Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Object-Oriented Framework for Teaching Introductory Programming

Master Thesis

Rolf Bruderer
bruderer@computerscience.ch

Supervising Professor: Prof. Dr. Bertrand Meyer
Supervising Assistants: Michela Pedroni, Till G. Bay

Department of Computer Science
Chair of Software Engineering
ETH Zurich

March 2005

To my parents

Abstract

The introductory programming course at ETH uses an object-oriented framework to teach first semester students. The framework consists of a library called TRAFFIC to model and to visualize the transportation network of a city. It enables the students to produce interesting applications with graphical output from the first exercise.

The existing framework used EiffelVision2 as graphical library. Due to the complexity of EiffelVision2 we decided to change it to use ESDL: a platform independent multimedia library for Eiffel. This thesis describes all the extensions that were made to ESDL in order to use it for the introductory programming course. This includes graphical primitives, classes for figures, coordinate transformations, mouse and keyboard events, and base classes for interactive applications.

The visualisation part of the TRAFFIC library has been redesigned to use ESDL and to support multiple views of a transportation network model.

The challenge of this thesis was to develop libraries that are easy to use for students and still reusable to produce interesting applications.

Contents

Chapter 1

Introduction

In the winter semester 2003/2004 a wind of change blew through the Department of Computer Science at ETH Zurich. Professor Bertrand Meyer held the course "Introduction to Programming" for first semester students. He redesigned the previous course using the "Inverted Curriculum" approach [1, 2]. In this approach students learn to grow from consumers to producers of object-oriented systems. Instead of starting with boring little programs like "Hello World" the students use an object-oriented framework in their programming exercises. This framework consists of libraries that enable the students to produce interesting applications right from the start.

One of the libraries provided to the students is called TRAFFIC [3, 4]. It is a library for modeling and visualizing the public transportation network of a city. By using TRAFFIC, students are able to produce fancy applications from the outset, just by writing a few lines of code. As they progress, the students learn to build more and more elaborate programs. Through the exercises they even start to understand the library from the inside and to modify it. Therefore this approach is also known as "progressive opening of black boxes" or as "outside-in approach" [1].

The first two runs of the course showed that the framework needs to be developed very carefully. The requirements on the design of the libraries are demanding. Even for students that have no programming experience yet the libraries must be easy to use. On the other hand the libraries should still be generic enough to provide the students with possibilities to build impressive applications. Developing such libraries is a big challenge.

It is very hard to understand a program when you can not see what it is doing. It is much easier to understand a program when most of its commands have a visual effect. Therefore, a framework for programming exercises should provide the students with an easy way to produce graphical output. The existing framework uses EiffelVision2 for this purpose, which is the GUI-library of Eiffel[5]. This library is very well designed and supports everything a developer needs to build graphical user interfaces. The problem is that students get lost very fast using this large library. There is another library that supports graphics in Eiffel: ESDL [6, 7]. This library is developed at the Chair of Software Engineering and is an Eiffel wrapper of the Simple Directmedia Layer (SDL)[8]: a C library for multimedia. ESDL provides an object-oriented API that is easy to understand and easy to remember for developing interactive graphical applications. We expect that students can produce impressive applications like little games using this library. ESDL has already been used once in another software engineering course for advanced students. The games the students programed using ESDL are an evidence of its power [9].

At the start of this master thesis there were still a lot of features missing in ESDL that are needed for introductory programming exercises. To provide students with a powerful graphic library, I made the following extensions to ESDL:

- Graphical primitives and figure classes: e.g. lines, rectangles and polygons
- Clipping
- Coordinate transformations: translation and scaling

- Zooming and rotation of images
- Keyboard and mouse events
- Base classes for applications

Using these ESDL extensions I redesigned the visualization part of the TRAFFIC library. The new design is more flexible since the model objects do not know the view objects anymore. This allows clients to have more than one view of a transportation map model. Creating a view for a map model has become much easier in the redesigned TRAFFIC library.

Chapter 2

Existing Framework

2.1 Overview

When I started my master thesis there was already a framework used for teaching introductory programming. This framework mainly consists of a library to model and visualize the traffic network of a city, called TRAFFIC. It contains two example applications built on top of this library: TOUCH and FLAT_HUNT. This existing framework was used in the winter semester 2004 / 2005 during the time of my master thesis to teach the introductory programming course for computer science students at ETH Zurich. The students were expected to understand the FLAT_HUNT application and to change or extend it during their exercises [4].

I helped maintaining the framework for the exercises during the semester and also supervised an exercise group of this course as a helping assistant. It was very interesting to accompany the application of the framework and to experience how the students learned to program with it.

In the following subsections I will shortly introduce the important parts of the existing version of the framework. I will discuss the drawbacks of its design that I have encountered. Furthermore, I will propose a possible solution for each drawback.

2.2 TRAFFIC Model

The existing TRAFFIC library provides the students with classes for modeling a city and its public transportation network. The model consists of places in a city and links between them. A link is a connection between two consecutive places of a line, e.g. a tram line or a bus line. All the places and links are stored in a graph that allows to calculate shortest paths. The following class diagram shows the classes of TRAFFIC to model such a network.

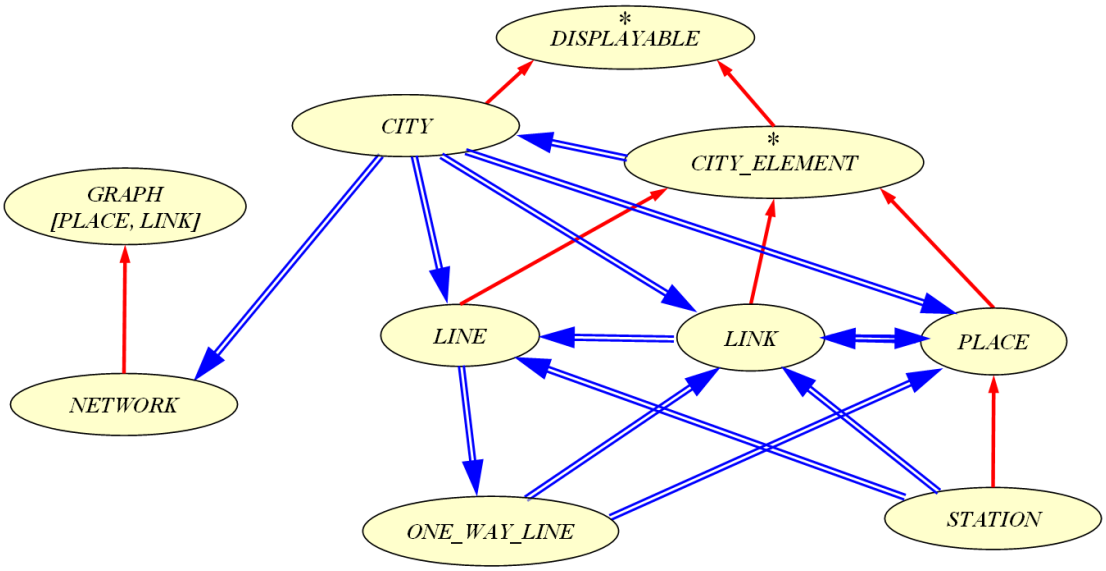

Figure 2.1: Classes in existing TRAFFIC library to model a city

Sibylle Aregger did a semester thesis to redesign this model during the time of my master thesis. Therefore all drawbacks we found about this part of the traffic library are written in her thesis [10] and I will not discuss them here. My work focused on the visualization part of the framework.

2.3 TRAFFIC Visualization

2.3.1 Overview

TRAFFIC also comes with classes to visualize the city model as a map. The visualization uses EiffelVision2 which is the GUI-library of Eiffel.

Figure 2.2: Visualization of city map in TOUCH application

The map is drawn using the class CANVAS which is a widget-component, developed by Till G. Bay [6]. CANVAS inherits from EV_PIXMAP, which is a bitmap inside an EiffelVision window onto which we

can draw. The class CANVAS comes with classes inheriting from DRAWABLE_OBJECT. Such drawable objects can draw themselves onto a CANVAS. There exist classes for rectangles, circles, polygons, polylines and images. A CANVAS takes a linked list of DRAWABLE_OBJECTs and displays them. It is possible to zoom or scroll the content of a CANVAS by setting a rectangular visible area. This *visible_area* defines which coordinates are projected onto the pixmap. There is another class TRAFFIC_CANVAS inheriting from CANVAS. TRAFFIC_CANVAS has additional features for convenience to change this visible area. It is quite easy to draw some drawable objects using the class CANVAS and even to zoom or scroll them. This class has been designed for the TRAFFIC library to make it easy to zoom or scroll the map of the city.

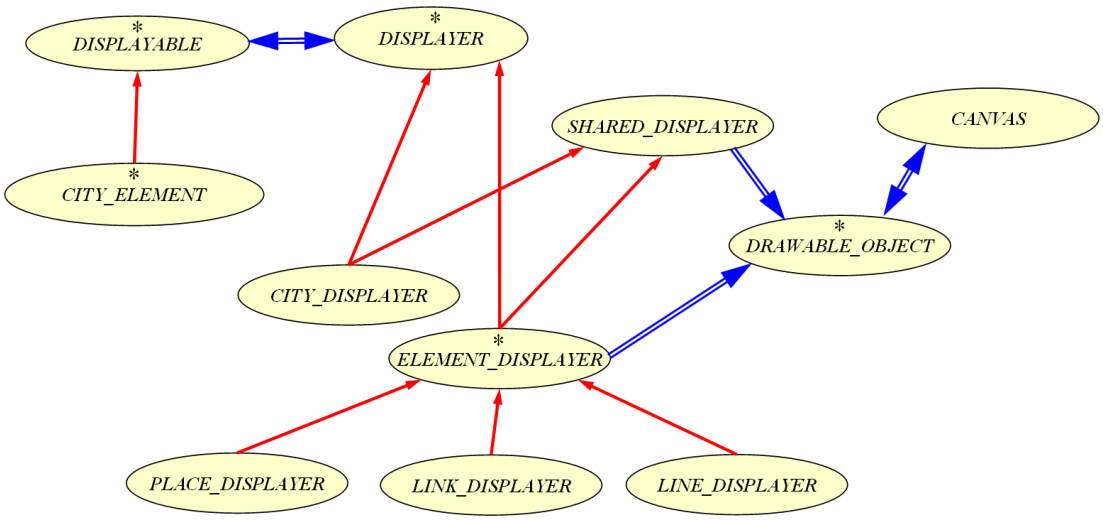

Figure 2.3: Classes in existing TRAFFIC library to visualize a city

To visualize the model elements of the city as a map there are displayer classes in the TRAFFIC library. For each city element there is a displayer class: PLACE_DISPLAYER, LINK_DISPLAYER and LINE_DISPLAYER. Each displayer holds a drawable representation of a city element as a linked list of DRAWABLE_OBJECTs. These DRAWABLE_OBJECTs can then be drawn onto a CANVAS to visualize the element.

Each city element is displayable and therefore has a displayer object to display it. Thus each PLACE has a PLACE_DISPLAYER, each LINK has a LINK_DISPLAYER and each LINE has a LINE_DISPLAYER. When clients want to change the way a place is displayed inside the map, e.g. if they want to highlight a place, they just call the feature *highlight* of its *displayer*. To ensure that this change directly gets effective inside the CANVAS that displays the map there is a class SHARED_DISPLAYER. This class provides a singleton of a linked list of drawable objects. Each displayer object accesses this linked list to put its drawable objects that represent its element into this list. When a displayer has to change the way its element is displayed it has to change its drawable objects inside this list accordingly. To display the map we have to initialize a CANVAS to draw the linked list of the SHARED_DISPLAYER.

I will discuss two drawbacks of this design in the following subsections.

2.3.2 Canvas and Drawable Objects

A DRAWABLE_OBJECT is an object that draws itself onto a CANVAS. That sounds quite easy and one could expect that it would be a good programming exercise for students to implement such a drawable object by inheriting from DRAWABLE_OBJECT and implementing the feature that performs the drawing onto the canvas. But there is a problem. The CANVAS can be zoomed or scrolled. The CANVAS does not ensure that all drawings that are performed onto it get scrolled or zoomed accordingly. Instead, it expects that all drawable objects perform this coordinate transformation on their own when drawing themselves.

This should not be the case. A zoomable and scrollable canvas should be aware of performing this coordinate transformation for all drawings that are performed on it. Contained objects should not have to care whether they are drawn inside a container that has been scrolled or zoomed.

I suggest that the drawable objects in our new framework will not have to care about the coordinate system of their parent container. Instead, there should be the possibility to change the coordinate system with which all drawings are performed. Therefore, the canvas has to perform the coordinate transformation in all its drawing commands. This would mean that we would have to redefine all drawing commands of CANVAS inherited from EV_PIXMAP to do the coordinate transformations.

2.3.3 Displayers

Another drawback of the existing design is, that it assumes that there is only one view of the map. It is not in the current design to introduce two visualizations of the same map. Of course we can introduce two CANVAS objects that both display the same drawable objects inside the linked list of SHARED_DISPLAYER. But this would mean that we get exactly the same view of the map multiple times. It is not possible to visualize the city differently using the displayer classes of the TRAFFIC library. For example it would not be possible to display places in one map as circles and in another map as rectangles. Since there is only one singleton linked list into which all displayers put their drawable objects, there can only be one kind of visualization at a time.

To change this, we have to do several changes to the current design. First of all, there should be no need for a class like SHARED_DISPLAYER holding a singleton of a list into which all displayers can put their drawable objects. Instead of this, a class that has the purpose to visualize a city element should be able to put a graphical representation of the element it visualizes into any canvas. Then we could create several objects of this class to visualize the same element inside different city maps. Possibly we could even set up different view objects visualizing the same model object in a different way (e.g. different color, different shape, different decorations etc.). We already have a class that we can use as a base class for these views, that is DRAWABLE_OBJECT. Therefore my proposal is to introduce classes, e.g. DRAWABLE_PLACE, DRAWABLE_LINK, etc. to draw the different city elements into a canvas. These classes would take over most of the functionality that the displayer classes were responsible for in the former framework. For example DRAWABLE_PLACE would have a feature *highlight* to highlight a place when it is selected. The advantage of this design is that we could create several view objects for the same place and put them into different CANVAS instances. This approach is much more reusable as the former design, where each city element had exactly one displayer object attached and all displayer objects were drawing into the same canvas or at least the same shared linked list of drawable objects.

Please consider section 4.3 for a more detailed discussion about the way this is implemented inside the redesigned TRAFFIC library.

Chapter 3

ESDL Multimedia Library

3.1 Overview

Students want to have fun when they start to learn programming. A program that does nothing else than displaying boring text output is not very thrilling for today's students [11]. Even more, it is very hard to understand a program, when you can not see what it is doing. It is much easier to understand a program when most of its commands have an effect that you can see directly on screen. Therefore, a good framework for teaching programming needs to provide the students with the possibility to produce graphical output very easily. Programming exercises that directly lead to visible results will not only be much more motivating for the students, but probably also lead to a better learning effect.

ESDL is an object-oriented wrapper in Eiffel for SDL, the Simple Directmedia Layer library [8]. SDL is a multimedia open source C library available for many platforms. It makes developing of graphical programs easy. Furthermore, SDL also supports sound, joystick, mouse, networking, CD-ROM etc. ESDL does not only provide the possibility to use SDL in Eiffel, but already comes with powerful classes that make it very easy to build object-oriented graphical applications. The API of ESDL is developed very carefully to keep it as simple as possible for the clients. Therefore, it is appropriate to use ESDL in an introductory programming course for giving the students the ability to produce graphical applications, for example small games.

In the latest version of ESDL a lot of features were missing that are needed to provide the students with an easy way to produce graphical applications. The object-oriented API mainly supported bitmap graphics and sound. A lot of other SDL functionality was already available through wrapper functions, but the object-oriented API to provide clients with an easy way for using these features was not yet developed.

I implemented the following extensions to ESDL:

- Graphical primitives and appropriate figure classes: lines, rectangles, circles, polygons
- Clipping
- Coordinate transformations: translation and scaling
- Bitmap transformations: zooming and rotation
- Keyboard and mouse events

Furthermore I did some changes in the existing ESDL design and added reusable base classes that make it easy for students to build their own interactive applications, e.g. a container that can zoom all contained graphical objects. I will discuss all these extensions in this chapter.

3.2 Requirements

The main goal of my thesis was to develop everything needed to visualize the city map of TRAFFIC using ESDL. The visualization part of the existing TRAFFIC library was designed for EiffelVision2, the GUI-library of Eiffel. This visualization part mainly consists of a widget component called CANVAS developed by Till G. Bay [6]. The class CANVAS inherits from EV_PIXMAP, which is a bitmap onto which you can draw. The widget comes along with a lot of DRAWABLE_OBJECT classes, that can draw themselves onto a CANVAS. There are classes for rectangles, circles, polygons, polylines and images. It is possible to scroll or even zoom the content of the widget. This existing visualization part of TRAFFIC was discussed in section 2.3 together with proposals on how to improve it for the redesigned framework.

To implement the visualization of the traffic map as proposed in section 2.3.2 we need the following functionality in ESDL:

- **Drawing Primitives:** To visualize the city map we need to have the possibility to draw lines with arbitrary line width, rectangles, circles, polygons, polylines and of course bitmap images. Bitmap images in various formats (e.g. GIF) are already supported quite perfectly in the existing ESDL library. All other needed drawing primitives, like line segments and polygons are missing.
- **Drawable Object Classes:** We need easy to use classes to represent the drawable objects, like rectangles for the city places, polylines for the traffic lines, and pictures for the landmarks and passengers. There is already a deferred base class ESDL_DRAWABLE for such classes that draw themselves onto a drawing surface. But concrete classes for graphical figures like polygons and polylines are missing, since there are no corresponding drawing commands yet.
- **Zoomable and Scrollable Container:** In order to visualize the traffic map I need to implement a container into which we can put all drawable objects and that can zoom and scroll its content. There is already a container class ESDL_DRAWABLE_CONTAINER that draws all its contained ESDL_DRAWABLE objects. But this class has no support to zoom or scroll the content.
- **Animation:** There needs to be an easy way to animate movable objects in the traffic map, e.g. passengers or transportation vehicles. ESDL already comes with some animation mechanism. But as discussed in subsection 3.3.2.5 this existing mechanism has its drawbacks.
- **Mouse and Keyboard:** Of course we also need support for interaction through keyboard and mouse events. Maybe ESDL should even support an easy way to get informed whenever a specific drawable object (like a rectangle representing some place of the city) has been clicked. There is already an event dispatching mechanism in ESDL which makes it possible to subscribe for mouse and keyboard events. But the event classes are incomplete. For example, it is not yet possible to query for the key that caused a keyboard event. Furthermore, there is no possibility to directly catch mouse events that occurred over a specific visible object.

3.3 Design Discussion

This section gives a short overview on the existing ESDL library and discusses some drawbacks of its design. Along with the drawbacks I will explain the changes I made to the ESDL design for lowering these drawbacks. The design of all my extensions will be explained in more detail in the following sections.

3.3.1 Existing Design

The most important class in the existing ESDL design is probably ESDL_SURFACE. An ESDL_SURFACE is a bitmap in memory representing an image. The display itself is represented through an ESDL_SUR-FACE object. Other ESDL_SURFACE objects can be created, e.g. by loading them from image files. An ESDL_SURFACE has two very important features *blit_surface* and *blit_surface_part* to draw other ESDL_SURFACE objects onto it.

```
blit_surface (a_surface: ESDL_SURFACE; an_x: INTEGER; an_y: INTEGER)
        -- Blit 'a_surface' at 'an_x' and 'an_y' onto 'Current'.
    require
        a_surface_not_void: a_surface /= Void
    ensure then
        surface_operation_successful: surface_operation_successful
                                      or last_error = Error_blit_surface


blit_surface_part (a_surface: ESDL_SURFACE; an_x, an_y, a_width, a_height: INTEGER;
                   dest_x, dest_y: INTEGER)
        -- Blit 'a_surface' the part from 'an_x' and 'an_y' to 'a_width' and 'a_height'
        -- onto 'Current' at 'dest_x', 'dest_y'.
    require
        a_surface_not_void: a_surface /= Void
    ensure then
        surface_operation_successful: surface_operation_successful
                                      or last_error = Error_blit_surface
```

Listing 3.1: Surface Blitting Features in Class ESDL_SURFACE

The most important deferred class in ESDL is ESDL_DRAWABLE (see Listing 3.2). An ESDL_DRAWABLE is something that can draw itself onto an ESDL_SURFACE. It has a deferred command *draw* to draw the drawable object onto an ESDL_SURFACE that is passed as an argument. An ESDL_DRAWABLE has a position (attributes x and y) that can be set (commands *set_x*, *set_y* and *set_x_y*) and a size (deferred features *width* and *height*).

```
deferred class interface
    ESDL_DRAWABLE

feature -- Commands

    draw (a_surface: ESDL_SURFACE)
            -- Draws 'current' to 'a_surface'
        require
            a_surface_not_void: a_surface /= Void

    draw_part (rect: ESDL_RECT; a_surface: ESDL_SURFACE)
            -- Draws rectangular part of 'current' defined by 'rect' to 'a_surface'
        require
            a_rect_not_void: rect /= Void
            a_surface_not_void: a_surface /= Void

feature -- Status

    x: INTEGER
            -- The distance of 'current' to the coordinate origin
            -- in 'x' (left -> right) direction

    y: INTEGER
            -- The distance of 'current' to the coordinate origin
            -- in 'y' (top -> down) direction

    width: INTEGER
            -- The 'width' of 'current'
            -- The 'width' is the position of the right most pixel
        ensure
            Result >= 0
```

```
    height: INTEGER
            -- The 'height' of 'current'
            -- The 'height' is the position of the bottom most pixel
        ensure
            Result >= 0

    set_x (an_integer: INTEGER)
            -- Sets 'x'

    set_x_y (an_x: INTEGER; an_y: INTEGER)
            -- Sets the 'x' and 'y' value

    set_y (an_integer: INTEGER)
            -- Sets 'y'

end -- class ESDL_DRAWABLE
```

Listing 3.2: Existing ESDL_DRAWABLE Class Interface

There are already a lot of classes in ESDL that inherit from ESDL_DRAWABLE to support an easy way to construct animatable bitmap graphics (see Figure 3.1). Even ESDL_SURFACE inherits from ESDL_DRAWABLE since an ESDL_SURFACE can also be drawn onto another one.

