Towards an object-oriented analysis and design method for hardware/software systems : a case study

Hansen, Erik

Award date:
1995

EB569

Technische Universiteit Eindhoven

Faculty of Electrical Engineering
Section of Digital Information Systems

Master's Thesis:

Towards an Object-Oriented Analysis and Design Method for Hardware/Software Systems. A Case Study.

Erik Hansen

Coach	: Ing. P.H.A. van der Putten
	Ir. J.P.M. Voeten
Supervisor	: Prof. M.P.J. Stevens
Period	: Sept 1994 - June 1995

Towards an Object-Oriented Analysis and Design method for Hardware/Software Systems. A Case Study.

Erik Hansen

coaches : ing. P.H.A. van der Putten
ir. J.P.M. Voeten
supervisor : prof. M.P.J. Stevens

June 22, 1995

Abstract

For several years now, the Digital Information Systems Group of Eindhoven University of Technology is doing research on structured specification, analysis and design methods for embedded systems. Part of the research focuses on an object-oriented method for hardware/software systems. This analysis and design method—SHE—consists of models and a framework to guide modeling activities. Research is assisted with practical input from this masters project. In this masters project existing ideas and theories are tried out and new ones are applied to a real-world problem and evaluated.

Research to currently available object-oriented approaches indicated OMT—Object-oriented Modeling Technique—of Rumbaugh et.al. [14] as a relative suitable method. OMT is cited by many prominants in the field of system design and is taught on several universities. OMT models show three different but related views of a system: object, dynamic and functional view. OMT's design framework is divided in four phases: analysis, system design, object design and implementation.

As part of this masters project, OMT is applied to an elevator problem. The elevator problem contains considerable dynamic behavior and physical and abstract hierarchies. Applying OMT to the elevator problem showed that it was not as suitable for analysis and design of hardware/software systems as expected. Experiences with OMT were used to build a requirements list for a method for software/hardware engineering. The new method SHE—Software/Hardware Engineering—was checked with the requirements when it became available in a first form.

This thesis describes SHE as far as it has evolved. SHE is applied to the elevator problem as well. SHE showed to tackle several problems that were encountered with OMT, but has some problems of its own as well. These problems are analyzed and described in this thesis. They may serve as input for discussions and future research. SHE seems to be a very promising object-oriented method for analysis and design of complex hardware/software systems.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Why structured analysis and design?

Today's information processing systems are very complex and involve a mix of both hardware and software. The customer requests high quality at a reasonable price. Time to market is a key factor for the success or failure of a product in the market. Development time must be minimized and the ability of reuse of previously developed modules is necessary. Therefore, a structured analysis and design method for hardware/software co-design is required. This thesis describes the results of a masters project that is part of the research in modeling techniques at the Digital Information Systems Group of the Eindhoven University of Technology.

There are several reasons to analyze and design a system in a structured way with a particular method. In general, models of a system that are build according a method are an abstraction of the real-world. They enable communication with customers and engineers by highlighting the relevant aspects of a system and suppressing the insignificant ones. During analysis and specification, models help to understand the problem domain. During design, models help to build a correct design. Four of the most important reasons are:

- Models help to understand the problem domain.
- Models of a system are a communication aid.
- Models serve as a specification of the desired system.
- Operations on models enable correctness-preserving manipulations on a system specification.

A structured analysis and design method consists of: 1) means to build models, 2) a framework that guides how to use the models.

A method must cover all phases from specification to design. Phase transitions must be seamlessly from one phase to another. A system design starts with specification. From requirements of the customer, models of the desired system are build. The models enable communication with the customer. We can distinguish informal models and formal models. Informal models are easy to understand and easy to learn. However, informal models are often ambiguous, hard to keep consistent and allow context sensitive and discipline sensitive interpretation. On the other hand, a formal model is unambiguous, allows verification, simulation and correctness-preserving transformations. A method should have both formal and informal models.

Concepts like hierarchy, distribution and parallelism are very important to be able to handle complex systems. It is impossible to understand and manipulate a complex system in its entirety. Hierarchy offers a way to partition a system into smaller subsystems. Partitioning may be physical and logical. These smaller subsystems may operate in parallel with each other. A modeling technique must be able to describe parallelism between subsystems and within subsystems. For communication between subsystems, a variety of communication primitives are required.

Besides complexity management issues, a design method for complex systems has means for project management. Project management allows a team of engineers to work on a single system simultaneously. This should speed up system development. Another way to speed up system development and to reduce costs is reuse of previously designed modules.

1.2 Masters project

Before developing an analysis and design method for embedded information processing systems with a mix of both hardware and software existing methods have been studied at the Digital Information Systems Group of the Eindhoven University of Technology. This study learns us about a variety of methods and whether or not a new method is necessary. Existing methods can be categorized in their formal or informal basis, their approach as being data analysis, a functional approach, or an object oriented approach.

Encapsulation of operations and data into objects is one of the most powerful object oriented concepts to simplify models. It makes the object oriented paradigm very useful. Examples of currently available informal object oriented analysis methods that have been studied are Jacobson's OOSE [7], OMT of Rumbaugh et.al. [14], Hayes and Coleman [6], the Fusion method of Coleman [3]. Examples of formal approaches that have been studied are ObjLog [2], FOOPS [5], Maude [8], ROOA [10] and SYSDAX [11]. All of these methods are suitable for pure software. Currently, there are no object oriented methods for hardware/software co-design available.

Developing an analysis and design method involves a lot of study, creative thinking and fruitful discussions with colleagues. Besides theory, another very important input is practical experience with existing methods and new theories. This masters project supplies the research at the Digital Information Systems Group with practical input. As a starting point I studied OMT of Rumbaugh et.al. [14] and applied it to the elevator problem of Yourdon [18].

OMT—Object-Oriented Modeling Technique—is selected for our case study for the following reasons. This method is cited by many prominents in the field of system design and is taught on several universities. It is an analysis and design method for software systems. Its models show three different views of the system: A static structural view, a dynamic view and a functional view. OMT has a four phase design framework from analysis to implementation. The four phase are: analysis, system design, object design and implementation. Especially the existance of a system design phase makes OMT a suitable method for our case study.

We chose the elevator problem of Yourdon [18] for our case study because it has some very important properties. It is a problem with significant dynamic behavior. Hierarchy is a very important factor in the system, especially physical and logical hierarchies. It is a problem with enough complexity to be useful, and yet it still is very common system. Everybody knows what an elevator looks like, how it operates and what its prime functions are. And last but not least, it is an embedded system that combines both hardware and software.

Application of OMT on the elevator problem showed it is not as suitable for hardware/software systems as initially was thought. This experience lead to the development of a new specification, analysis and design method SHE—Software/Hardware Engineering—. SHE got shape through study to existing methods together with practical input from this masters project.

Pillar of SHE is communicating objects. SHE focuses on instances that send messages to each other. This instance approach is more appropriate for hardware/software co-design than a class approach. Hardware has a static structure and is not dynamically allocated like data objects in software. High level dynamic behavior of the system is embedded in message flows and their sequence. Functionality of a system is divided among objects in the system's model. Every object provides part of the functionality. Objects send messages as service requests or to transport information. Modeling a system integrates both formal and informal techniques. The SHE method falls apart in a framework with four quadrants:

- Essential behavior model
- Architecture structure model
- Implementation structure model
- Extended behavior model

The essential behavior model is in the first quadrant. This quadrant catches the essence of the system to be designed. Although many correct models of the same problem can be build, only one model will be built. It must be suitable for design and implementation. To find this model, iteration between the essential behavior model and the architecture structure model—the second quadrant—is necessary. High level structuring requirements are graphically represented in the architecture structure model. It describes structure and timing requirements, but no behavior. The acceptance of the specification, as modeled in these two quadrants, should be confirmed by a sign off by all relevant parties and experts. After the sign off, design starts in the third quadrant with the implementation structure model. This model visualizes the implementation that must be designed before the behavior description can be extended accordingly. The implementation topology is visualized and design decisions are motivated and described. The design decisions will result in additional objects and communications. The extended behavior model is based on the essential behavior model, but contains the additional objects and communications.

SHE described in [13] shows the state of SHE at februari 1995. SHE at this state is applied to the elevator problem as well. This application is an excercise of new ideas and theories. They have been evaluated and compared to experiences with OMT.

Summarizing, this thesis presents the results of the OMT exercise on the elevator problem. It describes what we learned and formulates requirements for the new method for software/hardware co-design of embedded systems. SHE is explained as far as it has evolved. It still is under development. SHE is applied to the elevator problem as well. An essential behavior model of the elevator problem is presented. Finally, SHE is reviewed. This review presents weak and strong points of SHE which may serve as input for future research.

1.3 How to read this thesis

This chapter—contains an introduction to the masters project and shows you how to read this report. The **second chapter** contains the elevator problem. It is almost an exact copy of the elevator problem in [18]. The method OMT is applied to the elevator problem in the succeeding two chapters. **Chapter 3** contains a summary of OMT [14]—Object Oriented Modeling Technique—. It is intended for those readers unfamiliar with OMT. Notation is explained in Appendix B. For more information see [14]. **Chapter 4** describes the OMT model of the elevator problem and how it is build. The model is analyzed according the heuristics and guidelines of Rumbaugh et.al. It concludes with a review. **Chapter 5** formulates the requirements for a new method. It is build on requirements from the experiences with building the OMT model of the elevator.

Chapter 6 describes SHE—Software/Hardware Engineering—. **Chapter 7** contains the application of SHE on the elevator problem. The review section of this chapter evaluates SHE and its application on the elevator problem. The POOSL description of the model is in Appendix A. The **last chapter** of this thesis draws conclusions of the case study.

Chapter 2

The elevator problem

2.1 A narrative description

The elevator problem is an example in [18]. This section presents the unmodified elevator problem from [18]. The problem gives rise to some hierarchy and substantial dynamic behavior. This makes the problem complex enough to test OMT's suitability for the analysis and design of embedded hardware/software systems.

The general requirement is to design and implement a program to schedule and control four elevators in a building with 40 floors. The elevators will be used to carry people from one floor to another in the conventional way.

The program should schedule the elevators efficiently and reasonably. For example, if someone summons an elevator by pushing the down button on the fourth floor, the next elevator that reaches the fourth floor traveling down should stop at the fourth floor to accept the passenger(s). On the other hand, if an elevator has no passengers—no outstanding destination requests—, it should park at the last floor it visited until it is needed again. An elevator should not reverse its direction of travel until its passengers who want to travel in its current direction have reached their destinations. As we will see below, the program cannot really have information about an elevator's actual passengers; it only knows about destination button presses for a given elevator. For example, if some mischievous or sociopathic passenger boards the elevator at the first floor and then presses the destination buttons for the fourth, fifth, and twentieth floor, the program will cause the elevator to travel to and stop at the fourth, fifth, and twentieth floors. The computer and its program have no information about actual passengers boarding and exiting. An elevator that is filled to capacity should not respond to a new summon request. There is an overweight sensor[1] for each elevator. The computer and its program can interrogate these sensors.

The interior of each elevator is furnished with a panel containing an array of 40 buttons, one button for each floor, marked with the floor numbers 1 to 40. These destination buttons can be illuminated by signals sent from the computer to the panel. When a passenger presses a destination button not already lit, the circuitry behind the panel sends an interrupt to the computer. There is a separate interrupt for each elevator. When the computer receives one of these vectored interrupts, its program can read the appropriate memory mapped eight-bit input register that contains the floor number corresponding to the destination button that caused the interrupt. There is one input register for each interrupt, hence one for each elevator.

[1]It is actually a "filled to capacity" sensor.

Of course, the circuitry behind the panel writes the floor number into the appropriate memory mapped input register when it causes the vectored interrupt. Since there are 40 floors in this application, only the first six bits of each input register will be used by the implementation; but the hardware would support a building with up to 256 floors.

As mentioned earlier, the destination buttons can be illuminated by bulbs behind the panels. When the interrupt service routine in the program receives a destination button interrupt, it should send a signal to the appropriate panel to illuminate the appropriate button. This signal is sent by loading the number of the button into the appropriate memory mapped output register. There is one such register for each elevator. The illumination of a button notifies the passenger(s) that the system has taken note of his or her request and also prevents further interrupts caused by additional (impatient?) pressing of the button. When the controller stops an elevator at a floor it should send a signal to its destination button panel to turn off the destination button for that floor.

There is a floor sensor switch for each floor for each elevator shaft. When an elevator is within eight inches of a floor, a wheel on the elevator closes the switch for that floor and sends an interrupt to the computer. There is a separate interrupt for the set of switches in each elevator shaft. When the computer receives one of these vectored interrupts, its program can read the appropriate memory mapped eight-bit input register that contains the floor number corresponding to the floor sensor switch that causes the interrupt. There is one input register for each interrupt, hence one for each elevator.

The interior of each elevator is furnished with a panel containing one illuminable indicator for each floor number. This panel is located just above the doors. The purpose of this panel is to tell the passengers in the elevator the number of the floor at which the elevator is arriving and at which it may be stopping. The program should illuminate the indicator for a floor when it arrives at the floor and extinguish the indicator when it arrives at a different floor. This signal is sent by loading the number of the floor indicator into the appropriate memory mapped output register. There is one output register for each elevator.

Each floor of the building is furnished with an panel containing summon button(s). Each floor except the ground floor—floor 1—and the top floor—floor 40—is furnished with a panel containing two summon buttons, one marked UP and one marked DOWN. The ground floor summon panel has only an UP button. The top floor summon panel has only a DOWN button. Thus, there are 78 summon buttons altogether, 39 UP buttons and 39 DOWN buttons. Would-be passengers press these buttons in order to summon an elevator. Of course, the would-be passenger cannot summon a particular elevator. The scheduler decides which elevator should respond to a summon request. These summon buttons can be illuminated by signals sent from the computer to the panel. When a passenger presses a summon button not already lit, the circuitry behind the panel sends a vectored interrupt to the computer. There is one interrupt for UP buttons and another for DOWN buttons. When the computer receives one of these two vectored interrupts, its program can read the appropriate memory mapped eight-bit input register that contains the floor number corresponding to the summon button that caused the interrupt. Of course, the circuitry behind the panel writes the floor number into the appropriate memory mapped input register when it causes the vectored interrupt.

The summon buttons can be illuminated by bulbs behind the panels. When the summon button interrupt service routine in the program receives an UP or DOWN button vectored interrupt, it should send a signal to the appropriate panel to illuminate the appropriate button. This signal is sent by the program's loading the number of the button in the appropriate memory mapped output register, one for the UP buttons and one for the DOWN buttons.

The illumination of a button notifies the passenger(s) that the system has taken note of his or her request and also prevents further interrupts caused by additional pressing of the button. When the controller stops an elevator at a floor, it should send a signal to the floor's summon button panel to turn off the appropriate UP or DOWN button for that floor.

There is a memory mapped control word for each elevator motor. Bit 0 of this word commands the elevator to go up, bit 1 commands the elevator to go down, and bit 2 commands the elevator to stop at the floor whose sensor switch is closed. The elevator mechanism will not obey any inappropriate or unsafe commands. If no floor sensor switch is closed when the computer issues a stop signal, the elevator mechanism ignores the stop signal until a floor sensor switch is closed. The computer program does not have to worry about controlling an elevator's doors or stopping an elevator exactly at a level—home position—at a floor. The elevator manufacturer uses conventional switches, relays, circuits, and safety interlocks for these purposes so that the manufacturer can certify the safety of the elevators without regard for the computer controller. For example, if the computer issues a stop command for an elevator when it is within eight inches of a floor so that its floor sensor switch is closed, the conventional, approved mechanism stops and levels the elevator at that floor, opens and holds its doors open appropriately, and then closes its door. If the computer issues an up or down command during this period, the manufacturer's mechanism ignores the command until its conditions for movement are met. Therefore, it is safe for the computer to issue an up or down command while an elevator's door is still open. One condition for an elevator's movement is that its stop button not be depressed. Each elevator's destination button panel contains a stop button. This button does not go to the computer. Its sole purpose is to hold an elevator at a floor with its door open when the elevator is currently stopped at a floor. A red emergency stop switch stops and holds the elevator at the very next floor it reaches irrespective of computer scheduling. The red switch may also turn on an audible alarm. The red switch is not connected to the computer.

The elevator scheduler and controller may be implemented for any contemporary microcomputer capable of handling this application.

Chapter 3

Object modeling technique summary

3.1 Introduction

This chapter gives a summary of the object modeling technique as presented in [14]. To preserve the intentions of the authors of [14], definitions and figures are kept unmodified.

A model is an abstraction of something for the purpose of understanding it before building it. Because a model omits nonessential details, it is easier to manipulate than the original entity. Abstraction is isolating those aspects that are important for some purpose and suppress those aspects that are unimportant. The goal of abstraction is the selective examination of certain aspects of a problem. Many different abstractions of the same thing are possible, depending on the purpose for which they are made. There is no single "correct" model of a situation, only adequate and inadequate ones.

The Object Modeling Technique [14]—OMT—is the name for a software development methodology that combines three different but related viewpoints of a system:

- Object model
- Dynamic model
- Functional model

The *object model* represents the static, structural, "data" aspects of a system. The *dynamic model* represents the temporal, behavioral, "control" aspects of a system. The *functional model* represents the transformational, "function" aspects of a system. Each model contains references to entities in other models. Each of the three models evolves during the development cycles. During analysis, a model of the application domain is constructed without regard for eventual implementation. During design, solution domain constructs are added to the model. During implementation, both application domain and solution domain constructs are coded.

3.2 Object modeling concepts

An object model captures the static structure of a system by showing the objects in the system, relationships between the objects, and the attributes and operations that characterize

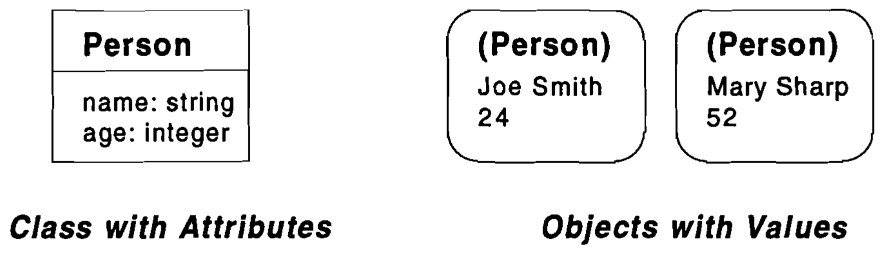


Figure 3.1: Class and objects

each class of objects. The object model is the most important of the three models. OMT emphasizes building a system around objects rather than around functionality, because an object oriented model more closely corresponds to the real world. An *object* is a concept, abstraction, or thing with crisp boundaries and meaning for the problem at hand. Decomposition of a problem into objects depends on judgment and the nature of the problem. Objects are distinguished by their inherent existence.

An *object class* describes a group of objects with similar properties (attributes) and common behavior (operations). A class serves as a template for a group of objects. Common definitions are stored once per class. An object is an instance of a class. An *attribute* is a data value held by the objects in a class. Different object instances may have the same or different values for a given attribute. An attribute should be a pure data value, not an object. Figure 3.1 shows some examples.

As you may have noticed, OMT has a different definition of *class* from what is generally referred to as class in the OO community. A *class* in OMT is a *class template*. In general, a *class* refers to the group of instances of a single *class template*.

An *operation* is a function or transformation that may be applied to or by objects in a class. Each operation has a target as an implicit argument. An operation may have arguments in addition to its target object.

Links and associations are the means for establishing relationships among objects and classes. A *link* is a physical or conceptual connection between object instances. A link shows a relationship between two (or more) objects. The link is not part of either object itself, but depends on both them together. An *association* describes a group of links with common structure. Associations are inherently bidirectional. The name of an association usually reads in a particular direction, but the binary association can be traversed in either direction. *Multiplicity* specifies how many instances of one class may relate to a single instance of an associated class. Multiplicity constrains the number of related objects. A *link attribute* is a property of the links in an association. A *role* is one end of an association. Each role on a binary association identifies an object or set of objects associated with an object at the other end. Use of role names provides a way of traversing associations from an object at one end, without explicitly mentioning the association. Figure 3.2 shows a simple class diagram.

Usually the objects on the "many" side of an association have no explicit order, and can be regarded as a set. Sometimes, however, the objects are explicitly ordered. In these cases, the ordering is an inherent part of the association and is indicated by writing "{ ordered }" on the many end of the association. A *qualified association* relates two object classes and a *qualifier*. The qualifier is a special attribute that reduces the effective multiplicity of

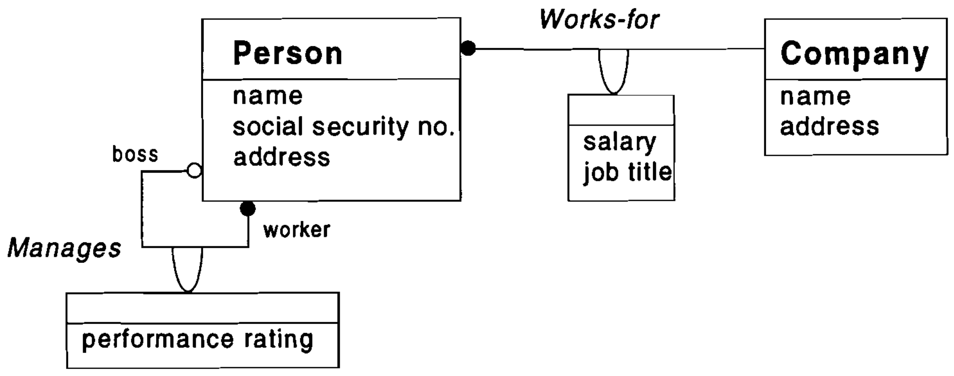


Figure 3.2: Simple class diagram

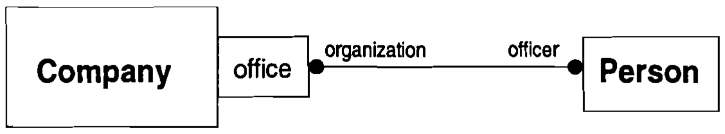


Figure 3.3: A qualified association

an association. The qualifier distinguishes among the set of objects at the many end of an association. Figure 3.3 shows a qualified association.

Aggregation is the "part whole" or "part of" relationship in which objects representing the components of something are associated with an object representing the entire assembly. The most significant property of aggregation is transitivity. It is antisymmetric as well. Some properties of the assembly propagate to the components, possibly with some local modifications. Unless there are common properties of components that can be attached to the assembly as a whole, there is little point in using aggregation.

Generalization and inheritance are powerful abstractions for sharing similarities among classes while preserving their differences. *Generalization* is the relationship between a class and one or more refined versions of it. The class being refined is called the *superclass* and each refined version is called a *subclass.* Each subclass is said to *inherit* the features—i.e. the attributes and operations—of its superclass. Generalization is sometimes called the "is a" relationship because each instance of a subclass is an instance of the superclass as well. The term *ancestor* and *descendant* refer to generalization of classes across multiple levels. Each subclass not only inherits all the features of its ancestors but adds its own specific attributes and operations as well. A subclass may override a superclass feature by defining a feature with the same name. The overriding feature refines and replaces the overridden feature. There are several reasons why you may wish to override a feature: to specify behavior that depends on the subclass, to tighten the specification of a feature, or for better performance. *Multiple inheritance* permits a class to have more than one superclass and to inherit features from all parents. A class with more than one superclass is called a *join class.* A feature from the same ancestor class found along more than one path is inherited only once; it is the same feature.

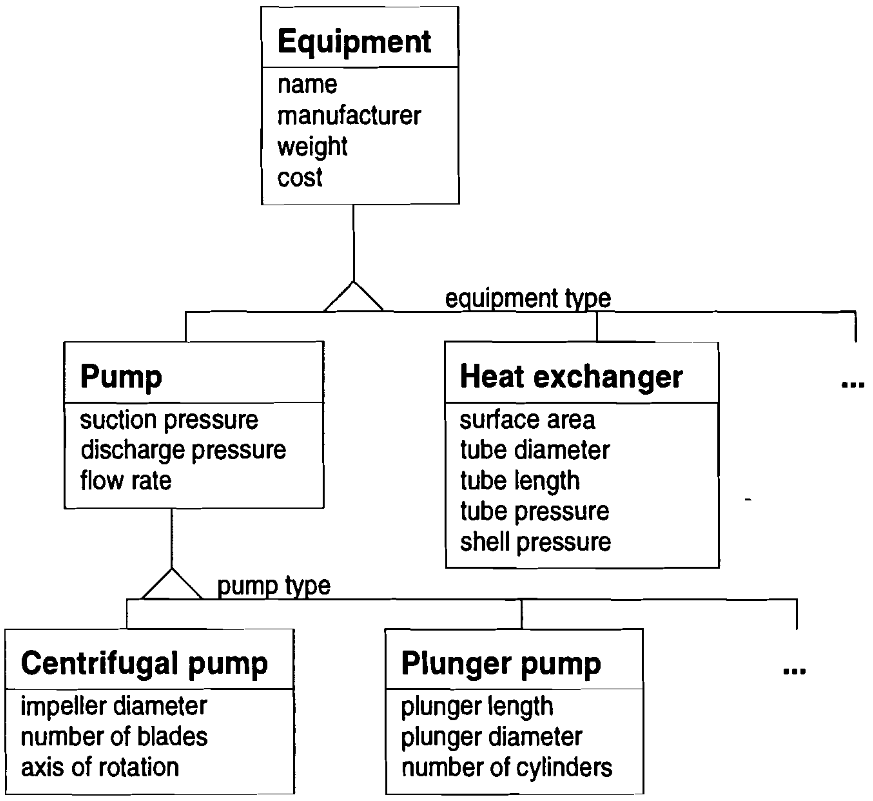


Figure 3.4: Multilevel inheritance

Figure 3.4 shows multilevel inheritance, while Figure 3.5 shows multiple inheritance.

3.3 Dynamic modeling concepts

The *dynamic model* captures those aspects of a system that are concerned with time and changes. The major dynamic modeling concepts are events, which represent external stimuli, and states, which represent values of objects. The attribute values and links held by an object are called its *state*. Over time, the objects stimulate each other, resulting in a series of changes to their states. An individual stimulus from one object to another is an *event*. The response to an event depends on the state of the object receiving it, and can include a change of state or the sending of another event to the original sender or to a third object. The pattern of events, states and state transitions for a given class can be abstracted and represented as a *state diagram*. The dynamic model consists of multiple state diagrams, one state diagram for each class with important dynamic behavior. Each state machine executes concurrently and can change state independently. The state diagrams for the various classes combine into a single dynamic model via shared events and thus shows the pattern of activity for an entire system.

An event is something that happens at a point in time. An event has no duration. Of course, nothing is really instantaneous; an event is simply an occurrence that is fast compared

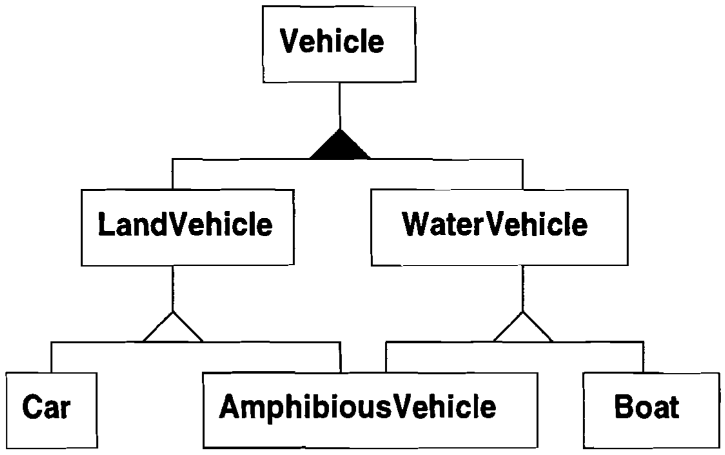


Figure 3.5: Multiple inheritance

to the granularity of the time scale of a given abstraction. An event is a one-way transmission of information from one object to another. It is not like a subroutine call that returns a value. Some events are simple signals, but most event classes have attributes indicating the information they convey. *Event classes* group events and give each event class a name to indicate common structure and behavior. This structure is a generalization hierarchy with inheritance of event attributes.

A state is an abstraction of the attribute values and links of an object. Sets of values are grouped together into a state according to properties that affect the gross behavior of the object. A state specifies the response of the object to input events. The response of an object to an event may include an action or a change of state by the object. A state is often associated with a continuous activity, or an activity that takes time to complete.

A *state diagram* relates events and states. The state diagram specifies the state sequence caused by an event sequence. If an object is in a state with more than one transition leaving it, then the first event to occur causes the corresponding transition to fire. If an event occurs that has no transition leaving the current state, then the event is ignored. State diagrams can represent one-shot life-cycles or continuous loops. One-shot diagrams represent objects with finite lives. See Figure 3.6. A one shot diagram has initial and final states. The initial state is entered on creation of an object; entering the final state implies destruction of the object.

A *condition* is a boolean function of object values. Conditions can be used as guards on transitions.

Operations attached to states or transitions are performed in response to the corresponding states or events. An *activity* is an operation that takes time to complete. An activity is associated with a state. Activities include continuous operations as well as sequential operations that terminate by themselves after an interval of time. A state may control an continuous activity that persists until an event terminates it by causing a transition from the state. If an event causes a transition from a state before a sequential activity is complete, then the activity is terminated prematurely. The lack of event labels at transitions indicate that the transition fires automatically when the activity in the state is complete. An *action* is an instantaneous operation. An action is associated with an event. An action represents

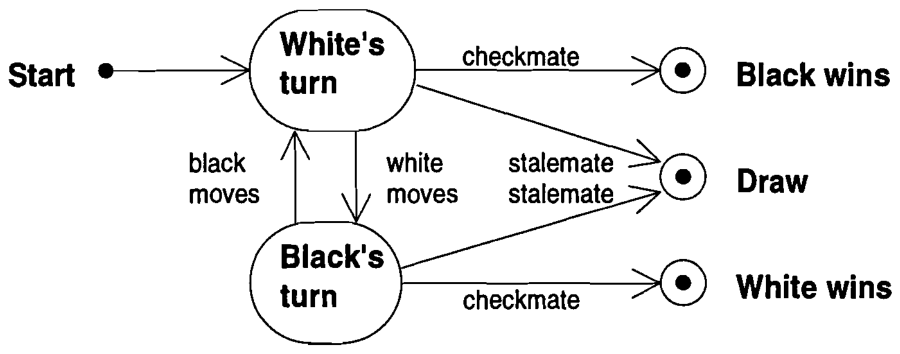


Figure 3.6: One-shot state diagram for chess game

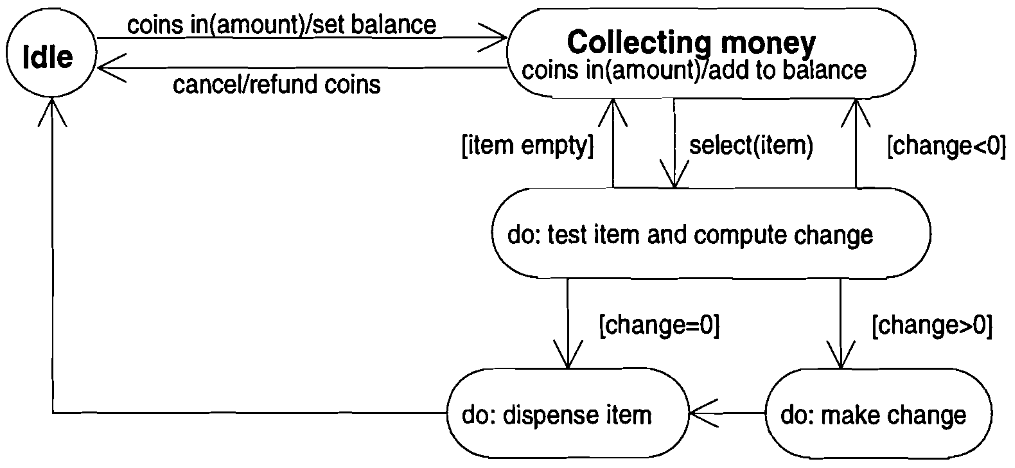


Figure 3.7: Vending machine model

an operation whose duration is insignificant compared to the resolution of the state diagram. An action could be setting attributes or generating other events.

