

Transportation System

16. 02. 2001

# **Transportation System**

## **Software and hardware model**

5<sup>th</sup> semester project

Student:

**Etienne Perron**

Professor:

**Alain Wegmann**

Assistant :

**Andrey Naumenko**

**EPFL**

**ICA-DSC**

**CH-1015 Lausanne**

**Switzerland**

## Table of Contents.

<b>1.</b>	<b>Introduction .....</b>	<b>3</b>
<b>2.</b>	<b>Transportation System.....</b>	<b>4</b>
	2.1 Before getting started.....	4
	2.1.1 Contract.....	4
	2.1.2 Techniques .....	4
	2.2 Partial Results.....	5
	2.2.1 Business Implementation .....	6
	2.2.2 System Design.....	7
	2.3 Business Implementation Spec .....	11
	2.3.1 Snapshots .....	11
	2.3.2 Collaboration Model.....	19
	2.3.3 Objects Model.....	21
	2.4 IT System Spec.....	23
	2.4.1 Snapshots .....	23
	2.4.2 Use Cases Model.....	27
	2.4.3 Conceptual Model.....	32
	2.4.4 Partial System Activity.....	36
	2.5 IT System Implementation Spec .....	38
	2.5.1 Scenarios.....	38
	2.5.2 Implementation .....	44
	2.5.3 The simulator.....	47
	2.6 Java implementation.....	48
	2.6.1 The code.....	48
	2.6.2 The lookForBestCabin – algorithm.....	48
	2.6.3 Simulator Tutorial .....	49
	2.7 Conclusion .....	53
<b>3.</b>	<b>Extended Transportation System.....</b>	<b>54</b>
	3.1 Business Implementation Spec .....	54
	3.1.1 Snapshots .....	54
	3.1.2 Collaboration Model.....	58
	3.1.3 Objects Model.....	60
	3.2 System Spec .....	61
	3.2.1 Snapshots .....	61
	3.2.2 Use Cases Model.....	67
	3.2.3 Conceptual Model.....	70
	3.2.4 Partial System Activity.....	73
	3.3 Conclusion .....	74
<b>4.</b>	<b>ETS Version Two .....</b>	<b>75</b>
	4.1 Business Implementation Spec .....	75
	4.1.1 Snapshots .....	75
	4.1.2 Collaboration Model.....	78
	4.1.3 Objects Model.....	80
	4.2 System Spec .....	82
	4.2.1 Snapshots .....	82
	4.2.2 Use Cases Model.....	85
	4.2.3 Conceptual Model.....	88
	4.2.4 Partial System Activity.....	90
	4.3 Conclusion .....	93
<b>5.</b>	<b>Appendix.....</b>	<b>94</b>
	5.1 Java Code .....	94

# 1. INTRODUCTION

This semester project is divided into three main parts.

- **Transportation System:** In this first part, I built the model of the controlling software of an elevator, that communicates with the elevator's hardware using input and output events.  
The idea for this project was born because the Software Engineering Laboratory (LGL, <http://lglwww.epfl.ch>) of the EPFL. All I know about that project is that the software to control an elevator was constructed, and I received a copy of the initial contract for this system. We decided to make such a system with the modeling methods used and developed by the Institute for computer Communications and Applications (ICA, <http://icawww.epfl.ch>).  
I modeled this system using the UML notation and Catalysis, a modeling technique with different phases. For this work, I used Cool:Jex by Sterling Software (<http://www.sterling.com>). This tool is great for drawing all the needed diagrams, defining properties of the used objects, and it generates Java Code corresponding to the Class Diagrams one made.  
(You will find the diagrams produced in this part in the transport3-repository, in the project "TransportationSystem")  
I programmed the functions for this generated Code, built a graphical user interface using Visual Café by Symantec (<http://www.symantec.com>), and I designed a very simple simulator that allows you to test the functionality of my elevator software.
- **Extended Transportation System:** In this second part, I tried to model not the software, but the whole elevator (hard- and software), and I made this for a very general transportation system, without specifying if it is an elevator or some other type of system. This work was extremely confusing and difficult, but I think that in the end, I have understood well the ideas of this approach. Unfortunately, my first version of the Extended Transportation System was not very useful for the optimization task that was one of the main goals of this model. In the ETS, the parallelisms of the different transports could not be treated using UML, so that there was still an algorithm to do by hand.  
(You will find the diagrams produced in this part in the transport3-repository, in the project "ExtendedTransportationSystem").
- **Extended Transportation System, Version Two:** For the reasons mentioned above, I started a third approach, where I tried to represent parallelisms in my diagrams. The goal was not to build a working system, but to try to find a general base for different transportation systems, to compare different implementations of it and to find optimal solutions for each case, depending on the requirements of the corresponding situation.  
I went on with this part as long as possible, but my time run out, and I had to stop far from the end. But this part lead to some interesting results, just as well.  
(You will find the diagrams produced in this part in the transport3-repository, in the project "ExtendedTransportationSystemTwo").

I hope that in the end, even if the project is not finished, this report is useful for someone who wants to get started with UML, and also for people that want to model real objects using OO techniques.  
My project is also an example how the "trial and error" principle can bring completely new viewpoints of a problem. Even if I made lots of trials and had to start over from the start more or less three times, those trials were not without success. Each of them has brought a couple of new interesting results that were used in the next approach.

Finally, I enjoyed this project very much and I had a lot of fun doing it. It would be really nice if someone had the courage to continue my work where I had to stop.

## **2. TRANSPORTATION SYSTEM**

### **2.1 Before getting started**

#### **2.1.1 Contract**

The first goal of the semester project was to develop a control software for an elevator. The hardware of the elevator is precisely determined and described in the following text, that the LGL published for their elevator.

The initial contract, containing all the requirements:

The system controls multiple lift cabins that all service the same floors of a building. There are buttons to go up and down on each floor to request the lift (apart of the top-most, bottom-most floors). Inside an elevator cabin, there is a series of buttons, one for each floor. The arrival of a cabin at a floor is detected by a sensor. The system may ask the elevator to go up, go down or stop. We assume that the elevator braking distance is negligible. The system may ask the elevator to open its door and it receives a notification when the door is closed - the door closes automatically after a predefined amount of time, which simulates the activity of letting the people on and off at each floor. However, neither this function of the elevator, nor the protection associated with the door closing (stopping it from squashing people) are part of the system to realize. The capability to cancel a request has been removed from the case.

So, in this first approach, I tried to build a system that does exactly what the software of the LGL does, but without having seen it, only using the contract above. This software is the “brain” of an elevator, listening and responding to events from the hardware it controls.

#### **2.1.2 Techniques**

I used the techniques used and developed by the ICA (Institute for computer Communications and Applications) that are described in the course of Professor Alain Wegmann (<http://icawww.epfl.ch/OOAD/>). These are the UML notation, combined with the method called catalysis, that splits the modelization of a software into 4 phases:

- As-is-Analysis: description of the business before the implementation of the system. This phase was left out in my model, because an elevator without system does not exist.
- To-be-Analysis: description of the business after the implementation of the system (who works with the system in which way?). This phase is called Business Implementation Specification in my model.
- System Design: description of the structure of the system (which objects of the real world should be represented in the system, what other objects do we need, how do we enchain the different actions?). This phase is called IT System Specification in my model (IT means Information Technology).
- Object Design: description of how the system works (which object is responsible for which action, how do they collaborate?). This phase is called IT System Implementation Specification in my model.

There is, of course, a fifth phase, called Implementation, that is the programming of the classes using Java and Visual Cafe.

## **2.2 Partial Results**

As explained in the diagram comment boxes, this first approach led to an interesting result. I tried to create two Use Cases, one for each of the following actions:

- Request Transport: a TransportedObject, that is most often a person, presses a button on a floor to request a transport. A cabin is chosen by the system and sent to the correct floor.
- Use Transport: a TransportedObject presses a button in a cabin to choose a floor as a destination. The corresponding cabin is sent to the right floor.

You can see that I went on to the System Design phase, where I suddenly realized, that the two Use Cases can not be treated separately.

The problem was:

If a cabin arrives at a floor, it opens the doors, closes the doors, but: the system does not know if there is a TransportedObject in the cabin or not.

One could say that this does not matter, but: if we free the cabin immediately after the doors are closed, it could immediately be requested and moved by another TOActor. What if there was a TOActor in the cabin, but pressed the button too late? It will be transported to a wrong direction. I resolved this problem by starting a time-counter each time the doors of a cabin are closed.

Of course, this time-counter is deleted as soon as someone presses a button in the Cabin.

But, as you can see in the following diagrams, this led to a new problem: The end of one Use Case is the beginning of the next. This means: if the Use Case Request Transport is terminated by a TO (Transported Object) pressing a button in a cabin, we know certainly, that as the very next action an instance of the Use Case Use Transport will be executed. This is not legal. After the end of a Use Case instance, we never know what happens! The only solution for this problem was to unite the two Use Cases to one single Use Case.

## 2.2.1 Business Implementation

### 20:BI\_Collaboration (Collaboration Diagram)

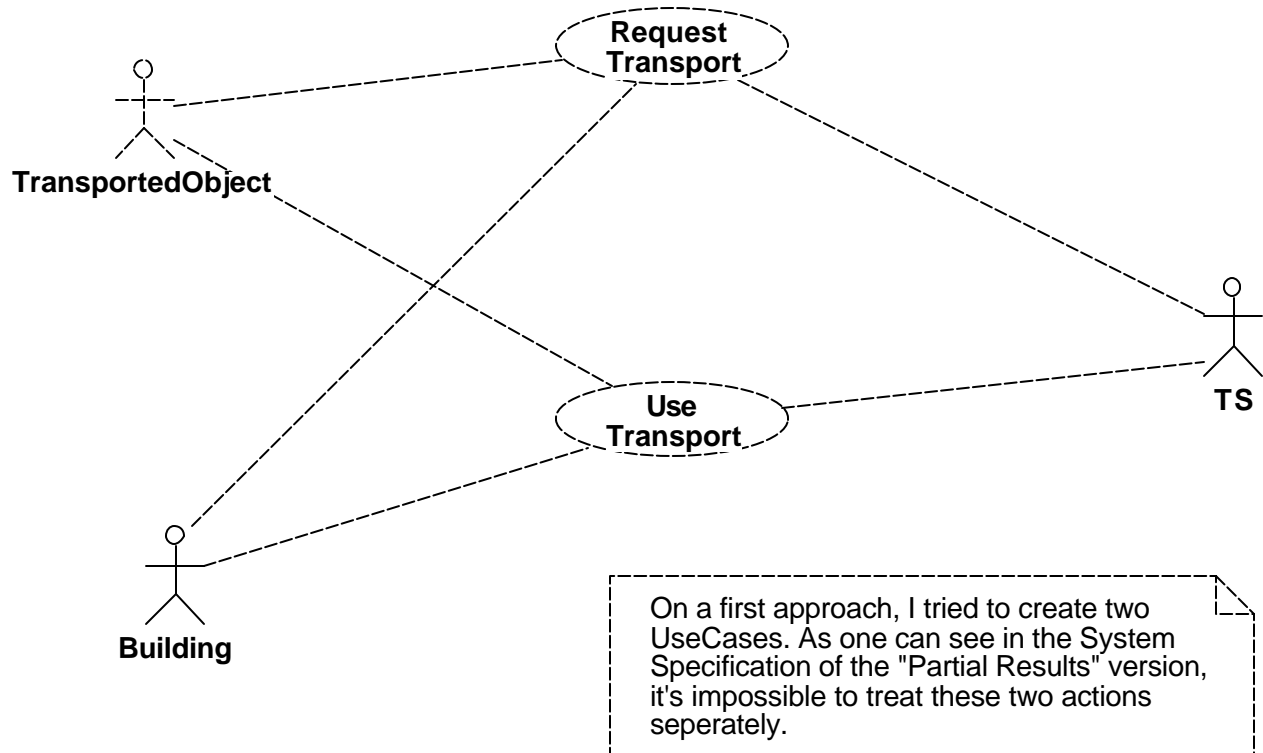


Figure 1. 20:BI\_Collaboration (Collaboration Diagram)

## 2.2.2 System Design

### 21:S\_Activity (Activity Diagram)

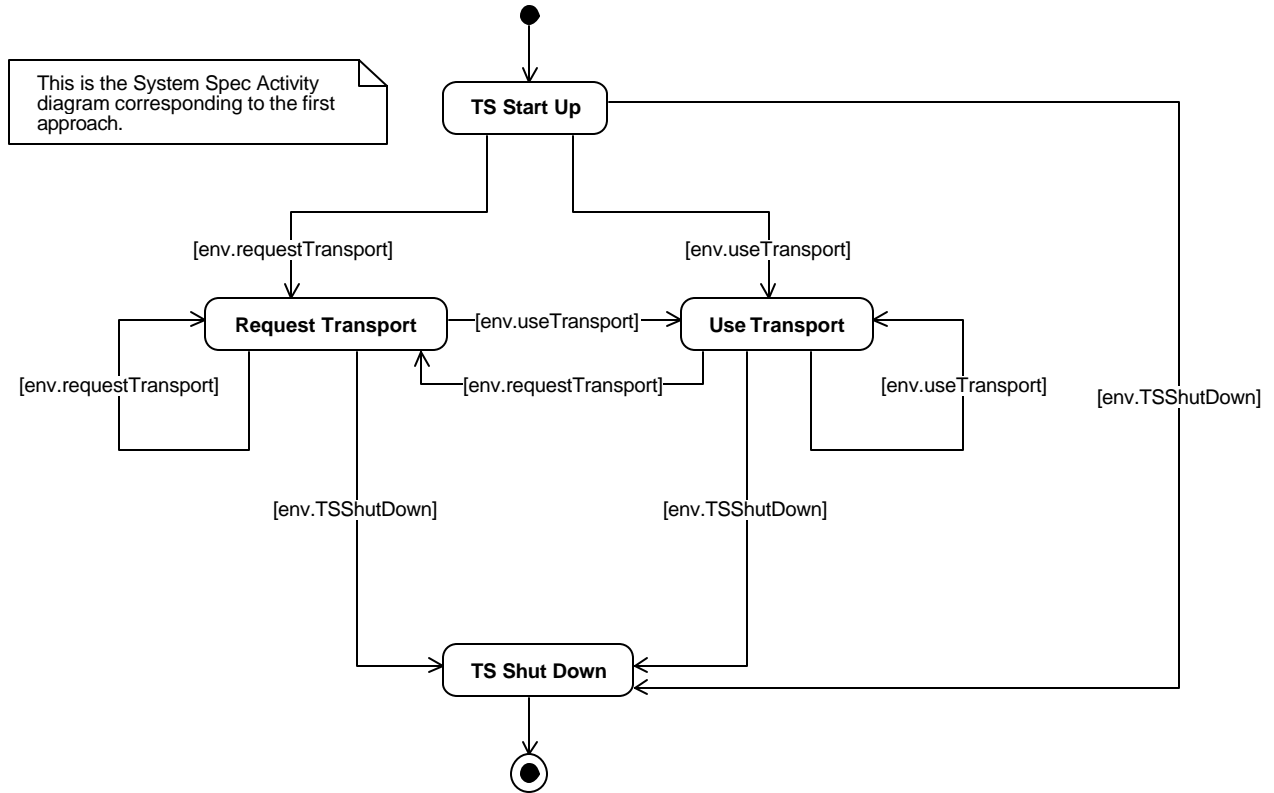
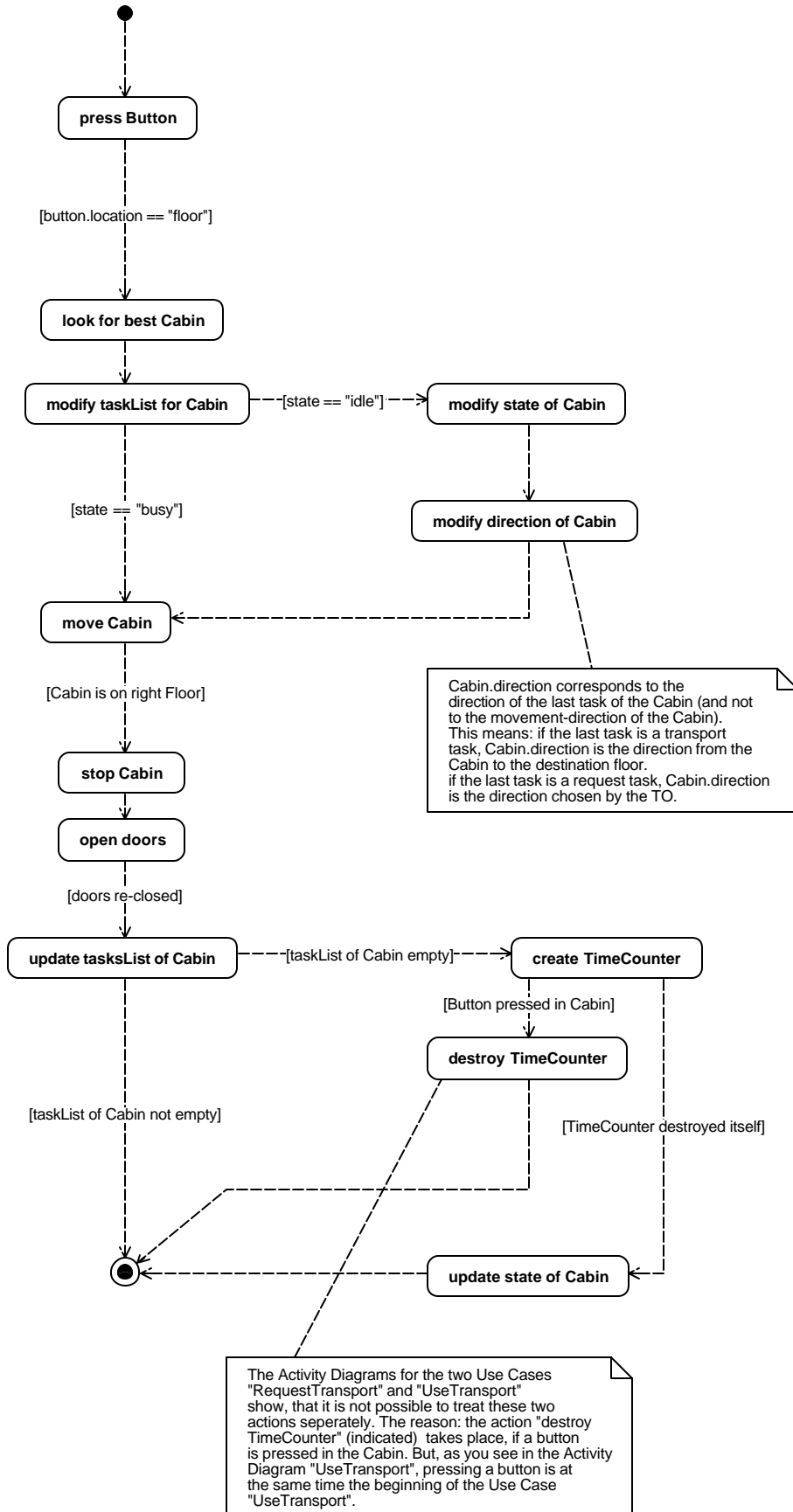


Figure 2. 21:S\_Activity (Activity Diagram)

### 30:S\_Activity<RequestTransport> (Activity Diagram)

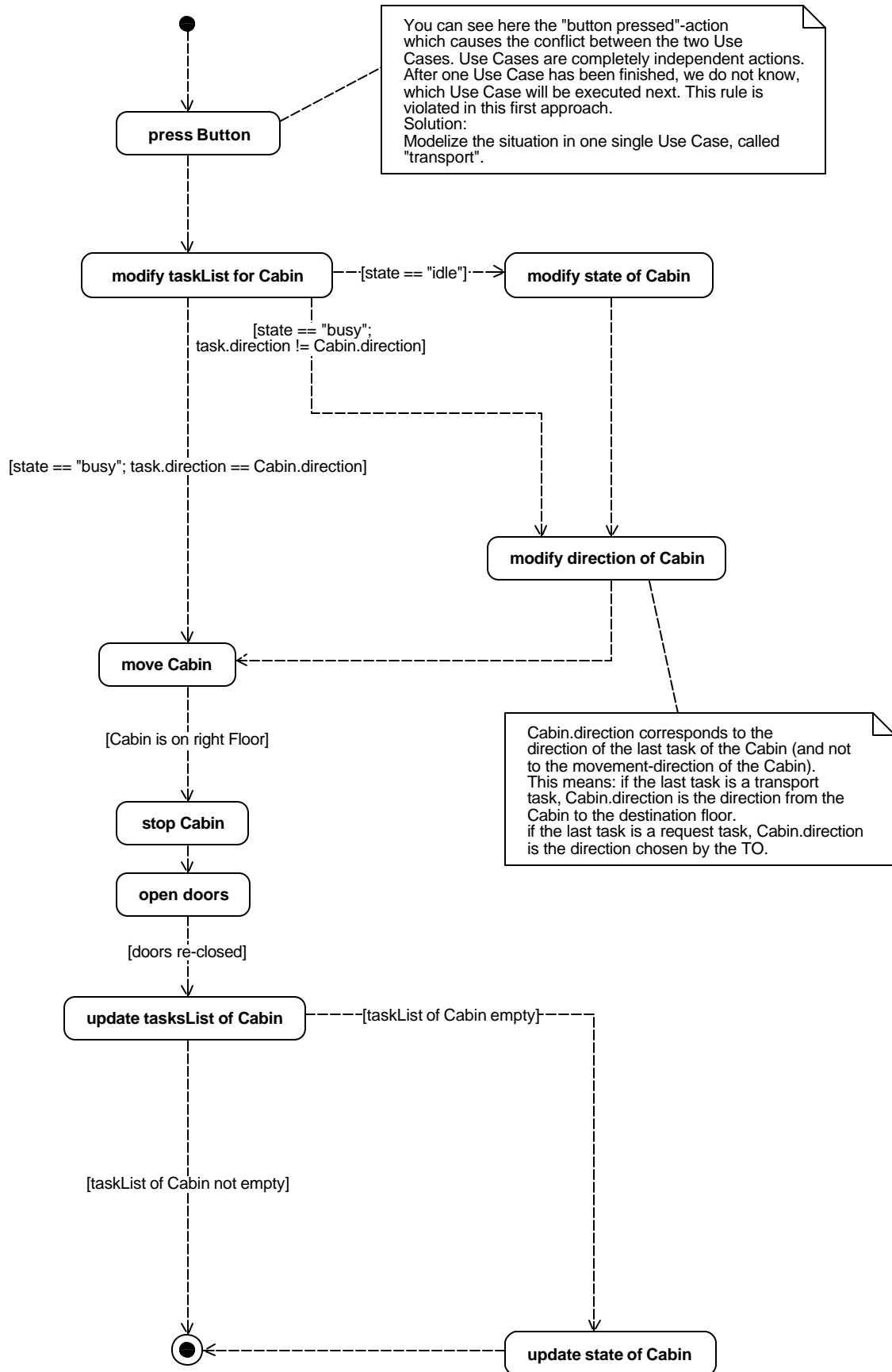
# Transportation System



**Figure 3. 30:S\_Activity<RequestTransport> (Activity Diagram)**

**31:S\_Activity<UseTransport> (Activity Diagram)**

# Transportation System



**Figure 4. 31:S\_Activity<UseTransport> (Activity Diagram)**

## 2.3 Business Implementation Spec

Here begins the complete and revised model of the TransportationSystem, that has lead to the software I programmed to show the system's functionality.

I went through the three modeling phases and, step by step, the structure of the whole system became clearer.

### 2.3.1 Snapshots

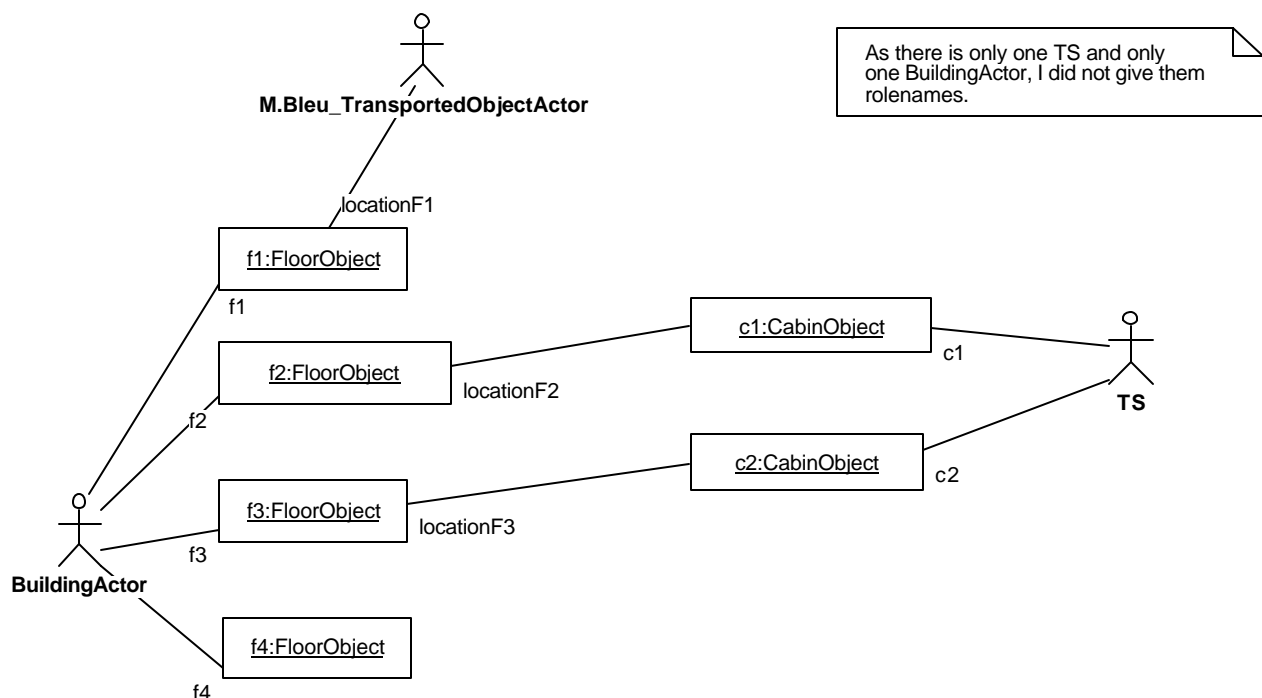
Snapshots are a rather new kind of diagrams, that are a very useful help to construct Object Diagrams. A Snapshot is, as one can imagine hearing that name, the description of one specific situation. Normally, in a running system, different object instances are being associated one to each other, new associations are created and existing associations are deleted as the systems state evolves. In an Object Diagram, all the connections of all the objects that could be important at any moment during the lifetime of the system should be represented.

If we make now Snapshot Diagrams for different situations, and we make enough of them, than we see all the associations that have to be present in the Object Diagram.

Attention: Snapshot Diagrams represent object instances. That's why for instance the fact that there are two cabins, leads to two instances c1 and c2 in my Snapshot Diagrams.

The scenario described in the following Snapshot Diagrams is easy. Pleas look at the explanations beneath every diagram for more details.

### 10:BI\_Snapshot<Initial> (Collaboration Diagram)



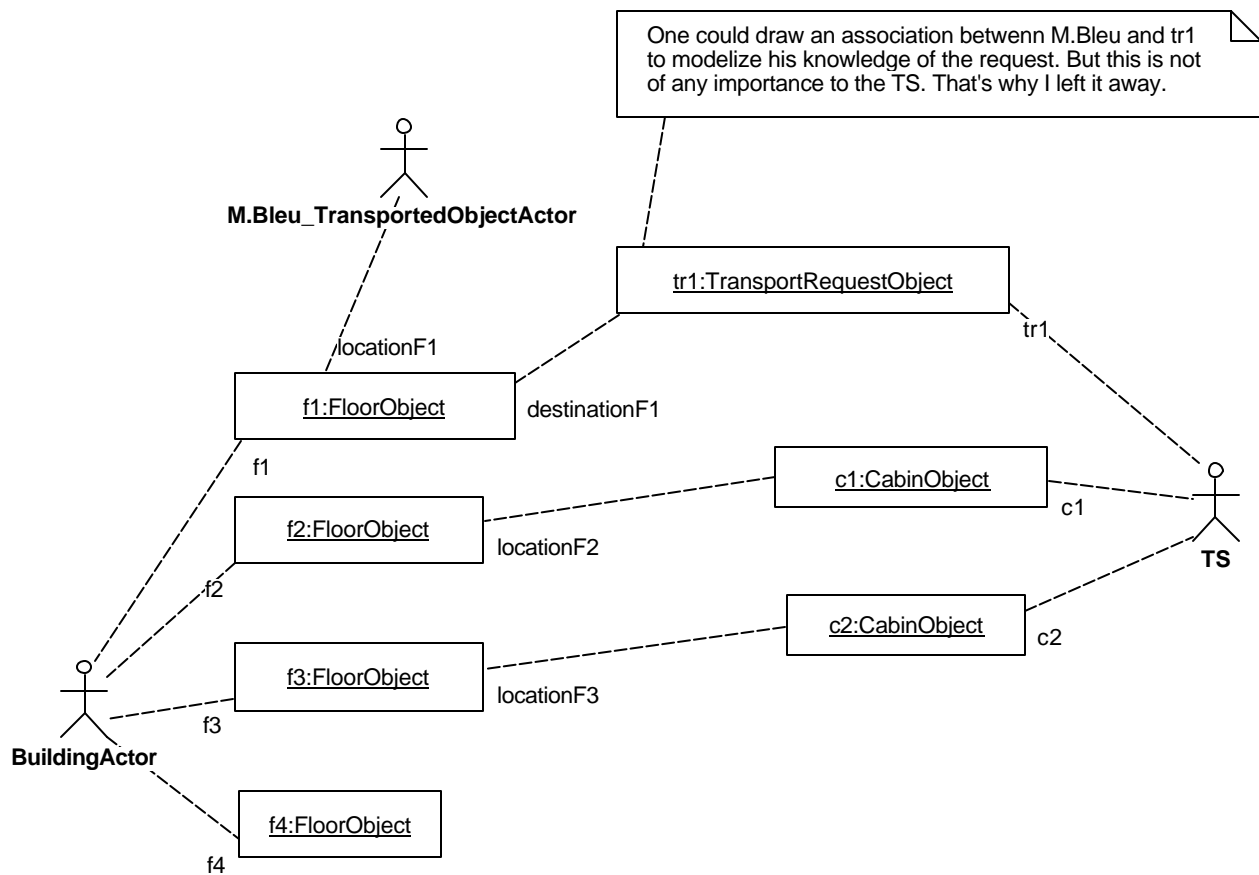
**Figure 5. 10:BI\_Snapshot<Initial> (Collaboration Diagram)**

There is a building with 4 floors (called f1 to f4). The TransportationSystem (TS) controls two cabins (c1 and c2). c1 is on the floor f2 and c2 is on the floor f3. M.Bleu, a TransportedObjectActor, is located on floor 1.

The postfixes ..-Object and ..-Actor are here to say if a “thing” is an Actor or an Object. Of course, TS is an actor, too, but as it is TS we want to model, this doesn’t need to be shown.

All those Objects and Actors are the representations of the corresponding real Objects and Actors in the viewpoint of an external observer. Sometimes, we say that they ARE the real objects, but this is of course incorrect, because one can not print real objects on the paper.

