Chair of Software Engineering

Traffic 3.1 - Designing Suitable Examples

Semester Thesis

By: Sarah Hauser
Supervised by: Michela Pedroni
Prof. Bertrand Meyer

Student Number: 01-908-912

Informatik
Computer Science

Table of Contents

1 Introduction

1.1 Personal Motivation

In the fourth semester I followed Professor Bertrand Meyer's lecture entitled "Informatik IV". At the time I managed to write some short programs and had only a vague idea of object-oriented programming. Nevertheless, I was completely lacking in understanding what good design decisions are and how to use techniques to produce reusable, extendible and reliable software. Many questions were answered in Meyer's book *Object Oriented Software Construction* [2]. I was fascinated by his mathematical approach to explaining the practice of object-oriented design and programming. At first, he taught us the theoretical basis of abstract data types. With this simple abstraction, the various pieces I knew of object technology began to come together and make sense, allowing me to understand enough to be able to proceed.

In a simple, natural way, several examples were presented which magically turned all these complicated ideas into a set of clear, concrete concepts.

This shows how important it is to break things down into core concepts, not to over-specify, and to use simple examples to illustrate theories.

1.2 Scope of work

1.2.1 Overview

The Chair of Software Engineering at the ETH Zurich is developing a programming module in order to support its first semester students in learning object-oriented programming concepts and software engineering issues. The module follows the objects-first teaching approach called the Inverted Curriculum. Working with a large software framework, students reuse existing components and gain experience with programming concepts by studying, reusing or modifying examples. This framework supports the development of multimedia and graphics applications both to attract students' interest in programming, and to allow them to enjoy their work.
In the first semester course "Introduction to Programming", students use *Traffic* [4], a software system that models transportation in a city. Professor Meyer has written an associated introductory programming textbook entitled *Touch of Class* [5]. This provides an introduction to programming and software engineering concepts.

1.2.2 Subject of this semester thesis

The textbook *Touch of Class* contains a range of illustrative examples. The subject of this semester thesis is to replace existing examples (or find new ones) that can be implemented using *Traffic*. The challenge is to collect core concepts and define suitable examples using existing codes or develop new parts for *Traffic*. As students learn by imitation, the examples have to be well designed and implemented. They should be simple, clear and well-documented. At the moment, the textbook consists of 31 chapters and examples exist for three chapters. The first step was therefore to study the textbook, gain insight into the Traffic framework and find suitable examples for some core concepts. One example for the concept of inheritance and two other examples had to be defined. Details of the ideas had to be collected because they could help other people to implement further examples.

2 Initial Situation

2.1 Existing System

The *Traffic library* was written to model a city and its public transportation system. It supports a city map with places, buildings, traffic lines and travelers.

There exist some sample applications:

- The Flathunt application is a simple adaptation of the well-known board game "Scotland Yard".
- The city_3d application visualizes the city with its traffic lines and buildings in 3d.
- The city_time application introduces the concept of time.

There are three applications used as examples in the *Touch of Class* textbook:

- The chapter2_ preview application shows feature calls.
- The chapter3_commands application illustrates commands.
- The chapter3_queries application illustrates queries.

For more information about the *Traffic* framework, refer to the online documentation [5]. The examples are available online on the *Traffic* homepage [4].

2.2 Proposed Work

I propose an inheritance example showing the means of transportation family. This example also introduces the concept of deferred classes. In addition, this domain contains multiple inheritance examples.
To continue, I propose an example for the use of event-based programming.

3 Result

3.1 Inheritance Example

"A mechanism whereby a class is defined in reference to others, adding all their features to its own." [2]
Inheritance is one of the central components in object technology. It supports the reusing of code and extendibility of classes. Inheritance can be used to define classes by extension, specialization and combination with other classes.
An example of inheritance should explain the following principles:

- Sub-typing
- Contracts with inheritance
- Extension of a class by adding new features
- Reusing of features by inheriting them from a class
- Redefinition of features in subclasses
- Dynamic binding
- Multiple Inheritance