An activity in a state can be expanded as a lower-level state diagram, each state representing one step of the activity. Nested activities are one-shot state diagrams with input and output transitions, similar to subroutines. Events can also be expanded into subordinate state diagrams. See Figure 3.7, 3.8 and 3.9.

A dynamic model describes a set of concurrent objects, each with its own state and state diagram. Concurrency within the state of a single object arises when the object can be partitioned into subsets of attributes or links, each of which has its own subdiagram. The state of the object comprises one state from each subdiagram. A transition to a state outside the composite state terminates all concurrent subdiagrams. However, merging of concurrent control is possible as well. For this purpose a forked arrow indicates the merged transition. Each subdiagram terminates as soon as its part of the transition fires, but all parts of the transition must fire before the entire transition fires and the composite state is terminated. The events need not be simultaneous.

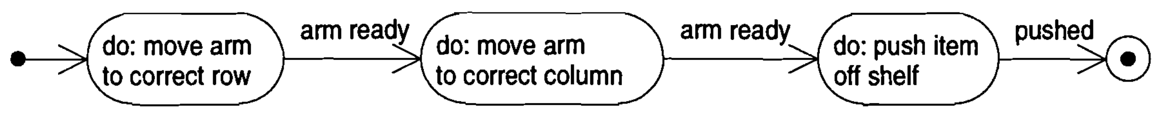


Figure 3.8: *Dispense item* activity of vending machine

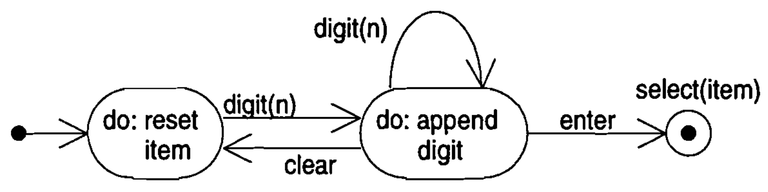


Figure 3.9: *Select item* transition of vending machine

The dynamic model of a class is inherited by its subclasses. The subclasses inherit both the states of the ancestor and the transitions. If the superclass state diagrams and the subclass state diagrams deal with disjoint sets of attributes, there is no problem. The subclass has a composite state composed of concurrent state diagrams. If, however, the state diagram of the subclass involves some of the same attributes as the state diagram of the superclass, a potential conflict exists. The state diagram of the subclass must be a refinement of the state diagram of the superclass. Usually, the state diagram of a subclass should be an independent, orthogonal, concurrent addition to the state diagram inherited from a superclass, defined on a different set of attributes.

3.4 Functional modeling concepts

The functional model describes computations within a system. The functional model specifies what happens, the dynamic model specifies when it happens, and the object model specifies what it happens to. It shows how output values in a computation are derived from input values, without regard for the order in which the values are computed. The functional model consists of multiple data flow diagrams. A data flow diagram is a graph showing the flow of data values from their sources in objects through *processes* that transform them to their destinations in other objects. A data flow diagram contains *processes* that transform data, *data flows* that move data, actor objects that produce and consume data, and data store objects that store data passively.

A process transforms data values. Each process has a fixed number of inputs an outputs. A process can have more than one output. A high level process can be expanded into an entire data flow diagram. A process is implemented as one or more methods of operations on object classes. The target object is usually one of the input flows, especially if the same class of object is also an output flow. A data flow connects the output of an object or process to the input of another object or process. It represents an intermediate data value within a computation. The value is not changed by the data flow. A data flow may generate an object that is used as a target of another operation.

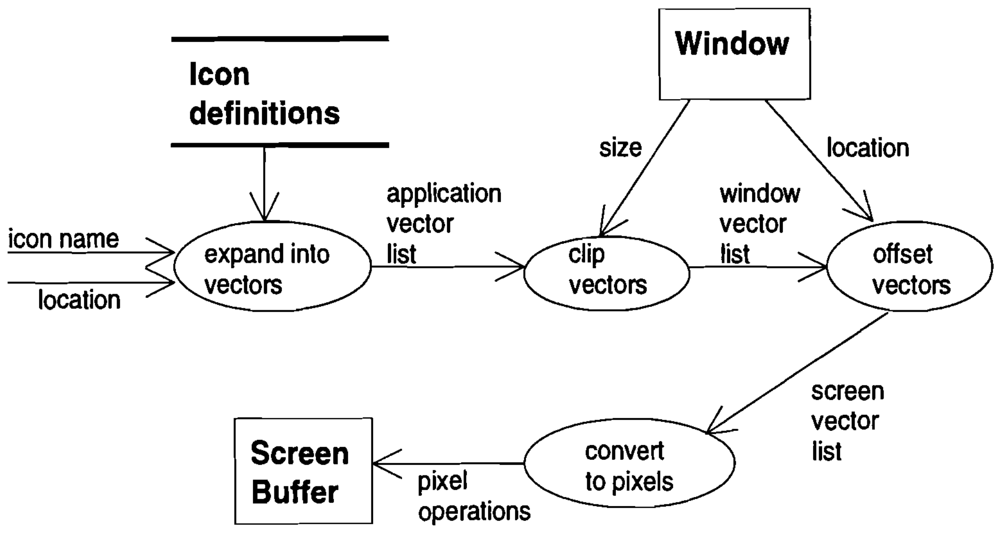


Figure 3.10: Data flow diagram for windowed graphics display

An *actor* is an active object that drives the data flow graph by producing or consuming values. In a sense, the actors lie on the boundary of the data flow graph but terminate the flow of data as sources and sinks of data, and so are sometimes called *terminators*. Actors are explicit objects in the object model. A *data store* is a passive object within a data flow diagram that stores data for later access. Data stores are also objects in the object model, or at least fragments of objects, such as attributes. Unlike an actor, a data store does not generate any operations on its own but merely responds to requests to store and access data.

The functional model shows what "has to be done" by a system. The leaf processes are the operations on objects. The object model shows the "doers"—the objects. Each leaf process is implemented by a method on some object. The dynamic model shows the sequences in which the operations are performed. The three models come together in the implementation of methods. The functional model is a guide to the methods.

3.5 Design methodology

The steps of software production are usually organized into a life cycle consisting of several phases of development. The complete software life cycle spans from initial formulation of the problem, through analysis ,design, implementation, and testing of the software, followed by an operational phase during which maintenance and enhancement are performed. The OMT methodology supports the entire software life cycle. Although the description of the Object Modeling Technique is of necessity linear, the actual development process is iterative. The methodology has the following stages:

1. *Analysis*: Starting from a statement of the problem, the analyst must work with the requester to understand the problem because problem statements are rarely complete or correct. The analysis model is a concise, precise abstraction of what the desired system must do, not how it will be done. The analysis model should not contain any implementation decisions.

2. *System design*: The system designer makes high-level decisions about the overall architecture. During system design, the target system is organized into subsystems based on both the analysis structure and the proposed architecture. The system designer must decide what performance characteristics to optimize.

3. *Object design*: The object designer builds a design model bases on the analysis model but containing implementation details. The designer adds details to the design model in accordance with the strategy established during system design. The focus of object design is the data structures and algorithms needed to implement each class.

4. *Implementation*: The object classes and relationships developed during object design are finally translated into a particular programming language, database, or hardware implementation. Programming should be a relatively minor and mechanical part of the development cycle, because all of the hard decisions should be made during design.

3.6 Comments after reading OMT

3.6.1 Conceptual remarks

The OMT model consists of three different views of a system: the object model, the dynamic model and the functional model. OMT is quite clear about the relation between the object model and the dynamic model. There is dynamic behavior description for every class template[1] in the object model. The functional model describes the functions that are performed by the system. The processes in the functional model are implemented as methods of operations in the object model. The mapping from processes in the functional model to methods of operation of class templates in the object model and operations/events in the dynamic model however, is not very clear. It may take several methods of operations on different class templates to implement a process. The relation between the functional model and object/dynamic model is not very clear.

During system design, the overall structure and style are decided. The system architecture is the overall organization of the system into subsystems. The decomposition of systems into subsystems may be organized as a sequence of horizontal layers and/or vertical partitions. The layers are an ordered set of virtual worlds at different levels of abstraction. The partitions vertically divide a system into several independent or weakly-coupled subsystems. Usually only the top and bottom layers are specified by the problem statement: The top is the desired system, the bottom is the available resources. If the disparity between the two is too big, then the system designer must introduce intermediate layers to reduce the conceptual gap between adjoining layers. The usefulness of a design method depends on its ability to support the system designer in filling the gap. OMT, however, only gives a list of suggestions how to divide a system into subsystems. Though the authors pretend to present a seamless design method, they do not use the three models made during analysis. Instead, totally out of the blue, the system design of a very important example in [14]—the automatic teller machine—is presented. Maybe the example is a very simple one, but to divide system into subsystems the system designer must be able to determine what the effect of his decisions are on, for example, performance. OMT does not show how to calculate or guess system performance.

[1]A *class* in OMT is a *class template*

These major limitations indicate that OMT can not be used as a combined hardware/software design tool yet.

The object model captures the static structure of a system. It is a diagram that contains class templates instead of objects. The dynamic model describes aspects of a system that are concerned with time and changes. There is a state diagram for every class template. The functional model describes computations within a system. The functional model is a guide to the implementation of methods. The models are built on class templates. OMT does not model a particular system, but a class of systems, because there are no instances of class templates in the system model. The three models serve as a system template. A real system however, consists of objects which perhaps have finite life time. A model of a particular system should therefore contain instances of class templates and a concept to model objects with finite life time. The OMT object model shows how to create an instance and what its possible links are to other instances. OMT has two types of object diagrams: class template diagrams and instance diagrams. OMT does not allow both class templates and objects in a single diagram.

3.6.2 Modeling remarks

Here is a list of some modeling remarks on OMT:

- OMT supplies two concepts two model a hierarchy in the object model. One is the inheritance hierarchy and the other is the aggregation hierarchy. Aggregation is the "part of" relationship in which objects representing the components of something are associated with an object representing the entire assembly. Unless there are common properties of components that can be attached to the assembly as a whole, there is little point in using aggregation. Unfortunately, the authors do not show how these common properties are modeled except for propagation of operations.

- An object is defined as a concept, abstraction, or thing with crisp boundaries and meaning for the problem at hand. Each object has its own state diagram. An event generated at one object could trigger state transitions of several other objects. The dynamic interaction between objects is through the use of the same event name at different locations in the dynamic model. This way, events are globally defined. The dynamic model does not have crisp boundaries as the object model has. The collection of state diagrams with globally used event names make it difficult to understand the dynamic behavior of a system.

- An individual stimulus from one object to another is an event. The attribute values and links held by an object are called its state. In the ATM example of [14] however, the authors use a broader definition of event and state. The dynamic model of the class template ATM contains many transitions, triggered by events from the userinterface, though the userinterface is not defined in the object model. Secondly, the state diagram of class template ATM contains more states then can be accounted for by differences in attribute values and links. Incomplete examples in [14], make OMT as a methodology hard to understand.

- The functional model consists of multiple data flow diagrams. A data flow diagram is a graph showing the flow of data values from their sources in objects through processes that transform them to their destinations in other objects. A data flow diagram does not show control information. Decisions and sequencing are control issues that are part of the dynamic model. A decision affects whether one or more functions are even performed, rather than supplying a value to the functions. Even though the functions do not have input values from these decision functions, it is sometimes useful to include them in the functional model so that they are not forgotten and so their data dependencies can be shown. This is done by including *control flows* in the data flow diagram. So, sometimes a data flow diagram does show control information even though the functional model should not contain control information. Control information in the functional model duplicates information of the dynamic model and will be prone to inconsistencies.

Chapter 4

An OMT model of the elevator problem

4.1 Initial elevator model

Analysis, the first step of the OMT methodology, is concerned with devising a precise, concise, understandable, and correct model of the real world. Before building anything complex, the builder must understand the requirements and the real-world environment in which it will exist. Analysis cannot always be carried out in a rigid sequence. Large models are build iteratively. First a subset of the model is constructed, then extended, until the complete problem is understood.

The first step in analyzing the requirements is to construct an object model. The object model describes real-world object classes and their relationships to each other. The object model precedes the dynamic model and functional model because static structure is usually better defined, less dependent on application details, more stable as the solution evolves, and easier for humans to understand. Information for the object model comes from the problem statement, expert knowledge of the application domain, and general knowledge of the real-world. It is best to get ideas down on paper before trying to organize them too much, even though they may be redundant and inconsistent, for not to loose important details. The following steps are performed in constructing an object model:

- Identify objects and classes
- Prepare a data dictionary
- Identify associations between objects
- Identify attributes of objects and links
- Organize and simplify object classes using inheritance
- Iterate and refine the model
- Group classes into modules

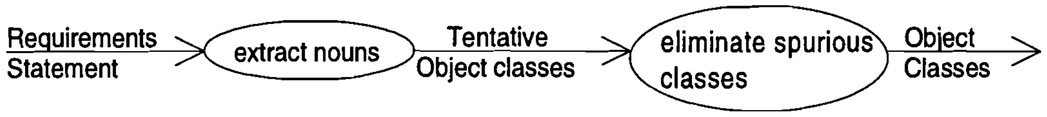


Figure 4.1: Identifying object classes

Destination	DownButton	Manufacturer
DestButton	StopButton	Passenger
DestButPanel	StopSwitch	Program
DestButPanCircuitry	ArrivalLight	Computer
DestButInputReg	ArrLightOutputReg	ControlWord
DestButOutputReg	ArrLightPanel	Door
Summons	Building	OverweightSensor
SumButton	TopFloor	ElevatorCage
SumButPanel	Floor	ElevatorMechanism
SumButPanCircuitry	BottomFloor	AudibleAlarm
SumButInputReg	FloorSensor	Interrupt
SumButOutputReg	FloorSensInputReg	IntServiceRoutine
UpButton	TargetMachine	

Table 4.1: Candidate object classes

4.1.1 Identifying object classes

Objects include physical entities as well as concepts, such as trajectories, seating assignment, and payment schedules. All classes must make sense in the application domain; computer implementation constructs must be avoided.

Classes often correspond to nouns. Searching for classes really is a search for objects. Every object belongs to some class. A class can have many instances or even just the one object found. Without being too selective, Table 4.1 contains a list of object classes that comes to mind when reading the problem statement.

This list probably isn't complete, because it only contains classes that appear directly in the problem statement. Some classes may be overlooked or it may be necessary to put classes in the object model that come from common knowledge of the problem domain. This should not be a problem, since modeling is an iterative process. The next step is to discard unnecessary and incorrect classes.

This list shows what classes are discarded and why:

- DestButPanCircuitry is redundant. The distinction between DestButPanCircuitry and DestButPanel doesn't seem to effect analysis. DestButPanCircuitry is merged into DestButPanel. SumButPanCircuitry is merged into SumButPanel as well.
- DestButInputReg, DestButOutputReg, SumButInputReg, SumButOutputReg, ArrLightOutputReg, FloorSensInputReg, ControlWord, Interrupt and IntServiceRoutine are implementation specific and should be eliminated from the analysis model. They may be needed

Destination	DownButton	FloorSensor
DestButton	StopButton	Passenger
DestButPanel	StopSwitch	Computer
Summons	ArrivalLight	OverweightSensor
SumButPanel	ArrLightPanel	ElevatorCage
UpButton	Floor	ElevatorMechanism

Table 4.2: Initial object classes

later during design, but not in this stage of analysis.

- SumButton is an abstract class. SumButton is a generalization of UpButton and DownButton. Inheritance is postponed until a more stable version of the object model is present.
- Building and TargetMachine are redundant. They are objects representing an assembly in an aggregation. Their components have no common properties in the problem domain, so there is no point in using an aggregation.
- Manufacturer is irrelevant. It has little to do with the problem.
- TopFloor and BottomFloor are specializations of Floor. At this time there is no need to make a distinction between an "ordinary" floor and a top or bottom floor. Also, inheritance is postponed until a more stable version of the object model is present.
- Door and AudibleAlarm are irrelevant. They are encapsulated in the object ElevatorMechanism. Door and AudibleAlarm are invisible for the Computer and its program. Therefore, they have no effect on the analysis.
- Program is redundant. The distinction between Program and Computer doesn't seem to effect analysis. Program is merged into Computer.

Table 4.2 shows the resulting initial object class list.

4.1.2 Identifying associations

Any dependency between two or more classes is an association. A reference from one class to another is an association. Associations show dependencies between classes at the same level of abstraction as the classes themselves. Associations often correspond to stative verbs or verb phrases. These include physical location, directed actions, communications, ownership, or satisfaction of some condition. Candidate associations can be extracted from the problem statement or directly by examining the relationship between classes from the class list. In the elevator problem, the latter is preferred since the number of classes is small. Table 4.3 contains a list of candidate associations.

Next step is to discard unnecessary and incorrect associations, but let us first take a closer look at the classes StopButton, StopSwitch and Passenger. The classes StopButton and StopSwitch are irrelevant. They are connected directly to the ElevatorMechanism and have nothing to do with the problem of designing and implementing a program to schedule and

DestButPanel, Destination	The passenger uses the panel to supply the system with his destination
DestButPanel, DestButton	Aggregation
Computer, OverweightSensor	Computer interrogates the sensor
Computer, DestButPanel	Computer reads destinations from the panel
Computer, ElevatorCage	Computer controls the movement of the cage
Computer, ElevatorMechanism	Computer issues commands to the mechanism
Computer, Summons	Computer keeps a list of summons
Computer, SumButPanel	Computer reads summons from the panel
Computer, FloorSensor	Sensor sends interrupts to the computer
Computer, ArrLightPanel	Computer sends signals to the panel
ElevatorCage, Destination	Destinations of a passenger automatically becomes a destination of the cage
ElevatorCage, DestButPanel	Aggregation
ElevatorCage, ElevatorMechanism	Aggregation: The mechanism is part of the cage. The mechanism controls the movement of the cage
ElevatorCage, StopButton	Aggregation
ElevatorCage, StopSwitch	Aggregation
ElevatorCage, Passengers	There are passenger in the cage
ElevatorCage, Floor	Cage stops at a floor
ElevatorCage, FloorSensor	Cage closes floorsensor
ElevatorCage, ArrLightPanel	Aggregation
ElevatorMechanism, StopButton	Button is connected to the mechanism
ElevatorMechanism, StopSwitch	Switch is connected to the mechanism
StopSwitch, AudibleAlarm	Switch turns alarm on
SumButPanel, DownButton	Aggregation
SumButPanel, UpButton	Aggregation
SumButPanel, Summons	The passenger uses the panel to supply the system with his summons
Passenger, Destination	Passenger has a destination
Passenger, DestButton	Passenger depresses button
Passenger, Summons	Passenger summons elevator
Passenger, SumButton	Passenger depresses button
Floor, Summons	There is a summon request at a particular floor
Floor, SumButPanel	Aggregation
Floor, Passenger	A passenger is at a floor and wants to travel to a different floor
Floor, FloorSensor	A floorsensor goes with a particular floor
ArrLightPanel, ArrivalLight	Aggregation

Table 4.3: Candidate associations

DestButPanel, Destination	ElevatorCage, FloorSensor
DestButPanel, DestButton	ElevatorCage, ArrLightPanel
Computer, ElevatorCage	SumButPanel, DownButton
Computer, Summons	SumButPanel, UpButton
ElevatorCage, OverweightSensor	SumButPanel, Summons
ElevatorCage, Destination	Floor, Summon
ElevatorCage, DestButPanel	Floor, SumButPanel
ElevatorCage, ElevatorMechanism	Floor, FloorSensor
ElevatorCage, Floor	ArrLightPanel, ArrivalLight

Table 4.4: Initial associations

control four elevators. The class **Passenger** seems to be very important. If it wasn't for the passengers there would be no elevator at all. However, the program cannot really have information about an elevator's actual passengers; it only knows about destination button presses for a particular elevator and summon button presses. Therefore, the class **Passenger** doesn't effect analysis. It can be discarded to simplify the model.

This list shows what associations are discarded and why:

- If one of the classes in the association has been eliminated, then the association must be eliminated. All associations with class **StopButton, StopSwitch** and **Passenger** must be eliminated.
- **Computer, OverweightSensor**: is a derived association if we give **ElevatorCage** a boolean attribute which shows if it is filled to capacity. The **Computer** can query this attribute.
- **Computer, DestButPanel**: is a derived association. The **Computer** communicates with **ElevatorCage**. **DestButPanel** is a part of **ElevatorCage**.
- **Computer, ArrLightPanel**: is a derived association. The **Computer** communicates with **ElevatorCage**. **ArrLightPanel** is a part of **ElevatorCage**.
- **Computer, ElevatorMechanism**: is a derived association. The **Computer** communicates with **ElevatorCage**. **ElevatorMechanism** is a part of **ElevatorCage**.
- **Computer, SumButPanel**: is a derived association. The **Computer** communicates with **Floor**. **SumButPanel** is a part of **Floor**.
- **Computer, FloorSensor**: is a derived association. The **Computer** communicates with **Floor**. **FloorSensor** is a part of **Floor**.

Table 4.4 shows what is left. Figure 4.2 shows the initial object model.

4.1.3 Refining the object model

The elevator object model doesn't feel right. There is an asymmetry in associations between **Computer, Summons** and **Destination**. It is a good idea to model **Destination** as an association between **ElevatorCage** and **Floor** instead as an object. An **ElevatorCage** has zero, one or many

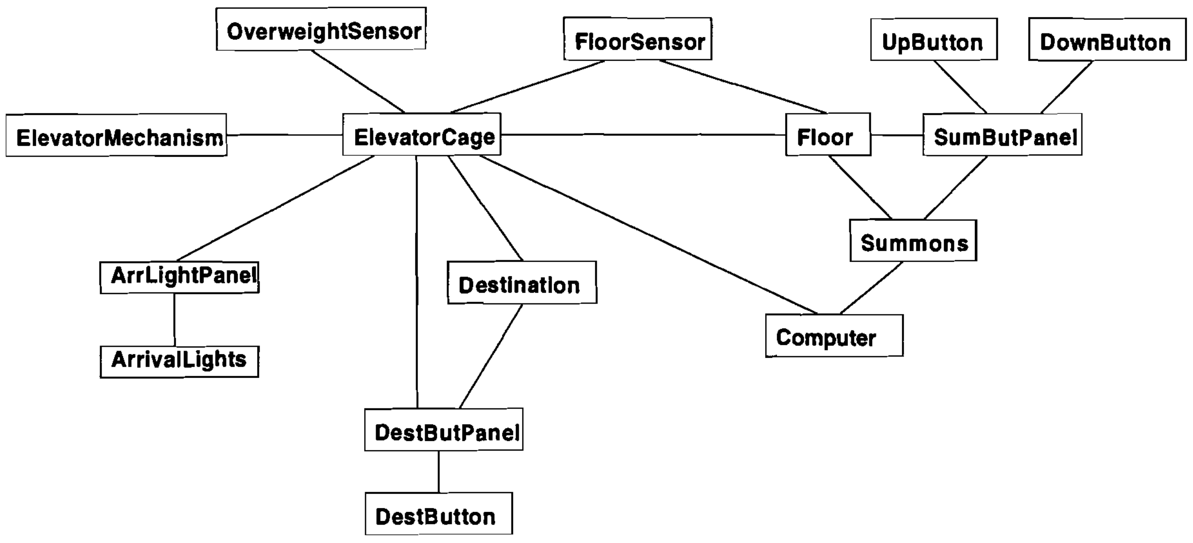


Figure 4.2: Initial object model

Floors as its destination. Another association between ElevatorCage and Floor is the "is at" relationship. Summons can not be modeled as an association. It is an object associated with a particular Floor. A Summons is a request for an elevator. The scheduler decides which elevator should respond to a summon request. The association between Computer and Summons can be discarded. It is a derived association. Computer communicates with Floor to see if there are any summon requests pending at a floor.

Refining the object model includes adding attributes of objects and associations, add multiplicity to associations, identify aggregations, qualifications. These small changes in the object model don't need a detailed explanation. Figure 4.3 reflects the object model up to this point.

Next step is to organize classes by using inheritance to share common structure. Inheritance can be discovered by searching for classes with similar attributes, associations, or operations. In the elevator problem, the object classes FloorSensor and OverweightSensor are both sensors with boolean output. Class BooleanSensor can serve as a generalization of FloorSensor and OverweightSensor. Similarly, class ButPanel is a generalization of DestButPanel and SumButPanel, and class Button is a generalization of UpButton, DownButton and DestButton. Figure 4.4 shows the generalization tree. Usually, the generalization tree is drawn in the object model, together with "ordinary" associations and aggregations. Because inheritance is a distinct type of associations, it is presented here in a separate figure. Notice that with the generalization tree, the association between DestButPanel and DestButton is not clear from the diagram. This applies for SumButPanel and its Buttons as well. It is probably best to leave inheritance out of the object model, because it does not add relevant or descriptive information to the models.

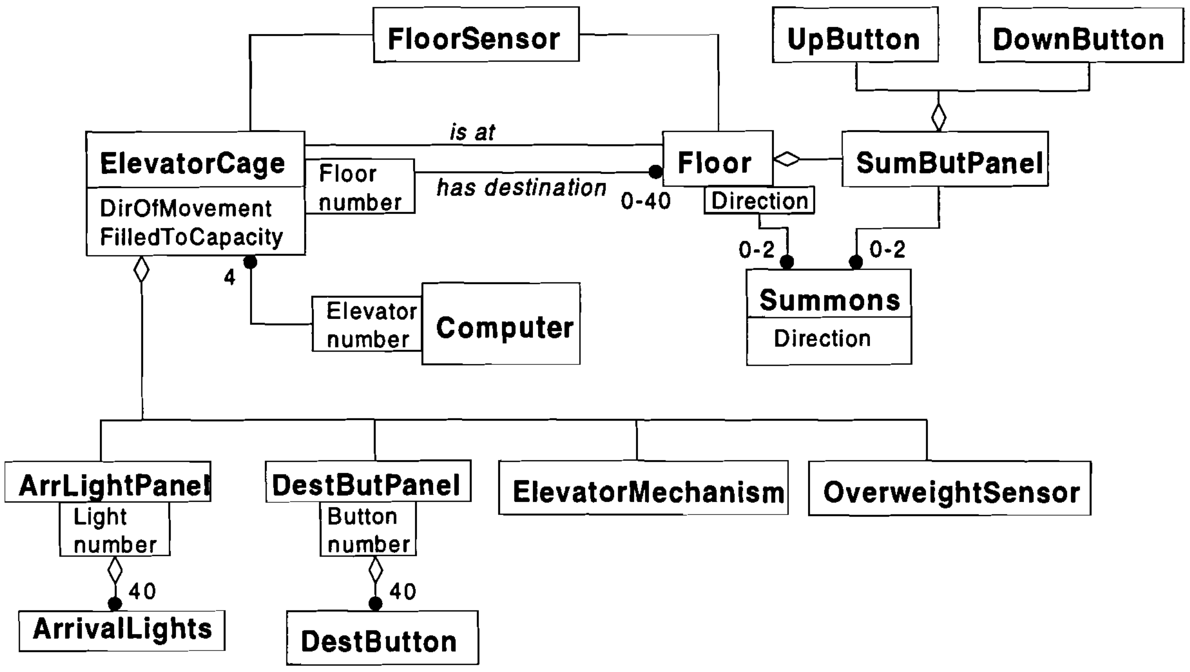


Figure 4.3: Refined object model

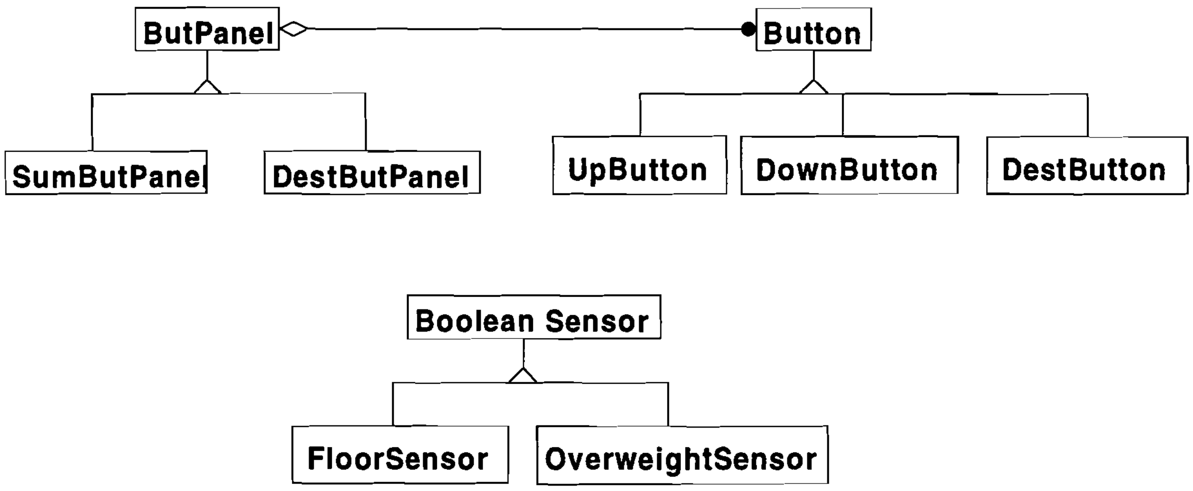


Figure 4.4: Inheritance part of the object model

4.1.4 Dynamic model

The dynamic model shows the time-dependent behavior of the system and the objects in it. First step is to prepare scenarios of typical dialogs. Even though these scenarios may not cover every contingency, they at least ensure that common interactions are not overlooked. Extract events from the scenarios. Organize the sequences of events and states into a state diagram. Finally compare state diagrams for different objects to make sure that the events exchanged by them match. The resulting set of state diagrams constitute the dynamic model. In summary, the following steps are performed in constructing the dynamic model:

- Prepare scenarios of typical interaction sequences
- Identify events between objects
- Prepare an event trace for each scenario
- Build a state diagram
- Match events between objects to verify consistency

Most object classes in the elevator problem have little dynamic behavior. There state diagrams are very simple. The state diagrams of these object classes can be constructed without the use of scenarios. Figure 4.5 shows the state diagrams. The dynamic model of a class is inherited by its subclasses. Subclass DestButton inherits the state diagram of Button. Subclasses FloorSensor and OverweightSensor inherit the state diagram of BooleanSensor. Note however that the inherited statediagrams differ in event and state names, and additional actions are attached to transactions.

There is not a dynamic model for class ElevatorMechanism. The dynamic behavior of class ElevatorMechanism is invisible. The manufacturer of the ElevatorMechanism designed and guarantees the internal dynamic behavior. The ElevatorMechanism accepts commands to control the movement of the elevator. These commands are predefined events.

