The evaluation of suitable Object-Oriented Development techniques in order to make a prototype Point Of Sales Terminal on a personal computer

van Bezooijen, H.

Award date:
1990

Eindhoven University of Technology
Department of Electrical Engineering
Digital Systems Group (EB)

Schlumberger RPS
Development and Engineering
Industrieweg 5, Bladel (Holland)

The evaluation of suitable Object-Oriented Development techniques in order to make a prototype Point Of Sales Terminal on a personal computer

H. van Bezooijen

Date : may 1989 - februari 1990

Supervisors : ir. J.M.H. Dassen
Schlumberger RPS
: ir. A.C. Verschueren
Eindhoven University of Technology

Supervising professor : Prof. ir. M.P.J. Stevens
Eindhoven University of Technology

ABSTRACT

In order to obtain an evaluation of Object-Oriented Developping Techniques Schlumberger RPS wanted to build a prototype Point Of Sales Terminal based on these techniques.
To accomplish this, at first a study was made of the concepts of Object-Oriented Development Techniques that are currently found in literature. This evolved in the implementation of two possible suitable techniques found in the literature part.

The approach for studying the literature was an Object-Oriented Development path analogous to the more or less 'usual' development path - analysis, design and implementation.

The literature study has lead towards a development path for Object-Oriented Analysis and Design that is primarly based on methodologies like Synthesis and Object-Oriented Domain Analysis. To these methodologies we added some features of Hierarchical Object-Oriented Design (ADA environment) and of the Shopping List Approach, advocated by Betrand Meyer.

The Object-Oriented Analysis is primarily based on an Information Model made by Shlaer - Mellor, to which we added class hierarchies and an External Events Model. The External Events Model is the starting point of the analysis.

The Object-Oriented Design is centered around a methodology developed by Schlumberger. This methodology is extended with an Object Diagram and a methodology that links the design to the analysis. This linking is a mapping of events, found in the analysis, to methods in the design.

An element of the development path that needs further investigation is concurrency modeling.

The Object-Oriented Programming language used for the prototype POS Terminal implementation is Smalltalk.

CONTENTS

1. INTRODUCTION 1

PART I: LITERATURE 2

2. OBJECT-ORIENTED PROGRAMMING 3
   2.1 Terminology and Definitions 3
      2.1.1 The Object 3
      2.1.2 The operations of the object 4
      2.1.3 Class hierarchy and inheritance 4
   2.2 Smalltalk: an Object-Oriented Programming System 7
      2.2.1 Object-oriented qualification 7
      2.2.2 Smalltalk 7

3. OBJECT-ORIENTED ANALYSIS/DESIGN 9
   3.1 The need for Object-Oriented Analysis/Design 9
   3.2 The development path 12
   3.3 Fundamentals of Object-Oriented Development 13

4. OBJECT-ORIENTED ANALYSIS/DESIGN METHODS 15
   4.1 Hierarchical Object-Oriented Design (HOOD)..... 15
   4.2 An integration of Objects in Structured
      Analysis and Design 20
   4.3 The Shopping List Approach 21
   4.4 Object-Oriented Analysis and Design 22
      4.4.1 Synthesis 22
      4.4.2 Object-Oriented Domain Analysis (OODA) 24
   4.5 Evaluation of the methods, ... 26
      4.5.1 Comparing the methods 26
      4.5.2 Conclusions 27

PART II: AN OBJECT-ORIENTED IMPLEMENTATION 30

5. THE POINT OF SALES SYSTEM 31
   5.1 The system's hardware 31
   5.2 The system functionality 33
   5.3 The user interface 35

6. OBJECT-ORIENTED ANALYSIS 37
6.1 The strategy 37
6.2 The Analysis 39
6.2.1 The External Events Model 39
6.2.2 The Information Model 41
6.2.3 The State Model 47
6.2.4 The Class Hierarchy 51
6.3 Results 52

7. OBJECT-ORIENTED DESIGN 53
7.1 Design considerations 53
7.2 The design 54
7.2.1 Step 1 (conceptual design) 54
7.2.2 Step 2 (detailed design) 56
7.3 Review 58

8. OBJECT-ORIENTED IMPLEMENTATION 59

9. CONCLUSIONS AND SUGGESTIONS 60

10. LITERATURE 62

APPENDIX 1 Context Diagram and External Events List 65
APPENDIX 2 Entity Relationship Diagrams 71
APPENDIX 3 Entity and Relationship Descriptions 76
APPENDIX 4 State Models 116
Communication Model 120
Events List 121
APPENDIX 5 The Class Hierarchies 129
APPENDIX 6 Method descriptions 130
APPENDIX 7 Object Diagram 174

1. INTRODUCTION

Koppens Schlumberger is a manufacturer of gas station equipment. This equipment can vary from a single dispenser to a complete gas station outfit.
An important part of this equipment is the Point Of Sales (POS) Terminal.
The POS Terminal functionality consists of:
(1) hardware control of dispensers, credit card and banknote acceptors, modems
(2) cash register functions
(3) management functions, like stock control, financial reports
(4) communication with a host for transaction clearance
(5) software downloading

Nowadays a POS Terminal has grown to a multi-processor system with several Mbytes of application software. Currently used techniques for the software development are based on the structured analysis and design methods (Ward-Mellor and Hatley-Pirbhay).
The applications developed at the Development and Engineering Department of Koppens Schlumberger are systems with only a certain basic functionality.
This functionality is extended and adapted to the divergent wishes of the clients in different countries.
This approach demands software requirements like reusability and extendability.

Over the last years a relatively new method of programming has come up, Object-Oriented Programming (OOP).
OOP is developed to support software requirements such as reusability, extendability and understandability. The current availability of analysis and design techniques for Object-Oriented Systems is limited. At Koppens Schlumberger one was interested in an object-oriented approach for software systems development.
The graduation subject is described as:

> *Make a small (working) prototype POS Terminal on a standard PC (personal computer). To accomplish this study the literature for suitable Object-Oriented Development Techniques and evaluate these.*

This report is divided into two main parts. The literature part includes the concepts of Object-Oriented programming, abstracts of the literature study and conclusions for the direction(s) to follow to obtain an Object-Oriented Design.
The second part deals with the implementation of two possible suited techniques, evolved from the first part, for this project. At the end of part two, recommendations will be given for further development of Object-Oriented systems.

2. OBJECT-ORIENTED PROGRAMMING

In the first section of this chapter an overview of the thinking and meaning behind Object-Oriented Programming (OOP) is introduced. Furthermore the most important terminology used in this programming will be explained and defined.
The second section deals with an Object-Oriented language, Smalltalk. This language will be used to implement our prototype POS terminal.

2.1 Terminology and Definitions

2.1.1 The Object

Everything in OOP focusses on objects. One can say that an object is a model of a "real world" entity. One can model a house, a car etc. Our problem domain is a subset of this "real world". How an object is modelled depends on the use of this object. If for example the criterium for a airplane is; it flies with an engine, then every flying thing with an engine is an airplane.
The object is fully characterized by its (internal) state and its interface. The internal state is described with what is called the attributes of the object.
In our example an airplane can have attributes like number of passengers, maximum fuel etc. The attributes are the private data of the object. The state of an object is comprised of the current values of the attributes.
The interface of an object consists of a set of actions.
We want objects to perform these actions. To accomplish this we need a set of operations that act upon the object.
Operations that could act upon an airplane are *take-off, land, take-passengers, fly-a-direction etc.*
These operations don't tell us how the object performs them. Neither do we know what and how its attributes are involved.
With help of these object descriptions we try to give a summary of what an object is.
An object is a combination of data (attributes) and operations, it has a hidden internal structure (a state), it is accessible through a set of requests (operations) - the object's interface - and has identity to distinguish it from other objects.
The hidden internal structure is the basis for information hiding. Information hiding is used to conceal properties that are unimportant to the user of a particular object. It also plays an important role in the analysis and design of object-oriented systems.

2.1.2 The operations of the object

The operations of the objects mentioned in the previous section can be viewed as services which are offered to the users of an object.
The object has two kinds of operations, public and private.
The public operations are the offered services which is called the object's interface. The private operations are solely used by the object itself.
The terms operations, services and methods (Smalltalk terminology, see section 2.2) are synonyms.
The set of required services that an object needs of other objects, to fulfill certain actions, is called the required interface.

The interaction mechanism between objects is a message passing model.
Message passing has a certain security advantage. Because the objects knows what methods it has to offer it will refuse an unknown message or it can reject a message that carries a wrong data type. The response to a known message depends on the internal state of the object.

A method can have zero or more arguments. The receiving object knows how many arguments it has to expect. Type checking of the received arguments can be done at compile time or at run time, depending on the used language.

2.1.3 Class hierarchy and inheritance

To organise the objects we need a kind of a catalog to put them in. These catalogs are called classes. A class defines the general characteristics of a group of objects. For instance the classes person and dog belong to class mammals. The person 'John' is an instance of the class man. From this moment on we refer to a class if we describe the general features and we refer to an object if a specific instance of a class is mentioned.
The further one extends the class hierarchy the more specific a certain characteristic of a class attribute is specialised. The subclass person can be divided into different subclasses like man and woman.

From an opposite point of view it is possible to grow to classes at a 'higher level'. This is a form of generalization of class characteristics. We can attach the class students as a subclass to the already mentioned class person. This process of generalization is actually a modelling for data abstraction.

To formalize the concept of class we can use abstract data types (ADTs). Combining the concept of data abstraction and classes it is possible to get an explicit (Meyer [3]) description of classes of data structures. This leads us to the theory of abstract data types.
This theory is based on the fact that abstract data structures are specified by their available services and properties of these data structures and not by their implementation. Meyer [3] gives a specification of abstract data types. This includes a formal syntax and semantics specification of the abstract data type.
The abstract data type is a class of data structures described by its interface. With these properties we add another element to the object description of section 2.1.1:
Objects are encapsulations of abstractions.

In Figure 2.1 we see a part of the class hierarchy of the class magnitude (Smalltalk [5]).

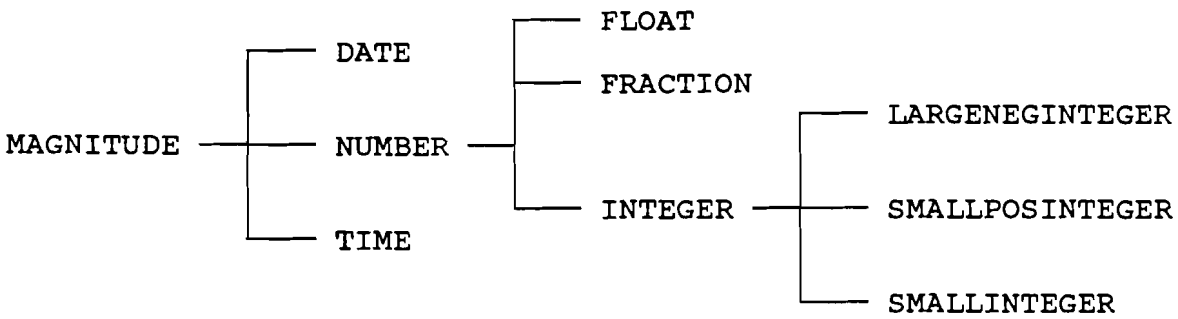


FIGURE 2.1 MAGNITUDE CLASS HIERARCHY

In Figure 2.1 we see that classes are identified by a name. This name is a short description of what the class represents.
Objects are instances of a class. They are represented by identifiers.

There is a second element to putting objects (classes) into class hierarchies: inheritance.
Inheritance is the property of ascribing features of a class to its subclasses. The inherited features can be external (methods) and internal (the different states).
Taken the example of the class hierarchy of MAGNITUDE of Figure 2.1 it is clear that a method 'add' belonging to the class NUMBER is inherited by the class INTEGER. Only the implementation differs. Another technique of reusing methods is to redefine them for the sub classes.

If a class inherits properties from its ancestors, like INTEGER inherits certain properties from MAGNITUDE, it is called single inheritance.
Multiple inheritance - inheriting properties of different other classes - is also possible but this problem area is out of our scope, so we will not go into this part of the object-oriented domain.

The features abstraction, class hierarchy and inheritance gives the user the possibility to reuse code.
Already defined messages in a higher rank of the class hierarchy can be freely used by sub classes if their description permits it.

It is also possible to send the same messages to different classes. We could think of an object STRING that is sent the message 'add' with as an argument another string. In this case we would expect a new string that is the union of the two strings.
If two classes respond to the same message, this is called 'operator overloading'. The parameters of these messages need not be the same. When two classes respond in a meaningful (but possible different way) to exactly the same message, this is called polymorphism.

2.2 Smalltalk: an Object-Oriented Programming System

2.2.1 Object-oriented qualification

Object-oriented programming is defined as programming in a language where objects form the basic structures while the language must support the object-oriented paradigm.
The object-oriented paradigm is a general term for a set of characteristics that is used to define and describe the terminology of objects.
These characteristics are abstraction, encapsulation, class hierarchies and polymorphism.

When we use the term objects we mean objects that are qualified by the above description.
There are a lot of languages with objects as the major structures but only if all the elements of the object-oriented paradigm can be applied to a such a language it is qualified as an object-oriented language.
Wegner [18] described language qualifications from object-based (ADA) via class-based (CLU) to object-oriented languages (Simula, Smalltalk).
Meyer [3] distinguished seven steps that lead to true object-orientedness. Only systems (languages) that come to the last step may be called object-oriented.
These steps cover beside the in this section already mentioned object-oriented features:

- automatic memory management
- dynamic binding
- multiple and repeated inheritance.

Except for the multiple inheritance, Smalltalk satisfies all these properties mentioned by Meyer.
Smalltalk is the language chosen to implement our prototype POS terminal.

2.2.2 Smalltalk

Smalltalk is more than just an object-oriented language, it gives the user a complete programming environment.
This environment supplies a user interface that contains multiple windows, icons, text and graphics windows, menus, mouse use and an extended class hierarchy.

Smalltalk is called a 'truly' object-oriented programming language. The classes, the methods and their responses are objects.
Classes are instances of higher-level classes, the "meta-class". There is one root class, called **OBJECT**.
The objects communicate through messages, which consist of an 'method selector' and, possibly, parameter objects.

The reception of a message invokes the evaluation of the 'method' which matches the 'method selector'. The method is a piece of executable Smalltalk code, which itself can send messages to other objects and manipulate (amongst others) the instance (private) variables of the receiving object.
When evaluated, a method always returns a result object to the sender of the message.
Whenever we talk about message or method sending to objects we actually mean the above described meachnism.