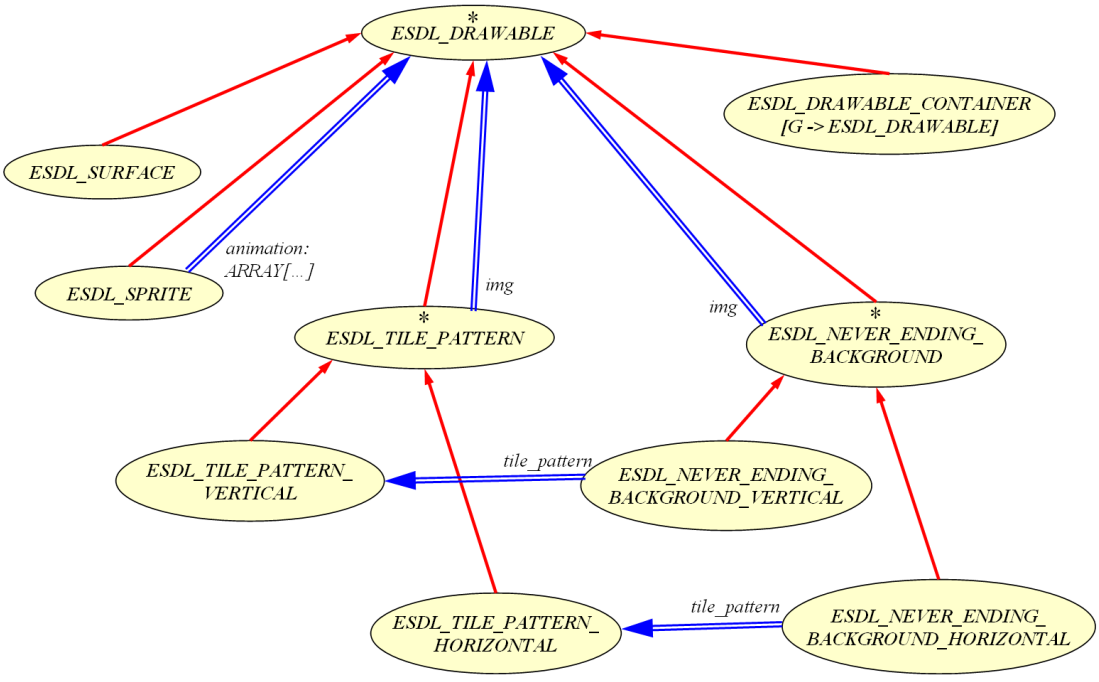

Figure 3.1: Most important classes inheriting from ESDL_DRAWABLE

All classes inheriting from ESDL_DRAWABLE, except for ESDL_SURFACE, are composed of other ESDL_DRAWABLEs. For example an ESDL_SPRITE is a viewer for showing an ESDL_ANIMATION. An ESDL_ANIMATION is nothing else than an ARRAY of ESDL_DRAWABLEs. When *draw* is called on an ESDL_SPRITE, it picks the next ESDL_DRAWABLE in the ESDL_ANIMATION that needs to be drawn and calls *draw* on this ESDL_DRAWABLE.

The only ESDL_DRAWABLE that directly draws itself onto a surface, without delegating the drawing to another ESDL_DRAWABLE is ESDL_SURFACE. This is not very surprising, since the only possibility to draw something onto an ESDL_SURFACE so far, is to draw an image (ESDL_SURFACE)

onto it, using the features *blit_surface* or *blit_surface_part*. To be completely honest, there is another possibility, namely to change the pixels of the bitmap. Anyway, this second possibility was not used by any ESDL_DRAWABLE so far.

There is also a base class for building interactive scenes, the deferred class ESDL_SCENE. This class is intended to make the live of interactive application developers much easier. It contains the reusable parts that are always needed to build an interactive scene in an ESDL application. It has a deferred feature *initialize_scene* where descendants are supposed to build the drawable objects the scene consists of. Then it has another feature *run* that has to be called when the scene has been initialized to execute it. Effective descendants of this class are supposed to draw the scene onto the screen that is passed as argument to the feature *run*. ESDL_SCENE provides the developer with an *event_loop*, which is an instance of the class ESDL_EVENT_LOOP. This event loop gets started when the *run* feature is called and dispatches all the events from SDL until the event loop is terminated. The class ESDL_SCENE already has some event handler features that are executed when some events (e.g. keyboard events) occur in the event loop.

Developers can simply inherit from this class ESDL_SCENE and implement the deferred feature *initialize_scene* to build the scene they want to display. To make the scene interactive developers can redefine the event handling features to perform some interaction, e.g. when a key is pressed. Of course more advanced programmers can also subscribe for events using the *event_loop*. There is also a feature *handle_outside_event* that is called from time to time and that can be used to redraw the screen after each frame, i.e. to perform animations.

There are a lot of other classes in ESDL, but the discussed classes are the most important ones needed to do graphics in ESDL. Most of my work will not need much more than these classes. Please consider other existing ESDL documentations for more information [6, 7].

3.3.2 Drawbacks and Changes in the Existing Design

Even though the existing design is very nice and provides ability to produce animations with bitmap graphics very easy, there are some drawbacks in the existing design. In the following subsections I will present drawbacks that made it difficult to extend ESDL with figures and a component that can zoom and scroll them. Furthermore, I will discuss the changes I made to the existing ESDL design to get rid of these problems.

3.3.2.1 ESDL_SURFACE as Drawing Interface

An ESDL_DRAWABLE directly gets an ESDL_SURFACE as an argument to draw itself onto. This leads to several problems. First of all it is generally not very nice to give any drawable the possibility to access the whole surface it draws itself on. There is no need for a drawable object to have access to all features of the drawing surface, because an ESDL_SURFACE is more than just an object you can draw onto. For example an ESDL_DRAWABLE could change the position of the surface onto which it is supposed to draw (feature *set_x_y*). Or even worse, a drawable could cause the surface to *redraw* itself which would flush the buffered display surface onto screen. There are other features not intended to be used when an ESDL_DRAWABLE draws itself. Since ESDL_SURFACE is supposed to grow, there are surely even more to come.

The major problem with drawable objects drawing themselves directly to an ESDL_SURFACE is, that there is no possibility to draw them elsewhere. You might think that an ESDL_DRAWABLE is not supposed to be drawn anywhere else than on an ESDL_SURFACE, and probably you are right. But what if you want to do coordinate transformations when drawing an ESDL_DRAWABLE inside a zoomed or scrolled component? Either the ESDL_DRAWABLE needs to know that it is zoomed and perform its drawing with transformed coordinates or the drawing interface with which the ESDL_DRAWABLE draws itself needs to do this transformation. As discussed in section 2.3 the latter possibility is the preferred one.

Therefore, the drawing interface passed as an argument to the feature *draw* of an ESDL_DRAWABLE needs to be able to transform the coordinates passed to its drawing commands before performing the

drawing. There are three possibilities to introduce such a coordinate transformation into all drawing commands used by ESDL_DRAWABLE :

1. Let ESDL_SURFACE itself do the coordinate transformations in all drawing commands. This would lead to a quite overloaded ESDL_SURFACE class text that is already difficult to maintain.

2. Introduce a new class inheriting from ESDL_SURFACE, let us call it ESDL_TRANSFORMATION_SURFACE for example. This class would have to redefine all drawing commands of ESDL_SURFACE to perform them with transformed coordinates. The major problem with this solution is, that transformed drawing would only be possible on such an instance of ESDL_TRANSFORMATION_ SURFACE. Zoomed drawing would not be possible on an ESDL_SURFACE directly. Aside from that, there would probably be a lot of code duplication because all drawing commands would have to be redefined in ESDL_TRANSFORMATION_SURFACE.

3. Let ESDL_DRAWABLE draw onto something else than directly onto an ESDL_SURFACE. We would have to introduce a new class, for example called ESDL_DRAWING_INTERFACE. This class defines the interface that all ESDL_DRAWABLEs can use to draw them selves. The *draw* feature of an ESDL_DRAWABLE would then take such an ESDL_DRAWING_INTERFACE as argument instead of an ESDL_SURFACE. This solution has the advantage that we can define exactly which features an ESDL_DRAWABLE is allowed to use to draw itself. There would be no problem any more to provide an implementation of ESDL_DRAWING_INTERFACE that performs all its drawing commands on an ESDL_SURFACE with some coordinate transformations. It would even be possible to provide an implementation that draws anywhere else. The only problem with this solution is, that we would have to change the design of the existing ESDL library.

Even though the last approach needs changes in the existing design, I decided to choose this solution, because it seems to be very powerful. You will find more information on this new drawing interface in section 3.4.

3.3.2.2 Position as Changeable Attributes in ESDL_DRAWABLE

Another problem with the existing ESDL design is that it is mainly designed for bitmap graphics. The design does not pay attention to the possibility of drawable objects that might be built of something else than bitmap images, e.g. vector-based figures like polygons or polylines. To realize this circumstance consider the following part of the ESDL_DRAWABLE class:

```
x: INTEGER
        -- The distance of 'current' to the coordinate origin
        -- in 'x' (left -> right) direction

y: INTEGER
        -- The distance of 'current' to the coordinate origin
        -- in 'y' (top -> down) direction

set_x (an_integer: INTEGER) is
        -- Sets 'x'
    do
        x := an_integer
    end

set_y (an_integer: INTEGER) is
        -- Sets 'y'
    do
        y := an_integer
    end
```

```
set_x_y (an_x: INTEGER; an_y: INTEGER) is
        -- Sets the 'x' and 'y' value
    do
        x := an_x
        y := an_y
    end
```

Listing 3.3: Position attributes and set features in ESDL_DRAWABLE

The problem with this class is that it already assumes that the position of every drawable object is an attribute that can be changed. This is no problem for bitmap graphics. As their position is only defined through one point (the upper left corner), it is quite easy to move them by changing the attributes x and y. But for vector graphics it is a little bit different. Usually a vector graphic is defined through a lot of points. For example a polyline consists of many points that are connected by line segments. Changing the position of such a polyline would mean to change the position of all its points. Instead of moving all points, we could also define the x and y attributes of such a figure as offset coordinate for all points it is built of. But this would complicate the work with such figures, since the coordinates of the figure points would not be absolute anymore but relative. Moving all points of the figure when the position is changed is acceptable when we do not care about the performance of this operation .

The much bigger problem is that x, y, *width* and *height* are supposed to define the area where the figure is drawn. This means for a polyline that all points are supposed to be inside this rectangle. Usually when you work with vector based figures you do not want to care about this bounding box. A user wants to define the points that define a figure and nothing more. For a polyline this means that whenever the points are changed, the bounding box (x, y, *width* and *height*) needs to be adapted accordingly. Since x and y are already defined as attributes in the deferred class ESDL_DRAWABLE, our figure implementations have no chance to implement x and y as functions that return the minimum coordinates of all points (Eiffel does not allow redefinition of attributes as functions). All figure implementations need to be aware of this and need to update the x and y attributes whenever their points are changed.

3.3.2.3 Changing Object Positions for Drawing

A very important class in ESDL is ESDL_DRAWABLE_CONTAINER. This class is a linked list of ESDL_DRAWABLEs. All contained drawables will be drawn relative to the position of the container. Let us briefly consider the *draw* implementation of this class:

```
draw (a_surface: ESDL_SURFACE) is
        -- Draws 'current' to 'a_surface'
    local
        cursor: DS_LINKED_LIST_CURSOR [G]
    do
        create cursor.make (current)
        from
            cursor.start
        until
            cursor.off
        loop
            cursor.item.set_x_y (cursor.item.x + x, cursor.item.y + y)
            cursor.item.draw (a_surface)
            cursor.item.set_x_y (cursor.item.x - x, cursor.item.y - y)
            cursor.forth
        end
    end
```

Listing 3.4: Implementation of feature draw in class ESDL_DRAWABLE_CONTAINER

This implementation excessively moves the contained objects to draw them at the right position and then moves them back again. Obviously this is not a very nice implementation for drawing objects at some position. Drawing a container of objects should not change the state of the objects (even not temporarily). For large polylines this implementation could become a performance issue, since moving a polyline involves moving all the points it consists of. This is another argument why a container should not move its objects for drawing them and why the deferred class ESDL_DRAWABLE should not define all drawables as movable.

I did not remove the features for changing the position of ESDL_DRAWABLE. It would have needed too many changes in the existing library. Particularly all existing ESDL examples rely on the possibility to move drawable objects. But I changed all those classes that moved the drawable child objects temporarily for drawing, like ESDL_DRAWABLE_CONTAINER did. Now they move the coordinate system instead (see section 3.4.2).

3.3.2.4 Feature *draw_part* in ESDL_DRAWABLE

There is another feature in the interface of the deferred class ESDL_DRAWABLE we need to discuss. All descendants of ESDL_DRAWABLE have to implement the deferred feature *draw_part* that is supposed to draw a specified rectangular part of an ESDL_DRAWABLE:

```
draw_part (rect: ESDL_RECT; a_surface: ESDL_SURFACE) is
        -- Draws rectangular part of 'current' defined by 'rect' to 'a_surface'
    require
        a_rect_not_void: rect/=void
        a_surface_not_void: a_surface/=void
    deferred
    end
```

Listing 3.5: draw_part feature in ESDL_DRAWABLE

Of course it is very useful to have the ability to draw a part of an ESDL_DRAWABLE. For example we could compose a picture from parts of other ESDL_DRAWABLEs, e.g. as in ESDL_TILE_PATTERN. But on the other hand it makes the implementation of descendants of ESDL_DRAWABLE much more complicated, because every descendant class has to implement this feature. As long as drawable objects are only composed of bitmap images, which was the case so far, this is not very difficult. But as soon as our drawable objects might also be composed of figures, an implementation of *draw_part* might not be that simple. Drawing a part of a polygon for example is much more complex than copying a rectangular part out of a bitmap area.

Anyway, the question is, if it is really necessary to have a feature in all drawable objects for drawing a part of them? I would say: No! Usually a graphical system provides the ability to set some clipping area and to draw anything into it. Everything outside this clipping area would then simply not be drawn. Like this it is easily possible to draw only parts of all drawable objects without any need for all descendants of ESDL_DRAWABLE to implement their own way of drawing a part of them. This would make it much easier to implement new classes inheriting from ESDL_DRAWABLE, even for students who just started to program, because they only need to implement one feature *draw* that draws the whole object.

Since there is a function in SDL to set a rectangular clipping area for all drawings performed to a surface, I decided to introduce this clipping into the API of ESDL. Then I introduced a default implementation of the *draw_part* feature in the class ESDL_DRAWABLE that sets the clipping accordingly to draw a part of the object using the normal *draw* feature. Descendants of ESDL_DRAWABLE could still redefine the *draw_part* feature, e.g. to provide a more efficient way to draw a part of them. For ordinary drawable objects there would be no need to provide their own implementation of this feature anymore. Instead, all drawable objects should directly consider the clipping rectangle in their implementation of *draw* to avoid drawing parts of them that are not visible.

3.3.2.5 Animation in *draw* Features

Another problem I discovered with ESDL is the way in which animations of drawable objects are implemented. I will explain the problem by discussing the animation of a never ending background in detail.

A never ending background is a drawable object that fills the whole drawing area with a pattern. For example, we could fill the background with a cloudy sky using an image of one single cloud (see figure 3.2).

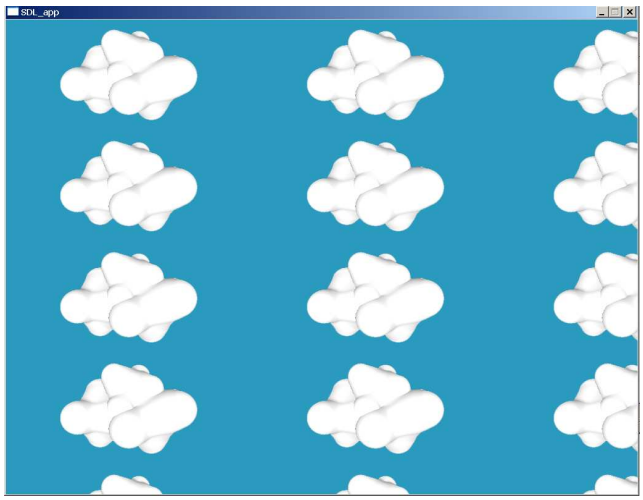

Figure 3.2: Cloudy never ending background

You can think of an ESDL_NEVER_ENDING_BACKGROUND as the kind of background used in old movies when a scene in a car is recorded. It is a large picture that is pulled behind the car. An ESDL_NEVER_ENDING_BACKGROUND is very similar to that concept. It takes an ESDL_DRAWABLE as a pattern to fill the screen. This ESDL_DRAWABLE is glued together over and over again, either in horizontal or vertical direction. Internally ESDL_NEVER_ENDING_ BACKGROUND uses an ESDL_TILE_PATTERN to draw the ESDL_DRAWABLE several times to fill the whole size of the screen (see figure 3.3).

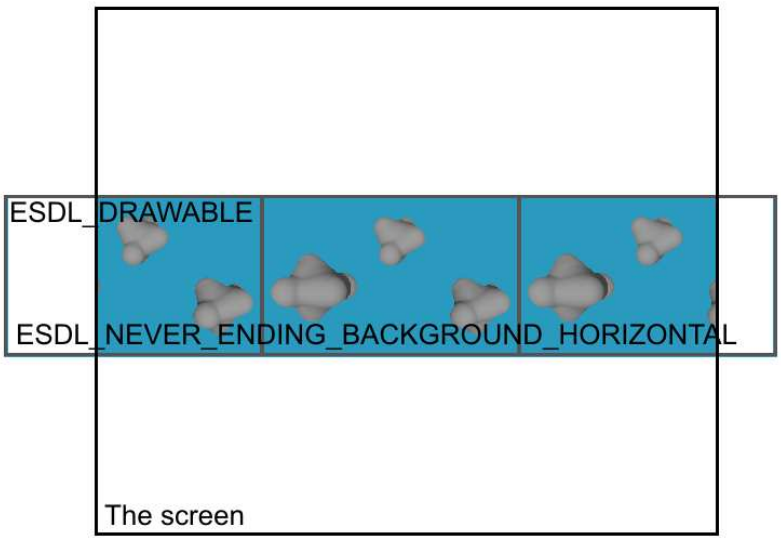

Figure 3.3: ESDL_NEVER_ENDING_BACKGROUND_HORIZONTAL filling the screen horizontally by using an ESDL_TILE_PATTERN_HORIZONTAL to draw an ESDL_DRAWABLE several times.

There are two effective classes inheriting from ESDL_NEVER_ENDING_BACKGROUND.

ESDL_NEVER_ENDING_BACKGROUND_HORIZONTAL uses an ESDL_TILE_PATTERN_HORIZONTAL to glue an ESDL_DRAWABLE together several times to fill the whole width of the screen. ESDL_NEVER_ENDING_BACKGROUND_VERTICAL does the same to fill the whole height of the screen. Figure 3.4 shows the class diagram of the relevant classes.

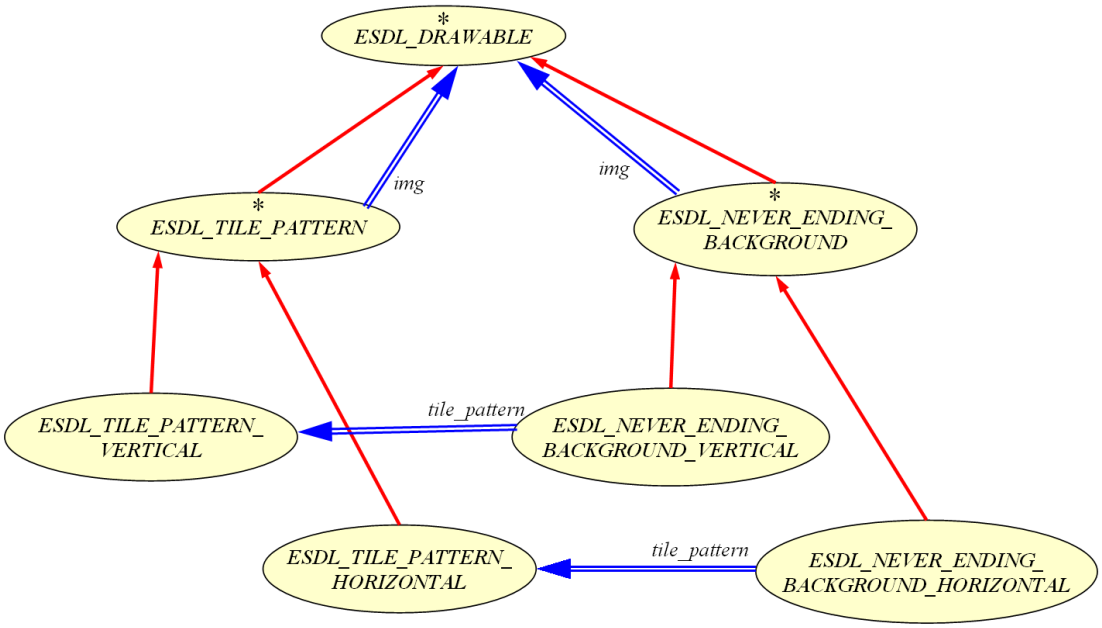

Figure 3.4: Relevant classes for ESDL_NEVER_ENDING_BACKGROUND

We could use an ESDL_NEVER_ENDING_BACKGROUND_HORIZONTAL as the ESDL_DRAWABLE in an ESDL_NEVER_ENDING_BACKGROUND_VERTICAL to fill the whole screen with a pattern (as in Figure 3.2). The code to create this never ending background looks as follows:

```
local
    endless_background: ESDL_NEVER_ENDING_BACKGROUND_VERTICAL
    endless_background_row:
        ESDL_NEVER_ENDING_BACKGROUND_HORIZONTAL
    cloud_image: ESDL_SURFACE
do
    -- Create never ending background to fill screen with clouds.
    create cloud_image.make_from_image ("cloud1.gif")
    create endless_background_row.make (cloud_image)
    create endless_background.make (endless_background_row)

    -- Extend scene container to draw 'endless_background'
    scene.extend (endless_background)
end
```

Listing 3.6: Creation of a never ending background filling the whole screen with clouds

ESDL_NEVER_ENDING_BACKGROUND also has a feature to set the speed for animating the background. It will then move either in vertical or horizontal direction, as the background in the movie with the car scene would. It is possible to let our clouds move into any possible direction by setting the speed of *endless_background* and *endless_background_row* accordingly.

