PRÁCTICA 4

Herencia, interfaces y excepciones

Inicio: Semana del 5 de abril
Duración: 3 semanas
Entrega: Semana del 26 de abril
Peso de la práctica: 30%

El objetivo de esta práctica es la simulación de una carrera entre vehículos de distinta clase utilizando técnicas de programación orientada a objetos más avanzadas que en las prácticas precedentes, como son la herencia, las excepciones y las *interfaces*. Se trata de servirse de estas técnicas para reducir código redundante, obtener una API fácilmente extensible, y desarrollar un programa bien estructurado que refleje de una forma directa los conceptos del dominio del problema.

En el desarrollo de esta práctica se utilizarán principalmente los siguientes conceptos de Java y de programación orientada a objetos:

- *Herencia y ligadura dinámica*
- *Manejo de excepciones*
- *Uso de interfaces Java*
- *Estructuración en paquetes*

Motivación

Uno de los ejemplos más naturales en los que el diseño de software orientado a objetos se vuelve especialmente intuitivo es en los **videojuegos**, donde tienes personajes u objetos con atributos específicos que los diferencian del resto de su misma clase, e interaccionan entre sí con un fin.

El objetivo de esta práctica es hacer uso de esa intuición para crear nuestra propia **carrera** de diferentes clases de vehículos. La carrera empezará siendo muy simple, con motos que son veloces y ganarán siempre, coches, y camiones algo más lentos. No obstante, le iremos incorporando elementos que la hagan más impredecible siguiendo una **lógica similar a la de juegos como Mario Kart**, primero añadiendo **ataques** que dañen los componentes de los vehículos, y finalmente añadiendo habilidades especiales, o ***power ups***.

Para hacer todo esto, iremos creando una librería de paquetes cada vez más extensa que vendrá dada por las interfaces que iremos proponiendo en el enunciado. Una recomendación muy aconsejable es pensar bien en cómo los diferentes elementos (**vehículos, la carrera, los componentes**, etc...) van a conectar entre sí antes de empezar a generar código, ya que en la evaluación de la práctica se tendrá en cuenta no sólo el propio código, sino su diseño y la explicación del mismo tanto en la **memoria como en el diagrama de clases** que hay que entregar.

Al igual que en las prácticas anteriores, resulta imprescindible hacer un código modular aprovechando al máximo los conceptos que ya conocéis de la programación orientada a objetos (herencia, polimorfismo etc.).

Apartado 1: Crear los vehículos de la carrera a partir de la lectura de un fichero (2,5 puntos)

En esta práctica vamos a simular una carrera (Race) entre distintos tipos de vehículos, que de momento serán de tres tipos, Camiones (Truck), Coches (Car) y Motos (Motorcycle). Aunque cada uno va a tener unas características que las diferencie de los demás tipos de vehículos, comparten una estructura mínima común. La forma de reflejar esto es mediante el uso de una interfaz, en este caso: IVehicle, que es la que vamos a usar. De momento, la interfaz es:

```
public interface IVehicle {
        public double getActualPosition();
        public void setActualPosition(double newPosition);
        public boolean canMove();
        public void setCanMove(boolean newMovement);
        public double getMaxSpeed();
        public String getName();

        //Para el apartado 3
        //public void addComponent(IComponent c) throws InvalidComponentException;
        //public List<IComponent> getComponents();
}
```

IMPORTANTE: la definición de las interfaces no debe modificarse salvo para añadir comentarios javadoc. No obstante se te da libertad para declarar clases e interfaces adicionales si lo requieren

La mayoría de los métodos de IVehicle nos dan la información necesaria de un vehículo para la carrera, o permiten modificarla, como la posición (getActualPosition(), setActualPosition()); si el vehículo se puede mover o no (canMove(), setCanMove()); la velocidad máxima (getMaxSpeed()) y el nombre del vehículo (getName()). Para poder distinguir entre los vehículos del mismo tipo se pide que el nombre de cada vehículo sea el tipo de vehículo más un identificador único por vehículo entre paréntesis que debe ser generado por el sistema (por ejemplo: Truck(1)). Los métodos addComponent(IComponent c) y getComponents() serán explicados más en detalle en el apartado 3, ya que para este apartado no son necesarios.