Smalltalk distinguishes three forms of messages: unary, keyword and binary.
Unary messages are messages with no arguments. A Smalltalk expression like: **'string' size**, will return the number of characters of the object **'string'**. The message is **'size'**.
Keyword messages are messages with one or more arguments.
The expression:**'strang' at:4 put:$i**, will replace the character 'a' by the character 'i'.
Binary messages are messages which are evaluated from the left to the right.
The expression: a + **b * c**, means actually (a + b) * c.
It is also possible to send more than one message to an object at the time by using a semicolon to separate messages.
The expression: **set add:2 ; add:3**, shows how two items (2,3) are added to an object **set**.

Smalltalk gives the possibility to define class methods and class variables. Smalltalk is case - sensitive.
The user is provided with a basic library of classes that has enough capacity to develop his own system.
The private variables of objects are called attributes. Smalltalk has no protection against the use of private methods, it is the responsibility of the user to make a correct use of the private methods.

Smalltalk is a dynamic typed language. The type checking is done at run-time.
Furthermore, it is equipped with an automatic garbage collector. The user doesn't have to worry about non-used objects that occupy unnecessary memory.

It is already mentioned that Smalltalk is more than a programming language.
The interaction between the Smalltalk system and the user is achieved through an extended user interface that is completely window driven.
It is easy to extend the system with new applications made by the user or with commercially purchased applications.

3. OBJECT-ORIENTED ANALYSIS/DESIGN

Although Object-Oriented Programming (OOP) and has become an area of great interest in recent years, object-oriented development methodologies are scarce.
Presently, the analysis for these designs is done with techniques introduced by Ward-Mellor [7], Hatley-Pirbhay [6] and Jackson [8]. As a result most of the CASE tools are based on these techniques and have strong similarities.
To use OOP as a new implementation strategy doesn't imply new or improved analysis and design techniques.
One of the first papers about what is called an object-oriented development (Booch [19]) suggested already that design methods would have to adapt to this environment.
In this chapter we describe some, what are believed, general advantages of such an object-oriented development environment.

An object-oriented development is the path that is gone through from the idea of a system to its object-oriented implementation. We shall call this trajectory, the (software) development path.

3.1 The need for Object-Oriented Analysis/Design

Programs are growing in code size and complexity. Also, there is an increase in speed of successively updating versions of programs. Software requirements like robustness, speed, correctness and reliability are not enough to fulfill these huge and fast changes.
Adapting the software development process, with respect to the fast changes, demands a more flexible approach to software design.

Methods that cope with the increasing complexity of software systems are based on an increased use of predesigned software libraries and/or are based on decomposing the system into smaller subsystems. Decomposition criteria that are in use are event partitioning and functional partitioning.

The methods based on decomposition can be characterized as Top-Down approaches (see [6,7,8]).
A Top-Down method that came recently in use is the analysis and design method described by Hatley-Pirbhay [6]. Their method is focused on functional decomposition. These Top-Down approaches have several advantages. Besides the fact that they are a well established and described discipline, they encourage a logical development, tackle the system complexity by making smaller subsystems (the functionally decomposed subsystems) and are supported by various CASE tools.

These advantages merely apply for slow evolving systems with a strong centralized development.
However, as we mentioned above, systems are changing rapidly. As a system evolves through its lifetime, it can be updated, replaced or even become redundant.

One of the most volatile parts of a system are the interfaces between the system and its terminators. These interfaces determine the 'top' of a system. When using a (functional) top-down analysis and design approach in such a rapidly changing system (Hatley-Pirbhay [6]), there are some serious drawbacks.

Most of these drawbacks are due to the fact that during the evolution of a system the criteria for system decomposition, the functions, are changing also.
The designer must try to avoid a premature binding of the functional relations. In this he/she can prevent that changes in the "top" lead to many changes in submodules and a lot of time and effort spent to the analysis and design is wasted.

A second important drawback of top-down design is that the interesting data structures are global. Any change in representation of these data structures will effect all the submodules that need these particular data structures.
Using global data structures has another disadvantage. Breaking up a system in order to decrease the complexity one focuses on for this decomposition important properties. Other not so important properties stay hidden.
This process of information hiding is as good as impossible when using global data structures.

The third drawback of the functional top-down approach is that it doesn't support data abstraction, which can be very useful in describing and designing software systems.

Other drawbacks mentioned in literature are the mismatch between the software model and the 'real world' and the lack of appropriate approaches to concurrency in top-down design.

It is obvious that with all the disadvantages mentioned above, the current (functional) top-down analysis and design methods are not the solution in reducing the system complexity, and hardly offer any methods for reusing code.

What we need is a new criterium for system decomposition. This criterium must be based on the most persistent elements in a software system: the data structures which from now on we shall call objects or entities.
We will use the description for objects as given in chapter 2 when referring to enties or objects.

Using abstract data types as the basis for system decomposition is believed to contribute to a more modular design and the reuse of code. In this way we obtain objects as the basis for system modularity. Modular building of the system must be based on cooperation between objects.

What about the other mentioned drawbacks of top-down development ?
Looking again at the description given for objects in chapter 2, it is not so difficult to understand that an object is a natural basis for concurrency.
Changes in object structures tend to be more local compared with the global changes of the top-down approach.
The concept of information hiding is also encapsulated in the description and use of objects.
An object knows what it must perform through its defined operations. The same argument applies for changes in data representation. If the internal representation of data changes, the external representation of this data inside the object does not change.

This leaves us with the matter of code reuse.
Only using objects as the decomposition basis itself does not imply reuse of code.
Putting the objects into abstract classes and include the feature of inheritance to the class structure makes code reuse possible.

We made a summary of a few of the expected advantages when using the object-oriented paradigm in system development.
The question that now arises is whether this different approach still follows the same analysis and design path that is currently used by the 'traditional' methods and what parts of these methods can be applied in an object-oriented environment.
In the next paragraph we will describe what development we have in mind when referring to the development trajectory.

3.2 The development path

The process of developing a software system is a long trajectory. Roughly one can distinguish the following phases in such a trajectory: user requirements negotiations, analysis, design and implementation.
We call this trajectory the development path.
In recent years this path has been refined by improved and integrated analysis and design methods.
The growing availability of CASE tools for these methods made the development process more controllable and verifiable.
The growing interest in object-oriented systems caused a different approach for the analysis and design phase.
At first most attention was paid to the design phase [16,19,21]. More recently there have been developments in the analysis area [4,10,17,23].
The goal is a fully object-oriented development path. Schematically depicted in Figure 3.1.

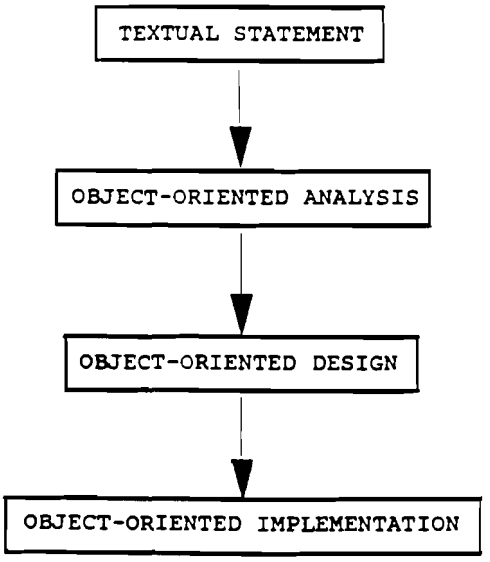


FIGURE 3.1 AN OBJECT-ORIENTED DEVELOPMENT PATH

Some of the above mentioned methodologies try to make a new development trajectory, while others are merely involved with one of these phases, or try to use existing methods for an object-oriented environment.
In chapter 4 we will summarize the features for four of the most interesting directions of research there are at this moment in literature, concerning object-oriented development.

3.3 Fundamentals of Object-Oriented Development

Booch [19] was one of the first who used an object-oriented approach for the decomposition of a software system. Although this approach was primarily intended for an ADA environment, the properties of his methodology can be seen as a general starting point in the object-oriented developments area.

The modelling is focussed around real world objects.
An object is defined as an problem domain entity with the following characteristics:

- has an identifier (a name)
- has state
- requires and delivers services for other objects
- has restricted visibility of and by other objects
- is an instance of a class
- may be viewed by its specification or by its implementation

Abstraction and information hiding are the bases of object-oriented development. The development of Booch [19] consists of five major steps:

1. Find the objects and their relations from the problem domain. To accomplish this, model them according to their role in the problem domain. Booch uses a kind of data flow diagrams to find the objects.
   Establish classes of found objects.
2. Identify the services (operations) delivered and used by an object. This description includes also the constraints upon the use of the services from an object.
3. The establishment of the visibility of the object in relation to other objects.
   This step places the object in a layered structure.
4. Establish the interface of each object. These are the static dependencies between the objects.
5. Implement the object.
   This step involves, if necessary, the decomposition of an object into other objects or - in the contrary - the composition of an object from other objects.

The object-oriented development steps described by Booch are mostly the design and implementation parts of the development path of Figure 3.1.
A separation between an object-oriented requirement and analysis phase preceding the design and implementation phase isn't made in this method.
The different layers of abstraction that come across the development are collections of objects and classes that are called subsystems.

As benefits of an object-oriented approach are mentioned:
- reduction of the total life-cycle software costs
- the implementation of robust systems
- The modeling of real world entities

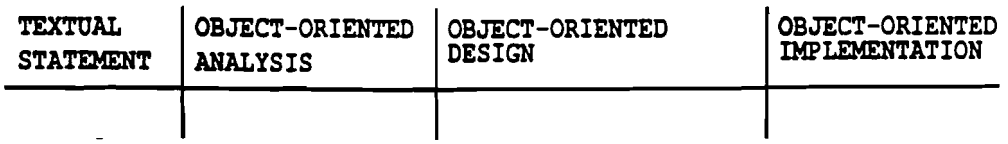
Booch doesn't mention inheritance as a tool for reusability, because ADA doesn't support inheritance.
Although the object-oriented development description of Booch isn't complete, it was a first step for building a new software development path.

4. OBJECT-ORIENTED ANALYSIS/DESIGN METHODS

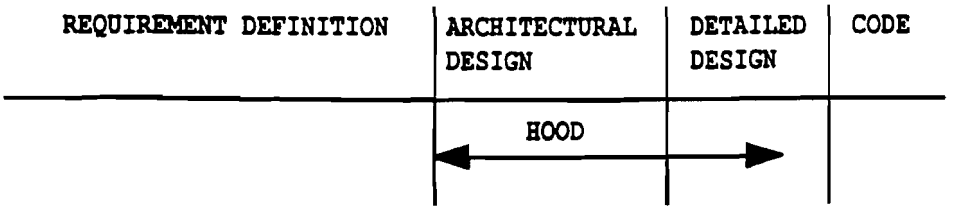
In this chapter a summary is given of four, in my opinion, important directions of research in the field of object-oriented development.
In the last paragraph of this chapter we make a comparison between the summarized methods and give an overview of what we think are useful parts for developing a fully object-oriented development path.

4.1 Hierarchical Object-Oriented Design (HOOD)

HOOD [16] is originally developed for an ADA environment. The ADA environment is more then just a language. We are only interested in the object-oriented features of ADA and HOOD. Because ADA doesn't include the inheritance mechanism it is clear that strictly following the definition for object-oriented languages (chapter 2), ADA is not an object-oriented language.
However, as we shall see, HOOD has some interesting elements which can be used in object-oriented modeling.
In Figure 4.1 The place of HOOD in the software development path is shown.



A. OBJECT-ORIENTED DEVELOPMENT PATH



B. HOOD LIFECYCLE

FIGURE 4.1 HOOD IN THE SOFTWARE LIFE-CYCLE (from [16])

Figure 4.1 indicates the potential use of HOOD in our development trajectory of section 3.2. In Hierarchical Object-Oriented Design the objects are the basic elements of modularity.

The implementation phase of the software development path crosses, what in HOOD is called, the detailed design phase. The pseudo code given in the detailed design is almost an implementation in object-oriented terms.

HOOD deals with problems like:
- Entity abstraction
- What is the relation between the objects
- Object decomposition
- What kind of control must be applied
- What kind of (graphical) representation is needed

An object in HOOD is a data package with related operations (procedures). It is a model of a problem domain entity.
HOOD enforces a structurized design onto a system.
This structure is characterized by three groups of principles (HOOD [16]):
- Abstraction, information hiding and data encapsulation
- Two (system) hierarchy structures (see below)
- Control structures

The first group of principles are characteristic for the object-oriented paradigm.
The object is described by its static and dynamic properties. The static properties are a description of the object's interface and its internals. The static properties also include a list of operations required from other objects. This description and use in HOOD differ little from their general application (chapter 2).
The dynamic properties describe the communication (protocol) between objects.
What is interesting are the other two principles, hierarchy and control.

Hierarchy in HOOD is twofold.
Every object can be decomposed into other (sub)objects. The decomposed object is called the father and the sub-objects are referred to as the children. This is called the **parent-child relationship.** An operation wanted from the father is actually executed by one its children, if it has children !
The second form of hierarchy is caused by the **use relationship.** An object can **use** the features of other objects. If the system is split into layers, the used objects are placed in junior (lower) layers as compared to the using objects, which are placed in the senior (higher) layers. How many layers a system needs depends on the system complexity. An object in a lower layer doesn't need to know about the objects in its senior levels. It only provides operations to them.

"Unintelligent" objects, for instance, databases will be placed in the lower layers while objects representing major sub systems are placed the higher ones.
Figure 4.2 shows the two relationships graphically.

