Università degli Studi di Verona

DIPARTIMENTO DI INFORMATICA
Corso di Laurea in Informatica

Software Engineering

Second project report

Candidati:
Giovanni Liboni
Matricola VR363021

Enrico Giordano
Matricola VR359169

Alberto Marini
Matricola VR359129

Alessandro Falda
Matricola VR359333

Anno Accademico 2013-2014

Contents

Part I
Introduction

This project implements an asynchronous system consists in three principal parts:

1. a station, that controls velocity of cars;
2. some automatic cars, that set their velocity randomly during the ride and set the ideal velocity thanks to the station;
3. some manual cars, that set their velocity randomly during the ride and receive "break" message (but they are not obliged to slow down).

During the execution, is istantiated 50 manual cars and 40 automatic cars and change the speed of each car randomly during the runtime. After a simple ride, the cars decide randomly if they will be exit or not.

Part II
Graphical interface

The graphical interface is built with *swing* Java library and consists in two JFrame called "WallGraphic" and "DebugInterface". The first interface (WallGraphic) is a representation of the situation, composed by a station, some automatic cars and some manual cars. The second interface (DebugInterface) is a textual console that show the program flow, cars display and state and station state. This interface has three option:

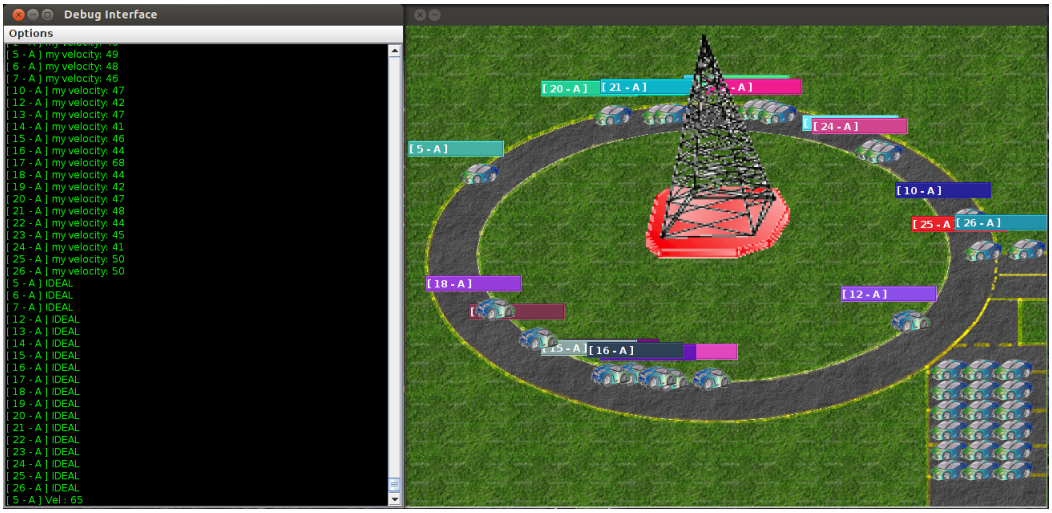

Figure 1: ScreenShot

1. "pause", that stops the program flow acquisition;
2. "resume", that resume the program flow acquisition (and enable autoscroll);
3. "watch", that disable or enable the autoscroll of the scrollbar.

Only 200 rows are shown in the screen; after 200 rows, DebugInterface clean itself and delete old rows. This was made because after 200 setText in the same JLabel probably will crash the program.

This is composed by a JFrame that contains a JPanel that contains a JLabel with a black background and a text that was updated by station and cars display.

WallGraphic is composed by a JFrame that contains a JPanel with a "wallpaper" (the circuit). This JPanel contains in different levels (Z ordered) a station (that is a JLabel with an image) and some cars (that are a JLabels with an image). The cars move theirself in asynchronous way into the ride in 10 different direction (in order to approximate the elliptical path).

The car label have a superior JLabel that contains its id and the car type: the "M" letter represent the "Manual car" and the "A" letter represent the "Automatic car". When a car change its direction (X orientation), it changes its image.

Figure 2: car label

Every car is a graphical thread that execute the move() method in order to move itself during the ride; when a car exit to the circuit, the thread dead. The velocity is graphically represented by a thread sleep during the movement.

Figure 3: car label (different orientation)

Figure 4: station label

This is the Z order of the JLabel:

-1: background image;

0: station;

-1: cars;

In this way, the cars will pass graphically behind the station and on the background image.
In the park, there is a lot of "dead car" that simply occupies the park. This was made because is a graphical strategy to fill a space that may seem empty or messy.

The graphical class are contained in the *"graphics"* package.

1 Design pattern for graphical interface

Every class of this project must use theese graphical interface, so it was implement a **Façade pattern**. In fact there is a general class, "ScenarioGraphic", that include in itself every graphical istance; in this way, when a class must use a graphical istante, simply call the ScenarioGraphic methods and ScenarioGraphic can modify the graphical istances.