Los detalles de la carrera, es decir, su longitud hasta la línea de meta y los vehículos participantes, se leerán de un fichero que procesará el método *read* de RaceReader. La primera línea del fichero establece la longitud de la carrera. Cada una de las siguientes líneas especificará (separados por espacios), el número de vehículos del tipo; el tipo de vehículo que hay que añadir a la carrera; y la velocidad máxima a la que puede llegar ese tipo de vehículo.

El siguiente ejemplo muestra el contenido del fichero de texto que usaremos para los apartados 1 y 2. Como puede observarse, la carrera es de longitud 100, y participan dos coches, un camión y una motocicleta:

```
100
2 Car 7
1 Truck 6
1 Motorcycle 9
```

Si la longitud de la carrera es demasiado pequeña como para que no se pueda avanzar, o hay pocos/demasiados participantes, la carrera no tendrá sentido, así que hay prevenir estas situaciones. Como muestra el ejemplo de más abajo, se deberá lanzar una excepción, si la longitud de la carrera es menor o igual a la velocidad máxima de cualquier vehículo, o si el número de vehículos es menor que 2, o mayor que 10.

El objetivo de este primer ejercicio es preparar todos los ingredientes necesarios para la carrera antes de hacer la simulación de la misma en el siguiente apartado. Como guía, el siguiente programa debe producir la salida de más abajo, cuando recibe como parámetro el fichero de texto con la información de la carrera.

MainEx1.java

```
public class MainEx1 {
	public static void main(String [] args) {
		Race r;
		try {
			r = RaceReader.read(args[0]);
			System.out.println(r);

		} catch (IOException e) {
			System.out.println("Error reading the file");
		} catch (RaceException e) {
			System.out.println(e);
		}
	}
}
```

cuya salida esperada es:

```
Race with maximum length: 100.0
Car(1). Speed 7.0. Actual position: 0.0.
  Car(1) distance to Car(2): 0.0
  Car(1) distance to Truck(3): 0.0
  Car(1) distance to Motorcycle(4): 0.0
Car(2). Speed 7.0. Actual position: 0.0.
  Car(2) distance to Car(1): 0.0
  Car(2) distance to Truck(3): 0.0
  Car(2) distance to Motorcycle(4): 0.0
Truck(3). Speed 6.0. Actual position: 0.0.
  Truck(3) distance to Car(1): 0.0
  Truck(3) distance to Car(2): 0.0
  Truck(3) distance to Motorcycle(4): 0.0
Motorcycle(4). Speed 9.0. Actual position: 0.0.
  Motorcycle(4) distance to Car(1): 0.0
  Motorcycle(4) distance to Car(2): 0.0
  Motorcycle(4) distance to Truck(3): 0.0
```

En este apartado aún no se va a desarrollar la simulación de la clase Race, pero tendrá que estar lista la mayor parte de su lógica. Por esa razón, se muestra por cada vehículo la distancia absoluta con el resto de vehículos de la carrera.

Por otro lado, antes de empezar a programar, es necesario diseñar bien en cómo estructurar la clase Race. Para ello, debéis pensar cuál es el objetivo último de esta librería, explicado en la motivación, y cómo podéis aplicar lo que habéis aprendido en teoría para hacer un buen uso de todos los conceptos nuevos, y así alcanzar un código ordenado y extensible. Tanto para este apartado como para los siguientes, se debe organizar el código en una estructura de paquetes coherente.

Apartado 2: Simulación básica de la carrera (2 puntos)

En este apartado se procederá a hacer la primera simulación de la carrera. Una carrera constará de turnos, en los que cada vehículo actualizará su posición, y terminará cuando el primer vehículo cruce la meta. Cada vehículo tiene unas normas diferentes de avance:

- Las motos siempre se moverán a su máxima velocidad.
- Los coches, que son más voluminosos, se moverán a su velocidad máxima con una probabilidad $p_{car} = 0.9$. Cuando no se mueven a su velocidad máxima, lo harán al 90% de su velocidad máxima.
- Los camiones, los más grandes de todos, se moverán a su velocidad máxima con una probabilidad $p_{truck} = 0.9$. Cuando no se mueve a su velocidad máxima, lo harán al 80% de su velocidad máxima.

