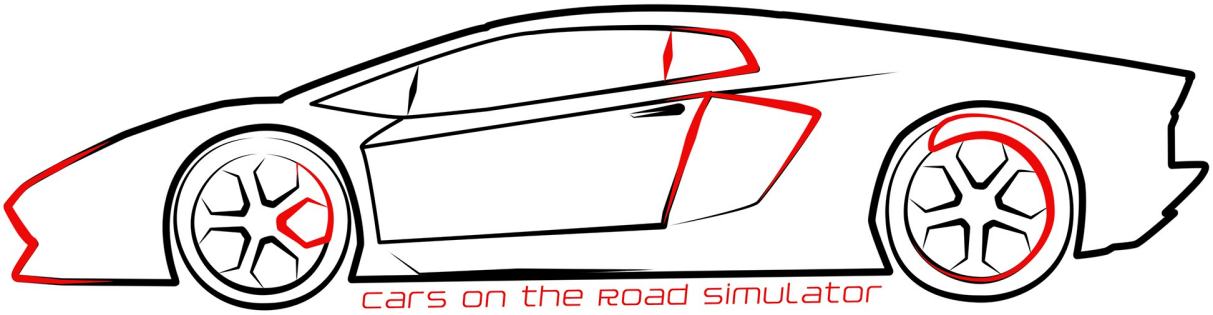
Course of Object-Oriented Programming - Academic Year 2013/2014
Laurea degree course of Computer Engineering and Science - University of Bologna

Report regarding the project CA.R.S.
CArs on the Road Simulator:
road traffic's computer simulation



Project's members: Masini Gioele, Pruccoli Andrea, Zamagna Marco
Available at https://bitbucket.org/GioeleMasini/oop13-cars

1. PROBLEM ANALYSIS

The problem for which this software has been designed is traffic management. Every time the public administration has to approve the project of a new important road it doesn't really know with certainty if it will be a good choice or it should be better to enlarge, reduce, deviate it.
With a good simulation of daily movements it should be possible to know it in advance. This possibility may save public administrations from an inefficient work thanks to the possibility to test it in advance and watch a graphical simulation before accepting the project permanently and starting to work.

The software has to be able to simulate movements of different vehicles, from cars to trucks to motorcycles, which will drive along the road following some grassroots road rules, like stopping at a cross moderated with or without traffic lights, turn the direction depending on the trend of the road, decelerate before stopping and then accelerate.

The user will be able to observe all these dynamic changes thanks by a graphical interface in which is represented the map used in the simulation, this last customized by user, and the vehicles driving on their own.

2. ARCHITECTURAL PLANNING

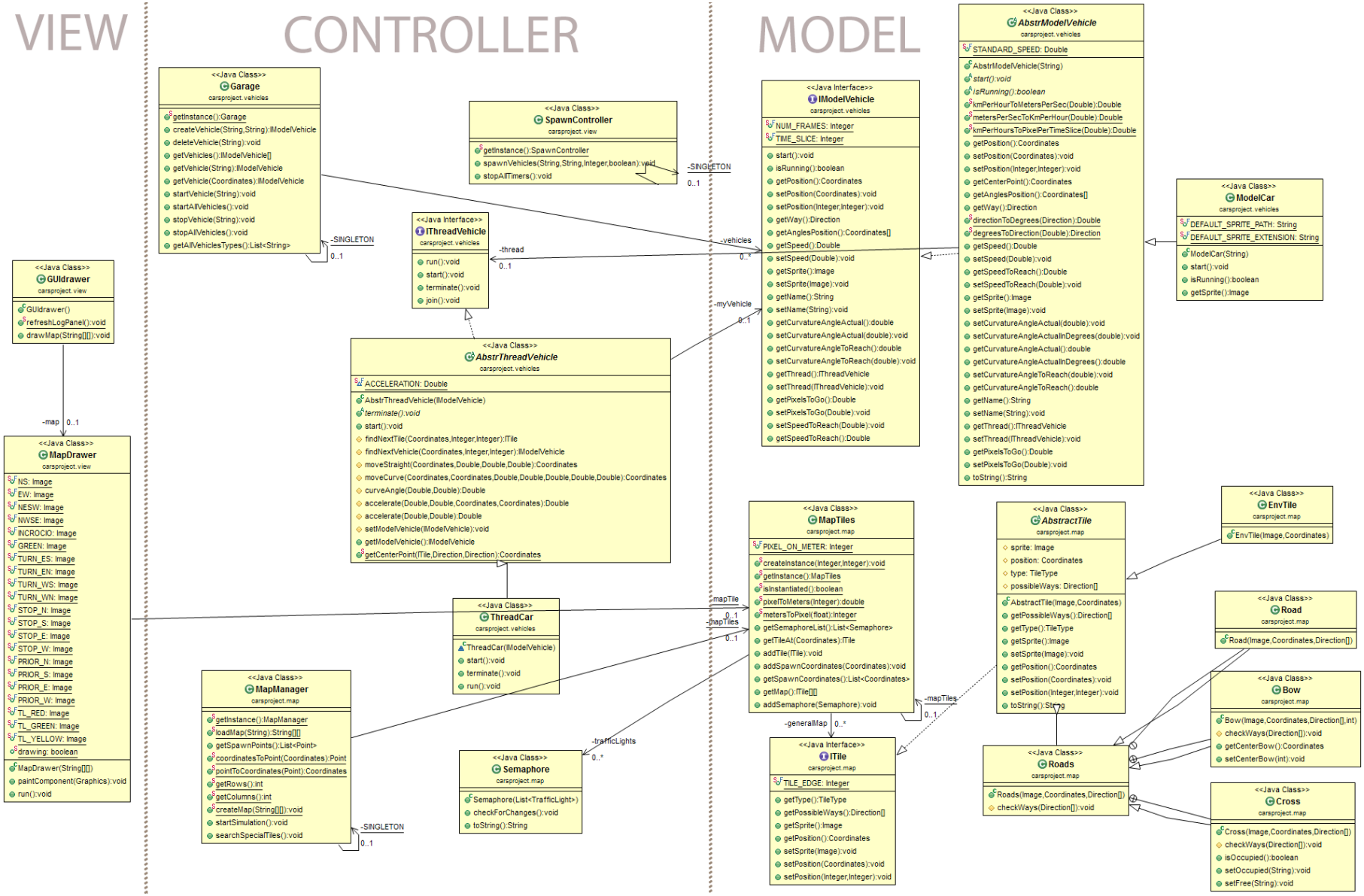
The program has been developed following Model-View-Controller architectural pattern, which permits to have an easier code to maintain and to improve with more elements and features. It has also been decided to use interfaces and abstract classes in order to provide common and useful functions and to simplify implementation of specialized classes.