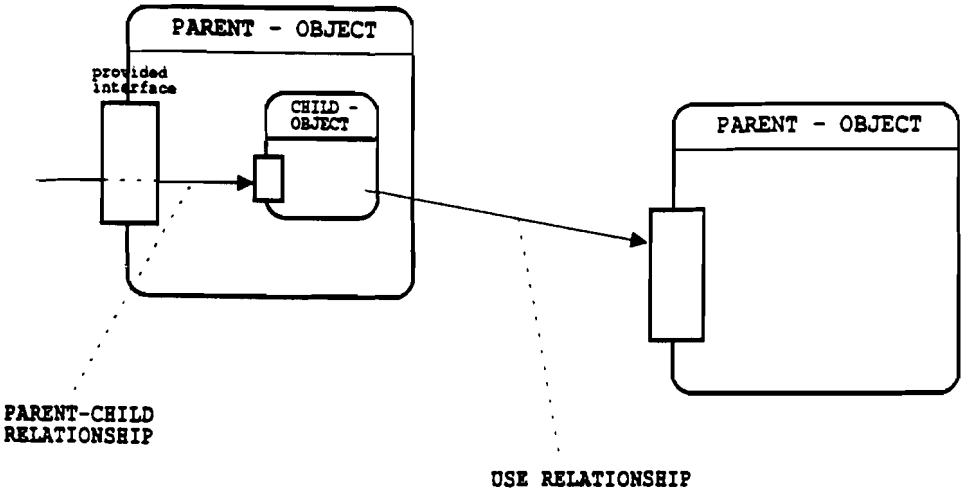


FIGURE 4.2 RELATIONSHIPS IN HOOD

Control deals with the level of parallelism. If the object that uses another object stays "active", there will be a new control flow created in the used object. This new control flow can act in parallel to the original one.

Control structures in HOOD and the **use relationship** are strongly coupled. Control structures are a description of the dynamic properties of the objects. It describes how the **use relationship** is executed, when looked at from the using object.

An execution can be performed sequentially or in parallel. In the second situation the response of the operation request depends on the internal state of the used object: the operation is said to have constraints and a new control flow is generated inside the used object. Based on the two execution types, HOOD divides the objects into two groups, active and passive objects. Passive objects have no form of parallelism at all. Active objects must have at least one constraint operation. In HOOD we make a distinction between two kind of operation constraints:

- The internal state of the object, the execution of the operation depends on the sequence of the previous operations.
- The type of the operation request, for example time-out operation types.

To avoid circular control it is forbidden for passive objects to use active objects. Of course it is still possible to get a deadlock, when two active objects are using each other !

Another way to look at the layered object model that is found in a HOOD like environment, is to consider it as layered virtual machine [21]. Between the layers there is a virtual machine interface. The set of objects in a layer satisfy a certain level of abstraction and information hiding. This view of the object model and its representation matches the hierarchy structures discussed above.

HOOD [16] advocates a designing trajectory started with a top object that is decomposed into sub objects. The process of decomposing continues until non decomposable (terminator) objects are found. This qualifies HOOD as a top-down design method. Seidewitz and Stark [21] also mention a splitting, not a decomposing, of the top as a possibility when the strong centralization causes bottlenecks.

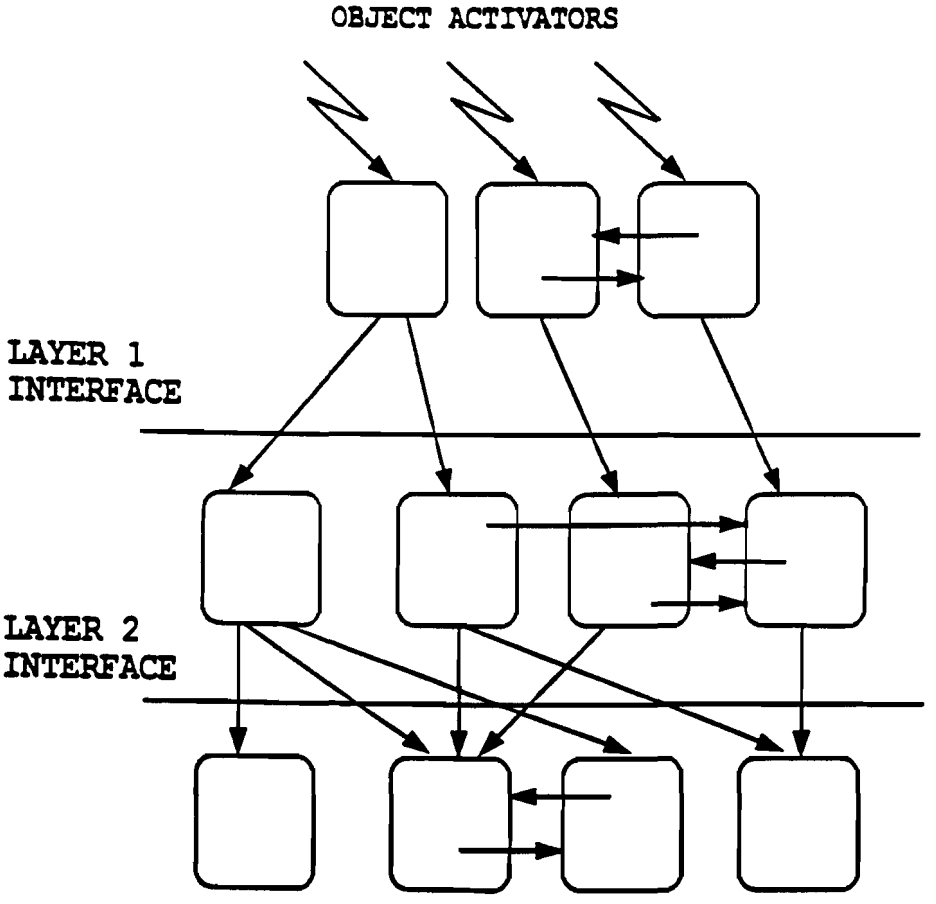
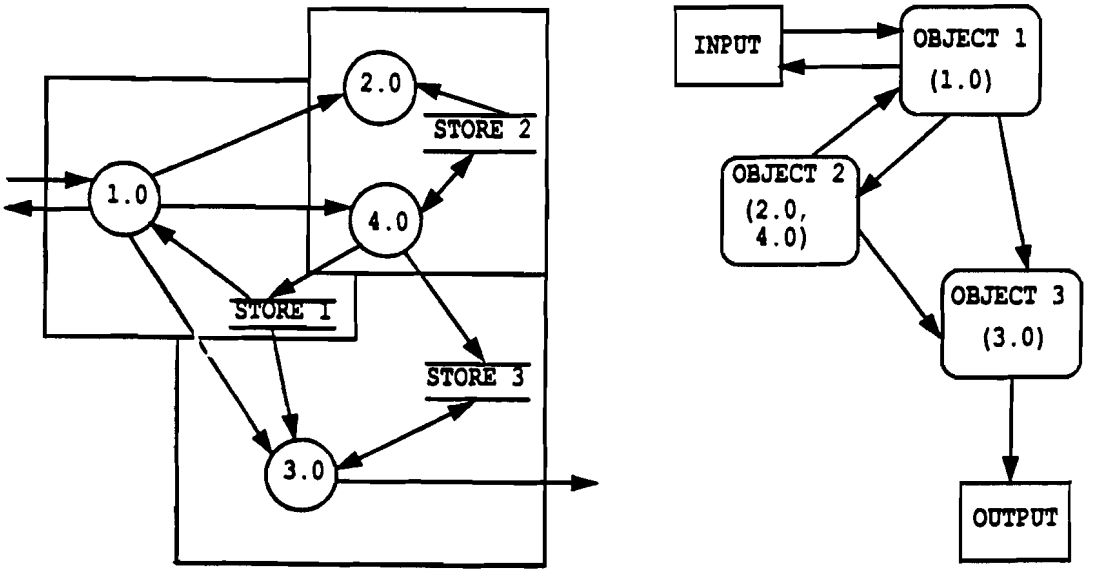


FIGURE 4.3 A HOOD OBJECT DIAGRAM

This causes multiple control and as a consequence there is concurrency at this level. As Meyer [3] stated, **real systems have no top.** In Figure 4.3 we see an example of the two hierarchies put together in a layered object model. One notices that in this object diagram the parent-child relationship and the provided interface are not shown. Object activators are kind of **use relationships** for active objects.

HOOD as the name refers to, is primarily targeted towards the design phase. Stark and Seidewitz [21] mention abstraction analysis preceding the design phase. Abstraction analysis is a transformation from structured analysis to an object-oriented design.

In Figure 4.4 we see such a transformation. The rectangles in the Data Flow Diagram are the identified objects. The heuristic for finding these objects is simply the detection of the data stores combined with the operations acting upon the data stores. Disadvantages are that one doesn't know whether the obtained interfaces are minimalized or not and if of all the proper-ties of objects can be applied to the in this way obtained objects.



SA LEVEL 0 DATA FLOW DIAGRAM OBJECT DIAGRAM

FIGURE 4.4 MAPPING FROM SA TO AN OBJECT ORIENTED MODEL

A look alike methodology is discussed in the next section. HOOD [16] refers also to SA methods but doesn't mention a mapping from analysis to design.

4.2 An integration of Objects in Structured Analysis and Design (SA/SD)

The SA/SD analysis and design method that is based on process decomposition. It is a purely top-down methodology. Elements used at the SA/SD process are context diagrams, entity relationship diagrams (ERD's), data flow diagrams (DFD's), structure charts and transformation schemes.
An integration of the his method with an object-oriented development is described by Ward [17]. His object-oriented development and SA/SD have some similarities. Both decompose a system into smaller sub systems, both use ERD's.

The basic difference is that the object-oriented approach uses events for the communication between objects. A list of events and a relation between objects can be used to identify the objects. This leads to the ERD's.

SA uses also ERD's, but the entities found here are nothing more than data stores. Ward [17] loosens this constraint in order to include all objects and to show a bond between respectively the SA and the object-oriented approach.
In these expanded ERD's abstract data types and inheritance (through subtyping) are identified.

The integration of the two techniques is done at a high level of detail. A high level transformation scheme together with ERD's is used to find the objects or group of objects.
Control processes of the transformation scheme become sort of 'control' objects.
Transformations or groups of transformations are transformed into respectively objects and groups of objects.
With the methodology of Ward [17] we come to a similar object partitioning as we have seen in section 4.1.
HOOD can also start with SA. Although the result may look alike, the road to it isn't.

Ward [17] makes an explicit use of the ERD and its heuristics to detect the objects. For instance to find subtypes that can be used for inheritance. In the contrary, we do not find inheritance in HOOD.
Ward believes that there is no fundamental difference between SA/SD and object-oriented design.

4.3 The Shopping List Approach

This "method" advocated by Meyer [3] is more a philosophy then a complete design method.
The shopping list approach is a description for the relation between the designers and users of objects.
The designer is a provider of classes of objects, object descriptions and a list of the various operations belonging to the objects.
The designer will postpone as long as possible the sequence specification of the operations. This makes using the objects more flexible.
An object is used by means of its messages. When a user or a designer of objects wants to assemble or design new objects he will have to make an arrangement of the sequence of operations of the used objects in order to fix the interface of these new objects.
At this point of the development cycle of object-oriented software the top-down approach and the bottom-up approach interfere.
This method of designing objects is opposed to methods where premature binding of the order of relationships is obliged, like the data flow diagrams used in top-down design.

The Shopping List Approach has the disadvantage that designers are tempted to the excessive use of classes. This can lead to a lot of code overhead.

Related to the Shopping List Approach is the idea that software components can be standardized and used in the same way as it is done with IC's, composing modules with already made library elements.
Ledbetter and Cox [20] call these reusable software components Software-IC's.
They can be compared with the classes of objects as discussed in chapter 2.
The basic idea behind building Software IC's is object programming through messages.
Requirements needed to build and use reusable components are:

- encapsulation
- inheritance
- dynamic binding
- direct mapping from real world functions to implementation

These four requirements are covered by the object-oriented paradigm.

4.4 Object-Oriented Analysis and Design

Recently two methods are published that have an object-oriented view as a starting point. These methods are Synthesis [23] and Object-Oriented Domain Analysis. In this section we give a summary of these methods.

4.4.1 Synthesis

The base for Synthesis [23] is information modeling. It uses the Context Diagram, Events Lists, ERD's and the State Model from the Information Model.
Other components are an object class hierarchy, event partitioned neighborhood diagrams and supporting textual statements. Central to the Synthesis Model is event partitioning. Neighborhood diagrams are used to show the detection and response for one, or a related group of events. In figure 4.5 a simple example of a neighborhood diagram is shown.

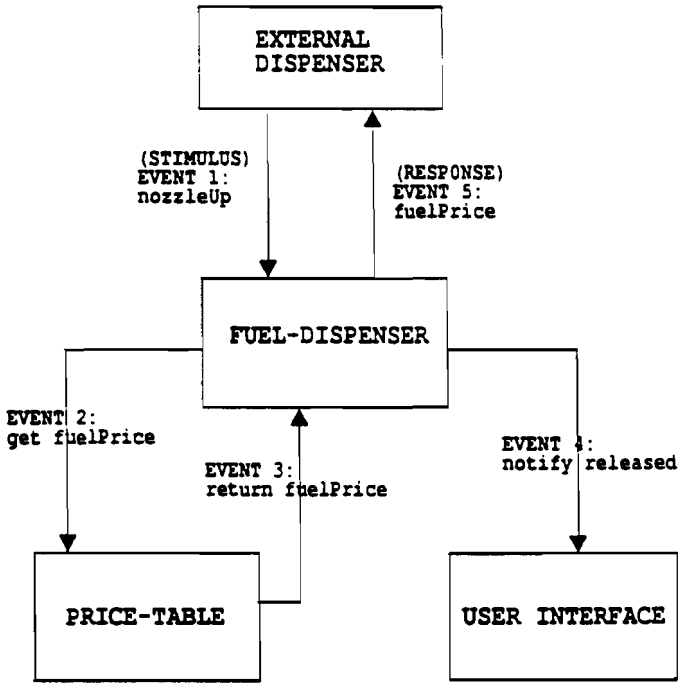


FIGURE 4.5 A NEIGHBORHOOD DIAGRAM EXAMPLE

The diagram from figure 4.5 is a simple example of the situation at an EXTERNAL DISPENSER where a customer takes a nozzle. The FUEL-DISPENSER gets a FUEL-PRICE from a PRICE-TABLE and releases the EXTERNAL DISPENSER by sending the fuel-price to it. Furthermore the USER INTERFACE is notified that a FUEL-DISPENSER is released.

The Synthesis Model is built into two steps.
First one makes in parallel an Augmented Information Model, a Context Diagram and an (External) Events Dictionary.
The Augmented Information Model is an extension of ERD's. They contain information about the entities and their relationships. The goal of this step is to centralize information around the entities of the problem domain.

We now have, what is called, an Object-Oriented Essential Model. This model is the basis for the next step.
In this step relevant objects are generated for entities, relationships and event types found in the Essential Model. This must be done using the properties of objects, putting them into classes and make use of inheritance. The objects are supplied with initial methods, attributes and state diagrams. Using the Events List a Neighborhood diagram is made for every event. Eventually all the neighborhood diagrams are put together in layered graphical representation.

