

Redesign of the TRAFFIC library
Final Report
SEMESTER THESIS

Sibylle Aregger

18th March 2005

http://se.inf.ethz.ch/projects/sibylle_aregger

Student: Sibylle Aregger (siaregge@student.ethz.ch)

Student-No: 99-905-564

Supervising Assistant: Michela Pedroni

Supervising Professor: Bertrand Meyer

Contents

1	Introduction	4
2	Description and Management	4
2.1	Use	4
2.2	Requirements	5
3	Initial State	5
4	Implementation (Design Decisions)	9
4.1	Class Names	9
4.2	Map	10
4.3	Line	12
4.4	Simple line	17
4.5	Line section	17
4.6	Place	19
4.7	Factory	20
4.8	Type	22
4.9	Line section state	23
4.10	Route	24
4.11	Station	25
4.12	Extension	26
4.13	XML-File	26
5	Test application (without GUI)	27
5.1	How to run the test application	30
6	Experiences	32
7	Conclusion	34
8	Appendix	34
8.1	Student Guide	34
8.1.1	TRAFFIC overview	35
8.1.2	TRAFFIC_COLOR	37
8.1.3	TRAFFIC_LINE	39

8.1.4	TRAFFIC_LINE_SECTION	44
8.1.5	TRAFFIC_LINE_SECTION_STATE	47
8.1.6	TRAFFIC_LINE_SECTION_STATE_CONSTANTS	48
8.1.7	TRAFFIC_MAP	50
8.1.8	TRAFFIC_MAP_FACTORY	54
8.1.9	TRAFFIC_PLACE	60
8.1.10	TRAFFIC_PLACE_INFORMATION	63
8.1.11	TRAFFIC_ROUTE	65
8.1.12	TRAFFIC_SIMPLE_LINE	69
8.1.13	TRAFFIC_TYPE	71
8.1.14	TRAFFIC_TYPE_FACTORY	72
8.2	Developer Manual	74
8.2.1	TRAFFIC_COLOR Class	75
8.2.2	TRAFFIC_LINE	76
8.2.3	TRAFFIC_LINE_SECTION	81
8.2.4	TRAFFIC_LINE_SECTION_STATE	83
8.2.5	TRAFFIC_LINE_SECTION_STATE_CONSTANTS	83
8.2.6	TRAFFIC_MAP	84
8.2.7	TRAFFIC_MAP_FACTORY	85
8.2.8	TRAFFIC_PLACE	86
8.2.9	TRAFFIC_PLACE_INFORMATION	87
8.2.10	TRAFFIC_ROUTE	87
8.2.11	TRAFFIC_SIMPLE_LINE	88
8.2.12	TRAFFIC_TYPE	90
8.2.13	TRAFFIC_TYPE_FACTORY	91
8.2.14	XML-File	92

1 Introduction

The intention of this semester thesis was to redesign the existing TRAFFIC framework with respect to ease of understandability, ease of useage and ease of extension.

This final report is separated into 9 sections. Section 2 is an introduction to the TRAFFIC library describing where it is used, what it is used for and what its requirements are. Section 3 gives a short overview of the initial state of the TRAFFIC library at the beginning of this semester thesis. Section 4 shows in full length the design decisions that have and could have been made, with its pros and cons. Section 5 presents a non graphic test application; the graphical part of the library was handled by Rolf Bruderer who wrote a Master Thesis[2] on that subject. Section 6 gives a short overview of the experience I made with the TRAFFIC library and the EIFFEL environment. Section 7 deals with the conclusion of this semester thesis. Finally, in the appendix, 8, you will find additional information such as developer manual (8.2) and students guide (8.1).

2 Description and Management

2.1 Use

In winter 2003/2004, Professor Bertrand Meyer redesigned the course “Introduction to Programming” for first semester students at the Department of Computer Science at ETH Zurich. He introduced the approach called “Inverted Curriculum”[5]. In this approach students start as consumers of a big object-oriented system to become producers. In contrary to this new approach, the standard way of teaching introductory programming starts with small producer programs like “Hello World”. Here the focus lies on syntax and other basic language constructs like how a program file looks like and how to use/include libraries.

In the “Introduction to Programming” course, the students work on an object-oriented library called TRAFFIC[6]. TRAFFIC models a city and its public transportation system. The combination of the Inverted Curriculum and the TRAFFIC library allows students to write interesting programs right from the start. As Bertrand Meyer puts it:

Teaching is better than preaching, and if something is better than teaching, it must be the demonstration, carried out by the students themselves, of the principles at work, producing “Wow” results.

The TRAFFIC library is supposed to build up on familiar constructs, containing interesting algorithms and data structure examples, support of graphics and multimedia. The combination of the “Inverted Curriculum” and the TRAFFIC library will make learning how to program not only efficient but fun for the students. The class book used in the “Introduction to Programming” is called “Touch of class”[4]. Along with the book comes a software students can perform their first programming experience. This software uses the TRAFFIC library.

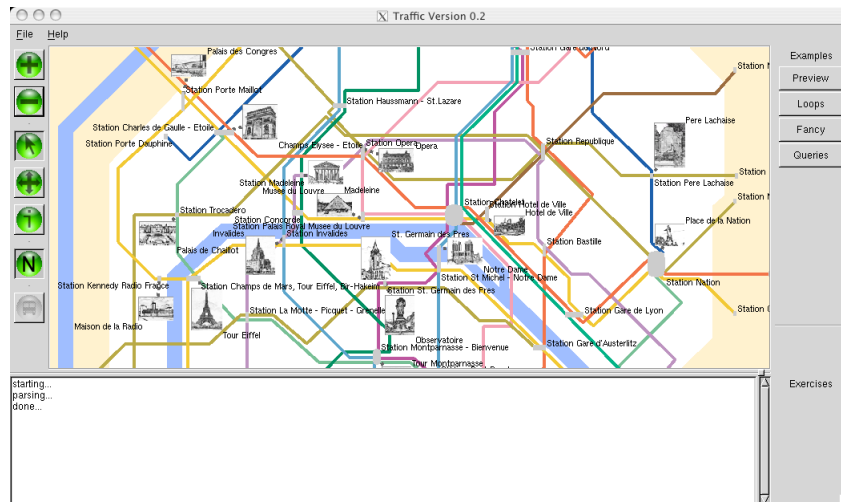


Figure 1: Touch screen shot

The execution of this software will give the students the “wow” results Bertrand Meyer mentioned. With a few lines of code the screen appears as seen in the figure above. What you can’t see in the picture is the animation effect those few lines accomplish.

2.2 Requirements

As the TRAFFIC library is used in a first year’s course it can be assumed that many students have no or little experience with programming. Therefore the overall goal of this library has to be exemplary object oriented and easy to use. In addition it has to be easy to understand, easy to remember and sound. The library code should make extensive use of the style guidelines[3], contracts and documentation of implementation (8.2) and interfaces (8.1). This documentation of the library has to be easy to understand, complete and useful.

Another requirement to the library is to make it as much independent of the graphical interface as possible.

3 Initial State

In the spirit of the saying “a picture says more than a thousand words”, you find overviews of classes of the original system on the next pages.

The first figure shows you the overall view of the TRAFFIC library. The main class is the CITY class. The city object contains all city elements like public transportation lines and places. They are administrated through the network.

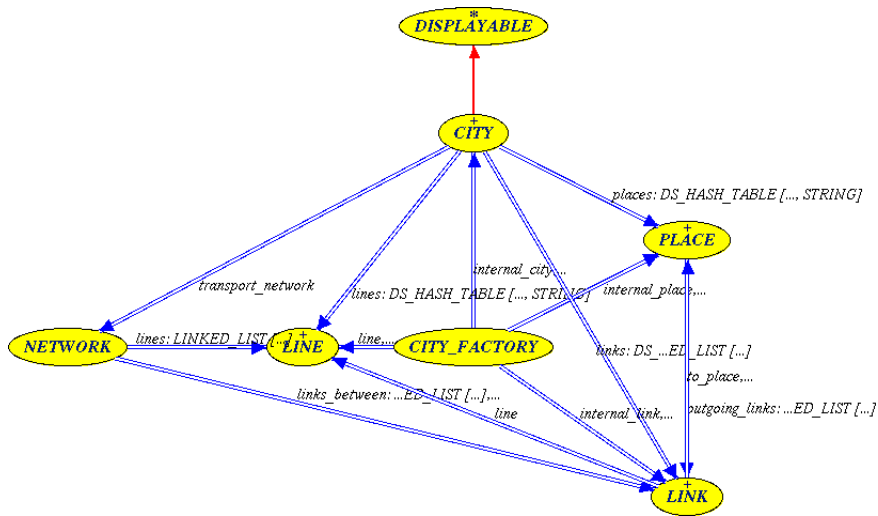


Figure 2: CITY (initial state)

The network is implemented as a graph to be able to represent lines (or more precise the links of the lines) as edges and places as vertices.

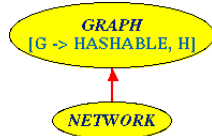


Figure 3: NETWORK (initial state)

The line object consists of two one way lines. They in turn are put together by links.

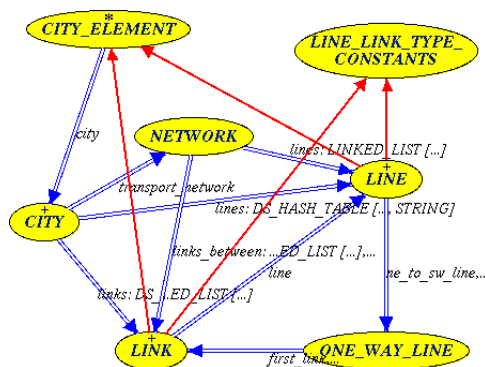


Figure 4: LINE (initial state)

The one way line represents the links as a list of edges.

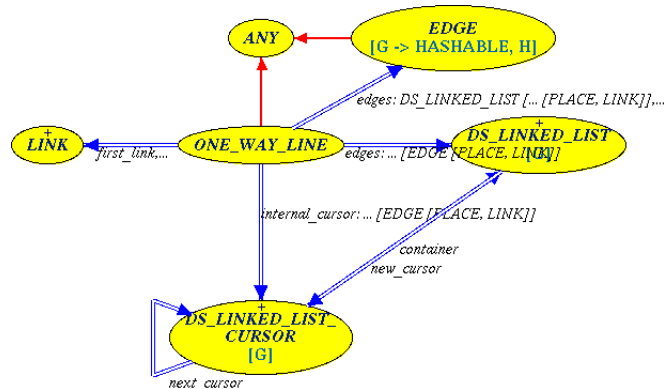


Figure 5: ONE_WAY_LINE (initial state)

The class `LINE_LINK_TYPE_CONSTANTS` is used to assure that the type of a link corresponds with the type of a line.

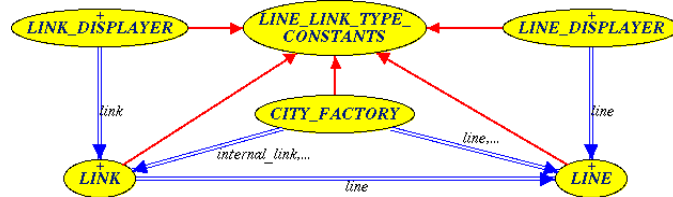


Figure 6: LINE_LINK_TYPE_CONSTANTS (initial state)

The link contains of two coordinates defining the start and end place of the link. Additionally the polypoints define their graphical representation as vertices.

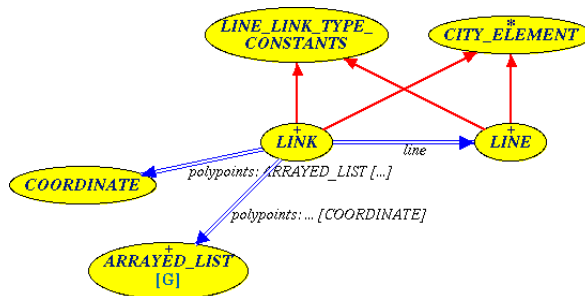


Figure 7: LINK (initial state)

Each place has a coordinate standing for a position.

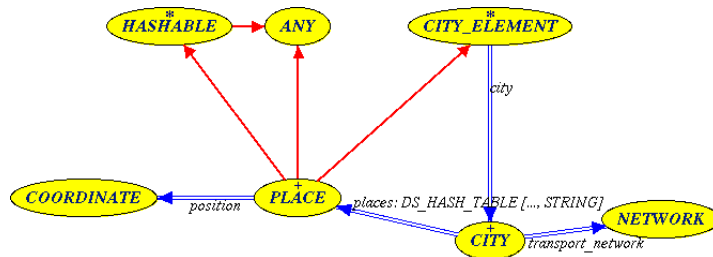


Figure 8: PLACE (initial state)

A station is a special place where public transportation lines come through.

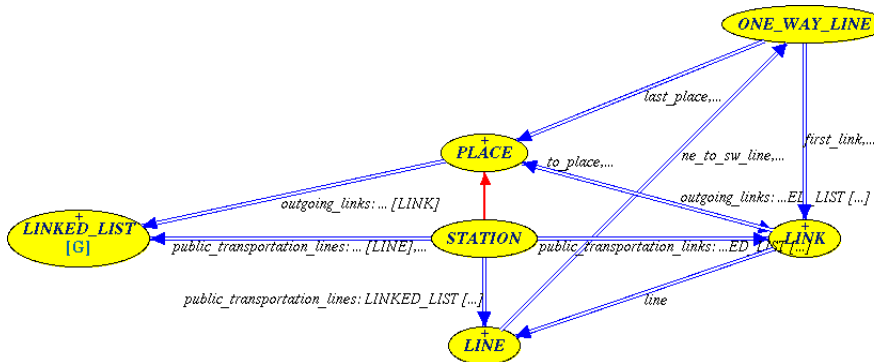


Figure 9: STATION (initial state)

The route class administrates a list with places that want to be visited, e.g. by a tourist.

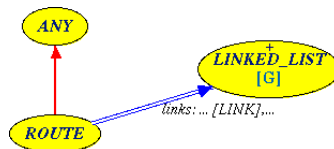


Figure 10: ROUTE (initial state)

The route calculator class calculates the route through all places of a route object.

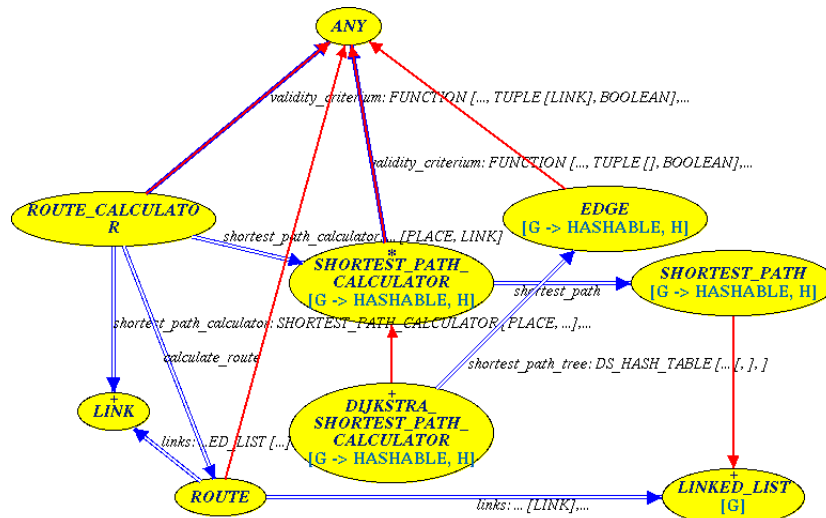


Figure 11: ROUTE_CALCULATOR (initial state)

4 Implementation (Design Decisions)

This section is about the TRAFFIC library as it is after the redesign.

4.1 Class Names

In the process of redesigning the library some class names have changed. They may represent more or less the same object as before but now are known under a different name. One general name-design decision is to name all classes belonging to the TRAFFIC library with the prefix TRAFFIC. In that way it can easily be identified which classes belong to the TRAFFIC library and which ones don't. Even though TRAFFIC appears to be a long name for a prefix there seems to be no satisfying short form of it.

In the following list I will present the most important name changes. The suffix TRAFFIC will be neglected.

- CITY becomes MAP. The idea is that, as a map is responsible for the administration of the connections, it is more appropriate to call it a map than a city. In addition it is possible to have several maps of the same city representing different kind of transportation systems. More on a map and what a map exactly is will be found in 4.2.
- LINK becomes LINE_SECTION. The main idea behind this new name is that the name "line section" indicates the fact, that a line section is a part of a line. A link can be thought of as simply a connection between two points. But a line section is more. It is a connection between two points belonging to a line. Another alternative would have been to call it segment or connector. But this again would have been too general.

4.2 Map

What was called the city in the initial state model is renamed to map. This is mainly due to the fact that a map should administer all places and line sections, which are connections from one place to another.

A logical consequence is to make the class TRAFFIC_MAP inherit from a graph class. Now the administration can take place implicitly. No helping object, as the network in the initial state model, is needed any more. That's why in the new TRAFFIC library the class NETWORK is absent and city is not the correct name for the class any more. NETWORK was the connection from the city to the graph representing the connections. Now the map is this graph representation, so it's a simple thing to have different maps of the same city representing different traffic transportation systems. You simply create two different map objects.

Another consequence of making TRAFFIC_MAP inherit from GRAPH is, that now the shortest path from one place to another is implicitly given. The graph library already contains a feature carrying out the algorithm. This is a good example for the usefulness of inheritance.

The second design decision for the map is to add a new field, called description, where a textual description to the map can be added. The idea is to add a short description of what exactly this map represents of a city. The final version of the TRAFFIC_MAP looks (simplified) like the following BON diagram[1].

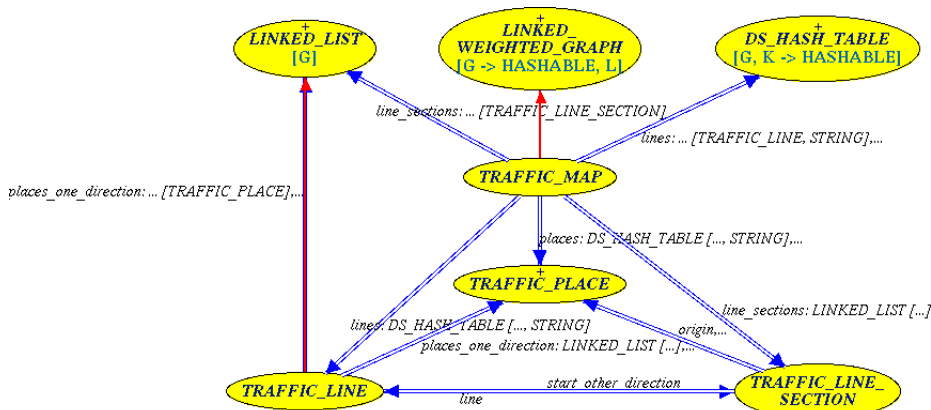


Figure 12: TRAFFIC_MAP (simplified overview)

Following is an overview of the discussed outcomes of the TRAFFIC_MAP class.