Como recordarás, se tiene pensado incorporar una serie de ataques a los vehículos más próximos, pero esto será abordado en el Apartado 3.

Como guía para tu diseño, el siguiente programa debería producir la salida de más abajo:

```
public class MainEx2 {
	public static void main(String [] args) {
		Race r;
		try {
			r = RaceReader.read(args[0]);
			r.simulate();
		} catch (IOException e) {
			System.out.println("Error reading the file");
		} catch (RaceException e) {
			System.out.println(e);
		}
	}
}
```

Las primeras líneas que deberían aparecer en la salida de la simulación deberían ser:

```
---------
Starting Turn: 1
Race with maximum length: 100.0
Car(1). Speed 7.0. Actual position: 0.0.
  Car(1) distance to Car(2): 0.0
  Car(1) distance to Truck(3): 0.0
  Car(1) distance to Motorcycle(4): 0.0
Car(2). Speed 7.0. Actual position: 0.0.
  Car(2) distance to Car(1): 0.0
  Car(2) distance to Truck(3): 0.0
  Car(2) distance to Motorcycle(4): 0.0
Truck(3). Speed 6.0. Actual position: 0.0.
  Truck(3) distance to Car(1): 0.0
  Truck(3) distance to Car(2): 0.0
  Truck(3) distance to Motorcycle(4): 0.0
Motorcycle(4). Speed 9.0. Actual position: 0.0.
  Motorcycle(4) distance to Car(1): 0.0
  Motorcycle(4) distance to Car(2): 0.0
  Motorcycle(4) distance to Truck(3): 0.0

Ending Turn: 1
---------
```

y las últimas:

```
Motorcycle(4). Speed 9.0. Actual position: 108.0.
 has won the race
```

El formato de los números debe coincidir con el mostrado.

Probad diferentes configuraciones de velocidades máximas, y comprobad que gana el vehículo adecuado. Si las motos y los camiones se dejan a la misma velocidad actual, ¿cuál es la mínima velocidad máxima a la que tienen que ir los coches para ganar siempre las carreras?

Apartado 3: Atacando al adversario (3 puntos)

Como se ha podido observar en los resultados del apartado anterior, tal y como está configurado, las motos siempre ganan la carrera. Para hacer un poco más interesante la carrera, vamos a configurar los vehículos para que puedan lanzar cáscaras de plátano a los vehículos más cercanos que tengan por delante. Cuando un vehículo es afectado por una cáscara de plátano puede causarle daño a uno de sus componentes, y según eso puede que tenga que esperar más o menos turnos para su reparación.

Por lo tanto, vamos a dotar a los vehículos de componentes. Por el momento tendremos cuatro componentes: Wheels, Engine, Window y BananaDispenser. Según podemos ver en la interfaz de IComponent, el componente en cuestión podrá estar dañado o no dañado.

```
public interface IComponent {
        public boolean isDamaged();
        public void setDamaged(boolean damage);
        public String getName();
        public int costRepair();
        public IVehicle getVehicle();
        public boolean isCritical();
        public void repair();
}
```

No es lo mismo que se dañen componentes como las ruedas o el motor, que impedirán completamente el avance, a que se dañen las ventanas o el dispensador. Por lo tanto, si se dañan los componentes más críticos (Wheels, Engine), el vehículo tendrá que parar completamente para repararlos, perdiendo 3 turnos. Si se dañan los otros componentes, su reparación costará 2 turnos para las ventanas y 4 turnos para el dispensador, pero se podrá seguir avanzando mientras son reparados. Si el dispensador está dañado, ese vehículo **no podrá atacar** hasta que se haya reparado.