The map has been designed to be customizable by the final user thanks to a simple representation on .txt files. In fact each character corresponds to an object called tile which is a square that, aligned with the others, will form the map. Tiles can be everything, from simple grass/tree, without any final purpose, to traffic lights related to each others in a same cross. While the interface is common to all of them, there will be two main families: the one including all road-like tiles (bows, crosses, etc.), and the one including all environment-like tiles.

Vehicles are the other main part of the program and are thought to be entirely represented in their multiplicity. They will be divided between model and controller, both with their interface and abstract class which provide useful functions and implement most of the functions requested by their own interface to permit faster implementations of specialized classes, without losing the focus on general aspects.

The last important part is the view. It permits the user to interact with the program by loading maps, adding vehicles and starting/stopping them. There will be also a panel on the right that will store informations about vehicles spawned and will permit to interact singly with them. A minor part but not less important is the possibility to load a pdf file containing a guide to write a custom map.

It has done a hard work while planning the project to provide most possible flexible classes, that is why there will be interfaces and abstract classes for both tiles and vehicles. Then some complete implementations of those classes will be provided to demonstrate potential of the application and to give an example of specialized class to other possible programmers. Every method will be written trying to make it reusable in future different implementations for other vehicles or other tiles, without have to modify the entire code, but making only some little changes.



- General UML diagram

3. PACKAGES' ORGANIZATION

All the classes have been organized into 4 packages, which will be shortly described now.

1) **carsProject.general:** contains the classes

    a. **Loader:** it's the main of the application
    b. **Amanuensis:** class collecting logs from the application. This permit to create different named logs and then add to it lines of text as String
    c. **Coordinates**: class representing the position of an object in the map, uses double values inside inside of it while returns always integer. This because we can have more precision while setting coordinates that will be automatically rounded when returned since this is wrote thinking to be used as pixel's coordinates.

2) **carsProject.map:** contains the classes which allocates the objects to manage the elements forming the environment (roads, traffic lights etc…). Will follow a short description of every class:

    a. **ITile**: interface which provides a first skeleton for every class which will represent a tile
    b. **AbstractTile**: abstract class, implements ITile, provides a first implementation for every class which will represent a tile
    c. **EnvTile**: extends AbstractTile, represents all the tiles of the environment which are not integral part of the road (like could be the grass, a tree or whatever; however, in this first implementation of the software, only the 'grass' has been taken under consideration)
        i) *Base*: extends EnvTile, is a nested class which represents the grass
    d. **Roads**: extends AbstractTile, represents the tiles of the environment which are integral part of the road
        i) *Road*: extends Roads, is a nested class which represents a straight or an oblique road
        ii) *Bow*: extends Roads, is a nested class which represents a bow
        iii) Zebra: extends Roads, is a nested class which represents a zebra crossing
        iv) *Cross*: extends Roads, is a nested class which represents the middle of a cross, allows vehicles choose different directions when possible
        v) *Stop*: extends Roads, is a nested class which represents a stop right before a cross
        vi) *Priority*: extends Roads, is a nested class which represents a priority right before a cross
        vii) *TrafficLight*: extends Roads, is a nested class which represents a traffic light of a cross (it has been implemented as last tile before the cross area)

e. **Semaphore**: this class represents a collection of TrafficLight objects in order to define a group of traffic lights involved in a same cross. They are synchronized thanks to priority values which define when every light need to be changed
f. **MapTiles**: class containing the objects which represent the environment (so the general map, the list of traffic lights' crosses and the list of zebra crossings) and almost all the methods which allow their full allocation
g. **MapManager**: class which handles the allocation of the map, by reading the .txt file, and the specific objects which have to change their states depending by the time
    i) **Point**: simple and general class that can store two final integer values. Used to store spawn points, different from spawn coordinates because are described by number of row and number of character of the tile subject to the spawn
h. **Direction**: enum class, represents all the possible directions kept by a tile (or a vehicle); in this first implementation have only been taken under consideration mutation of slope 0, 45 and 90 grades (even if the tiles which 45 grades of slope are not used)
i. **TileType**: enum class, represents all the types of possible tiles which can be present on the map
j. SpawnController: got methods to spawn, at one or at all points, one vehicle or to set a timer to spawn new vehicles at fixed time rate
    i) SpawnTimerTask: extension of TimerTask used to spawn vehicles at fixed time rate

3) carsProject.vehicles:

a. **IModelVehicle**: Interface to be implemented when writing a vehicle's model class. It is hardly recommended to extends AbstrModelVehicle that already got a lot of methods implemented. Classes that implements this interface will contains all methods necessary to store and modify any vehicle's data and methods to be runnable
b. **AbstrModelVehicle**: Partial implementation of interface IModelVehicle. Contains nearly 90% of methods already done. It is really recommended to extends this class instead of implements IModelVehicle. Abstract methods are start() and isRunning()
c. **ModelCar**: This is the final implementation of IModelVehicle, extends AbstrModelVehicle. This start() implementation will create and run a ThreadCar thread only if vehicle is already spawned in map, so as if coordinates are legal
d. **IThreadVehicle**: Interface to be implemented in order to create a thread to update a vehicle. Classes implementing this will have to contains methods to start, stop and join the thread
e. **AbstrThreadVehicle**: Partial implementation of IThreadVehicle, contains some protected methods to help programmer when writing final run() method

f. **ThreadCar**: Final implementation of IThreadVehicle, extends AbstrThreadVehicle. This contains implemented methods of the interface, contrary to abstract class that provide only some helping methods to programmer of thread class

4) carsProject.view:

a. **GUIDrawer**: The base class needed in order to draw the GUI and its components, contains methods to open the map and start the car, creates the object which will draw the map and monitor the movements of the vehicles
   i) **SpawnPanel**: Panel used to spawn one or more vehicles at chosen spawn point, with or without auto-start
b. **MapDrawer**: Contains the object and methods to draw and manage the Map panel and his components, extends JPanel class and overrides its PaintComponent method in order to draw and monitor the map.
c. **ImageTool**: It's an external class from "game-engine-for-java" available at https://code.google.com/p/game-engine-for-java/, distributed under GNU Lesser GPL and it can be used to manipulate images. Supports the following image operations:
   i) Convert between Image and BufferedImage
   ii) Split images
   iii) Resize image
   iv) Create tiled image
   v) Create empty transparent image
   vi) Create a colored image
   vii) Flip image horizontally
   viii) Flip image vertically
   ix) Clone image
   x) Rotate image

4. WORKING PARTITION

Classes built in synergy:

- **Masini Gioele** and **Pruccoli Andrea** worked together on the **MapTiles** class
- **Pruccoli Andrea** and **Zamagna Marco** worked together on the **MapManager** class
- The **Loader** class has been shaped by all the members

Masini Gioele: classes included in the package **carsProject.vehicles**, classes **Amanuensis** and **Coordinates** in **carsProject.general**;

