# IMPLEMENTATION OF OBJECT ORIENTED PRODUCT MODEL APPLICATIONS

Matti Hannus

VTT, Laboratory of Structural Engineering, Kemistintie 3, SF-02150 Espoo, Finland

**Abstract**

The paper describes implementation aspects of object oriented applications using different software tools such as a CAD-system, a relational data base management system and an object oriented programming language. The different implementations are based on a common generic product model and are integrated by means of neutral file transfer. The modules make up a toolbox from which various specific applications can be derived by adding application specific subclasses. The described development aims to provide steps along an evolutionary path from the dominating design tools of today towards the envisioned object oriented systems of tomorrow.

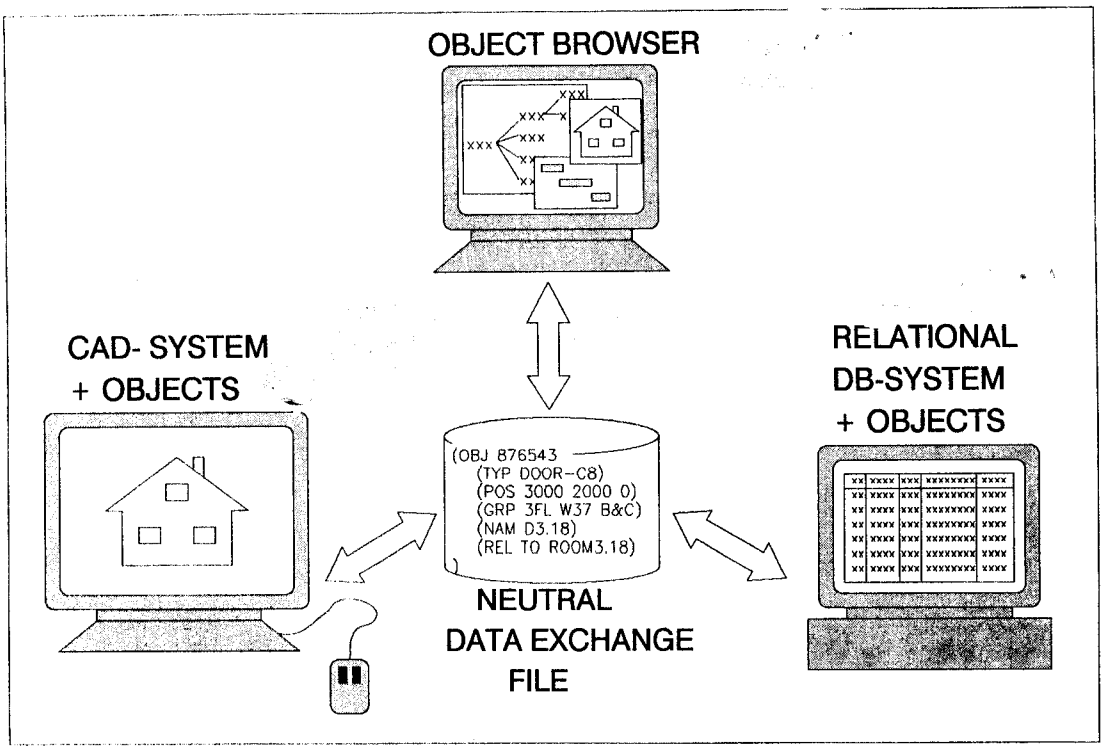


Figure 1. Object oriented implementations described in this paper.

## 1. INTRODUCTION

Because of the large variety of products and different viewpoints of numerous participants the construction industry needs generic i.e. application independent software tools which can be easily used and tuned for specific tasks. Construction industry is too fragmented to support any significant software development unless a common software basis can be used for different applications.

The success of the current generation of CAD-systems can be partly explained by the generic functions provided by these systems for geometric design and drawing preparation.

Unlike many other products which are designed with CAD systems buildings are mainly assembled from prefabricated components and materials. These basic parts are described to a large extent by non-geometrical information. This paper presents some possibilities to enhance CAD and other software tools with the management of non-graphical product data according to a simple generic product model. The model provides a common basis for **data exchange** between dissimilar design and data management systems. It also provides an application independent abstraction of data structures which can be used for the development of various **application systems**.