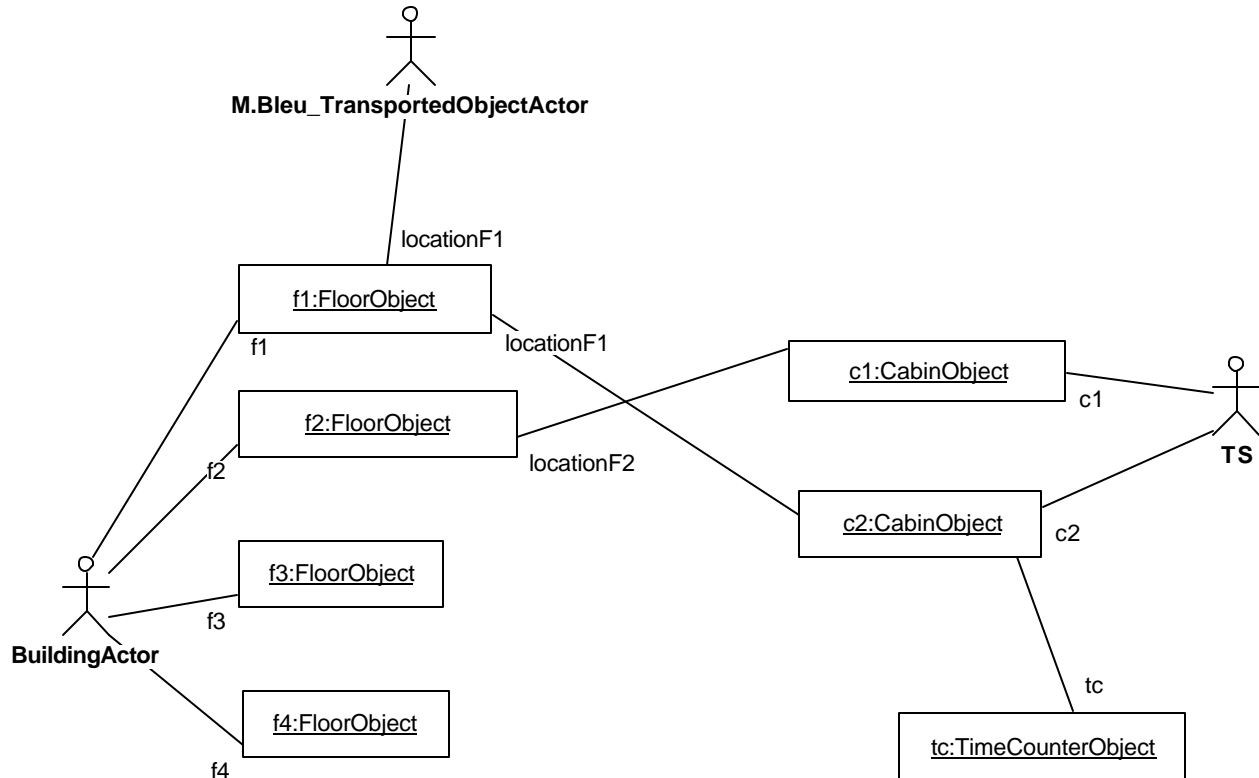
**11:BI\_Snapshot<AfterTransportRequested> (Collaboration Diagram)**



**Figure 6. 11:BI\_Snapshot<AfterTransportRequested> (Collaboration Diagram)**

This Snapshot shows the situation after M.Bleu has pressed a button of the elevator to go “up”. There is a new Object called TransportRequestObject, that represents the request M.Bleu has expressed by his action. The TS knows this “information-object” and has now the task to satisfy the request.

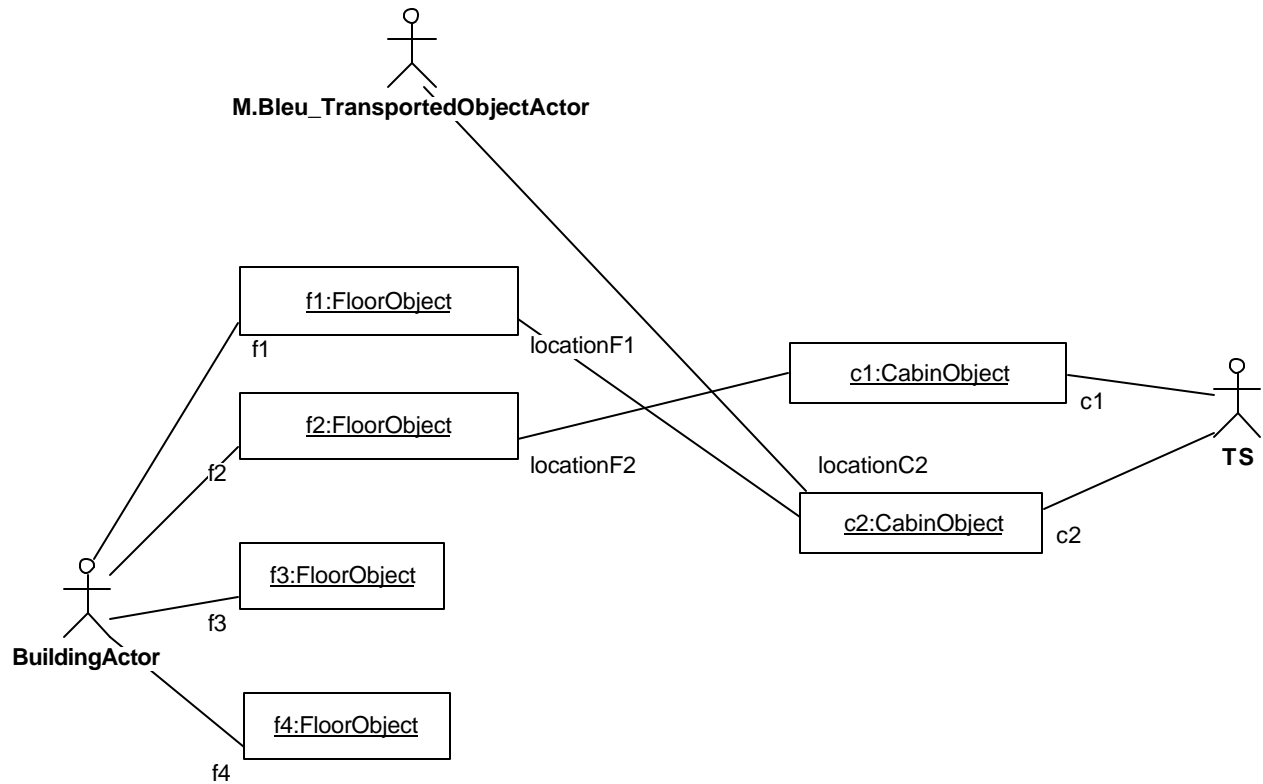
### 12:BI\_Snapshot<AfterTransportRequestSatisfied> (Collaboration Diagram)



**Figure 7. 12:BI\_Snapshot<AfterTransportRequestSatisfied> (Collaboration Diagram)**

This Snapshot shows the situation after arrival of c2 on the right floor. For some reason, TS has not sent c1, which would have been nearer to f1. Maybe c1 was occupied. A TimeCounterObject is created to show the fact that this c2 will not move for a certain time, to give M.Bleu the time to enter the cabin.

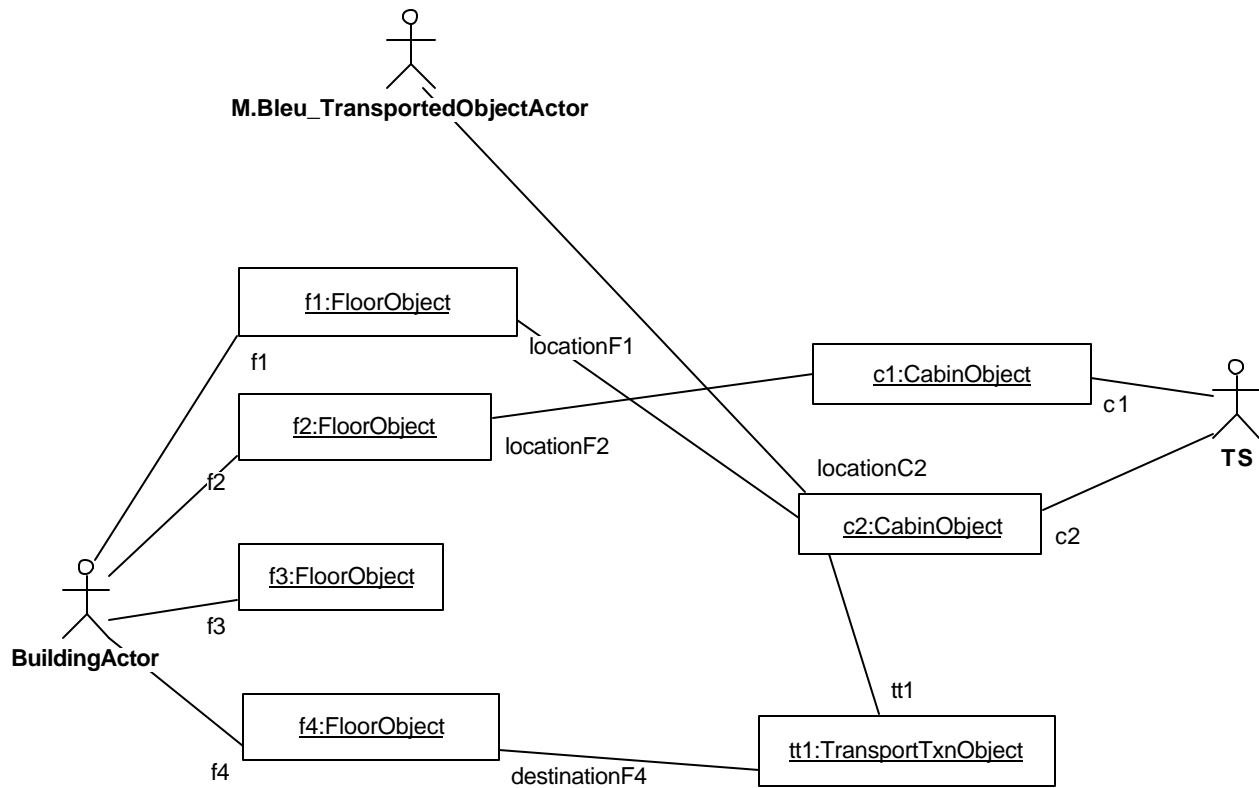
### 13:BI\_Snapshot<BeforeDestinationChosen> (Collaboration Diagram)



**Figure 8. 13:BI\_Snapshot<BeforeDestinationChosen> (Collaboration Diagram)**

The time-counter has expired, and M.Bleu has entered the cabin c2. But he has not yet pressed a button.

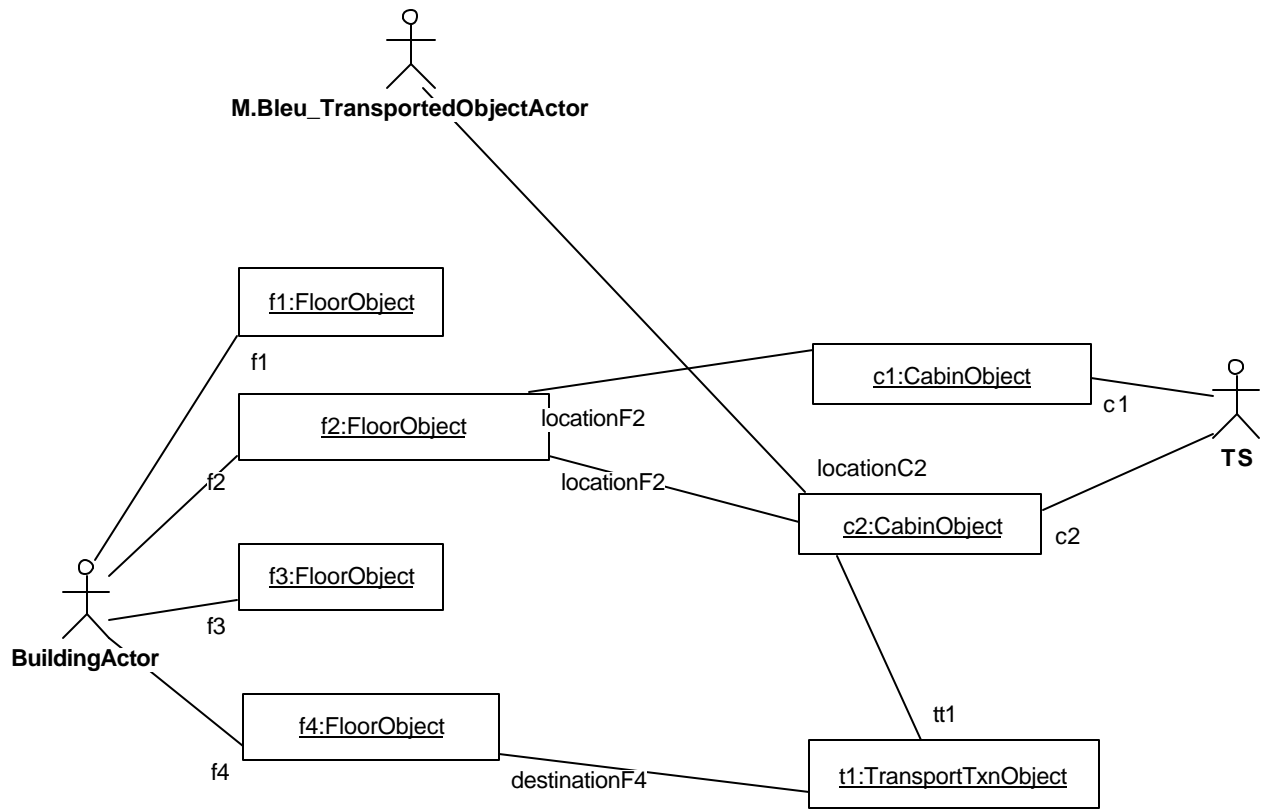
**14:BI\_Snapshot<AfterDestinationChosen> (Collaboration Diagram)**



**Figure 9. 14:BI\_Snapshot<AfterDestinationChosen> (Collaboration Diagram)**

This Snapshot shows the situation after that M.Bleu has pressed button number 4 in the cabin c2. A new information-object called TransportTxnObject has been created.

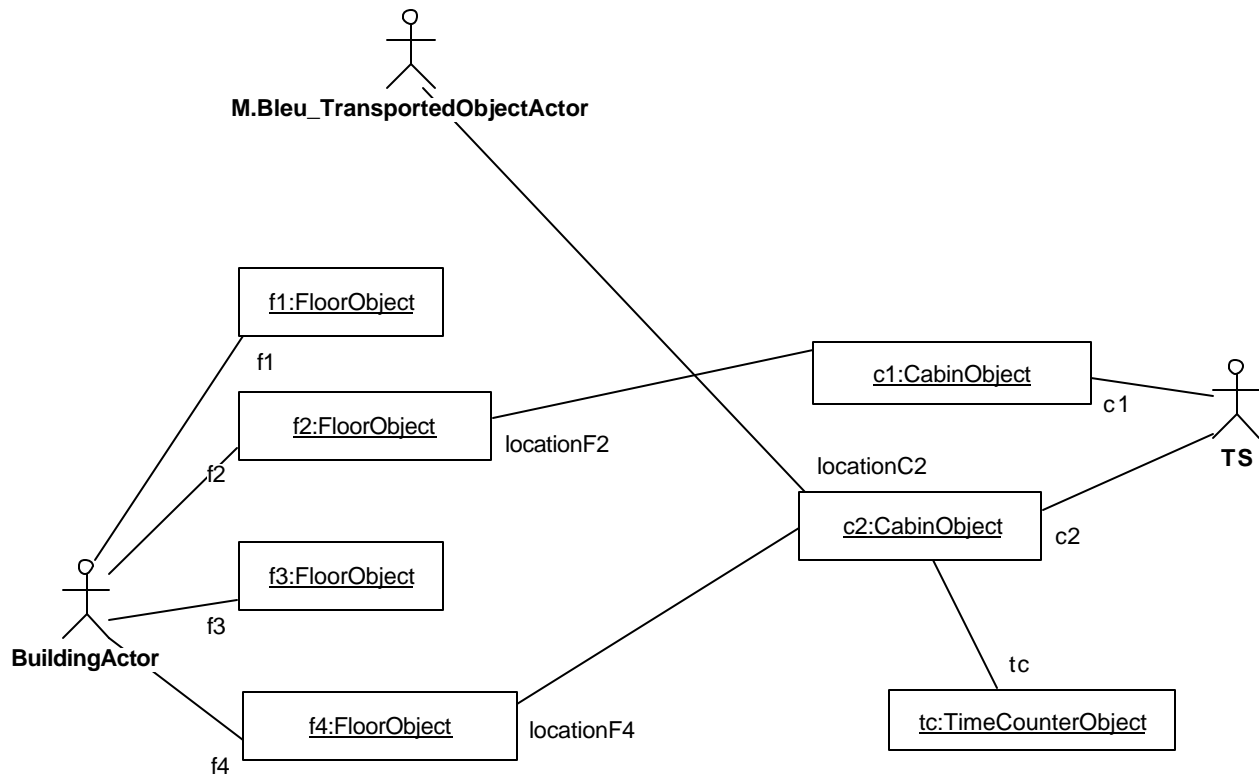
### 15:BI\_Snapshot<AfterDeparture> (Collaboration Diagram)



**Figure 10. 15:BI\_Snapshot<AfterDeparture> (Collaboration Diagram)**

The cabin c2 has moved to f2.

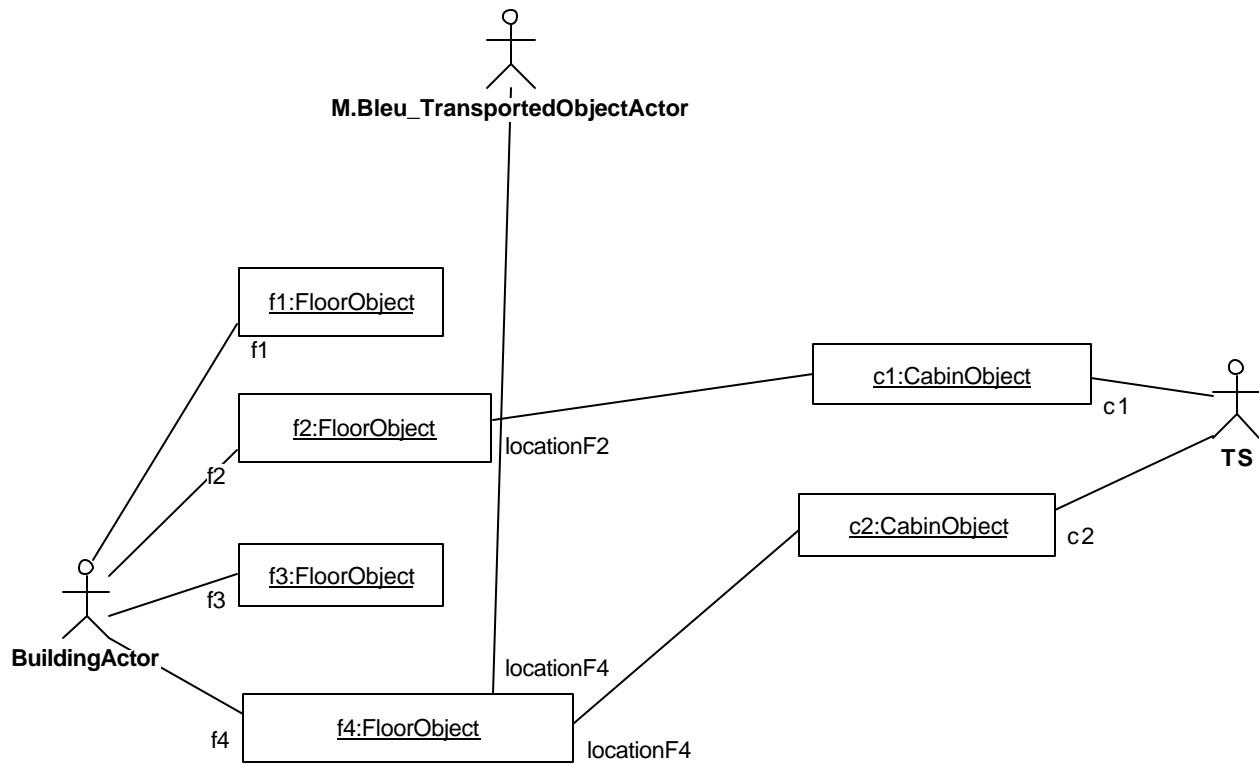
**16:BI\_Snapshot<AfterArrival> (Collaboration Diagram)**



**Figure 11. 16:BI\_Snapshot<AfterArrival> (Collaboration Diagram)**

The cabin c2 has arrived at f4 and stopped. A new TimeCounterObject has been created to let M.Bleu or other TOActors the time to choose a new destination.

**17:BI\_Snapshot<Final> (Collaboration Diagram)**



**Figure 12. 17:BI\_Snapshot<Final> (Collaboration Diagram)**

M.Bleu has left the cabin. The time-counter has expired. c2 is available for new requests.

## 2.3.2 Collaboration Model

As you can see on the next diagram, there is now only one Use Case called transport. This Use Case begins when a TransportedObjectActor presses a button. If the CabinObject arrives to a destination-FloorObject, the Use Case ends only if no TimeCounter is created (i.e. the CabinObject has other tasks to fulfill), or if the TimeCounter expires. If someone presses a button in the cabin during the lifetime of the TimeCounterObject, the same Use Case-instance continues.

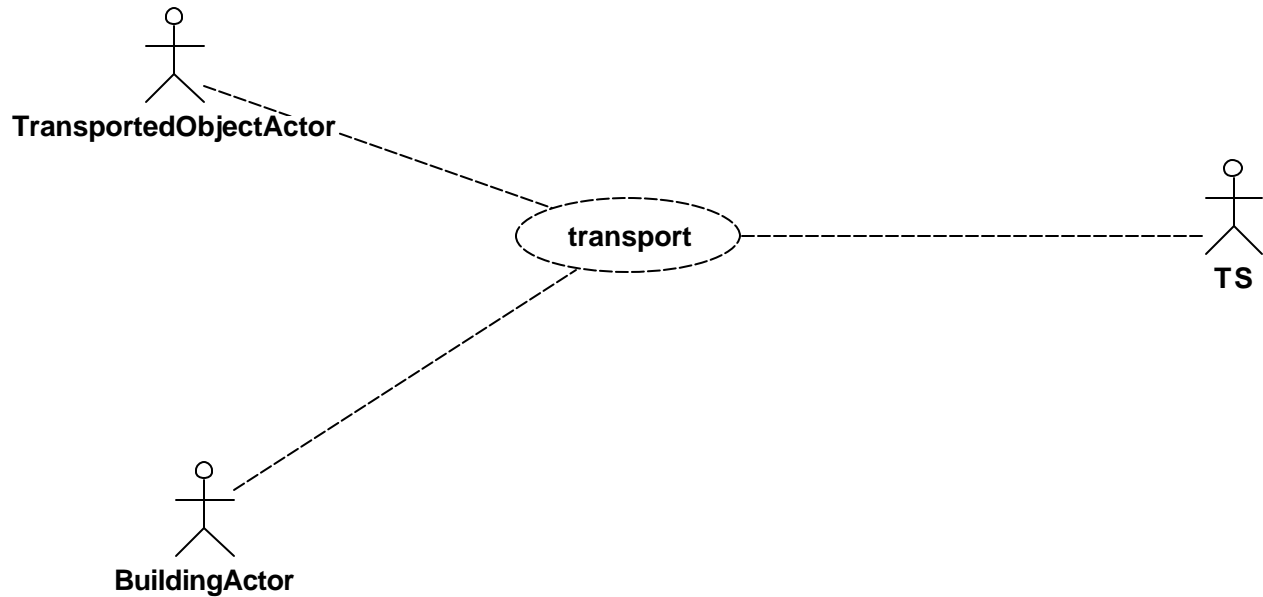


Figure 13. 20:BI\_Collaboration (Collaboration Diagram)

### Collaboration Description

#### Collaboration: transport

##### Purpose

One or more TransportedObjectActor request a transport from the TS and/or are transported to another floor.

##### Policies

UC\_P1: TransportedObjectActor cannot be identified. It doesn't have to be present during all the UC.

UC\_P2: If FloorObject and Direction have corresponding input parameters, they are given

by the TransportObjectActor (pressing a button) and the BuildingActor (detecting it).

UC\_P3: If CabinObject has a corresponding input parameter, it is given by the TransportObjectActor (pressing a button).

UC\_P4: The BuildingActor should be able to detect when a CabinObject is associated with a FloorObject.

UC\_P5: The TSActor should be able to compute the movements of each CabinObject by the input parameters it gets from the BuildingActor and the TransportedObjectActor.

UC\_P6: The TransportedObjectActor may have some wishes, that can be satisfied by the TSActor. Possible wishes are: to get any cabin to the floor on which the TOActor is located; or: to be transported to a specific destination floor in the cabin in which the TOActor is located.

### **Parameters**

one or more TransportedObjectActor, one BuildingObject, one TSActor,  
one or more FloorObject (with one Direction each) and/or one (or more) CabinObject

### **Pre-Conditions**

TransportedObjectActor exists

Cabin.weight  $\leq$  Cabin.weightLimit

FloorObject exists

The specified direction is possible on the Floor Object in question.

### **Post-Conditions**

All Objects created during UC (TransportRequestObject, DestinationObject or TimeCounterObject) have been destroyed.

The wish(es) of the TransportedObjectActor(s) that had initiated the UC have been satisfied.

### 2.3.3 Objects Model

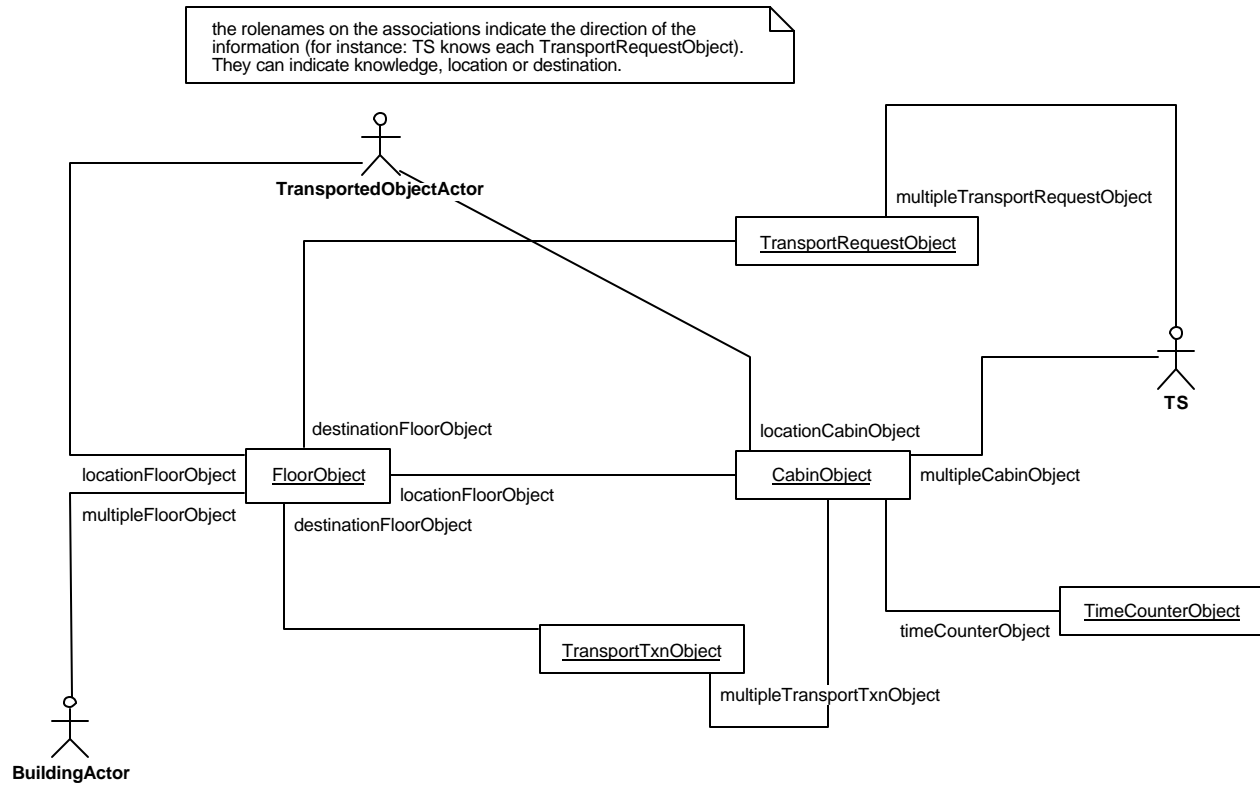


Figure 14. 21:BI\_Object (Collaboration Diagram)

## Glossary

### BuildingActor

Model of the building in which the TS is located and whose floors the TS serves.

### TransportedObjectActor

Model of a person interacting with the TS. A TransportedObjectActor (also: TOActor) can press one of the buttons on each floor or in each cabin, and it can be transported by a CabinObject.

## **TS**

Model of the system making sure that every TOActor can change locationFloorObject as wished.

## **CabinObject**

Model of a cabin of the elevator. CabinObjects are always associated with one FloorObject. They can be moved by the TS and can transport TOActors (number of transported TOActors is limited by their total weight).

## **FloorObject**

Model of a floor which belongs to BuildingActor and can be served by the Transportation System. A FloorObject can be the location of a TOActor or a CabinObject, and the destination of a TransportRequestObject or a TransportTxnObject.

## **TransportRequestObject**

Model of the task: to bring any Cabin to a specified FloorObject, on which a TOActor has requested a transport.

## **TimeCounterObject**

Model of the time-period during that the CabinObject it belongs to is "occupied". A CabinObject can be occupied after satisfaction of one TransportRequestObject or one TransportTxnObject. (If a new TOActor has entered the CabinObject, the system may give him some time to choose a destination, before the CabinObject can be requested and moved by another TOActor.)

## **TransportTxnObject**

Model of the task: to transport a TOActor which is in a CabinObject to a specified FloorObject.

## 2.4 IT System Spec

The goal of this phase is to build the framework for both the object-construction and the action-enchainement.

### 2.4.1 Snapshots

For finding out which of the objects of the previous phase should be represented in the systems viewpoint and what new objects should be added, I again used the Snapshot Diagrams. I drew the same Snapshots as in the Business Implementation Spec., with the following extension: in the left part of each diagram, you can clearly recognize the representations of all the Objects and Actors in the system-external viewpoint of a third observer, as they were drawn in the BI Spec. Those objects, that are very precise representations of the real objects, can be “traced” by the system. This means the system creates for the objects that are important to it an internal representation and connects these representations with the same associations that exist in the outer viewpoint. Like this, the system will be able to record each important change of the relations between the real objects by changing the relations between it’s internal representations.

So, in the right part of each diagram, you see the system-internal viewpoint.

To make an even clearer difference of the external and the system-viewpoint, objects that are representations of an external observer are called ..-Object and ..-Actor, while the system-viewpoint-representations do not carry this postfix.

### 10:S\_Snapshot<Initial> (Collaboration Diagram)

One object that has been added to the system’s viewpoint without having an external equivalent is the TaskList. It is needed to list several TransportTxn and TransportRequest objects for one Cabin.

