

---

# Práctica 2 parte II: Road Fighter Extended

---

**Fecha de entrega:** 29 Noviembre 2021, 9:00

**OBJETIVO:** Herencia, polimorfismo, clases abstractas e interfaces

## 1. Extensiones

En esta práctica vamos a extender el código con nuevas funcionalidades. Antes de comenzar recordad la advertencia:

**IMPORTANTE:** La falta de encapsulación, el uso de métodos que devuelvan listas, y el uso de “instanceof” o “getClass()” tiene como consecuencia un suspense 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`.

### 1.1. Objeto Wall, comando Shoot e interfaz `InstantAction`

**Objeto Wall.** El primer objeto que vamos a añadir es un nuevo tipo de obstáculo que llamaremos `Wall`. Su comportamiento es exactamente igual que `Obstacle` pero tiene 3 puntos de resistencia.

Al representarlo utilizaremos tres símbolos diferentes `⚡`, `⚡⚡`, `⚡⚡⚡` dependiendo de cuánta vida le queda (consulta el fichero `symbols.txt` incluido con la plantilla para ver los símbolos exactos). El `Player` va a poder destruir obstáculos disparando. Cuando destruya un `Wall` le daremos 5 coins (el obstáculo normal no le da ningún coin).

**Comando Shoot.** Para poder disparar vamos a crear el comando `Shoot`, que quita un punto de vida al primer obstáculo que está en el mismo carril que el `Player`. El usuario gasta 1 coin por disparo a un obstáculo. Además de ejecutar el disparo, los **`GameObjects` del juego también son actualizados** (los ciclos avanzan). Veamos un ejemplo:

```

Diferentes visualizaciones de un Wall

Command >
q
[DEBUG] Executing: q
Distance: 6
Coins: 10
Cycle: 3
Total obstacles: 1
Total coins: 0

=====
>          ■          |
=====
|
=====
|

=====

Command >
s
[DEBUG] Executing: s
Distance: 6
Coins: 9
Cycle: 4
Total obstacles: 1
Total coins: 0

=====
>          ■          |
=====
|
=====
|

=====

Command >
s
[DEBUG] Executing: s
Distance: 6
Coins: 8
Cycle: 5
Total obstacles: 1
Total coins: 0

=====
>          ☒          |
=====
|
=====
|

=====

Command >
s
[DEBUG] Executing: s
Distance: 6
Coins: 12
Cycle: 6
Total obstacles: 0
Total coins: 0

=====
>          |
=====
|
=====
|

=====

```

**Interfaz Collider.** El disparo sólo afecta a los obstáculos, así que para controlar cómo afecta a cada objeto, añadiremos el nuevo método `receiveShoot()` al interface `Collider`. El alcance del disparo es sólo la parte visible de la carretera.

**Interfaz InstantAction.** El comando `Shoot` no crea ningún nuevo objeto “bala” sino que se ejecuta instantáneamente como un rayo láser. Para encapsular las acciones puntuales

como esta, vamos a crear un nuevo interfaz muy sencillo al que llamaremos `InstantAction`:

```
package es.ucm.tp1.logic;

public interface InstantAction {
    void execute(Game game);
}
```

Cada acción instantánea *concreta* tendrá que implementar el método `execute` del interface.

## 1.2. Objeto Supercoin

Ahora vamos a crear un nuevo objeto que llamamos `SuperCoin` que representamos con el símbolo `$`. Cuando cogemos el `SuperCoin` nos da 1000 coins.

El `Supercoin` tiene la peculiaridad de que sólo puede haber uno por partida. Para controlar que sólo haya una instancia usaremos una variable estática. Fijaos que cuando el `Supercoin` está presente en la carretera, se muestra en la información `Supercoin is present`.