## 2. OBJECT ORIENTED GENERIC PRODUCT MODEL

### 2.1 Requirements on the generic model

The present generic product model is based on the following functional requirements:

(1) Feasibility for implementation using a variety of software tools.
(2) Open data structure allowing incorporation of new object/entity types, attributes and relationships. Implementations should be able to adopt ad-hoc enhancements and ignore non-relevant or unknown data.
(3) Hierarchical decomposition and networking of data in a viewpoint-dependent way.
(4) Grouping of data according any agreed basis. Standardized building classification codes are an example of a widely used grouping mechanism.

### 2.2 Object orientation

Object orientation was chosen as an approach to map the real world domain of interest directly into a conceptual model and further into software. It provides basis for data abstraction and software modularity. In this contex the folllowing interpretation of what is "object oriented" is adopted:

**Object**: A collection of data which describe a thing or concept in the application domain. Object assigns values to attributes which are defined by classes. Examples of things represented by objects: buildings, functional systems of buildings, building parts, rooms, surfaces, components of parts, connections between parts, activities, organizations etc.

**Complex object**: Object which consists of objects at lower levels of decomposition.

**Identification**: All objects have a unique identifier.

**Encapsulation**: An object contains all data about itself; also about it´s relationships with other objects.

**Classes**: Attributes of objects are defined by classes.

**Inheritance**: Classes may have subclasses which inherit their attributes from one or several superclasses. For example, class "FireDoor" may be subclass of class "Door". It should be noted while **multiple** inheritance is necessary at the the conceptual level there are ways to implement it using **single** inheritance only.

**Polymorphism**: Objects of different classes may have different local representations of corresponding properties. For instance, the "shape" of different things may be represented in class-specific ways.

Figure 2 shows the generic product data model in an IDEF1X diagram where the encapsulation of data in objects is also shown. The boxes are implemented as tables in a relational data base system. Object oriented implementation encapsulates interrelated data into "objects".

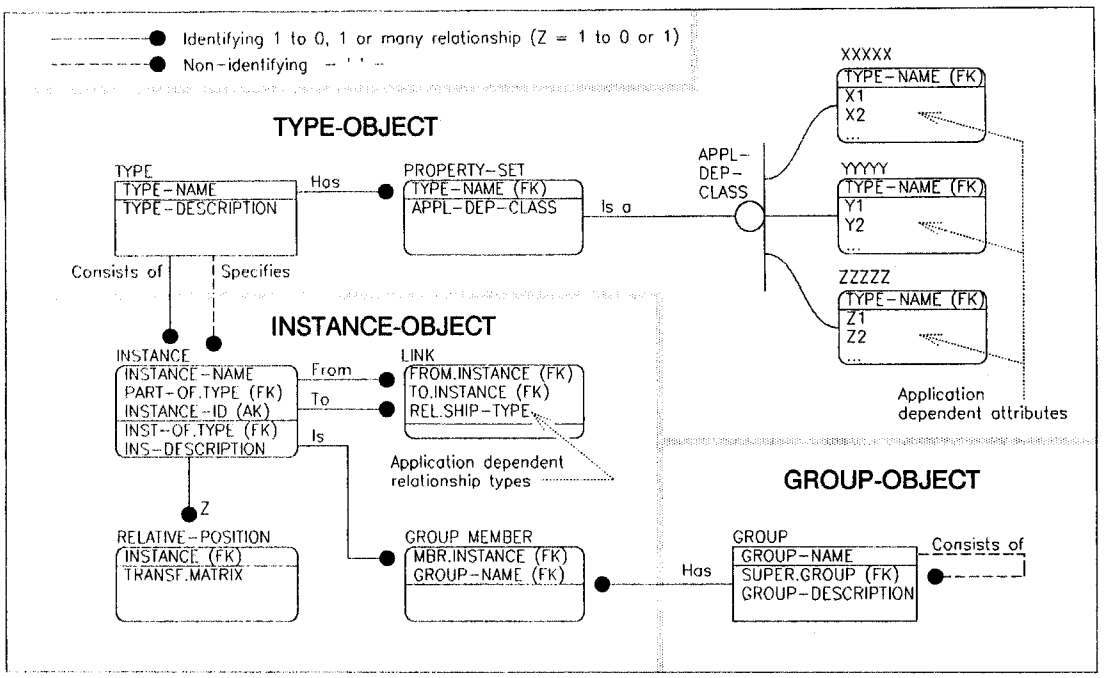


Figure 2. IDEF1X-diagram of the generic product model.

