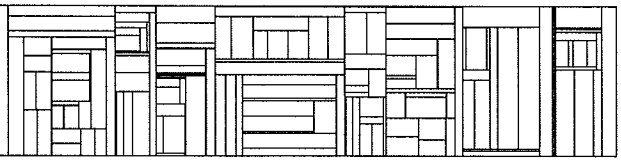
ISSN 0105-8517

Object-Oriented Development – Integrating Analysis, Design and Implementation

Elmer Sandvad

DAIMI PB – 302
April 1990

COMPUTER SCIENCE DEPARTMENT
AARHUS UNIVERSITY
Ny Munkegade, Building 540
DK-8000 Aarhus C, Denmark



Object-Oriented Development - Integrating Analysis, Design and Implementation

Elmer Sandvad
Computer Science Department, Aarhus University,
Ny Munkegade 116, DK-8000 Aarhus C, Denmark
Phone: +45 86 12 71 88 — Email: essandvad@daimi.dk

Abstract

Object-orientation is discussed in relation to the traditional division of system development activities: analysis, design and implementation. Object-orientation integrates analysis, design and implementation. This statement is illustrated by introducing an object-oriented graphical notation for analysis as well as design and by showing how this notation can be mapped into the object-oriented programming language BETA. This mapping can be automated. It is shown how the programming environment: the Mjølner BETA System can support integration of the object-oriented notation and BETA, and thereby providing an object-oriented CASE tool, that, at least partially, closes the CASE gap between design and implementation.

Keywords: Object-Orientation, Analysis, Design, Code Generation, Reverse Engineering, Traceability, Programming Environments, CASE

Contents

Chapter 1

Introduction

The report has mainly two purposes. The one is to examine the relationship between the object-oriented programming perspective and the system development activities: analysis, design and implementation. The second is to examine the possibilities for automated support in analysis, design and implementation. Not only support for creation and modification of documents, but also for the transitions between analysis, design and implementation. The conclusions are that the object-oriented programming perspective can integrate analysis, design and implementation and if all three models are object-oriented, then there can be a high degree of automated support.

Object-Oriented Programming

Object-oriented programming can reduce the total software life-cycle cost, because object-oriented programs are easier to write, understand, maintain and reuse [Booch 86] [Meyer 87]. When defining object-oriented programming the focus is often on programming language concepts. From a programming technical point of view important programming language concepts are often considered to be data abstraction, information hiding, inheritance and may be concurrency. From this point of view an object-oriented program execution is considered as a collection of interacting objects, that are instances of classes (abstract datatypes), and objects may have their own execution thread. The classes may be organized in inheritance hierarchies. This is in contrast to procedural programming languages, where a program execution can be characterized as a (partially ordered) sequence of procedure calls manipulating data structures.

There is however more to object-oriented programming than programming language concepts. From a modelling point of view, the power of object-oriented programming can be ascribed to the ability to model reality in a natural way [Booch 86] [Madsen 88b]. A program execution is regarded as a physical model simulating the behavior of either a real or imaginary part of the world. Phenomena and concepts in the real world are directly reflected in the physical model.

But how does the object-oriented perspective fit into the context of system development? Is it possible merely to substitute the currently used programming language with an object-oriented programming language, without influencing the methods used for analysis and design? Of course you can program "conventionally" in most object-oriented programming languages. But several software engineers [Booch 86] [Meyer 87] have pointed out, that to create optimal object-oriented implementations, a system-development perspective must be used, that is based on the same abstractions as used in the programming.

Structured Analysis and Structured Design

What about the well-known methods Structured Analysis and Structured Design (SA/SD) [DeMarco 78] [Yourdon]? What are the abstractions of SA/SD?

The Structured Analysis and Structured Design methods have been widely used for many years, and the popularity has increased after the provision of CASE tools, that support creation and modification of SA/SD diagrams. One of the reasons for the success of SA/SD is, that the methods are based on a set of graphical notations, that are easier to understand than textual alternatives. Another reason is the lack of modelling capability of the programming languages used. In industry system developers are often forced to use primitive programming languages like Fortran and COBOL. This is either because these languages are the only available languages on the computer used, or because new systems must build upon or be compatible with a huge amount of existing code.

The dataflow diagram of SA, has turned out to be suitable to model at least some aspects of reality, namely the flow of information through a set of processes. A process is considered as a functional transformation from a set of inputs to a set of outputs. The structure chart of SD is used as

an intermediate description between the data flow diagrams and the programming language. A structure chart is a functional decomposition of the data flow diagrams into modules that constitute the major functions in the overall process. These modules are in turn implemented as subprograms or procedures that manipulate common data structures. SA/SD can be characterized as a functional development method [Jackson 83] [Booch 86].

Apparently the abstractions of SA/SD are incompatible with the abstractions of object-oriented programming. Functional development is based on the concept of a function whereas object-oriented development is based on the concept of an object. But integration of object-orientation with Structured Analysis and Structured Design has been the subject of a substantial research effort the last few years [Booch 86] [Seidewitz 87] [Bailin 89] [Wasserman 89] [Ward 89]. It is in fact possible to use the well-known SA/SD techniques in an object-oriented manner as demonstrated by Bailin, Ward and Wasserman. But there is not much left of the original underlying concepts of SA/SD: data flow and functional development.

One of the "fathers" of SA/SD: Edward Yourdon has together with Peter Coad gone a step further. Instead of introducing object-orientation into SA/SD, they have given up SA/SD and proposed a method for object-oriented analysis [Coad 89].

The contribution of this report is to propose an object-oriented graphical notation for analysis as well as design. The notation is called Object-Oriented Analysis and Design Diagrams (OOAADD but the shorter term OAD is used) [1].

1.1 Object-Oriented Methods

[Booch 86] [Bailin 89] [Ward 89] give some techniques for finding the objects. In [Coad 89] a comprehensive method is described for identifying

[1] A note on terminology: analysis, design and implementation are well-established terms in the SA/SD and CASE communities. In [Andersen 86] the product-oriented functions in the system development process are called analysis, design and realization, where analysis is addressing the existing system (whether automated or not) whereas design and realization are addressing the future edp-based system. In that terminology traditional analysis and design are covered by the design function. In this report the terms analysis and design are used in their traditional meaning.

classes [2] with attributes and services, assembly and inheritance structures, object connections and message connections.

The OAD notation is a description tool, not a technique (not to mention a methodology), but the object-oriented perspective as described in chapter 2 is the underlying philosophy.

Analysis, Design and Implementation

[Coad 89] is focusing on analysis. [Booch 86], [Seidewitz 87] and [Wasserman 89] are focusing on design and implementation. [Ward 89] is focusing on analysis and design. [Bailin 89] is focusing on analysis, design and implementation.

In order to ease the transformation from the analysis model to the design model and in turn to the implementation model, object-orientation must be introduced in all three models [Coad 89].

The purpose of the OAD notation is to demonstrate the integration of analysis, design and implementation. Using this notation, the same model is used at all (three) levels: the model starts informally in analysis, is gradually extended and precisized in design and then automatically transformed to program templates.

Objects and/or Concepts

One important characteristic of the OAD notation is the emphasis on expressing objects, object hierarchies, object interaction and sequencing between objects (object diagrams), as well as concepts and concept hierarchies (pattern diagrams), and finally the connection between objects and concepts. Other notations tend primarily to focus on either the objects [Booch 86] [Seidewitz 87] [Bailin 89] [Ward 89] or the concepts [Wasserman 89] [Coad 89]. [Pun 89] is an example of a design method with graphical notations that can be used to express object relationships (object interaction diagrams) as well as concept relationships (class structure charts)

[2] A confusing detail about this method is that classes are called objects (objects are then called instances).

Generalization/Specialization

[Booch 86] [Seidewitz 87] [Bailin 89] have developed their methods with a particular programming language in mind, namely Ada. But is Ada object-oriented? No, but as Booch remarks: "languages such as Ada may be applied in an object-oriented fashion". Ada program executions can indeed be considered as a collection of interacting objects, i.e. instances of abstract datatypes (generic packages or types provided by packages), and concurrency is provided by means of tasks. What Ada lacks in order to be a full-fledged object-oriented language is the ability to express classification hierarchies (generalization/specialization). Generic packages and derived types can however be used to express a limited form of specialization, but only in two levels.

The fact that Ada is the "target" language, has influenced these methods and none of these approaches consider the generalization/specialization aspect of SA/SD diagrams.

Wasserman et al. give an example of the simple kind of specialization that can be provided by generic packages, i.e. parameterization by constants and types.

Ward shows how simple specialization, i.e. inheritance of attributes from a superclass and addition of attributes in the subclass, can be integrated with the data flow diagram of Real-time Structured Analysis and Structured Design [3] .

Coad & Yourdon support simple and qualified specialization to some degree, and the possibility to override attributes of a superclass. The latter facility does not conform with the view on generalization/specialization in this report.

In [Belsnes 87] a proposal is given for making SDL [4] object-oriented. The proposed language (OSDL) introduces not only simple specialization but also qualified specialization (virtual procedures and virtual transitions).

In the OAD notation simple as well as qualified specialization can be expressed in object diagrams as well as pattern diagrams.

[3] Real-time Structured Analysis and Structured Design (RT SA/SD) [Ward 85] [Ward 86] is an extension of the original SA/SD method, that is aimed at supporting the development of real-time systems.

[4] SDL is a CCITT standard [SDL] [Rockström 85], that is especially addressing specification of telecommunication systems. It has a textual as well as a graphical presentation.

1.2 Automated Support

Object-oriented development is amenable to automated support [Booch 86] and one consequence of introducing object-oriented concepts into analysis and design is the possibility to generate programming code from the design documents. [Booch 86] [Seidewitz 87] [Bailin 89] [Wasserman 89] propose and/or support mappings from the object-oriented structures of design documents to programming languages.

Concerning automated support the contribution of this report is to show how the OAD notation can be supported and especially the integration of OAD and BETA. It is shown how the notation directly can be mapped into the object-oriented programming language BETA [5]. Because the notation is directly mappable into an object-oriented programming language, the CASE gap [1], between analysis/design and implementation can be closed.

Not only the mapping of OADs into BETA can be automated but also mappings from BETA to OADs. OADs can be regarded as an alternative graphical abstract presentation of a BETA program. This means that OADs and the corresponding BETA program can share the same internal representation. The Mjølner BETA System [Knudsen 89] is a grammar-based programming environment and the internal program representation is abstract syntax trees. By letting the diagram editor and the textual editor manipulate common abstract syntax trees, modifications in the analysis and design diagrams can immediately be reflected in the program and vice versa.

The advantages of the grammar basis are many. Consistency can be maintained between analysis and design documents and programs. Code generation as well as and reverse engineering [6] can be provided by textual and graphical presentation of the common internal representation, respectively.

The CASE gap can however not be fully closed. OADs can be used to express the overall structure of a system, i.e. the aggregation and classification hierarchies of objects, and the sequencing and interaction between objects. The action sequences of the individual objects are not

[5] BETA [BETA 87] is an object-oriented programming language in the Simula [Simula67] tradition.

[6] This term is often used for automatic generation of design documents from program code.

expressed in the notation. Such details must be added manually in the generated program templates.

The preliminary notation could however be extended, for example in the direction of Real-time Structured Analysis and Design, that at least partly makes it possible to specify action sequences by adding control aspects to the traditional data flow diagrams. Other directions could be to specify the action sequences of objects like the process diagrams of OSDL, the structure diagrams and structured text in Jackson or Petri nets. In [Berthelsen 86] a graphical syntax for BETA is proposed, where every construct in the BETA programming language has a graphical presentation. This proposal could be used at the lower levels of abstraction. It is however my opinion, that graphical descriptions are most applicable (practical) at higher levels of abstraction. The purpose of the OAD notation is to introduce a graphical notation that can be used for modelling, not to provide a graphical syntax for a programming language.

1.3 This Report

The structure of the report is as follows. Chapter 2 gives a definition of object-oriented programming. Chapter 3 discusses how object-oriented development can integrate analysis, design and implementation. Chapter 4 presents the notation: Object-Oriented Analysis and Design Diagrams (OADs). Chapter 5 describes how abstraction, especially generalization/specialization, is supported in OADs. Chapter 6 shows how OADs can be mapped into BETA program templates. Chapter 7 discusses how OADs can be created and modified by means of a syntax-directed graphical editor, and how the mapping between OADs and BETA program templates can be automated. Appendix A presents some of the existing methods and notations for introducing object-orientation into analysis and design. Appendix B contains the BETA program parts that correspond to the diagrams in chapter 3. Appendix C shows how the transformation schema of Real-time SA/SD can be mapped into BETA.

Chapter 2

Object-Oriented Programming

This chapter elaborates on the definition of object-oriented programming given in the introduction. It represents the modelling view point on object-oriented programming in contrast to the the programming technical view point. For a more detailed discussion see [Madsen 88b].

The real world consists of phenomena and concepts.

Phenomena are characterized by having substance, measurable properties and some transformations that may alter the substance. An example is a person, that certainly has substance, a measurable property is the persons weight, and a transformation on the substance can be eating or jogging, which in turn may alter the measurable properties of the person. In a programming language, substance is modelled by means of variables or objects. Procedures (actions) are often used to examine the measurable properties or to alter the substance.

Concepts are created by abstraction, focusing on similar properties of phenomena and disregarding differences. Three well-known sub-functions of abstraction have been identified: classification, aggregation and generalization.

Classification is used to define which phenomena are covered by the concept. The reverse sub-function is exemplification. An example is again the concept of a person, that is a description of the properties of phenomena, that characterize a person. These properties can be a phenomena that walks on two legs, talks, destroys the ozone layer etc. An exemplification of the concept person is a specific person. In an object-oriented programming language the class and the procedure (or in BETA the pattern [1]), are used to model classification of phenomena. The class models

[1] The *pattern* concept in BETA unifies abstraction mechanisms like type, procedure, function

substance and the procedure models actions. Exemplification is simply an object, i.e. an instance of a class or an activation of a procedure.

Aggregation is used to express that concepts can be defined by using other concepts. An example is the concept of a car, that can be formed by using concepts like wheel, engine and seat. The reverse sub-function is called decomposition. Most programming language support aggregation by means of record, classes, procedures etc. A class has attributes, e.g. local objects, references to other objects, procedures or even local classes. Aggregation must be supported in an arbitrary number of levels in an object-oriented language, because the real world does not have any restrictions on the number of aggregation levels.

Generalization is used to organize concepts in classification hierarchies [2]. A concept can be regarded as a generalization of a set of concepts. An example is the classification of vehicles. A car is a generalization of bus, van and truck. A vehicle is a generalization of car and motor cycle. The reverse sub-function is specialization: a motor cycle is a specialization of vehicle. Generalization is perhaps that sub-function, which most clearly divides object-oriented languages from non-object-oriented languages. In an object-oriented language generalization/specialization is typically supported by super/sub-classing, respectively. Generalization must be supported in an arbitrary number of levels in an object-oriented language, because the real world does not have any restrictions on the number of generalization levels.

An object-oriented language must be able to model phenomena and concepts, as described above. A program execution in an object-oriented programming language is considered as a *physical model*, simulating the behavior of either a real or an imaginary part of the world.

In an object-oriented program, phenomena are modelled as objects and concepts are typically modelled as classes and procedures (or in BETA patterns).

---

and class

[2]The terms inheritance hierarchies or just class hierarchies are also often used.

The following quotation from [Madsen 88b] defines a physical model:

> "A physical model consists of *objects*, each object characterized by *attributes* and a sequence of *actions*. Objects organize the substance aspect of phenomena, and transformations on substance are reflected by objects executing actions. Objects may have part-objects. An attribute may be a reference to a part object or a separate object. Some attributes represent measurable properties of the object. The *state* of the object at a given moment is expressed by its substance, its measurable properties and the action going on then. The state of the whole model is the states of the objects in the model.
>
> Actions in the physical model are performed by objects. The action sequence of an object may be executed *concurrently* with other action sequences, *alternating* (that is at most one at a time) with other action sequences, or as part of the action sequence of another object."

A procedure is an example of a partial action sequence. The action sequence of a procedure is executed as part of the "calling" objects action sequence.

Sequencing between phenomena is modelled by concurrent or alternating objects.

Interaction or communication between phenomena is modelled by objects performing actions on other objects.

2.1 Language Support for Specialization

As mentioned in the introduction, special attention is given, in this report, to the support for generalization/specialization in object-oriented analysis and design languages; especially the graphical syntax for expressing generalization/specialization. The reason is, partly because it is a necessary part of modelling the real world, and partly because of the low priority these abstraction functions have had in many object-oriented method proposals.

Specialization can be categorized in

- *simple specialization* (also known as inheritance or subclassing)
  - a subclass inherits the attributes of the superclass and extends the superclass by adding attributes (Simula, BETA, Smalltalk)
  - a subprocedure inherits the attributes of the superprocedure and extends the superprocedure by adding attributes (including action, i.e. the body of the subprocedure can extend the body of the superprocedure) (BETA)
- *qualified specialization* (virtual procedures and virtual classes are used for this purpose)
  - a subclass overwrites or extends some of the attributes of the superclass; the attributes can be classes as well as procedures. (virtual procedure, overwriting: Simula, Smalltalk; virtual procedure, extending: BETA; virtual class, extending: BETA)
  - a subprocedure overwrites or extends some of the attributes of the superprocedure; the attributes can be classes as well as procedures. (virtual procedure, extending: BETA; virtual class, extending: BETA)

All methods in Smalltalk correspond to virtual procedures in Simula, because they overwrite methods with the same name in superclasses (control can however be directed to superclasses). One of the unique qualities of BETA is the ability to specialize not only classes but also procedures. Inheritance is also provided for procedures. Another unique quality is the virtual class concept. Virtual procedures and classes are not overwritten when bound, but extended. Virtual procedures and virtual classes are discussed in [Kristensen 87] and [Madsen 89], respectively.

Chapter 3

Integrating Analysis, Design and Implementation

This chapter discusses how the object-oriented programming perspective can integrate analysis, design and implementation. It is not an attempt to neglect the well-recognized and important division of these systems development activities. It is on the other hand an attempt to strengthen the interaction between analysis and design and the interaction between design and implementation.

Analysis is concerned with creating a model of a part of reality. This model is expressed in terms of application domain phenomena and concepts and can therefore be used as a communication medium between the users and the designers.

Design is concerned with transforming the analysis model into a model expressed in terms of phenomena and concepts that are closer to the computer. There is not a strict division between analysis and design; there is on the other hand a natural interaction between these activities, but at some time in the development process the main emphasis is moved from analysis to design.

Implementation is the final transformation of the design model into an executable program followed by testing. A similar interaction exists between design and implementation.

3.1 Traditional Analysis, Design and Implementation

One major problem in traditional (i.e. non-object-oriented) analysis, design and implementation is the amount of work needed when going from one model to another. This is especially the case in the transformation of the analysis model into the design model. The problem is that the two models have two logically different representations. One example is Structured Analysis (SA) and Structured Design (SD).

In SA the data flow diagram (together with textual descriptions like the data dictionary, event lists and minispecs) form the analysis model. The dataflow diagram models the flow of information through a network of processes (bubbles). A process is considered as a functional transformation from a set of inputs to a set of outputs.

In SD the dataflow diagrams are transformed into structure charts. A structure chart is a hierarchy of modules together with a description of parameters. These modules are in turn implemented as subprograms or procedures that manipulate common data structures.

The transformation from data flow diagrams to structure charts is a hard job because the two models have different representations. The transformation from design to implementation is much easier because procedural languages support hierarchies of procedures very well.

The same problem is present in the methods that only introduce object-orientation into design and implementation only [Booch 86] [Seidewitz 87] [Wasserman 89]. In order to ease the transformation from the analysis model to the design model and in turn to the implementation model, object-orientation must be introduced in all three models [Coad 89].

3.2 Object-Oriented Analysis, Design and Implementation

Object-oriented development means object-orientation in analysis, as well as design, as well as implementation.

Why object-oriented analysis? Because analysis and object-oriented programming have the same purpose: to model phenomena and concepts in a

part of reality. As described in the previous chapter the essence of object-oriented programming is to make a physical model of a part of the real world. A model where relevant phenomena from the real world are represented as objects, and where all relevant relationships and interactions between phenomena are reflected. Concepts are typically modelled by means of procedures and classes and all three (six) abstraction functions are supported.

Object-oriented development integrates analysis, design and implementation because the same representation can be used in all three activities. In the transformation from the analysis model to the design model and from the design model to implementation it is not necessary to change the representation. It is the same model that gradually is extended from a model in terms of application domain phenomena and concepts to a model that includes implementation domain phenomena and concepts. The final model is executable.

The design activity adds details to (precisizes) the analysis model. New objects and concepts are added. Objects and concepts that are directed against the implementation. It should be possible at any time to extract the analysis model from the design model. The analysis model can be considered as an abstract presentation of the design model, where all the technical details are suppressed. In the same way the design model and in turn the analysis model can be considered as an abstract presentation of the implementation model. These interesting relationships are very important when the discussion comes to automated support for systems development (CASE), as will be shown later.

Because the analysis model is expressed in terms of application domain phenomena and concepts, that are familiar to the user, the analysis model should be reflected in the user interface. In other words the object-oriented model should not only be considered as an internal "working model", the problem specific part of it should (at least partly) be visible in the user interface. One good example is the flight reservation system used later in this report. In this application it is natural to present a picture of the flight containing the seats, emergency exit, wings etc. Reservation is then made by selecting free seats from the picture. Special customer wishes can easily be satisfied by looking at the picture. The customer might for example want a seat near the emergency exit and not too close to the wings. Object-oriented user interfaces with direct

manipulation have proven to be especially suitable for human computer interaction [Smalltalk].

A word on programming: programming is not only an implementation activity. Programming takes place at all (three) levels. Programming is to make choices. In object-oriented programming it is to find the phenomena and concepts that we wish to model, and create corresponding objects, procedures and classes. Object-oriented development integrates analysis, design and implementation, but it does not remove the creative element that exists at each level.

3.3 The Notation

An important characteristic of many systems development methodologies is that the notations for expressing analysis and design models are graphical to a great extent. This is especially important in the analysis model, that is used as a communication medium between the users and the developers. It is a well-known phrase that "a picture can tell more than a thousand words", but there exist also situations where few words can tell more than thousand pictures. This is the underlying belief of the proposed notation: graphical descriptions are very suitable at higher levels of abstraction, but when it comes to details, these are more effectively expressed textually.

The notation is claimed to be object-oriented, i.e. it supports the object-oriented perspective. It can be used to model phenomena and concepts from a part of reality. The interactions and sequencing between phenomena can be expressed. Abstraction is supported. Phenomena can be classified into concepts and concepts can be organized in aggregation and classification hierarchies.

The notation is a description tool. The focus is on the description rather on the process of creating the description. There are no predescribed techniques (not to mention a methodology), but the object-oriented perspective as described in chapter 2 is the underlying philosophy. In addition this description of analysis, design and implementation is not concerned with the (although important) manual working routines of the edp-based system.

The creation of the analysis and design models involves:

- Identification of phenomena with substance, measurable properties and transformations on substance
- Identification of the aggregation hierarchies of objects
- Identification of interaction between objects
- Identification of sequencing between objects
- Identification of concepts that classify phenomena
- The concepts may be organized in classification and aggregation hierarchies

In the next section the notation is illustrated on the flight reservation system. In the chapters 4 and 5 the notation will be presented in details.

3.4 An Example

The flight reservation example is taken from [BETA 90]. The example and the diagrams will not be explained in detail. The intention of this section is to give an impression of the graphical notation. The corresponding BETA program parts are shown in appendix B.

Fig. 3.1 is an attempt to show that an OAD description can start informally, without many details. The upper diagram is called an *object diagram* and the lower diagram is called a *pattern diagram.* The object diagram shows that the flight reservation system consists of 4 objects, that are interacting in some way. The pattern diagram shows for each object the corresponding concept (pattern (class or procedure)). The pattern corresponding to the entire system object is shown on the left. It has 4 attributes, that also are patterns. Each of these patterns are shown in details in the square on the right. The entries of a pattern symbol can be part objects, object references to separate objects (either part objects or other external objects), procedure patterns or class patterns. There is currently no distinction in the graphics between part objects and object references, class patterns and procedure patterns.