SumButPanel is a "part-whole" class of an aggregation. Its state is composed of the states in the sub state diagrams. Class DestButPanel, ArrLightPanel, Floor and ElevatorCage are "part-whole" classes as well. Their state diagrams aren't drawn because they are simple aggregations, without additional dynamic behavior for the "part whole". Classes UpButton and DownButton inherit the state diagram of Button. Object class Summons does not have any dynamic behavior except for its one shot life cycle.

Most of the dynamic complexity is in the object class Computer. The Computer accepts summon and destination requests, schedules the elevators and issues control commands to them. The model of an ElevatorCage inside the Computer has an idle state. In this state the ElevatorCage is waiting at a Floor with its doors open. If a passenger depresses a SumButton or DestButton a transition may occur from the idle state to a state in which the ElevatorCage is moving with pending summon or destination requests in its direction of movement. If all requests in the direction of movement have been handled, the Computer checks to see if there are requests in the opposite direction. The ElevatorCage changes direction or stays at a Floor with its doors opened. Figure 4.6 shows a sketch of the state diagram. The Computer state diagram is composed of this state diagram and two state diagrams to handle summon and destination requests as in Figure 4.7. In the state diagrams event generalization is used. The event "sum. or dest. button depressed" is the ancestor of "sum. button up depressed" and "sum. button down depressed".

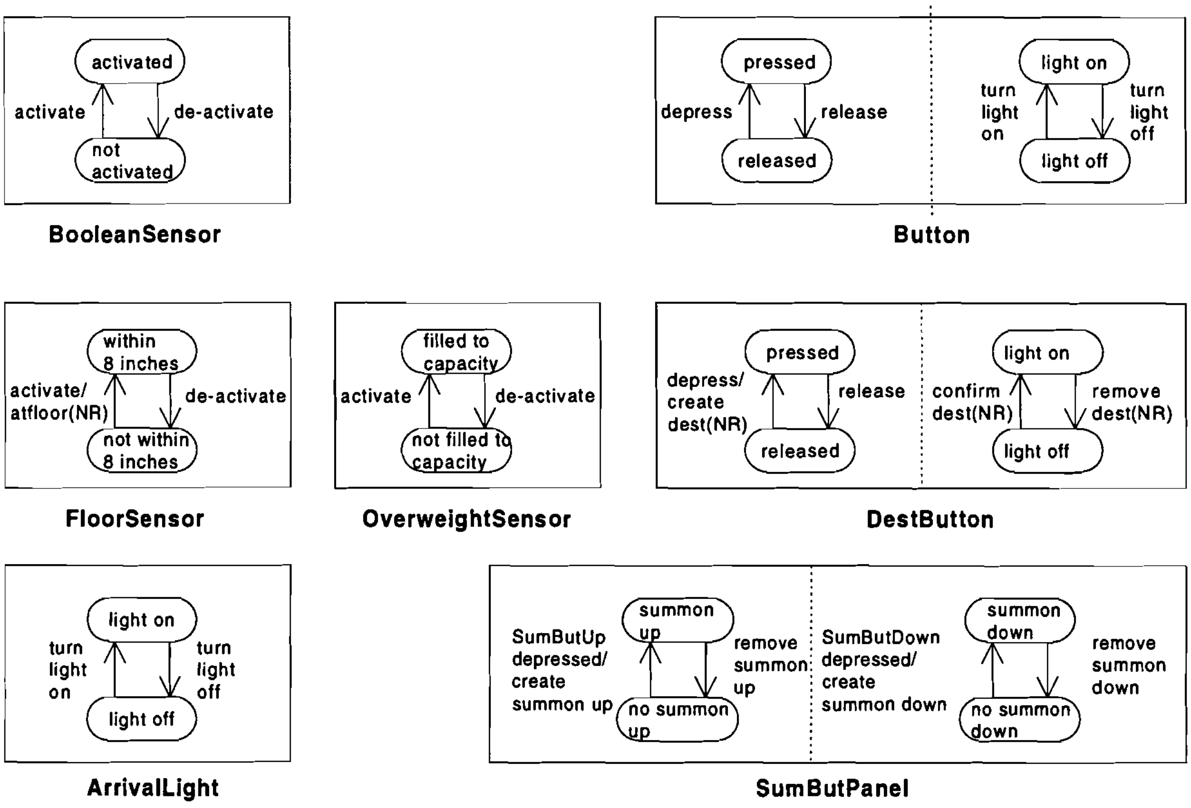


Figure 4.5: State diagrams of object classes with simple dynamic behavior

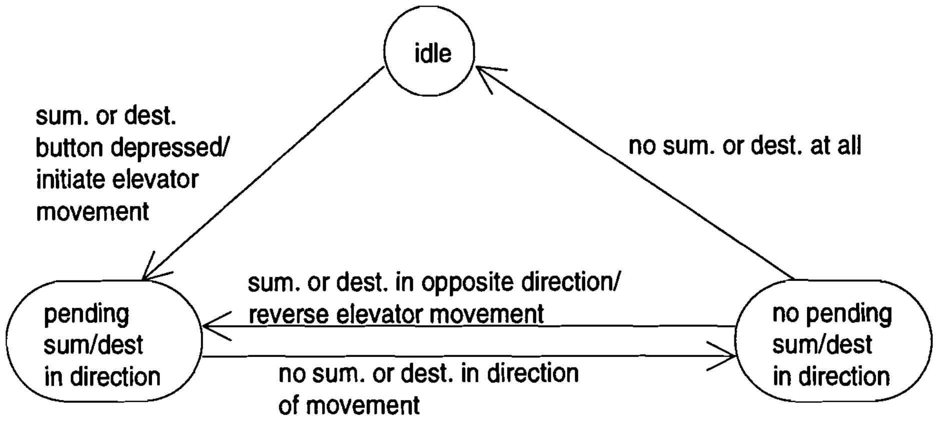

idle
sum. or dest. button depressed/ initiate elevator movement
no sum. or dest. at all
sum. or dest. in opposite direction/ reverse elevator movement
pending sum/dest in direction
no pending sum/dest in direction
no sum. or dest. in direction of movement

Figure 4.6: **Computer**: **ElevatorCage** state

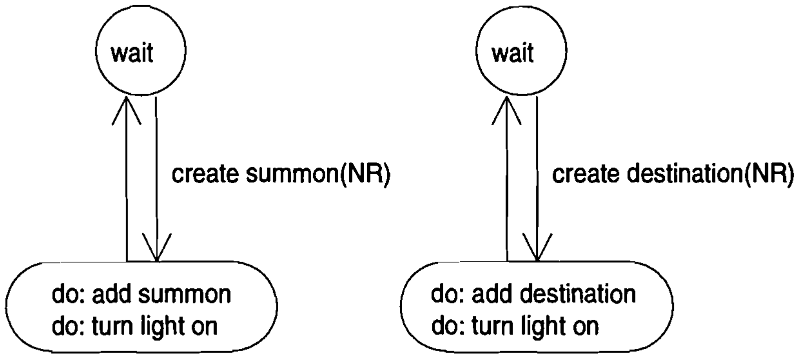


Figure 4.7: **Computer**: summon and destination requests

Figure 4.8 shows the sub state diagram of the state with pending summon or destination requests in the direction of movement. Figure 4.9 shows the sub state diagram of the state when all summon or destination requests in the direction of movement are handled.

The dynamic model of class **Computer** is a rough sketch which is not complete. It is however, a good starting point. Since the functional model serves as a guide to the methods in the dynamic model it may help to construct and refine the dynamic model. Because of the complexity of the dynamic behavior of the **Computer** it is a good idea to start working on the functional model before refining the dynamic model.

4.1.5 Functional model

The functional model shows how values are computed, without regard for sequencing, decisions, or object structure. The functional model shows which values depend on which other values and the functions that relate them. The following steps are performed in constructing a functional model:

- Identify input and output values

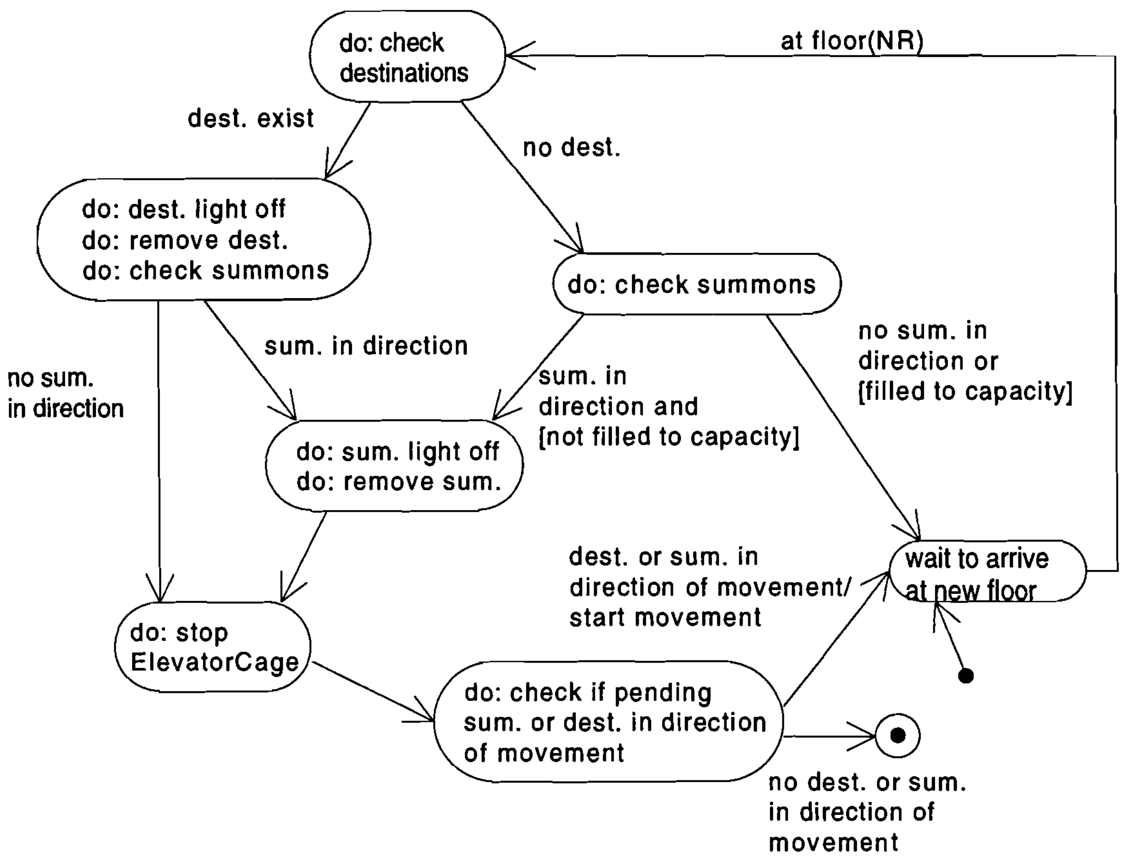

do: check destinations
at floor(NR)
dest. exist
no dest.
do: dest. light off
do: remove dest.
do: check summons
do: check summons
no sum. in direction
sum. in direction
sum. in direction and [not filled to capacity]
no sum. in direction or [filled to capacity]
do: sum. light off
do: remove sum.
dest. or sum. in direction of movement/ start movement
wait to arrive at new floor
do: stop ElevatorCage
do: check if pending sum. or dest. in direction of movement
no dest. or sum. in direction of movement

Figure 4.8: Computer: pending summon or destination requests

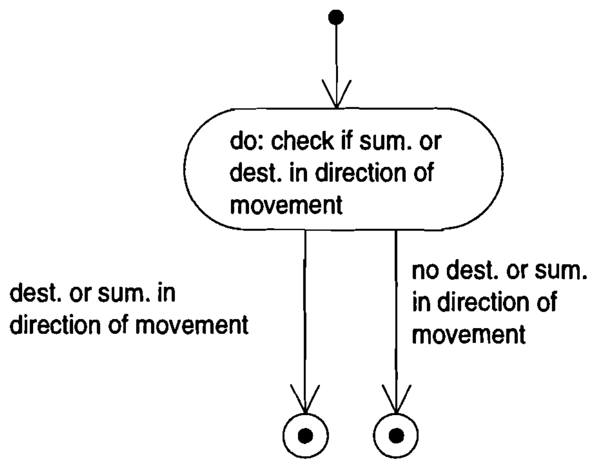
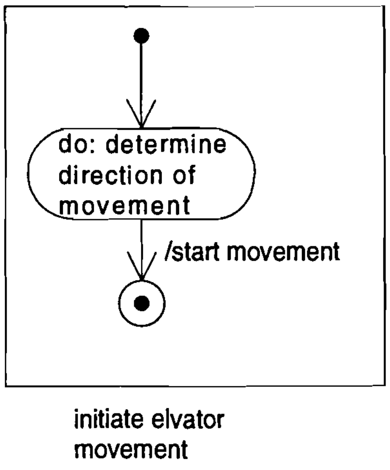

do: check if sum. or dest. in direction of movement
dest. or sum. in direction of movement
no dest. or sum. in direction of movement

Figure 4.9: Computer: no pending summon or destination requests


do: determine direction of movement
/start movement
initiate elvator movement

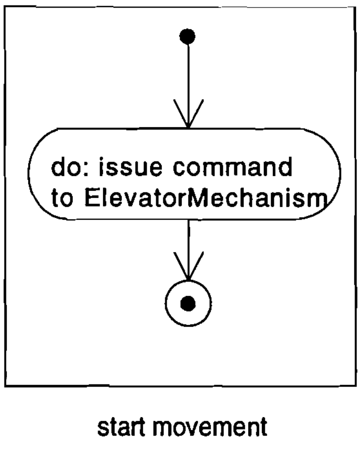


Figure 4.10: Computer: expanded actions: initiate elevator movement and start movement

INPUTS	OUTPUTS
Summon requests	Commands
Destination requests	Summons request accepted
Elevator direction	Destination request accepted
Elevator at floor	Elevator position
Efficiency rules	
Filled to capacity	

Table 4.5: Inputs and outputs for the functional model

- Build data flow diagrams showing functional dependencies
- Describe functions
- Identify constraints
- Specify optimization criteria

First step is to identify input and output values. Table 4.1.5 contains a list of inputs and outputs.

Starting with the inputs and outputs list, a functional model is build. Figure 4.11 shows a first sketch of the functional model. The functional model contains two levels of abstraction. The top level is shown in Figure 4.11 and the bottom level in Figure 4.12. Figure 4.12 is the expansion of the scheduler process.

Summons requests and destination requests result in summons and destinations inside the system. When they are added to their corresponding list, displays are updated as well. A scheduler queries the lists and determines the commands to send to the elevator mechanisms. Function elevator control translates commands from the scheduler into commands for the elevator mechanism. Whenever an elevator reaches a floor, elevator cage status is updated

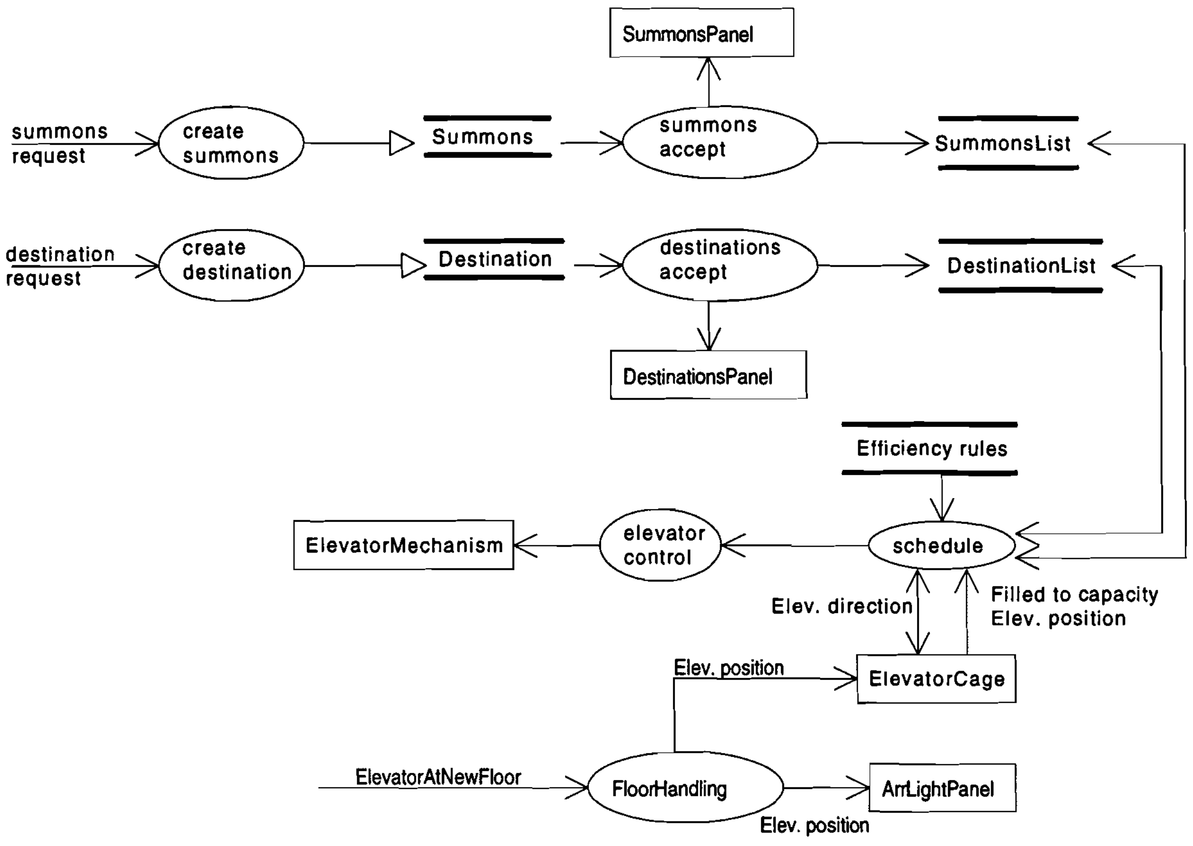


Figure 4.11: Functional model of the elevator problem

and the display inside the elevator is updated. The scheduler consists of two major parts: a summons part and a destinations part. Each part determines a desired elevator command according its request list and efficiency rules. The actual command that is send to the elevator mechanism is determined from these two desired commands.

As you can see the functional model contains the object Destination which is modeled as an association in the object model. This is inconsistant. Objects SummonsList and DestinationList in the functional model are not modeled in the object model. Both of these irregularities indicate that the OMT model of the elevator is not consistent or not complete yet. It may take one or more iterations to get the model right.

4.2 Review

The OMT model of the elevator problem is not entirely complete nor correct yet. At this stage, correcting the model requires at least another iteration through the object, dynamic and functional model. This will take a significant amount of time but won't learn us much new about modeling with OMT. We have already learned a lot. The main causes for the encountered problems are:

- Not enough notion about the difference between essence and implementation

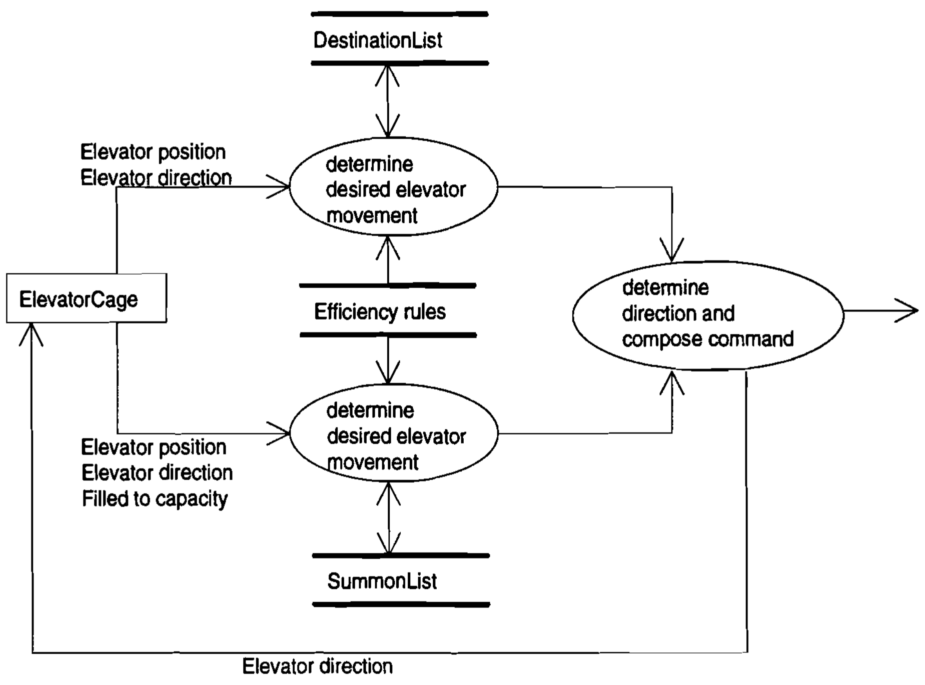


Figure 4.12: Expansion of the scheduler

- Class template view of the system is difficult to built
- Complicated hierarchy
- Unclear relation between the object, dynamic and functional model
- Many iterations necessary
- Ambiguity in the models

This section will explain each item in more detail.

4.2.1 Not enough notion about the difference between essence and implementation

Given that a system must function in a specific environment and given that it has a purpose to accomplish, it is possible to describe the system so that the description is true regardless of the technology to implement the system. This "description" is called an essential model. It is also possible to describe a system as actually realized by a particular technology. This is called an implementation model. Ward and Mellor [17] note the importance of separating essence from implementation.

During analysis an essential model of the system is build. The analysis of the elevator problem in this chapter should have resulted in an essential model. It contains however some implementation specific information. The choice of the class **Computer** is really a matter of implementation. A computer is a technology which is capable of performing the control task. Instead of this class, classes like SummonsHandler, DestinationsHandler, Scheduler etc. would be more appropriate. What about ArrivalLights, DestButton or DestButPanel? Are they implementations?

Whether or not ArrivalLights, DestButton or DestButPanel are implementations depends on the definition of the system. If, for example, the system definition only refers to the schedule and control part of the problem with the classes ArrivalLights, DestButton and DestButPanel in the environment, these classes are essential. If, however, the system definition refers to transporting people between floors, these objects are definitely implementations. We could have used parrots instead of buttons and lights to communicate with the passenger. It all depends on the system boundary. I quote Ward and Mellor [17], volume 1, page 10:

> It is impossible to distinguish between the essentials of a problem and the formulation of a solution unless the system boundary is carefully defined.

Unfortunately, OMT doesn't stress enough the importance of a system boundary. In fact, there is only one system boundary in [14] in one picture without any explanation. The absence of a system boundary in the elevator model caused several problems. Classes could not be judged if they are essential or implementation. The notion of the difference between essence and implementation was lost. The entire system is one class, class Computer, with all other classes in the environment. The complexity of the system is concentrated in class Computer.

4.2.2 Class template view of the system is difficult to built

People are not used to thinking in class templates. Instead, we think of objects and subconsciously we transform them into class templates. This transformation complicates the process of finding the right class templates. But that is not all. One of OMT's heuristics is to start looking for class templates in the requirements statement. The requirements statement contains information about the system and its environment. Most, if not all, of the class templates that will be found are templates for real-world objects. They belong in the environment, or they belong inside the system but are implementation biased. That the initial class template list needs a lot of editing is of no surprise. Even so, some of the necessary abstract class templates will be very hard to find.

Any dependency between two or more classes is an association. This implies that a lot associations will exist. Most of the association are derived associations. In this case, class templates are not related directly, but indirectly through one or more other class templates. For example the association between Computer and DestButPanel is indirect. The Computer is related to the ElevatorCage and the ElevatorCage is related to the DestButPanel. But doesn't the Computer gets its input from the DestButPanel? Is this relation really an indirect one? Because the the relation between ElevatorCage and DestButPanel is a part-of relation it is not hard to see that the relation between Computer and DestButPanel is indirect. However, it will not always be this obvious. Keeping the right associations and discarding the rest is difficult.

We have to choose to model something as a class template, an association or an attribute. OMT is very flexible at this point. It is for example allowed to transform an association into a class template or to transform an association into an attribute. How should the destinations be modeled in the elevator problem? Is it a class template associated with ElevatorCage and DestButPanel as in Figure 4.2, or should be an attribute of ElevatorCage? A destination could also be modeled as an association between ElevatorCage and Floor as in Figure 4.3. A choice must be made. It is needless to say that this choice has great impact on the object and dynamic models. OMT's flexibility is a strong modeling tool, but has as disadvantage that it may require additional iterations if a wrong choice is made. For an unexperienced modeler, flexibility gives him a feeling of uncertainty. Did he make the right choice? This problem is always there with a flexible modeling tool, but as the modeler becomes more experienced, flexibility gives him a lot of freedom.

4.2.3 Complicated hierarchy

OMT supports six types of hierarchy:

- Aggregation hierarchy (object model)
- Inheritance hierarchy (object model)
- Dynamic hierarchy within states and events (dynamic model)
- State generalization hierarchy (dynamic model)
- Event generalization hierarchy (dynamic model)
- Functional hierarchy (functional model)

The title of this section—complicated hierarchy—refers to the aggregation, inheritance and dynamic hierarchy of the models.

In OMT an aggregation is a special type of association, namely the part-of relation. It relates a class template acting as a whole with class templates being components of it. Because an aggregation is a special type of association, aggregation does not involve any encapsulation. Therefore, aggregation does not simplify a complex model with hiding of information.

Inheritance hierarchy relates class templates and more refined versions of it. It is an abstraction for sharing similarities, while preserving the differences. It's main purpose is in reuse of previously defined class templates. A descendant class template inherits the operations and attributes of its ancestor, while adding attributes and operations that refines it from its ancestor. I quote Rumbaugh et.al. [14], page 111:

> The dynamic model of a class is inherited by its subclasses. The subclasses inherit both the states of the ancestor and the transitions.
> ... The state diagram of the subclass must be a refinement of the state diagram of the superclass.

However, inheritance of dynamic behavior isn't alway possible. Lets take for example class template Airship. It has two descendants: Balloon and Helicopter. Inheritance of attributes is no cause for problems, but their dynamic behavior is very different. Their dynamic behavior is not a refinement of the dynamic behavior of class template Airship but are totally different state diagrams. This inability of reuse is commonly known as the inheritance anomaly. Inheritance does not simplify the understanding of dynamic behavior of class templates.

The dynamic model in OMT allows nesting within states and events. This nesting does not support encapsulation of dynamic behavior and all events are global. An event generated in one class template at a particular level in the dynamic hierarchy could cause a transition in any other class template at any level in the dynamic hierarchy. The models do not directly show which class templates interact. The lack of encapsulation and the fact that events are global make the dynamic models hard to understand and maintain.

4.2.4 Unclear relation between the object, dynamic and functional model

There exists a strong relationship between the object and dynamic model. There is a dynamic behavior description for each class template. The dynamic model specifies allowable sequences of changes of the instances of class templates. It is however unclear, which class templates interact. Associations do not help here, because OMT allows interactions between class templates without actually having an association between them.

The relation between the functional model and the object and dynamic model is vague. This is what Rumbaugh et.al. [14] say:

> The functional model shows what has to be done by the system. The leaf processes are the operations on the objects. The object model shows the "doers". Each process is implemented as a method on some object. The dynamic model shows the sequence in which the operations are performed. The three models come together in the implementation of methods. The functional model is a guide to the methods.

There is not a direct relation between the functional model and the object and dynamic model. Instead, the processes of the functional model are implemented as operations, scattered over the class templates in the object model. Not even the names of processes in the functional model will be found in the object or dynamic model. It seems, that the object and dynamic model contain all the information to describe the system, but the functional model helps to explain these models. The functional model also helped to find abstract classes in the elevator problem.

4.2.5 Many iterations necessary

OMT builds models iteratively. The first step in building the object model is identifying candidate class templates. Of this list an initial selection is made. Next step is to identify associations between class templates. This results in a long list of candidate associations of which also an initial selection must be made. While looking for and selecting class templates and associations other classes and associations will be found. The object model is build iteratively looking for class templates and associations. As mentioned in 4.2.2, making the right selections at this stage is difficult. This will cause several iterations.

The dynamic model is build after the object model. Building the dynamic model is an iterative process as well. Preparing scenario's, identifying events and building a state diagram with hierarchy. Building the state diagrams is difficult because most of the operations are not defined yet. This will cause several iterations. If the dynamic behavior of a class template is too complex, iterations must be made through the object model again to remodel it to reduce dynamic behavior complexity. The dynamic model cannot be completed at this point, because not all of the operations are already defined and maybe some of the necessary class templates are missing.

The functional model is build after the dynamic model. The functional model contains information about "what has to be done" by the system—see quotation in previous section—. However, this information was already needed building the previous two models. At this stage additional class templates and operations are found. This is cause for iterations concerning the object and dynamic model. It is of no surprise that iterations at this stage cost a significant amount of additional time to build the models. Building the models of the elevator problem took many iterations and are still not finished yet.

4.2.6 Ambiguity in the models

OMT models are informal. An informal approach is relatively easy to learn and understand. This is a big advantage. On the other hand, informal models may be ambiguous and hard to keep consistent. This is exactly the case with the models of the elevator problem.

The functional model contains several processes that will be implemented as methods of objects. The functional model is loosely related to the object and dynamic model. It is difficult to keep these models consistent, because consistency is not defined in OMT. Changing a process in the functional model causes changes in several methods over different objects. It is easy to overlook one of them, especially because the relation between functional model and object and dynamic model is vague.

In OMT, dynamic behavior is modeled as state diagrams and events. Events are an unbuffered asynchronous way of communication between interacting class templates. This type of communication is prone to ambiguities. The elevator model for example, contains an ambiguity in the dynamic behavior. See Figure 4.5 and Figure 4.6. A button press could cause the Computer to send a command to the ElevatorCage to start moving. At the same time, destinations lists and summons list are updated. However, the Computer needs information from the updated lists to compose its command. Are the lists updated in time before the Computer queries them?

Chapter 5

Software/hardware engineering requirements

This chapter shows and explains requirements for a new method. These requirements are based on the experiences with building the OMT model of the elevator problem. This wish-slip is divided into two parts: a modeling part and a framework part. The modeling part describes all requirements related to modeling issues. The framework part describes all the requirements related to building and manipulating the models. The items in this list are numbered and are referenced to in next chapter where the actual new method will be explained.

1. Modeling requirements
   (a) Allow specification, analysis, design and implementation
   (b) Type of system: embedded, real-time, combined hardware and software
   (c) Minimal, but sufficient concepts
   (d) Graphic representation of models for human understandability
   (e) Hierarchy, with encapsulation to conquer and divide complexity
   (f) Ability to model concurrency
   (g) Ability to model response times
   (h) Ability to model topology
   (i) Concepts to model dynamic visibility
   (j) Concepts for dynamic creation/deletion of objects
   (k) Models must have a formal basis
   (l) Verification must be possible
   (m) Transformation must be possible
   (n) Simulation must be possible
2. Framework requirements
   (a) Specification, analysis, design and implementation
   (b) Separating essence from implementation
   (c) Close relation between system functionality and models

(d) Equal emphasis on both environment and system model

(e) Minimizing iterations

(f) Communication with the customer at all stages

(g) Change development phase seamlessly

(h) Complexity management

(i) Project management

(j) Heuristics

5.1 Modeling requirements

The new method must support all development phases from analysis and specification to design and implementation. Development starts with an initial requirements description from the customer. It describes what the system must do in an informal way. A requirements description may be ambiguous and is usually far from complete. During analysis and specification, models of the system and its environment are build. Models should give an understanding of the problem domain. They specify the system and its environment in a rigorous manner. Once agreement exists with the customer about these essential models, design may take place. Design involves manipulating, modifying, extending or transforming models. During design, information about design decisions and their consequences are added to the models. Design in the new method must end with an implementation blueprint. The implementation blueprint is a model, that contains enough information for an engineer for final stage design and implementation.