1. *Map contains a graph.* This would have been similar to the class NETWORK in the initial state. The Network-class didn't have much of an interface but mapped the inheritance of the graph into a class of its own. The possibility of having several networks in one city wasn't implemented. So there didn't seem to be really an advantage of the map having a graph. The general problem of holding the map up to date isn't any simpler when

for each update this field has to be updated. A way to implement this update, if the map and the graph are different objects, is to make map and/or graph inherit from observable. In that way they will observe each other to assure, that those two heavily related objects are up to date all the time.

2. *Map is a graph.* To do so, map is a descendant of a graph. There exist many different graph interfaces in Eiffel. The decision will be for a multiple directed, weighted graph, because the weights can represent the ride time from one place to another. The advantage of the solution of the map inheriting from a graph is, that graph operations can be performed directly on the map. And the map really is just a map representation of a city. A city so can have several different maps representing different kind of views. If map is a graph no additional interface for map operations on the graph is needed, therefore the interface can be hold relatively simple.
3. Alternatively, there would have been the possibility that each *place know it's connected line sections* and the graph is thereby implicitly given. The problem with this solution is, that the shortest path calculation would has to be reimplemented on those classes (place, line section). With one of the two first solutions, the Eiffel-library implementation of the shortest path algorithm in a graph can be used.
4. A general design decision to keep in mind is the *path of changes*. It is important that the representation of a line, a line section, a place is consistent with what is in the graph. If the map is a graph it is easier to guarantee this because both operations, changing the TRAFFIC structure and the graph, can be done in the same class in the same feature call on the same object.
5. If the map is a graph it can be imagined, that one map represents the *public transportation system* of a city and another is a representation of the *streets*. Then when wanting to find a shortest path one has the option of only searching in one graph or in the union of the two graphs. The implementation of the needed operations is given in the Eiffel Graph library.
6. The above comment is also an answer to the question *how to represent streets*. With the possibility of different map representations of one city, representing different parts of the traffic system can easily be achieved. If line sections don't have to belong to lines, then line sections of the type street can be thought of belonging to one fictionary line in an arbitrary order and being added to the map in the same way. Ways from one place to the other could then be found either in a dedicated street map of the city or in a special line.
7. The last design decision on maps is the *interface to the graphical interface*. This should be as small as possible. In the former version a background was stored to the city. But the background of a map is quite a graphical detail, so it should be, if possible omitted from the general TRAFFIC library interface of the map class.

4.3 Line

The line class has undergone many design decisions and changes. Therefore first a simple overview of the class how it is now. And below, for simple comparison, the same overview of the line class in the initial state system.

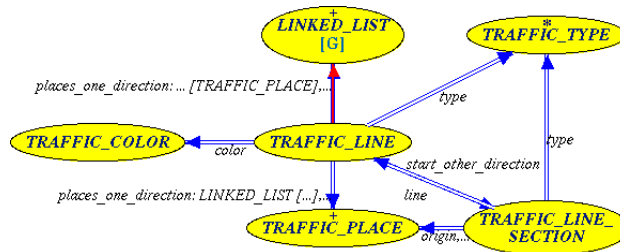


Figure 13: TRAFFIC_LINE

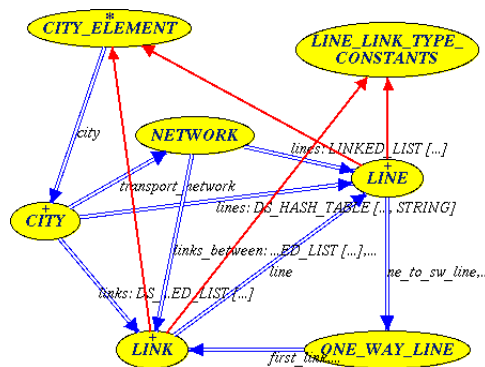


Figure 14: LINE (old)

As you can see, the TRAFFIC_LINE inherits from LINKED_LIST. In this linked list the two sub lines of the line are stored. How they are stored is more an implementation detailed and can be found more detailed in the developer manual(8.2.2). A line can, but does not have to, have two directions. And those two directions also don't have to drive by the same places. A line also has a color assigned. This color can be looked at as a graphical detail, but in many cities traffic lines are also provided with a color. This color is as much of an identification of a line as the name of the line itself. Therefore the color can be added to the traffic line without hesitation. The interface of the TRAFFIC_LINE is very simple, as can be seen in the appendix 8.1.

Following some examples of lines.

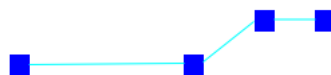


Figure 15: Line with only one direction

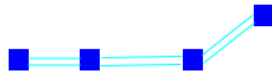


Figure 16: Line with two directions and same places

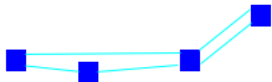


Figure 17: Line with two directions and different places

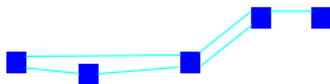


Figure 18: Line with two directions and different terminals

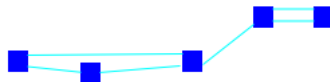


Figure 19: Invalid: line with hole

Following all possible design decisions listed.

1. Is it necessary to have a *one-way-line* class in order to model the two directions? The advantage is, that one can work on whole line directions independently. But usually a customer of a line class is just interested in the line sections he is going to pass from one place to another. He may also want to know the two terminal stations of the line in both directions. For all this functionality a one-way-line seems to be an overkill and makes the interface more complicated.
2. An important question that has to be answered is if a *line can be empty* (contain no line sections). This decision is important for the implementation. If a line can not be empty, line sections have to be added at the creation time in order to be able to have a proper instance. To make the interface more flexible, our implementation of a line can be empty. But keep in mind that an empty line may not be as easy to understand as a non empty line.
3. *Can walkways and streets also be represented as lines?* If so, they would have to be special lines, because they may not need to be entirely connected. A street can be thought of as a collection of street line sections. One way streets and ordinary streets can be added to the same big city street line. The same thoughts have to be made for walking ways. These are loose connections between different places where people can walk. Either all walking line sections are collected in one special walking line or each way is put together in a line of its own. Advantages of the first idea

is, that there is no need to name many different lines that are more or less the same. In addition one could imagine to have different types of streets put together in different lines (example: motor ways, cycle ways, ...). The disadvantage of this idea is, that the line, as it is implemented now, is too restrictive. A line section needs to fulfill special conditions in order to be able to be added to a line. The two directions have to be connected entirely. They can not have "holes". As a street is a net of line sections this condition is not appropriate any more. This problem would be solved with the second idea. But the problem this idea is, that you would need a huge amount of lines which would not represent a street as we would expect them.

4. If one decided that a *line is a linked list of line sections* there exist two possible implementation. Either the line implements a linked list with a similar interface or line inherits from linked list. The advantage of the first is, that it would show students how a linked list looks like and the class would not contain any inherit clause which makes it easier to understand for first year students. All they need is in the class. A disadvantage would be that one would have to reimplement almost every feature of the linked list. So this is the advantage of the second idea. If line is a subclass of linked list all those features could be used and the added interface of line is reduced to a minimum.
5. *Can a line be represented as a cycle?* If the design with two one-way-lines is dropped the question then is how the line with the two directions should be represented. An idea would be to represent a line as a cyclic linked list. The pointers to the two terminals are stored. The advantage of this idea is, that one can drive round on a line. Line sections that can not be driven through with passengers could be marked with a special state or an infinite weight. The problem is, that you usually also want to know, where the first and last place of a direction of a line is. So there would have to be at least 4 places marked. In addition many dummy line sections modeling this rounds would have to be added. There doesn't seem a practical use where this behavior of a line as a cycle is preferable.
6. To continue the discussion of the first point we add the question, *how are the two lines (directions) represented?* The possibility of the one-way-line was discussed above. One core decision has to be whether a line consists of two directions at all or if the two directions are separate lines. If the two directions are separate lines no problem with their access names and representation exists at all. But the problem is, that then two lines may be quite similar, namely the two directions of what is known as one line, or they be entirely different, as is with two directions of two different lines. Therefore it is nice to compact the two directions into one line. But then the problem is how to address these sub lines. They could be called `ne_to_sw`(north-east to south-west) and `sw_to_ne`(south-west to north-east). The advantage of this solution is, that the name indicates a characteristic of a sub line. But the problem is, that it is quite cumbersome to call features `ne_to_sw` etc. in addition the insertion of line sections gets more complicated, because this characteristic has to be restored and in this process the sub lines may be exchanged. So it would be possible

that after the insertion of a line section if you ask for the terminal in `sw_to_ne` direction you will get the terminal of the other direction of before. Additionally `sw_to_ne`, as intuitive they at first seem, are not very intuitive names. There needs to be an emphasis on one direction of the wind. If no classification from north to south can be made east to west is considered. Therefore what you really would get back when calling `sw_to_ne` could be a sub line going from se to nw. Furthermore as the starting place of a direction does not have to be the terminal of the other direction there can be imagined special cases, where there is no clear decision on sw to ne (see the next figure). The alternative is to call the directions `one_direction` and `other_direction`. The advantage of this solution is, that the implementation and the interface get much simpler. For the normal use-case when we know where we are and in which direction we want to go this is more than sufficient.

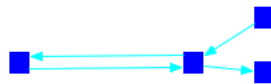


Figure 20: Possible line, but difficult to decide on `ne_to_sw`

1. The next issue to be discussed is the *type of a line* (The same discussion will take place for line sections as well). There exist two possibilities. Either you use a class for each type with a common superclass or a constants class with string representation for each type. The first approach seems better object oriented but also an overkill as each class will only contain of a textual representation of itself. In addition it will make the whole library much larger (many more classes) and this may not be very helpful when first year students need to work with the library. The second approach makes sense if there doesn't seem any other use for classes. So everything needed is indeed just a string to distinguish a bus line from a tram line.
2. If different classes were chosen as type representation one could think of making the class `TRAFFIC_LINE` *generic with respect to this type class*. The first question to be answered is whether this use of genericity is unusual. That's a question that depends severely on the use of the class. Another aspect of the use of genericity is that it has a great impact on several classes. The `PLACE` class seems to be nearly the only genericity-free class. This alone is no problem. Another problem is how to find out the type of a given line. This problem seems to be much more difficult to solve. At first it doesn't seem to be a problem, but when thinking of introducing for example busses that drive on bus lines you want to be sure, that a line you want to add the bus to really is a bus line. This task seems quite complicated just because we chose genericity. One more problem with genericity are the cat-calls. The main reason for introducing genericity is to ensure that no bus line sections are added to tram lines. But because of the static type checking in the case of genericity we could think of examples where exactly this happens. Another reason against genericity is, that this topic is covered at the end of the semester but as nearly all classes would use genericity in one way or the other the

students would have to work with it almost right from the start. So the core question seems to be if the use of genericity here is so exemplary and necessary that it can overcome its disadvantages. If one decides against genericity what would have been done by genericity (type safety) can be easily simulated with contracts and the interface even gets simpler.

3. One more aspect that has to be kept in mind is the *extendibility*. How will a simple line look like? A simple line is a line that passes through the same places in both directions. Could one also think of a fancy line with stops? How would they look like? Would it be possible to extend the TRAFFIC library to implement those extensions?
4. To come back to the discussion about names. It is quite clear that there has to exist a feature to get the terminal station of a direction. But is it also vital to know the *beginning of a direction*. If so should it be called directly like `start_1` and `start_2` or with help of a feature where you can get the starting place of a direction with help of its terminal place. The advantage of the first idea is, that you can use it like the terminal if terminals are accessed through `terminal_1` and `terminal_2`. On the other hand it may be sufficient to have a feature that returns the starting place to a terminal of a direction. Another advantage of the second idea is, that the interface gets smaller and is easier to understand. A feature call can have a better name description than just `start_1` and `start_2`.
5. Another issue is the *color* of a line. The first question to ask is, whether this is general information of a line or part of the graphical interface which will graphically represent a line. If the color is a general information of a line then it can be added to the interface without much restriction. If it was simply graphical information one can think about hiding this information in some general graphical information field.
6. On more question is if a line does need a *reference to the map*. In the section about the map it was said, that it would be good to make changes through the map interface in order to assure the consistency of the map and the graph. When it comes to the line, the question arises if this indeed can be managed. Can all changes be done through map without a reference to the map a line belongs to? If it can be assumed that there always will be a reference to the map in the application available, the line class doesn't need a reference to the map itself. In that way the interface can be reduced and simplified because all features connected with the map reference in the class can be omitted. The control flow also gets easier to understand as the potential places where a map reference is known are fewer than if every instance of a line does have a reference, too. In addition one can think about inserting the same line in different map. There is no reason why the same line can not be represented in different maps.
7. One last remark on the *final interface*: It was decided to call the directions `one_direction` and `other_direction` and the corresponding terminals `terminal_1` and `terminal_2`. For the starting place of a direction a feature call named `start_to_terminal` was implemented. To add a line the feature call `extend` was redefined. This makes the whole class very easy to understand as `extend` is a well known name for a feature that adds an

element to a data container. The whole interface is documented in the appendix (8.1, details on the implementation 8.2).

4.4 Simple line

The difference from a line to a simple line is, that a simple line contains of two opposite directions that visit the same places in the two directions. As a consequence a simple line always adds a line section into both directions. The general implementation of a simple line is easier as there are not many special cases that need to be payed attention to. The interface of when a line section can be added to a simple line differs from the one of a line and the implementation of the extend. But the rest is similar. The starting place of a terminal is the other terminal. But for the user this isn't relevant, as he can use the interface of a simple line as the one of a line. Unlike the line the simple line seems to need a reference to the map (or maps) to be able to insert a new line section for both directions into the graph. This must carefully be handled to guarantee the consistency of the graph and the traffic map.

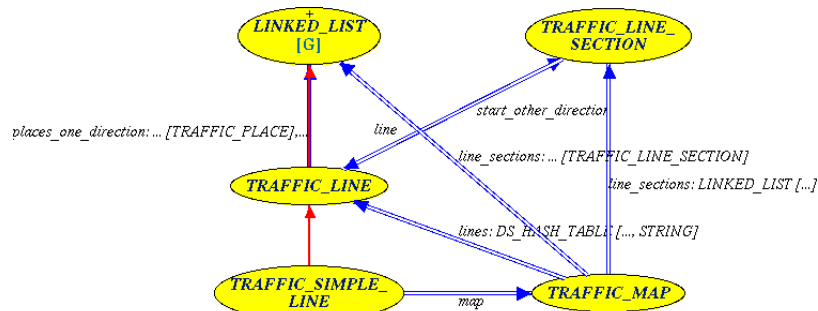


Figure 21: TRAFFIC_SIMPLE_LINE

4.5 Line section

As mentioned at the beginning of this section (4.1) a line section is a connection from one place to another. It is a specific TRAFFIC_TYPE and belongs to at most one line. This is an artificial restriction; we could just as well allow a line section to belong to several or no lines. The relating questions are handled in the following section. But first as for the other classes a short overview of the final class. More detailed information on the implementation and interface will be found in the appendix (8.1.4, 8.2.3).

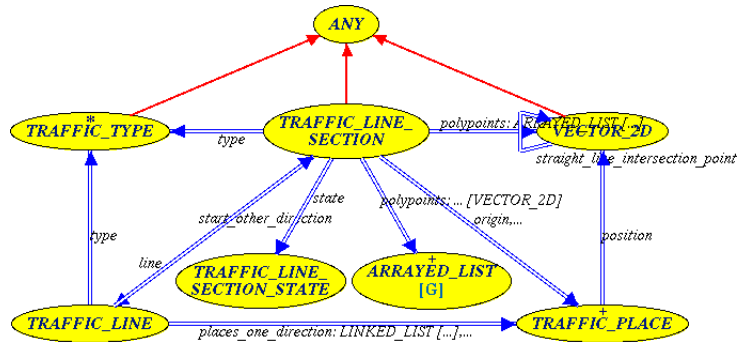


Figure 22: TRAFFIC_LINE_SECTION

1. The first question corresponds to the implementation of the line. The two classes TRAFFIC_LINE and TRAFFIC_LINE_SECTION are highly connected. A design decision for the one infects the other. So the first thought is about lines being an exemplary implementation of a linked list. Then the line section will be the *element of a linked list*. The interface will have to be accordingly. As the line were a sample implementation of a linked list the line section were a sample implementation of a linked list element. This has the same advantages and disadvantages as for the line. (See point 4 of line discussion points)
2. Another point to remember is the *addition of a line section into a line*. Should the line or the map control this action? What features of line section do have to be used? We can argue, that it makes sense to only allow lines to add itself as the owner line of a line section. And also only a line can remove itself from a line section. If this element modification were done by the map, the intention of only letting the map do these changes would show clearly. This would be to prevent abuse of other classes capturing a line section.
3. The question already mentioned above is, if a *line section has to be in a line* or if it can have no line added. The advantage of not having to have a line assigned is, that one can create a line section and add it later on. A problem would arise anyway if both the line and the line section have to have a reference to each other right from the start. The disadvantage is, that a line section, as the name implies, not always is connected to a line.
4. And a similar question to be considered is if a line has only to be *added* to the line section when the line section can be successfully inserted into the line. This makes sense because if a line section can not be inserted into the line it can not have a line as owner line. It isn't in this line.
5. Another question is if a line section always is *directed*. This is especially important together with TRAFFIC_SIMPLE_LINE. If a line section can be directed or undirected a simple line will only have to add one undirected line section whereas in the other case it has to make sure both directed line sections are added. This alone is not enough to come to a decision as we have to make sure that the map will be able to represent undirected line sections as well as directed at the same time. A disadvantage of an

undirected line section is, that what could have been called origin and destination does not make much sense for undirected line sections any more. And we will have to use less expressive names as it is the case for `terminal_1` and `terminal_2` of the class `TRAFFIC_LINE`.

6. Again a question related to the decision taken for line is the one of *genericity*. If `TRAFFIC_LINE` was generic with a formal parameter representing the type of the traffic line the line section would also have to be generic. The same advantages and disadvantages arise as for the line (point 2 of line discussion points).
7. A line section can be thought of as having a *state*. Such states could be state: “normal”, “collision”, ... The advantages of such states are that collisions, work and other things could be simulated. The disadvantage is however, that it would make the interface bigger and maybe more complicated. Furthermore, the disadvantages depend on the representation of the state. There mainly exists two possibilities. Either the states are instances of classes or constant strings. The discussion is more or less the same as for the type field for lines and lines sections.
8. And also for the line section the discussion about a strict separation of the graphical and model information remains. What needs to be in the interface for the graphical representation and what is ordinary information. An example are the polypoints that describe in detail where the line section goes through in image coordinates. These polypoints will also be used to calculate the distance which is an information that is needed independent of the graphical representation. But are polypoints needed? If they were empty the distance could be calculated from the coordinates of the origin place and the destination place, assumed that places always have a coordinate.

4.6 Place

A `TRAFFIC_PLACE` is a place in a city. It inherits from `HASHABLE` in order to be able to put it into a hash list where all places of a map can be administrated. Following a simple overview of the class `TRAFFIC_PLACE`. There weren't many design decisions to be taken, as a place is a relatively simple object.

