A UML Documentation for an Elevator System

Distributed Embedded Systems, Fall 2000
PhD Project Report

Lu Luo
December 2000

A UML documentation for an elevator system

1. Introduction

This paper is a PhD project report for the course *Distributed Embedded Systems* at Carnegie Mellon University. Throughout this course, a distributed real-time system – an elevator control system– is specified, designed, built, and simulated. Object Oriented Analysis and Design methods, in specific the Unified Modeling Language (UML) are used when designing the system.

A lot of corners in the design of this elevator system are cut in the regular class. The existing UML documentations for the elevator lab of this course are pretty lame compared to real elevators. It is therefore not so clear whether UML will really represent the design of an elevator well. In this report, a rigorous UML documentation package for the class project is given, based on current system design.

From different points of view how UML can be used in a real-time, distributed system, three groups of UML diagrams are given, the diagrams in these groups differ mostly in their class diagrams, focusing on the viewpoint of object architecture, software architecture and system architecture correspondingly.

In the following part of this report, overviews of UML and distributed embedded systems are given in section 2 and section 3, correspondingly. In section 4, the design of our elevator control system is presented from a static structural point of view, i.e. the Use Case diagram and the Class diagrams are presented and analyzed. The Sequence diagrams and State Chart diagrams given in Section 5 emphasize on the dynamic aspects of the system. Section 6 is the conclusion.

2. A Brief Introduction to UML

The Unified Modeling Language (UML) is the industry-standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as other non-software systems. UML simplifies the complex process of software design, making a "blueprint" for construction, and is now the standard notation for software architecture.

UML provides both the structural views and behavioral views of the system. A set of diagrams with different graphical elements is the core part as well as the most expressive presentation in UML. The UML includes nine kinds of diagrams, for the sake of grasp the most representative aspects of the design of elevator system, in this paper only following UML diagrams are used and analyzed:

Use Case diagram shows a set of use cases and actors (a special kind of class) and their relationships. Use case diagrams address the static use case view of a system, these diagrams are important in organizing and modeling the behaviors of a system.

Class diagram shows a set of classes, interfaces, and collaborations and their relationships. Class diagrams are the most common diagrams used in modeling object-oriented systems. Class diagrams address the static design view of a system.

Sequence diagram is an interaction diagram. Interaction diagrams address the dynamic view of a system, besides sequence diagram, the other interaction diagram in UML is the **Collaboration diagram**. Sequence diagram emphasizes the time ordering of messages between objects in the system, while collaboration diagram emphasizes the structural organization of the objects that send and receive messages. Sequence diagrams and collaboration diagrams are isomorphic, and can be transformed from one into the other. Since either of them contributes to the same extend of understanding of our system, while sequence diagrams give more ideas of time, which is essential for real time systems, only the sequence diagrams are given in this report.

State chart diagram shows a state machine, consisting of states, transitions, events and activities. State chart diagrams address the dynamic view of a system. State chart diagrams are especially important in modeling the behavior of an interface, class, or collaboration and emphasize the event-ordered behavior of an object, which is especially useful in modeling reactive systems.

The rest four kinds of UML diagrams are: **Object diagram**, showing a set of objects and their relationships; **Activity diagram**, a special kind of State chart diagram showing the flow from activity to activity within a system; **Component diagram,** showing the organizations and dependencies among a set of components; and **Deployment diagram** showing the configuration of run-time processing nodes and the components that live on them.

3. An Over view at Systems that are Real-time, Distributed, and Embedded

Before discussing the detailed problems involved in designing our elevator system with UML, the definitions of real time system, distributed system, and embedded system are necessary to be presented here, the difference between real-time distributed systems and software systems in the common sense that most object-oriented design and analysis pays effort on by UML are briefly stated. In the rest parts of this paper, the pros and cons of UML used in designing distributed embedded systems are largely discussed.

According to Kopetz, a **real time** computer system is a computer system in which the correctness of the system behavior depends not only on the logical results of the computations, but also on the physical instant at which these results are produced[1]. As the saying goes: *"The right answer late is wrong*", in real time systems, performance requirements are as important as functional requirements, so not only do we have to perform the correct functions, but there are clear bounds within which these must be completed. An **embedded computer system** is a system that uses a computer as a component, but whose prime function is not that of a computer.

As one of the object-oriented techniques, UML is basically suitable for real time system development. There are techniques within UML definition that are a natural fit for specifying and designing real-time systems. Use cases allow the designers to describe the way in which humans and external devices, interact with the system. Object sequence diagrams, describe for a given use case, the events which cause the interaction and the detailed system response, **including timing**.

Class diagrams helps to separate system components and define interfaces between them. These techniques are good enough to capture usage scenarios and identify likely time problems.

We've been trying to answer the question during our practice using UML to design the elevator system: *"Is UML a suitable modeling language for the development of real-time systems?"* We found that based on the features mentioned in last paragraph, UML is applicable but not yet enough. There are some main challenges in the design of real-time embedded system with UML, as follows:

- The definition of specific hardware elements and their characteristics;
- Specifying the time constraints at the object, task and hardware levels.
- Modeling the network.

In the following sections, we will attempt to figure out how to use UML for a better description for the elevator system. Some practical suggestions that might be a useful addition to the standard UML are given out.

4. Modeling the static aspects of the system

4.1 A snapshot of the elevator control system

The elevator system designed as our class project is rather an "ideal" elevator in which some of the technical corners are cut. Our elevator has the basic function that all elevator systems have, such as moving up and down, open and close doors, and of course, pick up passengers. The elevator is supposed to be used in a building having floors numbered from 1 to MaxFloor, where the first floor is the lobby. There are car call buttons in the car corresponding to each floor. For every floor except for the top floor and the lobby, there are two hall call buttons for the passengers to call for going up and down. There is only one down hall call button at the top floor and one up hall call button in the lobby. When the car stops at a floor, the doors are opened and the car lantern indicating the current direction the car is going is illuminated so that the passengers can get to know the current moving direction of the car. The car moves fast between floors, but it should be able to slow down early enough to stop at a desired floor. In order to certificate system safety, emergency brake will be triggered and the car will be forced to stop under any unsafe conditions.

4.2 Use Case Diagram

All systems interact with human or automated actors that use the system for some purpose, and both human and actors expect the system to behave in predictable ways. In UML, a use case models the behaviors of a system or a part of a system, and is a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor [2].

A use case diagram models the **dynamic design view** of systems. Use case diagrams are central to modeling the behavior of a system, a subsystem, or a class. Use case diagram shows a set of use cases and actors and their relationships. The main contents of a use case diagram are:

- Use Cases
- Actors
- Dependency, generalization, and association relationships

According to the requirements document in our class, the use case diagram of elevator systems is showed in Figure 1:

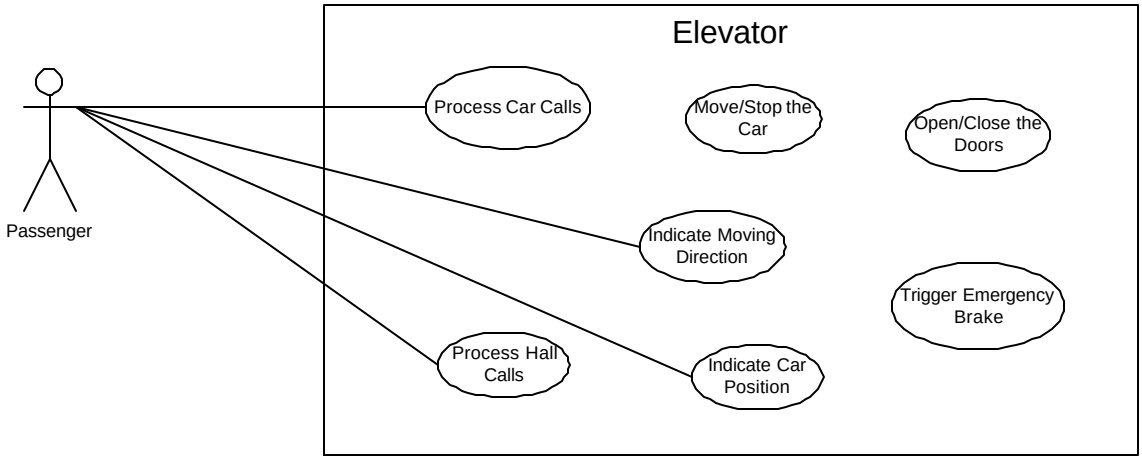

Figure 1: Use Case Diagram of Elevator System

There are seven use cases based on the requirement documentation of the elevator system in our class, as shown in Figure 1:

- Process Car Calls: This use case includes several scenarios, which will be described in detail in later sections of this paper. These scenarios includes that the elevator receives car calls from the passengers, turns on or turns off the light of car call buttons, updates the record of car calls stored in system controlling parts, etc.
- Process Hall Calls: Similar to Car Call processing, this use case includes that the elevator receives hall calls from the passengers, turns on or turns off the light of hall call buttons, updates the record of hall calls in system controlling parts, etc.
- Move/Stop the Car: The main function of an elevator, detailed action will include the changing of driving speed, how to make the decision of stop, and driving directions of the car.
- Indicating Moving Direction: The elevator should have this mechanism to let the passengers know the current moving direction of the car such that the passenger might decide whether to enter the car or not.
- Indicating Car Position: Similarly, the elevator should let the passengers know whether his/her destination floor is reached so that the passenger may decide to leave the car.
- Open/Close the Doors: The elevator should be able to open and close the doors for the passengers to get in and out of the car. The functional areas of this use case should also enable the passengers to make door reversals when the doors are closing and the passenger wants to get in the car.
- Trigger Emergency Brake: There is safety mechanism within the car to make sure that an unsafe state is not transiently generated.