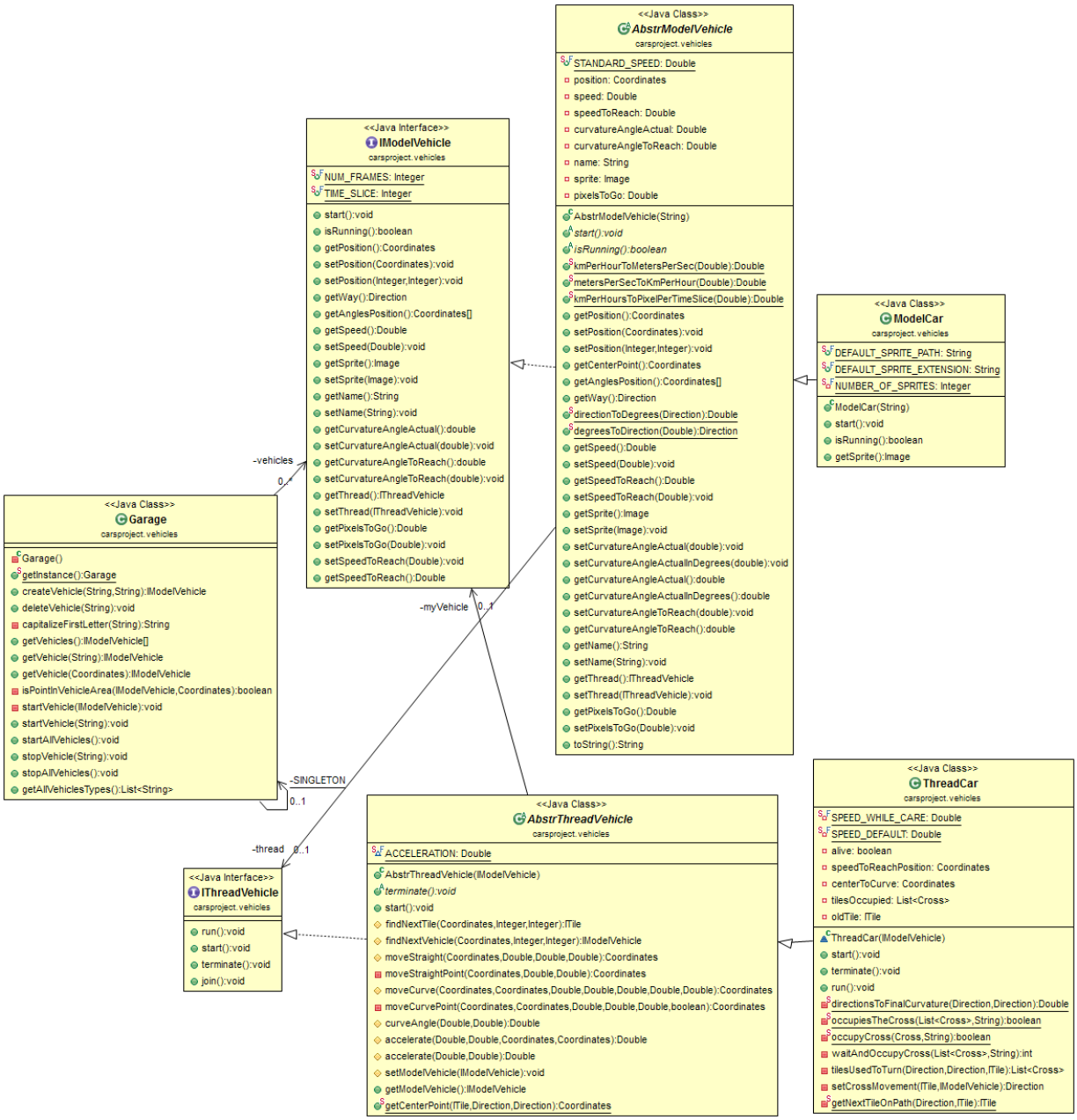
Pruccoli Andrea: classes included in the package **carsProject.map**;

Zamagna Marco: classes included in the package **carsProject.view**.

5.1 PLANNING'S DETAIL: MASINI GIOELE'S SIDE

Gioele Masini's work has been mostly based on management and movement of vehicles circulating on map. Following MVC pattern he worked dividing vehicles in two main groups of classes: one for models and one for controllers. They have both the same structure: an interface implemented by an abstract class that has been extended in a possible implementation class for cars. This is the list of classes:

	MODEL	CONTROLLER
Interface	IModelVehicle	IThreadVehicle
Abstract class	AbstrModelVehicle	AbstrThreadVehicle
Implementation	ModelCar	ThreadCar



Firstly an interface has been created for both model and thread:

1) **IModelVehicle**, containing methods to store and modify informations about the vehicle;
2) **IThreadVehicle**, with methods to make it moves around the map respecting highway code.

Those two classes are really important because they permit future implementations, different from the one presented which represents only cars. The entire vehicle's planning has been thought to be flexible and not rigid or limited to only some types of vehicles. This is why it was decided to write a first and partial implementation with two abstract classes:

1) **AbstrModelVehicle,** contains a lot of methods requested by interface. It has so much getters and setters which are mostly implemented here to provide a near-to-ready class for every new vehicle. It's also so much important because it contains a constructor with the needed *"String name"* parameter, which is a unique identifier for every vehicle used also from Garage *(which will be discussed below)* to manage all vehicles. This class stocks also some conversion's static methods to manage speed values which are stored as km/h, a user-friendly format also useful to have a well scaled simulation supported by a relation in map between virtual pixels of the screen and meters in reality.

   Its not-abstract extension **ModelCar** contains only methods to start a new thread, not implemented in abstract super class because it could be different from ThreadCar, and *isRunning()* to check if it's running;