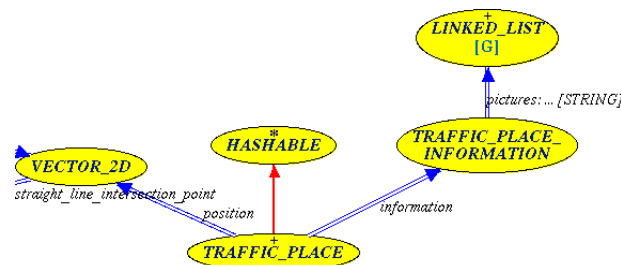


Figure 23: `TRAFFIC_PLACE`

1. A question that arises also for the TRAFFIC_PLACE is how the *interface to the graphical representation* should look like. It is clear that it would be nice to have one or several images showing the place. The first question is if this is just graphical information. The other question is, whether one image is enough or if there should be the possibility to have several images per place. One could argue, that one image is sufficient, but then if one could have several images of a place it could be shown from different directions and additional images of objects of interest could be added as well. Additionally a description would be desirable as well. There arises the question if this information can be organized. If one thinks that one or several images and a description are similar information one could put these information into one structure. The advantage is to have a smaller interface for the place class. Especially when this information is non mandatory one will not always need the bigger interface. A disadvantage is, that there will have to be introduced a new class which in return makes the library interface bigger. Another idea is to add this additional information into other classes that have this non mandatory additional information as well.
2. A connected question is whether a place *needs a coordinate*. In our view, it makes sense to assign a coordinate to a place. Even if we came across a place that does not yet have a specified position, we could assign it a default value at the coordinate origin. In that way even though a coordinate were non mandatory for a place one could assume a coordinate for calculations. A related question is if the position can be updated by polypoints of line sections that connect a place. This decision is depending on whether polypoints are mandatory and if there, too, exists a default.

4.7 Factory

The main idea and functionality of the factory has remained the same. The names have changed according to the general name changes(4.1). Because the line and line section types now are classes, those classes also need to be generated somehow. They are generated in the TRAFFIC_TYPE_FACTORY. Following the overviews of the TRAFFIC_MAP_FACTORY and the TRAFFIC_TYPE_FACTORY are the design questions on hand.

get the created object back. But this is not nice distinction between query and command. And the advantage of the second way is, that a created object is available as long as no other object (maybe even object of the same type) has been created. The same object can be fetched several times and it is possible to introduce a feature that gives feedback whether the creation of the object was successful at all.

4. In connection with line sections the question arises *who generates new line sections*. The answer to this question is found together with the implementation of the class `TRAFFIC_LINE_SECTION`. The main issue lies in the decision if a line section can be created without knowing to which line it belongs or if it has to belong to a line. In the second case it gets more complicated to create a line section, as the line has to be known. Therefore it may be more complicated to create a new line section, and the factory class has to consider the rules for line section creation.

4.8 Type

The initial TRAFFIC library had a type for the lines and line sections (called links in the initial state) assigned. The idea of this type was to make sure that only line sections of a given type could be added to a line and vice versa. Therefore the main information needed was an identifier for the type. A string will do just fine. Hence in the initial version a constants class with those identifiers and some needed features was designed and line and line section inherited from this class. For the redesigned TRAFFIC library many different possibilities have been taken into account. Finally the decision was on a hierarchies of classes. The advantages of this design are pointed out below where all design issues are discussed. First, as with the other classes, too, a simple overview of the class in it's final state.

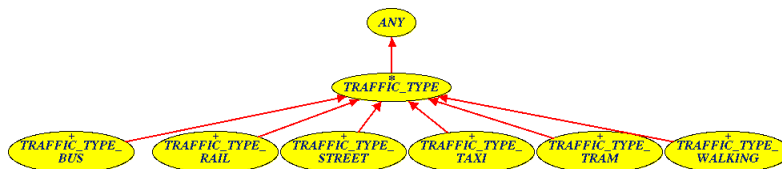


Figure 26: TRAFFIC_TYPE

1. All questions about the types of lines and line sections are connected to the question of *how line sections are inserted into lines*. The definition of a line section suggests, that a line section belongs to a line. In general a tram line and a bus line are quite the same except that on a tram line trams are driving and they drive on rail tracks rather than the pure street. Therefore a line section will most likely represents these rail tracks that connect one place with another. As a bus will not be likely to be able to drive on rail tracks a way is needed to ensure that only line sections of a given type can be added to a line of some type.
2. The first decision to come is whether there are to be *classes for each type* with a common superclass *or* a class with *constant strings* representing the

different types. The advantages of the second solution are, that it is easy to implement, easy to understand and sufficient. The main disadvantage is, that it is not a good object oriented example for students to learn from. The first solution will present a nice object oriented way of distinguishing different types but on the same time it makes the whole framework a lot bigger. There are many new classes which will mostly contain nothing more than a string representation. An advantage of this first idea is that those classes can be extended and used for other things of the same type. For example when vehicles are introduced that drive on a line or passenger that have only a ticket for one type of line, these classes can be reused and, if needed, extended. Another advantage is the fact, that it represents a better object oriented design. If the decision comes to the second solution the implementation is relatively simple and clear. The line and line section classes will have `STRING` field where the string representation of the type will be, making sure with pre- and post conditions that the type string is a valid one. When the decision comes to the first suggested idea, there are two possible implementation which will be discussed in the next paragraph.

3. The first idea that comes into mind when thinking about representing the types of a line as a class is to have a type field in the line that is of type `TRAFFIC_TYPE` and contains a concrete subclass of it. The other solution that is less usual is using *genericity*. A clear advantage of the genericity approach is that it makes the interface type-field free and easier to introduce new types. The type checking, that only bus line sections are added to a bus line is implicitly given. But here already comes the first problem. How can I really assure, that only bus line sections are added to a bus line, when a general notion of the abstract class is used in the declaration. I would have to make sure that really only allowed types of line sections were added to a line through pre- and post conditions and for these I would need some type of string representation again. How else could I find out what type a line section or line really is of? In addition genericity of a line or line section has quite some impact on so many classes (see notes on line, point 2, and line section, point 6).
4. When the decision falls on subclasses and a field indicating the type of a line (or line section) another simplification comes into mind. One can think of using *once* routines for the creation of the types in a factory. This ensures, that only one type object per type is in the system and so a simple reference comparison is sufficient to make sure two object are of the same traffic type.

4.9 Line section state

This class was introduced in order to be able to model states of a line sections. Such states can be “normal”, “collision”, “under construction” and many more. The idea is to make line sections more dynamic and model more complex situations like a collision. In such a situation one may think that in a tram an announcement would inform the passengers of switching routes. The design chosen for this class was a constants class and a kind of a proxy class. The discussion about the implementation of the idea of states comes right after the overview

of the layout of the classes `TRAFFIC_LINE_SECTION_STATE`, which models the state, and the `TRAFFIC_LINE_SECTION_STATES_CONSTANTS` which is responsible for the different possible states.

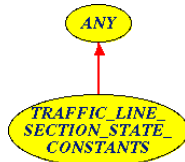


Figure 27: `TRAFFIC_LINE_SECTION_STATES_CONSTANTS`

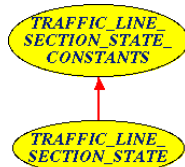


Figure 28: `TRAFFIC_LINE_SECTION_STATE`

1. The first decision to be taken is whether the states are to be *classes or constant strings*. The discussion is the same as with the traffic types argues for and against classes or strings (types points 2).
2. A new aspects comes into account for line section states. Here one can even think of *integer representations instead of strings*. The different states can easily be thought of as an integer instead of a meaningful names. If an integer representation is enough this is a good argument for introducing a constants class where all these integer states can be declared (possibly even with the use of the unique keyword, so one does not have to care about which number a state exactly has) and a way of displaying a state as a string. This design is easy to implement, understand and use.

4.10 Route

A route is a tour that visits a predefined set of places on a given map. Therefore the class `TRAFFIC_ROUTE` needs a reference to the map that contains the places that the route should visit. The design of the route class has changed quite a lot in comparison with the old design. The new route can calculate the route itself. In the old design a route calculator was needed in order to get the way through the city visiting all places. After the overview of the new route design as usual the discussion points on the `TRAFFIC_ROUTE` class.

be a walking station and one could walk to this entry from different public transportation stations and parking places.

1. The first question to be answered is whether *stations are needed* at all. A good reason to omit stations from the design is to make the overall design easier to understand and work with for first year students. An argument for stations is to have a distinction between places of interest to visit and public transportation stations.
2. The first design decision that will have to be made is, if stations can be *assigned to several lines or have to be line specific*. This decision depends on the use of stations. But a station to represent a stop where different lines of the same type or even different lines of different types can stop seems to be the most intuitive.
3. Another question to be answered is whether line sections are to connect stations and not places any more or if line sections and lines that drive to stations are some *specialized lines*.

4.12 Extension

Following just a few possible extensions that would be nice to have and make the TRAFFIC library model more realistic and fun to play with, but also more complicated to work with.

- time table
- cinema
- player
- car
- passenger (on places, between places, with moving direction)
- performance

4.13 XML-File

In order to be able to build up a predefined map as in the initial state model a XML-File is used. To be able to use this file as input a few changes had to be made (A more detailed overview will be given in [8.2](#)).

1. First the DTD file had to be updated with respect to the changes in the design. The initial state of the XML was retained for most parts; only minor changes were made.
2. The XML-File itself had to be updated with the changes.
3. The parser had to be updated as well. We could have made an analysis on the whole parser but this would have exceeded the scope of this semester thesis.

5 Test application (without GUI)

In order to test the most elementary functionality and show a small example of how to use the TRAFFIC library a test application has been written. To make the tests easier to read, we separated testing of the map and testing of the XML parsing. TRAFFIC_TEST_MAP tests the traffic map and all related classes, and TRAFFIC_TEST_XML tests the reading of a map from an XML-file. They are combined in the overall TRAFFIC_TEST class (see next figure).

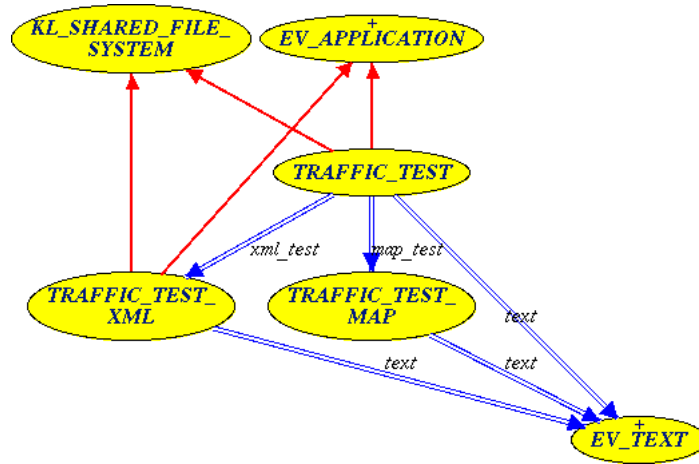


Figure 30: TRAFFIC_TEST

As can be seen in the figure, the test is simply embedded in the application. The core feature of the overall test class, TRAFFIC_TEST, is called test. First the traffic map test is called and then the XML-file-input test. The class is initialized with the window and the text window of the application.

```

test is
    -- Test Traffic framework.
    local
        b: BOOLEAN
    do
        text.append_text ("TRAFFIC TEST%N")
        text.append_text ("-----%N")
        text.append_text ("test map:%N
            -----%N%N")
        map_test.test
        text.append_text ("%Ntest xml:%N
            -----%N")
        xml_test.test
    end

feature {NONE} -- Implementation

text: EV_TEXT
    
```



```

visit_places is
  -- Visit places.
  local
    l_places: LINKED_LIST [TRAFFIC_PLACE]
  do
    create l_places.make
    l_places.extend (map.place ("Hauptbahnhof"))
    l_places.extend (map.place ("Kunsthaus"))
    create route.make (l_places, map)
    route.calculate_shortest_path
  end
    
```

Listing 1: test feature TRAFFIC_TEST_MAP

The XML test needs in addition to the text output field of the window a reference to the main window itself to be able to display the file dialog correctly. When a file name of an XML file has been entered, the application tries to parse the file into a map and generates a textual output of the generated map. For the city map of Zurich the same route is calculated as for the map test to have a comparison.

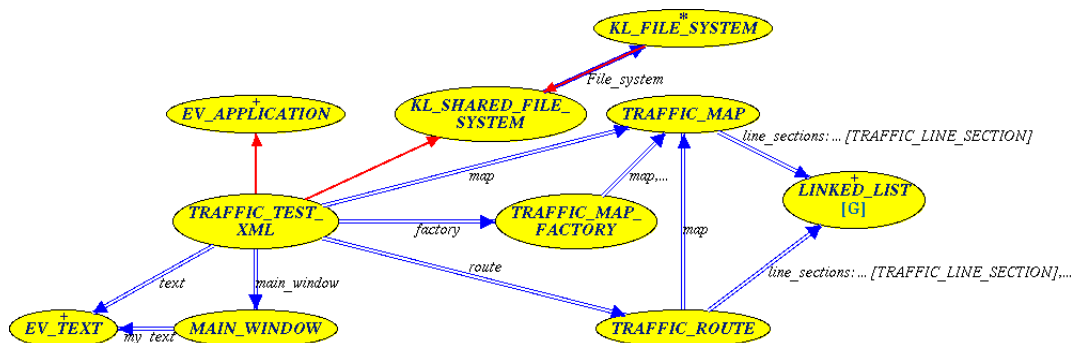


Figure 32: TRAFFIC_TEST_XML

```

test is
  -- Test XML and generate text output.
  do
    request_open_city -- get the path to the xml file
    text.append_text ("%NNxml-file parsed!%N%N")
    text.append_text (factory.map.out) -- display textual
      representation of generated map
    map := factory.map
    if equal (map.name.substring (1, 6), "Zurich") then
      visit_places_zurich
      text.append_text ("%N")
      text.append_text (route.out)
    end
  end
end
    
```

Listing 2: TRAFFIC_TEST feature test and needed references.

5.1 How to run the test application

To run the test, simply start the application then choose file > open.

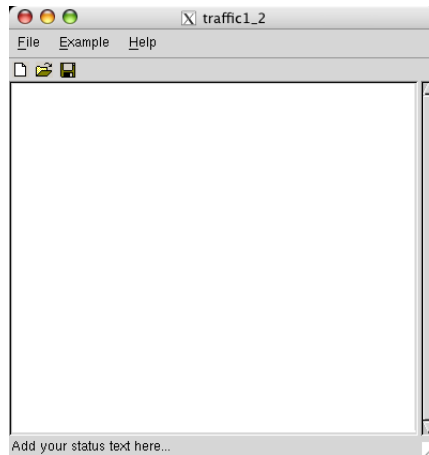


Figure 33: Choose file > open

For the test of the XML-file-input you are asked to choose an XML-file.

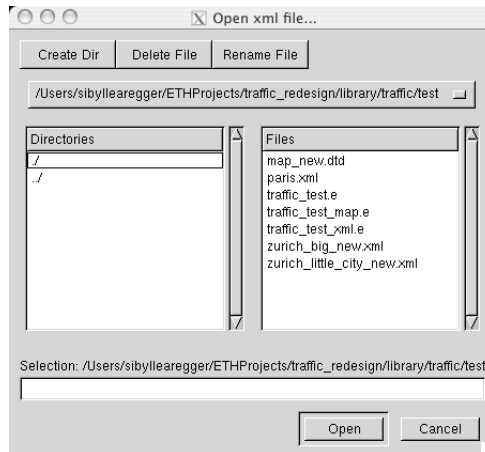
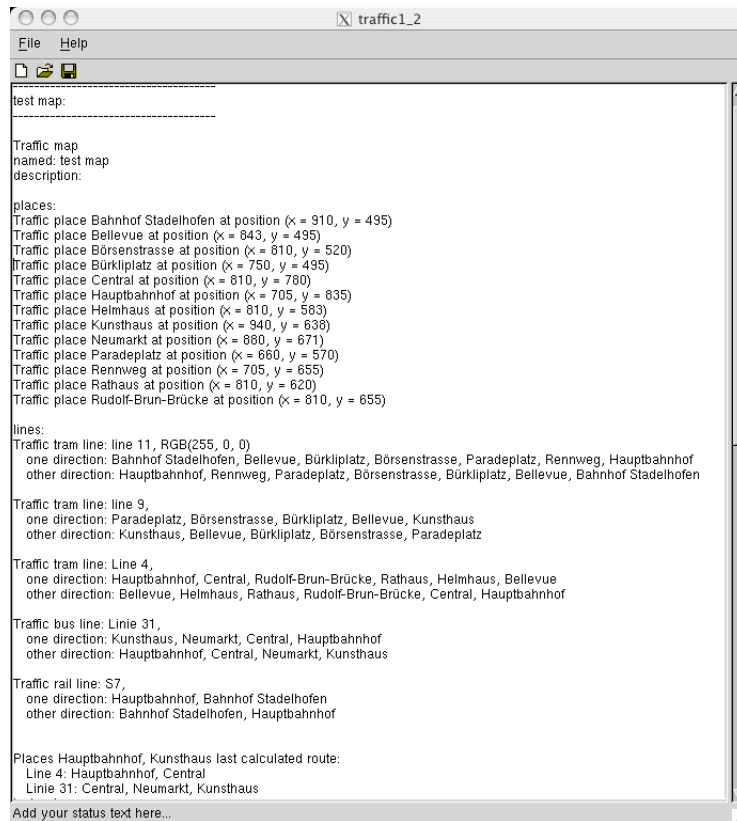