The only actor in elevator system is the passenger, which is the role that humans play when interacting with the system. The passenger interacts with the Elevator system by making car calls

and hall calls. The passenger also makes decision whether to enter/leave the car or not by observing the indication of moving direction and car position. Therefore the use case diagram shows that the actor has relationship with four use cases of the system: Process Car Calls, Process Hall calls, Indicate Moving Direction, and Indicate Car Position.

4.3 Class Diagram

Class diagram, one of the most commonly used diagrams in object-oriented system, models the **static design view** for a system. The static view mainly supports the functional requirements of a system – the services the system should provide to the end users. We will see from our practical experience that lots of fun comes out when modeling out system with class diagrams. The discussion on different views of class diagrams for the system will be put into emphasis later in this paper.

A class diagram shows a set of classes, interfaces, and collaborations and their relationships. Class diagrams involve global system description, such as the system architecture, and detail aspects such as the attributes and operations within a class as well. The most common contents of a class diagram are:

- Classes
- Interfaces
- Collaborations
- Dependency, generalization, and association relationships
- Notes and constraints

In the following subsections, three groups of class diagrams are presented and analyzed in detail. The corresponding sequence diagrams for each class diagram is introduced in section 4.4.

4.3.1 Class Diagram – the Object Construction View

From the elevator use cases we've got in section 4.2 and according to the requirements of the system, intuitively we get a class diagram as is shown in Figure 2.

We can now get some idea of how the system is composed of from the description of classes in Figure 2. We will not go any further into the detailed class components such as attributes and operations of each class, which is out of the scope of our current view. In this regard, we build this class diagram from the view of object composition of the system.

- **ElevatorControl:** The central controlling object in the elevator system. ElevatorControl communicates and controls all other objects in the system.
- **Door:** There are two doors in the system, the "god" object - the ElevatorControl – command the doors to open and close, according to the situation stated in the use case.

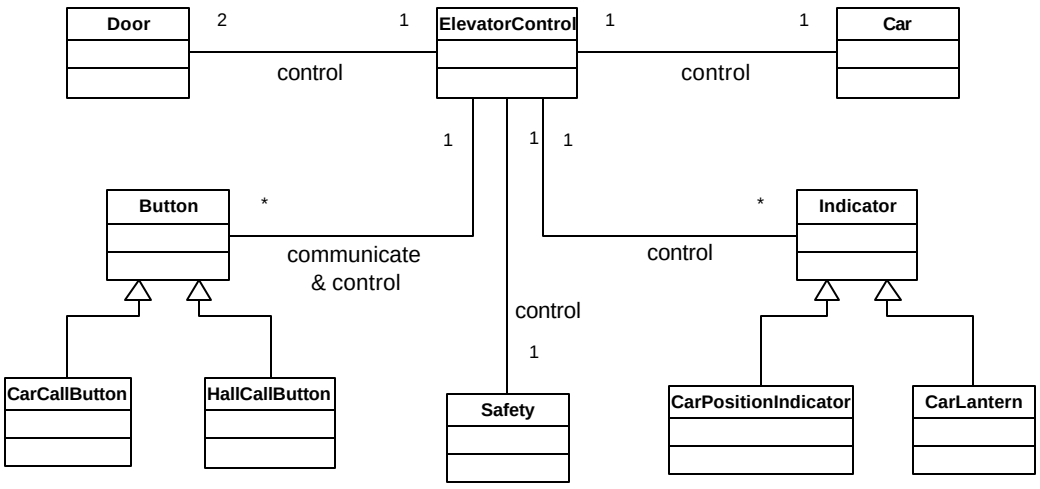

Figure 2: Class Diagram - the Object Construction View

- **Car:** The car is being controlled to move up and down (in different speeds), to make stops at floors when necessary.
- **Button:** The ElevatorControl class also controls the button class, which further generalizes two subclasses CarCallButton and HallCallButton. The control object communicates with the Button objects, get the information whether a button is pressed and in turn controls the illumination of Button lights.
- **Indicator:** There are two kinds of indicators in the system, the CarPositionIndicator and the CarDirectionIndicator (i.e. the CarLantern). The indicators are controlled to show the information about the current position and moving direction of the car.
- **Safety:** Whenever an emergency happens according to the definition of emergency brake trigger in the requirement documentation, the ElevatorControl commands the Safety

This version of class diagram is derived directly from the use cases stated in section 4.2. The classes captured in this diagram can cover all the functional aspects of the system: for moving or stopping the car, we have the class Car, and the control class ElevatorControl; for opening or closing the doors, we have the class Door; for the passenger to know the position and direction of the elevator, we have Indicator class, for the passenger to make car calls or hall calls, we have the Button classes; we also have the Safety class fulfilling the system need of emergency brake processing. All the classes have interfaces with the central controller, whose job is in charge of the actions of other objects. From the point of view of object division and system functioning, this class diagram helps understanding the basic design idea of the system.

When we tried to go further into the design of our elevator control system, and thus find our way toward a detailed design then a good implementation of our system starting from this class diagram, problems arose. In the context of this report, the order of designing the system with an existing architecture and capturing the UML documentation is reversed, i.e. rather than designing the system first using UML, we “inherit” a ready elevator system architecture from the instructor, and have part of the software design in hand before working on the UML. The reason why we know that the class diagram is not quite perfect for a final design before we really meet difficulties MIGHT because of above reasons, but it is almost certain in other cases that the

designers will find this the inappropriateness of this design for future development stages, sooner or later. Provided our system under discussion is a normal centralized system where every component (software and/or hardware) is controlled by one processor, our current class diagram solution may not cause future design defects. However, the nature of being a distributed embedded system determines that the class diagram merely from the point of view of objects in the elevator system is not adequate.

Considering the current class diagram in hand, potential defects of our future software design are stated as following. If a better solution is not found, the software design is going to be a failure.

- **Overburden of the central object:** From above analysis we can see that the ElevatorControl object has to act as the central controller that interacts with all other objects. All the computing and controlling tasks have to be done within this object
- **Idleness of some other objects:** While the ElevatorControl keeps working all the time, some of the other objects, such as Button and Indicator only act as system interface, even worse, as objects like the Door and the Car are actually part of the system, i.e. the "hardware", from the point of view of control software, they are outside the scope of the software system.
- **Competing for computing resource:** When more than one object want to get controlled by the central object at the same time, it is inevitable that these objects compete for the limited computing resources of the controller, and some of the objects may not get timely control messages for it to keep normal operation, which will cause a fatal defect in real-time system.
- **Low efficiency for the whole system:** Even if the computing resources in the controller are fast/large enough such that every control need is processed and taken into action in time, central node controlling is still not an efficient solution for a distributed system like the elevator.

4.3.2 Class Diagram – the Software Architecture View

Based on the analysis of last section and based on the software architecture from the class project, which is simulated and proved to be practically suitable of being the control system for an elevator, a class diagram is derived from this point of view.

This class diagram provides the solution for how to design and implement the control software. The software architecture of the actual elevator control system is reflected accurately in this diagram. Except for Dispatcher, all other control objects are derived from the super class ElevatorControl. The control objects share (some of) the property of ElevatorControl, and has its own attributes and operations used for the object it controlled. The objects controlled by the control objects are defined as environmental objects, which although exist in the elevator system, are in fact not belong to the software control system. In next section we will discuss these non-control objects in detail from the system architecture point of view.

- **DoorControl** controls the action of **DoorMotor**, each of the two DoorMotors on a car is controlled by a DoorControl object. DoorMotor can be commanded to open, close, or make a door reversal.
- **DriveControl** controls the elevator **Drive**, which acts as the main motor moving the car up and down, and stopping at floors when necessary.
- **LanternControl** are in the number of two, each controls a **CarLantern** indicating the current moving direction of the car.

- **HallButtonControl** exists in pair on each floor, where one controls the Up **HallCallButton** and the other Down. The HallButtonControl accepts hall call button presses controls as well as gives feedback to hall call lights.
- **CarButtonControl** is one for each floor and all locate in the car. The CarButtonControl accepts **CarCallButton** calls and is in charge of turning on/off the corresponding car call lights.
- **CarPositionIndicator** gives value to the **CarPositionIndicator** so that the passengers might know the current position of the car.

