AALBORG UNIVERSITY
INSTITUTE OF ELECTRONIC SYSTEMS
DEPARTMENT OF COMMUNICATION TECHNOLOGY

Frederik Bajersvej 7 ▪ DK-9220 AALBORG Ø Telefon 98 15 85 22

TITLE:	Elevator Control System
THEME:	Software Development / Simulation
PROJECT PERIOD:	3rd semester, September to December 1998
PROJECT GROUP:	355

Abstract

This report is the documentation of the object oriented development of an elevator control system test program. The program performs a statistical evaluation of the connected control system through simulation of the passenger flow and elevator mechanics in a building. The user can set up the passenger flow and elevator characteristics to match actual conditions.
To test the functionality of the program we have developed our own centralized control system in JAVA.

PARTICIPANTS:
Claus Albøge
Mads Græsbøll Christensen
Tonny Gregersen
Karsten Jensen
Peter Korsgaard
Lars Jochumsen Kristensen
Robert Stepien

SUPERVISOR:
Giedrius Slivinskas

PUBLICATIONS:	10
NUMBER OF PAGES:	143 pages
FINISHED:	December 22nd 1998

Preface

This report has been drawn up by project group D3-355 at Institute of Electronic Systems, Aalborg University. The purpose with the project is to achieve and demonstrate skills in object oriented analysis, design and programming. The report mainly apply to our supervisor and censor and other students at Aalborg University.

The report contains five main parts and an appendix. Four parts constitute the documentation for analysis, design, implementation and test of our program. The fifth part is our study journal were we reflect on our project course and the methods applied to it. The appendix contain the account for scientific theories used in the project, a description of parts of the progam we have opted not to treat thoroughly in our analysis and design and source code for parts of the program.
Figures and tabulars are numbered in succession according to chapters. To benefit fully from reading this report a basic knowledge to the method OOA&D, UML-notation and the JAVA programming language is necessary. A java-doc documentation for all classes can be found on the following url: http://www.kom.auc.dk/ jacmet/javadoc.

Aalborg, 21. december 1998

Claus Albøge

Lars Jochumsen Kristensen

Tonny Gregersen

Mads Græsbøll Christensen

Karsten Jensen

Peter Korsgaard

Robert Stepien

Contents

Part i

Analysis

Chapter 1

Preliminary Analysis

1.1 Purpose

The computer system is intended for use in an elevator control system company. The purpose is to help the user configuring, adapting and testing elevator control systems in multiple floor/elevator buildings, according to time efficiency, economy, etc. The user should be able to give environment-specific variables characteristic to the actual building, thus he should not be able to edit the main algorithm or create a new one, but the system must enable him to alter the parameters used in the algorithm. To test and measure the results of the configuration without actually installing the system in a building, a simulation of a given passenger flow will be included. It should be possible to change the flow of people in the building in order to simulate rush hours, etc.

1.2 System Definition

Conditions: A computer system for users with various qualifications.

Application domain: Adaption of an elevator control system to specific environments by a technician.

Technology: Java-console based on JDK 1.1.6 (PC-platform).

Object system: Elevators, passengers, floors.

Functionality: Adjustment and test of the elevator control system with a finite number of parameters.

Philosophy: Modification and integration.

This leads to the following system definition:

> The system is a tool for users with various qualifications to adapt an elevator control system to specific environments. The tool includes a simulation of both the elevator system and the passenger circuit in it. The user will be able to adjust a finite number of parameters according to a number of quality measurements. The computer system must support the user in modification and integration of the elevator control system. It will be based on a Java-console, and will run on a PC-platform

1.3 Surroundings

1.3.1 Problem Domain

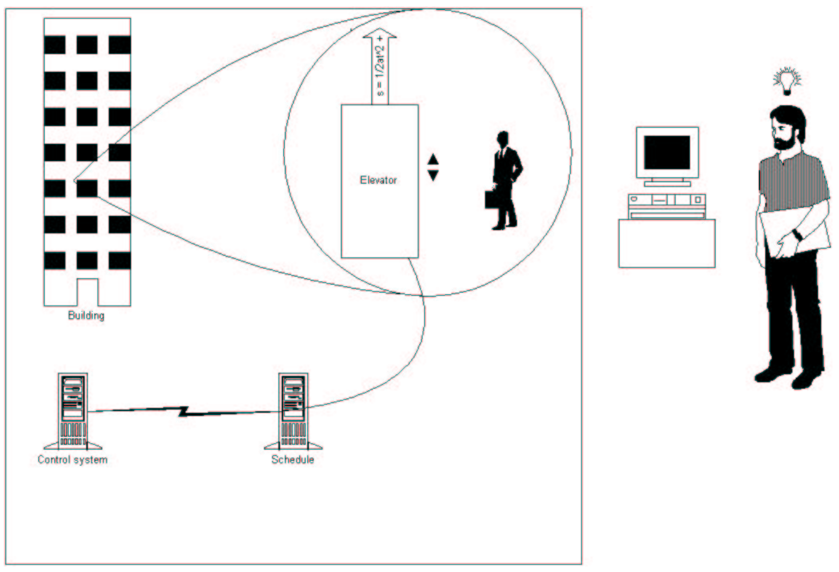


Figure 1.1: Overview showing the computer systems surroundings.

The computer system is a system to test an algorithm, with different parameters for an elevator control system. The algorithm simulate the elevator movement in a building with multiple floors and elevators.

To measure the quality of an algorithm you have to be able to register the travel time of each passenger in the passenger flow of the building. This makes it possible to make different statistic evaluations of the time used by the algorithm to move the passengers to their destination. Using the statistic evaluations it is possible to compare the algorithm with different parameters and thereby choosing the most efficient configuration of the elevator control system. Figure 1.1 shows an overview over the computer systems surroundings.

1.3.2 Application Domain

The computer system is a tool to find the most efficient configuration of an elevator control system. It can be used in two different kinds of environments:

- An already existing building with some predefined parameters for the elevator system.

- A non-existing building without predefined parameters for the elevator system.

It is possible to create a passenger flow like the one in the building, where the elevator control system is to be installed. This includes creating a passenger flow that simulates the peak flow of passengers between the floors and the time cycle of the passenger movement. When the passenger arrives at the destination the travel time is stored for later use in the statistic evaluation. Then the passenger continues in the passenger flow with a new destination and departure time. This results in a more realistic simulation of a passenger flow.
The elevator control system is based upon a general algorithm, which can be adjusted by the user of the computer system. The system specifies how the user can alter some parameters and thereby optimize the algorithm that is used in the elevator control system. The parameters in question will be those defining the relation between the waiting time for passengers on the floors and the travel time in the elevators. If the travel time for each elevator is reduced, the waiting time on the floors is increased and vice versa
It is possible to optimise the algorithm on the same passenger flow using the statistic evaluation to determine the parameters

1.4 Assumptions

In our work with the simulation we have made the following assumptions in order to simplify our problem domain models:

- People do not use stairs.
- The elevator system does not break down.
- The elevator motor is very simple.
- The motor supplies constant acceleration.
- The movement of the elevators is not affected by the load.

Chapter 2

Problem Domain

2.1 Structure

Figure 2.1 shows the coherence of the problem domain classes.

2.2 Classes

2.2.1 Building

The building (Figure 2.2) is an aggregation of the physical environment in which the elevator system exists and the passengers circulate.

2.2.2 Control System

The control system (Figure 2.3) handles all input regarding the elevator system and controls the destinations of the elevators.
To initialize the evaluation, one of the three following events have to appear:

- Elevator called: When a passenger invokes an elevator, the control system evaluates all schedules and selects, to which elevator schedule the destination should be added.

- Elevator ready: When an elevator is empty and ready to receive a new destination, it sends the elevator ready signal to the control system. Then if there is any destination

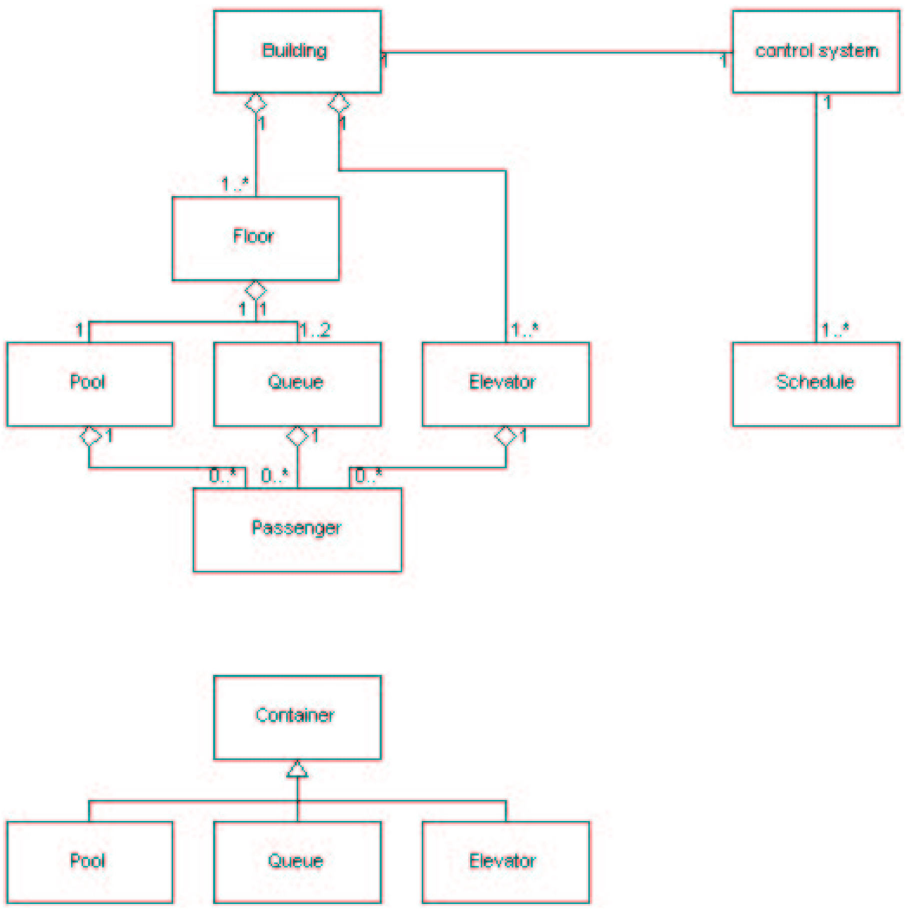


Figure 2.1: Structure diagram.

pending, the control system evaluates and adds a new destination to the elevator schedule.

- Destination selected: When a passenger selects a destination, from inside the elevator, the control system evaluates the local elevator schedule, and adds the destination according to the algorithm.

2.2.3 Schedule

Each elevator has a schedule (Figure 2.4) containing future destinations. Only the control system can change the schedule.

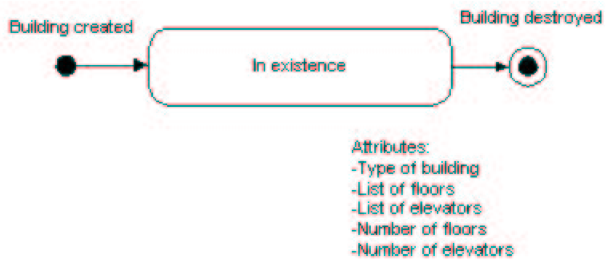


Figure 2.2: Building state diagram.

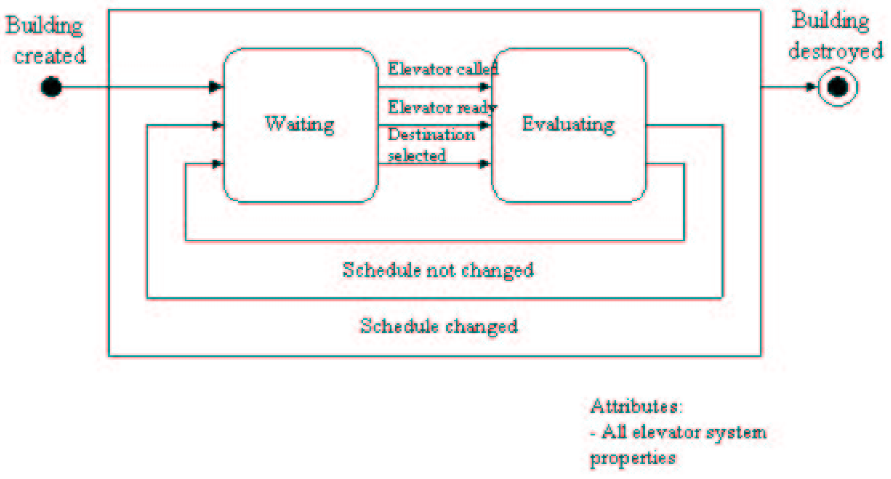


Figure 2.3: Control System state diagram.

2.2.4 Elevator

An elevator (Figure 2.5) is a container that moves passengers from one floor to another. It represents a real-life elevator but the mechanical properties of the elevator system (e.g. engine, wires) are also embodied in the elevator.

2.2.5 Floor

Each floor (see Figure 2.6) consists of a pool and two queues.

2.2.6 Container

The container class (Figure 2.7) embodies the general passenger circuit related properties of queue, pool and elevator such as methods for moving passengers and lists of contained

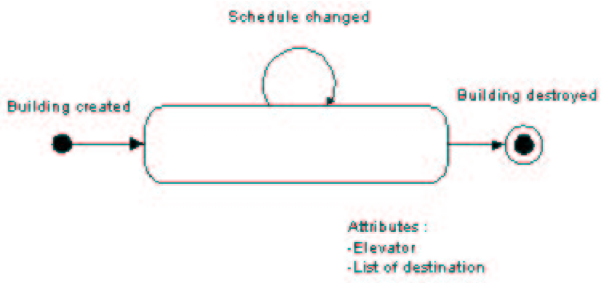


Figure 2.4: Schedule state diagram.

passengers.
A queue is a specialization of a container. Passengers waiting for an elevator wait in a queue on a floor. Each floor contains two queues, up and down.
The pools represent the whereabouts of the passengers, when they are not in the queue or the elevator, be it in an apartment, office, or store.

2.2.7 Passenger

The passenger (see Figure 2.8) is the entity that circulates in the building and is used for statistical evaluation (measurement) of the performance of the elevator system.

2.3 Events and Classes

Figure 2.9 shows the events and respective classes.

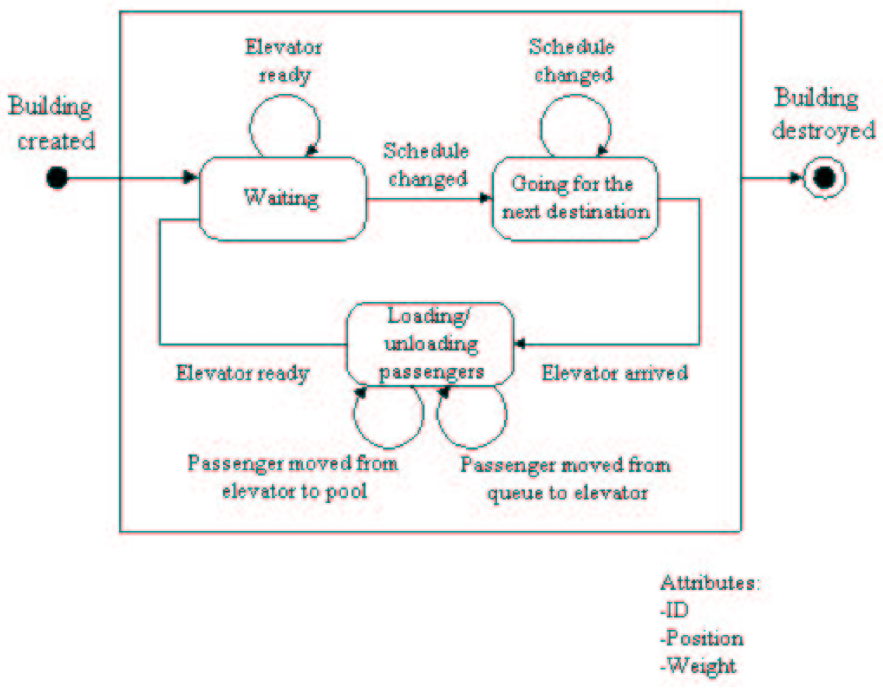


Figure 2.5: Elevator state diagram.

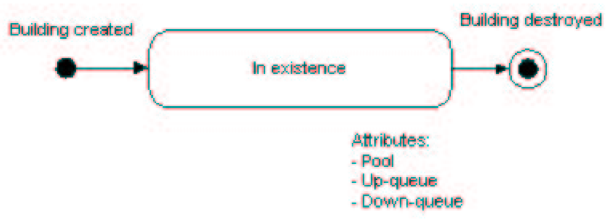


Figure 2.6: Floor state diagram

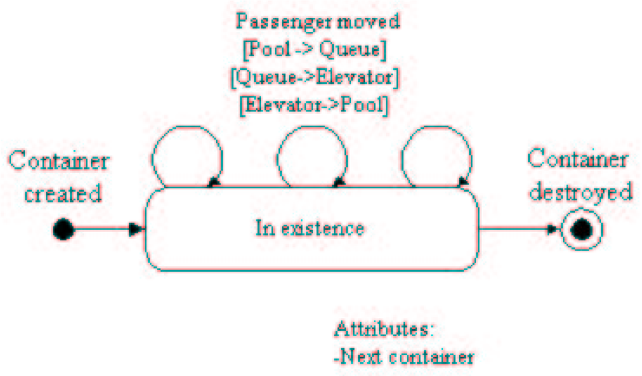


Figure 2.7: Container (Pool, Queue, Elevator) state diagram. See also the elevator state diagram.

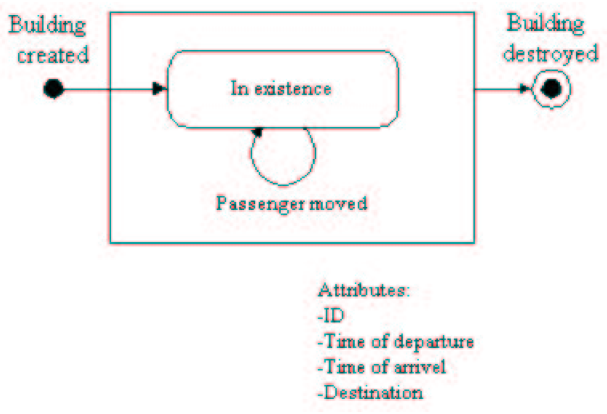


Figure 2.8: Passenger state diagram.

Events/Classes	Schedule	Elevator	Passenger	Queue	Pool	Control System
Elevator called				*		*
Elevator ready		*				*
Elevator arrived		*				
Destination selected		*				*
Schdule changed		*				*
Passenger moved from Elevator to Pool		*	*		*	
Passenger moved from Pool to Queue			*	*	*	
Passenger moved from Queue to Elevator		*	*	*		
Destination added	*					*
Destination deleted	*					*

Figure 2.9: Events and classes.

Chapter 3

Application Domain

3.1 Actors

There is only one user of the system. The following goes for him:

- **Purpose**
  A person who intends to adapt the given control system to a specific environment by setting up and simulating the passenger flow and adjusting the parameters in the control system algorithm.

- **Characteristics**
  The ability to obtain the mechanical properties of the elevator system (e.g. motor) and the building must be possessed, also the person must understand a statistical evaluation.

- **Example**
  A technician who is going to install the control system in a building. To make the elevator system run efficiently he must adapt the control system to the specific environment as defined by the user.

3.2 Patterns of usage

These patterns apply to the user:

- **Adaptation of Algorithm**
  Before the simulation is started, the physical properties of the building and the

Function	Complexity	Type
Parameter Adjustment	Medium	Update
Run Simulation	Hard	Calculation
Show Simulation Results	Medium	Read

Figure 3.1: Functions

elevator system is entered and the algorithm is adjusted to fulfill demands set by the user. The time efficiency is optimized through changes made in floor priorities and the travel time for the elevators.

- **Passenger Flow Control**
  The user defines the statistical behaviour of the passengers, thus making simulation of certain building types possible.

- **Statistical Evaluation**
  After the simulation has been run, a statistical evaluation is presented to the user as a measurement of the performance of the control system as set up by the user. This evaluation may e.g. include average travel time for each passenger and performance of each elevator.

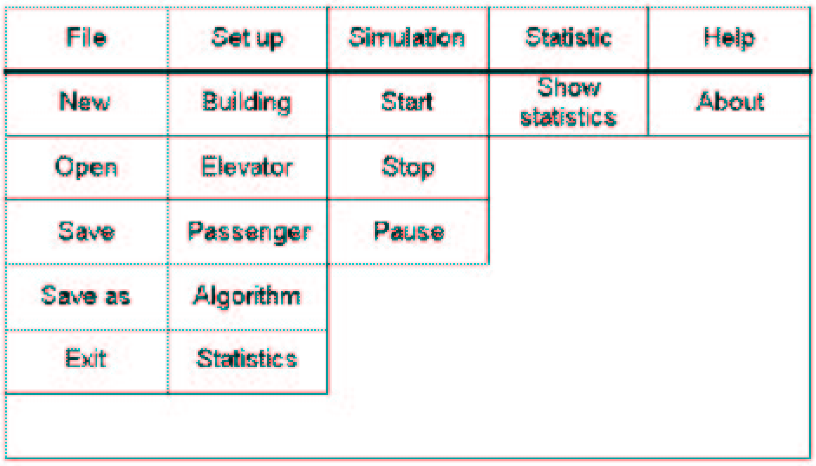
- **Iteration**
  After the statistical evaluation, the process can be iterated, until the most optimal configuration has been achieved.

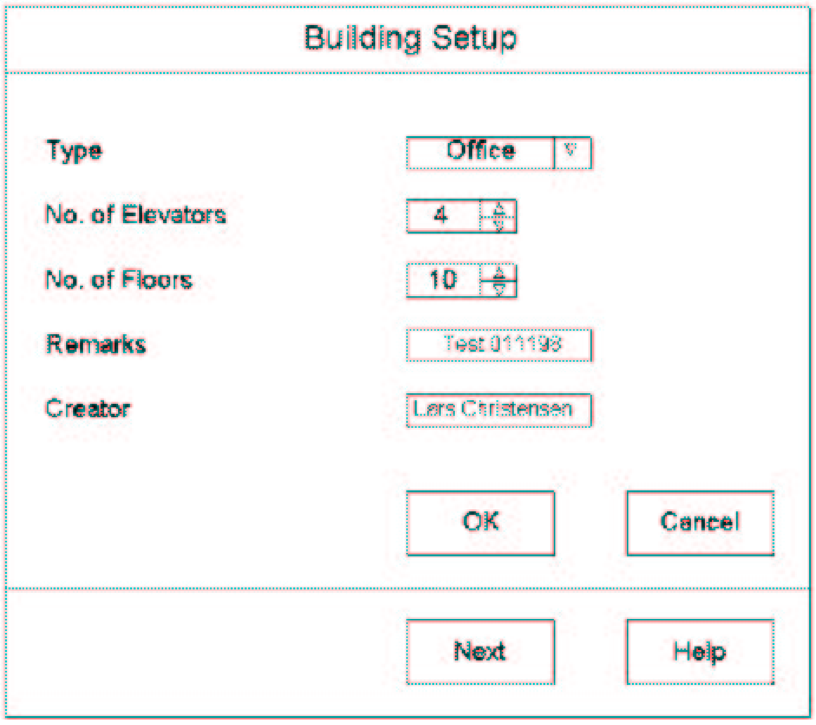
3.3 Functions

Figure 3.1 show the functions and their complexity and type.