Figure 34: Choose an XML file

Make sure that the path of the XML-file also contains the dtd file. As a result a text output will be generated that shows what has been created in the different maps. If a route has been calculated it shows its way from one place to another.



```
traffic1_2
File Help
test map:
-----
Traffic map
named: test map
description:

places:
Traffic place Bahnhof Stadelhofen at position (x = 910, y = 495)
Traffic place Bellevue at position (x = 843, y = 495)
Traffic place Börsenstrasse at position (x = 810, y = 520)
Traffic place Bürkliplatz at position (x = 750, y = 495)
Traffic place Central at position (x = 810, y = 780)
Traffic place Hauptbahnhof at position (x = 705, y = 635)
Traffic place Helmhaus at position (x = 810, y = 583)
Traffic place Kunsthaus at position (x = 940, y = 638)
Traffic place Neumarkt at position (x = 880, y = 671)
Traffic place Paradeplatz at position (x = 660, y = 570)
Traffic place Rennweg at position (x = 705, y = 655)
Traffic place Rathaus at position (x = 810, y = 620)
Traffic place Rudolf-Brun-Brücke at position (x = 810, y = 655)

lines:
Traffic tram line: line 11, RGB(255, 0, 0)
one direction: Bahnhof Stadelhofen, Bellevue, Bürkliplatz, Börsenstrasse, Paradeplatz, Rennweg, Hauptbahnhof
other direction: Hauptbahnhof, Rennweg, Paradeplatz, Börsenstrasse, Bürkliplatz, Bellevue, Bahnhof Stadelhofen

Traffic tram line: line 9,
one direction: Paradeplatz, Börsenstrasse, Bürkliplatz, Bellevue, Kunsthaus
other direction: Kunsthaus, Bellevue, Bürkliplatz, Börsenstrasse, Paradeplatz

Traffic tram line: Line 4,
one direction: Hauptbahnhof, Central, Rudolf-Brun-Brücke, Rathaus, Helmhaus, Bellevue
other direction: Bellevue, Helmhaus, Rathaus, Rudolf-Brun-Brücke, Central, Hauptbahnhof

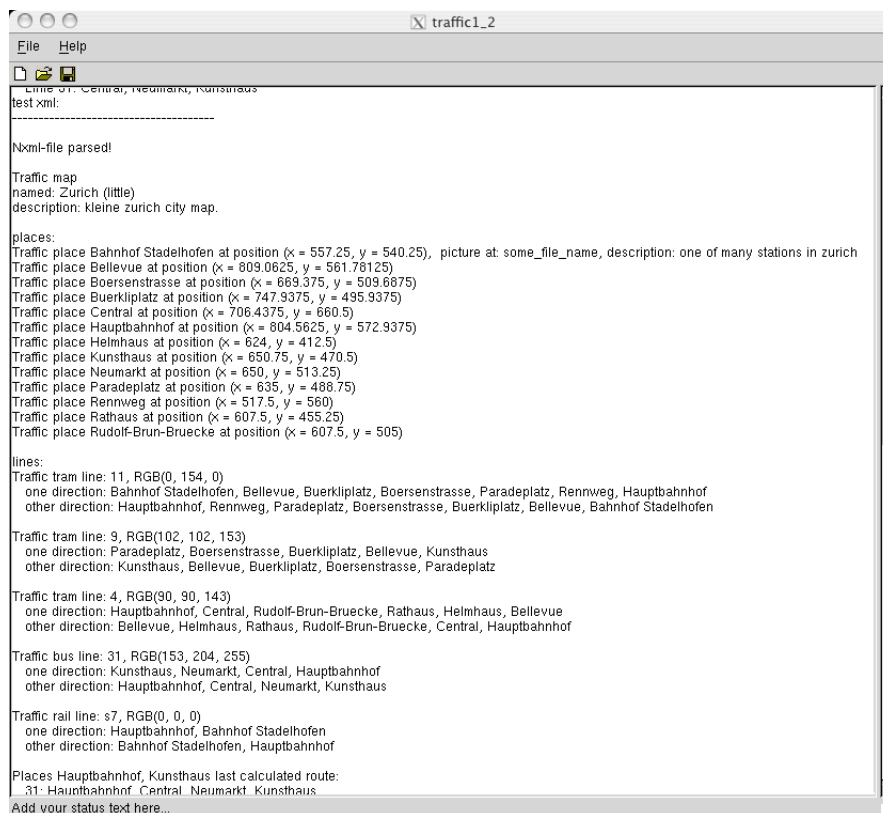
Traffic bus line: Linie 31,
one direction: Kunsthaus, Neumarkt, Central, Hauptbahnhof
other direction: Hauptbahnhof, Central, Neumarkt, Kunsthaus

Traffic rail line: S7,
one direction: Hauptbahnhof, Bahnhof Stadelhofen
other direction: Bahnhof Stadelhofen, Hauptbahnhof

Places Hauptbahnhof, Kunsthaus last calculated route:
Line 4: Hauptbahnhof, Central
Linie 31: Central, Neumarkt, Kunsthaus

Add your status text here...
```

Figure 35: Output of map test



```
Line of: Central, Neumarkt, Kunsthaus
test.xml:
-----
Nxml-file parsed!

Traffic map
named: Zurich (little)
description: kleine zurich city map.

places:
Traffic place Bahnhof Stadelhofen at position (x = 557.25, y = 540.25), picture at: some_file_name, description: one of many stations in zurich
Traffic place Bellevue at position (x = 809.0625, y = 561.78125)
Traffic place Boersenstrasse at position (x = 669.375, y = 509.6875)
Traffic place Buerkliplatz at position (x = 747.9375, y = 495.9375)
Traffic place Central at position (x = 706.4375, y = 660.5)
Traffic place Hauptbahnhof at position (x = 804.5625, y = 572.9375)
Traffic place Helmhaus at position (x = 624, y = 412.5)
Traffic place Kunsthaus at position (x = 650.75, y = 470.5)
Traffic place Neumarkt at position (x = 650, y = 513.25)
Traffic place Paradeplatz at position (x = 635, y = 466.75)
Traffic place Rennweg at position (x = 517.5, y = 560)
Traffic place Rathaus at position (x = 607.5, y = 455.25)
Traffic place Rudolf-Brun-Bruecke at position (x = 607.5, y = 505)

lines:
Traffic tram line: 11, RGB(0, 154, 0)
one direction: Bahnhof Stadelhofen, Bellevue, Buerkliplatz, Boersenstrasse, Paradeplatz, Rennweg, Hauptbahnhof
other direction: Hauptbahnhof, Rennweg, Paradeplatz, Boersenstrasse, Buerkliplatz, Bellevue, Bahnhof Stadelhofen

Traffic tram line: 9, RGB(102, 102, 153)
one direction: Paradeplatz, Boersenstrasse, Buerkliplatz, Bellevue, Kunsthaus
other direction: Kunsthaus, Bellevue, Buerkliplatz, Boersenstrasse, Paradeplatz

Traffic tram line: 4, RGB(90, 90, 143)
one direction: Hauptbahnhof, Central, Rudolf-Brun-Bruecke, Rathaus, Helmhaus, Bellevue
other direction: Bellevue, Helmhaus, Rathaus, Rudolf-Brun-Bruecke, Central, Hauptbahnhof

Traffic bus line: 31, RGB(153, 204, 255)
one direction: Kunsthaus, Neumarkt, Central, Hauptbahnhof
other direction: Hauptbahnhof, Central, Neumarkt, Kunsthaus

Traffic rail line: s7, RGB(0, 0, 0)
one direction: Hauptbahnhof, Bahnhof Stadelhofen
other direction: Bahnhof Stadelhofen, Hauptbahnhof

Places Hauptbahnhof, Kunsthaus last calculated route:
31: Hauptbahnhof, Central, Neumarkt, Kunsthaus
Add your status text here...
```

Figure 36: Output of XML test

6 Experiences

This section will cover my own experience with the EIFFEL system and the TRAFFIC library. The main idea is to show where problems can lie for the unexperienced user, as I was with the EIFFEL system and language. I've had a semesters course on object oriented languages some years ago where I collected some first step experience also on EIFFEL. So I nearly had to learn the environment from the scratch. Of course I have some knowhow with other programming environment systems so it can not be compared to a first years students knowledge.

- One of the first pitfalls was when wanting to show the BON overview of the class and remove a class from the view. I wasn't aware of the fact that arranging the view may changed the underlying framework. So creating an overview of my class always felt a little bit queasy to me at the beginning.
- Another problem not quite intuitive to me was to move a class into another cluster in the EIFFEL studio. I tried to just drag and drop it in the class panes overview. But this sure enough would not work. And as with the BON overview I felt not safe. Now I usually replace the classes on the file

system itself and recompiled the project. I am still not quite sure what the proper way would have been.

- Sometimes I had problems displaying the information in the lower window at the right hand side. I never found out exactly why. For a class not compiled with the project no such information would be available but it also seemed to have some problems with a special kind of layout of the class text.
- A similar displaying problem was feature information of a class on the left hand side window. If a class was not integrated in the project no such information would be available, so I used a dummy reference to an object of the class if I wanted to have the class information. The second problem with the overview of the feature was when a class had some compile errors. Then the overview would get illegible sometimes too, which was a pity, because it made debugging the class harder.
- When one wants to see how the code is being executed breakpoints are a nice way to stop at the place of interest. Unfortunately this too, wasn't an intuitive task for me. Possibly because of my experience with other developing environments and the resulting experience how breakpoints can be set.
- Another uncertain task for me was to compile a project. I didn't quite understand when I had to compile what to get the intended result. This maybe again comes from experiences with other systems and the resulting expectations on how this has to be done.
- The updating of the ace-file was a problem at the beginning, too. Again, it is something unknown and though an uneasy feeling accompanies each change one tries to do.
- When I started to document my classes with BON diagrams out of the EIFFEL studio I wanted to have some feature names assigned to classes as I had seen in Books and Papers. But unfortunately I did not find a way to do this without a big effort of copying.
- A problem I also had with the EIFFEL compiler, as with any other compiler, was that especially at the beginning outputs sometimes seem not very intuitive. I especially had this problem with genericity involved where I assume the problem lies in the fact that I am not used to working with genericity.
- The biggest trouble I had with adding and removing genericity. For adding it seemed to be essential in which order the genericity was introduced. The main problem lay in the fact that there was a circular dependency of two classes, each having a reference to the other and both becoming generic classes. Removing the once introduced genericity in the more evolved system afterwards was even a harder task to manage.
- The most enjoyable moments always were, when finally one of the above problems was solved. :-)

7 Conclusion

The goal of this semester thesis was to redesign the existing TRAFFIC framework with respect to ease of understandability, ease of usage and ease of extension. The library has to be exemplary object-oriented, easy to use and easy to remember. Additionally a good documentation for both students and developers has to be provided.

The implementation of the library has turned out to be a constant trade off in respect to all important goals. All goals have to be balanced all the time and finally a decision for or against one of the goals has to be taken and implemented as consequently as possible. The most important outcome of this semester thesis is, that those decisions are well documented. If the main reason on which any decision builds on changes, the implementation should change as well.

The documentation turned out to be more difficult than expected. Documenting first years students guide requires additional care.

Even though there some improvements and especially extensions remain, it can be said, that the overall architecture is deliberately designed and well documented.

8 Appendix

8.1 Student Guide

This student guide consists of a description of the intention and use of the TRAFFIC library with a description of each class and its simplified interface. The class overviews are sorted in ascending order to be able to find a class' description faster when knowing the name of it. For each class a short example of use will be added to show how it can be used. Those examples will be found in the cluster test ▷ example.

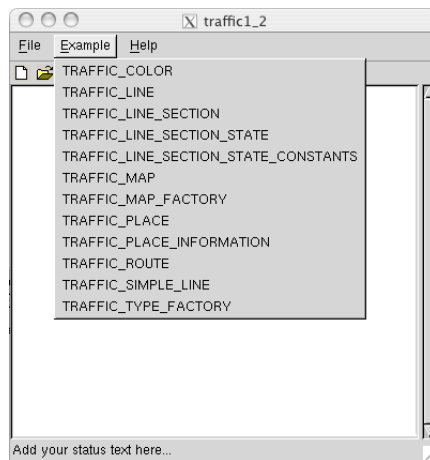


Figure 37: Window with test examples

In order to be able to see some textual output on the screen the example itself generates text output. To understand the code you need to know, that there always is defined a field

```
text := STRING -- Text output
```

in the class where textual output can be inserted. Text output can be generated by

```
text := "some text"
```

or

```
text := some_object.query_returning_text_output
```

or several strings can be added with the “+” operator. When appending more text output to a string

```
text := text + extended_string
```

is used.

8.1.1 TRAFFIC overview

Let us start with a short overview of the TRAFFIC library.

The TRAFFIC library was written to model a city and its public transportation system. Therefore you can imagine the library delivering you with all you need to build and work on a city map. This map can contain places with landmarks and public transportation lines. The library provides you with all you need and you even can get a tour through the city visiting all your places of interest. The overall model of the library is shown in the following figure.

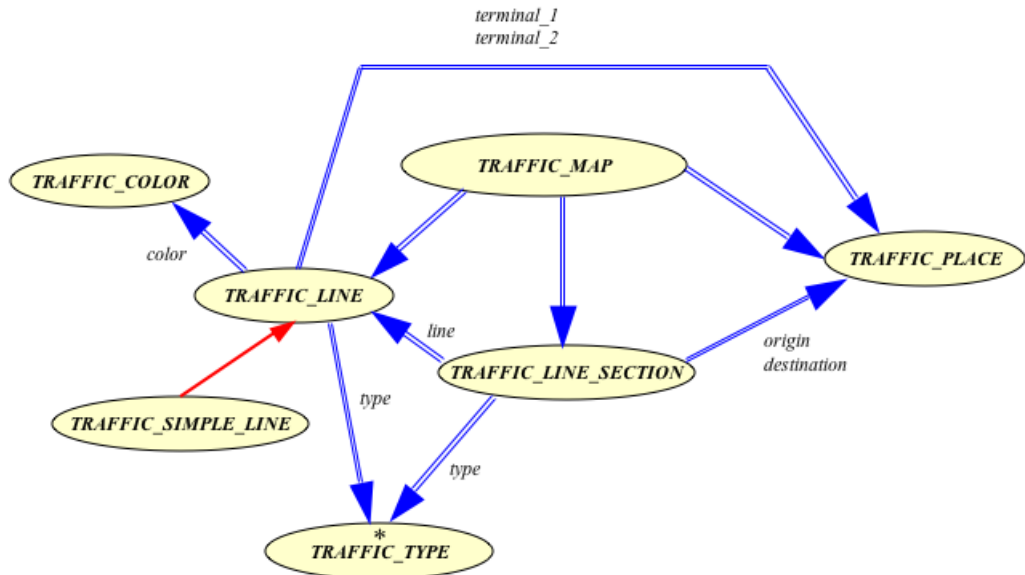


Figure 38: General view of the TRAFFIC library

As is visible from the picture, the most important class is the TRAFFIC_MAP class. It describes the map of a city and its public transportation system. Such a map consists of places and traffic transportation lines. Those lines in turn are made up of line sections. With this knowledge you can already build a map!

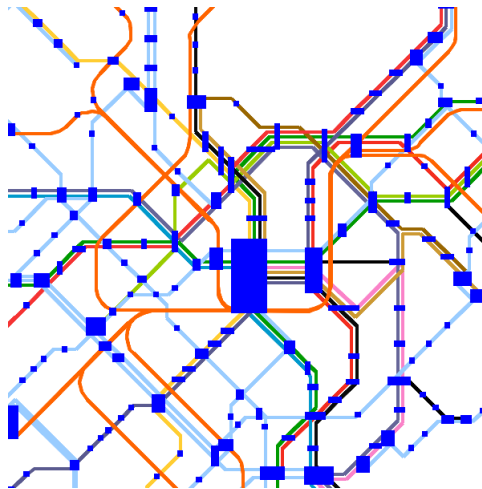


Figure 39: Visual representation of a map

In the following sections the classes will be described. The layout for each class is: Requirements, Description, Class overview, Class interface, and usually an Example. In the Requirements the name of classes you need to know in order to understand the class described is given. Description is a short description of

what the class does. It is more detailed than just the class description that you find in the class header. The class overview provides you with a class diagram and a rough description thereof. In the section about the class interface you find the interfaces of the most important features of a class and their more precise description. Finally a short example is delivered in the Example section.

8.1.2 TRAFFIC_COLOR

Requirements

None

Description

The class TRAFFIC_COLOR represents a RGB-Color. A RGB-Color is a color composed of the three additive components: red, green, blue.

Class overview



Figure 40: TRAFFIC_COLOR

As can be seen from the class overview the color class is a simple class implementing the most important features needed to describe and change a color.

Class interface

The three color parts can be accessed through the following three features.

feature -- Access

red: *INTEGER*
-- Red part of rgb-color.

green: *INTEGER*
-- Green part of rgb-color.

blue: *INTEGER*
-- Blue part of rgb-color.

An important feature is *is_valid_color_part* which makes sure only valid integer values for the rgb-parts are used.

feature -- Status report

```
is_valid_rgb_part (a_part: INTEGER): BOOLEAN
  -- Is 'a_part' a valid part of rgb-color?
```

The object is created through the *make* feature call which sets the color parts to the passed values.

create

```
make (a_red, a_green, a_blue: INTEGER)
  -- Create a rgb-color.
require
  a_red_valid: is_valid_rgb_part (a_red)
  a_green_valid: is_valid_rgb_part (a_green)
  a_blue_valid: is_valid_rgb_part (a_blue)
ensure
  red_set: red = a_red
  green_set: green = a_green
  blue_set: blue = a_blue
```

The individual color parts can be changed through the *set_color* commands. Make sure the value you want to change the color to is valid. For this the *is_valid_color_part* query can be used.

feature -- Element change

```
set_red (a_red: INTEGER)
  -- Set red to 'a_red'.
require
  a_red_valid: is_valid_rgb_part (a_red)
ensure
  red_set: red = a_red

set_green (a_green: INTEGER)
  -- Set green to 'a_green'.
require
  a_green_valid: is_valid_rgb_part (a_green)
ensure
  green_set: green = a_green
end

set_blue (a_blue: INTEGER)
  -- Set blue to 'a_blue'.
require
  a_blue_valid: is_valid_rgb_part (a_blue)
ensure
  blue_set: blue = a_blue
end
```

Example

```
feature -- Basic operation

  run is
    -- Run example demonstration.
  local
    b: BOOLEAN
  do
    -- create blue color
    create color.make (0, 0, 255)

    -- display color
    text := text + "initial color: " + color.out

    -- test if an integer is a valid part of a color
    b := color.is_valid_rgb_part (300)
    text := text + "%N300 is a " + b.out + " part of a color!"

    -- create new composition of color resulting in green color
    color.set_red (0)
    color.set_green (255)
    color.set_blue (0)

    -- display new color
    text := text + "%Nnewly created color: " + color.out
  end
```

```
TRAFFIC_COLOR example:
initial color: RGB(0, 0, 255)
300 is a False part of a color!
newly created color: RGB(0, 255, 0)
```

Figure 41: generated output

8.1.3 TRAFFIC_LINE

Requirements

TRAFFIC_COLOR (8.1.2), TRAFFIC_PLACE(8.1.9), TRAFFIC_LINE_SECTION (8.1.4), TRAFFIC_TYPE (8.1.13)

Description

The class TRAFFIC_LINE models a line of a public transportation system. Each line has a type, e.g. TRAFFIC_TYPE_BUS for a bus line, TRAFFIC_TYPE_RAIL for a rail line and so on. This type can be accessed through the feature *type*. In addition a line can have up to two directions. A line which only has one direction is also valid. Each direction has a terminal place.

Class overview

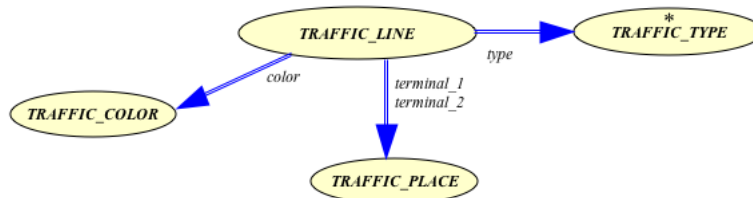


Figure 42: TRAFFIC_LINE

Class interface

A new line is created through the feature *make*. It takes as input the name of your new line and a traffic type.

```

create
  make (a_name: STRING; a_type: TRAFFIC_TYPE)
    -- Create a line with name 'a_name' of type 'a_type'.
  require
    a_name_exists: a_name /= Void
    a_name_not_empty: not a_name.is_empty
    a_type_exists: a_type /= Void
  ensure
    name_set: equal (name, a_name)
    type_set: type = a_type -- have to be same object
    count_line_section_not_void: count >= 0 -- List is
      initilalized.
    places_one_direction_exists: places_one_direction /= Void
    places_other_direction_exists: places_other_direction /=
      Void
  
```

The *name*, *color*, *type*, *terminal_1*, *terminal_2* can directly be called. If you want to get the starting place of a direction of your line use the feature *start_to_terminal*. It takes as input either the place in *terminal_1* or *terminal_2*. If you use just a place, make sure it is a terminal in a direction of the line.

```

feature -- Access

  name: STRING
    -- Name of line.

  type: TRAFFIC_TYPE
    -- Type of line.

  terminal_1: TRAFFIC_PLACE
    -- Terminal of line in one direction.
  