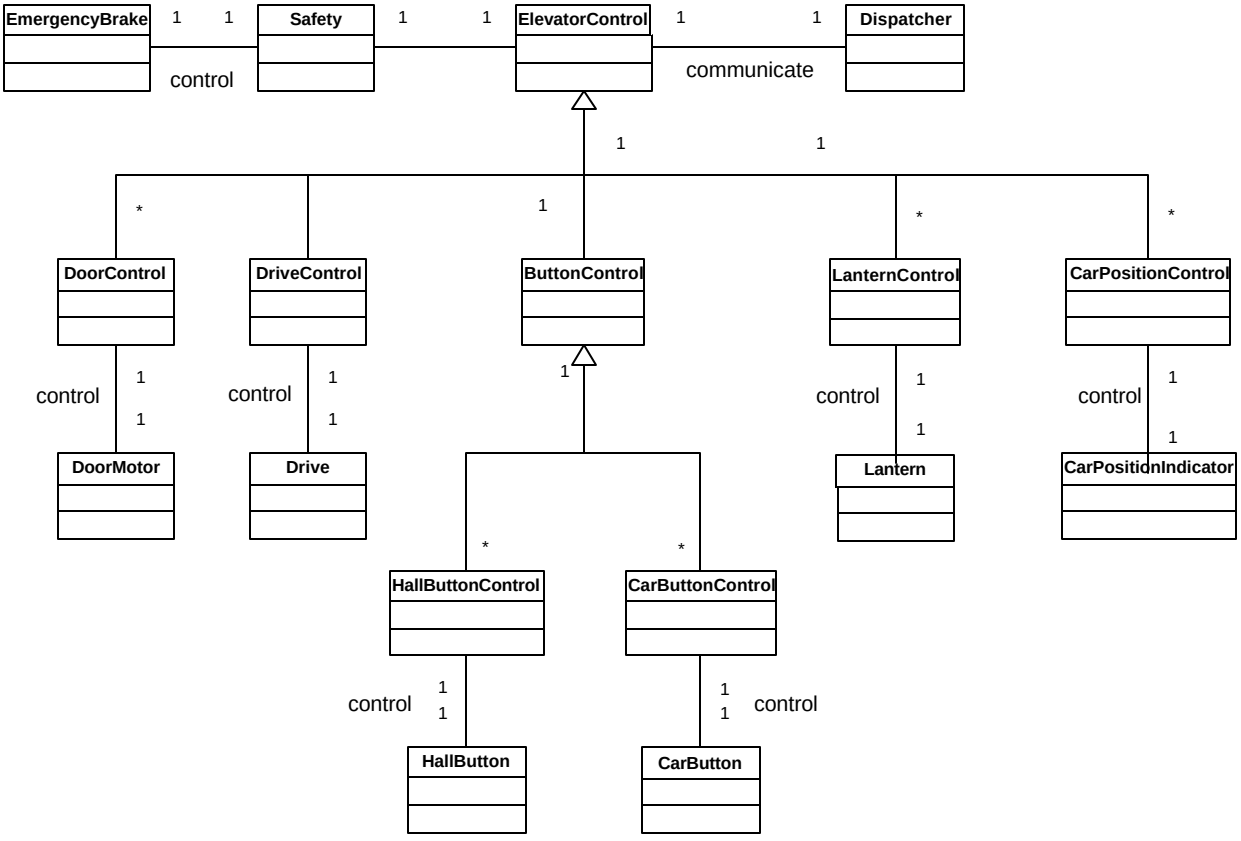

Figure 3: Class Diagram - the Software Architecture View

There are two non-control objects in the system:

- **Dispatcher** does not control actual elevator components, but it is important in the software system. There is one Dispatcher for each car, whose main function is to calculate the target moving direction and destination for the car, as well as to maintain the opening time for the doors. The Dispatcher interacts with nearly all the control objects in the system except for LanternControl.
- **Safety** is also an environmental object, which does not belong to the control software but is an important part of the system. In the real world, the safety actions vary if the

emergency brake of an elevator is triggered, in our elevator simulation system, though, if an emergency brake is triggered, only some message is displayed.

In our system, passenger is also modeled as an environmental object. Passengers interact with hall call buttons and car call buttons, make door reversals, observe the direction and position of elevator, etc. For the sake of simplicity, the passenger object is not shown (unlike other environmental objects) in Figure 3.

The software view of class diagram solved most of the problems put forward in last section. Since the control tasks are distributed to several control objects, each controlling one or a couple of environmental objects, no one is overburden or idle. There is no need competing for the computing resources of the central controller since the controllers are devoted to their controlled objects. However, starting from this class diagram issues about the implementation detail of our system will still arise, as following:

- How do the control objects control the environmental objects?
- How does one object get necessary information from other objects?
- How to model the network?

From the view of the system architecture, these questions will be answered.

4.3.3 Class Diagram – the System Architecture View

For answering the question brought out in last section, the class diagram is further detailed with classes as the network and sensors/actuators added to model the actual system architecture. Strictly, in this view of the system, the class diagram does not contain exactly the same meaning as in normal UML class diagrams. But as the class diagram is an excellent way of describing the static structural aspects of a system, why not use it to help express better the system architecture?

The components in the class diagram shown by Figure 4 can be sorted into eight categories, as follows:

Control Objects

- We have stated a lot on the control objects in our system in previous sections. From the system architecture view, the control objects including CarPositionControl, CarButtonControl, LanternControl, DoorControl, DriveControl, HallButtonControl and Dispatcher.
- All the control objects connect to the network, getting input messages from the network, and sending output messages to the network for other objects to use.
- Control objects controlling an system entity (such as the Doors and the Buttons) are connected to sensors and actuators, getting messages from sensors and send feedback to actuators to perform the control function.

Network

- All the control objects are connected with the communication network, which is modeled as the network class in the middle of the diagram. Network is the

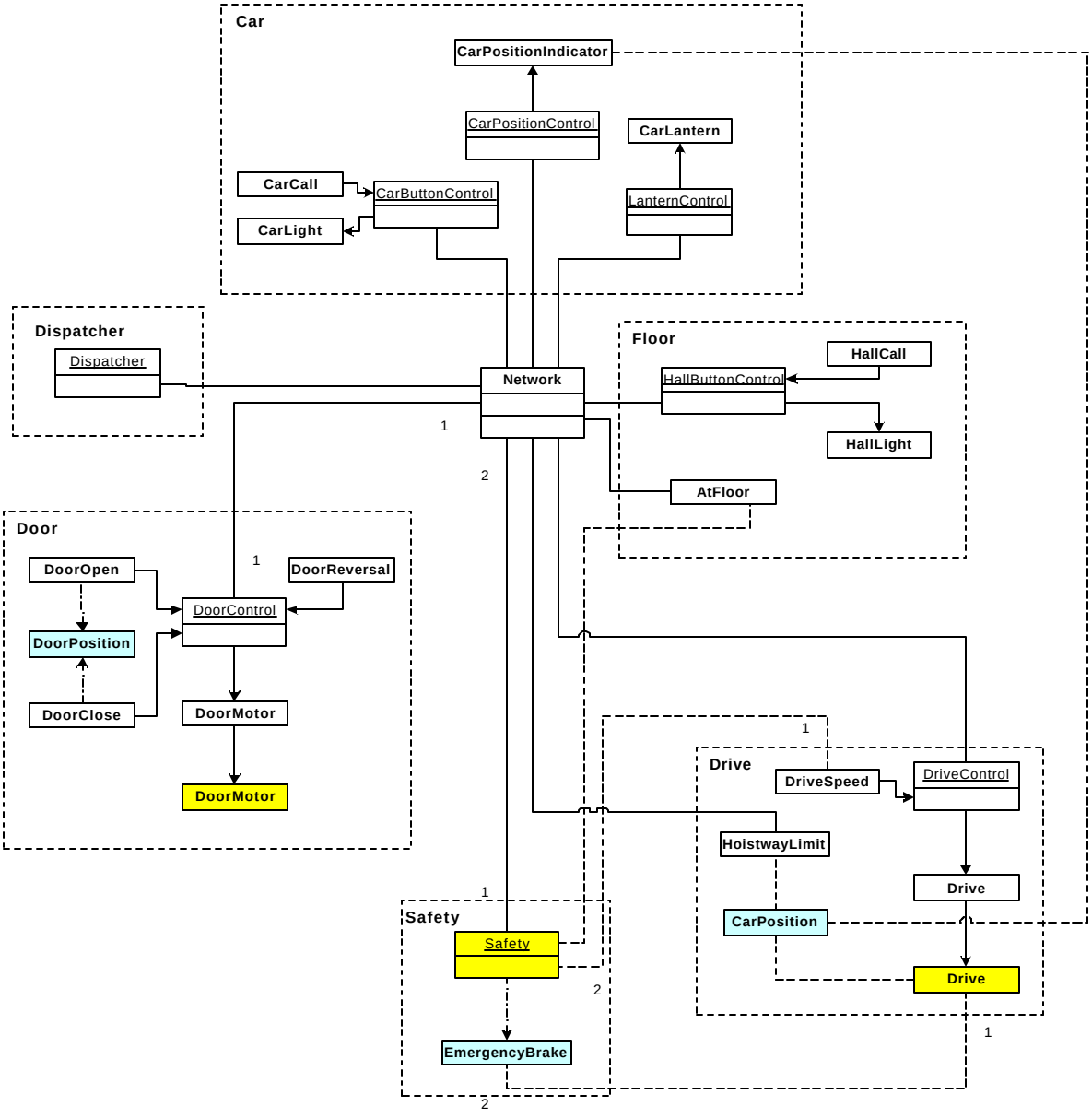

Figure 4: Class Diagram - the System Architecture View

System Sensors

- System values are available for use by the control system. On the class diagram the system sensors are connected with the control objects with an arrow pointing to the control objects.
- The system sensors in the class diagram include AtFloor, CarCall, DoorClosed, DoorOpen, DoorReversal, HallCall, and DriveSpeed.

- Except for AtFloor, all system sensors are connected to their control objects via physical network interface, the control objects get the message from both the sensors and the network to give out correct control messages.