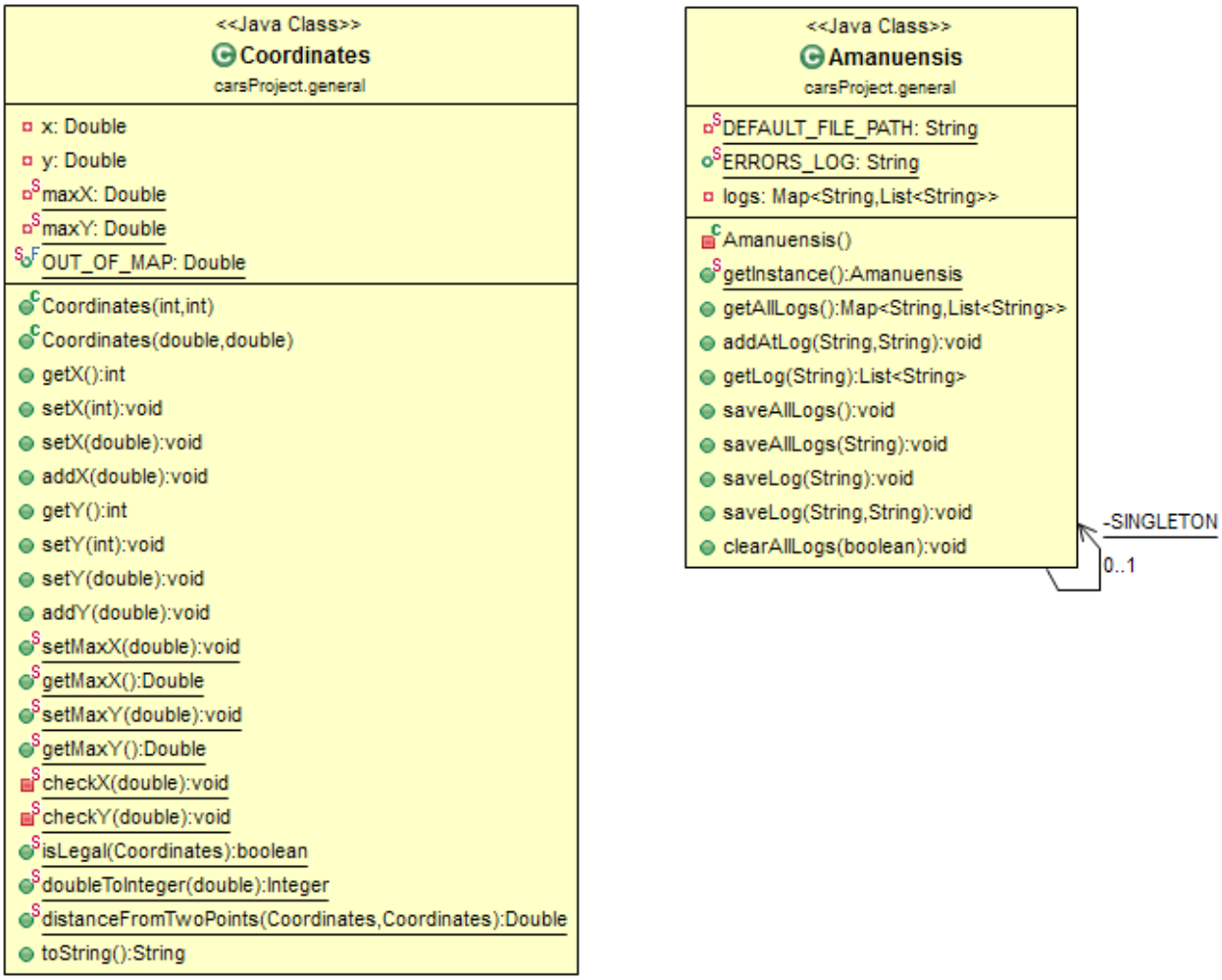
2) **AbstrThreadVehicle**, contains instead no necessary methods (excluding those inherited from its super class Thread) because it has been thought that they are too much important and related to the single vehicle implemented that could be more useful to provide some protected functions that can help programmers in future final implementations of the method *run()*, which has the task to move the vehicle. They can do mathematical calculations to retrieve updated coordinates of a vehicle moving straight or curving, to accelerate/decelerate until a new speed and to scan in front of itself in search of other vehicles or important tiles (all tiles that are no simply straight roads).

   Then it has been extended in **ThreadCar** which contains all methods requested by interface for threads implemented using all methods provided by its abstract super-class.

All model objects are instantiated by class **Garage** which has only a singleton instance and it's the only way that has the program to get access to spawned vehicles (via name). The *createVehicle*() method uses reflection to spawn any asked new vehicle permitting to create any other model class different from the ready *ModelCar* if it respects the same class name format: "Model_____", putting on blank space the common name of the vehicle, that will be

passed as input to the method when spawned. This way has been chosen to have the same code running with every new vehicle with no changes. This class provides also methods to get a vehicle by coordinates (used to search if there is a vehicle on given point) and to start/stop threads of one or all vehicles.

Others Masini's classes are more generals:



1. **Coordinates**, developed to store and manage points' coordinates at the same way in all the program. It has been designed to be precise and to be used with pixels. This is why all values are stored as Double and returned rounded as Integer and this is why it contains an "add" method with Double values as input that is the only way to benefit of the double precision. This class has also been planned to be used in a finite space which extends from (0,0) to (*maxX*, *maxY*), that's why all "*get*" methods can throw an exception if coordinates are out of those limits. Setters will not throw exceptions because values not legal are considered as "point out of map" that can be set but not returned. There are also two more general methods stored here:

doubleToInteger() that consists of two *Math.round()* concatenated to each others (to have a more readable code), and *distanceFromTwoPoints()* to calculate and return distance between two points;

2. **Amanuensis**, logger of the program. It can have only a singleton instance because it alone can store and modify different logs. They have to be named and then it's possible to add strings of text to them. There is no createLog method because Amanuensis itself provides to create the log if not already done. This in order to avoid to check log existence every time the program tries to add an error or every other type of text in a log. This contains methods to save to file those logs in a default or custom path. It have also a constant with the name of standard error log.

• Appendix: additional work

This appendix covers the additional 50 hours of work done after the first 100 hours. This had the purpose of improving the program under several aspects.

The first and more important part of work has been done on vehicles:

- **Management of crossing**: now when a vehicle arrive to a cross, wait until it is free trying to occupy it. Only at this moment it will pass the cross. About the code, when a car reaches Stop, Priority, or every other tile before a Cross, it chooses a random direction to go among possible ones and then wait until tiles it needs to move are free with method *waitAndOccupyCross()*. After that it sets them occupied with synchronized method *occupiesTheCross*() in competition with other cars and will free them only at the end of the movement.

A lot of work also affects the view and map:

- **Map reading and creation method partly rewrote:** implemented **cross tiles with four ways**. Old implementation could only create crosses of 3 ways (T crosses) from user customizable map. Now the program can read correctly "plus" crosses and every cross got a boolean flag to keep mind their occupation state and the name of the vehicle that set them occupied. Thanks to that only the vehicle that occupies a tile can set it free. Then also **bow read part has been rewrote** to permits bows to be to extremes of the map and be correctly read. **Others three important methods** are added to *MapManager* used on load of maps: *loadMap()* is the main method which start read and process of map file and configuration file -if there is one- both from file system and from jar file. It also create instance of *MapTiles* and start *threadTL* if needed. Then *readCfgFile()* and *processCfgFile()* are added to read and make changes described in **map configuration file**. First method search configuration file from given map file path then read it and pass it as input to process method. The latter take strings, ask to *decodeCfgLine()* an array of strings with configuration line name and parameters that will be processed as necessary from parent method.