Fig. 3.2 and Fig. 3.3 show a more detailed version of the system. New objects have been added to the object diagram and the objects have been

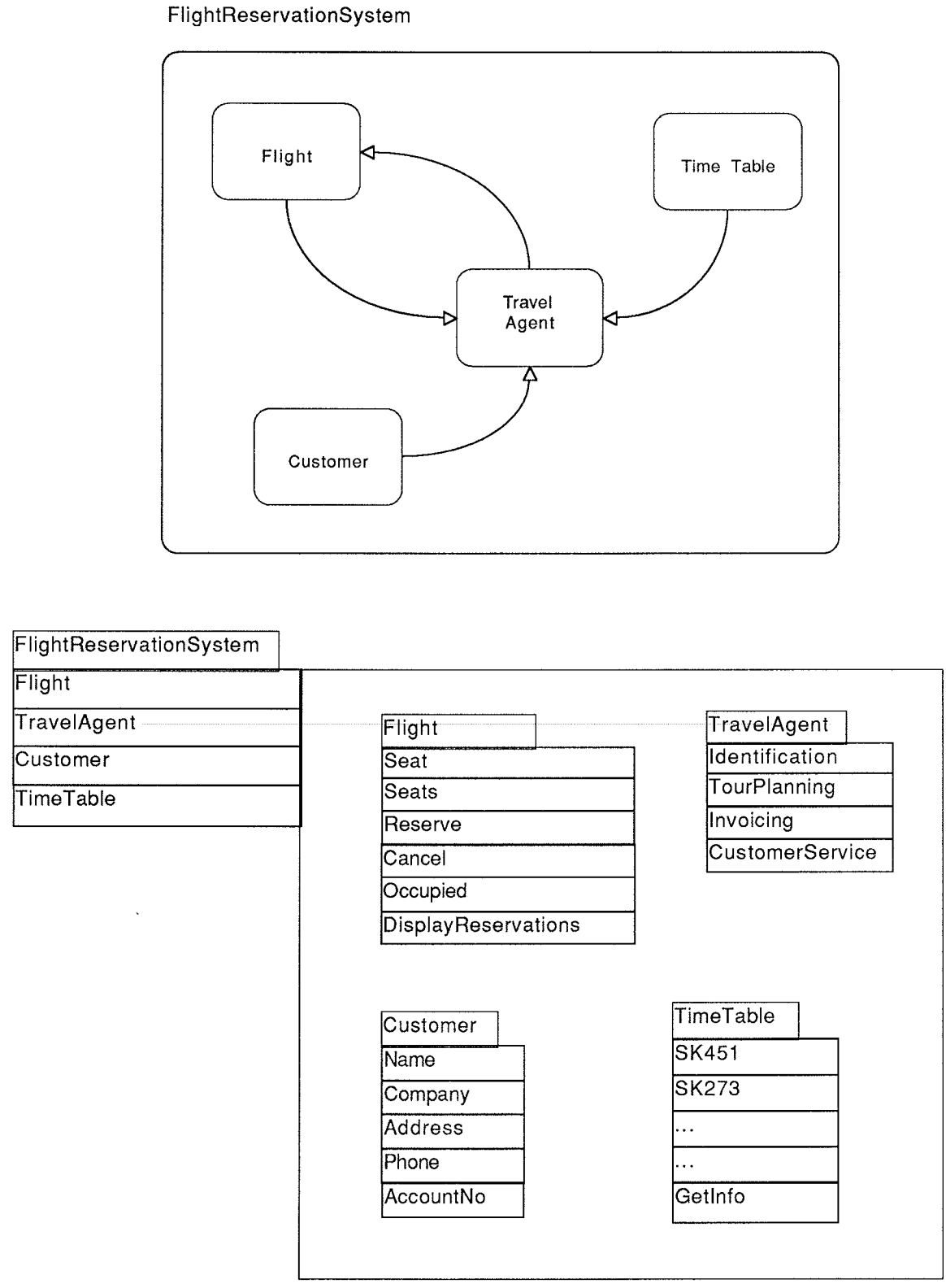


Figure 3.1: An Informal Model

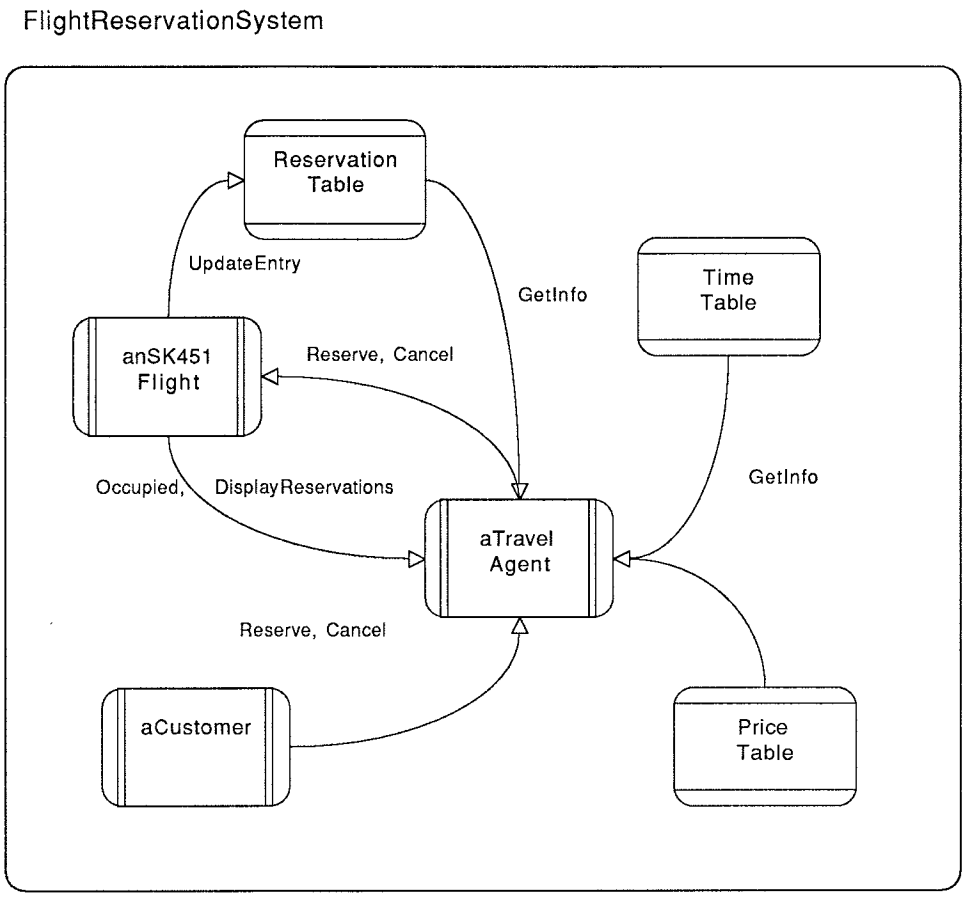


Figure 3.2: A More Detailed Model, Object Diagram

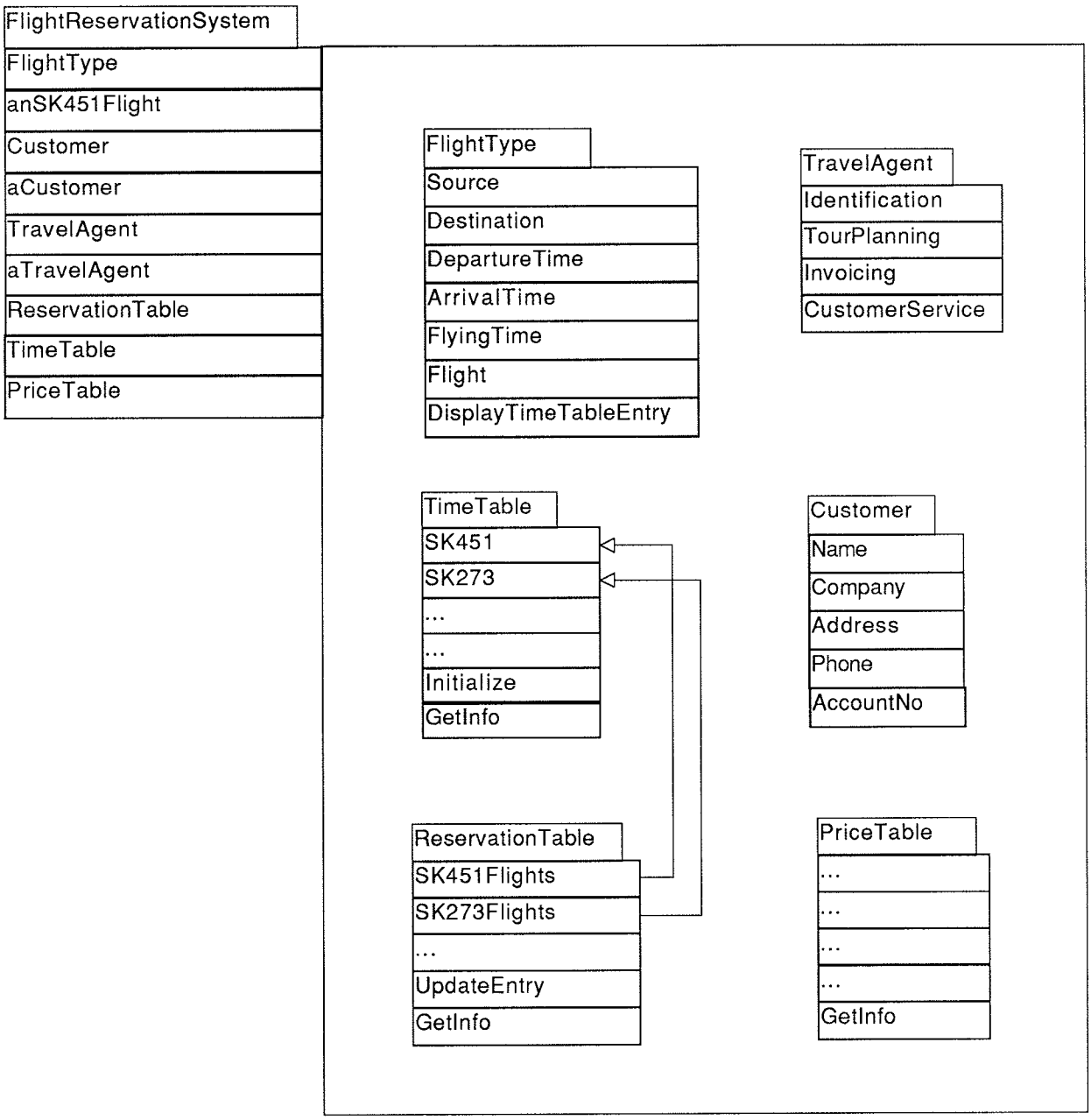


Figure 3.3: A More Detailed Model, Pattern Diagram

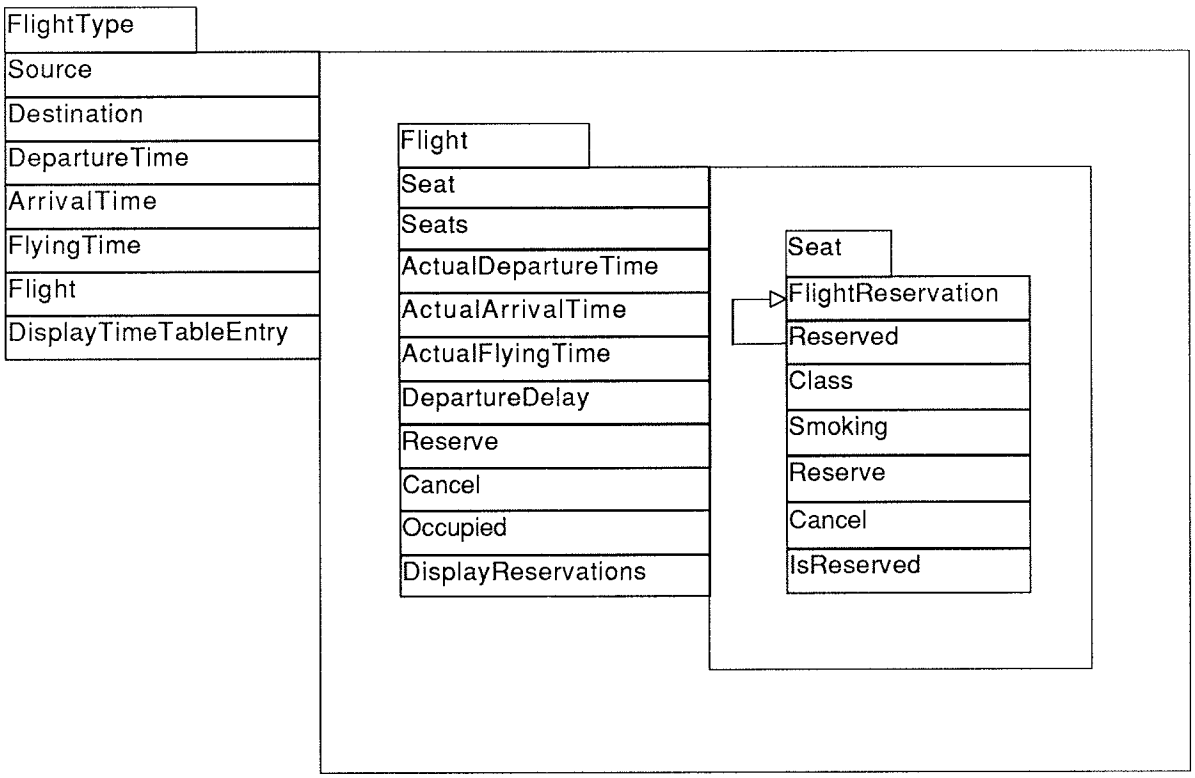


Figure 3.4: More Details, Flight Type

divided into active and passive objects. aFlight, aTravelAgent and aCustomer are concurrent (active) objects and ReservationTable, TimeTable and PriceTable are passive (data) objects. The interaction between the objects have been further specified.

The pattern diagram is similarly extended. In OAD it is possible to express *singular objects* as opposed to *pattern-defined objects*. Singular objects are directly defined without a pattern. The object descriptors of singular objects are also presented by means of pattern symbols in the pattern diagram. In Fig. 3.2 and Fig 3.3 it can be seen that aFlight, aTravelAgent and aCustomer are pattern-defined objects and ReservationTable, TimeTable and PriceTable are singular objects.

Object references are indicated by means of arrows from the attribute line to a pattern. This pattern is called the qualification of the reference. Objects referenced (at run time) must be instances of that pattern (or subpatterns of it). The part objects and the object references can be described in further details if wanted (this is done in appendix B).

Fig. 3.4 shows the FlightType pattern in more details. Fig. 3.5 shows

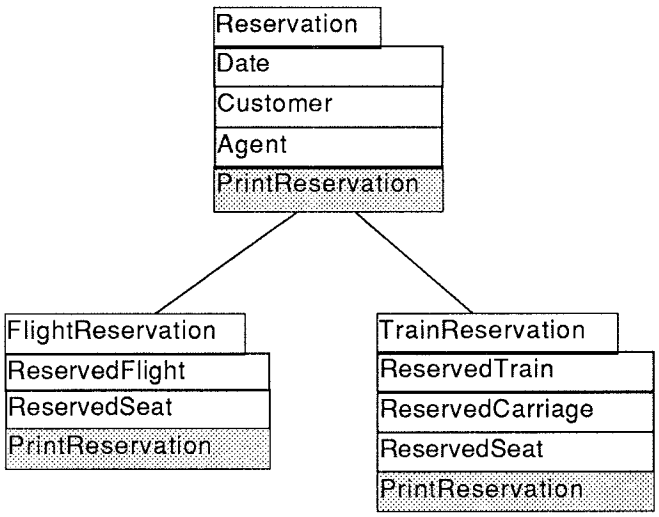


Figure 3.5: More Details, Specialization

that FlightReservation is a part of a classification hierarchy. FlightReservation and TrainReservation are subclasses of Reservation. PrintReservation is a virtual procedure pattern defined in Reservation and further extended in the subclasses. Class patterns can also be virtual. Agent for example could have been a virtual class pattern, that is defined in Reservation and further extended in the subclasses.

3.5 An Object-Oriented CASE Tool

Automated support for system development or computer aided software engineering (CASE) can be especially powerful in object-oriented development. The reason is the integration of the analysis model, the design model and the implementation model through the object-oriented representation.

A typical CASE tool provides diagram editors for the analysis model and the design model and program skeletons can be generated from the design model. Many CASE tools exist for different variants of SA and SD. The dataflow diagrams and the structure charts are integrated by means of the data dictionary, that is used as the basis for consistency and completeness check.

Object-oriented development is integrated by means of the object-oriented model. Because the analysis model, the design model and the implementation model have the same logical representation, consistency and completeness check should be more or less redundant. This is the case if the software representations of the three models actually are shared. The three models can then be considered as different presentations of the same underlying representation. In the proposal of this report the software representation of all three models are abstract syntax trees. The models are created and modified by means of a syntax-directed editor with a graphical and a textual interface. The graphical interface is used to create and modify the analysis and design model and the textual interface is used to create and modify BETA programs. Because of the integration of the models it is possible to generate program templates from the design model. On the other hand the design model and even the analysis model can be extracted from the program. The different models are considered as graphical or textual prettyprinting of the same underlying abstract syntax tree (AST). This is illustrated in Fig. 3.6.

Reverse Engineering

A lot of focus has been on *reverse engineering*. Reverse engineering is defined in [Chikofsky 90] as:

> "Reverse engineering is the process of analyzing a subject system to
>
> - identify the system's components and their interrelationships and
> - create representations of the system in another form or at a higher level of abstraction.
>
> Reverse engineering generally involves extracting design artifacts and building or synthesizing abstractions that are less implementation-dependent.
>
> ....
>
> In spanning the life-cycle stages, reverse engineering covers a broad range starting from the existing implementation, recapturing or recreating the design, and deciphering the requirements actually implemented by the subject system."

Some CASE tools support reverse engineering to some extent. Design models are generated from the programs but because of the different representations of the analysis and design there is no hope for extracting the analysis model.

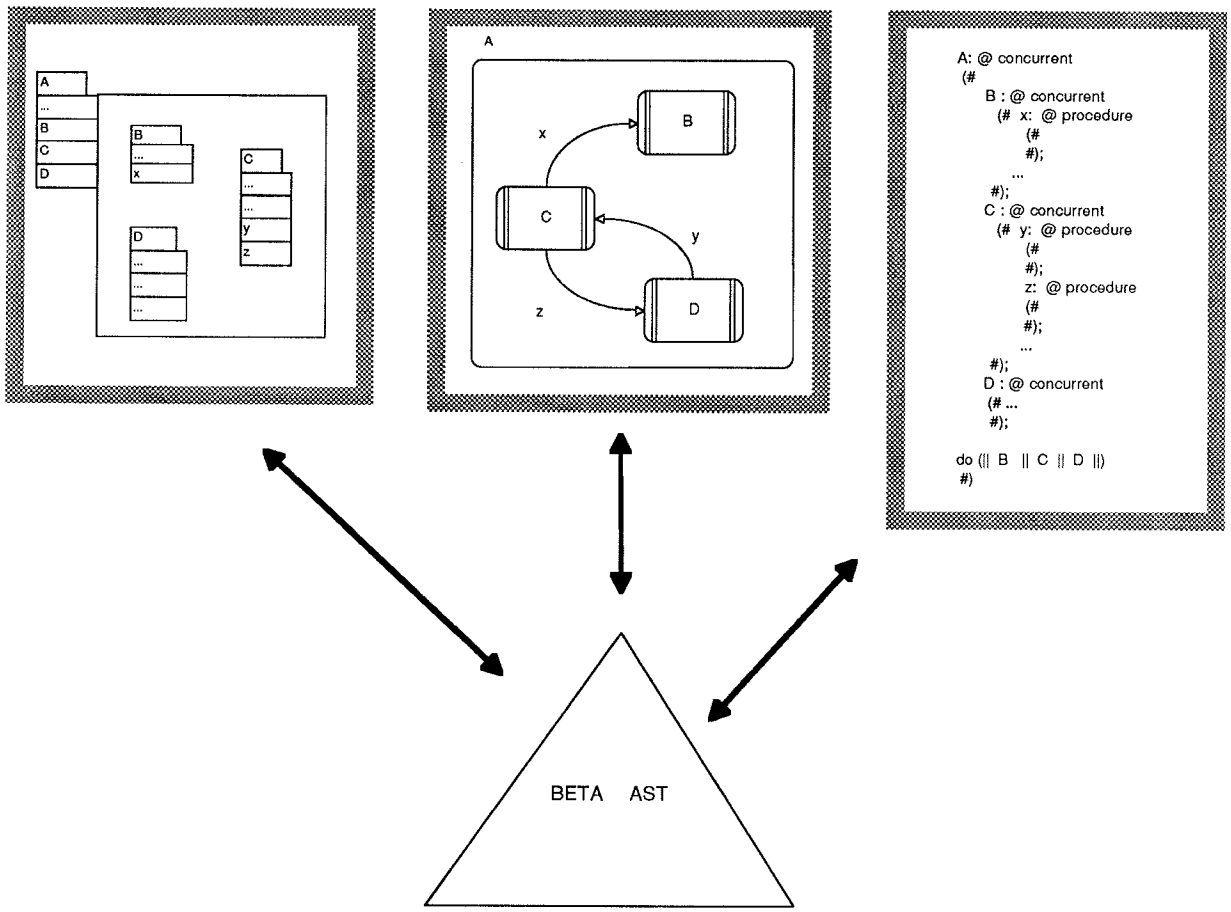


Figure 3.6: Integration of A, D, and I through ASTs

In object-oriented development it is not only possible partially to automate the transition from design to implementation but also to automate the transition from implementation to design and even to the analysis model. It might be difficult completely to distinguish between the analysis and design model, but a high level graphical presentation of the program should reflect the basic structure of the analysis model.

The proposed CASE tool has no support for the creation of user interfaces or databases.

In chapter 7 the object-oriented CASE tool will be further described.

Chapter 4

Object-Oriented Analysis and Design Diagrams

A preliminary notation is proposed for expressing object-oriented analysis and design diagrams (OADs). The purpose of the notation is to make it easier to create an object-oriented model, that in turn can be mapped into an object-oriented program. The choice of geometric symbols, shadowing etc. is more or less arbitrary; the important thing is, that the notation can be used to express the necessary aspects in object-oriented modelling.

The notation is addressing phenomena as well as concepts. One important characteristic of the notation is the emphasis on expressing objects, object interaction and sequencing between objects, as well as concepts and concept hierarchies, and finally the connection between objects and concepts. Objects and object relationships are expressed in *object diagrams.* Concepts and concepts relationships are expressed in *pattern diagrams.*

The notation distinguishes between the different kinds of action sequences, that may be executed. It is possible explicitly to express different kinds of objects: concurrent objects, alternation objects, procedure objects and passive objects.

The sub-functions of abstraction, including generalization and specialization, can be expressed in the notation. Abstraction in OADs is presented in chapter 5.

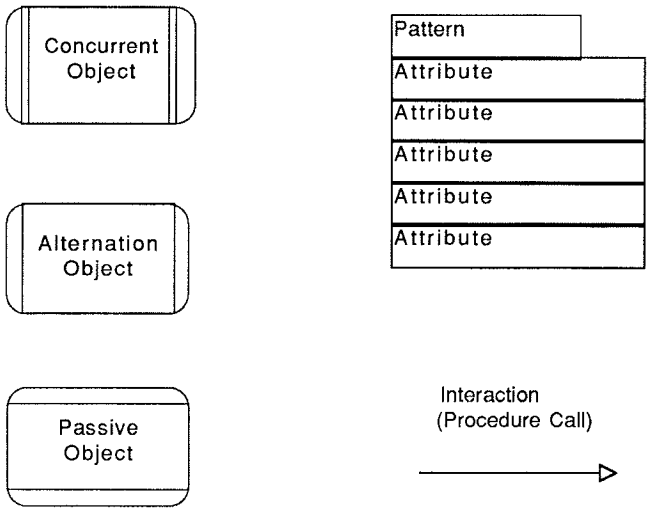


Figure 4.1: The OAD Notation

4.1 The Notation

Concurrent objects are objects, that have their own execution thread. Alternation objects are objects, that are executed by concurrent objects one at a time (they are alternating). Procedure objects are objects, that are executed as part of an objects action sequence. Passive objects are like instances of abstract datatypes, they are just providing access to (operations on) a data structure [1].

The symbols of the notation are shown in Fig 4.1.

All objects are shown as rectangles with rounded corners. This symbol has been chosen instead of the traditional circle, simply because there can be more text in a rectangle relatively to the area.

Concurrent objects are shown as rectangles with rounded corners and a pair of double-vertical lines symbolizing parallelism.

Alternation objects are shown as rectangles with rounded corners and a pair of single-vertical lines.

Passive objects are shown as rectangles with rounded corners and a pair of horizontal lines symbolizing a store. The difference between a passive

[1] Concurrent objects, alternation objects, procedure objects and passive objects correspond to system objects, component objects, item objects and data objects in BETA respectively.

object and a store (in SA data flow diagrams) is, that the passive object includes the operations that manipulate the store. A passive object corresponds to an instance of an abstract data type with no independent action sequence (except may be for initialization).

Procedure objects are only shown indirectly in the object diagrams. Interactions between objects, as defined below, corresponds to procedure calls i.e. creation or activation of procedure objects.

Interactions are shown as arrows. Interactions can be used like flows of SA and real-time SA/SD. An interaction can be considered as a transport medium for data, but it can also be considered as a transaction or a signal. An interaction can be bi-directional. The direction(s) of an arrow can reflect data flow, control flow or whatever the designer wishes to model. Interactions are mapped to procedure calls.

Interactions must be labelled. Corresponding to the labels, procedures must be provided by one of the connected objects. If an interaction is drawn between two concurrent objects, execution of the procedure is synchronized like the rendezvous in Ada. If no concurrency is involved, the interaction corresponds to a normal procedure call.

Patterns are shown as rectangles with a title. The entries of the rectangle express attributes of the pattern. A pattern is either a class or a procedure. Attributes may be part objects, object references, class patterns or procedure patterns. The purpose of introducing the pattern symbol in the object-oriented analysis and design diagram is to be able to model concepts. This will be further discussed in chapter 5.

4.2 An Example

To illustrate the OAD object diagram the flight reservation example will be used again. Fig. 4.2 shows the top level OAD object diagram. The system models a flight reservation system containing flights, a reservation table, a time table, a price table, travel agents and customers. Each of these phenomena is represented by an object in the model. It might not be surprising that the tables are modelled as objects. But to model the flight as an object that is requested for making reservations, might be unusual to some people. This is a consequence of making a physical model. Concerning flight reservation, a flight is considered as an object

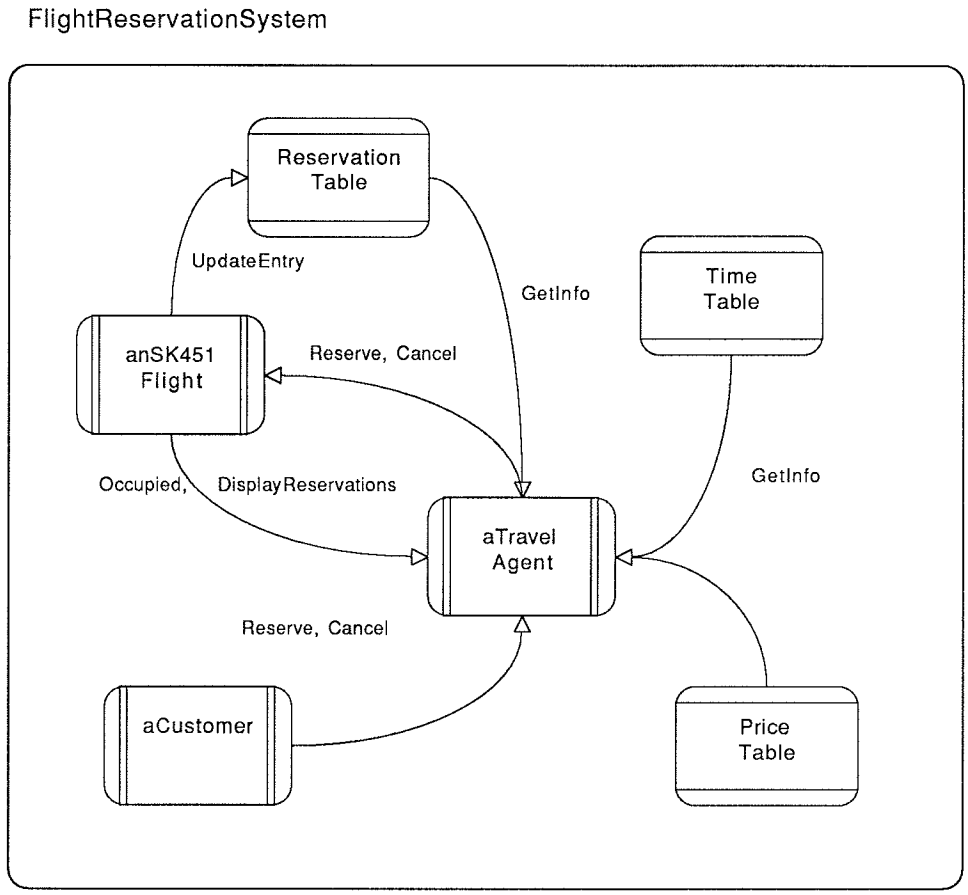


Figure 4.2: Flight Reservation, Concurrent Objects

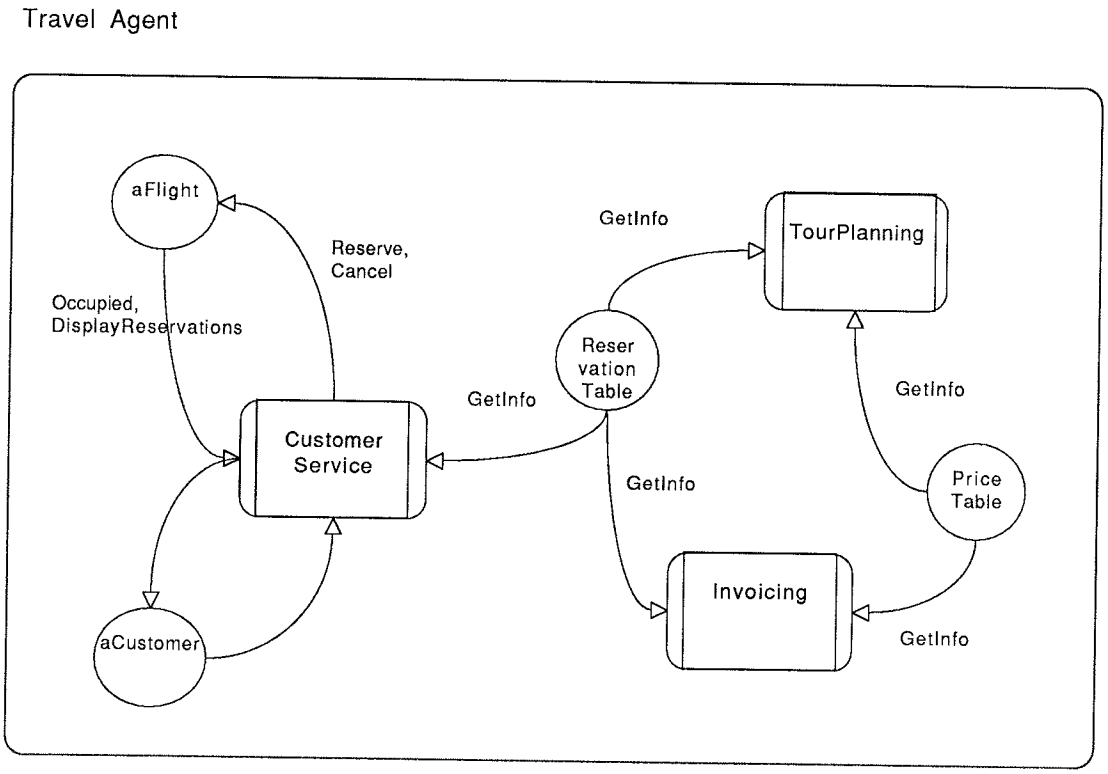


Figure 4.3: Travel Agent, Alternating Objects

with a number of seats, each of which can be reserved. The travel agent objects makes reservations by interacting with the flight objects.

