# Práctica 1: Road Fighter Car

**Fecha de entrega:** 18 de Octubre 2021, 09:00

**Objetivo:** Iniciación a la orientación a objetos y a Java; uso de arrays y enumerados; manipulación de cadenas con la clase String; entrada y salida por consola.

**Control de copias**

Durante el curso se realizará control de copia de todas las prácticas comparando las entregas de todos los grupos de TP. Se considera copia la reproducción total o parcial de código de otros alumnos o cualquier código extraído de Internet o de cualquier otra fuente, salvo aquellas autorizadas explícitamente por el profesor.

En caso de detección de copia se informará al Comité de Actuación ante Copias que citará al alumno infractor y si considera que es necesario sancionar al alumno propondrá una de las tres siguientes medidas:

- Calificación con cero en la convocatoria de la asignatura a la que corresponde la prueba.
- Calificación con cero en todas las convocatorias del curso actual.
- Apertura de expediente académico ante la Inspección de Servicios de la Universidad.

## 1. Introducción

Vamos a crear un juego de coches inspirado en los clásicos de carreras de las consolas de los años 90. En muchos de estos juegos, el jugador manejaba un coche por una carretera y tenía que sortear obstáculos y adelantar a otros coches hasta llegar a la meta. Estos juegos tenían la peculiaridad de que el coche del jugador siempre estaba en el mismo sitio y era la carretera la que avanzaba hacia él dando la ilusión del movimiento (ver Figura 1).

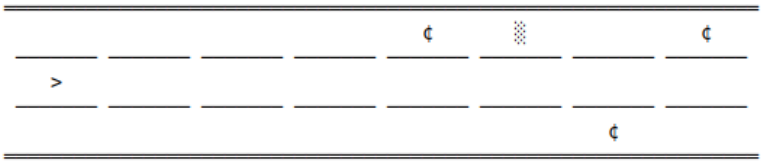
Podéis ver un ejemplo de este tipo de jugabilidad en el siguiente vídeo. El desafío de la práctica es hacer un juego de estilo *arcade* utilizando la línea de comandos.

https://www.youtube.com/watch?v=ha6vRnauX84.


Figura 1: Road Fighter - Nintendo 1991

```
Super cars 1.0

Level: EASY
Random generator initialized with seed: 366
Distance: 30
Coins: 5
Cicle: 0
Total obstacles: 8
Total coins: 18
Ellapsed Time: 0,00 s
```



```
Command >
```

Figura 2: Salida por pantalla esperada para la Práctica 1

En nuestra práctica, a nivel visual, el juego se presenta como una cuadrícula en la que el coche se sitúa en la primera columna, avanzando de izquierda a derecha (ver Figura 2). Como decimos el coche nunca va a cambiar de columna, van a ser el resto de elementos los que se desplacen hacia él. El coche podrá ir hacia arriba y hacia abajo para sortear a los enemigos. Durante el cuatrimestre vamos a ir desarrollando el juego progresivamente.

Empezaremos con una versión muy reducida e iremos incrementando su complejidad añadiendo nuevos tipos de obstáculos y enemigos. En la primera práctica tenemos como objetivo implementar la mínima versión jugable con la consola.

## 2. Descripción de la práctica

En nuestra primera práctica vamos a considerar que el juego consta de una carretera que tiene una longitud $L$ y una anchura $W$ (número de carriles), que dependerán del nivel. En pantalla sólo veremos un fragmento de carretera. Por ejemplo, en el nivel más fácil la carretera tiene un largo de 30 y ancho 3, pero en la pantalla sólo vemos una cuadrícula de $8 \times 3$ casillas (8 columnas por 3 filas). El 8 es lo que llamaremos visibilidad u horizonte, ya que es el número de casillas visibles a partir de la posición del coche.

