Redesign of Touch

Roger Küng

September 29, 2005

Abstract

The Touch application is the software used to show and implement the examples and exercises from the textbook “A Touch of Class” which will be used to teach programming to students. Since “A Touch of Class” is aimed at novice programmers the visual representation and the Source Code of Touch should be simple, entertaining and encouraging. The goal is to reimplement Touch using EiffelMedia and provide a good framework for examples together with some examples already implemented.

Contents

1 Introduction

Since Winter 2003 the Introduction to Programming Course at the department of Computer Science applies a teaching approach called Inverted Curriculum. In this approach the student is given a full Library to start with. In this case this is Traffic, a library that includes classes to model a city and it's transport connections. Accompanying are two applications given: Flat_Hunt and Touch. Flat_Hunt is a small game where players try to catch the hunted on a city map. Touch is used as a framework for the examples and exercises in the textbook "A Touch of Class" written by Bertrand Meyer, which will be used to guide the students through the course. During the winter semester 2004/2005 Sibylle Aregger [link] improved the Traffic library and Rolf Bruderer [link] extended ESDL to offer features that are needed for using it as visualization library for Touch instead of EiffelVision2. During my time working ESDL changed and was included in a much bigger library called EiffelMedia (EM), which will once finished include classes for sound, music, graphics, video, networking, collision detection, gui etc. This semester project aspired the redesign of Touch using EM. This included providing a framework where the examples and programming exercises from the book could be implemented. One of the key aspects of this work was to reimplement the examples from the textbook and to change the examples where necessary. The redesign of Flat_Hunt is done in parallel to my work by Ursina Caluori [Link]. Much of the work with the graphical representation was done in cooperation with Ursina.

2 Existing Frameworks

2.1 TOUCH

The existing Touch Application was written by Michela Pedroni [link] and is based on the old version of the traffic library. It uses Eiffelvision2 to visualize the application and the traffic map. The old Touch Application is also based on the textbook "A Touch of Class" and primarily shows its examples. So the goal of Touch was to implement the examples as close as possible the way they are written in the textbook. But the implementation could not really bring the same code to the examples as in the book. New classes had to be introduced to achieve the claimed functionality.

2.2 TRAFFIC

A major redesign of the traffic library has been done by Sibylle Aregger [link]. The new traffic library is slim and much easier to use than the old library was. The new city basically consists of objects which are instances of three classes: TRAFFIC_PLACE, TRAFFIC_LINE_SECTION and TRAFFIC_LINE. The TRAFFIC_MAP maintains a network which uses the places as nodes and the line sections as edges. Additionaly every line is a list of its line sections. A place has no information about the city at all. The Line does only know its line sections and the places it visits. Because the TRAFFIC_LINE class ist not so easy to handle there is a second class TRAFFIC_SIMPLE_LINE which inherits from TRAFFIC_LINE. Objects of this class can only have undirected line sections meaning if there is a line section in one direction there also must exist one in the other direction.

2.3 TRAFFIC VISUALISATION

In his Master Thesis Rolf Bruderer [Link] extended the ESDL Library and provided a first design and implementation of classes to help visualising the map in ESDL. His idea was to have a framework for any type of maps. The widget which displays all the items has access to the items in the map and their ordering in the map model is equivalent to the z ordering of the visualisation of the items. Every item has a category which can be queried from the map model. The renderer uses special render functions that convert the items into drawables for every item category. The so created drawables are stored, therefore the conversion only needs to be done if an item is changed.

3 Traffic Visualization

There were basically two parts of the existent design and implementation of the visualisation that needed further refinement. One was the implementation of the categorization and the other was the implementation of the item renderers.
The category of an item was a string. First of all a string is not very efficient in memory usage and time used to do an equal operation. The implementation which associated the item with its category was using the *generating_type* feature. This is only acceptable if we know all possible descendants of our base classes and this we cannot know.
In the existent implementation of the renderer an item was picked, its category was queried and then the correct render feature was called and the resulting drawable was stored. If you wanted to highlight a line by drawing it thicker the only thing you could do is change the category it belonged to. But since the category was its generating_type this was not easiliy possible. The possibility to add a feature *line_width* to the line is also not feasible because the different objects in a map should be independent of any visualisation and several different visualisations of the same map should be possible.
In a few meetings with Ursina and Michela we were able to develop a new model. To overcome the restrictions of the existent model we decided to drop the flexibility of putting objects of any type into the map model. We also replaced the rendering features by objects that provide a renderering feature. Special item renderer objects can be plugged in for evey single item. If no special item renderer object is plugged in, the item is rendered with the default item renderer object.

