición es incorrecta el programa deberá mostrar el mensaje siguiente:

```
[ERROR] Error: SetRoleCommand error (Incorrect position or no object in that position admits that role)
```

Sin embargo, si el rol no existe el programa deberá mostrar el siguiente error:

```
[ERROR] Error: Unknown Role
```

Para implementar el método `helpText()` de este comando es posible que necesites pedir a la factoría de roles su ayuda, por lo que si no la has implementado anteriormente deberías crear un método en ésta para que te devuelva su ayuda. Fíjate en el ejemplo que hay al principio de este documento, pues seguro que te hace falta añadir algún mensaje en la clase `Messages`.

Nuevo rol y objeto: DownCaverRole y MetalWall

Ahora que tenemos nuestra factoría de roles y hemos utilizado un interfaz para generalizar los roles de juego y herencia para los objetos del juego, resulta muy sencillo extender el juego con nuevos roles y objetos. En esta ocasión vamos a añadir el rol de **excavador** que permitirá cavar en vertical eliminando la tierra que hay debajo del lemming y cayendo este una posición. Al igual que el rol *Paracaidista* este se desactivará en caso de que el lemming no pueda cavar. Desactivar el role no cuenta como un *paso* de ejecución, por lo que si se desactiva se deberá pedir al lemming que realice un paso.

Para que este rol tenga distinto comportamiento con los distintos objetos del juego añadiremos una **pared de metal** que consideramos **dura**, a diferencia de las paredes actuales que se considerán **blandas**, y no podrá ser excavada por el excavador.

Para realizar esta extensión y conseguir que el comportamiento fuera diferente con cada uno de los objetos del juego se podrían añadir nuevas funciones a cada objeto para saber si es un objeto *duro* o *blando*, pero esto obligaría de nuevo a modificar varias clases, incluida `Game`. Además, no es la técnica más adecuada porque por cada extensión habría que añadir nuevos métodos que podrían considerarse `instanceof` encubiertos.

Por lo que primeramente generalizaremos las *interacciones* entre los objetos del juego.

Generalizando las **interacciones** entre los objetos del juego

Parece razonable considerar que los lemmings pueden interaccionar con los objetos del juego. Un ejemplo claro es que el lemming sale por la puerta cuando la alcanza, aunque esta situación está implementada directamente a través de consultas ad-hoc a otros objetos (método `isExit`), no como interacciones en un contexto más general.

Otro ejemplo claro que se va a dar ahora es la interacción entre el excavador y el suelo. Dicha interacción sobre el suelo *blando* hará que este desaparezca, pero sobre el suelo *duro* no tendrá ningún efecto, salvo el de desactivar el rol de excavador del lemming.

Para modelar estas *interacciones* entre objetos vamos a usar otra interfaz, `GameItem`. Esta interfaz nos va a permitir utilizar una técnica llamada ***double-dispatch***, que permite seleccionar en tiempo de ejecución el método que implementa la interacción en función del tipo de los dos objetos involucrados en la interacción. La clase `GameObject` implementará el interfaz `GameItem` porque la idea es que todos los objetos del juego deben tener la posibilidad de interaccionar con otros mediante los métodos de esta interfaz (y *solo* mediante esos métodos).

```
public interface GameItem {
        public boolean receiveInteraction(GameItem other);

        public boolean interactWith(Lemming lemming);
        public boolean interactWith(Wall wall);
        public boolean interactWith(ExitDoor door);

        public boolean isSolid();
        public boolean isAlive();
        public boolean isExit();
        public boolean isInPosition(Position pos);
}
```

Fíjate que en el método `interactWith` estamos haciendo uso de la sobrecarga de métodos. La interacción que implementaremos no consiste en una relación simétrica, pues el método *receive* indica que hay un receptor y un emisor. Es cierto que se podría generalizar más simplemente cambiando los nombres de los métodos, pero no es necesario. Una implementación válida, por ahora, para el método `receiveInteraction` podría ser la siguiente:

```
        public boolean receiveInteraction(GameItem other) {
                return other.interactWith(this);
        }
```

Es necesario que cada subclase concreta de `GameObject` cuente con esa implementación del método `receiveInteraction`. Fíjate que no es posible subir esa implementación a `GameObject` y compartir el código. El motivo es que estamos haciendo uso de la sobrecarga del método `interactWith` mencionados anteriormente. En la clase `Wall` el tipo estático de `this` es `Wall` y por lo tanto el método invocado es `interactWith(Wall)`. Análogamente, si ese código está en la clase `MetalWall` el tipo estático de `this` es `MetalWall` y por lo tanto el método sobrecargado al que se invoca es `interactWith(MetalWall)`.[1]

A continuación tenemos que extender el concepto al `GameObjectContainer`, de tal manera que crearemos un método que realiza todas las interacciones de un objeto dado con los existentes en el contendor:

```
        public boolean receiveInteractionsFrom(GameItem obj) {...}
```

En este caso, el booleano indicará si algún objeto del contenedor ha sido modificado por dicha interacción.

Por último, es conveniente que `GameWorld` tenga también dicho método para que los objetos del juego puedan utilizarlo para generar interacciones con él. En concreto nos será útil para programar el rol `DownCaverRole`.

Detalles: DownCaverRole y MetalWall

Este nuevo rol lo implementaremos en una clase llamada `DownCaverRole`. Como se ha comentado anteriormente, el rol excavador permitirá al lemming cavar en vertical el suelo blando. El icono que utilizaremos en este caso para representar un lemming excavador será: `Messages.LEMMING_DWON_CAVER`. Cada vez que un excavador excava un bloque de tierra caerá un nivel y, por lo tanto, no morirá. Sin embargo, si no pudiera cavar se desactivaría dicho rol para volver a ser un caminante en la misma dirección que llevaba anteriormente.