En la primera columna estará el coche del jugador que avanzará de izquierda a derecha. El jugador gana la partida cuando llega al final de la carretera (ver Figura 3). En el recorrido

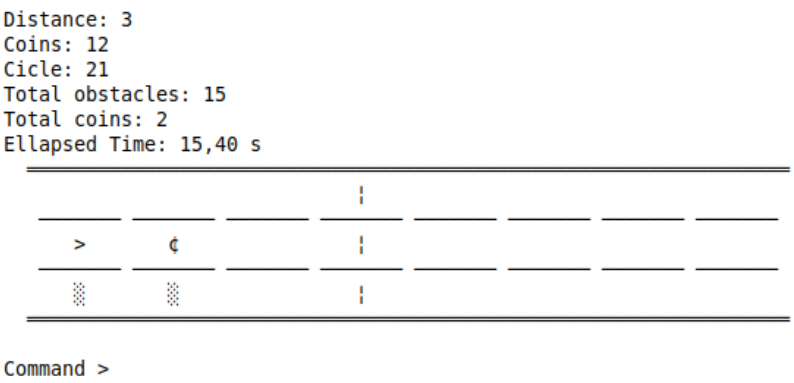



Figura 3: Salida por pantalla visualizando la meta

se enfrentará a diferentes obstáculos que tendrá que esquivar. En las siguientes prácticas también incluiremos otro tipo de objetos de juego como power-ups.

En la primera práctica sólo hay tres tipos de objetos de juego:

- ***Player***. Es el coche que representa al jugador. Está siempre en la primera columna. Se mueve hacia adelante, hacia arriba o hacia abajo, sin salirse del tablero, dependiendo de los comandos del jugador.
- ***Obstacle***. Aparecen aleatoriamente en la carretera. No se mueven, el coche muere si se choca con ellos.
- ***Coin***. El jugador puede recoger monedas con las que podrá comprar power-ups en futuras extensiones.

En cada ciclo del juego se realizan secuencialmente las siguientes acciones:

1. ***Draw***. Se muestra la información del juego y se pinta el tablero.
2. ***User action***. El usuario realiza su movimiento. Además de los movimientos también habrá otras acciones para pedir ayuda o mostrar información extra.
3. ***Update***. Se actualizan los objetos que están en el tablero. En este caso los obstáculos no se mueven, en futuras versiones sí tendrán comportamiento.
4. ***Remove dead objects***. Borramos los objetos muertos.
5. ***Check end***. Se analiza si el juego ha terminado.

Después de cada movimiento del coche, habrá que comprobar si hay alguna colisión con los Obstacle o si recoge algún Coin el jugador.

## 3. Objetos del juego

En esta sección describimos el tipo de objetos que aparecen en el juego y su comportamiento. Consulta el fichero **symbols.txt** distribuido junto al enunciado que contiene los caracteres concretos para representar cada uno de los elementos del juego:

**Player**

- **Comportamiento**: Avanza, va hacia arriba o va hacia abajo.
- **Velocidad**: 1 casilla por turno.
- **Resistencia**: Muere automáticamente con una colisión.

**Obstacle**

- **Comportamiento**: No se mueve. El coche muere cuando choca contra él.
- **Resistencia**: 1 punto de resistencia (en futuras extensiones tendremos muros más fuertes y el usuario podrá dispararles o tirarles bombas).

En esta primera práctica también añadiremos Coins, que son monedas que el usuario puede recoger y que le servirán en las próximas prácticas para comprar power-ups. Pero vamos a hablar primero del comportamiento básico y después añadiremos las monedas. Os recomendemos que también hagáis la práctica incrementalmente para que sea más sencillo.

**Coin**

- **Comportamiento**: No se mueve. Cuando el coche pasa por encima, el jugador incrementa su número de monedas en 1 unidad y el objeto Coin desaparece.

# 4. Actions

A continuación describimos lo que ocurre en cada parte del bucle del juego. El orden de las acciones es el especificado más arriba, en esta sección damos detalles de cada una de las partes.