3.1 TRAFFIC_MAP_WIDGET

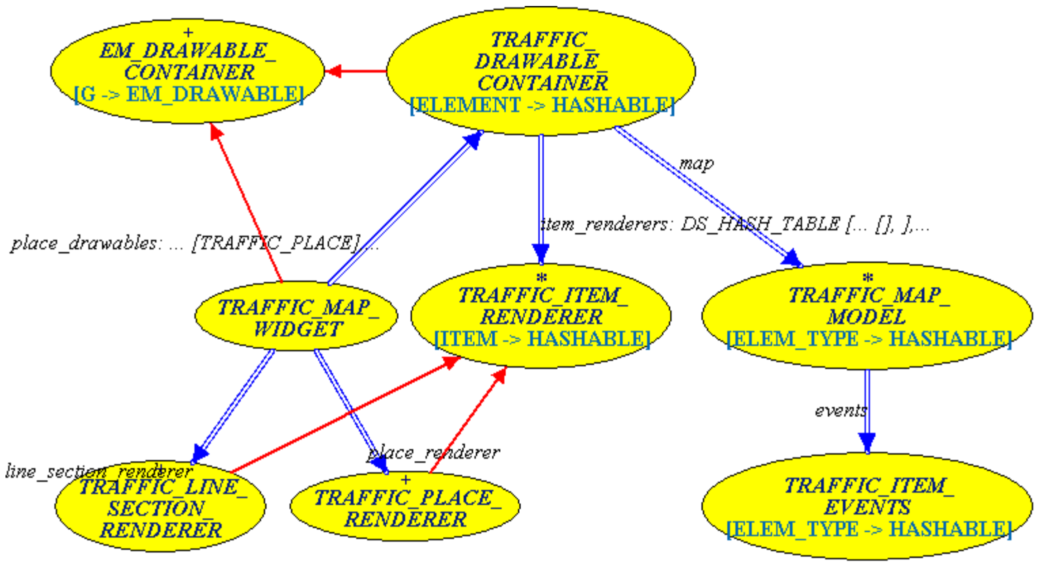

+ EM_DRAWABLE_ CONTAINER [G -> EM_DRAWABLE]
TRAFFIC_ DRAWABLE_ CONTAINER [ELEMENT -> HASHABLE]
map
item_renderers: DS_HASH_TABLE [... [],],...
place_drawables: ... [TRAFFIC_PLACE] ...
TRAFFIC_MAP_ WIDGET
* TRAFFIC_ITEM_ RENDERER [ITEM -> HASHABLE]
* TRAFFIC_MAP_ MODEL [ELEM_TYPE -> HASHABLE]
events
line_section_renderer
TRAFFIC_LINE_ SECTION_ RENDERER
place_renderer
+ TRAFFIC_PLACE_ RENDERER
TRAFFIC_ITEM_ EVENTS [ELEM_TYPE -> HASHABLE]

Figure 1: TRAFFIC_MAP_WIDGET

The TRAFFIC_MAP_WIDGET is what the user is supposed to use if he wants to display a map. It contains the two default item renderer objects, which are instances of specialized descendants of TRAFFIC_ITEM_RENDERER. One item renderer object for the line sections and the other for the places. It also contains two TRAFFIC_DRAWABLE_CONTAINERs, one for the places and one for the line sections. This class provides the following basic operations for the places.

place_renderer access the default place renderer

set_default_place_renderer set the default place renderer

set_place_special_renderer add an individual place renderer for a given place

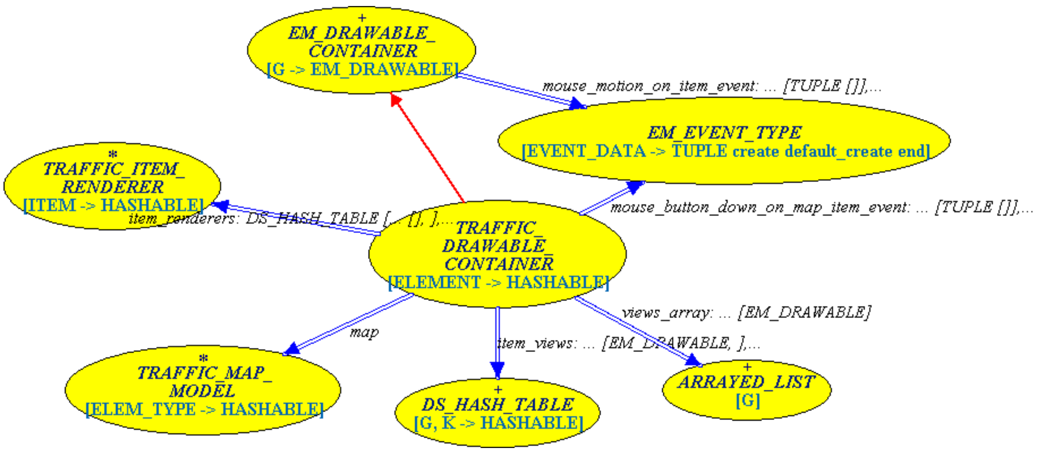
reset_place_special_renderer reset to the default place renderer for a given place

The same functionality exists for the line sections. But there are two operations more for convenience:

set_line_special_renderer add an individual line section renderer for all line sections of a given line

reset_line_special_renderer reset all line section renderers for all line sections of a given line

3.2 TRAFFIC_DRAWABLE_CONTAINER

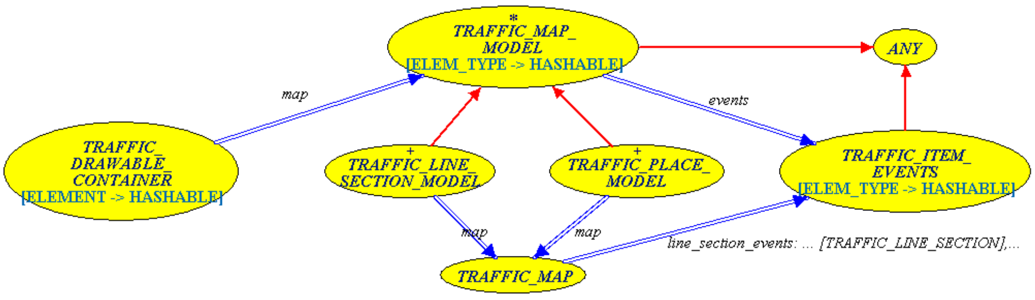


+ EM_DRAWABLE_ CONTAINER [G -> EM_DRAWABLE]
mouse_motion_on_item_event: ... [TUPLE []],...
EM_EVENT_TYPE [EVENT_DATA -> TUPLE create default_create end]
* TRAFFIC_ITEM_ RENDERER [ITEM -> HASHABLE]
item_renderers: DS_HASH_TABLE [... [],],...
TRAFFIC_ DRAWABLE_ CONTAINER [ELEMENT -> HASHABLE]
mouse_button_down_on_map_item_event: ... [TUPLE []],...
views_array: ... [EM_DRAWABLE]
map
item_views: ... [EM_DRAWABLE,],...
* TRAFFIC_MAP_ MODEL [ELEM_TYPE -> HASHABLE]
+ DS_HASH_TABLE [G, K -> HASHABLE]
+ ARRAYED_LIST [G]

Figure 2: TRAFFIC_MAP_CONTAINER

The TRAFFIC_DRAWABLE_CONTAINER converts all items provided to it and stores them. It only operates on items of one type. It does not get the items directly from a TRAFFIC_MAP but from a TRAFFIC_MAP_MODEL. To convert the items to drawables it has one default TRAFFIC_-ITEM_RENDERER and a hash table for the individual renderers of the items. The results of the TRAFFIC_ITEM_RENDERERs feature *render* are put to the EM_DRAWABLE_CONTAINER and are also stored in two separate structures. One is an array filled with the drawables and is used to track the mouse events back to the original item. The other structure is a hash map for finding the corresponding drawable to a given item. This can be used from outside the class to query the drawable corresponding to an item and it is used to find the old drawable and to remove it from the EM_DRAWABLE_CONTAINER if the item or its renderer has changed.