- **Semaphores implementation**: the user can now add traffic lights with the character "T". They must be near a cross of four asterisks. The program will find them and regroup under an only "*Semaphore*" that will be handled by "t*hreadTL*". This thread will made traffic lights to change their colour at fixed rate with 8 seconds of green, two seconds of yellow and others two seconds in which all lights are red. By default they will change colour singly, but it is possible to synchronize two different traffic lights of a cross by map configuration file. It admits the keyword "*TLpriority*" with three arguments between parentheses: number of row and number of character of the traffic light, and value of priority. Priorities have to be set from 1 to N, two synchronized traffic lights must have the same value of priority. If user don't set them, the program will set default ones. About the code it has been a hard work because of the high number of bugs in already present methods. At the end of map creation, *MapManager* starts method *searchSpecialTiles()* which searches traffic lights and starts *createAssociations()* on them, creating new Semaphores which are added to *MapTiles*. After that, if program founds some semaphores, will starts *threadTL*.
- **Customizable spawn points**: every map now can have customizable spawn points on Road tiles. Old fixed point has been removed and now they are set through a map configuration file which needs to have the same name of the map file but with different extension: ".cfg". With the keyword "*spawnpoint*" followed by number of row and number of character of the tile the user want to spawn vehicles, put between parentheses, it is possible to set a spawn point for the map. Those two values will be converted in the right coordinates and will be available at the **new spawn window**. The user can also choose to **spawn new vehicles at fixed time rate** at a chosen point and make them auto-start at creation. About the code, it has been created a new type of variable named *Point*, which stores two final values, different from Coordinates, to manage those spawn points. A new class has been wrote to manage them: *SpawnController()*. It can spawn one vehicle or set a new timer to spawn vehicles at fixed time rate. Holds references to all timers and there is a method *stopAllTimers()* to stop them. There is also an inner class, *SpawnTimerTask* which is the task used by timers to spawn new vehicles. Every spawned vehicle now is rotated at necessary grade.
- **Others**: some minor, but not less importants, changes of code are:
    - *Flickering of vehicles fixed*, founded an error in *paintComponent()* method;
    - *Improvement of lateral panel "Vehicles log"*, with more readable strings ordered by alive/not alive;
    - *Added different sprites for cars*, with different colors chosen random at spawn;
    - *Added some error messages on view* when user try to spawn without loaded map and when user try to load a map with another one running on program;
    - *Updated Help pdf*, now it includes traffic lights, screen of the new sample map and help section to explain map configuration file;

- *Map zoom*, now the map is zoomable both from "Map" menu and from buttons in right panel;
- *Vehicles' queue*, vehicles now stop themselves if in front of them is occupied (there is another vehicle);
- *More types of grass*, changed grass tile with a texture and added some different types of grass tiles, with a bush/tree which is rotated or flipped randomly. Those tiles are chosen randomly

Know bugs of old version now fixed:

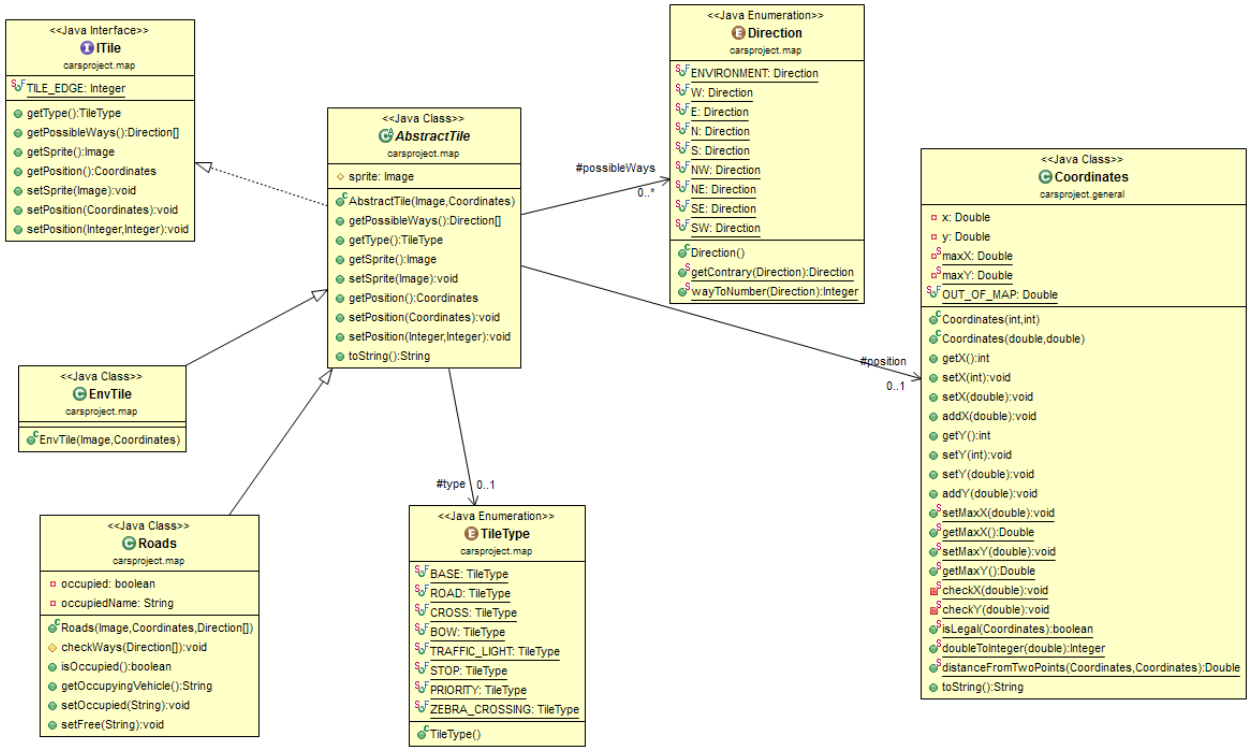
1. Vehicles are drawn out of their lanes because of the point chosen to calculate movements;
2. Vehicles curve too fast at left and too slow at right because of point chosen to calculate movements;
3. Vehicles don't turn at crosses;
4. Flickering of running vehicles;
5. Vehicles in foreground respect to menus;
6. Resizing of window after the selection of a map (if fullscreen).

5.2 PLANNING'S DETAIL: PRUCCOLI ANDREA'S SIDE

Pruccoli Andrea's work was based on the management of the elements composing the map.
Were first conceived an interface and an abstract class which give a frame and a first implementation to all the tiles which will form the world under consideration; these classes are **ITile** and **AbstractTile**. In these classes are defined the setter and getter methods related to the fields in common, such as the field *sprite* which will be used to view the tiles on the GUI, the field *position* (expressed in **Coordinates**) of the tile on the map, the field *type* (expressed in the enum object **TileType**) which express what kind of tile it is, and the field *possibleWays* (expresses as an array of the enum object **Direction**) which specifies the direction/directions kept by that tile.