The animation mechanism of an ESDL_NEVER_ENDING_BACKGROUND is rather simple. Whenever its *draw* feature is called, it calculates the position where the tile pattern has to be drawn from its speed and the current time and then draws the moved tile pattern.

Here comes the problem with this animation mechanism. ESDL_NEVER_ENDING_BACKGROUND_VERTICAL draws itself by drawing its ESDL_DRAWABLE several times below each other. In our example this ESDL_DRAWABLE used as a pattern *for endless_background* is *endless_background_row* which is an ESDL_NEVER_ENDING_BACKGROUND_HORIZONTAL. This means, that the *draw* feature of *endless_background_row* is called several times, once for each row in the background. Every time it is called, it calculates the position where the horizontal pattern needs to be drawn for the animation from current speed and time. Since drawing also needs some time, it is possible that one pattern row of the background is drawn with another current time and therefore at another position than the row above. This can lead to funny results, as in Figure 3.5

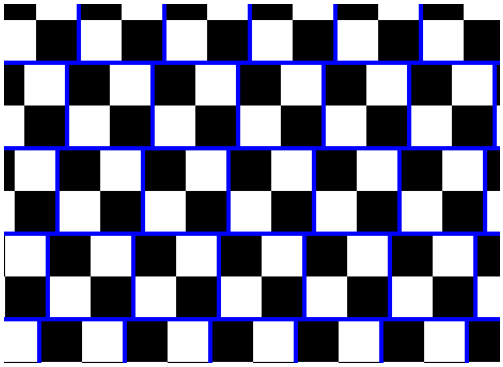

Figure 3.5: Never ending background filled with a checkerboard pattern image (blue squares) and animated with a fast horizontal speed. The rows are displaced relative to each other, because they are not drawn at exactly the same time.

Obviously it is not precise to animate objects by considering the current system time when they get drawn. One drawing pass is supposed to produce a snapshot of all drawables at exactly the same time. Therefore we need a way for all animatable objects to get animated with respect to the same time tick.

Doing the animation of an object together with its drawing is bad design, because drawing should not mean to move an object. I think that the animation mechanism needs to be separated from the drawing mechanism and all animatable objects should get the same time relative to which they all can do their animation.

To achieve this, I introduced a new deferred class ESDL_ANIMATABLE. An ESDL_ANIMATABLE has a deferred feature *go_to_time* that takes a time in milliseconds as argument.

```
deferred class
    ESDL_ANIMATABLE

feature -- Basic operations

    go_to_time (a_time: INTEGER) is
            -- Change 'Current's object state to
            -- state at 'a_time' (in milliseconds).
        deferred
        end

end -- class ESDL_ANIMATABLE
```

Listing 3.7: Deferred class ESDL_ANIMATABLE

In the class ESDL_SCENE there will be a new event called *animation_event*. This new event is published right before drawing is done together with a time tick as argument. To ensure that all animatable objects of a scene get animated when the scene is running, we have to subscribe their *go_to_time* feature to this event. I introduced two additional features *start_animating* and *stop_animating*, that make this subscription more convenient:

```
feature -- Animation

    animate
            -- Let all subscribed animatable objects perform their animation.
            -- (Calls 'go_to_time' of animatable objects with current time tick)

    animation_event: EVENT_TYPE [TUPLE [INTEGER]]
            -- Animation event, allows animatable objects to perform animation
            -- (i.e. moving them selves) before they get drawed.
            -- As an argument the reference time in milliseconds is passed
            -- up to which the animatable objects should draw them selves.
            -- This event gets published right before the scene is redrawed.

    start_animating (an_animatable: ESDL_ANIMATABLE)
            -- Subscribe 'an_animatable' to be animated when 'Current' is running.

    stop_animating (an_animatable: ESDL_ANIMATABLE)
            -- Unsubscribe 'an_animatable' from beeing animated when 'Current' is running.
```

Listing 3.8: Animation features in ESDL_SCENE

Like this, subscribed animatable objects will be animated right before being drawn and they will all get the same time tick to perform animation. Subsection 3.10.2.2 discusses this new animation mechanism in more detail.

3.4 Drawing Interface

As discussed in section 3.3.2.1, one of the major changes to the existing design is that drawable objects will not draw directly onto an ESDL_SURFACE anymore. Instead, I introduced a new deferred class ESDL_DRAWING_INTERFACE that drawable objects take as argument to draw themselves with. There is an effective descendant of this class, called ESDL_SURFACE_DRAWER that can draw onto an ESDL_SURFACE. Since all existing ESDL examples assume that ESDL_DRAWABLEs can be drawn directly onto an ESDL_SURFACE, I also let ESDL_SURFACE inherit from ESDL_SURFACE_DRAWER. One could think that this looks like a workaround. But we could also compare it to the concept of data structures in EiffelBase- or Gobo-libraries[5, 12], where LINKED_LIST is not only a data structure but also an iterator to traverse the data items inside of it. This is very easy to use, since we can directly iterate over a list without any need to create an iterator first. With ESDL_SURFACE and ESDL_SURFACE_DRAWER it is quite the same. Since ESDL_SURFACE inherits from ESDL_ SURFACE_DRAWER, we can directly draw onto an ESDL_SURFACE. It is also possible to create other ESDL_SURFACE_DRAWER instances drawing on the same surface, but maybe with another coordinate system.

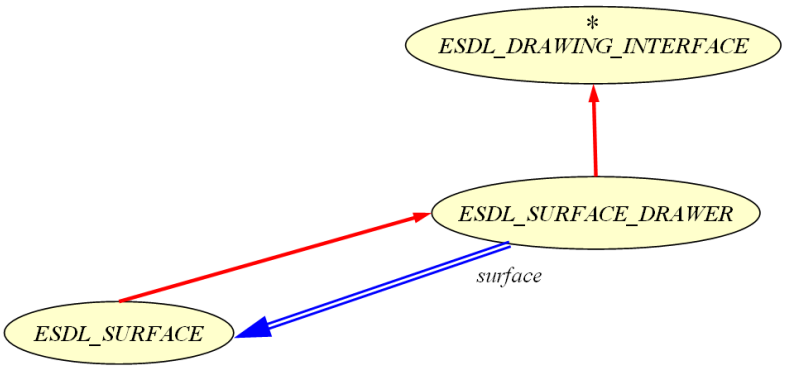

Figure 3.6: Effective descendants of ESDL_DRAWING_INTERFACE

The following subsections will discuss the features of this very important class ESDL_DRAWING_INTERFACE in detail. All classes inheriting from ESDL_DRAWABLE are supposed to use this interface to draw.

3.4.1 Drawing Commands

First of all, ESDL_DRAWING_INTERFACE has a lot of features to perform drawings. Most of these features are deferred. I designed them very carefully with respect to the operand principle (see [13], page 767). All drawing commands have no color or line width as an argument. Such options can be set through separate status setting features instead:

```
deferred class interface
    ESDL_DRAWING_INTERFACE

feature -- Status report

    drawing_color: ESDL_COLOR
            -- Color used to draw

    line_width: DOUBLE
            -- Line width used to draw lines

    is_line_width_scaling_enabled: BOOLEAN
            -- Does 'Current' scale line width,
            -- when coordinates are scaled?
            -- ('True' by default)

    is_line_point_rounding_enabled: BOOLEAN
            -- Are points of polylines and line segments drawed rounded
            -- for nice line endings and joins between polyline segments?

feature -- Status setting

    set_drawing_color (a_color: ESDL_COLOR)
            -- Set 'drawing_color' to 'a_color'.
        require
            a_color_attached: a_color /= Void
        ensure
            drawing_color_set: drawing_color.is_equal (a_color)

    set_line_width (a_width: DOUBLE)
            -- Set 'line_width' to 'a_width'.
        ensure
            line_width_assigned: line_width = a_width

    set_line_width_scaling_enabled (a_bool: BOOLEAN)
            -- Set 'is_line_width_scaling_enabled' to 'a_bool'.
        ensure
            is_line_width_scaling_enabled_set: is_line_width_scaling_enabled = a_bool

    set_line_point_rounding_enabled (a_bool: BOOLEAN)
            -- Set 'is_line_point_rounding_enabled' to 'a_bool'.
        ensure
            is_line_point_rounding_enabled_set: is_line_point_rounding_enabled = a_bool

feature -- Drawing commands
```

```
fill (a_color: ESDL_COLOR)
        -- Fill 'coordinate_area' with 'a_color'.
    require
        a_color_not_void: a_color /= Void

draw_point (a_point: VECTOR_2D)
        -- Draw a point at position 'a_point' with width 'line_width' using 'drawing_color'.
        -- Draw exactly one pixel if 'line_width' is 0.0.
    require
        a_point_attached: a_point /= Void

draw_line_segment (point1, point2: VECTOR_2D)
        -- Draw line segment from 'point1' to 'point2' using current 'line_width' and '
            drawing_color'.
    require
        point1_attached: point1 /= Void
        point2_attached: point2 /= Void

draw_rectangle (a_rectangle: ORTHOGONAL_RECTANGLE)
        -- Draw 'a_rectangle' with 'line_width' and 'drawing_color'.
    require
        a_rectangle_not_void: a_rectangle /= Void

fill_rectangle (a_rectangle: ORTHOGONAL_RECTANGLE)
        -- Draw filled rectangle 'a_rectangle' in 'drawing_color'.
    require
        a_rectangle_not_void: a_rectangle /= Void

draw_polyline (points: DS_LINEAR [VECTOR_2D])
        -- Draw line segments between subsequent points in 'points'.
    require
        at_least_two_points: points /= Void and then points.count >= 2

draw_polygon (points: DS_LINEAR [VECTOR_2D])
        -- Draw polygon defined by 'points' with 'line_width' and 'drawing_color'.
    require
        at_least_three_points: points /= Void and then points.count >= 3

fill_polygon (points: DS_LINEAR [VECTOR_2D])
        -- Draw filled polygon defined by 'points' in 'drawing_color'.
    require
        at_least_three_points: points /= Void and then points.count >= 3

draw_surface (a_surface: ESDL_SURFACE)
        -- Draw 'a_surface' into its 'bounding_box' onto 'Current'.
    require
        a_surface_not_void: a_surface /= Void

draw_surface_part (a_surface: ESDL_SURFACE; part_rectangle:
    ORTHOGONAL_RECTANGLE)
        -- Draw part of 'a_surface' specified by 'part_rectangle'
        -- at its original position inside 'a_surface.bounding_box' onto 'Current'.
    require
        a_surface_not_void: a_surface /= Void
        part_rectangle_not_void: part_rectangle /= Void
```

```
    draw_surface_stretched (a_surface: ESDL_SURFACE; a_destination_box:
        ORTHOGONAL_RECTANGLE)
            -- Draw 'a_surface' into 'a_destination_box' onto 'Current', stretch if necessary.
        require
            a_surface_not_void: a_surface /= Void
            a_destination_box_not_void: a_destination_box /= Void

    draw_surface_part_stretched (a_surface: ESDL_SURFACE; part_rectangle:
        ORTHOGONAL_RECTANGLE; a_destination_box: ORTHOGONAL_RECTANGLE)
            -- Draw part of 'a_surface' specified by 'part_rectangle' into 'a_destination_box'
                onto 'Current', stretch if necessary.
        require
            a_surface_not_void: a_surface /= Void
            part_rectangle_not_void: part_rectangle /= Void
            a_destination_box_not_void: a_destination_box /= Void

    draw_object (an_object: ESDL_DRAWABLE)
            -- Draw 'an_object'.
        require
            an_object_not_void: an_object /= Void

invariant
    drawing_color_not_void: drawing_color /= Void
    line_width_not_negative: line_width >= 0

end -- class ESDL_DRAWING_INTERFACE
```

Listing 3.9: Drawing features of class ESDL_DRAWING_INTERFACE

3.4.2 Coordinate System

An ESDL_DRAWING_INTERFACE provides more than drawing commands. It is also a coordinate system that clients can change for performing coordinate transformations when drawing. Like this, it is very easy to draw something scrolled or zoomed. It is also possible to clip the coordinate system for only drawing to a rectangular part of the target drawing surface.

We have to distinguish between at least two coordinate systems. First, we have coordinates that clients of the drawing interface use to draw. I call them *user coordinates*. All coordinates passed as arguments to the drawing commands are considered to be *user coordinates*. But these coordinates do not necessarily have to be the pixel coordinates on screen directly (see also [13], page 1070). A drawing interface has to transform these *user coordinates* to *device coordinates*, the pixel coordinates on the target drawing surface (e.g. the screen).

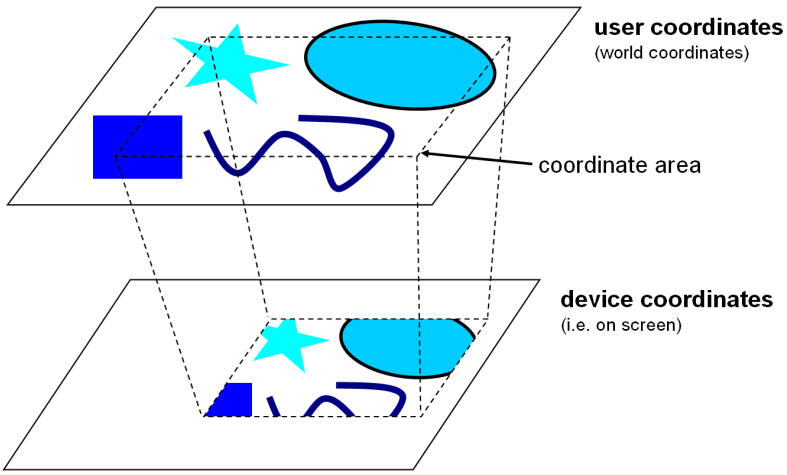

Figure 3.7: Coordinate area in user coordinates, clipped and projected into device coordinates.

ESDL_DRAWING_INTERFACE defines the user coordinate system through a feature *coordinate_area*. This rectangle defines which part of the user coordinate space gets drawn to the target drawing surface. All coordinates inside this rectangle are projected to *device coordinates*, e.g. to pixel coordinates on an ESDL_SURFACE. Clients are allowed to use coordinates outside this area in all drawing commands. But all drawings will be clipped to this *coordinate_area* on the graphical output device. Nevertheless, clients should try to avoid drawing things that are outside this area for better performance.

```
feature -- Coordinate System

    coordinate_area: ORTHOGONAL_RECTANGLE
            -- Coordinates inside which all drawing primitives can draw,
            -- coordinates outside this area will be clipped
            -- (usefull to avoid drawing objects, that are anyway off-screen),
            -- 'coordinate_area' can be changed using 'transform_coordinates',
            -- 'translate_coordinates' and 'clip_coordinates'.
        ensure
            result_not_void: Result /= Void

    clip_coordinates (an_area: ORTHOGONAL_RECTANGLE)
            -- Clip 'coordinate_area' to 'an_area'.
            -- Usefull to restrict drawing to 'an_area'
            -- and to just not draw anything outside.
            -- (Reset by calling again with backed up old 'coordinate_area'
            -- when done clipping)
        require
            an_area_not_void: an_area /= Void

    translate_coordinates (a_distance: VECTOR_2D)
            -- Draw all following drawing primitives as translated by 'a_distance'.
            -- Usefull for container drawables that want there childs to
            -- be drawed relatively to a new origin-point.
            -- (Reset by calling again with negation of 'a_distance' when done)
        require
            a_distance_not_void: a_distance /= Void


    transform_coordinates (an_area: ORTHOGONAL_RECTANGLE)
            -- Transform 'coordinate_area' to 'an_area'.
            -- Usefull for drawable containers that want there childs to
            -- be drawed scaled (zoomed or stretched).
```

```
        -- (Reset it by calling again with backed up old 'coordinate_area' when done).
    require
        an_area_not_void: an_area /= Void

disable_scaling_coordinates (a_reference_point: VECTOR_2D)
        -- Transform coordinates such as no scaling is done anymore and use
        -- 'a_reference_point' as the point that keeps its position.
        -- (Reset by calling 'transform_coordinates' with backed up
        -- old 'coordinate_area' when done)
    require
        a_reference_point_not_void: a_reference_point /= Void

device_resolution: DOUBLE
        -- Number of pixels/points on device per user coordinate unit
        -- (Usefull for drawing with adopted level of detail
        -- for better performance)
    ensure
        positive_display_resolution: Result > 0

device_point (a_point: VECTOR_2D): VECTOR_2D
        -- Device point where 'a_point' gets drawed to
    require
        a_point_not_void: a_point /= Void
    ensure
        result_calculated: Result /= Void

user_point (x, y: DOUBLE): VECTOR_2D
        -- Point in user coordinates that gets drawed
        -- to device point at 'x' and 'y'
    ensure
        result_calculated: Result /= Void
```

Listing 3.10: Coordinate system features in class ESDL_DRAWING_INTERFACE

There are a few features that allow clients to change the coordinate area. This is helpful when you only want to draw a part of the drawing or if you like to perform drawings that are translated or even scaled. There are two conceptual different ways of changing the coordinate area we have to distinguish:

- **Clipping:** Clipping means to restrict drawing to a rectangular area. Everything outside this area will not be drawn. Therefore, if we clip the coordinate area, it means that we change the area where we can draw to, both in user coordinates as well as in device coordinates (see figure 3.8). This does not change the way the coordinates are transformed. A point in user coordinates that was transformed to the center of the screen before the clipping is still transformed to the center of the screen after clipping. The only difference is which points are drawn and which are not.
- **Transformation:** A transformation changes the way the user coordinates are transformed to device coordinates. The coordinate system of ESDL_DRAWING_INTERFACE supports coordinate transformations through changing the coordinate area. This does not change the area where coordinates will be projected to in device space (see figures 3.10 and 3.12).

I will discuss all possibilities that ESDL_DRAWING_INTERFACE supports to change the *coordinate_area* in the following subsections.

3.4.2.1 Clipping

Clipping means to restrict drawing to a rectangular part of the coordinates. This does not change the way how user coordinates are transformed to device coordinates. When the coordinate area in user space

is clipped, the area on the target surface where the user coordinates get projected to is also changed accordingly.

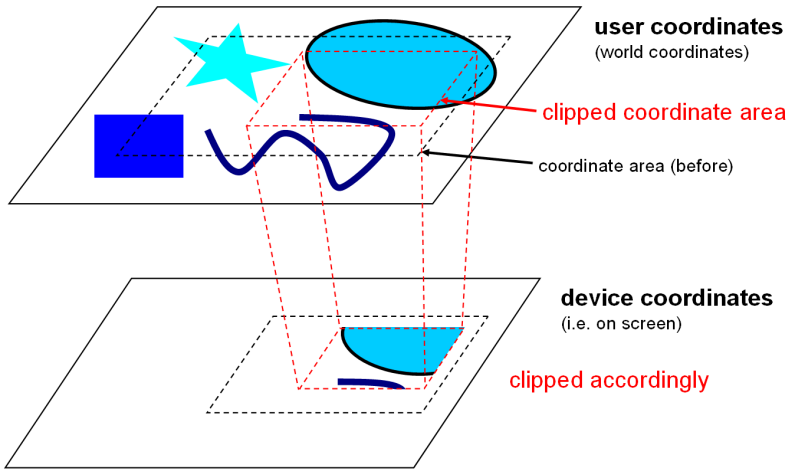

Figure 3.8: Clipped coordinate area

Feature *clip_coordinates* in ESDL_DRAWING_INTERFACE gives clients the ability to change the area to which all drawings are clipped. Using this command they can set a new *coordinate_area* defining the clipping in user coordinates. Have a look at the following little example, clipping the coordinate area to draw a part of a circle:

```
draw (drawing_interface: ESDL_DRAWING_INTERFACE) is
        -- Draw a circle piece using 'drawing_interface'.
    local
        center: VECTOR_2D
        circle: ESDL_CIRCLE
        rectangle: ORTHOGONAL_RECTANGLE
        old_coordinate_area: ORTHOGONAL_RECTANGLE
    do
        -- Create circle object to draw a circle
        -- at center (150, 150) with radius 100.
        create center.make (150, 150)
        create circle.make (center, 100)

        -- Backup current coordinate area before clipping it.
        old_coordinate_area := drawing_interface.coordinate_area

        -- Create rectangular part of circle we want to draw
        -- and restrict it to intersection with current coordinate area.
        create rectangle.make_from_coordinates (40, 40, 150, 150)
        rectangle := rectangle.intersection (old_coordinate_area)

        -- Clip coordinate area to rectangular area we want to draw.
        drawing_interface.clip_coordinates (rectangle)

        -- Draw the circle.
        drawing_interface.draw_object (circle)

        -- Reset clipping as it was before.
        drawing_interface.clip_coordinates (old_coordinate_area)
    end
```

Listing 3.11: Feature that draws a piece of a circle using clipping

As you can see in this example, clients should always backup the old coordinate area before clipping it, in order to reset it when done. The client should also ensure not to clip the coordinate area to an area that goes outside of the initial coordinate area that was passed to him.

Figure 3.9: Clipped circle drawn by feature in listing 3.11. Red square represents the clipping rectangle.

3.4.2.2 Translation

The feature *translate_coordinates* enables the user to move the coordinate system. Moving the coordinate system means that all following drawings are performed as if they were moved by the vector passed as argument to *translate_coordinates*. This transformation does not change the area on the target surface where drawings will be performed and thus does not change the clipping rectangle in device space.