Veamos un ejemplo donde el coche pasa por encima del dólar y aumentan sus coins:

```
[DEBUG] Executing:
Distance: 26
Coins: 5
Cycle: 4
Total obstacles: 20
Total coins: 11
Supercoin is present
Elapsed Time: 0.71 s
```

		¢			☹	⌘	⌘
>		\$					
⌘	¢		>>>	¢	¢		

Command >

```
[DEBUG] Executing:
Distance: 25
Coins: 5
Cycle: 5
Total obstacles: 20
Total coins: 11
Supercoin is present
Elapsed Time: 1.66 s
```

	¢				⌘	⌘	
>	\$			☹			¢
¢		>>>	¢	¢			⌘

Command >

```
[DEBUG] Executing:
Distance: 24
Coins: 1005
Cycle: 6
Total obstacles: 20
Total coins: 11
```

Elapsed Time: 5.04 s

¢				⌘	⌘		>>>
>						¢	¢
	>>>	¢	¢			⌘	

### 1.3. Objeto Turbo

Este objeto de juego lo representamos con el símbolo >>> y hace al coche *saltar* 3 casillas cuando el coche pasa por encima del mismo. El tablero se pinta avanzando tres casillas conservando siempre el coche en la primera columna. Si el coche cae sobre otro objeto entonces no pasa nada en relación con ese nuevo objeto, es decir, no colisiona (no se choca, ni coge monedas...), además los dos objetos se quedan en la misma casilla.

Veamos un ejemplo:

Command >

[DEBUG] Executing:  
Distance: 25  
Coins: 5  
Cycle: 5  
Total obstacles: 20  
Total coins: 11  
Supercoin is present  
Elapsed Time: 1.38 s

	¢			⌘	⌘		
>	\$			⊖			¢
¢		>>>	¢	¢			⌘

Command >

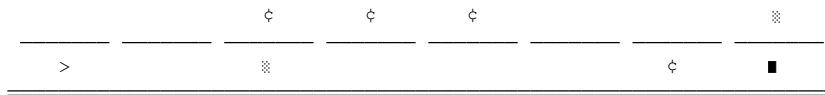
a  
[DEBUG] Executing: a  
Distance: 24  
Coins: 5  
Cycle: 5  
Total obstacles: 20  
Total coins: 11  
Supercoin is present  
Elapsed Time: 5.58 s

¢				⌘	⌘		>>>
\$			⊖			¢	¢
>	>>>	¢	¢			⌘	

Command >

[DEBUG] Executing:  
Distance: 20  
Coins: 5  
Cycle: 6  
Total obstacles: 20  
Total coins: 11  
Supercoin is present  
Elapsed Time: 6.50 s

⌘	⌘		>>>				
---	---	--	-----	--	--	--	--



Command >

### 1.4. Comando ClearCommand

Para testear los nuevos objetos vamos a crear un par de comandos para hacer *trampas*. El primero de ellos eliminará todos los objetos de la carretera (menos el coche). El ClearCommand usa la tecla 0.

### 1.5. Comando CheatCommand

Este comando sirve para añadir objetos avanzados, que son los que se listan a continuación y que detallaremos más adelante. El CheatCommand se ejecuta con las teclas numéricas [1..5]. Primero borra todos los objetos de la última columna visible y después añade un objeto en un carril aleatorio. Este comando es sólo para testear, así que para facilitarnos la tarea, utilizaremos los siguientes valores literales correspondientes a cada objeto:

1. Wall
2. Turbo
3. SuperCoin
4. Truck
5. Pedestrian

Para poder ejecutar el comando, saltando las restricciones del juego normal, añadiremos un método al GameObjectGenerator de la *Parte I*, similar a este:

```
public static void forceAdvanceObject(Game game, int id, int x) {
    GameObject o = null;
    switch (id) {
        case 1:
            o = new Wall(game, x, game.getRandomLane());
            break;
        case 2:
            o = new Turbo(game, x, game.getRandomLane());
            break;
        case 3:
            o = new SuperCoin(game, x, game.getRandomLane());
            break;
        case 4:
            o = new Truck(game, x, game.getRandomLane());
            break;
        case 5:
            o = new Pedestrian(game, x, 0);
            break;
    }
    game.forceAddObject(o);
}
```