```

```
terminal_2: TRAFFIC_PLACE
  -- Terminal of line in other direction.

color: TRAFFIC_COLOR
  -- Line color.
  -- Used as color representation.

start_to_terminal (a_terminal: TRAFFIC_PLACE):
  TRAFFIC_PLACE
  -- The start place to existing terminal 'a_terminal'.
require
  a_terminal_exists: a_terminal /= Void
  a_terminal_valid: is_terminal (a_terminal)
ensure
  result_exists: Result /= Void
```

To make sure that a place is a terminal the *is_terminal* query can be used. The features *one_direction_exists* and *other_direction_exists* return true if one or other direction (depending on the feature you call) exists. Only if a direction exists there exists a terminal and a starting place. The query *is_valid_for_insertion* tells you if a line section, as it is, can be inserted into the line. The query *is_valid_insertion* tells you if it is possible for a line section of the correct type from a origin place to a destination place is a possible extension of the line in any direction. So the argument in the second query (*is_valid_insertion*) is only the origin and destination of a fictionary or real line section.

feature -- Status report

```
is_terminal (a_terminal: TRAFFIC_PLACE): BOOLEAN
  -- Is 'a_terminal' a terminal of line?
require
  a_terminal_exists: a_terminal /= Void

one_direction_exists: BOOLEAN is
  -- Does line have line section(s) in one direction?

other_direction_exists: BOOLEAN
  -- Does line have line section(s) in other direction?

is_valid_for_insertion (a_line_section:
  TRAFFIC_LINE_SECTION): BOOLEAN
  -- Can 'a_line_section' be added to line?
  --
  -- This is the case if it can be added in front or back of an
  -- existing direction,
  -- or at least one direction does not yet exists.
  -- The destination of 'a_line_section' is
  -- not place allready use in this direction (circle).
  -- A line section can not be added twice to the same line.
```

```
-- A line section can not be added to two lines at the same  
time.
```

```
require  
  a_line_section_exists: a_line_section /= Void
```

```
is_valid_insertion (a_origin, a_destination: TRAFFIC_PLACE):  
  BOOLEAN
```

```
-- Can a line_section from 'a_origin' to 'a_destination' be  
added  
-- in front or back of this line?
```

```
require  
  a_origin_exists: a_origin /= Void  
  a_destination_exists: a_destination /= Void
```

The color of a line can be changed or removed.

feature -- Element change

```
set_color (a_color: TRAFFIC_COLOR)  
  -- Set color to 'a_color'.
```

```
require  
  a_color_exists: a_color /= Void  
ensure  
  color_set: equal(color, a_color)
```

feature -- Removal

```
remove_color  
  -- Remove color.
```

```
ensure  
  color_removed: color = Void
```

Finally the probably most important feature. *Extend* adds a line section to the line where it fits. So directions can be extended at both their beginning and their end.

feature -- Basic operations

```
extend (a_line_section: TRAFFIC_LINE_SECTION)  
  -- Add 'a_line_section' at beginning or end of existing  
  direction(s).
```

```
require else  
  a_line_section_exists: a_line_section /= Void  
  a_line_section_valid_for_insertion: is_valid_for_insertion (  
    a_line_section)  
ensure  
  a_line_section_in_line: has (a_line_section)  
  line_added_to_line_section: a_line_section.line = Current
```

Example

```
run is
  -- Run example demonstration.
local
  b: BOOLEAN
  blue: TRAFFIC_COLOR
  station_home, home_station: TRAFFIC_LINE_SECTION
  station, home: TRAFFIC_PLACE
do
  -- create station and home place
  create station.make ("rail station")
  create home.make ("sweet home")

  -- create blue line
  create line.make ("VIP bus", type)

  -- if station-home is a valid insertion, create line section
  -- from station to home and add it to the line
  if line.is_valid_insertion (station, home) then
    create station_home.make (station, home, type)
    -- add line section
    line.extend (station_home)
  end

  -- display line
  text := text + "initial line: " + line.out

  -- create blue color
  create blue.make (0, 0, 255)
  -- set the line's color to blue
  line.set_color (blue)

  -- create line section from home to station
  create home_station.make (home, station, type)
  -- is home-station a valid line section to be
  -- inserted in line?
  if line.is_valid_for_insertion (home_station) then
    -- add other line section
    line.extend (home_station)
  end

  -- display new line
  text := text + "%Nchanged line: " + line.out
end
```

```

TRAFFIC_LINE example:
initial line: Traffic bus line: VIP bus,
  one direction: rail station, sweet home
  other direction:
changed line: Traffic bus line: VIP bus, RGB(0, 0, 255)
  one direction: rail station, sweet home
  other direction: sweet home, rail station
    
```

Figure 43: generated output

8.1.4 TRAFFIC_LINE_SECTION

Requirements

TRAFFIC_PLACE (8.1.9), TRAFFIC_LINE_SECTION (8.1.13), TRAFFIC_LINE_STATE (8.1.5)

Description

The class TRAFFIC_LINE_SECTION represents a connection of a line from one place to another. Those places are called origin and destination. A line section is of some traffic type, e.g. TRAFFIC_BUS_TYPE. A line section can be added to a line, forming the lines connection. It can belong to at most one line. Additionally a line section can have different states.

Class overview

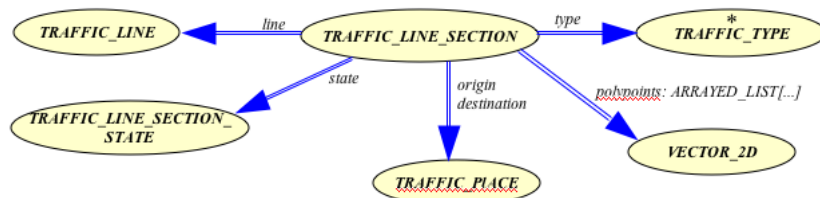


Figure 44: TRAFFIC_LINE_SECTION

Class interface

To create a new line section an origin place, a destination place and a traffic type have to be defined. Such a line section will not belong to any line, its state will be set to some normal state and the polypoints are empty.

create

```

make (a_origin, a_destination: TRAFFIC_PLACE; a_type:
    TRAFFIC_TYPE )
    
```

-- Initialize 'Current'.

require

a_origin_exists: a_origin /= Void

a_destination_exists: a_destination /= Void

```
    a_type_exists: a_type /= Void
  ensure
    origin_set: origin = a_origin
    destination_set: destination = a_destination
    state_exists: state /= Void
    type_set: type = a_type
    polypoints_exists: polypoints /= Void
```

All elements shown in the overview can be accessed. Additionally there is a feature called *length* which returns a calculated length of the line section. To calculate the length the polypoints are used, and if no polypoints exists, the position of the origin and destination place are used to calculate a length. This length becomes especially important when starting to calculate routes on lines and even a whole map of lines.

feature -- Access

```
line: TRAFFIC_LINE
  -- Line this line section belongs to.

type: TRAFFIC_TYPE
  -- Type of line section.

origin: TRAFFIC_PLACE
  -- Place of origin.

destination: TRAFFIC_PLACE
  -- Place of destination.

state: TRAFFIC_LINE_SECTION_STATE
  -- State of line section.

polypoints: ARRAYED_LIST [VECTOR_2D]
  -- position representation of line section.

length: DOUBLE
  -- Length from start of polypoints to end.
  -- If no polypoints exists, distance between origin and
  destination.
```

The state can be changed through the command *set_state*. Polypoints can be added or removed.

feature -- Element change

```
set_state (a_state: TRAFFIC_LINE_SECTION_STATE )
  -- Change state to 'a_state'.
  require
    a_state_exists: a_state /= Void
  ensure
    state_set: state = a_state
```

```
set_polypoints (a_polypoints: ARRAYED_LIST [VECTOR_2D])
  -- Set polypoints to 'a_polypoints'.
  require
    a_polypoints_exist: a_polypoints /= Void
  ensure
    equal (polypoints, a_polypoints)
    polypoints_exists: polypoints /= Void
    polypoints_equal: polypoints.count > 0 implies equal (
      polypoints, a_polypoints)

feature -- Removal

  remove_polypoints
    -- Remove polypoints.
```

The changement of the attached line can only be carried out by a line. This is due to the fact, that when a line section has a line attached it should be in this line. Therefore the line is responsible to update line sections that are added to or removed from it.

Example

```
run is
  -- Run example demonstration.
local
  b: BOOLEAN
  station, home: TRAFFIC_PLACE
  station_home: TRAFFIC_LINE_SECTION
  collision_state: TRAFFIC_LINE_SECTION_STATE
do
  -- create station and home place
  create station.make ("rail station")
  create home.make ("sweet home")

  -- create line section from station to home
  create station_home.make (station, home, type)

  -- display line
  text := text + "initial line section: " + station_home.out

  -- create collision state
  create collision_state.make
  collision_state.set_state (collision_state.State_collision)

  -- set collision on line section
  station_home.set_state (collision_state)

  -- display new line
```

```
text := text + "%Nchanged line section: " + station_home.out
end
```

TRAFFIC_LINE_SECTION example:
initial line section: Traffic bus line section, state: normal, from rail station to sweet home
changed line section: Traffic bus line section, state: collision, from rail station to sweet home

Figure 45: generated output

8.1.5 TRAFFIC_LINE_SECTION_STATE

Requirements

TRAFFIC_LINE_SECTION_STATE_CONSTANTS (8.1.6)

Description

The class TRAFFIC_LINE_SECTION_STATE provides the interface to define states and attach them to line sections. The available states are defined from the TRAFFIC_LINE_SECTION_STATE_CONSTANTS class.

Class overview

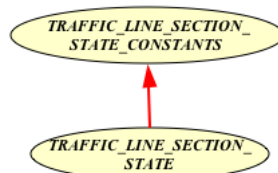


Figure 46: TRAFFIC_LINE_SECTION_STATE

Class interface

A state is simply created by calling its creation feature. The feature *make* always creates a normal state. To change it, call the feature *set_state* with one of the values defined in TRAFFIC_LINE_SECTION_STATE_CONSTANTS.

create

```
make
  -- Set state to normal.
ensure
  value_set: value /= Void
  value_valid: is_valid_state_value (value)
```

The state value can be accessed through the feature *value*.

feature *-- Access*

```
value: INTEGER
    -- Current state.
```

Finally the state can be changed returning you by the next call of *value* the new set state value.

feature *-- Element change*

```
set_state (a_value: INTEGER)
    -- Set 'value' to 'a_value'.
require
    is_valid_state_value (a_value)
ensure
    value_set: value = a_value
```

Example

```
run is
    -- Run example demonstration.
local
    state: TRAFFIC_LINE_SECTION_STATE
do
    -- create state
    create state.make

    -- display state
    text := text + "initial state: " + state.out

    -- change state to collision
    state.set_state (state.State_collision)

    -- display new state
    text := text + "%Nnewly created state: " + state.out
end
```

```
TRAFFIC_LINE_SECTION_STATE example:
initial state: state: normal
newly created state: state: collision
```

Figure 47: generated output

8.1.6 TRAFFIC_LINE_SECTION_STATE_CONSTANTS

Requirements

None

Description

The class `TRAFFIC_LINE_SECTION_STATE_CONSTANTS` defines all line section states and a feature that tests any integer value to the states value and a feature that returns a string representation of the state value.

Class overview

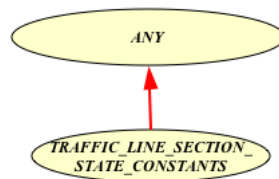


Figure 48: `TRAFFIC_LINE_SECTION_STATE_CONSTANTS`

Class interface

The states are defined as integer values.

feature `-- Access`

`State_normal, State_collision: INTEGER`
`-- State constants.`

For any integer value it can be tested if it is a valid state value.

feature `-- Status report`

`is_valid_state_value (a_state_value: INTEGER): BOOLEAN`
`-- Is 'a_state_value' a state constants?`
require
`a_state_value_set: a_state_value /= Void`

To generate readable state output the feature `value_to_string` is provided.

feature `-- Basic operation`

`value_to_string (a_state_value: INTEGER): STRING`
`-- String representation of state 'a_state_value'`
require
`a_state_value_exists: a_state_value /= Void`
`a_state_value_valid: is_valid_state_value (a_state_value)`

Example

`run is`
`-- Run example demonstration.`

```
local
  constants: TRAFFIC_LINE_SECTION_STATE_CONSTANTS
do
  -- create constants class
  create constants

  -- display normal state
  text := text + "normal state: " + constants.value_to_string (
    constants.State_normal)

  -- display collision state
  text := text + "%Nnormal state: " + constants.value_to_string
    (constants.State_collision)
end
```

```
ITRAFFIC_LINE_SECTION_STATE_CONSTANTS example:
normal state: normal
normal state: collision
```

Figure 49: generated output

8.1.7 TRAFFIC_MAP

Requirements

TRAFFIC_PLACE (8.1.9), TRAFFIC_LINE (8.1.3), TRAFFIC_LINE_SECTION (8.1.4)

Description

The TRAFFIC_MAP is the collection of all lines, line sections and places. All elements are administrated by the map. Through the name of a place or a line you can retrieve it, you can search for line sections and even search shortest paths from one place to another. A map has a name and can additionally have a description that gives more information on the purpose of the map.

Class overview

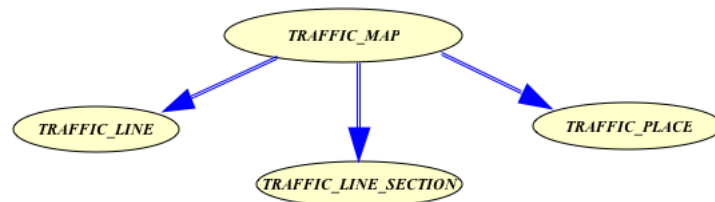


Figure 50: TRAFFIC_MAP

Class interface

Each map has a name. So when creating a new map you have to provide it with a name.

create

```
make (a_name: STRING)
  -- Create an empty map with name 'a_name'.
require
  a_name_exists: a_name /= Void
  a_name_not_empty: not a_name.is_empty
ensure
  name_set: equal (name, a_name)
  places_not_void: places /= Void
  lines_not_void: lines /= Void
  line_sections_not_void: line_sections /= Void
```

For all elements there exist features to find out if there is such an element in the map.

feature -- Status report

```
has_place (a_name: STRING): BOOLEAN
  -- Has traffic map place called 'a_name'?
require
  a_name_exists: a_name /= Void
  a_name_not_empty: not a_name.is_empty

has_line_section (a_origin_name, a_destination_name: STRING;
  a_traffic_type: TRAFFIC_TYPE; a_line: TRAFFIC_LINE):
  BOOLEAN
  -- Has traffic map line section 'a_line_section'?
require
  a_origin_exists: a_origin_name /= Void and not
    a_origin_name.is_empty
  a_destination_exists: a_destination_name /= Void and not
    a_destination_name.is_empty
  a_traffic_type_exists: a_traffic_type /= Void

has_line (a_name: STRING): BOOLEAN
  -- Has traffic map line 'a_name'?
require
  a_name_exists: a_name /= Void
  a_name_not_empty: not a_name.is_empty
```

For all elements exist features to add or set the values.

feature -- Element change

```
set_description (a_description: STRING)
```

```
    -- Set map description.
  ensure
    description_set: description = a_description

add_place (a_place: TRAFFIC_PLACE)
  -- Add place 'a_place' to map.
  require
    a_place_exists: a_place /= Void
    no_place_with_same_name_in_map: not has_place (
      a_place.name)
  ensure
    a_place_in_map: has_place (a_place.name)

add_line_section (a_line_section: TRAFFIC_LINE_SECTION)
  -- Add line section 'a_line_section' to map.
  require
    a_line_section_exists: a_line_section /= Void
    a_line_section_not_in_map: not has_line_section (
      a_line_section.origin.name, a_line_section.destination.
      name, a_line_section.type, a_line_section.line)
  ensure
    a_line_section_in_map: has_line_section (a_line_section.
      origin.name, a_line_section.destination.name,
      a_line_section.type, a_line_section.line)

add_line (a_line: TRAFFIC_LINE)
  -- Add line 'a_line' to map.
  require
    a_line_exists: a_line /= Void
    no_line_with_same_name_in_map: not has_line (a_line.
      name)
  ensure
    a_line_in_map: has_line (a_line.name)
```

The name and the description can be accessed through the features *name* and *description*. Additionally a place of a given name can be accessed through the features *place*.

```
feature -- Access

  name: STRING
    -- Name of region this map represents.

  description: STRING
    -- Textual description.

  place (a_name: STRING): TRAFFIC_PLACE
```

```
    -- Place named 'a_name'.  
require  
  a_name_exists: a_name /= Void  
  place_in_map: has_place (a_name)  
ensure  
  result_exists: Result /= Void
```

Example

```
TRAFFIC_MAP example:  
initial map: Traffic map  
named: vip map  
description:  
  
places:  
Traffic place rail station at position (x = 0, y = 0)  
Traffic place sweet home at position (x = 0, y = 0)  
  
lines:  
  
map with line: Traffic map  
named: vip map  
description:  
  
places:  
Traffic place rail station at position (x = 0, y = 0)  
Traffic place sweet home at position (x = 0, y = 0)  
  
lines:  
Traffic bus line: VIP bus,  
  one direction:  
  other direction:
```

Figure 51: generated output

```
run is  
  -- Run example demonstration.  
local  
  map: TRAFFIC_MAP  
  station_home, home_station: TRAFFIC_LINE_SECTION  
  station, home: TRAFFIC_PLACE  
  vip_line: TRAFFIC_LINE  
  type: TRAFFIC_TYPE_BUS  
do  
  -- create map  
  create map.make ("vip map")  
  
  -- create type  
  create type.make  
  
  -- create station and home place  
  create station.make ("rail station")  
  create home.make ("sweet home")  
  
  -- add places to map  
  map.add_place (station)  
  map.add_place (home)
```

```
-- create line sections
create station_home.make (station, home, type)
create home_station.make (home, station, type)

-- create line
create vip_line.make ("vip bus shuffle", type)

-- add line sections to line
vip_line.extend (station_home)
vip_line.extend (home_station)

-- display map
text := text + "initial map: " + map.out

-- create line
create vip_line.make ("VIP bus", type)

-- add line to map
map.add_line (vip_line)

-- display new map
text := text + "%Nmap with line: " + map.out
end
```

8.1.8 TRAFFIC_MAP_FACTORY

Requirements

TRAFFIC_PLACE (8.1.9), TRAFFIC_LINE_SECTION (8.1.4), TRAFFIC_LINE (8.1.3), TRAFFIC_SIMPLE_LINE (8.1.12), TRAFFIC_MAP (8.1.7), TRAFFIC_TYPE (8.1.13), TRAFFIC_TYPE_FACTORY (8.1.14)

Description

The TRAFFIC_MAP_FACTORY is used to create a map and its elements. The general principle is easy. Call the corresponding build feature. Test with the has-features if a valid object was created and access the last created object of a given element type with the corresponding query.