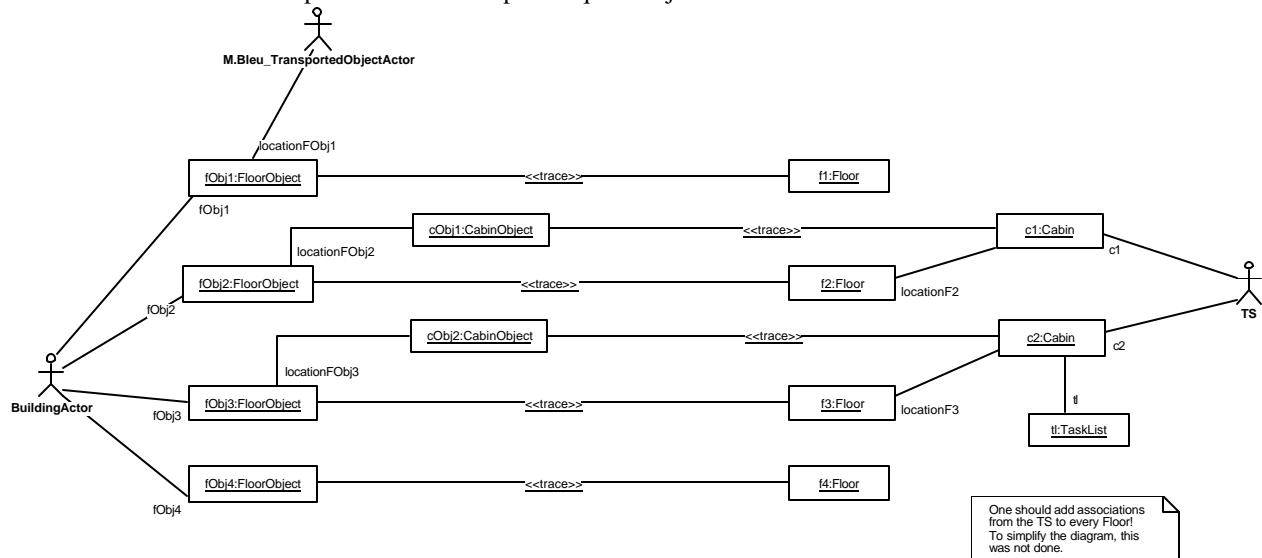


Figure 15. 10:S\_Snapshot<Initial> (Collaboration Diagram)

### 11:S\_Snapshot<AfterTransportRequested> (Collaboration Diagram)

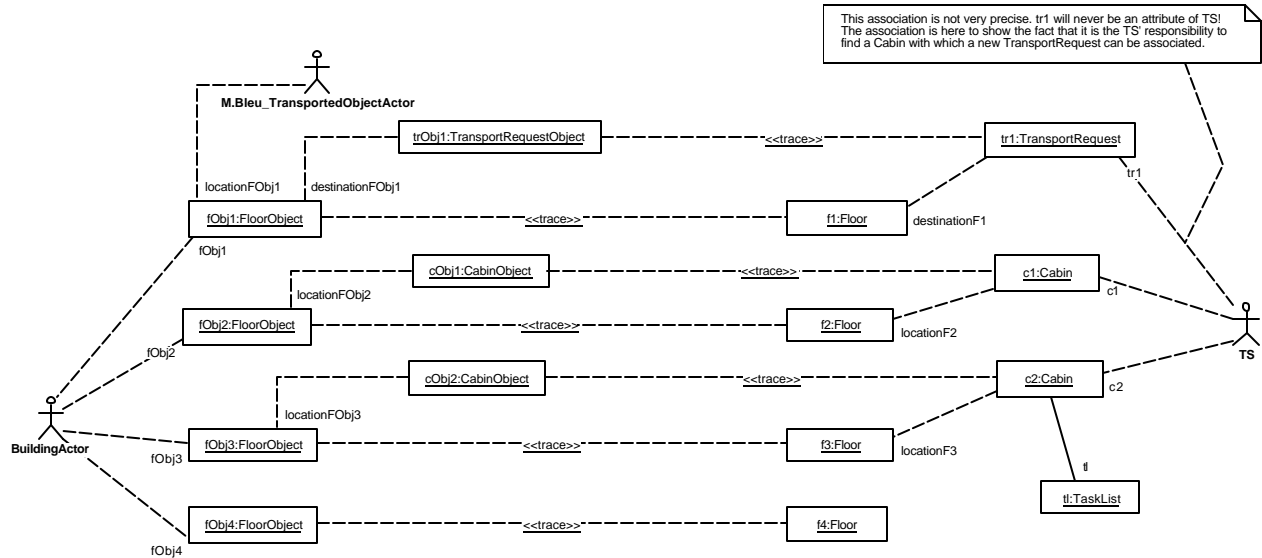


Figure 16. 11:S\_Snapshot<AfterTransportRequested> (Collaboration Diagram)

### 12:S\_Snapshot<AfterTransportRequestProcessed> (Collaboration Diagram)

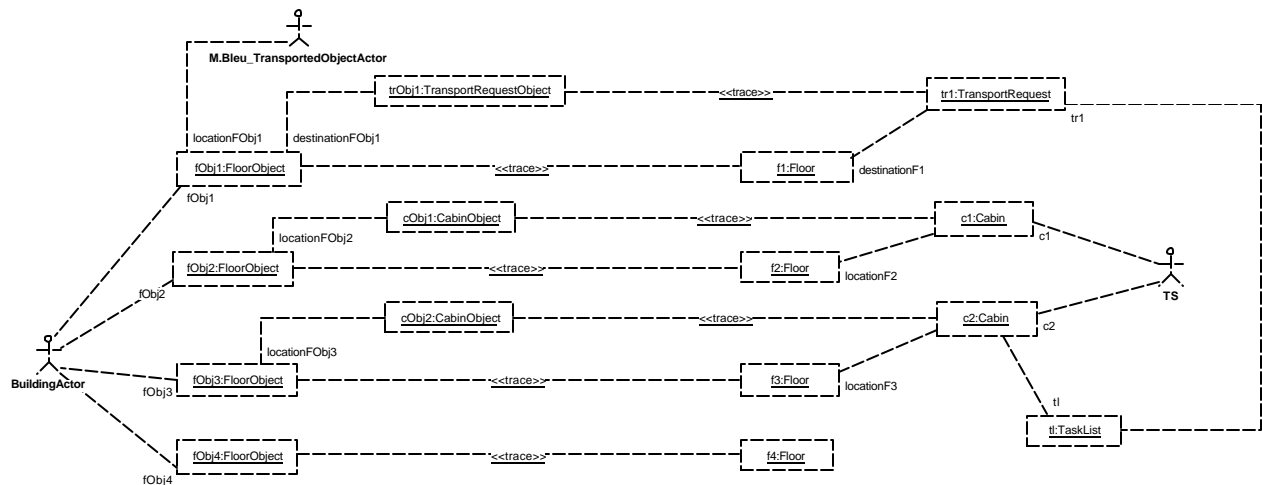
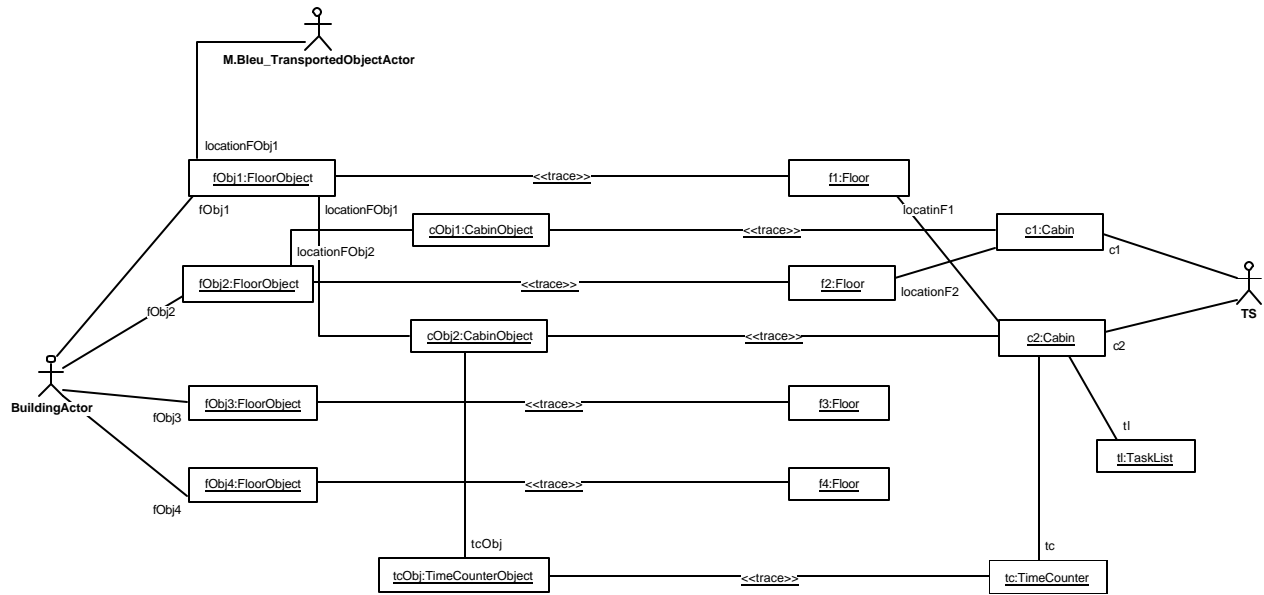


Figure 17. 12:S\_Snapshot<AfterTransportRequestProcessed> (Collaboration Diagram)

**13:S\_Snapshot<AfterTransportRequestSatisfied> (Collaboration Diagram)**



**Figure 18. 13:S\_Snapshot<AfterTransportRequestSatisfied> (Collaboration Diagram)**

### 14:S\_Snapshot<AfterDestinationChosen> (Collaboration Diagram)

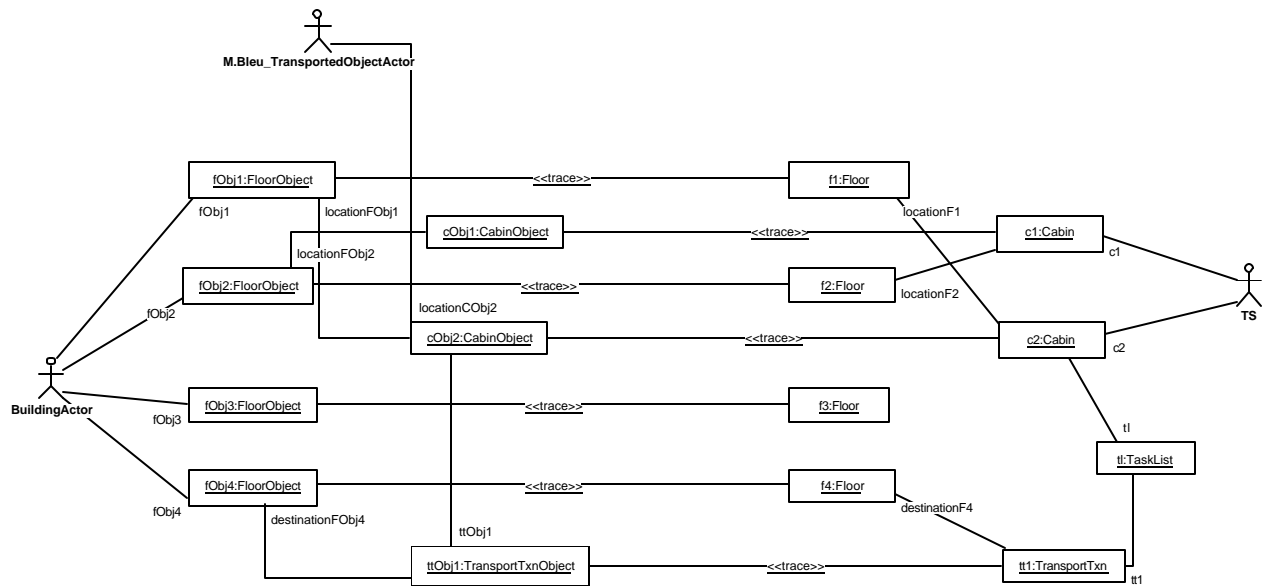


Figure 19. 14:S\_Snapshot<AfterDestinationChosen> (Collaboration Diagram)

### 15:S\_Snapshot<AfterArrival> (Collaboration Diagram)

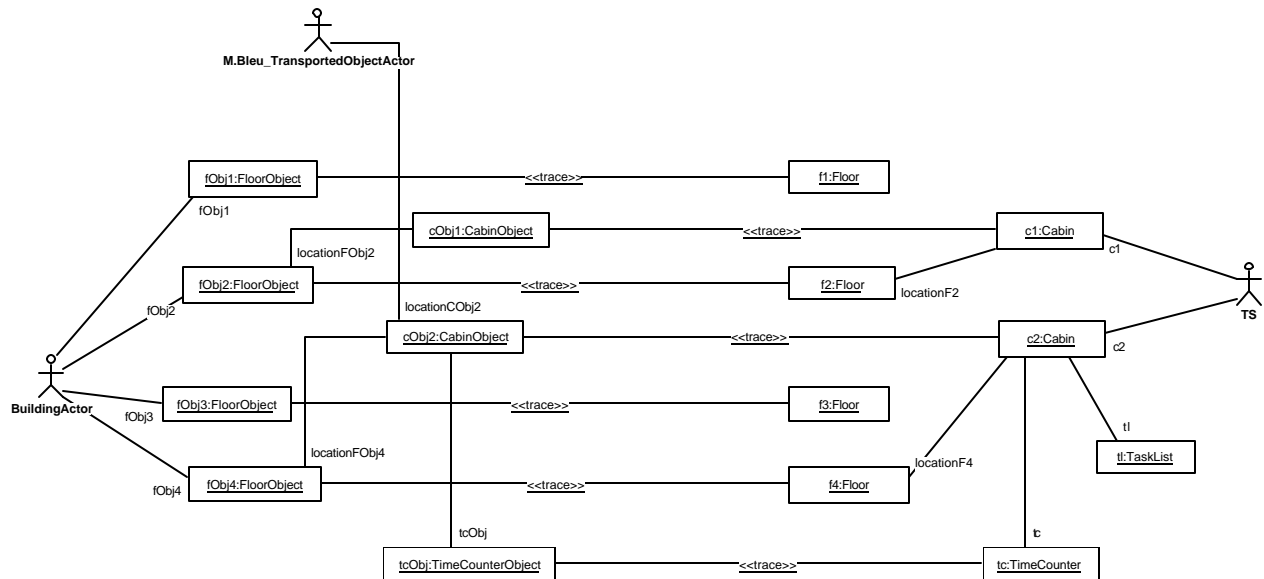


Figure 20. 15:S\_Snapshot<AfterArrival> (Collaboration Diagram)

## 16:S\_Snapshot<Final> (Collaboration Diagram)

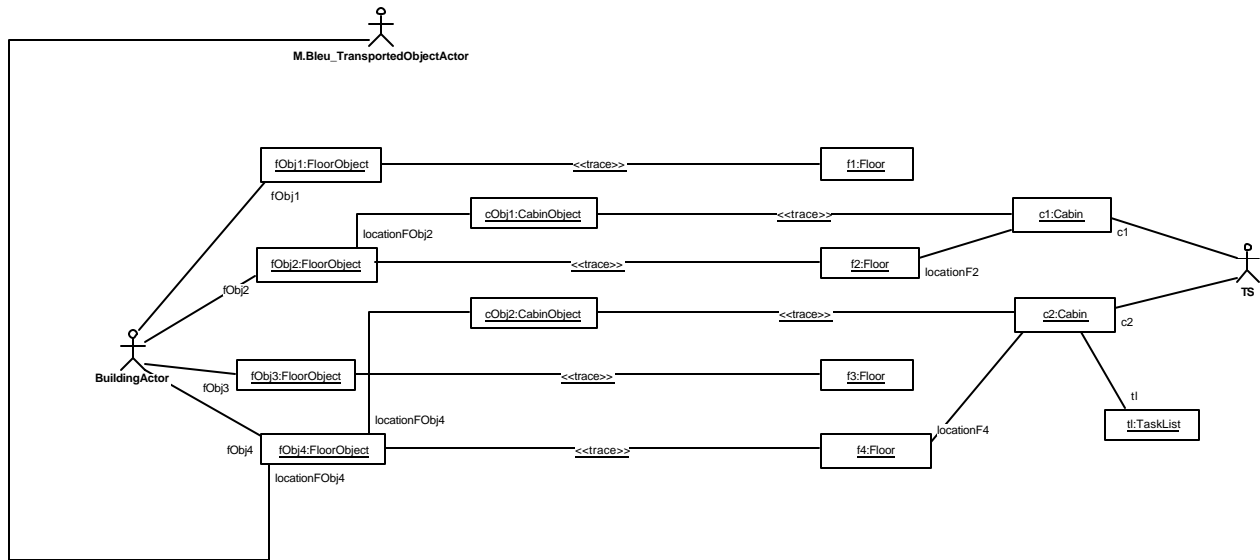


Figure 21. 16:S\_Snapshot<Final> (Collaboration Diagram)

In all the Snapshot Diagrams, you can clearly see that despite the fact that the TOActor is the initiator of the transport actions, no representation of it is maintained in the system. The reason why I decided to do this is described in the System Policies and Patterns in the next paragraph.

### 2.4.2 Use Cases Model

In this part of the phase, the collaboration diagram of the previous phase is limited to those parts of the collaborations that concern the system. That's why I placed the TS actor in a corner of the diagram, the TS is involved into every of those actions and could also be drawn as a rectangle containing the three Use Cases.

The two new Use Cases TSStartup and TSShutdown are not really important. As I implemented the software-version of the TS using java, startup- and shutdown-functionalities are automatically built in.

But very important in the Use Cases Model are the System Policies and the detailed Use Case description. These contain the rules to follow for all the next steps.

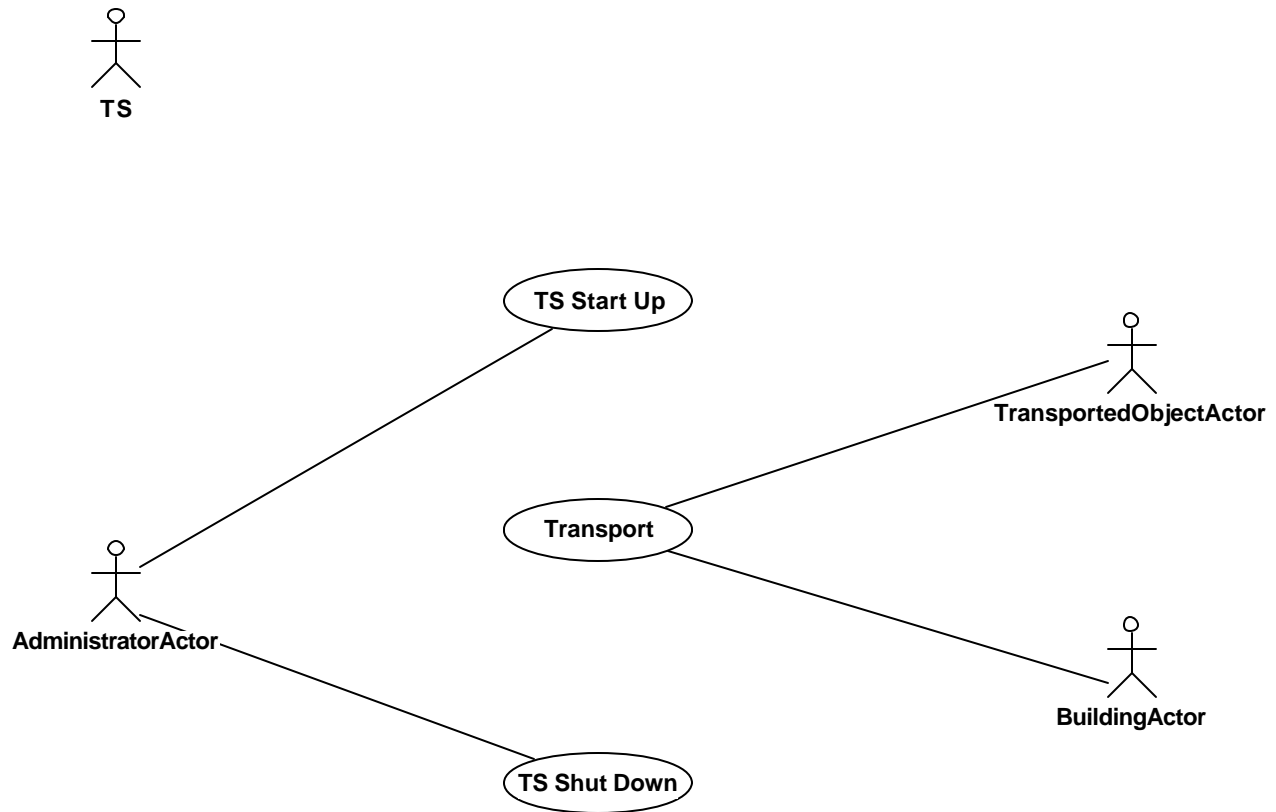


Figure 22. 20\_S\_Usecase (Use Case Diagram)

## System Policies and Patterns

S\_P1: TransportedObjectActor initialises all the actions of the TS.

S\_P2: The TS will process the actions (the tasks) in the order that corresponds to the chronological order in which they were initialised. Of course, these tasks are not always fulfilled in the same order they were processed, depending on the length of each task.

S\_P3: The TS has an internal representation of the important Objects like cabins or floors. It has to update the state of those representations using informational signals from the environment (doors closed, cabin x arrived on floor y, etc...)

S\_P4: The TS has no object representing the TransportedObjectActor, because there is no way to get any information about it. A TOActor can only initialize an action, it cannot modify or cancel it. It is not identified by the TS.

S\_P5: The input-signals sent to the TS by its environment are supposed to be correct. This means one instance of TS is configured for a specific hardware (a certain number of floors and cabins) and this hardware is expected to work well (to send the right signals).

## Use Case Description

### Use Case: Transport

#### UC Purpose

One of the following action-sequences is satisfied:

- a single TransportRequest

or

- one or several TransportTxn

or

- a TransportRequest immediately followed by one or several TransportTxn

#### UC Policies

P1: The TS does not know if the TransportedObjectActor that initialized this UC is still present. That's why no confirmation or cancel input from the TransportedObjectActor is allowed or required (no stop functionality inside a cabin is required by the initial contract).

P2: Depending on the type of the initialized action-sequence, the TS has to be able to process different calculations from different UC Parameters, and to give the right output signals to it's hardware (cabin-doors, cabin-motors, cabin-overweight-warning lamp).

#### UC Parameters

N times:

IN: one floorNumber (the number of the floor on which a button has been pressed)

IN: one direction (the direction of the button which has been pressed)

IN: one buttonType (in this case: "floor")

,  $0 \leq N \leq 1$

M times:

IN: one cabinNumber (number of the cabin in which a button has been pressed)

IN: one floorNumber (the number of the button which has been pressed)

IN: one buttonType (in this case: "cabin")

,  $0 \leq M$

additional Parameters:

IN: at each of the L arrivals: floorNumber and cabinNumber,  $0 \leq L$  (is sent to the TS each time a cabin arrives at a floor)

IN: at each of the P times of doors closing: cabinWeight and cabinNumber,  $1 \leq P$ ,

$P \geq N + M$  (is sent to the TS each time the doors of a cabin close)

### **UC Pre-Conditions**

At least one Cabin exists.

At least 2 Floors exist.

TS is running (the last Action performed by the AdministratorActor was TS Start Up).

Each Cabin is located on one Floor.

### **UC Exceptions**

E1: If  $\text{Cabin.weight} > \text{Cabin.weightLimit}$ :

E1.1: the TS announces the problem E1 in the cabin (a red light is activated).

E1.1.1: the cabin is blocked (cabin cannot move, no new tasks can be added to the Cabin's Tasklist), and the doors of the cabin are opened.

### **UC Post-Conditions**

No TransportTxn exists.

TransportRequest doesn't exist.

All TimeCounters that had been created have been destroyed or have destroyed themselves.

The Cabin that was moved is now associated with the destination Floor of the last performed Task.

The value of TaskList.size has been decreased as many times as it had been increased.

## 21:S\_Activity (Activity Diagram)

This diagram is in our case rather banal. It shows the order in which the different Use Case instances may occur. Unfortunately, the fact that two Use Case instances can take place in parallel was nearly impossible to represent. Using the parallelism syntax of complex transitions would result in listing an infinite number of Use Cases called transport, that can all begin at any time of the system's lifecycle. The problem is that a new UseCase instance can start while another one is being executed. But anyway, this is not too bad, because parallelism will be enabled by my event model.

Attention: this diagram does not show the possible parallelism between several instances of the UC transport. This would only be possible by drawing as many UC instances as allowed, which is too complicated.

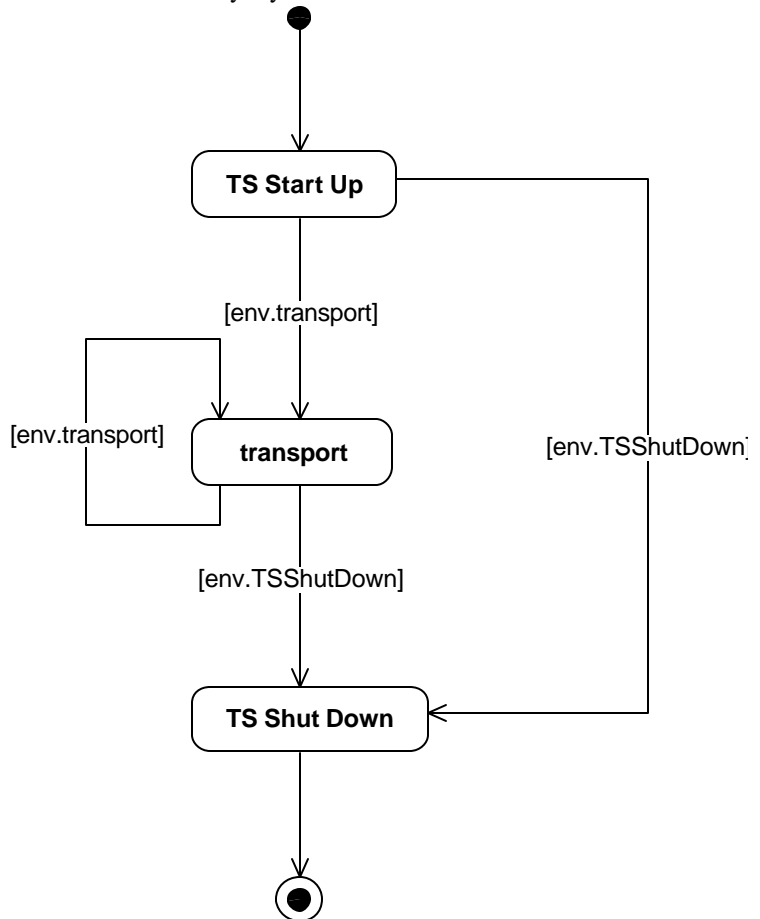


Figure 23. 21:S\_Activity (Activity Diagram)

### 2.4.3 Conceptual Model

Now, we use the Snapshot Diagrams to construct the Concept Diagram, representing the structure of the system. Each object or representation in the system is called a concept.

In addition to the information of the Snapshots, we add the attributes that are certainly necessary, like the floorNumber for a Floor.

The two lists CabinList and FloorList are used to handle multiple Cabin and Floor concepts in an easy way.

I also added a super-concept called Task, from that TransportRequest and TransportTxn inherit. This should show the similarities of the two task types. In fact, the two types differ only by their name. But, if I had made one single concept called Task, I would have been forced to add an attribute “type”. The inheritance is a much nicer solution, also because it the more intuitive choice.

You will also see that TS has become “Myself”. This is the main concept, representing the system itself. Starting from this concept, the system will access all the information it needs.

## 22\_S\_Concept (Class Diagram)

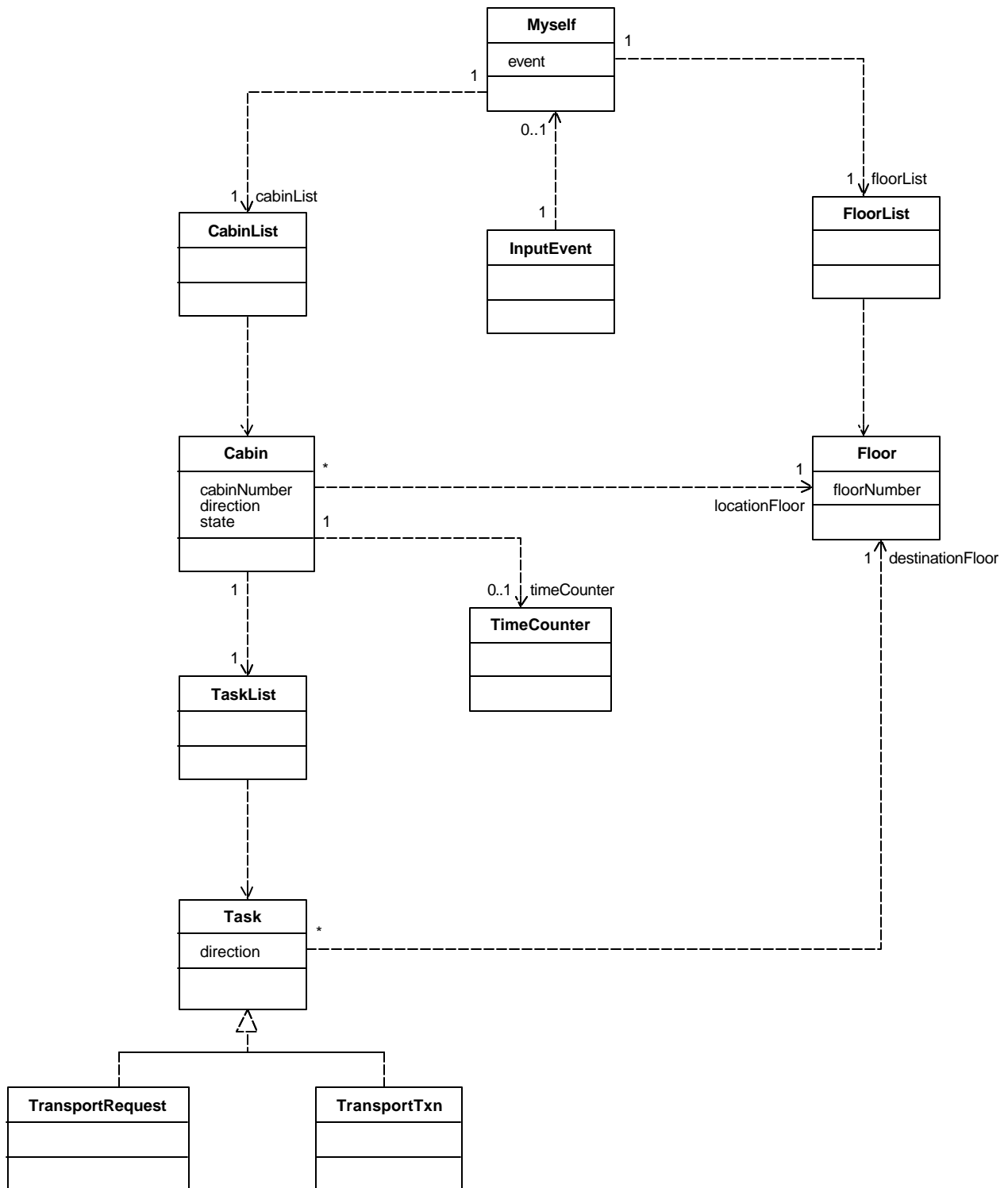


Figure 24. 22\_S\_Concept (Class Diagram)

## **Glossary**

### **Cabin**

Concept representing one CabinObject.  
States: busy, idle, blocked, occupied

### **CabinList**

Concept referencing all the Cabin concepts.