## 1.6. Comando Wave

Vamos a crear un comando que empuja todos los objetos una casilla hacia la derecha de la carretera. El comando tiene un coste de 5 coins. Para ello, crearemos una acción instantánea igual que **Shoot**. Se ejecuta con "w", o "wave". Tenemos que tener en cuenta lo siguiente:

- Empujaremos *todos* los objetos visibles de la carretera excepto el coche.
- No podemos empujar un objeto que tenga algún otro objeto detrás. No obstante queremos maximizar los objetos que empujamos, así que analiza con cuidado cómo y en qué orden vas a actualizar los objetos.

Veamos un ejemplo:

Command >

```
[DEBUG] Executing:
Distance: 6
Coins: 5
Cycle: 4
Total obstacles: 4
Total coins: 1
```

		ç	⊗				
>	⊗						
⊗					⊗		

Command >

```
w
[DEBUG] Executing: w
Distance: 6
Coins: 0
Cycle: 4
Total obstacles: 4
Total coins: 1
```

		ç	⊗				
>		⊗					
	⊗					⊗	

## 1.7. Comando Grenade y objeto Grenade

Este comando crea un objeto **Grenade** en la posición (x, y) de la carretera. Se ejecuta con "g" o "grenade", donde los parámetros son las coordenadas X e Y de su posición. La granada tiene un coste de 3 coins y explota en 3 ciclos. Para este comando tendremos en cuenta lo siguiente:

- El comando tendrá que sobrescribir el método **parse** de **Command** para aceptar los parámetros de la posición.
- La posición X es relativa a la posición del jugador, es decir,  $X = 5$  significa que la granada se lanzará 5 posiciones por delante del jugador.
- La posición de Y es absoluta respecto a la carretera, es decir,  $Y = 0$  hace referencia al primer carril (en la parte superior) de la carretera.

- Sólo podemos añadir granadas en la parte visible de la carretera y la casilla tiene que estar vacía.
- Tendremos que crear un nuevo objeto **Grenade** que tenga una cuenta atrás. Lo representaremos con el símbolo ⌚. Además, entre corchetes, representaremos los ciclos que faltan para que explote (ver ejemplo).
- Al terminar la cuenta atrás explotará causando daño sólo a los obstáculos y muros, así que deberemos extender con un método **receiveExplosion** el interface *Collider*. El comportamiento es igual al de **receiveShoot** así que podemos reutilizar el método.

Veamos un ejemplo:

```
Command >
g 5 1
[DEBUG] Executing: g 5 1
Distance: 25
Coins: 2
Cycle: 5
Total obstacles: 20
Total coins: 11
Supercoin is present
Ellapsed Time: 30.52 s
```

	¢				⌚	⌚	
>	\$			⊕	⌚[3]		¢
¢		>>>	¢	¢			⌚

```
Command >
[DEBUG] Executing:
Distance: 24
Coins: 1002
Cycle: 6
Total obstacles: 20
Total coins: 11
Ellapsed Time: 38.18 s
```

¢				⌚	⌚		>>>
>				⌚[2]		¢	¢
	>>>	¢	¢			⌚	

```
Command >
[DEBUG] Executing:
Distance: 23
Coins: 1002
Cycle: 7
Total obstacles: 20
Total coins: 11
Ellapsed Time: 38.80 s
```

			⌚	⌚		>>>	
>		⊕	⌚[1]		¢	¢	¢
>>>	¢	¢			⌚		

```
Command >
```

```
[DEBUG] Executing:
Distance: 22
Coins: 1002
Cycle: 8
Total obstacles: 20
Total coins: 11
Elapsed Time: 39.66 s
```

---

---

					>>>		
>				☺	☺	☺	
☺	☺			⌘			←

---

---

### 1.8. Interface Buyable

Como hay varios comandos que comparten la característica de tener un coste en monedas, vamos a crear un interfaz común para ellos. En este caso vamos a crear el interfaz **Buyable** para encapsular esta funcionalidad en un método **buy()**, que le restará las monedas del coste del objeto al **Player**:

```
package supercars.control;

import supercars.logic.Game;

public interface Buyable {

    public int cost();

    public default void buy(Game game){
        // TODO add your code
    };
}
```

Todos los comandos que cuesten **Coins** tendrán que implementar este interfaz.