Environment-only Sensors

- There are two environment-only sensors, DoorPosition and CarPosition in the system. They are illustrated with a dashed arrow line connected to the control objects.
- Environment-only sensors are pseudo-sensors and are not accessible to the control system, but used in the simulation.

System Actuators

- System actuators on the class diagram are connected with the control objects with an arrow pointing from the control objects.
- The system actuators in the class diagram include DoorMotor, CarLantern, CarLight, CarPositionIndicator, HallLight, and Drive.

Environme nt-only Actuators

- The EmergencyBrake is the environment-only actuator in the diagram. It is connected to the Safety object with dashed arrow line.

Environmental objects

- The Safety, Drive, and DoorMotor illustrated in the class diagram with shadow are environmental objects.
- The environmental objects are accessible to the control system indirectly via manipulation of actuators.

Grouped objects

- Grouped objects are Car, Door, Dispatcher, Drive, Floor, and Safety, each with a dashed box surrounded.
- The relationship of grouped objects in the system architecture is illustrated in Figure 5.
- Having a look at the very beginning, we can make a comparison of Figure 5 and Figure 2. We find that the object structure showed in Figure 5 is improved towards a more distributed system. Instead of having one central object taking care of every control task in the system, as implicated in Figure 2, each (group of) object has it own functional area and communicate and collaborate with other objects in the system. The cla ss diagram as we got in Figure 5 is an evolution of the original one, with the environmental class “Passenger” added to.

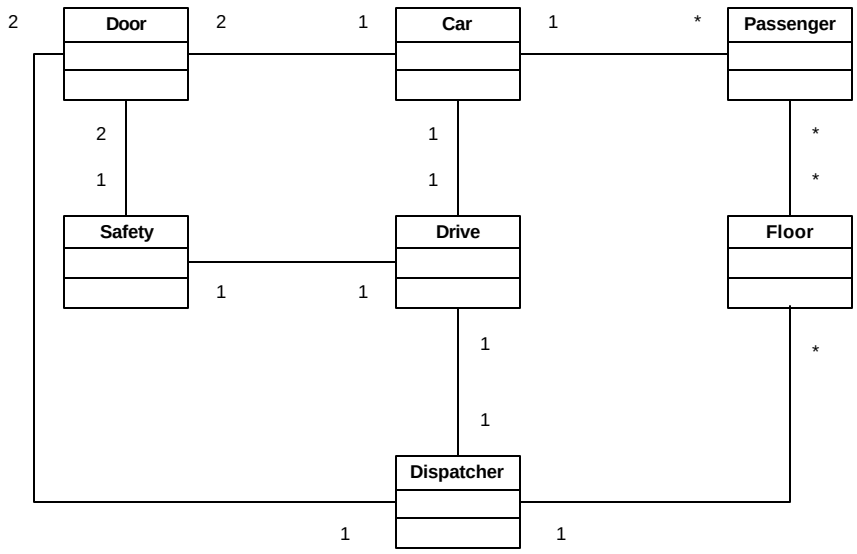

Figure 5: Class Diagram - the Revised Object Construction View

4.4 The Static Structure – Brief Summary

In section 4.3, three different class diagrams emphasizing on different views of the elevator system are given in an evolutionary way. Each of the three views captures one aspect of the system, and gives an overall understanding of the system design when combined together.

From the **view of object construction**, the class diagram describes a solution to the problem as the object architecture. The software design is captured by describing a collection of objects communicating and collaborating to implement a specific function. Objects communicate by sending messages to each other. Objects that share the same responsibilities are generalized into a class. The class diagram generated from this view grasps the main functional area of the system design, and gives a skeletal describing for the system.

From the **view of software architecture** , much more design and implementation details are captured. Based on the class diagram derived from this view, most of the future design of the software can be figured out.

The **system architecture view** provides the most complicated yet the most delicate description of not only the software, but also the structure of the whole system. Compared to normal software systems, it is very important to know in distributed embedded systems how the system components are working together.

After all, every class diagram is just a graphical presentation of the static design view of a system. No single class diagram can capture everything about a system's design view. Collectively, all the class diagrams of a system represent the system's complete static design view; individually, each represents just one aspect.

5. Modeling the dynamic aspects of the system

To model the dynamic aspects of a system, UML provides Sequence diagrams and Collaboration diagrams. In the context of this paper, only the Sequence diagrams for the elevator system are given, the collaboration diagrams can be derived from the sequence diagrams without too much effort.

State chart diagrams for the elevator system are also given in this section, based on the class project design of this semester. From our design practice, some experiential methods are given on how to migrate from the requirement to design.

5.1 Sequence Diagram

Sequence diagram is one kind of interaction diagrams, which shows an interaction among a set of objects and their relationships (another kind of interaction diagram is collaboration diagram). The purpose of the Sequence diagram is to document the sequence of messages among objects in a time based view. The scope of a typical sequence diagram includes all the message interactions for (part-of) a single use case. There may be multiple sequence diagrams per use case, one per use case scenario.

The state diagrams commonly contain:

- Objects
- Links
- Messages
- Respond Time (especially useful in real-time systems)

The vertical "lifelines" represents objects of interest. Messages are shown flowing between object lifelines. UML supports the notation of respond time in the sequence diagrams, which makes it feasible to specify the performance requirements for a real time system. Time flows from top to bottom.

In following sections, the objects in sequence diagrams are based on the class diagram from the software architecture view. The reason for doing that is we want to neither stay in the object construction view, in which the functions of objects are obscure and inadequate, nor go too further in the system architecture view, where many technical details obstruct a quick understanding of interaction among objects.

In some sequence diagrams the passenger appear to be an object of the system, since some of the messages are coming out from the passengers.

5.1.1 Use Case 1 – Process hall calls

There are two scenarios for this use case: when the passenger requests a hall call service by pressing hall call button(s), one scenario is that the elevator is moving the same direction as the passenger's destination, the other is vice versa. The two scenarios can share the same sequence diagram, the only difference is the driving time before the passenger could get on, i.e. the (x sec) in the diagram reflects the travel time of the elevator.

Scenario 1.1 Hall Call service – the elevator is moving towards the same direction as the passenger's destination.

Scenario 1.2 Hall Call service – the elevator is moving towards the opposite direction as the passenger's destination.

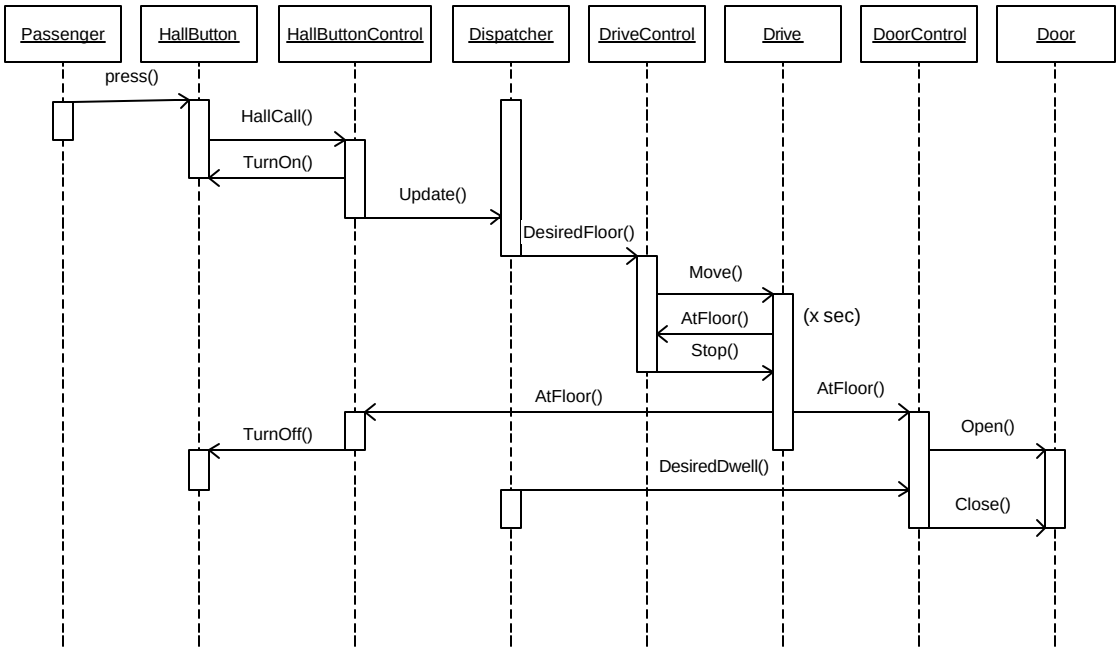

Figure 6: Scenario 1.1&1.2 - Hall Call Service

5.1.2 Use Case 2 – Process car calls

There are two scenarios for this use case: the passenger enters the car, presses a car call button. The passenger may either want to go to a upper floor or a lower one, depending on the current moving direction of the elevator, the passenger will either get to the destination floor when the elevator passes by it, or when the elevator turns around. Again, the two scenarios can share the

Scenario 2.1 Car Call service – the elevator is moving towards the same direction as the passenger's destination.

Scenario 2.2 Car Call service – the elevator is moving towards the opposite direction as the passenger's destination.

