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:

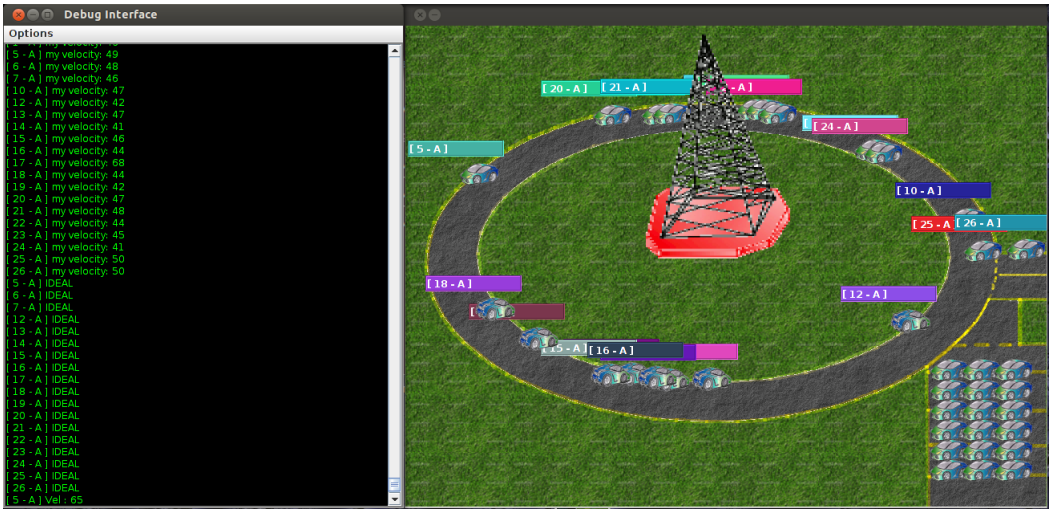

Figure 1: ScreenShot

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.