## 2.3 Objects of the generic model

Data of a real world target such as a building consists of two kinds of objects: types and instances. For example, a window type can be described as a complex composition of instances of a frame, glass sheets, a handle etc. Several instances of this window type can occur in different positions of a wall panel. The wall panel again is a type which may have one or several instances in a building etc. It should be noted that a composition of instances is always a type which may itself be instanciated at a higher level of composition. The model of figure 2 is further explained below.

A **type** object encapsulates a set of properties as attribute values. The properties of types are described as **property set**s i.e. one or more application dependent objects which encapsulate the type-specific assigned values of **application dependent class**es. For instance the type "theClassDoorTypeA" may assign specific values to the attributes of the classes "Window" and "Door".

An **instance** object represents an occurrence of one type object. The same type may be referred to by several instances. Thus, similar items can be represented without redundancy. An instance is also part of a type. This hierarchical composition implies existence dependy: if a type is deleted then all instances contained by it shall be deleted, too. The instance object encapsulates an optional **relative position** of the instance with respect to it's father object: a part is located with respect to the assembly. Exeptionally, relative position may be with respect to another instance. For practical reason it is suggested that an instance may be identified by it's name and father type as an alternative to a globally unique identifier. Thus several distinct type objects (e.g. assemblies) may contain instance objects (e.g. parts) which have the same name (e.g. part number).

**Group** objects encapsulate collections of instances as **group member**s.

Relationships between instance objects are represented by **link**s from an instance to another. A link is specified by a relationship type.

## 2.4 Relationships of the generic model

The present model provides three generic relationships which cover a wide range of application needs and can also be easily understood by a human:

**Hierarchy** is used as a tool for viewpoint dependent decomposition of data. Example: a building may consist of spatial, structural and distribution systems. A spatial system may consist of the spaces in horizontal floorplans and vertical communication spaces such as staircases and elevator shafts. A floorplan may consist of appartments which consist of rooms which contain pieces of furniture etc. This kind of hierarchically composed objects (which consist of subobjects etc) are called complex objects. A hierachy relationship as used here actually is a combination

of a) **existency dependency:** if the father object is deleted then all son objects are deleted as well and b) **relative location** (in case of physical things) of a son object with reference to its father object. In most cases these two relationships coinside. Only in exceptional cases separation of existency dependencies between objects and locations of objects relatively to other objects is needed. Splitting the two is necessary e.g in modelling compex connections between physical parts.

Network relationships as **link**s are used to express viewpoint dependent functional interrelations or physical connections between objects. Examples: a beam is connected to a column, a cost is caused by resource utilization, a task is preceded by another task etc. In this paper network relationships are called links emphasizing the encapsulation of a relationship in an object: the object has a link of a specific **relationship type** to another object. Relationship type is an application dependent enumeration which may be associated with application dependent rules concerning existency dependency etc. For instance, the interrelations of a physical joint and the connected structural members are represented by links. These application dependent rules are not considered in the generic model and should not usually be a concern of data exchange either.

**Grouping** structure can be used to create collections of objects for any purpose. Grouping is an important tool of data exchange: various authors of data are likely to organize their data mainly by using (viewpoint dependent) hierarchical and network relationships. However, the private organization of data by an author is not usually relevant to another party having another viewpoint. This is why **classification** standards are emphasized in the construction industry: classification of e.g building parts is used as a grouping criteria. Most CAD systems provide a simple grouping mechanism based on an analogy of transparent "layers": each drawing entity belongs to a layer. In practice more versatile grouping mechanisms are needed. The current model provides hierarchical grouping which can map current classification codes. Simultaneous grouping according to different criteria allows different receivers of data to create flexible access mechanisms rather than enforcing a fixed data structure.

### 2.5 Incorporation of application dependencies

Within an application domain (e.g. AEC-design) certain agreements or standards must be established concerning the attributes etc. of objects in order that different systems can be integrated. The present product model makes a clear distinction between the generic and application dependent parts. The application specific conceptual model (e.g. AEC product model) can be specified within the generic model in terms of groups, link (relationship) types and classes.

## 3. IMPLEMENTATION CONSIDERATIONS

### 3.1 Implementation with a CAD-system

Capabilities to store and manage complex design objects are added to a CAD-system. The focus is on the non-graphical data of objects in order to satisfy the vertical integration between different phases in the building process.