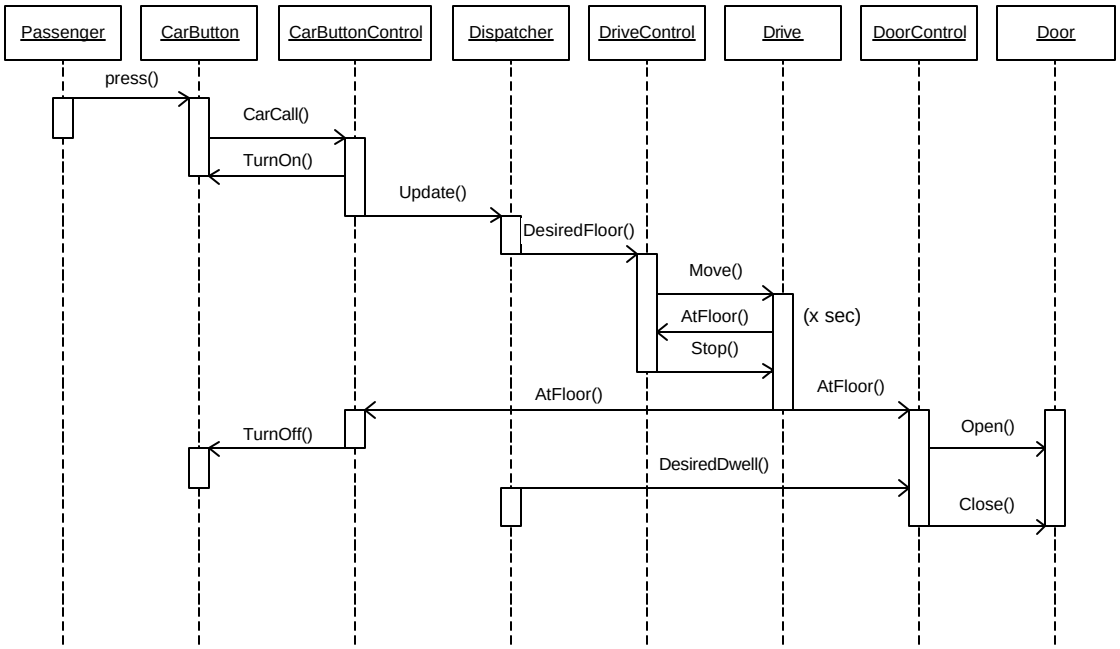

Figure 7: Scenario 2.1&2.2 - Car Call Service

5.1.3 Use Case 3 – Move/Stop the car

There are two scenarios for this use case:

Scenario 3.1&3.2 Move the car – the elevator is commanded to start moving from stop status. The moving direction and desired floor of the car are given by the Dispatcher. Within a safe scope, the car should move from slow speed to a fast speed. Scenario 3.1 is for moving Up, and Scenario 3.2 for moving Down.

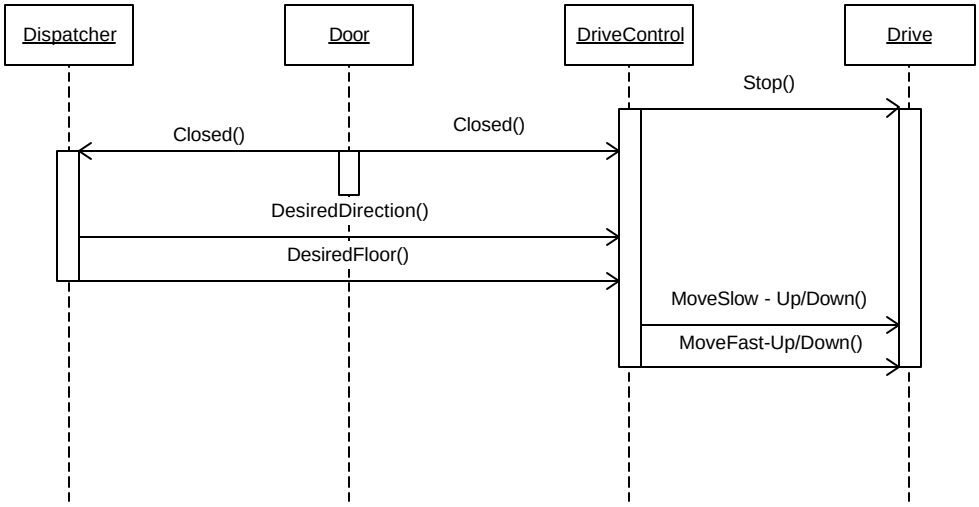

Figure 8: Scenario 3.1&3.2 - Moving the Car from Stop to Slow then to Fast

Scenario 3.3&3.4 Stop the car – when the elevator is approaching the desired floor, it should be commanded to slow down its drive speed, and at last stop at the floor.

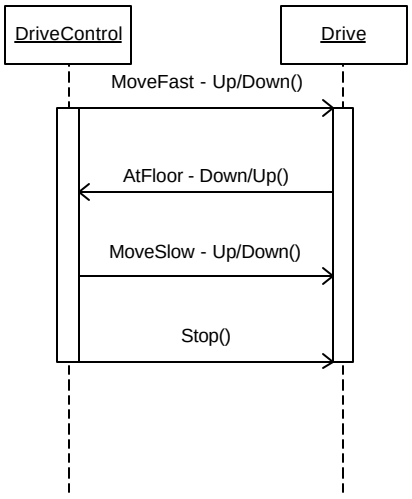

Figure 9: Scenario 3.3&3.4 - Moving the Car from Fast to Slow then to Stop

5.1.4 Use Case 4 – Indicate car position

There are two scenarios for this use case, which can share one sequence diagram:

Scenario 4.1 Indicating car position – whenever the doors of the elevator are open, the CarPositionIndicator should be commanded to illuminate to indicate the current car position.
Scenario 4.2 Finish indicating car position – when the doors are closed, the CarPositionIndicator should be commanded to indicate the desired floor.

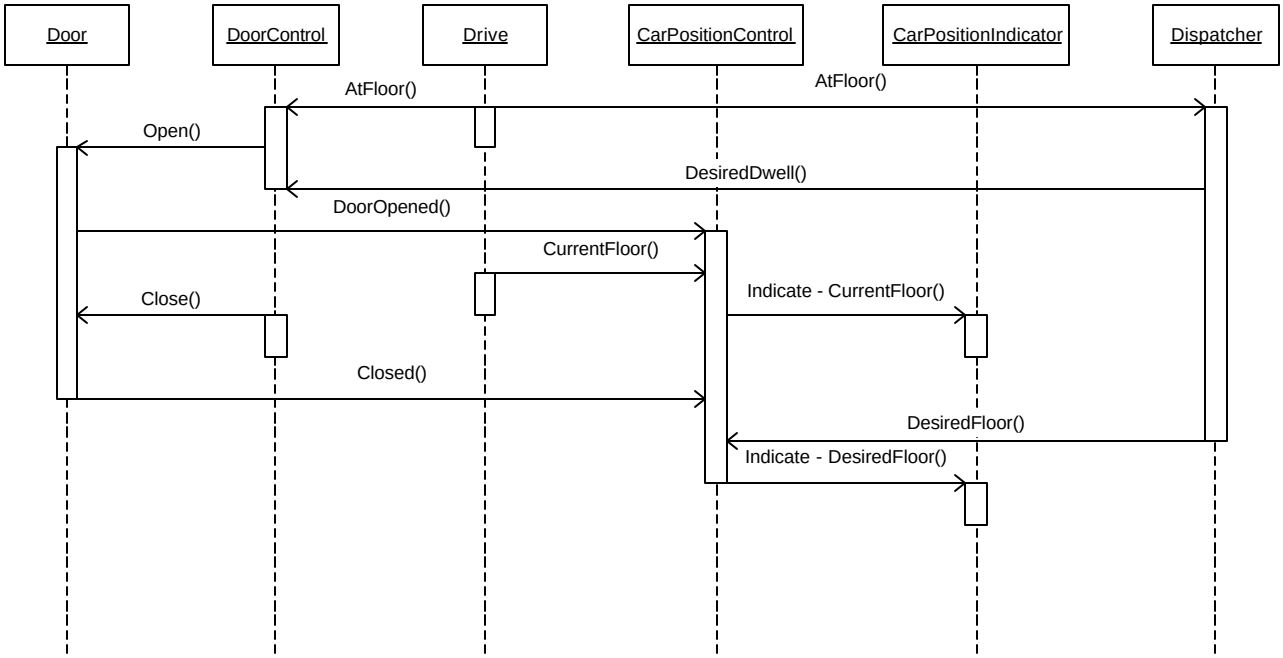

Figure 10: Scenario 4.1&4.2 - Indicating the Car Position

5.1.5 Use Case 5 – Indicate moving direction

There are two scenarios for this use case, which can share one sequence diagram:

Scenario 5.1 Indicating moving direction (up) – When the doors of the elevator are open and the desired direction of the car is UP, the UP CarLantern is illuminated. When the doors are closed, the CarLantern is turned off.

Scenario 5.2 Indicating moving direction (down) – When the doors of the elevator are open and the desired direction of the car is DOWN, the DOWN CarLantern is illuminated. When the doors are closed, the CarLantern is turned off.