The flights, travel agents and customer are modelled as concurrent objects. This is not an implementation decision. In the real world customers and travel agents are indeed independent phenomena that have their own lives. Flights may not be considered as living phenomena, but it this application it makes good sense to model flights as active objects, because several travel agents can interact with the same flight.

The tables are modelled as passive objects, that contain some data structures and a set of operations, e.g. ReservationTable with operations like UpdateEntry and GetInfo.

At the next level of detail we will explicitly model that the travel agent has several jobs to do. The primary job of the travel agent is to serve customers, but whenever there are no customers to serve, the travel agent must do invoicing and tour planning. Whenever a customer arrives or phones, the travel agent must shift to servicing the customer. The Travel Agent is decomposed as shown in the object diagram of Fig. 4.3.

To model the three alternating activities, that a travel agent has to do, three alternation objects have been introduced: CustomerService, Invoicing and TourPlanning. These objects can not be concurrent. The travel agent can only do one thing at a time, but she or he can have several ongoing activities, that she or he alternates between.

The different tables are used by the three alternating objects. The access to these objects must of course be synchronized, this will be further discussed later.

The connection to the outer objects are shown as small circles containing the name of the outer objects. Outer objects are the objects in the diagram, that contains the object, which this diagram is a decomposition of.

This example will later be mapped to BETA.

4.3 Traditional Structured Analysis

In the DFD of Structured Analysis, transformations model processes or functions, that according to inputs produce some outputs. Inputs and outputs are represented as data flows. Stores model storage of data, typically as files.

In OAD object diagrams the focus is on objects as defined in chapter 2. OADs are intended to express the physical model, at least at higher levels of abstraction. Therefore the emphasis is on explicit identification of the different kinds of objects, that model phenomena: concurrent objects, alternating objects, procedure objects and passive data objects. Modelling of concepts in OADs is presented in chapter 5. Like the flows in SA the interactions in OADs can be considered as inputs to and outputs from objects, but the data is "carried" by procedure objects. The very procedure call can be considered as an input (like a signal), or parameters can be used to "carry" data. Therefore an interaction can easily be bi-directional.

4.4 Real-time Structured Analysis

The application area of Structured Analysis is development of administrative systems where processing of files is a substantial part of the system. In real-time applications like processor control systems SA has appeared to be inadequate. Real-time Structured Analysis and Structured Design (RT SA/SD) [Ward 85] [Ward 86] is an extension of the original SA/SD method, that is aimed at supporting the development of real-time systems. RT SA/SD extends the original method to deal with concurrency, synchronization, and other control issues. One addition is a version of the entity-relationship diagram (ERD), another is the use of stimulus-response analysis. The ERD is used to describe the application domain. The stimulus-response analysis is reflected in an extended version of the dataflow diagram called the *transformation schema* (TS). The TS has additional symbols to represent real-time aspects. Data flows are categorized into *continuous* and *discrete* flows. *Event* flows (interrupt-type messages with no variable content) are added. In addition to the data transformation (process), a *control transformation* (process) is used to turn other transformations on and off by means of enabling and disabling event flows. Control transformations are further specified by means of *state transition diagrams* (STDs). *Terminators* (also known in traditional SA) represent a phenomena in the real world that provides information to the system, or receives some information from the system. A typical terminator represents either continuous data like a monitor of e.g. temperature or speed (this is typically provided by a hardware component), or discrete data like the dialogue between a person and an administrative system (user interface).

In OAD object diagrams there is no special symbol for control objects i.e. processes that are controlling other processes, because there is no principal difference between a controlling process and a "working" process. In real-time SA/SD control transformations are further specified by means of a state transition diagram (STD). STDs can be provided for concurrent objects in general.

In the OAD notation there is no distinction between value and non-value bearing interactions (data flows and signals respectively), as well as no distinction between discrete and continuous flows. Such specialized interactions may be added to the notation, but the intention behind this

notation, has been to keep it general and direct mappable to the basic constructs of BETA. In this way the graphical notation can be used as a graphical presentation of BETA (at least at higher levels of abstractions). Continuous flows and asynchronous signals can easily be expressed in BETA (this is further discussed in chapter 6).

There is no special terminator symbol, since a terminator object can be expressed by one of the 4 object types, typically a concurrent object or a passive object.

Chapter 5

Abstraction in OADs

As discussed in chapter 2 an object-oriented language must be able to model not only phenomena in the real world but also concepts. Phenomena in the real world are often classified into concepts by means of abstraction. Concepts may be organized into classification hierarchies, and concepts can be aggregated from other concepts. If the notation, as a description language, is supposed to be object-oriented, it must support the three subfunctions of abstraction (and their reverse subfunctions). Pattern diagrams are used to express concepts and concept relationships, but abstraction is also expressed in object diagrams.

5.1 Classification

A concept is modelled by means of an OAD pattern. A pattern can either be used as a class or a procedure. The OAD pattern is presented as a rectangle with a title and a number of attributes (lines) Attributes may be part objects, object references, class patterns or procedure patterns. See Fig. 5.1. The upper diagram is an object diagram and the lower diagram is a pattern diagram. In the example X, anX, anotherX, aY, Z, V, doSomething, doAnotherThing are attributes of A. Object references are indicated by means of arrows from the attribute line to a pattern. This pattern is called the qualification of the reference. Objects referenced (at run time) must be instances of that pattern (or subpatterns of it).

In the object diagram the substance and some dynamic aspects of the pattern A is shown. The substance is simply the part objects of the pattern. The dynamic aspects are the sequencing between objects and

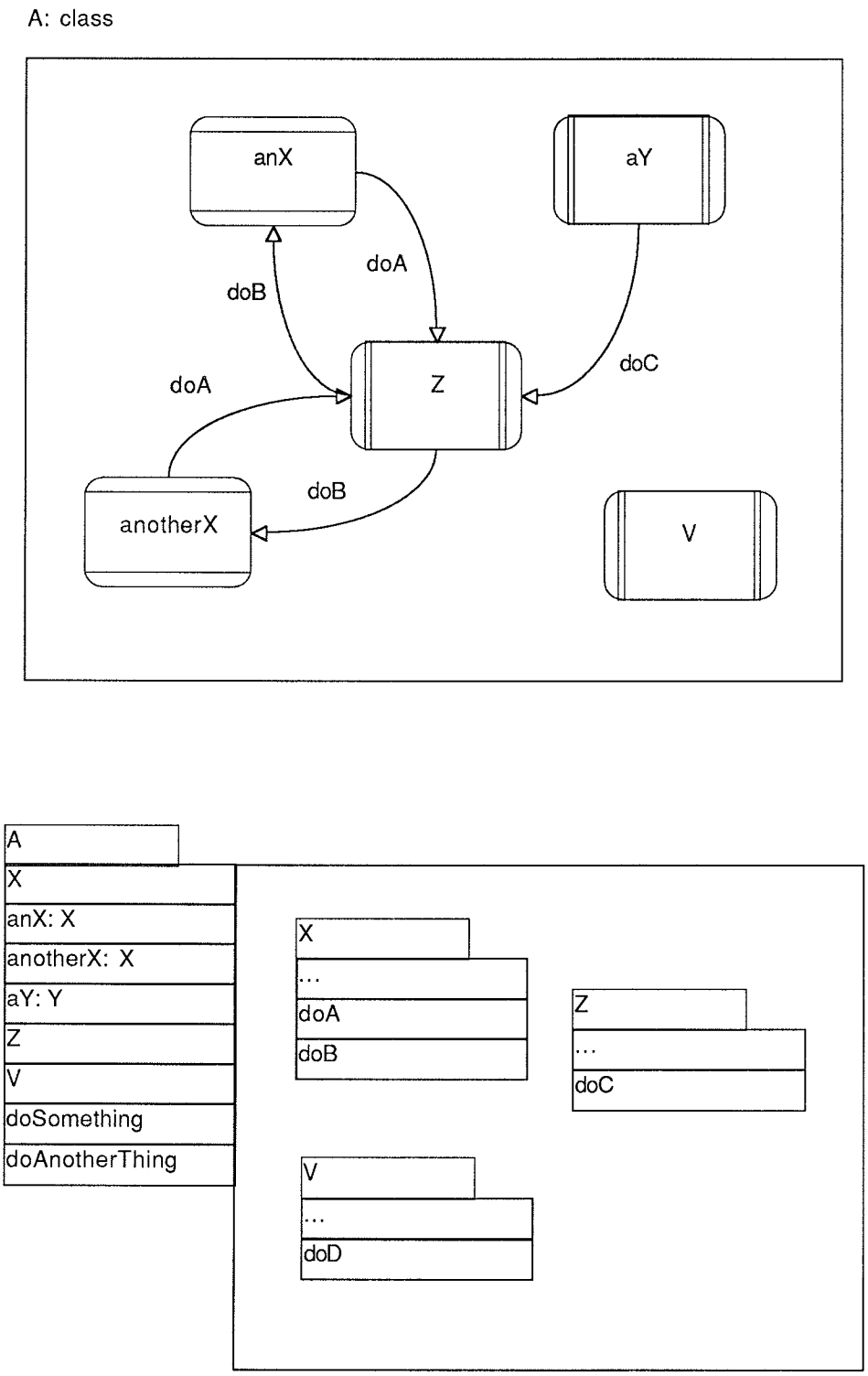


Figure 5.1: Classification/Exemplification

the interaction between objects. Exemplification is done differently for class patterns and procedure patterns [1]. Class patterns are instantiated as concurrent, alternation or passive objects. Procedure patterns are instantiated as interactions (procedure calls).

Concurrent, alternation or passive objects are created by incorporating an object symbol in an object diagram. When instantiating the class pattern, the type of object (concurrent, alternation or passive) is decided by choosing the corresponding object symbol. An arbitrary number of objects can (of course) be instantiated from the class pattern. Objects may be instantiated directly from a object descriptor (singular object) or by referring to a class name (class-defined object). In the latter case the name of the object and the name of the class are written (in the attribute symbol) separated by ':', just as in textual programming languages. If more than one object of a given class is needed, the objects must be class-defined. The object descriptors of singular objects are also presented by means of pattern symbols in the pattern diagram. See Fig. 5.1. The objects Z and V are singular concurrent objects, anX and anotherX are class-defined passive objects. Class X is local to class A, but class Y is defined at an outer level or in a superclass. The V object is not interacting with any other object, at least not at this stage in the analysis/design model.

An interaction is created by choosing a source object and a destination object. The distinction between source and destination is only a matter of specifying the direction of the arrow, but the arrow can also be bi-directional. An attribute in the corresponding class pattern of one of the two objects must be selected or created. This attribute will then be the procedure pattern that is instantiated in the interaction. See Fig. 5.1. The doA interaction between anX and Z has been created by selecting the doA attribute (in the X pattern that corresponds to the anX object) and connecting it to the Z object. In this case anX is the source object and Z is the destination object.

In the preliminary notation there is no graphical distinction between the different kind of attributes in a pattern: part objects, object references, class patterns or procedure patterns. The intention is to avoid overloading the diagrams with different graphical symbols. Such distinctions might

[1] In fact the distinction between class patterns and procedure patterns is determined by the object kind created from the pattern.

however be added.

5.2 Aggregation

OAD patterns/objects can be nested in an arbitrary number of levels. In the last section we have already seen how aggregation is expressed in pattern diagrams. In the lower diagram of Fig. 5.1 the pattern symbol for A on the left shows that A is composed of a number of attributes. The square area on the right is a further decomposition of some of the attributes, namely the pattern attributes.

The aggregation hierarchy of objects is reflected in a corresponding hierarchy of object diagrams. The correspondence between the aggregated object and the decomposition diagram is illustrated in Fig. 5.2. The lower diagram shows the decomposition of the A object in the upper diagram.

The connection to outer objects are shown as small circles containing the name of the outer objects. Outer objects are the objects in the diagram, that contains the object, which this diagram is a decomposition of. In the example B, C and D are outer objects of the lower diagram. An example is given in chapter 4.

Notice that concurrent, alternation and passive objects can be aggregated in this way. Aggregation of interactions is shown in the pattern diagrams.

5.3 Simple Specialization

It is relatively easy to express a general concept, but it is more difficult to make specializations without repeating the general concepts. Simple specialization (i.e. inheritance) is often used to aggregate attributes from the superclass together with added attributes in the subclass. Fig. 5.3 shows simple specialization.

Class AA is a simple specialization of class A, or in other terms AA is a subclass of A. This means, that everything in class A is visible to class AA, and the subclass can refer to any attribute in A.

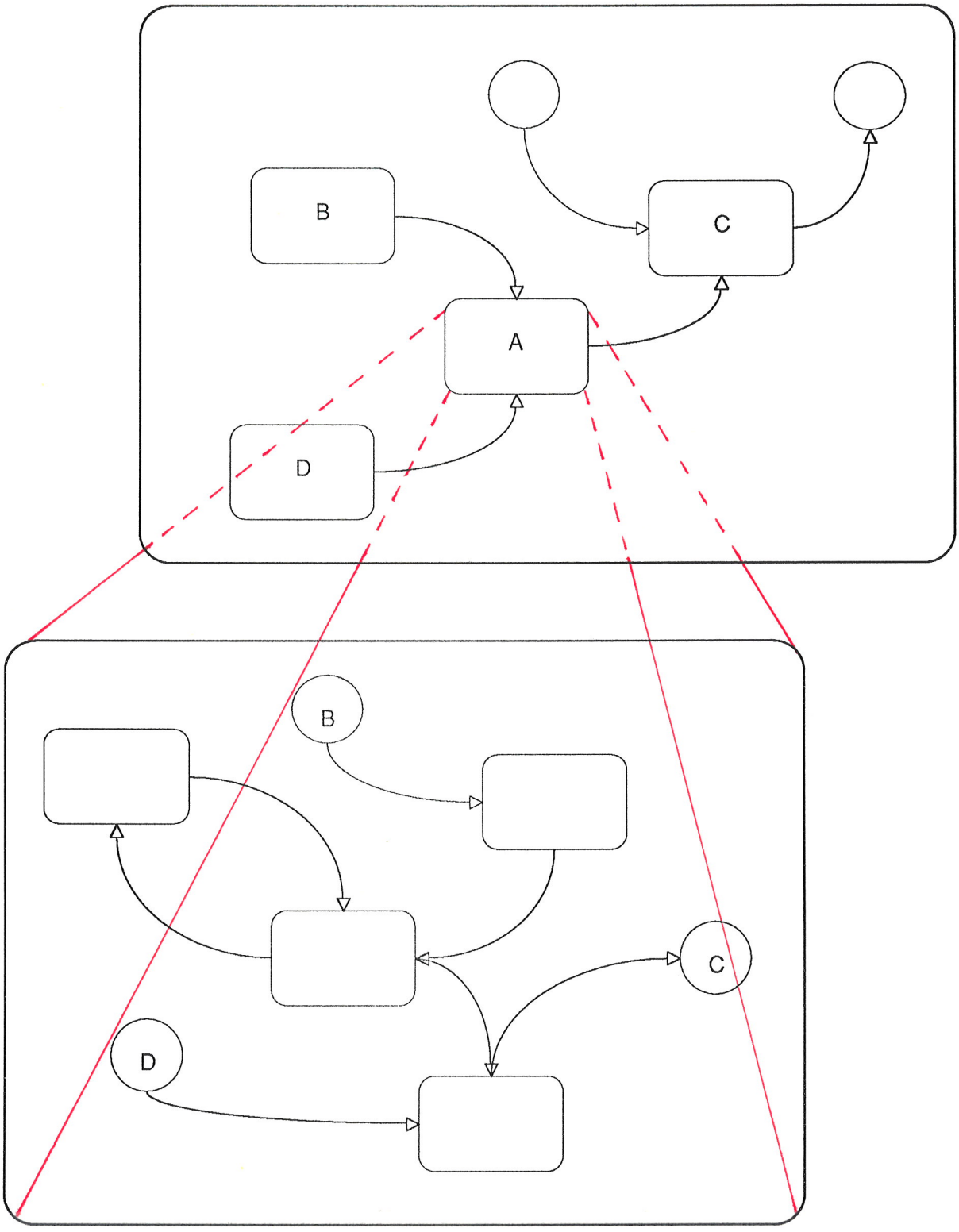


Figure 5.2: Aggregation/Decomposition

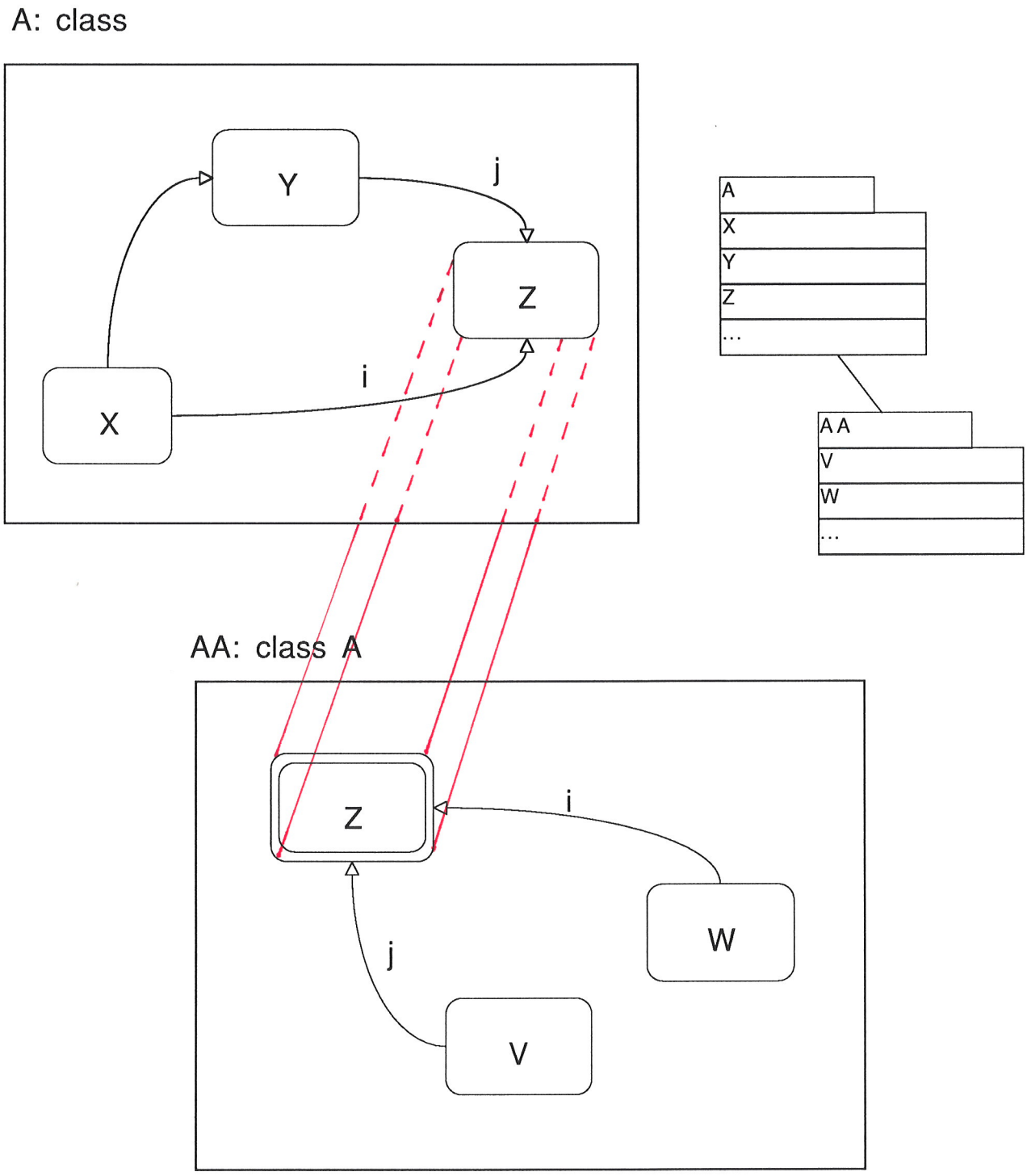


Figure 5.3: Simple Specialization

Object Diagrams

Objects in AA can for example be connected to objects in A by interactions, as shown in the figure. This is done by redrawing the relevant objects of A in AA. To indicate that they are inherited, they are drawn as object symbols with a double border.

Notice that concurrent, alternation and passive can be specialized in this way. This adds 3 symbol variants to the OAD language. Specialization of procedure objects/patterns is shown in the pattern diagram.

Pattern Diagrams

The pattern diagram shows the classification hierarchy of patterns as a corresponding tree of pattern symbols. The pattern symbols are connected by means of undirected lines. In the example class A has only one subclass. It is shown that subclass AA extends A with the attributes V and W.

5.4 Qualified Specialization

Qualified specialization was defined as overwriting or extending some of the inherited attributes in the subclass. Those attributes that can be overwritten or extended are called virtual attributes. In Smalltalk for example all methods are virtual procedures.

In OAD virtual attributes must be indicated in the superclass (*virtual definition*). This is done by shadowing the object and its corresponding pattern symbol. In the subclasses the overwriting or extension of the attribute must also be indicated (*further binding*). In the object diagram it is done by a combination of the virtual "shadowing" and the inheritance "double bordering". In the pattern diagram it is done by means of shadowing the virtual definition attribute (in the superpattern) as well as the further binding attribute (in the subpattern). An example of qualified specialization is shown in Fig. 5.4. In this example aZ is an object defined by the virtual pattern Z in class A. In the subclass AA, Z is further bound, which means that the aZ object of A now is extended.

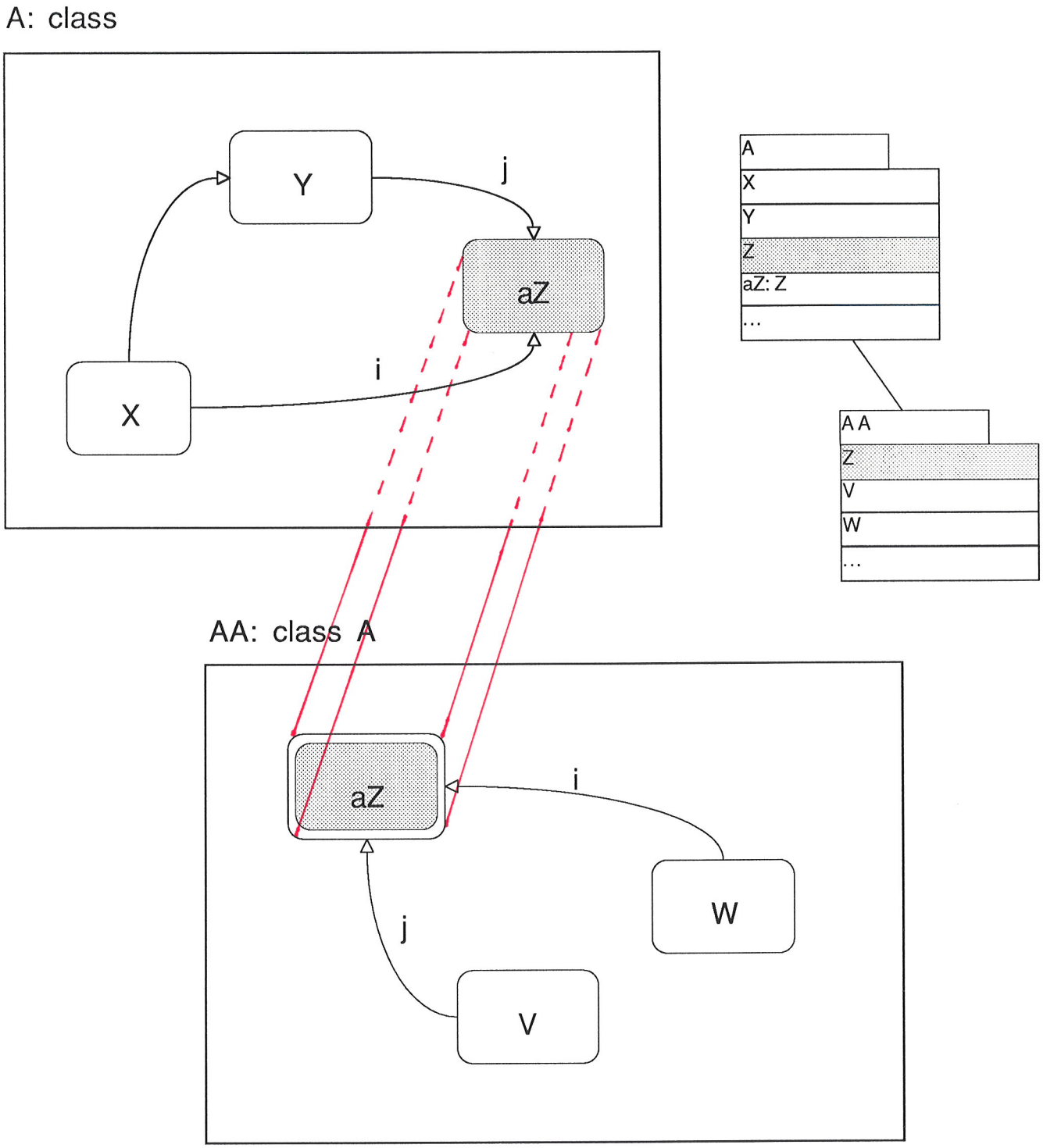


Figure 5.4: Qualified Specialization

Notice that concurrent, alternating and passive objects can be specialized in this way. This adds 6 symbol variants to the OAD language.