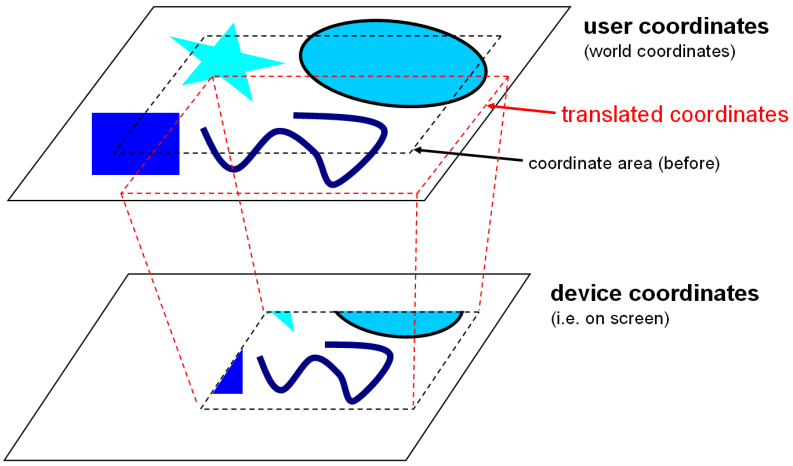

Figure 3.10: Translated coordinate area

This feature is very helpful, for example to implement containers of drawable objects that want their contained objects to be drawn as if coordinate (0, 0) was in the top left corner of the container.

The following little example demonstrates how to use the feature *translate_coordinates*:

```
draw (drawing_interface: ESDL_DRAWING_INTERFACE) is
        -- Draw translated rectangle using 'drawing_interface'.
    local
        rectangle: ORTHOGONAL_RECTANGLE
        translation: VECTOR_2D
    do
        -- Draw a red rectangle before translating the coordinate system.
        drawing_interface.set_drawing_color (red)
        create rectangle.make_from_coordinates (100, 100, 200, 200)
        drawing_interface.draw_rectangle (rectangle)

        -- Translate coordinate system.
        create translation.make (50, 25)
```

```
        drawing_interface.translate_coordinates (translation)

        -- Draw the same rectangle again, in blue.
        drawing_interface.set_drawing_color (blue)
        drawing_interface.draw_rectangle (rectangle)

        -- Retranslate coordinates to old state.
        drawing_interface.translate_coordinates (- translation)
    end
```

Listing 3.12: Example demonstrating use of feature translate_coordinates to draw a rectangle translated

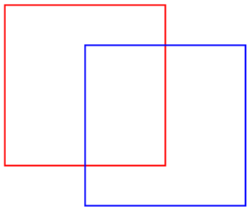

Figure 3.11: Rectangles drawn by feature in Listing 3.12

3.4.2.3 Scaling

Last but not least there is the possibility to transform the coordinate system such that it even gets scaled. The feature *transform_coordinates* sets a new coordinate area for all following drawing commands. This gives us all possibilities to choose a rectangular part of the user coordinate system that is projected onto the target area in device space. For example we could simply choose a bigger coordinate area for zooming out, such that everything fits into the target device space (see Figure 3.12).

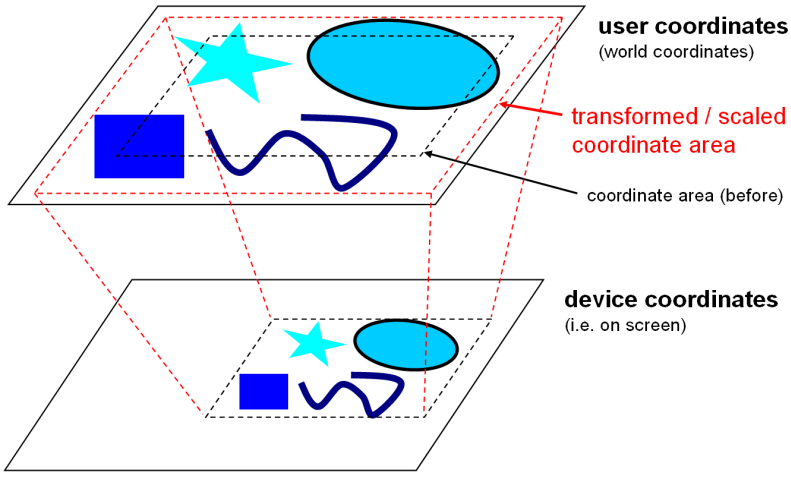

Figure 3.12: Transformed coordinate area

The following example shows how to use the feature *transform_coordinates* to draw something scaled:

```
    draw (drawing_interface: ESDL_DRAWING_INTERFACE) is
            -- Draw transformed square using 'drawing_interface'.
        local
            old_coordinate_area: ORTHOGONAL_RECTANGLE
            new_coordinate_area: ORTHOGONAL_RECTANGLE
            rectangle: ORTHOGONAL_RECTANGLE
        do
```

```
    -- Draw a red square before scaling the coordinate system.
    drawing_interface.set_drawing_color (red)
    create rectangle.make_from_coordinates (200, 200, 400, 400)
    drawing_interface.draw_rectangle (rectangle)

    -- Backup current coordinate area.
    old_coordinate_area := drawing_interface.coordinate_area

    -- Change size of coordinate area and keep center position.
    create new_coordinate_area.make_from_rectangle (old_coordinate_area)
    new_coordinate_area.set_size_centered (
        old_coordinate_area.width * 2,
        old_coordinate_area.height * 3)
    drawing_interface.transform_coordinates (new_coordinate_area)

    -- Draw the same rectangle again, in blue.
    drawing_interface.set_drawing_color (blue)
    drawing_interface.draw_rectangle (rectangle)

    -- Retranslate coordinates to old state.
    drawing_interface.transform_coordinates (old_coordinate_area)
end
```

Listing 3.13: Example demonstrating use of feature transform_coordinates to draw a square scaled

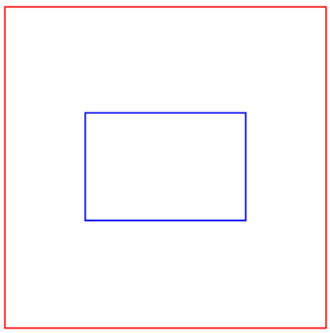

Figure 3.13: Rectangles drawn by feature in Listing 3.13

3.5 Extensions to ESDL_SURFACE

As discussed in subsection 3.3.1, ESDL_SURFACE is probably the most important class in ESDL. It represents a bitmap image in memory. It serves both as a drawing surface onto which we can draw and as an image that can be drawn onto other drawing surfaces. To implement an effective drawing interface that draws onto an ESDL_SURFACE as described in section 3.4, we need features in ESDL_SURFACE that support this. So far, there were only three features in ESDL_SURFACE that supported drawing onto an ESDL_SURFACE bitmap image: *blit_surface*, *blit_surface_part* and *draw_pixel*. The first two features, *blit_surface* and *blit_surface_part* copy an ESDL_SURFACE or part of it onto the target surface. The feature *draw_pixel* sets the color value of one specific pixel.

To implement all drawing commands in the class ESDL_SURFACE_DRAWER efficiently, these features are not sufficient. Of course we could implement our own polygon filling algorithm and call the feature *draw_pixel* for each pixel that needs to be drawn. But probably this approach would be quite slow. There are drawing primitives in SDL, like line segments, polygons and polylines. These C functions

have already been wrapped into ESDL using EWG [14]. I introduced these drawing primitives into the object-oriented API of ESDL to make them usable.

Furthermore, there were missing some other important features in ESDL_SURFACE. There was no clipping mechanism present yet and there was no possibility to zoom an ESDL_SURFACE bitmap. Both features are necessary when we want to implement all coordinate transformations as discussed in subsection 3.4.2.

All extensions that I added to the class ESDL_SURFACE are described in the following subsections.

3.5.1 Drawing Primitives

Introducing drawing primitives for drawing lines, rectangles, circles, polygons and so on onto a bitmap graphic seems to be an easy task. Especially when we have already some wrapped C functions available from SDL, that perform exactly these tasks (see EWG generated class SDL_GFXPRIMITIVES_FUNCTIONS). But there where some issues I had to take care of, when introducing these features into the object-oriented API of ESDL.

First of all I had to decide how to name these new features. Because ESDL_SURFACE will also inherit all drawing commands of ESDL_SURFACE_DRAWER as discussed in section 3.4, there are already some feature names starting with "draw_". But those features have a different meaning. The features inherited from ESDL_SURFACE_DRAWER do not directly draw to pixel coordinates of the ESDL_SURFACE. These features are supposed to do coordinate transformations first, before drawing onto the pixels of the surface using the drawing primitives that we want to introduce now. Therefore I had to choose different names for the drawing primitives. The names should express that the features directly manipulate the pixels of the surface. I decided to choose names starting with "put_" since these features simply put some values into the bitmap memory area. To avoid confusion between the old feature *draw_pixel* and all other drawing commands inherited from ESDL_SURFACE_DRAWER, I set the old feature *draw_pixel* to obsolete with an appropriate obsolete clause. Clients that use this feature in future will get a compilation warning, telling them to use *put_pixel* instead.

Most of the drawing primitives I introduced call some wrapped SDL functions to perform the appropriate drawing. There is one command that was not that easy to implement, called *put_wide_line_segment*. This feature is able to draw a straight line segment between two points that has a width of more than one pixel. SDL does not support drawing lines of arbitrary width. Since we need to be able to draw wide lines for visualizing the traffic lines, I had to find a possibility to draw lines with an arbitrary line width. I tried several approaches to implement this (see section 3.6). The feature *put_wide_line_segment* draws a rectangular polygon to represent the line segment with the given width.

Following you'll find the interface of all drawing primitives I introduced in the class ESDL_SURFACE:

```
put_pixel (x_pos, y_pos: INTEGER; a_color: ESDL_COLOR)
        -- Draw a pixel at pixel position 'x_pos', 'y_pos'
        -- with color 'a_color' onto 'Current'.
    require
        a_color_not_void: a_color /= Void
    ensure
        surface_operation_successful or else last_error = error_put_pixel

put_line_segment (x1, y1, x2, y2: INTEGER; color: ESDL_COLOR)
        -- Draw a line segment from pixel 'x1', 'y1'
        -- to pixel 'x2', 'y2' with 'color' onto 'Current'.
    ensure
        surface_operation_successful: surface_operation_successful or else last_error =
            error_put_line_segment

put_wide_line_segment (x1, y1, x2, y2: INTEGER; a_line_width: INTEGER; color:
    ESDL_COLOR)
        -- Draw a line segment from pixel 'x1', 'y1' to pixel 'x2', 'y2'
        -- with 'color' and 'a_line_width' onto 'Current'.
```

```
        require
            color_not_void: color /= Void
        ensure
            surface_operation_successful: surface_operation_successful or else last_error =
                error_put_line_segment or else last_error = error_put_polygon_filled

    put_rectangle (x1, y1, x2, y2: INTEGER; color: ESDL_COLOR)
            -- Draw a rectangle from pixel 'x1', 'y1'
            -- to pixel 'x2', 'y2' with 'color' onto 'Current'.
        require
            color_not_void: color /= Void
        ensure
            surface_operation_successful: surface_operation_successful or else last_error =
                error_put_rectangle

    put_rectangle_filled (x1, y1, x2, y2: INTEGER; color: ESDL_COLOR)
            -- Draw a filled rectangle from pixel 'x1', 'y1'
            -- to pixel 'x2', 'y2' filled with 'color' onto 'Current'.
        require
            color_not_void: color /= Void
        ensure
            surface_operation_successful: surface_operation_successful or else last_error =
                error_put_rectangle_filled

    put_circle (center_x, center_y: INTEGER; radius: INTEGER; color: ESDL_COLOR)
            -- Draw a circle around center at 'center_x' and 'center_y'
            -- with 'radius' and 'color' onto 'Current'.
        require
            color_not_void: color /= Void
        ensure
            surface_operation_successful: surface_operation_successful or else last_error =
                error_put_circle

    put_circle_filled (center_x, center_y: INTEGER; radius: INTEGER; color: ESDL_COLOR)
            -- Draw a filled circle around center at 'center_x' and 'center_y'
            -- with 'radius' filled with 'color' onto 'Current'.
        require
            color_not_void: color /= Void
        ensure
            surface_operation_successful: surface_operation_successful or else last_error =
                error_put_circle_filled

    put_polygon (x_coordinates, y_coordinates: ESDL_INTEGER_ARRAY; count: INTEGER;
        color: ESDL_COLOR)
            -- Draw polygon defined by first 'count' points
            -- in 'x_coordinates' and 'y_coordinates'
            -- with 'color' onto 'Current'.
        require
            x_coordinates_not_void: x_coordinates /= Void
            y_coordinates_not_void: y_coordinates /= Void
            enough_x_coordinates: x_coordinates.count >= count
            enough_y_coordinates: y_coordinates.count >= count
        ensure
            surface_operation_successful: surface_operation_successful or else last_error =
                error_put_polygon

    put_polygon_filled (x_coordinates, y_coordinates: ESDL_INTEGER_ARRAY; count:
```

```
    INTEGER; color: ESDL_COLOR)
      -- Draw filled polygon defined by first 'count' points
      -- in 'x_coordinates' and 'y_coordinates'
      -- filled with 'color' onto 'Current'.
  require
    at_least_3_points: count >= 3
    x_coordinates_not_void: x_coordinates /= Void
    y_coordinates_not_void: y_coordinates /= Void
    enough_x_coordinates: x_coordinates.count >= count
    enough_y_coordinates: y_coordinates.count >= count
  ensure
    surface_operation_successful: surface_operation_successful or else last_error =
        error_put_polygon_filled
```

Listing 3.14: New drawing primitives in class ESDL_SURFACE

As you can see, there is a special class ESDL_INTEGER_ARRAY in the interface of some drawing primitives (e.g. *put_polygon*). I had to introduce this class in the ESDL utility cluster, because there was no appropriate class in EWG yet that supported C compatible arrays with 16 bit integers. I know that it is not very nice to expose such low level classes in an interface of a feature. But on the other hand drawing something onto an SDL surface is quite a low level thing. Most clients are supposed to use the features of ESDL_DRAWING_INTERFACE to draw onto an ESDL_SURFACE where they don't have to care about these low level integer arrays. Of course the drawing primitives could also take normal arrays as arguments instead and then convert them to C compatible 16 bit integer arrays before passing them to SDL. But this would slow down the drawing primitives a lot. ESDL_INTEGER_ARRAY supports a very convenient way to to directly pass C compatible arrays.

3.5.2 Anti Aliasing

SDL also has support for drawing primitives with anti aliasing. For most of the drawing primitive functions in SDL there is an appropriate counterpart that performs the same drawing with anti aliasing. To give clients the ability to choose between using this anti aliasing or not, I introduced some status features in ESDL_SURFACE:

```
  is_anti_aliasing_enabled: BOOLEAN
        -- Is anti aliasing enabled for converting 'Current' and for drawing onto it?

  set_anti_aliasing_enabled (a_bool: BOOLEAN)
        -- Set 'is_anti_aliasing_enabled' to 'a_bool'.
    ensure
      is_anti_aliasing_enabled_set: is_anti_aliasing_enabled = a_bool
```

Listing 3.15: New anti aliasing status features in class ESDL_SURFACE

Like this clients can choose if all drawings to an ESDL_SURFACE should be done using anti aliasing or not. Per default anti aliasing is set to the same setting as the video surface (see *video_subsystem* in class ESDL_SHARED_SUBSYSTEMS) when creating an ESDL_SURFACE. For the video surface anti aliasing is enabled per default.

Anti aliasing could slow down your application of course, especially when debugging. For example drawing a polygon with anti aliasing means to fill the polygon first and then to draw an anti-aliased outline around it. SDL does not support to draw a filled anti-aliased polygon in one pass and thus our implementation of *put_polygon_filled* has to call two SDL functions for drawing a filled polygon with anti-aliased border line.

3.5.3 Clipping

SDL also comes with some functions to set a rectangular clipping area for all drawings performed to a SDL surface. I introduced appropriate features in ESDL_SURFACE to set this clipping rectangle:

```
clipping_rectangle: ESDL_RECT
        -- Clipping rectangle. Pixels outside this rectangle won't be drawn.
        -- (see 'coordinate_area' for according rectangle in user coordinates)
    ensure
        result_not_void: Result /= Void

set_clipping_rectangle (a_rectangle: ESDL_RECT)
        -- Set 'clipping_rectangle' to 'a_rectangle'.
    require
        a_rectangle_not_void: a_rectangle /= Void
```

Listing 3.16: New clipping features in class ESDL_SURFACE

3.5.4 Bitmap Transformations

We also need a possibility to scale bitmap graphics to draw them using a scaled user coordinate system (see subsection 3.4). Fortunately SDL has functions to zoom and even to rotate an SDL surface. I introduced according conversion features into ESDL_SURFACE:

```
part (x_pos, y_pos, a_width, a_height: INTEGER): ESDL_SURFACE
        -- Part at position 'x_pos' and 'y_pos'
        -- with size 'a_width' and 'a_height'
        -- inside 'Current'
    require
        a_width_positive: a_width > 0
        a_height_positive: a_height > 0
    ensure
        result_created: Result /= Void
        transparent_colorkey_transfered: has_transparent_colorkey implies Result.
            transparent_colorkey = transparent_colorkey
        anti_aliasing_as_in_source: Result.is_anti_aliasing_enabled =
            is_anti_aliasing_enabled

zoomed (a_zoom_factor: DOUBLE): ESDL_SURFACE
        -- New surface containing 'Current' zoomed by 'a_zoom_factor'
        -- (anti aliased if 'is_anti_aliasing_enabled')
    require
        positive_zoom_factor: a_zoom_factor > 0
    ensure
        result_created: Result /= Void
        transparent_colorkey_transfered: has_transparent_colorkey implies Result.
            transparent_colorkey = transparent_colorkey
        anti_aliasing_as_in_source: Result.is_anti_aliasing_enabled =
            is_anti_aliasing_enabled

stretched (x_factor, y_factor: DOUBLE): ESDL_SURFACE
        -- New surface containing 'Current' horizontaly stretched by 'x_factor'
        -- and verticaly stretched by 'y_factor'
        -- (anti aliased if 'is_anti_aliasing_enabled')
    require
        positive_factors: x_factor > 0 and then y_factor > 0
    ensure
```

```
        result_created: Result /= Void
        transparent_colorkey_transfered: has_transparent_colorkey implies Result.
            transparent_colorkey = transparent_colorkey
        anti_aliasing_as_in_source: Result.is_anti_aliasing_enabled =
            is_anti_aliasing_enabled

    rotated (angle: DOUBLE): ESDL_SURFACE
        -- New surface containing 'Current' rotated by 'angle' (in degrees)
        -- (anti aliased if 'is_anti_aliasing_enabled')
    ensure
        result_created: Result /= Void
        transparent_colorkey_transfered: has_transparent_colorkey implies Result.
            transparent_colorkey = transparent_colorkey
        anti_aliasing_as_in_source: Result.is_anti_aliasing_enabled =
            is_anti_aliasing_enabled

    transformed (x_stretch_factor, y_stretch_factor, angle: DOUBLE): ESDL_SURFACE
        -- New surface containing 'Current' rotated by 'angle' (in degrees)
        -- and stretched by 'x_stretch_factor' and 'y_stretch_factor'
        -- (anti aliased if 'is_anti_aliasing_enabled')
    require
        positive_factors: x_stretch_factor > 0 and then y_stretch_factor > 0
    ensure
        result_created: Result /= Void
        transparent_colorkey_transfered: has_transparent_colorkey implies Result.
            transparent_colorkey = transparent_colorkey
        anti_aliasing_as_in_source: Result.is_anti_aliasing_enabled =
            is_anti_aliasing_enabled
```

Listing 3.17: New conversion features in class ESDL_SURFACE

3.6 ESDL_SURFACE_DRAWER Implementation

ESDL_SURFACE_DRAWER is the effective descendant of ESDL_DRAWING_INTERFACE that implements all features of this interface to draw onto an ESDL_SURFACE. Please refer to section 3.4 for more informations on the design and the usage of this classes. With the extensions to ESDL_SURFACE as discussed in the previous section 3.5, there is now all functionality available for implementing this class. This chapter will explain implementation issues that might be important for further development of this class.

3.6.1 Clipping in User Coordinate Space

Since clients of the class are allowed to pass coordinates outside the user coordinate area, we need to have some sort of clipping to ensure that nothing gets drawn outside this area. Since ESDL_SURFACE supports clipping now, one might think that this is no problem at all. So did I. But testings showed, that when the drawing commands just blindly transform all incoming user coordinates and then pass these transformed coordinates to the drawing primitives of ESDL_SURFACE there are several problems that might arise:

- The transformed coordinates might get out of 16 bit integer bounds. Of course we could truncate these coordinates to 16 bit integers. But this could mean, that the performed drawing would probably not lead to the expected result. This might lead to ugly drawing results.

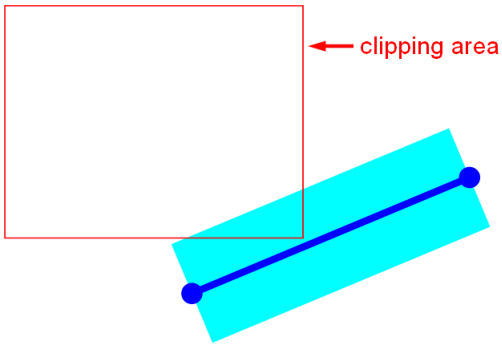

Figure 3.14: Line segment laying outside the clipping area, that is nevertheless partially inside.

- Polygon clipping performed by SDL seems to be rather primitive. Instead of first calculating the clipped polygon from the point coordinates, the polygon is directly passed to the scan-conversion algorithm that draws the polygon pixel per pixel. This means that this algorithm is very slow for polygons that have most of their pixels off-screen.

The only way to solve both problems is to implement a clipping that is done before transforming the user coordinates to device coordinates. If the clipping is done in user coordinates, there is no problem with coordinates that might become infinite when transforming them.

I implemented such a clipping mechanism for all drawing commands in the class ESDL_SURFACE_DRAWER. To make things a little bit easier when clipping lines having a line width, I used a little trick. Instead of clipping line segments (and polylines) exactly to the coordinate area, I used an area that is a little bit bigger. Consider a line segment with points laying outside the coordinate area but with a very big line width (as in Figure 3.14). It might be, that this line needs to be drawn anyway . Since SDL does an exact clipping, it is sufficient if we clip to the enlarged coordinate area by adding a border of the half line width. Like this, there is a little bit too much that gets drawn. But better drawing a bit too much, than drawing not enough.

