!TEX encoding = UTF-8 Unicode

Chapter 1

Assignment 2, part II: Road Fighter Extended

Submission date: 29 November 2021, 9:00

Objective: Inheritance, polymorphism, abstract classes and interfaces.

1. Extensiones

In this assignment, we will extend the code with new functionality. But first a warning:

IMPORTANT: Any of the following, on its own, is sufficient reason to fail the assignment:

- breaking encapsulation,
- the use of methods that return lists,
- the use of instanceof or getClass, since identifying the dynamic types of objects is simply a way of avoiding the use of polymorphism and dynamic binding, i.e. avoiding the use of OOP.

Regarding the last item, use of a "DIY" instanceof (e.g. each subclass of GameElement has a set of methods isX, one for each subclass of GameElement where in class Y, the method isX returns true if $X = Y$ and false if $X \neq Y$) is even worse since it is simply a clumsier, more verbose, way of doing the same thing.

1.1. The Wall game element, the `shoot` command and the InstantAction interface

Wall. We first add a new type of obstacle called Wall having the same behaviour as Obstacle except for the fact that it has a resistance of 3 instead of 1. Three different symbols are used to represent objects of this class, depending on how many lives the Wall object has left (see the file `symbols.txt` to for the exact symbols). As explained below, the player obtains coins from destroying obstacles by shooting them, in particular, the

player obtains 5 coins from destroying a wall (no coins are obtained from destroying a normal obstacle).

Shoot command. The `shoot` command removes a life from the first obstacle found in the same lane to the right of the player and costs the player 1 coin. The range of a shot is the visible part of the road, i.e. it has no effect on obstacles that are not inside the visibility. Note that use of the shoot command does not involve a cycle. Here follows a sample execution:

Collider interface. The `shoot` command only affects obstacles. To implement a behaviour which only affects certain game elements, we can use dynamic binding. To this end, to the Collider interface we add a method receiveShot() that can be given a default empty body in GameElement to be overwritten in the appropriate GameElement subclasses.

InstantAction interface. The `shoot` command does not create a bullet object that travel towards the obstacle and then collides with it but, instead, acts instantaneously like a laser. We create a new interface called InstantAction to represent the propagation of an effect to a game element without there being a collision.

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

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

The implementation of the execute method in each concrete instantaneous action, in this case the ShootAction, determines which game element (or elements) is affected by that action, in order to call the appropriate method, in this case receiveShot(), on that game element (or those game elements). The instantaneous action objects should be created in the corresponding command.

1.2. The SuperCoin game element

Next, we create a new game element subclass called SuperCoin, represented by the symbol $, which gives the player 1000 coins if the car drives over it. There can be only one supercoin per game; one way of ensuring this is by endowing the SuperCoin class with a static boolean attribute. When a supercoin is present on the road, the information Supercoin is present must be displayed each time the road is displayed. Here follows an example in which the car drives over a supercoin:

1.3. The Turbo game element

When the car passes over this game element it jumps forward three squares; note that this does not change the requirement that the display always show the car remains in the leftmost column. If, in the process of jumping forward three squares, the car passes over some other game element, this has no effect (there is no collision or coin grabbing), including if that game element is on the target square of the jump. In the latter case, the car and the other game element share the same square (the car is displayed). The Turbo element is represented by the symbol >>>. Here follows an example in which the car drives over a turbo:

1.4. The `clear` command and the `cheat` command

We now create two commands for use in debugging.

The `Clear` command. This command eliminates all game elements (except the car) from the road. The abbreviated form of this command is simply 0.

The `cheat` command. This command is used to add an advanced game element, i.e. an object of one of the GameElement subclasses appearing in the list given below, some of which have already been introduced and the rest of which will be introduced later in this document. The execution of this command results in the addition of a GameElement object of the chosen subtype in a random lane of the last visible column, after first deleting any game elements already present in that column. The advanced game element to be added is identified by its name: `wall`, `turbo`, `super`, `truck` and `ped`, respectively.