A call of a virtual or further bound procedure is indicated by a dashed arrow in the object diagram and as a shadowed attribute in the pattern diagram.

5.5 An Example

To illustrate specialization in the OAD notation, the Ward example [Ward 89] (See appendix A) is used. It is part of a system that controls filling and emptying a tank of liquid. Valves are used to fill or empty the tank. In the example a general valve is described. This valve is specialized into an inlet valve and an outlet valve. The inlet valve functions like the general valve, but the outlet valve must, in addition to the general properties of a valve, check, whether the valve status is manual or automatic. If the status is manual, the system may not operate on the valve, and the outlet valve must send a valve failure signal to the tank control transformation. Wards solution to this problem is using simple specialization. This section illustrates simple and qualified specialization in the OAD notation.

Simple Specialization

Fig. 5.5 shows how simple specialization can be expressed in the OAD notation.

The object diagram shows that Class Valve contains two concurrent objects and one passive object. Open Valve is a concurrent object that when enabled sends a signal to the physical Valve telling to it to open. The same signal is sent to the Monitor valve object that checks the actual on/off status of the physical valve according to some data (Operating time). If there is a mismatch between the desired and the actual on/off status a Valve failure signal is sent. This general valve is specified without a control object, that enables or disables the Open Valve object. The specification of valve control is deferred to subclasses of Valve. Class Inlet Valve has a Control Inlet Valve object and class Outlet Valve has a Control Outlet Valve object. In addition Outlet Valve has an object

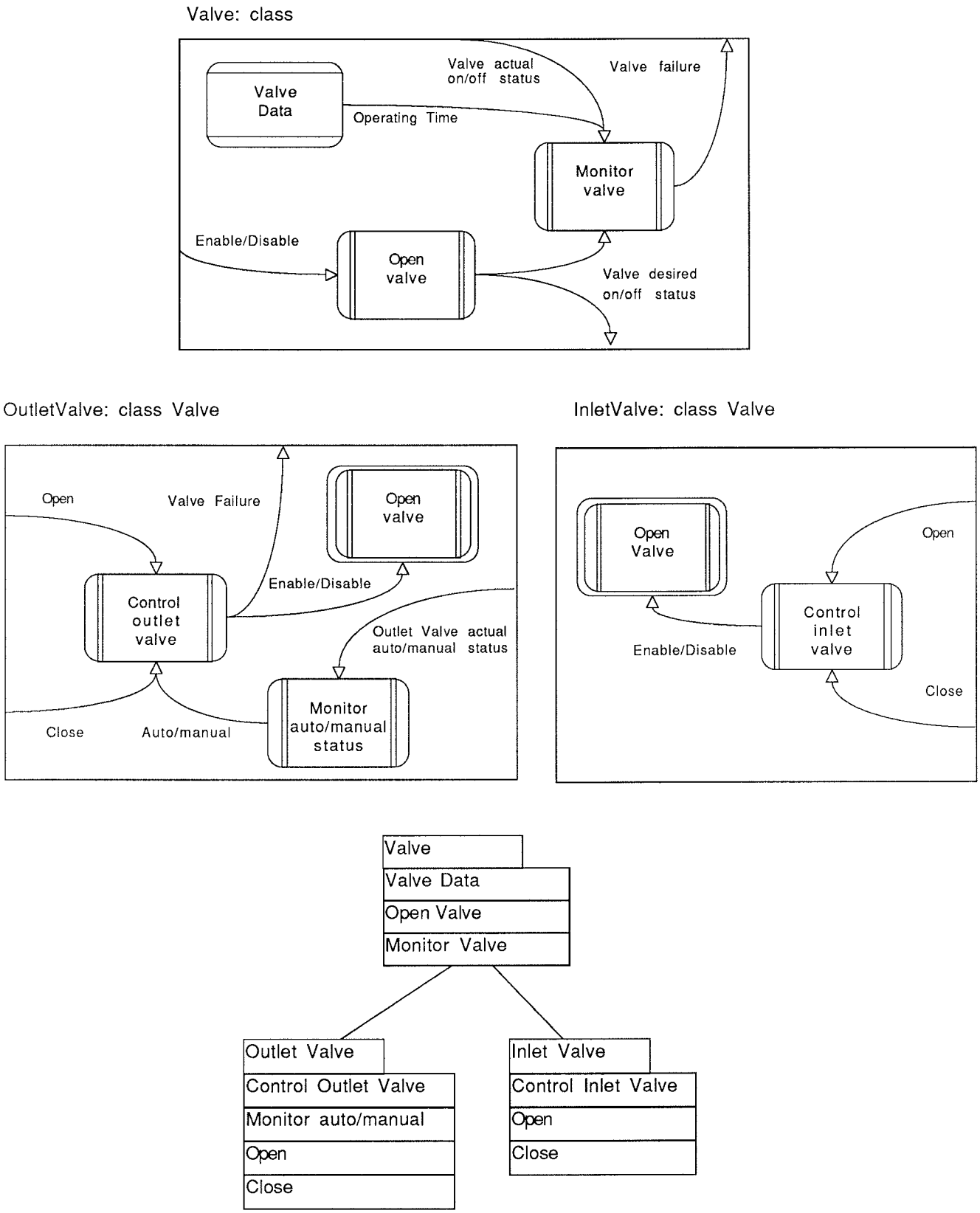


Figure 5.5: Simple Specialization in OAD

that signals the status of the auto/manual switch in the system. This additional signal is considered in the control logic of the Control Outlet Valve object. The output from the control objects must be send to Open Valve. To specify this, the Open Valve object of the superclass is repeated in the subclasses. The double border of the Open Valve object indicates, that it is inherited from a superclass.

The corresponding pattern diagram also shows the simple specialization of the Valve Class into Outlet Valve and Inlet Valve, but here the focus is only on the attributes.

The Ward Solution

This solution is different than Wards solution. Ward specifies inheritance by allowing diagrams of subclasses to contain their superclass as an object. To describe that the attributes of Valve are inherited in Outlet Valve and Inlet valve, the Valve symbol is repeated in the subclasses. This solution can be characterized as a kind of "downward aggregation". Valve is presented as an aggregate of attributes, but it is not possible to see the contents of an aggregate, except for the in-going and out-going interactions. It is therefore not possible to add an interaction to an internal object of the superclass, for example to call a procedure in a passive object of the superclass. This suggests, that the solution of Ward does not cover all cases of simple specialization.

Qualified Specialization

In the Ward example the Valve was defined as an abstract superclass, i.e. a class that only is used as a superclass. There will never be created objects of this class, because the control logic is missing. Another solution is to specify the Valve as a complete class including the control object. In this way a Valve object can be created directly from the Valve class, or from subclasses of Valve. See Fig. 5.6.

Two operations are added to the Valve class: Open and Close, as interactions to the Control object. The Control object must be able to function together with general Valves, and together with the more specialized Outlet Valve. This means that it must be modified in the Outlet

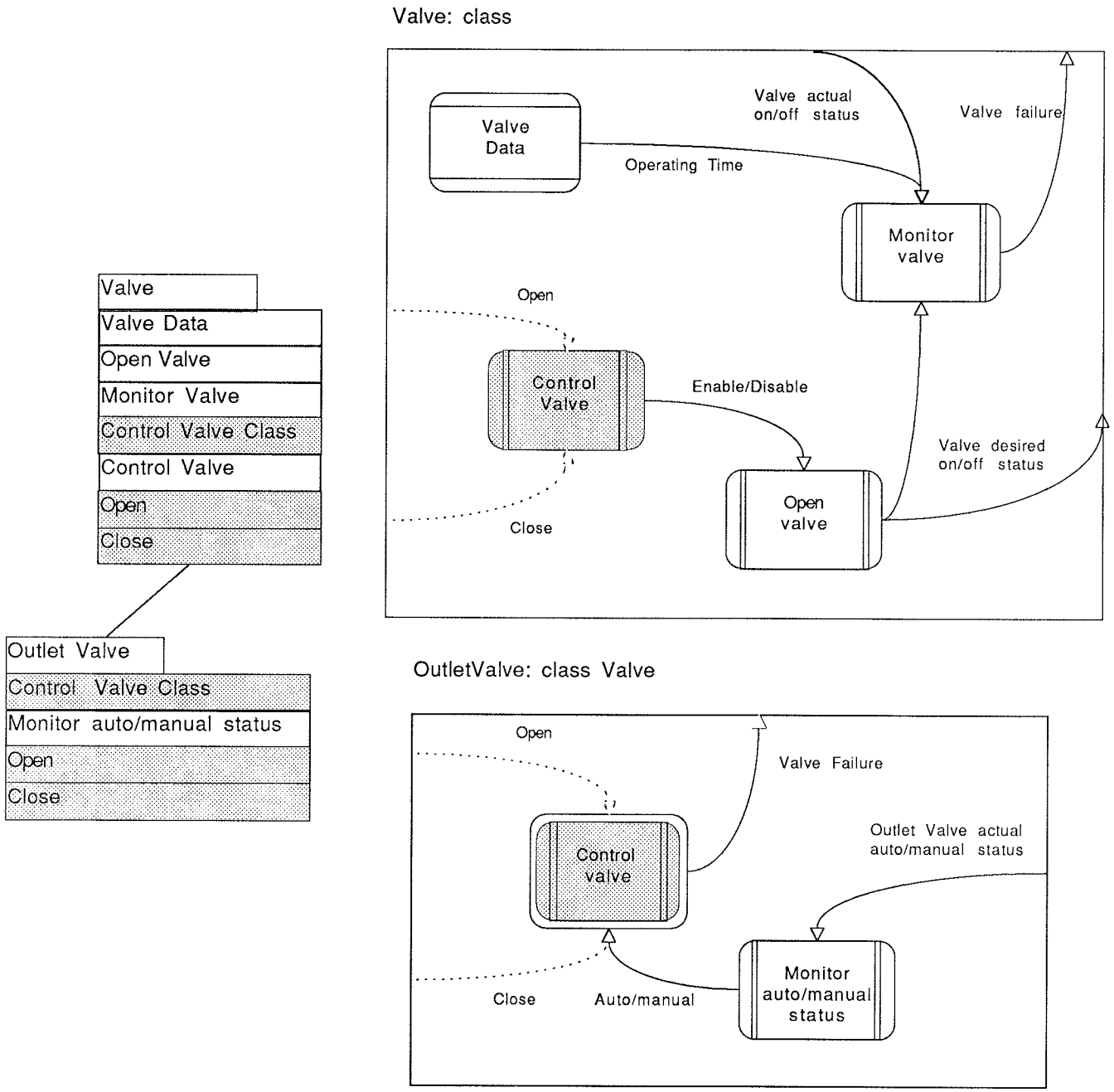


Figure 5.6: Qualified Specialization in OAD

Valve subclass. The modification might be a totally new specification of the control logic or just an extension of it.

Control Valve (the shadowed object) in class Valve is an instance of the virtual class Control Valve Class (the shadowed attribute in the pattern diagram). In the subclass (Outlet Valve) the definition of Control Valve is extended or overwritten. This is indicated by the shadowed and double bordered object as well as the shadowed attribute in the pattern diagram. What it really means, is that the Open and Close procedures are extended or overwritten, i.e. Open and Close are also virtual. This is indicated by dashed arrows. The Control Valve object of class Valve is also presented in the subclass (the shadowed and double bordered object) because it is referred to. Notice that the Control Valve in Outlet Valve is extended with the Auto/manual procedure object and the Valve Failure interaction.

Because the Valve Class is self-contained with control logic, the Inlet Valve object can be directly created as an instance of that class.

The corresponding pattern diagram also shows the simple specialization of the Valve Class into Outlet Valve, as well as the qualified specialization of the Control Valve Class and the procedures Open and Close. But again here the focus is only on the attributes.

Specialization of State Transition Diagrams

In the original example Control Inlet Valve and Control Outlet Valve were control transformations, i.e. they had state transition diagrams (STDs) associated with them. The control objects in the OAD examples can also have STDs associated. There is no problem in having a virtual STD. What it really means, is that some of the state transitions are virtual, i.e. the actions taken, when signals arrive, are specified generally (according to the given state and other conditions like values of variables and continuous input flows). These specifications can then be extended or overwritten in specializations of the STD.

The OSDL proposal [Belsnes 87] introduces among other things virtual procedures and virtual transitions in SDL process diagrams, that are a kind of state transition diagrams. The difference between SDL diagrams and the STDs of real-time SA/SD is basically, that SDL STDs are non-cyclic (provided by duplication of states). This makes it easier to introduce a graphical notation for simple and qualified specialization.

Chapter 6

Mapping from OADs to BETA

The intention behind the OAD notation was partly to contribute to the work on integrating object-orientation with analysis and design, and partly to introduce a notation that is directly mappable to BETA.

The preliminary notation however only expresses the overall structures of the physical model. The notation can be used to express aggregation hierarchies of concurrent, alternation and passive objects. Interaction between objects is expressed by procedure objects, that can be used as data flows between objects, operations performed on another object or signals from one object to another. There are no means for expressing the action sequence of an individual object [1]. Consequently, the corresponding BETA programs can only be templates of the higher abstraction levels. The action sequence for each object must be filled out manually, in order to be able to execute the physical model.

Aggregation hierarchies and classification hierarchies in the OAD notation are directly mappable to corresponding hierarchies in BETA. This is of course not surprising, because the OAD notation is based on the definition of object-oriented programming in chapter 2. This definition constitutes the basic principles underlying the design of BETA. The terminology used in this report is however different than the BETA terminology [2]. Moreover the syntax of BETA, that normally is difficult to grasp for ignorants of the language, has been slightly modified to reflect the concepts of OAD. It has been done by adding and changing keywords

[1] A state transition diagram or another graphical formalism, might however be associated with a control object, as discussed in the introduction.

[2] Concurrent objects, alternation objects, procedure objects and passive objects correspond to system objects, component objects, item objects and data objects in BETA respectively.

```
FlightReservationSystem: @ concurrent
  (#
      FlightType: <<... ObjectDescriptor ...>>;
      anSK451Flight: @ concurrent TimeTable.SK451.Flight;
      TravelAgent: <<... ObjectDescriptor ...>>;
      aTravelAgent: @ concurrent TravelAgent;
      Customer: <<... ObjectDescriptor ...>>;
      aCustomer: @ concurrent Customer;

      ReservationTable: @ passive <<... ObjectDescriptor ...>>;
      TimeTable: @ passive <<... ObjectDescriptor ...>>;
      PriceTable: @ passive <<... ObjectDescriptor ...>>

  do (|| anSK451Flight || aTravelAgent || aCustomer ||)
  #)
```

Figure 6.1: Flight Reservation, Concurrent Objects

[3].

The mapping of aggregation hierarchies is illustrated by means of the flight reservation example of chapter 4, and the mapping of classification hierarchies is illustrated by means of the Valve example of chapter 5.

6.1 Aggregation Hierarchies

The aggregation hierarchies of OADs are easily mapped into corresponding hierarchies in BETA.

Fig. 6.1 shows the BETA program fragment that corresponds to the top-level OAD object diagram of the flight reservation example (Fig. 4.2). FlightReservation is a concurrent object that contains three concurrent objects and three passive objects. The keywords: `concurrent` and `passive` should not be confused with a superclass specification. The imperative

```
do (|| anSK451Flight || aTravelAgent || aCustomer ||)
```

is called a *concurrent imperative*, and is used to specify that the three objects are executed concurrently. The construct `<<... ObjectDescriptor`

[3]The keywords: `concurrent` and `alternation` are used instead of '||' and '|' respectively. The keywords: `virtual` and `extended` are used instead of ':<' and '::<' respectively. The keywords `passive`, `procedure` and `class` are removed when transforming to real BETA.

```
TravelAgent:
  (#
      Identification: @ passive <<... ObjectDescriptor ...>>;
      CustomerService: @ alternation <<... ObjectDescriptor ...>>;
      Invoicing: @ alternation <<... ObjectDescriptor ...>>;
      TourPlanning: @ alternation <<... ObjectDescriptor ...>>

  do (| CustomerService | Invoicing | TourPlanning |)
  #)
```

Figure 6.2: Travel Agent, Alternating Objects

...>> is not part of the BETA syntax, it is rather part of the programming environment for BETA. It is called a *contraction*, and it is used to suppress details. The dots indicate the suppressed details and `ObjectDescriptor` is the syntactic category of the contracted construct. The program fragment is presented in this way in the editor of the Mjølner BETA System. This is further discussed in chapter 7.

Looking into the next level of abstraction the program fragment corresponding to the TravelAgent object diagram in Fig. 4.3 is shown in Fig. 6.2. TravelAgent is a pattern that contains one passive and three alternating objects. The imperative

```
do (| CustomerService | Invoicing | TourPlanning |)
```

is called an *alternation imperative*, and is used to specify that the two objects are executed alternatingly.

As indicated the BETA program templates are deduced from the object diagrams as well as the pattern diagrams. The more details that are specified in the pattern diagrams the more detailed program templates can of course be generated. A non-detailed BETA program template for the passive object TimeTable is shown in Fig. 6.3. The attribute kind of GetInfo (procedure pattern) is derived from the object diagram in Fig. 4.2 (The GetInfo arrow between TimeTable and TravelAgent).

The constructs `<<AttributeDecl>>` [4] and `<<ObjectDescriptor>>` are not part of the BETA syntax. They are placeholders (nonterminals) used in the syntax-directed editor of the Mjølner BETA System.

[4] This is a slight modification of the BETA grammar, but it is only for demonstration purpose.

```
TimeTable: @ passive
  (#
     SK451: <<AttributeDecl>>;
     SK273: <<AttributeDecl>>;
     Initialize: <<AttributeDecl>>;
     GetInfo: @ procedure <<ObjectDescriptor>>

  do <<Imperative>>
  #)
```

Figure 6.3: TimeTable, Program Template

```
TimeTable: @ passive
  (#
     SK451: @ passive FlightType;
     SK273: @ passive FlightType;
     Initialize: @ procedure <<ObjectDescriptor>>;
     GetInfo: @ procedure <<ObjectDescriptor>>

  do <<Imperative>>
  #)
```

Figure 6.4: TimeTable, More Details

A more detailed BETA program template for the passive object TimeTable could be as shown in Fig. 6.4. In the current notation there is no specification of parameters. Not even the direction of the flows can be used to determine the direction of parameters.

6.2 Interaction between Objects

Concerning interaction between objects, the basic principle when mapping OADs to BETA is to associate a BETA procedure pattern with an OAD interaction (procedure call). The template for the alternation object CustomerService is shown in Fig. 6.5. The action part of CustomerService contains an infinite loop and four communication imperatives. The order of the imperatives is arbitrary, it should just be considered as a list of communications that may take place from the CustomerService object. Further details must of course be specified.

```
CustomerService: @ alternation
  (# <<Attributes>>
  do cycle
       (#
       do anSK451Flight >? Reserve;
          anSK451Flight >? Cancel;
          anSK451Flight >? Occupied;
          anSK451Flight >? DisplayReservations;
       #);
  #);
```

Figure 6.5: Customer Service, Synchronized Communication

As mentioned when defining the notation in chapter 4, the procedure object that corresponds to an interaction between two concurrent objects is executed synchronized, like the Ada rendezvous. The imperative

```
anSK451Flight >? Reserve
```

is a *communication request.* It specifies that the CustomerService object is requesting to execute the Reserve procedure object of anSK451Flight. The corresponding *communication accept* in Flight, is shown in Fig. 6.6.

The imperative

```
<? Reserve
```

is a communication accept. The with-imperative is used to specify those concurrent objects, that communication requests will be accepted from. The nonterminal <<SomeProcess>> is indicating that the process that might request a communication is not specified yet.

Notice that the request-imperatives and the corresponding procedure patterns are specified in the alternation object CustomerService, that is local to the concurrent object TravelAgent. CustomerService could also have been a concurrent object. In BETA a concurrent object can communicate with internal concurrent objects of another concurrent object.

In further details the interaction Reserve could look like in Fig. 6.7 and the corresponding communication request could look like:

```
(5, False, Business) -> anSK451Flight >? Reserve -> theResult
```

```
Flight:
  (# Seats: [NoOfSeats] @ Seat;
     Seat: <<ObjectDescriptor>>
     ActualDepartureTime,
     ActualArrivalTime: @ TimeOfDay;
     ActualFlyingTime: @ TimePeriod;

     DepartureDelay: @ procedure
       (# <<Enter>> with <<SomeProcess>> <<ActionPart>> <<Exit>> #);
     Reserve: @ procedure
       (# <<Enter>> with TravelAgent <<ActionPart>> <<Exit>> #);
     Cancel: @ procedure
       (# <<Enter>> with TravelAgent <<ActionPart>> <<Exit>> #);
     Occupied: @ procedure
       (# <<Enter>> with TravelAgent <<ActionPart>> <<Exit>> #);
     DisplayReservations: @ procedure
       (# <<Enter>> with TravelAgent <<ActionPart>> <<Exit>> #);

  do cycle
       (#
       do <? Reserve;
          <? Cancel;
          <? Occupied;
          <? DisplayReservations;
          ReservationTable.UpdateEntry;
       #)
  #)
```

Figure 6.6: Flight, Synchronized Communication

```
Flight:
  (# ...
     Reserve: procedure
       (#
       enter (NoOfSeats, Smoking, SomeClass)
       with TravelAgent
       do ...
       exit Result
       #);
    ...
  do cycle
       (#
       do ...
          <? Reserve;
          ...
       #)
  #)
```

Figure 6.7: Reserve, Further Details

Notice that a synchronized procedure pattern can be fully parameterized. Synchronization takes place when parameters are transferred between the sender and the receiver.

The interaction with the passive object ReservationTable is indicated by means of a procedure call, that corresponds to the interaction. As mentioned the sequence and the parameters can not be deduced from the OAD.

The definition of the attributes Seats, ActualDepartureTime, ActualArrivalTime, and ActualFlyingTime are derived from the more detailed pattern diagram in Fig. B.2.

Mutual Exclusion

In general mutual exclusion is provided in BETA by the aggregation hierarchy. An internal concurrent object may execute procedure objects belonging to an enclosing concurrent object. Such procedure objects are executed one at a time.

6.3 Classification Hierarchies

The classification hierarchies in OADs are easily mapped into corresponding classification hierarchies in BETA. As described in chapter 2, BETA provides simple specialization as well as qualified specialization for procedures as well as classes.

There is still no specification of action sequences of individual objects. The sequences of imperatives should just be considerated as lists of communications that must be put in the right context.

Fig. 6.8, 6.9 and 6.10 show the mapping of class Valve, class Outlet Valve and class Inlet Valve (respectively) from Fig. 5.5. The nonterminal `SomeProcess` indicates, that the receiver of the communication request cannot be deduced from the class diagrams. If the class is used in a diagram (an object is instantiated from it), the receiver will probably be specified.

Fig. 6.11 and 6.12 show the mapping of class Valve and class Outlet Valve (resp.) from Fig. 5.6. In class Valve, ControlValve is a concurrent object, that is an instance of the virtual class ControlValveClass. Class Valve has two virtual procedures Open and Close. In the subclass Outlet-Valve, a new ControlValve object is not instantiated, but the definition of class ControlValve is extended. The Open and Close procedures are also extended. The keyword `extended` is used, because qualified specialization in BETA is considered as an extension of a more general concept as opposed to a redefinition (like in most other object-oriented programming languages). It is however also possible to program a redefinition.

The details of the ControlValve class is not shown, but in the actual implementation this class may also introduce two virtual procedures Open and Close, that are defined in the virtual definition of the ControlValve Class and further bound in the further binding of the ControlValve. In BETA it is very typical to several simultaneous layers of virtuals.

An interesting problem is how to handle interaction between the action parts of the super- and subclass. The INNER construct directs control to a possible subclass, but how do the two loops work together? In some cases the "inner" action part could be a single (possibly empty) sequence of actions, that are part of the "outer" cycle. In other cases a totally new cycle of actions is required. More generally the problem is how to handle

```
Valve: class
  (#
      ValveData: @ passive
        (# OperatingTime: @ procedure <<ObjectDescriptor>>
        do <<Imperative>>
        #);
      OpenValve: @ concurrent
        (# Enable: @ procedure <<ObjectDescriptor>>;
           Disable: @ procedure <<ObjectDescriptor>>
        do cycle
             (#
             do <? Enable
                <? Disable
                MonitorValve >? ValveDesiredOnOffStatus;
                <<SomeProcess>> >? ValveDesiredOnOffStatus;
             #);
        #);
      MonitorValve: @ concurrent
        (# ValveDesiredOnOffStatus: @ procedure
             (# <<Enter>> with OpenValve <<ActionPart>> <<Exit>> #);
        do cycle
             (#
             do <? ValveDesiredOnOffStatus;
                <<SomeProcess>> >? ValveActualOnOffStatus;
                <<SomeProcess>> >? ValveFailure;
                ValveData.OperatingTime;
             #);
        #);
  do <<Imperative>>
  #)
```

Figure 6.8: Class Valve

```
OutletValve: class Valve
  (#
     ControlOutletValve: @ concurrent <<ObjectDescriptor>>;
     MonitorAutoManualStatus: @ concurrent
       (# AutoManual: @ procedure
            (# <<Enter>> with ControlOutletValve <<ActionPart>> <<Exit>> #);
       do cycle
            (#
            do <<SomeProcess>> >? OutletValveActualAutoManualStatus;
               <? AutoManual;
            #);
       #);
     Open: @ procedure <<ObjectDescriptor>>;
     Close: @ procedure <<ObjectDescriptor>>;
  do cycle
       (#
       do <? Open;
          <? Close;
          MonitorAutoManualStatus >? AutoManual;
          OpenValve>? Enable;
          OpenValve>? Disable;
          <<SomeProcess>> >? ValveFailure;
       #);
  #)
```

Figure 6.9: Subclass OutletValve

```
InletValve: class Valve
  (#
     ControlInletValve: @ concurrent <<ObjectDescriptor>>;
     Open: @ procedure <<ObjectDescriptor>>;
     Close: @ procedure <<ObjectDescriptor>>
  do cycle
       (#
       do <? Open;
          <? Close;
          OpenValve>? Enable;
          OpenValve>? Disable;
       #);
  #)
```

Figure 6.10: Subclass InletValve

inheritance together with concurrent interacting objects.