3.6.2 Drawing Polylines with Line Width

Another thing that was quite difficult to implement, was the drawing of polylines having a line width of more than one pixel. Since SDL does not support this, but we need wide lines to draw the lines in the traffic map, I had to find a way to implement this. The first approach that I implemented did not work perfectly. I will shortly explain this approach and why it can not work properly. I hope that this will prevent anyone who wants to improve the final implementation from running into the same problems again. Afterwards I will shortly explain the finally chosen solution.

The first approach I tried was to calculate a polygon representing the polyline with its line width as in Figure 3.15.

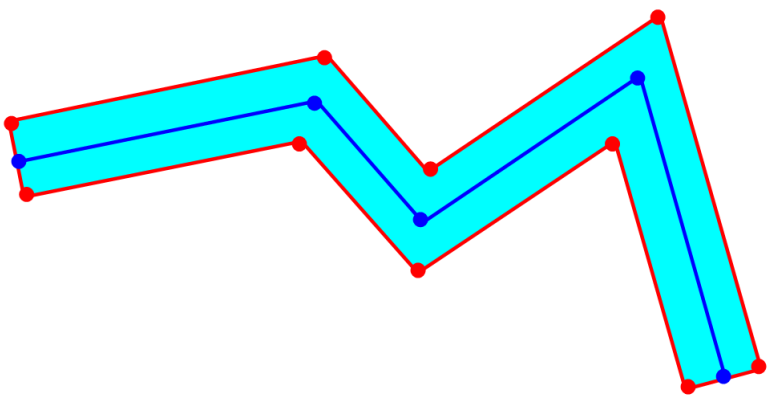

Figure 3.15: Polyline (blue points) with a wide line width drawn as a filled polygon (red points).

Using this approach the joints appear as quite nice closed pointed corners. The algorithm has to calculate two polygon points for each point of the polyline. This points can be calculated by translating the two adjacent line segments accordingly and intersecting them on both sides of the line segments.

I implemented this approach and it first seemed to work as expected. But later on I found out that there are cases where this approach does not work. Consider the case in Figure 3.16 with a very acute angle and a wide line width. As you can see the intersection points are very far away, possibly even farther than the length of the two adjacent line segments. In this case at least one of the two intersection points is not useful at all to draw this polyline as a polygon.

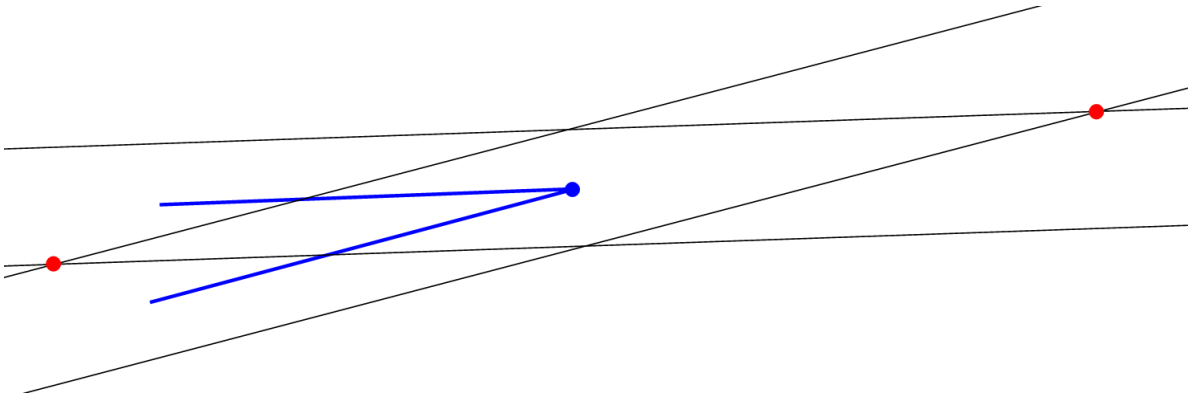

Figure 3.16: Construction of the two polygon points at an acute angle of a wide polyline

I finally had to discard this approach because the problem to find two accurate points for drawing the polyline as one polygon seems to be undecidable in some cases. The advantage of this approach would have been that we could have drawn the whole polyline as one polygon only needing one call to an SDL function. This probably would have been faster than the approach I finally chose.

The solution I used instead does not lead to ugly artifacts as the discussed approach did in some cases. I draw the polyline segment per segment, each as a simple rectangle.

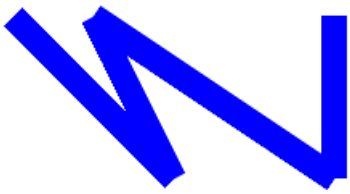

Figure 3.17: Polyline segments drawn as rectangles.

To draw the joints where two segments meet in a nice manner, I draw a circle with the same diameter as the line width over each point of the polyline.

Figure 3.18: Polyline with circles over each point to draw joints rounded

Like this all corners of the polyline are drawn rounded. Drawing a circle over each point of a polyline means some performance loss. Therefore clients can decide if they want to turn off this rounded drawing of the points of a polyline (see feature *set_line_point_rounding_enabled* in class ESDL_SURFACE_INTERFACE).

3.7 Figure Classes

I introduced classes to represent the most important figures, like rectangles, circles, polygons and polylines that are needed to create drawable objects for the places and lines in the traffic map. These figure classes also inherit from ESDL_DRAWABLE as they can draw themselves using the drawing commands of an ESDL_DRAWING_INTERFACE.

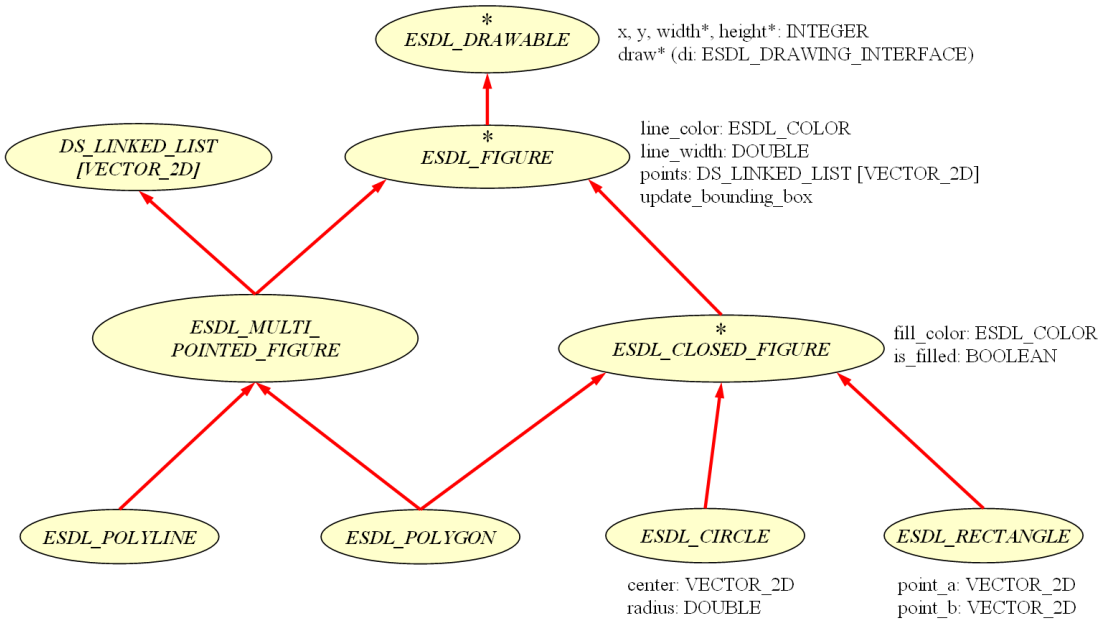

Figure 3.19: Figure classes in ESDL

There is a base class for all figures, called ESDL_FIGURE. A figure is always defined through one or more points and has a *line_color* and a *line_width* used to draw the figure. The position and the bounding box of a figure is normally defined through the points a figure is built of. Since the features x and y of ESDL_DRAWABLE are defined as attributes and are considered as the top left corner of the area inside which an object gets drawn, our figure implementations need to take care to update this attributes whenever the points get changed. Therefore ESDL_FIGURE has a feature *update_bounding_box* that ensures to update x, y, *width* and *height* according to the points the figure is built of. This issue has already been discussed in section 3.3.2.2. All descendants of ESDL_FIGURE should call *update_bounding_box* whenever the points they are defined through get changed.

Further there are two descendant classes of ESDL_FIGURE that serve as base classes for effective figure classes. ESDL_CLOSED_FIGURE is an abstraction for all figures that are closed, like circles and polygons. It is possible to fill this figures with a *fill_color*. The class ESDL_MULTI_POINTED_FIGURE is a base class for all figures that are defined through a list of points, like polylines and polygons. This class also inherits from DS_LINKED_LIST [VECTOR_2D]. Multi pointed figures are easy to build, by simply adding point vectors to them. ESDL_MULTI_POINTED_FIGURE is already an effective class since I introduced a default implementation of the *draw* feature that draws all contained points as points.

Finally there are the effective figure classes. I introduced ESDL_POLYLINE, ESDL_POLYGON, ESDL_CIRCLE and ESDL_RECTANGLE. Since a polygon is both, a list of points and a closed figure, it inherits from ESDL_CLOSED_FIGURE and ESDL_MULTI_POINTED_FIGURE.

Even though most school book examples for inheritance suggest to let the class of rectangles inherit from the class of polygons, ESDL_RECTANGLE does not inherit from ESDL_POLYGON. From a pure mathematical point of view, a rectangle might be considered as a special case of a polygon. From an object-oriented point of view this is only partially correct. A rectangle does not support the whole interface of a polygon. For example it is possible to move one vertex of a polygon or even to add one, and it is still a polygon. But in a rectangle it is not allowed to add or remove a point. Therefore I

let ESDL_RECTANGLE only inherit from ESDL_CLOSED_FIGURE, which defines all features an ESDL_POLYGON and an ESDL_RECTANGLE have in common.

Most of this classification has been adopted from the EV_FIGURE classes in EiffelVision. There is also a class for closed figures and one for multi pointed figures in EiffelVision. EV_RECTANGLE does also not inherit from EV_POLYGON.

For more informations about ESDL_FIGURE and all its descendant classes, I recommend looking at the documentation of the class interfaces.

3.8 Container Classes

The new drawing interface as introduced in section 3.4 gives us now the ability to draw a lot of drawable objects translated to another position without any need to move each object anymore. This makes implementation of container classes like ESDL_DRAWABLE_CONTAINER that draw all contained objects with respect to another coordinate system quite easy. Therefore I changed the implementation of the feature *draw* in ESDL_DRAWABLE_CONTAINER accordingly.

Since the new drawing interface also supports scaling of the coordinate system, I was able to introduce another container class that can even zoom or scroll its content. This new class is called ESDL_ZOOMABLE_CONTAINER and inherits from ESDL_DRAWABLE_ CONTAINER.

In the following subsections I will discuss the new implementation of ESDL_DRAWABLE_CONTAINER and the new class ESDL_ZOOMABLE_CONTAINER that can zoom and scroll its content.

3.8.1 ESDL_DRAWABLE_CONTAINER

An ESDL_DRAWABLE_CONTAINER defines a new coordinate system for its contained drawable objects. The content of such a container gets drawn relatively to the top left corner of the container (see Figure 3.20). Using this class it is easy to build composite objects of ESDL_DRAWABLEs.

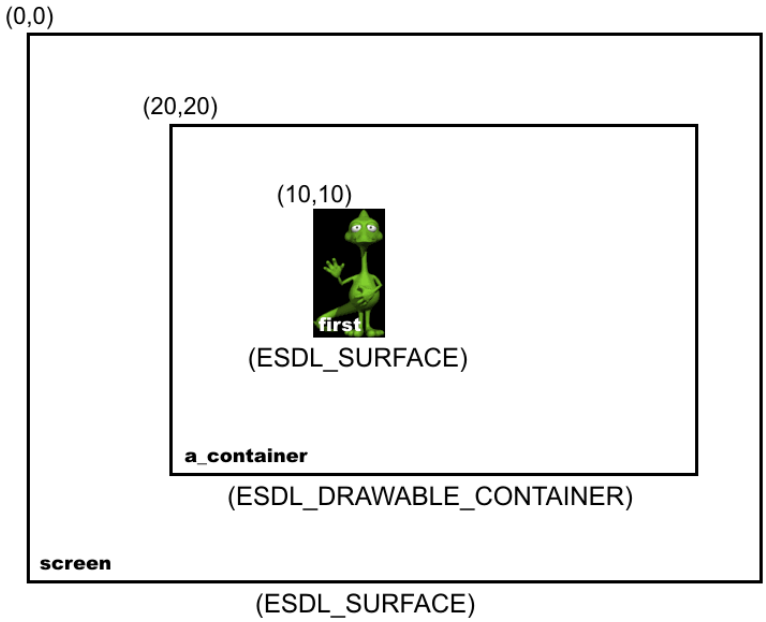

Figure 3.20: Image with position (10, 10) inside an ESDL_DRAWABLE_CONTAINER with position (20, 20)

As discussed in section 3.3.2.3 the implementation of ESDL_DRAWABLE_CONTAINER used to move its contained objects to draw them at the right position. Since the new drawing interface provides

the possibility to translate the coordinate system we are drawing to, there is no need anymore to move the contained objects for drawing them at the right position. Therefore I changed this implementation such as to translate the coordinate system for drawing all contained objects accordingly:

```
draw (drawing_interface: ESDL_DRAWING_INTERFACE) is
        -- Draw 'Current' using 'drawing_interface'.
    local
        cursor: DS_LINKED_LIST_CURSOR [G]
        translation: VECTOR_2D
        old_clipping_area, clipping_area: ORTHOGONAL_RECTANGLE
    do
        -- Translate coordinate system for drawing all contained objects.
        create translation.make (x, y)
        drawing_interface.translate_coordinates (translation)

        -- Change clipping area to clip all objects to container boundaries.
        old_clipping_area := drawing_interface.coordinate_area
        create clipping_area.make_from_coordinates (0, 0, width, height)
        clipping_area := clipping_area.intersection (old_clipping_area)
        drawing_interface.clip_coordinates (clipping_area)

        -- Draw all contained objects.
        cursor := new_cursor
        from
            cursor.start
        until
            cursor.off
        loop
            drawing_interface.draw_object (cursor.item)
            cursor.forth
        end

        -- Reset coordinate system.
        drawing_interface.clip_coordinates (old_clipping_area)
        drawing_interface.translate_coordinates (- translation)
    end
```

Listing 3.18: New implementation of feature draw in ESDL_DRAWABLE_CONTAINER

3.8.2 ESDL_ZOOMABLE_CONTAINER

An ESDL_ZOOMABLE_CONTAINER does not only draw its contained objects relative to its upper left corner as a new origin. An ESDL_ZOOMABLE_CONTAINER also provides the possibility to zoom and scroll its contents.

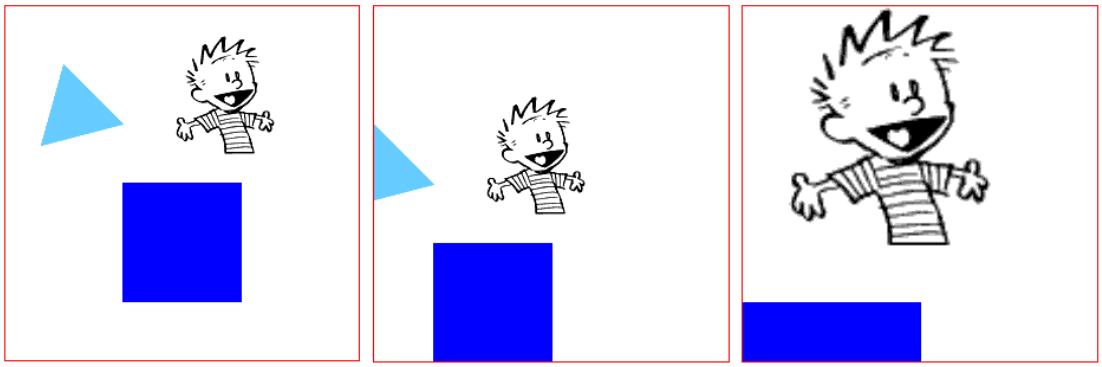

Figure 3.21: Three objects inside an ESDL_DRAWABLE_CONTAINER (red border): as is (left), scrolled (middle) and zoomed (right)

An ESDL_ZOOMABLE_CONTAINER defines its own coordinate system through the feature *visible_area*. This rectangle defines which coordinates of the contained objects are visible. These are the coordinates that finally get projected into the area where the container resides (*bounding_box* of the container defined by *x*, *y*, *width* and *height*). Everything in the container that is outside of this visible area will not be visible or will be clipped to this area. Figure 3.22 visualizes this concept:

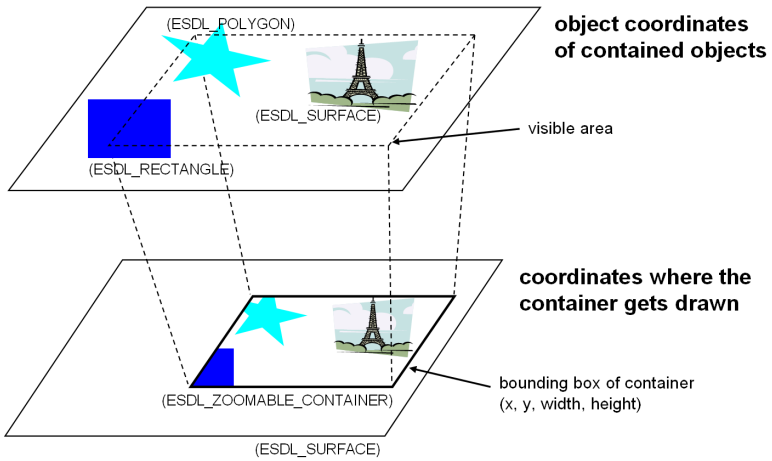

Figure 3.22: Concept of visible area in ESDL_ZOOMABLE_CONTAINER

By changing the *visible_area* of an ESDL_ZOOMABLE_CONTAINER we can define which part of the contained objects we want to draw. Like this we have the possibility to scroll the content (by moving the *visible_area*) or to zoom it (by changing the size of the *visible_area*).

There are several features in ESDL_ZOOMABLE_CONTAINER that allow the user to comfortably change the visible area such that the content gets scrolled or zoomed:

```
feature -- Status report

    visible_area: ORTHOGONAL_RECTANGLE
            -- Area defining coordinate system,
            -- coordinates inside this area will be stretched
            -- into area occupied by 'Current'
            -- ('bounding_box')

    object_area: ORTHOGONAL_RECTANGLE
            -- Area inside which all contained drawable objects
            -- should reside, scrolling is restricted to this area
            -- (scrolling commands ensure to not scroll 'center'
```

```
            -- of 'visible_area' outside of 'object_area')

        zoom_factor: DOUBLE
            -- Factor by which lengths will be scaled

    feature -- Status setting

        calculate_object_area
            -- Calculate 'object_area' from all contained objects
            -- to tightly surround them.

        set_object_area (an_area: ORTHOGONAL_RECTANGLE)
            -- Set 'object_area' to 'an_area'.
        require
            an_area_not_void: an_area /= Void
        ensure
            object_area_set: object_area = an_area

        set_zoom_factor (a_zoom_factor: DOUBLE)
            -- Zoom 'Current' such as 'zoom_factor' is 'a_zoom_factor'.
        require
            zoom_factor_above_minimum: a_zoom_factor >= min_zoom_factor
            zoom_factor_below_maximum: a_zoom_factor <= max_zoom_factor
        ensure
            zoom_factor_set: is_zoom_factor_equal (a_zoom_factor)

    feature -- Commands

        scroll (a_direction: VECTOR_2D)
            -- Scroll 'Current' by 'a_direction'
            -- in object coordinates.
        require
            a_direction_not_void: a_direction /= Void

        scroll_proportional (a_direction: VECTOR_2D)
            -- Scroll 'Current' by 'a_direction'
            -- in transformed coordinates
        require
            a_direction_not_void: a_direction /= Void

        center_on (a_position: VECTOR_2D)
            -- Scroll to have 'a_position' in the center of 'Current'.
        require
            a_position_not_void: a_position /= Void

        zoom (a_zoom_factor: DOUBLE)
            -- Zoom 'Current' by 'a_zoom_factor'.
        require
            a_zoom_factor_is_positive: a_zoom_factor > 0
        ensure
            not_zoomed_over_zoom_max: zoom_factor <= max_zoom_factor +
                zoom_factor_precision
            not_zoomed_over_zoom_min: zoom_factor >= min_zoom_factor -
                zoom_factor_precision

        scroll_and_zoom_to_rectangle (an_area: ORTHOGONAL_RECTANGLE)
            -- Zoom and scroll to fit 'an_area'
```

```
        -- into 'visible_area' (centered)
    require
        an_area_not_void: an_area /= Void
```

Listing 3.19: Most important features to control an ESDL_ZOOMABLE_CONTAINER and its visible area

ESDL_ZOOMABLE_CONTAINER also has the possibility to set an *object_area* to restrict scrolling to some boundaries. Furthermore there is a minimum and a maximum value for the *zoom_factor* that can be set accordingly (not listed in Listing 3.19). Please consider the documentation of the full class interfaces for more information on these features.

3.9 Events

ESDL supports several events like keyboard and mouse events. Clients can subscribe for these events in the class ESDL_EVENT_LOOP. The event classes like ESDL_KEYBOARD_EVENT or ESDL_MOUSEBUTTON_EVENT are passed to the subscribers of an event as argument. These classes were only partially implemented so far. For example there was no possibility to query a keyboard event for the key that caused the event. I implemented the missing queries for mouse and keyboard events to make it possible for students to use such events.

I also made changes to ESDL_EVENT_LOOP itself which keeps an ESDL application running and dispatches the events received from SDL.