3.4 User interface

The user should interact with the program in connection with parameter adjustment through a graphical user interface. In Figure 3.2 it is shown what this GUI may look like. Also the statistical evaluation of the simulation should be presented through GUI or optionally on a printer!




Building Setup

Type	Office
No. of Elevators	4
No. of Floors	10
Remarks	Test 011198
Creator	Lars Christensen

Figure 3.2: Graphical user interface.

Part ii

Design

Chapter 4

Preliminary Design

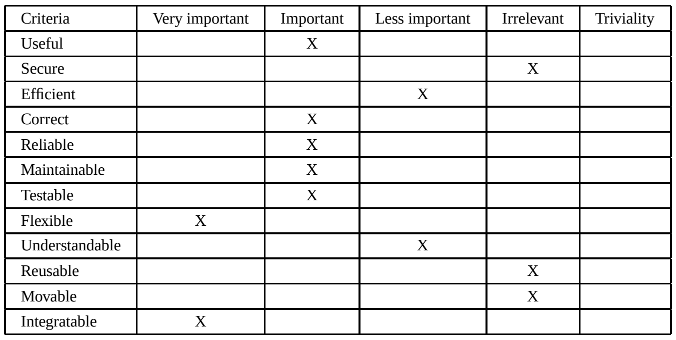
4.1 Analysis Corrections

The following amendments have been made to the analysis:

- The class Schedule has been removed from the structure diagram. It now exists as different attributes (lists of destinations) in several classes.
- Two classes, Control System Output and Passenger movement, was found when restructuring the state diagrams.
- It has been made more explicit that the control system and the simulation of the passenger flow are separate units. The simulation unit (SimElevator) and the control system unit are seen as actors using the other system. To handle this interaction two more classes, System Manager and Protocol, were added.
- The graphical user interface has been specified.

4.2 Criteria

Figure 4.1 shows the design criteria. The purpose of setting the design criteria is to get the priorities straight within the software development team.
To minimize the consequences of possible errors due to our inadequate knowledge of the physics of elevator systems the system must be designed flexible by extensive use of encapsulation.



Criteria	Very important	Important	Less important	Irrelevant	Triviality
Useful		X			
Secure				X	
Efficient			X		
Correct		X			
Reliable		X			
Maintainable		X			
Testable		X			
Flexible	X				
Understandable			X		
Reusable				X	
Movable				X	
Integratable	X				

Figure 4.1: Design criteria.

The importance of the system being integratable is based on the whole idea of the simulation - To test a control system algorithm, in principle an external system.

Chapter 5

Technical Platform

5.1 Equipment

The system is intended for use on a PC, but the platform will de facto be decided by JDK 1.1.6 compability and availability. It will be developed on PC-UNIX-terminals using the university facilities.

5.2 Basic Programs

Sun's JDK 1.1.6 will be used in the development. Also a basic text editor is required.

5.3 Systems and Devices

Besides a PC with standard equipment a JDK 1.1.6 compatible Java-console, a GUI-system, is needed, e.g. Windows 95/98/NT, X11, to run the program. Java Runtime Environment (JRE) is a Java virtual machine containing all you need to run the program. It can be downloaded at http://www.java.sun.com.

5.4 Design Language

The design language is generally based on the notation used in the OOA&D book by Peter Axel Nielsen et al.

Chapter 6

Architecture

6.1 Component architecture

Due to the fact that this is a program intended for test of a specific control system algorithm the control system is seen as an independent component with an interface to the simulated environment. Figure 6.1 show this dyarchic component architechture.
The design phase thus contains two parts. A part involving the simulation domain, and one concerning the control system. The simulation component has a system interface to the control system unit and a user interface. The control system unit has a system interface to the simulation component.
You must be able to extract the control system unit and use it in a real-life elevator system. The design of the system interfaces shall make this possible.

6.2 Processes

We have opted not to use multiple threads.

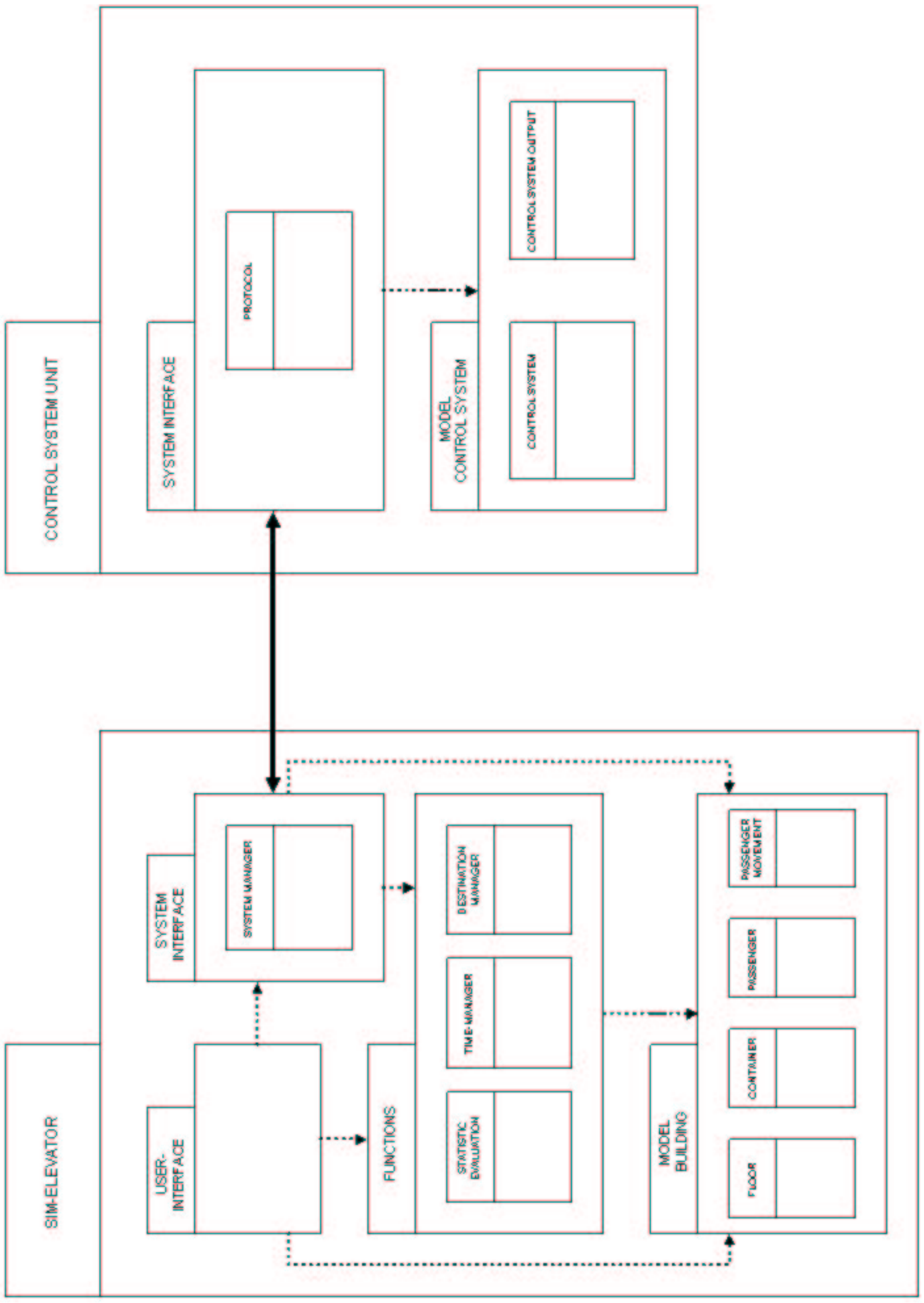


Figure 6.1: System architecture.

Chapter 7

Sim-elevator

7.1 Model Component

7.1.1 Structure

By substituting iterations in the state diagrams of the analysis document with classes (or attributes), we get a new structure diagram, Figure 7.4. The process of substituting the iterations lead to restructured state diagrams.

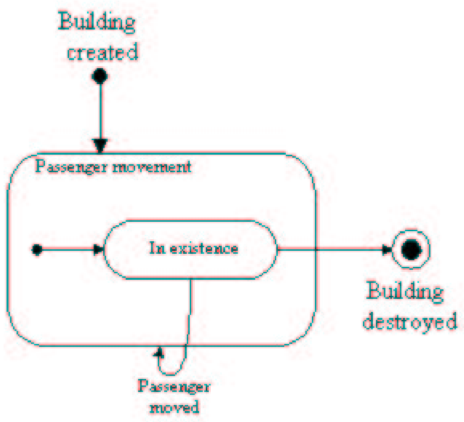


Figure 7.1: Restructured container state diagram.

7.1.2 Passenger Movement

Figures 7.1 and 7.2, showing the restructured state diagrams of the container and passenger classes, lead to the conception af a new class containing information on each move-

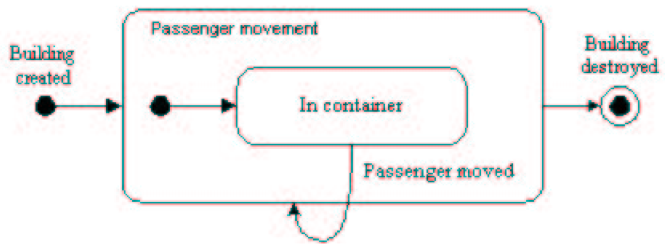


Figure 7.2: Restructured passenger state diagram.

ment of a passenger, called Passenger movement. This class is useful in connection with the statistical evaluation. It makes it possible to reconstruct the exact movement of a given passenger as a function of time.

- **Purpose:** Registration of passenger movements
- **Attributes:** Passenger ID, Time of movement
- **Operations:** N/A

7.1.3 Passenger

- **Purpose:** contains data for a passenger
- **Attributes:** ID number,group ID, List of movements (reference to passenger movement), last movement, destination, time of departure (when passenger moves from pool to queue), Passenger state (indicates if passenger has a new destination)
- **Operations:** setNewDestination(int Destination) (update passenger destination and departure time),
  LogPassengerMovement(double ActualTime, int ContainerID) (Add an object of passengerMovement with parameters to the end of a list of passenger movements), getActionTime() (returns passenger arrival time)

7.1.4 Container

- **Purpose:** Move passengers to next container
- **Attributes:** List of Passengers, No. of Passengers, Next Container, ID

- **Operations:** MovePassenger(Passenger PassengerToMove) (move a passenger object from one container to another),
  AddPassenger(Passenger AddPassenger) (Add a passenger to the list of passengers),
  DeletePassenger(Passenger DeletePassenger, Passenger TestPassenger) (Deletes a passenger),
  setNextContainer(Container SetNextContainer) (reference to next container),
  getContainerID() (returns containerID),
  getCommand() (returns the command to be executed by the observer)

7.1.5 Elevator

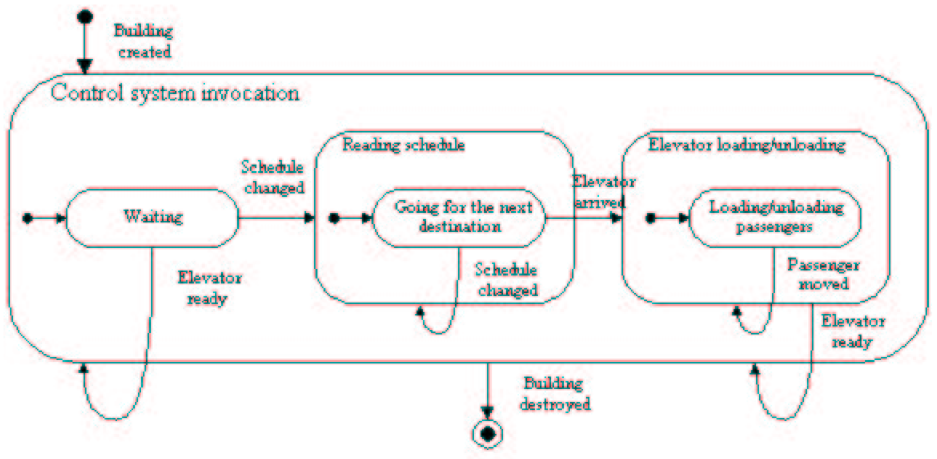


Figure 7.3: Restructured elevator state diagram.

In figure 7.3 you see the iteration Elevator ready. It has been substituted by an attribute in the elevator class called state.

- **Purpose:** Move passengers between floors

- **Attributes:** ID number, next destination, list of unprocessed destinations, state (ready/not ready), max capacity, actual capacity

- **Operations:** setDestination(int NewDestination) (update elevator destination),
  setUnprocessedDestinationFlag() (Notify observer (System Manager) with unpro-

cessed destination),
setElevatorReadyFlag() (notify observer (System Manager) with elevator ready),

7.1.6 Queue

- **Purpose:** Signals that a passenger needs an elevator
- **Attributes:** Next Container
- **Operations:** setInvokeElevatorFlag() (notify observer (System Manager) with call for an elevator)

7.1.7 Pool

- **Purpose:** Contain passengers when they're not circulating in the elevator system
- **Atributes:** N/A
- **Operations:** getNewDestination(Passenger NewPassenger) (notify observer (Destination Manager) to get new passenger destination),
  getPassengerToMoveObject() (returns reference to the passenger entering pool)

7.1.8 Floor

- **Purpose:** Connects the different Containers on each Floor
- **Attributes:** ID, References to pools and queues
- **Operations:** N/A

7.2 Function Component

Specifying operations and placing them in classes, it has been found that a function component is needed in the simulation unit. The function component contains three classes: Destination Manager, Time Manager and Statistic Manager.

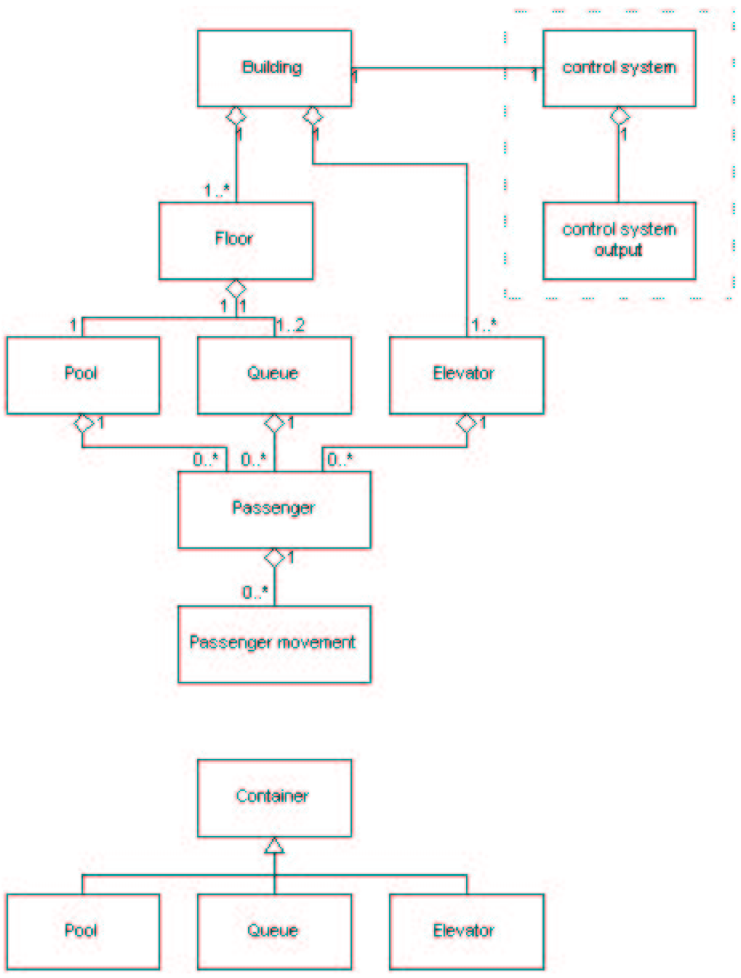


Figure 7.4: Model component structure diagram with new classes. Note that both the building model component and the control system component are included.

7.2.1 Destination Manager

- **Purpose:** Set up all passengers with destination and departure time, during initialisation, give new passenger destinations and departure times during simulation

- **Attributes:** Passenger flow algorithm

- **Operations:** N/A

7.2.2 Time Manager

- **Purpose:** Manage critical moments (events in the simulation like Passenger moved, Elevator arrived, etc.) and execute necessary tasks at these moments

- **Attributes:** List of Passenger events, List of Elevator events

- **Operations:** AddPassengerEvent(Pool PoolObject, Passenger PassengerToAdd), AddElevatorEvent(int ElevatorID, double ActionTime, int Direction)

7.2.3 Statistic Evaluation

- **Purpose:** Calculate statistics and extract the results

- **Attributes:** All Passengers

- **Operations:** CalculateAverage() (Calculate average travel time per floor traveled), CalculateSD() (Calculate standard deviation in travel time per floor traveled), calculateCycles() (transform passenger movements into cycles)

7.3 User interface component

7.3.1 Overview

The user interface includes two kinds of elements. The simulation is set up through a number of windows, and the statistic results of the simulation are shown in a transcript. A menu, common for all the windows, allows the user to navigate between the windows. Below is a description of the functionality of the menu items.

7.3.2 File

New

Activates the setup windows in the following order: Building setup, Elevator setup, Passenger setup and Parameter setup.

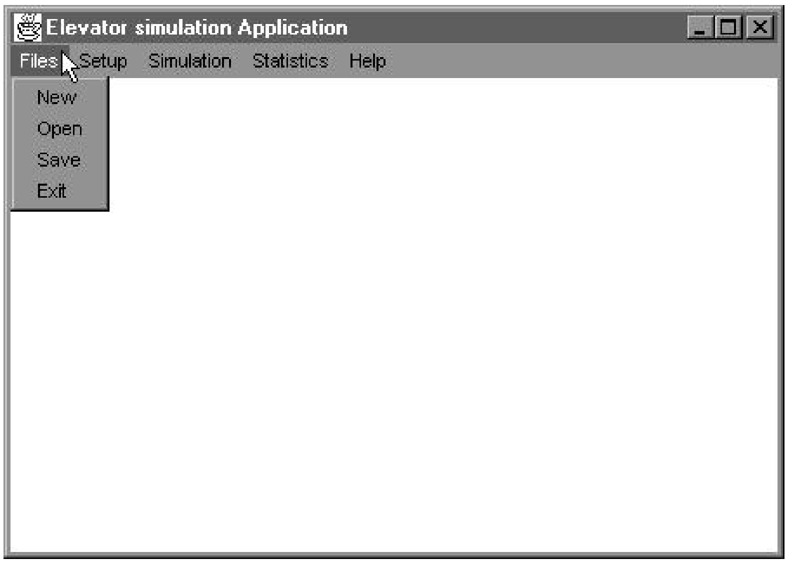


Figure 7.5: The pull down menu Files.

Open

Read the setup of a simulation from a disk. A file dialog box with options is shown.

Save

Save the setup of the actual simulation on a disk.

save as

Save a copy of the actual simulation on a disk. A file dialog box with options is shown.

Exit

Exit the program.

7.3.3 Setup

Building

Opens the window, where the setup for Building is specified. Allows changes to be made in the Building setup.

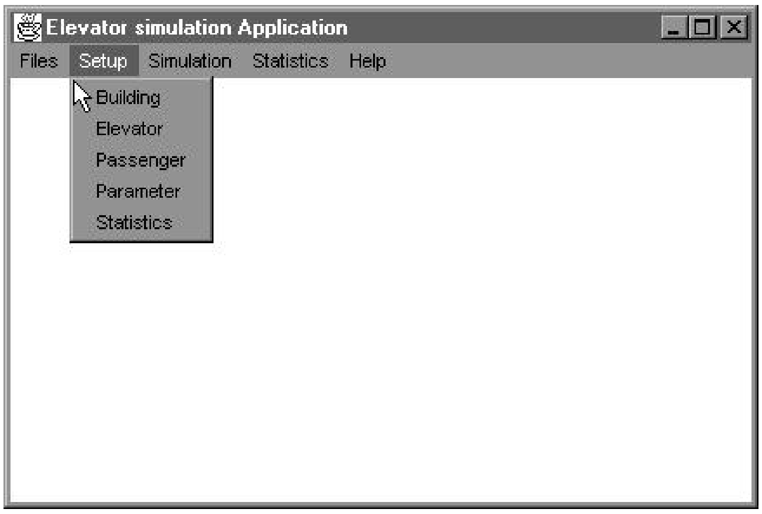


Figure 7.6: The pull down menu Setup.

Elevator

Opens the window, where the setup for Elevator is specified. Allows changes to be made in the Elevator setup.

Passengers

Opens the window, where the setup for Passenger is specified. Allows changes to be made in the Passenger setup.

Parameter

Opens the window where the parameters for the algorithm are specified. Allows changes to be made in the Parameter setup.

Statistics

Opens the window where the setup for the calculation of statistics is specified.

7.3.4 Simulation

Start

Opens the window where the period to be simulated is set and the simulation is started.

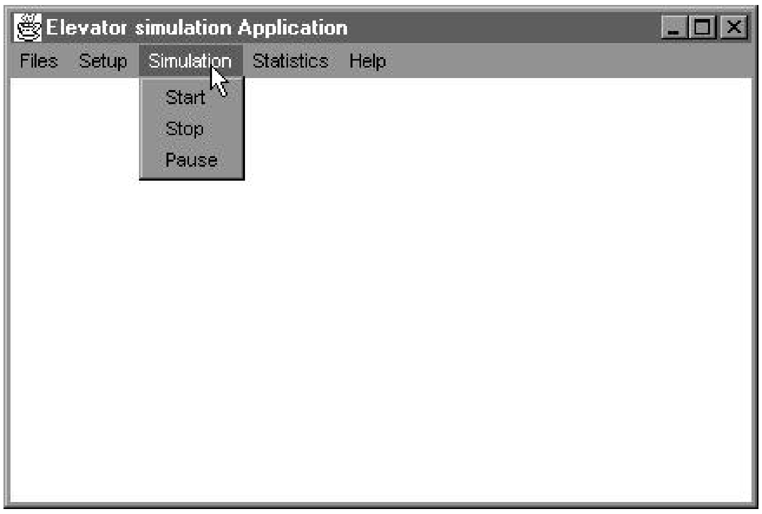


Figure 7.7: The pull down menu Simulation.