The new method must be able to model the type of systems we are interested in. These systems are embedded and real-time, which may be implemented in hardware, software or both. Concepts to model this type of system should be sufficient, but minimal. Minimal, because the learning curve of a method is directly related to the number and complexity of its concepts. Sufficient, because all relevant aspects of a system and its environment must be modeled.

Whenever two engineers are discussing a complex problem they usually resort to drawing each other pictures in order to explain the concepts involved. A graphic representation is usually easier to understand by humans than a textual representation. Therefore, graphic representations in the new method are required to allow at least communication with the customer. Graphic representation of the new method must be based on a, perhaps textual, formal basis to enhance rigor.

The new method must handle systems of great complexity gracefully. Since human limitations prevent us from keeping all the details of complex systems in our head at one time, the models must allow partitioning into a large number of chunks. However, it is also necessary to be able to perceive in some sense the system as a whole [17]. This implies representations at various levels of detail, together with a bookkeeping scheme to show the relationship between the different levels of representation. Encapsulation is required in this hierarchy. It allows us to understand the model at a particular level in the hiearchy without the necessity to dig into lower levels of representation. In order to handle complex systems models of the new method must allow the concept of hierarchy with encapsulation. The object-oriented paradigm with powerfull concepts to encapsulate operations and corresponding data into objects might be suitable [13].

To model real-time, embedded systems properties like concurrency, topology and response times are very important concepts. Models of the new method should show which parts operate concurrent. Response times are closely related to concurrency. A model may require concurrent parts. Several response times may be essential. In an essential model, concurrency and response times are independent. In the implementation view of the models however, concurrency may be simulated by an sequential implementation with sufficient fast enough response times. Topology shows the physical distribution of a system. Topology imposes constraints on both essential and implementation models. Topology requires additional modeling constructs to model separated system parts.

Usually, modeling techniques separate the static structure of a system from its dynamic behavior. However, the static structure of a system and its environment may not be as static as it seems. In the elevator problem for instance, passengers are dynamic objects. They encounter the system's environment if he/she wants to be transported between floors. The passenger notifies his summons request to the system. When an elevator arrives, the passenger enters the elevator cage and notifies his destination request. When he arrives at his destination floor, the passenger leaves the system's environment. The passenger physically moves and his communications and relations with the system moves with him accordingly. From the system's point of view a passenger comes into existence, moves and dies. This example shows two different concepts: 1) dynamic "visibility" of model parts and 2) dynamic creation/deletion of model parts. The new method must support both. Note that these concepts apply to both environment and system.

The informal approach of OMT was cause for ambiguities and in general makes a model susceptible for context sensitive interpretation. Therefore, the new method must have a formal basis. The rigor of a formal model description forces an engineer to understand "every corner" of the problem in order to build the model [13]. Other major advantages of a formal description over an informal one are ability for verification and correctness-preserving transformations. A formal description is also well suited for simulation.

5.2 Framework requirements

A methodology consists of modeling techniques and a framework that guides the designer how to use these modeling techniques. A methodology framework must support specification, analysis, design and implementation. A requirements description is the first step towards a working system. This description usually is ambiguous, far from complete and has an implementation bias. The designer should analyze the essentials of problem and build a model of it. The essential model serves as a specification. Analysis and specification is one major part of the methodology. The other major part is design. Design may take place when agreement with the customer about the essentials of the system is achieved. The starting point of design is the essential model. This model will be manipulated and extended during design. The result is a model that serves as an implementation blueprint. The methodology must not restrict the designer in the sequential order of analysis/specification before design. It should also be possible to start design on an abstraction level of which already agreement exists about the model before the entire essential model is finished. This will speed up development time.

A methodology must force the designer to separate essence from implementation. The objective of the designer is to build the "optimum" implementation(blueprint) of the system.

It is therefore not permitted to include design decisions in the specification as this limits the number of solutions to the problem. During design the designer may optimize for one or more properties. This will usually be system properties, but optimizing for development time is also allowed.

A system "must do" something. A system is its functionality. For many years, engineers used design methods based on functional approaches. It is very natural to think about a system in functions. When a feature is added to the system, many functions may change a little and/or new functions are added. Models of these functional approaches are not very stable when modified in practice. Since a system's functionality is what it is all about, the new method must show how functionality is distributed across the system. It must not only show how functionality is distributed, but also guides the designer to distribute in a way that is stable when requirements change.

In the OMT model of the elevator problem the environment is poorly modeled. It does not show how passengers move and change their visibility for different parts of the system. It does not model that a passenger can only enter an elevator when the elevator is at the correct floor with its doors open, nor does it model that an elevator always arrives at floors in an up or down sequence instead of randomly. Yet, this information is used building the dynamic models. In a new methodology, any specific information about environment and system that is used to model must be modeled. This will require an equal emphasis on both environment and system model. Only when the environment is understood in every detail, a correct system can be build for this environment.

As already mentioned in previous section, modeling is an iterative process. To reduce time spend building the models, the number of iterations must be minimized. A new method must guide the designer to make the right choices to limit iterations. Development phases must follow each other seamlessly. A seamless phase change does not require model transformations which are cause for errors and loss of information. While building the models, communication with the customer about the system is required. The method must allow communication with the customer, who may not be an expert or has knowledge of this method.

A new method must be able to handle systems of great complexity. A new method must guide the designer to divide and conquer system complexity. This is complexity management. Complexity management helps to partition a system and spots in an early stage model parts of which complexity is too great. A model with unbalanced complexity will be hard to understand and is difficult to manipulate.

Complex systems are usually build with a team of engineers. To allow several engineers to work on a single systemdesign simultaneously, the new method must support project management. Project management helps to divide the work to be done over several persons. The method must allow parallel development activities as many as possible.

At this point several requirements for a new method have been described. They are mostly based on experiences with the application of OMT on the elevator problem, but on literature research as well. Next chapter describes the new method. It is called SHE—Software/Hardware Engineering—. Next chapter describes SHE as far as it has evolved at the moment this thesis is made.

Chapter 6

Software/hardware engineering summary

6.1 SHE overview

Software/hardware engineering—SHE—is an object-oriented method, developed to support the co-design of complex hardware/software embedded systems—requirement 1b—. The method integrates both formal and informal techniques. SHE is composed of a method framework and modeling concepts. This section describes the general overview of SHE, while succeeding sections describe parts of SHE in more detail. This chapter is not a SHE manual, but a description of the current state of SHE. SHE is still under development. In this chapter references are inserted to requirements in previous chapter where appropriate.

Figure 6.1 shows SHE's framework. It offers specific views on the system to be designed and enables awareness of the activities on hand in the modeling process from analysis/specification to design. The framework is designed to limit the number of iterations between development phases. The top half of the framework focusses on system essentials, mixed with high level system design. The bottom half focusses solely on implementation. SHE separates essence from implementation—requirement 2b—. Essentials of the system are captured into models in the analysis/specification phase. If agreement with the customer about the essentials is achieved, design may take place. The design phase—the bottom half of the framework—results in an implementation blueprint. SHE's framework offers possibilities to anchor analysis/specification and design activities so that they can be performed concurrently or sequentially.

The framework falls apart into four quadrants, grouped as two pairs:

- Essential part of the framework
  - Essential behavior model
  - Architecture structure model
- Implementation part of the framework
  - Implementation structure model
  - Extended behavior model

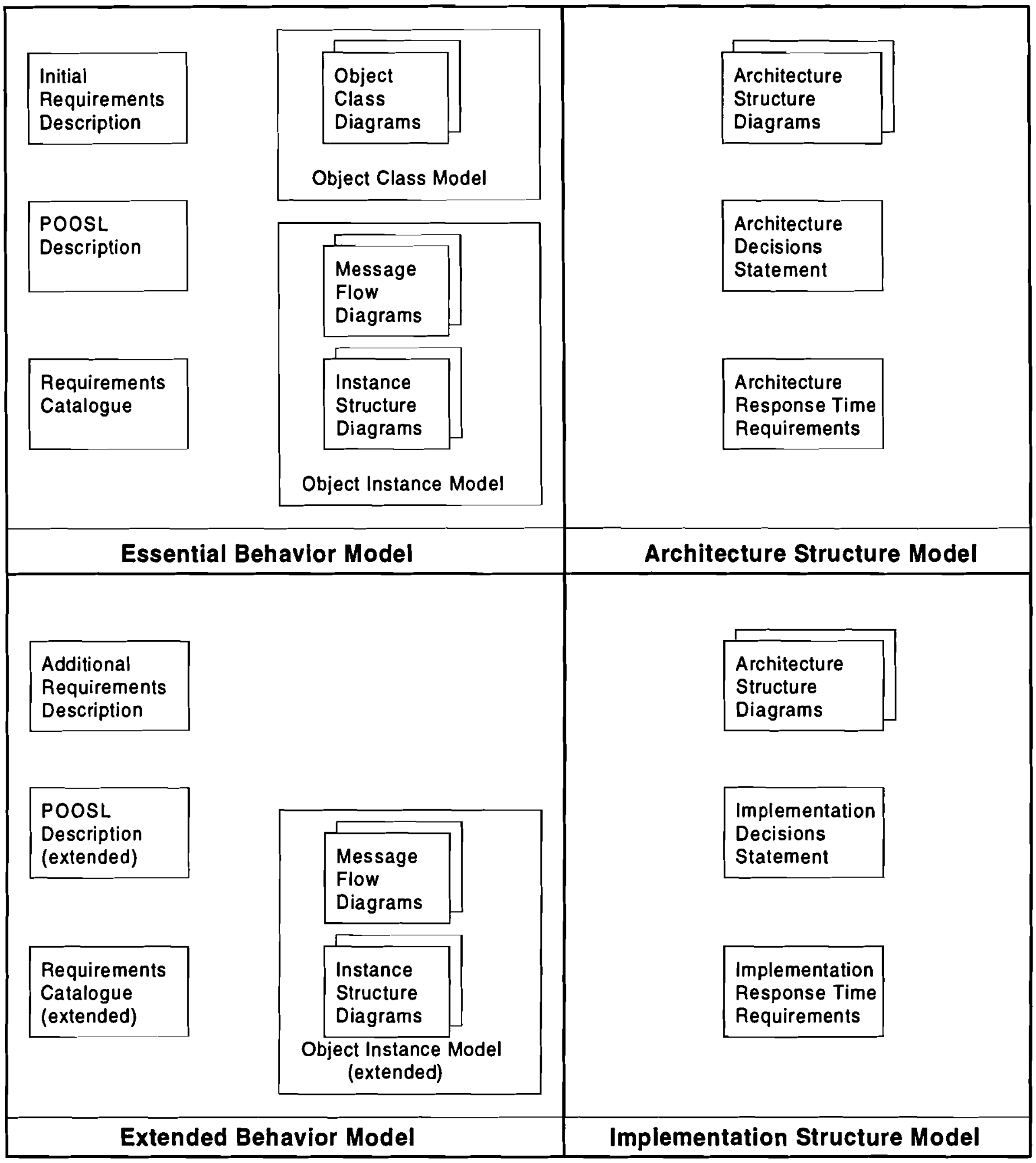

Initial Requirements Description
Object Class Diagrams
Object Class Model
Architecture Structure Diagrams
POOSL Description
Message Flow Diagrams
Architecture Decisions Statement
Requirements Catalogue
Instance Structure Diagrams
Architecture Response Time Requirements
Object Instance Model
Essential Behavior Model
Architecture Structure Model
Additional Requirements Description
Architecture Structure Diagrams
POOSL Description (extended)
Message Flow Diagrams
Implementation Decisions Statement
Requirements Catalogue (extended)
Instance Structure Diagrams
Implementation Response Time Requirements
Object Instance Model (extended)
Extended Behavior Model
Implementation Structure Model

Figure 6.1: SHE framework

6.2 Essential behavior model and architecture structure model

Figure 6.1 shows that the essential behavior model consists of the following parts:

- Initial requirements description (section 6.2.1)
- Object class model (section 6.2.1)
- Object instance model (section 6.2.2)
- POOSL description (section 6.2.3)
- Requirements catalogue (section 6.2.4)

and the Architecture structure model consists of the following parts:

- Architecture structure diagrams (section 6.2.5)
- Architecture decisions statement (section 6.2.5)
- Architecture response time requirements (section 6.2.5)

Each of these items will be explained in more detail together with the relations between them.

Starting point of SHE is the *initial requirements description.* It is the customer's description of the system to be designed. It is used to build the *object class model.* This model shows a graphical representation—requirement 1d—of object classes from the problem domain and their relationships. The *object class model* is intended as an initial study of the problem domain. Next step is to build the *object instance model.* It shows communicating objects of system and environment. The *object instance model* is a graphic representation, that incorporates boundaries. Boundaries show architecture structure. The *architecture structure model* describes high level structuring requirements and high level system design decisions, for example logic layering and topology may be imposed by the initial problem description and by physical geographical constraints. A specific topology may be a high level design decision of the system and will be modeled in the *architecture structure model* as well. Structure imposed by the *architecture structure model* is formalized in terms of boundaries and object choices in the *object instance model.* Therefore, the *object instance model* will be build in conjunction with the *architecture structure model.*

Formalization—requirement 1k—in POOSL is performed when the *object instance model* becomes stable. The *POOSL description* formalizes the *object instance model* and describes dynamic behavior. A *requirements catalogue* collects information to be stored during analysis and specification. It contains a data dictionary, object to boundary maps, traceability data and environmental conditions.

6.2.1 Initial requirements description and the object class model

First step in developing a system is to state requirements. The customer supplies the *initial requirements description.* It should state what the system "must do", and not how it is to be done. It should be a statement of needs, not a proposal for a solution [14]. The customer must state what is mandatory and what is optional. Performance specification, standards like modular construction, logic layering and provisions for future extension are also legitimate requirements. The *initial requirements description* should describe the essentials of the system.

In practice, an *initial requirements description* often contains both essentials and implementation specific information. Implementation specific information is in fact a collection premature design decisions, that limits the set of possible solutions. The analyst must be aware of this problem and should seperate essence from implementation. Other problems of the *initial requirements description* are its possible ambiguities, incompleteness or even its inconsistentness. The analyst works with the customer—requirement 2f—to refine the requirements so they represent the customer's true intent. The analyst does so when he builds the *essential behavior model* and the *architecture structure model.* The elevator problem of chapter 2 is an example of an *initial requirements description.*

SHE is able to handle systems of great complexity. These systems may require an initial study to get enough understanding of the problem domain. Building the *object class model* serves as an initial study of the problem domain and the result is a first inventory of candidate objects for the *object instance model.* The *object class model* shows object classes of system and environment and their relationships. It is much like the object model of OMT [14], there is one difference however. SHE *object class model* separates generalization-specialization trees from aggregation trees. SHE incorporates the Object Model approach used in the Fusion Method of Coleman et.al. [3], that shows aggregation hierarchically. Figure 6.2 shows an example *object class model.* It is the same model as Figure 4.3 and 4.4.

The *object class model* is build according heuristics and guidelines of OMT, but with notation of the Fusion method [3]. Here, SHE incorporates the good work that has been done by Rumbaugh and many others. OMT offers a comprehensive description of object modeling along with useful heuristics. The *object class model* will be used as an initial study of the problem domain and serves as a first inventory of candidate objects for the *object instance model.*

6.2.2 Object instance model

Pillars of SHE models are the *object instance model* and a formal *POOSL description.* The *object instance model* visualizes objects, their collaboration and message flows between them. It shows structure, boundaries and gives a first impression of dynamic behavior. The *object instance model* consists of two types of diagrams:

- Message flow diagrams
- Instance structure diagrams

Message flow diagrams

The *message flow diagrams* show process objects and message flows. A process object encapsulates data objects and dynamic behavior. Collaborating process objects perform some coherent part of the system behavior, called a scenario. The collection of all scenarios describes essential system behavior completely. Functionality is partitioned over process objects. Each process object provides a part of system functionality with its encapsulated data objects, dynamic behavior and communication with its collaborators.

Collaborating process objects communicate through messages. A message transports information from one process object to another. Information send may be the message itself

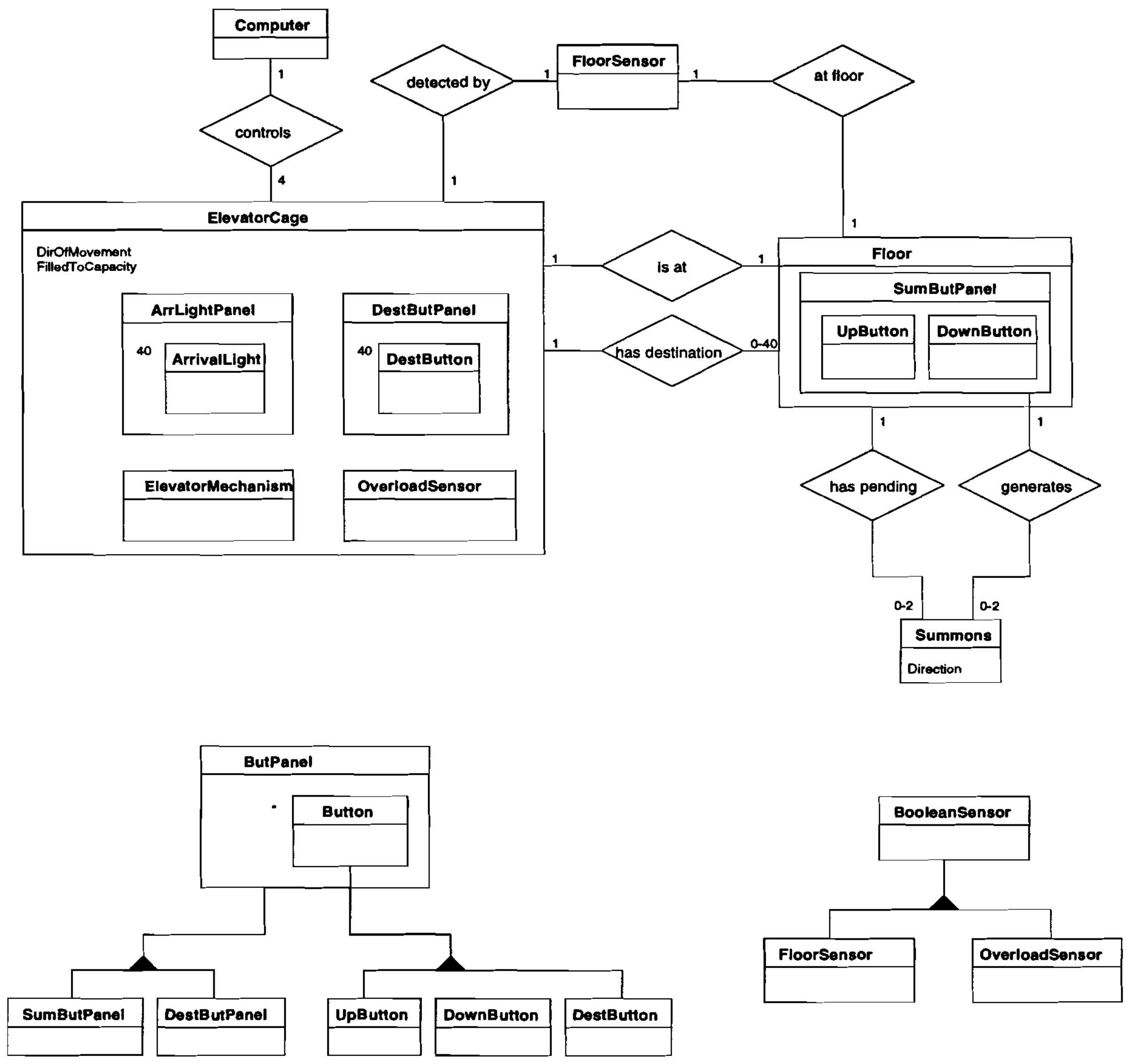


Figure 6.2: Example object class model

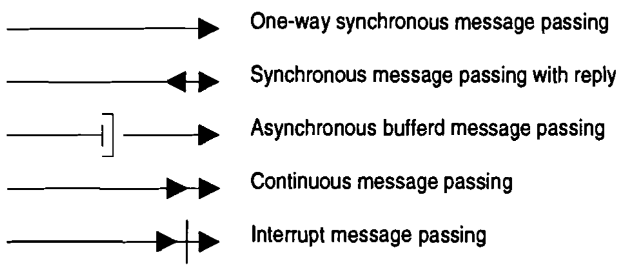

One-way synchronous message passing
Synchronous message passing with reply
Asynchronous bufferd message passing
Continuous message passing
Interrupt message passing

Figure 6.3: Message flow primitives

and/or time it is send, but it may also transport additional information in its parameters. Therefore, each message has a name and may have zero or more paramaters.

SHE has five message flow primitives, see Figure 6.3. The first message flow primitive is one-way synchronous message passing. Information flow is in one direction only and information transfer will take place at the instant both sender and receiver are willing to perform a rendez-vous. If for instance the sender is willing to send, but the receiver is not willing to receive, the sender will wait until the receiver is willing to accept the message. This applies the other way around as well. This synchronous communication is based on the rendez-vous principle. Information will never be lost.

Second message flow primitive is synchronous message passing with reply. It occurs often that a process object requires information from another process object. Communication in this situation is divided into two parts. In the first part the initiating process object sends a message requesting for information. This is one-way synchronous message passing. In the second part, the receiving process object will answer the information request with another one-way synchronous message passing mechanism, with a message containing the requested information. SHE has a shorthand for this message pair, the synchronous message passing with reply symbol.

In asynchronous buffered message passing there exist no synchronization between sender and receiver. The message is send, gets in the message buffer and sits there until the receiver is ready to receive. This mechanism always allows sender to send immediately. Buffer size is unlimited. If the receiver is ready, but the buffer is still empty, it waits until a message arrives. With asynchronous buffered message passing, information is never lost.

Some occasions require emergency communications. The sending process object requires immidiate message passing, whether the receiving process object is ready or not. SHE has the interrupt message passing for this purpose. Whenever an interrupt message is send, the receiving process object accepts this message, no matter what it was doing, and behaves accordingly.

Continuous message passing is used to model continuous information flows. Take for example a clock. The arms of a clock continuously display the current time. If someone wants to know the time, all it takes is to take a look at the clock. Though the clock continuosly transmits information, the contents and availability of the information is only important at the instant the person looks at the clock—is ready to receive information—. This is exactly what the continuous message passing models. Whenever this symbol is drawn, it means the sender is continuously ready to send the latest information. However, the actual information

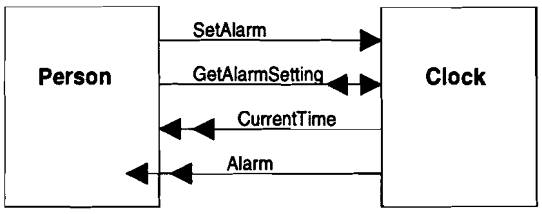


Figure 6.4: Message flow examples

transfer takes place at rendez-vous.

Except for the asynchronous bufferd message passing, Figure 6.4 contains all types of message passing mechanisms. The example shows two process objects: Person and Clock. A person may set the clock alarm by sending the SetAlarm message to the clock with the alarm time as its parameters. He may check the alarm setting with the GetAlarmSetting message. The clock will reply with the alarm time. The current time is always available. All a person needs to do is read the CurrentTime message. This message contains the current time in its parameters. If the alarm fires, the clock sends the Alarm message to the person. Whatever the person was doing, this message will always come through.

Building a message flow diagram involves adding process objects and message flows. Process objects may be real-world or abstract and may belong to the system or environment. The *object class model* serves as a candidate object list. This list will have a tendency towards real-world environment objects, as the OMT model of the elevator did. This is not a problem, because it is only a starting point. New objects must be found. The analyst must keep in mind that he is building an essential model. Therefore all objects and message flows must be essential. Ward and Mellor [17] offer a comprehensive description how to separate essence from implementation.

While new objects and communications are added to the model, system functionality is distributed among process objects. Each process object performs part of system or environment functionality. When building a message flow diagram, we only have to think about the "outside" of an object. What it should do and what communications with collaborating process objects are necessary to perform its task. Objects help to find message flows. At some time a message flow as a request for service of another process object is known, but the collaborating object does not exist yet. This is how message flows help to find new objects. There is a strong interaction between finding objects and message flows. This process of thinking about the "outside" of objects, finding and adding objects and communication feels very natural. *Message flow diagrams* evolve quickly. Note however that the *object instance model* is build in conjunction with the *architecture structure model.* How this is done will be explained in section 6.2.5.

Message flows of an object will give a first impression of its dynamic behavior, without going into detail. *Message flow diagrams* will give an idea about dynamic behavior and functionality of system and environment. Each *message flow diagram* models one or more scenarios, all scenarios model the problem domain.

Before describing hierarchy, messages from a single object to multiple objects need explanation. See Figure 6.5. Object A is a single object, object B is a multiple object. A multiple object is in fact a collection of instances of an object class. Each process object has its own

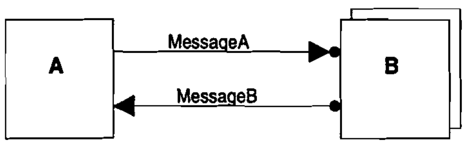


Figure 6.5: Message flow from a single object to a multiple object

private data objects and operates separate from other process objects in the multiple object. Each of these objects is unique and may be identified in some way. **MessageA** is a message from a single object to a multiple object. Every object of the multiple object "hears" the message, but only one is allowed to receive it. Which of the ready receiving objects will actually rendez-vous with the sending object is nondeterministically determined.

Whether or not an object is willing to receive a message depends on its current state of operation, but may also depend on parameters of the message to receive. It is a message selection mechanism. This mechanism may be used to send messages from a single object to another object of a multiple object. In this situation, one of the message parameters could identify a particular object of the multiple object. Remember, identification and selection is not limited to a single parameter, but is an evaluation of a boolean expression on all parameters. When an object of the multiple object sends a message to a single object, the single object does not automatically "know" where this message came from. The sending object may however identify itself in the parameters of the message. More information about messages, rendez-vous and identification can be found in Voeten's POOSL [16].

The last important concepts of the *message flow diagrams* to discuss are clusters and composite objects. Both add hierarchy—requirement 1e—to the *message flow diagrams*. Figure 6.6 shows a cluster of objects B and a composite object C. They both contain objects and messages. Inner objects are hidden from the outside world. All messages except to ones to the box boundary are hidden for the outside world as well. The outside world of clusters or composite objects only "sees" the communications to the box boundary. Objects outside cluster B only see the messages a, b and e. Objects outside composite object C only see messages d, c and e. In a cluster any object may export messages, while in a composite only one object exports messages.

In a composite object the object that exports messages plays a central role in the composite. The central object provides the composite's functionality to the outside. To accomplish its task, it collaborates with other objects inside the composite. To get an abstract description of the composite, only an abstraction of the central object of the composite needs to be taken. Generally, getting an overall abstraction is much more complicated. Functionality is provided by interaction of all objects inside a cluster. To get a cluster's abstraction, all interacting objects as whole needs to be abstracted. The resulting abstract description will be more complicated than in the case of the composite object.

Both clusters and composites add hierarchy to the *message flow diagrams*. SHE does not prescribe a bottom-up or top-down approach to build the hierarchy. Instead, objects are placed in a flat model initially. While finding new objects, complex objects are expanded to lower levels or collections of objects are grouped to form clusters or composites. Starting at some intermediate level in the hierarchy and extending the model in both direction is a very flexible method which does not hinder the analyst if he cannot determine the level of

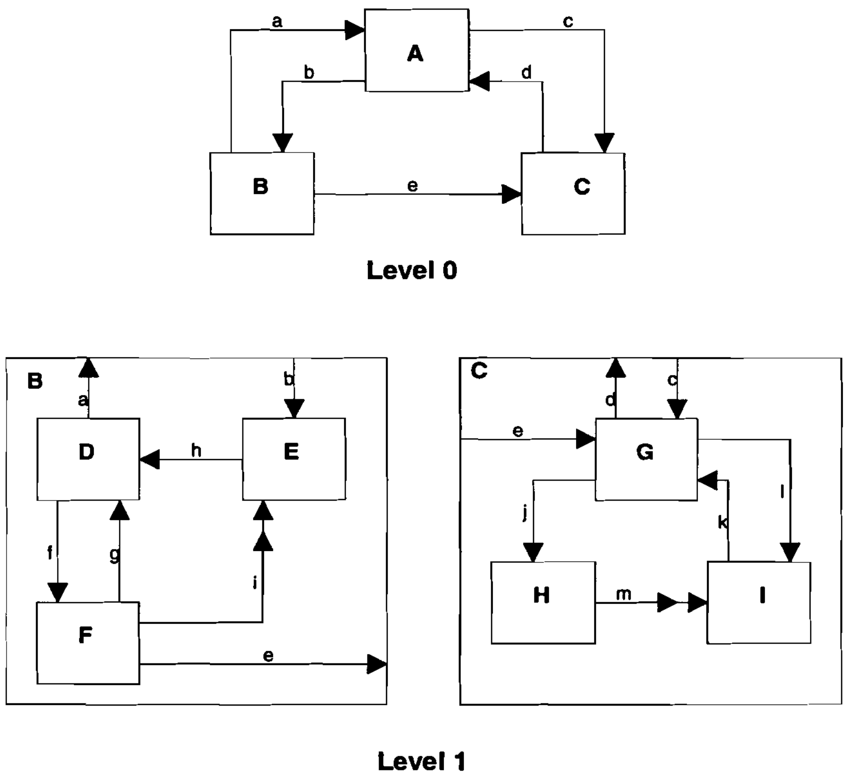


Figure 6.6: Example of a cluster and a composite object

abstraction of an object yet.

Message flow diagrams are build starting with the *object class model* and using it as a first list of candidate objects. The *object class model* should have given an understanding of the problem domain, so at least a few communication can already be added between objects. Communications help you find new objects and new objects will require new communications. Continuously check objects and communications for essence. See Ward and Mellor [17] how to do this. Besides the difference between essence and implemenation, the *architecture structure model* must be taken into account. How this is done is shown in section 6.2.5. Add hierarchy as the model evolves.

Instance structure diagrams

The *object instance model* contains a lot of information. Instead of putting it al together in one model, the *object instance model* is seperated in two parts: *message flow diagrams* and *instance structure diagrams. Instance structure diagrams* are a basis for showing architecture structure. The diagrams present instances—process objects—interconnected by simple static channels. The object hierarchy is exactly the same as in the *message flow diagrams.*

Figure 6.7 shows an example *instance structure diagram* which with Figure 6.6 forms an *object instance model.* A channel is responsible for the transport of one or more messages identified in the *message flow diagrams.* Each channel has a unique name. Channel hiding philosophy in the *instance structure diagrams* is the same as message hiding in the *message flow diagrams.* Channels inside a cluster or composite are invisible, except for the ones connected to the box boundary.