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

Since this is a debugging command, to avoid spending too much time implementing it, you are allowed to use a method in GameObjectGenerator (called from the execute method of the CheatCommand class) which contains a switch on the possible advanced game element names, in each case of which a GameElement of the type corresponding to that name is created (before calling a method of Game to introduce this new GameElement object into the game). However, a much better implementation would be to introduce the following code:

- a NAME constant in each GameElement subclass similar to the NAME constant in each Command subclass,
- a name attribute in GameElement similar to the name attribute in Command,
- a parse method in GameElement similar to the parse method in Command that uses this name attribute and is inherited by each of these game element subclasses,
- a static attribute AVAILABLE_GAMEELEMENTS in GameObjectGenerator similar to the AVAILABLE_COMMANDS attribute of Command,
- a static method getGameElement in GameObjectGenerator similar to the getCommand method of Command.

1.5. The `wave` command

This command, which costs the player 5 coins, pushes all the game elements within the visibility (except the car) one square to the right on the road. If a game element has another game element immediately to its right, the command has no effect. Note, however, that the order of updating of the game elements on the board may affect how many of them can be pushed to the right. You should try to update the elements in order to maximise the number that can be pushed to the right. To implement this command, you should create

another instantaneous action as was done for the `shoot` command. The abbreviated form of this command is `"w"`.

Here follows an example of the use of this command:

1.6. The `grenade` command and the Grenade game element

This command, which costs the player 3 coins, creates a grenade object at the position on the road provided as a parameter to the command; in fact, two parameters must be provided, the x and y coordinates, where the x coordinate is relative to the position of the car and where the value of x is less than or equal to the visibility. If there is already an object at the position provided as a parameter, the command has no effect. As for any command with parameters, the GrenadeCommand class will need a parse method that is different to the one in the Command class, i.e. it will need to overwrite the parse method of the Command class (and, to avoid writing *fragile code*, parse methods for commands with parameters should not return the value this).

The behaviour of a grenade object is as follows:

- It contains a cycle counter initialised to 3, which explodes when the value reaches 0[1].
- When the grenade explodes it causes damage of 1 life to all the obstacles (including walls) in its vicinity.

The impact of the explosion should be implemented by introducing a method receiveExplosion in the *Collider* interface. In those classes that are affected by the explosion, since the effect is the same as that of a shot, receiveExplosion can simply call receiveShot. A grenade is repesented by the symbol ð followed by, in square brackets, the number of cycles left until it explodes.

We would also like to implement the propagation of the explosion as an instantaneous action. However, the instantaneous actions defined so far either assume that the position of origin of the action is that of the player (ShootAction) or that the instantaneous action has no position of origin (WaveAction), whereas the position of origin of the explosion action is clearly that of the grenade object causing the explosion. For this type of instantaneous actions, we can simply define a constructor in the action class and use it to pass the action the position of its origin. In the case of the grenade, the ExplodeAction object should be so created by the constructor of the Grenade class (where an object of the Grenade class is itself created in the execute method of the GrenadeCommand class).

Here follows an example of the use of the `grenade` command:

1.7. Interface Buyable

There are various commands that share the characteristic of having a cost, so we can represent this behaviour in another interface called Buyable to be implemented by the Command class:

```
package supercars.control;

import supercars.logic.Game;

public interface Buyable {
```

[1] *hint*: ensure that a grenade whose counter is zero is considered dead, then call the explode method from the onDelete method of the Grenade class

```
    public int cost();

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

The body of the buy() method will decrease the coins of the player by the appropriate amount. In this case, instead of specifying the default behaviour in the abstract class Command, as we could have done, we have lifted it to the interface itself by using a default method. If the default behaviour is that of the commands that do not (resp. do) have a cost, the commands that do (resp. do not) have a cost will need to overwrite this default behaviour.

1.8. Game elements that move

All the game elements created until now (except the player) do not move. We now create two game elements that have their own movement.

The `Truck` game element: objects of this class advance from right to left one square each turn. They can collide with the player in the same way as an obstacle but they do not collide with obstacles. If, on a given cycle, the truck shares a square with another game element, either of the two can be represented on the screen as occupying that square (this will be dictated by the order of the game elements, itself dictated by the order of introduction into the game). You must not allow the the player and a truck to pass through each other without colliding; this will involve checking for collisions twice on every cycle. A truck object is represented by the symbol "←".

Here follows an example:

The `Pedestrian` game element: objects of this class are obstacles that constantly changing lanes in the same column of the road (not of the display), alternating between the directions upwards and downwards. If the car collides with a pedestrian, the game ends as just as it does if the car collides with an obstacle and, in addition, the player loses all their coins.

Here follows an example:

1.9. Thunder Action

... to be continued