Class overview

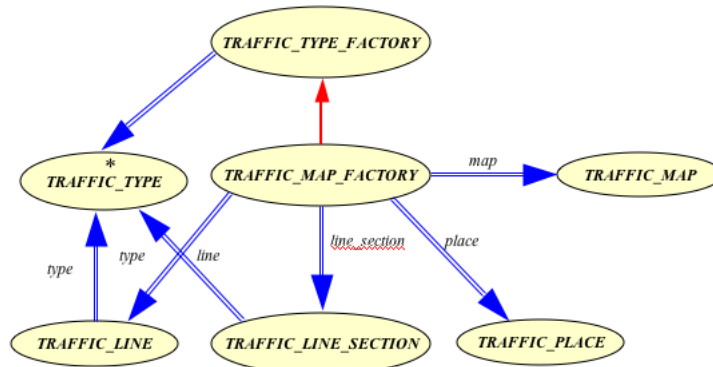


Figure 52: TRAFFIC_MAP_FACTORY

Class interface

Create a new factory by calling it's *make* feature. This creates a new, blank factory.

create

make

-- Initialize 'Current'.

ensure then

everything_void: *internal_map* = Void **and**
internal_place = Void **and**
internal_line_section = Void **and**
internal_line = Void

With the *reset* feature you can reset a factory to create a new map and it's elements from scratch.

feature -- Initialization

reset_factory **is**

-- Reset traffic map factory.

do

internal_map := Void
internal_place := Void
internal_line_section := Void
internal_line := Void

ensure

internal_traffic_type_is_void: *internal_traffic_type* = Void
type_table_exists: *type_table* /= Void
everything_void: *internal_map* = Void **and**
internal_place = Void **and**
internal_line_section = Void **and**
internal_line = Void

At first you possibly want to build a map to be able to insert the maps elements afterwards. Therefore this overview starts with the features related with the building of the map. The build feature that builds a named map is called *build_map*. The query *map* returns you the last created map. With the call to the query *has_map* you make sure a valid map exists. After a successful call to this feature it is safe to call *map*.

feature -- Traffic map building

```
build_map (a_name: STRING)
  -- Generate new map object with name 'a_name'.
  -- (Access generated object through feature 'map')
  require
    a_name_exists: a_name /= Void
    a_name_not_empty: not a_name.is_empty
  ensure
    map_created: map /= Void
    map_has_name: equal (map.name, a_name)

map: TRAFFIC_MAP
  -- Generated traffic map object.
  require
    map_available: has_map
  ensure
    Result_exists: Result /= Void

has_map: BOOLEAN
  -- Is traffic map object available?
```

For the building of a place two features exist. Either you build a standard place whose position is at the origin (0,0) or you build a place with another position. The call for the later option is called *build_place_with_position*. The features *has_place* and *place* have the same meaning as already mentioned generally in the description of this class.

feature -- Traffic place building

```
build_place (a_name: STRING; a_map: TRAFFIC_MAP)
  -- Generate new traffic place object with name 'a_name'
  -- belonging to traffic map 'a_map'.
  -- (Access generated object through feature 'place')
  require
    a_name_exists: a_name /= Void
    a_name_not_empty: not a_name.is_empty
    map_exists: has_map
    unique_name: not a_map.has_place (a_name)
  ensure
    place_created: place /= Void
    place_has_name: equal (place.name, a_name)
    place_in_map: a_map.has_place (a_name)
```

```

build_place_with_position (a_name: STRING; a_x, a_y:
  INTEGER; a_map: TRAFFIC_MAP)
  -- Generate new traffic place object with name 'a_name'
  and
  -- position ('a_x', 'a_y') belonging to traffic map 'a_map'.
  -- (Access generated object through feature 'place')
require
  a_name_exists: a_name /= Void
  a_name_not_empty: not a_name.is_empty
  a_x_exists: a_x /= Void
  a_y_exists: a_y /= Void
  map_exists: has_map
  unique_name: not a_map.has_place (a_name)
ensure
  place_created: place /= Void
  place_has_name: equal (place.name, a_name)
  place_in_map: a_map.has_place (a_name)

place: TRAFFIC_PLACE
  -- Generated traffic place object.
require
  place_available: has_place
ensure
  Result_exists: Result /= Void

has_place: BOOLEAN
  -- Is traffic place object available?
  
```

The building and accessing of a line section object works as with the other elements. The build features may look a little bit complicated, but that is just, because the definition of a line section is quite large. A line section has an origin and destination place and a type. Additionally it can have polypoints defining its appearance and a line it belongs to. Therefore so many arguments have to be given to the build features of the line section.

feature -- Line section building

```

build_line_section (a_origin, a_destination:STRING; a_polypoints
  : ARRAYED_LIST [VECTOR_2D]; a_map: TRAFFIC_MAP
  ; a_line: TRAFFIC_LINE)
  -- Generate new traffic line section object going from origin '
  a_origin' to place named 'a_destination'
  -- belonging to line 'a_line' in map 'a_map'.
  -- (Access the generated object through feature 'line_section
  ')
require
  a_map_exists: a_map /= Void
  a_origin_exists: a_map.has_place (a_origin)
  a_destination_exists: a_map.has_place (a_destination)
  a_line_exists: a_line /= Void
ensure
  
```

```
line_section_created: line_section /= Void
line_section_has_line: line_section.line = a_line
line_section_has_type: equal (line_section.type, a_line.type)
line_section_has_origin: line_section.origin = a_map.place (
  a_origin)
line_section_has_destination: line_section.destination =
  a_map.place (a_destination)
line_section_in_map: map.has_line_section (a_origin,
  a_destination, a_line.type, a_line)
```

```
line_section: TRAFFIC_LINE_SECTION
  -- Generated traffic line section object.
require
  line_section_available: has_line_section
ensure
  Result_exists: Result /= Void

has_line_section: BOOLEAN
  -- Is there a line section object available?
```

The building of a line works as defined as standard.

feature -- Traffic line building

```
build_line (a_name: STRING; a_type_name: STRING; a_map:
  TRAFFIC_MAP)
  -- Generate new traffic line object with name 'name' and
  -- type 'type'
  -- belonging to map 'a_map'.
  -- (Access the generated object through feature 'line')
require
  a_name_exists: a_name /= Void
  a_name_not_empty: not a_name.is_empty
  a_map_exists: a_map /= Void
  type_name_is_valid: valid_name (a_type_name)
ensure
  line_created: line /= Void
  line_has_name: equal (line.name, a_name)
  line_has_type: equal (line.type.name, a_type_name)
  line_in_map: map.has_line (a_name)

line: TRAFFIC_LINE
  -- Generated traffic line object.
require
  line_available: has_line
ensure
  Result_exists: Result /= Void

has_line: BOOLEAN
  -- Is there a traffic line object available?
```

The simple line, too, is a standard process.

feature *-- Traffic simple line building*

```
build_simple_line (a_name: STRING; a_type_name: STRING;
  a_map: TRAFFIC_MAP)
  -- Generate new traffic simple line object with name 'name'
  and type 'type'
  -- belonging to map 'a_map'.
  -- (Access the generated object through feature 'simple_line
  ')
require
  a_name_exists: a_name /= Void
  a_name_not_empty: not a_name.is_empty
  a_map_exists: a_map /= Void
  type_name_is_valid: valid_name (a_type_name)
ensure
  simple_line_created: simple_line /= Void
  simple_line_has_name: equal (simple_line.name, a_name)
  simple_line_has_type: equal (simple_line.type.name,
    a_type_name)
  simple_line_in_map: map.has_line (a_name)

simple_line: TRAFFIC_LINE
  -- Generated traffic simple line object.
require
  simple_line_available: has_line
ensure
  Result_exists: Result /= Void

has_simple_line: BOOLEAN
  -- Is there a traffic simple line object available?
```

Example

```
TRAFFIC_MAP_FACTORY example:
created map: Traffic map
named: simple vip map
description:

places:
Traffic place rail station at position (x = 0, y = 0)
Traffic place sweet home at position (x = 5, y = 5)

lines:
Traffic bus line: vip bus shuffle,
  one direction: rail station, sweet home
  other direction: sweet home, rail station
```

Figure 53: generated output

```
run is
  -- Run example demonstration.
local
```

```
map_factory: TRAFFIC_MAP_FACTORY
station, home: TRAFFIC_PLACE
type: TRAFFIC_TYPE_BUS
do
  -- create map_factory
  create map_factory.make

  -- create type
  map_factory.build_traffic_type ("bus")
  type ?= map_factory.traffic_type

  -- create map
  map_factory.build_map ("simple vip map")

  -- create station and home place
  map_factory.build_place ("rail station", map_factory.map)
  station := map_factory.place
  map_factory.build_place_with_position ("sweet home", 5, 5,
    map_factory.map)
  home := map_factory.place

  -- create line
  map_factory.build_line("vip bus shuffle", type.name,
    map_factory.map)

  -- create line sections
  map_factory.build_line_section (station.name, home.name, Void
    , map_factory.map, map_factory.line)
  map_factory.build_line_section (home.name, station.name, Void
    , map_factory.map, map_factory.line)

  -- display created map
  text := "created map: " + map_factory.map.out
end
```

8.1.9 TRAFFIC_PLACE

Requirements

TRAFFIC_PLACE_INFORMATION (8.1.10)

Description

The class TRAFFIC_PLACE represents a place in a city. It can have additional information like one or more pictures and a description attached to it. Additionally it can have a position.

Class overview

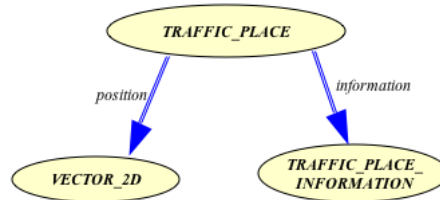


Figure 54: TRAFFIC_PLACE

Class interface

There exist two different ways to create a new place. The first way is to only define a name of the place, the position will be set to default. The second one is to also give its position to the creation feature.

create

```
make (a_name: STRING)
    -- Initialize 'Current'.
require
    a_name_exists: a_name /= Void
    a_name_not_empty: not a_name.is_empty
ensure
    name_set: equal (a_name, name)
    position_exists: position /= Void

make_with_position (a_name: STRING; a_x, a_y: INTEGER)
    -- Initialize 'Current' with name 'a_name' and position '
    a_x', 'a_y'.
require
    a_name_exists: a_name /= Void
    a_name_not_empty: not a_name.is_empty
    a_x_exists: a_x /= Void
    a_y_exists: a_y /= Void
ensure
    name_set: equal (a_name, name)
    position_exists: position /= Void
    position_set: position.x = a_x and position.y = a_y
```

The name, position and information can be accessed. The information can be *Void*.

feature -- Access

```
name: STRING
    -- Name of place.
```

```
position: VECTOR_2D
  -- Position on map.

information: TRAFFIC_PLACE_INFORMATION
  -- Additional information.
```

The information and position can be changed during the lifetime of a place object.

feature -- Element change

```
set_information (a_information:
  TRAFFIC_PLACE_INFORMATION)
  -- Set information to 'a_information'.
require
  a_information_exists: a_information /= Void
ensure
  information_set: information = a_information

set_position (a_position: VECTOR_2D)
  -- Set position to 'a_position'.
require
  a_position_exists: a_position /= Void
ensure
  position_set: position = a_position
```

Example

```
run is
  -- Run example demonstration.
local
  home: TRAFFIC_PLACE
  home_info: TRAFFIC_PLACE_INFORMATION
  position: VECTOR_2D
do
  -- create home
  create home.make ("home")

  -- display home
  text := text + "initial place: " + home.out

  -- create some home information
  create home_info.make
  home_info.set_description ("home sweet home")

  -- add some description
  home.set_information (home_info)

  -- set my homes coordinates
```

```
create position.make (17, 8)
home.set_position (position)

-- display new home
text := text + "%Nchanged place: " + home.out
end
```

TRAFFIC_PLACE example:
initial place: Traffic place home at position (x = 0, y = 0)
changed place: Traffic place home at position (x = 17, y = 8), description: home sweet home

Figure 55: generated output

8.1.10 TRAFFIC_PLACE_INFORMATION

Requirements

LINKED_LIST

Description

The class TRAFFIC_PLACE_INFORMATION is a collection of additional information for a place. This can be one or more pictures (path to the pictures) and a textual description.

Class overview



Figure 56: TRAFFIC_PLACE_INFORMATION

Class interface

To create a new traffic place information call the feature make.

create

make

-- Create empty information.

ensure

pictures_exists: pictures /= Void

A picture is added with the feature *extend_picture* and a description is added with the command *set_description*.

feature -- Access

pictures: LINKED_LIST [STRING]

```
-- Path to picture.
```

```
description: STRING  
-- Description.
```

To delete a picture you have to know its path. Then the command *remove_picture* has to be used. To remove a description just call *remove_description*.

```
feature -- Element change
```

```
extend_picture (a_picture_path: STRING)  
  -- Set picture path to 'a_picture_path'.  
  require  
    a_picture_path_exists: a_picture_path /= Void  
  ensure  
    picture_set: pictures.has (a_picture_path)
```

```
set_description (a_description: STRING)  
  -- Set description to 'a_description'.  
  require  
    a_description_exists: a_description /= Void  
  ensure  
    description_set: description = a_description
```

```
feature -- Removal
```

```
remove_picture (a_picture_path: STRING)  
  -- Remove picture path from pictures.  
  require  
    picture_in_pictures: pictures.has (a_picture_path  
    )  
  ensure  
    picture_removed: not pictures.has (  
      a_picture_path)
```

```
remove_description  
  -- Remove description.  
  ensure  
    description_removed: description = Void
```

Example

```
run is  
  -- Run example demonstration.  
local  
  home_information: TRAFFIC_PLACE_INFORMATION  
do
```

```
-- create home information
create home_information.make

-- display home_information
text := "initial place_information: " + home_information.out

-- add a description
home_information.set_description ("my home sweet home")

-- add a picture
home_information.extend_picture ("my_home_path")

-- display new place_information
text := text + "%Nplace_information: " + home_information.
out
end
```

```
TRAFFIC_PLACE_INFORMATION example:
initial place_information:
place_information: pictures: my_home_path, description: my home sweet home
```

Figure 57: generated output

8.1.11 TRAFFIC_ROUTE

Requirements

TRAFFIC_PLACE (8.1.9), TRAFFIC_LINE_SECTION(8.1.4)

Description

The class TRAFFIC_ROUTE calculates the shortest path for a set of places you want to visit on a map. You can change the route by adding or removing places you want to visit.

Class overview



Figure 58: TRAFFIC_ROUTE

Class interface

To create a new route you have to define the map on which the route is going to take place. The places you want to visit you can either set later, or you can initialize the route with a set of places of interest.

create

```
make_empty (a_map: TRAFFIC_MAP)
  -- Create empty route on map 'a_map'.
require
  a_map_exists: a_map /= Void
ensure
  places_to_visit_exists: places_to_visit /= Void
  places_on_route_exists: places_on_route /= Void
  line_sections_exists: line_sections /= Void
  map_exists: map /= Void
  map_set: map = a_map

make (a_places_to_visit: LINKED_LIST [TRAFFIC_PLACE];
     a_map: TRAFFIC_MAP)
  -- Create shortest path route through all places in '
  a_places_to_visit'.
require
  a_places_to_visit_exists: a_places_to_visit /= Void
  a_places_to_visit_more_than_one_place:
    a_places_to_visit.count > 1
  a_map_exists: a_map /= Void
  places_exist_on_map: places_on_map (a_places_to_visit,
    a_map)
ensure
  places_to_visit_exists: places_to_visit /= Void
  places_on_route_exists: places_on_route /= Void
  line_sections_exists: line_sections /= Void
  map_exists: map /= Void
  map_set: map = a_map
```

You have access to the places you entered. Additionally after calculating a route through the feature *calculate_shortest_path* the places and used line sections that are on the route can be accessed.

feature -- Access

```
places_to_visit: LINKED_LIST [TRAFFIC_PLACE]
  -- Places to visit on route.

places_on_route: LINKED_LIST [TRAFFIC_PLACE]
  -- All Places on route of last call to '
  calculate_shortest_path'.

line_sections: LINKED_LIST [TRAFFIC_LINE_SECTION]
```

```
-- Line sections to be used to visit 'places_to_visit'  
-- of last call to 'calculate_shortest_path'.
```

Places of interest can be added and removed.

feature -- Element change

```
extend (a_place: TRAFFIC_PLACE)  
  -- Add 'a_place' to the list of places to visit.  
  -- Call 'calculate_shortest_path' to recalculate the shortest  
  path  
  -- of all places to visit.  
require  
  a_place_exists: a_place /= Void  
  a_place_in_map: place_on_map (a_place)  
ensure  
  a_place_in_places_to_visit: places_to_visit.has (a_place)
```

feature -- Removal

```
remove (a_place: TRAFFIC_PLACE)  
  -- Remove 'a_place' from list of places to visit.  
  -- Call 'calculate_shortest_path' to recalculate the shortest  
  path  
  -- of all places to visit.  
require  
  a_place_exists: a_place /= Void  
  a_place_in_places_to_visit: places_to_visit.has (a_place)  
ensure  
  a_place_not_in_places_to_visit: not places_to_visit.has (  
    a_place)
```

For a single place or for a list of places it can be tested, that they really are on the map. Only then they can be added to the places to visit and a route can be calculated. It is not possible to calculate a route with places that are situated on different maps.

feature -- Status report

```
place_on_map (a_place: TRAFFIC_PLACE): BOOLEAN  
  -- Is a_place on map?  
require  
  a_place_exists: a_place /= Void  
  
places_on_map (a_places: LINKED_LIST[TRAFFIC_PLACE];  
  a_map: TRAFFIC_MAP): BOOLEAN  
  -- Are all places on map?  
require  
  a_places_exists: a_places /= Void  
  a_map_exists: a_map /= Void
```

The feature you will be most interest in is *calculate_shortest_path* which calculates a route which visits all your places to visit.

feature *-- Basic operation.*

```
calculate_shortest_path  
-- Calculate the shortest path from one place to  
visit to the next.
```

Example

```
TRAFFIC_ROUTE example:  
route: Places eth, home  
last calculated route:eth, station, home
```

Figure 59: generated output

feature *-- Basic operation*

```
run is  
-- Run example demonstration.  
local  
map: TRAFFIC_MAP  
home, station, eth: TRAFFIC_PLACE  
station_home, eth_station, home_eth:  
TRAFFIC_LINE_SECTION  
route: TRAFFIC_ROUTE  
position: VECTOR_2D  
type: TRAFFIC_TYPE_WALKING  
do  
-- create map  
create map.make ("simple map")  
  
-- create places  
create home.make_with_position ("home", 10, 10)  
create station.make_with_position ("station", 0, 1)  
create eth.make_with_position ("eth", 1, 0)  
  
-- add places to map  
map.add_place (home)  
map.add_place (station)  
map.add_place (eth)  
  
-- create line sections  
create type.make  
create station_home.make (station, home, type)  
create eth_station.make (eth, station, type)
```

```
create home_eth.make (home, eth, type)

-- add line sections to map
map.add_line_section (station_home)
map.add_line_section (eth_station)
map.add_line_section (home_eth)

-- create route
create route.make_empty (map)
route.extend (eth)
route.extend (home)

-- calculate shortest way from eth home
route.calculate_shortest_path

-- display route
text := "%Nroute: " + route.out
end
```

8.1.12 TRAFFIC_SIMPLE_LINE

Requirements

TRAFFIC_LINE (8.1.3)

Description

TRAFFIC_SIMPLE_LINE is a line that always has a line section in both directions. So if you add a line section from place A to place B the line section from place B to place A will be added as well. As a result you get a symmetric line.