Con estos componentes, las motos salen desfavorecidas, ya que no pueden tener ni ventanas ni dispensadores. El resto de vehículos sí tendrán todos los componentes. Para trabajar con los componentes debe hacerse uso de los métodos addComponent y getComponents definidos en la interfaz IVehicle.

Por lo tanto, la carrera ahora tiene la siguiente estructura:

- Empieza en el turno 1.
- Mostrar el estado actual de la carrera.
- Fase de ataque (disponible cada tres turnos empezando por el 3: 3, 6, 9...): todos los vehículos lanzan sus cáscaras de plátano siguiendo el orden que se ha seguido al incluirlos en la carrera.
- Fase de reparación: todos los vehículos comprueban si han sido atacados y comienzan su reparación de cada uno de los componentes dañados en caso de ser necesaria.
- Actualización de posiciones: actualizan las posiciones si no han sufrido daños en sus componentes más críticos.

La lógica de la fase de ataque es la siguiente:

- Un vehículo puede atacar si tiene el dispensador de cáscaras de plátano operativo (no dañado).
- El vehículo víctima será aquél que vaya inmediatamente delante del atacante. Si hay más de uno a la misma distancia, selecciona uno de ellos aleatoriamente.
- La distancia al vehículo que va a atacar tiene que estar a menos de 30 unidades.
- Un vehículo no puede dañarse a sí mismo.
- El ataque tiene una probabilidad de éxito de un 50%. En caso de que el ataque falle, no se hace nada.

- Si el ataque tiene éxito, le daña uno de sus componentes de manera aleatoria. En el caso en el que un componente sea dañado antes de que se haya reparado, el número de turnos de reparación no es acumulativo.

Crearemos un nuevo fichero de entrada para los apartados 3 y 4 que contenga los componentes que tiene cada vehículo de la carrera, mostrados a continuación de la velocidad máxima.

```
100
1 Car 7 Wheels Engine Window BananaDispenser
1 Truck 6 Engine Window BananaDispenser Wheels
1 Motorcycle 9 Engine Wheels
```

Ten en cuenta que la clase RaceReader debe ser capaz de leer tanto este fichero como el de los primeros apartados. Como se ha comentado, las motos sólo pueden tener motor y ruedas, por lo tanto, debe saltar una excepción si se intenta meterle un componente que no le corresponde que diga "Component XXX is not valid for Vehicle XXX(X)". Esa es la excepción (InvalidComponentException) que lanza el método addComponent de la interfaz de IVehicle.

Como guía a tu diseño, el siguiente programa debe resultar en la salida de más abajo.

```
public class MainEx3 {
	public static void main(String [] args) {
		Race r;
		try {
			r = RaceReader.read(args[0]);
			r.allowAttacks(true);
			r.simulate();
		} catch (IOException e) {
			System.out.println("Error reading the file");
		} catch (RaceException e) {
			System.out.println(e);
		}
	}
}
```

Debes procurar que el nuevo código desarrollado no modifique la salida de los apartados anteriores. La salida de este apartado tiene que añadir a las anteriores los componentes de cada vehículo, y además:

- Al inicio de cada turno, en la fase de mostrar el estado actual de la carrera, se tiene que comprobar el estado de todos los componentes de cada vehículo, indicando si están dañados, entre las líneas de velocidad máxima y posición actual, y antes de la distancia al resto de vehículos:
  (ejemplo de las primeras líneas de la salida, en verde las nueva incorporación)

```
Component Window is not valid for Vehicle Motorcycle(3)
---------
Starting Turn: 1
Race with maximum length: 100.0
Car(1). Speed 7.0. Actual position: 0.0.
->Wheels. Is damaged: false. Is critical: true
->Engine. Is damaged: false. Is critical: true
->Window. Is damaged: false. Is critical: false
->Banana dispenser. Is damaged: false. Is critical: false
  Car(1) distance to Truck(2): 0.0
  Car(1) distance to Motorcycle(3): 0.0
Truck(2). Speed 6.0. Actual position: 0.0.
->Engine. Is damaged: false. Is critical: true
->Window. Is damaged: false. Is critical: false
->Banana dispenser. Is damaged: false. Is critical: false
->Wheels. Is damaged: false. Is critical: true
  Truck(2) distance to Car(1): 0.0
  Truck(2) distance to Motorcycle(3): 0.0
Motorcycle(3). Speed 9.0. Actual position: 0.0.
```

```
->Engine. Is damaged: false. Is critical: true
->Wheels. Is damaged: false. Is critical: true
  Motorcycle(3) distance to Car(1): 0.0
  Motorcycle(3) distance to Truck(2): 0.0

 Not attacking turn
 Ending Turn: 1
 ---------
```