3.9.1 Keyboard Event Classes

The features to query for a key that caused a keyboard event where not implemented yet in the class ESDL_KEYBOARD_EVENT. Instead there was another class ESDL_KEY that provided clients with the possibility to query the current state of the keyboard. The feature *scan* enabled clients to check whether a specific key, passed as argument, is currently pressed or not. This lead to ugly keyboard event handling code in all existing ESDL examples that queried for all the keys they were interested in by sequences of if-statements.

I set this class ESDL_KEY to obsolete and introduced a new class ESDL_KEYBOARD. This new class primarily provides features to customize the keyboard, for example to customize whether keyboard events are repeated when the user keeps pressing a key for a long time. This class ESDL_KEYBOARD also has a feature to query whether a specific key is currently pressed since sometimes only knowing which key caused a keyboard event might not be sufficient. But developers are not supposed to use this feature often to query for the state of a key.

```
class interface
    ESDL_KEYBOARD

create
    make_snapshot

feature -- Initialization

    make_snapshot
            -- Create snapshot of current keyboard state.

feature -- Status report

    is_unicode_characters_enabled: BOOLEAN
            -- Is receiving of 'unicode_character' for keys
            -- in keyboard events enabled?
```

```
feature -- Status setting

    enable_unicode_characters
            -- Enable receiving of 'unicode_character' for keys
            -- in keyboard events.
            -- Fetching unicode characters incurs a slight
            -- overhead for each keyboard event
            -- and should therefore only be enabled
            -- if absolutly necessary (i.e. for textual input).
        ensure
            unicode_characters_enabled: is_unicode_characters_enabled

    disable_unicode_characters
            -- Disable receiving of 'unicode_character'
            -- in ESDL_KEYBOARD_EVENT.
        ensure
            unicode_characters_disabled: not is_unicode_characters_enabled

feature -- Basic operations

    enable_repeating_key_down_events (delay: INTEGER; interval: INTEGER)
            -- Enable repeating key down events,
            -- when the user keeps pressing a key
            -- for more than 'delay' milliseconds.
            -- Key down events will then occur every 'interval' milliseconds
            -- as long as the user keeps a key pressed.
        require
            positive_delay: delay > 0
            positive_interval: interval > 0

    disable_repeating_key_down_events
            -- Disable repeating key down events
            -- when the user keeps pressing a key for a long time.

feature -- Queries

    is_pressed (a_key: INTEGER): BOOLEAN
            -- Is 'a_key' pressed?
            -- (See sdlk_... features in SDLKEY_ENUM_EXTERNAL
            -- for possible 'a_key' values)

    key_name (key: INTEGER): STRING
            -- SDL name for 'key'
            -- (See sdlk_... features in SDLKEY_ENUM_EXTERNAL
            -- for possible 'key' values)
        require
            key_is_valid: is_valid_key (key)
        ensure
            key_name_returned: Result /= Void and then not Result.is_empty

invariant
    keyboard_state_not_void: keyboard_state /= Void

end -- class ESDL_KEYBOARD
```

Listing 3.20: Class interface ESDL_KEYBOARD

I implemented a lot of features in the class ESDL_KEYBOARD_EVENT to make it convenient for clients to query for the key that caused the event:

```
feature -- Queries

    type: INTEGER
            -- Event type value
            -- ('esdl_key_down_event' or 'esdl_key_up_event')

    is_key_down: BOOLEAN
            -- Is it a key down event?

    is_key_up: BOOLEAN
            -- Is it a key up event?

    key: INTEGER
            -- Key value
            -- (See sdlk_... features in SDLKEY_ENUM_EXTERNAL for possible values)

    key_name: STRING
            -- SDL-Name for key

    character: CHARACTER
            -- Character representing the key of the current event,
            -- not a meaningfull value for all keys,
            -- better use 'unicode_character' for reliable characters
            -- (i.e. for textual input)

    unicode_character: CHARACTER
            -- Character entered in the current key down event (not valid for key up events)
            -- only meaningful if 'is_unicode_characters_enabled' in ESDL_KEYBOARD.

    keyboard_modifiers: INTEGER
            -- Keyboard modifier flags specifying which keyboard modifiers are currently on
            -- (see kmod_... features in SDLMOD_ENUM_EXTERNAL for possible flags)

    is_keyboard_modifier_on (keyboard_modifier_flag: INTEGER): BOOLEAN
            -- Is 'keyboard modifier_flag' turned on in 'keyboard_modifier_flags'.
            -- (see kmod_... features in SDLMOD_ENUM_EXTERNAL for possible '
                keyboard_modifier_flag' values)
        require
            valid_keyboard_modifier_flag: is_valid_keyboard_modifier_flag (
                keyboard_modifier_flag)

    is_alt_pressed: BOOLEAN
            -- Is an alt key currently pressed?

    is_alt_left_pressed: BOOLEAN
            -- Is left alt key currently pressed?

    is_alt_right_pressed: BOOLEAN
            -- Is right alt key currently pressed?

    is_control_pressed: BOOLEAN
            -- Is a control key currently pressed?

    is_control_left_pressed: BOOLEAN
            -- Is left control key currently pressed?
```

```
is_control_right_pressed: BOOLEAN
        -- Is right control key currently pressed?

is_shift_pressed: BOOLEAN
        -- Is a shift key currently pressed?

is_shift_left_pressed: BOOLEAN
        -- Is left shift key currently pressed?

is_shift_right_pressed: BOOLEAN
        -- Is right shift key currently pressed?

is_caps_locked: BOOLEAN
        -- Is caps lock enabled?

is_num_locked: BOOLEAN
        -- Is num lock enabled?
```

Listing 3.21: Queries in class ESDL_KEYBOARD_EVENT

Instead of querying for the state of each key using ESDL_KEYBOARD, developers should rather use these queries of ESDL_KEYBOARD_EVENT to decide which key has been pressed or released whenever possible.

3.9.2 Mouse Event Classes

The queries in the mouse event classes were also not implemented yet. There are two different mouse events in ESDL: ESDL_MOUSEBUTTON_EVENT and ESDL_MOUSEMOTION_EVENT. There was no common ancestor class for all mouse events. Having such a class would be helpful, since both events have some queries in common. For example every mouse event knows the position where the mouse pointer was, no matter if a mouse button was pressed or the mouse was moved. Therefore I introduced a new class ESDL_MOUSE_EVENT as ancestor of the two mouse event classes.

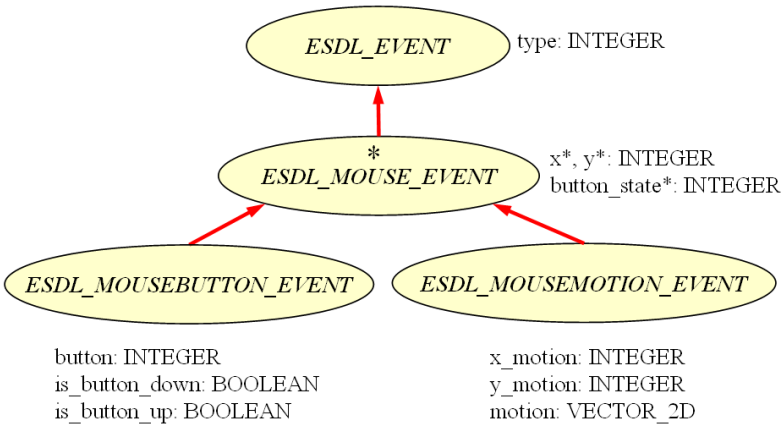

Figure 3.23: Mouse event classes as descendants of ESDL_MOUSE_EVENT

ESDL_MOUSE_EVENT has two features x and y to query for the position of the mouse pointer where the event occurred. Furthermore, it has features to query for the state of the mouse buttons. This features are for convenience, e.g. to make it easy to check whether two buttons are pressed together.

```
x: INTEGER
        -- X-coordinate of pointer position on screen
```

```
        -- where the mouse event has occured

    y: INTEGER
        -- Y-Coordinate of pointer position on screen
        -- where the mouse event has occured

    button_state: INTEGER
        -- Mouse button state specifying which mouse buttons
        -- are currently pressed
        -- (see button_state_xxx_flag features for possible
        -- flag values)

    button_state_left: BOOLEAN
        -- Is left mouse button currently down?

    button_state_middle: BOOLEAN
        -- Is middle mouse button currently down?

    button_state_right: BOOLEAN
        -- Is right mouse button currently down?
```

Listing 3.22: Common queries for all mouse events in class ESDL_MOUSE_EVENT

The following listing shows the queries I implemented in the class ESDL_MOUSEBUTTON_EVENT to query for the mouse button that caused an event. You should not confuse the mouse button state with the button that caused an event. For example it is possible that *button_state_left* is *True* and *is_left_button* is *False*. This means that the left mouse button is currently down but it was another mouse button that was pressed or released.

```
    type: INTEGER
        -- Event type value
        -- ('Esdl_mouse_button_down_event' or
        -- 'Esdl_mouse_button_up_event')

    is_button_down: BOOLEAN
        -- Is it a button down event?

    is_button_up: BOOLEAN
        -- Is it a button up event?

    button: INTEGER
        -- Number of the button that has been pressed or released

    is_left_button: BOOLEAN
        -- Has the left button been pressed or released?

    is_middle_button: BOOLEAN
        -- Has the middle button been pressed or released?

    is_right_button: BOOLEAN
        -- Has the right button been pressed or released?

    is_mouse_wheel_down: BOOLEAN
        -- Has the mouse wheel been moved down?

    is_mouse_wheel_up: BOOLEAN
        -- Has the mouse wheel been moved up?
```

Listing 3.23: Queries in class ESDL_MOUSEBUTTON_EVENT

The following listing shows the queries I implemented in the class ESDL_MOUSEMOTION_EVENT to query for the mouse motion that caused the event:

```
x_motion: INTEGER
        -- Relative motion in x-direction on screen

y_motion: INTEGER
        -- Relative motion in y-direction on screen

motion: VECTOR_2D
        -- Relative motion on screen
    ensure
        result_not_void: Result /= Void
```

Listing 3.24: Queries in class ESDL_MOUSEMOTION_EVENT

3.9.3 Event Loop

The class ESDL_EVENT_LOOP provides ESDL applications with a main loop that processes the events in the event queue of SDL and dispatches them. There are several EVENT_TYPE features to which clients can subscribe their event handler procedures passed as agents:

```
key_down_event: EVENT_TYPE [TUPLE []]
        -- Key down event

key_up_event: EVENT_TYPE [TUPLE []]
        -- Key up event

mouse_motion_event: EVENT_TYPE [TUPLE []]
        -- Mouse motion event

mouse_button_down_event: EVENT_TYPE [TUPLE []]
        -- Mouse button down event

mouse_button_up_event: EVENT_TYPE [TUPLE []]
        -- Mouse button up event

quit_event: EVENT_TYPE [TUPLE []]
        -- Quit event

outside_event: EVENT_TYPE [TUPLE []]
        -- Outside event, allows the user to perform actions outside the event loop
```

Listing 3.25: Most important events in class ESDL_EVENT_LOOP

I fixed two problems with the way the event loop was polling for events from SDL:

- There is a special event *outside_event* that is published whenever the event queue is empty. Most of the example ESDL applications use this event to redraw the screen after each frame. But when a lot of events occur, e.g. when the user moves the mouse a lot, the event queue might not get empty for a long time, since the application has to process all the events. Especially in interactive applications like games there are usually a lot of events that occur. In this case the outside event is not called and the application does not update the screen anymore. The user would have the impression that the application is frozen. This problem especially occurred when debugging an

ESDL application in EiffelStudio, since debugging an ESDL application makes it much slower. We do not want to provide the students with a framework that makes it almost impossible to run their application in debug mode.
Therefore I changed the implementation of the event loop a little bit. The new implementation does not publish an outside event only after all the waiting events in the event queue have been processed. I introduced an upper time limit during which the event loop processes the events. When this time is over the outside event is called once, no matter if there are still other events waiting. I set this time limit to 30 milliseconds, which should be small enough to make frame rates of more than 20 frames per second possible.
Testings showed that this new event loop implementation seems to run very well. I was even able to run my interactive test application in the debugger of EiffelStudio without having the problem that the application did not react when moving the mouse a lot anymore.

- Another problem with the polling event loop was, that it did not give the operating system any time to perform other tasks. Even when there where no events at all, the loop was publishing the outside event again and again, causing the application to redraw the screen much more than needed. A frame rate of 25 frames per second is usually sufficient. Therefore I introduced another time limit for the event loop to give the operating system some time when the application itself is not too busy. This time limit ensures that the outside event is not called more than once every 40 milliseconds. When one pass of the event loop needs less than this 40 milliseconds the event loop causes the system to delay for the remaining time and thus enables the operating system to perform other tasks.

For more detailed informations on the implementation of this event loop I recommend to look at the source code of the class ESDL_EVENT_LOOP.

I also introduced two new features *process_events* and *delay_and_process*. These features are useful if the application has to perform an extended calculation and still wants to keep the application interactive.

3.10 Base Classes for Interactive Applications

3.10.1 Overview

By looking at the source code of the existing ESDL examples it is very obvious that all this applications have some parts that always look very similar. All examples have to initialize the video subsystem first for getting the screen onto which they can draw. Then most of the examples create a scene that consists of drawable objects. Some applications introduce a class inheriting from ESDL_SCENE others build their scene directly inside the root class. Then all applications either create their own ESDL_EVENT_LOOP or they use the one from ESDL_SCENE. More complex applications that are built of several scenes implement a main loop to go to the next scene when the previous scene was terminated. Most examples have to redraw the screen for each frame of their animations by themselves. It is obvious that these parts should be turned into reusable classes. Instead of always copying them into each ESDL application, there should be some base classes from which a developer can derive his application.

As shortly introduced in section 3.3.1 there is already a base class ESDL_SCENE that developers can inherit from to build their interactive graphical scenes of an application. This class mainly provides an event loop and handler procedures that descendants can redefine to receive some of the events. There is a deferred feature *initialize_scene* that descendants have to implement to create the drawable objects the scene is built of. ESDL_SCENE even suggests that all the drawable objects in a scene are drawn onto the screen that is passed as an argument to the feature *run*. But in fact, this class does not provide any mechanism to draw anything nor does it have a container to hold all the drawable objects a scene is usually built of. All classes inheriting from ESDL_SCENE always had to implement this parts on their own.

In the following subsections I will discuss the parts of an ESDL application that I turned into reusable parts of the ESDL library. I extended the class ESDL_SCENE to provide mechanisms for displaying and animating a scene. I even introduced an effective descendant ESDL_SIMPLE_SCENE that makes it

possible to build very simple scenes without any need for inheriting from ESDL_SCENE. Furthermore I introduced a class ESDL_APPLICATION as a base class for ESDL Applications. This class provides all the functionalities needed to run one or more scenes. Using this classes it is now possible to develop little ESDL Applications with very few lines of code. For example the developer does not necessarily have to care about the settings of the video subsystem anymore for building some simple application.

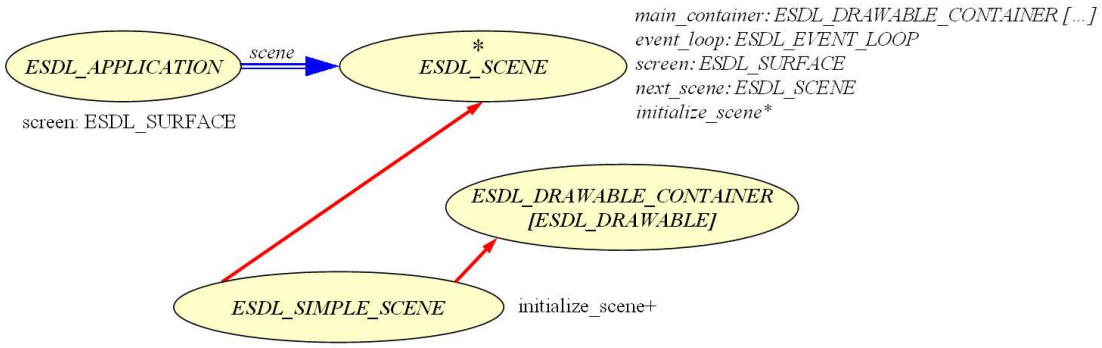

Figure 3.24: BON diagram of base classes for ESDL applications.

Last but not least I also introduced a mechanism to publish the mouse events to all ESDL_DRAWABLE objects contained in an ESDL_SCENE. This mechanism takes care of all the coordinate transformations for publishing only the mouse events to objects the mouse pointer is over. This makes it possible to subscribe for mouse events occurring over a specific drawable object. Using this mechanism it was also possible to introduce the class ESDL_ZOOMABLE_WIDGET inheriting from ESDL_ ZOOMABLE_ CONTAINER. This widget component allows the user to zoom and scroll the content inside an ESDL_ZOOMABLE_WIDGET using the mouse.

3.10.2 Base Classes for Interactive Scenes

Usually an ESDL application, for example a game, is composed of interactive scenes, similar to scenes in a movie or levels in a game. These scenes are composed of drawable objects that might be animated. The class ESDL_SCENE is a deferred base class for such scenes. The scenes might be interactive, which means that the user can control the scene, for example by steering the main character of the game.

3.10.2.1 ESDL_SCENE

Clients of an ESDL_SCENE (i.e. applications) are supposed to call the feature *initialize_scene* before launching the scene with the command *run:*

```
initialize_scene
        -- Initialize the scene by filling 'main_container' with drawables
        -- the scene consists of.

run (a_screen: ESDL_SURFACE)
        -- Run the scene and show it on 'a_screen'.
    require
        a_screen_not_void: a_screen /= Void

event_loop: ESDL_EVENT_LOOP
        -- Event loop that makes the scene running

screen: ESDL_SURFACE
        -- Surface where the scene is drawed
```

Listing 3.26: Features in class ESDL_SCENE

Developing a scene means to inherit from ESDL_SCENE and to implement the deferred feature *initialize_scene* to prepare drawable objects needed to draw the scene. This is especially helpful to preload all image files before the scene is running.

In the former version of ESDL, descendants also had to implement the deferred feature *handle_outside_event* to redraw the screen after each frame. Instead of this, I introduced two new features inside ESDL_SCENE. There is now a *main_container* supposed to contain all drawable objects a scene consists of. This enables us to introduce a default implementation for drawing the scene. Therefore, the new feature *redraw* draws the *main_container* onto *screen*. Descendant scenes only have to put their drawable objects into this main container. There is no need to implement an own redrawing mechanism anymore. The feature *redraw* is automatically called after each frame and ensures to draw the current state of a scene.

```
main_container: ESDL_DRAWABLE_CONTAINER [ESDL_DRAWABLE]
        -- Container that contains all drawable objects the scene is built of.

redraw
        -- Redraw 'Current' scene.
        -- Causes to clear 'screen' and redraw 'main_container' onto it.
        -- Does nothing if 'main_container' is 'Void'.
    require
        screen_not_void: screen /= Void
```

Listing 3.27: New features in class ESDL_SCENE

To add interaction to a scene, descendants can redefine the event handler features, for example to change the objects inside the *main_container* when a key is pressed. This event handler features are subscribed to the event loop of the scene, when the scene is running.

```
feature {NONE} -- Event handlers

    handle_key_down_event (a_keyboard_event: ESDL_KEYBOARD_EVENT) is
            -- Handle key down event.
            -- (Descendants can redefine this feature
            -- to do something when a key is pressed.)
        require
            a_keyboard_event_not_void: a_keyboard_event /= Void
        do
        end

    handle_key_up_event (a_keyboard_event: ESDL_KEYBOARD_EVENT) is
            -- Handle key up event.
            -- (Descendants can redefine this feature
            -- to do something when a key is released.)
        require
            a_keyboard_event_not_void: a_keyboard_event /= Void
        do
        end

    handle_quit_event (a_quit_event: ESDL_QUIT_EVENT) is
            -- Handle quit event.
            -- (Descendants can redefine this feature
            -- to do something when application is quit.)
        require
            a_quit_event_not_void: a_quit_event /= Void
        do
        end
```

```
    handle_mouse_button_down_event (a_mouse_button_event:
        ESDL_MOUSEBUTTON_EVENT) is
            -- Handle mouse button down event.
            -- (Descendants can redefine this feature
            -- to do something when a mouse button is pressed.)
        require
            a_mouse_button_event_not_void: a_mouse_button_event /= Void
        do
        end

    handle_mouse_button_up_event (a_mouse_button_event:
        ESDL_MOUSEBUTTON_EVENT) is
            -- Handle mouse button up event.
            -- (Descendants can redefine this feature
            -- to do something when a mouse button is released.)
        require
            a_mouse_button_event_not_void: a_mouse_button_event /= Void
        do
        end

    handle_mouse_motion_event (a_mouse_motion_event: ESDL_MOUSEMOTION_EVENT
        ) is
            -- Handle mouse button move event.
            -- (Descendants can redefine this feature
            -- to do something when the mouse is moved)
        require
            a_mouse_motion_event_not_void: a_mouse_motion_event /= Void
        do
        end

    handle_outside_event is
            -- Handle the outside event.
            -- (Descendants can redefine this feature
            -- to do something after each event loop pass.)
        do
        end
```

Listing 3.28: Event handler features in class ESDL_SCENE

To terminate a scene you can use the command *event_loop.stop*. This terminates the event loop of the scene. A good movie is usually composed of more than one scene. Therefore, there is a feature *next_scene* that can be set to the next scene the application has to initialize and to run when the scene has ended. Setting this feature to *Void* means that the movie is over and the application will terminate when the scene has ended.