Stop

Stops a running simulation.

7.3.5 Statistics

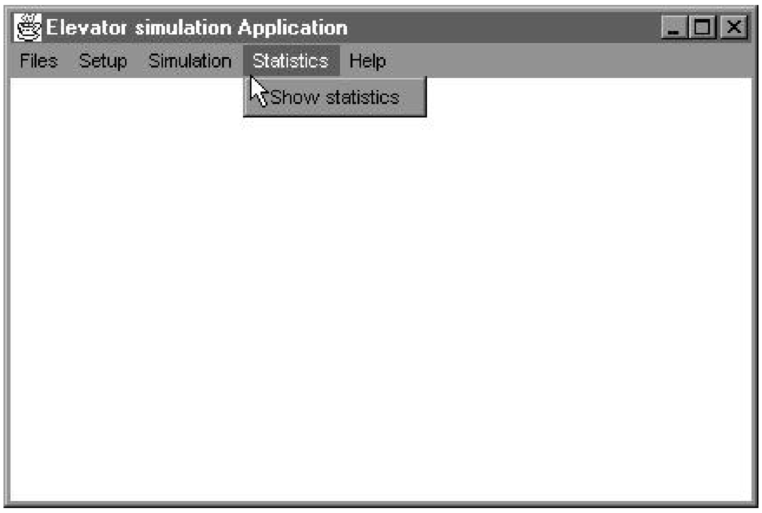


Figure 7.8: The pull down menu Statistics.

Show Statistics

Shows a transcript of the results of the calculation of the statistics .

7.3.6 Help

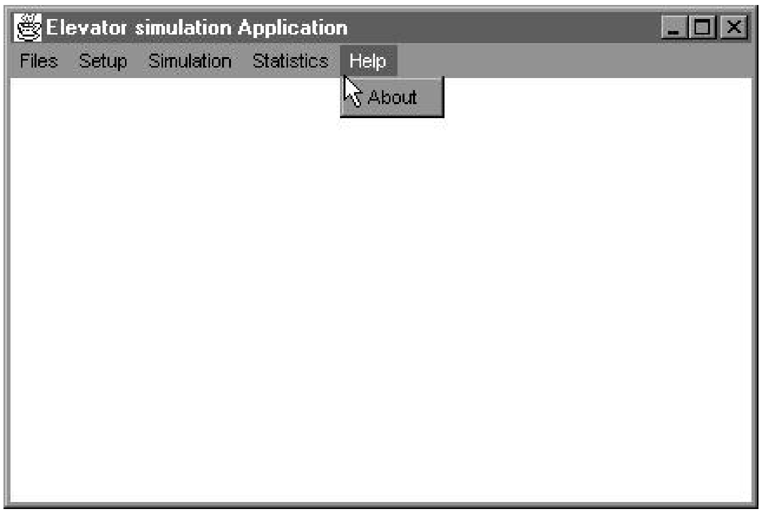


Figure 7.9: The pull down menu Help.

About

Opens the window with program information.

7.4 System interface component

7.4.1 System Manager

- **Purpose:** Manage the exchange of data between Sim-Elevator and the Protocol in the Control System Unit
- **Attributes:** Objects of Building, Time Manager, Elevator, Queue and Protocol
- **Operations:** N/A

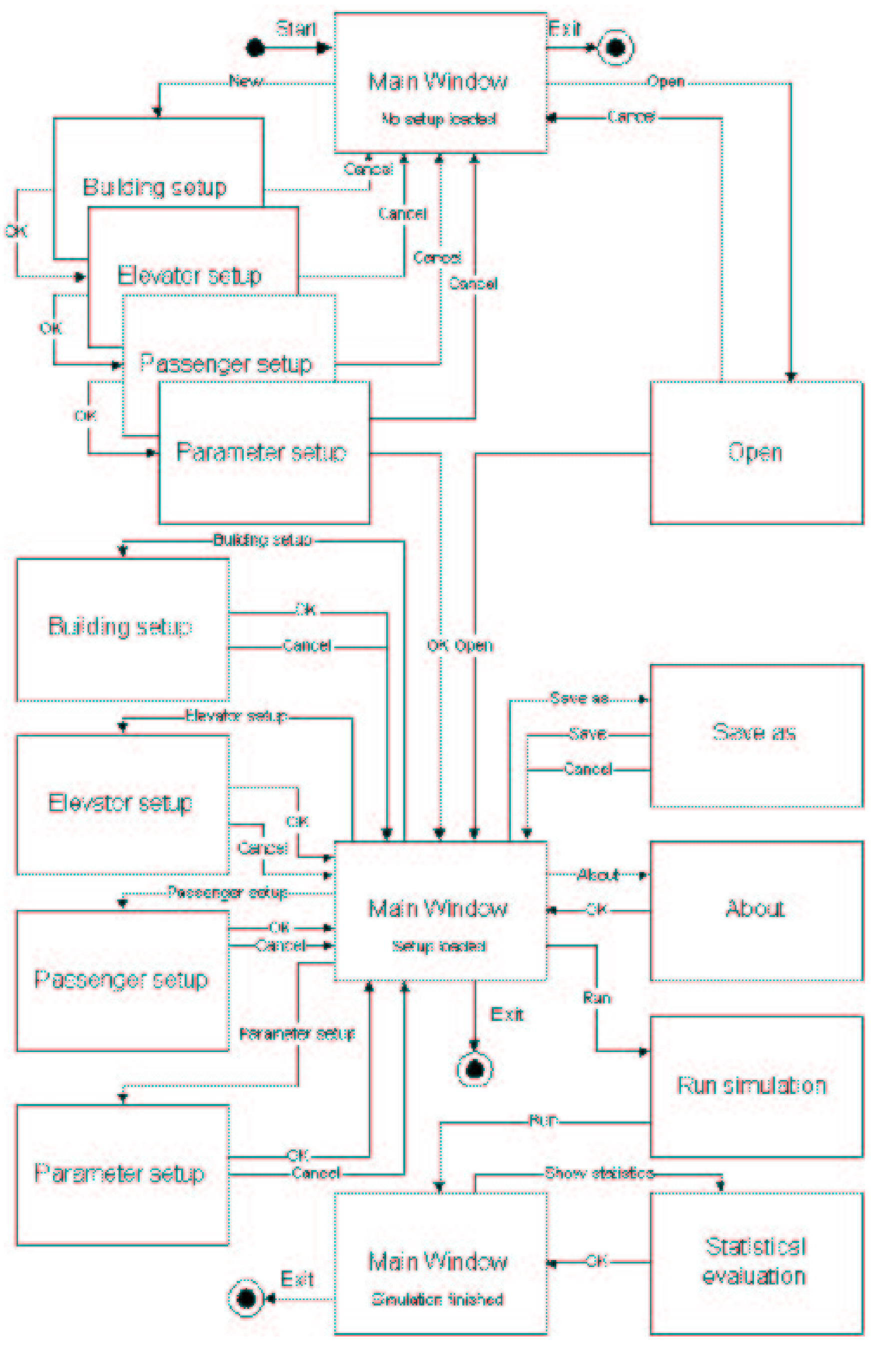


Figure 7.10: Navigation diagram for the user interface. The diagram shows three typical states for the main window on a vertical time axis with the earlier states on top and the later downwards. In the later states options in the earlier are still intact. This is not shown on the diagram.

Chapter 8

Control System

8.1 Model Component

8.1.1 Structure

The structure of the control system model component is depicted in Figure 7.4.

8.1.2 Control System

- **Purpose:** Adds and removes destinations from lists of destinations, evaluates order of elevator movements
- **Attributes:** State, List of Moveable Destinations, List of Control System output
- **Operations:** AddMovableDestination(Destination Dest,ElevatorInfo[] ElevatorInformation) (Adds a new movable destination (passenger destinations in queue)),
AddStaticDestinations (Adds a new List of Static Destinations (passenger destinations in elevators)),
DeleteDestination(Destination Dest, int ElevatorNumber,ElevatorInfo[] ElevatorInformation) (Deletes a destination)

8.1.3 Control System Output

By restructuring the control system state diagram (see Figure 8.1) we get a new class called control system output. The class contains the result of the evaluation algorithm in

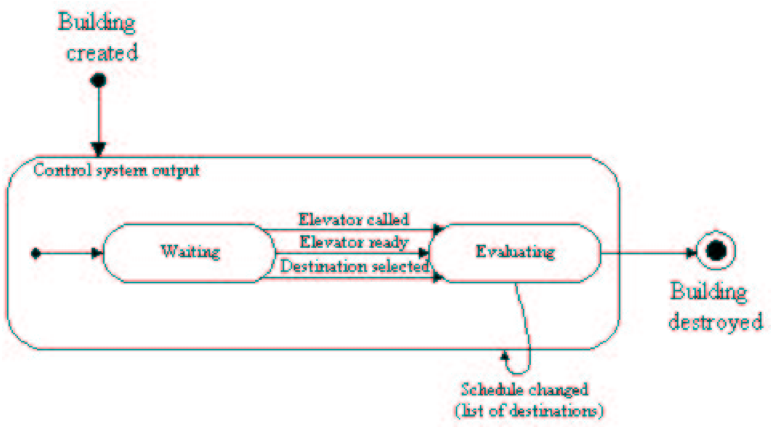


Figure 8.1: Restructured control system state diagram.

the control system in the form of lists of destinations for the elevators.

- **Purpose:** Supply the control system with methods to calculate new destinations for the elevators
- **Attributes:** List of static destinations, List of sorted destinations
- **Operations:** addMovableDestination(Destination Dest, ElevatorInfo ElevatorInformation) (adds the given movable destination to the ControlSystemOutput object at the position which gives the lowest TimeValue and follows the given guidelines), addStaticDestination(Destination Dest) (adds the given static destination to the Control System Output object),
  setScheduleChanged() (notify observer (Protocol) with schedule changed), getNewDestination() (returns new elevator destination)

8.2 Function Component

The control system main component does not contain any function component, all functions are implemented through the system interface component or are contained in the model component.

8.3 System Interface Component

8.3.1 Protocol

- **Purpose:**Manage the exchange of data between Sim-elevator and the control system
- **Attributes:** N/A
- **Operations:** addMovableDestination(Destination Dest,double ActionTime) (add destinations for passengers in elevators),
  addStaticDestination(ListOfDestinations DestinationList, int ElevatorNumber,double ActionTime,double ArrivalTime) (add destinations for passengers in queues),
  setScheduleChanged() (notify observer (System Manager) with schedule changed),
  getNewDestination() (returns new elevator destination), getNewActionTime() (returns the elevator arrival time),
  setElevatorReady(double ActionTime,int ElevatorNumber) (update ElevatorReady flag),

Chapter 9

Program Flow

9.1 Create Passenger

Figure 9.1.
When the simulation is started an object of the class Destination Manager is instantiated. The movement of the passenger is defined for groups of passengers through the setup in the user interface.

1. Call setDestination in Passenger to set the first passenger destination.

2. Call addPassenger in Pool to add the passenger to list of passengers.

3. Call addPassengerEvent in TimeManager to add the passenger to list of passenger events.

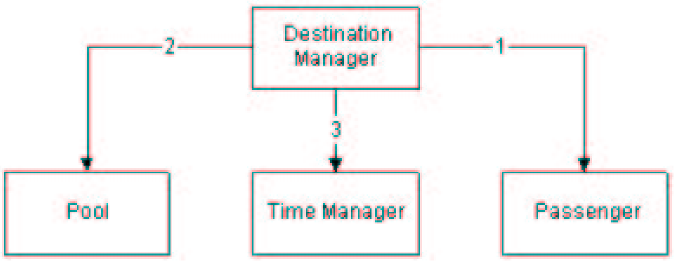


Figure 9.1: Create passenger.

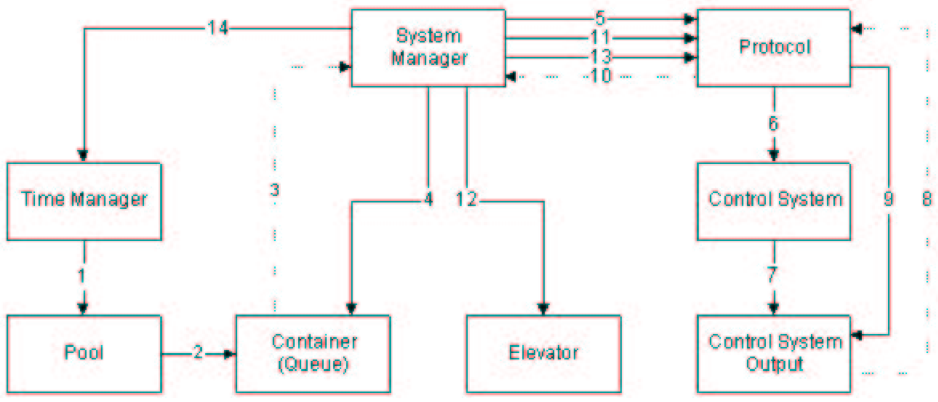


Figure 9.2: Passenger event.

9.2 Passenger Event

Figure 9.2.
Passenger departure time.

1. Call setNextContainer in Pool to set the next container.
   Call movePassenger in Pool to move the passenger.
2. Call addPassenger in Queue to add the passenger to list of passengers.
   Call setInvokeElevatorFlag in queue to invoke an elevator.
3. Notify observer (System Manager) with setInvokeElevatorFlag.
4. Call getContainerID in Container to get the floor number for the invokation.
5. Call addMovableDestination in Protocol to get the invokation processed.
6. Call addMovableDestination in Control System to get the destination added to list of elevator destinations.
7. If the first destination in list of elevator destinations is changed: call setScheduleChanged in Control System Output.
8. Notify observer (Protocol) with setScheduleChanged.
9. Call getNewDestination in Control System Output to get the new elevator destination.

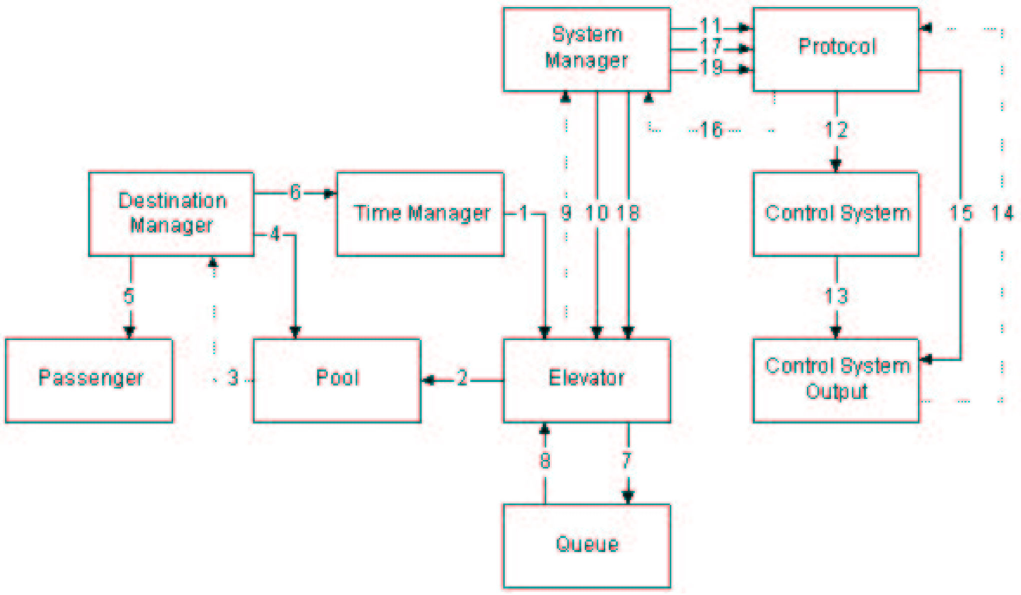


Figure 9.3: Elevator event.

10. Notify observer (System Manager) with schedule changed.

11. Call getNewDestination in Protocol to get the new elevator destination.

12. Call setDestination in Elevator to set the new elevator destination.

13. Call getNewActionTime in Protocol to get the elevator arrival time.

14. Call addElevatorEvent in TimeManager to add the next elevator arrival time.

9.3 Elevator Event

Figure 9.3.
The elevator arrives at a floor.

1. Call MovePassenger in Elevator to unload and load passengers.

2. Call addPassenger in Pool to add the passenger to list of passengers.

3. Notify observer (DestinationManager).

4. Call getPassengerToMoveObject in Pool

5. Call getActionTime in Passenger and calculate the new destination.
   Call setNewDestination in Passenger to set the new destination and depaturetime.

6. Call addPassengerEvent in TimeManager to add the passenger event to list of passenger events.

7. Call movePassenger in Queue to load the passenger from the queue to the elevator.

8. If passengers are loaded call setUnprocessedDestinationFlag in Elevator.
   If no passengers are loaded call setElevatorReadyFlag in Elevator.

9. Notify observer (System Manager) with either setElevatorReadyFlag or SetUnprocessedDestinationFlag.

10. If setElevatorReadyFlag (9)
    Call getCommand in Elevator. getCommand returns ElevatorReady.
    If setUnprocessedDestinationFlag (9)
    Call getCommand in Elevator. getCommand returns UnprocessedDestination.

11. If setElevatorReadyFlag (9)
    Call setElevatorReady in protocol.
    If setUnprocessedDestinationFlag (9)
    Call addStaticDestinations in protocol.

12. If setElevatorReadyFlag (9)
    Call deleteDestination in ControlSystem to delete the elevator destination.
    If setUnprocessedDestinationFlag (9)
    Call addStaticDestinations in ControlSystem to add the passenger destinations to the list of elevator destination.

13. If setElevatorReadyFlag (9)
    Evalaute the list of destinations and call setScheduleChanged in Control System Output.

14. If setUnprocessedDestinationFlag (9)
    Evalaute the list of destinations and call setScheduleChanged in Control System Output.

15. Notify observer (Protocol) with schedule changed.
16. Call getNewDestination in Control System Out to get the new elevator destination.
17. Notify observer (System Manager) with schedule changed.
18. Call getNewDestination in Protocol to get the new elevator destination.
19. Call setDestination in Elevator to set the new elevator destination.
20. Call getNewActionTime in Protocol to get the elevator arrival time.
21. Call addElevatorEvent in TimeManager to add the next elevator arrival time to list of elevator events.

Part iii

Implementation

Chapter 10

Implementation

This chapter describes the steps taken to convert the design into a JAVA program.
The program only uses classes and methodes from Sun's JDK 1.1.6.
This makes it possible to execute the program on different platforms.
This version of the program is tested succesfully on both a PC- and a UNIX platform.
The use of standard JDK classes makes the program able to load and save files using the file dialog implemented in the platform.
There are different ways of implementing the program. The order of the implementation determine the amount of extra sourcecode to be implemented in order to test the different classes as they are implemented. A combination of early user interface and button-up implementation, gives a graphic test platform and a small amount of extra test-sourcecode to be implemented.
Some parts of the program have not been implemented, specially parts of the GUI have been let out. E.g. only four passenger flows have been implemented though this is enough to show the funtionality of the program. The edit function in the GUI is not needed to show the functionality of the program and therefore not implemented. The statistic part of the simulation is not fully graphicly implemented.

10.1 Class Structure

The different classes described in the design chapter is implemented by the members of the group. Each member is responsible for the implementation of a number of classes. To make sure the communication between the classes is

in agreement with the design, each public method were designed with a contract defining what is required and what has to be insured through the communication and functionality of the method.
By fulfilling these contracts it should be possible to implement all the classes into one program.
The order of implementation is as follows:

- Userinterface
- Model component (Simulation)
- Model component (Control system)
- Function component
- System interface between the simulation and the control system.

10.2 User Interface

The userinterface is controlled by the use of pulldown menus and consist of a number of windows to change the setup of the program.

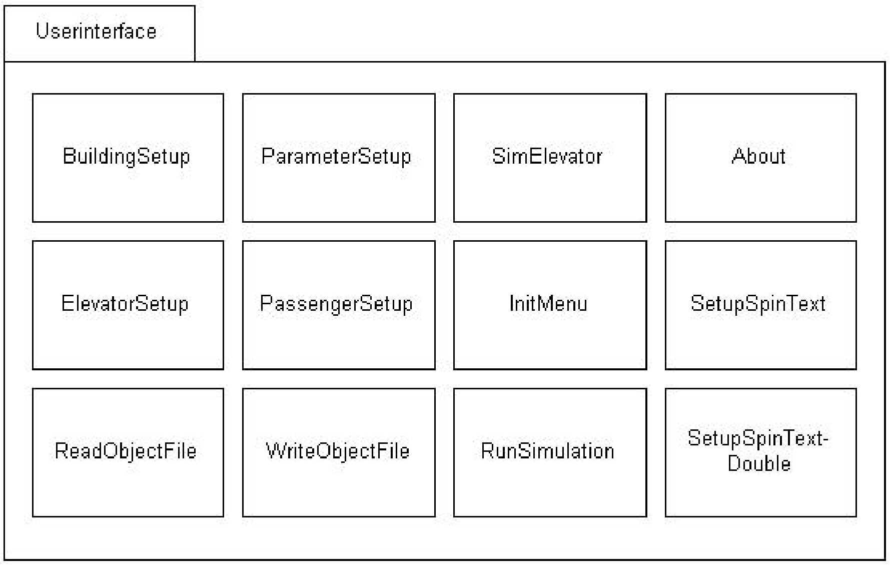


Figure 10.1: Userinterface Component.