Architecture structure is represented by drawing boundaries. SHE identifies four different types of boundaries necessary for complex hardware/software systems:

- Abstraction boundaries
- Concurrency boundaries
- Distribution boundaries
- Implementation boundaries

Abstraction boundaries encapsulate collections of objects. Examples are clusters and composite objects. Abstraction boundaries contain a group of objects to form an object of a higher level of abstraction. Abstraction boundaries are drawn as solid line boxes, just like objects.

Concurrency boundaries—requirement 1f—identify concurrent operating model parts. Inside a concurrency boundary, operation is sequential. Areas, each surrounded with a concurrency boundary, operate concurrent.

Synchronous/asynchronous operation of model parts and visibility of objects are important concepts in hardware/software systems. They are modeled as distribution boundaries. Currently, these concepts are researched and did not reach a final state yet. This applies to imlementation boundaries as well. Implementation boundaries—requirement 1h—show the allocation of objects to implementations. Ideas exist not to draw a separate type of border for each type of boundary, but to have one type of border and to attach attributes to it.

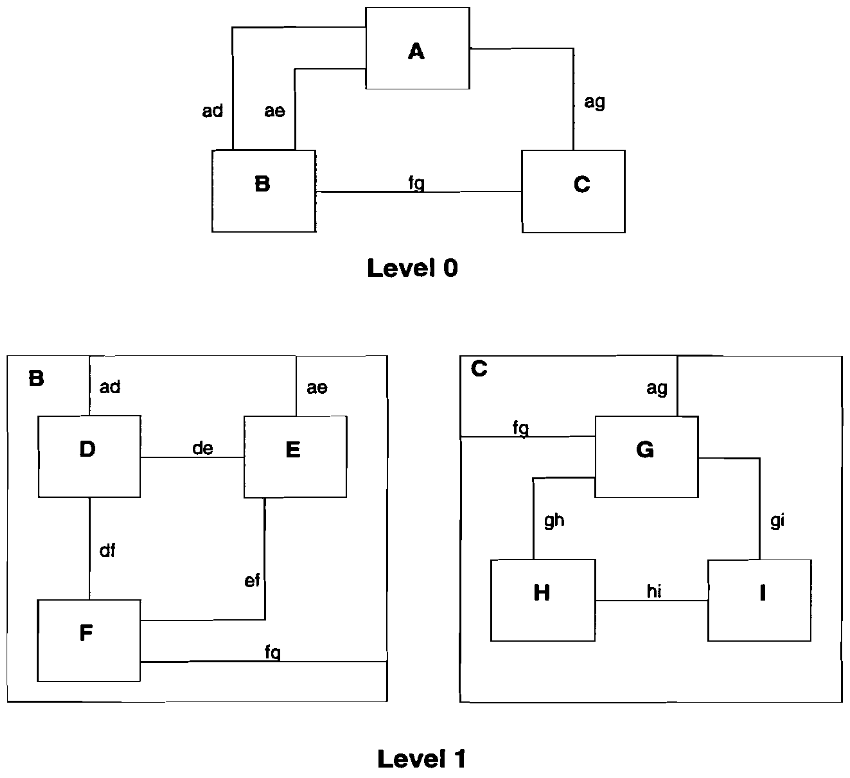


Figure 6.7: Example of an instance structure diagram

6.2.3 POOSL description

POOSL [15] is an acronym for Parallel Object-Oriented Specification Language. It is a formal language for the specification of hardware/software systems. POOSL is the formal basis of SHE. The *object instance model* is formalized in POOSL when dynamic behavior is added. A specification in POOSL consists of a fixed number of statically interconnected proces objects which are able to execute in parallel. Proces objects are connected to a fixed network of channels, through which they can communicate by sending messages. These messages may carry parameters in the form of data objects. Note the correspondence between the *object instance model* and the *POOSL description.*

The communication mechanism POOSL uses is based upon the synchronous—rendez-vous—pair-wise message passing mechanism of CCS [9]. When a process wants to send a message it explicitely states to which channel this message has to be sent. It also explicitely states when and from which channel it wants to receive a message. Here is an example process definition:

```
process class name              CName
instance variable names         IVar1, IVar2
communication channels          ch1, ch2, ch3
message interface               ch1!MessageA(p1)
                                ch2?MessageB(p1, p2)
                                ch3!MessageC
initial method call             Init
instance methods
        Init
           ch1!MessageA(p1);
           ch2?MessageB(p1, p2);
           if p1=IVar1 then
                ch3!MessageC
           fi
```

Note that a process class is described instead of a process—instance—. Each process class has a unique name. Instance variables are data objects local to the process class. A data object incorporates both data and operations. The communications channels and message interface part state all channels and messages known to the process class. The instance methods part defines the dynamic behavior of the process class.

Besides process classes, POOSL supports clusters. A cluster is built from other process classes, which can be either basic or clusters themselves. A process class definition of a cluster consists of a name, communication channels and a message interface. It also specifies a behavior description. The behavior description states the inner process classes of the cluster, hides channels and renames channels.

Besides process classes a *POOSL description* defines a system behavior and data object classes. A data object class definition defines data and operations. Process object classes use instances of data object class as local data or parameters for communications. A system behavior description uses instances of process object classes to define a system.

POOSL supports distribution and implementation boundaries via clusters. It does not support however, the flexibility concurrency boundaries have in the *object instance model.* The first step in formalizing the *object instance model* is to model all process objects. An process object in the *object instance model* becomes a process object in the *POOSL description* and a cluster or composite object becomes a cluster. All declarations of messages and channels are made according the *object instance model.* The POOSL hierarchy will resemble the *object instance model* hierarchy.

Next step is to add dynamic behavior to process objects in the *POOSL description.* When building the *object instance model* the analyst thought about dynamic behavior at a high level of abstraction. Choosing objects and messages, he partitioned overall dynamic behavior. Defining an object's dynamic behavior requires thorough investigation of the object itself, but of its collaborating objects only interfaces need to be understood. Objects have very little influence on dynamic behavior descriptions of objects other than its collaborators.

Messages must be modeled with POOSL's rendez-vous mechanism. Modeling one-way synchronous message passing is the easiest. It can be directly mapped to a send and receive message pair in the dynamic behavior descriptions of the objects. The sending object would have a `ch!Message(parameters)` and the receiving object would have a `ch?Message(parameters)` in its dynamic behavior description. Modeling synchronous message passing with reply requires two consecutive send/receive message pairs. Modeling asynchronous buffered message passing and continuous message passing is much more complicated. Both mechanisms force the process object to be modeled as a cluster with additional proces objects inside to perform the communication task. Interrupt message passing is explicitely modeled in POOSL through a dedicated interrupt method description.

The analyst may discover that dynamic behavior of an object may be too complex to describe. This will require remodeling the *object instance model* at the location of the object and its collaborators. He could move functionality to collaborators, add new objects or decide to make a complex object a composite one. Of course this will have effect on the *POOSL description*, but in all cases changes remain local—requirement 2h—. An *object instance model* with a homogeneous distribution of messages, modeled maximizing independence will minimize iterations back to the *object instance model*—requirement 2e—.

6.2.4 Requirements catalogue

A *requirements catalogue* collects information to be stored during analysis and design. It contains a data dictionary, object to boundary maps, tracebility data and environmental conditions [13].

The *requirements catalogue* is a textual document. Its data dictionary explains the *essential behavior model* in an informal way. About the *object class model* it states meaning of every class and association and their relation to the problem domain. Of the *object instance model* it contains a short description of functionality and intent of every object, message and boundary. It also describes how static structure of the *object instance model* is mapped to the *POOSL description.* This makes the *requirements catalogue* a lengthy document, but it is necessary for another person besides the analyst to understand the models.

SHE models are build starting from the *initial requirements description.* It is usually far from complete and while building the models new requirements appear. This new information is stated in the *requirements catalogue.* Most of the information in the *initial requirements description* will find its way in one of the SHE models, but some items cannot be modeled directly. Examples are preparations for future system expansion, test scenarios, accuracy requirements, environmental conditions such as temperature etc. They are stated in the *requirements catalogue.*

6.2.5 Architecture structure model

The *architecture structure model* describes structure and timing requirements, but no behavior. The model consists of three parts:

- Architecture structure diagrams
- Architecture decisions statement
- Architecture response time requirements

Architecture structure diagrams are free style drawings of architecture structures that are imposed by the initial problem description, by physical geographical constraints and by architecture design decisions. An examples is the OSI-reference model for open systems interconnection. The OSI-model consists of seven layers. Therefore, the OSI-model imposes an abstraction hierarchy constraint on the *object instance model* of the *essential behavior model.* Choosing objects, messages and defining a hierarchy must be conform the *architecture structure diagrams.* Structure imposed by the *architecture structure diagrams* has to be formalized in terms of boundaries in the *object instance diagrams.* The analyst must build the *object instance model* in conjunction with the *architecture structure model.* An example of a physical geographical constraint could be physical separation of elevators from each other and from a central controller.

Because response times are important in hardware/software systems, they must be specified. Response times between environment and system are essential. These response time impose response time requirements between parts of the architecture structure. The *architecture response time requirements* description is the place to state all essential response times—requirement 1g.

Analysis and design are activities that cannot be separated completely. Architectural constraints, as previously described, impose addition constraints on the problem domain. Design decisions to support these constraints are stated and motivated in the *architecture design decisions.* The *essential behavior model* is build in conjunction with the *architecture structure model.* Choosing objects and defining boundaries must be done according constraints in the *architecture structure model.* When the *essential behavior model* is build—analysis—it is biased with high level design decisions from the *architecture structure model.*

The *essential behavior model* and *architecture structure* model must be evaluated intensively with the customer. Especially the graphical models of SHE are designed for this goal. The formal POOSL description, can be used by an expert to explain the message flows and object interaction of the informal *object instance model* to the customer. The *essential behavior model* and *architecture structure model* are relatively implementation independent and freeze the specification of the product.

6.3 Implementation structure model and extended behavior model

Once agreement exists about the essentials, design may take place. The *implementation structure model* visualizes implementation structure that must be designed before the behavior description can be extended accordingly. The *implementation structure diagrams* are just like the *architecture structure diagrams* free style drawings. In the *implementation decisions* *statement* decisions are described about hardware or software implementation, concurrency, communication protocols, bus-structures etc. The *implementation response time requirements* state all response time requirements, both essential and implementation specific.

Design and implementation decisions add information. This requires an extension of the *essential behavior model.* Because the *essential behavior model* captures the essence, design and implementation decisions are added to a separate model, the *extended behavior model.* The *extended behavior model* is based on the *essential behavior model,* but contains additional objects, communications, boundaries etc. Architecture structure decisions are formalized in the *essential behavior model,* while implementation structure decisions are formalized in the *extended behavior model.* The *extended behavior model* serves as an implementation blueprint. However, this does not mean that design ends here. An implementation blueprint is a model which is detailed enough to describe system and environment the way the customer intended them to be. Further design before actual implementation is done according the implementation blueprint without further input from the customer.

At this moment not much more about the *implementation structure model* and *extended behavior model* can be said. It was not part of this masters project and this part of SHE is still under development.

Chapter 7

A SHE model of the elevator problem

7.1 Model overview

Previous chapter showed the principles of the SHE—Software/Hardware Engineering—method. This chapter describes how the SHE method is applied to the elevator problem. It shows the essential behavior model and architecture structure model of the elevator problem. The essential behavior model consists of five parts:

- Initial requirements description
- Object class model
- Object instance model
  - Message flow diagram
  - Instance structure diagram
- POOSL description
- Requirements catalogue

The architecture structure model consists of three parts:

- Architecture structure diagrams
- Architecture structure decisions statement
- Architecture structure response times requirements

The initial requirements description is the problem statement as stated in chapter 2. The *architecture structure model* contains initial requirements together with high level design decisions concerning architecture structure. The second part of the essential behavior model is the object class model. It is much like the object class model of OMT. This object class model serves as an initial study of the problem domain in the SHE method. It shows a first inventory of candidate objects for the object instance model. The OMT study of the object class model is used to build the SHE object class model. Figure 6.2 shows the object class model.

With the initial requirements description, the object class model and architecture structure model at hand, the object instance model can be built. It is a process of adding objects and communications between objects to achieve system functionality as described in the initial requirements description. The essential behavior model must only contain essential objects. The object class model does not contain all the object classes necessary for the object instance model, but while adding communications between the objects the necessary additional objects will be found. You may for instance need information from another object that does not exist yet, or you may need a service of it. Communications help you find new objects and objects on their turn will result in new communications.

Next step is to build the POOSL description. The object instance model forms the basic structure the POOSL description. Every object is described with its dynamic behavior.

The requirements catalogue collects information during analysis and design. It contains a data dictionary, object to boundary maps and environmental conditions. This chapter does not contain the requirements catalogue because focus has been mostly on the object instance model. However, some of this information will be described along with the description of the subsequent models in the following sections.

7.2 The architecture structure model

Figure 7.1 shows the architecture structure model of the elevator problem. This model consists of modules connected through communication channels. Two main parts can be distinguished:

- Control part
- Elevator part

Elevators are controlled by the control part. Because the system must handle summons requests and destination requests, control is divided into two parts: **Central Control** and **Individual Control**. **Central Control** is responsible for correct handling of summons requests. It accepts summons requests from **Floor Passenger** and efficiently schedules each elevator through communications with each the **Individual Control**.

Each **Individual Control** is responsible for the operation of an elevator. It accepts destination requests from **Elevator Passenger**, checks summons requests with the **Central Control** and issues commands to the **Elevator Mechanism** for elevator movement. **Individual Control** knows when an elevator arrives at a new floor through communications with **Floor Sensors** and wheter or not an elevator is filled to capacity through communication with an **Overweigh Sensor**.

Elevator Mechanism is a module that allows safe operation of an elevator cage. It is able to handle **Doors**, **Motor** and **Audible Alarm** control. The **Elevator Mechanism** performs commands from **Individual Control** like moving up/down or stopping the elevator. Elevator movement and door opening/closing is done safely. An **Elevator Passenger** may request the **Elevator Mechanism** to hold doors open or to stop and halt at the next floor the elevator arrives. Whenever movement of an elevator is halted by a passenger, the **Elevator Passenger** notifies the **Individual Control** about the suspension of operation.

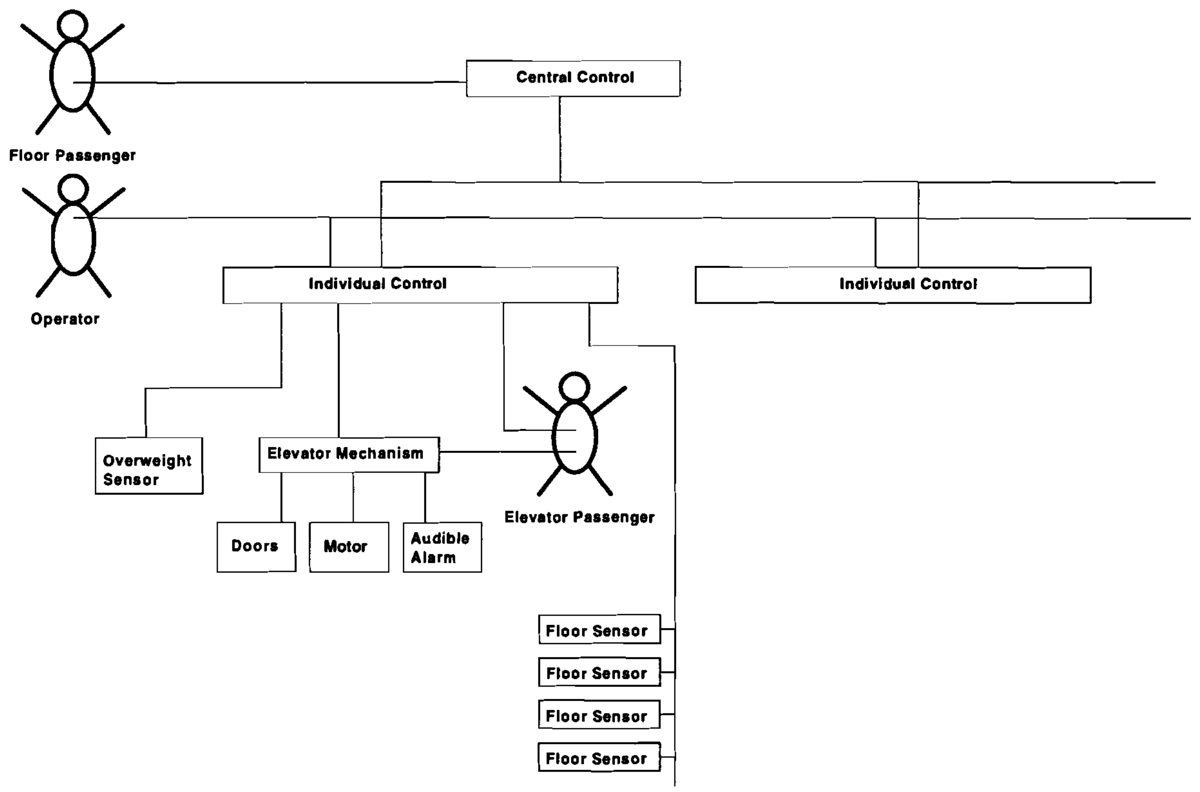


Figure 7.1: Architecture structure model

ENVIRONMENT	SYSTEM
ElevatorPassenger	ElevatorMechanism
FloorPassenger	ElevatorControl
Operator	ElevatorMaintenanceControl
OverweightSensor	FloorHandler
FloorSensor	ElevatorsSupervisor
ElevatorMotor	PassDestinationsInputControl
Doors	DestinationsList
AudibleAlarm	DestinationsAdministrator
	PassSummonsInputControl
	SummonsList
	SummonsAdministrator

Table 7.1: Objects of the flat object instance model

Operator communicates with Individual Controls to put an elevator in and out one of its possible maintenance modes. Whenever all elevators must be in maintenance mode, this must done by putting each elevator in maintenance mode separately.

At this time a few response time requirements need to be mentioned. All communication between a person and the system should be handled within todays human/system communication timing requirements. This means for example that a destination acceptance response should be within 0.3 second. Another important response time requirement concerns elevator control. Whenever an elevator reaches a floor within 8 inches, the system computes the desired movement of an elevator. If, for example, the elevator should stop at this floor, the system must control the elevator motor in time. Of course, a quantitive value of this response time requirement depends on implementation details such as type of motor. At this time only a qualitive response time requirement can be given: system response must be fast enough to allow correct operation. In a real elevator control design—instead of this case study—much more response time requirement study should be done. It would for example be very important to know the maximum allowed acceleration/deceleration of a passenger in the elevator cage. In this case study, awereness of these response time requirements suffices.

7.3 The essential behavior model

At this point, building the essential behavior model is a matter of building the object instance model and the POOSL description. The first step is to built a message flow model. It is difficult to built a correct hierarchy while adding objects and communications to the model. Therefore, initially a flat object instance model is built, while adding hierarchy to the model in a succeeding step. This is neither a top-down nor a bottom-up approach. One may start in some intermediate level of the hierarchy and combine both methods. In the elevator problem, modeling started on the bottom level. Table 7.1 shows all the objects in the initially flat object instance model. Once the flat model is finished, clusters or a composite objects are identified. Grouping objects is a bottom-up approach. In the elevator model this approach is used and resulted in a three level hierarchy. The hierarchy must comply with the architecture

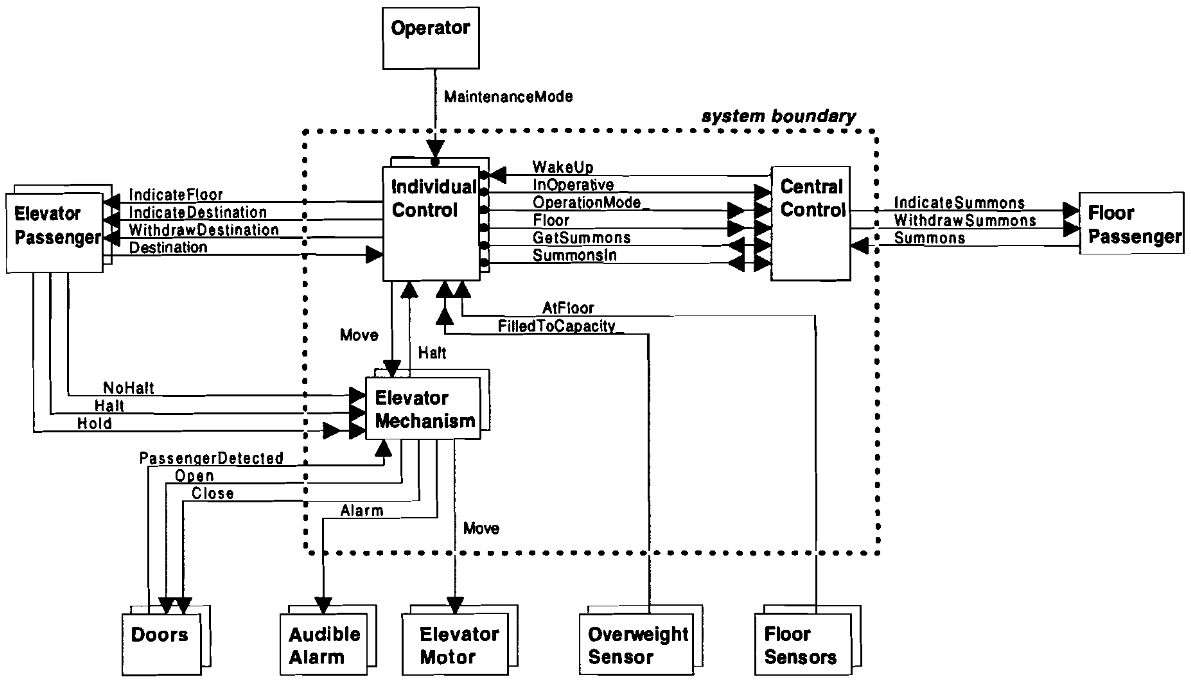


Figure 7.2: Level 0 of the message flow diagram

structure model.

The following two sections show the two parts of the object instance model. It is explained in a top-down fashion. It describes the objects, their connections and communications. This will give an idea of the dynamic behavior of the system. In section 7.3.3 a detailed description of the dynamic behavior is given.

7.3.1 Message flow diagrams

As you can see, Figure 7.2 shows the top level of the message flow diagram. The most important part is the system boundary. Objects outside the system boundary are objects in the environment and will not be designed. Objects inside the system boundary form the system to be designed and implemented. The easiest part to understand is the environment objects. Here is a list of the environment objects and their description:

- ElevatorPassenger is an object to represent a real-world passenger inside an elevator. This passenger is able to issue destination requests to the system with a Destination message. The message contains information about the requested destination floor. The system notifies the passenger about pending destination requests with the IndicateDestination and WithdrawDestination messages. The system gives the passenger the current floor number with the AtFloor message. A passenger is able to halt an elevator with the Halt message or enable it again with the NoHalt message. Halting an elevator causes the elevator cage to stop and halt at the next floor with its doors open. Hold_ holds the doors open when the elevator is at a floor.

- FloorPassenger is an object to represent a real-world passenger at a floor who wishes to travel with an elevator. He notifies his request to the system with a Summons message. This message contains information about the location of the passenger and the direction he wishes to travel in. The request is for any of the available elevators. The system decides which elevator the passenger will travel with. The system notifies the passenger about pending summons requests with the IndicateSummons and WithdrawSummons messages.

- Operator is an object to represent the person who has control over the operation mode of an elevator. He can put an elevator in normal operation, maintenance mode and lock and clear mode. Maintenance mode is a mode in which an elevator does not stop for any summons requests, but handles its destination requests as in the normal operation mode. In the lock and clear mode, the elevator clears its destination list and ignores destination and summons requests. The operator puts the elevator in one of these three modes with a MaintenanceMode message.
- FloorSensors is an object to represent a group of real-world sensors that detect if an elevator reaches a floor within eight inches. The AtFloor message sent to the system contains the floornumber.
- OverweightSensor is an object to represent a real-world sensor that senses if an elevator is filled to capacity. If an elevator is filled to capacity it should not stop for any pending summon requests at the floors it passes. The overweight sensor object has its information always available to the system. The system only needs to accept the FilledToCapacity_ message which contains a boolean value.
- The AudibleAlarm is an object to represent a real-world audible alarm. The AudibleAlarm sounds as long as the ElevatorMechanism is halted by an Halt message from an Elevator-Passenger.
- The Doors is an object to represent the real-world doors of an elevator, together with a mechanism to detect passengers passing through. Doors open at an Open message and close at a Close message. Every time a passenger passes through, the Doors send a PassengerDetected message.
- The ElevatorMotor is an object to represent a real-world elevator motor. It accepts commands through the Move message. This message starts the motor moving up or down or stops the motor. It is save to switch the motor directly from up to down and vice versa.

When the environment has been analyzed and its objects have been defined it is time to look at the system itself. We can distinguish three major parts. An IndividualControl part, a CentralControl part and an ElevatorMechanism part.

The IndividualControl part encompasses all the necessary objects to describe all the functionality that is specific to a single elevator. This means handling destinations, control its movement and update the display inside the elevator cage. It must provide information for the CentralControl about its operationmode, position and notify the CentralContral if it cannot continue operation. The IndividualControl queries the CentralControl about pending summons requests it may have to handle. If there is nothing to be done for an IndividualControl it goes to "sleep" in its idle state. In this state the elevator cage is at a floor with its doors open.

The CentralControl part describes all the functionality that is not specific to a single elevator, but for all elevators as a whole. It handles summons requests, wakes up idle elevators and updates the summons panel. The CentralControl wakes up an idle elevator if this elevator must handle a summons request or if one of the other elevators fails to continue operation. In Figure 7.3 the internals of the IndividualElevator and CentralControl are shown.

The ElevatorMechanism part is responsible for safe operation of an elevator cage. The elevator mechanism sends the Halt message to the system if it is unable to continue operation, because of an halt request from an elevator passenger. It accepts the Move message. This message contains information about the desired movement of the elevator cage: up, stop or down. If the cage is stopped, the doors are automatically opened. It will only start moving again after an up or down command and all passengers have entered or leaved the elevator cage. The doors will automatically be closed. With a Hold_ the elevator cage will hold at a floor with its doors open. The Halt message from an ElevatorPassenger halts the elevator at a floor with its doors open.

Let's take a closer look at level 1 now. The FloorSensors is a collection of 40 FloorSensors, one at each floor. If an elevator reaches a floor within 8 inches, the FloorSensor sends the AtFloor message to the system with information about the floor number. This message is handled by the FloorHandler in the IndividualControl part. The IndividualControl is a cluster of four objects:

- FloorHandler handles AtFloor messages from the FloorSensors. If this message arrives it notifies the ElevatorPassenger through an IndicateFloor message and notifies the ElevatorControl that the elevator has arrived at a new floor with an AtFloor message.

- ElevatorControl controls the movement of the elevator cage by sending Move messages to the ElevatorMechanism. It checks destination requests and summons requests. With the GetDestination message the ElevatorControl asks the DestinationsHandler if a destination request is pending for a particular floor. The DestinationsIn message returns information about pending destination requests in some direction relative to a floor except for the floor itself. For instance, the DestinationsIn(3, down) message returns true if there is a destination request pending at floor 0, floor 1 or floor 2. Summons information is retrieved in the same manner. If nothing is left to be done, the elevator goes to "sleep" in its idle state. A WakeUp message from the DestinationsHandler or CentralControl will wake it up again. If an elevator is FilledToCapacity_ or in Maintenance mode it will not handle any summons requests. The Normal message puts the elevator from maintenance mode back to normal mode. If an ElevatorPassenger sends a Halt message, the ElevatorMechanism sends a Halt message to the ElevatorControl. This causes the ElevatorControl to notify the CentralControl it is temporarily InOperative.

- The DestinationsHandler handles information requests from the ElevatorControl and wakes it up if a new destination request arrives. While keeping an administration of destination requests, it updates the DestinationsPanel accordingly with Light messages. New destination requests can be Enable-ed, Disable-ed or the entire destination administration can be cleared with the ClearAll message.

- ElevatorMaintenanceControl is an object that interprets MaintenanceMode messages from the Operator and instructs the DestinationsHandler and ElevatorControl to operate accordingly the requested mode. If the Operator requests the maintenance mode, the

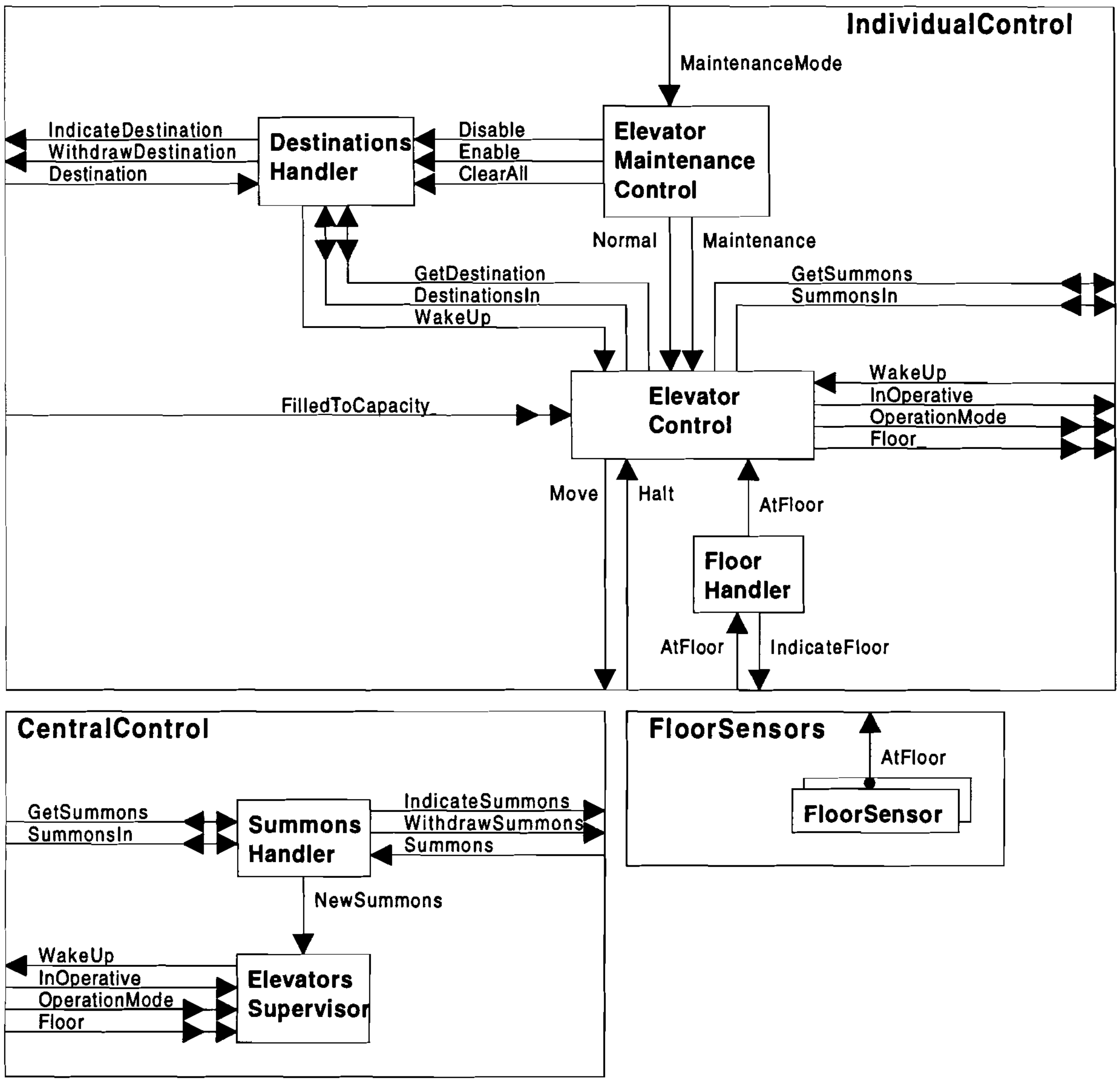


Figure 7.3: Level 1 of the message flow diagram

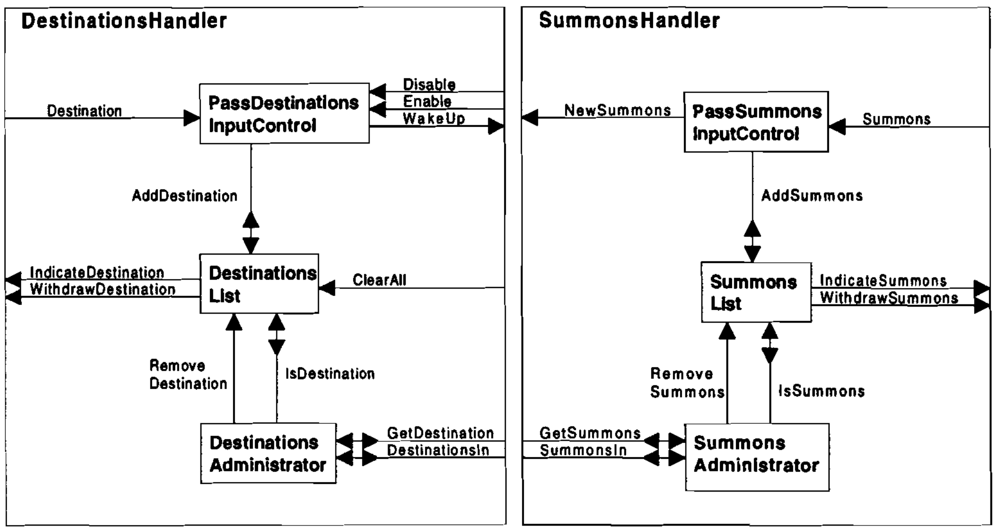


Figure 7.4: Level 2 of the message flow diagram

Maintenance message is send to the ElevatorControl. This causes the elevator not to stop for any summons requests. The ElevatorControl informs the CentralControl it is InOperative for a while. In the lock and clear mode, the ElevatorMaintenanceControl Disable-s new destinations requests and clears the destination administration in the DestinationsHandler with the ClearAll message.