### **FloorList**

Concept referencing all the Floor concepts.

### **Floor**

Concept representing one FloorObject.

### **InputEvent**

There are three possible input events:

1. ButtonEvent:

is sent to the TS each time a button of the elevator is pressed by a TOActor.

It contains the following values:

one buttonType (in this case: "cabin"), one cabinNumber, one floorNumber

OR:

one buttonType (in this case: "floor"), one floorNumber, one direction

(if floorNumber is the number of the lowest floor, direction must be "up",

if floorNumber is the number of the highest floor, direction must be "down";

the hardware has to make sure that this error does not occur!)

2. ArrivalEvent:

is sent to the TS each time a cabin arrives at a floor.

It contains the following values:

one floorNumber, one cabinNumber

3. DoorEvent:

is sent to the TS each time the doors of a cabin close.

It contains the following values:

one cabinWeight, one cabinNumber

### **Myself**

Concept used to represent what the TS knows about itself.

It is created when the TS is started up.

It is destroyed when the TS is shut down.

### **Task**

Concept representing one entry in the TaskList concept of one Cabin concept.

It is either a TransportRequest or a TransportTxn.

### **TaskList**

Concept referencing all the Task concepts of one Cabin concept.

### **TransportRequest**

Concept representing one TransportRequestObject.

### **TransportTxn**

Concept representing one TransportTxnObject.

### **TimeCounter**

Concept representing one TimeCounterObject.

## 2.4.4 Partial System Activity

Up to this point, we described the system's state before and after the occurrence of the Use Case "transport", how several of those Use Cases can be chained and who is taking part in it. Now, it's time to go inside the Use Case "transport" and to see what actions should be done in what case and in which order. But, we do not yet decide which of the concepts is doing the work. Also, each of the actions shown in diagram 30 is another blackbox containing another sequence of actions.

### 30:S\_Activity<transport> (Activity Diagram)

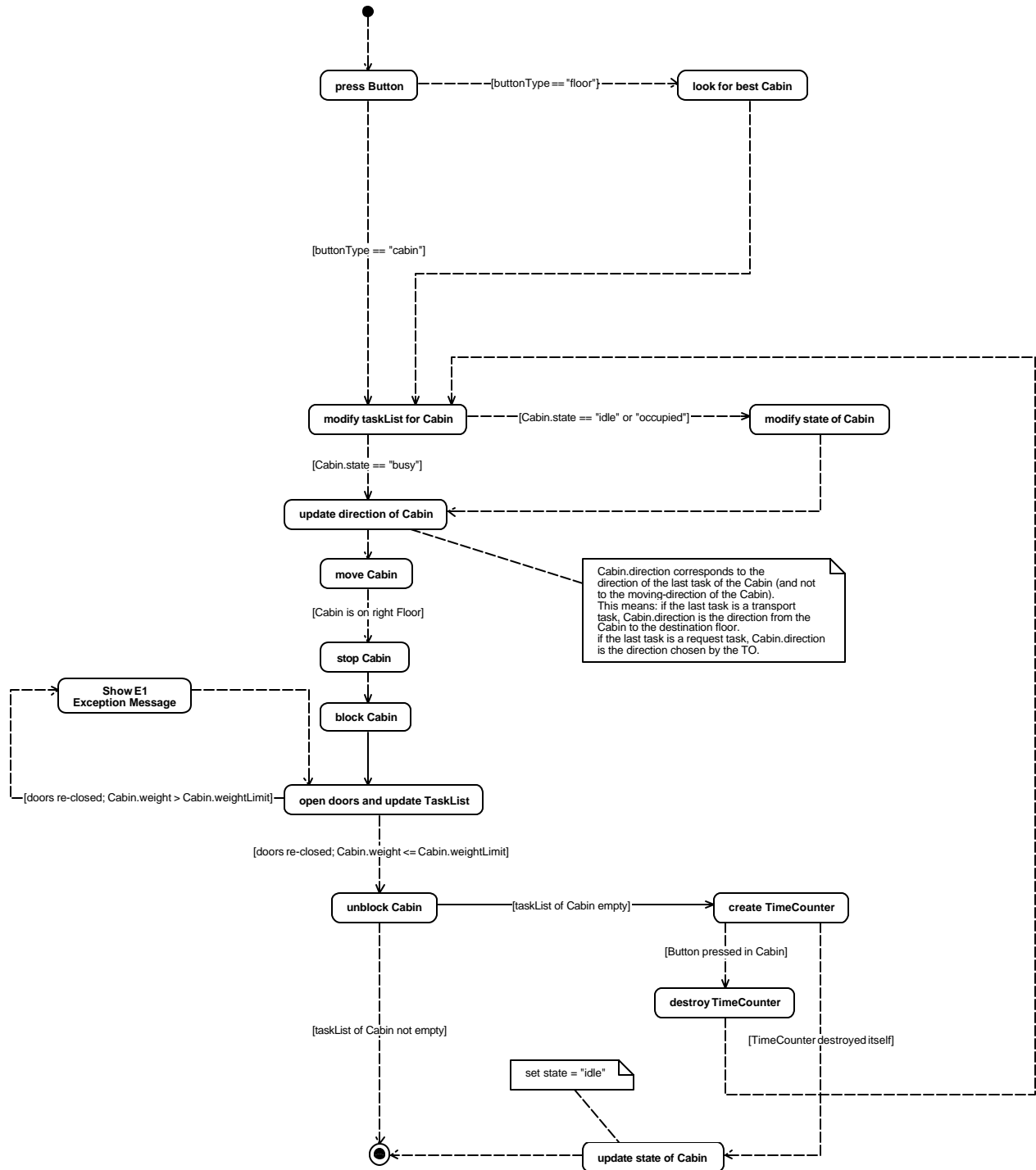


Figure 25. 30:S\_Activity<transport> (Activity Diagram)

## 2.5 IT System Implementation Spec

The aim of this last modelling phase is to make clear what work is done in which instance of which concept in what case, and to draw a final Class Diagram, that contains the static part of the system information, and from which we can generate the Java code.

The general ideas and functionalities of my elevator are:

A cabin can be:

- with open doors (state="blocked")
- with a running time-counter (state="occupied")
- moving to some direction (state="busy")
- idle(state="idle")

If a TransportedObjectActor requests a transport from a Floor, the best Cabin for this Task is found, and the Task is added to it's TaskList. The Cabin move in a way to satisfy as fast as possible the oldest Task in the TaskList, but if it arrives on the destinationFloor of another Task, this task is first satisfied and then the Cabin moves on.

If a TransportedObjectActor presses on a button in a Cabin, this Task is of course added to the TaskList of the corresponding Cabin.

If the doors of a Cabin are closed, and there are no more Tasks in it's TaskList, a TimeCounter is created. During the lifetime of this TimeCounter, a Cabin is "occupied" and can only be moved by a TOActor IN the Cabin.

### 2.5.1 Scenarios

In a Sequence Diagram, one decides exactly what scenario he wants to show. The diagram is then like a film with a vertical time-axis, starting at the top and descending.

The arrows show function calls, the value after the double point is the returned value. It is either the name of an instance like aTransportRequest or a name representing a java-type value like found, that is a boolean. Instead of writing "found", I could have written "true" or "false", but I used this little unexactness to give the description of what would happen if found was true in the comment box at the bottom. Like this I avoided drawing another sequence diagram for that case.

Some words about the events: I decided to implement the communication interface between the TS and it's hardware using events. An event can be sent to and from the hardware, has a type and some attributes that are sent with it. Finally, those events are used by simply calling the ActionPerformed function of either the TS or it's hardware, but it is still a nice idea. An event could theoretically also be an electrical impulse or a binary signal transmitted by a cable to the computer in a real elevator.

Those events allow parallelism for different Use Case instances. You will see in the following diagrams that the system does not remember if a new Use Case instance has begun, if an old one has continued or if one has ended. It simply gets an input event, processes it, updates the state of the concept instances and returns, if necessary, an output event. The TS does not care at all which of those events belong to the same Use Case instance. And that is why parallelism is automatically possible.

Another interesting point is the function lookForBestCabin(aTransportRequest):aCabin. This is the place where the system decides which cabin should get a new entered TransportRequest.

### 10:SI\_Scenario<initializeTransportRequest> (Sequence Diagram)

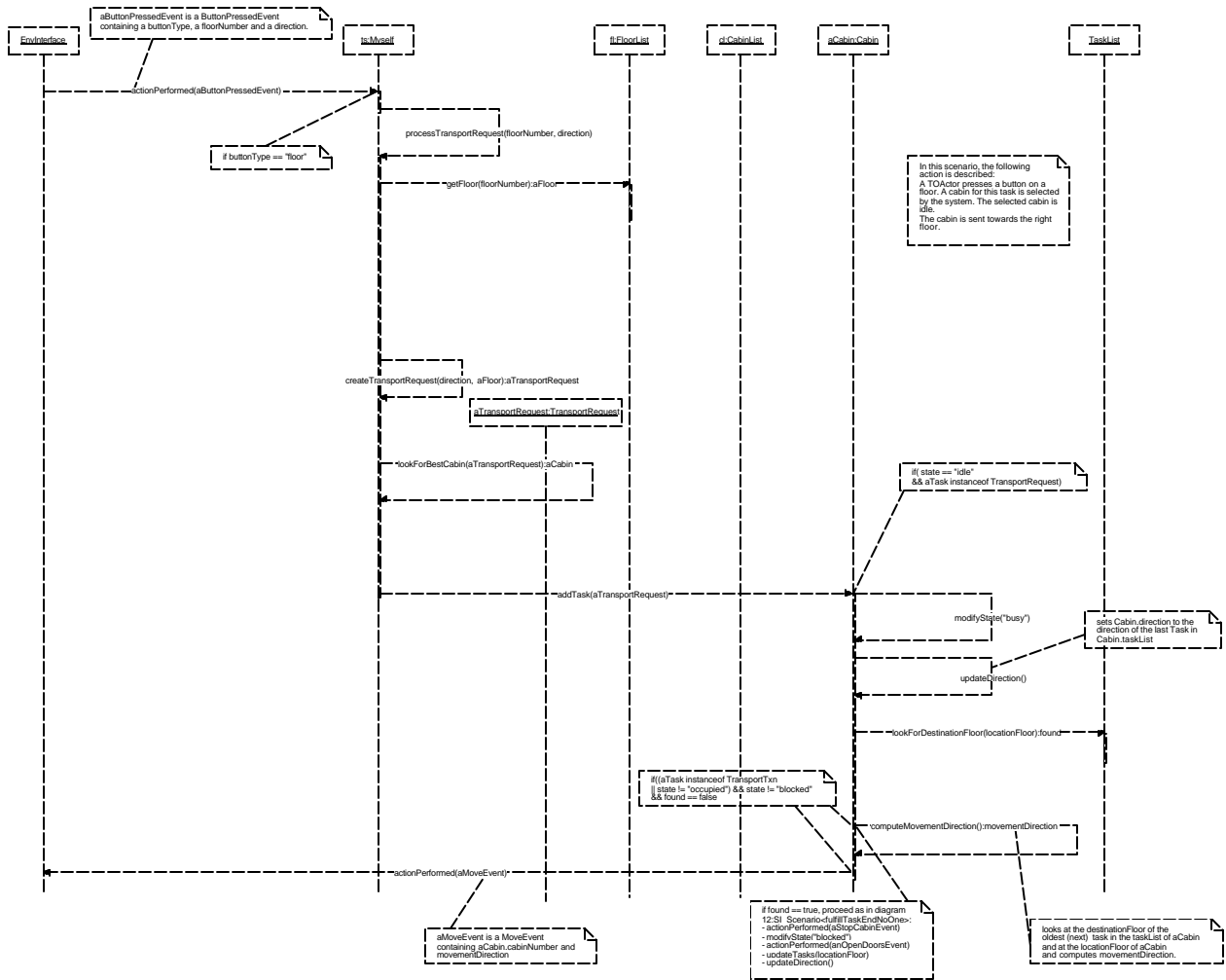


Figure 26. 10:SI\_Scenario<initializeTransportRequest> (Sequence Diagram)

### 11:SI\_Scenario<moveCabin> (Sequence Diagram)

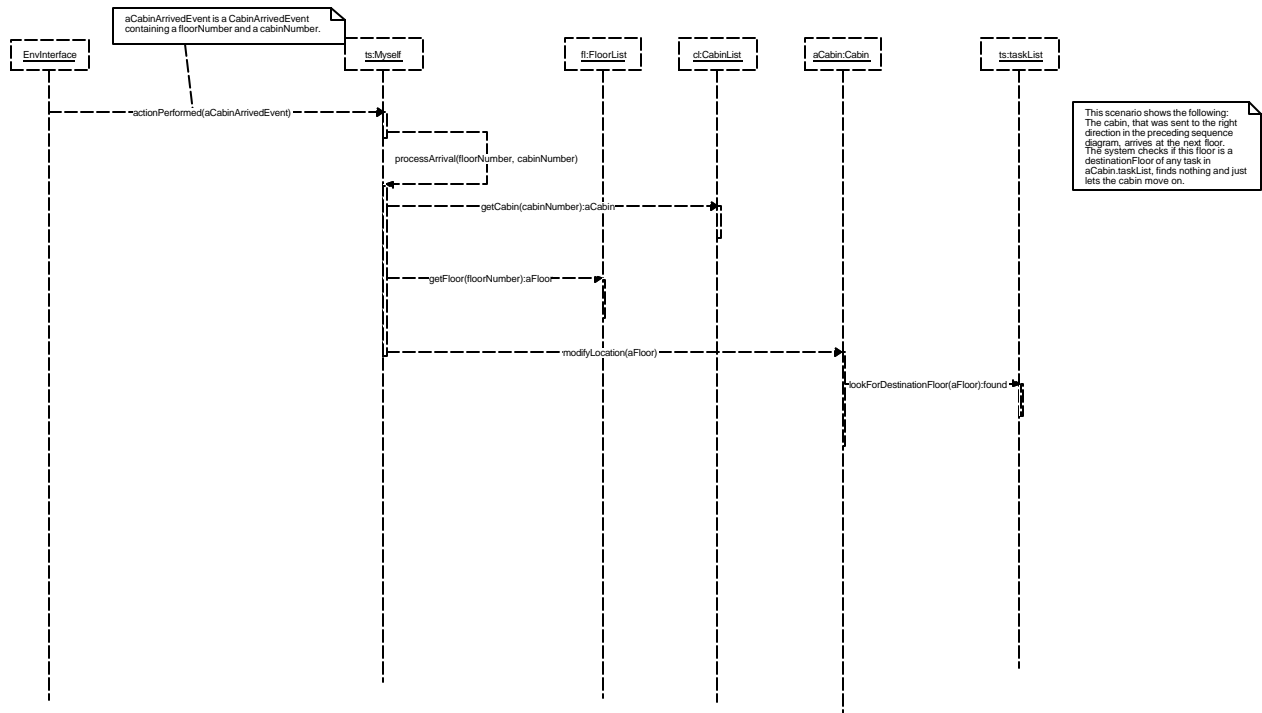


Figure 27. 11:SI\_Scenario<moveCabin> (Sequence Diagram)

## 12:SI\_Scenario<fulfillTaskEndNoOne> (Sequence Diagram)

In this scenario, the TS gets three times an input event and reacts to each of those separately.

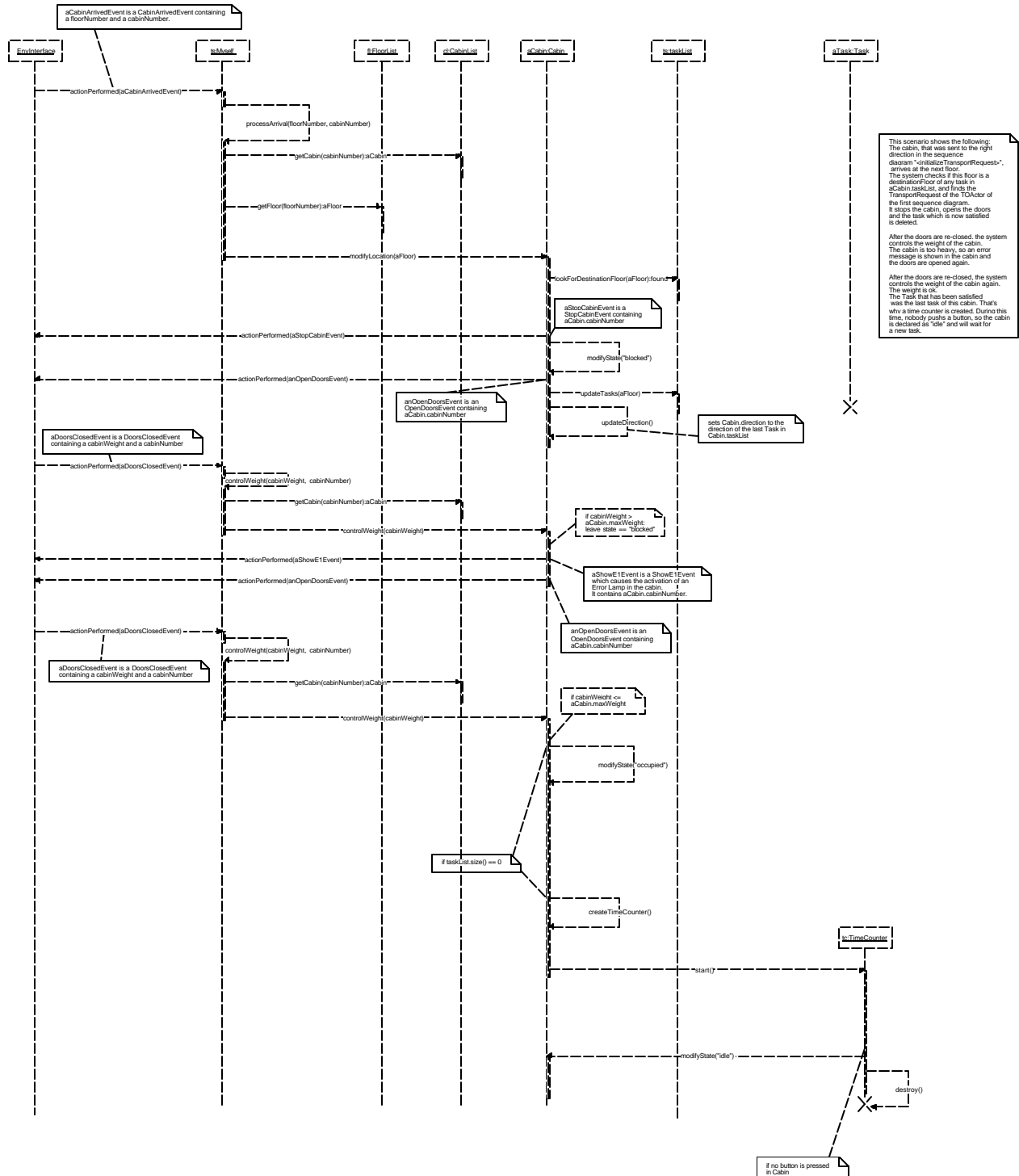


Figure 28. 12:SI\_Scenario<fulfillTaskEndNoOne> (Sequence Diagram)

### 13:SI\_Scenario<fulfillTaskEndNoTwo> (Sequence Diagram)

Attention: this scenario represents another possible end for the scenario in diagram 12 (figure 28)!

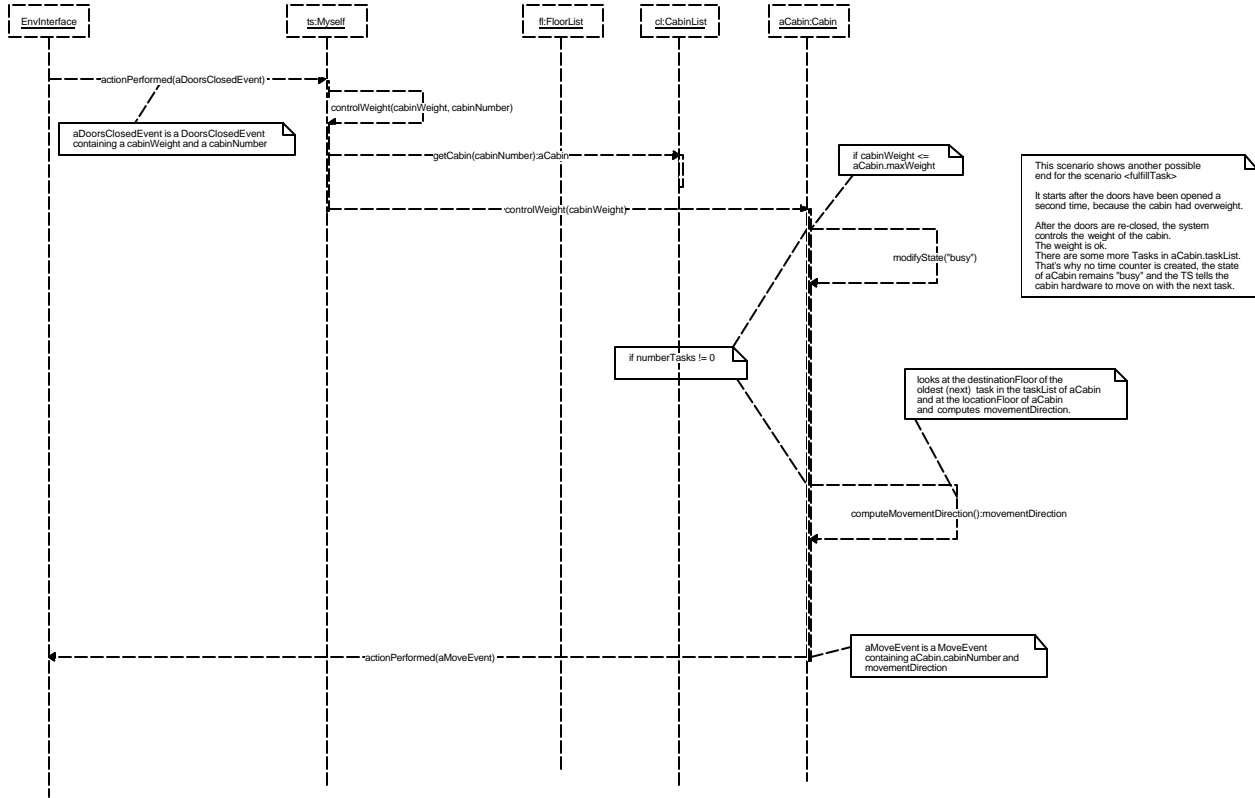


Figure 29. 13:SI\_Scenario<fulfillTaskEndNoTwo> (Sequence Diagram)

## 14:SI\_Scenario<fulfillTaskEndNoThreeAndInitializeTransportTxn> (Sequence Diagram)

Attention: this scenario shows yet another end for the scenario in diagram12 (figure28)!

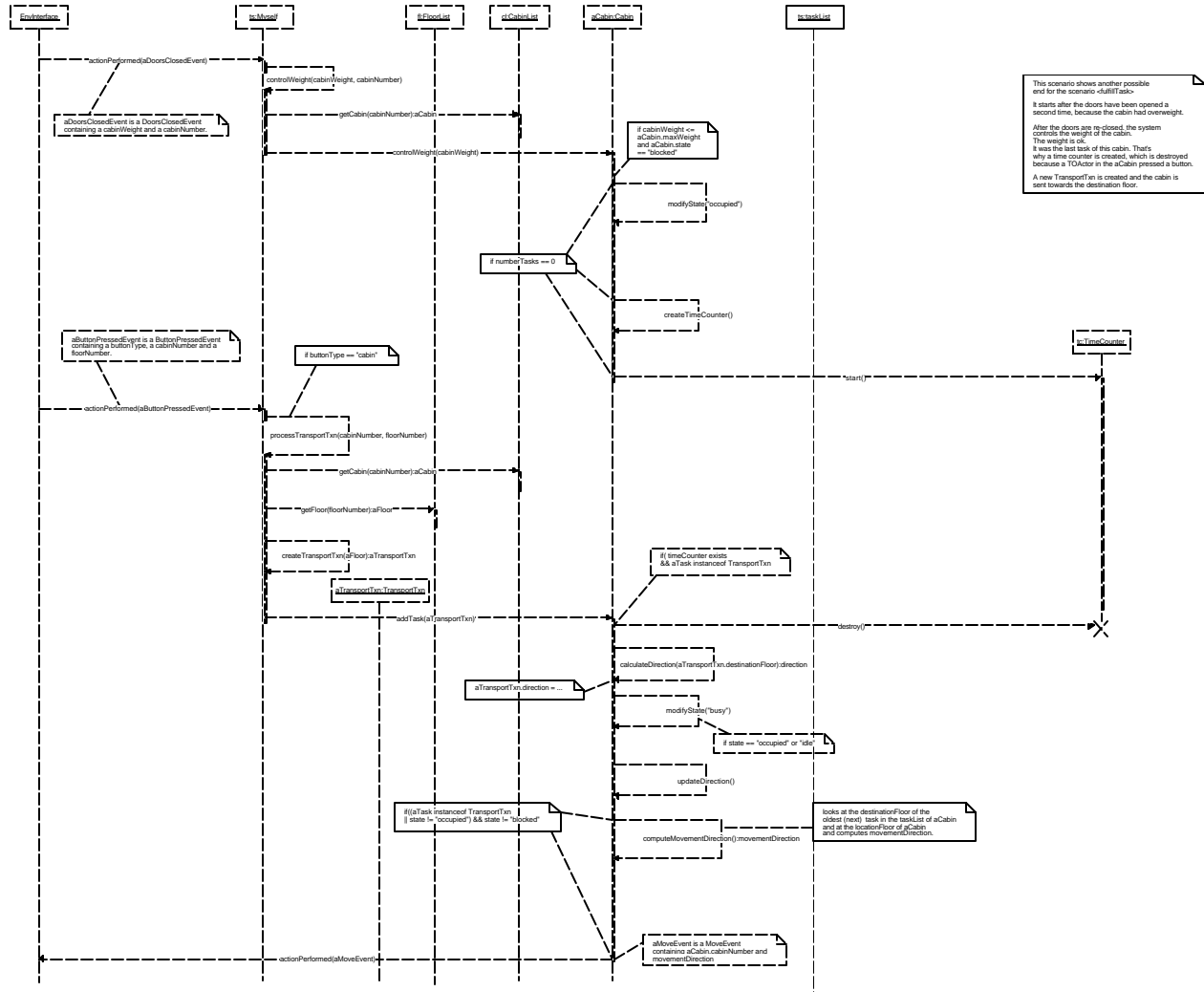


Figure 30. 14:SI\_Scenario<fulfillTaskEndNoThreeAndInitializeTransportTxn> (Sequence Diagram)

You have probably noticed, that most of the work is done by instances of the concept Cabin. This seemed the most logical way to me. I tried to delegate tasks to objects with less knowledge of the system whenever possible. So, as the taskList and the locationFloor of the current Cabin are used in almost every action, the work is very often done by the Cabin.

## 2.5.2 Implementation

The final modelling step is the complete class diagram. I took the object diagram of the IT System Spec and just had to add the functions and the new attributes to each concept.

This was an easy job, because once all the sequence diagrams were carefully finished, I just had to go through them and find out what function has to be in which concept.

Note that I omitted the InputEvent concept. This concept was not known by Myself anyway, and it has been split up to several sub-events as you could see in the sequence diagrams and you will see in the class diagram "SI\_Events".

To allow parallelisms of different Use Case instances was again the reason why I decided to create an own thread for each TimeCounter instance. A TimeCounter will as a Thread be able to run in parallel to the main process of the application and can be stopped or influence the system whenever needed.

## 20\_SI\_Object (Class Diagram)

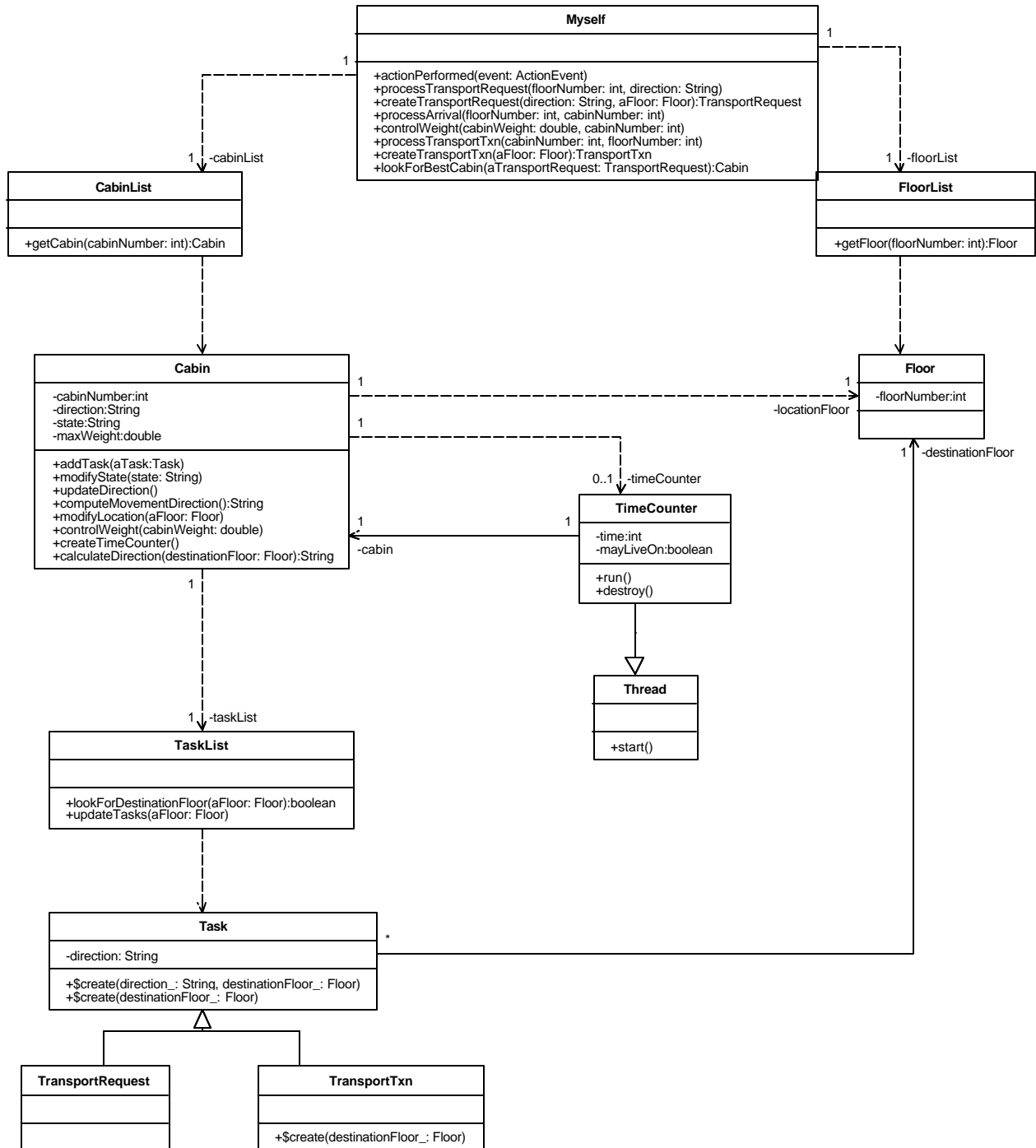


Figure 31. 20\_SI\_Object (Class Diagram)

## 21\_SI\_Events (Class Diagram)

You see that each event inherits from the ActionEvent class, that belongs to the java.awt.event package. Like this, Myself just has to possess a normal ActionListener function (and, for this reason, has to implement ActionListener), that can be called using such an event.