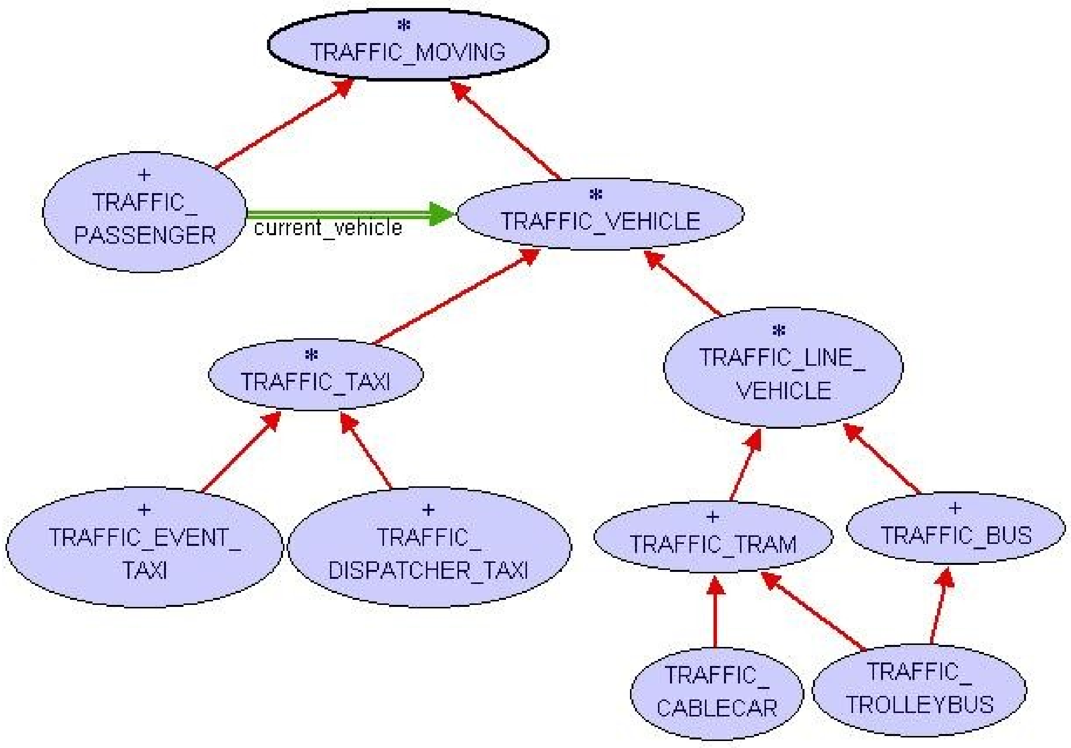
The topic of inheritance is closely related to the topic of deferred classes.
I will therefore present an example of deferred classes in this section, too.

3.1.1 Sub-typing

"A subtype must require no more and promise no less than its supertype." [2]

The inheritance example I provide models the family of various means of transportation. It reuses the existing class `TRAFFIC_MOVING` and extends it. `TRAFFIC_MOVING` defines a moving thing on the map. This can be anything that moves on the map, e.g. a traveler or a vehicle. This inheritance example builds a hierarchy of types that defines special moving objects like trams, buses, taxis, or humans walking around etc.

To return to the above definition: wherever you expect a `TRAFFIC_MOVING` you may also use a `TRAFFIC_TRAM`, for example, as the interface offered by `TRAFFIC_MOVING` is also supported by `TRAFFIC_TRAM`.



3.1.2 Adding features

The subclasses of `TRAFFIC_MOVING` are special kinds of moving objects. Each one has its own purpose and therefore supports new features.

`TRAFFIC_PASSENGER`:
A passenger moves on the map by using a means of transportation. A passenger therefore adds features like `current_vehicle`, `board` and `deboard`.
By the way, a passenger can also walk around reusing the features `move`, introduced in section 3.1.3, and `take_tour`, in section 3.1.4. In section 4.1.3, I describe the future work on `TRAFFIC_PASSENGER`.

`TRAFFIC_VEHICLE`:
A vehicle is also a moving object, as it can move on the map. A vehicle is, however, a special kind of moving object. It can carry something. A tram, for example, can carry passengers, or a truck can carry cargo. Therefore, a vehicle adds attributes like `capacity`, `current_load` and methods `load` and `unload`.

TRAFFIC_TAXI:
A taxi is a specialization of a vehicle, as it adds attributes such as taxi_office and is_busy and the method take to fulfill a request.

TRAFFIC_LINE_VEHICLE:
A line-bound vehicle is a special kind of vehicle. Like any vehicle it carries passengers or cargo, but it is not as free as a general vehicle as it is bound to a distinct transportation line. A line-bound vehicle adds the features line and schedule.

TRAFFIC_TRAM:
A tram is a special kind of line-bound vehicle carrying passengers. A tram is line-bound but a tram can do more. During rush hours it is possible to increase a tram's capacity by adding new wagons. Therefore a tram adds the features wagons, wagons_limitation, add- and remove_wagon.

TRAFFIC_BUS:
A bus, like the tram, is a special kind of line-bound vehicle. Buses can be used to replace other vehicles, e. g. if a tram-line gets blocked. A bus therefore adds a replace feature.

3.1.3 Reusing features

Subclasses inherit all the features provided by their ancestors.
Some subclasses reuse the features provided by their ancestors in their own features.

Some examples:

The TRAFFIC_MOVING has a feature polypoints that defines a list of coordinates through which the moving object will go. In addition, features like origin, destination and speed are defined in the TRAFFIC_MOVING and reused by its subclasses.
The feature move moves the moving object from origin to destination.