Class overview

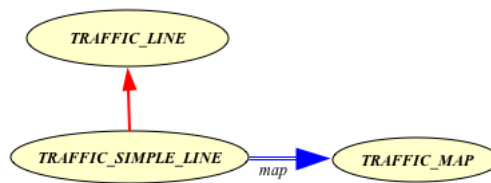


Figure 60: TRAFFIC_SIMPLE_LINE

Class interface

The only feature that differs from the traffic line interface is the creation feature. The simple line needs a traffic map in addition. This reference to the map has to be the map, that the line (and consequently its line sections) is in.

create

```
make (a_name: STRING; a_type: TRAFFIC_TYPE; a_map:
      TRAFFIC_MAP)
  -- Create a line with name 'a_name' of type 'a_type'.
require
  a_name_exists: a_name /= Void
  a_name_not_empty: not a_name.is_empty
  a_type_exists: a_type /= Void
  a_map_exists: a_map /= Void
ensure
  name_set: equal (name, a_name)
  type_set: type = a_type -- have to be same object
  count_line_section_not_void: count >= 0 -- List is
    initilalized.
  places_one_direction_exists: places_one_direction /= Void
  places_other_direction_exists: places_other_direction /=
    Void
  map_set: map = a_map
```

Example

```
TRAFFIC_SIMPLE_LINE example:
initial line: Traffic bus line: simple VIP bus,
  one direction: rail station, sweet home
  other direction: sweet home, rail station
changed line: Traffic bus line: simple VIP bus, RGB(0, 0, 255)
  one direction: rail station, sweet home
  other direction: sweet home, rail station
```

Figure 61: generated output

```
run is
  -- Run example demonstration.
local
  simple_line: TRAFFIC_SIMPLE_LINE
  blue: TRAFFIC_COLOR
  station_home: TRAFFIC_LINE_SECTION
  station, home: TRAFFIC_PLACE
  map: TRAFFIC_MAP
  type: TRAFFIC_TYPE_BUS
do
  -- create map
  create map.make ("simple map")

  -- create station and home place
  create station.make ("rail station")
  create home.make ("sweet home")

  -- add places to map
  map.add_place (station)
```

```
map.add_place (home)

-- create bus type
create type.make

-- create simple line
create simple_line.make ("simple VIP bus", type, map)

-- if station-home is a valid insertion, create line section
-- from station to home and add it to the line
if simple_line.is_valid_insertion (station, home) then
  create station_home.make (station, home, type)
  -- add line section
  simple_line.extend (station_home)
end

-- display line
text := text + "initial line: " + simple_line.out

-- create blue color
create blue.make (0, 0, 255)
-- set the line's color to blue
simple_line.set_color (blue)

-- display new line
text := text + "%Nchanged line: " + simple_line.out
end
```

8.1.13 TRAFFIC_TYPE

Requirements

None

Description

The class TRAFFIC_TYPE is used to identify the type of a line or line_section. TRAFFIC_TYPE is the abstract class of all possible traffic types. Traffic types are mainly used to make sure that only line sections of a given type can be added to a line of some type. This is done to ensure that no bus drives on a rail way and so on.

Class overview

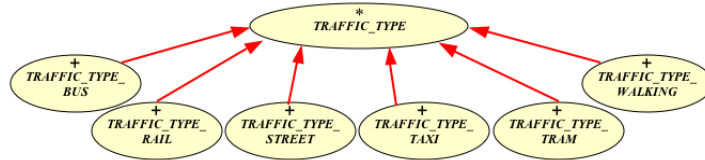


Figure 62: TRAFFIC_TYPE

The class overview reflects the simple construction of the type hierarchies.

Class interface

The traffic type classes have one interesting query. This query is called *name* and returns a textual representation of the traffic type.

```
feature -- Access

    name: STRING
        -- String representation.
```

8.1.14 TRAFFIC_TYPE_FACTORY

Requirements

TRAFFIC_TYPE (8.1.13)

Description

The class TRAFFIC_TYPE_FACTORY creates singleton traffic types. This means you get the same type object every time you build a type. This makes it easier to compare two objects of a given traffic type. You can simply compare the references of the type. If they are identical the type of the two objects those type references belong to are the same.

Class overview

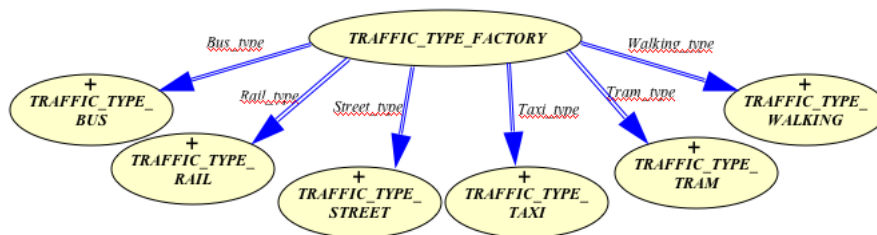


Figure 63: TRAFFIC_TYPE_FACTORY

Class interface

To create a type factory simply call the *make* feature.

create

```
make
  -- Set internal traffic type representation to void.
ensure
  internal_traffic_type_is_void: internal_traffic_type = Void
  type_table_exists: type_table /= Void
```

The feature *valid_name* tests, if a name is a valid name of a traffic type. For valid names, traffic types can be built. The query *has_type* tells you whether the last call to *build* was successful.

feature -- Status report

```
valid_name (a_name: STRING): BOOLEAN
  -- Is 'a_name' a valid traffic type name?
require
  a_name_exists: a_name /= Void
  a_name_not_empty: not a_name.is_empty

has_type: BOOLEAN
  -- Has a type object been generated?
```

The factory can be reset by the *reset* feature call. The most important feature about the factory is *build*. This generates a traffic type of the specified type. If you are not sure about the string, call *valid_name* to assure, that the type you want to build is a valid one. You can retrieve it with the call to *traffic_type*. The query *has_type* returns true if the type could be built.

feature -- Basic operation

```
reset
  -- Reset internal traffic type representation.
ensure
  internal_traffic_type_is_void: internal_traffic_type = Void

build (a_name: STRING)
  -- Build traffic type depending on type name 'a_name'.
  -- Access traffic type with 'traffic_type'.
require
  a_name_valid: valid_name (a_name)
  a_name_not_empty: not a_name.is_empty
ensure
  type_exists: has_type
```

The last created types of the factory can be accessed through the feature *traffic_type*. Remember to call it after the *build* which generates you a traffic

type of the specific type. Also make sure with the feature *has_type* that a valid object has been created, thus you will get back a useful reference for the access of *traffic_type*.

feature *-- Access*

```
traffic_type: TRAFFIC_TYPE
  -- Traffic type last created by 'build'.
```

Example

```
TRAFFIC_TYPE_FACTORY example:
  built type: tram
```

Figure 64: generated output

```
run is
  -- Run example demonstration.
local
  type_factory: TRAFFIC_TYPE_FACTORY
  type_name: STRING
  type: TRAFFIC_TYPE
do
  -- create type_factory
  create type_factory.make

  -- build type tram
  type_name := "tram"
  if type_factory.valid_name (type_name) then
    type_factory.build (type_name)
  end

  -- if successful build assign result to type variable
  if type_factory.has_type then
    type := type_factory.traffic_type
    -- display type
    text := "built type: " + type.out
  else
    -- no successful build
    -- display information
    text := "no type built."
  end
end
end
```

8.2 Developer Manual

In the following you will find descriptions, use and connection between classes that are important for the developer of the TRAFFIC library. Important design decisions of a class and their effect on the implementation of a class are mentioned and explained where necessary.

8.2.1 TRAFFIC_COLOR Class

The TRAFFIC_COLOR class is a very simple class.



Figure 65: TRAFFIC_COLOR

You as a developer have to make sure, that only valid values for the RGB parts are set. For this you have the feature *is_valid_rgb_part*. As you can see in the following code extracts, this feature is used as precondition for the *make* and *set_<some_rgb_part>* features and in the invariant of the class to assure that the class always has a valid state of color parts.

feature {NONE} -- Initialisation

```

make (a_red, a_green, a_blue: INTEGER) is
  -- Create a rgb-color.
  require
    a_red_valid: is_valid_rgb_part (a_red)
    a_green_valid: is_valid_rgb_part (a_green)
    a_blue_valid: is_valid_rgb_part (a_blue)
  do
    ...
  end

```

feature -- Element change

```

set_red (a_red: INTEGER) is
  -- Set red to 'a_red'.
  require
    a_red_valid: is_valid_rgb_part (a_red)
  do
    ...
  end

```

feature -- Status report

```

is_valid_rgb_part (a_part: INTEGER): BOOLEAN is
  -- Is 'a_part' a valid part of rgb-color?
  require
    a_part_exists: a_part /= Void
  do
    ...
  end

```

invariant

```

red_valid: red /= Void and then is_valid_rgb_part (red)

```

green_valid: *green* /= Void **and then** *is_valid_rgb_part* (*green*)
blue_valid: *blue* /= Void **and then** *is_valid_rgb_part* (*blue*)

8.2.2 TRAFFIC_LINE

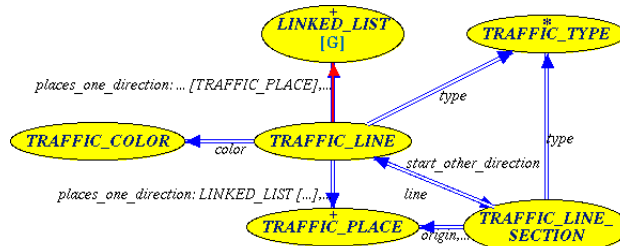


Figure 66: TRAFFIC_LINE

General

The TRAFFIC_LINE class is the class containing all line section elements that it is composed of. To be able to administrate them in an easy way they are put in a linked list. There are different possibilities to implement this list (see 4 and 6). The current implementation of the line is a linked list by inheriting from LINKED_LIST. Only those features of LINKED_LIST that really are needed of the user of the TRAFFIC_LINE are exported. All other features are hidden. This is done to make the interface for the students as small as possible. And as the linked list is only used for internal use of representation of a line and its line sections customers of the TRAFFIC_LINE class do not have to know, that a line is a linked list thus they don't rely on the interface provided by LINKED_LIST.

```

class
    TRAFFIC_LINE

inherit
    LINKED_LIST [TRAFFIC_LINE_SECTION]
    rename
        make as make_linked_list,
        out as linked_list_out,
        extend as put_end
    export
        {NONE} all
    end
    
```

Existence of one and other direction

For the implementation of *one_direction_exists* and *other_direction_exists* there exists different possibilities. As it is this feature tests whether *terminal_1* (or *terminal_2* respectively) exist. If so, there must be a direction in

this way. Therefore it is vital to ensure, that the terminals are always up to date. Another implementations can be to test the *places_one_direction* (*places_other_direction* respectively). The criterion (*places_one_direction*), here too, has to be kept up to date.

```
one_direction_exists: BOOLEAN is
  -- Does line have line section(s) in one direction?
do
  Result := terminal_1 /= Void
end
```

Implementation of one and other direction

The two directions, calls *one_direction* and *other_direction*, are implemented in the same linked list. By definition, *one_direction* is inserted at the beginning of the linked list followed by *other_direction*, if *other_direction* exists at all. At first always the *one_direction* is created. If *one_direction* exists, we know, that the first list element is the starting line section in this direction. The starting line section in the other direction is internally kept as reference called *start_other_direction*. With those two references we can go through the linked list from start until we come to *start_other_direction* or the end of the list (in this case no *other_direction* exists).

Testing if the line can be extended with a line section

The feature *is_valid_for_insertion* test for an existing line section, if it can be inserted to the line. There exist also the possibility to test for a hypothetical line section from an origin to a destination if it can be inserted into the line. Therefor the implementation for the test of an existing line section can be reduced to test if a line section from its origin to its destination can be inserted into the line. How this is done is explained after the code for the feature *is_valid_for_insertion*.

feature -- Status report

```
is_valid_for_insertion (a_line_section:
  TRAFFIC_LINE_SECTION): BOOLEAN is
  -- Can 'a_line_section' be added to line?
  --
  -- This is the case if it can be added in front or back of an
  -- existing direction,
  -- or at least one direction does not yet exists.
  -- The destination of 'a_line_section' is
  -- not place allready use in this direction (circle).
  -- A line section can not be added twice to the same line.
  -- A line section can not be added to two lines at the same
  -- time.
require
  a_line_section_exists: a_line_section /= Void
```

```

do
  if a_line_section.line /= Void or has (a_line_section) then
    -- 'a_line_section' is in other line or in this
    Result := False
  else
    Result := is_valid_insertion_one_direction (
      a_line_section.origin, a_line_section.destination) or
      is_valid_insertion_other_direction (a_line_section.
        origin, a_line_section.destination)
  end
end
end

```

Testing if the line can be extended with a line section from an origin place to a destination place

The features that provides this test is called *is_valid_insertion* with two places (following called *a_origin* and *a_destination* of a line section) as its arguments. The implementation tests if a line section from *a_origin* to *a_destination* can be added in front of or at the end of *one_direction* or in front of or at the end of *other_direction*. If either *one_direction* or *other_direction* do not exists, that means if their terminal are empty, any line section can be inserted. Otherwise we have to test the front and end of both directions. Following the explanation for *one_direction* (*other_direction* is analogical, just keep in mind, that the internal representation of *other_direction* starts somewhere in the middle and goes to the end of the internal linked list).

For *one_direction* to add a line section in front, *a_destination* has to be the same as the *origin* of the starting line section in *one_direction*. This first line section in *one_direction* is the first line section in the internal linked list (for *other_direction* the first line section is stored in the reference called *start_other_direction*). Additionally it has to be made sure, that the line has no circles. Therefore *a_origin* has not yet to be a place in *one_direction*. To test this, an internal list of all places in *one_direction* and all places in *other_direction* are stored. They are called *places_one_direction* (respectively *places_other_direction*). So if the first test has been successful we test *a_origin* against the *places_one_direction*. If the place is not in the list, the line section from *a_origin* to *a_destination* can be added. The test for the end of *one_direction* is similar. Just use the destination of *terminal_1* instead of the origin of the first line section.

feature -- Status report

```

is_valid_insertion (a_origin, a_destination: TRAFFIC_PLACE):
  BOOLEAN is
  -- Can a line_section from 'a_origin' to 'a_destination' be
  -- added
  -- in front or back of this line?
require
  a_origin_exists: a_origin /= Void
  a_destination_exists: a_destination /= Void

```

```
do
  Result := is_valid_insertion_one_direction (a_origin,
    a_destination) or
    is_valid_insertion_other_direction (a_origin,
    a_destination)
end

feature {NONE} -- Implementation

is_valid_insertion_one_direction (a_origin, a_destination:
  TRAFFIC_PLACE): BOOLEAN is
  -- Can line section from 'a_origin' to 'a_destination' be
  -- added in one direction?
  require
    a_origin_exists: a_origin /= Void
    a_destination_exists: a_destination /= Void
  do
    if terminal_1 = Void then -- no line sections added so far
      Result := True
    else
      Result := is_valid_insertion_one_direction_front (
        a_origin, a_destination) or
        is_valid_insertion_one_direction_end (a_origin,
        a_destination)
    end
  end
end

is_valid_insertion_one_direction_front (a_origin, a_destination:
  TRAFFIC_PLACE): BOOLEAN is
  -- Can line section from 'a_origin' to 'a_destination' be
  -- added in front of one direction?
  require
    a_origin_exists: a_origin /= Void
    a_destination_exists: a_destination /= Void
    terminal_1_exists: terminal_1 /= Void
  do
    Result := False
    if equal (a_destination, start_to_terminal (terminal_1))
      then
        if not places_one_direction.has (a_origin) then -- no
          circle
          Result := True
        end
      end
    end
  end
end

is_valid_insertion_one_direction_end (a_origin, a_destination:
  TRAFFIC_PLACE): BOOLEAN is
  -- Can line section from 'a_origin' to 'a_destination' be
  -- added at end of one direction?
```

```

require
  a_origin_exists: a_origin /= Void
  a_destination_exists: a_destination /= Void
  terminal_1_exists: terminal_1 /= Void
do
  Result := False
  if equal (a_origin, terminal_1) then
    if not places_one_direction.has (a_destination) then
      Result := True
    end
  end
end

```

Extending the line

When extending a line with a line section the line section has to be a valid insertion. Therefore the feature *is_valid_for_insertion* has to return true when called with the line section to be extended with.

To know where to put the new line section to, the tests for front and end of the directions have to be redone. So if the terminal do not exists, the line section is the first line section in the direction. Therefore the terminal and for other_direction *start_other_direction* have to be updated. Else depending on whether the line section is added in front of back of the direction the *terminal_1*, *terminal_2*, *start_other_direction* and the *places_one_direction*, *places_other_direction* have to be updated. And the line section has to be inserted at the correct position in the internal linked list. Finally the line has to insert itself as the line the line section that has been inserted belongs to.

If a line section was to be removed. Those changes have all to be made undone.

```

extend (a_line_section: TRAFFIC_LINE_SECTION) is
  -- Add 'a_line_section' at beginning or end of existing
  direction(s).
require else
  a_line_section_exists: a_line_section /= Void
  a_line_section_valid_for_insertion: is_valid_for_insertion (
    a_line_section)
local
  position: INTEGER
  origin, destination: TRAFFIC_PLACE
do
  origin := a_line_section.origin
  destination := a_line_section.destination
  if terminal_1 = Void then -- no direction exists yet
    put_front (a_line_section)
    terminal_1 := destination
    places_one_direction.extend (origin)
    places_one_direction.extend (destination)
  end

```



```
inherit
  HASHABLE
  redefine
    is_equal,
    out
  end

feature -- Basic operation

  hash_code: INTEGER is
    -- Hash code value
  local
    line_name: STRING
  do
    if line = Void then
      line_name := ""
    else
      line_name := line.name
    end
    Result := ([origin, destination, type.name, line_name]).
      hash_code
  end
```

Two line section are the same, if they go from the same place to the same destination, have the same type and the same line they belong to. Therefore the *is_equal* feature is redefined for a line section.

```
feature -- Comparasion

  is_equal (other: like Current): BOOLEAN is
    -- Is 'other' attached to an object considered
    -- equal to current object?
    -- (from ANY)
  do
    Result := equal (origin.name, other.origin.name) and
      equal (destination.name, other.destination.name) and
      equal (type, other.type) and
      equal (line, other.line)
  end
```

The features *set_line* and *remove_line* are only exported to the TRAFFIC_LINE to ensure, that a line is only added to a line section, if the insertion of the line section in this line was successful.