The second major part of the system is the CentralControl. It is a cluster of two objects:

- The SummonsHandler handles information requests from the ElevatorControl and alerts the ElevatorsSupervisor if a new summons request arrives. While keeping an administration of the summons requests, it updates the SummonsPanel accordingly with Light messages.
- The ElevatorsSupervisor WakeUps elevators. This may occur when a new summons request arrives or an elevator becomes inoperative. If a new summons request arrives, the ElevatorsSupervisor wakes up the closest idle elevator. If an elevator becomes inoperative, the ElevatorsSupervisor wakes up the first idle elevator it can find and lets it move in the same direction the currently inoperative elevator was going before it was inoperative.

Level 2 is the bottom level of the object instance diagrams. It is shown in Figure 7.4. The DestinationsHandler is a cluster of two objects:

- The PassDestinationsInputControl is an object that handles passenger destinations input. A passenger supplies the system with destination requests through the Destination message. This input can be Disable-ed and Enable-ed by the ElevatorMaintenanceControl. The PassDestinationsInputControl sends a WakeUp message to the ElevatorControl each time a new destination request arrives and has been added to the DestinationsList with the AddDestination message.

- The DestinationsList holds all destination requests and informs the ElevatorPassenger about pending destination requests with IndicateDestination and WithdrawDestination messages. ClearAll clears all destination requests and AddDestination is used to add destinations to the list. IsDestination replies with a message containing a boolean whether or not a particular floor has a destination request pending. RemoveDestination removes a destination from the list.
- DestinationsAdministrator is able to answer the questions from the ElevatorControl about pending destination requests. The message GetDestination replies with a message containing a boolean which is set if the appointed floor has a destination request pending. The destination request will be removed. The DestinationsIn message returns information about pending destination requests in some direction relative to a floor except for the floor itself. For instance, the DestinationsIn(3, down) message returns true if there is a destination request pending at floor 0, floor 1 or floor 2.

The SummonsHandler is very similar to the DestinationsHandler. The exceptions is the PassSummonsInputControl. It cannot be disabled and instead of the WakeUp message, it sends the NewSummons message to the CentralControl each time a new summons request arrives. All other objects and messages are similar to the ones of the DestinationsHandler.

7.3.2 Instance structure diagrams

Figures 7.5, 7.6 and 7.7 show the instance structure. They show objects connected through channels. A channel exist between two objects whenever they have at least one message flow between them. Each channel must have a unique name. This name may be arbitrary, but in order to improve readability of the diagrams it is best to use a proposed convention. In the elevator problem, all leaf objects—objects that are neither a composite object nor a cluster—have a two character abbreviation of their name. Table 7.2 shows their name abbreviations. The name of a channel between two collaborating objects is the concatenation of both object name abbreviations. The concatenation must be in alphabetical order. There exist a difficulty if there are several instances of an object—a multiple object—. In the elevator problem for example, the model contains four elevators. This implies the communication between four DestinationsLists and four DestinationsPanels. To identify an object, a number is needed besides its name. This number may then be used to create the channel name as well. To get unique names for the channels between multiple objects a number is added after the concatenation of abbreviations. See Figure 7.5. The channel name between ElevatorControl and OverweightSensor is ecos#, where # is a number between 0 and 3. Inside the IndividualControl the name of the channel is ecos, see Figure 7.6. This channel name is local to the object IndividualControl and there exist no conflict between multiple channels.

The instance structure diagram does not only show interconnections. The diagram is also a basis for architecture structure. The architecture is represented by drawing boundaries around groups of objects. There are four different types of boundaries:

- Abstraction boundary
- Concurrency boundary
- Distribution boundary
- Implementation boundary

NAME	ABBREV.
AudibleAlarm	aa
DestinationsAdministrator	da
DestinationsList	dl
Doors	do
ElevatorControl	ec
ElevatorMotor	em
ElevatorPassenger	ep
ElevatorsSupervisor	es
FloorHandler	fh
FloorPassenger	fp
FloorSensor	fs
ElevatorMaintenanceControl	mc
ElevatorMechanism	mh
Operator	op
OverweightSensor	os
PassDestinationsInputControl	di
PassSummonsInputControl	si
SummonsAdministrator	sa
SummonsList	sl

Table 7.2: Name abbreviations of leaf objects

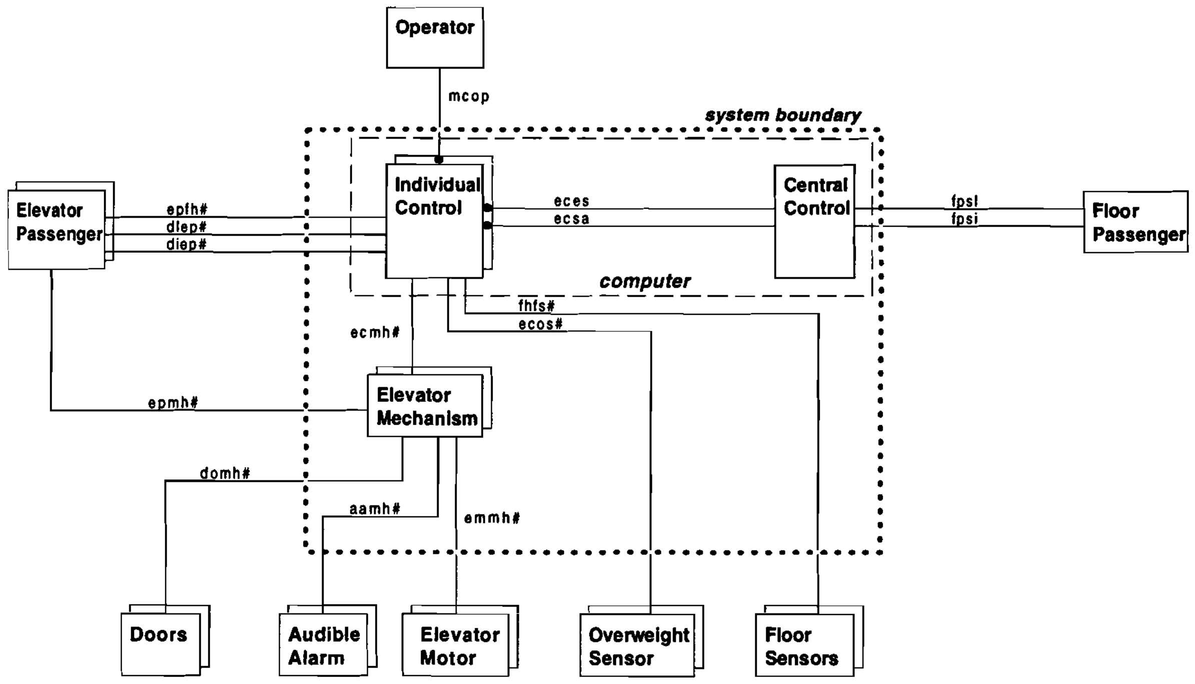


Figure 7.5: Level 0 of the instance structure diagram

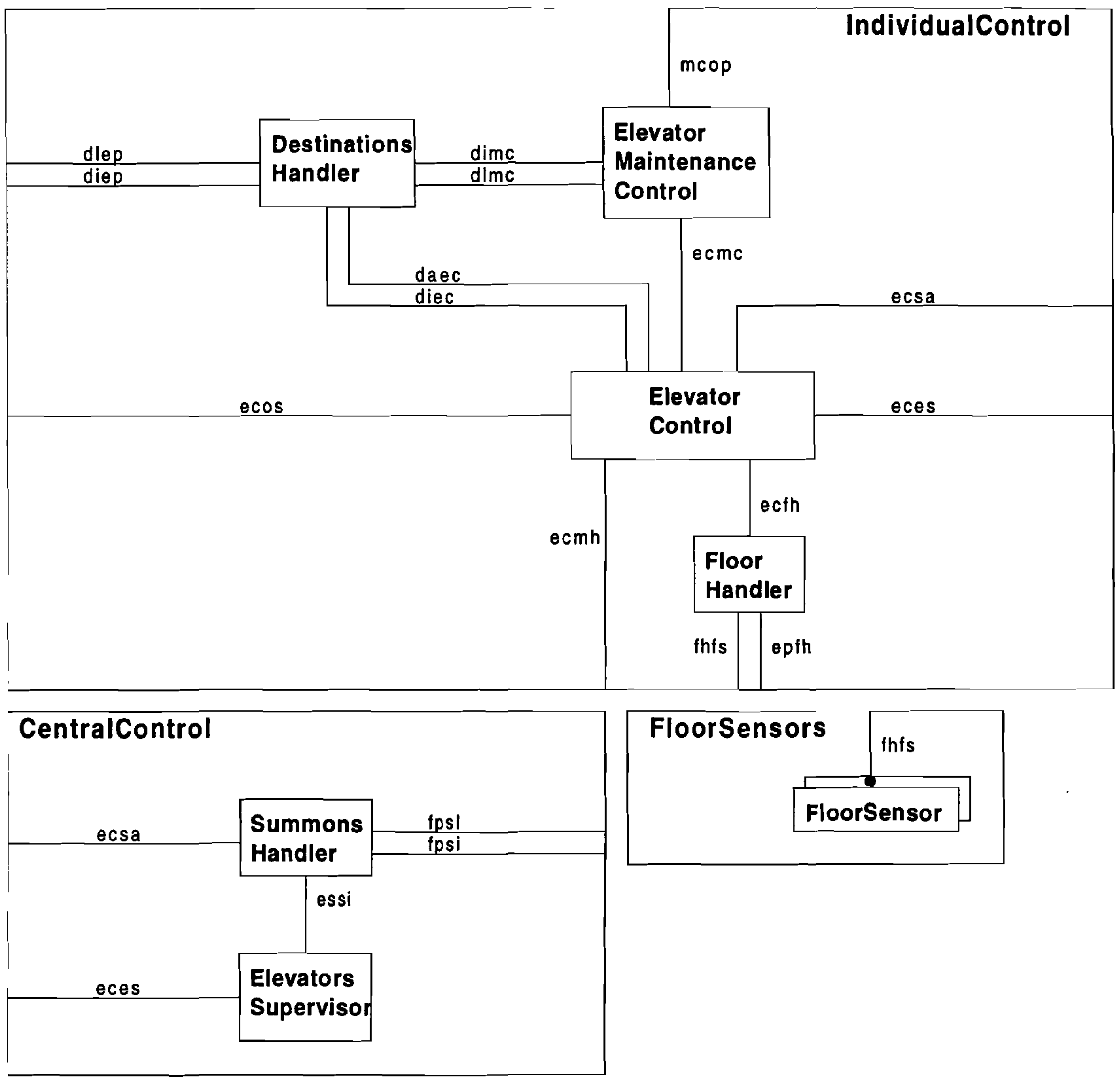


Figure 7.6: Level 1 of the instance structure diagram

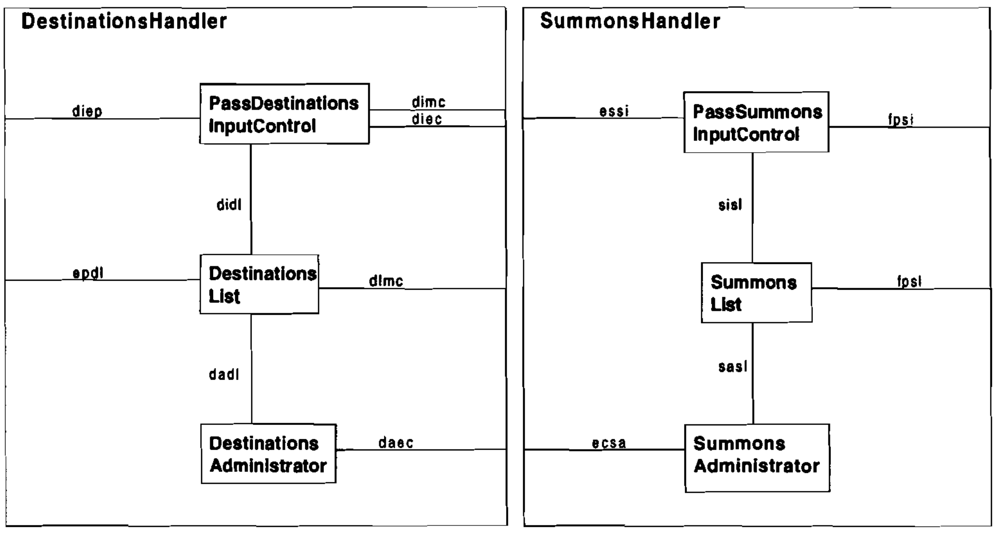


Figure 7.7: Level 2 of the instance structure diagram

The abstraction boundaries are in fact the leaf objects itself, composite objects and clusters. Concurrency boundaries show of course concurrency between groups of objects. At this point a concurrency boundary exist around every leaf object. This means that all leaf objects operate concurrently, but inside the objects the operation is sequentially. The instance structure diagrams of the elevator problem does not contain distribution boundaries yet, but does have one implementation boundary. An implementation boundary exist around IndividualControl and CentralControl. These clusters contain all objects that control four elevators as described in the problem statement and will be implemented in software on a single computer that is capable of performing this task.

7.3.3 POOSL description

The POOSL description describes the static structure of the object instance model together with the dynamic behavior of the objects. The POOSL description of the elevator problem is in Appendix A of this report. The description is divided into three parts:

- System description
- A system of process objects
- A system of data objects

This is a top-down order of the elevator problem. The system description shows the static structure of connected process objects that comprise the system. A process object represents an object in the object instance model. So, to build a POOSL description, one starts with describing all objects of the object instance model as process objects in the POOSL description. At a later stage the dynamic behavior of each object is added to the POOSL description. The data objects are an extension of the standard data types of the POOSL language. For more information about POOSL see [15].

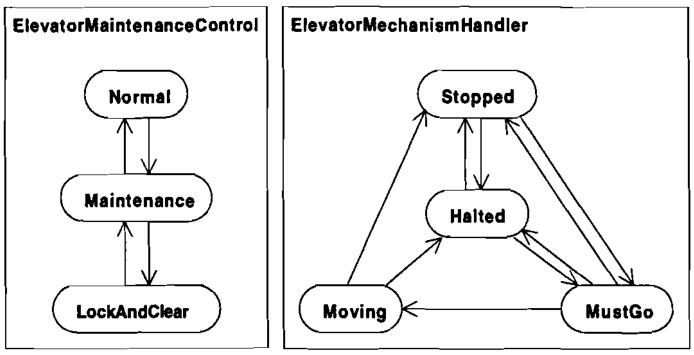


Figure 7.8: States of the ElevatorMaintenanceControl and the ElevatorMechanism

The description of most objects is a very straightforward sequence of sending and receiving messages. The dynamic behavior of the objects in the environment is not described, only their interfaces. Three process objects need further explanation: ElevatorMaintenanceControl, ElevatorControl and ElevatorMechanism.

The dynamic behavior of a process object is some sequence of sending and receiving messages. What is received or send depends on a certain "state" the process object is in. The "state" is composed of the execution point in the dynamic behavior description and the values of the variables of the process object. To understand the behavior of a process object, it may be necessary to take subsets of all possible states and name them.In the elevator problem the process objects ElevatorMaintenanceControl and ElevatorMechanism have variables that hold a value about the state the are in.

The ElevatorMaintenanceControl can be in one of three states: Normal, Maintenance and LockAndClear. The variable MaintenanceMode holds the value of the state. Figure 7.8 shows how the value of the variable MaintenanceMode of the ElevatorMaintenanceControl is allowed to change, and hence the state of the ElevatorMaintenanceControl. In the Normal state, the elevator handles both destination requests and summons requests. In the MaintenanceMode the elevator is instructed not to handle any summons requests any more. The LockAndClear state causes the elevator to not handle both destination requests and summons requests. All pending destination requests will be cleared as well. The elevator does not accept any destination request input. The ElevatorMaintenanceControl changes state upon the reception of messages from the Operator.

The ElevatorMechanism has two variables that are very important to its state: MoveTo and MoveState. Figure 7.8 shows the variable MoveState. It can be one of four states. Normal operation is a cycle through three states: Stopped–MustGo–Moving. In the Stopped state the elevator is at a floor with its doors open. If the ElevatorMechanism receives a Move(up/down) message a transition to the state MustGo occurs. This state is active until no more passengers enter or leave the elevator. The doors will then be closed and the elevator starts moving. At this time the ElevatorMechanism is in the Moving state. A Move(stop) message causes the elevator to stop and open its doors. Now, the ElevatorMaintenanceControl is in its Stopped state again.

When the Halt message arrives, the elevator must halt at the next floor it arrives on and must open its doors. Therefore, at any of the three previous described states a transition may occur to the Halted state. While halted, the ElevatorMechanism still handles Move messages and updates the variable MoveTo. This variable holds the direction the elevator must go to. If the NoHalt message arrives, ElevatorMechanism goes to the Stopped or MustGo state depending on the variable MoveTo. In the Stopped state the variable MoveTo always holds stop and in the MustGo state it always holds up or down. It is also possible to go from the MustGo state to the Stopped state. This will happen when a Move(stop) message arrives. Variable MoveTo will then be set to stop. For a detailed description, see the POOSL description itself in Appendix A.

The ElevatorControl is not difficult to understand concerning states, but its actions to AtFloor and WakeUp messages need further explanation. The initiative of elevator control lays at the ElevatorControl. The OperationMode of an elevator is Idle, Up or Down. In the Up or Down mode, the elevator handles destination requests and summons requests in the up or down direction. It will not reverse direction until all requests in a direction are handled. The precedence of requests is as follows:

1. Destination requests
2. Summons requests in the same direction as the OperationMode
3. Summons requests at floors in the direction of OperationMode, but with a request in the opposite direction. The elevator will travel to the farthest summons request before reversing its direction.

The elevator will not handle any summons requests if it is filled to capacity or it is in maintenance mode. The variable MaintenanceMode holds Maintenance or Normal if the elevator is in maintenance mode or it is not in maintenance mode.

Whenever the elevator arrives at a new floor the ElevatorControl is send the AtFloor message. This message contains the floornumber. The ElevatorControl queries the DestinationsAdministrator and the SummonsAdministrator to see if it must stop at this floor according the requests precedence. If there are no pending requests in the direction of the OperationMode, the elevator reverses its direction and continues oparation. If there is nothing to be done in this direction as well the OperationMode switches to the Idle state. In this state the elevator is at a floor with its doors open. The elevator will start operation again if the ElevatorControl receives a WakeUp message. This message is send to it when a new destination request arrives or the CentralControl decides to wake it up.

The ElevatorMechanism sends the Halt message to the ElevatorControl if it has a Halt message received from the ElevatorPassenger. This is nothing to worry about, but the CentrolControl must know that the elevator is temporarily inoperative. Because halting an elevator does not effect the operation of itself, all that needs to be done is to send the InOperative message to the CentralControl. This message is also send to the CentralControl if the elevator switches to maintenance mode, because it does not handle summons requests any more.

7.4 Heuristics with the SHE method

The heuristics presented in this section serve as guidelines to build the essential behavior model and architecture structure model. These guidelines may help you, but they are not compulsory. The essential behavior model consists of:

- Initial requirements description

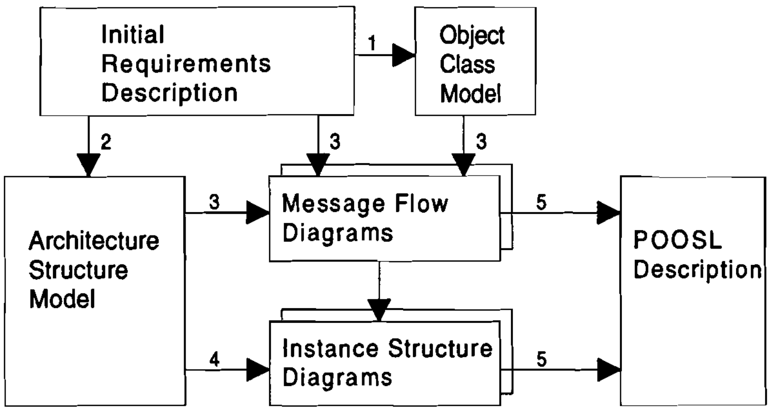

Initial Requirements Description
1
Object Class Model
2
3
3
Architecture Structure Model
3
Message Flow Diagrams
5
POOSL Description
4
Instance Structure Diagrams
5

Figure 7.9: The information flow when building the models that comprise the essential behavior model

- Object class model
- Object instance model
  - Message flow diagrams
  - Instance structure diagrams
- POOSL description
- Requirements catalogue

The architecture structure model consists of:

- Architecture structure diagrams
- Architecture decisions statement
- Architecture response time requirements

Of course modeling starts with the *initial requirements description.* It is the problem statement that contains information about the system from the customer's point of view. From this starting point the other models will be built in several phases as depicted in Figure 7.9. The numbers near the arrows indicate the phase.

The essential behavior model is build in five phases:

1. An initial study is done of the problem domain. The *object class model* is build from the *initial requirements description*
2. Essential architecture structure requirements imposed by the *initial requirements description* along with physical geographical constraints are put in the *architecture structure model*
3. The *initial requirements description* and the *architecture structure model* are used to create the *message flow diagrams.*

4. The *message flow diagrams* and the *architecture structure model* are input for the *instance structure diagrams.*
5. The *POOSL description* is actually build in two sub phases:
   (a) A textual form of the basic system structure is build from the *message flow diagrams* and *instance structure diagrams.*
   (b) Dynamic behavior of every object is described.

The *requirements catalogue* which contains a data-dictionary, object to boundary maps, traceability data and environmental conditions is build from the *initial requirements description* and updated with information with every analysis and design phase. The succeeding section describe each phase building the *essential behavior model* in more detail.

7.4.1 Phase 1 and 2 in building the essential behavior model

SHE is capable of handling systems of great complexity. These systems may require an initial study of the problem domain. Building the *object class model* serves two purposes: 1) it is an initial study of the problem domain and 2) classes of this model serve as a first candidate list for objects in the *object instance model.* Instead of giving a thorough description on how to build a class model, I would like to refer to the good work done by Rumbaugh et.al. [14]. Rumbaugh et.al. give an exhaustive description and a rich set of heuristics for class models.

The next step is to build the *architecture structure model.* It contains essential architecture structure requirements imposed by the problem description. Examples are abstract layering of the OSI-reference model or physical separation of elevators from each other and from a central controller. Though the *architecture structure model* requires information from the *initial requirements description* it also requires common knowledge of the problem domain.

7.4.2 Phase 3 in building the essential behavior model

In the third phase, the *message flow diagrams* are build from the *initial requirements description*, the *object class model* and the *architecture structure model.* The *object class model* serves as a first list of candidate objects. Because this list will not contain all required objects—most of the classes of the *object class model* belong to the environment—, find objects in the *initial requirements description* as well. Because an essential model is build here, every object must be essential to the problem.

Another important issue to take into account when choosing objects is the *architecture structure model.* This model describes high level architecture as is imposed by the problem description. Architecture structure boundaries may not intersect with objects. Objects must be carefully chosen to locate them inside boundaries.

Next step is to add message flows to the objects found thisfar. While doing so, additional objects will be found. Message flow diagrams are initially build from an intermediate level of abstraction. They are then extended to higher and lower levels. If it is difficult to stay at one level of abstraction doesn't matter. Objects of different levels are put together in the diagram along with their message flows and hierarchy is added to the model at a later stage.

When choosing objects and adding messages, overall functionality is partitioned over objects. To make the model cohesive, organize the model for maximum independence. A good model keeps related things together and unrelated things separate [17]. Changes will then remain local. When messages are added to objects, new objects can be found. This occurs for example when an object needs functionality—a service—of another object to accomplish its task and this object is not exist in the model yet. A service request is modeled as a message between collaborating objects. Messages helps you find objects. New objects require adding messages. Therefore, interaction exist between finding objects and messages.

Try to avoid continuous message flows and buffered message flows. They cannot be described in POOSL directly and therefore make the POOSL description harder to understand. Giving good names to the message flows is very important. A message flow must be named from the point of view of the initiator of the communication.

7.4.3 Phase 4 in building the essential behavior model

Once you have the *message flow diagrams* it is very easy to make the *instance structure diagrams*. The object hierarchy of the *object instance structure* is exactly the same as the hierarchy of the *message flow diagrams*. Whenever there exist message flows between two objects, they are replace by a channel. Name these channels. It is a convention to build channels names from object name abbreviations. This makes it possible to see what objects a channel connects just by looking at its name. On page 71 the construction of channel names is explained.

In the *object instance structure diagrams* the boundaries of architecture structure from the *architecture structure model* are formalized. When the *message flow diagrams* were build, object were chosen not to intersect with any of these boundaries. Objects themselves are abstraction boundaries. So, only concurrency, distribution and implementation boundaries need to be added.

The *instance structure diagrams* serve as a skeleton for the construction of the *POOSL description*. The *instance structure diagrams* contain concurrency, distribution and implementation boundaries. However, POOSL doesn't support them directly, but they can be specified as clusters.

7.4.4 Phase 5 in building the essential behavior model

The first step in building the *POOSL description* is transforming the *message flow diagrams* and *instance structure diagrams* into a textual form as a *POOSL description* skeleton. This step is mechanical. Objects become process objects, message flows are declared in the message interface part of process objects and communication channels are declared. Composite objects and clusters will be supplied with a behavior description. Of every process object its functionality and operation must be described in an informal way as a comment. This will make the *POOSL description* much more readable.

In the second step the dynamic behavior of the process objects is coded in POOSL. The hierarchy in the *object instance model* has reduced the complexity of every object to an acceptable level. The message flows of a process object and the informal description documented in the *POOSL* description in the first subphase serve a starting point to build the dynamic behavior description of the process object. When it is too difficult to make the description, one could start with a less complex object. Objects at the lowest level in the hierarchy are usually less complicated than higher levels. When describing the dynamic behavior is difficult, even when all its collaborating objects can be described, the *object instance model* needs remodeling. Objects must be added to the *object instance model* or the complex object must be made a composite object to reduce object complexity. This will however influence dynamic behavior descriptions of collaborating objects. They must be checked and modified if necessary. If messages are homogeneous distributed across the *object instance model* and the model is build maximizing independence, modeling iterations are minimized and modifications will stay local.

Keep the description of the objects in the *POOSL description* ordered in hierarchy level and order each level alphabetically.

7.5 Review

This section reviews the application of SHE on the elevator problem. Though especially items of the applied SHE part are reviewed, succeeding sections will give also—a less extensive—review of other SHE parts. The applied SHE part consists of all parts of the *essential behavior model.* They are build in conjunction with the *architecture structure model.*

The SHE review is based on a single case study. Of course this is not enough to fully evaluate an analysis and design methodology, but gives a good first impression. Future studies of other complex embedded systems are required.

7.5.1 Four framework quadrants

SHE framework consists of four quadrants, see Figure 6.1. The idea of using four quadrants is to enhance awareness about essence versus implementation and behavior versus structure [12]. The top half of the framework—the *essential behavior model* and *architecture structure model*— captures essence while the bottom half—the *extended behavior model* and *implementation structure model*—captures implementation. The left half—the *essential behavior model* and *extended behavior model*—captures behavior, while the right half—the *architecture structure model* and *implementation structure model*—captures structure.

In SHE not all models are strictly separated according the four quadrant principles. Take for example:

- Boundaries. Boundaries, are structuring concepts but are modeled in the *essential behavior model* and the *extended behavior model,* both behavior quadrants.

- The *initial requirements description.* It contains both behavior and structure as well essentials and implementations. In SHE it is part of the *essential behavior description.* The *initial requirements description* serves as an information source during analysis and specification. Its information will be formalized in essential models: the top half of the quadrant framework. So, probably a better place for the *initial requirements description* would be outside of the quadrant framework, above the essential part. This would emphasize that the *initial requirements description* is input for both *essential behavior model* and *architecture structure model.*

- The *requirements catalogue.* It collects information to be stored during analysis and design [13]. It contains a data dictionary, object to boundary maps, tracebility data and environmental conditions. It does not contain essential dynamic behavior. Why put it in the *essential behavior description*? Maybe, a better place would be outside the quadrant framework.

- The *object class model.* It serves as an initial study of the problem domain and as a first candidate list of objects for the *object instance model.* The *object class model* does not contain any dynamic behavior. Yet, it is part of a behavior quadrant.

Not strictly keeping to the quadrant principles may be cause for confusion or worse, misplacing information. SHE separates essence from implementation, but mixes structure with behavior in the quadrant framework. Restructuring parts of the framework should therefore be considered. One could think of separating into three major parts:

- Initial requirements and initial study: *initial requirements description* and *object class model.*
- Essential part: *architecture structure model and essential behavior model* except the *initial requirements description* and *object class model.* This part is build during analysis of the problem domain and will formalize the customers true intent.
- Implementation part: *extended behavior model* and *implementation structure model.* This part is build during design.