TRAFFIC_LINE_VEHICLE defines the feature set_line_route (a_line: TRAFFIC_LINE). This feature sets the polypoints to follow the route given by the parameter a_line. Here, the line vehicle reuses the polypoints from its superclass.

A bus can serve as a replacement bus when a traffic line is interrupted. For example, a bus can replace a broken tram line. To support this functionality, TRAFFIC_BUS has a feature replace (a_line: TRAFFIC_LINE). In this feature, the bus reuses the set_line_route feature provided by TRAFFIC_LINE_VEHICLE to set its polypoints in order to follow the new line.

Reuse is supported if inherited features are also available as external features for clients of the subclasses. Here are some further examples:

TRAFFIC_VEHICLE defines features capacity, current_load, load and unload which are reused by all its subclasses TRAFFIC_TRAM, TRAFFIC_BUS, TRAFFIC_TAXI etc.

The TRAFFIC_TAXI office defines a feature call(origin: TRAFFIC_PLACE; destination: TRAFFIC_PLACE). Clients can use this call method to order a taxi to an address origin to go to destination. The taxi office then passes the request on to the nearest available taxi. The call feature is reused by the subclasses TRAFFIC_EVENT_TAXI and TRAFFIC_DISPATCHER_TAXI.

3.1.4 Dynamic Binding and deferred classes

Consider again the TRAFFIC_MOVING hierarchy. Moving things like trams or passengers move around in a city. To simulate this using the *Traffic* library, you can apply take_tour to a TRAFFIC_MOVING. The underlying dynamic binding mechanism then picks the appropriate version depending on the dynamic type of the moving object, it could be a passenger or kind of vehicle.

TRAFFIC_MOVING is a general notion of a passenger or a vehicle. It does not make sense to define a general take_tour feature. Some vehicles like public transportation take a tour by adhering to a timetable, other objects like human beings can just walk around. Some moving objects use streets like buses, others such as trams use tracks etc.

Therefore, TRAFFIC_MOVING is a deferred class and the feature take_tour is also deferred. Each particular moving thing has its own behavior to take a tour on the map and will have to implement the take_tour feature to define it.

“If a class is marked as deferred it must have at least one deferred feature. But a class may be deferred even if it does not declare any deferred feature of its own: it might inherit a deferred feature that it does not effect.” [2]

TRAFFIC_VEHICLE inherits take_tour from TRAFFIC_MOVING. TRAFFIC_VEHICLE is still a general notion covering line-bound vehicles like trams and other vehicles like taxis. Therefore, TRAFFIC_VEHICLE still does not effect take_tour and is also a deferred class.

Classes that effect take_tour are the TRAFFIC_PASSENGER that just walks around

on the map or uses a vehicle and the `TRAFFIC_LINE_VEHICLE` that pursues a line and a schedule.
Here, I will not provide any code. However, you may want to have a look at the code documentation [5] or the *Traffic* framework [4].

The same is the case with the `TRAFFIC_TAXI`: taking a tour, the taxi has to keep its office informed about its state, for example, whether it is busy or not. But the general `TRAFFIC_TAXI` has not defined how to communicate with its taxi office. Therefore, `TRAFFIC_TAXI` does not effect `take_tour`. Its subclasses `TRAFFIC_EVENT_TAXI` and `TRAFFIC_DISPATCHER_TAXI` effect it. The first uses event based communication, the latter method call communication. For more about this and another example of deferred classes, see section 3.2.3.

3.1.5 Contracts with inheritance

“Assertions will give us insights into the nature of inheritance. It is in fact not an exaggeration to state that only through the principles of Design by Contract can one finally understand what inheritance is really about.” [2]

Contracts of a class are the pre- and postconditions of exported routines and class invariants. In a descendant class, all contracts of its ancestors still apply.

3.1.5.1 Class Invariants

A class invariant defines an assertion that must be satisfied whenever the object is in a stable state. The class invariant must be true on creation of an instance of the class. Further, it must be satisfied on entry to and on exit from each exported routine in the class.
A descendant class combines its ancestors' invariants and its own invariant: ”The invariant property of a class is the boolean and of the assertions appearing in its invariant clause and of the invariant properties of its parents, if any.” [2]