As the UML diagram above represents, two specializations of the abstract class **AbstractTile** were been conceived, the **EnvTile** and the **Roads** classes.
The first one includes in itself all those tiles representing the environment, so the ones which are not fundamental for the goal of the application, such as a green area, a sidewalk, a gas station etc…

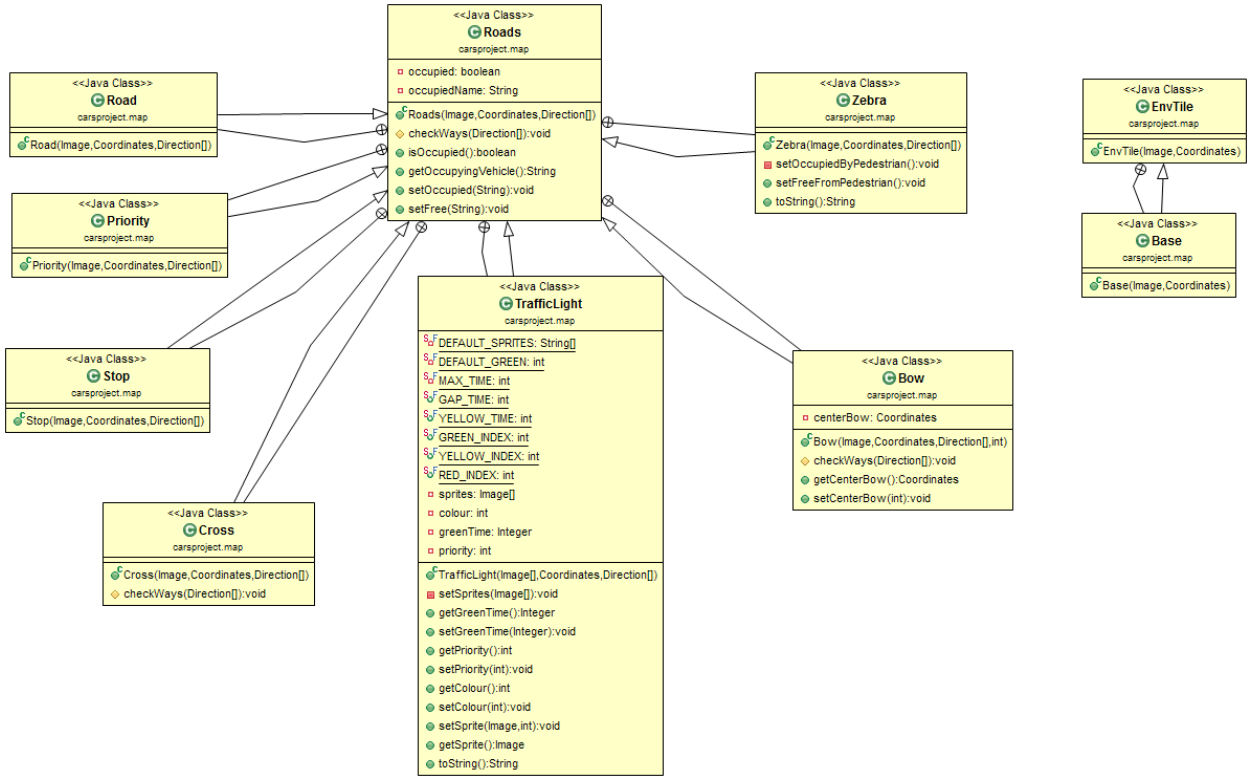
> **Notes:** since they are not fundamental (as just said), in this first version of the application the only EnvTile objects used are the ones representing a green area (the **Base** specialization).



The second one includes in itself all those tiles representing all the common kinds of road which can be found on a street; these ones are the ones fundamental for the goal of the application, because, depending on the kind of these tiles, the actions of the vehicles will be influenced. Now will follow a repeal of the types which have been implemented as specialization of the **Roads** class:

1) The simplest specializations are:
    a) *Stop*, which represents the area with obligatory arrest right before a cross;
    b) *Priority*, which represents the area right before a cross with obligatory arrest only if vehicles are coming in this direction;
    c) *Road*, which represents the most common straight or oblique road.

2) The most elaborated specializations are:
    a) *Cross*, which represents a cross area;
    b) *Bow*, which represents a bow;
    c) *Zebra*, which represents a zebra crossing;
    d) *TrafficLight*, which represents a traffic light right before a cross area.



Except for the Bow and Cross classes, each tile can (and have to) keep only one direction of those present in the enum class **Direction**, which is stored inside an array of **Direction** object. The classes earlier excluded need to keep, as concerns the **Bow**, exactly 2 directions (still there isn't a check on the regularity of the pair), and as concerns the **Cross** class, at least 2 directions (in this first implementation, the field *possibleWays* contains the directions of all the streets involved in the cross).

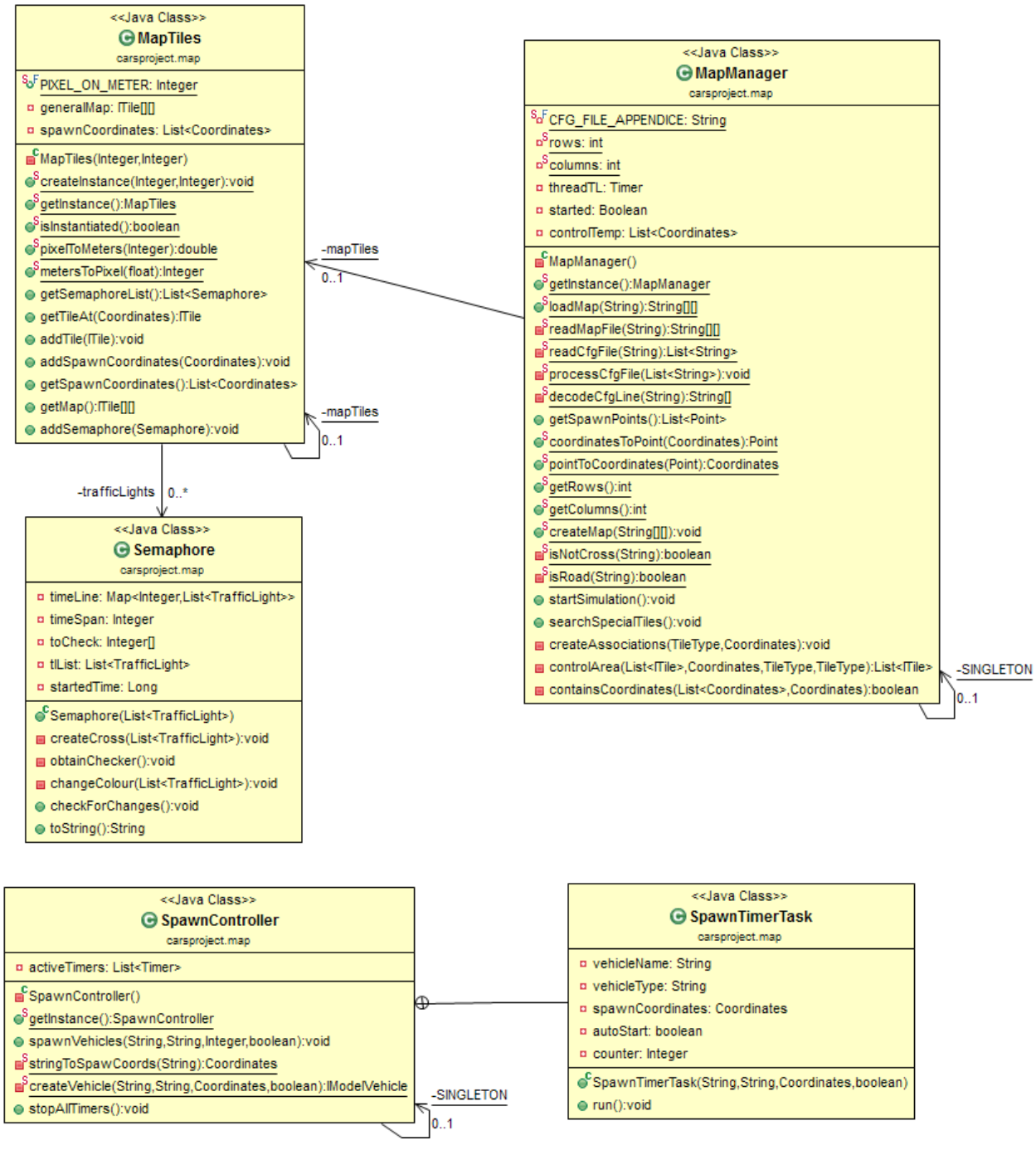
The inner class **Zebra** implements, in addition to the methods of the superclass, the changeState() method needed in order to change the state of the field *crossed*, which shows the current state of the zebra crossing (if it is crossed by some pedestrians or not), and its getter method isOccupied().