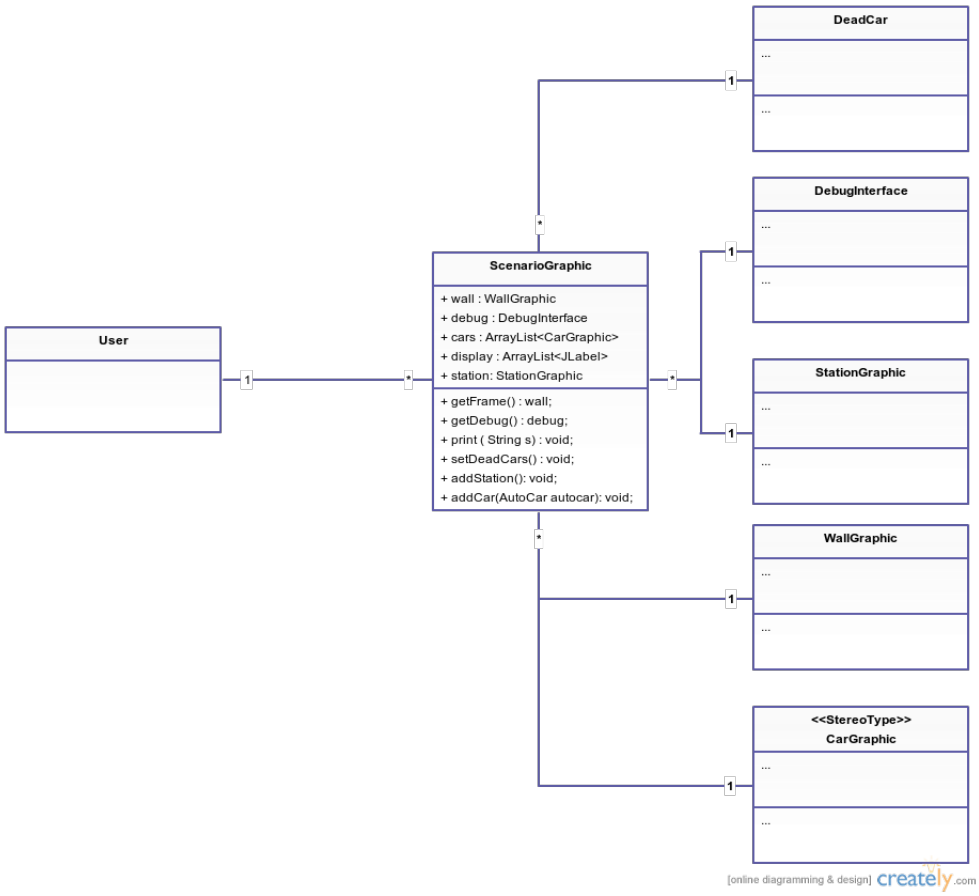

Figure 5: Façade pattern

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 Class

Queste classi sono delle implementazioni delle interfacce di tipo Runnable e InterfaceAutoCar (e il rispettivo InterfaceManCar) per questi motivi:

- contengono un metodo

  ```
  public void run()
  ```

  che rappresenta ciò che ogni macchina deve fare dal punto di vista del movimento (cioè eseguire il tragitto nel circuito);

- implementano le corrispettive classi della loro interfaccia.

Il fatto di implementare due interfacce differenti è stato utile per rappresentare un comportamento diverso come da specifiche per il messaggio di "break" ricevuto dalla stazione e in particolar modo per utilizzare il design pattern di tipo **Adapter**. Infatti avendo due implementazioni differenti, per utilizzare diversamente un metodo non è possibile fare un Override e quindi bisogna appoggiarsi ad una classe intermedia (AdapterAutoToManual) che estende la classe chiamante ma implementa la classe da cui si vuole ereditare il comportamento.

Quindi di fatto esistono due interfacce distinte di AutoCar e ManCar che vengono implementate dalle rispettive classi e che hanno comportamenti differenti; se si vuole utilizzare la AutoCar come ManCar bisogna istanziare la classe AdapterAutoToManual e utilizzare tramite essa i metodi della classe ManCar.

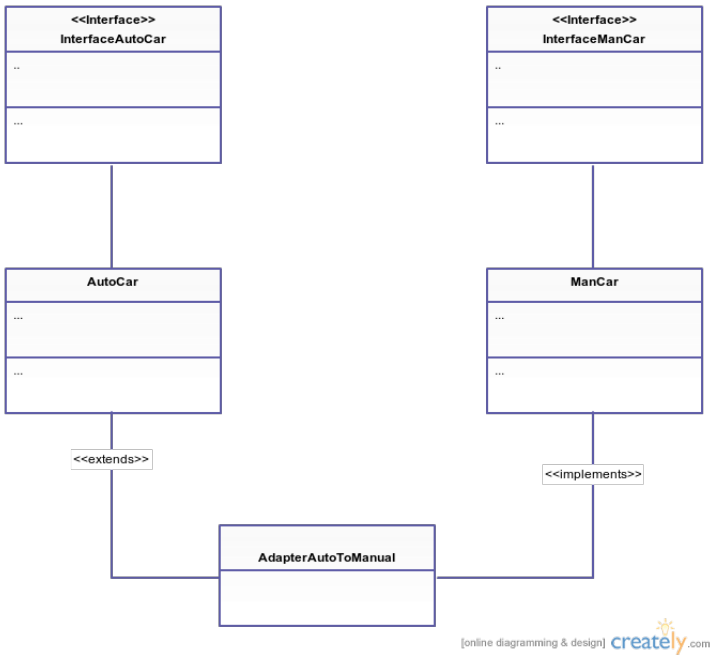

Figure 6: Adapter pattern

Per rendere possibile l'utilizzo della propria parte grafica da parte della classe AutoCar e ManCar, è stato fatto in modo che questa classe istanziasse dentro di essa l' istanza grafica "CarGraphic", in modo da avere pieno controllo sulla propria parte grafica.

Questo è stato utile per gestire il movimento delle macchine in maniera asincrona rispetto alle altre macchine ma anche all'esecuzione del programma. Inoltre in questo modo anche alla classe ScenarioGraphic è permesso di avere controllo del comportamento e della parte grafica sia delle AutoCar sia delle ManCar; istanziando infatti la classe AutoCar o ManCar, si crea in automatico l'istanza grafica, quindi è possibile gestirla facendo accesso alle singole istanze delle loro classi.

Questo rende inoltre utile l'utilizzo del **Façade Pattern**, perché si vuole utilizzare una singola classe per controllare tutte le istanze grafiche (dove è permesso).