ASSIGNMENT 4

Inheritance, interfaces and exceptions

Beginning: week starting on April 5th
Duration: 3 weeks
Delivery: week starting on April 26th
Weight: 30%

The goal of this assignment is the simulation of a race between different types of vehicles using more advanced object-oriented programming techniques than those of previous assignments: inheritance, exceptions and interfaces. We will use these techniques to reduce redundant code, obtain APIs that are easy to extend, and develop a well-structured program that reflects in a direct way all concepts of the domain.

During the development of this assignment the following concepts of Java will be used:

- Inheritance and dynamic binding
- Exception handling
- Use of Java interfaces
- Grouping of related elements into packages

Motivation

One of the most natural examples in which object-oriented software design is specially intuitive is **videogames**, where there are characters and objects with specific attributes that differ from the rest of the ones of the same class, and that interact one with each other for different purposes.

The goal of this assignment is to use that intuition to define our own race between different classes of vehicles. This race will start being very simple, with fast motorcycles that will always win, and slower cars and trucks. However, we will be adding elements that make the race more unpredictable following a **logic similar to games like Mario Kart**: first adding attacks that damage components of other vehicles, and finally adding special abilities, or **power ups**.

In order to do all this we will create the code of the game step-by-step, guided by the Java interfaces that each part of the assignment will propose. We strongly recommend thinking about how the different elements (**vehicles, race, components**, etc…) should interact with each other before writing any code, since the assignment evaluation will take into account not only the code, but also the design and its explanation, which needs to be submitted along the code.

Similar to the previous assignments, it will be essential to design the code modularly exploiting already known concepts of object-oriented programming, like inheritance or polymorphism.

Part 1: Creating vehicles for the race from a text file (2,5 points)

In this assignment we will simulate a race between different types of vehicles. We will consider three types for the moment: trucks, cars and motorcycles. Although each one will have particular characteristics that will make them unique, they share a minimum common structure. The easiest way to show this is through an interface, called IVehicle, defined as follows:

```
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();

        //For Exercise 3
        //public void addComponent(IComponent c) throws InvalidComponentException;
        //public List<IComponent> getComponents();
}
```

IMPORTANT: The interface definitions cannot be modified except for adding javadoc comments. However, you are free to declare additional classes and interfaces if required.

Most methods in IVehicle provide information about the vehicle for the race, and permit modifying it, like the position (getActualPosition(), setActualPosition()); if the vehicle can or cannot move (canMove(), setCanMove()); its maximum speed (getMaxSpeed()); and the name of the vehicle (getName()). In order to distinguish between vehicles of the same type, it is mandatory that every vehicle's name is made of its type followed by a unique identifier inside parenthesis, which should be generated by the system (for example: "Truck(1)"). The addComponent(IComponent c) and getComponents() methods will be explained in part 3 of the assignment, since they are not needed by now.

Every detail of the race -- its length and the participating vehicles -- will be read from a text file, to be processed by the *read* method from class RaceReader. The first line of the file indicates the length of the race. The next lines specify (separated by spaces) the number of vehicles of that class; the type of vehicle, and the maximum speed that type of vehicle can achieve.

The following example shows the file contents we will use for parts 1 and 2. The example race will have length 100, with two cars, one truck and a motorcycle:

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

If the race is too small so that vehicles can't move forward, or there are not enough or too many participants, the race won't make any sense, so we must prevent these situations. As shown in the example below, exceptions must be thrown if the length of the race is less than or equal to the maximum speed of any of the vehicles, or if there are less than two vehicles, or more than 10.

The goal of this first exercise is to prepare all the necessary ingredients to set the race, before making any simulation (this will be done in the next part). As a guide for your design, the next program needs to produce the output below when receiving as an input the path to the text file.

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);
		}
	}
}
```

Expected output:

```
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
```

We will not be programming the simulation part yet within class Race. However, most logic of that class needs to be ready. For this purpose, the output shows the absolute distance of each vehicle to each other.

Please remember that before starting to write any code, it will be essential to plan the design of Race carefully. Think about what is the purpose of that class, as explained in the motivation at the beginning of the text, and how you can apply object oriented principles to achieve clean and extensible code. Don't forget to organize your code in a coherent package structure.

Part 2: Basic simulation of the race (2 points)

In this exercise, we will build the simulator of the race. The race will consist of turns, in which each vehicle will update its position. The race will end when the first vehicle crosses the finish line. Each vehicle has different rules:

- Motorcycles will always move at their maximum speed
- Cars, since they are bigger, will move at their maximum speed with a probability of $p_{car} = 0.9$. The rest of the time, they will move at 90% of their maximum speed.
- Trucks, the biggest of all vehicles, will move at their maximum speed with a probability of $p_{truck} = 0.9$. The rest of the time, they will move at 80% of their maximum speed.

As you may remember, during this assignment we will develop a series of attacks to nearest vehicles, but this will be done in part 3.

As a guide for your design, the following program should produce the outputs below:

```
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);
		}
	}
}
```

The first output lines of the simulation should be the following:

```
---------
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
---------
```

and the last lines:

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

The output format of the numbers should be equal to the one shown.

Try out different configurations of maximum speeds, and check that the fastest vehicle always wins. If motorcycles and trucks maximum speeds stay as in part 1, what is the least maximum-speed for cars to win the race?

Part 3: Attacking the opponents (3 points)

As it can be seen in the last exercises, if we leave the race as it is configured, motorcycles will always win the race. To be more interesting, we will configure vehicles so they can throw banana peels to any nearby vehicle that is ahead of them. When a vehicle is hit by a banana peel, one of its components will be damaged, and because of that, some turns will be lost for its repair.

This way, we will introduce different components into vehicles. For the moment, we will have four basic components: Wheels, Engine, Window and BananaDispenser. As you can see in the IComponent interface below, there is an indication of whether the component is damaged or not:

