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 and consists in three principal parts:

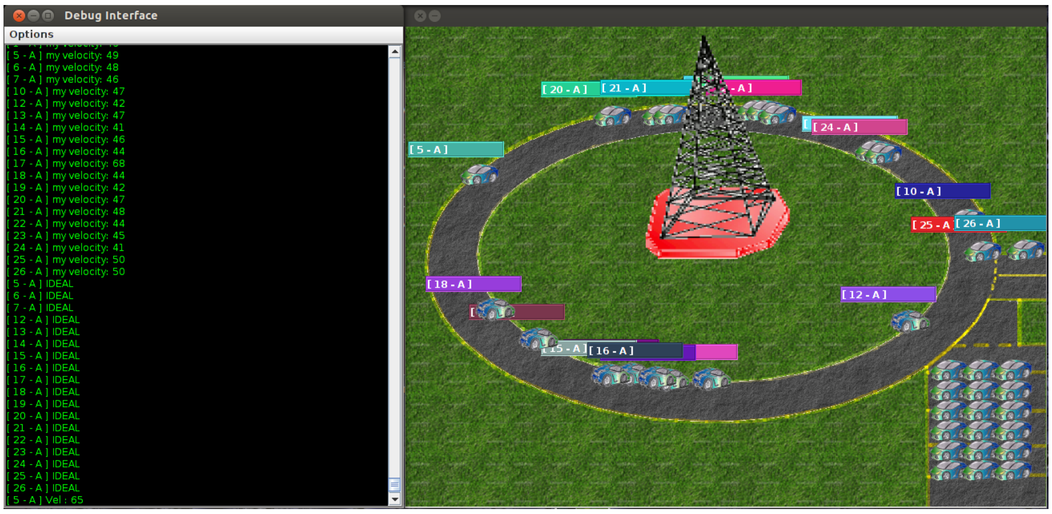
1. a station, that controls speed of cars;
2. some automatic cars, that set their velocity randomly during the ride and gain the ideal velocity thanks to the station;
3. some manual cars, that set their velocity randomly during the ride and receive “break” message (but they are not obliged to slow down).

During the execution, 50 manual cars and 40 automatic cars are instantiated; during the runtime their speeds changes randomly. After a simple ride, the cars decide randomly if they will exit or not.

Part II

Graphical interface

The graphical interface is built with *swing* Java library and consists of 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 shows the program flow, cars' display and state and station state. This interface has three option:



```
[ 5 - A ] my velocity: 49
[ 6 - A ] my velocity: 48
[ 7 - A ] my velocity: 46
[ 10 - A ] my velocity: 47
[ 12 - A ] my velocity: 42
[ 13 - A ] my velocity: 47
[ 14 - A ] my velocity: 41
[ 15 - A ] my velocity: 46
[ 16 - A ] my velocity: 44
[ 17 - A ] my velocity: 68
[ 18 - A ] my velocity: 44
[ 19 - A ] my velocity: 42
[ 20 - A ] my velocity: 47
[ 21 - A ] my velocity: 48
[ 22 - A ] my velocity: 44
[ 23 - A ] my velocity: 45
[ 24 - A ] my velocity: 41
[ 25 - A ] my velocity: 50
[ 26 - A ] my velocity: 50
[ 5 - A ] IDEAL
[ 6 - A ] IDEAL
[ 7 - A ] IDEAL
[ 12 - A ] IDEAL
[ 13 - A ] IDEAL
[ 14 - A ] IDEAL
[ 15 - A ] IDEAL
[ 16 - A ] IDEAL
[ 17 - A ] IDEAL
[ 18 - A ] IDEAL
[ 19 - A ] IDEAL
[ 20 - A ] IDEAL
[ 21 - A ] IDEAL
[ 22 - A ] IDEAL
[ 23 - A ] IDEAL
[ 24 - A ] IDEAL
[ 25 - A ] IDEAL
[ 26 - A ] IDEAL
[ 5 - A ] Vel : 65
```

Figure 1: ScreenShot

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

The DebugInterface is composed by a JFrame that contains a JPanel that contains a JTextArea with a black background and a text that is updated by both the station and the cars' displays.

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 JLabels with an image). The cars move in asynchronous way into the ride in 10 different direction (in order to approximate the elliptical path).

On the car label there is a Jlabel showing its id and 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), its image changes.