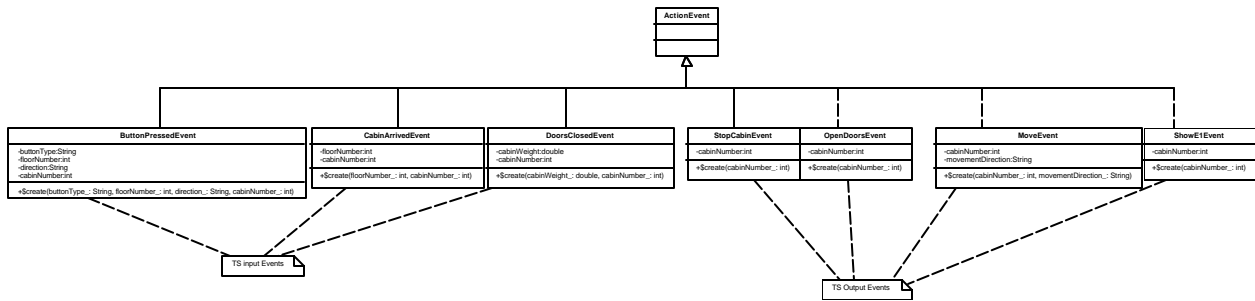


Figure 32. 21\_SI\_Events (Class Diagram)

## 22\_SI\_Technology (Class Diagram)

This diagram shows that the three lists of the TS inherit all from java.util.ArrayList. ArrayLists are very useful for handling a dynamic number of instances of the same type. But they also give a very nice object-oriented representation of multiplicity in general (better than arrays).

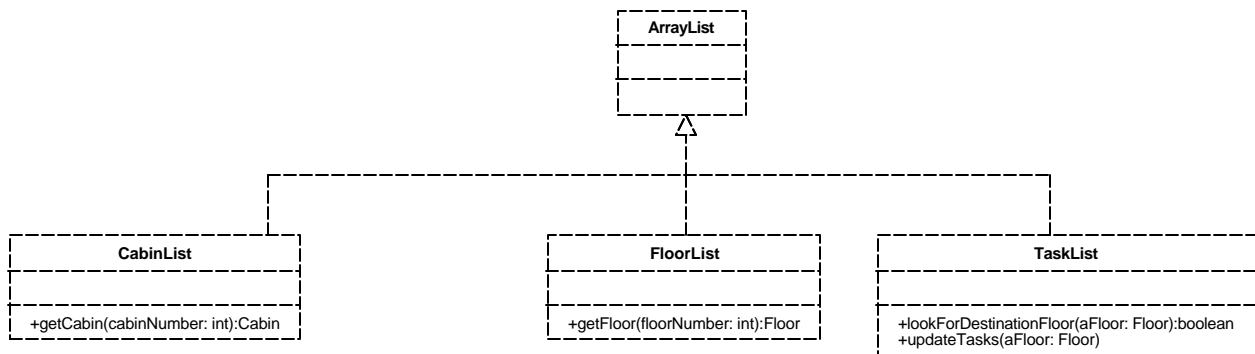


Figure 33. 22\_SI\_Technology (Class Diagram)

### 2.5.3 The simulator

Now, all this is very nice, but how to test the TS, once it is programmed? I needed a way to send input events to the TS and receive output events from it, to test if the TS works fine. This is done by an interface I called “simulator”, even if it does not simulate an elevator at all. It just gives the possibility to send and receive events using a graphical user interface.

That’s why in the following class diagram, a new class is shown, the TransportSimulator. It has a constructor without parameters, because it has to be able to initialize all it’s attributes itself, and it inherits from java.awt.Frame.

You can see in the sequence diagrams, that all possible output events are sent to the hardware by a Cabin. For this reason, the only concept that needs to know the TransportSimulator (to be able to call it’s ActionPerformed function) is the Cabin.

### 23\_SI\_Simulator (Class Diagram)

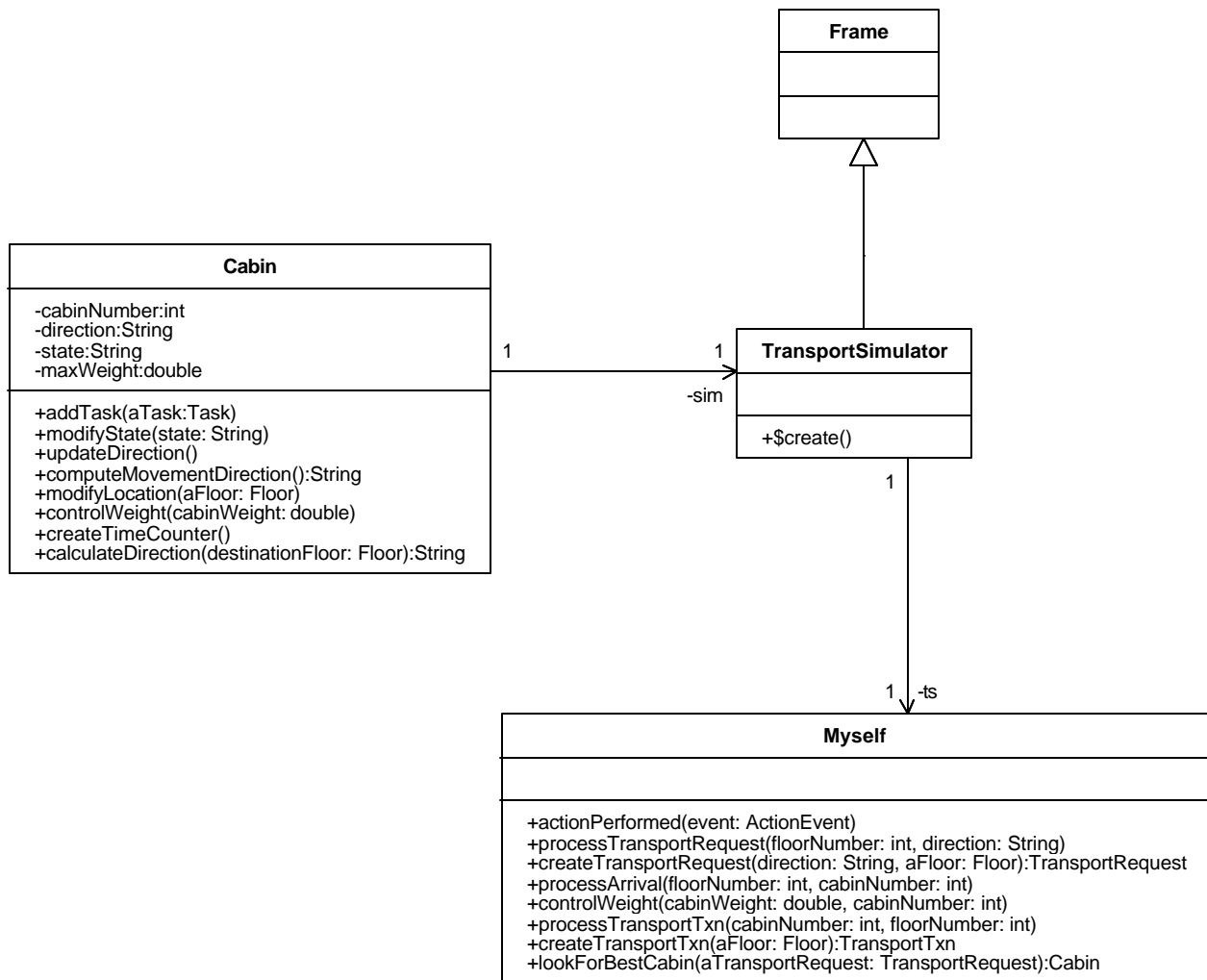


Figure 34. 23\_SI\_Simulator (Class Diagram)

## **2.6 Java implementation**

### **2.6.1 The code**

#### **Code generation**

Cool:Jex has a plug-in that is able to generate ready-to-use Java code from the class diagrams. I decided to design the GUI of the TransportSimulator with Symantec's Visual Café, version 4.0. Unfortunately, Visual Café writes lots of comment tags into the frame files to allow a graphical design of the awt elements. Cool:Jex needs it's own tags for re-generation of the code over existing files. That's why I made two subfolders in my ProjectCode-folder: one called CafeCode, the other called CoolJexCode. The TransportSimulator.java file is in both of the folders, but the version in the CooljexCode folder is not used by the Visual Café project.

This little trick gave me also the possibility to add the classes AttentionDialog and QuitDialog to the project, that helped me make the application a little more comfortable.

#### **TestThread application**

To test the correct functionality of the Thread-TimeCounter Idea, I made a small test-application that shows how TimeCounter work in my system.

You will find this application in the folder TestThread in the project directory. The file to open is called TestThread.vcp.

In this application, you have two buttons, one to create a TestThread, the other to kill it. If you do not kill it within a certain time, it will die. Look at the standard output to see the messages of the application. (Run the project without debugger using Ctrl-F5!)

Attention: you may only create one TestThread a time!

#### **Other coding information**

To simplify the application, I limited the number of floors to 7 and the number of cabins to 2. The maximum weight of each cabin is set to 600. Of course, the TS Software (everything but the TransportSimulator) is designed to cooperate with any simulator or hardware, having as many cabins or floors as desired.

Those values are set after creation of Myself. Look at the constructor of TransportSimulator to see how this is done. You can also change the numbers of floors and cabins at that place, if you want.

### **2.6.2 The lookForBestCabin – algorithm**

This algorithm has the task to find the best cabin for a new entered TransportRequest. TransportTxns are entered in a cabin and are therefore not concerned by this problem.

The algorithm, that is located in the function lookForBestCabin in the Myself class, does the following:

1. If an idle cabin is found, take it. (Idle means, it is not moving and it's TaskList is empty.)  
if not:
2. If a cabin is found that is moving towards the floor the request came from and that has the right final direction, take it. (Cabin.direction corresponds to the direction of the last task in Cabin.taskList.)  
if not:
3. If a cabin is found that is moving towards the floor the request came from and that is at the moment moving to the right direction and will certainly move in this direction to the last floor of the building, take it.  
if not:
4. If there is a cabin that is neither occupied nor blocked (means: busy), take it. (A cabin is occupied if it's doors are closed but it has a TimeCounter running.)  
if not:
5. If there is a cabin that is not blocked, take it. (A cabin is blocked as long as it's doors are open.)  
if not:
6. Take any cabin.

If one of those tests returns more than one cabin, take the one with the shortest TaskList.

## 2.6.3 Simulator Tutorial

You will find the complete TransportSimulator in the folder ProjectCode. Open the file transport.vep in Visual Café.

### Initial Situation:

After the startup of the system, both cabins are on floor 1, have their doors closed, and are idle.

### Legal input events:

- ButtonpressedEvent / buttonType = "floor":  
This means a button has been pressed on a floor. You have to specify the floorNumber and the direction in which a TOActor wants to travel.
- ButtonpressedEvent / buttonType = "cabin":  
This means a button has been pressed inside a cabin. You have to specify the cabinNumber and the number of the floor to which a TOActor wants to travel.
- CabinArrivedEvent:  
You need to send this event to the TS each time a cabin has moved from one floor to another. You have to specify the cabinNumber and the number of the floor on which the cabin has arrived.
- DoorsClosedEvent:  
You may only send this event if an OpenDoorsEvent has been output by the TS for this cabin. You have to specify the cabinNumber and the total weight of the people in the cabin.

## Output events:

-StopCabinEvent:

This means the cabin corresponding to the attached cabinNumber has to stop immediately.

- OpenDoorsEvent

This means the doors of the cabin corresponding to the attached cabinNumber has to open it's doors.

- MoveEvent

This means the cabin corresponding to the attached cabinNumber has to start now moving to the specified movementDirection.

- ShowE1Event

This means that a red alarm light has to be activated in the cabin corresponding to the attached cabinNumber, indicating overweight.

## Idea:

It is recommended to make a small drawing of the scenario you want to play on a sheet of paper. Afterwards, take care to really input one event after the other in a legal order. If you want for instance to move cabin 1 from floor 1 to floor 4, you have to send three times a CabinArrivedEvent:

cabinNumber = 1, floorNumber = 2;

cabinNumber = 1, floorNumber = 3;

cabinNumber = 1, floorNumber = 4;

And, of course, this is only legal if cabin 1 has received a MoveEvent from the TS with movement-Direction="up".

You should also try to represent the current state of a cabin on your drawing.

Doors can be open or closed, a cabin can be moving to a certain direction or idle.

Try to represent the different TransportedObjectActors on your drawing. This will help you to remember which Tasks have been satisfied and which TOs are still waiting.

## Example:

On floor 2, there is a TO wanting to go down.

On floor 7, there is a TO wanting to go down, too.

After they have entered their requests in this order, there is a TO on floor 4, wanting to go up.

### Input-event

*Output-event*

Comment.

- **ButtonPressedEvent / buttonType="floor" / floorNumber=2 / direction="down"**

*System output: MoveEvent / cabinNumber=1 / direction="up"*

**- ButtonPressedEvent / buttonType="floor" / floorNumber=7 / direction="down"**  
*System output: MoveEvent / cabinNumber=2 / direction="up"*

We see that the first task has been given to cabin 1 and the second to cabin 2.

**- ButtonPressedEvent / buttonType="floor" / floorNumber=4 / direction="up"**  
*System output: no output events*

The task has been added to one cabin, we do not know which one.

**- CabinArrivedEvent / cabinNumber=1 / floorNumber=2**  
*SystemOutput: StopCabinEvent / cabinNumber=1 followed by OpenDoorsEvent / cabinNumber=1*

So first task has been satisfied.  
now, let's move the second cabin as ordered by the system:

**- CabinArrivedEvent / cabinNumber=2 / floorNumber=2**  
*System output: no output events*

**- CabinArrivedEvent / cabinNumber=2 / floorNumber=3**  
*System output: no output events*

**- CabinArrivedEvent / cabinNumber=2 / floorNumber=4**  
*System output: StopCabinEvent / cabinNumber=2 followed by OpenDoorsEvent / cabinNumber=2*

So the third task had been added to cabin 2 and has now been satisfied. This makes sense, because cabin 2 is moving to the top of the building anyway (to fulfill the second task). On it's way up there, it can take with it the TO wanting to go up on floor 4.

Now both cabins have their doors opened, we have to close the doors to let them move on:

**- DoorsClosedEvent / cabinNumber=1 / cabinWeight=80**  
*System output: no output events*

Cabin one is now waiting for an action from it's inner buttons. As there is no action within the next 50 seconds, the cabin will become "idle", but all this is not visible on the interface.

If you want to see what happens inside the Transportation System, you just have to look at the standard output window of Visual Café. I added some println-calls to the code, that indicate the creation and death of a TimeCounter and the state or the direction of a Cabin.

**- DoorsClosedEvent / cabinNumber=2 / cabinWeight=650**  
*System output: ShowE1Event / cabinNumber=2 followed by OpenDoorsEvent / cabinNumber=2*

The cabin is too heavy (cabinWeight > 600), so let's chose a smaller weight:

**- DoorsClosedEvent / cabinNumber=2 / cabinWeight=120**  
*System output: MoveEvent / cabinNumber=2 / direction="up"*

The cabin has to go pick up the TO on floor 7. The TO that entered the cabin can chose now the destination-floor:

**- ButtonPressedEvent / buttonType="cabin" / floorNumber=6 / cabinNumber=2**  
*System output: no output events*

The system has added this new task to cabin 2.

Let's move on the cabin:

**- CabinArrivedEvent / cabinNumber=2 / floorNumber=5**

*System output: no output events*

**- CabinArrivedEvent / cabinNumber=2 / floorNumber=6**

*System output: StopCabinEvent / cabinNumber=2 followed by OpenDoorsEvent / cabinNumber=2*

We let the TO from floor 4 go out of the cabin.

**- DoorsClosedEvent / cabinNumber=2 / cabinWeight=0**

*System output: MoveEvent / cabinNumber=2 / direction="up"*

The cabin still moves towards floor 7

**- CabinArrivedEvent / cabinNumber=2 / floorNumber=7**

*System output: StopCabinEvent / cabinNumber=2 followed by OpenDoorsEvent / cabinNumber=2*

The TO on floor 7 has finally a cabin to travel downstairs.

This is the best solution for this problem, because cabin 1 was requested to floor 2 with the wish to go downwards. That's why it would not have been a good idea to add the third task to this cabin.

Of course, the TS does not always find such a good solution. You will surely find situations in which the system is not reacting the best way. But the aim of this software is not to implement a perfect algorithm, but to show that the UML model works.

## 2.7 Conclusion

### What has been reached:

My work so far has brought the following results:

- an easy-to-understand demonstration of the possibilities of UML and Catalysis. Everyone knows how an elevator works, and I have shown on this common example how to make a (more or less) well-structured, object-oriented model of it's controlling software. Such a model allows now the detailed optimization of the "lookForBestCabin"-algorithm.
- the use of events as a communication interface. In this model, the advantages of events are clearly recognizable. Using events makes it extremely easy to build our system into any type of simulator or hardware.
- a nice software that proofs once more that the applied modeling methods work fine.

### But...

My elevator does **not** work better than any other elevator.

The reason for this is:

The hardware was not changed at all. Actually used software for standard elevators has been developed over several years and works well. It is almost impossible to invent a software that reaches a much better performance than commercial elevators using the same hardware.

I also had to do the least methodical part, the creation of the "lookForBestCabin"-algorithm, by hand. There is no technique in UML that allows the optimization of such an algorithm. If I tried to avoid this algorithm, I would maybe get a better elevator.

But, even if I had found a better elevator software, this would not have been very useful for finding solutions to general transport problems, because my approach is too much restricted by the given hardware requirements.

### New Idea:

To model the whole transportation system, and not only it's software, to keep the model as general as possible and to try to find a better transport system.

Often, informational systems are built starting with a precise idea about what the result should look like. This is the reason why often, the solutions are not very creative, they are what one expected, and just work more or less well. In the next approach, I tried to define only the most basic conditions for the situation before and after the work of the system, without fixing how this will happen. Like this, I hope we will find some new aspects of a transportation system.

### **3. EXTENDED TRANSPORTATION SYSTEM**

The most important difference to the previous approach is, that the Extended Transportation System is considered as a complete system, able to transport TransportedObjects from one floor to another. We do not specify the exact form of this system, as we had done it before. The system can be an elevator, an escalator, etc... The system's software is represented as a black box, that is the "brain", and that is supposed to work. With this approach, it should be possible to find a better and more general model for any transportation system. To design the software for the system would then be the next step.

Basic ideas are:

- a TransportedObjectActor is identified by the system. For the example of an elevator, this means: the system knows which person entering a cabin has pressed which button.
- TransportedObjectActors have to specify the destinationFloor at the moment they request a transport. This will be a great help for the optimization of the system.
- TransportedObjectActors can cancel the requested transport as long as they are not yet being transported.
- The wishes entered by TransportedObjectActors are managed by the ETS itself, centrally, and not given to a transport-mean (like the cabin). This should let the system be able to react more flexible to changes of the wishes.

### **3.1 Business Implementation Spec**

#### **3.1.1 Snapshots**

As you will see in the Snapshot Diagrams, there are no cabins or similar elevator-hardware objects. This is because at this point of abstraction, the ETS is considered as one actor, the decomposition of it into its parts will happen in the next phase.

You will also notice that I did no longer differentiate between TransportRequest and TransportTxn. The only "thing" a TOActor can create is a WishObject, that is a representation of its real "to-be-transported" wish.

All we know and are able to draw is the situation before, during and after the existence of such a WishObject.

### 10:BI\_Snapshot<initial> (Collaboration Diagram)

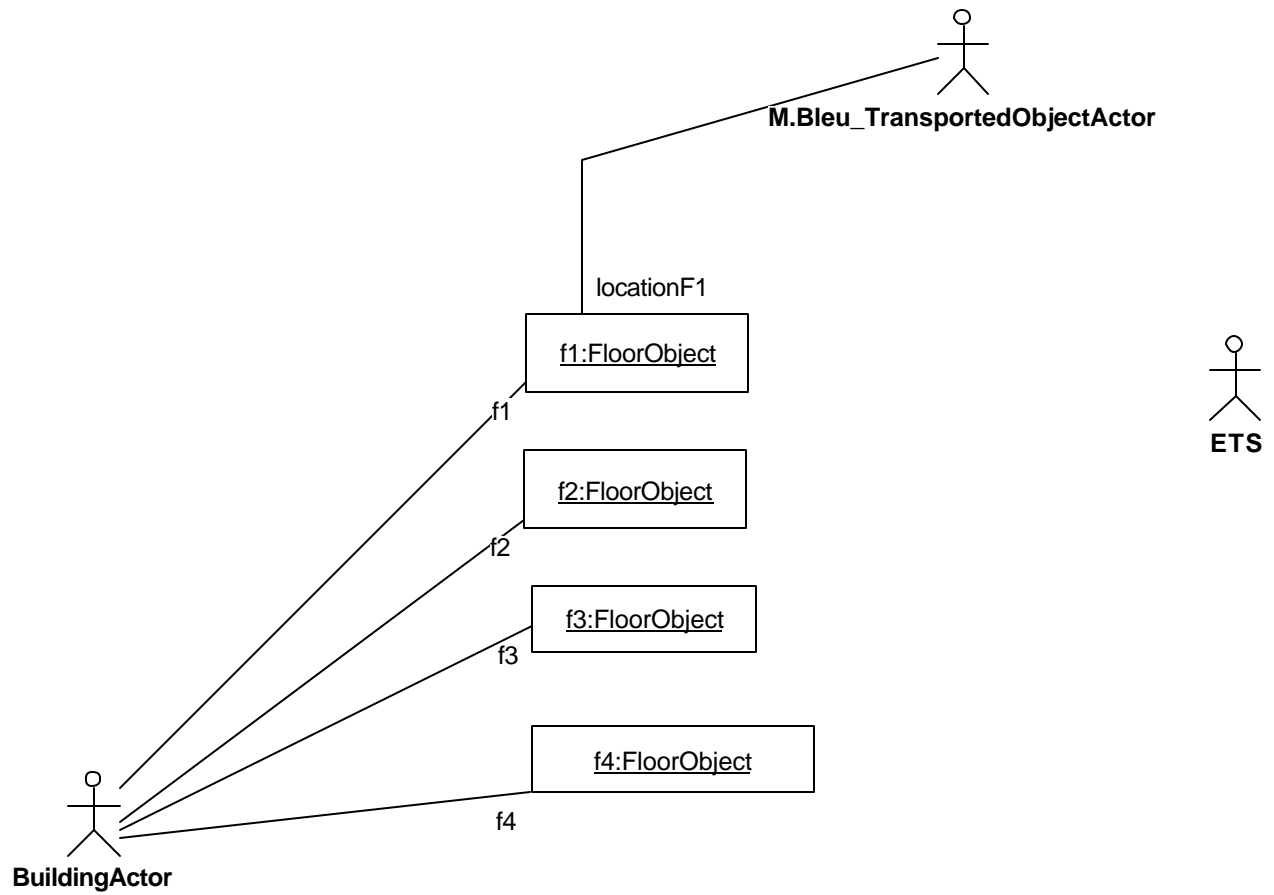


Figure 1. 10:BI\_Snapshot<initial> (Collaboration Diagram)

### 11:BI\_Snapshot<afterWishEntered> (Collaboration Diagram)

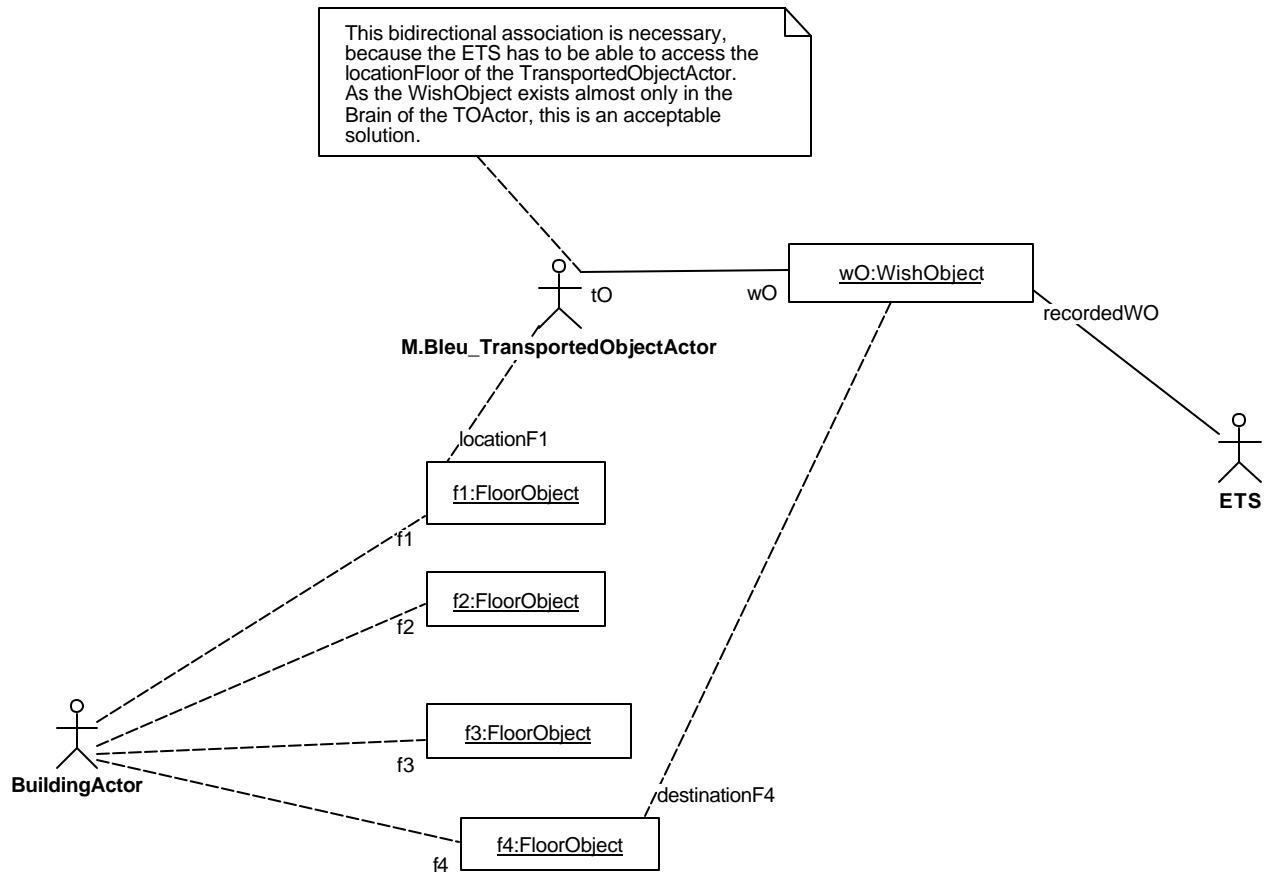


Figure 2. 11:BI\_Snapshot<afterWishEntered> (Collaboration Diagram)

### 12:BI\_Snapshot<final> (Collaboration Diagram)

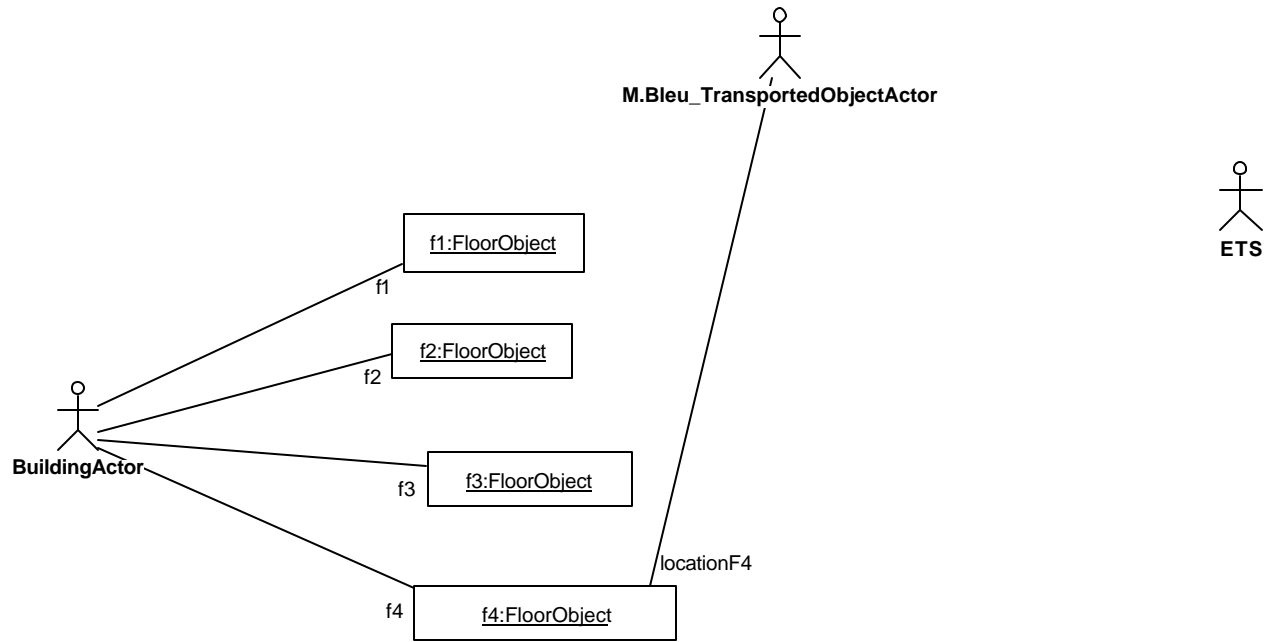


Figure 3. 12:BI\_Snapshot<final> (Collaboration Diagram)

### 3.1.2 Collaboration Model

The Collaboration Diagram looks exactly as the one for the previous Transportation System, but it's description has changed a lot.

One transport Use Case represents the satisfaction of one wish of one TransportedObjectActor, and that's it.