- Separar con una línea vacía la salida de las siguientes fases.
- En la fase de ataque, mostrar si el ataque de cada vehículo se ha efectuado con éxito o no.
- Si no es turno de ataque mostrar "Not attacking turn" antes de la fase de reparación.
- En la fase de reparación, mostrar qué vehículo está siendo reparado, y en qué fase de la reparación está.

A continuación se muestra un ejemplo de la salida en las fases de ataque y reparación. Recuerda que los turnos 1 y 2, por ejemplo, no tienen fase de ataque, por lo que el ataque y la reparación, empiezan en el tercer turno; al tener la carrera elementos probabilísticos, pueden salir resultados diferentes a este.

```
[...] // turns 1 and 2
 Not attacking turn
 Ending Turn: 2
 ---------
 ---------
 Starting Turn: 3
 Race with maximum length: 100.0
 Car(1). Speed 7.0. Actual position: 14.0.
 ->Wheels. Is damaged: false. Is critical: true
 ->Engine. Is damaged: false. Is critical: true
 ->Window. Is damaged: false. Is critical: false
 ->Banana dispenser. Is damaged: false. Is critical: false
   Car(1) distance to Truck(2): 3.1999999999999993
   Car(1) distance to Motorcycle(3): 4.0
 Truck(2). Speed 6.0. Actual position: 10.8.
 ->Engine. Is damaged: false. Is critical: true
 ->Window. Is damaged: false. Is critical: false
 ->Banana dispenser. Is damaged: false. Is critical: false
 ->Wheels. Is damaged: false. Is critical: true
   Truck(2) distance to Car(1): 3.1999999999999993
   Truck(2) distance to Motorcycle(3): 7.199999999999999
 Motorcycle(3). Speed 9.0. Actual position: 18.0.
 ->Engine. Is damaged: false. Is critical: true
 ->Wheels. Is damaged: false. Is critical: true
   Motorcycle(3) distance to Car(1): 4.0
   Motorcycle(3) distance to Truck(2): 7.199999999999999

 Starting attack phase
 Car(1) fails attack
 Truck(2) attacks Car(1) Banana dispenser
 Motorcycle(3) can not attack
 End attack phase
 Car(1) Banana dispenser is being repaired 1/4
 Ending Turn: 3
 ---------
 ---------
 Starting Turn: 4
 Race with maximum length: 100.0
 Car(1). Speed 7.0. Actual position: 21.0.
 ->Banana dispenser. Is damaged: true. Is critical: false
 ->Window. Is damaged: false. Is critical: false
 ->Engine. Is damaged: false. Is critical: true
 ->Wheels. Is damaged: false. Is critical: true
   Car(1) distance to Truck(2): 4.199999999999999
   Car(1) distance to Motorcycle(3): 6.0
```

```
Truck(2). Speed 6.0. Actual position: 16.8.
->Engine. Is damaged: false. Is critical: true
->Window. Is damaged: false. Is critical: false
->Banana dispenser. Is damaged: false. Is critical: false
->Wheels. Is damaged: false. Is critical: true
  Truck(2) distance to Car(1): 4.199999999999999
  Truck(2) distance to Motorcycle(3): 10.2
Motorcycle(3). Speed 9.0. Actual position: 27.0.
->Engine. Is damaged: false. Is critical: true
->Wheels. Is damaged: false. Is critical: true
  Motorcycle(3) distance to Car(1): 6.0
  Motorcycle(3) distance to Truck(2): 10.2

Not attacking turn
Car(1) Banana dispenser is being repaired 2/4
Ending Turn: 4
---------
```