### 1.9. Objecto Truck

Hasta ahora todos los objetos de juego (salvo el **Player**) eran estáticos, es decir, estaban siempre fijos en su posición inicial. Vamos a crear algunos objetos, entre ellos **Truck**, que se mueven solos en cada ciclo al igual que el **Player**.

Los objetos **Truck**, que representaremos con el símbolo **←**, avanzarán una casilla hacia la izquierda en cada turno y colisionarán con el **Player** exactamente igual que un obstáculo.

Para este nuevo objeto tenemos que tener en cuenta dos cosas importantes:

- Como el camión avanza y no colisiona con los otros obstáculos, podría ocurrir que existan dos objetos en la misma posición. En este caso, se pintarán en el mismo orden en el que fueron introducidos en el juego.
- Es posible que el **Truck** se cruce con el **Player** en vez de chocar. Esta situación debe evitarse detectando la colisión, ya que hay dos avances de una casilla en sentidos opuestos. Para evitar esta situación **tenemos que comprobar las colisiones dos veces cada ciclo, una al principio del ciclo y otra tras haber movido el Player**.





Total coins: 0

```

=====
_____  @ _____ | _____
|_____> _____ | _____
|_____ | _____
=====

```

Command >

[DEBUG] Executing:  
 Distance: 5  
 Coins: 5  
 Cycle: 5  
 Total obstacles: 1  
 Total coins: 0

```

=====
_____ | _____
|_____> _____ @ _____ | _____
|_____ | _____
=====

```

Command >

[DEBUG] Executing:  
 Distance: 4  
 Coins: 5  
 Cycle: 6  
 Total obstacles: 1  
 Total coins: 0

```

=====
_____ | _____
|_____> _____ | _____
|_____ @ _____ | _____
=====

```

Command >

[DEBUG] Executing:  
 Distance: 3  
 Coins: 5  
 Cycle: 7  
 Total obstacles: 1  
 Total coins: 0

```

=====
_____ | _____
|_____> _____ | _____
|_____ | _____
=====

```

Command >

[DEBUG] Executing:  
 Distance: 3  
 Coins: 0  
 Cycle: 8  
 Total obstacles: 0  
 Total coins: 0

```

=====
_____ | _____
|_____ @ _____ | _____
|_____ | _____
=====

```

|

### 1.11. Thunder Action

Hasta ahora todos los objetos o acciones se crean al principio de la partida, o por medio de un comando del usuario. Ahora vamos a crear otros objetos sin participación del usuario, durante la partida (en *runtime*). Vamos a crear una acción instantánea **Thunder**, que es un trueno que cae aleatoriamente en la parte visible de la carretera. Si cae sobre un obstáculo entonces lo elimina. Se va a crear un **Thunder** en cada ciclo del juego, como en muchas ocasiones no dará a ningún objeto vamos a incluir una salida que muestre dónde cae. Veamos ejemplos de cómo funciona:

Command >

```
[DEBUG] Executing:
Thunder hit position: (4 , 0)
Distance: 94
Coins: 6
Cycle: 6
Total obstacles: 45
Total coins: 25
Supercoin is present
Elapsed Time: 2,77 s
```

```

=====
      ←                               ¢
=====
>      _____ ■ _____ ¢
=====
      ⌘                               ⌘
=====
```

Command >

```
[DEBUG] Executing:
Thunder hit position: (4 , 1) -> Obstacle hit
Distance: 93
Coins: 6
Cycle: 7
Total obstacles: 45
Total coins: 24
Supercoin is present
Elapsed Time: 3,27 s
```

```

=====
                               ¢
=====
>      _____ ¢ _____ ¢
=====
      ⌘                               ⌘
=====
```

Como podemos observar, **la posición del trueno es relativa a la columna donde se encuentra el jugador**, es decir, 0 siempre representa la primera columna que se pinta.