The following little example shows how to implement a very simple scene with an image, that is moved a little bit whenever the user presses a key:

```
class
    MY_SIMPLE_SCENE

inherit
    ESDL_SCENE
        redefine
            handle_key_down_event
        end

feature -- Initialization
```

```
    my_image: ESDL_SURFACE
            -- Image that is moved.

    initialize_scene is
            -- Initialize the scene.
        do
            create my_image.make_from_image ("./pics/world.gif")
            my_image.set_x_y (300, 300)
            main_container.extend (my_image)
        end

    handle_key_down_event (a_keyboard_event: ESDL_KEYBOARD_EVENT) is
            -- Move 'my_image' whenever a key is pressed.
        local
            new_x, new_y: INTEGER
        do
            new_x := my_image.x + 20
            new_y := my_image.y - 10
            my_image.set_x_y (new_x, new_y)
        end

end -- class MY_SIMPLE_SCENE
```

Listing 3.29: Little example of an interactive scene

3.10.2.2 Animation Mechanism

As already discussed in section 3.3.2.5 there is a new animation mechanism for all drawable objects. Objects that are animatable, like ESDL_SPRITE are not automatically animated anymore when they get drawn. To ensure that objects inheriting from ESDL_ANIMATABLE are animated when a scene is running, we need to subscribe them to *animation_event* of ESDL_SCENE. I introduced the features *start_animating* and *stop_animating* to make this subscription more convenient and also usable for developers (e.g. students) that do not know anything about agents yet.

```
feature -- Animation

    animate
            -- Let all subscribed animatable objects perform their animation.
            -- (Calls 'go_to_time' of animatable objects with current time tick)

    animation_event: EVENT_TYPE [TUPLE [INTEGER]]
            -- Animation event, allows animatable objects to perform animation
            -- (i.e. moving them selves) before they get drawed.
            -- As an argument the reference time in milliseconds is passed
            -- up to which the animatable objects should draw them selves.
            -- This event gets published right before the scene is redrawed.

    start_animating (an_animatable: ESDL_ANIMATABLE)
            -- Subscribe 'an_animatable' to be animated when 'Current' is running.

    stop_animating (an_animatable: ESDL_ANIMATABLE)
            -- Unsubscribe 'an_animatable' from beeing animated when 'Current' is running.
```

Listing 3.30: Animation features in ESDL_SCENE

The feature *animate* ensures to call the feature *go_to_time* of all subscribed descendants of ESDL_ANIMATABLE to animate them at the same time tick passed as an argument. The scene already subscribes the *animate* feature to the outside event to ensure that it is called right before the scene is drawn.

The following little example shows how to initialize a scene with a sprite showing an animation loaded from a file:

```
initialize_scene is
        -- Initialize the scene.
    local
        my_animation: ESDL_ANIMATION
        my_sprite: ESDL_SPRITE
    do
        create my_animation.make_from_file ("./pics/alien.anim")
        create my_sprite.make (my_animation)
        my_sprite.set_x_y (500, 100)
        main_container.extend (my_sprite)
        start_animating (my_sprite)
    end
```

Listing 3.31: Initializing a scene with an animated sprite

3.10.2.3 ESDL_SIMPLE_SCENE

ESDL_SIMPLE_SCENE is an effective descendant of ESDL_SCENE. This class can be used for building very simple scenes without any need to introduce a new class for the scene. ESDL_SIMPLE_SCENE even inherits from ESDL_DRAWABLE_CONTAINER, which makes it more convenient to extend the scene with objects it consists of. The *main_container* of an ESDL_SIMPLE_SCENE is set to the ESDL_SIMPLE_SCENE itself. Therefore, we can simply create an instance of this class and add drawable objects to it for creating a scene. This makes it very simple to create a little scene showing some images, figures, or even animations.

There are three additional features in ESDL_SIMPLE_SCENE. First of all there is a creation feature to initialize the empty scene and a feature *set_background_color* to set the background color of the scene. Then the *set_next_scene* feature enables clients to set another scene that will be executed when the scene is finished.

Consider the class interface documentation for more information. A very simple example of using this class can be found in the new ESDL Hello World example in Listing 3.34 in the next section.

3.10.2.4 ESDL_SHARED_SCENE

The currently running scene is obviously a very important object in an ESDL application, since it holds the event loop that keeps the scene running and dispatches all events. Since it might be necessary for any object in an ESDL application to have access to the currently running scene and its event loop I introduced a class ESDL_SHARED_SCENE that holds a singleton cell always containing the currently running scene:

```
class
    ESDL_SHARED_SCENE

feature -- Access

    running_scene: ESDL_SCENE is
            -- Scene that is currently running.
        do
            Result := running_scene_cell.item
```

```
        end

    feature -- Element change

        set_running_scene (a_scene: ESDL_SCENE) is
            -- Set 'running_scene' to 'a_scene'.
        do
            running_scene_cell.put (a_scene)
        ensure
            running_scene_set: running_scene = a_scene
        end

    feature {NONE} -- Implementation

        running_scene_cell: CELL[ESDL_SCENE] is
            --
        once
            create Result
        end

    end -- class ESDL_SHARED_SCENE
```

Listing 3.32: Class ESDL_SHARED_SCENE

Using this class we can always access the currently running scene in an ESDL application. This might be necessary for example to keep the event loop running even when performing extended calculations or to subscribe to its events, e.g. to keyboard events.

3.10.3 Base Class for Applications

So far there was no base class for ESDL Applications. Therefore every ESDL Application had to perform the initialization of the screen on its own. Such simple examples as the Hello World example that loads an image and displays it looked as follows:

```
class
    HELLO_WORLD

inherit
    ESDL_CONSTANTS
        export
            {NONE} all
        end

    ESDL_SHARED_SUBSYSTEMS
        export
            {NONE} all
        end

create
    make

feature --Creation

    make is
            -- Create the main application.
        do
            video_subsystem.set_video_surface_width (width)
```

```
            video_subsystem.set_video_surface_height (height)
            video_subsystem.set_video_bpp (resolution)
            video_subsystem.enable
            if
                video_subsystem.is_enabled
            then
                screen := video_subsystem.video_surface

                create image.make_from_image (Image_file_name)
                screen.blit_surface (image, 0, 0)

                screen.redraw

                -- Event loop
                create event_loop.make_wait
                event_loop.quit_event.subscribe (agent handle_quit_event (?))
                event_loop.dispatch

                video_subsystem.disable
            else
                print ("Video mode could not be enabled!%N")
            end
        end

feature -- Event handling

    handle_quit_event (a_quit_event: ESDL_QUIT_EVENT) is
            -- Handle quit events.
        require
            a_quit_event_not_void: a_quit_event /= Void
        do
            event_loop.stop
        end

feature {NONE} -- Implementation

    screen: ESDL_SURFACE
            -- The screen

    image: ESDL_SURFACE
            -- The image

    event_loop: ESDL_EVENT_LOOP
            -- The event loop

    width: INTEGER is 283
            -- The width of the surface

    height: INTEGER is 274
            -- The height of the surface

    resolution: INTEGER is 16
            -- The resolution of the surface

    Image_file_name: STRING is "hello_world.gif"
            -- Name of image we want to display
```

```
end
```

Listing 3.33: Hello World example in former ESDL version

I introduced the class ESDL_APPLICATION to provide developers with a base implementation for this initialization parts. By simply inheriting from ESDL_APPLICATION and using an ESDL_SIMPLE_SCENE the Hello World Example now looks as follows:

```
class
    MY_APPLICATION

inherit
    ESDL_APPLICATION

create
    make_and_launch

feature -- Creation

    make_and_launch is
            -- Create and execute the application.
        local
            my_scene: ESDL_SIMPLE_SCENE
            image: ESDL_SURFACE
        do
            -- Initialize the screen.
            initialize_screen

            -- Create the first scene.
            create my_scene.make
            create image.make_from_image ("hello_world.gif")
            my_scene.extend (image)

            -- Set and launch the first scene.
            set_scene (my_scene)
            launch
        end

end
```

Listing 3.34: Hello World Example using ESDL_APPLICATION and ESDL_SIMPLE_SCENE

ESDL_APPLICATION not only supports to initialize the screen but also to run several scenes after each other. Applications do not need anymore to provide a main loop that goes from scene to scene until *next_scene* is *Void*. Descendants of ESDL_APPLICATION only have to set the feature *scene* to the first scene they want to run and then call the *launch* feature.

```
class interface
    ESDL_APPLICATION

feature -- Status report

    scene: ESDL_SCENE
            -- The scene that 'launch' executes

    screen: ESDL_SURFACE
            -- Screen where application is displayed
```

```
feature -- Status setting

    set_scene (a_scene: ESDL_SCENE)
            -- Set 'scene' to 'a_scene'.
        ensure
            scene_set: scene = a_scene

feature -- Commands

    initialize_screen
            -- Initialize 'screen' to the 'video_surface'
        ensure
            screen_initialized: screen /= Void

    initialize_video_subsystem
            -- Configure the video subsystem.
            -- (Descendants can redefine this feature
            -- to set up the video system before the screen is
            -- initialized)

    launch
            -- Launch the application by running 'scene'.
        require
            screen_initialized: screen /= Void
            scene_not_void: scene /= Void

end -- class ESDL_APPLICATION
```

Listing 3.35: Class interface of ESDL_APPLICATION

3.10.4 Mouse Event Mechanism

So far there is only one possibility to receive mouse events: subscribing for the mouse events in ESDL_EVENT_LOOP. This mouse events are published whenever a mouse event occurs, no matter over which part of the screen. This makes it very complicated to only catch mouse events that occur over a specific part of the screen, for example over a specific drawable object. It gets even more complex when you want to catch a mouse event that occurred over a drawable object that is inside an ESDL_ZOOMABLE_CONTAINER. Then you would have to transform the mouse coordinates into the object space of the container, that might be zoomed or scrolled.

To make it much more simpler to decide which object has been clicked on, no matter if it is inside a container that defines its own coordinate system or not, I introduced the possibility to subscribe for mouse events over each drawable object separately. Therefore I introduced some mouse event features in ESDL_DRAWABLE:

```
feature -- Mouse events
    mouse_button_down_event: EVENT_TYPE [TUPLE [ESDL_MOUSE_EVENT]]
            -- Mouse button down event,
            -- gets published when the mouse button is pressed over 'Current',
            -- an ESDL_MOUSEBUTTON_EVENT is passed as argument

    mouse_button_up_event: EVENT_TYPE [TUPLE [ESDL_MOUSE_EVENT]]
            -- Mouse button up event,
            -- gets published when the mouse button is released over 'Current',
            -- an ESDL_MOUSEBUTTON_EVENT is passed as argument

    mouse_motion_event: EVENT_TYPE [TUPLE [ESDL_MOUSE_EVENT]]
```

```
        -- Mouse button up event,
        -- gets published when the mouse button is released over 'Current',
        -- an ESDL_MOUSEMOTION_EVENT is passed as argument

    publish_mouse_event (a_mouse_event: ESDL_MOUSE_EVENT)
        -- Publish mouse event when 'a_mouse_event' occured on 'Current'.
        -- Descendants should redefine this feature
        -- for only catching and publishing their mouse events when mouse pointer
        -- is realy inside object or for
        -- distributing mouse events to child objects.
      require
        a_mouse_event_not_void: a_mouse_event /= Void
```

Listing 3.36: Mouse event features in ESDL_DRAWABLE

Each drawable object has now a feature *publish_mouse_event* that is responsible to publish its appropriate mouse events and also to inform its child objects, e.g. if the drawable object is a container. The scene publishes the mouse events to its *main_container* and thus to all drawable objects inside the scene.

Furthermore I introduced the attribute *proportional_position* in the class ESDL_MOUSE_EVENT:

```
    proportional_position: VECTOR_2D
        -- Mouse pointer position in possibly transformed coordinates
        -- inside the publishing container object

    set_proportional_position (a_position: VECTOR_2D)
        -- Set 'proportional_position' to 'a_position'.
      require
        a_position_not_void: a_position /= Void
      ensure
        proportional_position_set: proportional_position = a_position
```

Listing 3.37: Proportional position in ESDL_MOUSE_EVENT

Initially the *proportional_position* is the same as the absolute position of the mouse pointer when the event occurred. Containers that define a new coordinate system for their contained objects are now responsible to set the *proportional_coordinate* to the transformed coordinate inside their new coordinate system. Otherwise their child objects could not decide if the mouse event really occurred over them and thus if they have to publish their mouse events.

With this mechanism it is very easy to subscribe for mouse events over a specific drawable object. Consider the following little example that moves an animated image (*my_sprite*) when the mouse is moved over it:

```
class
  MY_MOUSE_SCENE

inherit
  ESDL_SCENE

feature -- Initialization

  my_sprite: ESDL_SPRITE
      -- Sprite that is moved on mouse overs.

  initialize_scene is
      -- Initialize the scene.
    local
```

```
        my_animation: ESDL_ANIMATION

    do
        -- Add some simple sprite containing an animation and animate it.
        create my_animation.make_from_file ("./pics/alien.anim")
        create my_sprite.make (my_animation)
        my_sprite.set_x_y (500, 100)
        main_container.extend (my_sprite)
        start_animating (my_sprite)

        -- Subscribe for mouse motion events over 'my_sprite'
        my_sprite.mouse_motion_event.subscribe (agent move_my_sprite)
    end

  move_my_sprite (a_motion_event: ESDL_MOUSEMOTION_EVENT) is
        -- Move 'my_sprite' to the position of the cursor,
        -- when mouse is moved over it.
    local
        new_x, new_y: INTEGER
    do
        -- Move center of 'my_sprite' to mouse position.
        new_x := a_motion_event.proportional_position.x.floor
        new_x := new_x - my_sprite.width // 2
        new_y := a_motion_event.proportional_position.y.floor
        new_y := new_y - my_sprite.height // 2
        my_sprite.set_x_y (new_x, new_y)
    end

end -- class MY_MOUSE_SCENE
```

Listing 3.38: Example using mouse events of ESDL_DRAWABLE

Furthermore I introduced the following additional mouse events in ESDL_DRAWABLE_CONTAINER:

```
feature -- Mouse events
  mouse_button_down_on_item_event: EVENT_TYPE [TUPLE [G]]
        -- Mouse button down on item event,
        -- gets published when the mouse button event
        -- is caught by an item inside 'Current',
        -- item is passed as first argument to subscribers,
        -- an ESDL_MOUSEBUTTON_EVENT is passed
        -- as optional second argument

  mouse_button_up_on_item_event: EVENT_TYPE [TUPLE [G]]
        -- Mouse button up on item event,
        -- gets published when the mouse button event
        -- is caught by an item inside 'Current',
        -- item is passed as first argument to subscribers,
        -- an ESDL_MOUSEBUTTON_EVENT is passed
        -- as optional second argument

  mouse_motion_on_item_event: EVENT_TYPE [TUPLE [G]]
        -- Mouse motion on item event,
        -- gets published when the mouse motion event
        -- is caught by an item inside 'Current',
        -- item is passed as first argument to subscribers,
        -- an ESDL_MOUSEBUTTON_EVENT is passed
        -- as optional second argument
```

Listing 3.39: Additional mouse events in ESDL_DRAWABLE_CONTAINER

This events only get published when a mouse event occurred over one of the contained drawable objects. The event handler even gets the drawable object over which the event occurred as an argument. This makes it very simple to subscribe for mouse events of all objects inside a container without any need to subscribe to the mouse events of all these objects.

Using this possibility we could change the example in listing 3.38 accordingly to enable the user to move any drawable object inside the main container :

```
class
    MY_MOUSE_CONTAINER_SCENE

inherit
    ESDL_SCENE
        redefine
            handle_key_down_event
        end

feature -- Initialization

    initialize_scene is
            -- Initialize the scene.
        do
            -- Add a lot of items into 'main_container'
            ...

            main_container.mouse_motion_on_item_event.subscribe (agent move_item)
        end

    move_item (an_item: ESDL_DRAWABLE; a_motion_event:
          ESDL_MOUSEMOTION_EVENT) is
            -- Move 'an_item' to the position of the cursor,
            -- when mouse is moved over it.
        local
            new_x, new_y: INTEGER
        do
            -- Move center of 'an_item' to mouse position.
            new_x := a_motion_event.proportional_position.x.floor
            new_x := new_x - an_item.width // 2
            new_y := a_motion_event.proportional_position.y.floor
            new_y := new_y - an_item.height // 2
            an_item.set_x_y (new_x, new_y)
        end

end -- class MY_MOUSE_CONTAINER_SCENE
```

Listing 3.40: Example using mouse events of ESDL_DRAWABLE_CONTAINER

3.10.5 ESDL_ZOOMABLE_WIDGET

Using the new mouse event publishing mechanism as discussed in subsection 3.10.4 I introduced a new class ESDL_ZOOMABLE_WIDGET inheriting from ESDL_ZOOMABLE_CONTAINER. Consider the following source code of this class which should be self explaining:

```
class
    ESDL_ZOOMABLE_WIDGET
```

```
inherit
    ESDL_ZOOMABLE_CONTAINER
        redefine
            make
        end

create
    make

feature {NONE} -- Initialization

    make (a_width, a_height: INTEGER) is
            -- Make empty container with size 'a_width' and 'a_height'.
        do
            Precursor (a_width, a_height)
            mouse_motion_event.subscribe (agent process_mouse_move_event)
        ensure then
            no_contained_drawables: is_empty
        end

feature {NONE} -- Implementation

    process_mouse_move_event (a_event: ESDL_MOUSEMOTION_EVENT) is
            -- Process mouse move event over 'Current' to zoom or scroll Current.
        local
            zoom_step: DOUBLE
        do
            if a_event.button_state_right then
                scroll_proportional (- a_event.motion)
            end
            if a_event.button_state_middle then
                zoom_step := 1.0 - (a_event.y_motion / 100)
                if zoom_step <= 0.5 then
                    zoom_step := 0.5
                end
                if zoom_step > 2.0 then
                    zoom_step := 2.0
                end
                zoom (zoom_step)
            end
        end

end -- class ESDL_ZOOMABLE_WIDGET
```

Listing 3.41: Class text of ESDL_ZOOMABLE_WIDGET

The user can scroll or zoom the content of an ESDL_DRAWABLE_WIDGET. He has to move the mouse into the direction he wants to scroll while keeping the right mouse button pressed. For zooming he has to keep the middle mouse button pressed and move the mouse forward to zoom into the content or back to zoom out again.

Chapter 4

TRAFFIC Library

4.1 Overview

TRAFFIC [3, 2, 4] is an object-oriented library to model the transportation network of a city. TRAFFIC comes with classes to model the city with its places and connections between these places. Furthermore TRAFFIC provides classes to visualize this model of the city. The intention of TRAFFIC is to provide students with a simple object-oriented library for modeling and visualizing the transportation network of a city. Using this library, the students can solve interesting exercises to learn object-oriented programming. I described the former version of the TRAFFIC library in chapter 2 together with its drawbacks.

The intention of my thesis was to redesign the visualization part of the TRAFFIC library using ESDL. There were more features missing in ESDL than expected and the needed extension was more time-consuming than estimated. In the end there was only little time left to implement the TRAFFIC visualization part. Therefore, this part of my work is more a prototype and there is future work to do on this part to make it usable for students. Following up, I will discuss the parts that were implemented and will propose some ideas on how to continue the development.

First, I will give a short overview of the TRAFFIC model that has to be visualized in section 4.2. In the succeeding section 2.3 I will explain the new design to visualize this model using ESDL.

4.2 TRAFFIC Model

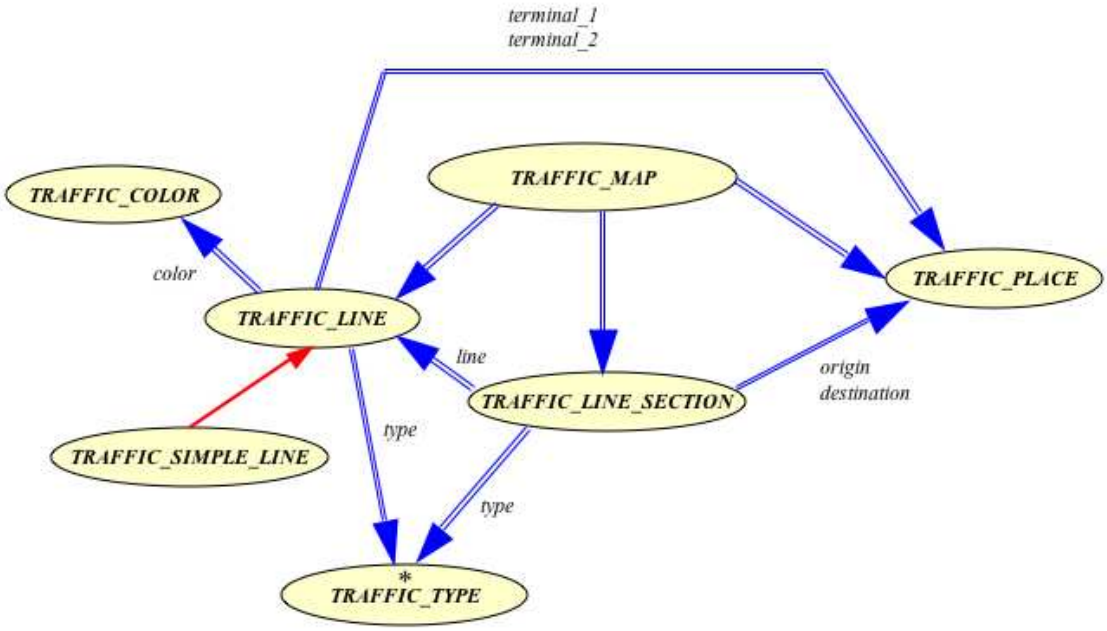

Figure 4.1: Class diagram of new TRAFFIC model design

The model of the transportation network of the city mainly consists of places and line sections. The most important class is the class TRAFFIC_MAP since it holds the whole representation of the transportation map. TRAFFIC_LINE is a transportation line, e.g. “Tram No. 7” or “Bus No. 33”. A line consists of TRAFFIC_LINE_SECTIONs. A line section is a connection between two consecutive places. Places are represented through the class TRAFFIC_PLACE.

The new design of this TRAFFIC model has been developed by Sibylle Aregger as a semester thesis in parallel to my work. More information about the new design can be found in her semester thesis report [10].

4.3 TRAFFIC Visualization

4.3.1 Design Overview

We want to visualize the TRAFFIC model as a map of the city with its places and transportation lines. This means that we have to draw the places and the line sections somehow. I tried to abstract this problem and to generalize it not only to our specific map that we have in TRAFFIC with its places and line sections, but to practically any map you could think of. There are many applications where you could have a map, for example strategy games or geographical information systems. These maps do not necessarily have to consist of places and line sections. There could be many other items inside them, for example buildings, streets, resources, players, mountains, restaurants, vehicles and others.