Each menu item is implemented as a class. (See Figure 10.1) This makes it easy to implement and test one window at a time. Each
window is an unit with its own actionlistener, this makes it possible to add or remove a window without too many changes in the other user interface classes. The class containing a GUI interface has the
responsibility of saving the data specified on the GUI. The class DataContainer keeps track of all parameters specifying a simulation.
The parameters from BuildingSetup is saved as attributes to the DataContainer class. These data is used to limit some of the other parameters e.g. Start Floor of a passenger is limited to the number of
floors in the building. The parameters of the elevators (ElevatorSetup) is saved in an array of ElevatorData objects, one for each elevator in the building. The array is an attribute of the class DataContainer. The parameters of the ParameterSetup are attributes of ElevatorData too and will be added to each ElevatorData object.
The most complex GUI is the PassengerSetup. It consist of two windows. The first GUI of the PassengerSetup is used to specify the group. The second GUI depend on the type of travel performed by the passengers of the group. This GUI is based upon three GUIs containing specification of the passengers timetable, and three GUIs containing the specificaton of the passengers travel table. The second GUI consists of a timetable in the upper part of the GUI and a travel table in the lower part. By combining these six half part GUIs, it is possible to show nine different GUIs. Each one of the nine GUIs got its own data class with attributes specified for the parameters of the GUI. The data classes are saved in form of a linked list and represented at the bottom level in Figure 10.2.
The classes SetupSpinText and SetupSpinTextDouble are two components used as input controls in the setup GUIs, these components uses standard JAVA components. ReadObjectFile and WriteObjectFile are the classes that controls the access of the setup files. The class SimElevator and InitMenu is used to generate and control the main window. RunSimulation is resonsible for the execution of an simulation.
When the user interface is implemented it is possible to use this to instantiate the classes of the model component and test the instantiation of the classes. This will minimize the amount of sourcecode used to test the classes of the model component.

10.3 Model Component (Simulation)

It is important to implement the classes of the model component early in the implementation phase, because these classes are used in the test of the function component classes. It is very important that the contracts between the methods are fulfilled. This makes the connecting of the classes easier. The use of button-up implementation makes it possible to use the tested classes of the model component when implementing the classes of the function component. The model component is divided in two, a model component of the simulation and the model component of the control system. The simulation part contains the classes of the system that uses the control system to circulate in the building.

10.4 Model Component (Control System)

It is possible to look upon the control system as a small part of the entire program. Using the user interface to setup the control system and redefining some of the menu items it is possible to test the control system. The data from the setup windows combined with the already implemented classes can be used to check if the program is working properly. When using already implemented and tested classes to test the implementation of the control system classes it is possible to make a more complete test of the user interface and minimize the chances of making testcode with bugs.

10.5 Function Component

The implementation of the function component classes starts when the implementation of the model component classes is in the final state (test of the functionality and contract). The first class of the function componet to be implemented is the DestinationManager. This class makes it possible to create the passengers and place them on the right floor.
The next class to implement is SystemManager. The SystemManager is special, because it is an observer of the classes Queue, Elevator and Protocol. The purpose of the SystemManager class is to ensure encapsulation. It is necessary to implement the SystemManager as three classes (SystemManagerObserverForQueue, SystemManagerObserverForElevator and SystemManagerObserverForProtocol), each of these classes in the SystemManager are observers. At this time in the total implementation phase, it is possible

to test the implementation in a way, that makes a passenger capable of traveling from one floor to another, with the use of an elevator.
The last of the funtion component classes is the DestinationManager. When the DestinationManager is fully implemented the passenger is capable of obtaining a new destination and departuretime. This end the cycle of the passengerflow. The DestinationManager gets the information about the passenger flow for a passenger from the DataContainer.
The structure of the passenger flow in the DataContainer is as shown in Figur 10.2. Each passengers flow is defined in DataPassengerFlow. The DataPassengerFlow has an attribute of the type Group. The class Group is an abstract class containing two abstract methods (getNewTime and getNewDestinaton).
The classes GroupGetTypeSpecificTime and the other classes on the same level inherits from the class Group and they also imply the method getNewTime.
The classes GroupGetTypeSpecificTimeAndOneFloor and the other classes on the same level inherits from the classes on the level above and inherits the method getNewDestination.
GetNewDestination has to be called first, because it gets the information from data classes placed on the bottom level of the data structure.
Because all of the getNewTime is called with the same type of parameters it is posible to overload the method getNewTime in Group.
The same applies to the method getNewDestination.
The implementation is at this time in a final state.
Now it is possible to try the different types of passengerflow.

10.6 System Interface

The system interface between the simulation and the control system consists of the SystemManager and the Protocol. It is based on the use of the design pattern Observer. The SystemManager is implemented as the Protocols observer and controls the communication between the simulation and the control system through the Protocol. The purpose of the Protocol is to act as converter between the elevator and the control system. Depending on how advanced the elevator is the Protocol must contain methods to calculate the information needed for the control system to determine the flow of the elevators.

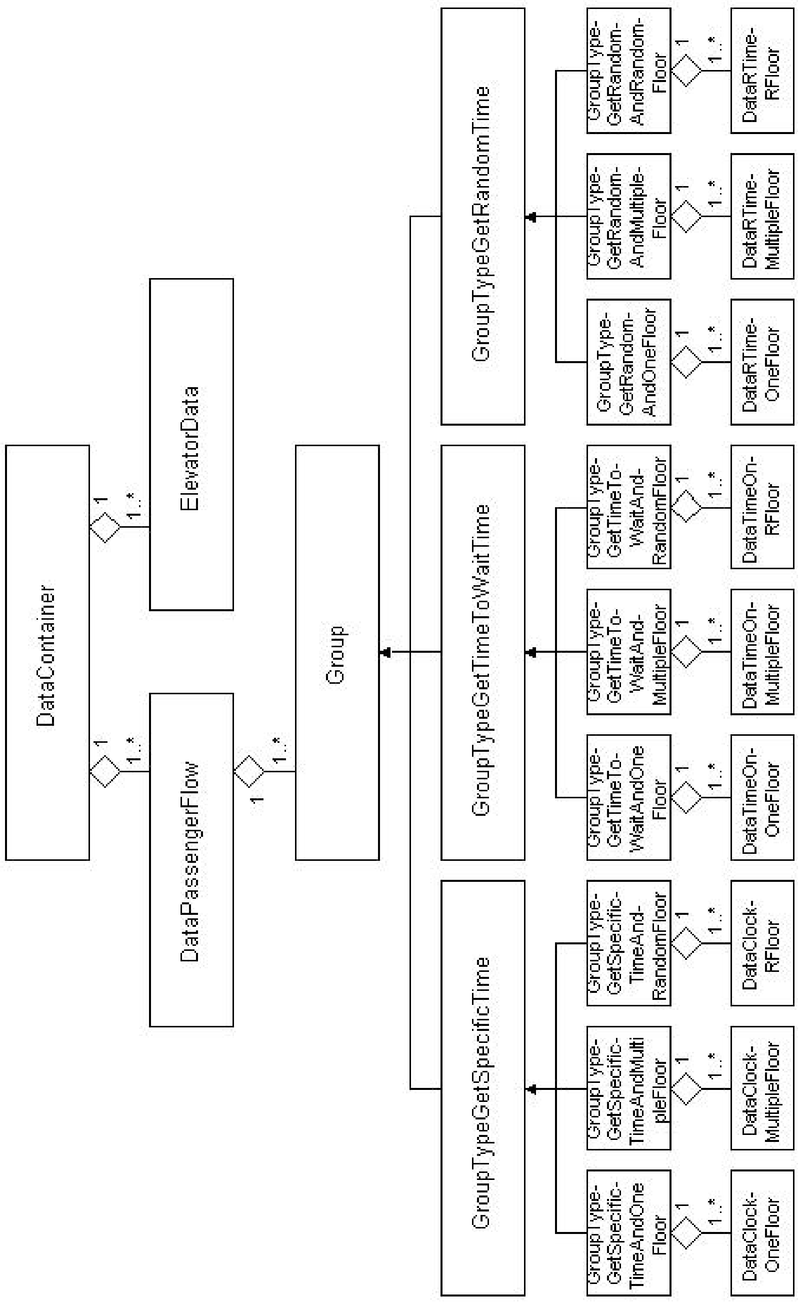

DataContainer
1
1..*
ElevatorData
1
1..*
DataPassengerFlow
1
1..*
Group
GroupTypeGetSpecificTime
GroupTypeGetTimeToWaitTime
GroupTypeGetRandomTime
GroupType-GetSpecific-TimeAndOne Floor
GroupType-GetSpecific-TimeAndMultipleFloor
GroupType-GetSpecific-TimeAnd-RandomFloor
GroupType-GetTimeTo-WaitAndOne Floor
GroupType-GetTimeTo-WaitAnd-MultipleFloor
GroupType-GetTimeTo-WaitAnd-RandomFloor
GroupType-GetRandom-AndOneFloor
GroupType-GetRandom-AndMultiple-Floor
GroupType-GetRandom-AndRandom-Floor
1
1..*
DataClock-OneFloor
DataClock-MultipleFloor
DataClock-RFloor
DataTimeOn-OneFloor
DataTimeOn-MultipleFloor
DataTimeOn-RFloor
DataRTime-OneFloor
DataRTime-MultipleFloor
DataRTime-RFloor

Figure 10.2: Data Class Structure.

Part iv

Test

Chapter 11

Test

11.1 Test Strategy

As an overall test strategy it has been decided that each programmer is responsible for the test of the classes he has programmed. In the process of integration tests have been performed “bottom up”, that is classes in the model component were tested thouroughly before integrated with the classes in the function component. This procedure were then repeated before integration with the user and system interfaces. Finally the two main parts of the program were combined and tested up against each other.

11.2 White Box

11.2.1 Method

The principle of a white box test is that you identify all possible independent paths through a given method and use them as test cases.
This test method is very extensive and time-consuming. Thus we have not been able to use it throughout the whole program. We have, however, performed it thouroughly on a single sequence of the movePassenger method in the Elevator class to demonstrate the principle.

11.2.2 Elevator.movePassenger

The following is a part of the method movePassenger in Elevator. We presume that the variable NextDirection is Undecided:

```
if (NextDirection == UNDECIDED) {
  if (!CurrentFloor.getUpQueue().isEmpty() &&
      !CurrentFloor.getDownQueue().isEmpty()) {
    if (CurrentFloor.getUpQueue().getFirstPassenger().getDepartureTime()<
        CurrentFloor.getDownQueue().getFirstPassenger().getDepartureTime())
      NextDirection = UP;
    else
      NextDirection = DOWN;
  }
  else {
    if (CurrentFloor.getUpQueue().isEmpty() &&
        CurrentFloor.getDownQueue().isEmpty()) {
      setElevatorReadyFlag();
    }
    else {
      if (CurrentFloor.getUpQueue().getNoOfPassengers()>
          CurrentFloor.getDownQueue().getNoOfPassengers())
        NextDirection = UP;
      else
        NextDirection = DOWN;
    }
  }
}
```

11.2.3 Flow Graph

Based on the source code the possible ramifications of the method movePassenger are displayed in Figure 11.1, each node refering to a line in the code and the arrows being the transition to the next line.

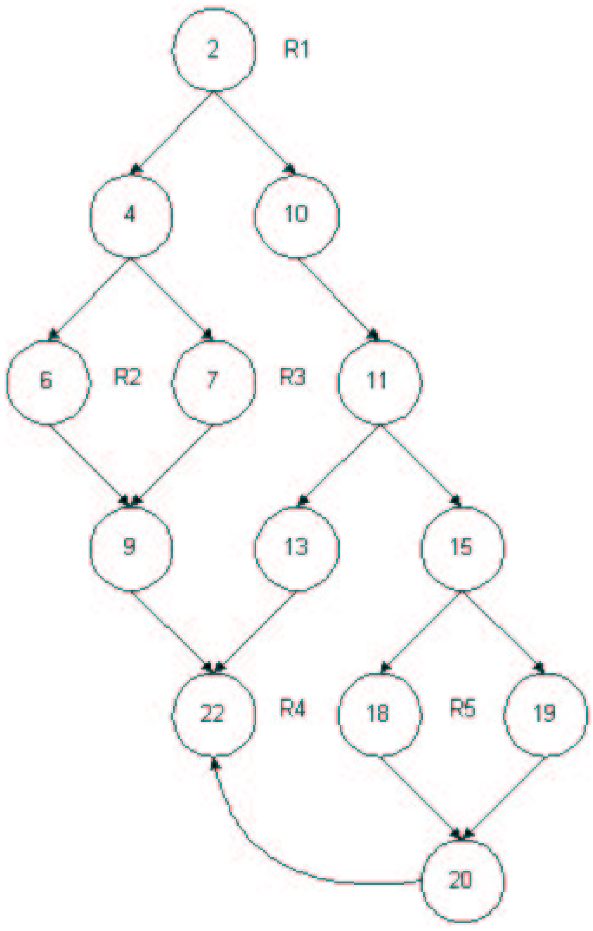


Figure 11.1: Flow graph of Elevator.movePassenger.

11.2.4 Indentification of Independent Paths

There are various ways to indentify the different independent paths, the number of independent paths being $V(G)$:

- Count the regions (the R's), giving $V(G) = 5$.
- $V(G) = arrows - nodes + 2 = 15arrows - 13nodes + 2 = 5$.
- $V(G) = ramifications + 1 = 4ramifications + 1 = 5$.

Figure 11.2.4 shows the independt paths of the method.

Test Case	Path
1	$2 - 4 - 6 - 9 - 22$
2	$2 - 4 - 7 - 9 - 22$
3	$2 - 10 - 11 - 13 - 22$
4	$2 - 10 - 11 - 16 - 18 - 20 - 22$
5	$2 - 10 - 11 - 16 - 19 - 20 - 22$

Figure 11.2: The independent paths of Elevator.movePassenger.

11.2.5 Test Cases

The following test cases completes a white box test of Elevator.movePassenger. All details are not included.
Once again, we presume that NextDirection is UNDECIDED.
The results (NextDirection, setInvokeElevatorFlag) of each test case can be obtained by substituting (or NextDirection and setInvokeElevatorFlag with System.out.println(<string>).

Case 1

```
Passenger Karsten = new Passenger(1,355);
Passenger Tonny = new Passenger(2,355);

Floor A5 = new Building(NoOfElevators, NoOfFloors);

Karsten.setDepartureTime(60);
Tonny.setDepartureTime(90);

A5.getFloor(ThisFloor).getUpQueue().addPassenger(Karsten);
A5.getFloor(ThisFloor).getDownQueue().addPassenger(Tonny);

A5.getElevator(ThisElevator).setDestination(ThisFloor);
A5.getElevator(ThisElevator).movePassenger(TimeOfMovement);
```

Case 2

```
Passenger Karsten = new Passenger(1,355);
Passenger Tonny = new Passenger(2,355);
```

```
Floor A5 = new Building(NoOfElevators, NoOfFloors);

Karsten.setDepartureTime(90);
Tonny.setDepartureTime(60);

A5.getFloor(ThisFloor).getUpQueue().addPassenger(Karsten);
A5.getFloor(ThisFloor).getDownQueue().addPassenger(Tonny);

A5.getElevator(ThisElevator).setDestination(ThisFloor);
A5.getElevator(ThisElevator).movePassenger(TimeOfMovement);
```

Case 3

```
Floor A5 = new Building(NoOfElevators, NoOfFloors);

A5.getElevator(ThisElevator).setDestination(ThisFloor);
A5.getElevator(ThisElevator).movePassenger(TimeOfMovement);
```

Case 4

```
Passenger Karsten = new Passenger(1,355);

Floor A5 = new Building(NoOfElevators, NoOfFloors);

A5.getFloor(ThisFloor).getUpQueue().addPassenger(Karsten);

A5.getElevator(ThisElevator).setDestination(ThisFloor);
A5.getElevator(ThisElevator).movePassenger(TimeOfMovement);
```

Case 5

```
Passenger Tonny = new Passenger(2,355);

Floor A5 = new Building(NoOfElevators, NoOfFloors);

A5.getFloor(ThisFloor).getDownQueue().addPassenger(Tonny);

A5.getElevator(ThisElevator).setDestination(ThisFloor);
```

```
A5.getElevator(ThisElevator).movePassenger(TimeOfMovement);
```

Note that these tests were performed before the SimElevator model, function and system interface components were integrated. No coupling existed between the components and the constructors were simpler than the final ones.

11.3 Black Box

11.3.1 Method

Most of our program has been tested by the black box method.

11.3.2 Performing The Test

The following is the source code used for the black box test of the SimElevator model-component. The main purpose of this component is circulation of passengers.

```
class Test {

  public static void main(String[] args) {

    // Create building and passengers

    Building A5 = new Building(1,3);
    Passenger Karsten = new Passenger(666,1);
    Passenger Tonny = new Passenger(4,1);
    Passenger Lars = new Passenger(9,1);
    Passenger Claus = new Passenger(13,1);

    // Circulate passengers

    Tonny.setNewDestination(2);
    Karsten.setNewDestination(2);
    Lars.setNewDestination(3);
    Claus.setNewDestination(2);

    A5.getElevator(0).setDestination(1);
    A5.getElevator(0).setNextDirection(ELEVATOR.UP);
```

```
A5.getFloor(1).getPool().addPassenger(Tonny);
A5.getFloor(1).getPool().addPassenger(Claus);
A5.getFloor(1).getPool().addPassenger(Karsten);
A5.getFloor(1).getPool().addPassenger(Lars);

A5.getFloor(1).getPool().movePassenger(Tonny,1);
A5.getFloor(1).getPool().movePassenger(Karsten,2);
A5.getFloor(1).getPool().movePassenger(Claus,3);
A5.getFloor(1).getPool().movePassenger(Lars,1);

A5.getElevator(0).movePassenger(4);

A5.getElevator(0).setDestination(2);
A5.getElevator(0).movePassenger(7);

Claus.setNewDestination(1);
Tonny.setNewDestination(1);

A5.getFloor(2).getPool().movePassenger(Tonny,8);
A5.getFloor(2).getPool().movePassenger(Claus,8);

A5.getElevator(0).setDestination(3);
A5.getElevator(0).setNextDirection(ELEVATOR.DOWN);
A5.getElevator(0).movePassenger(10);

A5.getElevator(0).setDestination(2);
A5.getElevator(0).movePassenger(12);

A5.getElevator(0).setDestination(1);
A5.getElevator(0).movePassenger(15);


// Verify movement of passengers and elevator

for (int i = 0; i < A5.getElevator(0).getMovementArray().length; i++)
  System.out.println("ELEVATOR: Ankomst kl. "+
                    A5.getElevator(0).getMovementArray()[i].getTimeOfMovement()+
                    " til "+A5.getElevator(0).getMovementArray()[i].getContainerID()+
                    ". etage");
```

```
    for (int i = 0; i < Karsten.getMovementArray().length; i++)
      System.out.println("KARSTEN: Flytning kl. "+
                                    Karsten.getMovementArray()[i].getTimeOfMovement()+
                                    " fra "+Karsten.getMovementArray()[i].getContainerID());

    for (int i = 0; i < Tonny.getMovementArray().length; i++)
      System.out.println("TONNY: Flytning kl. "
                                    +Tonny.getMovementArray()[i].getTimeOfMovement()+
                                    " fra "+Tonny.getMovementArray()[i].getContainerID());

    for (int i = 0; i < Lars.getMovementArray().length; i++)
      System.out.println("LARS: Flytning kl. "
                                    +Lars.getMovementArray()[i].getTimeOfMovement()+
                                    " fra "+Lars.getMovementArray()[i].getContainerID());

    for (int i = 0; i < Claus.getMovementArray().length; i++)
      System.out.println("CLAUS: Flytning kl. "
                                    +Claus.getMovementArray()[i].getTimeOfMovement()+
                                    " fra "+Claus.getMovementArray()[i].getContainerID());

    // Calculate statistics

    StatisticManager BigBrother = new StatisticManager();

    A5.endSimulation(BigBrother);

    BigBrother.calculateCycles();

    System.out.println("Gennemsnitlig rejsetid per rejst etage: "
                                  +BigBrother.calculateAverage());
    System.out.println("Spredning: "+Math.sqrt(BigBrother.calculateSD()));
    System.out.println("Antal rejser: "+BigBrother.getNoOfCycles());

  } // main
} // Test
```

The output is then compared to the expected results.
By substituting all notify(this) with something like System.out.println("Invoke Elevator"); the outgoing signals to the function and interface components are tested.

Part v

Study Journal

Chapter 12

Study Journal

This Study report is intended to explain the origin of this project as well as sum op the most important selections and decisions during the developing process. Further more the used methods and programming tools will be commented to make a basis to evaluate the experiences in object oriented analysis, design and programming.

12.1 Method

The method used in this software development project is Object Oriented Analysis and Design (OOAD). During the course in OOAD we used [3]. The purpose of the course was to learn how to think object oriented and analyse and design by that method.
A short comment to the OOAD book, is the lag of examples of dynamic programs, simulations, etc. It shows only administrative programs like electronic schedules, which makes it difficult to abstract from the book examples to our project which is a little more complex as we simulate a dynamic environment with moving passengers and elevators.

12.2 Programming Tools

With reference to the PE-course Object Oriented Programming (OOP), the source-code for the program have been written in the object oriented programming-language JAVA, being specific JDK v. 1.1.6, as this is the version used in the OOP-course, and available on the computer system at KOM-department on Aalborg University.
The literature used in the course is "Java Software Solutions, Foundation of Program

Design", [5] by John Lewis and William Loftus. As supplemental literature we have received documents about design patterns [2], threads [1], tests[7] and contracts [6]. Considering the difficulties in creating the graphical user interface we might have obtained great advantage by using Swing (integrated in JDK 1.2.0) which is a GUI component kit that simplifies the development of window components in JAVA.

12.3 The Working Process

12.3.1 Time Schedule

During the project period, we have not used a specified time plan, but a calendar with milestones and deadlines. This type of plan have been satisfying for all members in the group, and there have not occurred any kind of problems respecting the deadlines. To make a few comments to our plan, we might have disposed more time to the implementation and testing. But we decided from the beginning to spend quite a lot of time to get acquainted with the methods, furthermore we used 3 weeks of our project period on the mini-project in the "Analog Elektronik" course which demanded our full attention.

12.3.2 Group Decisions

The Group have worked as a democratic group, but the decision-making process have been characterized by listening to those with experience in a specific field. This type of decision-making have especially in the design phase been practiced as we distributed the work to single persons or small subgroups. Then is was up to the person(s) in the subgroups to design parts of the system, thus they should observe certain rules and specifications set by the whole group.

12.4 Choosing the Project

Friday the 4th of September all students in the D3 - Computer Engineering semester had to choose a main project. The theme of all projects was, due to an exemption of the study arrangement middle of 97, controlling of dynamic systems e.g.. traffic lights, elevators, network traffic, etc. As this project group had worked together in former projects, we made a briefly discussion, and decided that we did not want to do a project which would demand a lot of specialized knowledge (we tried that last semester), but a "simple" project so we could devote this semester to learning OOP and OOAD. Therefore we chose the Elevator Control system project as we saw this as a more simple project than e.g.. computer network traffic controlling where we first had to learn about computer network protocols that might have meant learning almost every rfc-document!

12.5 Analysis

After choosing the project "Elevator Control system", we started to do the first exercises in the analysis phase according to the OOAD course.