The applied CAD environment, Autocad 11, allthough being basically drafting oriented, provides means for the management of structured (graphical and non-graphical) data:

- BLOCK: Collection of any entities (a symbol) or inserted blocks defined as a single entity. Hierarchical composition is provided by nested blocks. Block itself is not a visible entity.
- INSERT: Instance of a block at a given position.
- ATTRIBUTE DEFINITION: Name, default value and prompt text of a property.
- ATTRIBUTE: Value of an attribute assigned at insertion time.
- HANDLE: Internal unique identifier of an entity.
- LAYER: Grouping mechanism for visibility and selection control.
- AUTOLISP: Interpreted programming language for user defined functions.
- C-INTERFACE: C-language interface for application developers.
- MENU: Taylored pull-down menus.

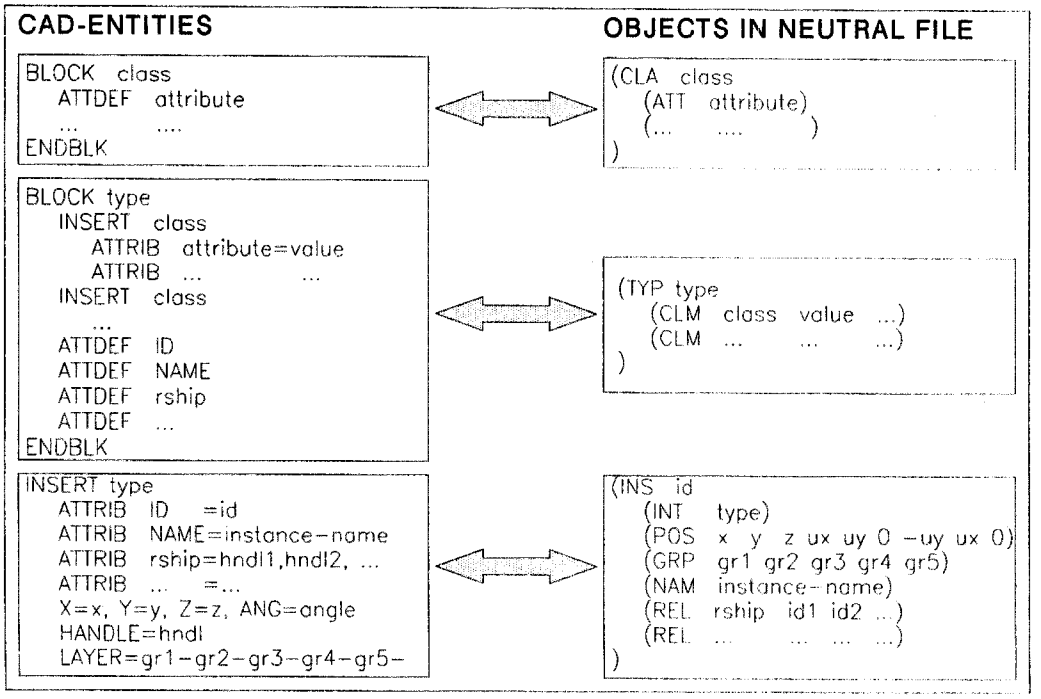


Figure 3. Principle of the CAD-implementation illustrated by mapping between CAD-entities and the physical file.

These capabilities are used in the following way to implement the prescribed generic model:

Attributes of an **application dependent class** are stored as attribute definitions in a block which is named according to the class. The class-block may also contain any class-specific fixed graphics. At the least, a graphical marker symbol needs to be included in the class block in order to provide a visual notification to the interactive user on the screen. The class may refer to an optional Autolisp-program which generates attribute-value dependent graphical presentation of a class member.

**Type**-objects are defined as blocks which contain one or several inserted class blocks i.e. property sets. Types also have attributes which define name, id and links (relationships) of instances. The values of the class-specific attributes or targets of links are given (by the user or by an application program) at insertion time. The type may also include any type-specific fixed graphics, class-related parametric graphics or user-supplied interactive graphics.

**Instance**-objects are created by inserting type-blocks to given positions. Upon insertion time the values of name and id are stored. Relationships with other instances are stored as attribute values containing the relationship type and a reference to the linked object. These references are stored internally as internal identifiers, called handles.

**Grouping** of data is done by means of layers. A layer may have a descriptive alphanumeric name. Given a flat layer structure, hierarchical groups are implemented by using layer names which contain a sequence grouping codes e.g. classification codes as layer names. A variation of this is to use an intuitive grouping mechanism based on layer names as combinations of nemonic codes [Pöyry 1991]. Example: non-bearing internal walls may be assigned onto layer name WAL-INT-NBE-. Because of limited length of layer name at most 5 short codes can be combined into a layer name in this manner which has been proved to be quite useful in practical design.