6.4 Specifying Action Sequences

The specifications in the Jackson System Development method [Jackson 83] are in principle executable because the action sequence of each process is specified by means of a structure diagram and structure text. The structure diagram is a graphical representation of control flow, it expresses sequence, selection and repetition. Structure text is a textual version of the structure diagram extended with constructs for inputs and outputs to and from the process.

In Real-time SA/SD the transformation schema is (at least partly) executable, because the control transformations are specified by means of state transition diagrams. Data transformations are however not specified graphically. Transformation schemas can be executed in a 'Petri net'-like manner, by means of placement of tokens [Ward 86].

State transition diagrams could also be associated with concurrent objects in OADs. In appendix C it is illustrated how a transformation schema with one control transformation and two data transformations can be mapped into BETA. The same principles can be used for other object-oriented notations that include some kind of state transition diagrams.

6.5 Mapping Other Notations to BETA

Mapping Real-time SA/SD to BETA

Besides the state transition diagram Real-time SA/SD extends traditional SA with asynchronous signals and continuous flows. These constructs can also be modelled in BETA.

Modelling asynchronous signals by means of synchronized communication can be done by making a special concurrent object, a signal handler. The signal handler is always ready to accept a communication request, and it queues the signals until the receivers are interested in them. The predefined signals (trigger, enable, disable, suspend and resume) can also be modelled by prefixing each concurrent object with a superclass that

```
Valve: class
  (#
     ValveData: @ passive
       (# OperatingTime: @ procedure <<ObjectDescriptor>>
       do <<Imperative>>
       #);
     OpenValve: @ concurrent
       (# Enable: @ procedure <<ObjectDescriptor>>;
          Disable: @ procedure <<ObjectDescriptor>>;
        do cycle
             (#
             do <? Enable;
                <? Disable;
                MonitorValve >? ValveDesiredOnOffStatus;
                <<SomeProcess>> >? ValveDesiredOnOffStatus;
             #);
       #);
     MonitorValve: @ concurrent
       (# ValveDesiredOnOffStatus: @ procedure
             (# <<Enter>> with OpenValve <<ActionPart>> <<Exit>> #);
       do cycle
             (#
             do <? ValveDesiredOnOffStatus;
                <<SomeProcess>> >? ValveActualOnOffStatus;
                <<SomeProcess>> >? ValveFailure;
                ValveData.OperatingTime;
             #);
       #);
     ControlValveClass: virtual <<ObjectDescriptor>>;
     ControlValve: @ concurrent ControlValveClass;
     Open: virtual procedure <<ObjectDescriptor>>;
     Close: virtual procedure <<ObjectDescriptor>>;
  do cycle
       (#
       do <? Open;
          <? Close;
          OpenValve>? Enable;
          OpenValve>? Disable;
       #);
  #)
```

Figure 6.11: Class Valve

```
OutletValve: class Valve
  (#
     MonitorAutoManualStatus: @ concurrent
       (# AutoManual: @ procedure
            (# <<Enter>> with ControlValve <<ActionPart>> <<Exit>> #);
       do cycle
            (#
            do <? OutletValveActualAutoManualStatus;
               ControlValve >? AutoManual;
            #);
       #);
     ControlValveClass: extended <<ObjectDescriptor>>;
     Open: extended procedure <<ObjectDescriptor>>;
     Close: extended procedure <<ObjectDescriptor>>;
  do cycle
       (#
       do <? Open;
          <? Close;
          OpenValve>? Enable;
          OpenValve>? Disable;
          MonitorAutoManualStatus >? AutoManual;
          <<SomeProcess>> >? ValveFailure;
          INNER
       #);
  #)
```

Figure 6.12: Subclass OutletValve

provides these operations. A kernel-like object can then be used to control the concurrent objects.

Continuous data flows can be modelled by allocating a concurrent object that periodically updates a variable, that is accessed by a procedure object.

Mapping OSDL Process Diagrams to BETA

In OSDL [Belsnes 87], as well as SDL, each process is specified by means of a graphical, as well as textual, state transition diagram. OSDL process diagrams can be mapped to BETA following the same principles as described in appendix C. The virtual procedure and virtual transitions of OSDL are easily supported by BETA. The virtual transition is modelled by the virtual procedure of BETA.

Chapter 7

Automated Support

Computer support in software development has been a research area for a long period of time. In the beginning the primary focus was on providing tools for what traditionally is called programming. Integrated and language-oriented programming environments that support creation, modification and debugging of programs have emerged. Lately there has also been a lot of effort in supporting other activities than traditional programming in systems development. Tools have been developed that support creation and modification of graphical analysis and design descriptions. This has introduced the popular term CASE (Computer Aided Software Engineering). Apparently this term covers computer support for all kind of system development activities.

The previous chapter has hopefully shown, that object-orientation as a bridge between analysis, design and implementation is not a castle in the air. This chapter will show how the mappings between OADs and BETA program fragments can be supported by the programming environment of BETA: the Mjølner BETA System.

The Mjølner BETA System [1] is a programming environment for object-oriented programming. The environment consists of an implementation of BETA and a set of grammar based tools, that can be generated for any language with a context free grammar. These tools are: an integrated text and structure editor [Borup 88], a metaprogramming system [Madsen 88a] and a fragment system. In addition the environment contains a user interface toolkit (build on top of the X Window System and the Macintosh Toolbox) and a UNIX ensemble that gives access to the

[1] The Mjølner BETA system [Knudsen 89] is one of several results of the Nordic Mjølner project [Dahle 87].

UNIX system.

The editor, the metaprogramming system and of course the compiler are heavily used in the integration of OADs and BETA programs. The metaprogramming system makes it possible to manipulate programs as data. A programming interface to abstract syntax trees (ASTs) can be generated from a context free grammar of any language. The editor and the compiler are built on top of the metaprogramming system, because they both manipulate ASTs.

In [Sandvad 88] it is shown how the syntax-directed textual editor can be extended also to provide syntax-directed graphical editing, i.e. creation and modification of graphical diagrams according to a context free grammar of the graphical language. The design is especially useful for graphical languages that have a high degree of text associated with them. This is the case with OSDL, that has been used as an example language in that proposal. The grammar basis ensures that the graphical diagrams always are syntactically correct. Error prevention is very useful if the graphical language is complex or it has a high degree of text associated with it. This is not the case with the OAD language. A grammar basis can however provide a lot of support in integrating OADs with BETA program fragments. The grammar basis can give:

- syntax-directed editing of OADs and BETA programs
- automatical creation of BETA programs from OADs (code generation)
- automatical creation of OADs from BETA programs (reverse engineering)
- traceability and maintenance of consistency between OADs and BETA programs
- traceability to other kind of descriptions e.g. documentation
- abstract presentation of BETA programs according to the abstraction levels in the corresponding OADs
- integration with the BETA compiler
- execution of the BETA program after completion of programming details like the action sequences of objects

7.1 The User Interface

The user interface to the syntax-directed textual editor will not be presented here. It is presented in [Borup 88].

The graphical editor follows the principles presented in [Sandvad 88]. The OAD symbols are presented in a palette, from where the user can drag them into the diagrams: either the object diagrams or the pattern diagrams. When creating, modifying and browsing the diagrams, the user typically alternates between manipulating the object diagram and the pattern diagram.

The syntax-directed editing techniques imply that the OADs are manipulated according to their logical structure, as opposed to the plain "collection" of graphical symbols. Another consequence is that whole subtrees of the abstraction hierarchies typically are manipulated in one operation. E.g. when cutting an object symbol out of an object diagram, the whole underlying decomposition of that object (if any) is also cut out. Although "standard" structural cut, copy and paste operations on OADs will do the work, some specialized operations might be required to ease the work. E.g. operations for moving collections of objects or patterns up and down in the abstraction hierarchies.

Editing of interactions typically involves the object diagram as well as the corresponding pattern diagram. If the interaction symbol is selected, the mouse must be dragged from the source object to the destination object. If the destination object is in another diagram for example in an outer diagram or in a superclass, the outer symbol or inherited symbol (resp.) is automatically duplicated in the diagram that contains the source object. The distinction between source and destination is only a matter of specifying the direction of the arrow, but the arrow can also be bidirectional. An attribute in the corresponding class pattern of one of the two objects must be selected or created. This attribute will then be the procedure pattern that is instantiated in the interaction. The creation of an interaction thus involves determination of ownership of the corresponding procedure, but this might be relaxed upon in the early informal stages of analysis.

The syntax-directed textual editor is used to enter and modify the although sparse text.

If an object is selected, its definition can be specified in another diagram or the user can decide to switch to the syntax-directed BETA editor to fill out the details of an object.

Navigation is provided in several ways. The abstraction hierarchies in the OADs can be browsed. Semantic links can be followed, for example if an outer object symbol or an inhereted object is selected, links to the outer diagram or the superclass diagram (resp.) can be followed. Links can be followed to the corresponding constructs in the associated BETA program, and vice versa. Finally links can be created and followed to supplementary descriptions or documentation in general. In traditional SA and real-time SA/SD, textual descriptions are often associated with nodes in the DFDs, for example mini-specs, process descriptions, pseudocode, pre and post conditions etc.

7.2 OADs as Graphical Presentation of BETA Programs

The basis for automating the mapping from OADs to BETA is to extend the BETA editor with a graphical prettyprinter, that presents the higher abstraction levels of BETA programs as OADs. The idea is that the underlying abstract syntax tree (BETA AST) can be manipulated through a textual as well as a graphical interface. The textual interface is the usual integrated text and stucture editor, that manipulates BETA programs at all abstraction levels. The graphical interface is a diagram editor that works corresponding to the underlying BETA grammar, but only at the higher levels of abstraction, namely those levels that have an OAD presentation. A more detailed description of how to support syntax-directed graphical editing in general is given in [Sandvad 88]. The basic idea of the two interfaces to the BETA AST is illustrated in Fig. 7.1.

Editing in one editor may of course affect the other editor because both presentations have the same internal representation. Due to the different nature of the two languages there is not symmetry in the amount of work that the editors impose on each other. There will be many creations and modifications in the text editor, that will not affect the graphical editor, because they are at a detailed level. Conversely, practically every creation and modification in the graphical editor will affect the text editor, because

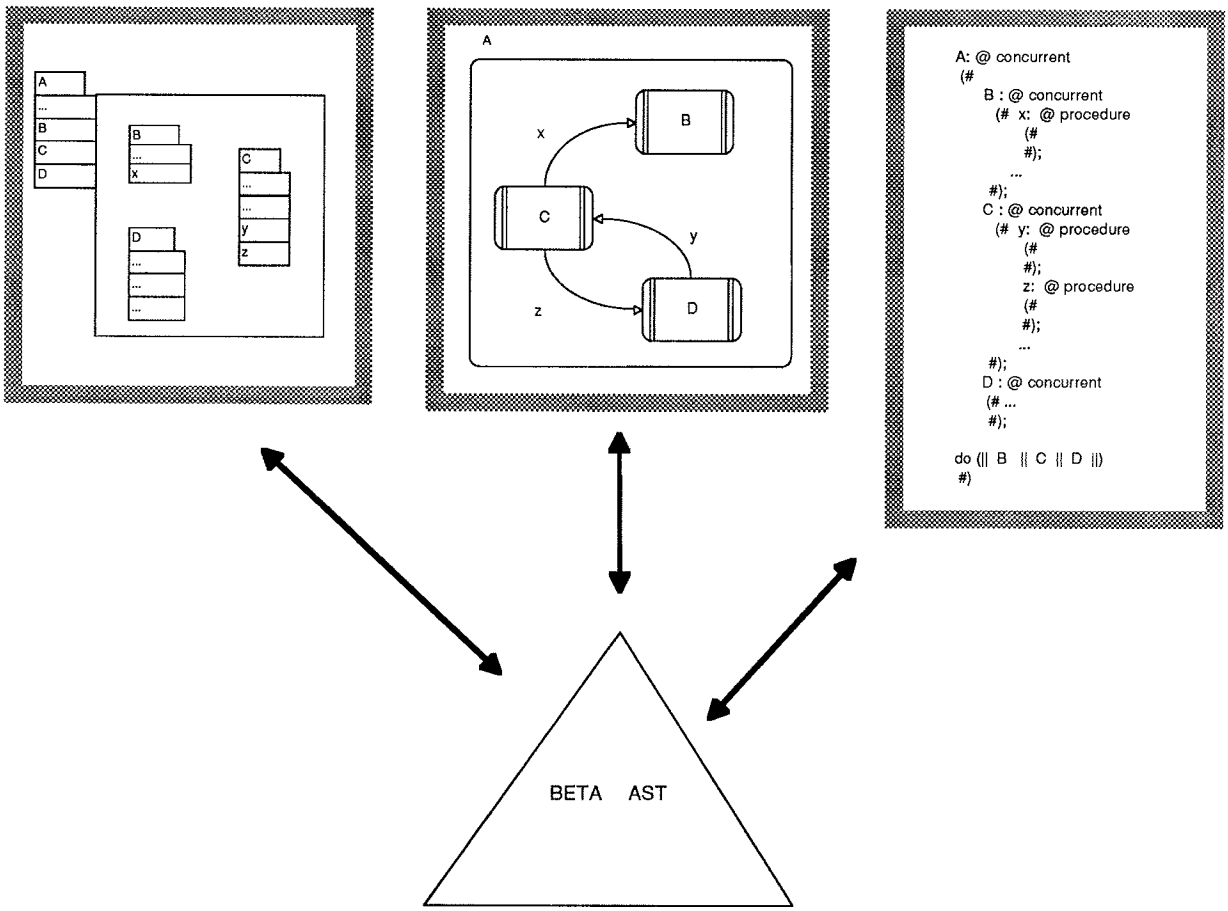


Figure 7.1: Integrated Editing of OADs and BETA Fragments

the OADs express the overall structures of the model.

The textual prettyprinter is (of course) based on the BETA grammar. For each production in the grammar it is specified how to present it on textual form. The graphical prettyprinter is however more complicated. The graphical presentation is not merely a mapping from the (context free) structures of the AST to corresponding graphical elements. The interactions (procedure calls) reflect (static) semantic information. It must therefore be possible to perform semantic analysis on the ASTs, in order to do the graphical prettyprinting. This is provided by the AST interface of the metaprogramming system. It is possible to operate on ASTs at three levels: the tree level that considers ASTs as simple trees with sons and siblings etc; the context free level that operate on ASTs in terms of the language specific constructs and finally the semantic level that makes it possible to create and maintain semantic information in the ASTs. The BETA compiler is an example of a tool, that operates through the semantic level of the AST interface to BETA ASTs. When using the semantic level of the AST interface, it is for example possible to deduce whether creation of a construct in one of the editors imply creation of a similar construct in the other editor.

7.3 Document Generation

The framework sketched in the previous section deals with simultaneous creation and modification of OADs in the graphical editor and BETA program fragments in the text editor. But it is also possible to generate documents in a more batch-oriented manner.

The user may then decide only to use the OAD editor to generate some preliminary sketches of a system. When these diagrams are acceptable the user may decide to generate BETA program templates from them.

Conversely, if existing BETA programs, that have not been created together with OADs, are going to be maintained or documented, it must be possible to generate OADs from them. This is often called reverse engineering. Another typical situation might be, that the analysis and design documents have been "turned off" for a period of time while concentrating on implementing the system. When the system has reached a stable state, the analysis and design documents must be updated according to

the system.

Both cases can be supported. It is just a matter of "turning on or off" the textual presentation or the graphical presentation. Textual prettyprinting of the whole BETA AST is no problem, but the generation of the graphical diagrams may not satisfy the aestetic sense of the user. The user must be able to adjust the layout.

7.4 Traceability

The ability to maintain consistency between analysis and design documents and programs is very valuable. The documentation of a system can easily become obsolete because the system evolves over time. Current CASE tools lack this ability.

The integration of OADs and BETA programs by being two alternative presentations of the same underlying abstract syntax tree also supports navigation. If an object is selected in the OAD editor for example, the user can follow the link to the corresponding object in a BETA program fragment, and a text editor may be opened on this fragment (if not already opened).

In general, several kinds of links are provided between documents in the programming environment. An experimental version of the environment, the hyperobject system [Sandvad 89], has been developed in order to extend and generalize the hypertext capabilities in the environment. The hyperobject system is intended to support links between any combination of text and structure. Programs as well as documentation may be represented as text or as ASTs, i.e. the syntax-directed textual editor can be used not only on programs but also on documentation. This provides links between and within program fragments; between and within documentation fragments; and between program fragments and documentation fragments. Because graphical notations can be created and modified structurally by extending the syntax-directed editor, the general linking capability is also provided between graphical descriptions and between textual and graphical descriptions.

This can be used in the integrated OAD and BETA environment to support documentation and navigation in the different kinds of related doc-

uments. The variety of relationships between BETA programs, OAD and other kind of documentation may be supported.

7.5 Abstract Presentation

In the examples used when illustrating the mappings between OADs and BETA programs, abstract presentation was used. Abstract presentation has turned out to be a very useful facility in the syntax-directed textual editor. It is used to create an overview of large program fragments, and to browse through programs by detailing selected abstractions. It is also used to provide parts of technical documentation, because snapshots of a program at different abstractions levels together with comments are a good supplement to the documentation.

In the integration between OADs and BETA, abstract presentation can be used to visualize the correspondance between the abstraction levels in the OADs (the diagrams) and the abstraction levels of the BETA programs.

7.6 Integration with the BETA Compiler

In the Mjølner BETA System the editor and the BETA compiler are integrated in two ways. The editor and the BETA compiler work on the same AST, and the compiler can be activated from the editor. The editor ensures that a BETA program is correct according to the context free grammar. The static semantic checks are performed by the compiler. If there are semantic errors, the compiler inserts information in the ASTs. This information is provided to the user by the editor. If there are no semantic errors, code is generated from the AST and the program can be activated from the editor.

Due to the integration of the editor and the compiler, OADs can be executed/simulated. When an OAD hierarchy has been created together with the corresponding BETA templates, the BETA editor is used to fill out necessary details like the action sequences of the objects. Then the BETA program is compiled and executed.

This framework can be used to support the experimental (prototyping) style. The programmer can decide only to fill out some of the details of

the objects, inorder to examine the behavior of the overall system before supplying more details. The result of executing the experimental version of the program may be that the overall model should be modified in the OADs.

7.7 Status

The integration of OADs and BETA as described above is at the design stage. But it has been illustrated how the implementation can be done, by reusing and extending the Mjølner BETA System.

Chapter 8

Conclusion

The message of this report is that object-orientation integrates the traditional activities analysis, design and implementation. The purpose of analysis as well as object-oriented programming is to build a model of part of the real world in terms of phenomena and concepts in the application area. The difference between the two models is that the object-oriented program can be executed, which makes it possible to simulate that part of the world that we wish to model. A valuable quality of most analysis models is however, that they are partly graphical. In this report the valuable qualities of both models are combined.

Object-oriented development integrates analysis, design and implementation because the same representation can be used in all three activities. In the transformation from the analysis model to the design model and from the design model to implementation it is not necessary to change the representation, but there may of course exist several versions of the models. However it is the same logical model that gradually is extended from a model in terms of application domain phenomena and concepts to a model that includes implementation domain phenomena and concepts. The design and implementation activities may of course affect the analysis model. There is no clear distinction between analysis and design. Analysis starts informally and the analysis model is gradually precisized. At some time in the development the primary focus moves from analysis to design. Instead of explicitly defining the border between analysis and design, the border can be characterized by the degree of formalization.

A preliminary notation for object-oriented analysis and design diagrams has been proposed, in order to illustrate the integration. An object-oriented CASE tool that supports OADs has been proposed. It has

been shown, how OADs easily can be mapped into BETA programs and how this mapping (and its inverse!) can be supported by the computer. Another reason for introducing the notation has been to improve the specification of abstraction in analysis and design diagrams, especially generalization and specialization.

The experimental (prototyping) style is to some extent supported by generating program templates from design diagrams. These program templates can gradually be detailed to a complete program, but the overall behavior of the program can be observed at an early stage. The OADs may be supplemented by user interface support. E.g. the objects might have windows, menus and dialog boxes associated with them. The menus containing the available operations on the objects. Conversely, user interface prototyping can be used as a basis for the creation of the OADs.

The integration of programs, analysis and design diagrams and other kind of documentation in this proposal, makes it possible to maintain consistency between documentation and programs. This is an important quality, because maintenance is a major problem in system development projects. Analysis and design documents are used as a starting point for implementation. When the implementation has reached a stable state, it is very likely, that the documentation has become obsolete and must be updated. The same problem arises when the programs are maintained. Because OADs are considered as a high-level graphical presentation of BETA programs, this part of the documentation is always consistent with the programs (reverse engineering). Other relationships between documentation and programs are supported by hypertext-like links.

The OAD notation is a description tool, not a technique (not to mention a methodology), but the object-oriented perspective as described in chapter 2 is the underlying philosophy. The creation of the analysis and design models involves:

- Identification of phenomena with substance, measurable properties and transformations on substance
- Identification of the aggregation hierarchies of objects
- Identification of interaction between objects
- Identification of sequencing between objects
- Identification of concepts that classify phenomena
- The concepts may be organized in classification and aggregation hierarchies

The graphical notation can help in expressing the physical model, but a specific order can not be prescribed. The developer may start by drawing OADs or by writing BETA programs, or both notations may be used simultaneously or alternatingly.

In the future development of the OAD notation two topics will be further explored: specification of action sequences of individual objects and specification of dynamically created objects.

Acknowledgement

I wish to thank Merete Bartholdy, Britta Shaw Bjerregaard, Merete Hess, Gitte Hjorth, Anette Hviid, Jørgen Lindskov Knudsen, Morten Kyng, Ole Lehrmann Madsen, Preben Thalund Madsen, Ivan Aaen and the participants in the graduate courses "Programmeringsprogsseminar 89" and "Design by Doing 89" for many helpful comments on this report. This work has been supported by *The Danish Natural Science Research Council*, FTU grant no. 5.17.5.1.25 and the Mjølner project, that has been partly funded by a grant from *The Nordic fund for Technology and Industrial Development.*

Bibliography

[Andersen 86] N.E. Andersen, F. Kensing, M. Larsen, J. Lundin, L. Mathiassen, A. Munk-Madsen, P. Sørgaard. Professionel Systemudvikling. Teknisk Forlag, København, 1986.

[Bailin 89] S.C. Bailin: An Object-Oriented Requirements Specification Method. Comm. ACM, vol. 32, no. 5, p. 608, May 1989.

[Belsnes 87] D. Belsnes, B. Møller-Pedersen, H. P. Dahle: Rationale and Tutorial on OSDL an Object-oriented Extension of SDL, In SDL '87: R. Saracco and P.A.J Tilanus (eds.), State of the Art and Future Trends, Elsevier Science Publishers B.V. (North-Holland), 1987.

[Berthelsen 86] S. Berthelsen, S. Hvidbjerg, P. Sørensen: Graphical Programming Environments Applied to Beta. Computer Science Department, Aarhus University, DAIMI IR-64, Sept. 1986

[BETA 87] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: The BETA Programming Language. In: B.D. Shriver, P. Wegner (eds.): Research Directions in Object Oriented Programming, MIT Press, 1987.

[BETA 90] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: Object Oriented Programming in the BETA Programming Language. Book in preparation, 1990.

[Booch 86] G. Booch: Object-Oriented Development. IEEE Trans. Software Eng., vol. SE-12, no. 2, p. 211, Feb. 1986.

[Borup 88] K. Borup, E. Sandvad: Users and Programmers Guide for Sif - A Syntax-Directed Editor. Project Mjølner Working Note DK-SYS-29.2, December 1988.

[Bruyn 87] W. Bruyn, R. Jensen, D. Keskar, P. Ward: ESML: An Extended System Modeling Language Based on the Data Flow Diagram.

Proceedings, Twelfth Structured Methods Conference, Chicago, Aug. 1987.

[1] CASE Industry Directory: An Introduction to CASE. CASE Consulting Group, 224 S. W. First Avenue, Portland Oregon 77204, 1988.

[Chikofsky 90] E.J.Chikofsky, J.H. Cross II: Reverse Engineering and Design Recovery: A Taxanomy. IEEE Software, p. 13, Jan. 1990.

[Coad 89] P.Coad, E.Yourdon: Object-Oriented Analysis. Prentice Hall, Englewood Cliffs, New Jersey, 1989.

[DeMarco 78] T.DeMarco: Structured Analysis and System Specification. Yourdon Press/Prentice Hall, 1978.

[Dahle 87] H.P. Dahle, M. Løfgren, O.L. Madsen, B. Magnusson (eds): The Mjølner Project. In: Proceedings of EUROSOFT '87, London, June 1987.

[Harel 87] D. Harel: State Charts: A Visual Formalism for Complex Systems. Science of Computer Programming 8, North-Holland, Amsterdam,1987, pp. 231-274.

[Jackson 83] M.A. Jackson: System Development. Prentice-Hall, Englewood Cliffs, N.J., 1983.

[Knudsen 89] J.L. Knudsen, O.L. Madsen, C. Nørgaard, L.B. Petersen, E. Sandvad: An Overview of the MjølnerBETA System. Mjølner Informatics ApS, Science Park Aarhus, Nov. 1989.

[Kristensen 87] B.B. Kristensen, O.L. Madsen, B. Møller-Pedersen, K. Nygaard: Classification of Actions or Inheritance also for Methods. In: Proceedings of ECOOP'87, Paris, June 1987.

[Madsen 88a] O.L. Madsen, C. Nørgaard: An Object-Oriented Metaprogramming System. In: Proceedings of Hawaii International Conference on System Sciences - 21, Jan. 1988.

[Madsen 88b] O.L. Madsen, B. Møller-Pedersen: What object-oriented programming may be - and what it does not have to be. In: S. Gjessing, K. Nygaard (eds.): ECOOP'88, Springer Verlag, Aug. 1988.

[Madsen 89] O.L. Madsen, B. Møller-Pedersen: Virtual Classes - A powerful mechanism in object-oriented programming. In: Proceedings of OOPSLA'89, New Orleans, 1989.

[Meyer 87] B. Meyer: Reusability: The Case for Object-Oriented Design. IEEE Trans. Software Eng, p. 50, Mar. 1987.

[Müller 89] G. Müller, A.-K. Pröfrock: Four Steps and a Rest in Putting an Object-Oriented Programming Environment to Practical Use. In: Steve Cook (ed.): ECOOP'89, Cambridge, July 1989.

[Nørgaard 89] C. Nørgaard, E. Sandvad: Reusability and Tailorability in the Mjølner BETA System. In: TOOLS'89: Technology of Object-Oriented Languages and Systems, Paris, Nov. 1989.