3.3 TRAFFIC_MAP_MODEL



* TRAFFIC_MAP_ MODEL [ELEM_TYPE -> HASHABLE]
ANY
map
events
TRAFFIC_ DRAWABLE_ CONTAINER [ELEMENT -> HASHABLE]
+ TRAFFIC_LINE_ SECTION_MODEL
+ TRAFFIC_PLACE_ MODEL
TRAFFIC_ITEM_ EVENTS [ELEM_TYPE -> HASHABLE]
map
map
line_section_events: ... [TRAFFIC_LINE_SECTION],...
TRAFFIC_MAP

Figure 3: TRAFFIC_MAP_WODEL

This class serves as an adaptor between the TRAFFIC_DRAWABLE_CONTAINER and a class that contains items. The two descendants TRAFFIC_LINE_SECTION_MODEL and TRAFFIC_-PLACE_MODEL build the link to the places and line sections from the TRAFFIC_MAP. They also set the internal TRAFFIC_ITEM_EVENTS object to the corresponding events from the map.

3.4 TRAFFIC_ITEM_RENDERER

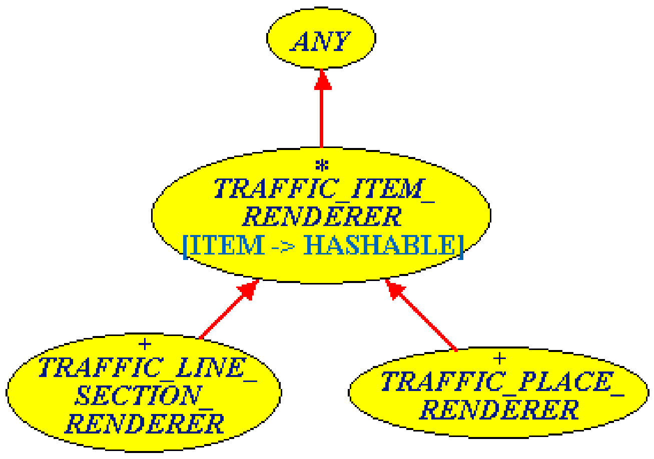


Figure 4: TRAFFIC_ITEM_RENDERER

The TRAFFIC_ITEM_RENDERER is the class that provides the rendering function. This function simply generates an EM_DRAWABLE from an ITEM. There are two descendant renderers for ITEMs of type TRAFFIC_PLACE and TRAFFIC_LINE_SECTION. The TRAFFIC_LINE_SECTION_RENDERER renders the TRAFFIC_LINE_SECTIONs as colored EM_POLYLINEs. The TRAFFIC_PLACE_RENDERER renders the TRAFFIC_PLACEs as EM_RECTANGLEs covering all the ends of the incident line sections.

4 Touch Redesign

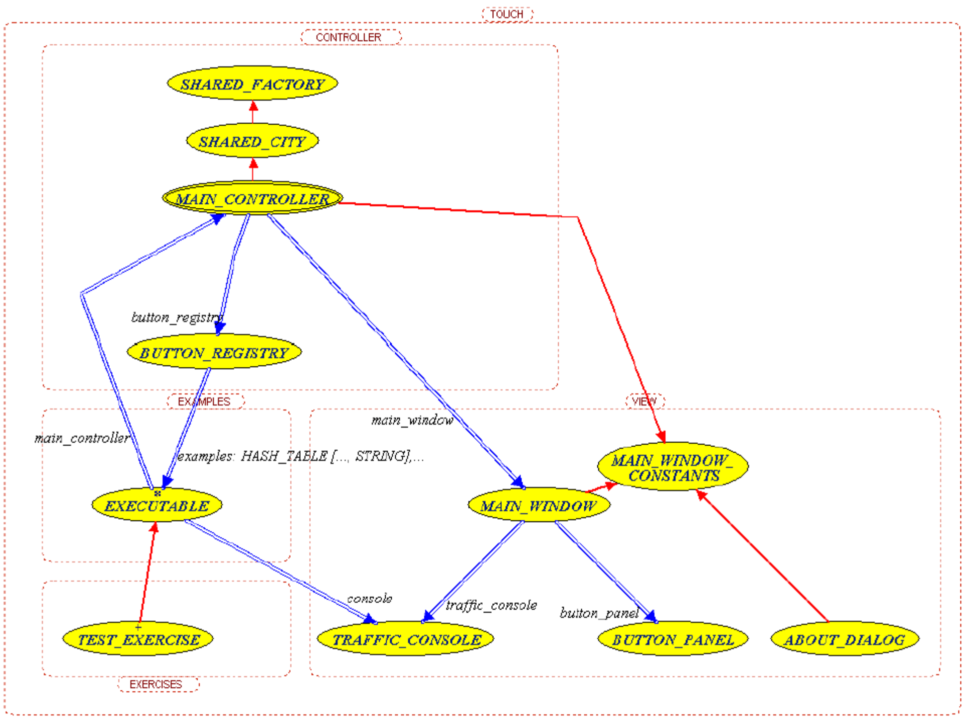


Figure 5: Old TOUCH Classes

Old Touch The old Touch Application is written to work with the old version of the Traffic library and it uses Eiffelvision2 for the visualisation part.

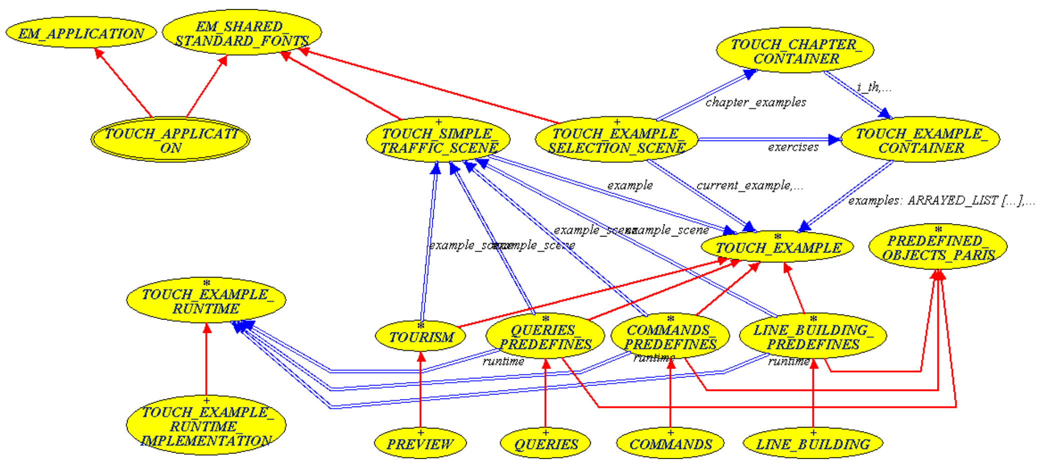


Figure 6: New TOUCH Classes

New Touch Since the new Touch Application is quite different from the old version I did not stick to the old design although it seemed very clean and simple. There are basically four blocks of classes. The TOUCH_APPLICATION which is the entry point for the application. The tho EM_SCENEs: TOUCH_SIMPLE_TRAFFIC_SCENE and TOUCH_EXAMPLE_SELECTION_SCENE. The TOUCH_EXAMPLEs including the TOUCH_EXAMPLE_RUNTIME. And the two example container in the top right corner.