El mensaje de ayuda que mostrará dicho rol es el siguiente: `[D]own[C]aver: Lemming caves downwards.` Obviamente, el nombre corto será `DC` y su nombre largo `DownCaver`.

La pared de metal la implementaremos en una clase llamada `MetalWall`. La principal diferencia con respecto a la pared normal `Wall` es que no interacciona con los objetos del juego.

Por cierto, para conseguir que los lemmings interaccionen con la `Wall` será necesario sobrescribir el método:

```
    public boolean interactWith(Wall obj){...}
```

en el lemming.

Fíjate que todos los métodos de interacción entre los lemmings y los objetos del juego pueden ser cambiados por el rol que tengan en cada momento. Por lo que el lemming debería preguntar al rol cual es el comportamiento. Para ello será necesario añadir en el interfaz `LemmingRole` los métodos:

```
        public boolean receiveInteraction(GameItem other, Lemming lemming);

        public boolean interactWith(Lemming receiver, Lemming lemming);
        public boolean interactWith(Wall wall, Lemming lemming);
        public boolean interactWith(ExitDoor door, Lemming lemming);
```

El segundo parámetro de estos métodos, de tipo `Lemming`, juega el mismo papel que el parámetro de tipo `Lemming` del método `play`, es decir, el lemming que ejerce el rol.

Hemos de tener en cuenta que la gran mayoría de lemmings no interaccionan con los elementos. Para establecer que el comportamiento general del rol con respecto a todos los objetos del juego sea el de no realizar ningún cambio es deseable crear una clase abstracta `AbstractRole` que implemente ese comportamiento por defecto, de modo que los roles concretos solo tengan que reescribir el comportamiento en los casos en los que sí haya interacción.

Una vez realizadas estas extensiones, es momento de implementar el rol `DownCaverRole`. Para saber si se ha cavado o no (es decir, se ha interaccionado con una `Wall` o no) consideraremos en `DownCaverRole` un atributo booleano `hasCaved`.

Aplicando *double-dispatch* al ExitDoor (opcional)

Para comprobar la versatilidad del ***double dispatch*** se os propone eliminar el método `isExit` de todos los objetos del juego y, por tanto, también de `Game` y de `GameObjectContainer´ e implementar en el lemming únicamente el método:

```
    public boolean interactWith(ExitDoor obj){...}
```

Si aplicáis estos cambios simples veréis que el código se simplifica, ya que desaparece mucho del código realizado con anterioridad. Además, seguro que si lo pensáis un poco veréis otros sitios donde se podría haber aplicado dicho método simplificando las interacciones entre los lemming y el `game`. Pero no es necesario que realicéis más modificaciones.

Extendiendo el commando `reset` (opcional)

En esta sección vamos a realizar una pequeña extensión de funcionalidad del comando `reset` para que no solo permita resetar el juego en ejecución, sino que también permita cargar un mapa diferente.

La extensión consiste en permitir un argumento opcional en la orden `reset` del usuario con un número de mapa a cargar. De tal forma que la ayuda nos quedará de la siguiente forma:

```
Command > h
[DEBUG] Executing: h

Available commands:
   [s]et[R]ole ROLE ROW COL: sets the lemming in position (ROW,COL) to role ROLE
      [D]own[C]aver: Lemming caves downwards
          [P]arachuter: Lemming falls with a parachute
      [W]alker: Lemming that walks
   [n]one | "": user does not perform any action
   [r]eset [numLevel]: reset the game to initial configuration if not numLevel else load the numLevel map
   [h]elp: print this help message
   [e]xit: exits the game
```

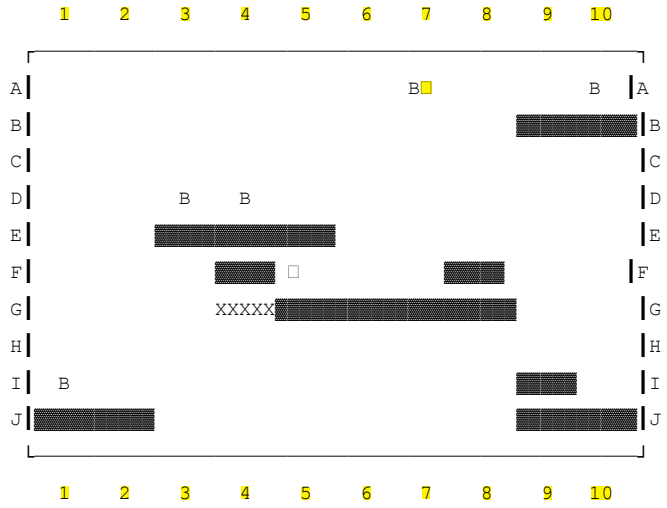
Se mantiene la funcionalidad antigua para el `reset` sin argumento y para el `reset` con número de nivel se cargará el mapa con dicho nivel. En caso de que no existiera dicho nivel solo se debería mostrar por pantalla el siguiente error:

```
[ERROR] Error: Not valid level number
```

Nuevo mundo: mapa 2

Se ha añadido en las pruebas el **mapa 2**, con pequeños añadidos con respecto a los anteriores. Este mapa se podrá al igual que los anteriores cargar desde los argumentos del juego y debería cargarse a través de la función `initGame2()`. El mapa es el siguiente:

```
Number of cycles: 0
Lemmings in board: 6
Dead lemmings: 0
Lemmings exit door: 0 |2
```



El símbolo `XXXXX` representa una pared de metal.

---

1. Existe una forma de evitar la copia de este código pero implica usar reflexión, que es otra técnica de programación más avanzada y que no veremos en este curso.↩