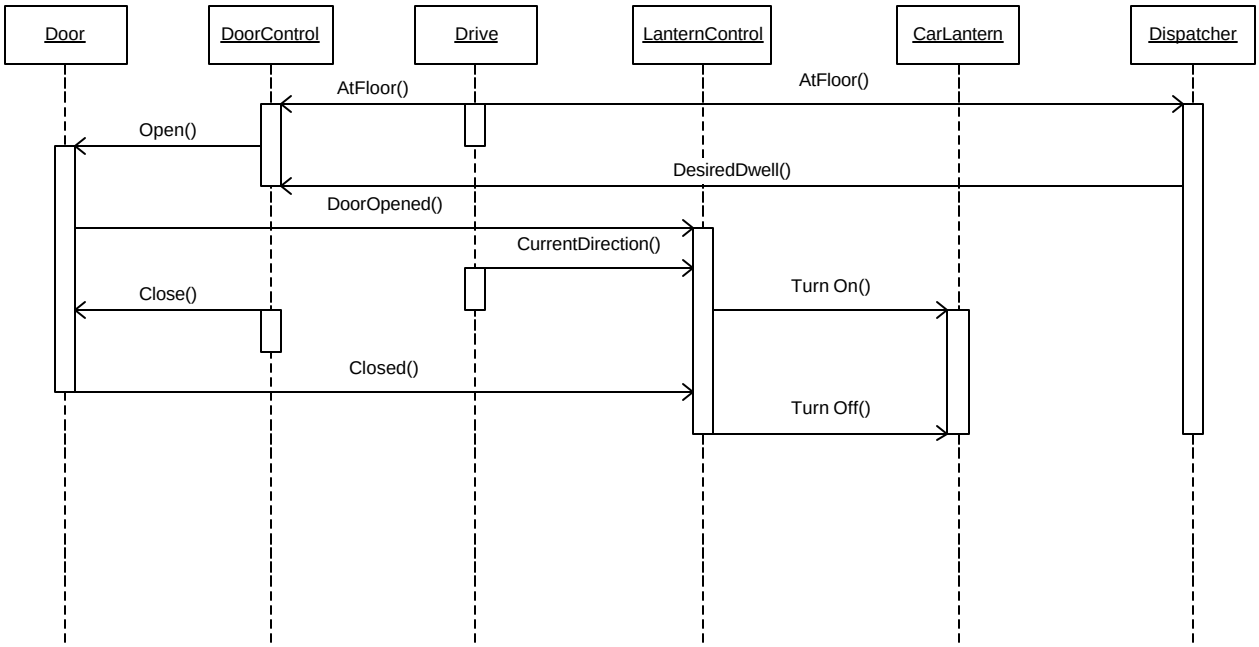

Figure 11: Scenario 5.1&5.2 - Indicating the Moving Direction

5.1.6 Use Case 6 – Open/Close the doors

There are three scenarios for this use case:

Scenario 6.1 Open the doors – When the car stops at a floor, the doors should open for a period of time (DesiredDwell), so that the passengers may get in the car.

Scenario 6.2 Close the doors – After opening for a specific period of time (Desiredperiod), the doors should close so that the car can move to the next destination.

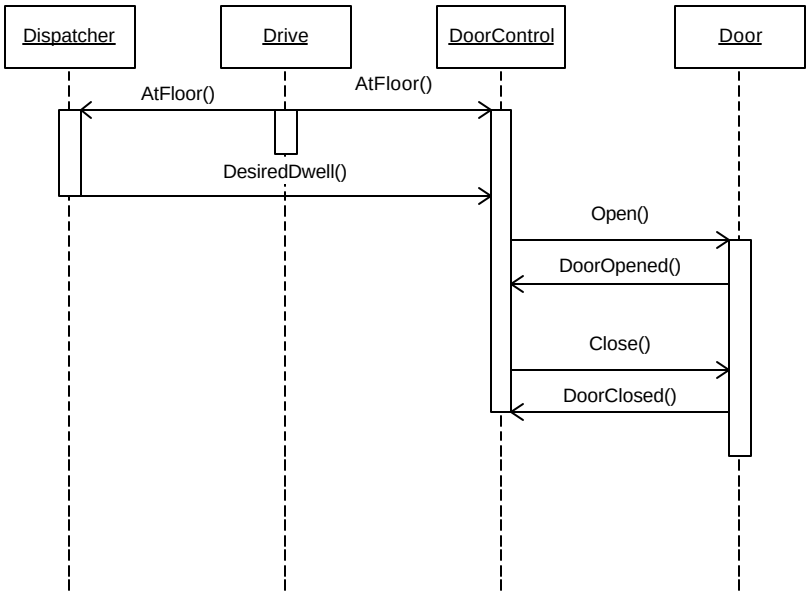

Figure 12: Scenario 6.1&6.2 - Open and Close the doors

Scenario 6.3 Door reversals – When the doors are closing but not fully closed, if there are passengers who want to get into the car, the doors should open again for another period of time, then close again.

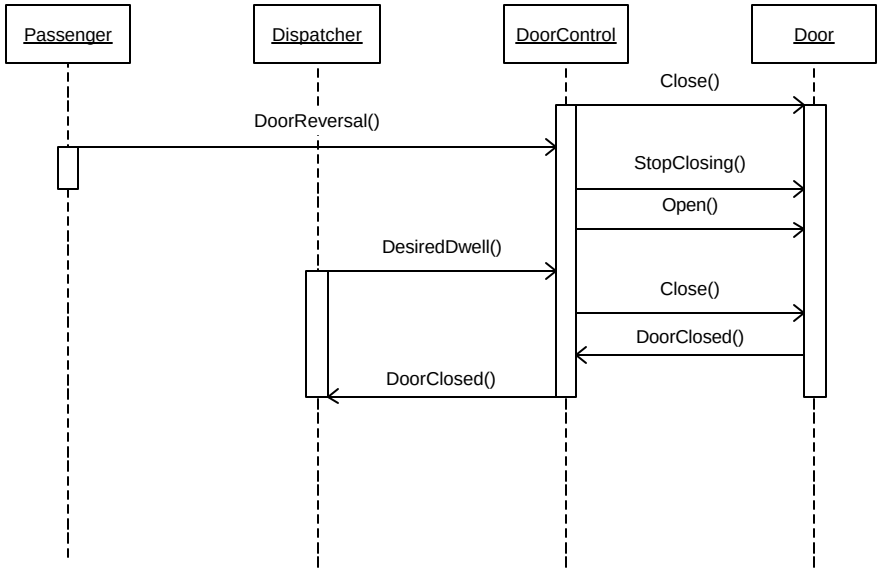

Figure 13: Scenario 6.3 - Door Reversal

5.1.7 Use Case 7 – Trigger emergency brake

There are five scenarios for this use case:

Scenario 7.1 Emergency Brake 1 – If the car is commanded to stop but it won't stop at a desired floor, the emergency brake will be triggered.

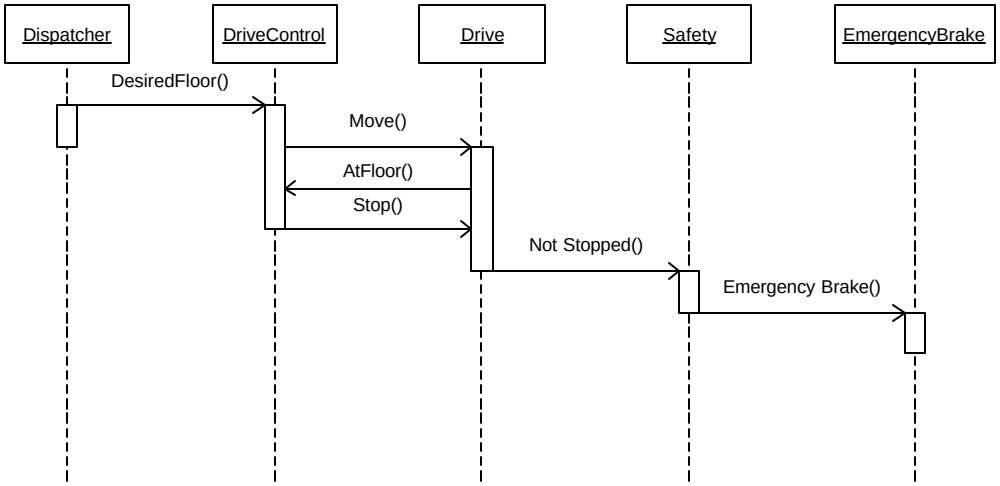

Figure 14: Scenario 7.1 - Emergency Brake - The car won't stop at desired Floor

Scenario 7.2 Emergency Brake 2 – If the car is commanded to move but it does not move, the emergency brake will be triggered.

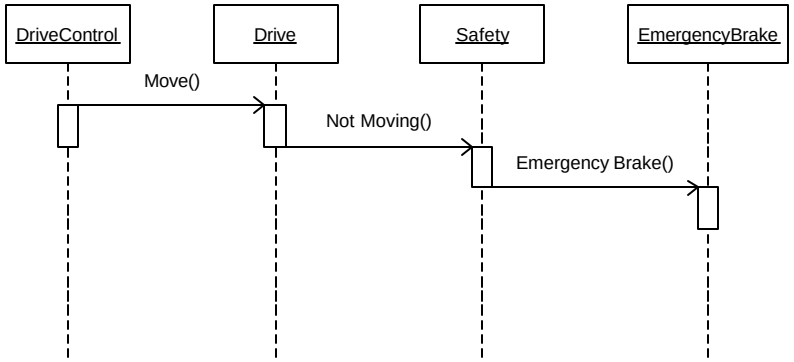

Figure 15: Scenario 7.2 - Emergency Brake - The car won't move

Scenario 7.3 Emergency Brake 3 – If the doors are commanded to open when the car stops at a floor, but the doors won't open, the emergency brake will be triggered.