[Peterson 77] J.L. Peterson: Petri nets. Comput. Surveys, vol. 9, no. 3, Sept. 1977.

[Pun 89] W.W.Y. Pun, R.L Winder: A Design Method For Object-Oriented Programming. In: Steve Cook (ed.): ECOOP'89, Cambridge, July 1989.

[Rockström 85] A. Rockström: An Introduction to the CCITT SDL, Televerket, Sweden, 1985.

[Sandvad 88] E. Sandvad: Syntax-Directed Graphical Editing. Computer Science Department, Aarhus University, Draft, June, 1988.

[Sandvad 89] E. Sandvad: Hypertext in an Object-Oriented Programming Environment. In: J. Andre, J. Bezivin (eds.): WOODMAN'89: Workshop on Object-Oriented Document Manipulation, Rennes, France, BIGRE, May 1989.

[SDL] Recommendation Z.100 SDL Specification description language, CCITT.

[Seidewitz 87] E. Seidewitz, M. Stark: Towards a General Object-Oriented Software Development Methodology. ACM SIGAda Ada Letters, vol. 7, no. 4, p. 54, July/Aug. 1987.

[Shlaer 89] S. Shlaer, S.J.Mellor: An Object-Oriented Approach to Domain Analysis. ACM SIGSoft, Software Engineering Notes, vol. 14, no. 5, p. 66, July 1989.

[Simula67] O.J. Dahl, B.Myrhaug, K. Nygaard: SIMULA 67 Common Base Language, Norwegian Computing Center, Oslo, 1984.

[Smalltalk] A. Goldberg. D. Robson: Smalltalk 80: The Language and its Implementation. Addison Wesley 1983.

[Ward 85] P.T Ward, S.J. Mellor: Structured Development for Real-Time Systems. vol. 1-3, Yourdon Press, 1985.

[Ward 86] P.T. Ward: The Transformation Schema: An Extension of the Data Flow Diagram to Represent Control and Timing. IEEE Trans. Software Eng., vol. SE-12, no. 2, p. 198, Feb. 1986.

[Ward 89] P.T Ward: How to Integrate Object Orientation with Structured Analysis and Design. IEEE Software, p. 74, Mar. 1989.

[Wasserman 87] A.I. Wasserman, P.A. Pircher: A Graphical, Extensible Integrated Environment for Software Development. Proceedings, 2nd Symposium on Practical Software Development Environments. ACM SIGPLAN Notices, vol. 22, no. 1, pp. 131-142, Jan. 1987.

[Wasserman 89] A.I. Wasserman, P.A. Pircher, R.J Muller: An Object-Oriented Structured Design Method for Code Generation. ACM SIGSoft Software Eng. Notes, vol. 14, no. 1, p. 32, Jan. 1989.

[Yourdon] E. Yourdon: Managing the System Life Cycle. Yourdon Press.

Appendix A

Object-Orientation in Analysis and Design

This appendix gives a brief survey of some proposals for introducing object-orientation into analysis and design. All the chosen proposals except [Coad 89] have some relation to Structured Analysis (SA) and/or Structured Design (SD). The focus is on how the the different methods support object-orientation in the graphical notations, i.e how the notations support:

- modelling of phenomena (objects)
- interaction between phenomena
- sequencing between phenomena (concurrency)
- modelling of concepts and abstraction
  - classification of phenomena into concepts
  - aggregation hierarchies of phenomena and concepts
  - generalization hierarchies

A.1 Booch

Object-Oriented Development (OOD) [Booch 86] is a partial life-cycle software development method, that focuses on design and implementation. The relationship to analysis is however emphasized: "It is necessary to couple object-oriented development with appropriate requirements and

analysis methods in order to help create our model of reality". Jacksons Structured Development [Jackson 83] is mentioned as being "a promising match". The most important difference between OOD and functional development methods like SA/SD, is that the decomposition of a system is based on the concept of an object, rather than the concept of a function. The major steps of OOD is:

1. identify the objects and their attributes
2. identify the operations suffered by and required of each object
3. establish the visibility of each object in relation to other objects
4. establish the interface of an object
5. implement each object

Step 3 and 4 result in graphical diagrams that are called *object diagrams.* See Fig. A.1 (Booch Fig. 6, 7, 9 and 10).

An object diagram is a directed graph, where each node constitutes an object and each edge indicates the dependencies between objects. Special graphical symbols exist that indicate the type of an object: task, package, generic package, subprogram or generic subprogram. Step 5 is then a straight forward transformation of the object diagrams and the descriptions from step 1 and 2 into Ada program templates.

OOD models phenomena by regarding objects as instances of abstract datatypes. Concerning modelling of concepts OOD is weaker.

Classification of objects in types (classes) is supported.

Aggregation is not expressed in more than one level, objects/classes do not have local objects/classes. The only graphical structuring mechanism is the *subsystem*, that is used to group objects into layers; where each layer denotes a collection of objects having restricted visibility to other layers.

Generalization is supported in a very limited way by means of generic packages, parameterized by constants and types. There are only two levels in the classification hierarchy. Booch discusses inheritance and how it can be supported in Ada, but he suggests: "that inheritance is an important, but not necessary, concept. On a continuum of 'object-orientedness', development without inheritance still constitutes object-oriented development".

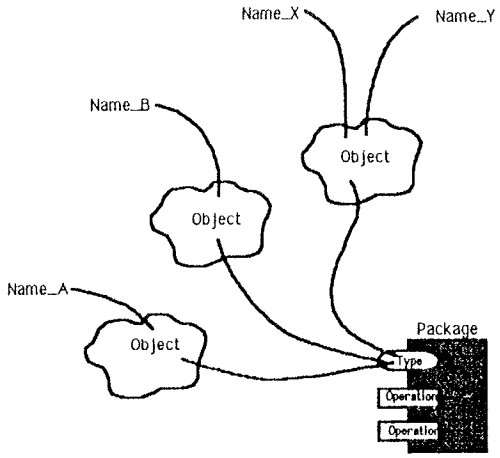


Fig. 6. Names, objects, and classes.

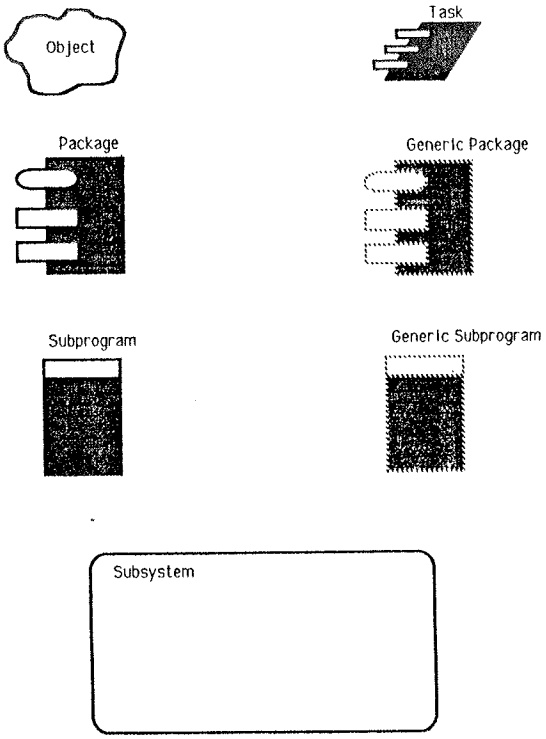


Fig. 7. Symbols for object-oriented design.

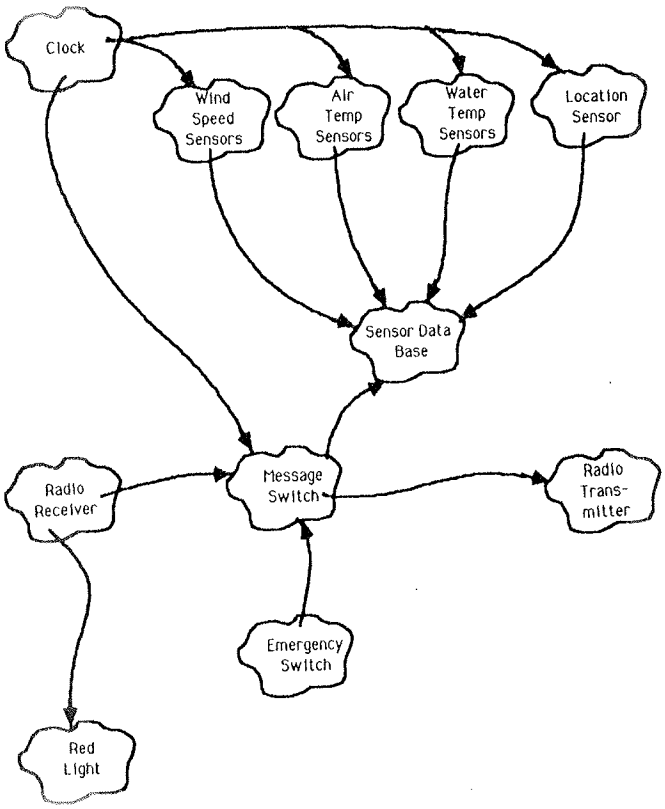


Fig. 9. Host at sea buoy objects.

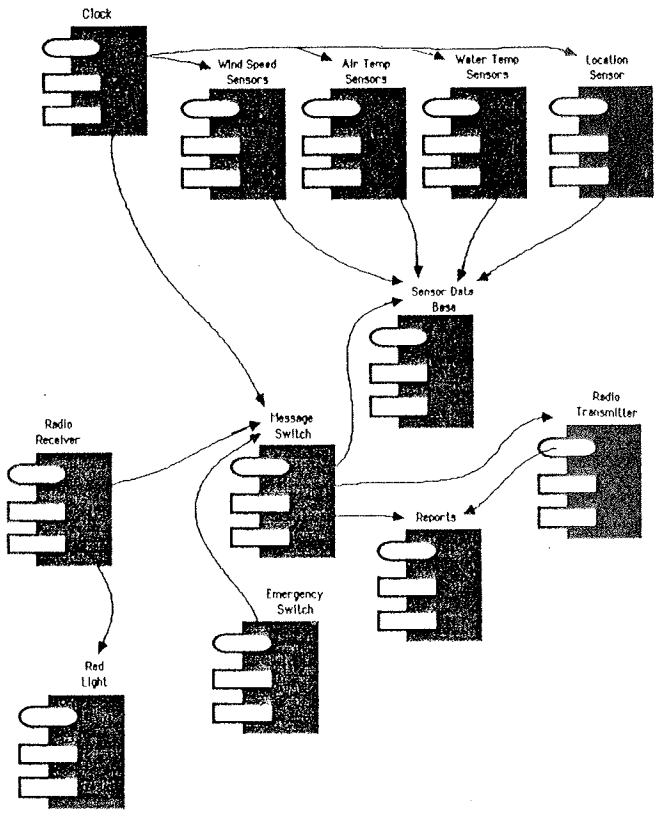


Fig. 10. Host at sea buoy objects.

Figure A.1: OOD Figures

Action sequences can be concurrent. The mapping to concurrent objects and communication between concurrent objects is not discussed. The object diagrams express only the dependencies between the objects, there is no specification of control.

A.2 Wasserman

Object-oriented Structured Design (OOSD) is an architectural design method [Wasserman 89]. OOSD provides a language independent architectural design notation, that is based on the structure chart of SD. The nodes of the OOSD structure chart are however more detailed, and constitute objects (classes) instead of functions. They are similar to the object diagrams of OOD [Booch 86], but are substantially more elaborated. See Fig. A.2 (Wasserman Fig. 19 and 20).

A graphical editor is being developed as the newest member of a family of graphical editors (Software through Pictures [Wasserman 87], that supports creation of OOSD-charts, checking of design rules and code generation. The editor can be configured for particular programming languages, by associating language-specific templates with the editor.

Classification is supported. Aggregation is supported.

Generalization: Simple specialization in several levels is mentioned, but only generic packages are shown in the example. However generic tasks are also supported in the notation, despite the fact that these are not supported in Ada.

Action sequences can be concurrent. OOSD supports the definition and use of asynchronous processes, similar to Ada tasks. The notation is however designed to support traditional fork/join communication and message passing, as well as the Ada rendezvous. Ada designers are suggested to extend the notation with symbols for communication between concurrent objects. Buhr's notation for showing guarded entry calls and sequencing of entry calls is recommended.

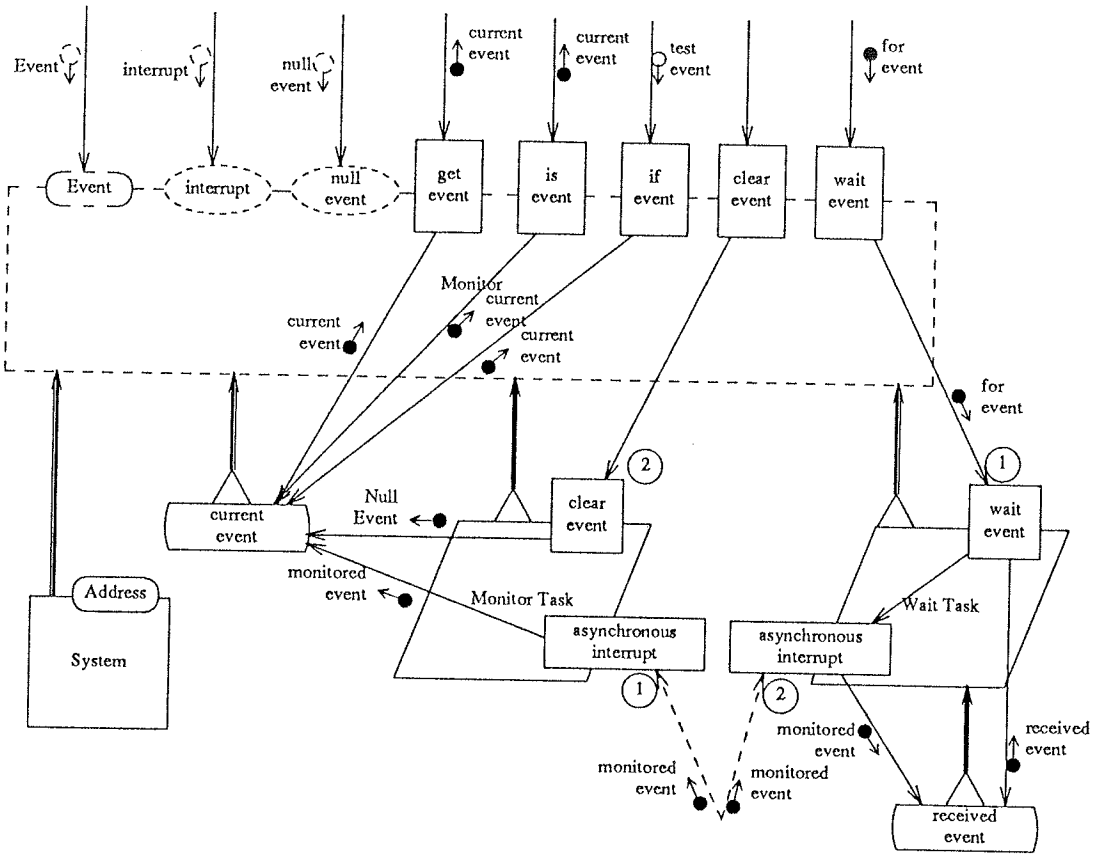


Figure 19 -- OOSD chart for the Monitor generic package

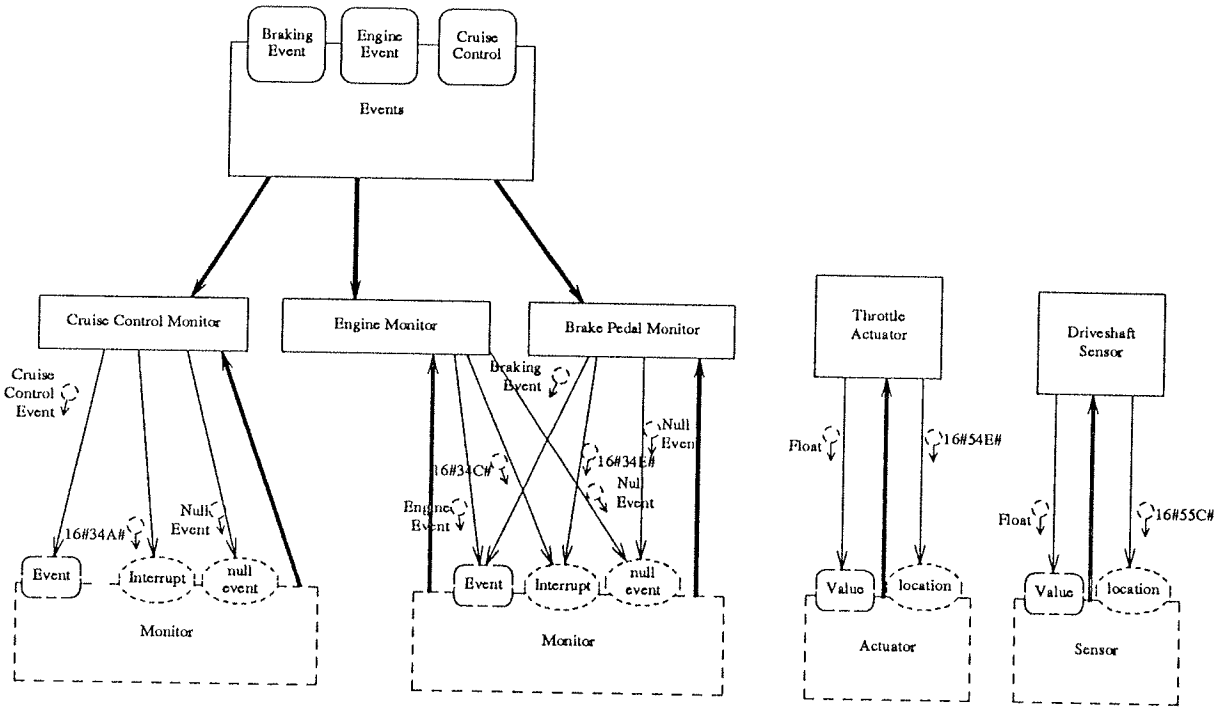


Figure 20 -- OOSD chart for Auto_Parts

Figure A.2: OOSD Figures

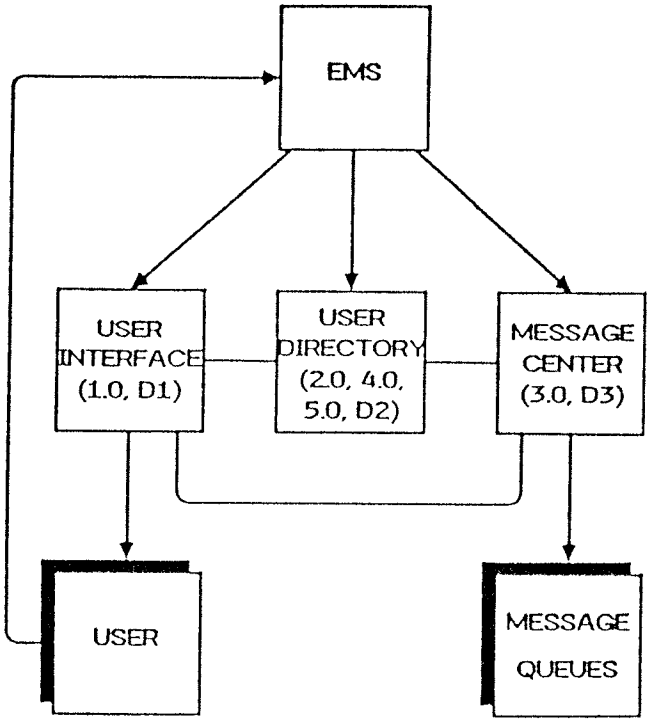


FIGURE 12 EMS entity graph

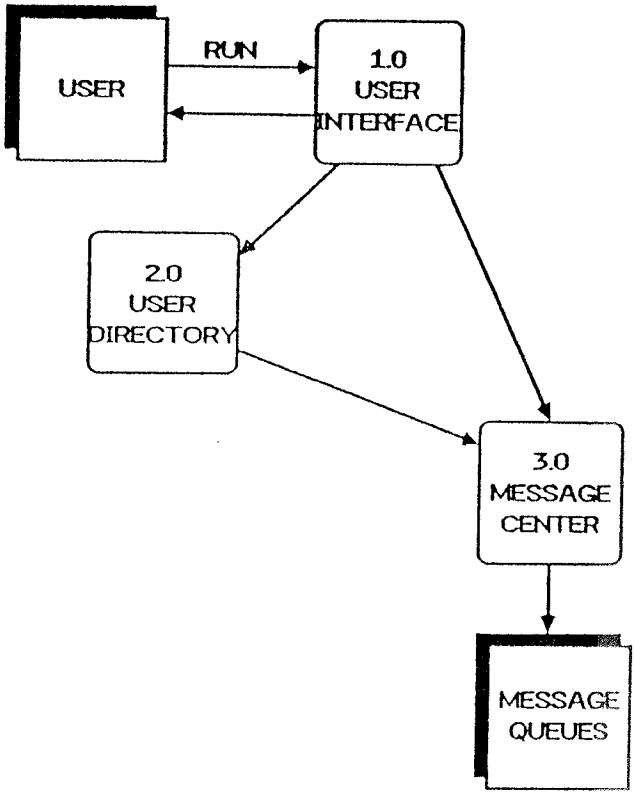


FIGURE 13 EMS object diagram

Figure A.3: GOOD Figures

A.3 Seidewitz and Stark

General Object-oriented Software Development Methodology (GOOD) [Seidewitz 87], is based on the work of Booch. The product of GOOD is an *object-oriented design*, i.e. a version of Booch' object diagram together with *object descriptions* specifying detailed 'provides' and 'uses' information. See Fig. A.3 (Seidewitz Fig. 12 and 13).

The object-oriented design can then be transformed into Ada program templates as demonstrated by Booch. Like OOD, GOOD is focusing on design and implementation. The contribution of Seidewitz and Stark is a method called *abstraction analysis*, for making a transition from a structured specification (SA) to an object-oriented design.

Phenomena are modelled as objects that are regarded as abstract data types.

Classification seems not to be supported.

Aggregation is supported in the object diagrams, much like aggregation in DFDs, but it is unclear whether the decomposition of objects can be done on more than two levels. It seems like the lower levels of decomposition is used for describing the operations of the decomposed object.

Generalization is not supported.

Action sequences can be concurrent and "rendezvous" is mentioned as the communication mechanism between concurrent objects, but is not discussed further. Because of the uncertainty about object aggregation on more than two levels, it is unclear whether concurrent objects can be nested. The object diagrams express only the dependencies between the objects, there is no specification of control.

A.4 Bailin

Bailin describes a method for analyzing requirements for object-oriented software (OOS) [Bailin 89]. The starting point of OOS is a textual statement of requirements or ideally a database of discrete, traceable requirements for a system. OOS is inspired by the notation of Structured Analysis (SA), but the data flow diagrams (DFDs) are constructed by means of object-oriented decomposition as opposed to functional decomposition. An *object-oriented specification*, the product of OOS, is a hierarchy of entity data flow diagrams (EDFDs) and a set of entity relationship diagrams (ERDs). ERDs serve as the context for the specification, they provide a comprehensive view of the problem domain. EDFDs are like conventional DFDs but the nodes fall into two categories: *entities* and *functions*. Bailin uses the term entity instead of object. A process in SA may be a valid entity, but the emphasis is not on transformation of inputs to outputs, but on the underlying *content*. Content is regarded as:

- the data structures that define the entity
- the underlying state of a process as it evolves in time
- that aspect of a process that is persistent across repeated execution cycles

A function is purely a transformation of inputs to outputs. It has no underlying state that it remembers across successive invocations. Every

function must occur in the context of an entity, i.e. it must be performed by or act on the entity. Entities are divided into *active* and *passive* entities. Active entities are processes (they have their own execution thread) or "important" abstract datatypes (ADTs). Passive entities are "less important" ADTs that are represented as data flows or data stores.

The specification method (OOS) consists of seven steps:

1. identify key problem-domain entities
2. distinguish between active and passive entities
3. establish data flow between active entities
4. decompose entities (or functions) into sub-entities and/or functions.
5. check for new entities
6. group functions under new entities
7. assign new entities to appropriate domains

Steps 4 through 7 should be iterated until the desired level of detail is reached.

Some tools have been developed, that support OOS. A rule checker checks consistency between the ERDs and the EDFDs. A design generator tool reads a hierarchy of EDFDs and outputs a hierarchy of objects diagrams as defined by Seidewitz and Stark. Active entities are mapped into ordinary objects, that corresponds to Ada packages. Only the highest-level functions are converted to procedure objects, that corresponds to Ada tasks and procedures. The object diagrams are in turn transformed to Ada program templates following the same principles as in [Seidewitz 87].

OOS models phenomena more detailed as seen in the previous methods. Objects can be active or passive, and functions can also be represented as nodes in the EDFDs. The latter is of course not a development in relation to traditional SA, but the nodes of the object diagrams as described by Booch and Seidewitz were only objects. See Fig. A.4 (Bailin Fig. 4 and 5).

Classification is not supported.

Aggregation is supported in an arbitrary number of levels. Entities can be decomposed into sub-entities or functions. Functions can however only

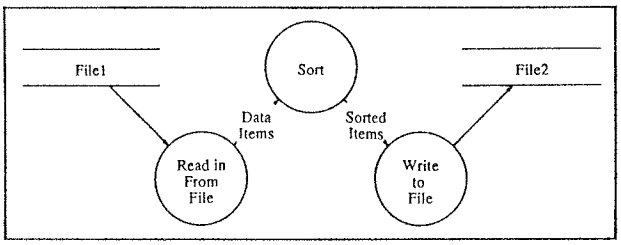


FIGURE 4. Processes Operate on Incoming Data in Structured Analysis

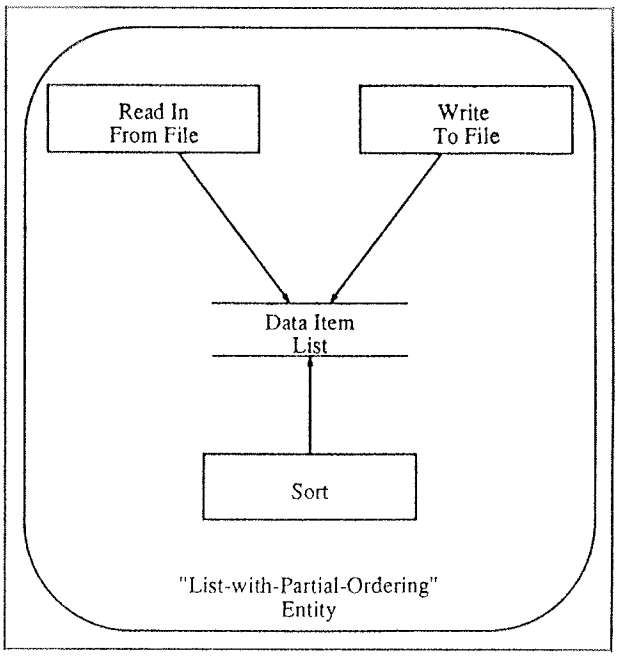


FIGURE 5. In OOS, the Operation and the Data are Part of the Same Entity

Figure A.4: OOS Figures

be decomposed into sub-functions. The graphical notation for aggregation/decomposition is the same as in conventional DFDs. The important development in OOS is aggregation of objects, not only functions. This means that objects can contain part objects and the operations (functions) on the object, and functions can be expressed by subfunctions.