Figure 2: car label

Every car is a graphical thread that execute the move() method in order to change the position forward the ride; when a car exits the circuit, the thread dies. The speed is represented graphically 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 are a lot of "dead cars" that simply stand still in there. This was made to fill a space that could seem empty or messy.

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

1 Design pattern for graphical interface

Every class of this project must use these graphical interface, so a **Façade pattern** was implemented. In fact there is a general class, "ScenarioGraphic", including every graphical instance; in this way, when a class needs to use a graphical instance, it simply calls the ScenarioGraphic methods and ScenarioGraphic can modify the graphical istances.

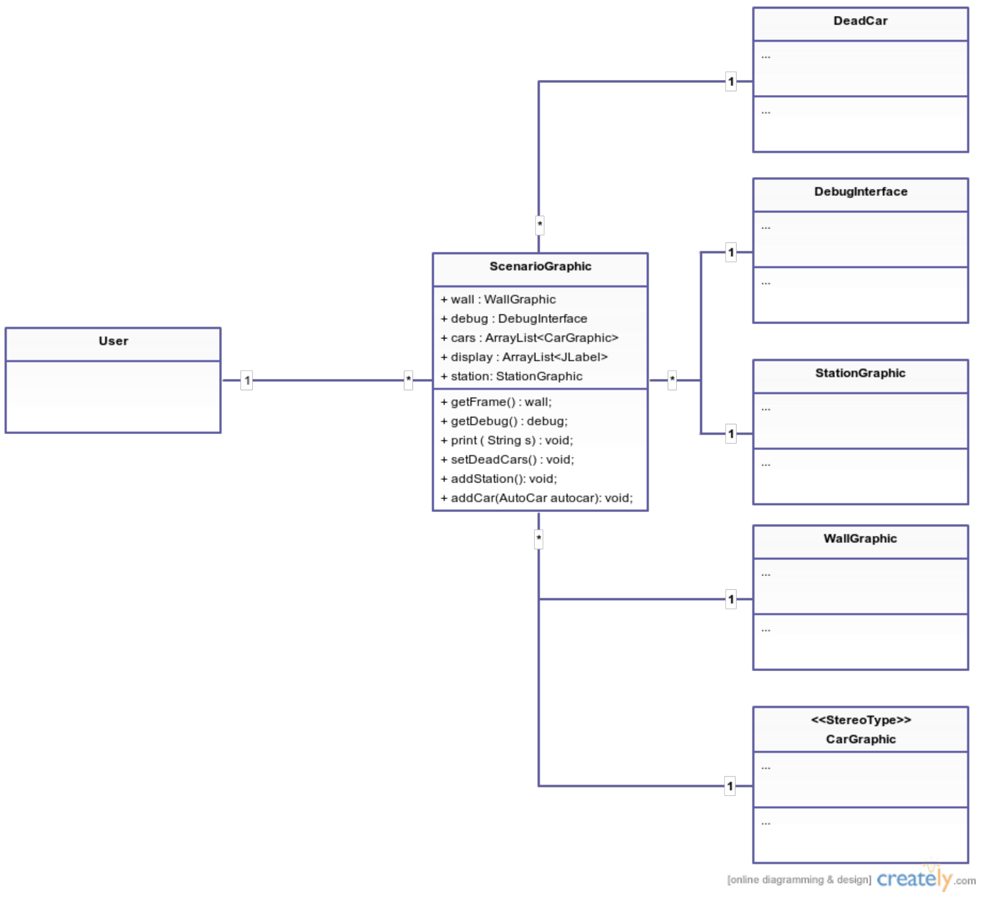


Figure 5: Façade pattern

2 Swing bug

The Swing graphical library is a simple and useful library for java applications, 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). During the execution the GUI could show this exception:

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

This is a bug of Swing library dealing with the management of asynchronous threads that have a very high refresh. Anyway it is not important for the program's execution, given its rare occurrence.

Part III
AutoCar and ManCar Classes

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

- they contain a method

  ```
  public void run()
  ```

  which represents the mouvement of each car (i.e. run the route in the circuit);

- they implement the corresponding classes of their interface.

Implementing two different interfaces is useful in order 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 cannot make an override and so you have to rely on an intermediate class (called AdapterAutoToManual) that extends the calling class but implements the class which from 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.