8.2.4 TRAFFIC_LINE_SECTION_STATE

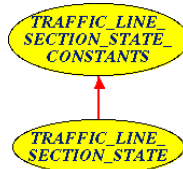


Figure 68: TRAFFIC_LINE_SECTION_STATE

General

The TRAFFIC_LINE_SECTION_STATE inherits from the class TRAFFIC_LINE_SECTION_STATE_CONSTANTS to have the features of the constants class available.

```
class
    TRAFFIC_LINE_SECTION_STATE

inherit
    TRAFFIC_LINE_SECTION_STATE_CONSTANTS
    rename
        out as state_out
    end
```

The TRAFFIC_LINE_SECTION_STATE is a class providing the interface for a state value. This value is stored in the reference called *value*. This state value can be changed through the *set_state* feature. The only thing you as a developer of the class have to make sure is, that the value always is a valid state value. This functionality is provided by the constants class.

8.2.5 TRAFFIC_LINE_SECTION_STATE_CONSTANTS

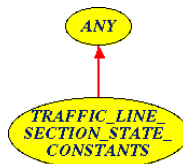


Figure 69: TRAFFIC_LINE_SECTION_STATE_CONSTANTS

General

The class administers all traffic line section state values. They are represented as an integer. Therefore the Eiffel construct unique can be used.

```
State_normal, State_collision: INTEGER is unique
-- State constants.
```

How to Insert a New State

If a new state is to be defined it has to be added in the state definitions (see code extract above). In addition the inspect statement in the *is_valid_state_value* and *value_to_string* have to be extended with this new identifiers implementation.

8.2.6 TRAFFIC_MAP

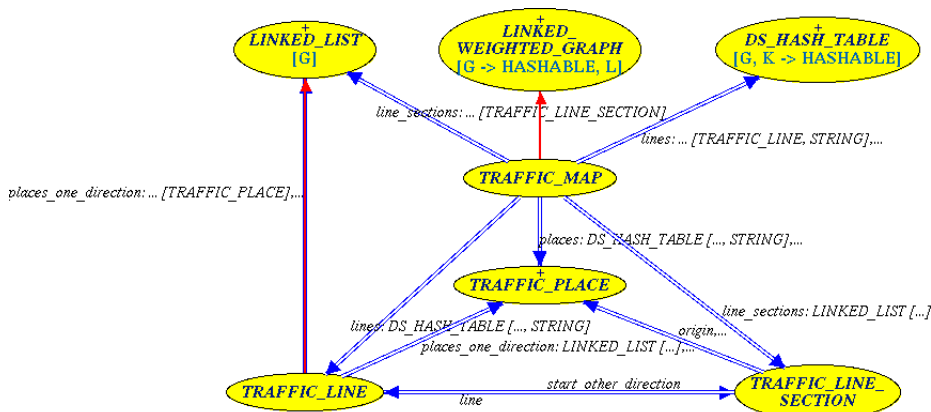


Figure 70: TRAFFIC_MAP

General

The TRAFFIC_MAP is a LINKED_WEIGHTED_GRAPH in order to represent the places and line sections as vertices and edges, respectively. Therefore you as a developer have to be very careful to make sure that the representation of the map and the graph are consistent. The map objects are stored in the map in corresponding containers (*internal_places*, *internal_lines*, *internal_line_sections*).

```

class
    TRAFFIC_MAP

inherit
    LINKED_WEIGHTED_GRAPH [TRAFFIC_PLACE,
        TRAFFIC_LINE_SECTION]
    rename
        item as graph_item
    export
        {NONE} all
        {ANY} find_shortest_path, shortest_path, path_found
    redefine
        out
    end
    
```

For each type of object a *has*-feature is implemented. The implementation then just looks in the corresponding internal container if an object identical with the one asked exists.

Adding an Object to the Map

When a new object is to be added to the map it first has to be made sure, that this object does not yet exist. An object can only be added once to the same map. If this is a valid extension, then the internal representation of the corresponding type has to be extended with the object. If the object is a line section or a place the graph has to be extended with the edge or the vertex (see the following code example for the addition of a place).

```

add_place (a_place: TRAFFIC_PLACE) is
    -- Add place 'a_place' to map.
    require
        a_place_exists: a_place /= Void
        no_place_with_same_name_in_map: not has_place (
            a_place.name)
    do
        internal_places.force (a_place, a_place.name)
        put_node (a_place)
        ...
    ensure
        a_place_in_map: has_place (a_place.name)
    end
    
```

8.2.7 TRAFFIC_MAP_FACTORY

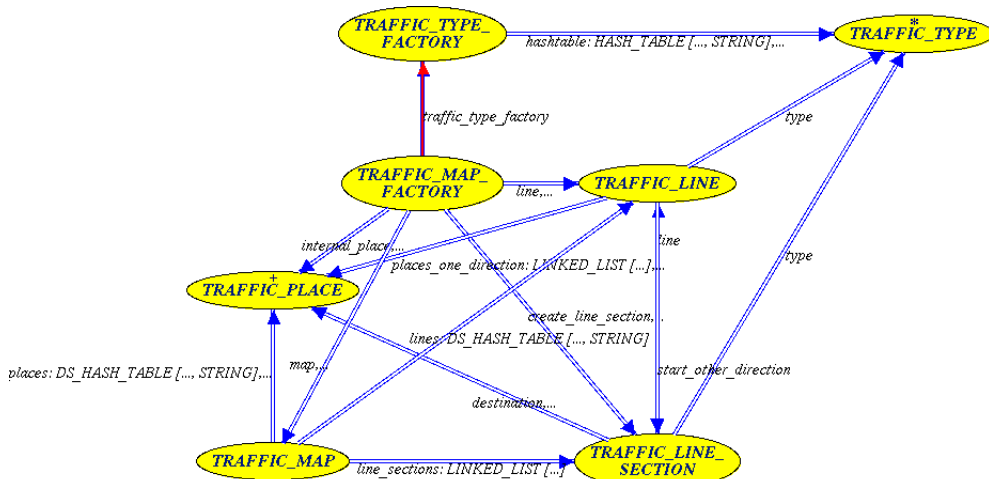


Figure 71: TRAFFIC_MAP_FACTORY

General

The TRAFFIC_MAP_FACTORY builds, as its name is indicating, a map and its objects. To be able to build the types of the objects as well, the TRAFFIC_MAP_FACTORY inherits from TRAFFIC_TYPE_FACTORY. In this way it can use the creation of types implicitly.

```

class
    TRAFFIC_MAP_FACTORY

inherit
    TRAFFIC_TYPE_FACTORY
    rename
        make as make_type_factory,
        build as build_traffic_type
    end
    
```

The general algorithm is to call a build command that created a new object and, if successful, adds it to the map, then a query that tells if the build was successful or not and then a query that returns the created object. For the created objects the factory has the internal references *internal_map*, *internal_place*, *internal_line_section*, *internal_line*.

Creation of an Object

The creation of the line, line section and simple line are implemented the implementation part of the class. For the line section the creation means to create a line section object, to extend it with polypoints, and to extend a line with the newly created line section.

How to Inset a new Class in the Map Factory

For a new type of class in the map the three types of features mentioned in the general section have to be implemented. An internal representation for the last built object of the new type, a build command, a has query and a get query. Remember to add a successfully created new object to the map.

8.2.8 TRAFFIC_PLACE

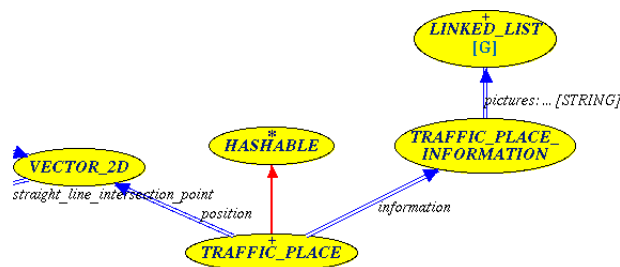


Figure 72: TRAFFIC_PLACE

the reference called *map*. When a user calls *calculate_shortest_path*, the route through all places is calculated, thereby using the shortest path from one place to another. The line sections used to come from one place to another are stored in *line_sections*. The places from the line sections in *places_on_route*.

Recalculating the Route

An essential design decision is when the route has to be recalculated. It is implemented that the customer of a class has to call *calculate_shortest_path* in order to get the route through all *places_to_visit*. More on this design decision can be found in 4.10.

Extending the Places to Visit

When extending the *places_to_visit*, you, as a developer, have to make sure that the place to be added is a place of the map. Only then a route can be calculated correctly. To ensure this, the features *place_on_map* and *places_on_map* are implemented.

8.2.11 TRAFFIC_SIMPLE_LINE

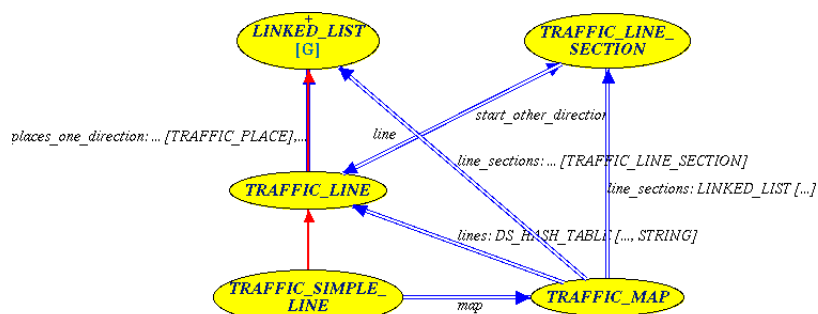


Figure 75: TRAFFIC_SIMPLE_LINE

General

The TRAFFIC_SIMPLE_LINE is a specialized line. It goes through the same places in both direction. This implicates that the terminals are the start places in the other direction.

```

class
    TRAFFIC_SIMPLE_LINE

inherit
    TRAFFIC_LINE
    rename
        make as make_line
    redefine
    
```

```
    start_to_terminal,  
    extend  
end
```

Creation

The reason why the simple line needs a reference to the map it belongs to will be seen when talking about extending the simple line with a line section. Until now, all map objects can easily have been in several maps, because they do not know in which map they are. Therefore a place can be in different maps, where different views of a city can be modeled. This is also true for lines. Therefore a way has to be thought of how to insert the simple line in several maps as well. At the time being, this is not possible.

Start to Terminal

For a simple line the *start_to_terminal* is always the other terminal. Therefore this feature has been reimplemented in that way.

Extend

The extension of a simple line is a little bit more complicated. In order to preserve the property of always going through the same places in both directions, extending means extending the line in both directions. Therefore the simple line has to create a new line section in the opposite direction and to add this in itself as well as the map. Thus the simple line needs a reference to the map it is in. The *is_valid_for_insertion* and *is_valid_insertion* features do not have to be reimplemented. When the consistency of the simple line is given, a valid insertion in one direction is always also a valid insertion in the other direction with the inverse line section.

```
extend (a_line_section: TRAFFIC_LINE_SECTION) is  
  -- Add 'a_line_section' at beginning or end of existing  
  direction(s).  
local  
  origin, destination: TRAFFIC_PLACE  
  other_line_section: TRAFFIC_LINE_SECTION  
do  
  origin := a_line_section.origin  
  destination := a_line_section.destination  
  create other_line_section.make (a_line_section.destination,  
    a_line_section.origin, a_line_section.type)  
  map.add_line_section (other_line_section)  
  
if terminal_1 = Void then -- no direction exists yet  
  put_front (a_line_section)  
  terminal_1 := destination  
  places_one_direction.extend (origin)
```

```

places_one_direction.extend (destination)

put_end (other_line_section)
start_other_direction := other_line_section
terminal_2 := origin
places_other_direction.extend (destination)
places_other_direction.extend (origin)
else
if is_valid_insertion_one_direction_end (origin,
destination) then
insert_one_direction_end (a_line_section)
insert_other_direction_front (other_line_section)
else
if is_valid_insertion_one_direction_front (origin,
destination) then -- put front of list
insert_one_direction_front (a_line_section)
insert_other_direction_end (other_line_section)
else
if is_valid_insertion_other_direction_end (origin,
destination) then -- put end of list
insert_other_direction_end (a_line_section)
insert_one_direction_front (other_line_section)
else -- is_valid_insertion_other_direction_front
insert_other_direction_front (a_line_section)
insert_one_direction_end (other_line_section)
end
end
end
end
a_line_section.set_line (Current)
other_line_section.set_line (Current)
end
    
```

8.2.12 TRAFFIC_TYPE

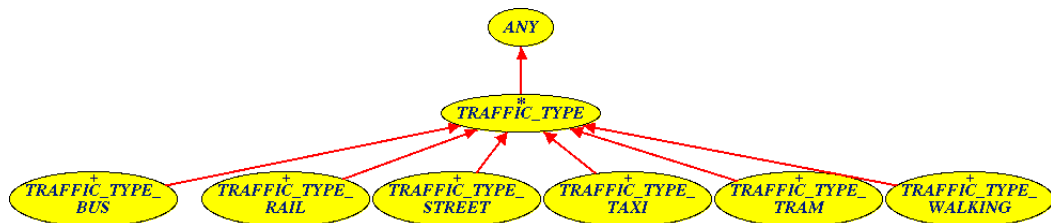


Figure 76: TRAFFIC_TYPE

General

The classes inheriting from TRAFFIC_TYPE define the possible types in the TRAFFIC library. The only feature they have to implement is a name. This

name should be some textual description of the type.

feature *-- Access*

name: STRING
-- String representation.

invariant

name_not_empty: not name.is_empty

How to Insert a new Traffic Type

To inset a new traffic type this new type has to inherit from TRAFFIC_TYPE and implement the *name* feature. Additionally the TRAFFIC_TYPE_FACTORY has to be extended to be able to create this new type, too.

A nice method to implement the name is to define it in the creation feature, as it is done in the TRAFFIC_TYPE_WALKING.

```
make is
  -- Create new walking type.
do
  name := "walking"
end
```

8.2.13 TRAFFIC_TYPE_FACTORY

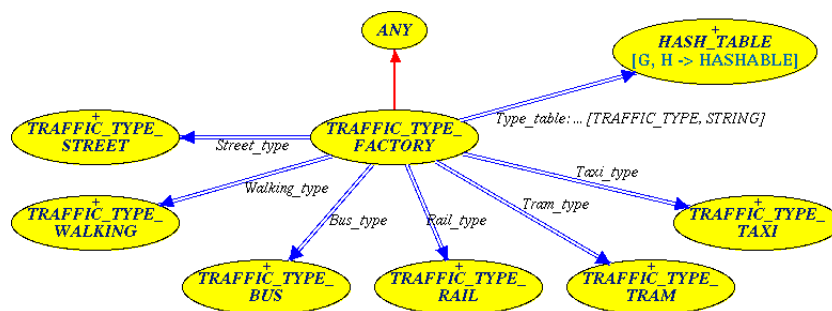


Figure 77: TRAFFIC_TYPE_FACTORY

General

The TRAFFIC_TYPE_FACTORY creates, as its name suggests, traffic types. The type objects are all once in the class. To make the accessing and the *valid_name* query easier to read and implement, the type objects are stored in a hash table. This hash table is initialized on creation of teh type factory.

```
make is
  -- Set internal traffic type representation to void.
```

```
do
  internal_traffic_type := Void
ensure
  internal_traffic_type_is_void: internal_traffic_type = Void
  type_table_exists: type_table /= Void
end
```

The building of types is similar to building objects for the map in the TRAFFIC_MAP_FACTORY. A customer of the type factory calls *build* with the name of the type, then he checks with the *has_type* if the build was successful and if so he fetches the created type by calling *traffic_type*.

The implementation of *build* uses the hash table. It accesses the type object through its name. Simple as that.

How to Insert a new Traffic Type

Add a new internal representant of the new type with a once feature and extend the type table with it. You can see how this is done for the walking type.

```
walking_type: TRAFFIC_TYPE_WALKING is
  -- Walking traffic type.
  once
    create Result.make
  end

type_table: HASH_TABLE [TRAFFIC_TYPE, STRING] is
  -- Table with all types.
  once
    ...
    Result.extend (walking_type, walking_type.name)
  end
```

8.2.14 XML-File

Map

The city has also in the XML-file been renamed to map. A map now has a description instead of the background and lines instead of links.

File, Color, Point

The *file*, *color* and *point* are still the same.

Description

Description is a new type consisting of a text.

Place

A place can now have zero to N file paths denoting paths to images of the place. Additionally it can have at most one description.

Line

A line can have at most one color assigned to it. Additionally it consists of an arbitrary number of line sections. A line also has a *name* and a *type* argument.

Line section

What was called link in the initial state model has also become line section in the XML-file. The new line section has no type as this is given implicitly in the line a line section is defined. It also has to have at least two points defined to indicate the coordinates from where to where the link is to be going.

Example

Following a short example of how the XML-file will look like.

```
<?xml version="1.0"?>
<!DOCTYPE city SYSTEM "example map.dtd">
<map name="Example City">

  <description text="An example city map."/>

  <places>

    <place name="Bahnhof Stadelhofen" x="" y="">
      <file name="some_file_name"/>
      <file name="some_other_file_name"/>
      <description text="one of many stations in zurich"/>
    </place>
    <place name="Kunsthaus"/>
    <place name="Rennweg"/>
    <place name="Rathaus"/>

  </places>

  <lines>

    <line name="11" type="tram">
      <color red="0" green="154" blue="0"/>
      <line_section from="Bahnhof Stadelhofen" to="Kunsthaus"
        direction="undirected">
        <point x="910" y="495"/>
        <point x="843" y="495"/>
      </line_section>
    </line>
  </lines>
</map>
```

```
</line_section>
<line_section from="Kunsthhaus" to="Rennweg" direction="
  undirected">
  <point x="843" y="495"/>
  <point x="750" y="495"/>
</line_section>
<line_section from="Rennweg" to="Rathaus" direction="
  undirected">
  <point x="705" y="655"/>
  <point x="705" y="835"/>
</line_section>
</line>

</lines>

</map>
```

References

- [1] Karine Arnout. *From Patterns to Components, A.5*. ETH Zurich, 2004.
<http://se.inf.ethz.ch/people/arnout/patterns>.
- [2] Rolf Bruderer. *Rolf Master Thesis*. ETH Zurich, 2005.
http://se.inf.ethz.ch/projects/rolf_bruderer.
- [3] Marcel Kessler. *DO IT WITH STYLE - A Guide to the Eiffel Style*, 2004.
<http://archive.eiffel.com/doc/manuals/language/style/style.pdf>.
- [4] Bertrand Meyer. *Touch of Class: Learning to Program Well - With Object Technology, Design by Contract, and Steps to Software Engineering*. To be published.
<http://se.inf.ethz.ch/touch>.
- [5] Bertrand Meyer. *The Outside-In Method of Teaching Introductory Programming*, 2003.
<http://www.inf.ethz.ch/~meyer/publications/teaching/teaching-psi.pdf>.
- [6] Michela Pedroni. *Teaching Introductory Programming with the Inverted Curriculum Approach*. ETH Zurich, 2003.
http://se.inf.ethz.ch/projects/michela_pedroni.