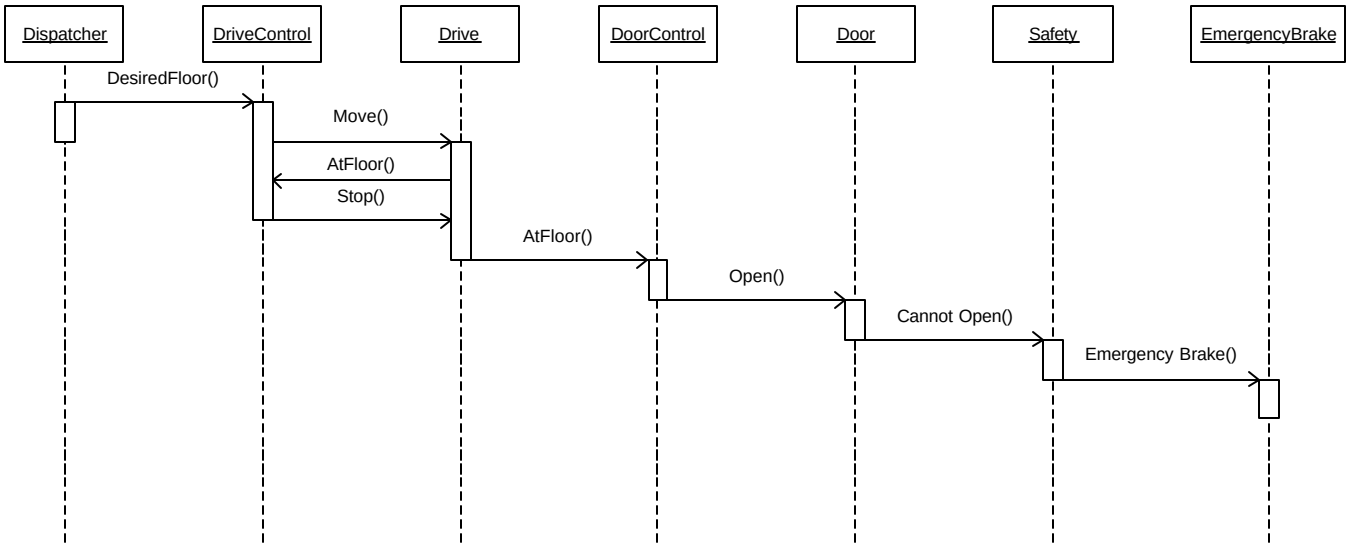

Figure 16: Scenario 7.3 - Emergency Brake - The doors won't open when the elevator stops at desired floor

Scenario 7.4 Emergency Brake 4 – If the doors open when the car is moving, the emergency brake will be triggered.

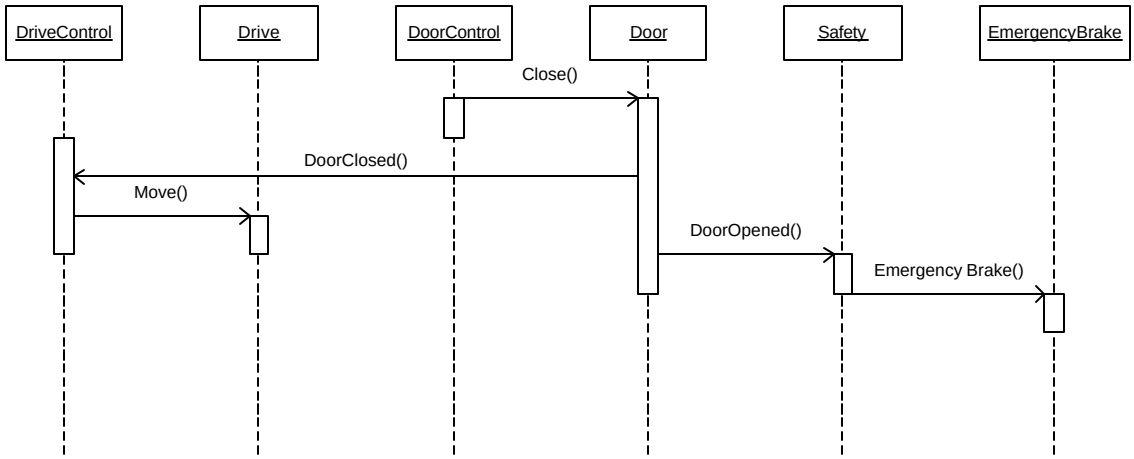

Figure 17: Scenario 7.4 - Emergency Brake - The doors open when the elevator is moving

Scenario 7.5 Emergency Brake 5 – If the car keeps going when the hoist way limit is reached, the emergency brake will be triggered.

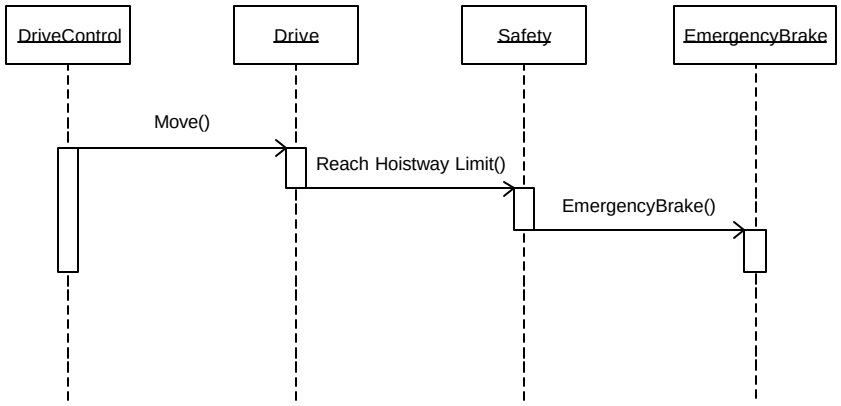

Figure 18: Scenario 7.5 - Emergency Brake - The elevator keeps going when hoistway limit is reached

5.2 State chart Diagram

A State chart diagram shows a state machine. Usually the state machine in a state chart models the behavior of a reactive object, whose behavior is best characterized by its response to events dispatched from outside its context. The object has a clear lifetime whose current behavior is affected by its past. State chart diagrams are important for constructing executable systems through forward and reverse engineering.

It is admitted that there exists a gap in the process of designing a system from requirements to state charts, not enough direction methods can be followed when drawing the state chart diagram from the requirements. In this section, some practical methods used during our designing the state charts for the elevator system are introduced. These methods may not be as serious as rules or instructions of how to draw state chart diagrams from the requirement document, but they are helpful in practice.