Synthesis uses many already known tools. The neighborhood diagram is actually a DFD for object-oriented systems.
The analysis to design mapping part of Synthesis isn't very well worked out. This is probably done because Synthesis is meant to be an object-oriented analysis for a lot of different implementations. The boundary between analysis and design is probably the step of the creation of neighborhood diagrams.
The design consists of the mapping of neighborhood diagrams to an object-oriented programming language.

4.4.2 Object-Oriented Domain Analysis (OODA)

Object-Oriented Domain Analysis [4,10] has a very similar approach to analysis as Synthesis.
OODA is based on building an integrated set of three formal models, the Information Model, the State Model and the Process Model.
To build these models one follows prescribed rules and a specific order of integration.
In Figure 4.6 the sequence and the integration of the three models is shown.

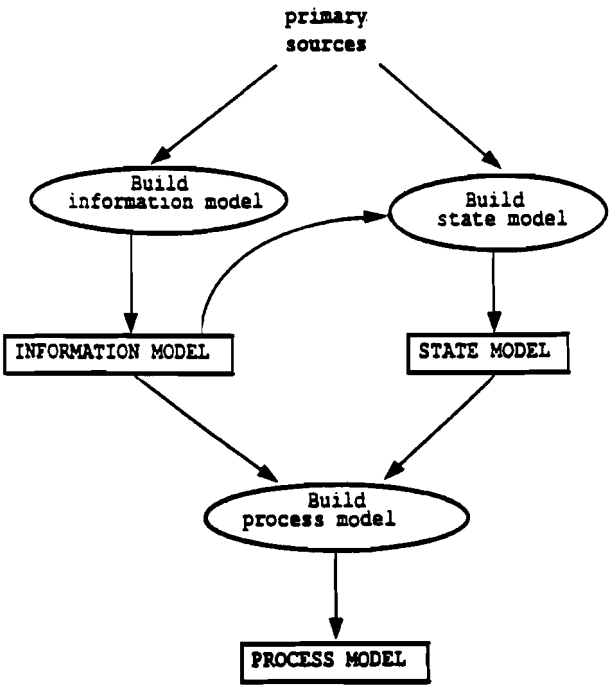


FIGURE 4.6 ANALYSIS PHASE OF SYSTEM DEVELOPMENT (from [10])

The graphical form of the Information Model is a kind of extended ERD.
The extension pays attention to the kind of relationship there exist between the entities and it formalizes them. Also It formalizes the description of the objects. Further it pays attention to the typing of objects and the construction of super and sub types.
The Information Model emphasizes on the relation between the problem domain entities. The documentation for every individual information model has a textual and a graphical part. It is a passive description of the problem domain.

After finishing the Information Model (see Figure 4.6) the dynamic behaviour of the entities is described in the State Model. It is assumed that every entity has state. It must be possible to make a state machine for every entity from the Information Model.
The graphical notation of the state-machines is called a life-cycle diagram. In describing the life-cycle of an object it is important to examine the following features: states, events and actions. State descriptions are made in pseudo code. The events cause the state transformations. An action can be seen as one state process.

The interactions between the different individual state models is graphically shown in what is called the Object Communication Model. This diagram show the input and output events for each state model (object) in the system.

The Information Model and the State Model together form the basis for the Process Model. The Process Model consists of separate DFD's for each state of each entity of the State Model.
The Process Model adds little new knowledge to the analysis. It is a refinement for the analyst to find generic processes that drive objects through their life-cycles.
The genericity found in this way can be used to maximize thereusability of objects and processes.

OODA doesn't refer to the use and specification of externals in the three previously described parts of the analysis phase. This is the successive step that has to be carried out. The external specification phase is a description of what the automated system must do and how it communicates with the outside world. It states what processes have to carried out by humans and what processing is done by the system.

Shlaer Mellor [10] mention two alternative methods for the external specification phase. Making a list of external events reflects the external view of the system. An alternative way of external specification is focusing on what's inside the system. The second alternative is the less recommended method to use [10, pages 99/100].

Although the external specification phase is certainly not a part of the analysis in OODA it is described in this section to show the basic difference between OODA an Synthesis.
This is the moment in the software development cycle where the externals are introduced.
Synthesis start the analysis phase with a context diagram and a list of external events. OODA calls this the external specification phase, which comes after the analysis.

4.5 Evaluation of the methods

In the previous sections we have summarized four different methods that describe parts of the software development process when using objects as essential units.
In this chapter we will compare these methods regarding their place in the development path and we will try to answer the question whether a fully object-oriented development path is needed or that individual steps can be carried out without the object-oriented view.

4.5.1 Comparing the methods

Mapping the described methods from the previous sections onto the steps of the object-oriented software development path (Figure 3.2 without the object-oriented meaning), we notice that there is not one method that covers the complete development path.

HOOD (section 4.1) is a design method that needs a supplementary analysis method, Stark, Seidewitz [21] and Booch [19] mention the use of structured analysis. Furthermore HOOD is primarily intended for one specific language, ADA.

An object-orientation integration with SA/SD (section 4.3) leads to an object-oriented design but starts with a structured analysis approach. It is questionable if the analysis of this method can be qualified as an object-oriented analysis.

The Shopping List Approach (section 4.3) is a special case, it is more a philosophy than a method and is used in the design and implementation phase of the development route.
This approach is only useful if there are class producers and class users. To compose his own software toolkit, the user makes a list of the generic classes he needs.
Building applications with the Shopping List Approach demands a proper use of the inheritance concept.
The preparation of the generic modules requires object-oriented analysis and design techniques in order to capture all the object-oriented properties.

Object-Oriented Domain Analysis (OODA) and Synthesis (both section 4.4) develop an object-oriented analysis. They focus on the information centralization around the domain or real world entities. Apart from their way of using the external events these methods look alike. For example the Communication Model found in OODA can be compared with the set of all the neighborhood diagrams from Synthesis.

Placing all the described methods in an object-oriented development path we get a table as shown in Figure 4.7.

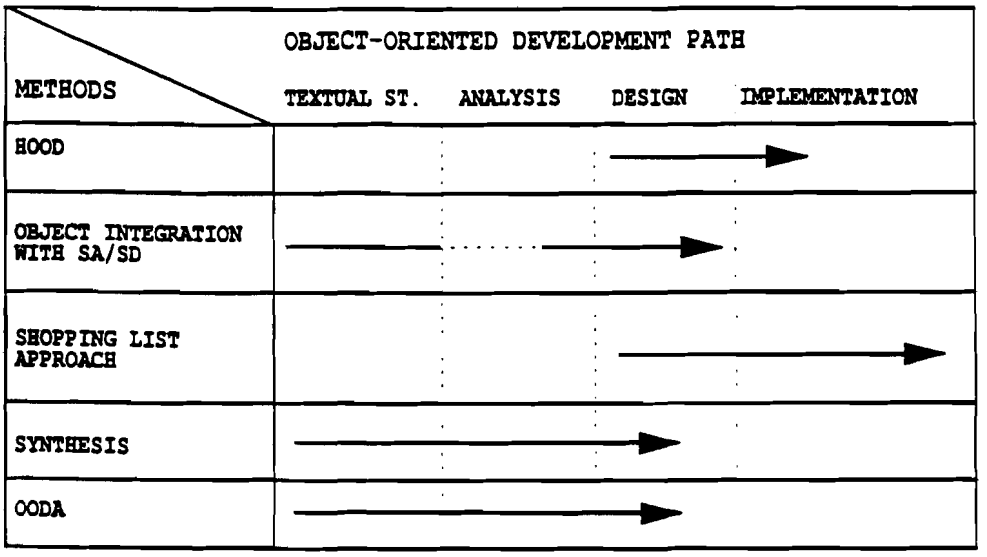


FIGURE 4.7 OBJECT-ORIENTED DEVELOPMENT OVERVIEW

The dashed arrow part of the integrated SA/SD method is used to referto the uncertainty if this part belongs to an object-oriented approach of the analysis phase.
The continuation of OODA and Synthesis into the design phase is due the fact that certain ingredients of the analysis phase of these methods belong to design.
For instance the neighborhood diagrams of Synthesis and the coding in the State Model of OODA.
HOOD extension into the implementation phase was already noticed in section 4.1.

4.5.2 Conclusions

The goals of object-oriented development are mainly twofold. We want to decrease software development and maintenance costs. On the other hand we need system development approaches that can cope with the increasing complexity of these systems. Object-oriented system design contributes to the decrease of these problems in several ways.

It encourages the reuse of code, implemented by inheritance. The understandability of systems is improved by modelling real world entities by their behaviour instead of their implementation. In this way the feedback between the analyst

and his information suppliers about the system is improved. The same reason holds for the requirements negotiations between the client and his suppliers.
Describing entities as abstract data types (ADT's) contributes to the reduction of these problem areas.
Using objects as the basic components in a system design can contribute to an easier redesign and maintenance.
To explain this view it is necessary to know how the objects are used and what they do. This brings us back to the properties inheritance (class hierarchies), ADT's and object communication.
The specialization of existing objects to create new objects through inheritance is an obvious example.
The objects communicate with each other using messages. If the relation changes or a new relation is added between two objects, the modification of that system part can be achieved by making local adaptions.
In this case one can think of adding new methods or changing existing methods (inheritance).

The objective is to find a software development path that supports the mentioned advantages of object-orientedness. This leads us the interpretation of the usefulness of the summarized methods to the object-oriented development path from Figure 4.6.
From this point of view, the integration of SA/SD in object-oriented development is of little use.
Although this method supports properties like subtyping and inheritance it doesn't contribute to easier system modifications. What to do if the translation from SA/SD to an object-oriented has been made and changes have to be made ?
Probably one has to do the SA/SD integration once more because the system structures are processes and not objects.
In my view this is a one-way method that is only useful to translate current systems to an object-oriented environment.

Synthesis and OODA are pure object-oriented methods. They are based on modelling the real world entities and the relations between the entities. Supporting of this kind of modelling I would favour starting an analysis with an external events list. This makes the system easier to understand. This is done in Synthesis and not in OODA. Therefore I would start an object-oriented analysis with a Context Diagram and an external events list. The result of the analysis is a description of the entities, relations and class hierarchies.

The next step to implementation is probably to to find the messages belonging to the objects and describing the communication between the objects through these messages.

This can be done with object diagrams using a similar kind of control structures as we saw in HOOD. Another aspect of HOOD is its hierachical structure. Besides the representation facet it can be useful to monitor the control structures.

The Shopping List Approach is a methodology that must be included in the design and analysis phase. This way of thinking forces the designer to make 'good' objects.

PART II: AN OBJECT-ORIENTED IMPLEMENTATION

5. THE POINT OF SALES SYSTEM

In part I and especially in paragraph 4.5 we made some remarks and conclusions about Object-Oriented Development methods. In order to verify these assumptions we want to develop a prototype object-oriented system.
The results of this evaluation are not only intended to show the advantages and disadvantages of these methods but can also be used to give a judgement of the relevancy about object-oriented development in general.

Our prototype system is a small Point Of Sales (POS) terminal application for a gasoline station. In this chapter we will give a description of the hardware connections with and the functionality of this POS Terminal.

5.1 The system's hardware

The POS terminal is based on a standard personal computer (PC), IBM compatible. The POS terminal is the "heart" of a gasoline station. It controls all the devices needed to run such a station. In Figure 5.1 the POS terminal and the devices connected to it is shown.

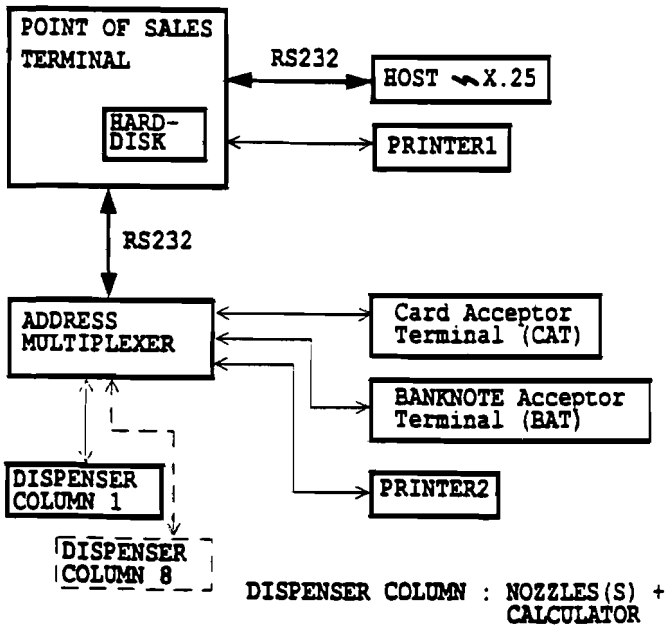


FIGURE 5.1 THE SYSTEM HARDWARE

Each of the items of Figure 5.1 is described below.

HOST
The HOST is mainly used to obtain on-line credit card authorization. It can also be used for software downloading, receiving of blacklists, price tables etc.

PRINTER1
This printer is used for the printing of customer tickets or is used to produce different kinds of reports.

CARD ACCEPTOR TERMINAL (CAT)
A CAT is a device that can be used by a customer for prepaid fuel deliveries. The customer has to put in his credit card and type some data like his PIN code, a Dispenser (Column) number, etc. The CAT checks this data.
Depending of the card type the system may decide for an on-line authorization via the HOST.

BANKNOTE ACCEPTOR TERMINAL (BAT)
Is used in the same manner as CAT, i.e. as a prepayment device, but accepts only banknotes as input. One needs only a validity checking of the banknotes.

PRINTER2
This printer is used for making backups of credit card paid transactions and transactions paid by a BAT.

ADDRESS MULTIPLEXER (KCD)
This device is the connection between the DISPENSER COLUMN(S), BAT, CAT, PRINTER2 and the PC. It simply reroutes the messages coming from and going to the PC. Every device (PRINTER2, BAT, CAT and DISPENSER COLUMN(S)) has an unique address.
The system isn't aware of the existence of this device.

Dispenser Column(s).
A Dispenser Column exists of a Calculator and one or more Nozzles and is used to dispense fuel. A Nozzle is identified by it's fuel type (diesel, super, blended mixture, etc.).
A Nozzle is connected to a fuel storage tank. Only when a Nozzle is used for a blended mixture it is connected to two fuel storage tanks.
The Calculator has a display for volume, total price and price/volume. The Calculator communicates with the PC by means of a set of messages. It has one pump that is put on during the dispensing. Only one fuel type can be dispensed at a time.