4.1 TOUCH_EXAMPLE_SELECTION_SCENE

This is the main scene and it is also what the user sees first when he starts the Touch application. In the feature *initialize_scene* everything is set up. All the drawables are created and placed correctly. The two features which set up the examples and exercises are also called in this feature: *fill_chapter_examples* and *fill_exercises*. *fill_chapter_examples* creates and fills a TOUCH_CHAPTER_CONTAINER with the currently implemented examples from the Touch Textbook: PREVIEW, QUERIES, COMMANDS and LINE_BUILDING. *fill_exercises* creates and fills a TOUCH_EXAMPLE_CONTAINER. This is where the custom examples and exercises should be placed. The chapter and example containers are then put into a TOUCH_EXAMPLE_CONTAINER_WIDGET which visualizes the examples. The example container widget features an event we can subscribe to for being noticed when the user has selected an example. If this happens *example_selected* is called and information about the selected example is displayed.

4.2 TOUCH_CHAPTER_CONTANER

This class is very simple, since it is basically an array of TOUCH_EXAMPLE_CONTAINERs.

4.3 TOUCH_EXAMPLE_CONTAINER

This simple class just holds some TOUCH_EXAMPLEs.

4.4 TOUCH_EXAMPLE

TOUCH_EXAMPLE is the abstract base class for all examples. It has three features to include information about the example: The STRINGs *name* and *description* and *pictures* which is a list of EM_BITMAPs. *run* is the entry point for the execution of every TOUCH_EXAMPLE implementation. This is where the actual code for the example should be placed. This feature takes an object of type TOUCH_EXAMPLE_RUNTIME as an argument which then provides access to most of the important objects for the example. The feature *run_with_scene* which takes an EM_SCENE and returns an EM_SCENE is called by the client before *run* is called and it should return the scene where the example wants to run in. The input parameter can be used to set *next_scene* in the scene returned in order to get back to the main selection scene when the example scene ends.

4.5 TOUCH_EXAMPLE_RUNTIME

The TOUCH_EXAMPLE_RUNTIME class provides the example with the information necessary to interact with the rest of the application. It has features for accessing a TRAFFIC_MAP, TRAFFIC_MAP_WIDGET and a TOUCH_CONSOLE. As additional feature there is *console_out* taking a STRING as an argument. There is no feature for accessing the current scene since this can be queried with other classes and if *run_with_scene* has been redefined the example already knows the exact object and class of the EM_SCENE running.

5 The Examples and Library as in the Textbook

5.1 PREVIEW

5.1.1 Code in Book

```
class PREVIEW
    inherit
        TOURISM
feature
    explore is
            -- Show some city info
        do
            Paris. display
            Louvre. spotlight
            Line8. highlight
            Route1.animate
        end
end
```

5.1.2 Suggested Code

```
class PREVIEW inherit
    TOURISM
feature
    explore is
            -- Show some city info
        do
            Paris. display
            Louvre. spotlight
            Line8. highlight
            Route1.animate
            Passenger.move_route (Route2)
        end
end
```

5.1.3 Description

The biggest problem was posed by the first example. Almost every line of it contains a feature that is not present in the classes suggested. The four problematic features are:

- Paris.*show*
- Line8.*highlight*
- Louvre.*spotlight*
- Route1.*animate*

The book introduces the objects as instances of classes from the TRAFFIC library, but the desired functionality does not exist in these classes. To be more precise the following classes for the objects are intuitively expected:

- Paris: TRAFFIC_MAP
- Line8: TRAFFIC_LINE
- Louvre: TRAFFIC_PLACE
- Route1: TRAFFIC_ROUTE

The problem with *show*, *highlight*, *spotlight* and *animate* is that the TRAFFIC-model is free of any features regarding the visual representation. Let us have a closer look at Line8.*highlight*. *Highlight* is supposed to highlight the line visually. So this is obviously an operation on the visual representation and thus not present in TRAFFIC_LINE. One could add it to the class and mark it as deferred or let it do nothing. So classes that have TRAFFIC_LINE as parent could implement *highlight*. But these enhanced classes would have to be inserted into the map when the map is constructed. This could be done by designing a special factory that already produces objects of these classes or one could rebuild the whole city after the initial construction with objects of classes containing the features for the visual representation. The problem would be if one inserted an object of the base type TRAFFIC_LINE into the map. Furthermore we would lose the generality of having different graphical representations of the same map. One call to *highlight* would affect all views.
Another solution is to use aggregation by using TRAFFIC_LINE as a client. Such a class could provide us with all the features we would like to have, be it *highlight*, *hide*, *show* .... An object of such a class could otherwise not be used where a TRAFFIC_LINE is expected. One would have to take the indirection over a feature like *line*. With this design the features *highlight* and also *spotlight* for the TRAFFIC_PLACE are easy to implement and these new classes could be extended easily because they can be created independently from the map. So they do not have to be taken into account when creating the map.
Using this solution the objects Line8 and Louvre can't be looked at as objects of type TRAFFIC_LINE respectively TRAFFIC_PLACE. I implemented TOUCH_GRAPHICAL_TRAFFIC_LINE and TOUCH_GRAPHICAL_TRAFFIC_PLACE using client relationships for MAP_WIDGET and TRAFFIC_LINE respectively TRAFFIC_PLACE.
So what about Route1? Do we have the same problems as with Line8 and Louvre? The answer is no, although TRAFFIC_ROUTE also does not have the desired feature. But since a TRAFFIC_ROUTE is not an element of the map we can use our own class that just has to inherit from TRAFFIC_ROUTE.

The class for the object Paris could be implemented the same way we implemented the classes for Line8 and Louvre, but I decided not to do so because the feature display really is a feature that belongs more to the widget than it belongs to the map. So in the implementation Paris is just of type TRAFFIC_MAP_WIDGET which provides display and hide and it also offers access to the underlying map.