Management of application dependent entities and relationships is done by Autolisp- and C-language programs which maintain the non-ambiguity of the data structure and also automate some tedious functions.

The ASCII neutral file interface is also implemented as an Autolisp or C program. The interface maps the prescribed internal data structure to the neutral file format. For instance, the internal Autocad handles are translated to identifiers.

Methods of object classes are implemented as Autolisp programs. The names of methods are stored as attribute values to type blocks. Specific user interface programs allows the user to access and exequte methods of an object.

### 3.2 Implementation with a relational database system

A relational data base management tool, dBase IV, is enhanced to allow the user to communicate with the system via an object oriented view in addition to the relational view. The user interface hides the internal data structure which is imposed by the relational implementation and breaks up the encapsulation of data into objects. The system provides functions to extract various reports from the data base. The implementation based on figure 2 is quite straight forward. The main purpose of the relational database implementation is report generation and reorganization of data.

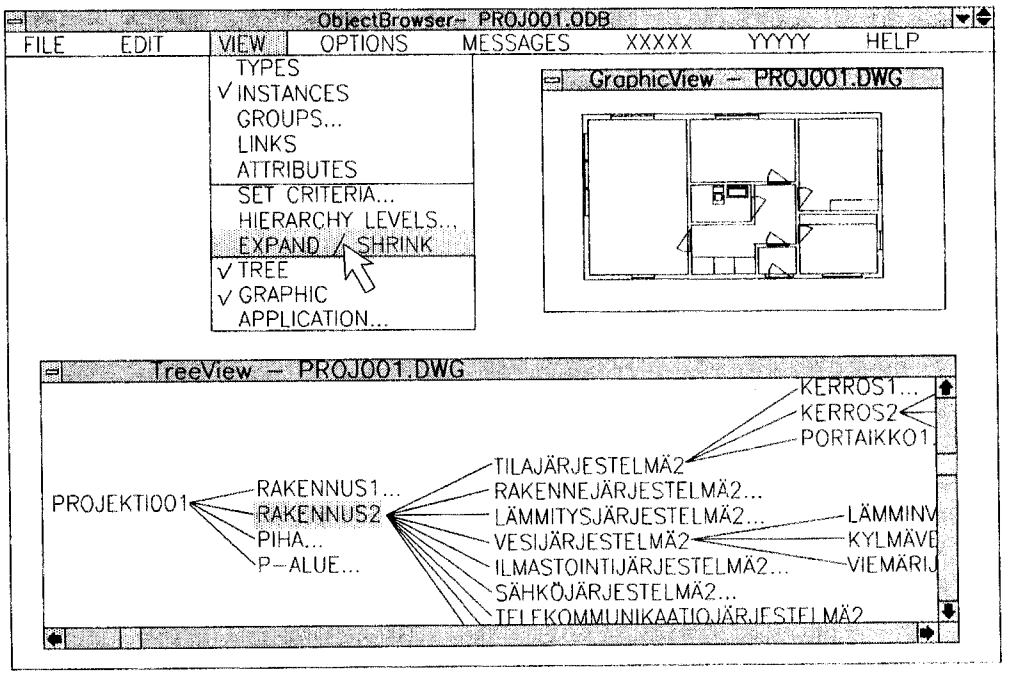


Figure 4. A planned screen view of the OBJECT BROWSER program.

### 3.3 Implementation with object oriented programming

An **object browser** program is implemented using an object oriented programming environment: Actor and MS-Windows. The program provides a set of basic functions to manage object oriented product data, to navigate in the complex object network, select specific views of data and edit data. The program can also be used as a software sceleton which can be enhanced into specific application programs by adding application dependent classes and methods. The object oriented programming language provides a close mapping between the conceptual model and implementation. A library of re-usable object classes essentially reduces the programming effort. The programming environment allows incremental and exploratory application development.