How and when behavior and structure should be separated should be researched, especially with computer aided SHE support in mind.

7.5.2 Modeling the environment

SHE should have equal emphasis on modeling environment and system. The SHE model of the elevator problem mostly focussed on modeling the system. This is cause for problems.

The SHE model of the elevator problem models the environment in not enough detail. Dynamic behavior of the system relies on environment properties that are not modeled. If, for example the elevator motor runs the elevator cage down, floor sensors are activated at succeeding lower floors. The dynamic behavior description of system objects expect floors to be numbered in succeeding order, with floor 0 at the bottom and floor 39 at the top of the elevator shaft. Any other numbering of floors will make the dynamic behavior description of many objects invalid.

Another example is modeling a passenger. A passenger arrives at a floor and issues a summons request to the system. The system sends an elevator at the floor the passenger is at. The passenger enters the elevator when it arrives and has its doors opened. Here is a constraint: A passenger can only enter an elevator that is at the correct floor with its doors open. The SHE model of the elevator problem contains floor passengers and elevator passengers. These passengers don't move between system parts but are statically "connected" to a floor or an elevator cage. Without the previously described constraints in the model, the model is not suited for validation.

If we took a closer look to the model we would note several other missing constraints. The key line here is: too much emphasis has been put on modeling the system itself and too little on its environment. Modeling an environment is as important as modeling the system itself. Each embedded system is designed for a particular environment. Changes in environment influences system operation. At worst case, it could malfunction. To model an environment, these properties should be taken into account:

- Environment objects interact just as environment objects interact with system objects or system objects interact. However, terminology about interaction between environment objects lays at the problem domain, so it is most easily described with problem domain constructs/relations. In SHE, they must be converted into communicating objects, which may cause loss of information or introduce errors.

- Objects could be moving. An example is the passenger of the elevator problem. Passenger communication with the system "moves" to other system parts as time passes by. In fact, the passenger even physically moves through system and environment. At a particular instant of time, a moving objects is "visible" only to a model part. SHE is based on a static structure of communicating objects. To model "movement" and "visibility" of an object, in SHE these properties must be embedded in the dynamic behavior description of the moving object or they must be modeled as traveling data-objects.
- Objects could come into existence and die at some other time. This is common practice in software systems. Even todays hardware like runtime programmable logic, could have this property. SHE however, doesn't support creation and deletion of objects—requirement 1j—. As SHE is a method for hardware/software systems, maybe it should.

SHE should have equal emphasis on modeling environment and modeling system—requirement 2d—. The system model and environment model as a whole act as a closed system. The environment model should have enough detail, so any information that is used to model, is modeled. The resulting closed system model is executable and can be used for simulation.

7.5.3 Mixing analysis and design

Analysis models should not contain implementation details. However, for co-design, high level architecture structure must be incorporated [13] page 2. High level architecture design is motivated and documented in the *architecture structure model.* Examples of high level architecture structure are (standardized) layering and topology that may be imposed by the initial problem description and by physical geographical constraints [13] page 5.

The examples of layering and topology constraints could be essential or implementations. It depends on the definition of the system boundary [17]. Topology may be essential or a possible solution, layering may be essential or optional. The analyst is responsible for separating essence from implementation. However he may be tempted to incorporate design decisions—implementations—in the *architecture structure model.* Implementation boundaries in the *essential behavior model* only enhances the risk of misjudgment.

7.5.4 Functionality distribution

A system "must do" something. Functionality is a system's most important property. There must be a close relation between system functionality and models —requirement 2c—. SHE does not include a functional model as OMT [14] does, but uses a few concepts in its framework to directly relate to functionality instead of a separate model. They are:

- Objects and message flows
- Clusters
- Scenarios

When the *object instance model* is build, overall functionality is divided among objects. Each object provides part of the functionality with its encapsulated data, dynamic behavior and communications with collaborating objects. Objects and communications are chosen to maximize independence. In the *object instance model* the objects name and its communications give a clue what it "does"—functionality—.

Clusters group objects that are tied to each other in some way. In the elevator problem for example, clusters IndividualControl and CentralControl separate to functional different parts: functionality concerning a single elevator in the *IndividualControl* and concerning all elevators as a collection in the CentralControl. Another example is cluster *DestinationsHandler* and cluster *SummonsHandler*. Clusters group collaborating objects that perform a part of system functionality.

Collaborating instances should perform some coherent part of the system behavior, called a scenario [13] page 4. The collection of all scenarios describes system behavior completely. Therefore, scenarios divide system functionality.

SHE has means to divide functionality and shows the relation between functionality and models. However, the analyst may not be aware enough of the relation between functionality and models. SHE should stress the importance of this relation and have heuristics how to apply the proper concepts when dividing functionality.

7.5.5 Object class model

The application of OMT on the elevator problem showed that it is difficult to build a class template view, see 4.2.2. SHE has an *object class model* in its *essential behavior model*. SHE's *object class model* serves two purposes [13]:

- Initial study of the problem domain
- First list of candidate objects for the *object instance model*

Building an *object class model* certainly helps to understand the static structure of the problem domain. However, the model gives no clue about any dynamic behavior and its emphasis is on static structure of the environment—see 4.2.1—. Because the system itself is not studied in enough detail, the *object class model* is of limited value as initial study. As a first list of candidate objects it will mostly generate objects in the environment.

Considering the results of building an *object class model*, should SHE keep it for the purposes previously described? Ideas exist to use the model as a guide for building the *object instance model*. The *object class model* is considerably stable as requirements evolve [14]. Associations between class templates serve as candidate communications. When the *object class model* structure is used as a template for the *object instance model* it will inherit the stability. Classes and associations guide the analyst to add objects and communications. Because, initially the emphasis of the *object class model* is on environment, the *object class model* and *object instance model* should be build concurrently, in conjunction with each other. The *object instance model* helps to find new objects and classes, while the *object class model* serves as a skeleton for the *object instance model*.

At this time, development of SHE focused mostly on hardware. In future much more attention must be paid to software as well. Then, the *object class model* may have other important purposes as well.

7.5.6 Instance approach

Building the OMT object model of the elevator problem required many iterations—see 4.2.5—. Building the *object instance model* of the elevator problem on the other hand, was done in a single iteration. SHE's instance approach with objects and messages does minimize iterations.

The *object instance model* of the elevator problem consists of 8 environment objects and 11 system objects. These are small numbers for a problem with considerable dynamic complexity. Building the *object instance model*, choosing objects and communications, was a continuous process without many alterations in the model or points at which it is difficult to find new objects and communications. The model was build in conjunction with the *architecture structure model* without difficulty. While defining dynamic behavior in the *POOSL description*, only minor changes in the *object instance model* were needed.

7.5.7 Hierarchy issues

Human limitations prevent us from keeping all details of complex systems in our head at one time. Hierarchy enables us to describe every detail of a system, while it is also possible to perceive a system as a whole [17]. Hierarchy represents a system at various level of detail, together with a bookkeeping scheme to show the relationship between different levels of representation.

Hierarchy is a concept to divide and conquer complexity. Building a hierarchy can be done top-down or bottom-up. SHE mixes both methods. In SHE, the analyst is allowed to start at some intermediate level of abstraction and work his way up and down in the hierarchy. Whenever the analyst adds objects and messages, he puts them at the proper level of abstraction. If he is unable to decide the level of abstraction of an object, he is allowed to add hierarchy afterwards. At a later time, when the model has evolved to a more complete model, judging level of abstraction is easier. This flexible approach speeds up modeling considerably, as experienced with the SHE model of the elevator problem.

Encapsulation is required to understand the model at a particular level of abstraction—requirement 1e—. In SHE, hierarchy is build using composite objects and clusters. To understand the model at a certain level of abstraction, it is necessary to hide lower levels of abstraction. To hide lower levels of abstraction in a composite object, the composite object could be given the abstract description of its central object—page 54—. SHE should allow hiding lower levels of abstraction of a composite object by attaching an abstract description of its internals. The abstract description of the internals of a cluster will be much more complicated—page 54—. With a cluster, digging into lower levels of abstraction is necessary to understand the model. Examples are the IndividualControl and CentralControl of the elevator problem. A cluster may not be the proper concept to divide and conquer complexity. The relation between clusters and functionality is already pointed out in section 7.5.4, maybe clusters should be replaced by functionality boundaries.

7.5.8 Boundary issues

The *instance structure diagrams* contain boundaries. One of its possible types of boundaries is a concurrency boundary. A concurrency boundary restricts the interior to be implemented sequentially, [13] page 5. What exactly is a concurrency boundary's role in the *essential behavior model*?

A concurrency boundary restricts all possible solutions to those with a sequential implementation inside the concurrency boundary. However, it is often possible to satisfy a concurrent processing requirement with a sequential implementation, [17] page 12. Why should a design decision like this be put in the *essential behavior model*? Concurrency and response times are very closely related concepts in an implementation model. Maybe it is better to let all objects in the *essential behavior model* operate concurrently to prevent from making premature design decisions. Concurrency boundaries could be added to the *extended behavior model* to formalize implementation decisions in the *implementation structure model*.

Distribution boundaries model constraints imposed by physical distribution like asynchronous/synchronous operation of model parts or accessibility of objects. The analyst must be very careful to separate essence from implementation as he adds distribution boundaries. Judging essence from implementation with topology is difficult: a premature design decision is easily made. In SHE, the *instance structure diagrams* of the *essential behavior model* may also contain implementation boundaries, [13] page 5. How is essence from implementation separated if implementation boundaries are allowed in an essential model? Implementation boundaries should only be allowed in the *extended behavior model* to formalize implementation decisions in the *implementation structure model*.

7.5.9 Formal versus informal

SHE integrates informal techniques with formal ones. Building models in SHE starts with defining an informal model which will be formalized in POOSL [15]. These steps are taken during both analysis/specification and design. This approach proved to be very useful when SHE was applied to the elevator problem.

The informal *object instance model* has a formal basis. A succeeding formalization step forces to avoid being vague or having ambiguities in the model. Graphic representation of the informal model enhances communication with the customer. The *POOSL description* helps to explain communications between objects in the *object instance model*. Messages in the *object instance model* forces to think about high level dynamic behavior while structure is defined before it is formalized. This initial study of high level dynamic behavior helps to define formal dynamic behavior descriptions in the *POOSL description*. Formal and informal techniques interact "natural" in SHE.

Currently, POOSL is not able to formalize all aspects of the *object instance model* yet. For example, it only supports abstraction boundaries of all boundary types. In POOSL, process objects operate concurrently, while its internal operation is sequential. Unfortunately, this property can not be exploited to model concurrency boundaries. If, for example, a concurrency boundary is modeled with a basic process object, internal sequential operating objects must be modeled as data objects. These data objects however, don't support message flows between them, so modeling an object of the *object instance model* as a data object is impossible. In future, POOSL should support all types of boundaries.

Message passing in POOSL is based on the rendez-vous mechanism. All message flows in the *object instance model* are formalized in POOSL using the rendez-vous mechanism. One-way synchronous message passing can be modeled using the rendez-vous mechanism directly. All other types of message passing need special POOSL constructs to model. Continuous message passing and asynchronous message passing even need additional process objects to model communications. SHE must prescribe how to model all types of message flows in POOSL. They serve as formal definitions of message flow primitives.

7.5.10 Channels in an essential model

The *instance structure diagrams* of the *object instance model* contain channels. A channel exists between two objects if they have at least one message connection in the *message flow diagrams*. Channels form basic concept in POOSL through which process objects communicate. Identifying channels in the *object instance model* is necessary to be able to formalize the model in POOSL. But, are these channels essential to the problem domain?

Channels in POOSL are means for identification. Whenever a process object communicates with another process object, selecting a channel is a way to "identify" or "select" the other party. Identification in the *message flow diagrams* is done differently. Instead of a channel, the objects themselves are identified. Each object is unique in its existence. Communicating objects are identified by drawing a message primitive symbol between them. Channels in the *instance structure diagrams* translate object identification in the *message flow diagrams* to a channel structure form suitable for POOSL. They do not add information to the *essential behavior model*, nor are they essential to the problem domain.

Channels could be essential though. Model parts could be separated by distribution boundaries. Objects in one part cannot communicate to objects in the other part directly, they are "invisible" to each other. If communication between separated parts is required, they could be connected through one or more channels. Message flows will be redirected to these channels. SHE should only allow essential channels in its *essential behavior model.* Channels could be essential when messages crosses certain types of boundaries.

Chapter 8

Conclusions

At Einhoven University of Technology the Digitial Information Systems Group does research to structured specification, analysis and design methods for hardware/software systems. This research requires both theoretical and practical input. Besides studying existing method theories, they must be tried out and new theories and ideas must be evaluated after application to a real-world problem. This masters project supplies the research with practical input.

This thesis describes the application of OMT—Object-oriented Modeling Technique—and SHE—Software/Hardware Engineering—on an elevator problem. The analysis and design of the elevator control showed to be a problem with enough complexity like hierarchy and extensive dynamic behavior to be very useful in this case study.

First step in the master project was the application of OMT on the elevator problem. OMT is a method for software systems, but is used for combined software/hardware in the case study. Though promising at first sight, applying OMT to the elevator problem gave quite a few problems. Building the object model was difficult due to complex hierarchy concepts in OMT, many iterations necessary and difficulty choosing objects. Relations between OMT models is not always clear, models are hard to keep consistent and are prone to ambiguities. In OMT, not separating essence from implementation is an easily made mistake.

OMT is a method used with great succes on software problems, but showed not to be useful in combined software/hardware applications. Instead of extending OMT for use with software/hardware systems, information from the application of OMT was used to build an initial requirement list for a new method named SHE.

SHE does not focus on classes and associations like OMT, but on instances and communications between them. This instance approach feels more "natural" because it closely corresponds to the way humans think about objects. This instance approach does solve many of the problems encountered with OMT. Iterations are minimized, choosing objects is less difficult and SHE hierarchy is not complicated. The formal POOSL basis of SHE avoids ambiguities and inconsistencies. Mixing formal techniques with informal ones shows to be a successful combination: it is easy to understand and learn while keeping the rigor of a formal method.

Besides many positive properties and solving most of the problems OMT gave, SHE does have deficiencies of its own. Some SHE models are misplaced in the quadrant framework, partly mixing analysis and design in SHE weakens the ability to separate essence from implementation and the difficulties the object class model gave in OMT still remain in SHE. Also the essential models contain non-essential channels and the relation between concurrency boundaries and essence is unclear.

It is clear that SHE is still under development. SHE is promising for analysis and design of hardware/software systems, but a lot of research still needs to be done. In the near future, the quadrant framework should be remodeled and the role of the object class model together with its relation to other models should be more clearly determined. Practical experience with an analysis and design method such as in the case study of this masters project is very important input for SHE development. As SHE evolves, more case studies should be done.

Bibliography

[1] Bowers, D.S.
"Some principles for the encapsulation of the behavior of aggregate objects"
Conf. Title: IEE Colloquium on "Recent Progress in Object Technology" (Digest No.1993/238)
London: IEE, 1993

[2] Briggs, T
"A Specification Language for Object Oriented Analysis and Design"
In: "Proceedings of the 8th European Conference on Object-Oriented Programming", Bologna, Italy, july 1994

[3] Coleman, D; P. Arnold, S. Bodoff, C. Dolling, H. Gilchrist, F. Hayes and P. Jeremaes
"Object-Oriented Development: The Fusion Method"
Englewood Cliffs: Prentice Hall, 1994

[4] Eckert, G. P. Golder
"Improving object-oriented analysis" Information and Software Technology
Vol: 36 Iss: 2 p. 67-86
UK: Butterworth-Heinemann Ltd. 1994

[5] Goguen, J; J. Messeguer
"Unifying Functional, Object-Oriented and Relational Programming with Logical Semantics"
In: "Research Directions on Object-Oriented Programming"
MIT press, 1987

[6] Hayes, F and D. Coleman
"Coherent models for object-oriented analysis"
In: "Proceedings of the 1991 Conference on Object-Oriented Programming, Systems, Languages and Applications", Phoenix, Arizona, oct 1991

[7] Jacobson, I
"Object-Oriented Software Engineering: A Use Case Driven Approach"
New York: ACM Press Books, 1992

[8] Messeguer, J
"A Logical Theory of Concurrent Objects"
In: "Proceedings of the 1990 Conference on Object-Oriented Programming, Systems, Languages and Applications"

[9] Milner, R.
"A Calculus of Communicating Systems", Lecture notes in Computer Science 92.
Berlin, Germany: Springer Verlag, 1980

[10] Moreira, A.M.D. and R.G. Clark
"Combining Object-Oriented Analysis and Formal Description Techniques"
In: "Proceedings of the European Conference on Object-Oriented Programming", Bologna, Italy, july 1994

[11] Narfelt, K.H.
"SYSDAX – an object oriented design methodology based on SDL"
In: "Proceedings of SDL'87: State of the Art and Future Trends", Amsterdam, apr 1987

[12] Putten, P.H.A.
"A Specification Method for Digital Systems: Stratagies for the specification, architecture design and implementation modeling of digital hardware/software systems"
Eindhoven: Instituut Vervolgopleidingen, Technische Universiteit Eindhoven, mrt 1993
ISBN: 90-5282-238-7

[13] Putten, P.H.A. van der; J.P.M. Voeten, M.P.J. Stevens
"Object-Oriented Co-Design for Hardware/Software Systems"
To be published in proceedings of Euromicro 95.

[14] Rumbaugh, James; Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen
"Object Oriented Modeling and Design",
London: Prentice Hall, 1991

[15] Voeten, J.P.M.
"An Object-Oriented Language for the Specification, Design and Description of Hardware/Software Systems"
Eindhoven: Eindhoven University of Technology, Digital Information Systems Group, dec 1994
EUT report 95-E-290

[16] Voeten, J.P.M.
"Semantics of POOSL: An Object Oriented Specification Language for the Analysis and Design of Hardware/Software Systems"
Eindhoven University of Technology, Digital Information Systems Group, feb 1995
To be published as EUT report.

[17] Ward, P.T.; S.J. Mellor
"Structured Development for Real-Time Systems"
Volume 1,2 and 3
Englewood Cliffs.,New Jersey: Prentice Hall, 1985