For example the `TRAFFIC_VEHICLE` ensures that a vehicle cannot have a negative `capacity`, and that the `load` has to be smaller than its `capacity`.
The `TRAFFIC_TRAM` adds new invariants ensuring that a tram cannot ride on a bus line, and further that it has a `wagons` array, and does not pull more wagons than its `wagon_limitation` allows.

```
indexing
  description: "Objects to transport cargo or passengers"
deferred class TRAFFIC_VEHICLE inherit
               TRAFFIC_MOVING
...
invariant
  capacity_non_negative: capacity >= 0
  load_smaller_or_equal_than_capacity: current_load <= capacity
end -- class TRAFFIC_VEHICLE

indexing
  description: "Tram"
class TRAFFIC_TRAM inherit
      TRAFFIC_LINE_VEHICLE
...
invariant
  valid_line_type: line.type.name.is_equal ("tram")
  wagons_not_void: wagons /= void
  wagons_count_allowed: wagon_limitation >= wagons.count
end -- class TRAFFIC_TRAM
```

3.1.5.2 Pre- and Post-conditions

“Preconditions are assertions attached to a routine, which must be guaranteed by every client prior to any call to the routine. Postconditions are assertions attached to a routine, which must be guaranteed by the routine's body on return from any call to the routine.” [2]
Inherited preconditions can only become weaker, whereas postconditions only become stronger.
“A routine re-declaration may only replace the original precondition by one equal or weaker, and the original postcondition by one equal or stronger.” [2]

3.1.6 Redeclaration and Redefinition

The `TRAFFIC_VEHICLE` has a private attribute `unit_capacity` which defines the maximum possible load the vehicle is able to carry. It defines a function `capacity` to query the quantity it is possible to load.
Usually, a tram is an arrangement of a motorized carriage and a number of wagons. Some trams do not have wagons attached. The motorized carriage is also a wagon and carries a number of passengers.

Therefore, both motorized carriage and wagons have a capacity. The overall capacity of a tram is the addition of the capacity of the motorized carriage with the capacity of its wagons. In the code I use `engine` as identifier for the motorized carriage.

As `TRAFFIC_TRAM` inherits `TRAFFIC_VEHICLE` the `capacity` function inherited from `TRAFFIC_VEHICLE` does not take into account the attached `wagons`. Therefore the `capacity` feature has to be redefined to include the capacities of the `wagons`.
For better naming conventions I have renamed the `unit_capacity` to `engine_capacity`. The `engine_capacity` defines the capacity of the pulling engines wagon.
The redefined version of the `capacity` feature in `TRAFFIC_VEHICLE` takes the `engine_capacity` and loops through its `wagons` array to add the `capacity` of each wagon.
This is an example of redefinition for the sake of correctness, as it is incorrect to return only the capacity of its first wagon for the capacity of a tram.

```
indexing
  description: "Objects used to transport cargo or passenger"
deferred class TRAFFIC_VEHICLE inherit
               TRAFFIC_MOVING
feature --Access
...
  capacity:INTEGER is
      -- Maximum possible load
    do
      Result := unit_capacity
    end
       ...
feature{NONE} -- Implementation
  unit_capacity: INTEGER
      -- Maximum load motor can manage
...
invariant
  unit_capacity_non_negative: unit_capacity >= 0
  load_smaller_or_equal_than_capacity: current_load <= capacity
end -- class TRAFFIC_VEHICLE
```

```
indexing
  description: "Tram"
class TRAFFIC_TRAM inherit
      TRAFFIC_LINE_VEHICLE
       rename unit_capacity as engine_capacity
       redefine capacity end
feature -- Access
  wagon_limitation: INTEGER
      -- Maximum number of wagons allowed
  wagons: ARRAYED_LIST[TRAFFIC_WAGON]
      -- Attached wagons
feature -- Basic operations
  capacity: INTEGER is
      -- Wagon capacities plus and engine_capacity
    local
      cap: INTEGER
    do
      cap := engine_capacity
      from
       wagons.start
      until
       wagons.after
      loop
       cap := cap + wagons.item.capacity
       wagons.forth
      end
      wagons.start
      Result := cap
    end
  add_wagon is
      -- Attach new wagon
    require
      wagons_not_full: wagon_limitation >= wagons.count + 1
    local
      wagon: TRAFFIC_WAGON
    do
      wagon := create {TRAFFIC_WAGON}.make_default
```

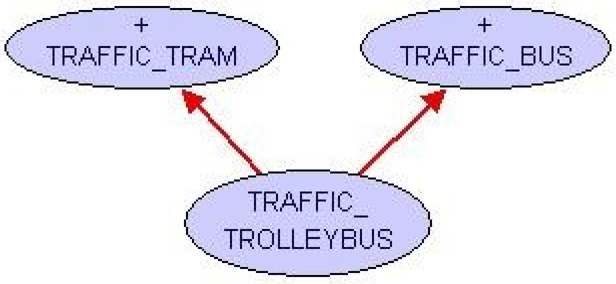
```
            wagons.force (wagon)
          ensure
            wagon_added: wagons.count = old wagons.count + 1
          end
      remove_wagon(i: INTEGER) is
            -- Remove wagon at position i
          require
           wagons_not_empty: wagons.count > 0
          do
           wagons.start
           wagons.go_i_th (i)
           wagons.remove
          ensure
           wagon_removed: wagons.count = old wagons.count -1
          end
  invariant
      wagons_not_void: wagons /= Void
      wagons_count_allowed: wagon_limitation >= wagons.count
    ...
  end -- class TRAFFIC_TRAM
```