### 1.12. Factoría GameObject Generator 2.0

Vamos a desarrollar una nueva versión de la Factoría. El **GameObjectGenerator** va a ser el encargado de crear los objetos en tiempo de ejecución. Para ello, desde el **Game**, en cada ciclo llamaremos al método **GameObjectGenerator.generateRuntimeObjects(this)**;

El código de este método se muestra a continuación:

```

public static void generateRuntimeObjects(Game game) {
    // Note we use this method to create and inject new objects or actions on runtime.

    if (game.getLevel().hasAdvancedObjects()) {
        game.execute(new ThunderAction());
    }

}

```

### 1.13. Niveles del juego, Level 2.0

Vamos a modificar los niveles del juego. En el Level añadiremos un nuevo valor que será la frecuencia de aparición de los objetos avanzados. En los niveles definidos en la primera práctica, la frecuencia para los nuevos objetos será de 0. Además, añadiremos un nuevo nivel, tal y como se muestra en la Tabla 1.1.

Nivel	length	width	visibility	Obs freq	Coin freq	advObjFreq
TEST	10	3	8	0.5	0	0
EASY	30	3	8	0.5	0.5	0
HARD	100	5	6	0.7	0.3	0
ADVANCED	100	3	8	0.3	0.3	0.1

Tabla 1.1: Configuración para cada nivel de dificultad

Con estos cambios, La creación de objetos se hará así:

```

public static void generateGameObjects(Game game, Level level) {

    for(int x = game.getVisibility()/2; x < game.getRoadLength(); x++) {

        game.tryToAddObject(new Obstacle(game, x, game.getRandomLane())
            , level.obstacleFrequency());
        game.tryToAddObject(new Coin(game, x, game.getRandomLane())
            , level.coinFrequency());

        if (level.hasAdvancedObjects()) {
            game.tryToAddObject(new Wall(game, x, game.getRandomLane()), level.
                advancedObjectsFrequency());
            game.tryToAddObject(new Turbo(game, x, game.getRandomLane()), level.
                advancedObjectsFrequency());
            if (!SuperCoin.hasSuperCoin()) {
                game.tryToAddObject(new SuperCoin(game, x, game.getRandomLane())
                    , level.advancedObjectsFrequency());
            }
            game.tryToAddObject(new Truck(game, x, game.getRandomLane()), level.
                advancedObjectsFrequency());
            game.tryToAddObject(new Pedestrian(game, x, 0), level.advancedObjectsFrequency());
        }

    }

}

```

Para controlar si está permitido crear objetos avanzados, tenemos el método `hasAdvancedObjects()`, que devolverá `true` si la frecuencia de objetos avanzados es mayor que 0. Aunque, recordemos que incluso en el nivel `Test`, donde la frecuencia es 0, **podemos forzar la aparición de objetos avanzados con el comando Cheat**.

## 2. Puntualizaciones importantes

Una manera de asegurarnos de que estamos gestionando correctamente la abstracción, es evitando que el `Game` acceda a ningún objeto o acción concretos. Para ello, en vez de importar paquetes, vamos a importar sólo las clases que debemos usar. La clase `Game` sólo necesita estos imports:

```
import java.util . Random;
import supercars.control . Level;
import supercars.logic . gameobjects.GameObject;
import supercars.logic . gameobjects.Player;
```

y la clase `GameObjectContainer` sólo necesita estos:

```
import java.util . ArrayList;
import java.util . List;
import supercars.logic . gameobjects.GameObject;
```