5.2 QUERIES

5.2.1 Code in Book

```
class QUERIES inherit
    feature
        make is
            -- Try out queries and commands on lines.
        do
            Console.show (Line8.count)
            Line8. i_th (1)
        end
end
```

5.2.2 Suggested Code

```
class
    QUERIES
inherit
    QUERIES_PREDEFINES
feature -- Basic operations
    queries is
            -- Do some queries
        do
            Console.show ("Places")

            Console.show (Simple_Line_8.i_th (1))
            Console.show (Simple_Line_8.i_th (Simple_Line_8.count))

            Console.show ("Line_Sections")

            Console.show (Line_8.i_th (1))
            Console.show (Line_8.i_th (Line_8.count))
        end
end
```

5.2.3 Description

The implementation of this example was straightforward and no additional classes had to be introduced

5.3 COMMANDS

5.3.1 Code in Book

```
class COMMANDS inherit
    TOUR
feature
    make is
            -- Recreate a partial version of Line 8
        do
            Line8.remove_all_segments

            -- No need to add Station_Balard, since
            -- remove_all_segments retains the SW end.
            Line8.extend (Station_Lourmel)
            Line8.extend (Station_Boucicaut)
            Line8.extend ( Station_Felix_Faure )

            -- We stop adding stations, to display some results:
            Console.show (Line8.count)
            Console.show (Line8.ne_end.name)
        end
end
```

5.3.2 Suggested Code

```
class
    COMMANDS
inherit
    COMMANDS_PREDEFINES
feature -- Basic Operations
    build_line_8 is
            -- Recreate part of Line 8
        do
            Simple_Line_8. remove_all_sections

            -- No need to add Station_Balard, since
            -- remove_all_sections retains the 'one_end'.
            Simple_Line_8. extend_place (Place_La_Motte_Picquet_Grenelle)
            Simple_Line_8. extend_place ( Place_Invalides )

            -- We stop adding stations, to display some results:
            Console.show (Simple_Line_8.count)
            Console.show (Simple_Line_8.other_end.name)
            Console.show (Simple_Line_8.one_end.name)
        end
end
```

5.3.3 Description

For this example to work I had to extend the class TRAFFIC_SIMPLE_LINE. I added the feature *remove_all_sections* which just as the texbooks *remove_all_segements* removes all line sections and keeps the *one_end*. To extend the Line with line sections to a place, one can now call *extend_place* and the *other_end* gets connected to the new place.
The only problem left was that the map had no simple lines, it only had instances of normal TRAFFIC_LINEs. Therefore I changed the definition in the dtd and added an attribute *simple* to the line. It can be set to *true* or *false*, default is *false*. To make it work with the example I have set all lines in the mapfile for paris to simple lines. With this changed the assignment attempt to a simple line from a line in the map does not fail anymore.

5.4 LINE_BUILDING

5.4.1 Code in Book

```
class LINE_BUILDING inherit
    TOUR
feature
    build_a_line is
        -- Build an imaginary line and highlight it on the map.
    do
        Paris. display
        Metro. highlight
        -- Create the stops and associate each to its station :
        -- Link each applicable stop to the next:
        create stop1
        stop1. set_station (Station_Montrouge)

        create stop2
        stop2. set_station ( Station_Issy )

        create stop3
        stop3. set_station (Station_Balard)

        stop1. link (stop2)
        stop2. link (stop3)
        -- "Create fancy_line and give it the stops just created"
        fancy_line . illuminate
    end
end
```

5.4.2 Suggested Code

```
class
    LINE_BUILDING
inherit
    LINE_BUILDING_PREDEFINES
feature -- Basic operation
    build_a_line is
            -- Build a fancy new line
        local
            fancy_line : TRAFFIC_SIMPLE_LINE
        do
            create fancy_line . make ("fancy_line", tram_type, Paris)

            -- Extend the fancy_line by three stations
            fancy_line . extend_place (Place_Balard)
            fancy_line . extend_place ( Place_Issy )
            fancy_line . extend_place (Place_Montrouge)

            -- Add the fancy line to the map
            Paris. add_line ( fancy_line )
        end
end
```

5.4.3 Description

This example does not fit into the new library in any way. Building a new line is done very different and there is no METRO_STOP or a similiar class in the new design of the library. One could write a custom TRAFFIC_STOP class, so TRAFFIC_STOPs could be assigned to TRAFFIC_PLACEs and then linked with other TRAFFIC_STOPs to form a new line. To make this construct an effective line in the map one could provide a special feature to add it to the map. I also implemented a class

SIMPLE_STOP like in the textbook and could almost recreate the original example from the book. The class SIMPLE_STOP has a feature called *build_line* which creates a TRAFFIC_SIMPLE_LINE. The implemention is the following.

```
    build_line (a_line_name: STRING;
                a_line_type : STRING;
                a_map: TRAFFIC_MAP): TRAFFIC_SIMPLE_LINE is
            -- Convert the linked List to a Line
        require
            a_line_type_exists : a_line_type /= Void
            a_line_name_exists : a_line_name /= Void
            a_map_exitst: a_map /= Void
        local
            simple_line : TRAFFIC_SIMPLE_LINE
            stop: SIMPLE_STOP
            factory: TRAFFIC_MAP_FACTORY
        do
            -- Create factory
            create factory.make

            -- Build simple line
            factory. build_simple_line (a_line_name, a_line_type , a_map)
            -- Get created simple line
            simple_line := factory. simple_line

            -- Convert linked list of SIMPLE_STOPs to simple_line
            from
                stop := Current
            until
                stop = Void
            loop
                -- If Place is set then
                if stop.place /= Void then
                    simple_line . extend_place (stop. place)
                end
                -- Go to the next item in the list
                stop := stop.next
            end

            -- Save Result and return
            Result := simple_line
        end
```

5.5 TRAFFIC_PLACE.is_exchange

The query feature *is_exchange* which is a member of METRO_STATION in the design proposed by the textbook would be a feature of a TRAFFIC_PLACE in the new design. But since in the new design a place is not necessary connected to a line the query *is_exchange* does not always make sense. Another problem is that a place does not know anything about the map, so it can not know what lines pass through it. *Is_exchange* can be added to be a query in the map and it can be added to TOUCH_GRAPHICAL_TRAFFIC_LINE which knows the place, the map widget and therefore also knows the map.

