Fourth Assignment

Inheritance, interfaces and exceptions

Kevin de la Coba Malam
Marcos Bernuy López

Section 1

In this section we first created the folder vehicle which will be the package where the interface IVehicle with the classes which will implemented where situated. These classes which we created represent the vehicles which must be created after reading the txt file. We decided to create an abstract class called Vehicle which implements the interface and from which it inherits into de subclasses Motorcycle, Truck and Car. We decided to create an abstract class, to decrease the repetition of code, since the vehicles implemented part of the functions exactly the same way. It was also needed to create the id of the vehicles created which needed a private static variable shared by all vehicles. Once this classes were implemented, we implemented the class RaceReader which implements the function read() which reads the data from the txt file and returns a Race, an object from the Race class. We implement this function making it read each line splitting the words received, this way we can get the quantity, and the maximum speed of each vehicle. Then we just make a loop to create each of the vehicles adding them to an ArrayList. Once we have the ArrayList and the length of the race, we can create the race which will be the return of the function read() Finally in this part we implement the exceptions package in which we implement the RaceException.java which extends from the Exception class so not many changes are needed.

Section 2

In this part we have to implement the function simulate() from the Race class, which simulates the race. In this simulate function we had to take a couple things into account. First of all, we had to implement the probability of the vehicles which participate in the race as it is asked. To make this possible, we implemented the function getRealSpeed() in the Vehicle class and its subclasses which returns a double with the speed after using the Math.random() function to represent the probability of getting the maximum speed. Once the real speed was calculated, we just had to change the actual position to its position plus the actual speed. Afterwards we made sure the output was printed correctly and exactly as the example given.

Section 3

This section's goal was to add components to the already created vehicles in the program. For this, we created first an abstract class *component* which implemented the *IComponent* interface.

With this solution the repetition of code is reduced, but we must do some castings. All the vehicles have *IComponents*, not *Components*, in the *Components* class there are defined some methods which cannot be called from an *IComponent* reference. If we decide to not implement that abstract class, we would have the problem of having repeated code in each new component that we create.

The other part of this section was implementing an attacking phase, for this we added some code in the *simulate* function in the class *race*, basically we allowed the race to have a phase each 3 turns in which attacks can be done. Repairing actions were also considered and added.