The inner class **TrafficLight** implements, in addition to the methods of the superclass, getter and setter methods related to the fields present in this class. There are setGreenTime(Integer) and getGreenTime() which sets/returns the value of the field *greenTime* (which represents the time given to this traffic light for its green sign time); setPriority(int) and getPriority() which sets/returns the field *priority*, which represents the turn of the traffic light inside the cross in which is evolved; setCoulor(int) and getColour() which sets/returns the value of the field *colour*, which represents the state of the traffic light in that moment.

> **Notes:** both **TrafficLight** and **Zebra** do override of the toString() method because, as for what concerns these classes, it is more important to represent the state in which the object of these class are.

The **Semaphore** class gathers the traffic lights involved in a same cross, organizing through an **HashMap<Integer,List<Roads.TrafficLight>>** the change of states of the traffic lights. The key (**Integer**) represents the moment in which the related traffic lights (**List<Roads.TrafficLight>**) have to change their state. The change of state is realized by the method changeColour(Integer, int), invoked by the method checkForChanges(), which defines the actual interval and the colour the related traffic lights have to assume.
The checkForChanges() method is invoked by the Timer object in the **MapManager** class.
The **HashMap** is allocated thanks to the createCross(**List<ITile>**) method when a **Semaphore** object is created, which splits the list passed as argument depending on the value the *priority* field has assumed, and defines the keys depending on the value the *greenTime* field has assumed. Inside the class is saved the list of traffic lights passed as argument, and is obtained an array of the key of the **HashMap** in order to make the checks quicker and in order to custom the toString() method.

> **Notes:** a control to check the relationship of green sign time between traffic lights of the same turn is not implemented.



The **MapTiles** class is the model of the application for what concerns the map. Handled as a SINGLETON, this class has the fields needed to the storage of the objects forming the map and particular objects which will need to be handled by the **MapManager**. The field representing all the map is *generalMap*, implemented as a matrix of generic object ITile.
The objects are allocated inside the matrix thanks to the addTile(ITile) method, called by the createMap(String[][]) method in **MapManager**. The class also implements the getTileAt(Coordinates) method, which returns the ITile object at the specified coordinates.

As said before, the **Zebra** and **TrafficLight** objects need to be handled in order to change their states (occupied field for **Zebra** and colour field for **TrafficLight**), so was needed a method able to group all these types in one object. That's why are present two fields: *zebraCrossing*, a list of **Roads.Zebra** arrays; *trafficLights*, a list of **Semaphore** objects.
Both are allocated when an instance of **MapTiles** is created, then they are filled thanks to the searchSpecialTiles() method, which calls the createAssociations(**TileType**, **Coordinates**), the one which actually groups the tiles of the same type, in the same area of map.
In order to help this method to easily and quickly create all the necessary associations, a support list has been created, *controlTemp*, a list of **Coordinates**, which contains all the coordinates of tiles which have been checked by the controlArea(**List<ITile>**, **Coordinates**, **TileType**, **TileType**) method, avoiding, by making a check of the coordinates of the tile under consideration, a useless re-search.
Are so present two getter methods for the retrieving of these lists, and they are getZCList() as concerns the list of zebra crossings, and getSemaphoreList() as concerns the list of traffic lights.

The **MapManager** class is the controller of the application for what concerns the map. Handled as a SINGLETON, this class handles the reading and creation of the map from the .txt file, the threads needed in order to change the states of **Zebra** and **TrafficLight** objects retrieved by the **MapTiles**. If there are no tiles of these last two types, the threads are never allocated.
The thread regarding the zebra crossings is implemented through an inner class presents in the **MapManager** which extends the abstract class **Thread**; this class is **ThreadZC**.
The thread regarding the traffic lights is implemented through the **Timer** class, useful because allows the execution of the routine described after a time decided by the programmer; in this situation has been thought that an updating check could have been necessary every second.

5.3 PLANNING'S DETAIL: ZAMAGNA MARCO'S SIDE

Zamagna Marco's work was based on the aspects of the graphical design, from the simple GUI to the creation of the map.
The main class is **GUIdrawer** that contains the object which will draw the map. Its most important part is the constructor which contains all the objects which compose the graphical interface and its actionEvent.
The GUI is composed by one **JMenuBar** which contains three **JMenu: "Map**", "**Car**" and "**?**". The **Map** JMenu contains two **JMenuITem: Open** and **Default**. When is clicked **Open** the program creates a **JFileChooser** object which allows the user to select a file outside the project through a window preset and then calls the **ReadFile** method which returns a matrix which contains the String representation of the map. Works in the same way the **Default** JMenuItem but instead of using filechooser, it uses a default path. The **Car** JMenu contains **Start** and **Stop** which are used to start and stop the cars. Lastly the **'?'** JMenu