## 4.1. Inicialización

Los obstáculos se colocan al principio de la partida, el número de obstáculos que hay en la carretera va a depender del nivel. Uno de los parámetros de entrada del programa será el nivel. En la primera versión tendremos tres niveles TEST, EASY, HARD (ver Tabla 1.1). Cada nivel determinará varias opciones de configuración del juego:

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

Tabla 1.1: Configuración para cada nivel de dificultad

- La longitud de carretera (length).
- El ancho de la carretera (width).
- La visibilidad, que indica el número de columnas del tablero (visibility).
- La frecuencia de obstáculos (Obs freq).

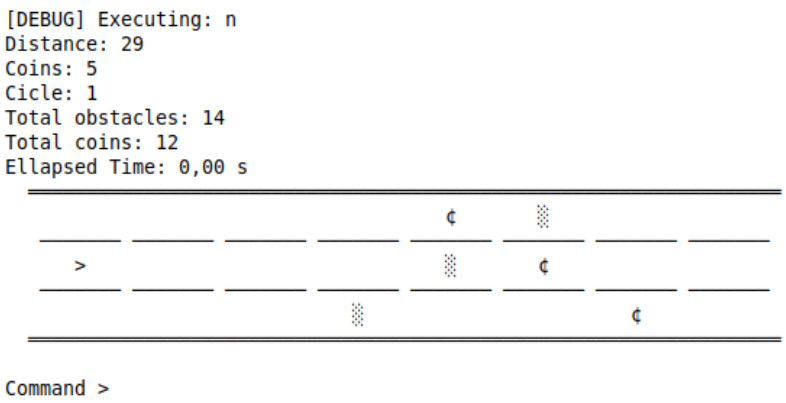



Figura 4: Salida al ejecutar el comando "n"

- La frecuencia de monedas (Coin freq).

La frecuencia determina la probabilidad de que aparezca un determinado objeto en una columna. Así pues, si la frecuencia es 0.5, habrá un obstáculo en una de cada dos columnas. Aunque al tratarse de una probabilidad, pueden salir más espaciados o menos según la aleatoriedad. En el nivel TEST solo hay objetos Obstacle y, además, también **se activa el modo test**. En las primeras columnas de la carretera no va a haber objetos, empezaremos a colocarlos a partir de la columna visibility / 2.

Si vamos a colocar un objeto en una casilla donde ya hay otro objeto de juego entonces no se coloca [1].

## 4.2. Draw

En cada ciclo se pintará el estado actual del tablero; así como el ciclo de juego en el que nos encontramos (inicialmente 0), la distancia a meta, el número de objetos Obstacle y Coins que hay en el tablero, y el tiempo que ha transcurrido desde el inicio de la partida. El tablero se pintará por el interfaz consola utilizando caracteres ASCII, como muestra el siguiente ejemplo.

El tiempo empezará a contar después del primer movimiento del coche. Utilizaremos la función System.currentTimeMillis() para saber el tiempo transcurrido.

Al principio de cada ciclo también mostraremos el último comando introducido ([DEBUG] Executing: n) y al final mostraremos el prompt del juego para pedir al usuario la siguiente acción (Command >)

Todo el juego se dibuja en relación a la posición del coche, es decir la posición de las Xs se ajusta a la posición relativa del coche. El número de columnas que se pintan es la visibilidad que también depende del nivel.

## 4.3. Update

En la primera práctica sólo se mueve el coche hacia arriba, hacia abajo o avanzando. Tenemos que comprobar que no se sale de la carretera. Cuando el coche va hacia arriba y hacia abajo también avanza, por lo que el movimiento es en diagonal.

[1] Puede darse el caso de que la casilla donde se iba a poner un Coin ya esté ocupada por un Obstacle, entonces habría dos opciones: buscar una casilla vacía para colocarlo o no colocarlo. Hemos optado por la segunda opción.