**Objetos disponibles.** Los objetos avanzados los escribimos con mayúscula y los normales en minúscula. Estos son todos los objetos:

```
[DEBUG] Executing: i
Available objects:
[Car] the racing car
[Coin] gives 1 coin to the player
[Obstacle] hits car
[GRENADe] Explodes in 3 cycles, harming everyone around
[WALL] hard obstacle
[TURBO] pushes the car 3 columns
[SUPERCOIN] gives 1000 coins
[TRUCK] moves towards the player
[PEDESTRIAN] person crossing the road up and down
```

**Comandos disponibles.** Estos son todos los comandos:

```
[DEBUG] Executing: h
Available commands:
[h]elp: show this help
[i]nfo: prints gameobject info
[n]one | []: update
[q]: go up
[a]: go down
[e]xit: exit game
[r]eset [<level> <seed>]: reset game
[t]est: enables test mode
[s]hoot: shoot bullet
[g]renade <x> <y>: add a grenade in position x, y
[w]ave: do wave
Cheat [0]: Clears the road
Cheat [1..5]: Removes all elements of last visible column, and adds an Advanced Object
```

### 3. Testing

Para esta práctica, se han adaptado y extendido los tests que debes pasar en tu práctica. Debes de tener en en cuenta que:

- Al utilizar el comando reset avanzado (con parámetros) los límites de la carretera y la visibilidad pueden cambiar al cambiar el nivel **HARD**.
- El Player siempre avanza en la ejecución de los comandos Q, A. No obstante, hay que controlar al jugador para que siempre se mantenga dentro de la carretera.
- Hemos cambiado los la nomenclatura de los ficheros de pruebas para aprovechar el comando reset avanzado. El nuevo formato es el siguiente:
  - **00\_easy\_s666.txt**: Es la entrada del caso de pruebas **00** con nivel **easy** y semilla **666**.
  - **00\_easy\_s666\_output.txt**: Es la salida esperada para la entrada anterior.

#### 3.1. Casos de prueba

En esta parte de la P2 tenemos una lista de casos de prueba más extensa. A continuación se incluye un listado de todos los casos de prueba que os suministramos y el objetivo que tiene cada uno de ellos:

- **00\_easy\_s666.txt**: Comprobar la ejecución básica de comandos.
- **01\_easy\_s666.txt**: Juega una partida en nivel EASY con semilla 666 que termina mal.
- **02\_easy\_s100.txt**: Juega una partida en nivel EASY con semilla 100 que termina bien.
- **03\_test\_s100.txt**: Juega una partida en nivel TEST con semilla 100 que termina bien.
- **04\_test\_s100.txt**: Juega una una partida en nivel HARD con semilla 100 que termina bien.
- **05\_easy\_s100.txt**: Prueba la gestión del jugador al intentar salir de los bordes de la carretera.
- **06\_easy\_s25.txt**: Prueba el comando Reset básico.
- **07\_easy\_s37.txt**: Prueba el comando Reset avanzado.
- **08\_easy\_s37.txt**: Prueba el comando Info.
- **09\_easy\_s37.txt**: prueba el comando Shoot.
- **10\_easy\_s37.txt**: prueba **Supercoin**.
- **11\_easy\_s37.txt**: prueba el comando turbo.
- **12\_easy\_s37.txt**: Prueba el comando clear, tanto en nivel easy como avanzado.
- **13\_easy\_s37.txt**: prueba el comando Wave.
- **14\_easy\_s37.txt**: prueba la granada.
- **15\_easy\_s37.txt**: Prueba las trampas (cheats).