12.5.1 Defining The System

First thing to do was defining the base system, and then setup some criteria explaining how the system should work, e.g. what tasks the user of the system should be able to accomplish, what philosophy and idea to be underlying the system, etc. The project group had nearly similar thoughts about what the system should look like, thus we had quite different opinions whether the system should be central or decentral controlled. The majority decided central control, with the argument that we wanted the control system to be an entire unit, that is the control system for all times should keep track of and know everything about all elevators.
The fact that there is not an actual user of our program, might seem to have made it easier to develop the elevator control system, but actually it has unnecessary complicated the analysis of the problem domain, because we have spent a lot of time discussing things about the elevator control system that would have been given if we had an actual user (e.g. elevator design).
To sum op the analysis phase, we planned the basic functionality of the system. Perhaps we should have split the program in two parts, a control system and a simulation each with corresponding interfaces. If we had done so we could have divided the project group into two, one designing the simulation and the other designing the control system. By doing this we could have taken more advantage of the resources in the group.

12.6 Design

When we entered the design phase we still had to finish some minor things in the analysis document. But not more than we could start designing the program. First of all we had to set up some design criteria specifying whether the system should be secure, efficient, reusable, etc.

12.6.1 Criteria

As we were going to make a control system to an elevator system there were specific criteria we had to give higher priority than other. Specially we wanted the system to be integratable, as we wanted to be able to take out the control system an integrate it in an existing building. On the other hand we did not want a very secure system as it should run on a stand-alone computer. All in all we wanted a flexible and reliable system, which was easy to maintain. Furthermore it should be rather easy to make some test on it, because we had to simulate the passenger/building environment.
During the design phase we could have used some experience in programming in Java. This would have made us able to include some of the Java specific parts e.g. observers and thereby avoiding inconsistence between design and implementation. Further more it would generally have made it easier to design the program.

12.7 Implementation

The implementation phase has been the most hectic period in the project, as said earlier in this chapter, we could have used more time to implement the program. Before we began to write the source code we sat down and wrote some simple contracts, telling what the different pieces of the program should require, contain and ensure. Then we divided the tasks among single persons or subgroups letting each one decide how the code should be implemented. It showed out, that it might have been a good idea to use a little more time making the contracts, thus we were not able to write all the code from the contracts only. But it was a step in the right direction, and we learned that it is definitely a necessary process to complete before making programs bigger than this. Nevertheless we learned a great deal about teamwork, supporting one another to get the information needed to write the pieces of the program in a way that made it possible to integrate all parts easily.

The graphical user interface is not implemented using contracts. Fortunately we heard rumors that it is a little tricky getting the graphical user interface to look as intended, so we began working on it at the end of the design phase. This showed out to be a good idea, because it turned out that the rumors about the GUI were true. We might have been able to save some time if we had used SWING, but we decided not to do this, because SWING only existed as an beta version when we began to implement the program.
By letting the subgroups decide for themselves how they should implement the code resulted in making some minor new classes and methods, which were not described in the analysis and design documents. This happened because we was not aware of the need for these classes and methods to realize the program parts. Anyway most of these classes and methods are private and therefor they cannot be seen from other objects.

12.8 Testing

During the implementation phase we actually did some of the testing. First of all, the syntax and semantic failures had to be fixed, so the program parts could compile! Then we had to set up some test routines to check whether, the parts would act like we wanted to. As the process went on more and more parts of the program could be joined, and we could make further tests to see if the parts could work together.
This kind of testing can only be considered as a basic test. To do a more structured test we could have specified some test routines to different parts of the program, from the contracts for all the methods and classes. By doing that we could more easy ensure the tests were performed on the right methods and classes. Of course it would be an overkill to test all the trivial methods.

12.9 Experience

The purpose with this project has from the beginning been to develop an almost fully operating elevator control system and make a simulation of a building with passengers and elevators to test the efficiency of the program , supported by a graphical user interface. And of course learn the methods of object oriented analysis and design. Our main benefit must be to gather knowledge to support our further studies.
Before we started this project only a few of the group members had actually written some

Java code, so we had to learn the most of Java from basics which was not that hard as we learned to think object oriented, though some aspects of learning Java were complicated due to lack of adequate explanations in Sun’s Java Development Kit Documentation [4]. Still as a result of this project we now have obtained a good overview of the possibilities of Java.
It has been a challenge to make this program executing a dynamic simulation, because not only did we have to create the simulation part, which is quite complicated, but we also had to make the surroundings, so we could test the simulation. A result of this report is our insight in simulation and the complications in testing a given simulation.

Part vi

Appendix

Chapter 13

Statistics

To give a crude evaluation of the performance of the control system, different statistical values are calculated.
If time-efficiency is important, the time consumation of each passenger from the time he enters the queue till he leaves the elevator, is of interest. This must be seen in connection with the number of floors traveled.
The following equation gives the average time consumation per floor traveled:

$$\mu = \frac{1}{n}\sum_{i=1}^{n}\frac{\Delta t_i}{\Delta s_i} \tag{13.1}$$

n is the number of cycles of all passengers, Δt the time consumation from the passengers entering the queue until he leaves the elevator, and Δs_i is the number of floors traveled.
Also, the standard deviation gives a picture of the variation in the time consumation per floor traveled.

$$SD(\frac{\Delta t}{\Delta s}) = \frac{1}{n}\sum_{i=1}^{n}(\frac{\Delta t_i}{\Delta s_i} - \mu)^2 \tag{13.2}$$

$$\sigma = \sqrt{SD(\frac{\Delta t}{\Delta s})} \tag{13.3}$$

A good measurement for the time consumation of each passenger would then be $\mu \pm 2\sigma$. 95,4% of the cycles would have consumed time within this interval. In other words: One passenger using the elevator once is 95,4% likely to spend this time in queue and elevator combined.

Similar values can be calculated for floors (e.g. queue length, average waiting time) and the elevators (e.g. utilization of max speed)

Chapter 14

Elevator mechanics

To calculate the position of the elevators at a given time and the time of arrival at a given place, we have to simulate the mechanical behaviour of the elevator. To simplify things we have decided that the elevators of our system will move at constant acceleration. This leaves the following characteristics for the elevator:

1. Maximum velocity going up
2. Maximum velocity going down
3. Constant acceleration going up
4. Constant decceleration going up
5. Constant acceleration going down
6. Constant decceleration going down

Since the elevators move at constant acceleration, the elevator can be in 4 different states:

1. Elevator is accelerating
2. Elevator is moving at maximum speed
3. Elevator is braking (decccelerating)
4. Elevator is waiting at a floor

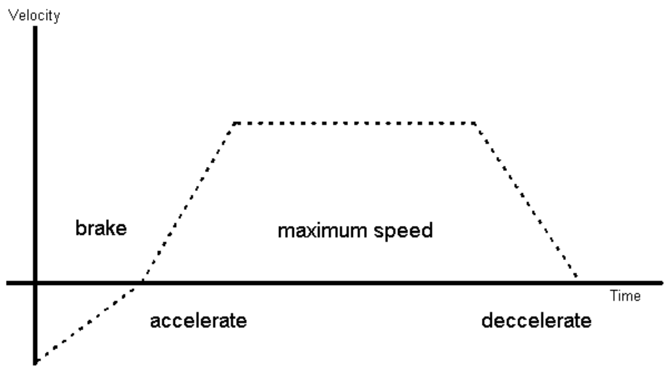


Figure 14.1: the four states of the elevator

In the following equations, these formulars for movement with constant acceleration known from physics are used:

- $V = V_0 + a \cdot t$
- $X - X_0 = V_0 \cdot t + \frac{1}{2} \cdot a \cdot t^2$
- $V^2 = V_0^2 + 2 \cdot a \cdot (X - X_0)$

The following notation will be used

- S_{index} = Position after [index]
- T_{index} = Time needed to perform this part of the movement (Δ time)
- V_{index} = Velocity after [index]

The time needed to travel from one position to another is given by

$$T_{total} = T_{brake} + T_{acceleration} + T_{maximum\_speed} + T_{deccelaration} \tag{14.1}$$

if the elevator is moving in the opposite direction of the destination, or if the elevator has a V_{start} so high that it cannot stop at the destination, thus it has to go past it and afterwards travel back to the destination, then it has to brake.
To calculate the time needed to stop the elevator and the position after it has stopped we use (Speed after braking $= 0$)

$$\begin{gathered} V = V_0 + a \cdot t, V = 0 \Rightarrow t = -\frac{V_0}{a} \\ X - X_0 = V_0 \cdot t + \frac{1}{2} \cdot a \cdot t^2 \Rightarrow X = X_0 + V_0 \cdot t + \frac{1}{2} \cdot a \cdot t^2 \end{gathered} \tag{14.2}$$

If the elevator is moving in the direction of the destination, we have to calculate if the elevator will be able to stop at the destination, or if it has to go past it, and afterwards travel back to the destination. This is calculated using:

$$V^2 = V_0^2 + 2 \cdot a \cdot (X - X_0) \tag{14.3}$$

Using this formula we can calculate the velocity as a function of time:

$$Velocity = V_{start}^2 + 2 \cdot decceleration \cdot (S - S_{start}) \tag{14.4}$$

All put together this gives:

```
if ( S_destination > S_start ) {
   ( if V_start < 0 ) brake; }
   else {
     if ( (V_start^2 +
        2 * Updecceleration * ( S_brake - S_start ))
        > 0 ) brake; }
else {
  if ( V_start > 0 ) brake; }
  else {
     if ( (V_start^2 +
     2 * Downdecceleration * ( S_brake - S_start ))
     < 0 ) brake; }
}
```

To calculate at which position the elevator has to start deccelerating to stop at the destination, we use formula 14.4:

$$\begin{aligned} Velocity &= V_{maximum\_speed}^2 + 2 \cdot decceleration \cdot (S - S_{maximum\_speed}) \\ V_{maximum\_speed} &= \sqrt{V_{brake}^2 + 2 \cdot acceleration \cdot (S_{maximum\_speed} - S_{brake})} \end{aligned} \tag{14.5}$$

Combined this gives ($V_{destination} = 0$):

$$\begin{aligned} 0 =& (V_{brake}^2 + 2 \cdot acceleration \cdot (S_{maximum\_speed} - S_{brake})) \\ &+ 2 \cdot decceleration \cdot (S_{decceleration} - S_{maximum\_speed}) \Rightarrow \\ 0 =& V_{start}^2 + 2 \cdot acceleration \cdot S_{maximum\_speed} - 2 \cdot acceleration \cdot S_{brake} \\ &+ 2 \cdot decceleration - 2 \cdot decceleration \cdot S_{brake} \end{aligned}$$

Isolating $S_{maximum\_speed}$ this gives:

$$S_{maximum\_speed} = \frac{\frac{1}{2} \cdot V_{brake}^2 + decceleration \cdot S_{decceleration} - acceleration \cdot S_{brake}}{decceleration - acceleration} \tag{14.6}$$

This is only true if $V_{maximum\_speed}$ is below or equal to the elevator's maximal velocity. If not then we have the following equation:

$$S_{maximum\_speed} = \frac{V_{maximal\_velocity}^2}{2 \cdot decceleration} + S_{decceleration} \tag{14.7}$$

The time needed to deccelerate is calculated using the same methods as in formula 14.2

$$T_{decceleration} = -\frac{V_{maximum\_speed}}{decceleration} \tag{14.8}$$

Now the time needed to accelerate and the position after acceleration can easily be calculated by using the same procedure as in 14.2:

$$\begin{aligned} T_{acceleration} &= \frac{V_{maximum\_speed} - V_{brake}}{acceleration} \\ S_{acceleration} &= V_{brake} \cdot T_{acceleration} + \frac{1}{2} \cdot T_{acceleration}^2 + S_{brake} \end{aligned} \tag{14.9}$$

Likewise the time moving at maximum speed is:

$$T_{maximum\_speed} = \frac{S_{maximum\_speed} - S_{acceleration}}{V_{maximum\_speed}} \tag{14.10}$$

Finally the time of arrival of the elevator is

$$T_{destination} = T_{start} + T_{brake} + T_{acceleration} + T_{maximum\_speed} + T_{decceleration} \tag{14.11}$$

Chapter 15

Control System evaluation

15.1 General Description

The control system contains two different types of destinations, movable and static destinations. The static destinations are the ones coming from the passengers inside an elevator, therefore they are connected to a certain elevator. The movable destinations, on the other hand, are coming from passengers waiting on the floors, and are thus not connected to a specific elevator.
The static destinations contain a time of insertion and a destination floor, while the movable destinations contain a time of insertion, starting floor and a direction, which indicates in which direction the passenger wants to go (up or down).
Because of this, the internal structure of the control system is divided into a list of movable destinations, and a list for each elevator for the static destinations. The destinations are not sorted in any way in the static lists. In the list of movable destinations the destinations are sorted in the way that the eldest destination is first in the list and the youngest destination is last in the list.
To add or remove destinations from the lists, the control system can be invoked by three actions. These are:

1. AddMovableDestination. This action inserts a new destination into the list of movable destinations. A destination with the same destination floor and direction cannot be added twice in the list.

2. AddStaticDestinations. This action inserts new destinations into the list belonging

to the elevator which the action is sent from. Destinations that already exists in the list will not be added.

3. DeleteDestination. This action deletes a static destination in a certain elevator. If there is a movable destination at the same floor with the right direction it is also deleted, because the waiting passengers on that floor now can walk into the elevator and the movable destination in the list is no longer needed. (see the direction part).

Each time a new destination is added, or a destination is deleted (when an elevator arrives), the control system recalculates the best order of the destinations. The control system has a sorted list of destinations for each elevator for this purpouse. First the sorted list is flushed. The static destinations get sorted and put in the sorted list (for all elevators). Then the movable destinations get distributed between the sorted lists of destinations for all elevators. This sorting is done in a way that gives the lowest time-value, and follows certain guidelines which are:

1. An elevator carrying passengers cannot pass by a floor if any of its passengers wants to get off at that floor.

2. An elevator carrying passengers cannot change its direction if there's any passengers left wanting to go in that direction.

These guidelines are included because passengers probably would find it very strange and stressing if the elevator didn't follow them, but also because they are greatly limiting the number of possible combinations to test. In the sorting of the static destinations these guidelines limit the sorting to only one combination. The sorting algorithm uses the current elevator position and current direction for the sorting.

15.1.1 Time-Value

The time-value is a sum of two values each weighted by a factor that is adjustable for the user. The idea behind our time-value algorithm is to minimize the total expected squared waiting time for each destination, both movable and static. The reason why the time is squared for each destination is that passengers that have waited a long time must be given higher priority ([8]). The first adjustable factor represents the importance of the movable destinations (the passengers waiting for an elevator), while the second adjustable factor represents the importance of the static destinations (the passengers travelling inside the elevators):

$$TimeValue = F1 \cdot TotalSquaredWaitingTime + F2 \cdot TotalSquaredTravelTime \tag{15.1}$$

where F1 and F2 are the two factors, and:

$$\begin{aligned} TotalSquaredWaitingTime &= \\ &\sum_{i=1}^{j}(M(i) \cdot ((CurrentTime - ActionTime(i)) + CalculatedTravelTime(i))^2) \\ TotalSquaredTravelTime &= \\ &\sum_{i=1}^{j}(S(i) \cdot ((CurrentTime - ActionTime(i)) + CalculatedTravelTime(i))^2) \end{aligned} \tag{15.2}$$

Where:
j = number of sorted destinations
M(i) = 1 if i points to a movable destination in the sorted destination list, otherwise M(i) = 0
S(i) = 1 if i points to a static destination in the sorted destination list, otherwise S(i) = 0
CurrentTime = the current time of the evaluation
ActionTime(i) = the time when the i'th destination was invoked

$$\begin{aligned} CalculatedTravelTime(i) = \\ DirectTravelTime(fromPositionAfterBraking(CurrentPosition)toPosition(1)) + \\ \sum_{k=1}^{i-1}(AverageWaitingTime + DirectTravelTime(fromPosition(i)toPosition(i+1))) \end{aligned} \tag{15.3}$$

CalculatedTravelTime(i) is the total expected travel time from the current elevator position to the i'th destination. All destinations and stops between these two positions are also counting and each stop adds an average waiting time. DirectTravelTime and PositionAfterBraking are methods, which use elevator mechanics for calculation (14)

15.1.2 Complexity

When the destinations from the movable list are distributed between the sorted destination lists, each movable destination can be inserted in one of two possible ways for each sorted destination list because of the restrictions given above and the fact that the sorted lists of destinations now holds static destinations that are sorted. The first of the possible insertion points in a sorted destination list is the point in the list, where the movable destination lies between two other destinations and has the right direction between those two. A movable destination for example with destination floor 5 and direction down could be inserted between 6 and 2, but not between 2 and 6 or between 3 and 2. It is not always possible to find such insertion point. The second possible insertion point in a sorted destination list is always present and is at the end of the list.

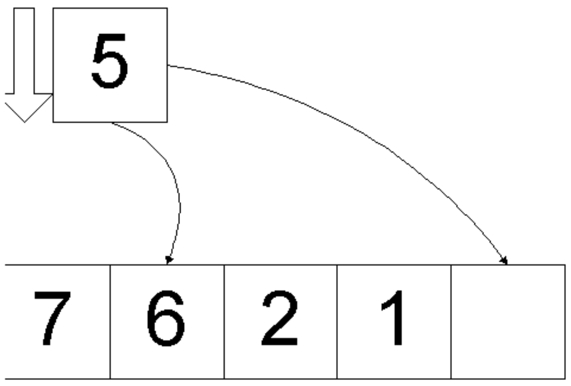


Figure 15.1: the two possible insertion points for a movable destination in a sorted list of destinations

If we should try to insert the movable destinations in all possible ways it would give a total of $(2 \cdot m)^n$ combinations, where m is the number of elevators and n is the number of movable destinations. A system with 10 floors and 10 movable destinations would give 20^{10} combinations (approx. $1.024 \cdot 10^{13}$), which unfortunately is too many combinations to be tried out in realtime. Since the perfect solution requires too many calculations we have to use another approach. One approach would be to select the eldest moveable destination and locate the best elevator and position in the temporary elevator destination list for it, and then repeat this procedure with the next movable destination until all movable destinations are inserted. With this approach the complexity is reduced to $2 \cdot n \cdot m$ combinations. This of course is not the perfect solution, but it is a pretty good approximation

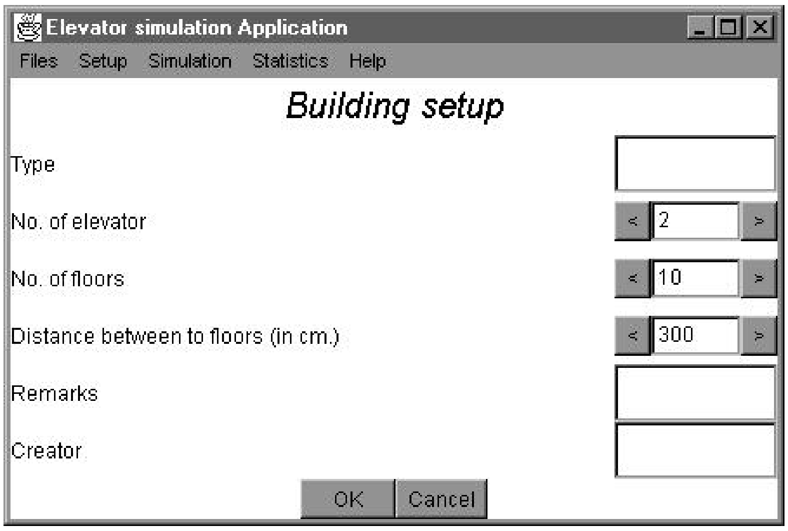
of it.
To locate the best elevator and position for the movable destination in the sorted lists of destinations we compare the change in time-values for the $2 \cdot m$ possible insertion points and choose a point that gives the lowest change of time-value. The movable destination is then inserted in the correct elevator's sorted list at the correct position and do the same with the next movable destination. This process is repeated until there is no more movable destinations to insert.

15.1.3 Directions

When the temporary elevator destination lists have been sorted the top destination from each of the lists must be sent to each corresponding elevator, if they are changed since last evaluation. Here the control system also gives each elevator a next direction. This next direction is sent with the new destination (the new direction replaces the direction of the new destination when sent). This next direction is used by the passengers to know in which direction the elevator will go next. It is also the direction that is returned form the protocol when deleting a destination. This way the delete method knows if or which movable destination to delete. The next direction is calculated from the static destinations in the sorted destination list. If the number of static destinations is less than 2, the next direction will equal NODIRECTION, else it will be either UPDIRECTION or DOWNDIRECTION depending on the order of the two top destinations.
The control system also calculates a current direction for each elevator. This direction is calculated by comparing the elevator's current position to the top static destination in the list of sorted destinations. If there's no static destinations the direction will be NODIRECTION. This current direction is used when sorting the static destinations the next time that the elevator system is called.