### 4.4. User command

Se le preguntará al usuario qué es lo que quiere hacer, a lo que podrá contestar una de las siguientes alternativas:

- info: Muestra la información de los objetos de juego.
- [q] up: El coche se desplaza hacia arriba.
- [a] down: El coche se desplaza hacia abajo.
- none: El coche avanza. Si no se introduce nada (simplemente al pulsar return el programa hace un update).
- reset: Este comando permite reiniciar la partida, llevando el juego a la configuración inicial.
- test: Este comando, como veremos más abajo, nos va a permitir ejecutar el juego en modo test en el que no se imprime el tiempo recorrido.
- exit: Este comando permite salir de la aplicación, mostrando previamente el mensaje "Game Over".
- help: Este comando solicita a la aplicación que muestre la ayuda sobre cómo utilizar los comandos. Se mostrará una línea por cada comando. Cada línea tiene el nombre del comando seguida por ':'y una breve descripción de lo que hace el comando.

```
[DEBUG] Executing: help
Available commands:
[h]elp: show this help
[i]nfo: prints gameobjet info
[n]one | []: update
[q]: go up
[a]: go down
[e]xit: exit game
[r]eset: reset game
[t]est: enables test mode
```

**Observaciones sobre los comandos:**

- La aplicación debe permitir comandos escritos en minúscula, mayúscula, o mezcla de ambos.
- La aplicación debe permitir el uso de la primera letra del comando en lugar del comando completo [N]one, [R]eset, [H]elp, [E]xit, ...
- Si el comando es vacío se identifica como none o update y se avanza al siguiente ciclo de juego.
- Si el comando está mal escrito, no existe, o no se puede ejecutar, la aplicación mostrará un mensaje de error.
- En el caso de que el usuario ejecute un comando que no cambia el estado del juego, o un comando erróneo, el tablero no se debe repintar.

El juego finalizará cuando el jugador llegue al final del recorrido, o cuando se choca contra un obstáculo. Cuando el juego termine se debe mostrar cómo ha terminado. En el caso de que llegue a meta se mostrará Player wins junto con el nuevo récord. En esta versión el récord no se guarda en un fichero entre partida y partida, así que cada vez que reseteemos el juego, el récord cambiará.

```
[GAME OVER] Player wins!
New record!: 4.85 s
```

En el caso de que el coche choque:

```
[GAME OVER] Player crashed!
```

**Aclaración:** Para controlar el comportamiento aleatorio del juego y poder repetir varias veces la misma ejecución utilizaremos una semilla, o como se conoce en inglés, seed. Este valor lo puede introducir, o no, el usuario. Proporciona un control del comportamiento del programa, lo que nos permitirá repetir exactamente una misma ejecución. Si el usuario no introduce ninguna semilla, entonces la semilla se genera automáticamente. Es muy importante que sólo creemos un objeto Random en el ciclo de vida de la aplicación y que todos los objetos lo compartan para que los resultados sean reproducibles.

Tanto la semilla como el nivel del juego al que queremos jugar se pasarán como argumentos al programa principal.

# 5. Implementación

La implementación propuesta para la primera práctica no es la mejor; ya que la vamos hacer sin utilizar **herencia** ni **polimorfismo**, dos herramientas básicas de la programación orientada a objetos. Durante el curso veremos formas de mejorarla mediante el uso de las herramientas que nos brinda la programación orientada a objetos. Además, para la primera versión, especialmente al añadir las monedas, tendremos que copiar y pegar mucho código, y esto casi siempre es una mala práctica de programación. La duplicación de código implica que va a ser poco mantenible y testeable. Hay un principio de programación muy conocido que se conoce como **DRY (Don't Repeat Yourself)** o **No te repitas** en castellano. Según este principio, toda "pieza de información" nunca debería ser duplicada debido a que la duplicación incrementa la dificultad en los cambios y evolución posterior, y puede perjudicar la claridad y crear un espacio para posibles inconsistencias.

A lo largo de las siguientes prácticas veremos cómo refactorizar el código para evitar repeticiones.