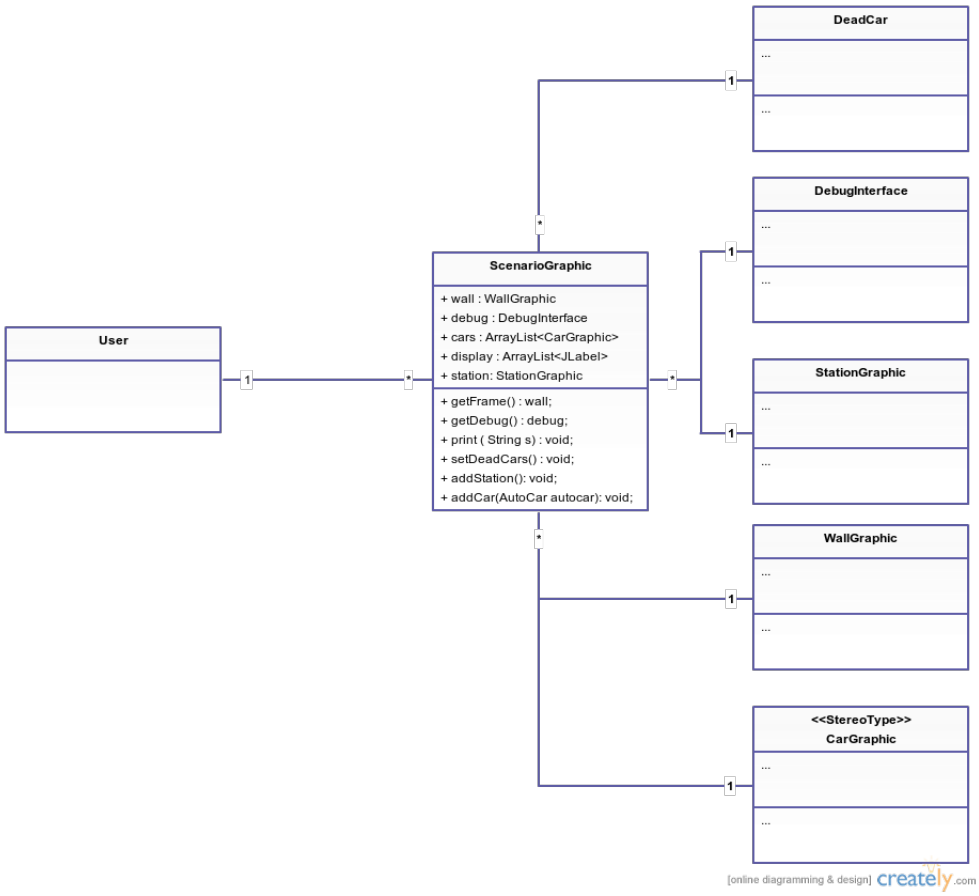

Figure 5: Façade pattern

2 Swing bug

The Swing graphical library is a simple and usefull library for java application, but in this project it was used at the limit of its potential. So, every car has two graphic threads that work asynchronously with a fast refresh (20 milliseconds). So, during the execution the GUI may be throws this exception:

```
Exception in thread "Thread-1" java.lang.NullPointerException
at javax.swing.BufferStrategyPaintManager.flushAccumulatedRegion
...
```

This is a bug of Swing library of the management of asynchronous threads that have a very high refresh. But is not important for the program execution, simply is rarely thrown this exception.

Part III
AutoCar and ManCar Classes

These classes are implementations of Runnable and InterfaceAutoCar interfaces (and the respective InterfaceManCar) for these reason:

- contain a method

  ```
  public void run()
  ```

  which represents what each machine must do from the point of view of the movement (i.e. run the route in the circuit);

- implement the corresponding classes of their interface.

Implement two different interfaces is useful to represent a different behavior for the "break" message received from the station and in particular for using **Adapter** design patterns type. In fact, to use a different method you can not make an override and so you have to rely on a intermediate class (called AdapterAutoToManual) that extends the calling class but implements the class which you want to inherit the behavior.

So in fact there are two distinct interfaces to AutoCar and Mancar that are implemented by the respective classes and have different behaviors; if you want to use the AutoCar as a ManCar you must instantiate the AdapterAutoToManual class and use through it the ManCar class methods.

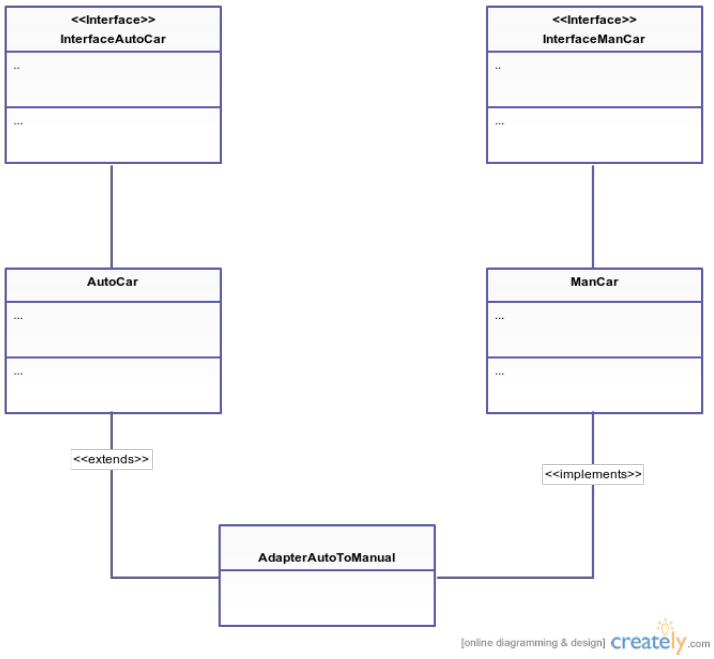

Figure 6: Adapter pattern

To make possible the use of graphics part by AutoCar and ManCar classes, this class can instantiate in it the "CarGraphic" graphics instance, to have full control over their graphics part.

This was useful for managing the asynchronously machine movements compared the other machines and the execution of the program too. Moreover, in this way is allowed to ScenarioGraphic class to have control of behavior and the graphics parts of both AutoCar and Mancar classes; in fact, by instantiating Mancar or AutoCar class it automatically creates the graphic instance, so you can manage it by accessing the individual instances of their classes.

This make useful to use **Façade Pattern**, because you want to use a single class to control all graphics instances (where permitted).