3.1.7 Multiple inheritance

This is defined as “the unrestricted form of inheritance, whereby a class may have any number of parents.” [2]

A trolleybus increases its capacity by adding wagons like a tram. In contrast to a normal bus, however, the trolleybus is powered by electricity, and therefore has more power and is able to carry more weight.
A trolleybus does not run on tracks and has the ability to uncouple itself from the contact wire to drive freely. A trolleybus, then, is as flexible as a normal bus. Like a bus, a trolleybus can serve as a replacement vehicle in case of breakdowns on the traffic line.

Therefore `TRAFFIC_TROLLEYBUS` inherits both `TRAFFIC_TRAM` and `TRAFFIC_BUS`. It takes advantage of reusing the features `wagons`, `add`- and `remove_wagon` and the redefined `capacity` feature of the tram, and the `replace` feature to serve as replacement trolleybus from bus.



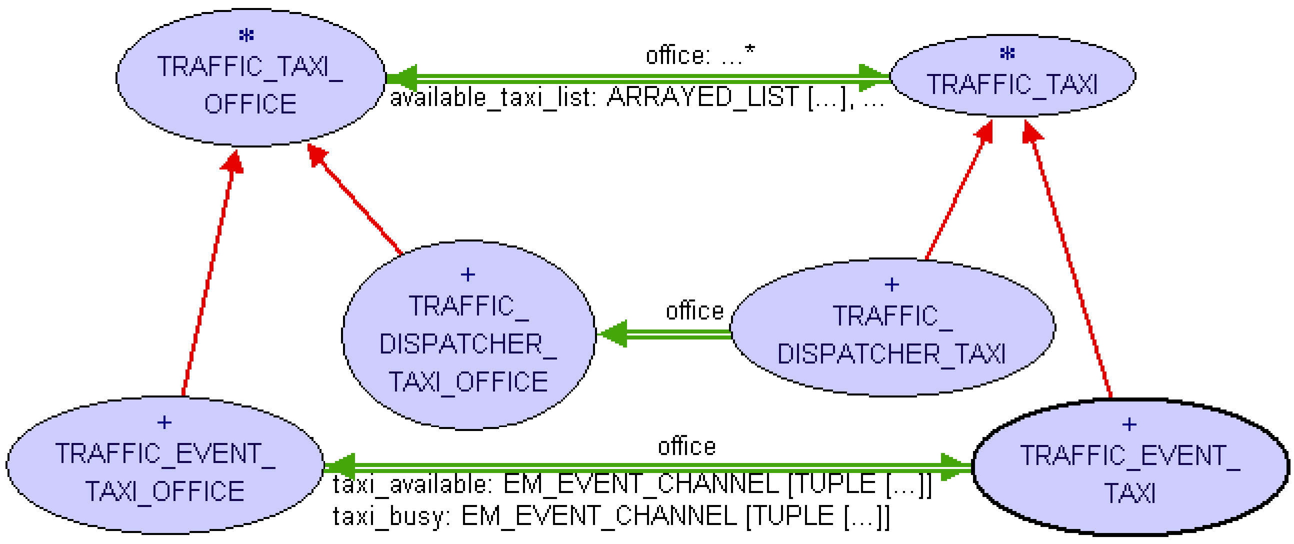
Other possible examples for multiple inheritance could be:

- Cargo tram inherits from disposal and tram.
- Eating tram inherits from restaurant and tram.
- Party tram inherits from party and tram.
- Caravan inherits from flat and bus.
- Emergency car inherits from emergency room and car

3.2 Taxi example

"Dispatchers are communication centers responsible for receiving and transmitting messages, tracking vehicles and equipment, and recording other important information." [7]
Many establishments like the police and fire departments, hospitals, trucking organizations, train stations and even taxi company offices use dispatchers to coordinate their operations.
Taxi dispatchers dispatch taxis in response to requests for services.
A taxi dispatching mechanism should of course be added to the *Traffic* framework.
In reality, communication between clients and the taxi office is usually by telephone.
The communication between taxi office and taxis is by radio.

In *Traffic*, a taxi office object takes care of dispatching taxis. To show different approaches in object-oriented software design, I will provide two possible solutions to simulate a taxi dispatch, one where the objects communicate with each other using event-based communication, and the other using client feature call communication.