## 5.1. Detalles de implementación

Para lanzar la aplicación se ejecutará la clase es.ucm.tp1.SuperCars, por lo que se aconseja que todas las clases desarrolladas en la práctica estén en paquetes que cuelguen de es.ucm.tp1. Recordad que según las convenciones de nomenclatura para Java, todos los paquetes deben ir en minúsculas, y las clases deben empezar siempre por mayúsculas.

Para implementar la práctica necesitarás, al menos, las siguientes clases:

- Coin, Obsctacle: Estas clases encapsulan el comportamiento de los elementos del juego. Tienen atributos privados, como su posición x, y, ... También tienen un atributo en el que almacenan una referencia al juego, esto es, una instancia de la clase Game (única

en el programa) que como veremos contiene la lógica del juego. De este modo, estas clases podrán usar los métodos de la clase Game para consultar si pueden hacer, o no, una determinada acción.

  - La contabilización de *cuántos obstáculos/coins* hay se realizará desde la propia clase Coin u Obstacle utilizando atributos estáticos. Game accederá a esas variables para mostrar la información.

- Player: Esta clase sirve para manejar el coche. Contiene la posición del coche, el número de monedas que ha adquirido el jugador, etc. En el update del coche se modifica la posición y se comprueba si ha habido alguna colisión con obstáculos o monedas.

- ObstacleList, CoinList: Contienen arrays de los respectivos elementos del juego, así como métodos auxiliares para su gestión. En esta práctica usaremos obligatoriamente arrays convencionales. En las siguientes prácticas las substituiremos por ArrayLists.

- Game: Encapsula la lógica del juego. Tiene, entre otros, el método update que actualiza el estado de todos los elementos del juego. Esta clase debe gestionar el contador de ciclos de juego y guardar una referencia a la instancia de Player.

- Level: Es una clase tipo Enum que contiene los valores correspondientes a cada nivel de juego.

- GamePrinter: Recibe el objeto game y tiene un método toString que sirve para pintar el juego como veíamos anteriormente.

- Controller: Clase para controlar la ejecución del juego, preguntando al usuario qué quiere hacer y actualizando la partida de acuerdo a lo que éste indique. La clase Controller necesita al menos dos atributos privados:

```
private Game game;
private Scanner in;
```

  El objeto in sirve para leer de la consola las órdenes del usuario. La clase Controller implementa el método público public void run() que controla el bucle principal del juego. Concretamente, mientras la partida no esté finalizada, solicita órdenes al usuario y las ejecuta.

- SuperCars: Es la clase que contiene el método main de la aplicación. En este caso el método main lee los valores de los parámetros de la aplicación (nivel y semilla), crea una nueva partida (objeto de la clase Game), crea un controlador (objeto de la clase Controller) con dicha partida, e invoca el método run del controlador.

**Observaciones a la implementación:**

- Durante la ejecución de la aplicación solo se creará un objeto de la clase Controller. Lo mismo ocurre para la clase Game (que representa la partida en curso y solo puede haber una activa).

- Junto con la práctica os proporcionaremos unas plantillas con partes del código.

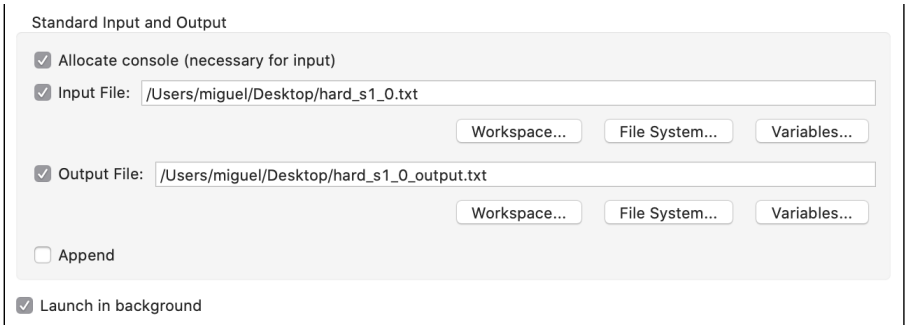



Figura 5: Redirección entrada y salida

El resto de información concreta para implementar la práctica será explicada por el profesor durante las distintas clases de teoría y laboratorio. En esas clases se indicará qué aspectos de la implementación se consideran obligatorios para poder aceptar la práctica como correcta y qué aspectos se dejan a la voluntad de los alumnos.