Chapter 16

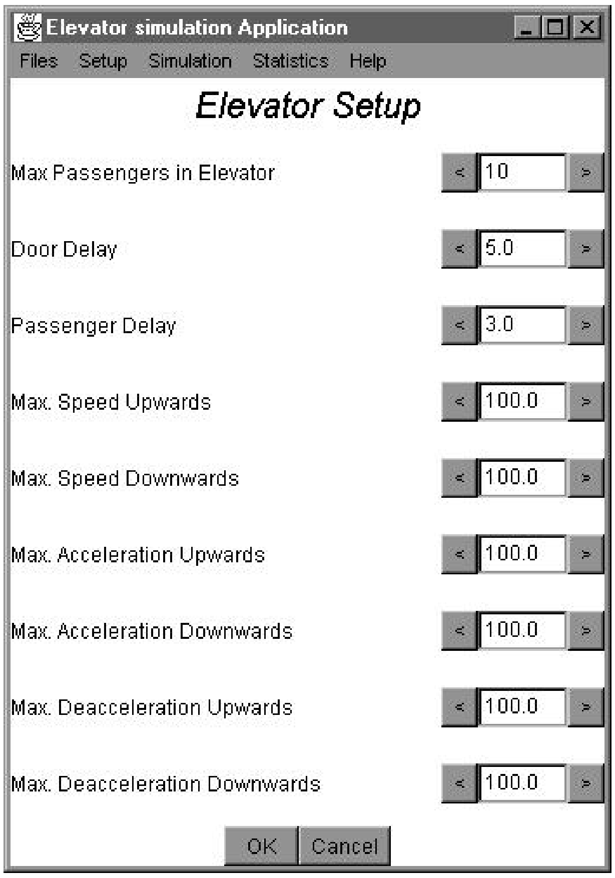
Screendumps


Elevator simulation Application
Files Setup Simulation Statistics Help
OK
Cancel

Building setup

Type	
No. of elevator	2
No. of floors	10
Distance between to floors (in cm.)	300
Remarks	
Creator	

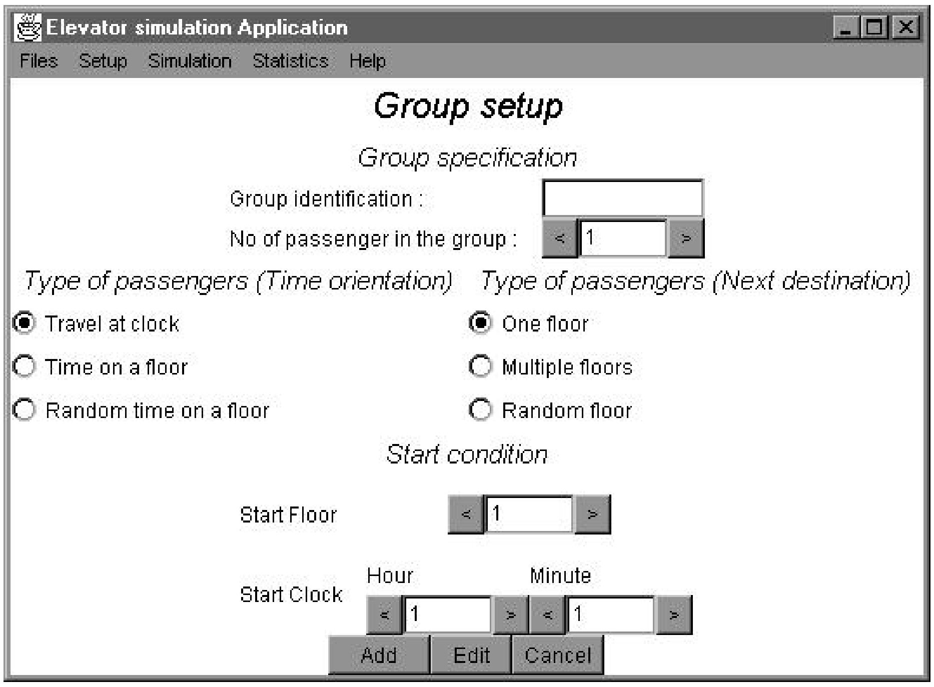
Figure 16.1: The Building setup window.



Elevator Setup

Max Passengers in Elevator	10
Door Delay	5.0
Passenger Delay	3.0
Max. Speed Upwards	100.0
Max. Speed Downwards	100.0
Max. Acceleration Upwards	100.0
Max. Acceleration Downwards	100.0
Max. Deacceleration Upwards	100.0
Max. Deacceleration Downwards	100.0

Figure 16.2: The Elevator setup window.


Group setup

Group specification

Group identification :

No of passenger in the group : 1

Type of passengers (Time orientation)

[✓ Travel at clock]

Time on a floor

Random time on a floor

Type of passengers (Next destination)

[✓ One floor]

Multiple floors

Random floor

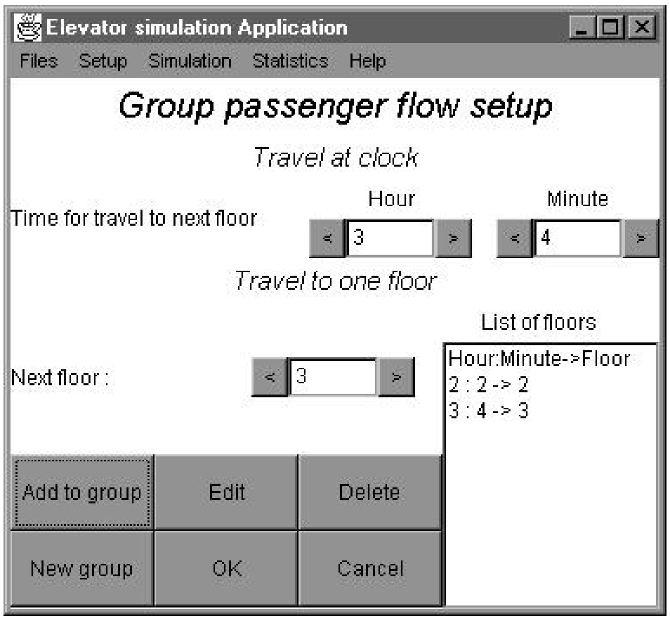
Start condition

Start Floor 1

Start Clock Hour 1 Minute 1

Elevator simulation Application
Files Setup Simulation Statistics Help
Add
Edit
Cancel

Figure 16.3: The group setup window.


Group passenger flow setup

Travel at clock

Time for travel to next floor Hour 3 Minute 4

Travel to one floor

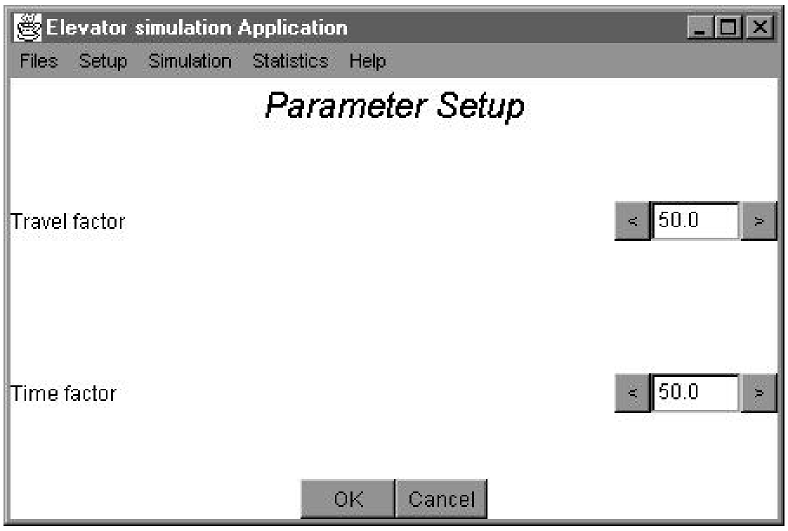
Next floor : 3

List of floors

Hour:Minute->Floor
2 : 2 -> 2
3 : 4 -> 3

Elevator simulation Application
Files Setup Simulation Statistics Help
Add to group
Edit
Delete
New group
OK
Cancel

Figure 16.4: The Passenger group flow setup window.

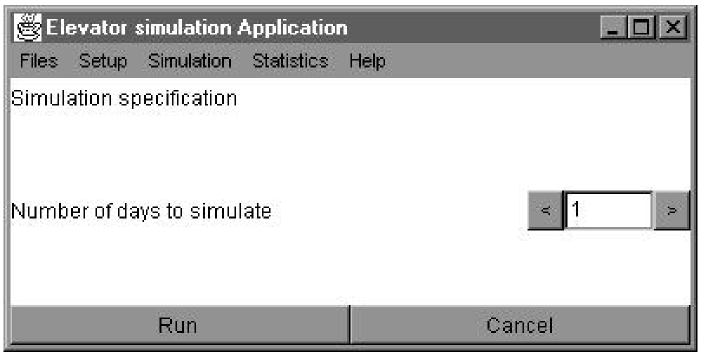


Parameter Setup

Travel factor 50.0

Time factor 50.0

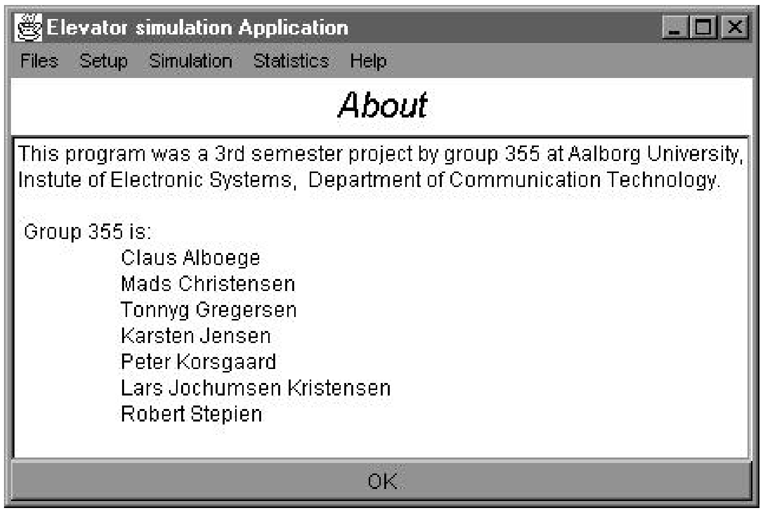
Figure 16.5: The Parameter setup window.



Simulation specification

Number of days to simulate 1

Figure 16.6: The start simulation window.


Elevator simulation Application
Files Setup Simulation Statistics Help
About
OK

This program was a 3rd semester project by group 355 at Aalborg University, Instute of Electronic Systems, Department of Communication Technology.

Group 355 is:

Claus Alboege
Mads Christensen
Tonnyg Gregersen
Karsten Jensen
Peter Korsgaard
Lars Jochumsen Kristensen
Robert Stepien

Figure 16.7: The about window.

Chapter 17

Abstracts from Sourcecode

The following sections in this appendix contains abstracts from the full sourcecode for the program. Each group member is beeing help responsible the piece of sourcecode marked with their name.

17.1 Tonny Gregersen

Part of the method actionPerformed in actionListener:

```
public void actionPerformed(ActionEvent action){
  ..
  ..
    case viewObject.SAVE:
      viewObject.clearMenuSetupItem();
      FileDialog saveDialog = new FileDialog(viewObject,
                                "Save setup");

      saveDialog.setFile(viewObject.FileName);

      saveDialog.setMode(FileDialog.SAVE);
      saveDialog.show();

      if (saveDialog.getFile() != null){
      viewObject.FileName = saveDialog.getFile();
      writeObjectFileObject = new WriteObjectFile();
      try{
         writeObjectFileObject.openWriteObjectFile(
                                                   viewObject.FileName);
         writeObjectFileObject.writeObjectToLog(
                                                   viewObject.DataVar);
      }finally{
         writeObjectFileObject.closeWriteObjectFile();
      }
      }
      break;
   . .
   . .
}
```

The **class** WriteObjectFile:

```
import java.io.*;
import actionListener;
import DataContainer;
import InitMenu;

/*****************************************************
 * The WriteObjectFile class controls the storing of the
 * configuration objects.
 *****************************************************/
class WriteObjectFile{

  /***************************************************
   * Sets FileObject as an instance of File
   ***************************************************/
  private File FileObject;

  /***************************************************
   * Sets FileOutput as an instance of FileOutputStream
   ***************************************************/
  private FileOutputStream FileOutput;

  /***************************************************
 * Sets Out as an instance of ObjectOutputStream
 ****************************************************/
private ObjectOutputStream Out;

/****************************************************
 * The name of the file that is to be stored
 ****************************************************/
private String FileName;

/****************************************************
 * The constructor of WriteObjectFile
 ****************************************************/
public WriteObjectFile(){
}

/****************************************************
 * Method openWriteObjectFile opens a given file for
 *  output to a specified stream
 ****************************************************/
public void openWriteObjectFile(String FileName){
  this.FileName = FileName;
      try{
        FileObject = new File(FileName);
        FileOutput = new FileOutputStream(FileObject);
        Out = new ObjectOutputStream(FileOutput);
      }catch(IOException e){
        System.out.println(" (WriteObjectFile
          IOException): Unable to create " + FileName);
      }
}//End Method openWriteObjectFile

/****************************************************
 * Method writeObjectToLog writes a given object to a
 * specified stream
 ****************************************************/
public void writeObjectToLog(Object LogThisObject){
      try{
        Out.writeObject(LogThisObject);
      }catch(IOException e){
        System.out.println(" (WriteObjectFile
            IOException): Unable to write to "+ FileName);
        }
    }//End Method writeObjectToLog

    /*****************************************************
     * Method closeWriteObjectFile closes the the output file
     *****************************************************/
    public void closeWriteObjectFile(){
        try{
            Out.close();
        }catch(IOException e){
            System.out.println(" (WriteObjectFile
              IOException): Unable to close " + FileName);
        }
    }//End Method closeWriteObjectFile

}//END CLASS WriteObjectFile
```

The Container Class:

The **class** Container

```
import java.lang.*;
import java.util.*;
/*****************************************************
 *  Super class of the following classes : Elavator, Queue,
 *  Pool.

 *****************************************************/
abstract class Container extends  Observable{

    protected int ContainerID;
    protected int Command;
    protected Container NextContainer = null;
    protected Passenger ListOfPassengers = null;
    protected int NoOfPassengers = 0;
    protected Passenger LastPassenger = null;
    private Passenger NewPassengerToAdd;
```

```
  protected Passenger SeekPassenger;

  /****************************************************
   * Instantiates (Constructor) a new Container Object
   * which gets a ContainerID
   * param SetContainerID The container identification
   * number.
*/
  public Container(int SetContainerID){
    ContainerID = SetContainerID;
  } // Constructor

  public Container() {

  } // Constructor

  /****************************************************
   * Returns the number of passengers in the container.
   * return  The number of passengers in the container.
*/
  public int getNoOfPassengers() {
    return NoOfPassengers;
  }

  /****************************************************
   * Returns the command to be executed by the observer.
   * return  The type of operation.
   ****************************************************
  public int getCommand(){
    return Command;
  }

/****************************************************
   * Instantiates a new command object with a command.
   * param NewCommand New command.
*/
  public void setCommand(int NewCommand) {
    Command = NewCommand;
  }
  /*****************************************************
   * Method to check if a container is empty.
   * return Wether the container is empty.
*/
  public boolean isEmpty() {
    if (NoOfPassengers==0) return true;
    else return false;
  }

  /*****************************************************
   * Returns the passengers in the container in an array.
   * return Array of passengers.
*/
  public Passenger[ ] getPassengerArray() {
    Passenger[ ] PassengerArray = new Passenger[
                                        NoOfPassengers];
    SeekPassenger = ListOfPassengers;
    for (int i = 0; i < NoOfPassengers; i++) {
      PassengerArray[i] = SeekPassenger;
      SeekPassenger = SeekPassenger.getNextPassenger();
    }
    return PassengerArray;
  } // getPassengerArray

  /*****************************************************
   * Returns the ID of the container.
   * return  The container ID.
   ****************************************************
  public int getContainerID() {
    return ContainerID;
// getContainerID

  /*****************************************************
   * Get the first passenger of the container.
   * return  The first passenger in the container.
*/
  public Passenger getFirstPassenger() {
    return ListOfPassengers;
  } // getFirst
  /**********************************************************
   * Add a passenger to the list of passengers.
   * param AddPassenger Which passenger to add.
*/
  public void addPassenger(Passenger AddPassenger){

    try {
      NewPassengerToAdd = (Passenger)AddPassenger.clone();
    }
    catch (CloneNotSupportedException e) {
      System.exit(0);
    }

    if(ListOfPassengers == null)
      ListOfPassengers = NewPassengerToAdd;
    else{
      SeekPassenger = ListOfPassengers;

    while(SeekPassenger.getNextPassenger() != null){
      SeekPassenger = SeekPassenger.getNextPassenger();
    }

    SeekPassenger.setNextPassenger(NewPassengerToAdd);
    }
    NoOfPassengers++;
  } // addPassenger

  /**********************************************************
   * Sets the reference to the next container in the
   * curcuit.
   * param SetNextContainer Which container to set as next
   * container.
*/
  public void setNextContainer(Container SetNextContainer){
    this.NextContainer = SetNextContainer;
  }

  /**********************************************************
   * Returns the next container.
   * return Next container.
```

```
*/
 public Container getNextContainer(){
   return NextContainer;
 }

 /*****************************************************
  * Deletes a passenger and decrease number of passengers
  * by 1.
  * param DeletePassenger    The passenger that should be
  * deleted from a container.
  * param TestPassenger       List control object, e.g.
  * ListOfPassengers.
  *****************************************************/
 protected void deletePassenger(Passenger DeletePassenger,
                                Passenger TestPassenger) {

   SeekPassenger = ListOfPassengers;

   if(SeekPassenger.getPassengerID() ==
      DeletePassenger.getPassengerID()){
     ListOfPassengers = null;
     ListOfPassengers = SeekPassenger.getNextPassenger();
     NoOfPassengers−−;
   }
   else{

   while((SeekPassenger.getNextPassenger() != null) &&
         (DeletePassenger.getPassengerID() !=
          SeekPassenger.getNextPassenger().
            getPassengerID()))
     SeekPassenger = SeekPassenger.getNextPassenger();

   if(DeletePassenger.getPassengerID() ==
      SeekPassenger.getNextPassenger().getPassengerID()){
     if(SeekPassenger.getNextPassenger().
          getNextPassenger()
        != null){
        Passenger DelPassenger = SeekPassenger.
          getNextPassenger();
        SeekPassenger.setNextPassenger(SeekPassenger.
          getNextPassenger().getNextPassenger());
        DelPassenger.setNextPassenger(null);
        DelPassenger = null;
      }
      else{
        SeekPassenger.setNextPassenger(null);
        ListOfPassengers = SeekPassenger;
      }
      NoOfPassengers−−;
    }
    else
      System.exit(0);
    }
  } // deletePassenger


  /****************************************************
   * Move a given passenger to the next container. The
   * NextContainer must be set!
   * param PassengerToMove  The passenger to move.
*/
  public void movePassenger(Passenger PassengerToMove) {

    NextContainer.addPassenger(PassengerToMove);
    deletePassenger(PassengerToMove, ListOfPassengers);

  } // movePassenger
} // Container
```

17.2 Karsten Jensen

```
import java.awt.*;
import java.awt.event.*;
import java.lang.*;

class SetupSpinTextDouble{
  final public static int BDEC = 1;
  final public static int BINC = 2;
  final public static int TEXT = 3;

  private SetupSpinTextDoubleActionListener BDec = new
SetupSpinTextDoubleActionListener (this,BDEC);
  private SetupSpinTextDoubleActionListener BInc = new
SetupSpinTextDoubleActionListener (this,BINC);
  private SetupSpinTextDoubleActionListener Text = new
SetupSpinTextDoubleActionListener (this,TEXT);
  private SetupSpinTextDoubleFocusListener FocusText = new
SetupSpinTextDoubleFocusListener (this,TEXT);


  private Panel SetupSpinTextDoublePanel;
  private GridBagLayout gridbag;
  private GridBagConstraints c;
  private TextField eText = new TextField ("",4);
  private Button eDec = new Button("<");
  private Button eInc = new Button(">");
  private Double Number;
  private double Min;
  private double Max;
  private double Step;

  public SetupSpinTextDouble(Panel SetupSpinTextDoublePanel,
double InitNumber, double Min, double Max, double Step){
  this.SetupSpinTextDoublePanel = SetupSpinTextDoublePanel;
  this.Min = Min;
  this.Max = Max;
  this.Step = Step;

  gridbag = new GridBagLayout();
  c = new GridBagConstraints();
  SetupSpinTextDoublePanel.setLayout(gridbag);
  c.fill = GridBagConstraints.BOTH;
  SetupSpinTextDoublePanel.add(eDec);
  SetupSpinTextDoublePanel.add(eText);
  SetupSpinTextDoublePanel.add(eInc);
  setNumber(InitNumber);
```

```
eDec.addActionListener (BDec);
eInc.addActionListener (BInc);
eText.addActionListener (Text);
eText.addFocusListener (FocusText);

}

void setNumber(double intNumber){
   Number = new Double(intNumber);
   eText.setText(Number.toString());
}

void setNumber(String NumberString){
   Number = new Double(NumberString);
   eText.setText(Number.toString());
}

Button getIncButton(){
   return eInc;
}

Button getDecButton(){
   return eDec;
}

TextField getTextField(){
   return eText;
}

double getNumber(){
   return Number.doubleValue();
}

String getNumberInString(){
   return Number.toString();
}

public double getMin(){
   return Min;
```

```
  }

  public double getMax(){
    return Max;
  }


  public double getStep(){
    return Step;
  }
}


class SetupSpinTextDoubleActionListener implements
ActionListener{

  private SetupSpinTextDouble viewObject;
  private int command;
  private double Number;
  private Double INumber;
  private String NumberString;

  public
SetupSpinTextDoubleActionListener(SetupSpinTextDouble
viewObject, int listening_command){
    this.viewObject = viewObject;
    command = listening_command;
  }


  public void actionPerformed(ActionEvent action){
    switch(command){


   case viewObject.BDEC :
      Number = viewObject.getNumber();
      if(Number > viewObject.getMin())
        Number = Number - viewObject.getStep();
      viewObject.setNumber(Number);
      break;
      case viewObject.BINC :
        Number = viewObject.getNumber();
        if(Number < viewObject.getMax())
          Number = Number + viewObject.getStep();
        viewObject.setNumber(Number);
        break;

      case viewObject.TEXT :
        if((viewObject.getMin() <= viewObject.getNumber()) &&
(viewObject.getMax() >= viewObject.getNumber()))
          Number = viewObject.getNumber();
        break;
      }
    }
}


class SetupSpinTextDoubleFocusListener implements
FocusListener{

    private SetupSpinTextDouble viewObject;
    private int command;
    private double Number;
    private Double INumber;
    private String NumberString;

    public
SetupSpinTextDoubleFocusListener(SetupSpinTextDouble
viewObject, int listening_command){
      this.viewObject = viewObject;
      command = listening_command;
    }

    public void focusLost(FocusEvent focuslost){
      switch(command){

      case viewObject.TEXT :
        try{
        NumberString = viewObject.getTextField().getText();
```

```
          if((viewObject.getMin() <=
(INumber.valueOf(NumberString)).doubleValue()) &&
(viewObject.getMax() >=
(INumber.valueOf(NumberString)).doubleValue()))
             viewObject.setNumber(NumberString);
        else
           {viewObject.setNumber(viewObject.getNumber());}
          }catch(NumberFormatException
e){viewObject.setNumber(viewObject.getNumber());}
        break;

      }

   }

   public void focusGained(FocusEvent focusgained){

      switch(command){
      }
   }

}


/********************************************************
********************************************************/


import java.util.*;

public class Passenger implements Cloneable {

   private int PassengerID;
   private int GroupID;
   private int Destination = -1;
   private int TravelState;
   private int Command;
   private double ActionTime;
   private double DepartureTime;
   private Passenger NextPassenger = null;
```

```
  private Movement ListOfPassengerMovement = new
Movement(−3,−1);
  private Movement LastMovement = null;
  private int NoOfMovements = 0;

  public Passenger(int PassengerID, int GroupID){
    this.PassengerID = PassengerID;
    this.GroupID = GroupID;
    TravelState = 0;
  }


  public void setTravelState(int New) {
    TravelState = New;
  }


  public void logPassengerMovement(double ActualTime, int
ContainerID){
    if (ListOfPassengerMovement.getNextMovement()  == null){
        ListOfPassengerMovement.setNextMovement(new
Movement(ActualTime, ContainerID));
        LastMovement =
ListOfPassengerMovement.getNextMovement();
    }
    else {
      LastMovement.setNextMovement(new
Movement(ActualTime,ContainerID));
      LastMovement = LastMovement.getNextMovement();
    }
    NoOfMovements++;
    LastMovement.setNextMovement(null);
  }

  public Object clone() throws CloneNotSupportedException{
    return super.clone();
  }

  public void setNewTime(double DepartureTime){
    TravelState++;
  this.DepartureTime = DepartureTime;
}

public void setNewDestination(int Destination){
  this.Destination = Destination;
}

public void setPassengerID(int PassengerID){
  this.PassengerID = PassengerID;
}

public int getPassengerID(){
  return PassengerID;
}

public int getNoOfMovements() {
  return NoOfMovements;
}

public void setGroupID(int GroupID){
  this.GroupID = GroupID;
}

public int getGroupID(){
  return GroupID;
}

public int getDestination(){
  return Destination;
}

public Movement[] getMovementArray() {
  Movement[] MovementArray = new Movement[NoOfMovements];
  Movement TempMovement = ListOfPassengerMovement;
  for (int i = 0; i < NoOfMovements; i++) {
    MovementArray[i] = TempMovement.getNextMovement();
    TempMovement = TempMovement.getNextMovement();
  }
  return MovementArray;
}
```

```
public int getTravelState(){
  return TravelState;
}


public int getCommand() {

  return Command;
}


public double getActionTime() {
  return ActionTime;
}

  public void setActionTime(double NewActionTime) {
  ActionTime = NewActionTime;
}

  public double getDepartureTime(){
  return DepartureTime;
}


public void setNextPassenger(Passenger NextPassenger){
  this.NextPassenger = NextPassenger;
}


public Passenger getNextPassenger(){
  return NextPassenger;
}


public void setPassengerState(){

}
```

```
}
```