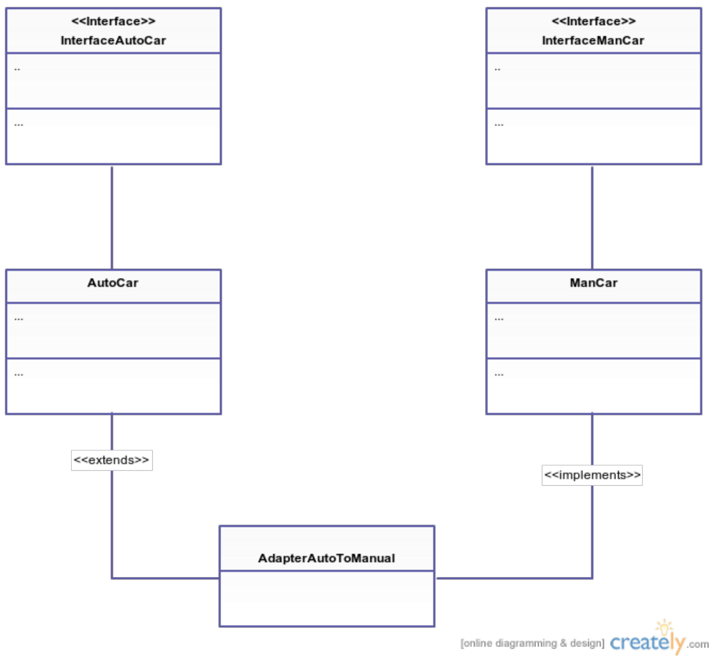


Figure 6: Adapter pattern

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

This was useful for managing the asynchronously car movements compared the other cars, and for the execution of the program too. Moreover, in this way ScenarioGraphic is allowed to have control of behavior and the graphical 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.

As a consequence the use of **Facade Pattern** is useful, because you can use a single class to control all graphical instances (where permitted)

Part IV
Communication

The communication is based on two main components: the network (called Net) and the communication module (called Comunication). The network stores the cars within the range of action and within the circuit. The communication manages the exchange of messages, in receiving and in transmission, and notifies to the upper component the receiving of a message.

The project specified a limitation of the available band of a station to receive and a limitation of number of car to send. This problem has been simulated using the sleep() method, so we send and receive only the predetermined number of times. During the sleep period, the car can receive messages, thanks to a tail that serves as a buffer. When the car receives a message, the module notifies that to the car using the update() method.

The station receives the messages, stores them into a vector that serves as a buffer and then, when the principal thread wakes up, the station download the vector. So, the received packets are passed to the station to be processed. The station uses a broadcast method to send to all the cars into the circuit the same packet. A package is composed by two objects: the sender's ID and the messages. Every message is composed of the data and the recipient. In this way, you minimize the number of packets sent from the station and you can send more broadcasted messages simultaneously, in order to get faster feedback by cars.

The communication part has been specialized depending on whether the component was a station or a car. The car module will send messages at a rate defined by ts pps; on the other side, the station hasn't this limitation. The network's operation can be grouped into stages: the entry into the field of action of the network by a car, the join request from the station, the station response, the entry in the circuit and the exit from the circuit.