## 6. Entrega de la práctica

La práctica debe entregarse utilizando el mecanismo de entregas del campus virtual, no más tarde de la fecha y hora indicada en la cabecera de la práctica.

El fichero debe tener al menos el siguiente contenido [2]:

- Directorio `src` con el código de todas las clases de la práctica.
- Fichero alumnos.txt donde se indicará el nombre de los componentes del grupo.

Recuerda que no se deben incluir los .class. **Nota:** puedes utilizar la opción File >Export para ayudarte a generar el .zip.

## 7. Testing

Junto con las instrucciones de la práctica tendrás una carpeta con trazas del programa. Encontrarás dos tipos de ficheros con la siguiente nomenclatura:

- easy_s5_0.txt: es la entrada del caso de prueba 0 con nivel easy y semilla 5.
- easy_s5_0_output.txt: es la salida esperada para la entrada anterior

En Eclipse, para usar un fichero de entrada y volcar la salida en un fichero de salida, debes configurar la redirección en la pestaña Common de la ventana Run Configurations, tal y como se muestra en la Figura 5.

Para generar nuestros tests, debes tener en cuenta que:

- Hemos usado el método Random.nextDouble() para saber si hay que añadir un objeto o no.

[2] Puedes incluir también opcionalmente los ficheros de información del proyecto de Eclipse

- Si frecuenciaObjetoNivel es la frecuencia que aparece en la Tabla 1.1 para cada objeto, entonces añadimos el objeto al tablero si Random.nextDouble() <frencuenciaObjetoNivel
- Siguiendo las indicaciones de la Sección 4.1, solo se añaden elementos al tablero a partir de la columna visibility / 2.

Puedes ver en el listado 1.1 un pseudocódigo para generar los elementos del juego. Ten en cuenta que:

- tryToAddObstacle y tryToAddCoin tienen por objetivo añadir, si es posible, un elemento al juego.
- getVisibility() devuelve la visibilidad del nivel actual.
- getRandomLane() devuelve un entero aleatorio correspondiente al carril donde se va a intentar añadir el objeto.

```
for (int x = getVisibility() / 2; x < roadLength; x++) {
  tryToAddObstacle(new Obstacle(game, x, getRandomLane()), level.
      obstacleFrequency());
  tryToAddCoin(new Coin(game, x, getRandomLane()), level.
      coinFrequency());
}
```

Listing 1.1: Pseudocódigo para generar los objetos del juego

Cuando generes una salida de tu juego, debes compararla con la salida correcta que te entregamos. Hay multitud de programas gratuitos para comparar visualmente ficheros, por ejemplo Eclipse ya tiene integrada una herramienta para comparar archivos que puedes lanzar al seleccionar dos archivos, pulsar con el botón derecho y en el menú emergente seleccionar **Compare With >Each other** (ver Figuras 6 y 7)

Otra posibilidad es utilizar la aplicación *Beyond compare* (https://www.scootersoftware.com/) que se encuentra disponible para todas las plataformas y es muy útil para asegurarte de que tu salida corresponde con la deseada.[3]

Ten mucho cuidado con el orden de las instrucciones porque puede cambiar la salida considerablemente. Por supuesto, nuestra salida puede tener algún error, así que si detectas alguna inconsistencia por favor comunícanoslo para que lo corrijamos.

Durante la corrección de prácticas os daremos nuevos ficheros de prueba para asegurarnos de que vuestras prácticas se generalizan correctamente, así que asegúrate de probar no sólo los casos que te damos, sino todas otras posibles ejecuciones que no estén presentes en ellos.