6 Conclusions

Being new to Eiffel programming language I first had to find my way into the world of assertions and assignment attempts. But once I got used to it I thought it was quite nice to have consctructs like require or ensure. After having learned the basics of Eiffel I had to understand the TRAFFIC library and of course the ESDL library as well. I think it was a good decision to do a redesign on TRAFFIC since the new version is quite easy to understand if you don't want to look at every little detail. During the time of my semester thesis ESDL began to change and about 10 new semester thesis started to extend ESDL and build a multimedia library. The so called EiffelMedia(EM) was further on the replacement for ESDL.

I had a look at the first 5 chapters where I found 4 examples and some other code that conflicted with the library. I had to rewrite the examples and change it where necessary and I also added code to the library where I thought it could simplify the examples and the real executable code. I also added to most of the examples code that shows how you can do the same thing but in a different way. Inventing new examples is one thing I droppped because the examples are very tightly bound to the book itself, so changing an example means changing a whole chapter.

7 Future Work

7.1 Other Chapters

I only took a look at the first 5 chapters. While the first examples are the most important ones there are still many examples left having not been checked for compatibility with the new TRAFFIC library.

7.2 EiffelMedia

Although I ported the code from ESDL to EiffelMedia I did not use the full potential of the library. This has mainly two raisons

- When I started my work EiffelMedia was not even beeing planned and so I implemented for example own GUI-Widgets.
- EiffelMedia ist still being developed and it evolved so rapidly that I could not adapt my code to the new features being added daily.

7.3 Visualization

In parallel with all the other semester theses that started their work on EiffelMedia another group began to write a 3d visualization of a map using OpenGL for graphics output. This of course looks much fancier and it is clear that this visualization should be integrated into TOUCH.

References

[1] Ursina Caluori *Flat Hunt*. ETH Zurich, 2005.
`http://se.inf.ethz.ch/projects/ursina_caluori`

[2] Sibylle Aregger. *Redesign of the TRAFFIC library*. ETH Zurich, 2005.
`http://se.inf.ethz.ch/projects/sibylle_aregger`

[3] Rolf Bruderer. *Object-Oriented Framework for Teaching Introductory Programming*. ETH Zurich, 2005.
`http://se.inf.ethz.ch/projects/rolf_bruderer`

[4] Marcel Kessler. *Exercise Design for Introductory Programming. "Learn-by-doing" basic OO-concepts using Inverted Curriculum*. ETH Zurich, 2004.
`http://se.inf.ethz.ch/projects/marcel_kessler`

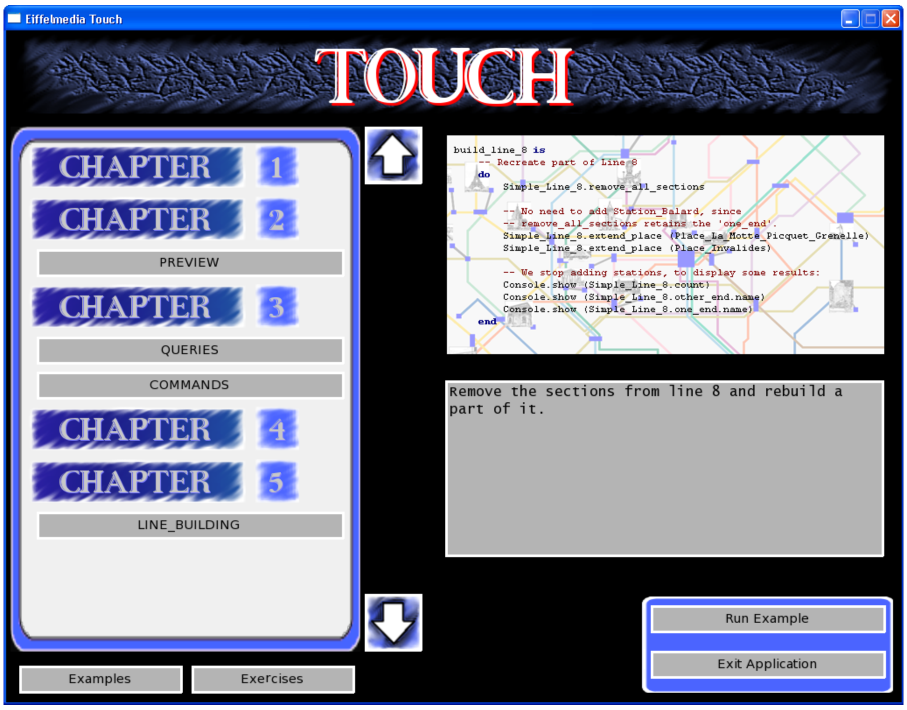
[5] Benno Baumgartner. *ESDL - Eiffel Simple Direct Media Library*. ETH Zurich, 2004.
`http://se.inf.ethz.ch/projects/benno_baumgartner`

[6] Michela Pedroni. *Teaching Introductory Programming with the Inverted Curriculum Approach*. ETH Zurich, 2003.
`http://se.inf.ethz.ch/projects/michela_pedroni`

[7] Till G. Bay. *Eiffel SDL Multimedia Library (ESDL)*. ETH Zurich, 2003.
`http://se.inf.ethz.ch/projects/till_bay`

A Touch User Guide

TOUCH is the Application that comes with the textbook “Touch of Class”. TOUCH implements some of the examples of the book. Note that the examples as described in TOUCH are not totally equivalent to their description in the textbook. At the time of this writing not all the examples of the book are implemented yet. So TOUCH needs to be taken with a pinch of salt!



```
build_line_8 is
    -- Recreate part of Line 8
  do
    Simple_Line_8.remove_all_sections

    -- No need to add Station_Balard, since
    -- remove_all_sections retains the 'one_end'.
    Simple_Line_8.extend_place (Place_La_Motte_Picquet_Grenelle)
    Simple_Line_8.extend_place (Place_Invalides)

    -- We stop adding stations, to display some results:
    Console.show (Simple_Line_8.count)
    Console.show (Simple_Line_8.other_end.name)
    Console.show (Simple_Line_8.one_end.name)
  end
```

Remove the sections from line 8 and rebuild a part of it.

Figure 7: The main screen

A.1 How does this Application work?

On the left side you can see the chapters and their examples. To change between the examples from the book and the exercises use the two buttons on the bottom left of the screen. Simply click on an example and some information and a picture will pop up on the right side. To see the example in action use the “Run example” button on the right. What happens further depends on the implementation of the example, but with most of the examples you will get a screen like this

You can navigate in the map by pressing the mouse buttons.