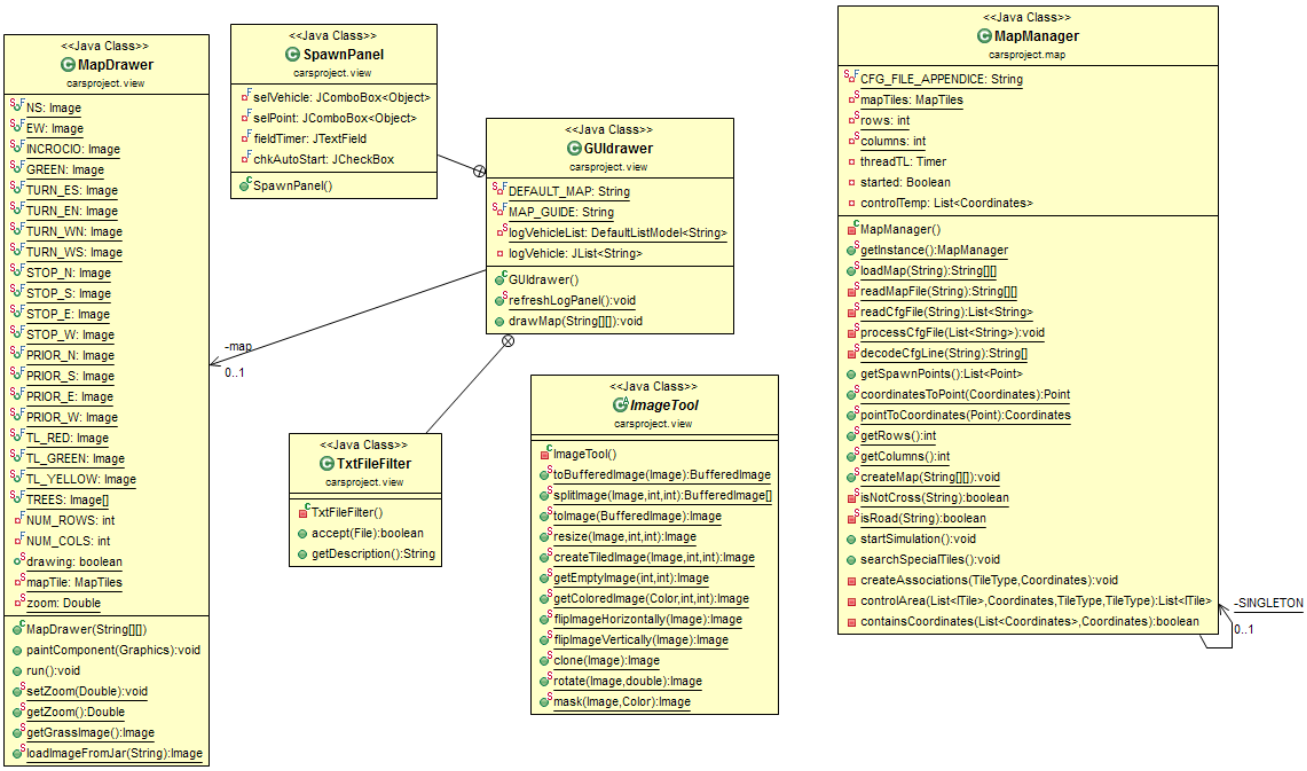
contains the **Help** which opens a pdf file with the instructions to draw a custom map in a .txt file and **About** which opens a **JDialog** with the informations about the project.
The other part of the window is composed by two **JPanel**, one at east of the window which contains a JLabel and a JList and the latter, which contains the log of vehicles. In order to use the JList was necessary the creation of another object type, **DefaultModelList**, and when the elements are added in the **DefaultModelList** the **JList** prints the element.
The remaining part is used for the **MapDrawer** panel created by the **drawMap** method. The **TxtFileFilter** is the class used to filter the files, between the ones chosen by the **FileChooser**, which ends with ".txt".

The specialized class charged with the drawing of the map is the **MapDrawer**, which is an extension of the **JPanel** class and overrides the **PaintComponent** method in order to allow the drawing of the map inside it. First of all, it invokes the method of its super class **JPanel** to allow the drawing of the panel, and then continues with the actual design of the parts of the map. The class contains all the images used to create the matrix Tile by MapManager as static attributes. The class which is responsible for managing all operations related to the creation of the map is **MapManager**, composed by two methods: **readFile** and **createMap**. **readFile** method consists of two readings, the first is only used to count the number of rows and columns which compose the map, dimensions that will be used to create a matrix of characters which will be filled by the second reading. To read the map is used a **BufferedReader** that cyclically reads a line of the text file. Firstly it checks the length of lines and chooses the largest, then it reads another time the entire file and fills each cell of matrix with a character. **createMap** method contains the code for the effective creation of the map inside **MapTiles** class. It's composed by a switch that checks if the characters within the character matrix are those accepted:

- "**-**": horizontal road;
- "**|**": vertical road;
- "*****": bow or crossing;
- "**Z**": crosswalk or zebra crossing;
- "**P**": priority;
- "**S**": stop;
- "**T**": semaphore;

All characters different from those listed above are converted into "base" tiles with green images.

(Example of default loaded map with running vehicles.)



6. TESTING

The whole application has been tested thanks to its graphical interface. All main bugs which didn't permit a sufficiently good experience have been fixed but there are surely some more minor bugs due to secondary features not tested and with known bugs (such as scansion methods in *ThreadCar*). Some maps were wrote to test some movements and are available within the program in "maps" folder. Considering that not all actions are implemented (ie.: curving in crosses is missing, car move only straight, traffic lights are implemented but not displayable and not used by cars.) the program is well working.

As said the application has been tested thanks to the graphical interface, and that's true but only if we talk about the last step of the project. Masini Gioele and Marco Zamagna worked a lot basing their code on what the graphical interface gave on output, the first in order to examine the cars' execution, the second in order to examine the graphical execution, while Pruccoli Andrea worked almost entirely with the console offered by the development environment in order to examine the traffic lights' and zebra crossings' execution, as well the effective allocation and the fairness of the values taken by the fields of the objects.

The application has been tested both on Linux (distro Chakra with OpenJDK 7) and on windows (Win 7 and 8 with Java 7).

Known bugs

These are the known bugs not fixed because of time limit of 300 hours:

1. Sometimes maps need some escape characters at the end of the ".txt" in order to work;

Others have been fixed by Gioele Masini in his additional work.

7. ENDING NOTES

As visible this program is so far from be finished. The work has been focused on provide all essential methods and classes before of a beauty and stable code. This could be a more and more complex application depending on hours the team can work on it. There are probably some missing controls on setting data and we got some classes not implemented in toto or not used in the simulation, like traffic lights. This because this project could fit hardly more hours than the 300 worked until now thanks to it's improvability, given by team's decision of made a flexible code, easily extensible providing others new features. So those are not to be understood as deficiencies of the team's work but as examples of the potentials of this program.

Some other parts that could be implemented in future are:

- Manage of fuel and gas stations;
- Graphical WYSIWYG editor with drag and drop features and possibility to set some parameters like vehicles' spawn points;
- Implementation in vehicle's movements of two and more lanes in each direction and one-way traffic;
- Implementation of overtaking;
- Draw of different "base" tiles with parks, cities and countries;
- Add of walkways and walking people in addition to zebra tiles;
- Other vehicles different from cars (motorcycles, trucks, etc.)