### 3.4 Physical file format

The neutral data exchange file format is based on the syntax of the LISP programming language and provides a close mapping to the generic model, is easily human- and computer-readable and can be flexibly enhanced with new capabilities. The basic element of the syntax is a list which is enclosed by paranthesis. The first item of any list is a keyword which defines the meaning of the remaining parameters. Any parameter may be a list itself etc. Thus a hierarchical structure is described by the nested parenthesis. Objects are stored as packages which encapsulate all data of themselves and are clearly distinguished from other objects. Preliminary comparision indicates that translation to ISO/STEP physical file syntax is fairly straight forward.

```
(CLA Door                          ; Application dependent class
   (ATT width REAL mm)             ; .. defines a set of
   (ATT height REAL mm)            ; .. properties
   (ATT thickness REAL mm)         ; .. as attributes
)                                  ; End of class definition

; Note: If application dependent classes are predefined by
; a standard then class definitions need not be included in
; a data exchange file.

(TYP DoorTypeA                     ; Type object assigns
   (CLM Door  800 2100 50)         ; .. values to properties
   (INS ...)                       ; .. and consists of parts
)                                  ; End of type object
                                   ;
(TYP OfficeTypeX
   (CLM ...)
   (INS 123456                     ; Instance as part of type
      (COM 'Door to room 3.18')    ; Comment text
      (NAM D318)                   ; Instance-name
      (OTY DoorTypeA)              ; Occurrence of type
      (POS 2400 0 0                ; Relative position
           1 0 0 0 1 0)            ; .. and orientation
      (REL FromRoom 234567)        ; Relationships of given kind
      (REL ToRoom 345678)          ; ... with other objects
      (GRM 3rdFloor Week37)        ; Member of multiple groups
   )                               ; End of instance-object
   (INS ...)                       ; More parts of type etc
)                                  ; End of type object

; Note: group objects (GRP ...) need not be included if
; group members encapsulate reference to groups. Group
; names can be part of an application standard.
```

Figure 5. Principle of the data exchange file syntax.

## 4. ACKNOWLEDGEMENTS

The described development is part of an on-going project "Object Oriented CAD" (OOCAD) carried out at VTT with partial funding from TEKES, the national Technology Development Centre.

The current development is a continuation of several previous efforts in Finland: CAD-systems based on a generic object oriented product model were developed by CADEX Oy since 1986 for the design and detailing of building structures. The Finnish concrete industry developed since 1986 a national industry standard, called BEC/STD, concerning the product model and data exchange of precast concrete structures. A generic building product model was sketched out in 1987 within a project called RATAS which has influenced many subsequent efforts.

Some ideas have been adopted from the object oriented programming paradigm as expressed by e.g. the SMALLTALK programming language, but also from the LISP programming language, the PHIGS standard for 3D computer graphics, common data structures of many CAD systems, the well known GARM model [Gielingh 1987] and the evolving ISO/STEP standard.

## 5. REFERENCES

1. Cadex Oy, ConcreteCad and SteelCad systems, Internal information, 1986...90
2. Wim Gielingh: General AEC reference Model, IBBC-TNO, 1987
3. RATAS phase II, part 3: Building product model, Report in Finnish, 1987
4. Matti Pöyry: Private communication on architectural design using Autocad, 1991
5. C++, Borland International Inc, 1991
6. Dewhurst & Stark: Programming in C++, Prentice Hall, 1989
7. C++/Views, CNS Inc, 1991
8. Pinson & Wiener: An introduction to object oriented programming and Smalltalk, Addison-Wesley, 1988
9. Smalltalk/V Windows, Digitalk Inc, 1991
10. Shlaer & Mellor: Object-oriented systems analysis, Yourdon Press, Prentice Hall, 1988
11. Winston & Horn, LISP, Addison-Wesley, 1984
12. AutoLISP programmer's reference, Autodesk Co, 1990
13. AutoCAD user's manual, Autodesk Co, 1991
14. AutoCAD development system programmer's reference manual, Autodesk Co, 1991
15. Actor User's Manual, the Whitewater Group, 1990
16. Object Graphics user's manual, the Whitewater Group, 1990
17. Whitewater Resource Toolkit user's manual, the Whitewater Group, 1990
18. Marty Franz: Object oriented programming featuring Actor, Scott, Foresman and Company, 1990
19. Microsoft Windows Development Kit, Microsoft Co., 1991
20. Charles Petzold: Programming Windows, Microsoft Press, 1990
21. Bertrand Mayer: Object oriented software construction, Prentice Hall, 1988
22. Grady Booch: Object oriented design with applications, The Benjamin/Cummings publishing company, 1991
23. Physical file, Part 21, Standard for the Exchange of Product Data (STEP), Working documents, ISO TC184/SC4