- Pressing the right mouse button while moving the mouse will scroll the map
- Moving the mouse up or down while pressing the middle mouse button will zoom in or out
- Clicking the left mouse button will select places or lines

To get back to the menu simply click the “End Example” button.

References

[1] Ursina Caluori. *Flat Hunt Redesign*. ETH Zurich, 2005. `http://se.inf.ethz.ch/projects/ursina_caluori`

[2] Sibylle Aregger. *Redesign of the TRAFFIC library*. ETH Zurich, 2005. `http://se.inf.ethz.ch/projects/sibylle_aregger`

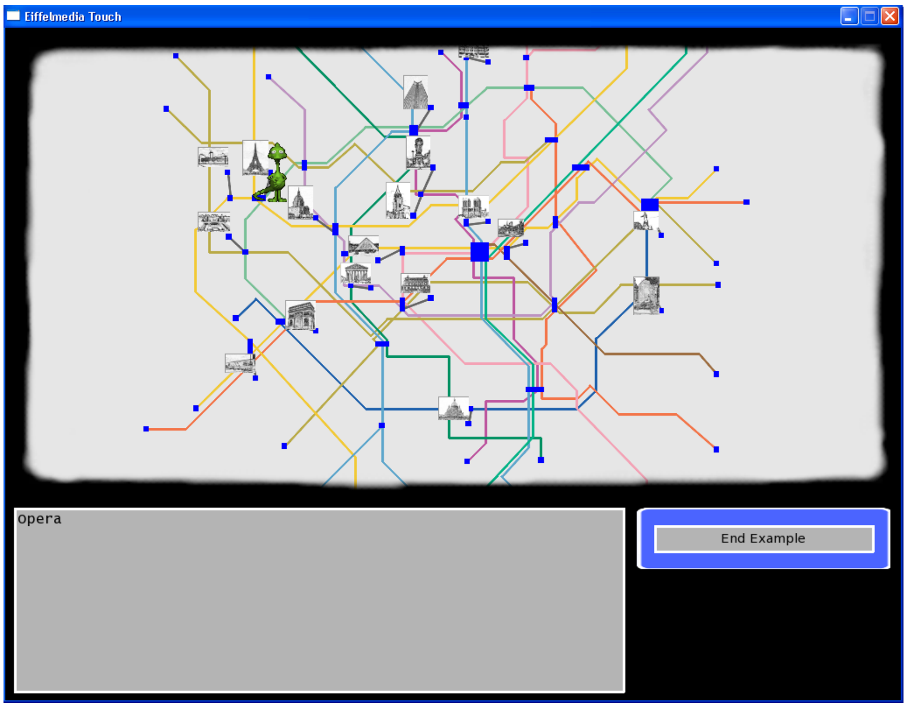


Figure 8: map example screen

[3] Till G. Bay. *Eiffel SDL Multimedia Library (ESDL)*. ETH Zurich, 2003. `http://se.inf.ethz.ch/projects/till_bay`

B Touch Developer Guide

B.1 Overview

This part describes the design and structure of the classes that build up TOUCH. This first chapter gives a short overview of the whole system. The next chapter describes each cluster and its most important classes. Figure 9 shows the clusters as they are displayed in EiffelStudio. The clusters that are important for you to work with TOUCH are found in top-level cluster Touch. This cluster has itself four subclusters called examples, utils, visualization and widgets. These cluster are described in the next chapter

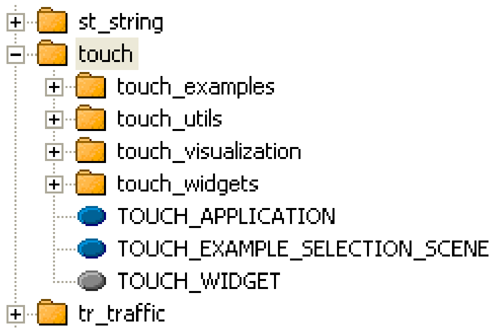


Figure 9: Clusters of Touch

B.2 Clusters

Cluster Touch contains only a few classes. But one of them is probably the most important: TOUCH_EXAMPLE_SELECTION_SCENE. It is the main scene for TOUCH. The example registration and example selection is coded within this class. The mechanism of inserting new examples and exercises is explained in the next section.

Cluster examples This Cluster contains the classes used to build an example: The deferred class TOUCH_EXAMPLE and TOUCH_SIMPLE_TRAFFIC_SCENE. It also contains the class TOUCH_EXAMPLE_RUNTIME that provides the running example with access to the map etc. The implemented examples that come with the application can also be found here.

Cluster utils In this cluster there are only three simple helper classes used for basic timing and for getting an unique id.

Cluster visualization Several classes that have to do with the graphical representation are contained in this cluster.

Cluster Widgets Since the main part of this application was written before EiffelMedia was created, TOUCH comes with its own widgets.

B.3 Touch classes

In figure 10 you see the most important classes and their inheritance and client relations.

B.3.1 Most important classes for developers

TOUCH_EXAMLE_SELECTION_SCENE is the main scene. It provides the user with the ability to select an example and run it. if you build a new example or a new exercise then you must register your class here. The next section explains how to implement a new example.

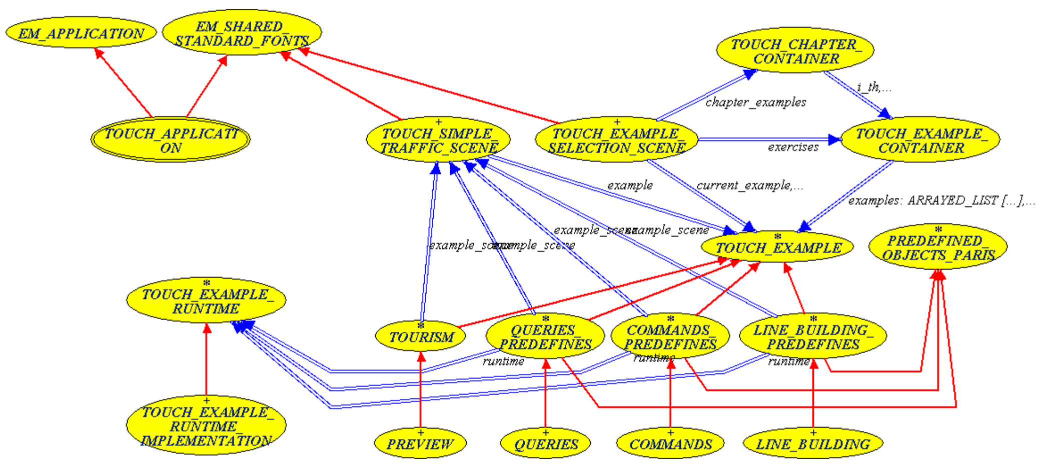


Figure 10: Touch Class Overview

TOUCH_EXAMPLE is the class you inherit from if you want to build a new example or exercise. It has the following features

- *name*
- *description*
- *run_with_example*
- *run*

TOUCH_EXAMPLE_RUNTIME is the interface for a running example. It allows an example to access

- *map*
- *map_widget*
- *console*

TOUCH_SIMPLE_TRAFFIC_SCENE is the scene used to run the first four examples. The scene loads the designated map and displays it.