El indicador del ganador de la carrera será diferente al del apartado anterior:

```
[...]
---------
Motorcycle(3). Speed 9.0. Actual position: 108.0.
->Wheels. Is damaged: false. Is critical: true     (← new lines are added when printing the object)
->Engine. Is damaged: false. Is critical: true
 has won the race
```

Apartado 4: Añadiendo habilidades especiales (2,5 puntos)

La carrera ya tiene algunos componentes dinámicos, pero vamos a mejorarla aún más añadiendo habilidades especiales, o *power ups*. Estos deben seguir la siguiente interfaz:

```
public interface IPowerUp {
        public void applyPowerUp(IVehicle v);
        public String namePowerUp();
}
```

Consideraremos dos tipos de power ups:

- *Swap*: intercambio de posición con el vehículo que esté inmediatamente delante. En caso de que haya más de un vehículo a la misma distancia se selecciona aleatoriamente uno de ellos.
- *AttackAll*: atacar a todos los demás vehículos independientemente de la distancia, de si se encuentran detrás o delante, y de quién sea el vehículo atacante (es decir, las motos pueden atacar). No obstante, mantenemos la probabilidad del 50% de acertar el ataque.

Además de estos dos *power ups* tenéis que inventar y diseñar un *power up* más que puede ser lo que queráis, a excepción de “ganar la carrera” y modificaciones de los anteriores. Describid brevemente dicho *power up* en la memoria: en qué consiste, su nombre, y cómo lo habéis implementado.

En los turnos que no sean de ataque, con una probabilidad de un 10%, todos los vehículos activarán un *power up* aleatorio de los tres disponibles, por lo que la carrera ahora queda finalmente con la siguiente estructura:

- Empezar el turno.
- Mostrar el estado actual de la carrera.
- Fase de ataque (cada 3 turnos) / Fase de *power ups* (resto de turnos a partir del primer turno de ataque)
- Fase de reparación.
- Actualización de posiciones.

El *main* de este apartado es:

```
public class MainEx4 {
	public static void main(String [] args) {
		Race r;
		try {
			r = RaceReader.read(args[0]);
			r.allowAttacks(true);
			r.allowPowerUps(true);
			r.simulate();

		} catch (IOException e) {
			System.out.println("Error reading the file");
		} catch (RaceException e) {
			System.out.println(e);
		}
	}
}
```

Como puedes crear tu propio power up, la salida que ponemos a continuación es orientativa, pues será diferente para cada pareja. Tendrás que escribir el output completo de este apartado en la memoria.

Ejemplo de salida indicando los power ups:

[...]

```
Not attacking turn
Turn with no power ups          (← no power up was activated due to its low probability)
Ending Turn: 4
---------
---------
Starting Turn: 5
Race with maximum length: 100.0
Car(1). Speed 7.0. Actual position: 27.3.
->Wheels. Is damaged: false. Is critical: true
->Engine. Is damaged: false. Is critical: true
->Window. Is damaged: false. Is critical: false
->Banana dispenser. Is damaged: false. Is critical: false
  Car(1) distance to Truck(2): 3.3
  Car(1) distance to Motorcycle(3): 8.7
Truck(2). Speed 6.0. Actual position: 24.0.
->Engine. Is damaged: false. Is critical: true
->Window. Is damaged: false. Is critical: false
->Banana dispenser. Is damaged: false. Is critical: false
->Wheels. Is damaged: false. Is critical: true
  Truck(2) distance to Car(1): 3.3
  Truck(2) distance to Motorcycle(3): 12.0
Motorcycle(3). Speed 9.0. Actual position: 36.0.
->Engine. Is damaged: false. Is critical: true
->Wheels. Is damaged: false. Is critical: true
  Motorcycle(3) distance to Car(1): 8.7
  Motorcycle(3) distance to Truck(2): 12.0

Not attacking turn
Turn with no power ups          (← no power up was activated due to its low probability)
Ending Turn: 5
---------
---------
Starting Turn: 6
Race with maximum length: 100.0
Car(1). Speed 7.0. Actual position: 34.3.
->Wheels. Is damaged: false. Is critical: true
->Engine. Is damaged: false. Is critical: true
->Window. Is damaged: false. Is critical: false
```