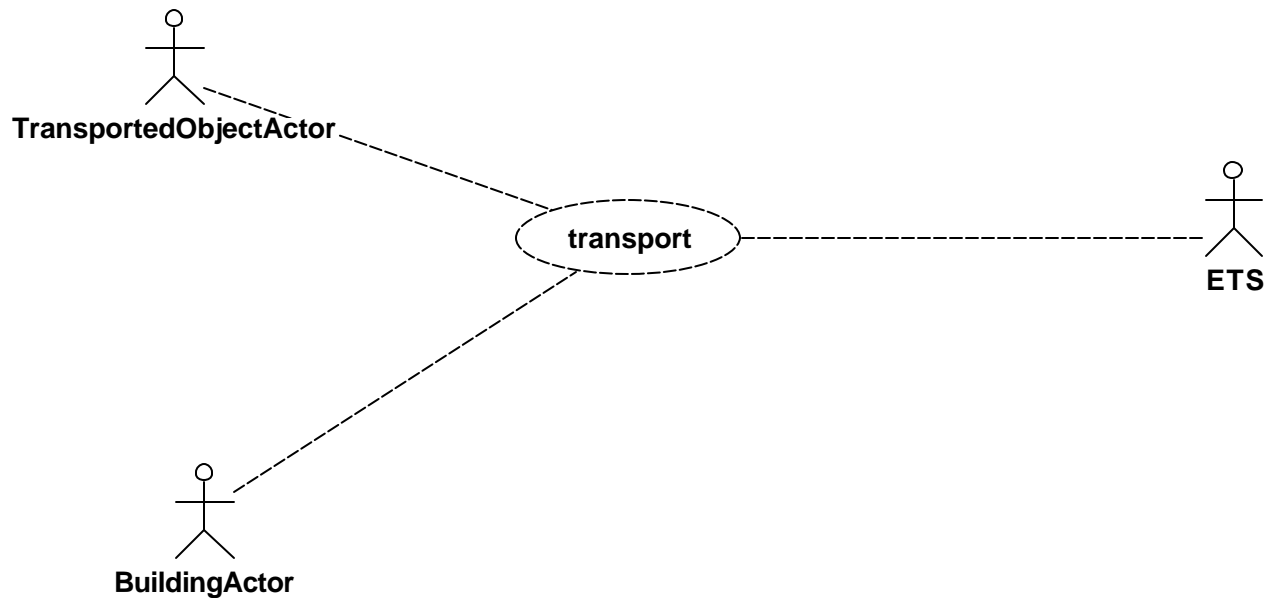


Figure 4. 20:BI\_Collaboration (Collaboration Diagram)

### Collaboration Description

#### Collaboration: transport

##### Purpose

Satisfy the Wish of a TransportedObject. A Wish is: get from one floor to another floor.

##### Policies

UC\_P1: A Wish can be expressed by any actor interacting with the Extended Transportation System. For reason of simplicity, this actor is not represented in the diagram: we assume the TransportedObject enters the Wish. At this level of abstraction, this is acceptable; it does not matter who expressed the Wish.

UC\_P2: The ExtendedTransportationSystem should provide the necessary hard- and software to fulfill a Wish (or more Wishes at the same time).

UC\_P3: The ExtendedTransportationSystem should be able to process several Wishes at the same time. Each of those Wishes is treated in one cycle of the transport Use Case (several instances of this UC can occur at the same time).

UC\_P4: The ratio  $(\text{arrival\_time} - \text{request\_time}) / \text{abs}(\text{destination\_floor} - \text{departure\_floor})$  should be as small as possible for each Wish!

### **Parameters**

one TransportedObject, one locationFloor, one destinationFloor

### **Pre-Conditions**

A Building with a certain number of Floors exists.  
TransportedObject exists.

### **Post-Conditions**

The Wish is satisfied, i.e. the TransportedObject is now associated with the destinationFloor of the Wish.  
The Wish has disappeared.

### 3.1.3 Objects Model

In this diagram, only the objects that are not a part of the system are represented. The WishObject is a kind of a representation of an idea in the brain of the TransportedObjectActor, but in the viewpoint of an external observer. All we see in this WishObject is that a certain TOActor wants to be transported from its current locationFloor to a certain destinationFloor. If it was the real wish inside the TOActor's head, we would also know why it wants to be transported, how important the wish is to it etc.

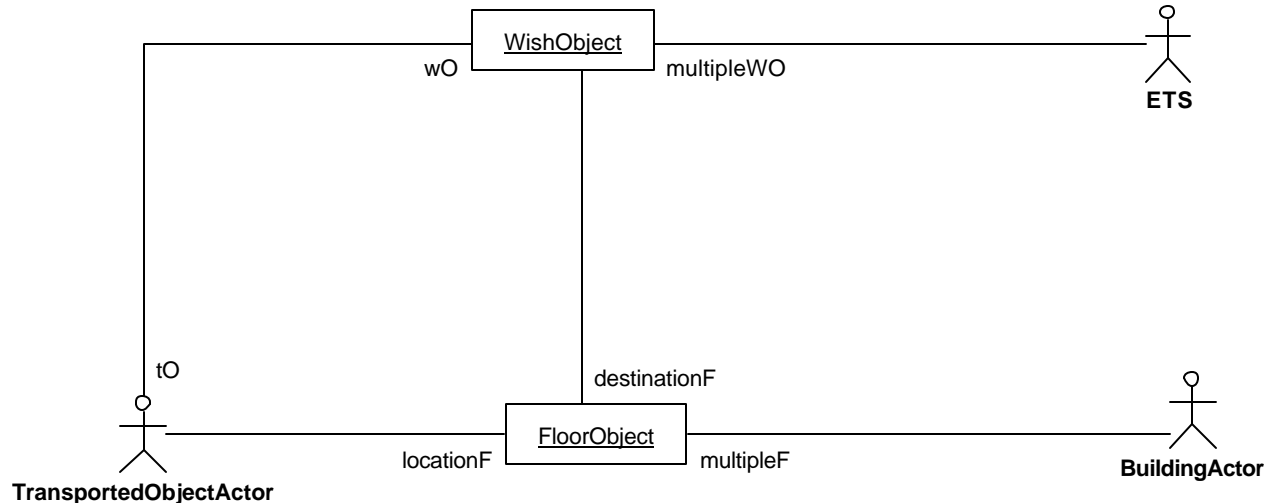


Figure 5. 21:BI\_Object (Collaboration Diagram)

## Glossary

### BuildingActor

Model representing the building. A BuildingActor can have several FloorObjects. The only action of this actor is to "tell" the TransportedObject on which floor it is.

### ETS

Model of the Extended Transportation System, which represents any imaginable solution to transport TransportedObjects from one Floor to another.

### TransportedObjectActor

Model of an object that "wants" to be transported. As we assume that it has expressed the

Wish itself, it is an actor.

### **FloorObject**

Model representing one floor of the building, which can be served by the ETS.

### **WishObject**

Model of the wish a TransportedObject has communicated to the ETS. A Wish can only be: to be transported from one locationFloor to a specified Floor.

## **3.2 System Spec**

In the System Specification (I dropped the “Information Technology” in the title, because we are building a non-informational model.), I tried to show the concepts that are within the ETS. But this time, we have to be very careful about what an “internal representation” of an object or an actor is.

### **3.2.1 Snapshots**

In this first diagram, for instance, there are “system-internal” concepts, that trace the FloorObjects. A FloorObject was defined above as the representation of a floor in the viewpoint of an third-person observer. Now, what is the representation of a FloorObject in the system? It is, what the system knows about a floor, i.e. where the floor can be found, i.e. it is the connection of the system to a floor.

The TransportMean concept could for instance be a cabin. Such an object has to be present in the system, but it has no representation outside the system’s viewpoint.

If I wanted, I could have connected a concept called “brain” or “computer” to the ETS. This would have been the blackbox representing the software. But, as the software has to communicate with all the other concepts of the system, we can simplify the diagram by saying that the software is built into the ETS.

### 10:S\_Snapshot<initial> (Collaboration Diagram)

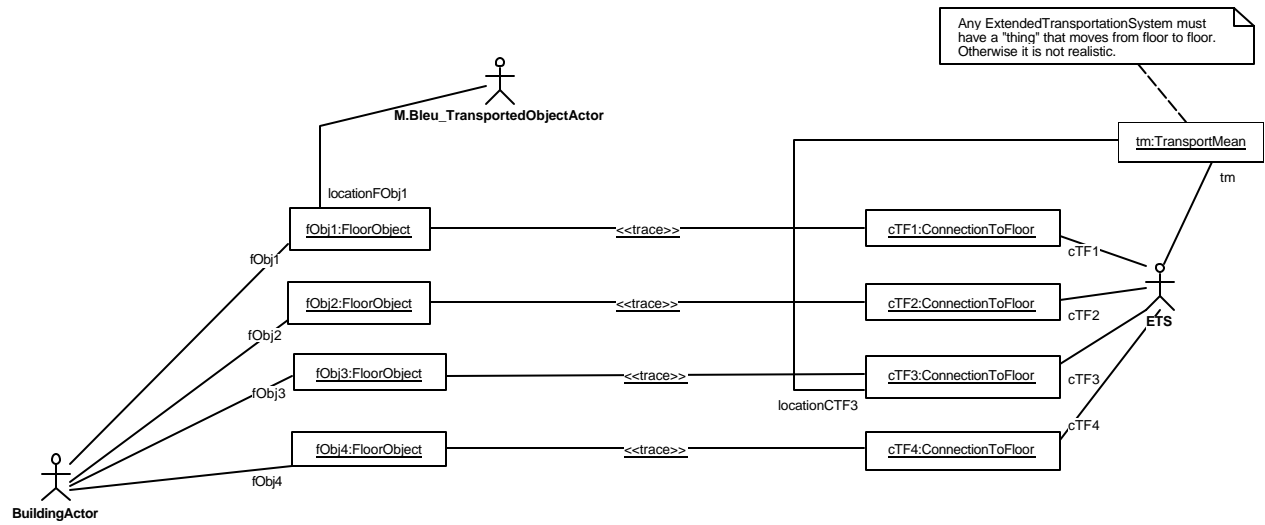


Figure 6. 10:S\_Snapshot<initial> (Collaboration Diagram)

### 11:S\_Snapshot<afterWishEntered> (Collaboration Diagram)

In this diagram, the TOActor M.Bleu is also traced by a “system-internal” concept. How is this possible? Isn’t there only one TransportedObject, that enters into the system? If we draw an internal TransportedObject, don’t we create a software model of it, what we wanted to avoid?

Look at the diagram 14:S\_BusinessModelizationExplanation to find the answers to those questions!

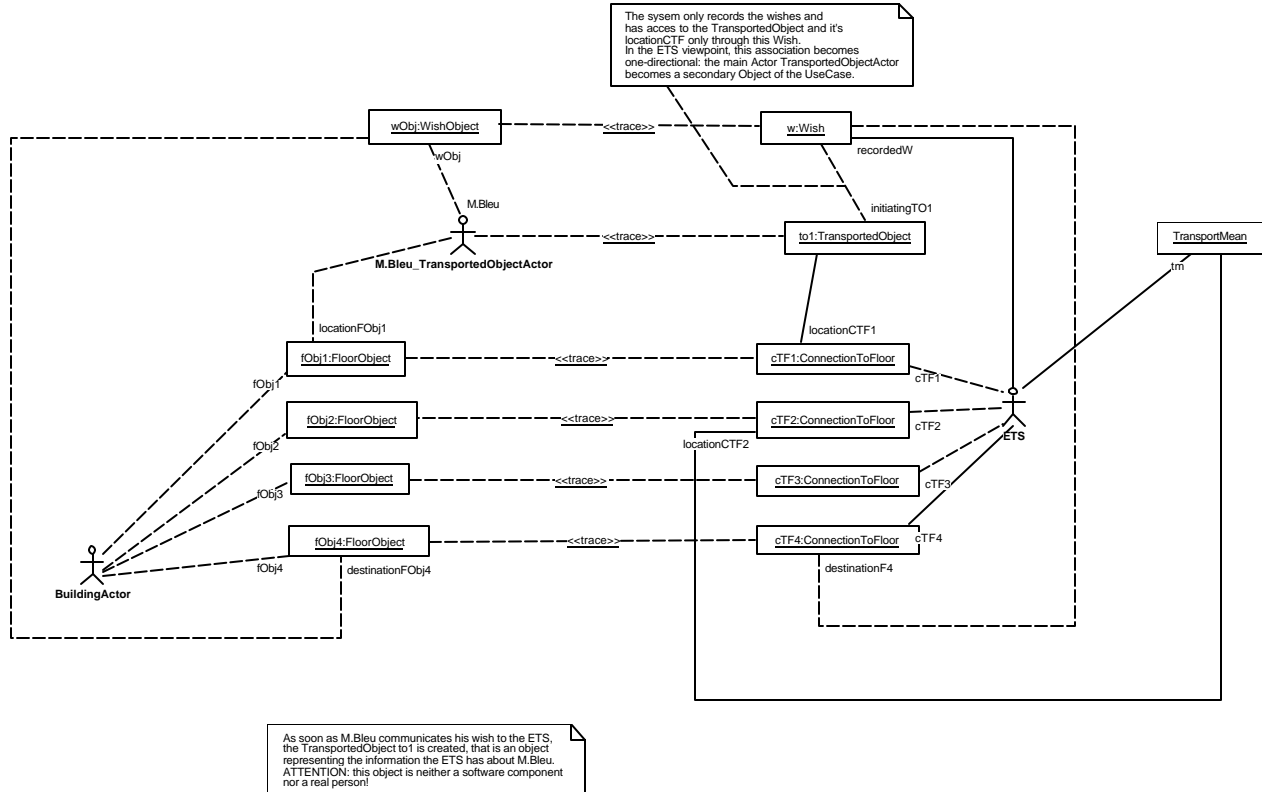


Figure 7. 11:S\_Snapshot<afterWishEntered> (Collaboration Diagram)

## 12:S\_Snapshot<afterTOActorAssociatedWithTransportMean> (Collaboration Diagram)

The association between to1 and the TransportMean means that M.Bleu is now “on board” and being transported. This causes that M.Bleu is no longer able to cancel the transport.

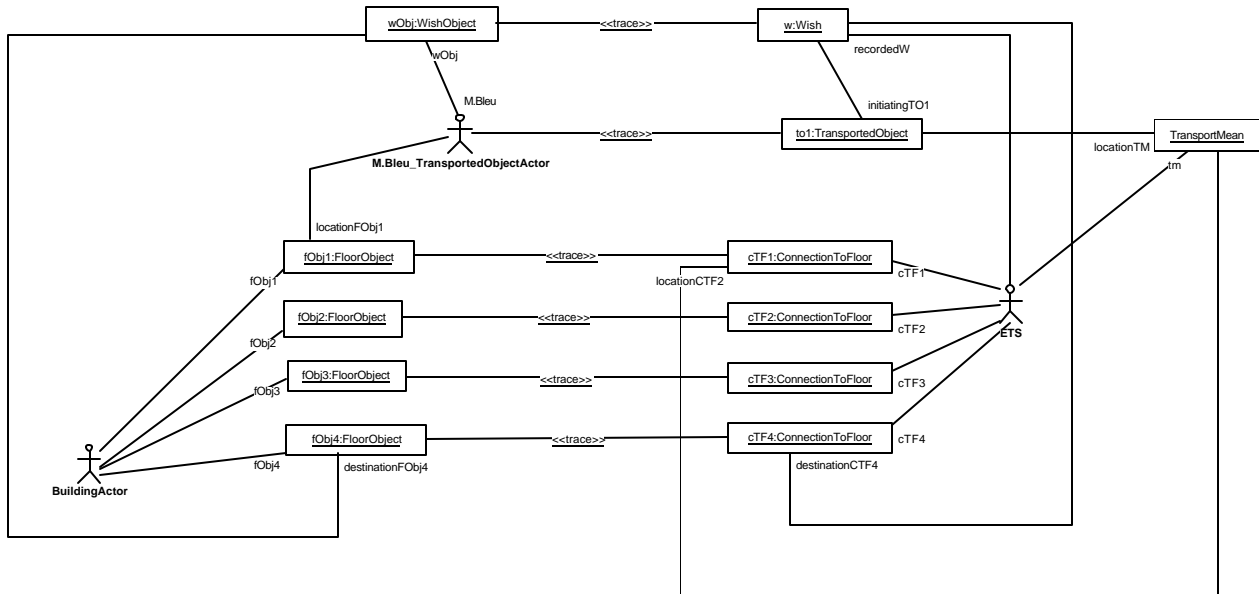


Figure 8. 12:S\_Snapshot<afterTOActorAssociatedWithTransportMean> (Collaboration Diagram)

### 13:S\_Snapshot<final> (Collaboration Diagram)

After the wish of M.Bleu is fulfilled, the internal representation to1 is deleted. If M.Bleu requests another transport, another TransportedObject concept with a different name will be created.

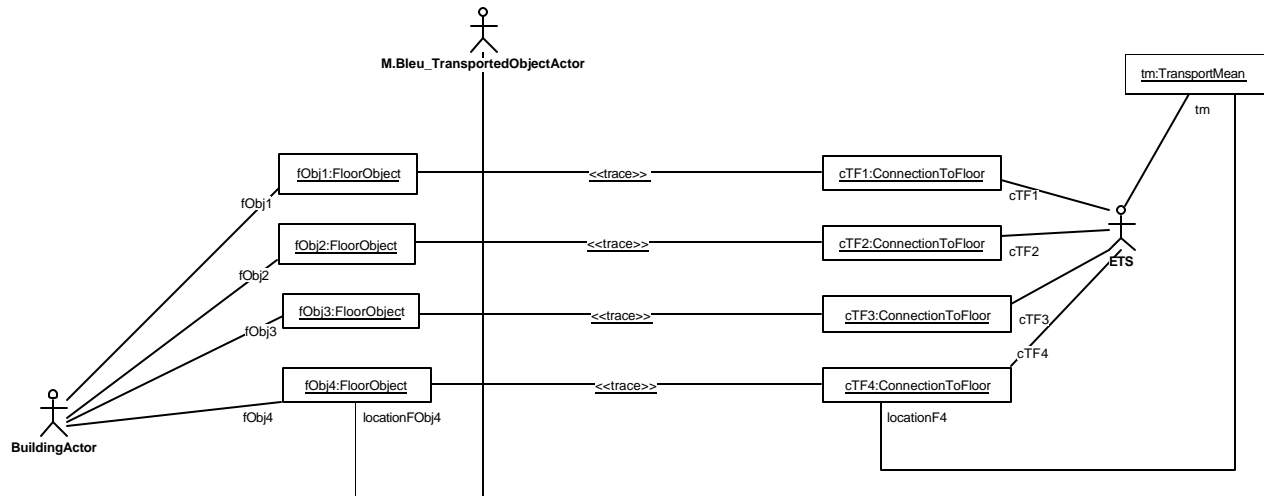


Figure 9. 13:S\_Snapshot<final> (Collaboration Diagram)

### 14:S\_BusinessModelizationExplanation (Collaboration Diagram)

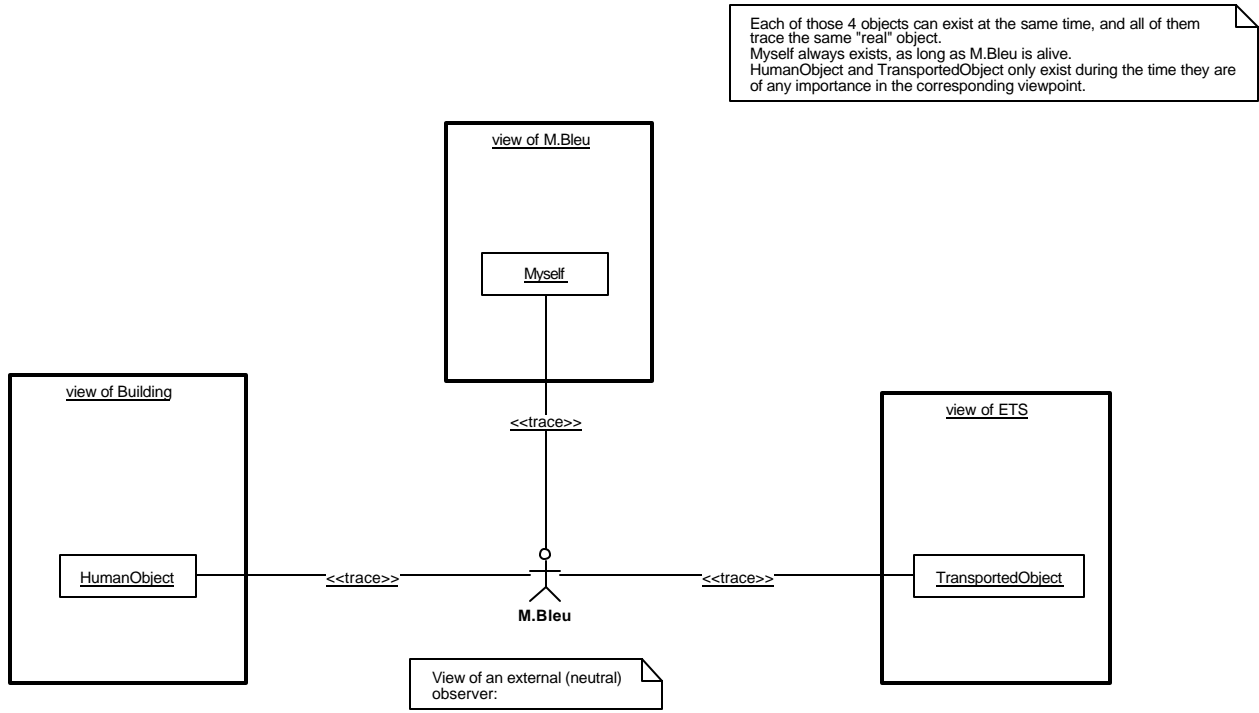
Now, with this diagram I tried to explain the basic idea of those representation concepts, that seem to be a bit confusing if we model a non-software system.

M.Bleu, a TransportedObjectActor, is the representation of a person in the viewpoint of an external observer. This representation is very precise, it really contains everything this observer knows about M.Bleu.

This is the viewpoint of the Business Specification Snapshot Diagrams, and also the viewpoint of the left hand part of the System Specification Snapshot Diagrams above.

In the right hand part of those diagrams, we see the representation of M.Bleu in the viewpoint of ETS. This representation stands for what the ETS knows about M.Bleu, and it is called TransportedObject. Attributes of this concept are for instance the locationConnectionToFloor, a Wish, etc.

This representation exists only as long as the Wish M.Bleu has entered exists. Afterwards, it becomes unimportant to the system and is dropped.



**Figure 10. 14:S\_BusinessModelizationExplanation (Collaboration Diagram)**

### 3.2.2 Use Cases Model

In this section of the ETS model, the diagrams have not changed since the TS model, but take a look at the descriptions, they have changed a lot.

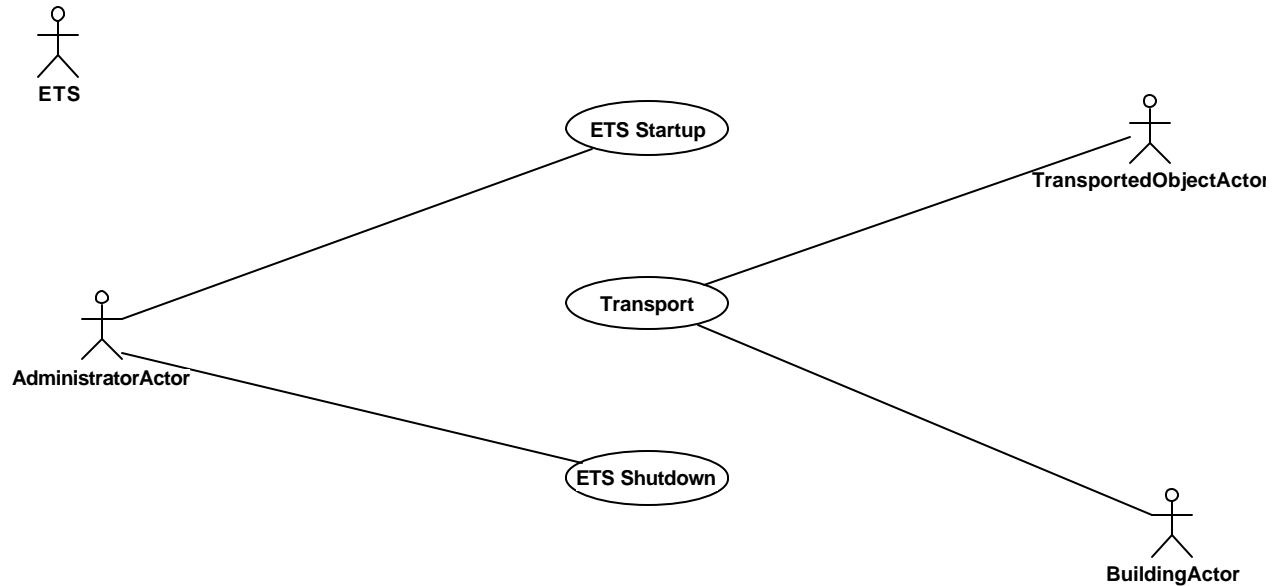


Figure 11. 20\_S\_Usecase (Use Case Diagram)

### System Policies and Patterns

S\_P1: TransportedObjectActor initialises the actions of the ETS.

S\_P2: The ETS has internal representations of the FloorObjects, of the WishObjects, AND of the TransportedObjectActor. Like this, it is able to identify TransportedObjectActors communicating with it, and to record the Tasks to fulfill and the location of a TransportedObjectActor.

S\_P3: The ETS makes sure that no environmental Actor can input illegal information. This aim is reached by limiting the possibilities of the input interfaces ( e.g. several numbered buttons instead of a textfield).

S\_P4: The ETS tries to minimize the transport-time of each TOActor by adapting it's decisions as the number of Wishes evolves.

## Use Case Description

### Use Case: Transport

#### UC Purpose

A Wish of a TransportedObjectActor is satisfied.

#### UC Policies

P1: Each time a TransportedObjectActor communicates with the ETS, it is identified by the ETS (associated with a new or an existing TransportedObject concept).

P2: The representation of a TransportedObjectActor in the ETS is limited to the time during which it is present in the System's environment. (After a fulfilled Wish, the corresponding TransportedObject is deleted.)

P3: The TransportedObjectActor can cancel the UC, as long as it is not yet being transported (as long as the TO is not yet associated with a TransportMean).

P4: Once the TransportedObjectActor is associated with a location TransportMean, it can not change it's Wish anymore. The ETS is now responsible to get it to the right destinationFloor.

#### UC Parameters

IN: one destinationFloorNumber

IN: zero or one CancelParameter

#### UC Pre-Conditions

At least two Floors exist.

ETS is running (the last Action performed by the AdministratorActor was TS Start Up).

#### UC Post-Conditions

Wish does not exist.

The TransportedObjectActor is now associated with the destinationFloor of the (disappeared) Wish.

The TransportedObjectActor has no TransportedObject concept representing it.

## 21:S\_Activity (Activity Diagram)

Here, we have the same parallelism problem as in the TS model. Again, I left this little problem unresolved. This is one of the details that indicate that this approach is not really satisfying. As you will see, I had to start over a third time.

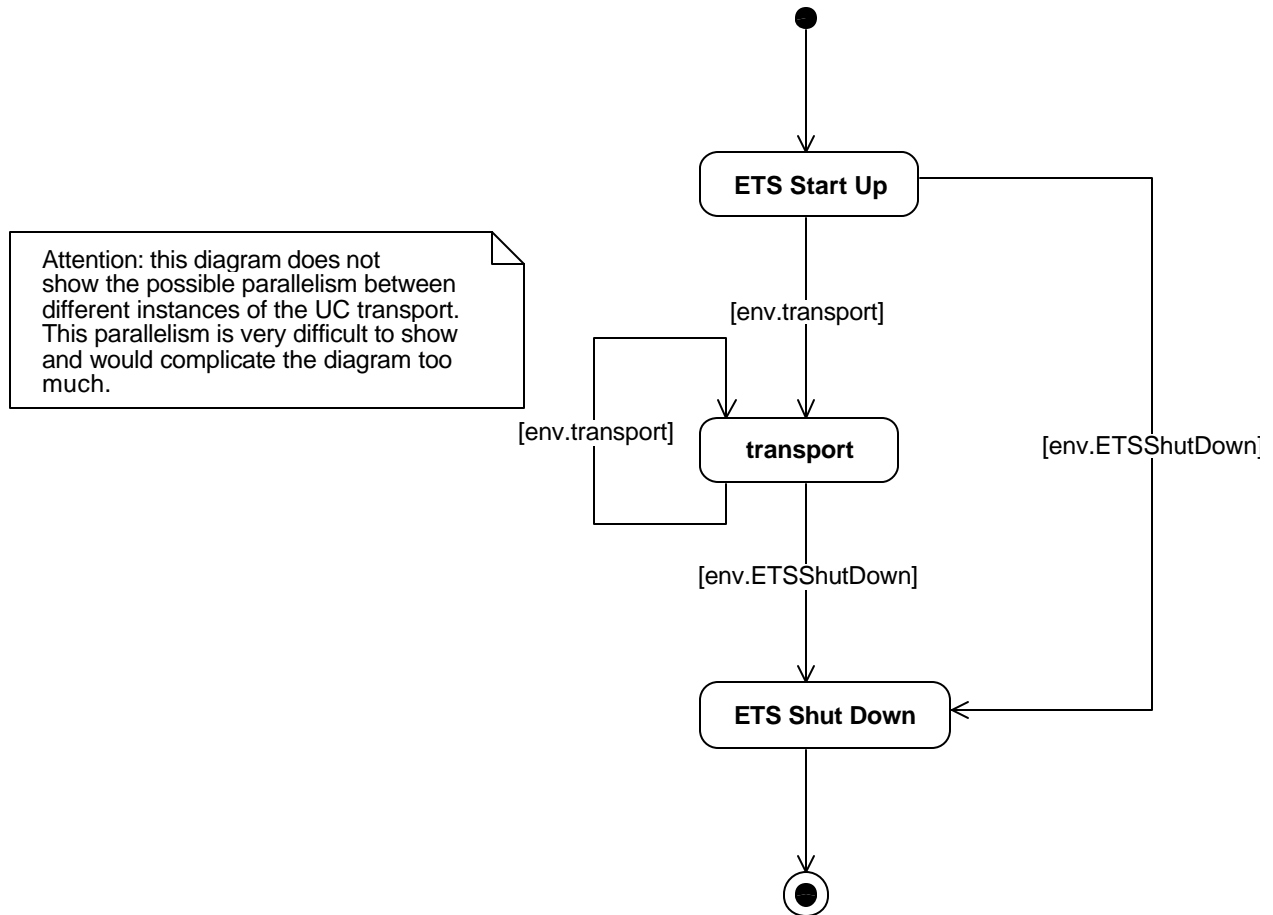


Figure 12. 21:S\_Activity (Activity Diagram)

### 3.2.3 Conceptual Model

#### 22\_S\_Concept (Class Diagram)

The conceptual model was not very difficult to construct. On the System Snapshots, one sees every association that was put into this diagram.

ATTENTION: the three lists may let you think of Java classes again. This is not the case! A list is, as it is written in the glossary, just a concept referencing other concepts. I used them to show that there are multiple TransportMeans, Wishes etc.