I provide deferred classes `TRAFFIC_TAXI_OFFICE` and `TRAFFIC_TAXI`.
The code of the deferred feature `take` of `TRAFFIC_TAXI` and its effective versions of `TRAFFIC_EVENT_TAXI` and `TRAFFIC_DISPATCHER_TAXI` is given in section 3.2.3.

3.2.1 Taxi events

The `TRAFFIC_EVENT_TAXI` and the `TRAFFIC_EVENT_TAXI_OFFICE` use event-based communication.

Events:

```
request: EM_EVENT_CHANNEL[TUPLE[EM_VECTOR_2D, EM_VECTOR_2D]]
      -- Event when taxi needed, contains address and intended
      -- destination coordinates

taxi_available: EM_EVENT_CHANNEL[TUPLE[TRAFFIC_EVENT_TAXI]]
      -- Event when taxi gets available

taxi_busy: EM_EVENT_CHANNEL[TUPLE[TRAFFIC_EVENT_TAXI]]
      -- Event when taxi gets busy
```

```
reject_request: EM_EVENT_CHANNEL[TUPLE[EM_VECTOR_2D, EM_VECTOR_2D]]
      -- Event when taxi rejects taking a request,
      -- contains address and intended destination coordinates
```

I chose to let the taxi office provide these events.

Publishers:

If a taxi is needed, a customer fires the `request` event.
The `TRAFFIC_EVENT_TAXI` publishes a `taxi_available` event when it is available to serve a request. It publishes a `taxi_busy` event when it gets busy and it publishes a `reject_request` event when it rejects to take a request.

Subscriber:

The `TRAFFIC_EVENT_TAXI_OFFICE` subscribes for all the events described above:

```
request.subscribe(agent call(?, ?))
taxi_available.subscribe(agent enlist(?))
taxi_busy.subscribe(agent delist(?))
reject_request.subscribe(agent recall(?,?))
```

Whenever a `request` event is fired, the `call` method of the taxi office handles the event and receives the request. The taxi office then determines where the nearest taxi is, and forwards the request to this one.
When a taxi gets busy, it fires the `taxi_busy` event and the taxi office handles the event, and takes the taxi out of its list of available taxis. When the taxi becomes available again, it fires the `taxi_available` event, and the taxi office deals with it, putting the taxi into the list of available taxis again.
When the taxi office forwards a request to a taxi that has just taken a passenger on board, the taxi may fire the `reject_request` event and the taxi office will handle it by recalling the request.

The advantage of event-based communication is that we may define events wherever we want. Several taxi offices can subscribe to the same events and clients publish a request event without caring about which taxi office will deal with it.
Events use asynchronous calls to let the caller proceed before the call is completed.

3.2.2 Taxi dispatcher

The `TRAFFIC_DISPATCHER_TAXI_OFFICE` and `TRAFFIC_DISPATCHER_TAXI` use client relationship and feature call communication.

"Let S be a class. A class C which contains a declaration of the form a:S is said to be a client of S. S is then said to be a supplier of C. " [2]

A customer orders a taxi by calling the `call` feature on a taxi office. The taxi office forwards the request to the nearest taxi by calling the feature `take` on the taxi. The taxi then serves the request or calls the `reject` feature on its associated office. When a taxi becomes busy it calls the `delist` feature and the `enlist` feature when it becomes available again.

For example, a customer is a client of the taxi office. The taxi office provides a service to its customers.

In contrast to event based taxi dispatch, this solution uses synchronous calls. A feature call forces the caller to wait until it is completed.
The advantage of this example is that it models reality. In reality, you call a particular taxi office. You do not send a request out and any one of several taxi offices deals with it. Also, a taxi calls its particular taxi office to inform it about its current state. But because of synchronous communication, this version is not as flexible as the above alternative.

3.2.3 City_taxi application

The city_taxi application shows how taxi dispatch works. You can click on a place on the map and a taxi will pick you up. There are two taxi offices at your beck and call: an `EVENT_TAXI_OFFICE` with several `EVENT_TAXI`s colored in yellow and colored in turquoise, a `DISPATCHER_TAXI_OFFICE` with `DISPATCHER_TAXI`s.
For the sample application city_taxi have a look at the examples on the *Traffic* homepage [4].