->Banana dispenser. Is damaged: false. Is critical: false
  Car(1) distance to Truck(2): 4.2
  Car(1) distance to Motorcycle(3): 10.7
Truck(2). Speed 6.0. Actual position: 30.0.
->Engine. Is damaged: false. Is critical: true
->Window. Is damaged: false. Is critical: false
->Banana dispenser. Is damaged: false. Is critical: false
->Wheels. Is damaged: false. Is critical: true
  Truck(2) distance to Car(1): 4.2
  Truck(2) distance to Motorcycle(3): 15.0
Motorcycle(3). Speed 9.0. Actual position: 45.0.
->Engine. Is damaged: false. Is critical: true
->Wheels. Is damaged: false. Is critical: true
  Motorcycle(3) distance to Car(1): 10.7
  Motorcycle(3) distance to Truck(2): 15.0

Starting attack phase
Car(1) fails attack
Truck(2) attacks Car(1) Window
Motorcycle(3) can not attack
End attack phase
Car(1) Window is being repaired 1/2
Ending Turn: 6

Starting Turn: 7
Race with maximum length: 100.0
Car(1). Speed 7.0. Actual position: 41.3.
->Window. Is damaged: true. Is critical: false
->Engine. Is damaged: false. Is critical: true
->Banana dispenser. Is damaged: false. Is critical: false
->Wheels. Is damaged: false. Is critical: true
  Car(1) distance to Truck(2): 5.2
  Car(1) distance to Motorcycle(3): 12.7
Truck(2). Speed 6.0. Actual position: 36.0.
->Engine. Is damaged: false. Is critical: true
->Window. Is damaged: false. Is critical: false
->Banana dispenser. Is damaged: false. Is critical: false
->Wheels. Is damaged: false. Is critical: true
  Truck(2) distance to Car(1): 5.2
  Truck(2) distance to Motorcycle(3): 18.0
Motorcycle(3). Speed 9.0. Actual position: 54.0.
->Engine. Is damaged: false. Is critical: true
->Wheels. Is damaged: false. Is critical: true
  Motorcycle(3) distance to Car(1): 12.7
  Motorcycle(3) distance to Truck(2): 18.0

Not attacking turn
Turn with power ups
Vehicle: Car(1) applying power-up: AttackAllPowerUp
Car(1) attacks Truck(2) Window
Car(1) attacks Motorcycle(3) Engine
Vehicle: Truck(2) applying power-up: SwapPowerUp
Truck(2) was on 36.0 with swap is now on 41.3
Car(1) was on 41.3 with swap is now on 36.0
Vehicle: Motorcycle(3) applying power-up: SwapPowerUp
Car(1) Window is being repaired 2/2
Truck(2) Window is being repaired 1/2
Motorcycle(3) Engine is being repaired 1/3
Ending Turn: 7

Normas de Entrega:

Se deberá entregar:

- Un directorio **src** con el código Java de todos los apartados, incluidos los datos de prueba y testers adicionales que hayas desarrollado en los apartados que lo requieren.
- Un directorio **doc** con la documentación generada.
- Una memoria en formato PDF con una pequeña descripción de las decisiones del diseño adoptado para ejecución de cada apartado, así como con la respuesta a la pregunta del apartado 2, y la explicación del *power up* ideado del apartado 4.
- El **diagrama de clases** final resultante.

Se debe entregar un único fichero ZIP con todo lo solicitado, que deberá llamarse de la siguiente manera: GR<numero_grupo>_<nombre_estudiantes>.zip. Por ejemplo Marisa y Pedro, del grupo 2213, entregarían el fichero: GR2213_MarisaPedro.zip, de manera que cuando se extraiga haya una carpeta con el mismo nombre, y dentro de la misma, el directorio src/, el doc/ y el PDF. El **incumplimiento** de la entrega en este formato supondrá una **penalización en la nota de la práctica**.