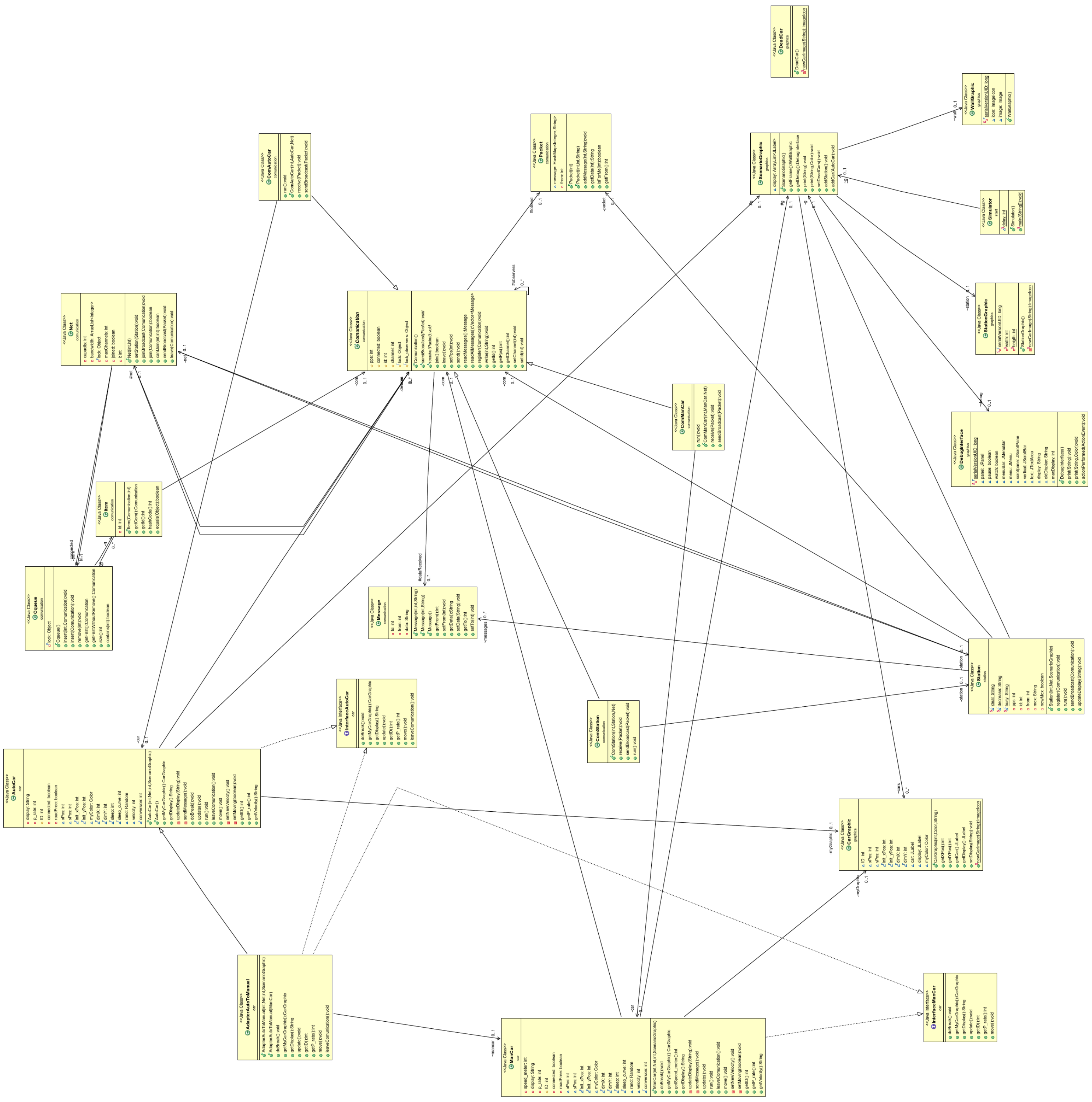
When a car enters into the field of action of the station and its network, the station sends a broadcast message to all cars with a JOIN message. This message will be ignored by the cars already in the circuit and will be managed by the cars into the parking and the cars just entered in the field. When a car receives a station ok to get in, it enters into the circuit and starts sending the data of its speed at the station. When the car must leave the circuit, the car is removed from the cars inside the circuit and inside the range of network.

If a car does not receive the station OK, it goes into the park and it waits a free place into the network. The parking is managed with a FIFO queue, to comply the order of entry into the field. The station sends a broadcast message when a car enters into the field or if there are cars into the parking.

The station manages the received packets from the cars and for each message into the packet responds with an IDEAL message, if the car is going at the right speed, otherwise responds with a BREAK message, if the car is going too fast. It also manages the JOIN request by the cars, by checking if is possible to add them into the network, and, if so, by sending an OK message. In the event that there was no place available, the station sends a BUSY message, denying access to the circuit. The speed of message receiving is simulated by a sleep, calibrated to receive packets 20 times per second. Every time the station reads the messages from the Comunication module, moves all messages in a internal vector and manages them all at once.

The communication module has been implemented using the **observer pattern**: it is a software design pattern in which an object, the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes, usually by calling one of their methods.

Part V
UML Class Diagram


Net
Cqueue
Item
ComAutoCar
Comunication
Packet
Message
InterfaceAutoCar
AutoCar
AdapterAutoToManual
ManCar
CommManCar
ComStation
Station
CarGraphic
InterfaceManCar
ScenarioGraphic
WallGraphic
Simulator
StationGraphic
DebugInterface
DeadCar