17.3 Mads Græsbøll Christensen

```
/******************************************************
 * Class        : Elevator
 ******************************************************/
class Elevator extends Container {

  private int NextDirection = UNDECIDED;
  private int Destination = 1;
  private int MaxNoOfPassengers = 10;

  private int[] UnprocessedDestination;
  private Building BuildingRef;
  private boolean[] ButtonPressed;

  private Movement ListOfElevatorMovement =
    new Movement(−1,−1);

  private Movement LastMovement = null;
  private int NoOfMovements = 0;

  private double ActionTime = 0;

  private double DoorDelay = 5;
  private double PassengerDelay = 3;
private SystemManagerObserverForElevator
   SystemManagerObserverForElevatorObject;

/*****************************************************
 * Elevator constructor.
 *****************************************************/
public Elevator(int SetElevatorID,
                int SetMaxNoOfPassengers,
                double SetPassengerDelay,
                double SetDoorDelay,
                Building Ref,
                SystemManagerObserverForElevator
                SystemManagerObserverForElevatorObject) {

   addObserver(SystemManagerObserverForElevatorObject);

   ContainerID = SetElevatorID;
   BuildingRef = Ref;

   MaxNoOfPassengers = SetMaxNoOfPassengers;
   PassengerDelay = SetPassengerDelay;
   DoorDelay = SetPassengerDelay;

   ButtonPressed = new
      boolean[BuildingRef.getNoOfFloors()];

   UnprocessedDestination = new int[MaxNoOfPassengers];

   for (int i=0; i < BuildingRef.getNoOfFloors(); i++)
      ButtonPressed[i] = false;
   for (int i=0; i < MaxNoOfPassengers; i++)
      UnprocessedDestination[i] = 0;

} // Constructor


/*****************************************************
 * Log elevator arrival.
 *****************************************************/
public void logElevatorMovement(double ActualTime,
                                int Floor) {

  if (ListOfElevatorMovement.getNextMovement()  == null){
    ListOfElevatorMovement.
      setNextMovement(new Movement(ActualTime, Floor));

    LastMovement =
      ListOfElevatorMovement.getNextMovement();
  }
  else {
    LastMovement.setNextMovement
      (new Movement(ActualTime, Floor));

    LastMovement = LastMovement.getNextMovement();
  }
  NoOfMovements++;
  LastMovement.setNextMovement(null);
} // log ElevatorMovement

/****************************************************
 * Unloads and loads passengers.
 ****************************************************/
public void movePassenger(double Time) {


  NextContainer = BuildingRef.getFloor(Destination).
    getPool();

  logElevatorMovement(Time, Destination);

  Passenger Next = null;
  ActionTime = Time + DoorDelay;

  SeekPassenger = ListOfPassengers;

  while(SeekPassenger != null){
    if(SeekPassenger.getDestination() ==
        this.Destination) {

      SeekPassenger.
      logPassengerMovement(Time,ContainerID);

    SeekPassenger.setActionTime(Time);
    ActionTime += PassengerDelay;

    ((Pool)NextContainer).
      getNewDestination(SeekPassenger);

    super.movePassenger(SeekPassenger);
  }
  SeekPassenger = SeekPassenger.getNextPassenger();
}

NextContainer = null;

int NoBefore = NoOfPassengers;
ActionTime += DoorDelay;

Floor CurrentFloor = BuildingRef.getFloor(Destination);
CurrentFloor.getDownQueue().getNoOfPassengers();
CurrentFloor.getDownQueue().getNoOfPassengers();

if (NextDirection == UNDECIDED) {
  if (!CurrentFloor.getUpQueue().isEmpty() &&
      !CurrentFloor.getDownQueue().isEmpty()) {
    if (CurrentFloor.getUpQueue().getFirstPassenger().
        getDepartureTime()<
        CurrentFloor.getDownQueue().getFirstPassenger()
        .getDepartureTime())
      NextDirection = UP;
    else
      NextDirection = DOWN;
  }
  else {
    if (CurrentFloor.getUpQueue().isEmpty() &&
        CurrentFloor.getDownQueue().isEmpty()) {
      setElevatorReadyFlag();
    }
    else {
      if (CurrentFloor.getUpQueue().
          getNoOfPassengers()>
          CurrentFloor.getDownQueue().
          getNoOfPassengers())
        NextDirection = UP;
      else
        NextDirection = DOWN;
    }
  }
}

switch (NextDirection) {
case UP : {
  if(CurrentFloor.getUpQueue().isEmpty())
    setElevatorReadyFlag();
  else{
    CurrentFloor.getUpQueue().
      setNextContainer(((Container)this));
    CurrentFloor.getUpQueue().
      movePassenger(MaxNoOfPassengers−
                     NoOfPassengers, Time);
    CurrentFloor.getUpQueue().setNextContainer(null);
  }
  break;
}
case DOWN: {
  if(CurrentFloor.getDownQueue().isEmpty())
    setElevatorReadyFlag();
  else{
    CurrentFloor.getDownQueue().
      setNextContainer(((Container)this));
    CurrentFloor.getDownQueue().
      movePassenger(MaxNoOfPassengers−
                     NoOfPassengers, Time);
    CurrentFloor.getDownQueue().setNextContainer(null);
  }
  break;
}
}

CurrentFloor = null;
```

```
  ActionTime += (PassengerDelay*(NoOfPassengers−
                                        NoBefore));

  if (NoBefore < NoOfPassengers) {

    int j = 0;
    for (int i=0; i < BuildingRef.getNoOfFloors(); i++) {
      if (ButtonPressed[i]==true) {
        setUnprocessedDestination(j,(i+1));
        j++;
      }
    }

    if (UnprocessedDestination[0] != 0)
      setUnprocessedDestinationFlag();
    for (int i = 0; i < MaxNoOfPassengers; i++)
      UnprocessedDestination[i] = 0;
    for (int i=0; i < BuildingRef.getNoOfFloors(); i++)
      ButtonPressed[i] = false;

  }
} // movePassenger

/****************************************************
 * Returns the log of the elevator's movement.
 ****************************************************/
public Movement[ ] getMovementArray() {
  Movement[ ] MovementArray = new Movement[NoOfMovements];
  Movement TempMovement = ListOfElevatorMovement;
  for (int i = 0; i < NoOfMovements; i++) {
    MovementArray[i] = TempMovement.getNextMovement();
    TempMovement = TempMovement.getNextMovement();
  }
  return MovementArray;
} // getElevatorMovementArray

/****************************************************
 * Notify the System Manager.
 ****************************************************/
public void setElevatorReadyFlag(){
        Command = ELEVATORREADY;
        setChanged();
        notifyObservers(this);
    }

     /****************************************************
        * Notifies the System Manager with UnprocessedDest.
        ****************************************************/
    public void setUnprocessedDestinationFlag(){

        Command = UNPROCESSEDDESTINATION;
        setChanged();
        notifyObservers(this);
    }
} // Elevator

/******************************************************
  * Class        : Container
  ******************************************************/
abstract class Container extends  Observable{

    protected int ContainerID;
    protected int Command;
    protected Container NextContainer = null;
    protected Passenger ListOfPassengers = null;
    protected int NoOfPassengers = 0;
    protected Passenger LastPassenger = null;
    private Passenger NewPassengerToAdd;
    protected Passenger SeekPassenger;

    /****************************************************
      * Add a passenger to the list of passengers.
      ****************************************************/
        public void addPassenger(Passenger AddPassenger){

            try {
            NewPassengerToAdd = (Passenger)AddPassenger.clone();
            }
            catch (CloneNotSupportedException e) {
```

```
        System.exit(0);
      }

      if(ListOfPassengers == null)
        ListOfPassengers = NewPassengerToAdd;
      else{
        SeekPassenger = ListOfPassengers;

        while(SeekPassenger.getNextPassenger() != null){
          SeekPassenger = SeekPassenger.getNextPassenger();
        }

        SeekPassenger.setNextPassenger(NewPassengerToAdd);
      }
      NoOfPassengers++;
    } // addPassenger
} // Container
```

17.4 Robert Stepien

```
/*****************************************************
 * Evaluates the Destinations. Calls the calcNewDirections
 * to find the new elevator directions. Calls
 * calcNewDestination in the ControlSystemOutputs to find
 * the newDestination for each elevator. The protocol is
 * not notifyed here.
 *****************************************************/
 private void evaluate(ElevatorInfo[] ElevatorInformation){
  // first sort the static destinations and calculate the
     new directions

  for (int i = 0; (i < NumberOfElevators); i++) {
    ListOfOutputs[i].updateListOfSortedDestinations(
    ElevatorInformation[i]);
    ListOfOutputs[i].calcNewDirections(Elevator
    Information[i]);
  }

  // then insert the movable destinations into the static
```

```
    ones

  int BestInsertionElevatorNumber = 0;
  double LowestDeltaTimeValue;
  double MaybeLowestDeltaTimeValue;

  for (int i = 0; (i < ListOfMovableDestinations.getNumber
  OfDestinations()); i++) {
    BestInsertionElevatorNumber = 0;

    LowestDeltaTimeValue = ListOfOutputs[0].tryMovable
    Destination(ListOfMovableDestinations.getDestination(
    i), ElevatorInformation[0]);

    for (int j = 1; (j < NumberOfElevators); j++) {
      MaybeLowestDeltaTimeValue = ListOfOutputs[j].try
      MovableDestination(ListOfMovableDestinations.get
      Destination(i), ElevatorInformation[j]);
      if (LowestDeltaTimeValue > MaybeLowestDeltaTime
      Value) {
        LowestDeltaTimeValue = MaybeLowestDeltaTimeValue;
        BestInsertionElevatorNumber = j;
      } // lower deltatime?

    } // inner for-loop
    ListOfOutputs[BestInsertionElevatorNumber].addMovable
    Destination(ListOfMovableDestinations.getDestination(
    i), ElevatorInformation[BestInsertionElevatorNumber]);

  } // outer for-loop

  //now we just have to send some dests to the protocol.
  for (int i = 0; i < NumberOfElevators; i++) {
    ListOfOutputs[i].calcNewDestination();
  } // for loop
} // method evaluate

/*******************************************************
* calculates the directions used for updating and for
* calculating the new destination. This method must be
* called before the insertion of the movable destinations
* in the sorted destination list (but after the update of
* the static desitnations)
*****************************************************/
public void calcNewDirections(ElevatorInfo Elevator
Information) {
  // first calculate the current direction
  if (ListOfSortedDestinations.getNumberOf
  Destinations() > 0) {
    if (ElevatorInformation.getPosition() > ListOfSorted
    Destinations.getDestination(0).getDestination
    Floor()) {
      CurrentElevatorDirection =
      Destination.DOWNDIRECTION;
    }
    else {
      CurrentElevatorDirection = Destination.UPDIRECTION;
    }
  }
  else { // if no static destinations in the list then
            give the elevator NODIRECTION
    CurrentElevatorDirection = Destination.NODIRECTION;
  } // end of current direction calculation


  //then calculate the next direction (used by the
    passenger curcuit)
  if (ListOfSortedDestinations.getNumberOf
  Destinations() < 2) { //less than 2 dests => NODIRECTION
    NextElevatorDirection = Destination.NODIRECTION;
  }
  else { // more than 1 destination means DOWN or UP
    if (ListOfSortedDestinations.getDestination(0).get
    DestinationFloor() >
        ListOfSortedDestinations.getDestination(1).get
    DestinationFloor()) {
      NextElevatorDirection = Destination.DOWNDIRECTION;
    }
    else {
      NextElevatorDirection = Destination.UPDIRECTION;
      // no other possibilities becauce we cannot add two
         equal static destinations to the list.
    }
  } // end of next direction calculation
} // method calcNewDirections

/*****************************************************
* changes the NewDestination and LastDestination and tells
* the protocol to get the data.
*****************************************************/
public void calcNewDestination() {

  if (ListOfSortedDestinations.getNumberOf
  Destinations() == 0) { // no more destinations, but the
                            elevator must always have at
                            least 1 destination
    NewDestination = new Destination(0.0, LastDestination
    .getDestinationFloor(), Destination.NODIRECTION);
  }
  else {
    NewDestination = new Destination(0.0, ListOfSorted
    Destinations.getDestination(0).getDestination
    Floor(), NextElevatorDirection);
  }
} // method sendDestToProtocol

/*****************************************************
* returns a possible insertion point for a movable
* destination in the ListOfSortedDestinations (follows the
* given guidelines)
*****************************************************/
private int possibleInsertionPoint(Destination Dest,
ElevatorInfo ElevatorInformation) {

  int ReturnValue = ListOfSortedDestinations.getNumberOf
  Destinations(); //Returns this value if this is the only
                    possible insertion point

  if (ReturnValue>0) {
    if (Dest.getDirection() == Dest.UPDIRECTION) {
```

```
    if ((Math.round(positionAfterBraking(Elevator
    Information)) <= Dest.getDestinationFloor()) &&
        (Dest.getDestinationFloor() <= ListOfSorted
    Destinations.getDestination(0).getDestination
    Floor())) {
      ReturnValue = 0;
    }

    for (int i = 1; i < ListOfSortedDestinations.get
    NumberOfDestinations(); i++) {
      if ((ListOfSortedDestinations.getDestination(i−1).
      getDestinationFloor() <= Dest.getDestination
      Floor()) &&
          (Dest.getDestinationFloor() <= ListOfSorted
      Destinations.getDestination(i).getDestination
      Floor())) {
        ReturnValue = i;
      }
    }
  }

  if (Dest.getDirection() == Dest.DOWNDIRECTION) {
    if ((Math.round(positionAfterBraking(Elevator
    Information)) >= Dest.getDestinationFloor()) &&
        (Dest.getDestinationFloor() >= ListOfSorted
    Destinations.getDestination(0).getDestination
    Floor())) {
      ReturnValue = 0;
    }

    for (int i = 1; i < ListOfSortedDestinations.get
    NumberOfDestinations(); i++) {
      if ((ListOfSortedDestinations.getDestination(i−1).
      getDestinationFloor() >= Dest.getDestination
      Floor()) &&
          (Dest.getDestinationFloor() >= ListOfSorted
      Destinations.getDestination(i).getDestination
      Floor())) {
        ReturnValue = i;
      }
      }
    }
  }
  return ReturnValue;
} //method possibleInsertionPoint

/*****************************************************
* adds the given movable destination to the
* ListOfSortedDestinations in the ControlSystemOutput
* object at the position which gives the lowest TimeValue
* and follows the given guidelines.
*****************************************************/
public void addMovableDestination(Destination Dest,
ElevatorInfo ElevatorInformation) {
  double OldTimeValue = getTimeValue(ElevatorInformation);

  ListOfSortedDestinations.insertDestination(Dest,
  ListOfSortedDestinations.getNumberOfDestinations());

  double LowestDeltaTimeValue = getTimeValue(Elevator
  Information)−OldTimeValue;
  ListOfSortedDestinations.removeDestination(Dest);

  ListOfSortedDestinations.insertDestination(Dest,
  possibleInsertionPoint(Dest, ElevatorInformation));
  double MaybeLowestDeltaTimeValue = getTimeValue(Elevator
  Information)−OldTimeValue;
  if (MaybeLowestDeltaTimeValue > LowestDeltaTimeValue) {
    ListOfSortedDestinations.removeDestination(Dest);
    ListOfSortedDestinations.insertDestination(Dest,
    ListOfSortedDestinations.getNumberOfDestinations());
  }
} // method addMovableDestination

/*****************************************************
* returns the change of the timevalue after trying to
* insert the given movable destination in the
* sorted list in the ControlSystemOutput object at the
* position which gives the lowest change in TimeValue and
* follows the given guidelines.
```

```
*****************************************************/
public double tryMovableDestination(Destination Dest,
ElevatorInfo ElevatorInformation) {
  double OldTimeValue = getTimeValue(ElevatorInformation);

  ListOfSortedDestinations.insertDestination(Dest,
  possibleInsertionPoint(Dest, ElevatorInformation));
  double LowestDeltaTimeValue = getTimeValue(Elevator
  Information)-OldTimeValue;
  ListOfSortedDestinations.removeDestination(Dest);

  ListOfSortedDestinations.insertDestination(Dest,
  ListOfSortedDestinations.getNumberOfDestinations());
  double MaybeLowestDeltaTimeValue = getTimeValue(Elevator
  Information)-OldTimeValue;
  ListOfSortedDestinations.removeDestination(Dest);

  if (MaybeLowestDeltaTimeValue < LowestDeltaTimeValue) {
    LowestDeltaTimeValue = MaybeLowestDeltaTimeValue;
  }

  return LowestDeltaTimeValue;
} // method tryMovableDestination

/*****************************************************
* updates the ListOfSortedDestinations by sorting
* ListOfStaticDestinations in the ControlSystemOutput
* object. The sorting can only be done in one way because
* of the given guidelines. ALERT! The direction in the
* elevatorinformation must indicate the current direction
* of the elevator! (NODIRECTION will work as
* DOWNDIRECTION!)
*****************************************************/
public void updateListOfSortedDestinations(ElevatorInfo
ElevatorInformation) {

  int FloorAfterBraking = (int)Math.round(
  positionAfterBraking(ElevatorInformation));

  ListOfSortedDestinations.flush();
```

```
if (ListOfStaticDestinations.getNumberOf
Destinations()>0) {
  ListOfSortedDestinations.insertDestination(ListOf
  StaticDestinations.getDestination(0),0);
  if (ElevatorInformation.getDirection() ==
  Destination.UPDIRECTION) {//current direction of elev.
    for (int i = 1; i<ListOfStaticDestinations.getNumber
    OfDestinations(); i++) {
      if (ListOfStaticDestinations.getDestination(i).get
      DestinationFloor() >= FloorAfterBraking) {
        int j;
        for (j = 0; ( (ListOfSortedDestinations.get
        NumberOfDestinations() > j) &&
                      (ListOfStaticDestinations.get
        Destination(i).getDestinationFloor() >
                       ListOfSortedDestinations.get
        Destination(j).getDestinationFloor()) &&
                      (ListOfSortedDestinations.get
        Destination(j).getDestinationFloor() >=
                       FloorAfterBraking) ); j++);
        ListOfSortedDestinations.insertDestination(List
        OfStaticDestinations.getDestination(i),j);
      }
      else {
        int j;
        for (j = 0; ( (ListOfSortedDestinations.get
        NumberOfDestinations() > j) &&
                      (ListOfStaticDestinations.get
        Destination(i).getDestinationFloor() <
                       ListOfSortedDestinations.get
        Destination(j).getDestinationFloor()) ); j++);
        ListOfSortedDestinations.insertDestination(List
        OfStaticDestinations.getDestination(i),j);
      }
    }
  }
  else {
    for (int i = 1; i<ListOfStaticDestinations.getNumber
    OfDestinations(); i++) {
      if (ListOfStaticDestinations.getDestination(i).get
            DestinationFloor() <= FloorAfterBraking) {
              int j;
              for (j = 0; ( (ListOfSortedDestinations.get
              NumberOfDestinations() > j) &&
                              (ListOfStaticDestinations.get
              Destination(i).getDestinationFloor() <
                               ListOfSortedDestinations.get
              Destination(j).getDestinationFloor()) &&
                              (ListOfSortedDestinations.get
              Destination(j).getDestinationFloor() <=
                               FloorAfterBraking) ); j++);
              ListOfSortedDestinations.insertDestination(List
              OfStaticDestinations.getDestination(i),j);
            }
            else{
              int j;
              for (j = 0; ( (ListOfSortedDestinations.get
              NumberOfDestinations() > j) &&
                              (ListOfStaticDestinations.get
              Destination(i).getDestinationFloor() >
                               ListOfSortedDestinations.get
              Destination(j).getDestinationFloor()) ); j++);
              ListOfSortedDestinations.insertDestination(List
              OfStaticDestinations.getDestination(i),j);
            }
          }
        }
      }
    } // method updateListOfSortedDestinations
```