THE USER INTERFACE
The user interface is a part of the POS Terminal.
Further explanation is given in section 5.3.

5.2 The system's functionality

The POS terminal is placed inside the kiosk (also called a c-shop). The kiosk attendant (an operator) uses the POS terminal for cash register applications and uses it to monitor and control the dispenser (columns).
The user interface displays the data of operator's current transaction (payments and deliveries) he is handling.
The operator can authorize and stop a dispenser. During its operation cycle the dispenser status is monitored on the user interface. There is also an option for the operator to monitor the calculator data during the dispensing.
The POS terminal has three access levels:

(1) a service level
(2) a manager level
(3) an operator level

The service level has the highest access mode, an operator level the lowest. This service level mode is used by a company service man for changing the configuration settings, error handling, program updates etc.
The manager level is intended for the station manager only.
The station manager is for instance allowed to change prices, to produce all kinds of reports, update local accounts and to change access codes of the operators.
An operator has limited rights on the POS terminal. He may enter all the functions for the cash register applications and dispenser control.
The period that an operator is logged in is called a shift.
All the transactions handled by the operator during his shift are filed with his shift number. In this way a manager is capable of producing shift reports which allow him to monitor the operators.
A manager can have also operator access rights. But before entering management functions he has to quit his operator mode.

The functionality of the POS terminal can be divided into five groups,see also chapter 1:

(1) control of peripheral devices
(2) cash register functions
(3) management functions
(4) communication with a host
(5) software downloading

The primarily occupation of a gasoline station is the selling, paying and accounting of gasoline - and c-store products.
This is, what we call transaction handling.

Transaction handling comprises elements of the first four functionalities mentioned above whereby the emphasis is on dispenser column control and cash register functions.
We use the transaction handling mechanism as the starting point for the prototype system to be built.
We are going to make a small subsystem that is capable of handling the following ingredients:

- Dispenser column control
- Status displaying of dispenser columns, printer(s) and other connected devices
- Prepayment with credit cards
- Cash payment
- C-store dispensing
- Transaction accounting
- Basic management functions like changing prices
- Operator functions needed for transaction handling

5.3 The user interface

The user interface is an essential part of the system.
With "user" we mean whether an operator, a manager or a service man, not a customer !
Of course, the customer must be able to view his transaction data. This demands a second monitor. However, this customer 'view' is outside the scope of our system and project.

We distinct four important main windows in the user interface:
- A window that displays the status of the dispenser columns and other peripheral devices.
- A console window used for dispenser authorization and control functions.
- A command line window which displays a number of last entered commands together with the system's responses.
- A transaction window for displaying tickets.

The status of a dispenser column is shown by a collection of rectangles. The combination of the rectangle's colour and it's place in the collection is characteristic for a certain dispenser status.

The window that includes all the above mentioned (sub) windows is called the Main - or Working Window.
The Main Window can be present in two modes. The normal or working mode and a degraded mode.
The degraded mode is depicted when no operator or manager is logged in, i.e. static forecourt control.
An example of a Working Window is displayed on the next page.

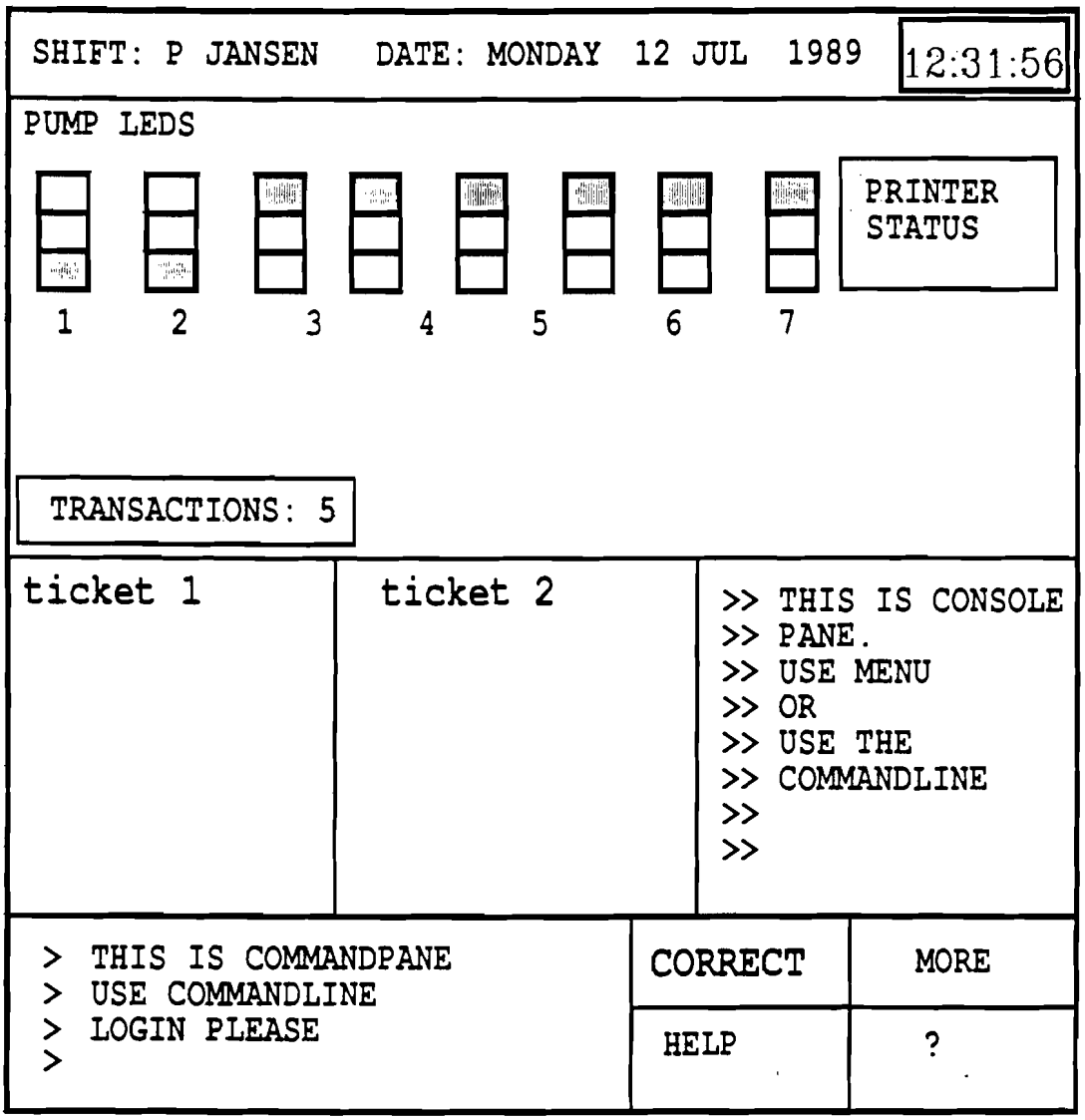


FIGURE 5.2 A WORKING WINDOW

6. OBJECT-ORIENTED ANALYSIS

The goal of the analysis phase is to make and evaluate a model which is based on entities as the basic structures for the system (de)composition.
To accomplish this goal we will first have to determine what road has to be taken.

6.1 The Strategy

In section 5.2 we discussed the transaction handling mechanism. From this point of view, the transaction handling mechanism, we start the analysis of our prototype system.

The fundamentals of the analysis modeling will be the first two models used by Shlaer - Mellor [7,10], the Information Model and the State Model (this includes the Object Communication Model).

To this analysis modeling we add the concept of external event modeling. This methodology is found in the Synthesis [23] model, see sections 4.4 and 4.5.
This external event modeling is the starting point of analysis. This approach is also referred to as the "outside in" approach.
A second item that is added to the analysis phase is the classification of entities, i.e. the building of hierarchies. This hierarchy building is done in parallel with the previously mentioned steps in the analysis phase.
Aiming at an object-oriented system the class hierarchy becomes the backbone.
Although this hierarchy modeling wasn't discussed as a separate step in literature it was already used indirectly when making Entity Relationship Diagrams (ERD'S) with subtyping in it. So this step is nothing more than a formalization of a heuristic used in ERD's.

This Object-Oriented Analysis combination is strongly based on the use of event modeling. Because of this event partitioning, we need to make some remarks about the use of events in our model (see also Ward -Mellor [7, part 2]).
Event modeling is a mapping between environment and a system. It is a stimulus response mechanism. This means that an event that is sent to, or coming from the system's environment, expects a (preplanned) response. This preplanned response can also be empty !
The events that occur between the system and its environment are called external events. Events that occur inside the system are called the internal events, we refer to both as events.

External events are the most important ones. They describe the system's interface with it's outside world entities, called terminators.
An external event can be described as a significant occurrence with a preplanned response.

To identify external events one needs to recognize them. Event recognition can be divided into two parts:
- direct recognition
- indirect recognition

The difference between those two is more or less the fact that indirect event recognition is used when having continuous events and direct event recognition is used when dealing with discrete events.
Because we are having a system that has only discrete events we use the direct event recognition.

Ward - Mellor [7, part 2] mention two strategies for creating a list of events:
- The active event modeling approach.
  An event is found by answering the question:
  What effects have the actions of the terminator on the system ?
- The passive event modelling approach.
  Here one doesn't have a well defined context (terminators).
  Now search for associations between objects in the system's environment.
  The use of ERD's is recommended.

The first strategy is appropriate to us. We have well defined terminators.

In literature a distinction is made between data and control events. We don't make that distinction. We use an event as a package of control and data.

6.2 The Analysis

As already mentioned in the previous section we use the transaction view for the analyzing of our system. We define a transaction as the combination of one or more payments with one or more deliveries. In section 5.2 we described the functionality of a subsystem that could manage transaction handling. Such a system focuses especially on the relations between the peripherals; DISPENSER COLUMN(S), CAT and (partially) the USER INTERFACE. The PRINTER1, PRINTER2, HOST, and BAT are occasionally referred to but are of less importance.

In describing the analysis process we will go through the following steps:

(1) External Events Model
    - Context Diagram
    - External Events List
(2) Information Model
    - ERD
    - Entity descriptions
    - Relationship descriptions
(3) State Model
    - lifecycles
    - (object) communication diagram
(4) The Class Hierarchy

The execution of each step is discussed in the next four sections.
Although class hierarchy building is numbered as step 4 it occurs in parallel with the two previous steps.

6.2.1 The External Events Model

Extracted from the hardware description of chapter 5 we start with a Context Diagram (see Figure 6.1).
The Address Multiplexer is transparent to the system, from now on called EXOOS, which stands for EXperimental Object-Oriented System, and is left out of the Context Diagram.

The peripheral devices of the system, the CAT and the DISPENSER COLUMN(S) are existing devices.
Their interaction with the system is well documented.
From these documents we extract their external events. We will only type the names of the events. We left the data types out for convenience. In Appendix 1, all the external events are listed.
The external events associated with the user interface that we are interested in, are those related to the DISPENSER COLUMN and some simple cash register functions.

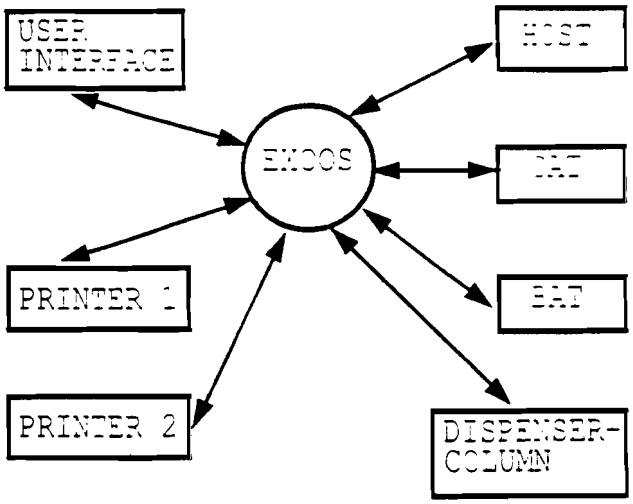


FIGURE 6.1 THE CONTEXT DIAGRAM

The events representing management functions are also included in this document but this part of the design is left out of our project.

6.2.2 The Information Model

The Information Model is based on Shlaer - Mellor [7,10]. We won't strictly follow all the conventions used by Shlaer-Mellor. We just simplified some of the descriptions used in [7] because we found it more convenient.
The Information Model consists of Entity Relationship Diagrams (ERD's) together with sets of descriptions of the elements of the ERD's (see also Appendix 2 and 3).
We will give a short description for some of the ERD's that we have constructed with the transaction view in mind.

A transaction was defined as a combination of payments and deliveries. A transaction can be prepaid or postpaid. The CAT (Figure 6.1) is a prepayment device. The CAT's application as a prepayment device is only used for fuel deliveries at the moment. In Figure 6.2 an ERD with CAT as a centre is shown. Relationships will be typed as:a relationship.

To come to a delivery the following steps are taken:
- The client puts a card in CAT and types a dispenser number, the CAT sends a message to OPT (controls).
- Before entering the validation phase the OPT reserves the wanted fuel dispenser (reserve dispenser).
  Then the OPT asks a card validation from the CARD VALIDATOR (asks validation).
- The OPT asks an authorization from the CARD AUTHORIZER AND PAYMENT HANDLER (asks authorization).
- If everything to this step is done successfully the OPT creates an OPT TRANSACTION CONTROLLER (TRCTR).
  This entity is responsible for handling transactions.
  There is one TRCTR for every transaction.
- With the authorization data, i.e what type of fuel, amount of money or volume is permitted, the OPT sends a release request to the FUEL DISPENSER (requests fuel-delivery).
  The request arguments include amongst others the OPT TRCTR and authorization data for the dispenser.

Although we refer to it as a prepayment mechanism, it would be better to call it a pre-authorize mechanism. There is no payment at the moment of a fuel delivery request, only the authorization !
A TRCTR is responsible for putting payments and deliveries in a transaction and for the storing of the same transaction.
One could say it is the link between the delivery and payment mechanisms.

We turn our attention to the FUEL DISPENSER. In Figure 6.3 the fuel delivery ERD is shown.