A code example for the deferred method take:

`TRAFFIC_TAXI` has the deferred feature `take` which is called by the taxi office to forward a request to the nearest taxi to serve it.

```
take(from_location: EM_VECTOR_2D; to_location: EM_VECTOR_2D) is
      -- Pick up someone at from_location, bring to to_location.
    require
     from_location_not_void: from_location /= Void
     to_location_not_void: to_location /= Void
    deferred
    end
```

TRAFFIC_EVENT_TAXI effects it to and uses events.

```
take(from_location: EM_VECTOR_2D; to_location: EM_VECTOR_2D) is
      -- Fulfill request, set busy and call taxi_busy or call
      -- reject_request.
     do
      if not busy then
       set_request_information(from_location, to_location)
       busy := True
       office.taxi_busy.publish ([Current])
      else
       office.reject_request.publish([from_location, to_location])
      end
     end
```

TRAFFIC_DISPATCHER_TAXI effects it and uses method call communication

```
take(from_location: EM_VECTOR_2D; to_location: EM_VECTOR_2D) is
      -- Set busy, fulfill request or let office recall request.
     do
      if not busy then
        set_request_information(from_location, to_location)
        busy := True
        office.delist(Current)
      else
        office.recall(from_location, to_location)
      end
     end
```

3.2.4 Covariance

"The policy allowing a feature redeclaration to change the signature so that the new types of both arguments and result conform to the originals." [2]

The TRAFFIC_EVENT_TAXI_OFFICE should accept in its taxi list only TRAFFIC_EVENT_TAXIs as they are the ones using event-based communication. As TRAFFIC_EVENT_TAXI_OFFICE inherits TRAFFIC_TAXI_OFFICE it needs to redefine the enlist method.

```
indexing
  description: "Taxi office"
deferred class TRAFFIC_TAXI_OFFICE
...
feature --Basic Operations
  enlist(a_taxi: TRAFFIC_TAXI)is
       -- Put a_taxi in available list.
     require
      a_taxi_not_busy: a_taxi.busy = False
     do
      available_taxi_list.extend(a_taxi)
     ensure
      a_taxi_added: available_taxi_list.count = old
     available_taxi_list.count + 1
     end
...
end -- class TRAFFIC_TAXI_OFFICE


indexing
  description: "Taxi office, that customers and taxis
          -- communicate with using events"
class TRAFFIC_EVENT_TAXI_OFFICE inherit
       TRAFFIC_TAXI_OFFICE
        redefine enlist end
...
feature -- Basic Operations
  enlist(a_taxi: EVENT_TRAFFIC_TAXI) is
       -- Put event taxi into available list.
     do
      available_taxi_list.extend(a_taxi)
     end
...
end -- class TRAFFIC_EVENT_TAXI_OFFICE
```

4 Conclusion and Perspectives

4.1 Future Work

4.1.1 Enhancement of inheritance example

A suitable example of “redefinition for efficiency” and for “redefinition for correctness” could be: vehicles can provide information about how long it takes to go to somewhere, and how much it will cost. The following is a possible scenario: somebody wants to go from the main station to Bellevue. He asks a tram, a train and a taxi the time they need to take him there, and how much it costs. Depending on this information, he then decides which transportation to use.
The following is a short sketch: vehicles provide the features `time` and `price`. These features take the parameters `origin` and `destination` and return the time and costs to travel from origin to destination.
A normal vehicle will use a rather complicated shortest path algorithm on the map, and the computation of its average speed to calculate the time. Further, it will use something like average costs per kilometer to compute the price. A line vehicle will redefine both features `time` and `cost`. For greater efficiency, it can just look at its schedule to return the time needed. For correctness, it looks at its tariffs to give information about prices.
For the following overview, I simplified things so that a line vehicle has only one pay scale group, e.g. a day ticket. The units used are: shortest_path [m], average_speed [m/min] and average_cost [Fr/m].

Code sketch:

```
class TRAFFIC_MOVING
...
feature --Access
  map: TRAFFIC_MAP
       -- Map Current moves on
...
end -- class TRAFFIC_MOVING

class TRAFFIC_VEHICLE
...
feature -- Basic operations
time(a_origin: TRAFFIC_PLACE; a_destination: TRAFFIC_PLACE):INTEGER
```

```
  is
       -- Time from a_origin to a_destination
  do
     Result := map.shortest_path(a_origin, a_destination)/
     average_speed
  end
cost(a_origin: TRAFFIC_PLACE; a_destination: TRAFFIC_PLACE):INTEGER
  is
      -- Costs from a_origin to a_destination
  do
     Result := map.shortest_path(a_origin, a_destination)*
     average_cost
  end
...
end -- class TRAFFIC_VEHICLE

class TRAFFIC_LINE_VEHICLE inherit
     TRAFFIC_VEHICLE
        redefine time, cost end
...
feature --Basic operations
time(a_origin: TRAFFIC_PLACE; a_destination: TRAFFIC_PLACE):INTEGER
  is
      -- Time from a_origin to a_destination
  do
     Result := schedule.time(a_origin, a_destination)
  end
cost(a_origin: TRAFFIC_PLACE; a_destination: TRAFFIC_PLACE):INTEGER
  is
      -- Costs from a_origin to a_destination
  do
     Result := pay_scale
  end
...
end -- class TRAFFIC_LINE_VEHICLE
```

4.1.2 Taxi Exercise

As a suggested exercise, students may program an application simulating a special taxi-needed event. For example there is a party at the “Rote Fabrik”, which is now over and many are waiting for taxis on the street. A taxi informs its central office to send more taxis to this place. The central office publishes a taxi-needed event to which all its taxis are subscribed. All its taxis, therefore, receive the information, but it is their decision whether they go to the “Rote Fabrik”.
Another suggested exercise could be a taxi-challenge. The Taxi office publishes a taxi-challenge event. Every taxi receives it but the fastest taxi wins, can fulfill the request and gets money.