Therefore we can abstract a model of a map as an aggregation of items that are inside this map. Usually these items are divided into several categories. Each item has to be visualized according to its category. For example a place is visualized differently than a line section, a building or a mountain.

I introduced a deferred class MAP_MODEL that abstracts this model of a map. This allows us to introduce another class MAP_WIDGET that visualizes such a MAP_MODEL without having to know much about the items inside the MAP_MODEL. The only thing the map widget needs is a renderer for each category of items inside the map. Renderers have the purpose of creating the graphical representation for map items.

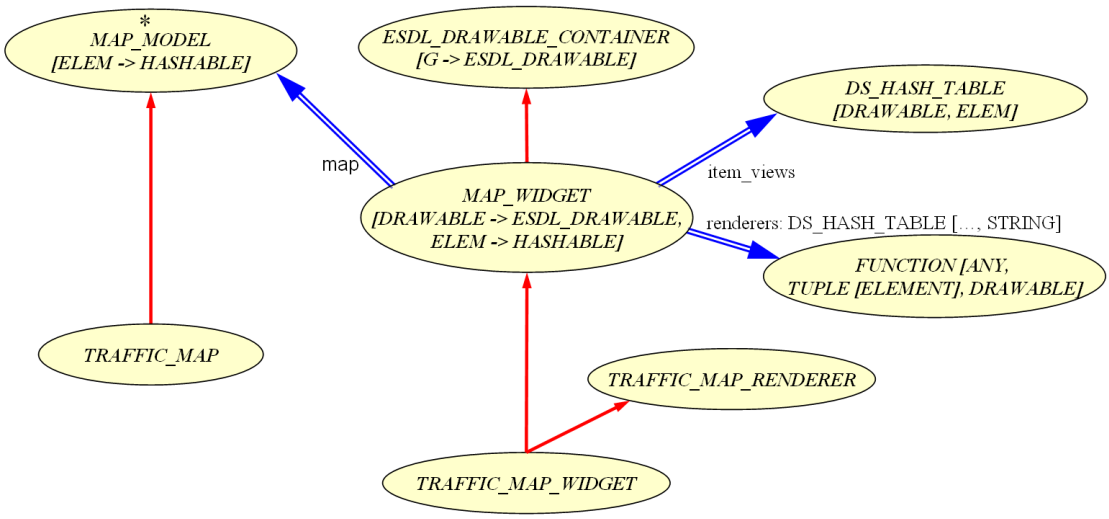

Figure 4.2: Class diagram of new TRAFFIC visualization design

As already discussed in section 2.3.3 the new design will not have any displayer objects anymore that are attached to the objects of the map model. The model objects do not know anything about the view objects in the new design. This enables us to visualize the model in different ways and to introduce several views for the same map model. Creating such a view of a map is easy, since all the clients of the library have to do is creating a MAP_WIDGET and setting it up with renderers to map the model items into drawable objects.

The class TRAFFIC_MAP_RENDERER provides customizable renderers to visualize places and line sections of a TRAFFIC_MAP. TRAFFIC_MAP_WIDGET inherits these renderers and uses them to visualize a TRAFFIC_MAP. Using a TRAFFIC_MAP_WIDGET to visualize a TRAFFIC_MAP is therefore simple.

4.3.2 MAP_MODEL

I introduced a deferred class MAP_MODEL as an abstraction for any map model that can be visualized by the MAP_WIDGET. This class defines what a map widget needs to know from a map model to visualize it. As already explained in the previous section we consider something as a possible model for a map if it consists of several items each of them belonging to a *category*. The only thing we assume about the items inside a map model is, that they are HASHABLE.

```
deferred class
    MAP_MODEL [ELEM_TYPE -> HASHABLE]

    ...

feature -- Queries

    item (i: INTEGER): ELEM_TYPE is
            -- The 'i'-th item to be visualized inside the map,
            -- ordered by z-coordinates from bottom to top.
        require
            i_is_valid_index: 1 <= i and then i <= count
        deferred
        ensure
            result_not_void: Result /= Void
        end

    count: INTEGER is
```

```
        -- Number of items in the map.
    deferred
    ensure
        result_not_negative: Result >= 0
    end

category (i: INTEGER): STRING is
        -- Unique string representing category of i-th item,
        -- all items with same category are rendered in the same way.
        -- (Per default 'generating_type' is used as category,
        -- can be redefined).
    require
        i_is_valid_index: 1 <= i and i <= count
    do
        Result := item (i).generating_type
    ensure
        result_not_empty: Result /= Void and then not Result.is_empty
    end
```

Listing 4.1: Queries in deferred class MAP_MODEL

The *category* is needed since an implementer of a MAP_MODEL might want to use another categorization than the categorization introduced by the classes the map items are instances of. For example you might want to use another renderer for some places even though all places are of the same class. But since we expect the classes of the model items to reflect their categorization, there is a default implementation of the feature *category* that returns the generating type of a map item (type of which it is a direct instance).

Furthermore, a MAP_MODEL is observable, such that views can observe the map model and update, if there has something changed inside the map. Therefore, a map model has several events that have to be published whenever the model changes:

```
changed_event: EVENT_TYPE [TUPLE []]
        -- Event to inform views of 'Current'
        -- when 'Current' changed such that
        -- views need to re-render

item_changed_event: EVENT_TYPE [TUPLE [INTEGER]]
        -- Event to inform views of 'Current'
        -- when item has been changed
        -- at index passed as argument

item_inserted_event: EVENT_TYPE [TUPLE [INTEGER]]
        -- Event to inform views of 'Current'
        -- when item has been inserted
        -- at index passed as argument

item_removed_event: EVENT_TYPE [TUPLE [INTEGER]]
        -- Event to inform views of 'Current'
        -- when item has been removed
        -- at index passed as argument
```

Listing 4.2: Events in MAP_MODEL

There are some more effective features that make it easy to publish the events when something changes. Please consider the documentation of the class interface for more information about that.

4.3.3 MAP_WIDGET

The class MAP_WIDGET is a widget for ESDL that visualizes a MAP_MODEL. It takes care of the mapping from items of the model to drawable objects for displaying these elements in a map.

```
class interface
    MAP_WIDGET [DRAWABLE -> ESDL_DRAWABLE, ELEMENT -> HASHABLE]

create
    make_with_map

feature -- Access

    has_view (an_item: ELEMENT): BOOLEAN
            -- Is there a view for 'an_item' in 'Current'?
        require
            an_item_not_void: an_item /= Void

    item_view (an_item: ELEMENT): DRAWABLE
            -- Drawable representation of 'an_item' inside 'Current'
        require
            an_item_not_void: an_item /= Void
            view_for_item_exists: has_view (an_item)

    map: MAP_MODEL [ELEMENT]
            -- Model of the map visualized by 'Current'

feature -- Status setting

    set_renderer (a_renderer: FUNCTION [ANY, TUPLE [ELEMENT], DRAWABLE];
          a_category: STRING)
            -- Set 'a_renderer' to visualize all items inside 'map' with category 'a_category'.
            -- Renderer is a function that takes an item of the map that has category '
                a_category'
            -- and returns a DRAWABLE to visualize the item in the map.

feature -- Commands

    render
            -- Create all drawable objects to represent 'map'
            -- inside 'Current'.
            -- (wipe out before)
```

Listing 4.3: Features of class MAP_WIDGET

Even though a MAP_WIDGET is already an effective class it will not do much when creating it. The widget needs to know how each item category of the model should be visualized. Therefore, the MAP_WIDGET needs a renderer for each category. A renderer is a function that takes a model item as input and returns a drawable representation for this object as output. Look at the following little example to visualize TRAFFIC_LINE_SECTIONs:

```
    traffic_map: TRAFFIC_MAP
            -- Model of the traffic map.

    map_widget: MAP_WIDGET [HASHABLE, ESDL_DRAWABLE]
            -- Map widget to visualize 'traffic_map'.
```

```
line_section_view (a_line_section: TRAFFIC_LINE_SECTION): ESDL_POLYLINE is
        -- Polyline to visualize 'a_line_section'.
    do
        create Result.make_from_list (a_line_section.polypoints)
        Result.set_line_color (line_section_color)
    end

build_map_widget is
        -- Build 'map_widget' to visualize 'traffic_map'.
    do
        create map_widget.make_with_map (traffic_map)
        map_widget.set_item_renderer (agent line_section_view, "
            TRAFFIC_LINE_SECTION")
        map_widget.set_item_renderer (agent place_view, "TRAFFIC_PLACE")
    end
```

Listing 4.4: Example using a MAP_WIDGET

MAP_WIDGET also provides the possibility to query for the drawable object (feature *item_view*) that currently represents a map item. This is helpful when we want to change the drawable representation of a map item temporarily, for example if we want to highlight a place. The mapping from model objects to view objects is implemented using a hash table, therefore the model items have to be HASHABLE.

Furthermore, the MAP_WIDGET also supports to subscribe for mouse events, that may occur inside the widget. There are some special "on map item"-events that directly pass the map item over which the mouse event occurred to the subscriber.

```
mouse_motion_on_map_item_event: EVENT_TYPE [TUPLE [ELEMENT]]
        -- Mouse motion over map item event,
        -- gets published when mouse is moved over drawable item,
        -- map element is passed as first argument to subscribers,
        -- as optional second argument the appropriate ESDL_MOUSE_EVENT is passed

mouse_button_up_on_map_item_event: EVENT_TYPE [TUPLE [ELEMENT]]
        -- Mouse button pressed over map item event,
        -- gets published when mouse is pressed over item,
        -- map element is passed as first argument to subscribers,
        -- as optional second argument the appropriate ESDL_MOUSE_EVENT is passed

mouse_button_down_on_map_item_event: EVENT_TYPE [TUPLE [ELEMENT]]
        -- Mouse button released over map item event,
        -- gets published when mouse is released over item,
        -- map element is passed as first argument to subscribers,
        -- as optional second argument the appropriate ESDL_MOUSE_EVENT is passed
```

Listing 4.5: Mouse on map item events in MAP_WIDGET

I think that there should even be a way to subscribe for mouse events only over the items of a specific category of items, for example if you are only interested in mouse events over places. This has not been implemented yet.

4.3.4 TRAFFIC_MAP_WIDGET

Using the map widget as introduced in the last few subsections it is already very easy to visualize a TRAFFIC_MAP, since TRAFFIC_MAP inherits from MAP_MODEL. Everything that is missing so far are the renderers to turn the model items into drawable objects. Therefore, I introduced some default

renderers for TRAFFIC_PLACEs and TRAFFIC_LINE_SECTIONs that give developers already a default implementation to visualize a transportation network. These renderer functions are implemented in the class TRAFFIC_MAP_RENDERER. This class allows to customize the way the items are rendered. For example, we can set a line color for each traffic type, e.g. a color for all tram lines.

```
class interface
    TRAFFIC_MAP_RENDERER

create
    make_with_map

feature -- Access

    map: TRAFFIC_MAP
            -- Map of which 'Current' can visualize places.

    traffic_type_colors: DS_HASH_TABLE [ESDL_COLOR, STRING]
            -- Colors for traffic types

    traffic_type_line_widths: DS_HASH_TABLE [DOUBLE, STRING]
            -- Line widths for traffic types

feature -- Status report

    line_color: ESDL_COLOR
            -- Color used to draw traffic line sections
            -- that have no appropriate 'traffic_type_color' set
            -- (if 'Void' either the color of the line is used
            -- or simply white).

    line_width: DOUBLE
            -- Line width used to draw traffic line sections.
            -- (if none set for the traffic

    place_color: ESDL_COLOR
            -- Color used to fill places.

feature -- Status setting

    set_line_color (a_color: ESDL_COLOR)
            -- Set 'line_color' to 'a_color'.
        ensure
            line_color_set: line_color = a_color

    set_line_width (a_line_width: DOUBLE)
            -- Set 'line_width' to 'a_line_width'.
        require
            a_line_width_positive: a_line_width > 0
        ensure
            line_width_set: line_width = a_line_width

    set_place_color (a_color: ESDL_COLOR)
            -- Set 'place_color' to 'a_color'.
        require
            a_color_not_void: a_color /= Void

feature -- Renderers
```

```
    line_section_renderer: FUNCTION [ANY, TUPLE [TRAFFIC_LINE_SECTION],
        ESDL_POLYLINE]

    place_renderer: FUNCTION [ANY, TUPLE [TRAFFIC_PLACE], ESDL_RECTANGLE]

end -- class TRAFFIC_MAP_RENDERER
```

Listing 4.6: Class interface of TRAFFIC_MAP_RENDERER

To make it convenient for students to visualize a traffic map there is another class TRAFFIC_MAP_WIDGET that inherits both from TRAFFIC_MAP_RENDERER and from MAP_WIDGET. This makes visualizing a traffic model as easy as the following example:

```
traffic_map: TRAFFIC_MAP
    -- Model of the map

traffic_map_example is
        -- Build a zoomable visualization
        -- of the map model.
    local
        traffic_map_widget: TRAFFIC_MAP_WIDGET
        zoomable_widget: ESDL_ZOOMABLE_WIDGET
    do
        create traffic_map_widget.make_with_map (traffic_map)
        create zoomable_widget.make (500, 500)
        zoomable_widget.extend (traffic_map_widget)
        main_container.extend (zoomable_widget)
    end
```

Listing 4.7: Example creating a zoomable view of a TRAFFIC_MAP

4.3.5 Summary

The new visualization part of the TRAFFIC library provides clients with the possibility to easily set up a visualization for the model of the TRAFFIC library. It is possible now to have more than only one view of the same map at a time. The map widget observes the model and therefore updates the view when there are changes.

The implementation of the new visualization part of the library has not yet been finished. There are still some parts that need improvement. For example places and line sections are currently visualized only as rectangles and polylines. The old library supported a way to highlight places using the displayer classes that are not present anymore in the new library. Therefore, I suggest to introduce some view classes for places and line sections that make it easy to highlight them.

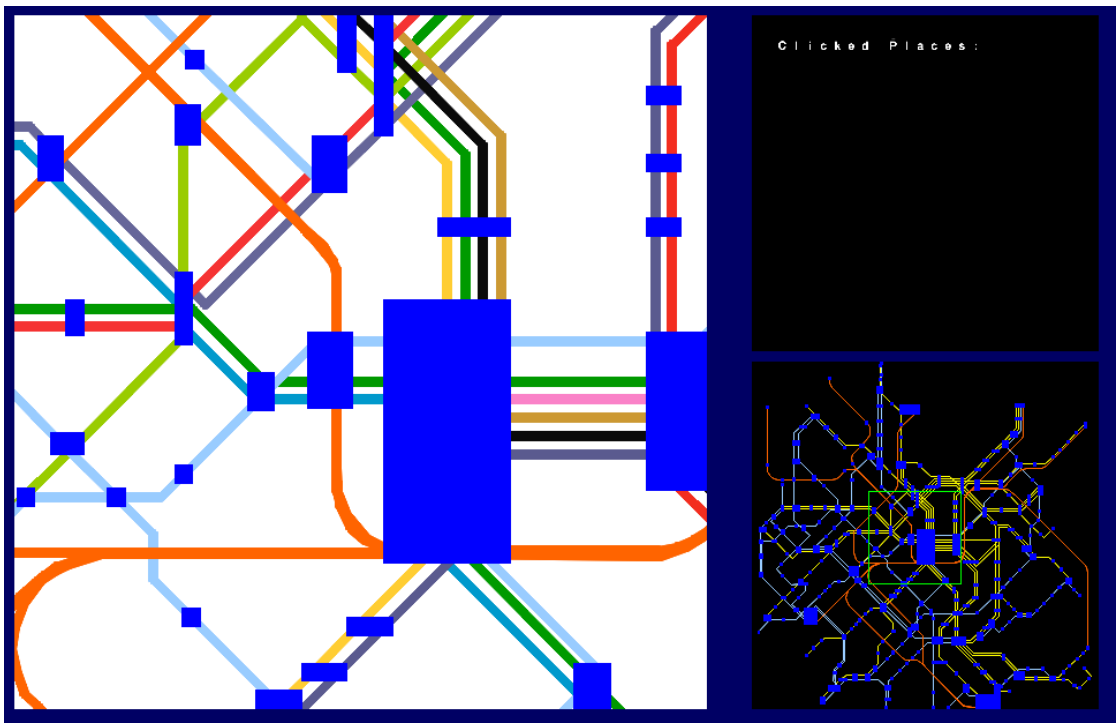

Figure 4.3: Two views of the same traffic map (Zurich), left side: zoomed with colored lines and white background, right side: different colors for different traffic types and black background

Chapter 5

Conclusion

5.1 Summary

I extended the ESDL library such that it can be used for teaching introductory programming. There is now the possibility to draw figures like lines and polygons. I introduced a drawing interface that defines the commands a drawable object is allowed to use when drawing itself. The drawing interface also has the possibility to change the coordinate system onto which drawings are performed. This makes it very easy for composite drawable objects to draw all child objects with respect to arbitrary coordinates. Using this mechanism I introduced a container that can zoom or scroll all its contained drawable objects. Furthermore, I improved the event loop of ESDL and implemented the queries of the mouse and keyboard event classes. There is now a mechanism to publish mouse events to all drawables belonging to an interactive scene. This makes it possible to implement widgets like ESDL_ZOOMABLE_WIDGET that allows the user to zoom and scroll its content using the mouse. Finally, I introduced some base classes that facilitate starting a new ESDL application without copying code from existing ESDL examples.

Based on these extensions to ESDL I redesigned the visualization part of TRAFFIC. The new design allows to create multiple views of a map model. Clients can easily customize the way the map items are visualized through passing renderer functions to the map widget.

Since all the classes I developed are assumed to be handed out to students, I always paid attention to produce good quality of code. To achieve this I wrote all class interfaces very carefully including contracts and header comments. I always chose meaningful names, in most cases even for local variables. Needless to say that I tried to stick to the Eiffel style guidelines, as introduced in chapter 26 of [13]: “A sense of style”.

Furthermore, I wrote a guide to introduce students to ISE EiffelStudio, the development environment for Eiffel. This “Eiffel Studio Student Guide” is written in German since most of the students at ETH are German speaking. The guide is intended to help the students with their first steps in using a development environment. The student guide can be found in the appendix.

During my time at the Chair of Software Engineering I also had the pleasure to accompany the course “Introduction to Programming” as a helping assistant. It was very interesting to see how the students learned to program using an object-oriented framework. I think that the Inverted Curriculum has a lot of advantages to introduce students to the art of programming. One of the major advantages from my point of view is probably that the students have much more fun when they start to program by using libraries that enable them to produce graphical effects. The only problem is that designing such libraries for the students is very difficult.

I hope that my hard work will be appreciated by hundreds of new students starting to study computer science in the forthcoming winter semester at ETH.

5.2 Future Work

The ESDL library has been improved such that it is ready to be used by students that start to program. Nevertheless, there are a lot of possibilities to continue the development of the ESDL library and also to improve some of the parts that I implemented:

- The implementation for drawing lines with a line width of more than one pixel is not very efficient. More about this issue can be found in subsection 3.6.2
- The mouse publishing mechanism that I introduced might need to be improved. For example a drawable object does not get an event when the mouse has been moved out of it.
- Vector fonts are still missing.

Furthermore, the time of my thesis was not sufficient to finish the development of the TRAFFIC library. The extension of ESDL was more time consuming than expected. There are still some parts that need to be improved in the visualization part of TRAFFIC:

- Review of the map visualization. Some features are not implemented clean and efficiently yet.
- Introduce special view classes for places and line sections instead of using figures to make it possible to highlight or even animate them.
- It would be possible to introduce style sheets for the map views. This would allow to store color of the lines and other style properties inside a separated style file instead of in the same file as the map model. Different styles could then be attached to different views of the map.

To complete the framework for teaching introductory programming the following has to be developed:

- The applications TOUCH and FLAT_HUNT have to be reimplemented using the new libraries.
- A student guide to introduce the students to the new framework needs to be developed.
- Some of the exercises have to be rewritten for the new framework.

Acknowledgments

I would like to thank my supervisors Michela Pedroni and Till G. Bay for their support and valuable feedback. I am grateful to Prof. Bertrand Meyer for giving me the opportunity to do my master thesis in the area of teaching. Furthermore I want to thank all people at the Chair of Software Engineering. Working in this group has been a pleasure.

Special thanks go to Olivier Jeger and Joseph Ruskiewicz for their valuable inputs and to Sibylle Aregger for the collaboration.

I also would like to thank Hans Dubach for always being a great help in all administrative questions during the whole study at ETH.

Finally, I am deeply grateful to my parents for their support in every respect.

References

[1] Bertrand Meyer. *The Outside-In Method of Teaching Introductory Programming*, 2003.
http://www.inf.ethz.ch/~meyer/publications/teaching/teaching-psi.pdf.

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

[3] 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.

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

[5] Eiffel Software. Eiffel.
http://www.eiffel.com.

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

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

[8] SDL. Simple DirectMedia Layer.
http://www.libsdl.org.

[9] ESDL Games.
http://se.inf.ethz.ch/download/games.

[10] Sibylle Aregger. Redesign of the traffic library. ETH Zurich, 2005.
http://se.inf.ethz.ch/projects/sibylle_aregger.

[11] Mark Guzdial, Elliot Soloway. *Teaching the Nintendo Generation to Program. Communications of the ACM*, 45(4):17–21.

[12] Eric Bezault. GOBO.
http://www.gobosoft.com.

[13] Bertrand Meyer. Object-Oriented Software Construction, 2nd edition. Prentice Hall PTR, 1997.

[14] Andreas Leitner. EWG - Eiffel Wrapper Generator.
http://ewg.sourceforge.net.

Appendix A

Eiffel Studio Student Guide

The following guide is intended to assist the students with their first steps in the ISE EiffelStudio development environment. It is written in German since most of the students at ETH are German speaking.

The guide only covers the most important topics the students have to know when doing their first steps in a development environment. It is intended to be a base for further extensions. I hope that someone will continue the writing of this student guide, for example by adding additional chapters about more advanced topics.

You can download the guide from http://se.inf.ethz.ch/projects/rolf_bruderer