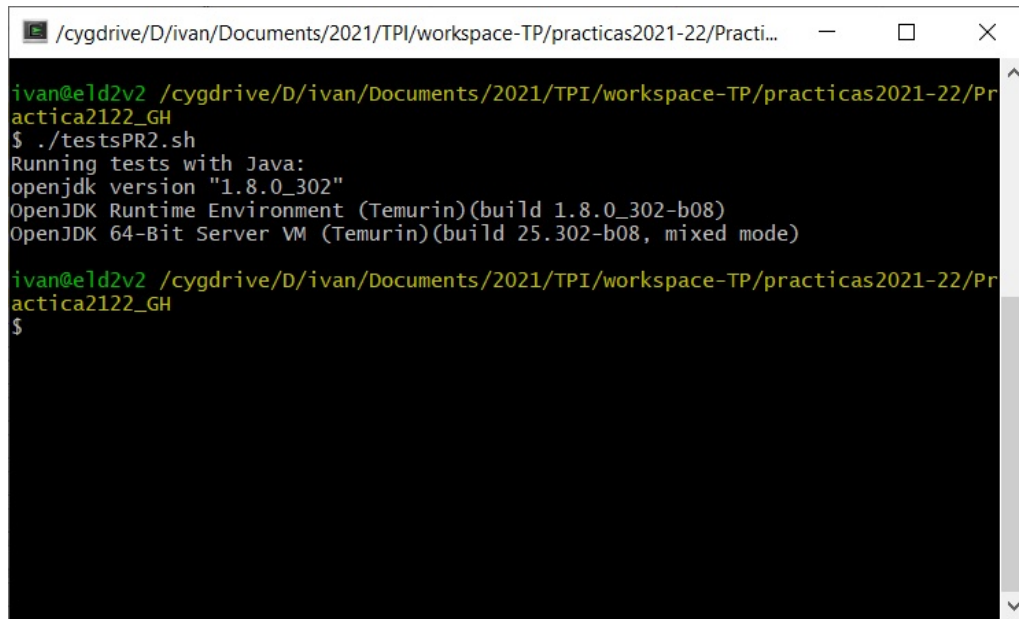
A screenshot of a Cygwin terminal window. The title bar shows the path: /cygdrive/D/ivan/Documents/2021/TPI/workspace-TP/practicas2021-22/Practica2122\_GH. The terminal content shows a user prompt 'ivan@eld2v2 /cygdrive/D/ivan/Documents/2021/TPI/workspace-TP/practicas2021-22/Practica2122\_GH' followed by the command '\$ ./testsPR2.sh'. The output of the script is: 'Running tests with Java:', 'openjdk version "1.8.0\_302"', 'OpenJDK Runtime Environment (Temurin)(build 1.8.0\_302-b08)', and 'OpenJDK 64-Bit Server VM (Temurin)(build 25.302-b08, mixed mode)'. The prompt returns to '\$'.

Figura 1: Ejecutar pruebas de la P2 en cygwin

### 3.2. Ejecución de las pruebas

Ya que tenemos bastantes casos de prueba, también os facilitamos un script `testsPR2.sh` que podéis utilizar para lanzar todos los casos de prueba contra vuestra práctica.