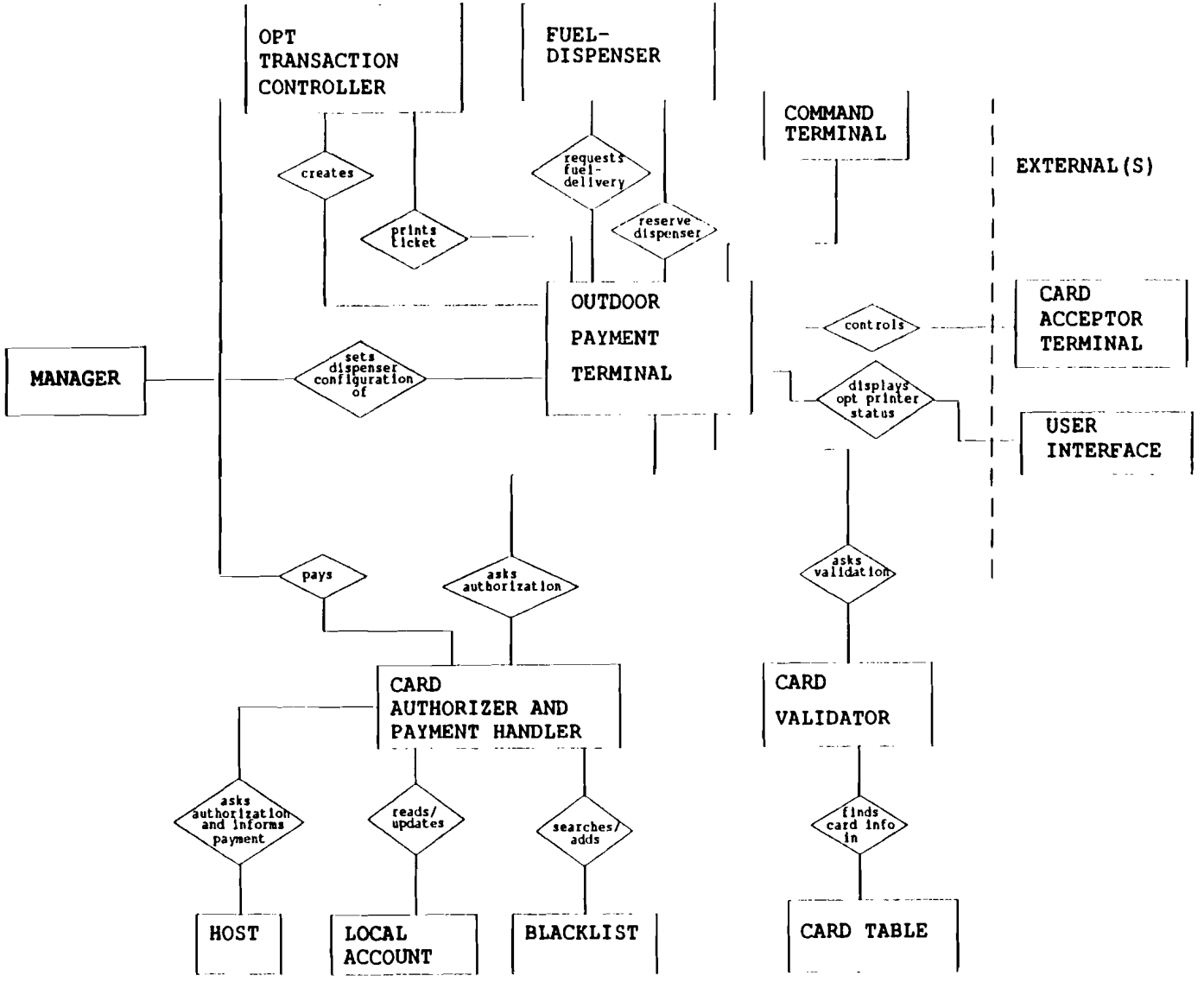
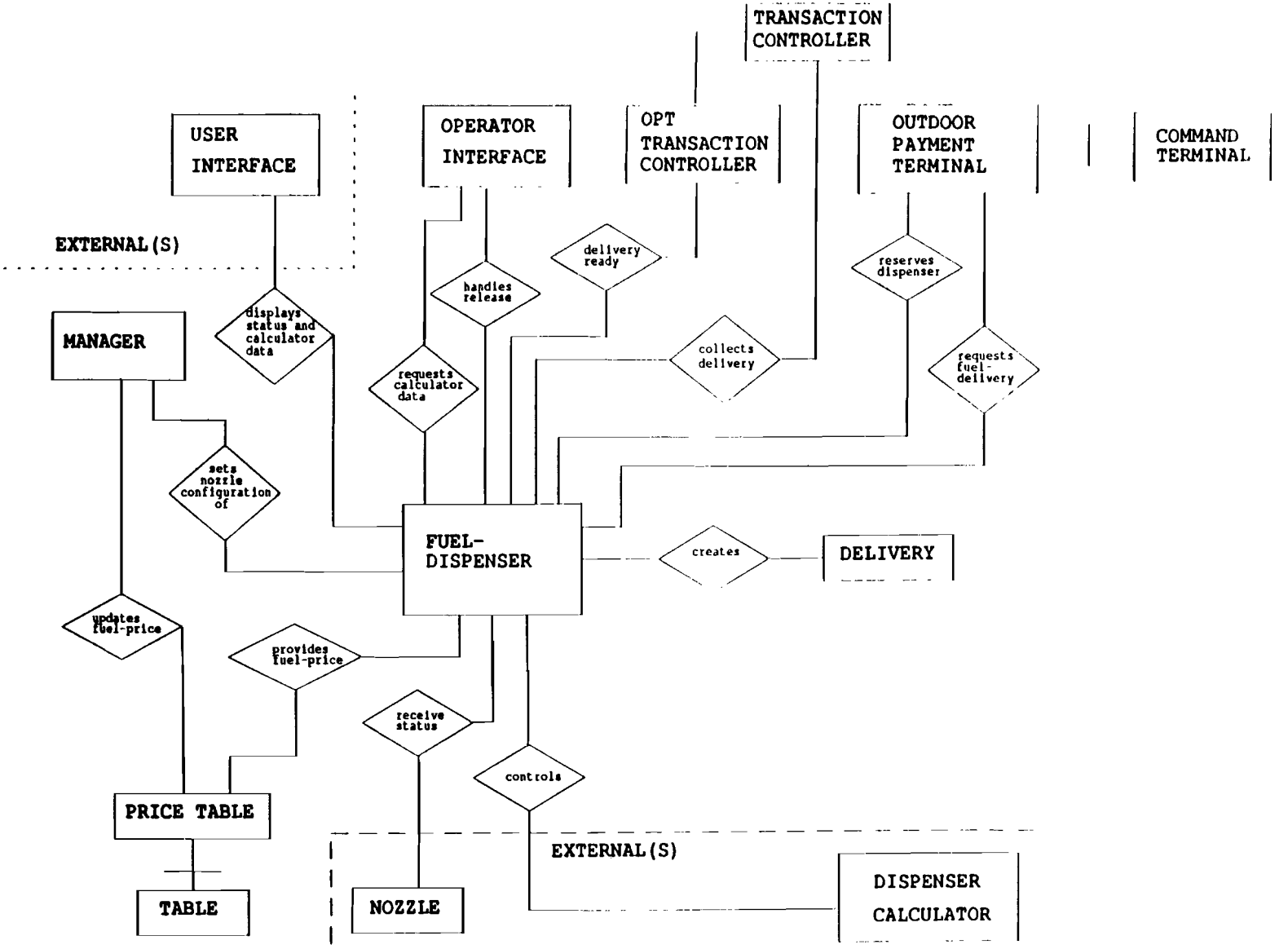


FIGURE 6.2 THE OUTDOOR PAYMENT ERD

OUTDOOR PAYMENT ERD

pt-erd:13
15/02/90



FUEL DELIVERY ERD

del-erd:6
15/02/90

FIGURE 6.3 THE FUEL DELIVERY ERD

The FUEL DISPENSER steps after a release request are:

- Waiting until the customer lifts up a nozzle (receives status). A nozzle stands for a fuel type. If this type is authorized the FUEL DISPENSER needs a price (provides fuel-price) to send to the DISPENSER CALCULATOR (controls).
  This fuel-price is a unit price. It depends on the customers card type what he actually will pay.
- After the ending of the delivery, the nozzle is put back by the customer (receives status), the OPT TRCTR is signalled by the FUEL DISPENSER (delivery ready).
  The FUEL DISPENSER creates a DELIVERY (creates).
- The OPT TRCTR collects the delivery data (collects delivery).
  The FUEL DISPENSER is ready for another release.

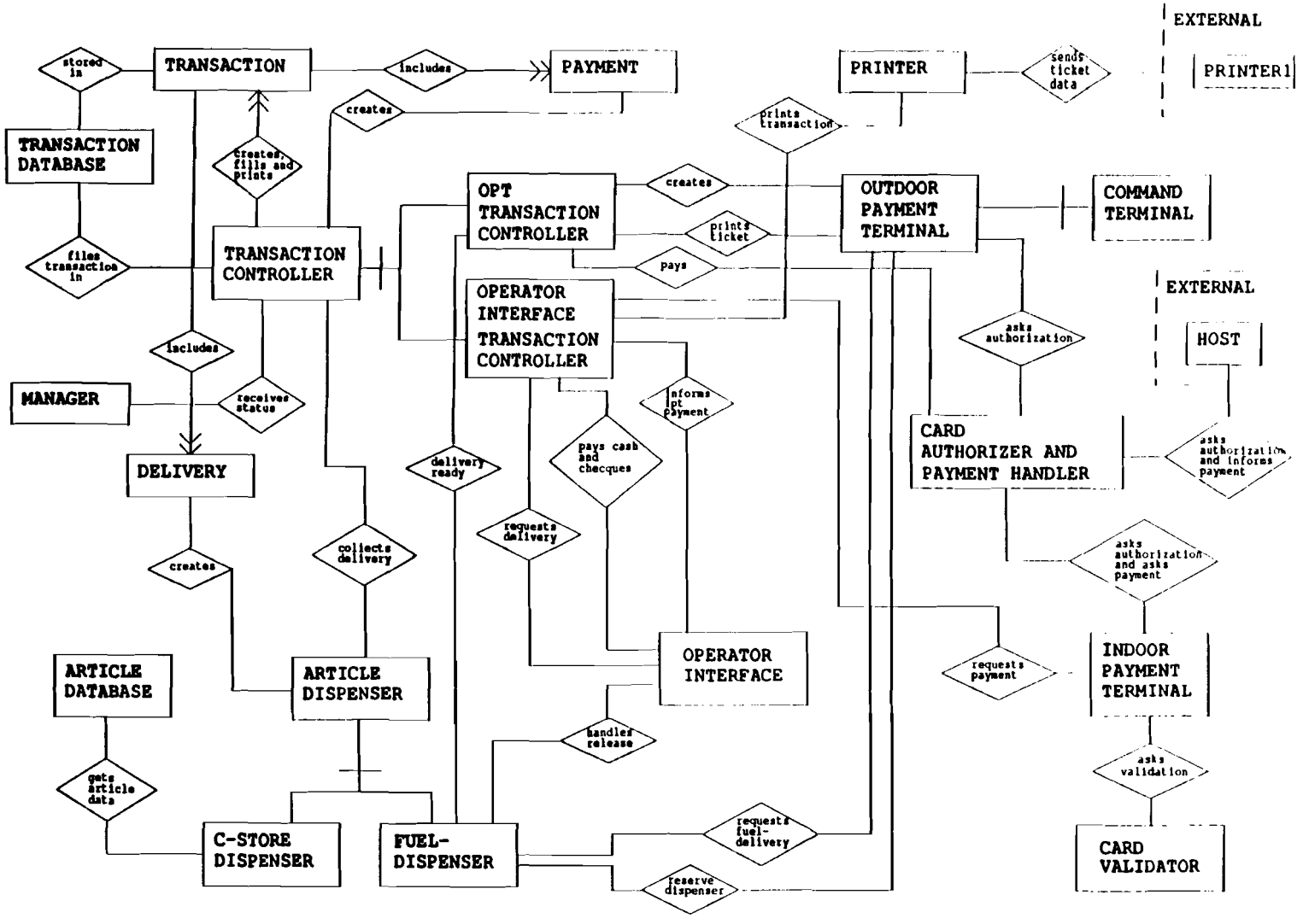
The TRCTR creates a PAYMENT (create) and sends a payment request to the CARD AUTHORIZER AND PAYMENT HANDLER. This entity pays (pays) the delivery and the transaction is complete. The TRCTR sends the OPT a message with data for the client's ticket (prints ticket).
The FUEL DISPENSER's status is continually displayed at the USER INTERFACE (displays status and calculator data) of the POS Terminal, see Figure 6.1. The commands entered by the operator enter the system through the entity OPERATOR INTERFACE.
The transaction mechanism for a shop payment shows a lot of similarities. Now it is the OPERATOR INTERFACE that must release a fuel dispenser and not only fuel is dispensed but shop products also. In Figure 6.4 the transaction controller ERD is shown.

The process of building ERDs can't be separated from the description of its elements. In Appendix 3 the description of the entities and their relations can be found.

The description of entities is a textual statement. An important item of this description are the attributes.
An attribute is an abstraction of a single characteristic of an entity. In the analysis all the attributes must be found.
Roughly we distinguish two important types of attributes:

1. Referential attributes, they are used to couple objects.
2. Private variables, these are objects needed by an entity to perform it's functions.

For instance the FUEL DISPENSER will have referential attributes for the PRICE-TABLE, the OPT TRCTR, the USER INTERFACE etc.



TRANSACTION CONTROLLER ERD

tc-erd:5
29/01/90

FIGURE 6.4 THE TRANSACTION CONTROLLER ERD

The attributes can also be described in textual documents. In Appendix 3 there is no special description of the attributes. Most of the descriptions can be found implicitly in the entity descriptions.
The relationship descriptions tend to be protocol like and give an indication of possible events between the entities.

In the ERDs we see relations with a crossbar. They indicate subtyping. A FUEL DISPENSER is a subtype of ARTICLE DISPENSER. We choose OPT and the USER INTERFACE as a subtype of COMMAND TERMINAL. The reason behind this is that one needs at least an OPT or an USER INTERFACE to obtain a working system.
Working with subtypes leads to generic relationships. In Figure 6.4 one notices that the relation collects delivery belongs to two types of transaction controllers. Subtyping is a useful tool for class hierarchy building, see also section 6.2.4.