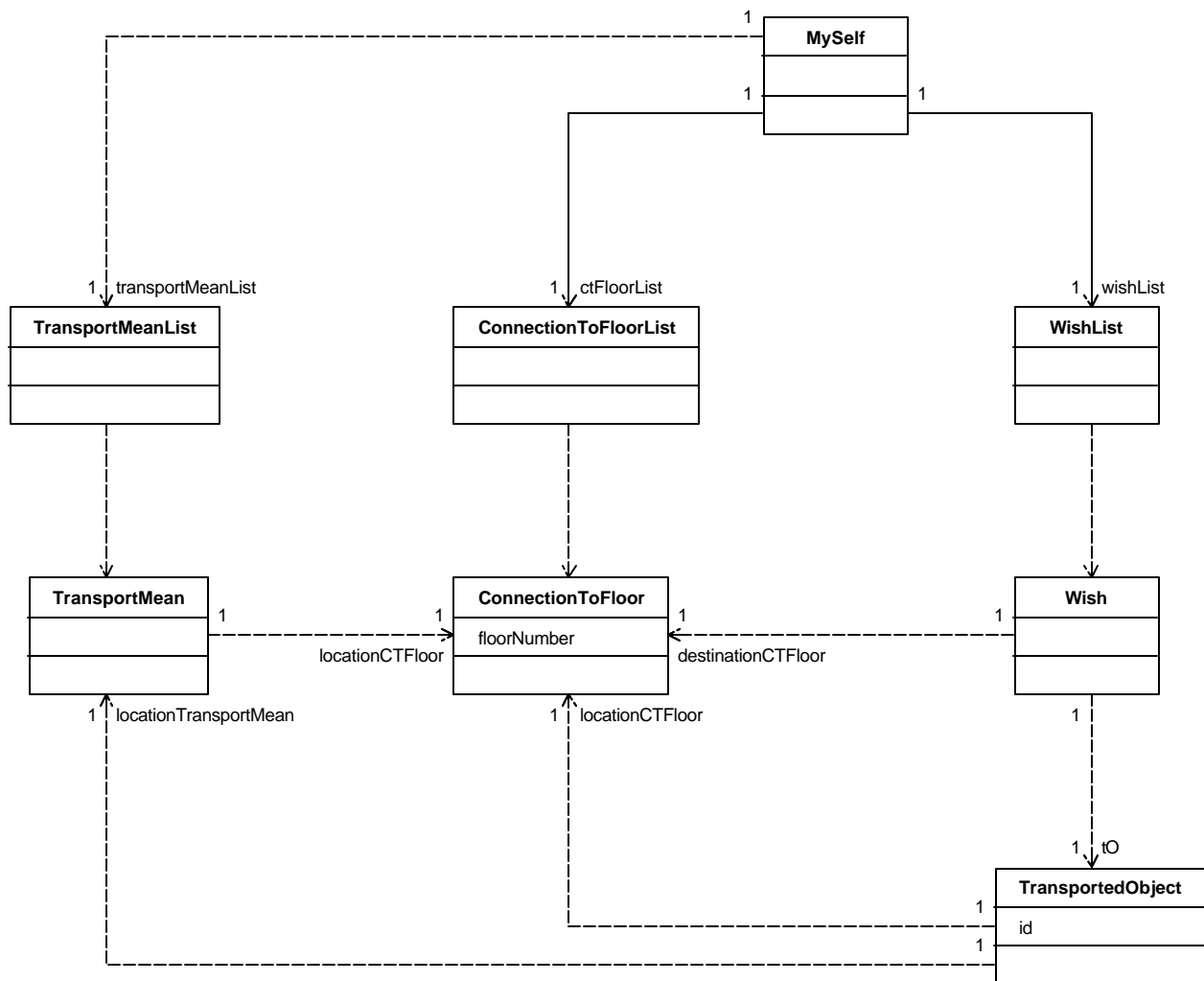


Figure 13. 22\_S\_Concept (Class Diagram)

## **Glossary**

### **ConnectionToFloor**

The connection of the ETS to one FloorObject (for instance a door).

### **ConnectionToFloorList**

Concept referencing all the ConnectionToFloor concepts.

### **MySelf**

Concept used to represent what the ETS knows about itself and about its environment.  
It is created when the ETS is started up.  
It is destroyed when the ETS is shut down.

### **TransportedObject**

Concept representing one TransportedObjectActor.  
(Look at the diagram 14:S\_BusinessModelizationExplanation for more details about what "representing" means.)

### **TransportMean**

The mean by which the ETS transports TransportedObjects (ETS has 1 or more TransportMeans). A TransportMean is simply a "thing" that can carry at least one TransportedObject and that is able to move from floor to floor.

### **TransportMeanList**

Concept referencing all TransportMean concepts.

## **Wish**

Concept representing one WishObject.  
(Look at the diagram 14:S\_BusinessModelizationExplanation for more details about what "representing" means.)

## **WishList**

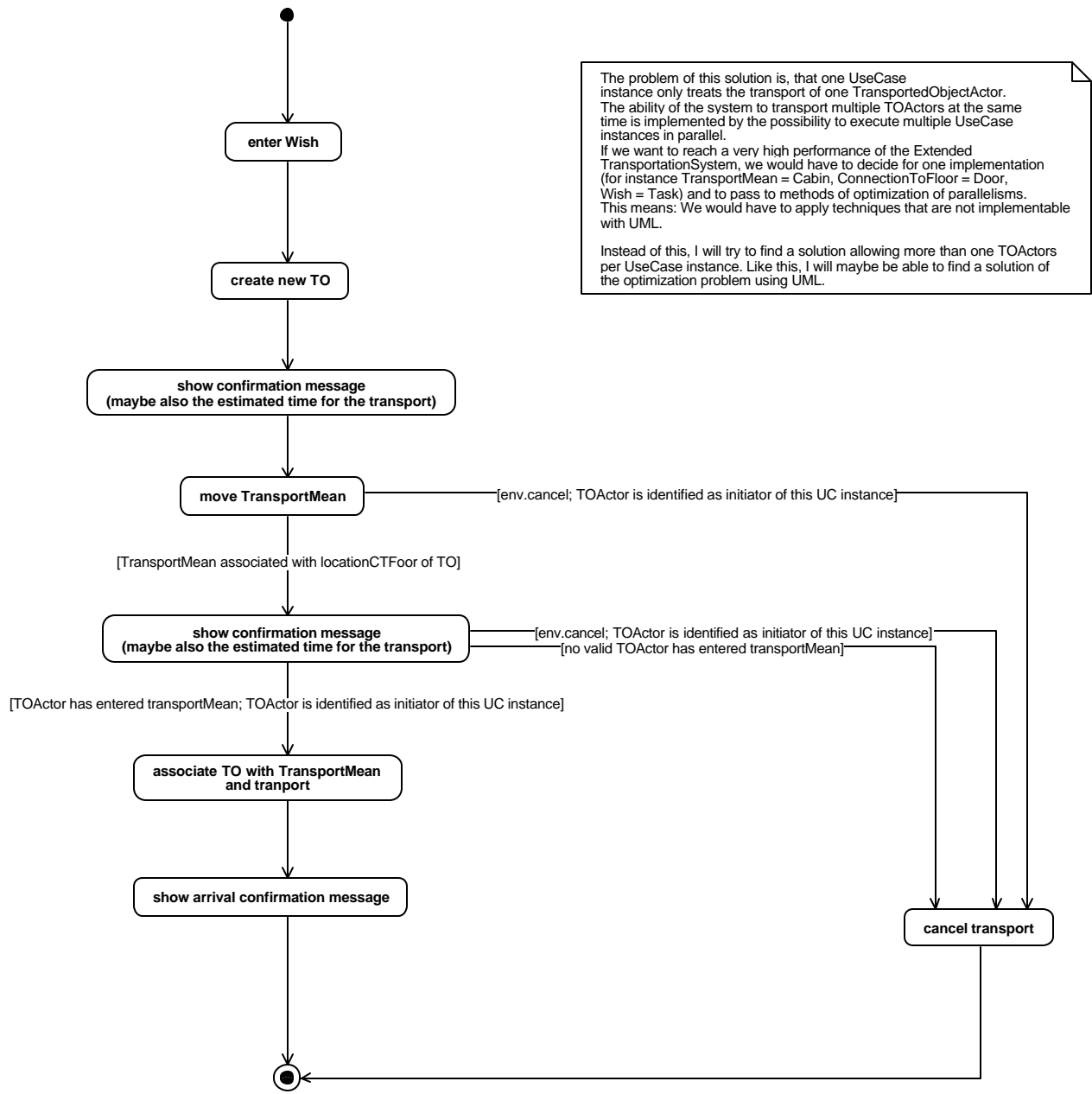
Concept referencing all the Wish concepts.

### 3.2.4 Partial System Activity

This diagram explains itself. Just note the possibility for the TO to cancel a transport as long as it is not yet associated with a TransportMean.

Please read the text in the comment box, that explains why this model is not what I expected.

### 30:S\_Activity<transport> (Activity Diagram)



The problem of this solution is, that one UseCase instance only treats the transport of one TransportedObjectActor. The ability of the system to transport multiple TOActors at the same time is implemented by the possibility to execute multiple UseCase instances in parallel. If we want to reach a very high performance of the Extended TransportationSystem, we would have to decide for one implementation (for instance TransportMean = Cabin, ConnectionToFloor = Door, Wish = Task) and to pass to methods of optimization of parallelisms. This means: We would have to apply techniques that are not implementable with UML.

Instead of this, I will try to find a solution allowing more than one TOActors per UseCase instance. Like this, I will maybe be able to find a solution of the optimization problem using UML.

Figure 14. 30:S\_Activity<transport> (Activity Diagram)

### **3.3 Conclusion**

#### **What has been reached:**

- In this second approach, I made my first experiences with a completely new and rather confusing idea: to model a real object like an elevator using UML. Even if it is really difficult, it works.
- I have understood now the difference between real objects and their representations in concepts. I have to admit that, if one designs software, this difference seems to be clear, but in my case, it was not. It was now, working on this new model, that I understood the meaning of a trace of an object into the viewpoint of some other actor.

I found some interesting ideas about how to make a better transportation system. These are:

- The TOActor should be identified and represented in the system.
- The TOActor should specify the destinationFloor at the moment it requests a transport.
- The TOActor should be able to cancel a transport.
- The Wishes should be managed centrally.

#### **But...**

As I have explained in the comment box of the last diagram, this model can not be optimized using UML. And using other techniques is not what I wanted. I wanted to find out if there is a possibility to do this optimization with UML. And, there is a possibility. Please read on.

## 4. ETS VERSION TWO

The idea of this new approach is to allow multiple TOActors entering wishes for one Use Case instance. As we always treat one Use Case a time in UML, we did not have the possibility to model the parallelism of different transports before. With this new approach, this will be possible.

### 4.1 Business Implementation Spec

#### 4.1.1 Snapshots

In each Snapshot, we have now two TransportedObjectActors, that have different desires and that have to be satisfied somehow.

#### 10:BI\_Snapshot<initial> (Collaboration Diagram)

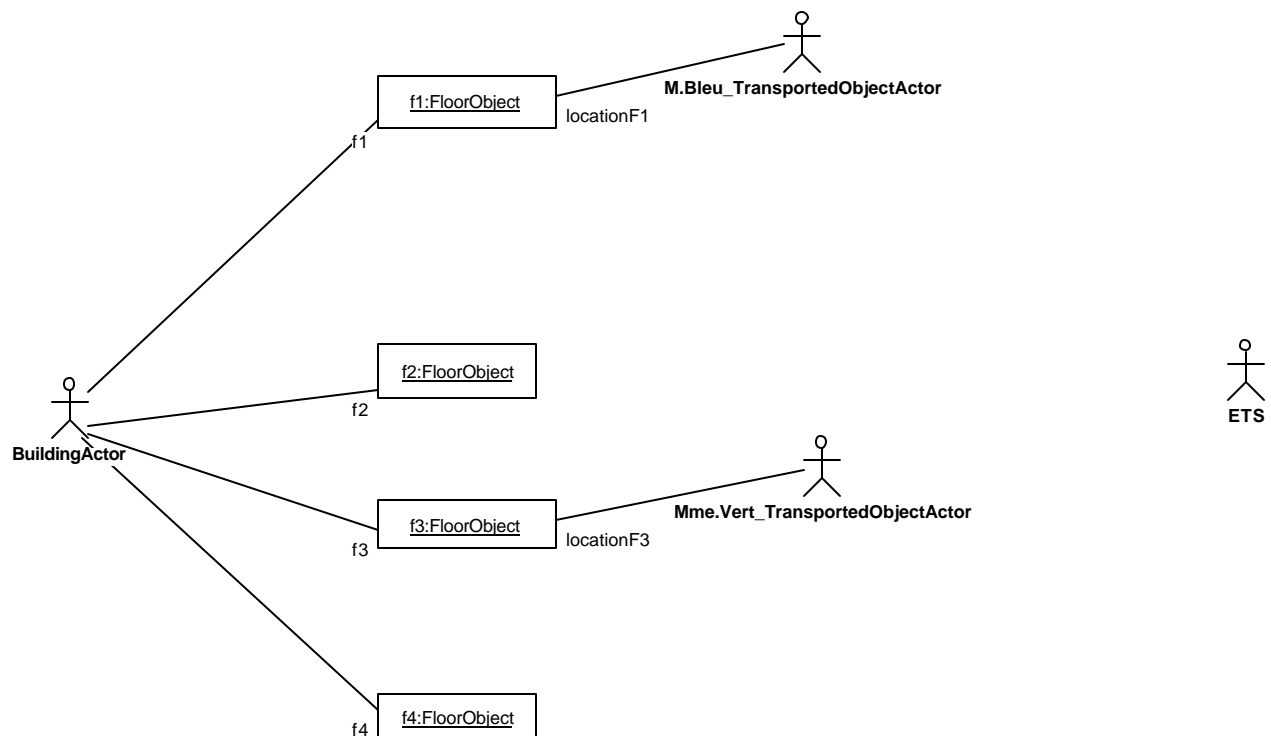


Figure 1. 10:BI\_Snapshot<initial> (Collaboration Diagram)

### 11:BI\_Snapshot<afterWishesEntered> (Collaboration Diagram)

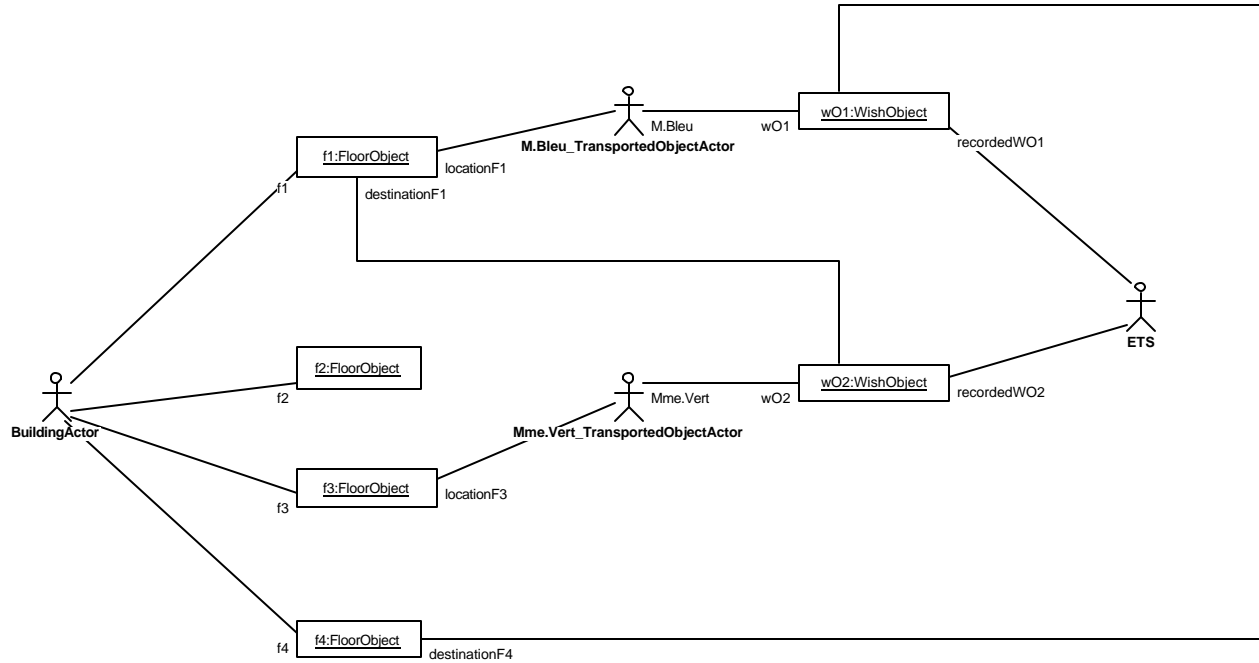


Figure 2. 11:BI\_Snapshot<afterWishesEntered> (Collaboration Diagram)

## 12:BI\_Snapshot<final> (Collaboration Diagram)

This is the final Snapshot. Until now everything works exactly in the same manner as in the previous approach, simply for two TOActors instead of one.

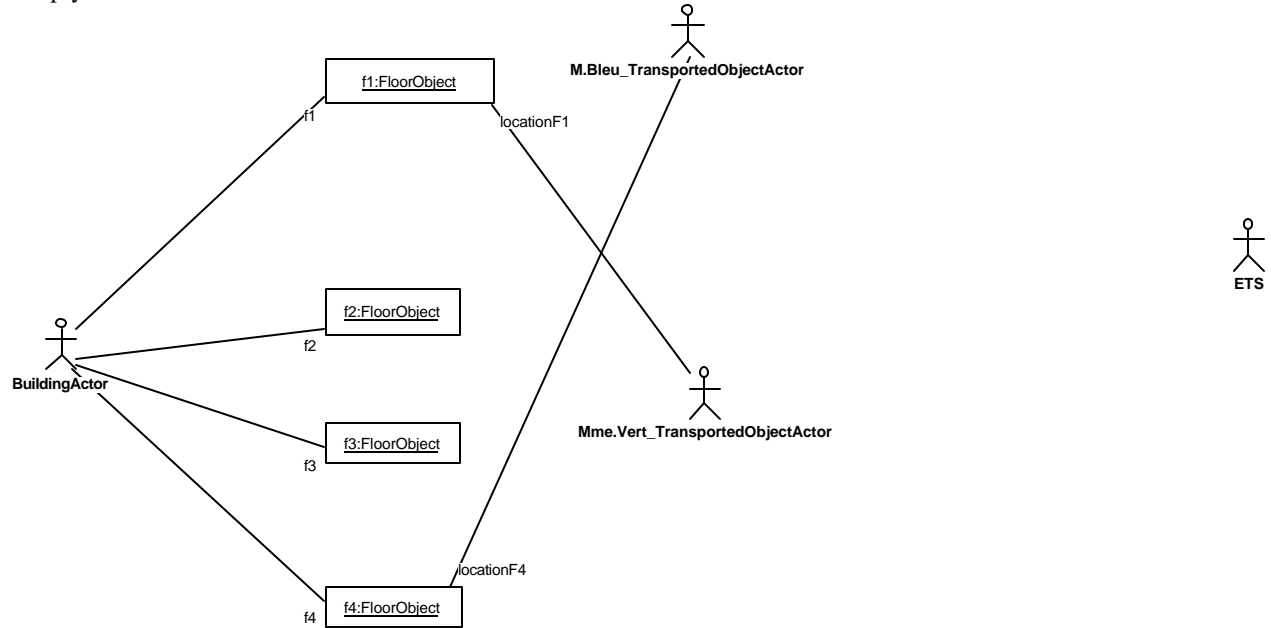


Figure 3. 12:BI\_Snapshot<final> (Collaboration Diagram)

## 4.1.2 Collaboration Model

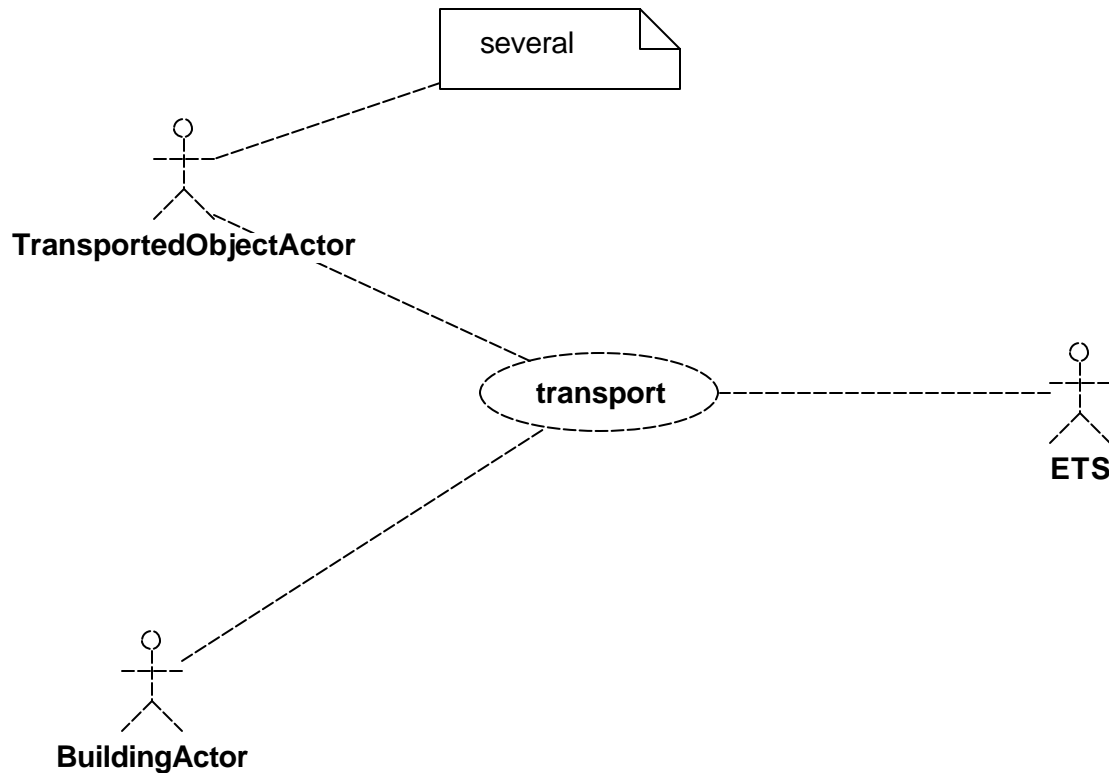


Figure 4. 20:BI\_Collaboration (Collaboration Diagram)

### Collaboration Description

#### Collaboration: transport

##### Purpose

Satisfy the Wishes of several TransportedObjects (at least one Wish per TO). A Wish is: get from one floor to another floor.

##### Policies

UC\_P1: A Wish can be expressed by any actor interacting with the Extended Transportation System. For reason of simplicity, this actor is not represented in the

diagram: we assume that each TransportedObject enters its Wish. At this level of abstraction, this is acceptable; it does not matter who expressed the Wish.

UC\_P2: The ExtendedTransportationSystem should provide the necessary hardware and software to fulfill all Wishes of all TransportedObjects.

UC\_P3: The ratio  $(\text{arrival\_time} - \text{request\_time}) / \text{abs}(\text{destination\_floor} - \text{departure\_floor})$  should be as small as possible for each Wish!

UC\_P4: Only one UC instance may occur at a time. UC starts when system is idle and one TOActor enters a Wish, and it ends when no more Wishes are pending.

UC\_P5: During one lifecycle of a transport UC, new Wishes can be entered at any time.

### **Parameters**

one or more TransportedObjects, one or more locationFloors, one or more destinationFloors

### **Pre-Conditions**

A Building with a certain number of Floors exists.

At least one TransportedObject exists.

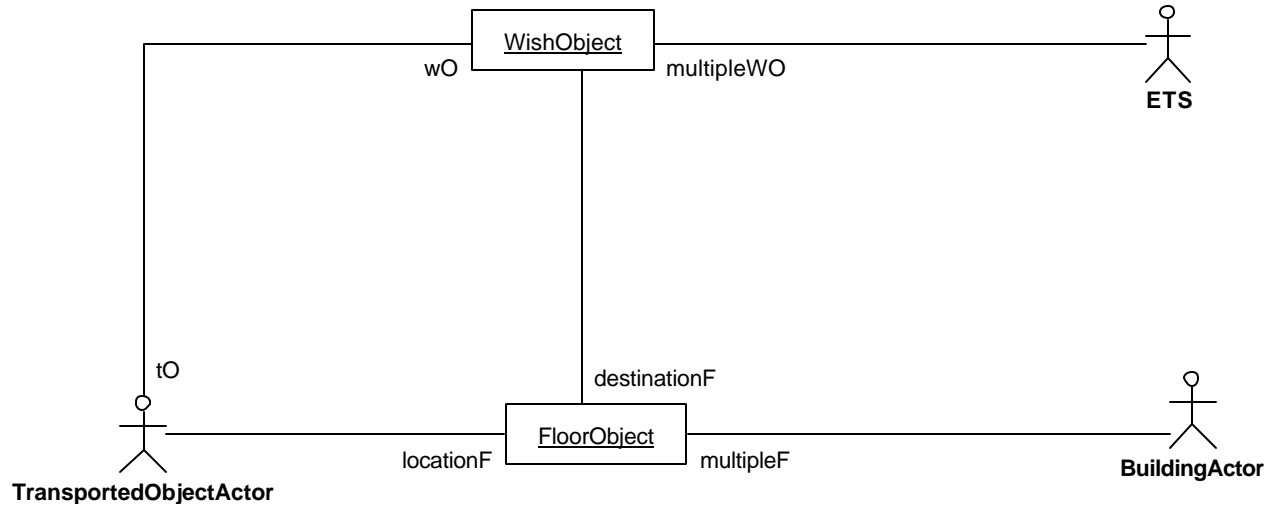
### **Post-Conditions**

All Wishes that were entered are satisfied, i.e. every TransportedObjectActor is now associated with the destinationFloor of its last Wish.

All Wishes have disappeared.

### 4.1.3 Objects Model

The Object Diagram has not changed at all. The framework of the objects outside the system remains the same.



**Figure 5. 21:BI\_Object (Collaboration Diagram)**

## Glossary

### BuildingActor

Model representing the building. A BuildingActor can have several FloorObjects. The only action of this actor is to "tell" a TransportedObject on which floor it is.

### ETS

Model of the Extended Transportation System, which represents any imaginable solution to transport TransportedObjects from one Floor to another.

### TransportedObjectActor

Model of an object that "wants" to be transported. As we assume that it has expressed the Wish itself, it is an actor.

**FloorObject**

Model representing one floor of the building, which can be served by the ETS.

**WishObject**

Model of the wish a TransportedObject has communicated to the ETS. A Wish can only be: to be transported from one locationFloor to a specified Floor.

## 4.2 System Spec

### 4.2.1 Snapshots

The System Snapshots are exactly the same as in the last approach, too. But: I cut away some Snapshots. This is because we do not know how the system transports the two TOActors, which one is served first, which one is associated to which TransportMean, how many TransportMeans are available, etc.

So I only made one Snapshot before the wishes have been expressed, one after, and one after the satisfaction of both wishes.