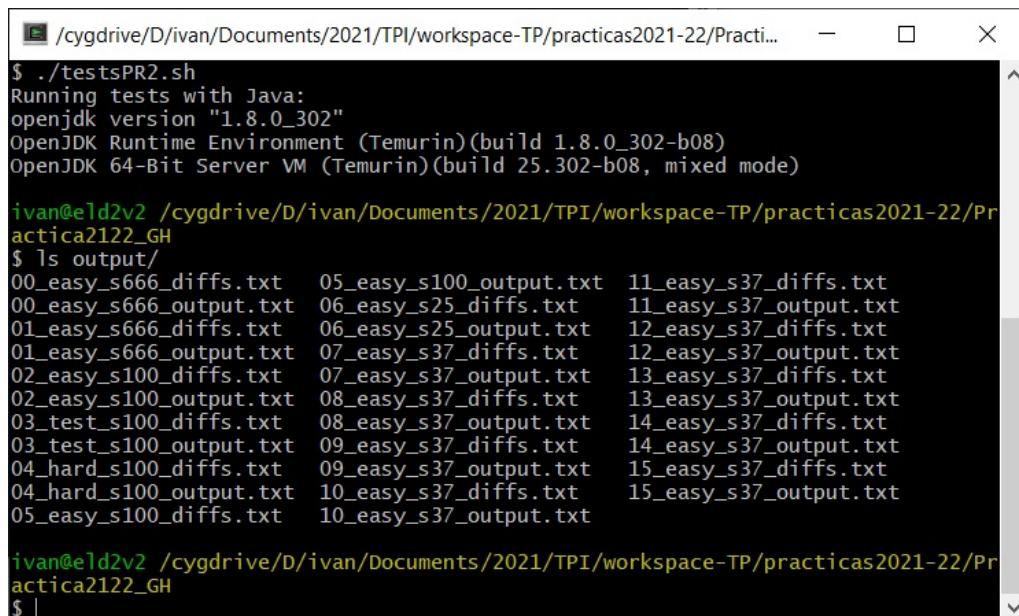
Este script lo podéis utilizar tanto en Linux, MacOS (asegúrate de utilizar bash como terminal) o Windows. En el caso de Windows, es necesario utilizar Cygwin (puede que ya lo tengas instalado de FP1/2 al haber utilizado MinGW).

Para ejecutar las pruebas correctamente, tienes que:

1. Colocar el script `testsPR2.sh` en la raíz del proyecto de Eclipse.
2. Colocar todas las pruebas que te ha facilitado tu profesor en una carpeta `test` dentro del proyecto de Eclipse.
3. Abrir un terminal y ejecutar `testsPR2.sh`.

La figura 1 muestra un terminal de Cygwin con la salida del script. Una vez que has ejecutado las pruebas, se habrá creado un directorio `output` donde podrás encontrar la salida de tu programa para cada una de las pruebas (archivos `_output.txt`) y un fichero que incluye las diferencias de la salida de tu programa con la salida esperada (archivos `_diffs.txt`). Si tienes dificultades para interpretar los ficheros de diferencias, puedes consultar los siguientes recursos:

- [StackOverflow: Interpreting git diff output](#).
- [What is git diff and how do we read the output](#).



```
$ ./testsPR2.sh
Running tests with Java:
openjdk version "1.8.0_302"
OpenJDK Runtime Environment (Temurin)(build 1.8.0_302-b08)
OpenJDK 64-Bit Server VM (Temurin)(build 25.302-b08, mixed mode)

ivan@eld2v2 /cygdrive/D/ivan/Documents/2021/TPI/workspace-TP/practicas2021-22/Practica2122_GH
$ ls output/
00_easy_s666_diffs.txt    05_easy_s100_output.txt    11_easy_s37_diffs.txt
00_easy_s666_output.txt  06_easy_s25_diffs.txt     11_easy_s37_output.txt
01_easy_s666_diffs.txt    06_easy_s25_output.txt    12_easy_s37_diffs.txt
01_easy_s666_output.txt  07_easy_s37_diffs.txt     12_easy_s37_output.txt
02_easy_s100_diffs.txt    07_easy_s37_output.txt    13_easy_s37_diffs.txt
02_easy_s100_output.txt  08_easy_s37_diffs.txt     13_easy_s37_output.txt
03_test_s100_diffs.txt    08_easy_s37_output.txt    14_easy_s37_diffs.txt
03_test_s100_output.txt  09_easy_s37_diffs.txt     14_easy_s37_output.txt
04_hard_s100_diffs.txt    09_easy_s37_output.txt    15_easy_s37_diffs.txt
04_hard_s100_output.txt  10_easy_s37_diffs.txt     15_easy_s37_output.txt
05_easy_s100_diffs.txt    10_easy_s37_output.txt

ivan@eld2v2 /cygdrive/D/ivan/Documents/2021/TPI/workspace-TP/practicas2021-22/Practica2122_GH
$ |
```

Figura 2: Ficheros generados tras las pruebas