The relations in an ERD are logic relations. There are several types of relations.
For instance the 'include relationship'.
Examples can be found in Figure 6.4, TRANSACTION includes PAYMENT and in Figure 6.3, CARD VALIDATOR finds card info in CARD TABLE.
In most cases one of the entities that contains an 'include' relationship can be found as an attribute of another entity.
Shlaer - Mellor [7], chapter 5, gives a survey of a number of relationships.

6.2.3 The State Model

A lifecycle represents the dynamic behaviour of the entities. It is the intention that every entity found in the Information Model is formalized by a lifecycle.
We used the graphical representation and notations of the lifecycles found in [7].

It is in this phase that the attribute 'status' of an entity is actually assigned with values.
We notice that when modeling lifecycles we have the requirement of one action per state.

A lot of table like entities have a very simple lifecycle. The PRICE TABLE (see Figure 6.2) is such an example. As we shall see it has only one state.
From the more complicated entities we took four to investigate. These entities are (see also Appendix 4):

- FUEL DISPENSER
- OUTDOOR PAYMENT TERMINAL (OPT)
- OPT TRCTR
- UI TRCTR.

Lifecycles are completely event driven. This means that in the lifecycle of FUEL DISPENSER, which controls the (external) DISPENSER COLUMN, one should recognize all the events found in the External Events List.
The lifecycle diagram of the FUEL DISPENSER is shown in Figure 6.5. Putting such a lifecycle together is a time-consuming job. Not only the information what an entity stands for and what it is supposed to do is important. But also the information, when and in what order the entity is supposed to handle it's requests, is needed. In this case it means that one has to investigate every relationship in the information model of FUEL DISPENSER with the other entities. In our case we made a simplified lifecycle by not implementing the relationship between FUEL DISPENSER and MANAGER.

Shlaer - Mellor [7] state that a state model of an entity's lifecycle is analogous to pure code. This is actually an anticipation of an aspect of the development trajectory that belongs to the design phase.
We used it as a more global high level description for the states of an entity. Furthermore it is difficult to speak about pure code if it isn't obvious what kind of implementation language is chosen.

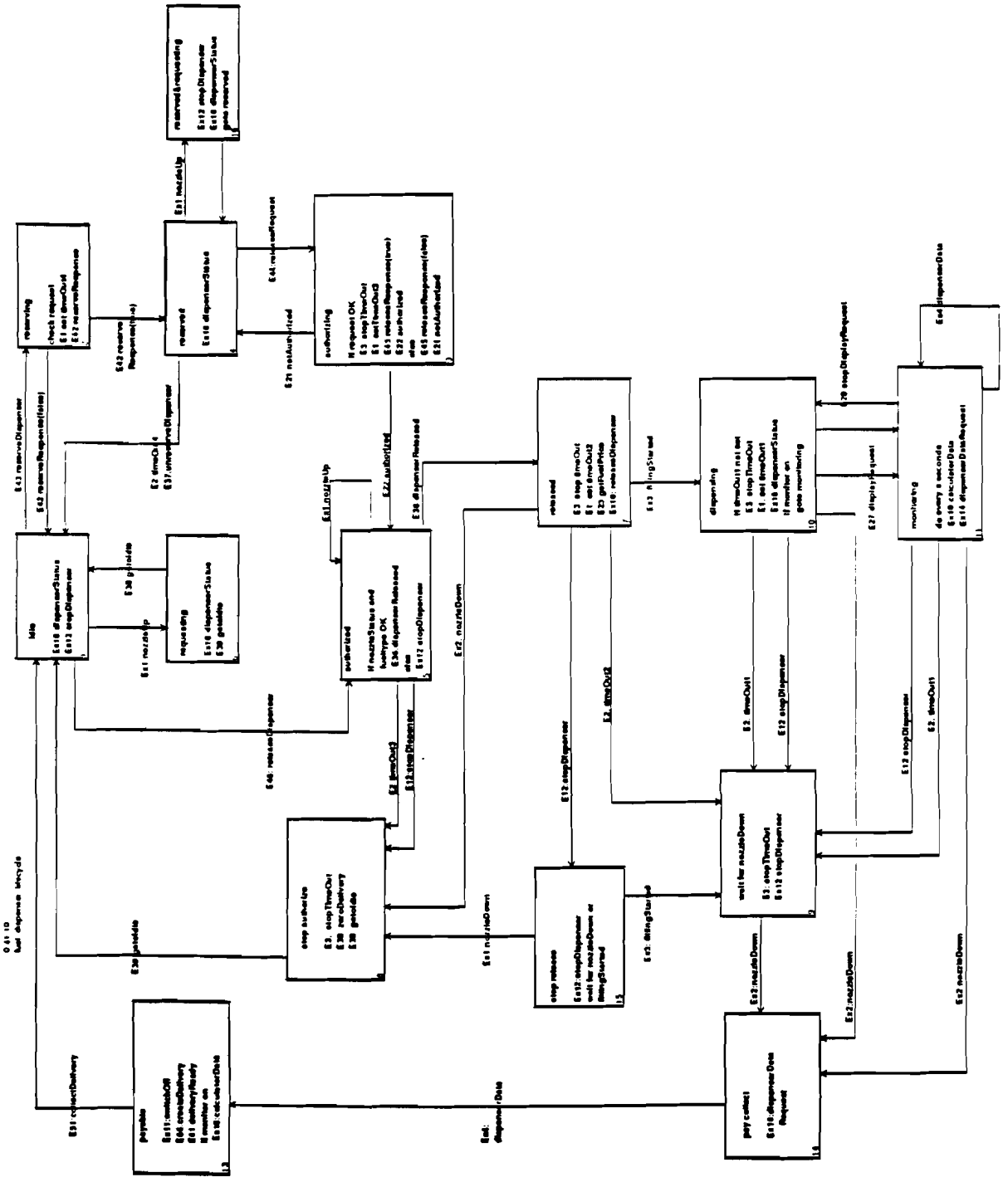


FIGURE 6.5 LIFECYCLE DIAGRAM OF A FUEL DISPENSER

The events with names starting with EX in Figure 6.5 are external events. Some of the events, like **E43:reserve-Dispenser**, come directly from entity and relationship descriptions. Others, like **E22:authorized**, are a result of the one action per state.

In Figure 6.5 one can see the dispenser release through an OPT using the states:*idle, reserving, reserved & requesting, reserved, authorizing* and *authorized*.
The release through means of an OPERATOR INTERFACE that the states:*idle, requesting* and *authorized*.

Comparing the lifecycles of the two transaction controllers (see **Appendix** 5) we notice that the generic relationship, collects delivery, leads to an event, **E51:collect delivery**, which is generated by both the transaction controller and sent to a dispenser.

All the individual lifecycles together form the State Model. From the the lifecycle models we can construct, what is called (Shlaer - Mellor [10]), a Communication Model. It shows the interaction between the individual state models.
In Figure 6.6 the Communication Model is shown. We created some relevant events for the entities which hadn't a life-cycle to complete a working model.

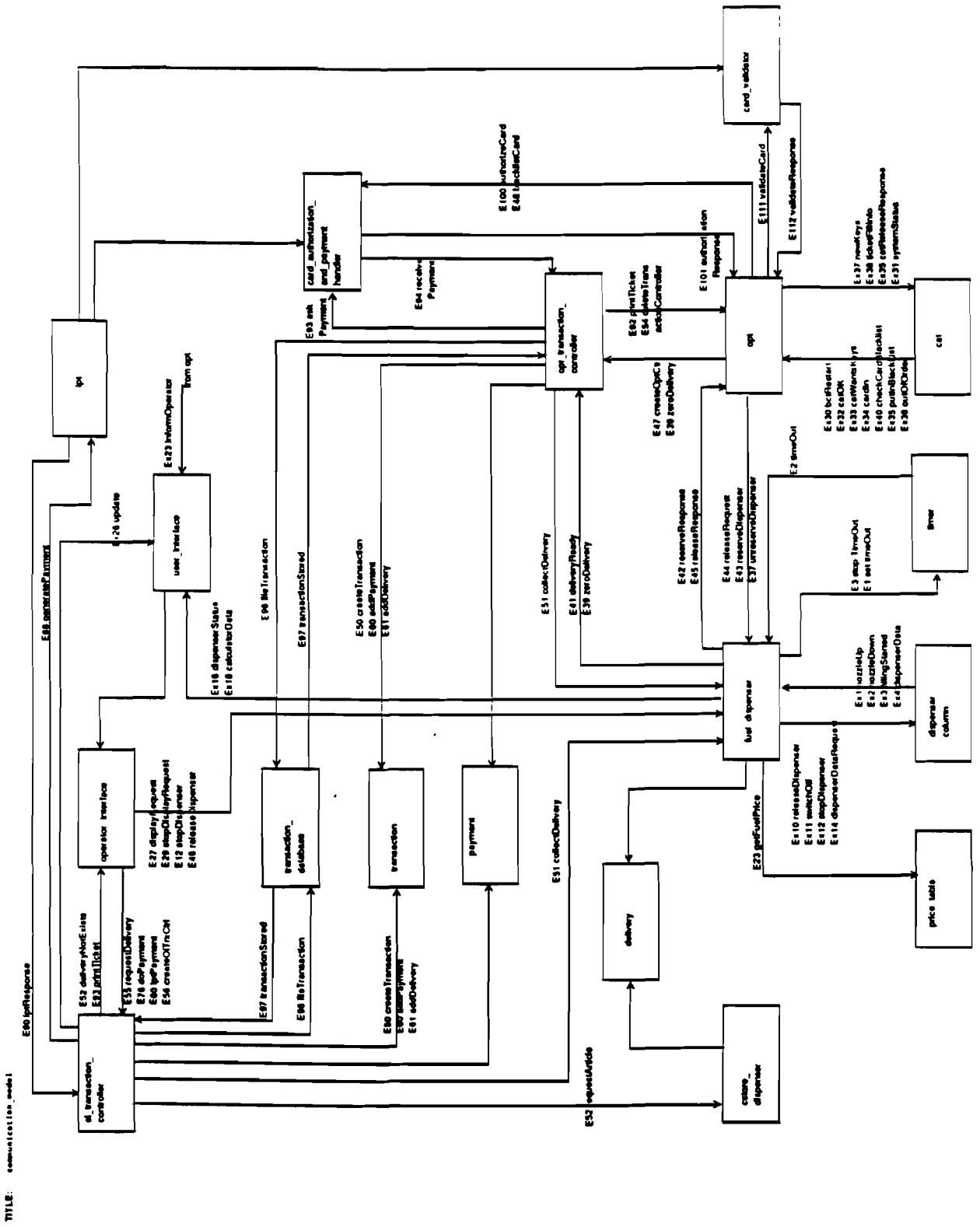


FIGURE 6.6 (OBJECT) COMMUNICATION MODEL

6.2.4 The Class Hierarchy

Essential for Object-Oriented modeling is the building of class hierarchies. This is the basic structure for the current design and future changes in the same design.
Every entity found must be placed a the class hierarchy.
In Figure 6.7 a part of the class hierarchies is shown.
For the complete class hierarchies see Appendix 5.

We already discussed the parent class COMMAND TERMINAL of USER INTERFACE and OPT.
From Figure 6.4 we extract two other super/sub classes.
The TRANSACTION CONTROLLER with the subclasses OPT TRCTR and OI TRCTR, and the ARTICLE DISPENSER with the subclasses FUEL DISPENSER and CSTORE DISPENSER.
Finding these higher level data abstractions is not an easy job. This process is simplified by searching for generic relationships and comparing different, but similar lifecycle diagrams. For instance compare the two transaction controller types and their relations with other entities.

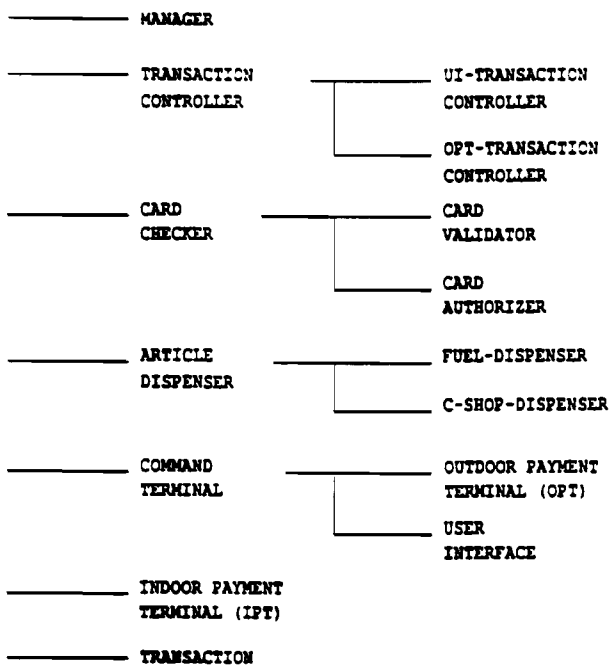


FIGURE 6.7 CLASS HIERARCHIES EXAMPLE

6.3 Results

Shlaer - Mellor [7,10] mention another step following the State Model, the Process Model. It comprises separate data flow diagrams for each state. They add little information to the analysis and it takes a lot of time to construct them.
Shlaer - Mellor use the processes of the data flow diagrams mainly for identifying identical processes inside one object and inside related objects. They call these related objects the supertypes. Capturing the same properties in such a manner is almost identical with putting objects into classes. Therefore the Process Model is most useful in clarifying the different states of a lifecycle. Because it is not an essential part we left it out the development path.

The most difficult part of the analysis phase is the building of ERD's with relevant relations. This includes finding the entities and their relations.
The "outside - in" approach is a good start to detect high level entities. Further analyzing can lead to the decomposition of entities and new entities.

The detection of higher level data abstractions of entities is encouraged by generic relationships and by comparing lifecycle diagrams (see Appendix 4).
This implies that making an object-oriented analysis is a recursive process.

7. OBJECT-ORIENTED DESIGN

The goal of the design phase is to translate analysis to such an arrangement that the system is ready for implementation.