Generalization is not considered at all.

Action sequences can be concurrent. Active entities are considered as concurrent objects and functions can also have their own execution thread. The support for aggregation means also, that it is possible to express hierarchies of concurrent objects, but the type of communication between such objects is not discussed. The EDFDs and the object diagrams express only the dependencies between the objects, there is no specification of control. Bailin mentions however integration of EDFD and state machines as a subject for further experiments. Ward and Mellor have done this integration, as will be described in chapter A.5.

A.5 Ward

Real-time Structured Analysis and Structured Design (RT SA/SD) [Ward 85] [Ward 86] is an extension of the original SA/SD method, that is aimed at supporting the development of real-time systems. RT SA/SD extends the original method to deal with concurrency, synchronization, and other control issues. One addition is a version of the entity-relationship diagram (ERD), another is the use of stimulus-response analysis. The ERD is used to describe the application domain. The stimulus-response analysis is reflected in an extended version of the dataflow diagram called the *transformation schema* (TS). The TS has additional symbols to represent real-time aspects. Data flows are categorized into *continuous* and *discrete* flows. *Event* flows (interrupt-type messages with no variable content) are added. In addition to the data transformation (process), a *control transformation* (process) is used to turn other transformations on and off by means of enabling and disabling event flows. Control transformations are further specified by means of *state transition diagrams* (STDs).

The transformations (processes) are identified, not by functional decomposition, but by stimulus-response partitioning. Stimuli arise in the application and require a response from the system. For each stimulus the

system may require transformations to recognize the stimulus and transformations to carry out the response.

RT SD is carried out not by transforming the SA model into a hierarchy of structure charts (like in traditional SD), but by reorganizing the highest-level transformations from the RT SA into a new top-down model using the same notation and representing the level of target hardware/software environment. RT SD model organization is strongly influenced by concurrency requirements. Only on the lowest levels of the top-down model, where the transformations represent single tasks in single processors, will the internal structure of these transformations be described by structure charts.

Ward has described how to integrate object-orientation with Structured Analysis and Design (OO RT SA/SD) [Ward 89]. He emphasizes two major themes in object-oriented design: "the closely related ideas of object, class and abstract datatype and the idea of inheritance".

As mentioned, the transformations in the TS are identified not by function decomposition but by stimulus-response partitioning. This approach however has the same tendency as conventional SA to create a large number of processes (one for each function). See Fig. A.5 (Ward Fig. 5).

The heuristic for making the TS object-oriented is then to let object-identification take precedence over potential concurrency. Fig. A.6 (Ward Fig. 6 and 7) show partion of Ward Fig. 5 based on concurrency-partition and object-partition, respectively.

The RT SA is reinterpreted in the following way:

- the entity-relationship analysis is used to identify both identity and the data content of the application domains abstract datatypes
- the stimulus-response analysis is used to identify the operations (transformations) by which the data types need to be accessed
- object identification is used as a heuristic for grouping lower level transformations (operations) and stores (data content) into higher level transformations (objects)

Classification is supported. An abstract data type is represented as a disembodied transformation with its inputs and outputs; it is not instan-

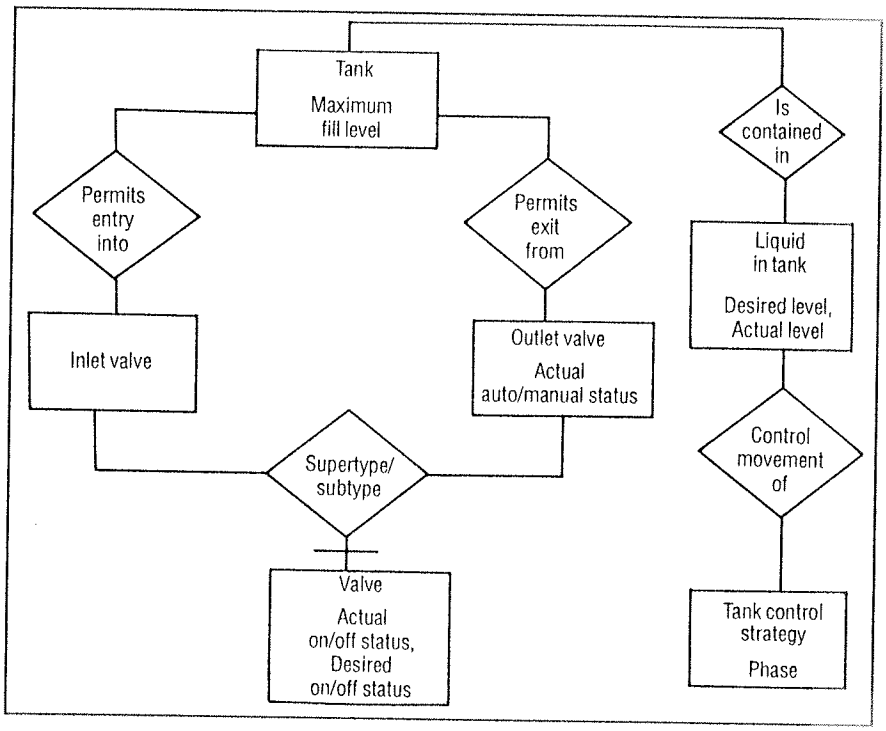


Figure 4. Entity-relationship diagram.

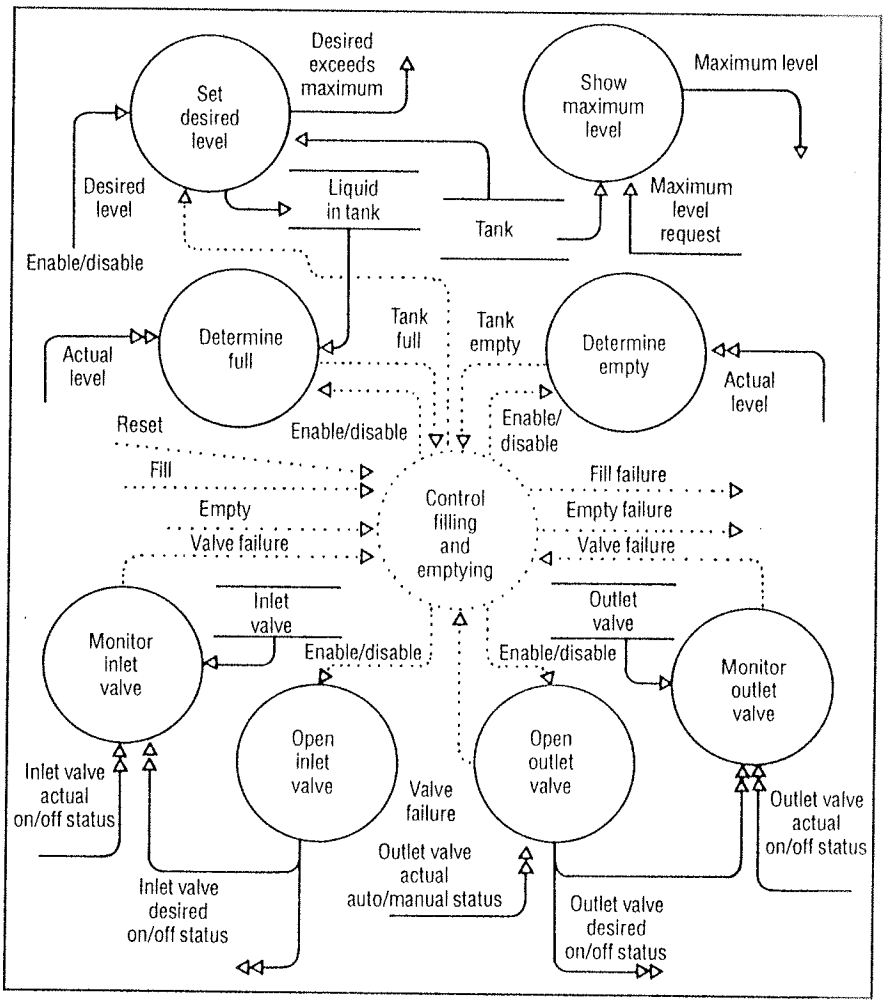


Figure 5. Real-time structured-analysis transformation schema.

Figure A.5: Ward Figures

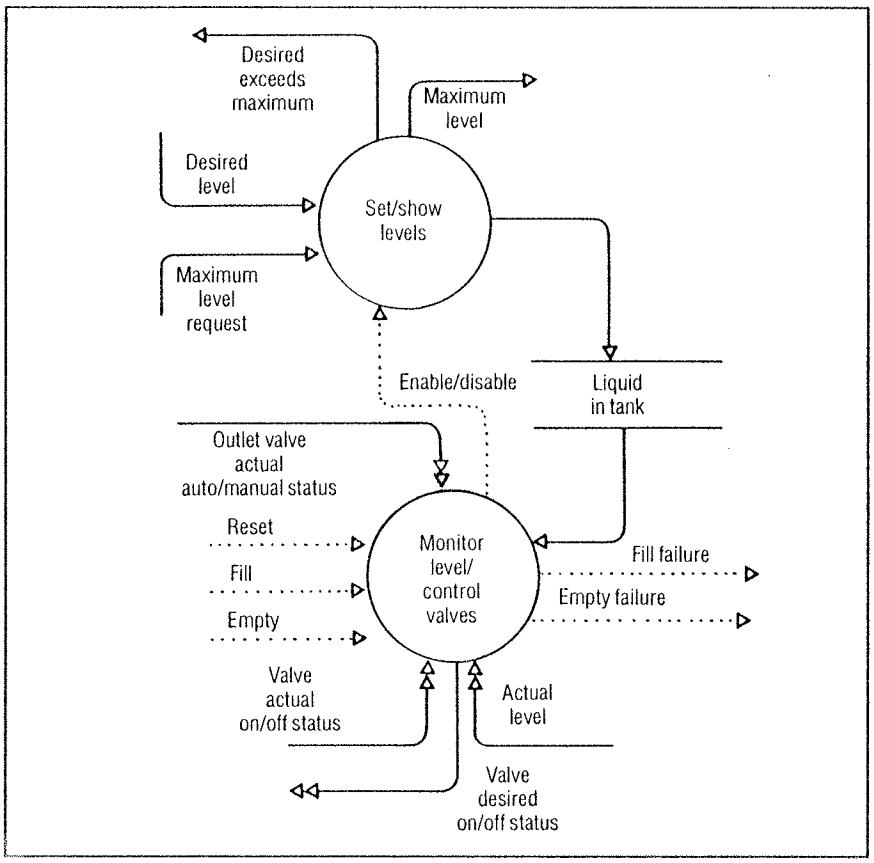


Figure 6. Top level of structured-design model partitioned on concurrency.

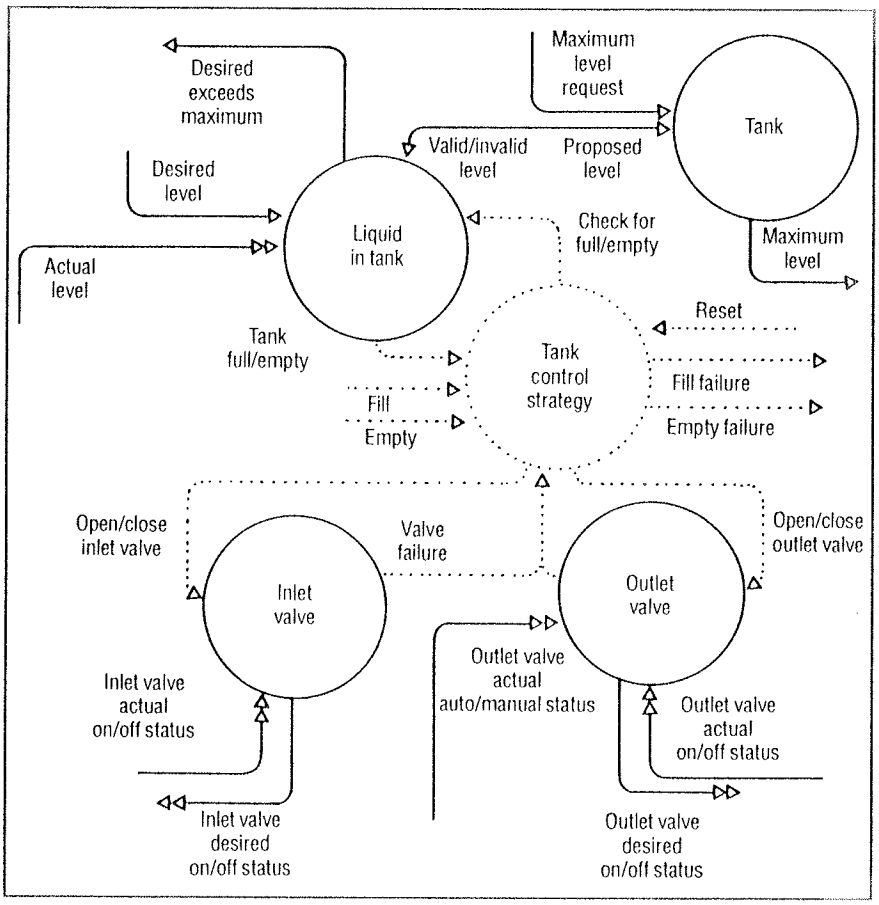


Figure 7. Object-partitioned transformation schema.

Figure A.6: Ward Figures

tiated if it is not incorporated in a model. A single object is represented as a transformation in the context of a model.

Aggregation is supported in an arbitrary number of levels; the usual graphical aggregation/decomposition techniques of DFDs is used on objects.

Generalization is supported. Inheritance, i.e. simple specialization, is expressed in the diagrams by allowing different schemas (subclasses) to contain the same transformation (superclass). This requires some relaxation of the normal conventions for TSs (enforced by hierarchical numbering of schemas and transformations), for example that distinct transformations must be represented at a lower level by distinct schemas. The ERD provides guidelines for this partitioning because it may express supertype and subtypes.

Wards example of expressing inheritance is shown in Fig. A.7 (Ward Fig. 8 and 9).

In the example a general valve is described (Ward Fig. 9). This valve is specialized into an inlet valve and an outlet valve (Ward Fig. 8). The valves are used to fill or empty a tank of liquid. The inlet valve functions like the general valve, but the outlet valve must, in addition to the general properties of a valve, check, whether the valve status is manual or automatic. If the status is manual, the system may not operate on the valve, and the outlet valve must send a valve failure signal to the tank control transformation.

The solution that Ward has chosen, is to describe the valve without control transformation. The specification of valve control is deferred to subclasses of Valve. 'Inlet valve' has a 'Control inlet valve' control transformation and 'Outlet valve' has a 'Control outlet valve' control transformation. In addition 'Outlet valve' has a data transformation that signals the status of the auto/manual switch in the system. This additional signal is considered in the control logic of the 'Control outlet valve' control transformation.

To describe that 'Valve' is inherited in 'Outlet valve' and 'Inlet valve' its symbol is repeated in the subclasses. This solution can be characterized as a kind of "downward aggregation". Valve is presented as an aggregate of the attributes in Ward Fig. 9, but it is not possible to see the contents of an aggregate, except for the in-going and out-going flows. It is therefore

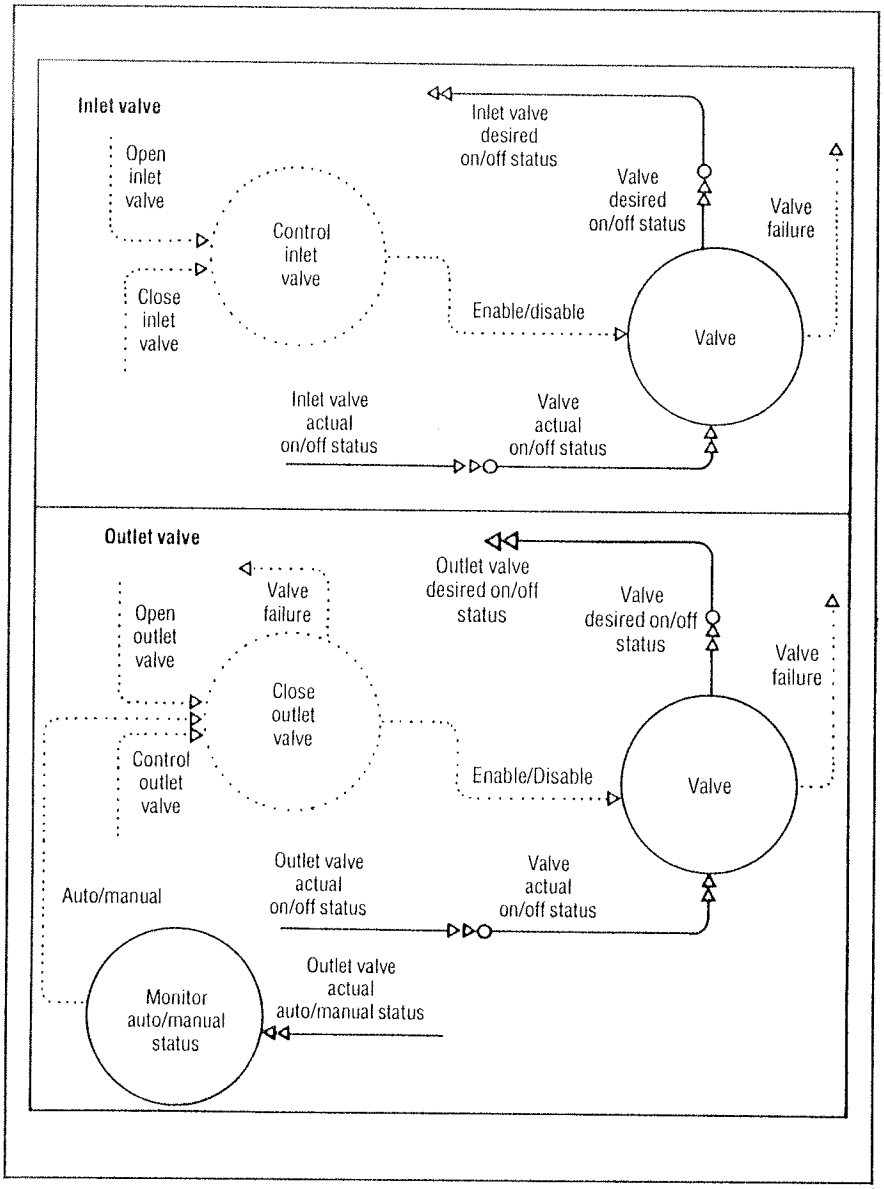


Figure 8. Inlet-valve and outlet-valve processing.

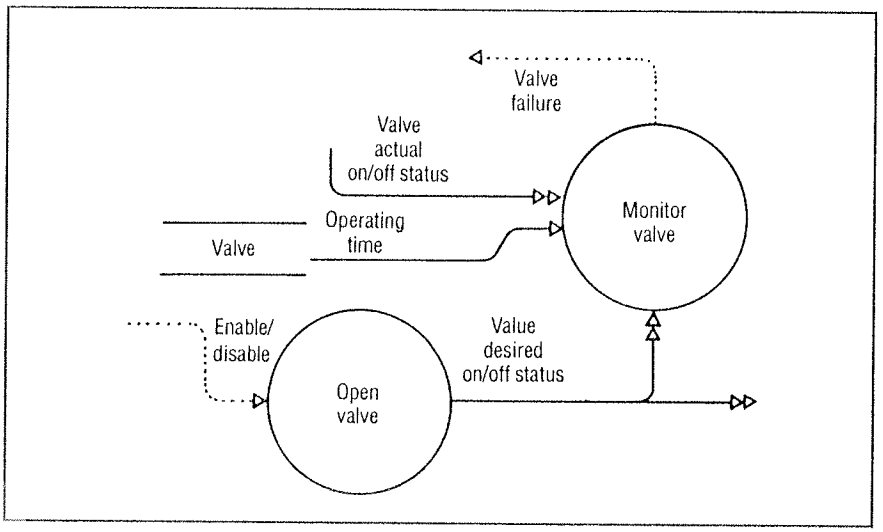


Figure 9. Common valve processing.

Figure A.7: Simple Specialization in Real-time SA/SD

not possible to add a flow to an internal object of the superclass, for example to call a procedure in a passive object of the superclass. This suggests, that the solution of Ward does not cover all cases of simple specialization.

Action sequences can be concurrent. The control of and communication between objects is made very explicit in the graphical notation. The addition of control transformations, signals and state transition diagrams makes it possible to execute the transformation schema. In [Ward 86] it is described how a transformation schema can be executed/simulated. The execution model is based loosely on the execution of a Petri net [Peterson 77]. Execution of the schema is visualized in terms of the placement of tokens. Execution of a notation similar to that of real-time SA/SD has been implemented on a commercially available CASE product [Harel 87].

A.6 Coad and Yourdon

Object-Oriented Analysis (OOA) [Coad 89] is first of all an analysis method, but it is not supposed to stand alone. One goal has been to avoid the traditional "representation shift" when moving from analysis to design. "Design should consist solely of expanding the requirements model to account for the complexities introduced in selecting a particular implementation." ... "It's a matter of using the same underlying representation in analysis and design. This concept is the the foundation of the OOA approach." [Coad 89].

OOA consists of five major steps:

1. Identifying Objects
2. Identifying Structures
3. Defining Subjects
4. Defining Attributes (and Instance Connections)
5. Defining Services (and Message Connections)

The model is presented and reviewed in five major layers:

1. Subject layer
2. Object layer
3. Structure layer
4. Attribute layer
5. Service layer

Subject layer. A Subject is a mechanism for controlling how much of the model that a reader considers at one time.

Object layer. An Object is an abstraction of data and exclusive processing of data. Despite the name 'Object' this term is used for modelling concepts. Phenomena are modelled as 'instances of Objects'. In other words, what is "normally" called classes is in OOA called Objects, and what is "normally" called objects is in OOA called instances of Objects. In OOA the primary focus is on concepts (classes) as opposed to phenomena (objects). The OOA terms will be used in the following (Starting with capital letters).

The Structure layer is used to represent complexities in two ways: Classification Structure is used to express generalization/specialization and Assembly Structure is used to express aggregation/decomposition.

Attribute Layer. An Attribute is a data element used to describe an instance of an Object or Classification Structure. Attributes are shown on the diagram. See Fig. A.8 (Coad Fig. 1.9). Instance Connections represent a mapping of an instance of an Object or Classification Structure with other instance(s). They are shown with a solid line, with markings indicating multiplicity and participation constraints. Instance connections are used to express relationships similar to those described in Entity-Relationship Diagrams.

The Service layer is used to express the processing that can be performed by the Objects. A Service is the processing to be performed upon receipt of a message. Services are listed in the bottom of the Object and Structure symbols. See Fig. A.9 (Coad Fig. 1.10). Services are categorized into three fundamental services: 'Occur' (add, change, delete, test and connect an occurrence), 'Calculate' and 'Monitor'. The 'Occur' Services are not shown in the diagrams, they are specificed once, as an implicit Service for all Objects and Classification Structure. A 'Calculate' Service

calculates results for an instance, or on behalf of another instance. The 'Monitor' Service performs on-going monitoring of an external system, device or user.

Message Connections are used to express the sending of a message from one Object to another, to get some processing done on their behalf. They are shown with dashed arrows, pointing from the sender of the message to the receiver of the message.

As mentioned the primary focus of OOA is on concepts as opposed to phenomena. The Instance Connections are however used to express relationships between phenomena.

Abstraction is supported at a high degree.

Aggregation is supported by means of Assembly Structures, but the nesting of Objects is not shown by a corresponding nesting in the diagrams. The aggregation hierarchies of Objects are shown by means of interconnecting arrows in the same diagram. One of the possible further developments of OOA is mentioned to be to allow Attributes to be Objects themselves. This would support aggregation in a more general way. Another future topic is adding and distinguishing between Objects and Collections of Objects.

Generalization is supported by means of Classification Structures. Simple specialization is supported by putting common Attributes and Services higher in the structure and showing specialization below. Qualified specialization is supported for Services; a Service specified at a higher level in the structure can be extended below. Finally it is possible to override (cancel) attributes. The "x" notation used in Fig. A.8 (Coad Fig. 1.9) indicates that an inherited attribute does not supply to a particular specialization. This kind of attribute cancellation has been chosen as an alternative to multiple inheritance. It does not conform with the view on generalization/specialization in this report. A specialized concept is here considered to be an extension of a more general concept.