B.4 Building a new example

To successfully integrate an example into TOUCH you have to do the following three steps

B.4.1 Create a new Class

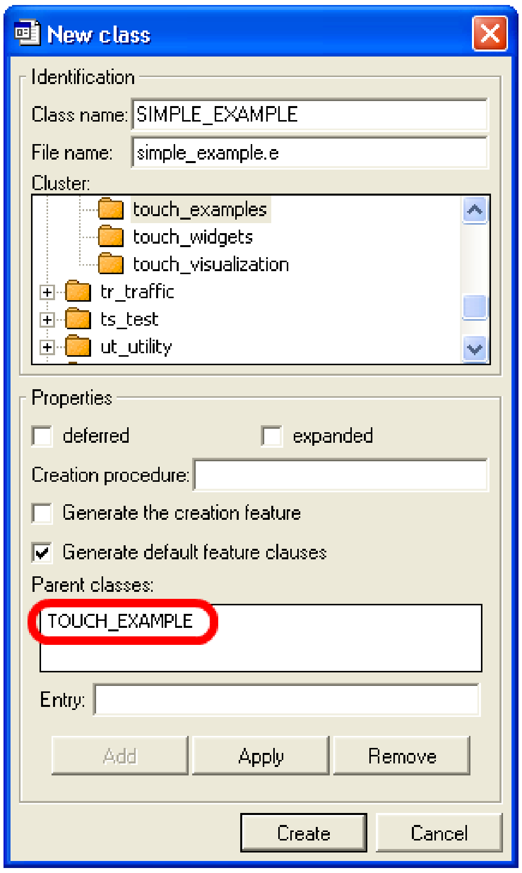
Just create a new class and add TOUCH_EXAMPLE to the parent classes. Figure 11 shows how it should look like in Eiffelstudio.

B.4.2 Implement the needed features

Inheriting from TOUCH_EXAMPLE requires you to give an implementation for the two features *description* and *run*. You can also redefine name to give your example an other name than the name of your class. If you don't redefine *run_with_scene* a standard scene and an empty map is set up for you. A simple example could look like the following:

```
class
    SIMPLE_EXAMPLE

inherit
    TOUCH_EXAMPLE
```



Class name: SIMPLE_EXAMPLE
File name: simple_example.e

Cluster:
- touch_examples
- touch_widgets
- touch_visualization
- tr_traffic
- ts_test
- ut_utility

[ ] deferred [ ] expanded
Creation procedure:
[ ] Generate the creation feature
[✓] Generate default feature clauses
Parent classes:
TOUCH_EXAMPLE
Entry:

Figure 11: Creating a new example

```
        redefine
            run_with_scene
        end

feature -- Access
    description : STRING is
        once
            Result := "Hello_World_Example"
        end

feature -- Basic operations
    run_with_scene ( exit_scene : EM_SCENE): EM_SCENE is
        local
            example_scene: TOUCH_SIMPLE_TRAFFIC_SCENE
        do
            -- Create the scene we want our example to run
            create example_scene. make_with_zurich_little (Current)
            -- Set the exit_scene
            example_scene. set_next_scene ( exit_scene )
            -- Return scene
            Result := example_scene
        end

    run (a_runtime: TOUCH_EXAMPLE_RUNTIME) is
        do
```

```
            -- Ouptut Hello World
            a_runtime.console_out ("Hello World")
        end

end -- class SIMPLE_EXAMPLE
```

B.4.3 Register the example

So let's say you have written your own example and want to test it. You compile the project but you can't find your example anywhere. This is because you have to register your example to TOUCH first. To do this you have to load the TOUCH_EXAMPLE_SELECTION_SCENE class and edit one of its features.

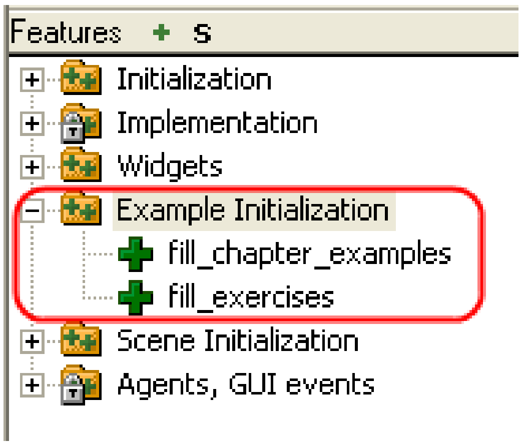


Figure 12: Example initialization

If you want your example to appear in one of the chapters then you must add your example in fill_chapter_examples. If you wanted to add SIMPLE_EXAMPLE to chapter 1 you would have to add the follwing:

```
chapter_examples.i_th (1).subscribe (create {SIMPLE_EXAMPLE})
```

To change the number of chapters. Just adapt the following line in the code or call it later to create more chapters:

```
chapter_examples.set_count (5)
```

If your example does not belong to any chapter you can put it to the exercises. Just add the following line to fill_exercises:

```
exercises.subscribe (create {SIMPLE_EXAMPLE})
```

References

[1] Ursina Caluori. *Flat Hunt Redesign*. ETH Zurich, 2005.
`http://se.inf.ethz.ch/projects/ursina_caluori`

[2] Sibylle Aregger. *Redesign of the TRAFFIC library*. ETH Zurich, 2005.
`http://se.inf.ethz.ch/projects/sibylle_aregger`

[3] Till G. Bay. *Eiffel SDL Multimedia Library (ESDL)*. ETH Zurich, 2003.
`http://se.inf.ethz.ch/projects/till_bay`