### 10:SI\_Snapshot<initial> (Collaboration Diagram)

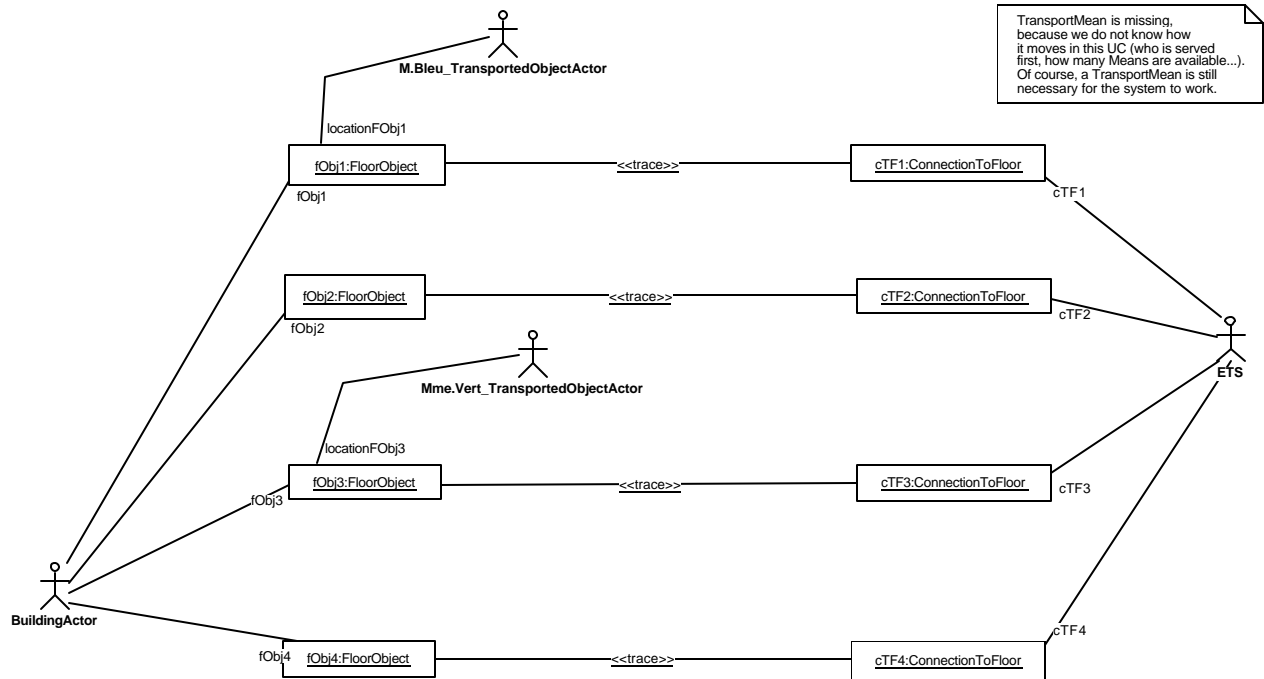


Figure 6. 10:SI\_Snapshot<initial> (Collaboration Diagram)

### 11:SI\_Snapshot<afterWishesEntered> (Collaboration Diagram)

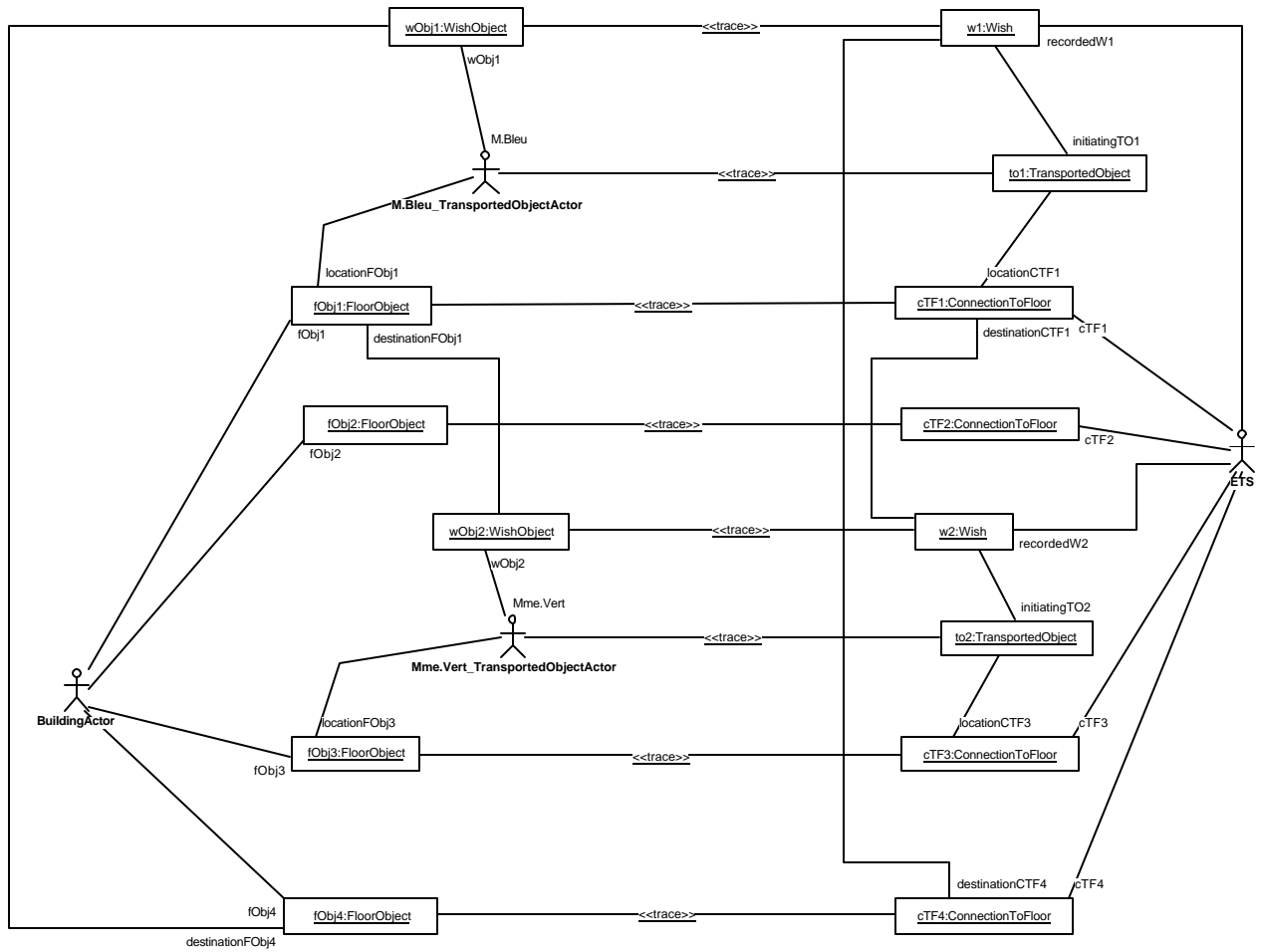


Figure 7. 11:SI\_Snapshot<afterWishesEntered> (Collaboration Diagram)

## 12:SI\_Snapshot<final> (Collaboration Diagram)

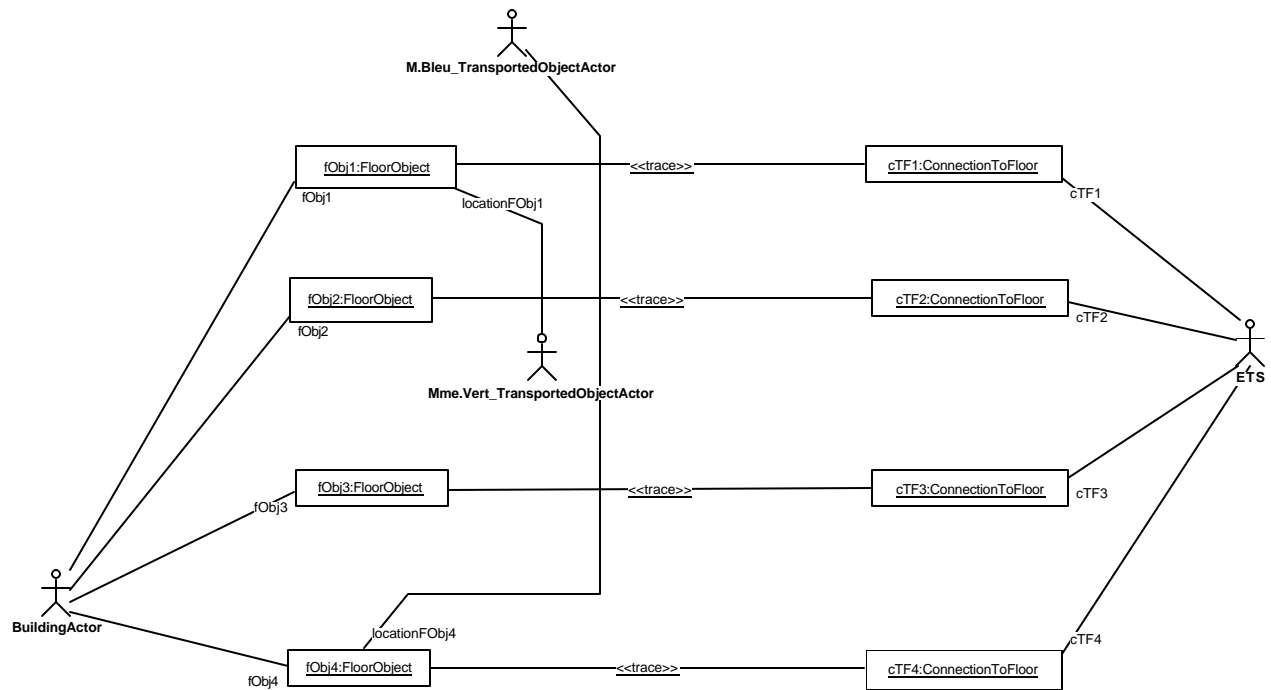


Figure 8. 12:SI\_Snapshot<final> (Collaboration Diagram)

## 4.2.2 Use Cases Model

In the new system policies and in the Use Case description, I set up the rules for the multiple-user-Use Case “transport”.

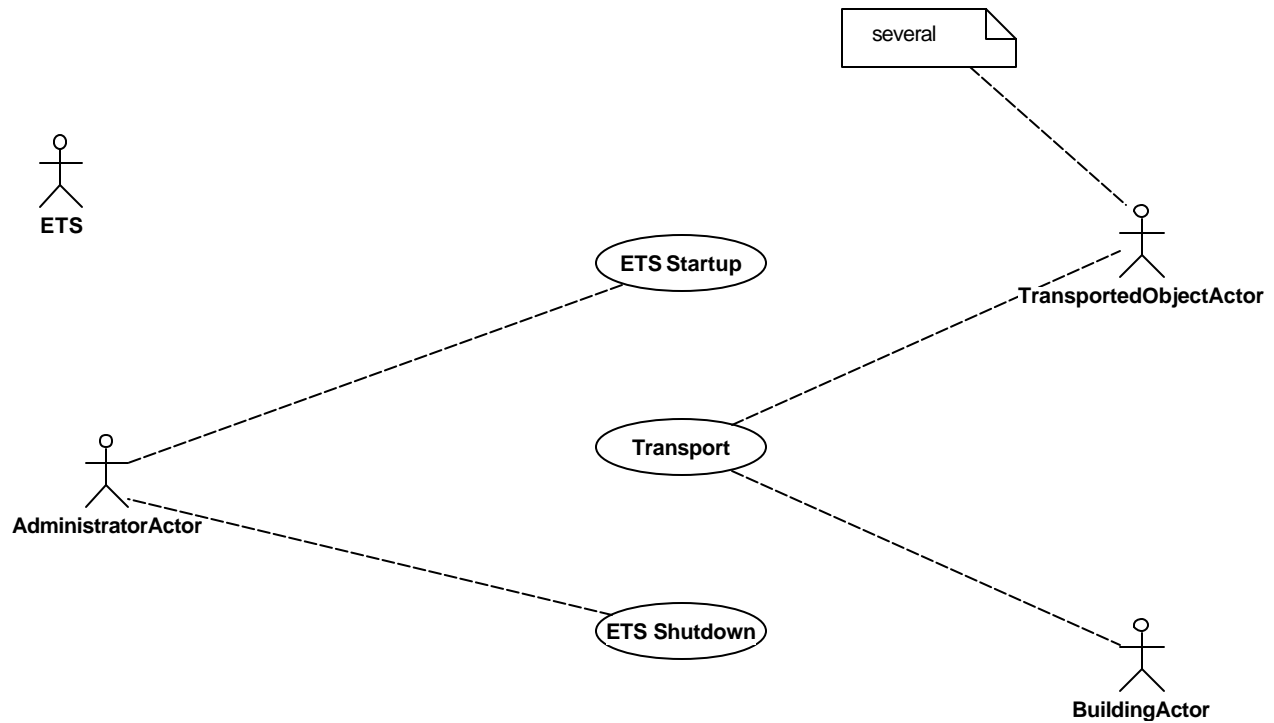


Figure 9. 20\_S\_UseCase (Use Case Diagram)

## System Policies and Patterns

S\_P1: TransportedObjectActors initialise the actions of the ETS.

S\_P2: The ETS has internal representations of the FloorObjects, of the WishObjects, AND of the TransportedObjectActors. Like this, it is able to identify TransportedObjectActors communicating with it, and to record the Tasks to fulfill and the location of each TransportedObjectActor.

S\_P3: The ETS makes sure that no environmental Actor can input illegal information. This aim is reached by limiting the input interfaces ( for instance several numbered buttons instead of a textfield).

S\_P4: The ETS tries to minimize the transport-time of each TOActor by adapting it's decisions as the number of Wishes evolves.

## Use Case Description

### Use Case: Transport

#### UC Purpose

The Wishes of one or several TransportedObjectActors are satisfied.

#### UC Policies

P1: Each time a TransportedObjectActor communicates with the ETS, it is identified by the ETS (associated with a new or an existing TransportedObject concept).

P2: The representation of a TransportedObjectActor in the ETS is limited to the time during which it is present in the System's environment. (After a Wish is fulfilled, the corresponding TransportedObject is deleted.)

P3: The TransportedObjectActor can cancel it's transport, as long as it is not yet being transported (i.e. not yet associated with a TransportMean).

P4: Once the TransportedObjectActor is associated with a location TransportMean, it can not change it's Wish anymore. The ETS is now responsible to get it to the right destinationFloor.

P5: Each TransportedObject (i. e. representation of a TransportedObjectActor) can only have one Wish and exists only as long as this Wish exists.

#### UC Parameters

IN: one or more IdentificationParameters of a TransportedObjectActor

IN: one or more destinationFloorNumbers

IN: zero or more CancelParameters

#### UC Pre-Conditions

At least two Floors exist.

ETS is running (the last Action performed by the AdministratorActor was TS Start Up).

#### UC Post-Conditions

No Wish exists.

Every TransportedObject has been associated with the destinationCTF of it's Wish and has been deleted.

### 21:S\_Activity (Activity Diagram)

This System Activity Diagram is completely correct. Only one instance of the Use Case “transport” is running at a time. It ends, if all wishes that have ever been entered are satisfied, and now, at the next entered wish, a new Use Case instance starts.

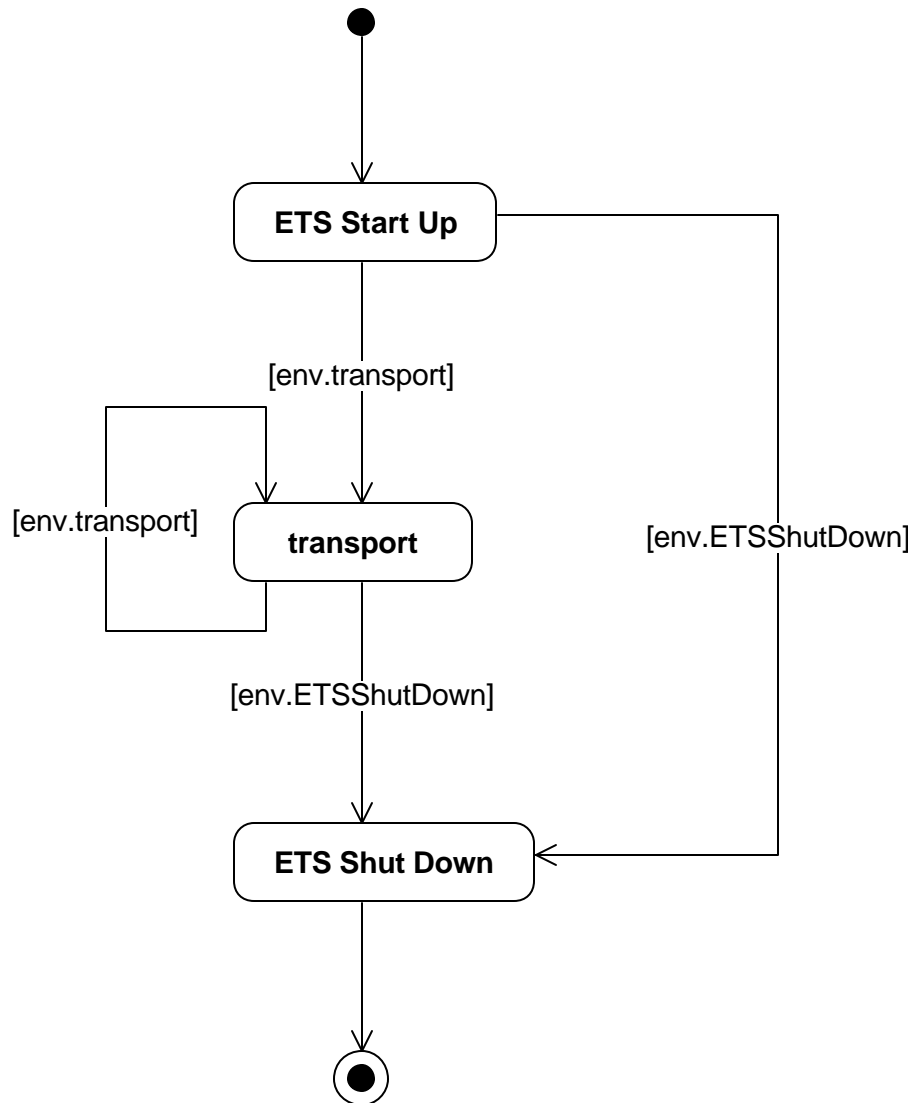


Figure 10. 21:S\_Activity (Activity Diagram)

### 4.2.3 Conceptual Model

The Concept Diagram is exactly the same as in the ETS, version1. The framework of the whole system did not change. But what has changed, is the way in which actions are described. That is why the hard part begins with the next step.

### 22\_S\_Concept (Class Diagram)

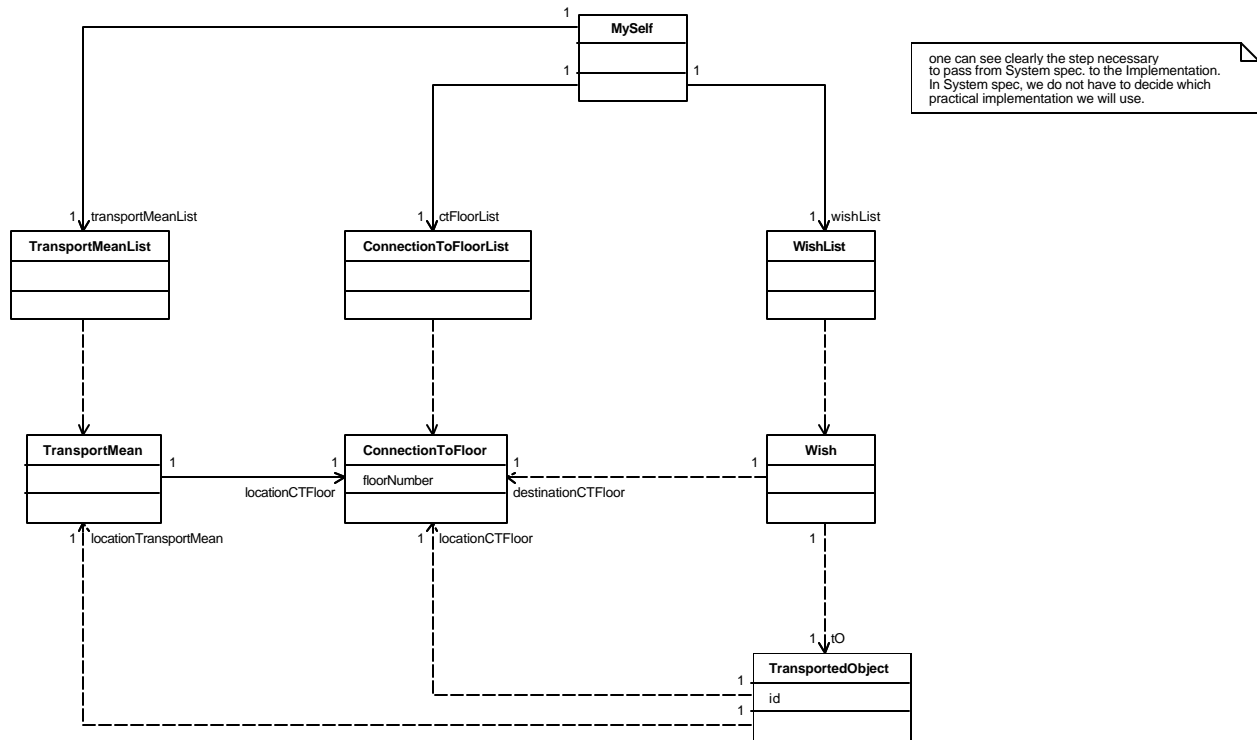


Figure 11. 22\_S\_Concept (Class Diagram)

## Glossary

### ConnectionToFloor

The connection of the ETS to one FloorObject (for instance a door).

### **ConnectionToFloorList**

Concept referencing all the ConnectionToFloor concepts.

### **MySelf**

Concept used to represent what the ETS knows about itself and about it's environment.  
It is created when the ETS is started up.  
It is destroyed when the ETS is shut down.

### **TransportedObject**

Concept representing the information about one TransportedObjectActor in the ETS' viewpoint.

### **TransportMean**

The mean by which the ETS transports TransportedObjects (ETS has 1 or more TransportMeans). A TransportMean is simply a "thing" that can carry at least one TransportedObject and that is able to move from floor to floor.

### **TransportMeanList**

Concept referencing all TransportMean concepts.

### **Wish**

Concept representing the information about one WishObject in the ETS' viewpoint.

### **WishList**

Concept referencing all the Wish concepts.

## 4.2.4 Partial System Activity

Now, we have to describe the possible order of actions during the lifecycle of one “transport” Use Case instance. So, how do we show the option of several actions occurring in parallel? The problem is, that we do not know when a new transport will be requested. It can happen at any time during the runtime of the other transports.

### 30:S\_Activity<transportExperimental> (Activity Diagram)

I first made a simplified model with only three actions (enter Wish => transport TO => delete Wish). As you can see in this diagram, the used technique leads to a monstrous web of wildly connected actions. I have even left away the conditions on the graphs, and it is, for only two transports at one time, a complicated web of actions. There is no hope in finding a solution like this.

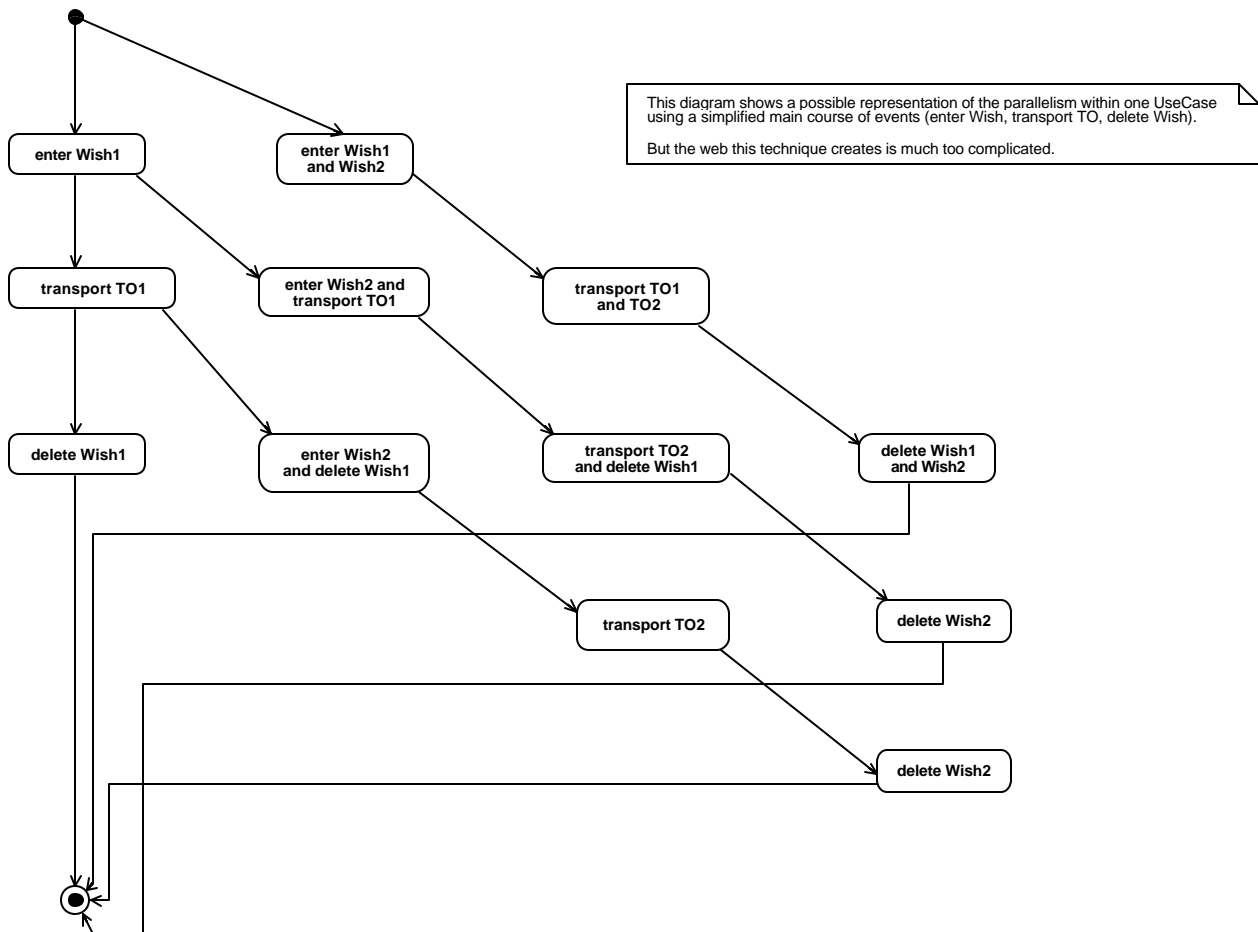


Figure 12. 30:S\_Activity<transportExperimental> (Activity Diagram)

### 31:S\_Activity<transportExperimentalTwo> (Activity Diagram)

But fortunately, I have found a really nice solution for this problem. Read the comment box for further details.

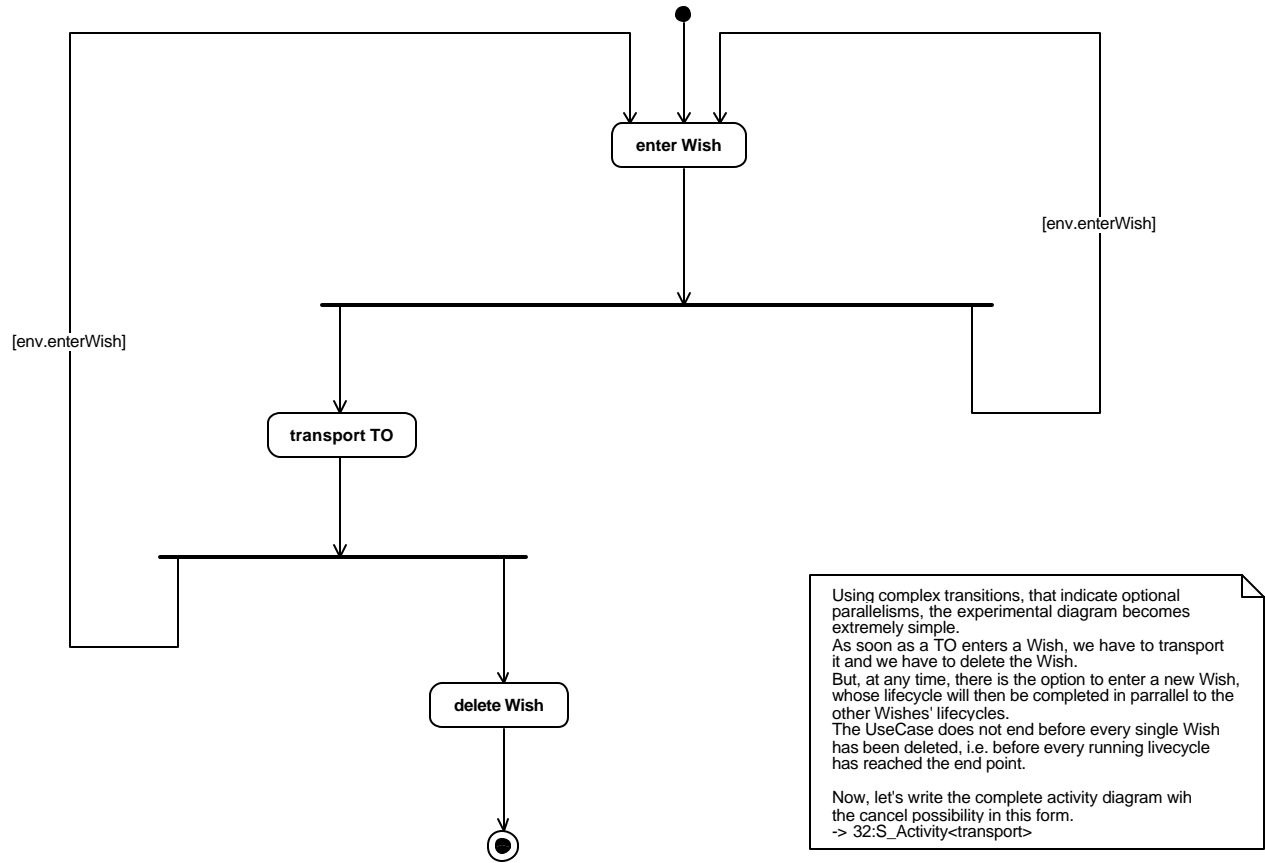


Figure 13. 31:S\_Activity<transportExperimentalTwo> (Activity Diagram)

### 32:S\_Activity<transport> (Activity Diagram)

And here is the complete transport Activity Diagram in this new form allowing parallelisms using complex transitions.

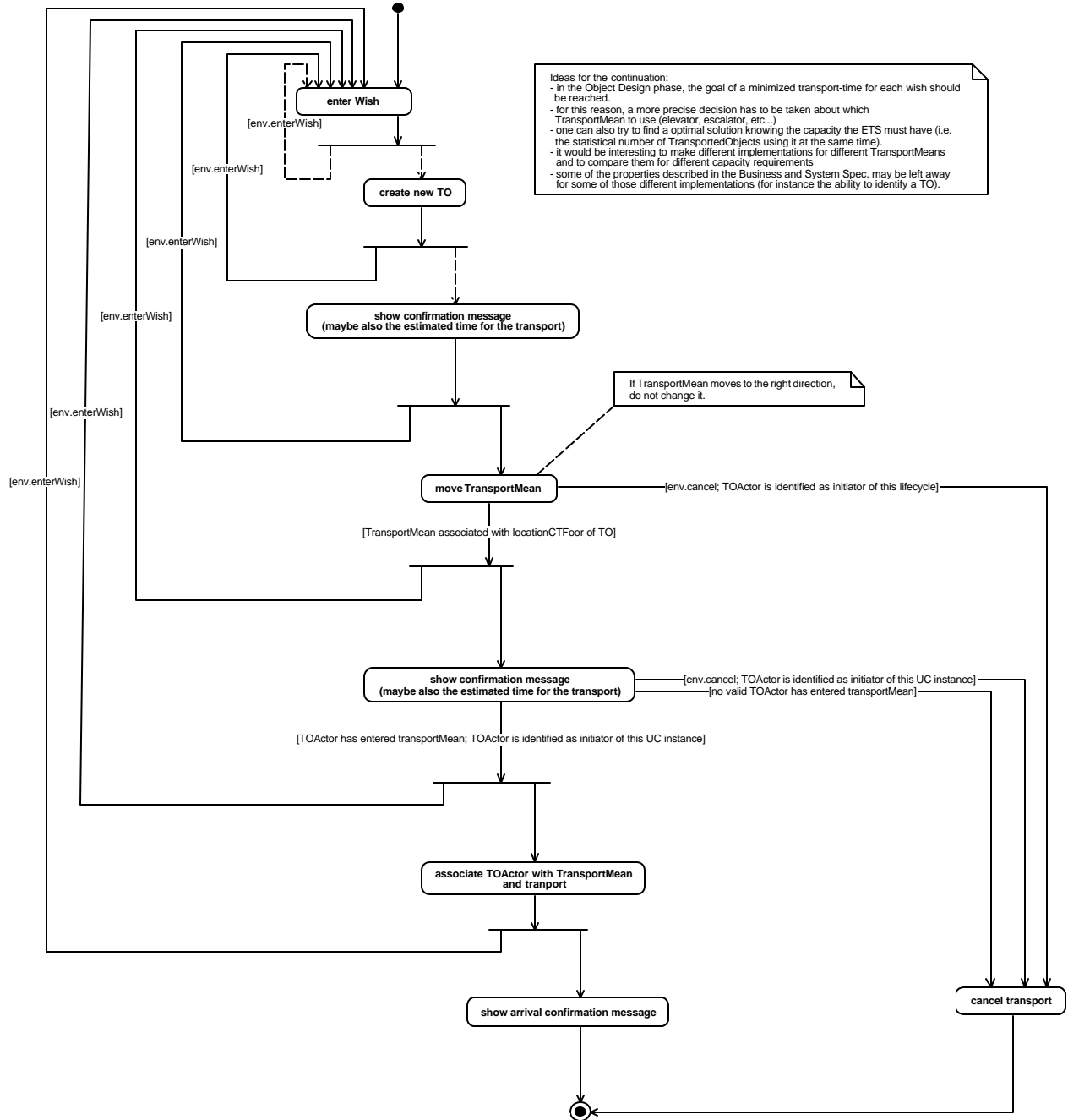


Figure 14. 32:S\_Activity<transport> (Activity Diagram)

## 4.3 Conclusion

### What has been reached:

- I made an important first step to a very interesting model. Everything is ready to continue this work and I am convinced that this time, we are on the right way
- The recursive parallelism-representation is a nice idea. What advantages it really brings is not yet known. This would be discovered by a continuation of the project.
- I think what I have done so far can be considered as a proof that UML can be used to optimize the handling of actions occurring in parallel.

### How to continue:

As you can read in the comment-box of the last diagram, the plan is to try out models of different hardware implementations.

For the example of an escalator, you would probably find the following properties:

- A TransportedObjectActor requests a transport by stepping onto the escalator. This means the time of the request equals the time of it's association to the TransportMean.
- A Wish can only have a destinationFloor that is the locationFloor + or - 1.
- The TransportedObjectActor is identified by it's weight, and by the moment this weight is detected by the motor of the escalator. (This does not mean the TOActor is identified by the software of the escalator, but that the motor is aware of the presence of the TO.)

### Now...

Unfortunately, I had to stop this project, because the semester is finished.

Now, it's your turn, dear reader. Are you interested in how this development could go on?

Have you some good ideas how to work on with this project?

Or would you start it over from the very beginning and do everything in a better way?

Well, just do it.

Dive into the world of UML and Object Oriented Analysis and Design and find out how interesting and useful all this can be.

## **5. APPENDIX**

### **5.1 Java Code**

You will find the Java files in the project directory.