[18] Yourdon, Edward
"Modern Structured Analysis"
London: Prentice Hall, 1989

Appendix A

POOSL description of the elevator problem

```
POOSL SYSTEM SPECIFICATION OF THE ELEVATOR PROBLEM

//The ElevatorControlSystem is the overall system description of the
//elevator problem. It includes the embedded system itself as well as
//objects in the environment. The environment objects descriptions
//however, do not include instance method descriptions.

ElevatorControlSystem = <
    ElevatorPassenger[diep0/diep, dlep0/dlep, epfh0/epfh, epmh0/epmh] ||
    ElevatorPassenger[diep1/diep, dlep1/dlep, epfh1/epfh, epmh1/epmh] ||
    ElevatorPassenger[diep2/diep, dlep2/dlep, epfh2/epfh, epmh2/epmh] ||
    ElevatorPassenger[diep3/diep, dlep3/dlep, epfh3/epfh, epmh3/epmh] ||
    Operator ||
    FloorPassenger ||
    IndividualControl(0)[diep0/diep, dlep0/dlep, epfh0/epfh, ecmh0/ecmh,
                         ecos0/ecos, fhfs0/fhfs] ||
    IndividualControl(1)[diep1/diep, dlep1/dlep, epfh1/epfh, ecmh1/ecmh,
                         ecos1/ecos, fhfs1/fhfs] ||
    IndividualControl(2)[diep2/diep, dlep2/dlep, epfh2/epfh, ecmh2/ecmh,
                         ecos2/ecos, fhfs2/fhfs] ||
    IndividualControl(3)[diep3/diep, dlep3/dlep, epfh3/epfh, ecmh3/ecmh,
                         ecos3/ecos, fhfs3/fhfs] ||
    CentralControl ||
    OverweightSensor[ecos0/ecos] ||
    OverweightSensor[ecos1/ecos] ||
    OverweightSensor[ecos2/ecos] ||
    OverweightSensor[ecos3/ecos] ||
    ElevatorMechanism[ecmh0/ecmh, epmh0/epmh, domh0/domh, aamh0/aamh,
                     emmh0/emmh] ||
    ElevatorMechanism[ecmh1/ecmh, epmh1/epmh, domh1/domh, aamh1/aamh,
                     emmh1/emmh] ||
    ElevatorMechanism[ecmh2/ecmh, epmh2/epmh, domh2/domh, aamh2/aamh,
                      emmh2/emmh] ||
    ElevatorMechanism[ecmh3/ecmh, epmh3/epmh, domh3/domh, aamh3/aamh,
                      emmh3/emmh] ||
    Doors[domh0/domh] ||
    Doors[domh1/domh] ||
    Doors[domh2/domh] ||
    Doors[domh3/domh] ||
    AudibleAlarm[aamh0/aamh] ||
    AudibleAlarm[aamh1/aamh] ||
    AudibleAlarm[aamh2/aamh] ||
    AudibleAlarm[aamh3/aamh] ||
    ElevatorMotor[emmh0/emmh] ||
    ElevatorMotor[emmh1/emmh] ||
    ElevatorMotor[emmh2/emmh] ||
    ElevatorMotor[emmh3/emmh] ||
    FloorSensors[fhfs0/fhfs] ||
    FloorSensors[fhfs1/fhfs] ||
    FloorSensors[fhfs2/fhfs] ||
    FloorSensors[fhfs3/fhfs]
    POD,
    DOD
    >

PROCESS OBJECTS

POD = <

//An ElevatorPassenger is a passenger inside an elevator. He is
//capable of issueing destination requests

process class                   ElevPassenger {ep}
instance variables
communication channels          diep dlep epfh epmh
message interface               diep!Destination(floor)
                                dlep?IndicateDestination(floor)
                                dlep?WithdrawDestination(floor)
                                epfh?IndicateFloor(floor)
                                epmh!Halt
                                epmh!Hold_(state)
                                epmh!NoHalt
initial method call


//A FloorPassenger is a passenger on some floor. He is capable of
//issueing summons requests
```

```
process class                   FloorPassenger {fp}
instance variables
communication channels          fpsi fpsl
message interface               fpsi!Summons(floor, direction)
                                fpsl?IndicateSummons(floor, direction)
                                fpsl?WithdrawDestination(floor, direction)
initial method call
instance methods


//The Operator sets an individual elevator operation mode to one of
//three modes: normal operation, maintenance mode and a mode in which
//an elevator is disabled. The mode requests must sequence up or down
//through these modes.

process class                   Operator {op}
instance variables
communication channels          mcop
message interface               mcop!MaintenanceMode(shaft, mode)
initial method call
instance methods


//The OverweightSensor senses if an elevator is filled to capacity

process class                   OverweightSensor {os}
instance variables
communication channels          ecos
message interface               ecos!FilledToCapacity_(state)
initial method call
instance methods


//The FloorSensors is a collection of floorsensors, one per floor, for
//a particular shaft.

cluster                         FloorSensors
communication channels          fhfs
message interface               fhfs!AtFloor(floor)
behaviour specification
  ( FloorSensor(0)  || FloorSensor(1)  || FloorSensor(2) ||
    FloorSensor(3)  || FloorSensor(4)  || FloorSensor(5) ||
    FloorSensor(6)  || FloorSensor(7)  || FloorSensor(8) ||
    FloorSensor(9)  || FloorSensor(10) || FloorSensor(11) ||
    FloorSensor(12) || FloorSensor(13) || FloorSensor(14) ||
    FloorSensor(15) || FloorSensor(16) || FloorSensor(17) ||
    FloorSensor(18) || FloorSensor(19) || FloorSensor(20) ||
    FloorSensor(21) || FloorSensor(22) || FloorSensor(23) ||
    FloorSensor(24) || FloorSensor(25) || FloorSensor(26) ||
    FloorSensor(27) || FloorSensor(28) || FloorSensor(29) ||
    FloorSensor(20) || FloorSensor(31) || FloorSensor(32) ||
    FloorSensor(33) || FloorSensor(34) || FloorSensor(35) ||
    FloorSensor(36) || FloorSensor(37) || FloorSensor(38) ||
    FloorSensor(39) )


//The IndividualControl is one of two major control parts of the
//elevator problem. This one handles all control aspects that are
//specific to an individual elevator.

cluster                         IndividualControl<SHAFT>
communiction channels           epfh dlep diep ecmh ecos fhfs ecsa
                                eces mcop
message interface               diep?Destination(floor)
                                dldp!IndicateDestination(floor)
                                dldp!WithdrawDestination(floor)
                                eces!Floor_(floor)
                                eces!InOperative
                                eces!OperationMode_(mode)
                                eces?WakeUp(floor)
                                ecmh!Move(direction)
                                ecmh?Halt
                                ecos?FilledToCapacity_(answer)
                                ecsa!GetSummons(shaft, floor, direction)
                                ecsa!SummonsIn(shaft, floor, direction)
                                ecsa?GetSummons_(shaft, answer)
                                ecsa?SummonsIn_(shaft, answer)
                                epfh!IndicateFloor(floor)
                                fhfs?AtFloor(floor)
                                mcop?MaintenanceMode(mode)
behaviour specification
  ( DestinationsHandler ||
    ElevatorMaintenanceControl<SHAFT> ||
    ElevatorControl<SHAFT> ||
    FloorHandler )
    \ { dimc, dlmc, daec, diec, ecmc, ecfh }


//The CentralControl is one of two major control parts of the
//elevator problem. This one handles all control aspects that are
//specific to all elevators as a whole.

cluster                         CentralControl
communication channels          fpsi fpsl eces ecsa
```

```
message interface               eces!WakeUp(floor)
                                eces?Floor_(floor)
                                eces?InOperative
                                eces?OperationMode_(mode)
                                ecsa!GetSummons_(shaft, answer)
                                ecsa!SummonsIn_(shaft, floor, direction)
                                ecsa?GetSummons(shaft, floor, direction)
                                ecsa?SummonsIn(shaft, floor, direction)
                                fpsi?Summons(floor, direction)
                                fpsl!IndicateSummons(floor, direction)
                                fpsl!WithdrawSummons(floor, direction)
behaviour specification
  ( SummonsHandler ||
    ElevatorSupervisor )
    \ { essi }


//The DestinationsHandler contains all processes associated with
//destinations.

cluster                         DestinationsHandler
communications channels         diep dimc diec dlmc daec dlep
message interface               daec!DestinationsIn_(answer)
                                daec!GetDestination_(answer)
                                daec?DestinationsIn(floor, direction)
                                daec?GetDestination(floor)
                                diec!WakeUp
                                diep?Destination(floor)
                                dimc?Disable
                                dimc?Enable
                                dlep!IndicateDestination(floor)
                                dlep!WithdrawDestination(floor)
                                dlmc?ClearAll
behaviour specification
  ( PassDestinationsInputControl ||
    DestinationsList ||
    DestinationsAdministrator )
    \ { didl, dadl }


//The ElevatorMaintenanceControl is a process which accepts commands
//from the operator and puts an elevator in the required mode. These
//modes are (1) normal: normal operation of an elevator, (2)
//maintenance: as in normal, but summons are ignored and (3) lock and
//clear: summons and destinations are ignored and the destinationslist
//is cleared.
```

```
process class                ElevatorMaintenanceControl<SHAFT> {mc}
instance variables           SHAFT MaintenanceMode
communication channels       mcop dimc dlmc ecmc
message interface            dimc!Disable
                             dimc!Enable
                             dlmc!ClearAll
                             ecmc!Maintenance
                             ecmc!Normal
                             mcop?MaintenanceMode(shaft, mode)
initial method call          init
instance methods
    init
        MaintenanceMode := new(MaintenanceMode) SetNormal;
        loop

    loop
    |mode|
        (
            [MaintenanceMode IsNormal]
            mcop?MaintenanceMode(shaft, mode | shaft=SHAFT and
                                               mode IsMaintenance)
                ecmc!Maintenance;
                MaintenanceMode SetMaintenance
        )
        or
        (
            [MaintenanceMode IsMaintenance]
            mcop?MaintenanceMode(shaft, mode | shaft=SHAFT and
                                               mode IsLockAndClear)
                dimc!Disable;
                dlmc!ClearAll;
                MaintenanceMode SetLockAndClear
        )
        or
        (
            [MaintenanceMode IsMaintenance]
            mcop?MaintenanceMode(shaft, mode | shaft=SHAFT and
                                               mode IsNormal)
                ecmc!Normal;
                MaintenanceMode SetNormal
        )
        or
        (
            [MaintenanceMode IsLockAndClear]
            mcop?MaintenanceMode(shaft, mode | shaft=SHAFT and
                                               mode IsMaintenance)
                dimc!Enable;
                MaintenanceMode SetMaintenance
        );
        loop


//The ElevatorControl schedules the movement of an elevator cage. It
//queries the destinations administrator and summons administrator and
//issues commands to the ElevatorMechanism. If this elevator can't
//continue to do its job, it notifies the elevators supervisor.

process class                   ElevatorControl<SHAFT> {ec}
instance variables              SHAFT OperationMode MaintenanceMode
                                Floor
communication channels          daec diec ecos ecmh ecfh eces ecsa ecmc
message interface               daec!DestinationIn(floor, direction)
                                daec!GetDestination(floor)
                                daec?DestinationIn_(answer)
                                daec?GetDestination_(answer)
                                diec?WakeUp(floor)
                                eces!Floor_(shaft, floor)
                                eces!Inoperative(shaft)
                                eces!OperationMode_(shaft, mode)
                                eces?WakeUp(shaft, floor)
                                ecfh?AtFloor(floor)
                                ecmc?Maintenance
                                ecmc?Normal
                                ecmh!Move(direction)
                                ecmh?Halt
                                ecos?FilledToCapacity_(state)
                                ecsa!GetSummons(shaft, floor, direction)
                                ecsa!SummonsIn(shaft, floor, direction)
                                ecsa?GetSummons_(shaft, answer)
                                ecsa?SummonsIn_(shaft, answer)
initial method call             init
instance methods
    init
        //Move elevator to floor 0
        OperationMode := new(OperationMode) SetIdle;
        MaintenanceMode := new(MaintenanceMode) SetNormal;
        Floor := 0;
        loop

    loop
        |floor, sdaf, sdi|
        (
            ecfh?AtFloor(Floor)
                CheckSumDestAtFloor()(sdaf);
            if sdaf then
                ecmh!Move(new(Direction) SetStop)
            fi;
            CheckSumDestIn()(sdi);
            if sdi not then
                if OperationMode IsUp then
                    OperationMode SetDown
                else
                    OperationMode SetUp
                fi;
                CheckSumDestAtFloor()(sdaf);
                if sdaf then
                    ecmh!Move(new(Direction) SetStop)
                fi;
                CheckSumDestIn()(sdi);
                if sdi not then
                    OperationMode SetIdle
                fi
            fi;
            (
                [OperationMode IsUp]
                    ecmh!Move(new(Direction) SetUp)
            )
            or
            (
                [OperationMode IsIdle]
                    ecmh!Move(new(Direction) SetStop)
            )
            or
            (
                [OperationMode IsDown]
                    ecmh!Move(new(Direction) SetDown)
            )
    )
    or
    (
        ecmh?Halt
            eces!Inoperative(SHAFT)
    )
    or
    (
        diec?WakeUp(floor)
            if OperationMode IsIdle then
                if floor<Floor then
                    OperationMode SetDown;
                    ecmh!Move(new(Direction) SetDown)
                else
```

```
                    OperationMode SetUp;
                    ecmh!Move(new(Direction) SetUp)
                fi
            fi
        )
        or
        (
            eces?WakeUp(shaft, floor | shaft=SHAFT)
                if floor<Floor then
                    OperationMode SetDown;
                    ecmh!Move(new(Direction) SetDown)
                else
                    OperationMode SetUp;
                    ecmh!Move(new(Direction) SetUp)
                fi
        )
        or
        (
            ecmc?Normal
                MaintenanceMode SetNormal
        )
        or
        (
            ecmc?Maintenance
                MaintenanceMode SetMaintenance;
                eces!InOperative
        )
        or
        (
            eces!Floor_(SHAFT, Floor)
        )
        or
        (
            eces!OperationMode_(SHAFT, OperationMode)
        );
        loop

    CheckSumDestAtFloor ()(answer)
        |destansw fill2cap shaft sumansw|
        ecos?FilledToCapacity_(fill2cap);
        if fill2cap not and MaintenanceMode IsNormal then
            ecsa!GetSummons(SHAFT, Floor, OperationMode);
            ecsa?GetSummons_(shaft, sumansw | shaft=SHAFT);
        else
            sumansw := FALSE;
        fi;
        daec!GetDestination(Floor);
        daec?GetDestination_(destansw);
        answer := sumansw or destansw

    CheckSumDestIn ()(answer)
        |destansw fill2cap shaft sumansw|
        ecos?FilledToCapacity_(fill2cap);
        if fill2cap not and MaintenanceMode IsNormal then
            ecsa!SummonsIn(SHAFT, Floor, OperationMode);
            ecsa?SummonsIn_(shaft, sumansw | shaft=SHAFT);
        else
            sumansw := FALSE;
        fi;
        daec!DestinationIn(Floor, OperationMode);
        daec?DestinationIn_(destansw);
        answer := sumansw or destansw


//The FloorHandler updates the display inside the elevator cage and
//notifies the elevator control that the elevator cage has reached a
//floor within 8 inches.

process class                   FloorHandler {fh}
instance variables
communication channels          ecfh fhfs epfh
message interface               ecfh!AtFloor(floor)
                                epfh!IndicateFloor(floor)
                                fhfs?AtFloor(floor)
initial method call             loop
instance methods
    loop
        |floor|
        fhfs?AtFloor(floor);
        epfh!IndicateFloor(floor);
        ecfh!AtFloor(floor);
        loop


//The SummonsHandler contains all processes associated with summons.

cluster                         SummonsHandler
communications channels         fpsi fpsl essi ecsa
message interface               ecsa!GetSummons_(shaft, answer)
                                ecsa!SummonsIn_(shaft, answer)
                                ecsa?GetSummons(shaft, floor, direction)
                                ecsa?SummonsIn(shaft, floor, direction)
                                essi!NewSummons(floor, direction)
                                fpsi?Summons(floor, direction)
                                fpsl!IndicateSummons(floor, direction)
                                fpsl!WithdrawSummons(floor, direction)
behaviour specification
  ( PassSummonsInputControl ||
    SummonsList ||
    SummonsAdministrator )
    \ { sisl, sasl }


//The ElevatorSupervisor supervises all elevators. If an new summons
//arrives it sends the closest idle elevator. If an elevator becomes
//inoperative the supervisor starts an idle elevator to move in the
//same direction.

process class                   ElevatorsSupervisor {es}
instance variables
communication channels          eces essi
message interface               essi?NewSummons(floor, direction)
                                eces?InOperative(shaft)
                                eces?Floor_(shaft, floor)
                                eces?OperationMode_(shaft, mode)
                                eces!WakeUp(shaft, floor)
initial method call             loop
instance methods
    loop
        |b ElevNr ElevDist direction floor i mode shaft
        thisfloor thismode|
        (
            essi?NewSummons(floor, direction)
                //Send closest idle elevator
                ElevNr := -1;
                ElevDist := 40;
                i := 0;
                do i<4 then
                    eces?OperationMode_(shaft, mode | shaft=i);
                    if mode IsIdle then
                        eces?Floor_(shaft, thisfloor | shaft=i);
                        if thisfloor-floor abs < ElevDist then
                            ElevNr := i;
                            ElevDist := thisfloor-floor abs
                        fi
                    fi
                    i := i + 1
                od
                if ElevNr > -1 then
                    eces!WakeUp(ElevNr, floor)
                fi
        )
        or
        (
            eces?Inoperative(shaft)
                //Start the first idle elevator in the direction the
                //inoperative elevator was going.
                eces?OperationMode_(i, mode | i=shaft);
                i := 0;
                b := TRUE;
                do (i<4 and b) then
                    eces?OperationMode_(shaft, thismode | shaft=i);
                    if thismode IsIdle then
                        (
                            [mode IsUp]
                                eces!WakeUp(i, 39)
                        )
                        or
                        (
                            [mode IsDown]
                                eces!WakeUp(i, 0)
                        );
                        b := FALSE
                    fi
                od
        );
                    //The elevators supervisor shouldn't wake up
                    //elevators that are halted or in maintenance
                    //mode. This isn't correctly implemented yet.
        loop


//The FloorSensor senses if an elevator cage reaches a floor within 8
//inches.

process class                   FloorSensor(FLOOR) {fs}
instance variables              FLOOR
communication channels          fhfs
message interface               fhfs!AtFloor(floor)
initial method call
instance methods


//The ElevatorMechanism controls the mechanical part of an
//elevator cage. It controls the motor, doors and audible alarm. It
//accepts commands from the elevator controller. The
//ElevatorMechanism is in one of four states: (1) moving: the
//elevator cage is moving, (2) stopped: the elevator cage has stopped
```

```
//at a floor, (3) mustgo: the elevator must go, but waits until all
//passengers has entered and (4) halted: the halt switch is operated.

process class                   ElevatorMechanism {mh}
instance variables              MoveState LastPassDetTime MoveTo
communication channels          ecmh epmh domh aamh emmh
message interface               aamh!Alarm(state)
                                domh!Close
                                domh!Open
                                domh?PassengerDetected
                                ecmh!Halt
                                ecmh?Move(direction)
                                emmh!Move(command)
                                epmh?Halt
                                epmh?Hold_(state)
                                epmh?NoHalt
initial method call             init
instance methods
    init
        //Elevator initially stopped
        LastPassDetTime := 0;
        MoveState := new(MoveState) SetStopped;
        MoveTo := new(Direction) SetStop;
        loop

    loop
        |direction move musthold|

        epmh?Hold_(musthold);
        (
            [MoveState IsMoving]
            ecmh?Move(direction | direction IsStop)
                emmh!Move(new(Direction) SetStop);
                domh!Open;
                MoveTo SetStop;
                MoveState SetStopped
        )
        or
        (
            [MoveState IsMoving]
            ecmh?Move(direction | direction IsUp or direction IsDown)
                emmh!Move(direction)
        )
        or
        (
            [MoveState IsMoving]
            epmh?Halt
            //Stop elevator at next floor
            emmh!Move(new(Direction) SetStop);
            domh!Open;
            aamh!Alarm(TRUE);
            ecmh!Halt
            MoveState SetHalted
    )
    or
    (
        [MoveState IsStopped]
        ecmh?Move(direction | direction IsStop)
    )
    or
    (
        [MoveState IsStopped or MoveState IsMustGo]
        ecmh?Move(direction | direction IsUp or direction IsDown)
            MoveTo := direction;
            MoveState SetMustGo
    )
    or
    (
        [MoveState IsStopped]
        epmh?Halt
            aamh!Alarm(TRUE);
            ecmh!Halt;
            MoveState SetHalted
    )
    or
    (
        [MoveState IsMustGo]
        ecmh?MoveState(direction | direction IsStop)
            MoveTo SetStop;
            MoveState SetStopped
    )
    or
    (
        [MoveState IsMustGo]
        epmh?Halt
            aamh!Alarm(TRUE);
            ecmh!Halt;
            MoveState SetHalted
    )
    or
    (
        [MoveState IsMustGo and (LastPassDetTime Time)>10 and
         musthold not]
            domh!Close;
```

```
                emmh!Move(MoveTo);
                MoveState SetMoving
        )
        or
        (
            [MoveState IsHalted]
            epmh?NoHalt
                aamh!Alarm(FALSE);
                if MoveTo IsStop then
                    MoveState SetStopped
                else
                    MoveState SetMustGo
                fi
        )
        or
        (
            [MoveState IsHalted]
            ecmh?Move(direction)
                MoveTo := direction
        )
        or
        (
            domh?PassengerDetected
                LastPassDetTime ResetTimer
        );
        loop


//The AudibleAlarm is an audible alarm

process class                   AudibleAlarm {aa}
instance variables
communication channels          aamh
message interface               aamh?Alarm(state)
initial method call
instance methods


//The ElevatorMotor is an interface process to the real-world elevator
//motor.

process class                   ElevatorMotor {em}
instance variables
communication channels          emmh
message interface               emmh?Move(command)
initial method call
instance methods
```

```
//The Doors are the doors of an elevator cage, together with detection
//of passengers entering and leaving the elevator cage.

process class                  Doors {do}
instance variables
communication channels         domh
message interface              domh!PassengerDetected
                               domh?Close
                               domh?Open
initial method call
instance methods


//The PassDestinationsInputControl is an interface process that
//communicates with an elevator passenger. Each time a new destination
//request arrives it notifies the elevator control so.

process class                  PassDestinationsInputControl {di}
instance variables
communication channels         diep dimc diec didl
message interface              didl!AddDestination(floor)
                               didl?AddDestination_
                               diem!WakeUp(floor)
                               diep?Destination(floor)
                               dimc?Disable
                               dimc?Enable
initial method call            loop
instance methods
    loop
    |floor|
        (
            diep?Destination(floor)
                didl!AddDestination(floor);
                didl?AddDestination_;
                diem!WakeUp(floor)
        )
        or
        (
            dimc?Disable
                dimc?Enable
        );
        loop


//The DestinationsList contains all the destination requests. They can
```

```
//be added, queried and removed. The destination display is updated
//according the contents of the list.

process class                  DestinationsList {dl}
instance variables             DestL
communication channels         didl dadl dlep dlmc
message interface              dadl!IsDestination_(answer)
                               dadl?IsDestination(floor)
                               dadl?RemoveDestination(floor)
                               didl!AddDestination_
                               didl?AddDestinat ion(floor)
                               dlep!IndicateDestination(floor)
                               dlep!WithdrawDestination(floor)
                               dlmc?ClearAll
initial method call            init
instance methods
    init
        DestL := new(BooleanArray) Size(40);
        ClearList;
        loop

    loop
    |floor|
        (
            didl?AddDestination(floor)
                DestL SetElement(floor, TRUE);
                dlep!IndicateDestination(floor);
                didl!AddDestination_
        )
        or
        (
            dadl?IsDestination(floor)
                dadl!IsDestination_(DestL GetElement(floor))
        )
        or
        (
            dadl?RemoveDestination(floor)
                DestL SetElement(floor, FALSE);
                dlep!WitdrawDestination(floor)
        )
        or
        (
            dlmc?ClearAll
                ClearList()()
        );
        loop
    ClearList ()()
        |i|
        i := 0;
        do i<40 then
            DestL SetElement(i, FALSE);
            dlep!WithdrawDestination(i);
            i := i + 1
        od


//The DestinationsAdministrator answers questions from the elevator
//control about destinations.

process class                 DestinationsAdministrator {da}
instance variables
communication channels        dadl daec
message interface             dadl!IsDestination(floor)
                              dadl!RemoveDestination(floor)
                              dadl?IsDestination_(answer)
                              daec!DestinationsIn_(answer)
                              daec!GetDestination_(answer)
                              daec?DestinationsIn(floor, direction)
                              daec?GetDestination(floor)
initial method call           loop
instance methods
    loop
        |answer b direction floor|
        (
            daec?GetDestination(floor)
                dadl!IsDestination(floor);
                dadl?IsDestination_(answer);
                if answer=TRUE then
                    dadl!RemoveDestination(floor);
                    daec!GetDestination_(TRUE)
                else
                    daec!GetDestination_(FALSE)
        )
        or
        (
            daec?DestinationsIn(floor, direction)
                (
                    [direction IsUp]
                        b := FALSE;
                        floor := floor + 1;
                        do floor<40 then
                            dadl!IsDestination(floor);
                            dadl?IsDestination_(answer);
                        b := b or answer;
                        floor := floor + 1
                    od
            )
            or
            (
                [direction IsDown]
                    b <= FALSE;
                    floor := floor - 1;
                    do floor>=0 then
                        dadl!IsDestination(floor);
                        dadl?IsDestination_(answer);
                        b := b or answer;
                        floor := floor - 1
                    od
            );
            daec!DestinationIn_(b)
        );
        loop


//The PassSummonsInputControl is an interface process that
//communicates with a floor passenger. Each time a new summons request
//arrives it notifies the elevators supervisor so.

process class                   PassSummonsInputControl {si}
instance variables
communication channels          fpsi sisl essi
message interface               essi!NewSummons(floor, direction)
                                fpsi?Summons(floor, direction)
                                sisl!AddSummons(floor, direction)
                                sisl?AddSummons_
initial method call             loop
instance methods
    loop
    |direction floor|
        fpsi?Summons(floor, direction)
        sisl!AddSummons(floor, direction);
        sisl?AddSummons_;
        essi!NewSummons(floor, direction);
        loop


//The SummonsList contains all the Summons requests. They can be
//added, queried and removed. The summons display is updated according
//the contents of the list.
```

```
process class                   SummonsList {sl}
instance variables              SummonsUpL SummonsDownL
communication channels          sisl fpsl sasl
message interface               fpsl!IndicateSummons(floor, direction)
                                fpsl!WithdrawSummons(floor, direction)
                                sasl!IsSummons_(answer)
                                sasl?IsSummons(floor, direction)
                                sasl?RemoveSummons(floor, direction)
                                sisl!AddSummons_
                                sisl?AddSummons(floor, direction)
initial method call             init
instance methods
    init
        SummonsUpL := new(BooleanArray) Size(40);
        SummonsDownL := new(BooleanArray) Size(40);
        ClearLists()();
        loop

    loop
    |direction floor|
        (
            sisl?AddSummons(floor, direction)
                (
                    [direction IsUp]
                        SummonsUpL SetElement(floor, TRUE)
                )
                or
                (
                    [direction IsDown]
                        SummonsDownL SetElement(floor, TRUE)
                )
                fpsl!IndicateSummons(floor, direction);
                sisl!AddSummons_
        )
        or
        (
            sasl?IsSummons(floor, direction)
                (
                    [direction IsUp]
                        sasl!IsSummons_(SummonsUpL GetElement(floor))
                )
                or
                (
                    [direction IsDown]
                        sasl!IsSummons_(SummonsDownL GetElement(floor))
                )
        )
        or
        (
            sasl?RemoveSummons(floor, direction)
                (
                    [direction IsUp]
                        SummonsUpL SetElement(floor, FALSE)
                )
                or
                (
                    [direction IsDown]
                        SummonsDownL SetElement(floor, FALSE)
                )
                fpsl!WithdrawSummons(floor, direction)
        );
        loop

    ClearLists()()
        |direction floor i|
        i := 0;
        do i<40 then
            SummonsUpL SetElement(FALSE);
            fpsl!WithdrawSummons(floor, direction SetUp);
            SummonsDownL SetElement(FALSE);
            fpsl!WithdrawSummons(floor, direction SetDown);
            i := i+1
        od


//The SummonsAdministrator answers questions from the elevator
//control about summons.

process class                   SummonsAdministrator {sa}
instance variables
communication channels          sasl ecsa
message interface               ecsa!GetSummons_(shaft, answer)
                                ecsa!SummonsIn_(shaft, answer)
                                ecsa?GetSummons(shaft, floor, direction)
                                ecsa?SummonsIn(shaft, floor, direction)
                                sasl!IsSummons(floor, direction)
                                sasl!RemoveSummons(floor, direction)
                                sasl?IsSummons_(answer)
initial method call             loop
instance methods
    loop
        |answer b direction floor shaft|
        (
            ecsa?GetSummons(shaft, floor, direction)
            sasl!IsSummons(floor, direction);
            sasl?IsSummons_(answer);
            if answer=TRUE then
                sasl!RemoveSummons(floor, direction);
                ecsa!GetSummons_(shaft, TRUE)
            else
                ecsa!GetSummons_(shaft, FALSE)
            fi
    )
    or
    (
        ecsa?SummonsIn(shaft, floor, direction)
            (
                [direction IsUp]
                    b := FALSE;
                    floor := floor + 1;
                    do floor<40 then
                        sasl!IsDestination(floor, direction);
                        sasl?IsDestination_(answer);
                        b := b or answer;
                        floor := floor + 1
                    od
            )
            or
            (
                [direction IsDown]
                    b := FALSE;
                    floor := floor - 1;
                    do floor>=0 then
                        sasl!IsDestination(floor, direction);
                        sasl?IsDestination_(answer);
                        b := b or answer;
                        floor := floor - 1
                    od
            );
            ecsa!SummonsIn_(shaft, b)
    );
    loop

>

-------------------------------------------------------------------------
DATA OBJECTS

DOD = <
```

```
//BooleanArrayElement is an element of an boolean array implemented as
//a single linked list.

data class                      BooleanArrayElement
instance variables              Bool NextElement
instance methods
    GetBoolean
        Bool

    SetBoolean(bool)
        Bool := bool;
        self

    GetNext
        NextElement

    SetNext(nextelement)
        NextElement := nextelement;
        self


//The BooleanArray is an array of booleans.

data class                      BooleanArray
instance variables              FirstElement
instance methods
    Size(number)
        |i element|
        FirstElement := new(BooleanArrayElement);
        element := FirstElement;
        i := 1;
        do i<number then
            element SetNext(new(BooleanArrayElement));
            element := element GetNext(element);
            i := i+1
        od;
        self

    GetElement(elementnumber)
        |i element|
        element := FirstElement;
        i := 0;
        do i<elementnumber then
            element := element GetNext(element);
            i := i+1
        od;
        element GetBoolean
    SetElement(elementnumber, bool)
        |i element|
        element := FirstElement;
        i := 0;
        do i<elementnumber then
            element := element GetNext(element);
            i := i+1;
        od
        element SetBoolean(bool);
        self


//The Direction is a data object to hold the direction.

data class                      Direction
instance variables names        state
instance methods
    SetDown
        state := 0;
        self

    IsDown
        state = 0

    SetStop
        state := 1;
        self

    IsStop
        state = 1

    SetUp
        state := 2;
        self

    IsUp
        state = 2


//The MaintenanceMode is a data object to hold the maintence mode of
//an elevator.

data class                      MaintenanceMode
instance variables              mode
instance methods
    SetNormal
```

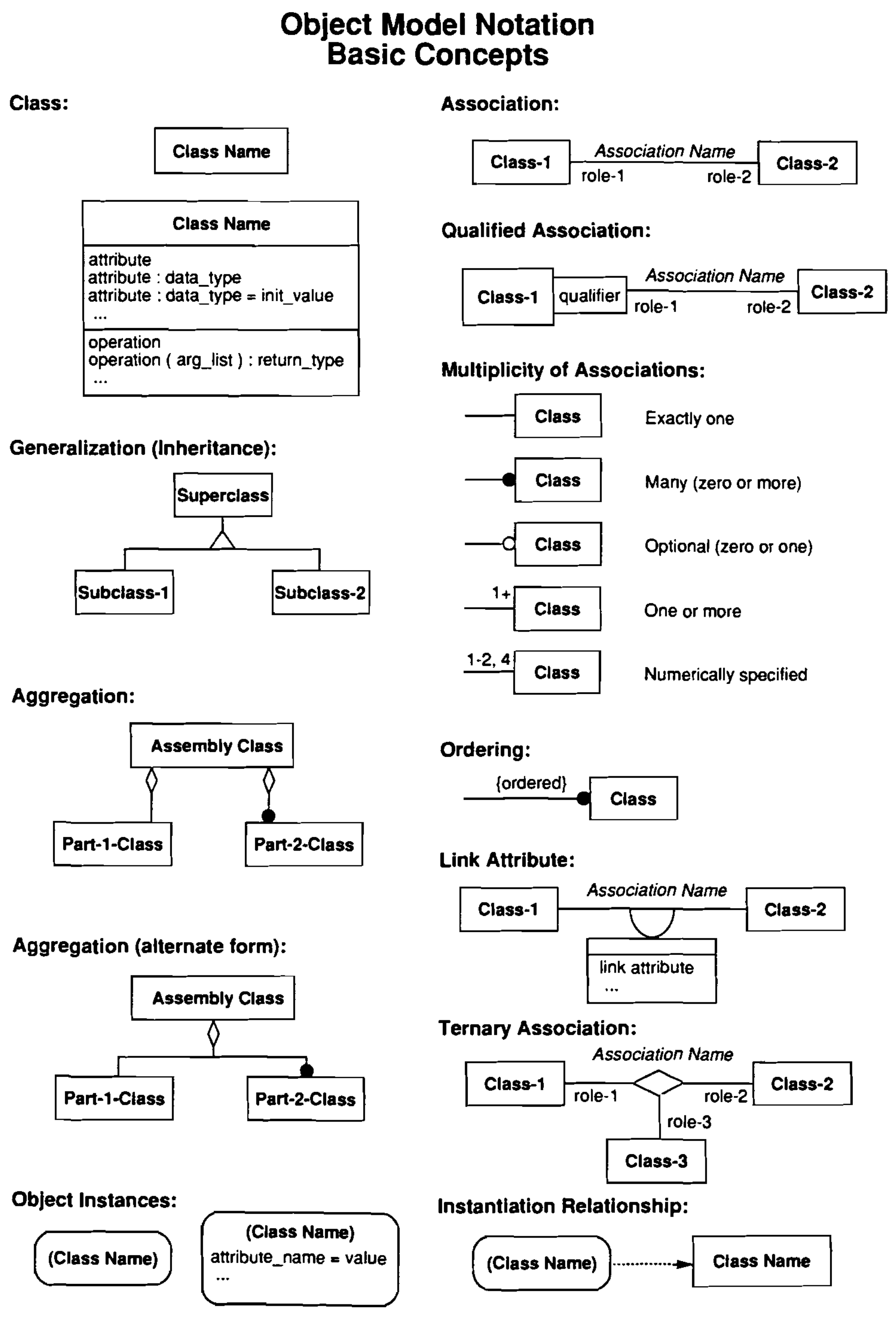
```
        mode := 0;
        self

    IsNormal
        mode = 0

    SetMaintenance
        mode := 1;
        self

    IsMaintenance
        mode = 1

    SetLockAndClear
        mode := 2;
        self

    IsLockAndClear
        mode = 2


//The OperationMode is a data object to hold the operation mode of an
//elevator.

data class                      OperationMode
instance variables              mode
instance methods
    SetDown
        mode := 0;
        self

    IsDown
        mode = 0

    SetIdle
        mode := 1;
        self

    IsIdle
        mode = 1

    SetUp
        mode := 2;
        self

    IsUp
        mode = 2
```

```
//The MoveState is a data object to hold the move state of the
//ElevatorMechanism.

data class                      MoveState
instance variables              state
instance methods
    SetStopped
        state := 0;
        self

    IsStopped
        state = 0

    SetMustGo
        state := 1;
        self

    IsMustGo
        state = 1

    SetMoving
        state := 2;
        self

    IsMoving
        state = 2

    SetHalted
        state := 3;
        self

    IsHalted
        state = 3


//The Timer is data object acting as a general timer. It increments
//its count every second.

data class                      Timer
instance variable               Count
instance methods
    Time
        Count

    ResetTimer
```

```
        Count := 0;
        Count
//Count is automatically incremented every second.

>
```

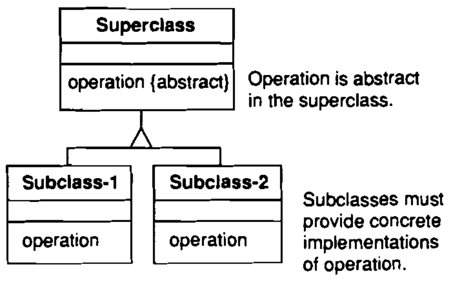
Appendix B

OMT notation

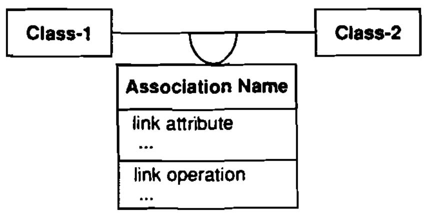


Object Model Notation
Advanced Concepts

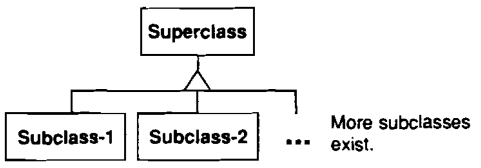
Abstract Operation:

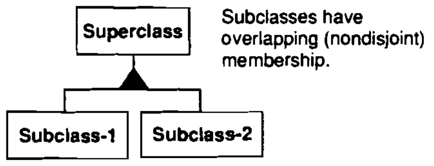

Superclass
operation {abstract}
Operation is abstract in the superclass.
Subclass-1
operation
Subclass-2
operation
Subclasses must provide concrete implementations of operation.

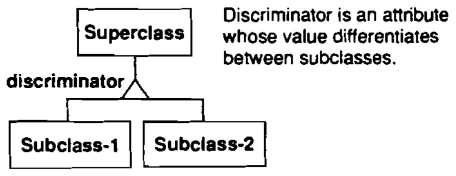
Association as Class:



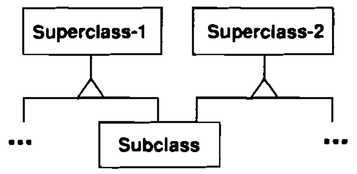
Generalization Properties:



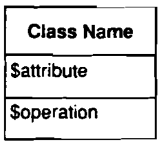



Superclass
Discriminator is an attribute whose value differentiates between subclasses.
discriminator
Subclass-1
Subclass-2

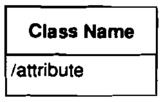
Multiple Inheritance:



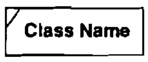
Class Attributes and Class Operations:



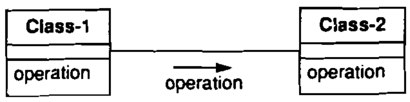
Derived Attribute:



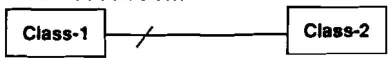
Derived Class:



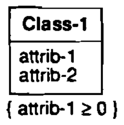
Propagation of Operations:



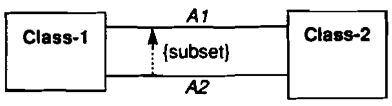
Derived Association:



Constraints on Objects:

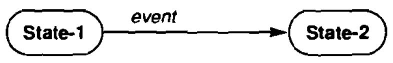


Constraint between Associations:

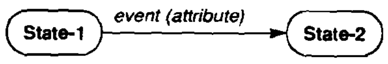


Dynamic Model Notation

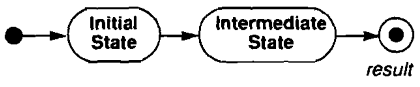
Event causes Transition between States:



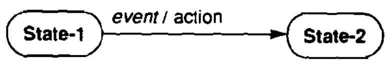
Event with Attribute:



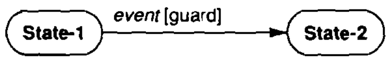
Initial and Final States:



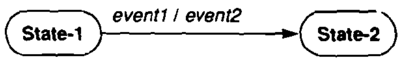
Action on a Transition:



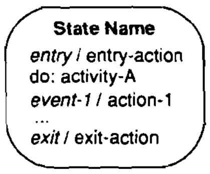
Guarded Transition:



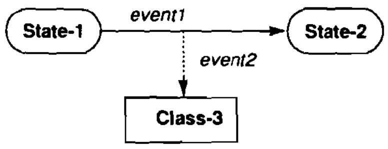
Output Event on a Transition:



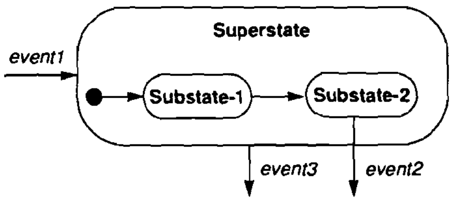
Actions and Activity while in a State:



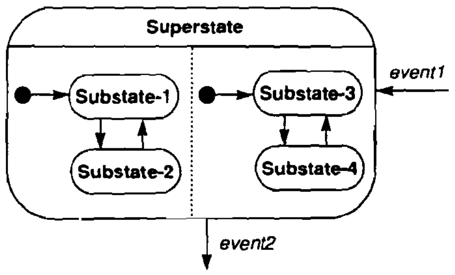
Sending an event to another object:



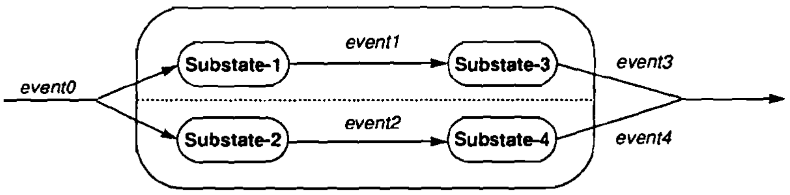
State Generalization (Nesting):



Concurrent Subdiagrams:

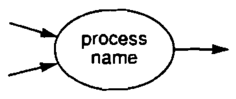


Splitting of control: **Synchronization of control:**

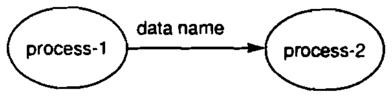


Functional Model Notation

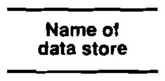
Process:



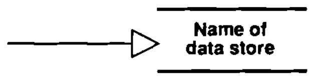
Data Flow between Processes:



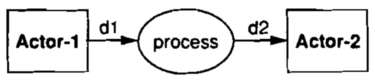
Data Store or File Object:



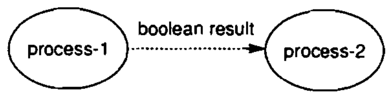
Data Flow that Results in a Data Store:



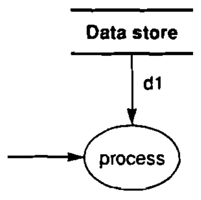
Actor Objects (as Source or Sink of Data):



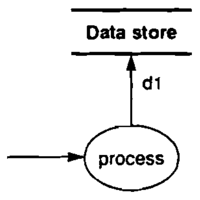
Control Flow:



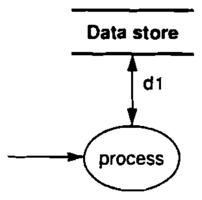
Access of Data Store Value:



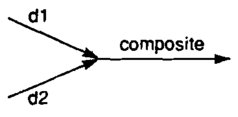
Update of Data Store Value:



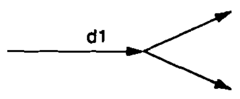
Access and Update of Data Store Value:



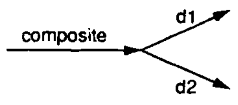
Composition of Data Value:



Duplication of Data Value:



Decomposition of Data Value:


composite
d1
d2