```
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();
}
```

Damage to components such as the wheels or the motor, which will completely impede progress of the vehicle, is not the same as damage to the windows or the dispenser. Therefore, if more critical components are damaged (Wheels, Engine), vehicles will have to stop completely to repair them, losing 3 turns. If any other component is damaged, its repair will cost 2 turns for windows, and 4 for dispenser, but the vehicle will be allowed to move in the meanwhile. While the banana peel dispenser is damaged or being repaired, that vehicle **won't be able to attack** until it is fully repaired.

Given these components, motorcycles lose their advantage, since they cannot have windows nor dispensers. The rest of the vehicles will have all components. In order to work with components, you will have to use the methods addComponent and getComponents defined in the IVehicle interface.

Therefore, the race now has the following structure:

- Start in turn 1
- The current state of the race must be shown
- Attacking phase (available every 3 turns, starting on 3: 3, 6, 9, …): all vehicles throw their banana peels following the same order as their inclusion in the race.
- Repairing phase: all vehicles check if they have been shot, and start their respective repairs of every damaged component if necessary.
- Position updating: vehicles update their positions if their most critical components are not damaged.

The logic of the attacking phase is the following:

- A vehicle can attack if its banana dispenser is available (undamaged)
- The target vehicle will be the vehicle that is immediately ahead. If there are two or more vehicles at the same distance, one of them is selected randomly.
- The target vehicle must be at 30 units away or less
- A vehicle cannot harm itself
- Any attack has a probability of success of 50%. If the attack fails, nothing happens.
- If the attack is successful, a random component is damaged. If it hits an already damaged component, the number of repairing turns resets, it is not cumulative.

We will create a new input file for parts 3 and 4, which also contains the components that each type of vehicle has, specified after the maximum speed.

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

Take into account that the class RaceReader must be capable of reading both this file format and the format from parts 1 and 2. Since motorcycles have just engines and wheels, an InvalidComponentException must be thrown by method addComponent (from IVehicle) if an invalid component is detected. Such exceptions should inform of the error with the message "Component XXX is not valid for Vehicle XXX(X)".

As a guide for your design, the following main should produce the output below.

```
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);
		}
	}
}
```

The new developed code should not modify the expected output from previous parts. The output for this part should print the components of each vehicle, and in addition:

- At the beginning of every turn, in the "state of the race" phase, for every vehicle, after printing its maximum speed and current position, and before printing distance to the rest of vehicles, a check of all its components, indicating if they are damaged. Example of first output lines, where the new lines are printed in green:

```
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
---------
```

- Separate with empty line the output of the next phases
- In the attacking phase, display if the attack of each vehicle was or wasn't successful
- If it is not an attacking turn, print "No attacking turn" before the repairing phase
- In the repairing phase, show what vehicles are being repaired, and its repairing stage.

The following is an example output of the attacking and repairing phases. Remember that turns 1 and 2, for example, don't have attacking phases, so attacks and repairing start on the 3rd turn. Since the race contains probabilistic elements, your results can be different from this:

```
[...] // 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): 3.2
  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): 3.2
  Truck(2) distance to Motorcycle(3): 7.2
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): 7.2

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): 4.2
  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): 4.2
  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
---------
```

The output indicating the winner is different from the previous exercise:

```
[...]
---------
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
```

Part 4: Adding special abilities or power ups (2,5 points)

Our race has now dynamic components, but we will improve it adding *power ups*, which are special abilities for the vehicles. Power ups should follow this interface:

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

We will consider two types of power ups:

- *Swap*: swap positions with the vehicle that is immediately ahead. If there are several vehicles ahead at the same distance, it randomly picks one of them.
- *AttackAll*: throws banana peels to all vehicles. It doesn't matter the separation distance, nor if they are ahead or back, nor the type of vehicle (motorcycles don't have banana dispensers but could have this power up too). However, the probability of success is 50%.

In addition to these two power ups, you have to propose and design one more power up, that can be anything you can imagine except "win the race", or variations of Swap and AttackAll. Explain briefly your power up in the report: what it does, its name, and how you implemented it.

The way to use power ups will be the following: in non-attacking turns, with a probability of 10% all vehicles will activate randomly one of the three available power ups, so the race will have this final structure:

- A new turn starts
- The current state of the race is shown
- Attacking phase (every 3 turns) / Power up phase (the other turns, but starting from the first attack turn)
- Repairing phase
- Update positions

The main for this part is the following:

```
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);
                }
        }
}
```

Since you have designed your own power up, the following output is orientative. Please describe the output in the report.

Example of output indicating 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): 4.3
  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): 4.3
  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): 5.3
  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): 5.3
  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

Submission rules:

You should submit:

- A src directory with all the Java code in its final version of part 4, including the test data and additional testers that you developed in the sections that require it.
- A doc directory with the generated documentation
- A PDF file with the class diagram, an explanation and a brief justification of the decisions that have been made in the development of this assignment, the main problems that have faced and how they have been solved, as well as the pending problems. It should include the answer to question in part 2, and an explanation of your power up of part 4.

Package everything in a single ZIP file, with name GR<group_number>_<student_names>.zip. For example Marisa and Pedro, from group 2213, would hand in the file: GR2213_MarisaPedro.zip. Inside the zip file there should be a directory src/, a folder doc/ and the PDF. Failure to deliver in this format will result in a penalty in the mark.