4.1.3 Model and View

The *Traffic* application uses an adaptation of the model view controller paradigm.
The model includes `TRAFFIC_MOVING`, `TRAFFIC_BUILDING`, `TRAFFIC_LINE`, `TRAFFIC_PLACE` and other classes. All model objects are collected on the `TRAFFIC_MAP`, the core component.
The view consists of `TRAFFIC_3D_REPRESENTATION`s. These representations are containers for the model objects to be drawn. The visualization container for moving objects is the `TRAFFIC_3D_TRAVELER_REPRESENTATION`. There, the decision as to how to represent a moving object is done in a switch statement in the `add_traveler` method: for each type of moving using the `TRAFFIC_3D_OBJECT_LOADER`, a graphical object is created and put into the container. At present, graphics for trams and passengers are supported. It is planned to add graphics for other vehicle types.
Extending the model to support other moving objects like buses, trams etc. is not that simple. For instance, to add a bus I had to create a new `TRAFFIC_TYPE_BUS,` a new model class `TRAFFIC_BUS`, adapt the switch in the `3D_TRAVELER_REPRESENTATION`, and a new graphical object for bus is needed. To improve the extendibility of the system, it is planned to substitute the `3D_TRAVELER_REPRESENTATION` by visualization containers for each type of moving object and to use factories to get the graphical objects.
This further implementation will avoid the switch and improve the extendibility of the system by using concrete model view mapping.

At the moment, the separation of model and view is not always preserved. For example moving objects do sometimes use OpenGL coordinates, but they would better use coordinates defined by the model component `TRAFFIC_MAP`. It may be a good decision to follow the model view controller paradigm strictly, in order to have less coupling and better supported reusability.

I added the feature `is_marked` to the model component `TRAFFIC_MOVING`. The

TRAVELER_3D_REPRESENTATION then draws marked moving objects bigger than normal ones. At the moment, this is only used for taxis. If no marking is needed for other moving objects this should be changed so that TRAFFIC_TAXIs with feature busy true are marked. Furthermore, it would be good to change this to highlighting busy taxis not drawing them bigger, but changing their color. However, this option for the graphical object is not yet supported. Taxis have different colors depending on their type, TRAFFIC_EVENT_TAXI or TRAFFIC_DISPATCHER_TAXI. But taxis should be assigned the same color as the taxi office for which they work. For example, a taxi office could have a corporate identity such as a number or a name on the model. This should be considered when changing the visualization component or adding a new one for taxis.

Inherited preconditions can only become weaker, whereas postconditions only become stronger. An illustrating example should be done in future work.

The TRAFFIC_PASSENGER boards or deboards a means of transportation. At present, this does not affect its tour. One possible solution is to set the new coordinates to follow the line of the vehicle using a method like set_request_information on TRAFFIC_TAXI. The other is, to let the passenger disappear until he deboards and to draw the vehicle as loaded.

4.2 Conclusion

I was immediately interested in the didactic aspect of this semester thesis, and wondered if I would manage to explain the computer science concepts it involved in a sufficiently clear and simple way. I also felt nervous about whether I would manage to achieve the objectives of the project.
In retrospect, it was not easy to fulfill this task, but it was very interesting and I learned a lot. I wish to thank Michela Pedroni, who was an admirable supervisor.

5 References

[1] Chair of Software Engineering: *Semester-/Diplomarbeiten*; Online at: http://se.inf.ethz.ch/projects/index.html, consulted in October 2002.
[2] Bertrand Meyer: *Object-Oriented Software Construction, 2nd edition*, Prentice Hall,1997.
[3] Bertrand Meyer with Michela Pedroni: *The Inverted Curriculum in Practice*, in *Proceedings of SIGCSE 2006*, ACM, Houston, Texas, 1-5 March 2006, pages 481-485.
[4] Traffic library http://se.inf.ethz.ch/traffic
[5] Traffic library developer documentation
http://se.ethz.ch/traffic/doc/traffic3.0_doc/traffic/developer/index.html

[6] Bertrand Meyer: *TOUCH OF CLASS, Learning to program well with Object Technology and Design by Contract*, *AN INTRODUCTION TO SOFTWARE ENGINEERING* http://se.inf.ethz.ch/touch
[7] U.S Departement of labor http://www.bls.gov/oco/ocos138.htm