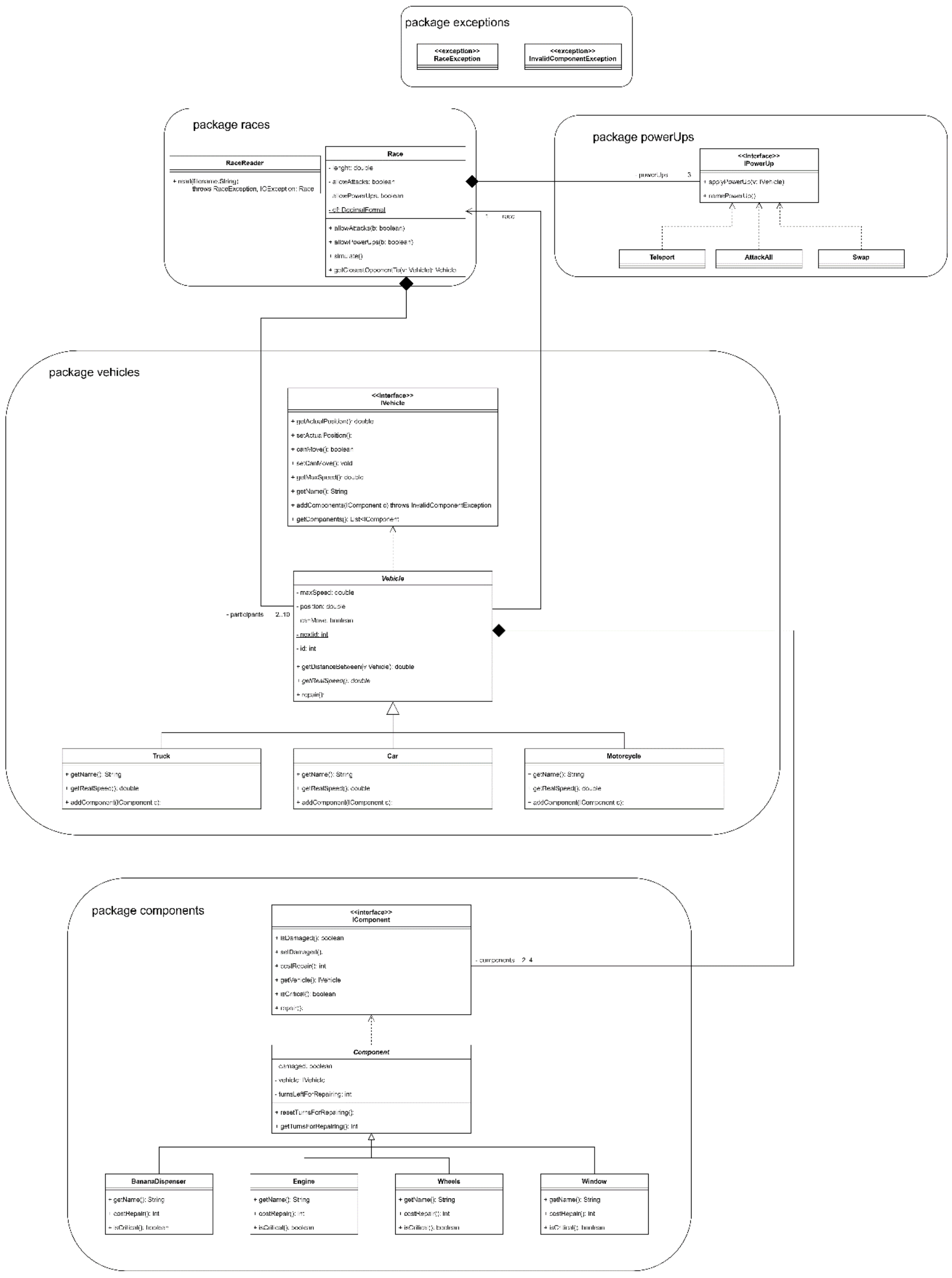
Section 4

Adding power-ups was the goal of this section, for this purpose we created the requested power-ups by simply creating one class for each one, and each class must implement the *IPowerUP* interface.

In this part we decided to not implement an abstract class because the power-ups are very simple, the needed methods are just the ones stated in the interface.

The power-up we implemented was **Teleport**, it's a simple power up that allows a vehicle to teleport into the position it would have been in the next turn, if the speed were the maximum, it's something like skipping a turn and moving into that position.

Class Diagram


package exceptions
<<exception>> RaceException
<<exception>> InvalidComponentException
package races
RaceReader
+ read(filename:String): throws RaceException, IOException: Race
Race
- lenght: double
- allowAttacks: boolean
- allowPowerUps: boolean
- df: DecimalFormat
+ allowAttacks(b: boolean)
+ allowPowerUps(b: boolean)
+ simulate()
+ getClosestOpponentTo(v: Vehicle): Vehicle
1 race
package powerUps
- powerUps 3
<<Interface>> IPowerUp
+ applyPowerUp(v: IVehicle)
+ namePowerUp()
Teleport
AttackAll
Swap
package vehicles
<<Interface>> IVehicle
+ getActualPosition(): double
+ setActualPosition():
+ canMove(): boolean
+ setCanMove(): void
+ getMaxSpeed(): double
+ getName(): String
+ addComponents(IComponent c) throws InvalidComponentException
+ getComponents(): List<IComponent
Vehicle
- maxSpeed: double
- position: double
- canMove: boolean
- nextId: int
- id: int
+ getDistanceBetween(v Vehicle): double
+ getRealSpeed(): double
+ repair():
- participants 2..10
Truck
+ getName(): String
+ getRealSpeed(): double
+ addComponent(IComponent c):
Car
+ getName(): String
+ getRealSpeed(): double
+ addComponent(IComponent c):
Motorcycle
+ getName(): String
+ getRealSpeed(): double
+ addComponent(IComponent c):
package components
<<interface>> IComponent
+ isDamaged(): boolean
+ setDamaged():
+ costRepair(): int
+ getVehicle(): IVehicle
+ isCritical(): boolean
+ repair():
- components 2..4
Component
- damaged: boolean
- vehicle: IVehicle
- turnsLeftForRepairing: int
+ resetTurnsForRepairing():
+ getTurnsForRepairing(): int
BananaDispenser
+ getName(): String
+ costRepair(): int
+ isCritical(): boolean
Engine
+ getName(): String
+ costRepair(): int
+ isCritical(): boolean
Wheels
+ getName(): String
+ costRepair(): int
+ isCritical(): boolean
Window
+ getName(): String
+ costRepair(): int
+ isCritical(): boolean