5.2.1 State chart for DoorControl

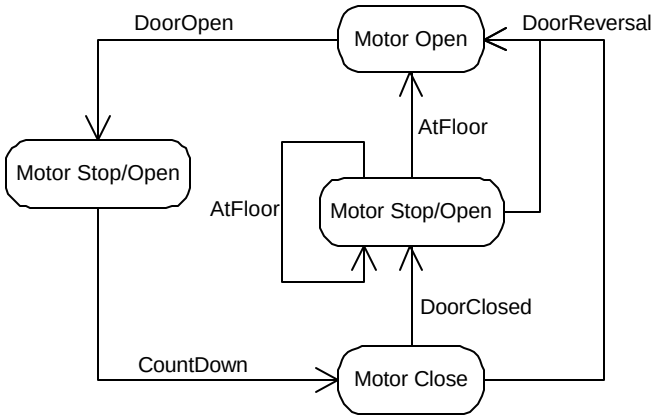

Figure 19: Statechart for DoorControl

5.2.2 State chart for DriveControl

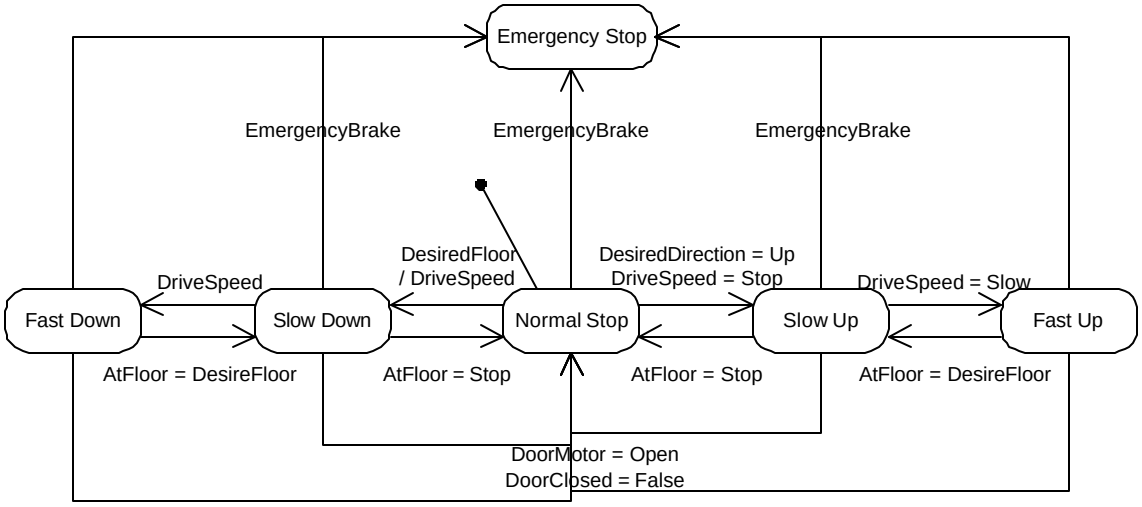

Figure 20: Statechart for DriveControl

5.2.3 State chart for LanternControl

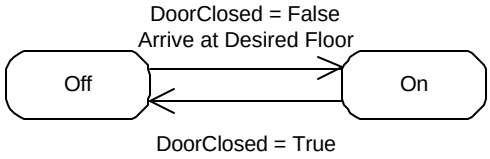

Figure 21: Statechart for LanternControl

5.2.4 State chart for HallButtonControl

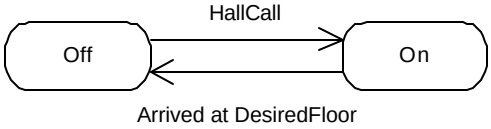

Figure 22: Statechart for HallButtonControl

5.2.5 State chart for CarButtonControl

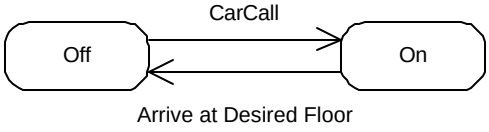

Figure 23: Statechart for CarButtonControl

5.2.6 State chart for CarPositionControl

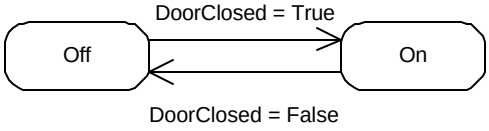

Figure 24: Statechart for CarPositionControl

5.2.7 State chart for Dispatcher

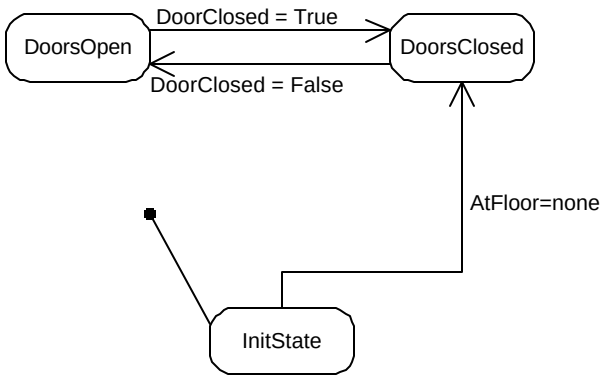

Figure 25: Statechart for Dispatcher

5.3 Practical methods that fill in the gap between requirements to state charts

State chart diagrams can model the behavior of a class, a use case, or the system as a whole. In the context of this paper, the state chart diagram is chosen to model the behaviors of each object in the system, in the periods later on, such as the implementation period and testing period, the state machine for each object will be used.

There is no detailed process in UML telling us how to draw state charts from the requirements documentation or UML diagrams such as class diagrams. From the experience on the course project, I am trying to summarize here some practical methods that are used when producing the state charts from the requirement documents, as follows:

Step 1: Make thorough analysis on the object structure and system architecture, if you want to draw state chart diagrams for each object. In the context of our class project, this part of work is well done by the instructors before we started working on the project by ourselves.

Step 2: For each class or object in the system, read the requirement carefully. Most of the information needed for the state charts can be found by this way, provided a good enough requirement document is already there. I am not sure whether there is any standard for the format of requirement documents. Taking the one used in our class for example, there are several items in the requirement document of our elevator system, on each one different attention should be paid.

- **Replication:** this part briefly declares the main function as well as the existence conditions of this specific object. This part doesn't help too much when drawing the state charts, but could be used to check when the state charts are finished whether the functions are fulfilled or not.

- **Instantiation:** The initial state can be chosen by the information given here. Take the HallButtonControl as an example, the Instantiation is "All HallCalls are false at

initialization" and "All HallLights are off at initialization". From these statements we can at least conclude that the initial state will at least be either in "Hall Call is False" or "Hall Light is off", but it's not complete, let's continue.

- **Input Interface:** The input messages that this object will get from other objects. The input variables will be the triggers of state changes, i.e. transitions in the state charts. In the HallButtonControl example, the change of the values of DesiredFloor and HallCall build the set of events triggering all the state changes in our future state chart.

- **Output Interface:** States what will the object being studied do to the outside world. The information here helps build what the states will be in the state chart. In our example, the HallLight is the single output this object deals with. We can think how many states could the HallLight be in. Intuitively, the Hall Light can be turned on and off, so the state in our future state chart will probably have two states: On and Off. Let's wait to see whether something else needs to be added.

- **State:** This state has nothing to do with the state chart, in this item some notation of variables used for shorthand in the description of behaviors.

- **Constraints and Behaviors:** Both of them will be used to check whether the state machine of this object fulfills the functional requirements of the system, at the same time does not break the constraints. What is important, we know how the state machine change are fired from the behaviors description.

Step 3: Now we have some ideas from the requirement items. The original state machine can be now generated. For the HallButtonControl, we have two states: "Hall Light is On" and "Hall Light is Off". From the information given in the **Instantiation** part, the initial state should be "Hall Light is Off". The behavior definition says "When HallCall[f, d] is True, command HallLight[f,d] to On", which makes up the first state change from On to Off; similarly, "If DesiredFloor.d is Stop, command both HallLight to Off" makes it change from On state back to Off. The state machine stays there waiting for a new hall call to come.

Step 4: Decide and add the pre-conditions, post-conditions, actions, entry codes and exit codes for the state machine we've got according to the Constraints and Behaviors from the requirement document.

Step 5: Check whether all the states are reachable under the combination of events.

Step 6: Check whether there is dead state that no (combination of) events can rescue the state machine from being stuck in that state.

Step 7: According to the behaviors, item by item, execute the state machine by hand, make sure all the requirement conditions are traversed and the state machine changes states, takes actions, modifies variables correctly. Make sure that nothing is missing or redundant.

Step 8: Draw out the state charts for each object, label the states, guard conditions, entry/exit codes and transitions correctly, record the corresponding requirement for tracing.

6 Conclusion

In this report, a detailed UML documentation for a simulated elevator control system is given. The UML diagrams used in this documentation are Use Case Diagram, Class Diagram, Sequence Diagram, and State Chart Diagram. Throughout the process of working on the class project, how can UML be used in real time systems are paid great attention to, the successful results of our project can be a good answer for the question.

Given the popularity and notational robustness of the current version of UML, OO technology can be reasonably exploited in real time developments. Current object-oriented analysis and design methods focus solely on the software of a system, which is not quite acceptable to real time systems, which are demanding a more pragmatic and comprehensive approach to system development, rather than just software. There are some aspects of real-time systems that need to be addressed:

- Definition of hardware elements and their characteristics;
- Definition of task and task communication;
- Time constraints;
- Modeling of the network.

UML, however, if used properly with attention paid to the real-time features of a system and combination of different notes, helps a lot the design and analysis of real-time system and can to some extend address above real-time system aspects.

To describe the hardware elements and model the network, we use three different views to model the structure of the system: the object construction and the software structure view focus on the software architecture of the system, while the system architecture view gives out an sketch of the system hardware and the communication method among system components. To describe the time constraints, UML supplies sequence diagrams and collaboration diagrams, which are able to specify the real-time feature of the system by way of marking time constraints aside by the names of message and object.

Every diagram in UML is just a graphical presentation of some of the aspects of a system. No single diagram could capture everything about a system's design view. The UML diagrams have to be combined to express a complete description of a real-time system. The three different views of class diagrams of the system can help to understand better the structural aspect of the system.

Some pragmatic methods are given in this paper based on my project experience, which may help filling the gap between requirement and design.

When creating the UML diagrams for the system, there already exist some components such as the system structure and the state charts. It is not clear whether in a normal analysis and design process, the methods concluded above will be still valid or not. For example, the system-architecture-viewed class diagram is based on the Elevator Architecture by Phil Koopman (attached to this report), in which no standard UML notation is used. The question here is: does UML notation good enough to design the system architecture provided that we don't have this architecture diagram?

The functions of the elevator system described in this paper are still limited to the class project, with little additional features that are more likely needed in the real world, such as a fire alarm

button or a fan lock. However, given the skeleton of the system, these additional functional modules can be added to both the static and the dynamic descriptions of the system, without too much effort.

7 Reference

[1] Hermann Kopetz. *Real-Time Systems, Design Principles for Distributed Embedded Applications*.
[2] Grady Booch, James Rumbaugh and Ivar Jacobson. *The Unified Modeling Language User Guide.*
[3] Perdita Stevens and Rob Pooley. *Using UML, Software Engineering with Objects and Components.*
[4] Martin Fowler and Kendall Scott. *UML Distilled, A Brief Guide to the Standard Object Modeling Language.*
[5] Bruce Powel Douglass. *Doing Hard Time: Developing Real-time Ssystems with UML, Objects, Frameworks, and Patterns.*
[6] Desmond F. D'Souza and Alan Cameron Wills. *Objects, Components, and Frameworks with UML.*
[7] Alan Moore and Niall Cooling. *Developing Real-Time Systems using Object Technology, A white paper from Artisan Software Tools.*