17.5 Claus Albøge

```
/*******************************************************
public class ElevatorSetup extends MenuWindow {

/*******************************************************
MaxUpDeAccelerationbSetupSpinTextDouble =
new SetupSpinTextDouble(PMaxUpDeAcceleration,100,10,1000,1)
```

```
MaxDownDeAccelerationbSetupSpinTextDouble =
new SetupSpinTextDouble(PMaxDownDeAcceleration,100,10,1000,1);

/*****************************************************
* Sets the input data in DataVar.
*****************************************************/
  public void setData(){
      mWindow.DataVar.createListOfElevatorData();
      ListOfElevatorData =
new ElevatorData[mWindow.DataVar.getNoOfElevatorInBuilding()];
      ElevatorDataObject =
new ElevatorData(
       ~
       ~
     − MaxUpDeAccelerationbSetupSpinTextDouble.getNumber() /
mWindow.DataVar.getDistanceBetweenFloor(),
    MaxDownDeAccelerationbSetupSpinTextDouble.getNumber() /
mWindow.DataVar.getDistanceBetweenFloor());

  for(int i = 0; i < mWindow.DataVar.getNoOfElevatorInBuilding(); i++)
       ListOfElevatorData[i] = ElevatorDataObject;
 mWindow.DataVar.setListOfElevatorData(ListOfElevatorData);
mWindow.DataVar.testPrintElevatorData();
    }



/*****************************************************
 public abstract class Group implements Serializable{

/*****************************************************
* Private reference to the setup of  the entire simulation
*****************************************************/
 private DataContainer DataContainerObject;

/****************************************************
* Cannot be called directly because it is abstract.
* param DataContainerObject   Reference to the setup of t
* he entire simulation
```

```
*****************************************************
public Group (DataContainer DataContainerObject)
   {
   this.DataContainerObject = DataContainerObject;
// constructer Group

/****************************************************
* Get new departure time for the passenger. (abstract)
* param PassengerObject            Which passenger to change
* param DataPassengerFlowObject The passengerflow of the
* passenger.
*****************************************************
abstract public void getNewTime(Passenger PassengerObject, D
ataPassengerFlow DataPassengerFlowObject);

/****************************************************
* Get new destination for the passenger. (abstract)
* param PassengerObject            Which passenger to change
* param DataPassengerFlowObject  The passengerflow of
* the passenger.
*****************************************************
abstract public void getNewDestination(Passenger Passenger O
bject, DataPassengerFlow DataPassengerFlowObject);

} // class Group
/****************************************************


/ ****************************************************
abstract class GroupTypeGetSpecificTime extends Group
implements Serializable{

/****************************************************
* The list of departure times for the getNewTime(Passenger
* P) method.
****************************************************/
 private int ClockHour;
 private int ClockMinute;
 private int DayInSimulation;
 private Passenger PassengerObject;
```

```
  private DataPassengerFlow DataPassengerFlowObject;
  private DataContainer DataContainerObject;

/*******************************************************
Cannot be called directly because it is abstract.
* param DataContainerObject       The DataContainerObject!
*/
  public GroupTypeGetSpecificTime (DataContainer DataContainerObject) {
      super(DataContainerObject);
   this.DataContainerObject = DataContainerObject;

  } // constructer GroupTypeGetSpecificTime

/****************************************************
* Get specific departure time for the passenger
* (from departureTime[TravelState]).
* param PassengerObject          Which passenger to change.
* param DataPassengerFlowObject Where to get the data for
* the passenger.
******************************************************* //
setTimeData must be executed before getNewTime is called

  public void getNewTime(Passenger PassengerObject,
   DataPassengerFlow DataPassengerFlowObject) {

    this.PassengerObject = PassengerObject;
    this.DataPassengerFlowObject = DataPassengerFlowObject;

    PassengerObject.setNewTime(60*60*((24*DayInSimulation)+
ClockHour)+60*ClockMinute);
    //    PassengerObject.setNewTime(ClockHour+ClockMinute);
  } // method getNewTime

   protected void setTimeData(int ClockHour,int ClockMinute,
int DayInSimulation){
        this.DayInSimulation = DayInSimulation;
        this.ClockHour = ClockHour;
        this.ClockMinute = ClockMinute;
      }
```

```
} // class GroupTypeGetSpecificTime
/*****************************************************/


/*****************************************************
class GroupTypeGetSpecificTimeAndOneFloor extends
GroupTypeGetSpecificTime implements Serializable {

  private DataClockOneFloor ListOfDataClockOneFloor = null;
  private DataClockOneFloor SeekListOfClockOneFloor = null;
  private Passenger PassengerObject;
  private DataPassengerFlow DataPassengerFlowObject;
  private DataContainer DataContainerObject;
  private int DayInSimulation;


/*****************************************************
Instantiates a new Group Object which gives a specific de∗ parture time and a
∗ specific destination each time.
∗ <br><b>Alert!</b>The Destination parameter must not have ∗ two adjecent destinations
∗ of the same value (not even first/last)<br>
∗ param DataContainerObject    Reference to the setup of th∗ e entire simulation
****************************************************
 public GroupTypeGetSpecificTimeAndOneFloor(DataContainer
  DataContainerObject) {
    super(DataContainerObject);
    this.DataContainerObject = DataContainerObject;
    DayInSimulation = 0;

// constructer GroupTypeGetSpecificTimeAndOneFloor

/***************************************************** ∗
Get a new destination for the passenger. This method
∗ returns one specific floor
∗ from DestinationFloor[TravelState]
∗ <br><b>Must be called before the getNewTime()
∗ method!</b><br>
∗ param PassengerObject            Which passenger to change.
∗ param DataPassengerFlowObject Where to get the data for ∗ the passenger.
****************************************************/
```

```
public void getNewDestination(Passenger PassengerObject,
 DataPassengerFlow DataPassengerFlowObject) {

  this.PassengerObject = PassengerObject;
  this.DataPassengerFlowObject = DataPassengerFlowObject;
  SeekListOfClockOneFloor = ListOfDataClockOneFloor;

  // seek throug the list of destination
  if(PassengerObject.getDestination() != −1){
    while((SeekListOfClockOneFloor.getNextInList()
    != null) && (PassengerObject.getDestination()
    != SeekListOfClockOneFloor.getNextFloor())){
    SeekListOfClockOneFloor
    = SeekListOfClockOneFloor.getNextInList();
    }

    if(SeekListOfClockOneFloor.getNextInList() == null){
      DayInSimulation++;
      SeekListOfClockOneFloor = ListOfDataClockOneFloor;
      PassengerObject.setTravelState(1);
    }
    else
      SeekListOfClockOneFloor
      = SeekListOfClockOneFloor.getNextInList();
  }

  //    System.out.println("DayInSimulation =   " + DayInSimulation);

  setTimeData(SeekListOfClockOneFloor.getClockHour(),
  SeekListOfClockOneFloor.getClockMinute(),DayInSimulation     );

  PassengerObject.setNewDestination(SeekListOfClockOneFloor.getNextFloor());

} // method getNewDestination

   public DataClockOneFloor getListOfDataClockOneFloor(){
       return ListOfDataClockOneFloor;
   }

   public void setListOfDataClockOneFloor(DataClockOneFloor
ListOfDataClockOneFloor){
        this.ListOfDataClockOneFloor
        = ListOfDataClockOneFloor;
    }

} // class GrouptypeGetSpecificTimeAndOneFloor
/*****************************************************/
```

17.6 Peter Koorsgaard

```
/*****************************************************
 * A class which calculates the time needed to travel from
 * the current position to it's destination. (Only positive
 * direction shown because of space restrictions)
 *****************************************************/
class TravelTime {

 private ElevatorInfo StartPosition;
 private ElevatorInfo BrakePosition;
 private ElevatorInfo AccelerationPosition;
 private ElevatorInfo MaximumSpeedPosition;
 private ElevatorInfo DeAccelerationPosition;
 private ElevatorData DataOfElevator;

 public TravelTime(ElevatorData DataOfElevator,
                   Destination DestinationFloor,
                   ElevatorInfo CurrentPosition) {


   double T_start = CurrentPosition.getTime();
   double S_start = CurrentPosition.getPosition();
   double V_start = CurrentPosition.getVelocity();

   double T_brake = 0.0;
   double S_brake = S_start;
   double V_brake = V_start;

   double S_acc; double T_acc; double T_deacc;
double S_max; double V_max; double T_max;

StartPosition  = CurrentPosition;

if (DestinationFloor.getDestinationFloor() > S_start) {
   if (V_start<0) {
      // brake - because we want to go up
      T_brake = − V_start /
                 DataOfElevator.getDownDeAcceleration();
      S_brake = V_start*T_brake + 0,5*
                 DataOfElevator.getDownDeAcceleration()
                 *T_brake*T_brake + S_start;
      V_brake = 0.0;
   } // brake - for updirection
  // are we going to miss the destination in first try?
  else if ( ( 0.5*V_start*V_start +
           DataOfElevator.getUpDeAcceleration()*
           DestinationFloor.getDestinationFloor()
           − DataOfElevator.getUpAcceleration()*S_start)
           / ( DataOfElevator.getUpDeAcceleration()
           − DataOfElevator.getUpAcceleration() )
           < S_start) {
      // brake - because we are going too far
      T_brake = − V_start /
                  DataOfElevator.getUpDeAcceleration();
      S_brake = V_start*T_brake + 0.5*
                 DataOfElevator.getUpDeAcceleration()
                 *T_brake*T_brake + S_start;
      V_brake = 0.0;
   } // brake - because we are going too far
 } // going up

else {
     // do the same thing for down direction...
     }

if ( DestinationFloor.getDestinationFloor()>S_brake ) {
   // going up
   S_max = ( 0.5*V_brake*V_brake
            + DataOfElevator.getUpDeAcceleration()*
```

```
                DestinationFloor.getDestinationFloor()
                − DataOfElevator.getUpAcceleration()∗S_brake
                ) /
                ( DataOfElevator.getUpDeAcceleration()
                − DataOfElevator.getUpAcceleration() );
        V_max = Math.sqrt(2∗
                DataOfElevator.getUpDeAcceleration()∗
           (S_max − DestinationFloor.getDestinationFloor()));
        if ( V_max > DataOfElevator.getMaxUpVelocity() ) {
          // V_max bigger than maxspeed
          // recalculate deacceleration position
          S_max = ( DataOfElevator.getMaxUpVelocity()∗
                    DataOfElevator.getMaxUpVelocity() ) /
                  ( 2∗DataOfElevator.getUpDeAcceleration() )
                   + DestinationFloor.getDestinationFloor();
          // maximum speed is reset..
          V_max = DataOfElevator.getMaxUpVelocity();
          } // V_max > MaxUpVelocity

       T_deacc = − V_max/DataOfElevator.getUpDeAcceleration();
        T_acc = ( V_max − V_brake ) /
                DataOfElevator.getUpAcceleration();
        S_acc = V_brake∗T_acc + 0.5∗
               DataOfElevator.getUpAcceleration()∗T_acc∗T_acc
                + S_brake;
        if (!(V_max == 0)) { T_max = (S_max−S_acc) / V_max; }
        else { T_max = 0; }

        } // going up
       else {
        // do the same for downdirection
       } // going down


BrakePosition          = new ElevatorInfo (S_brake,
                 V_brake,CurrentPosition.getDirection(),
                 T_brake);
AccelerationPosition   = new ElevatorInfo (S_acc,V_max,
                 CurrentPosition.getDirection(),
                 T_acc+T_brake);
```

```
    MaximumSpeedPosition    = new ElevatorInfo (S_max,V_max,
                          CurrentPosition.getDirection(),
                          T_max+T_acc+T_brake);
    DeAccelerationPosition = new ElevatorInfo
                         (DestinationFloor.getDestinationFloor(),
                          0.0,CurrentPosition.getDirection(),
                          T_deacc+T_max+T_acc+T_brake);

  } // constructor TravelTime

  public double getTotalTime () {
    return StartPosition.getTime() + BrakePosition.getTime()
           + AccelerationPosition.getTime()
           + MaximumSpeedPosition.getTime()
           + DeAccelerationPosition.getTime();
  } // method getTotalTime

} // class TravelTime

/*********************************************************
 * The Group SuperClass.
 *********************************************************/
public abstract class Group implements Serializable{

  private DataContainer DataContainerObject;
  public Group (DataContainer DataContainerObject) {
    this.DataContainerObject = DataContainerObject;
  } // constructer Group

  /*********************************************************
  * Get new departure time for the passenger. (abstract)
  *********************************************************/
  abstract public void getNewTime(Passenger PassengerObject,
                         DataPassengerFlow DataPassengerFlowObject);

  /**********************************************************
  * Get new destination for the passenger. (abstract)
  **********************************************************/
  abstract public void getNewDestination(Passenger
                                    PassengerObject, DataPassengerFlow
```

```
                              DataPassengerFlowObject);

} // class Group


/******************************************************
 * The Protocol. This class is the interface between the
 * control system and the model. (only partly included
 * because of space restrictions)
 ******************************************************/
class Protocol extends Observable implements Observer{

 public static final int SCHEDULECHANGED = 60;
 private int CurrentElevatorNumber;
 private Destination CurrentDestination;
 private double CurrentTime;
 private ElevatorMovement[ ] ListOfElevatorMovements;
 private ControlSystem ControlSystemObject;

 /******************************************************
 * constructor protocol.
 ******************************************************/
 public Protocol (int NumberOfFloorsInBuilding,
                  int NumberOfElevatorsInBuilding,
                  ElevatorData[ ] DataOfElevators,
                  Observer ObserverObject) {

   // add the observer, so that we can notify it later.
   addObserver(ObserverObject);
   // create listOfElevatorMovements of correct size
   ListOfElevatorMovements =
         new ElevatorMovement[NumberOfElevatorsInBuilding];

   // instantiates the ListOfElevatorMovements
   for (int i=0; i<NumberOfElevatorsInBuilding; i++) {
     ListOfElevatorMovements[i] =
     new ElevatorMovement(DataOfElevators[i],
     new Destination(0.0,1,Destination.NODIRECTION),
    new ElevatorInfo(1.0,0.0,Destination.NODIRECTION,0.0));
   } // for loop
```

```
    // create ControlSystem object
    ControlSystemObject =
      new ControlSystem (NumberOfFloorsInBuilding,
          NumberOfElevatorsInBuilding,DataOfElevators,this);
  } // constructor Protocol

  /*****************************************************
  * Update method. This method is invoked by the
  * notifyobservers of an observable of this observer.
  * ( see Observer interface and Observable class )
  *****************************************************/
  public void update (Observable o, Object args) {
    if ( ((ControlSystemOutput)args).getCommand()
      == ControlSystemOutput.SCHEDULECHANGED ) {

    CurrentElevatorNumber =
          ((ControlSystemOutput)args).getElevatorNumber();
    CurrentDestination      =
          ((ControlSystemOutput)args).getNewDestination();

    ListOfElevatorMovements[CurrentElevatorNumber].
    setNewDestination(CurrentDestination,CurrentTime);

    CurrentDestination = new
      Destination(ListOfElevatorMovements
      [CurrentElevatorNumber].getNextActionTime(),
      CurrentDestination.getDestinationFloor(),
      CurrentDestination.getDirection() );

      setChanged();
      notifyObservers(this);
    } // is it a ControlSystemOutput object?
  } // method update
} // class Protocol
```

17.7 Lars Jochumsen Kristensen

```
/*******************************************************
```

```
 * The class DestinationManager
 ******************************************************/
public class DestinationManager implements Observer{

  private RunSimulation rSimulation;
  private DataPassengerFlow SeekDataPassengerFlow;

  private int GroupID;
  private int NoOfPassengerInGroup;
  private int StartFloor;
  private int StartHour;
  private int StartMinute;
  private int PassengerID = 0;
  private Pool PoolObject;
  private Passenger PassengerObject;

/******************************************************
 * Instantiates the DestinationManager which give all the
 *      Passengers a start destination and departuretime.
 *      (specific destination each time.)
 ******************************************************/
  public DestinationManager(RunSimulation  rSimulation){

    this.rSimulation = rSimulation;

  } // constructor DestinationManager


/******************************************************
 * Creates the passenger with there destiantions and
 *      departure times.
 * The passengers is placed on there start floors.
 * The departure times is added to the TimeManagers list
 *      of passenger events
 ******************************************************/
public void createPassengerInBuilding(){
  SeekDataPassengerFlow =
        rSimulation.getMenuWindow().DataVar.
        getListOfDataPassengerFlow();
  PassengerID = 0;

  while(SeekDataPassengerFlow != null){
  // get specific data about the group
  GroupID = SeekDataPassengerFlow.getGroupID();
  NoOfPassengerInGroup =
        SeekDataPassengerFlow.getNoOfPassengerInGroup();
  StartFloor = SeekDataPassengerFlow.getStartFloor();
  StartHour = SeekDataPassengerFlow.getStartHour();
  StartMinute = SeekDataPassengerFlow.getStartMinute();
  //Create all passenger in the group with a destination
  //     and a departuretime
  for(int i = 0; i < NoOfPassengerInGroup; i++,PassengerID++){
     PoolObject = rSimulation.getBuildingObject()
        .getFloor(StartFloor).getPool();

     PassengerObject = new Passenger(PassengerID,GroupID);
     // get new destination and departuretime
     SeekDataPassengerFlow.getListOfGroupEvent().getNewDestination
        (PassengerObject,SeekDataPassengerFlow);
     SeekDataPassengerFlow.getListOfGroupEvent().getNewTime(
        PassengerObject,SeekDataPassengerFlow);
    // Save new destination and departuretime in the object of
    //  the passenger and timemanager
    PoolObject.addPassenger(PassengerObject);
    rSimulation.getTimeManagerObject().addPassengerEvent
        (PoolObject,PassengerObject);
  }
  SeekDataPassengerFlow = SeekDataPassengerFlow.getNextInList();
} // loop
} // end while


/*********************************************************
 * This metode is called the obsevable class pool notifyes
 * this observer. Get the type of command to be executed
 * in pool. The command be executed is to get a new
 * destination for the passenger.
 *********************************************************/
 public void update(Observable o, Object arg){
```

```
 if(((Pool)arg).getCommand() ==
      ((Pool)arg).GETNEWDESTINATION){
   // get information from pool
   PassengerObject = ((Pool)arg).getPassengerToMoveObject();
   PoolObject = ((Pool)arg).getPoolObject();
   // find reference to passengerflow of the group the
   // passenger are palced in.
   SeekDataPassengerFlow = rSimulation.getMenuWindow()
      .DataVar.getListOfDataPassengerFlow();
   while((SeekDataPassengerFlow != null) &&
       (SeekDataPassengerFlow.getGroupID() !=
       PassengerObject.getGroupID())){
     SeekDataPassengerFlow = SeekDataPassengerFlow
      .getNextInList();
   }

   // if passengerflow of the group exist then get the
   // passenger a new desination and departuretime
   if(SeekDataPassengerFlow != null){ SeekDataPassengerFlow
   .  getListOfGroupEvent().getNewDestination
      (PassengerObject,SeekDataPassengerFlow);

      SeekDataPassengerFlow.getListOfGroupEvent()
        .getNewTime(PassengerObject,SeekDataPassengerFlow);

      // Save new destination and departuretime in the object of
      // the passenger and timemanager
      PoolObject.addPassenger(PassengerObject);

      rSimulation.getTimeManagerObject().addPassengerEvent
        (PoolObject,PassengerObject);
      }
   } // is command = GETNEWDESTINATION
} // method update

} // class DestinationManager



/*******************************************************
```

```
 * The class DataClockOneFloor
 *****************************************************/
class DataClockOneFloor implements Serializable{

  private int ClockHour;
  private int ClockMinute;
  private int NextFloor;

  private DataClockOneFloor NextInList = null;

 /*****************************************************
 * Constructor of the class DataClockOneFloor
 *****************************************************/
  public DataClockOneFloor(int ClockHour, int ClockMinute,
    int NextFloor){
        this.ClockHour = ClockHour;
        this.ClockMinute = ClockMinute;
         this.NextFloor = NextFloor;
  }

 /*****************************************************
 * Add an object of DataClockoneFloor to the linked list of
 * DataClockOneFloor
 *****************************************************/
  public DataClockOneFloor addDataClockOneFloor(int
     ClockHour, int ClockMinute, int NextFloor){
        NextInList = new DataClockOneFloor(ClockHour,
        ClockMinute, NextFloor);
    return NextInList;
  }

        [...]

 /*****************************************************
 * Returns the next floor
 *****************************************************/
  public int getNextFloor(){
    return NextFloor;
  }
```

```
/*****************************************************
* Returns the next element in the linked list of
* DataClockOneFloor
*****************************************************/
  public DataClockOneFloor getNextInList(){
     return NextInList;
  }
}


/*****************************************************
* The class GroupTypeGetSpecificTimeAndOneFloor
*****************************************************/

class GroupTypeGetSpecificTimeAndOneFloor extends
GroupTypeGetSpecificTime implements Serializable {

  [. . .]

/*****************************************************
* Get a new destination for the passenger. This method
* returns one specific floor
*****************************************************/
public void getNewDestination(Passenger PassengerObject,
       DataPassengerFlow DataPassengerFlowObject) {

   this.PassengerObject = PassengerObject;
   this.DataPassengerFlowObject = DataPassengerFlowObject;

   SeekListOfClockOneFloor = ListOfDataClockOneFloor;

   // seek throug the list of destination
   if(PassengerObject.getDestination() != −1){
     while((SeekListOfClockOneFloor.getNextInList() != null)
        && (PassengerObject.getDestination() !=
        SeekListOfClockOneFloor.getNextFloor())){
      SeekListOfClockOneFloor =
        SeekListOfClockOneFloor.getNextInList();
     }
```

```
        if(SeekListOfClockOneFloor.getNextInList() == null){
          DayInSimulation++;
          SeekListOfClockOneFloor = ListOfDataClockOneFloor;
          PassengerObject.setTravelState(1);
        }
        else
          SeekListOfClockOneFloor = SeekListOfClockOneFloor
            .getNextInList();
      }

      setTimeData(SeekListOfClockOneFloor.getClockHour(),SeekListO
            fClockOneFloor.getClockMinute(),DayInSimulation);

    PassengerObject.setNewDestination(SeekListOfClockOneFloor
            .getNextFloor());

  } // method getNewDestination

  public DataClockOneFloor getListOfDataClockOneFloor(){
    return ListOfDataClockOneFloor;
  }

  public void setListOfDataClockOneFloor(DataClockOneFloor
            ListOfDataClockOneFloor){
    this.ListOfDataClockOneFloor = ListOfDataClockOneFloor;
    }

}
```

Bibliography

[1] Arnold and Gosling. *The Java Programming Language*.

[2] Gamma et al. *Design Patterns - Elements of Reusable Object Oriented Software*.

[3] Lars Mathiassen et al. *Objekt Orienteret Analyse og design*. Forlaget Marko Aps, 2nd edition, 1998.

[4] Sum Microsystems Inc. Jdk 1.1.6 documentation. December 1998. http://www.java.sun.com/products/jdk/1.1/docs/index.html.

[5] John Lewis and William Loftus. *Java Software Solutions*. Addison Wesley, 1st edition, 1998.

[6] Betrand Meyer. *Applying Design by Contract*. IEEE Computer, October 1998.

[7] Roger S. Pressman. *Software Engineering - a Practitioners approach*.

[8] Richard S. Sutton. Elevator dispatching. December 1998. www-anw.cs.umass.edu/ rich/book/11/node5.html.