7.1 Design considerations

In Part I we discussed an Object-Oriented Design method, the HOOD (section 4.1, [16]) approach. HOOD is especially designed for an ADA environment but some features as mentioned in chapter 4, the Object Diagram, parent-child modeling and concurrency, could be useful in other design methodologies too.
The use of Object (neighborhood) Diagrams was also advised by Synthesis (section 4.4) and used in HOOD [16].
The only other 'methodology' mentioned in literature, was the Shopping List Approach, but this was more a philosophy than a useful tool (section 4.3 and 4.5).
Neither of these approaches can be used as a continuation of the (Object-Oriented) Analysis from chapter 6.
Schlumberger has internally come up with an Object-Oriented Design Development trajectory which consists of two steps called conceptual - and detailed design.
This method was not supported by an analysis method.
Neglecting this deficiency because we had already an independent Analysis Model, we evaluated these steps, and added the concept of object diagrams that was already found in literature.
At the same time we tried to make a link between the Object-Oriented Analysis and this Design Trajectory.
The result, as we will see in section 7.2, was a remarkably simple step that is executed in combination with the conceptual design step.

In the next section we will describe the design steps we have followed.

7.2 The Design

The methodology developed internally consists of two steps-conceptual and detailed design. In this section we will discuss both.

7.2.1 Step 1 (conceptual design)

The first step consists of finding the classes, put them in a hierarchy and fill them with public methods. From our analysis we already have the hierarchy. A method in this context is a message and a response. The response is obligatory, but may be empty.
If we want to fill the class hierarchy we will have to find the methods first.
The heuristic is very simple: Use the events !
We distinguish two types of event mapping:

1. One event results in one method.
2. Two events (a request and a response) result in a method.

An example of the first mapping is the event:**deliveryReady**, generated by FUEL DISPENSER, that can be transformed into the method:**deliveryReady** (see Figure 6.3).
The combination of the two events **reserveDispenser** and **reserveResponse** to a method:**reserveDispenser** is an example of the second type of mapping (see Figure 6.3).
This mapping is the main link between the analysis and the design. In Appendix 6 one can find all the public messages that are put into the class hierarchy.

Looking at the class ARTICLE DISPENSER and especially at the method, **collectDelivery**, one notices that this method is reimplemented at its subclasses. Such a construct was already expected when we looked at the lifecycle diagrams in chapter 6.

Stripping the event responses from the Object Communication Model leads us to a new type of diagrams, the Object Diagrams. See also part I: the Synthesis and HOOD overview.
Object Diagrams are not included in the original conceptual design step but they are a logical graphical addition. In Figure 7.1 we show such an object diagram.

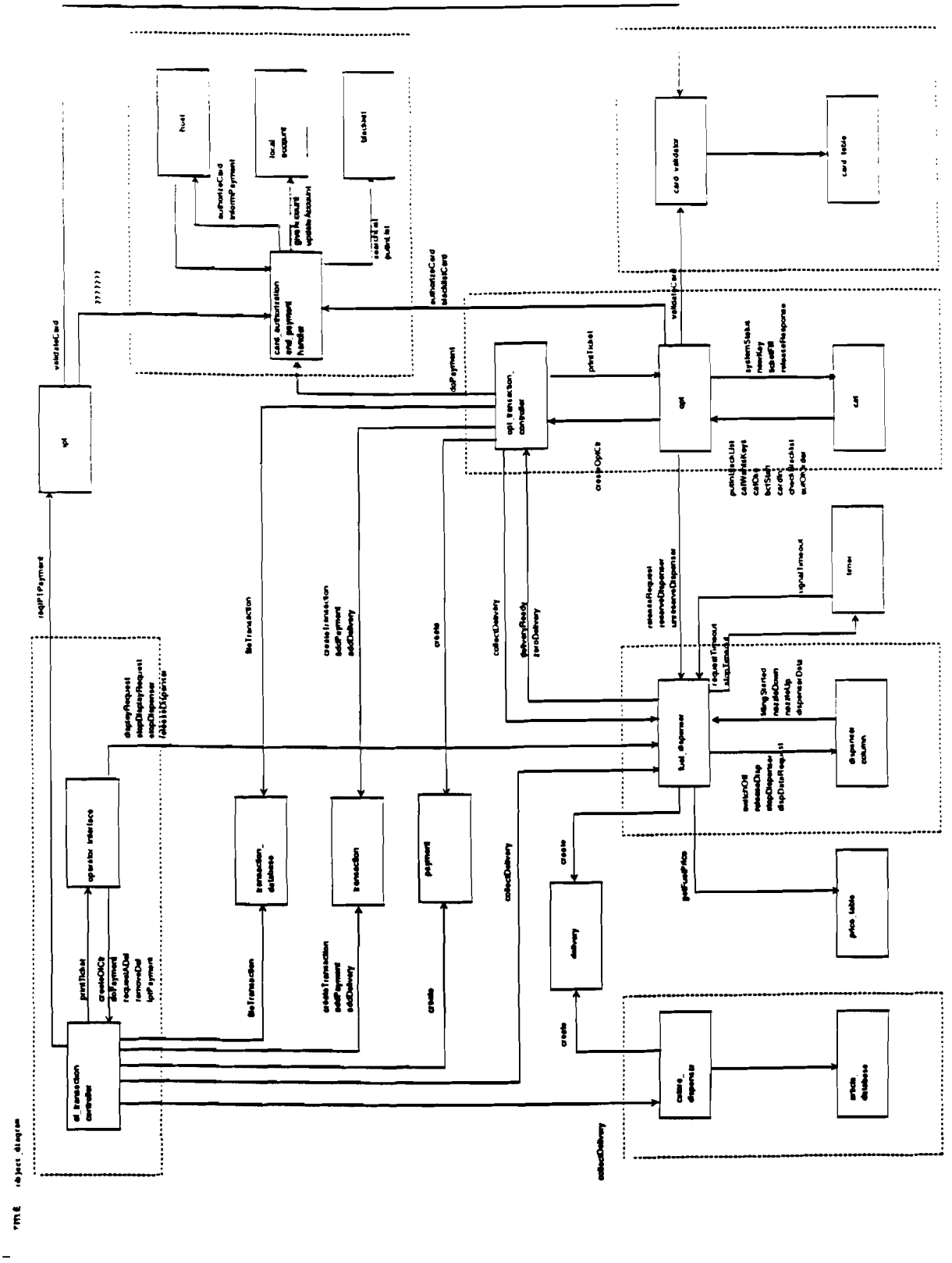


FIGURE 7.1 OBJECT DIAGRAM

7.2.2 Step 2 (detailed design)

This step has two main substeps which we will describe with some examples taken from Appendix 6.

In the first substep, declarations of public methods are made. This includes the description of the arguments, the return value and the class types for each value.
Figure 7.2 shows an example of public method declaration:

```
object      : FUEL DISPENSER
method      : reserveDispenser

argument    : anOriginator
type        : CommandTerminal

returnValue : ^aBoolean
type        : Boolean
```

FIGURE 7.2 AN EXAMPLE OF PUBLIC METHOD DECLARATION

A method can have zero or more arguments, it always has one return value. We used the SmallTalk notation (^) referring to return values.
The arguments can be deduced from the events that lead to a particular method.

A second substep that is performed is the production of the definitions of the public methods.
To accomplish this we use the lifecycle diagrams. We do not produce a copy of the "code" of the lifecycles but an interpretation. We produce a kind of pseudo code.
In Figure 7.3 an example of pseudo code is given. This example shows the use of a private variable, *originator* and the creation of a private method, **gotoReserved**.
The private variable must be one of the list of attributes found in the analysis. The private methods found in this manner must be declared also. The is analogous to the declarations of the public methods. Together with this declaration a textual description is produced. An example is given in Figure 7.4.

```
reserveDispenser:anOriginator

(* anOriginator wants to reserve this fuel dispenser.
   This only possible when fuel dispenser is in state
   idle. If already reserved check anOriginator and
   if oke reserve again.
*)

status = idle
ifTrue:[save anOriginator in originator.
        self gotoReserved.
        ^true
       ]
ifFalse:[originator = anOriginator
         ifTrue:[self gotoReserved.
                 ^true
                ]
         ifFalse:[^false].
        ]
^false
```

FIGURE 7.3 AN EXAMPLE OF PUBLIC METHOD PSEUDO CODE

```
gotoReserved

(* Set time out for reserved.
   Update dispenser status.
*)
```

FIGURE 7.4 AN EXAMPLE OF PRIVATE METHOD DECLARATION

Strictly following this methodology would imply making declarations and default values of the private variables. The declaration of the privates is already done in the analysis, this need not be done again.
The declarations of the default values is left to the implementation phase.

7.3 Recapitulation

An item that is fairly important but which is not discussed in these design steps is concurrency. If an object has sent a message to another object and must be able to respond to a receiving message at the same time, there is concurrency.
Looking at Figure 7.1, one could imagine such a situation in case the OPT sends the *authorizeCard* message to CARD AUTHORIZATION AND PAYMENT HANDLER while the CAT sends a *bctRestart* to the OPT. A restart means that the CAT has returned the credit card to the client and the authorization can be stopped. The CAT expects a status message from the OPT. In the analysis and design models there is no special mechanism for concurrency recognition. The only thing left to do is placing remarks in the design and analysis phase where to expect concurrency.

Another item, the **parent - child relationship**, found in HOOD [16], is not used in this design phase. We already mentioned the **include relationship**, which is similar, in the analysis phase and its connection to the attributes.
It is not clear what the influence is on the design.

The mapping of events to methods and the use of lifecycle diagrams for generating pseudocode is the link between analysis and design.

8. OBJECT-ORIENTED IMPLEMENTATION

The implementation language is SmallTalk, see chapter 2.
Due to the effort put into the analysis and design phase there was no time left for the implementation.

The implementation phase of the development trajectory consists extending the design phase with coding details.
This step is focused mainly on the coding details of the private methods and transforming the pseudocode of the public methods to the actual code.
When looking at Figure 7.3 we see a strong resemblance with Smalltalk. The pseudocode developed in the design phase is close to code.
The expectation is that the coding takes less effort than the design itself. Provided that problems such as concurrency can be simply resolved.

During the analysis and design phase some tests with a SmallTalk communication port and a Dispenser Column were carried out. This communication port of Smalltalk is interrupt driven.
From this test we know that it is possible to control peripheral devices with a Smalltalk application. Conclusions about real-time aspects cannot be drawn yet.

All the interrupt types (communication, keyboard, mouse and time) in the Smalltalk (version V286 1.0) system are rerouted through one event before entering the system. Given the fact that there is only one process running (the user interface process) the complications of parallelism are enormously reduced.

9. CONCLUSIONS AND SUGGESTIONS

In chapter 3 we discussed the need for a complete Object-Oriented Development to come to object-oriented implementations. Although the implementation is not completed we can conclude that the composed Object-Oriented Analysis (OOA) and Design (OOD) models are fit for an Object-Oriented Implementation in Smalltalk. The pseudo code, made in design phase, is close to an implementation in Smalltalk.

The OOA and OOD are based on different ideas and items found in literature. During the development and the evaluation of our object-oriented trajectory we did not follow the sequence of steps as described in the chapters 6 and 7. This 'jumping' was necessary to get an insight in the different methodologies and their integration. However, when one wants to obtain a smooth OOA and OOD trajectory one must follow the sequence of these steps as they are reported.

The link between the analysis and design models by mapping the events is a reasonable solution to couple the design to the analysis. The different models can probably be more integrated and leveled to each other, when the attention is turned over to make a complete development path, instead of separate models for analysis and design. An example is the lifecycle modeling (Shlaer - Mellor, analysis phase) that includes pseudo code. This coding is actually a part of the design phase. The lifecycle modeling must be done with sort of a high level description but not with pseudo code.

In the design phase the concurrency is an underdeveloped subject that needs more investigation and a place in the development trajectory.

An open issue is how to integrate the parent-child relations in the development path. It is obvious that this is related to the (de)composition of a system but is not clear what to do with it.

An important characteristic of object-oriented programming is reusability. This reuse is obtained by inheritance.
In our development trajectory we make little use of reusable objects. We simply have no reusable objects (classes) available. A testcase for the reusability of the created classes will be future adaptions or alterations of the designed system.

10. LITERATURE

Books:

[1] A Little Smalltalk.
Timothy Budd.

[2] An introduction to Object-Oriented Programming and Smalltalk.
Lewis J. Pinson, Richard S. Wiener.

[3] Object-Oriented Software Construction.
Bertrand Meyer.

[4] Object-Oriented Systems Analysis:
Modelling the World in Data.
S. Shlaer, S. J. Mellor.

[5] Smalltalk V286, Tutorial and Programming Handbook.
Digitalk Inc.

[6] Strategies for Real-Time System Specification.
Derek J. Hatley, Imitiaz A. Pirbhai.

[7] Structured Development for Real-Time Systems.
Paul T. Ward and Stephen J. Mellor.
Volumes 1,2 and 3.

[8] System Development.
Michael A. Jackson.

Papers:

[9] An Object-Based Development Model.
David M. Bulman.
COMPUTER LANGUAGE, august 1989.

[10] An Object-Oriented Approach to Domain Analysis.
Sally Shlaer, Stephen J. Mellor.
ACM SIGSOFT, jul 1989, pages 66 - 77.

[11] An Object-Oriented Requirements Specification Method.
Sidney S. Bailin.
Communications of the ACM, Vol 32 Nr. 5, pages 609..623.

[12] An Object-oriented Structured Design Method for Code Generation.
A.I. Wasserman, P.A. Pircher, R. J.Muller.
Interactive Development Environments. Inc.

[13] Assuring Good Style for Object-Oriented Programs.
K. J. Lieberherr, I. A. Holland.
IEEE Software, september 1989, pages 38 - 48.

[14] Design Principles behind Smalltalk.
Daniel H. H. Ingalls.
BYTE, august 1981, pages 286 - 298.

[15] Experimental Prototyping in Smalltalk.
Jim Diedrich, Jack Milton.
IEEE Software, may 1987, pages 50-64

[16] HOOD MANUAL, Issue 2.2 april 1988, ESA/ESTEC and Issue 3.0 september 1989, ESA/ESTEC

[17] How to Integrate Object Orientation with Structured Analyses and Design.
Paul T. Ward.
IEEE Software March 1989, pages 74...82.

[18] Learning the language.
Peter Wegner.
BYTE, march 1989, pages 245 - 249.

[19] Object-Oriented Development.
Grady Booch.
IEEE Transactions on Software Engineering, Vol SE-12, February 1986, pages 211...221.

[20] Software-IC's.
L. Ledbetter, B. Cox.
BYTE, june 1985, pages 28 - 36.

[21] Towards a general Object-Oriented Software development methodology.
E. Seidewitz, M. Stark.
Goddard Space Flight Center.

[22] What's in an Object?
Dave Thomas.
Byte, march 1989.

[23] What's The Object?
Presented by Meilir Page-Jones and Steven Weiss.