[3]Otro software que puedes usar es DiffMerge (https://sourcegear.com/diffmerge/)

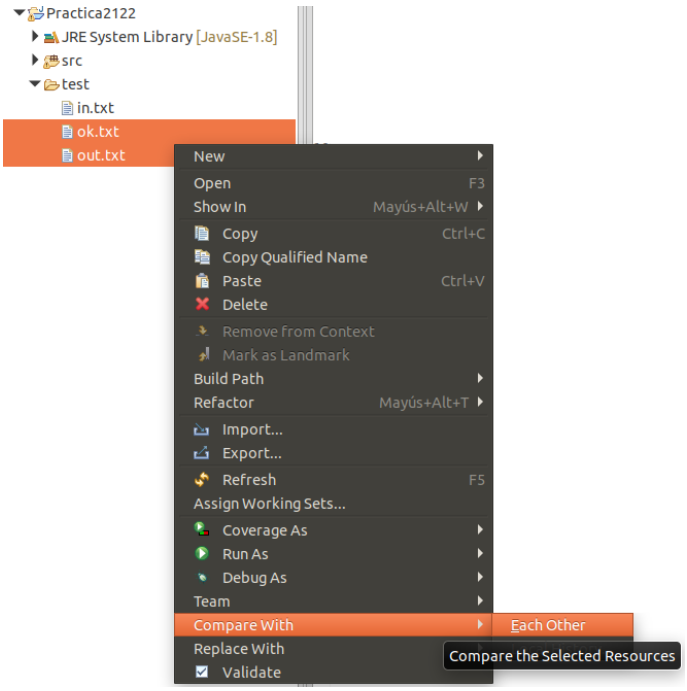



Figura 6: Cómo comparar dos archivos en Eclipse

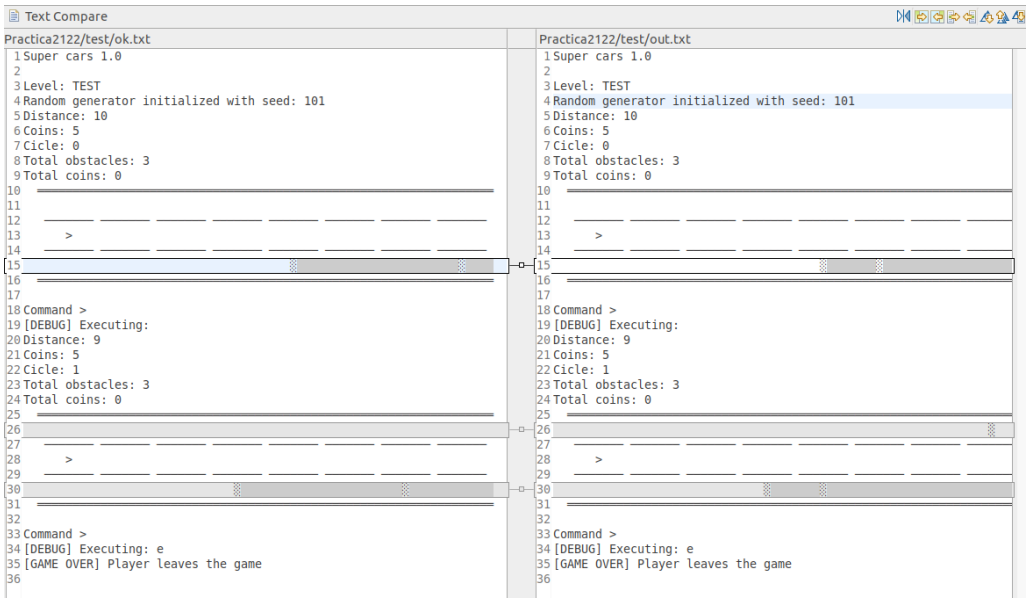



Figura 7: Salida de la herramienta de comparación de archivos de Eclipse