Interaction between Objects is indicated by Message Connections. Concurrency is not considered in the OOA model. This is postponed to the object-oriented design (OOD) model. In OOD the OOA model is expanded according to the hardware and software architecture. A separate task diagram can show the tasks, priority, multiple instantiations, plus communication and coordination (via rendez-vous, or with more tradi-

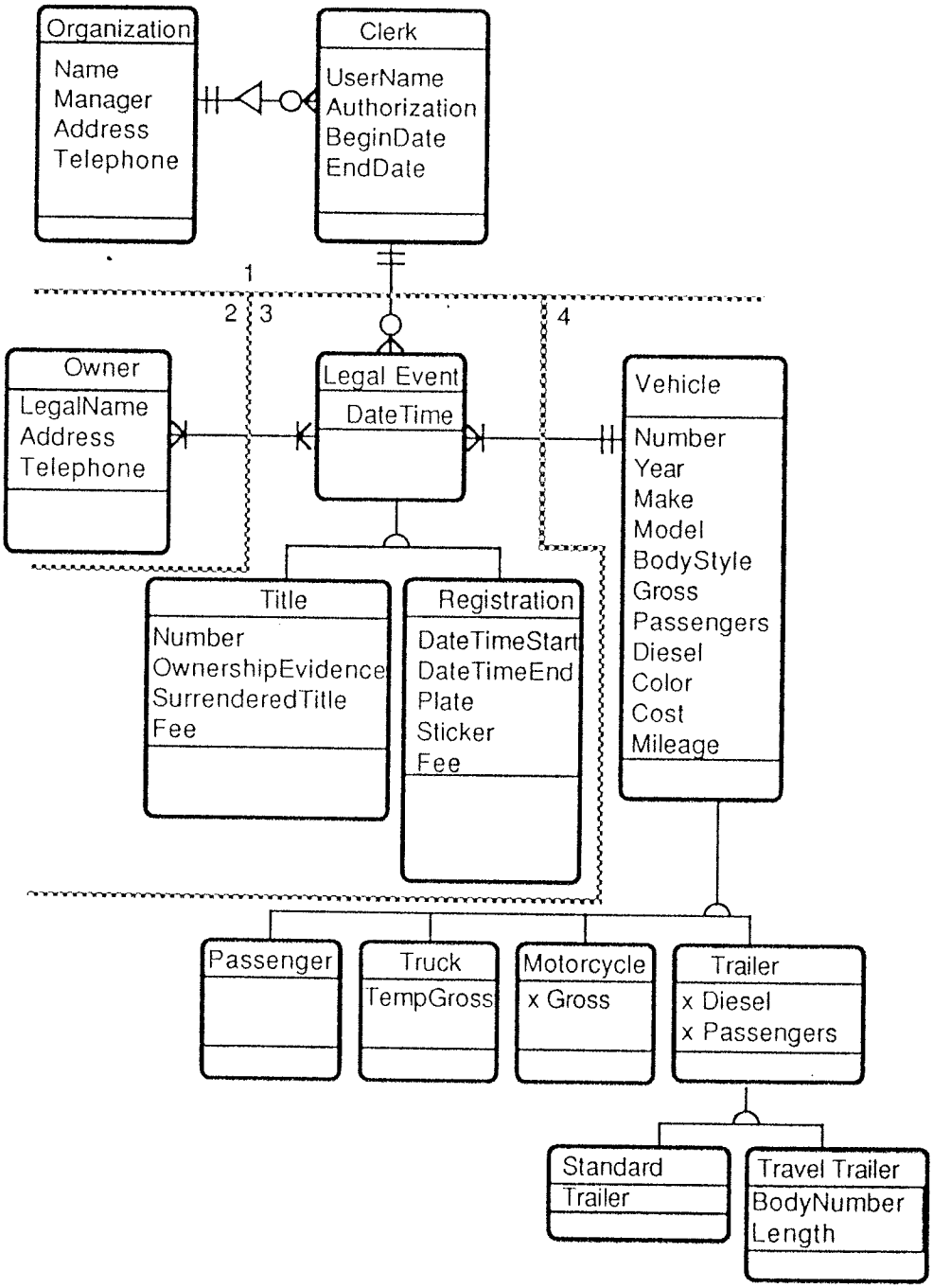


Figure 1.9: *An OOA Example: Attribute Layer*

Figure A.8: OOA Figure

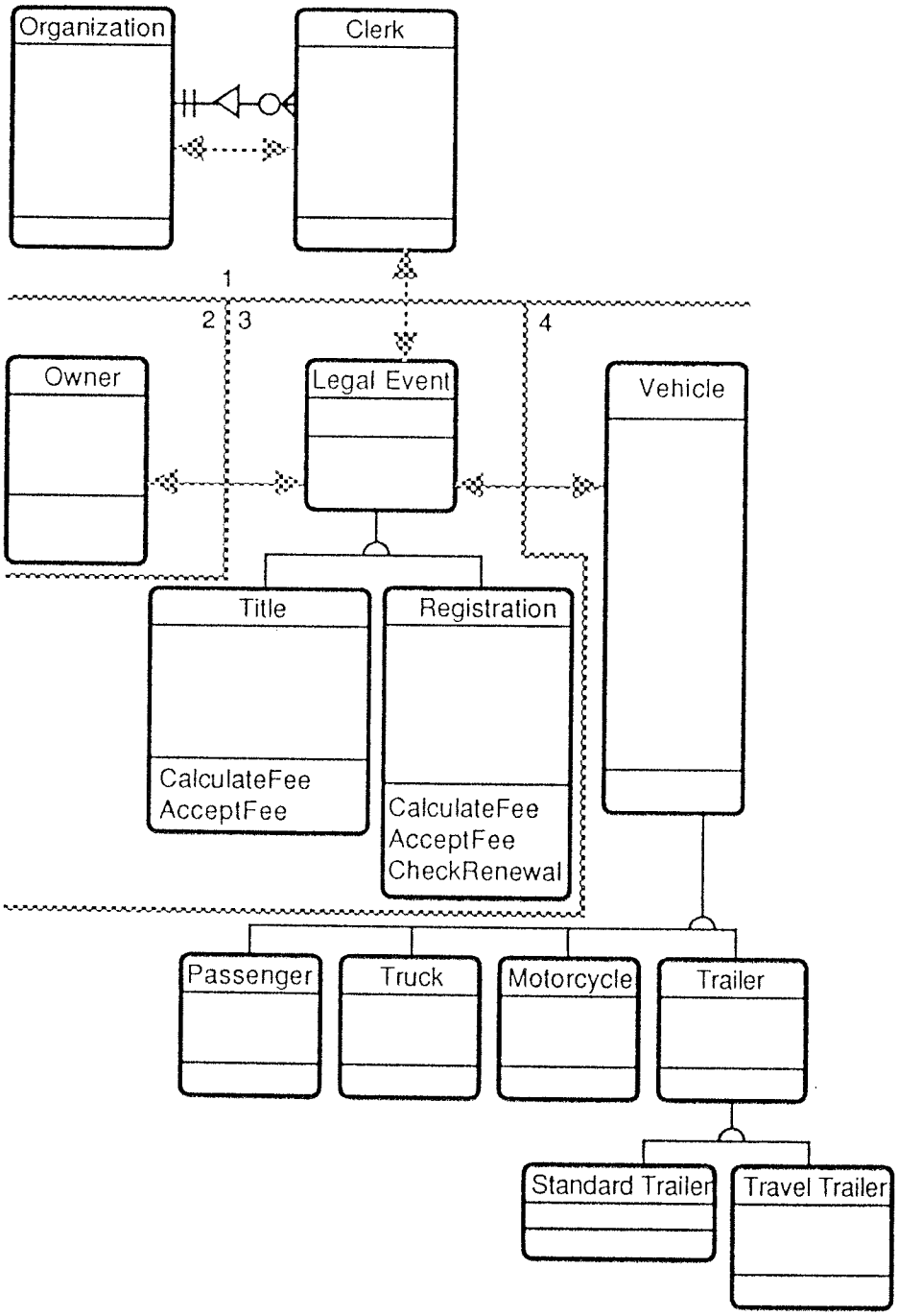


Figure 1.10: *An OOA Example: Service Layer*

Figure A.9: OOA Figure

tional mailboxes, semaphores and interrupts). The OOD model itself can be annotated with task names.

Appendix B

The Flight Reservation System in BETA

This appendix contains the BETA program pieces that correspond to the example in chapter 3. Fig. B.1 and Fig. B.2 show the object diagram and the pattern diagram that are mapped to BETA.

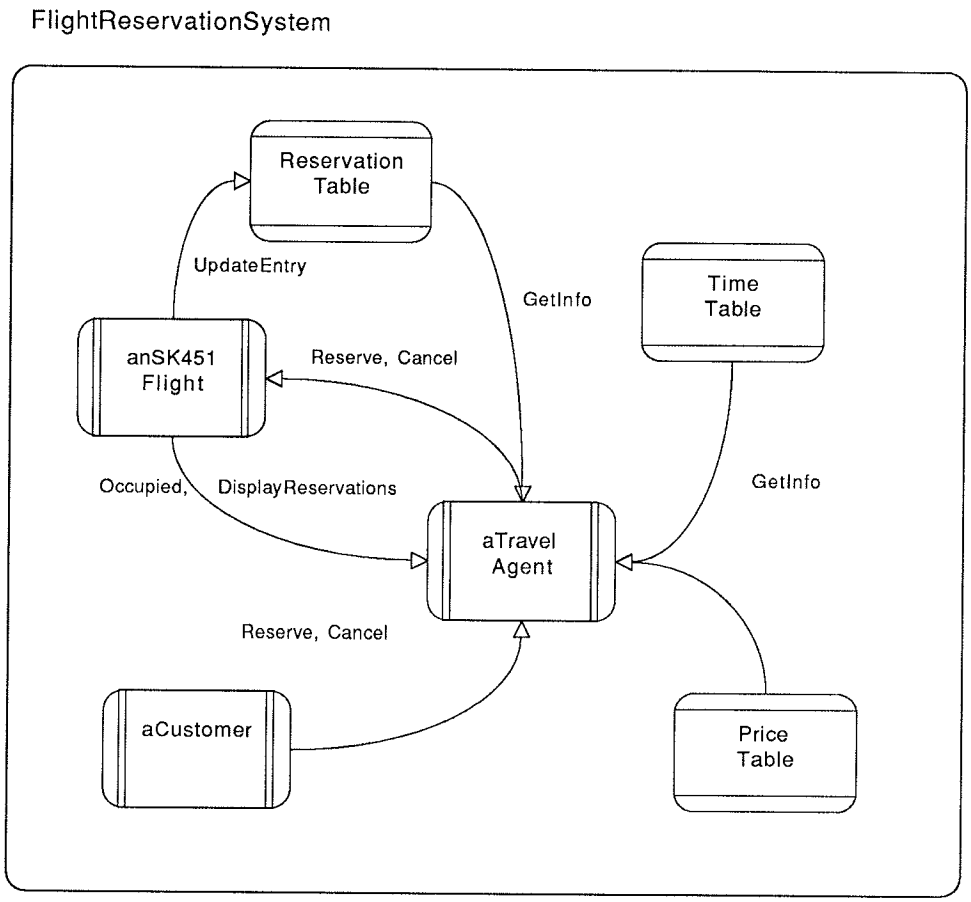


Figure B.1: Flight Reservation, Object Diagram

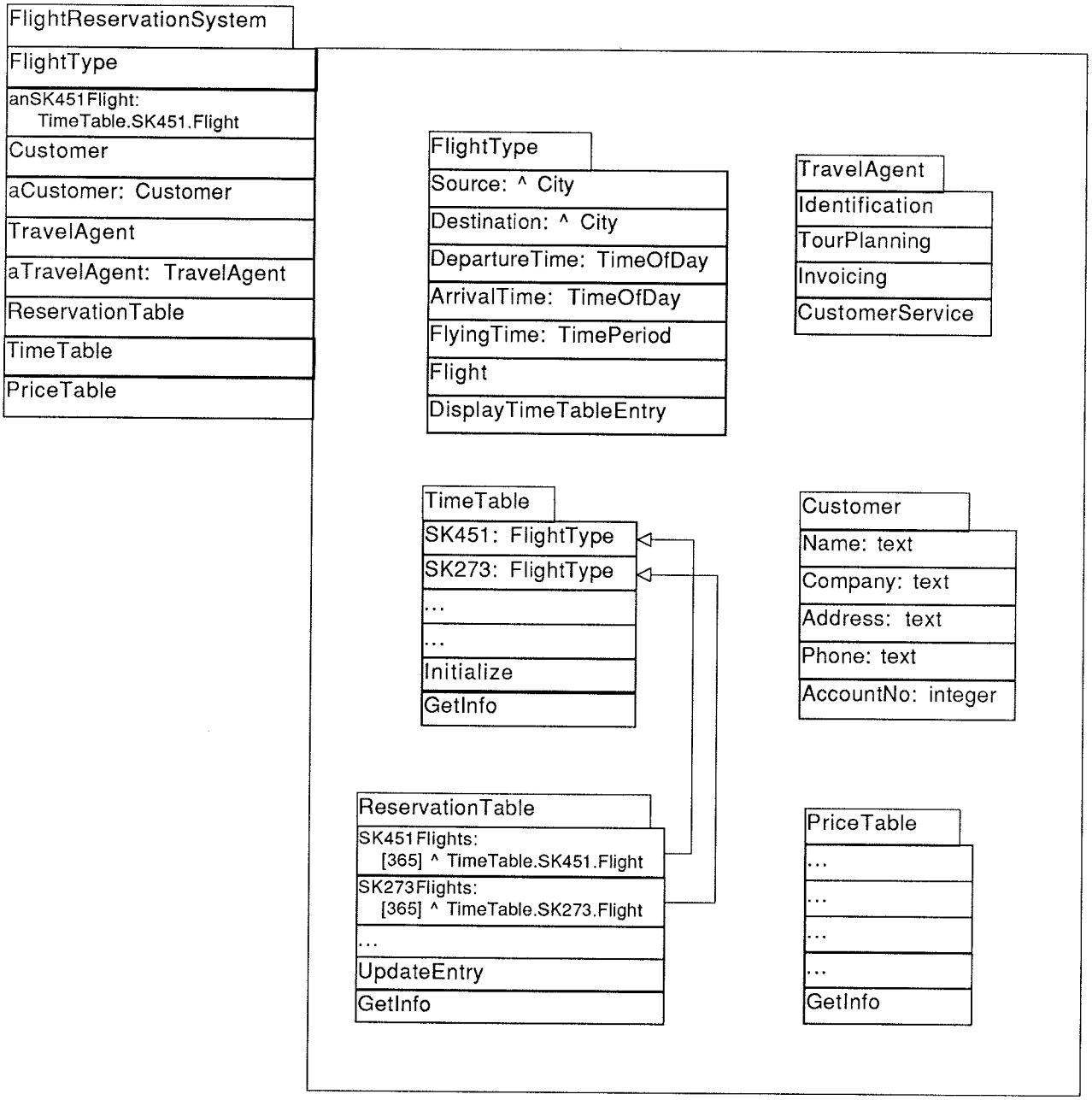


Figure B.2: Flight Reservation, Pattern Diagram

```
FlightReservationSystem: @ concurrent
  (#
      FlightType: <<... ObjectDescriptor ...>>;
      TravelAgent: <<... ObjectDescriptor ...>>;
      Customer: <<... ObjectDescriptor ...>>;

      anSK451Flight: @ concurrent TimeTable.SK451.Flight;
      aTravelAgent: @ concurrent TravelAgent;
      aCustomer: @ concurrent Customer;

      ReservationTable: @ passive <<... ObjectDescriptor ...>>;
      TimeTable: @ passive <<... ObjectDescriptor ...>>;
      PriceTable: @ passive <<... ObjectDescriptor ...>>

  do (|| anSK451Flight || aTravelAgent || aCustomer ||)
  #)
```

Figure B.3: Flight Reservation

```
FlightReservationSystem: @ concurrent
  (# FlightType:
       (# Source, Destination: ^ City;
          DepartureTime,
          ArrivalTime: @ passive TimeOfDay;
          FlyingTime: @ passive TimePeriod;
          Flight: <<... ObjectDescriptor ...>>;
          DisplayTimeTableEntry: @ procedure <<... ObjectDescriptor ...>>
       #);
     TravelAgent:
       (# Identification: @ passive <<... ObjectDescriptor ...>>;
          CustomerService: @ alternation <<... ObjectDescriptor ...>>;
          Invoicing: @ alternation <<... ObjectDescriptor ...>>;
          TourPlanning: @ alternation <<... ObjectDescriptor ...>>
       do (| CustomerService | Invoicing | TourPlanning |)
       #);
     Customer:
       (# Name,
          Company,
          Address,
          Phone: @ passive Text;
          AccountNo: @ passive Integer;
       do <<... Imperatives ...>>
       #);
     anSK451Flight: @ concurrent TimeTable.SK451.Flight;
     aTravelAgent: @ concurrent TravelAgent;
     aCustomer: @ concurrent Customer;
     ReservationTable: @ passive
       (# SK451Flights: [365] ^ TimeTable.SK451.Flight;
          SK273Flights: [365] ^ TimeTable.SK451.Flight;
          ...
          GetInfo: @ procedure <<... ObjectDescriptor ...>>
       do <<... Imperatives ...>>
       #);
     TimeTable: @ passive
       (# SK451: @ passive FlightType;
          SK273: @ passive FlightType;
          ...
          Initialize: @ procedure <<... ObjectDescriptor ...>>;
          GetInfo: @ procedure <<... ObjectDescriptor ...>>
       do <<... Imperatives ...>>
       #);
     PriceTable: @ passive <<... ObjectDescriptor ...>>
  do (|| anSK451Flight || aTravelAgent || aCustomer ||)
  #)
```


Figure B.4: Flight Reservation

```
Flight:
  (# Seat:
       (# FlightReservation: <<... ObjectDescriptor ...>>;
          Reserved: ^ FlightReservation;
          Class @ passive ClassType;
          Smooking: @ passive Boolean;
          Reserve: @ procedure <<... ObjectDescriptor ...>>;
          Cancel: @ procedure <<... ObjectDescriptor ...>>;
          isReserved: @ procedure <<... ObjectDescriptor ...>>
       #);
     Seats: [NoOfSeats] @ Seat;
     ActualDepartureTime,
     ActualArrivalTime: @ TimeOfDay;
     ActualFlyingTime: @ TimePeriod;

     DepartureDelay: @ procedure <<... ObjectDescriptor ...>>;
     Reserve: @ procedure <<... ObjectDescriptor ...>>;
     Cancel: @ procedure <<... ObjectDescriptor ...>>;
     Occupied: @ procedure <<... ObjectDescriptor ...>>;
     DisplayReservations: @ procedure <<... ObjectDescriptor ...>>;
  do cycle
       (#
       do ...; <? Reserve;
          ...; <? Cancel;
          ...; <? Occupied;
          ...; <? DisplayReservations;
          ...; ReservationTable.UpdateEntry; ...;
       #)
  #)
```

Figure B.5: Flight

```
Reservation:
  (# Date: @ passive DateType;
     theCustomer: ^ Customer;
     theAgent: ^ TravelAgent;
     Smooking: @ passive Boolean;

     PrintReservation: virtual procedure <<... ObjectDescriptor ...>>;
  #);
FlightReservation: Reservation
  (# ReservedFlight: ^ Flight;
     ReservedSeat: ^ Seat;

     PrintReservation: extended procedure <<... ObjectDescriptor ...>>;
  #);
TrainReservation: Reservation
  (# ReservedTrain: ^ Train;
     ReservedCarriage: ^ Carriage;
     ReservedSeat: ^ Seat;

     PrintReservation: extended procedure <<... ObjectDescriptor ...>>;
  #)
```

Figure B.6: Reservation, Simple and Qualified Specialization

Appendix C

Mapping Transformation Schemas to BETA

The transformation schema of Real-time SA/SD can also be mapped into BETA. In chapter 6 it has been shown, how the OAD notation is mapped into BETA. In order to model the transformation schemas some specialized constructs must be used in order to express state transition diagrams and the predefined signals. Fig. C.1 shows a transformation schema. The associated state transition diagram is shown in Fig. C.2. The example is taken from [Bruyn 87].

Fig. C.3 shows the mapping to BETA at the top level. Transformations in the transformation schema are concurrent objects that respond to a set of predefined signals: trigger, enable, disable, suspend and resume. These signals are defined in class Transformation, that is used as a superclass for all transformations. The handling of transformations is done in the TransformationSystem. Class Transformation is illustrated in Fig. C.4. The construct `with some Transformation` means that communication requests will be accepted from concurrent objects, that are qualified by Transformation, i.e. all transformations will accept these predefined signals from all transformations.

The next level of detail is shown in Fig. C.5. The basic idea is to model each state in the transition diagram as an alternation object. This makes good sense, because the state transition diagram is always in exactly one state. The basic alternation objects (components) in BETA are so-called *semi-coroutines*, because there is an asymmetry between the calling coroutine and the coroutine being called. The calling coroutine explicitly names the coroutine to be called, whereas the called coroutine returns

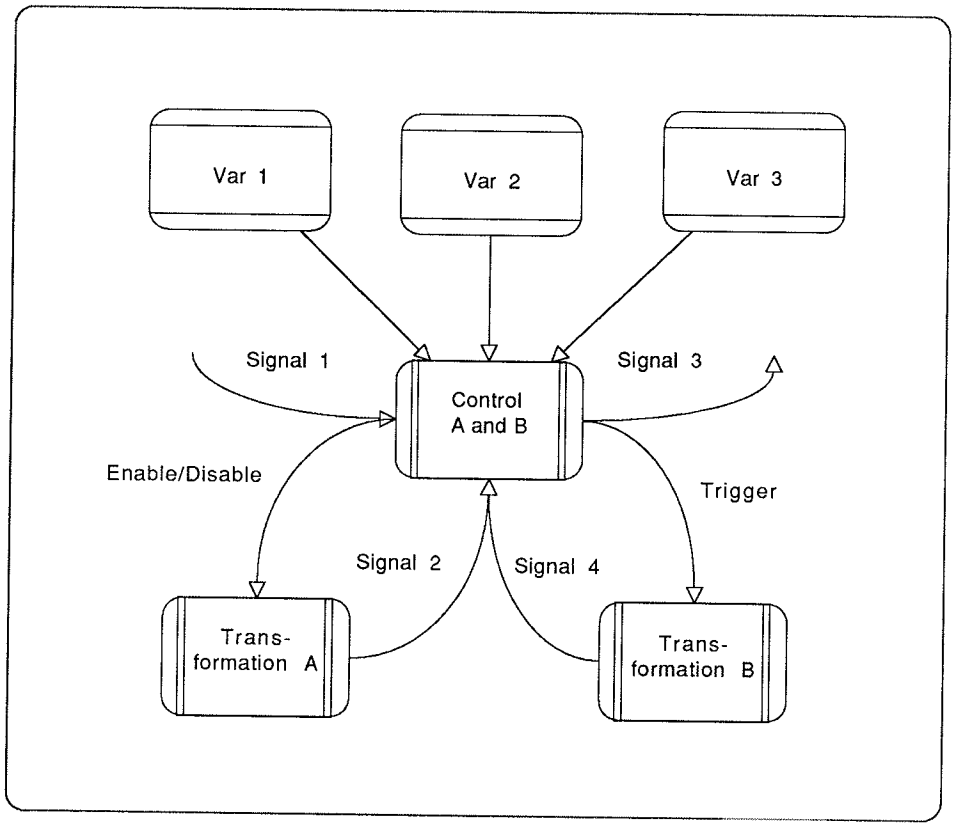


Figure C.1: A Transformation Schema in Real-time SA/SD, but in the OAD Notation

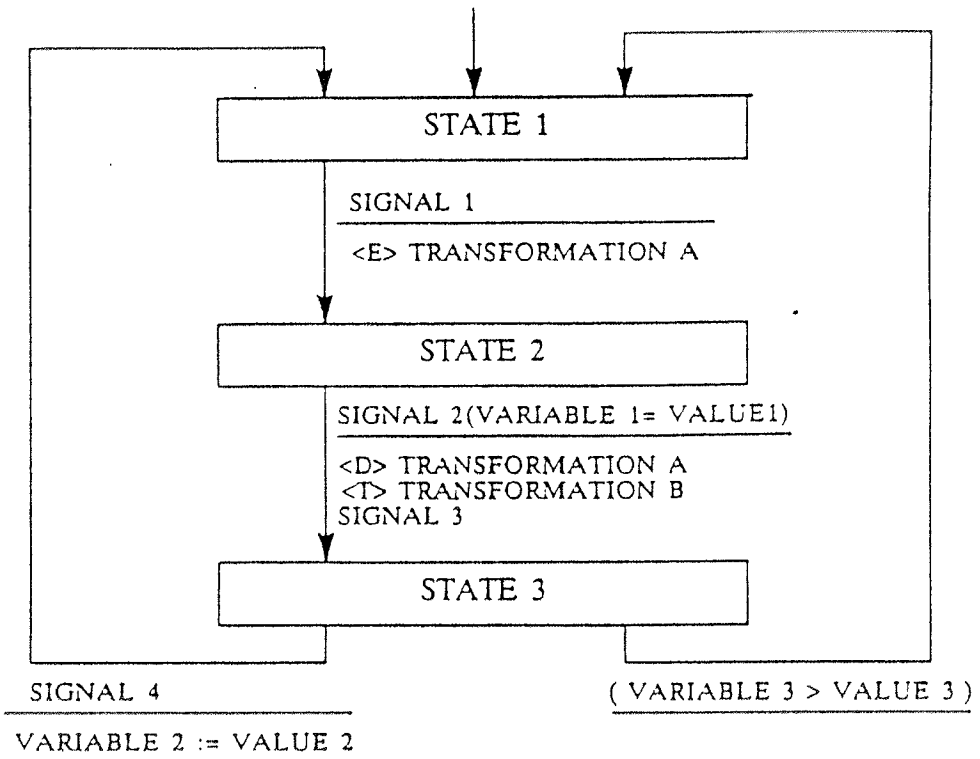


Figure C.2: A State Transition Diagram in Real-time SA/SD

```
TransformationSchema: @ concurrent TransformationSystem
  (#
      ControlAandB: @ concurrent Transformation
        <<... ObjectDescriptor ...>>;
      TransformationA: @ concurrent Transformation
        <<... ObjectDescriptor ...>>;
      TransformationB: @ concurrent Transformation
        <<... ObjectDescriptor ...>>

  do (|| ControlAandB || TransformationA || TransformationB ||)
  #)
```

Figure C.3: Real-time Transformation Schema

```
Transformation: class
  (#  Trigger: @ procedure (# with some Transformation do #);
      Enable: @ procedure (# with some Transformation do #);
      Disable: @ procedure (# with some Transformation do #);
      Suspend: @ procedure (# with some Transformation do #);
      Resume: @ procedure (# with some Transformation do #)
  <<... ActionPart ...>>
  #)
```

Figure C.4: Class Transformation

```
ControlAandB: @ concurrent Transformation
  (#
      STD: @ alternation SymmetricCoroutineSystem
        (#
           State1: @ alternation SymmetricCoroutine
             <<... ObjectDescriptor ...>>;
           State2: @ alternation SymmetricCoroutine
             <<... ObjectDescriptor ...>>;
           State3: @ alternation SymmetricCoroutine
             <<... ObjectDescriptor ...>>
        do State1[] -> Run (* activate State1 *)
        #)
  do STD
  #)
```

Figure C.5: State Transition Diagram

to the caller by executing **suspend**. There is another kind of coroutine called *symmetric coroutines*. A symmetric coroutine explicitly calls the coroutine to take over. In the modelling of a state transition diagram in BETA we need symmetric coroutines, because a state transition among other things explicitly defines the next state. Symmetric coroutines can easily simulated in BETA. The SymmetricCoroutineSystem as described in [BETA 87] and [BETA 90] is used. Each alternation class is prefixed with a SymmetricCoroutine superclass, that among other things provides the **resume** operation as shown in Fig. C.6.

```
ControlAandB: @ concurrent Transformation
  (#
      STD: @ alternation SymmetricCoroutineSystem
        (#
           State1: @ alternation SymmetricCoroutine
             (# Signal1: @ procedure (# with some Transformation do #)
             do cycle
                (#
                do <? Signal1;
                   TransformationA >? Enable;
                   State2.Resume
             #) #);
           State2: @ alternation SymmetricCoroutine
             (# Signal2: @ procedure (# with some Transformation do #)
             do cycle
                (#
                do <? Signal2;
                   (if (var1 = value1)
                    //true then
                      TransformationA >? Disable;
                      TransformationB >? Trigger;
                      <<ProcessIdentification>> >? Signal3;
                      State3.Resume
             if) #) #);
           State3: @ alternation SymmetricCoroutine
             <<... ObjectDescriptor ...>>
        do State1[] -> Run
        #)
  do STD
  #)
```

Figure C.6: Signals and Transitions

ISSN 0105-8517