Copyright Warning & Restrictions

The copyright law of the United States (Title 17, United States Code) governs the making of photocopies or other reproductions of copyrighted material.

Under certain conditions specified in the law, libraries and archives are authorized to furnish a photocopy or other reproduction. One of these specified conditions is that the photocopy or reproduction is not to be "used for any purpose other than private study, scholarship, or research." If a, user makes a request for, or later uses, a photocopy or reproduction for purposes in excess of "fair use" that user may be liable for copyright infringement,

This institution reserves the right to refuse to accept a copying order if, in its judgment, fulfillment of the order would involve violation of copyright law.

Please Note: The author retains the copyright while the New Jersey Institute of Technology reserves the right to distribute this thesis or dissertation

Printing note: If you do not wish to print this page, then select "Pages from: first page # to: last page #" on the print dialog screen

NJIT
New Jersey's Science &
Technology University

The Van Houten library has removed some of the personal information and all signatures from the approval page and biographical sketches of theses and dissertations in order to protect the identity of NJIT graduates and faculty.

ABSTRACT

OODINI 2.0: AN ENHANCED GRAPHICAL SCHEMA REPRESENTATION FOR OBJECT-ORIENTED DATABASES

by
Kamil Shahab

The graphical representation of object-oriented database (OODB) schemas is useful for the designers and users of a database system. The aim of our project was to enhance the existing version of OOdini, an interactive graphical tool for editing OODB schema. The new features include interactive modification and description of objects in the schema. Data structures for representing classes and attributes have been altered to incorporate object/data types as well as a descriptive string. The software has been implemented using ObjectMaker, a toolkit to design your own methodology using the ObjectMaker Extension Language.

OODINI 2.0: AN ENHANCED GRAPHICAL SCHEMA REPRESENTATION FOR OBJECT-ORIENTED DATABASE

by
Kamil Shahab

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
In Partial Fulfillment of the Requirements for the Degree of
Master of Science in Electrical Engineering

Department of Electrical and Computer Engineering

May 1999

APPROVAL PAGE

OODINI 2.0: AN ENHANCED GRAPHICAL SCHEMA REPRESENTATION FOR OBJECT-ORIENTED DATABASE

Kamil Shahab

Dr. Mengchu Zhou, Thesis Advisor — Date
Associate Professor of Electrical and Computer Engineering, NJIT

Dr. Raashid Malik, Committee Member — Date
Visiting Professor of Electrical and Computer Engineering, NJIT

Dr. Yehoshua Perl, Thesis Co-Advisor — Date
Professor of Department of Computer and Information Science, NJIT

BIOGRAPHICAL SKETCH

Author: Kamil Shahab

Degree: Master of Science in Electrical Engineering

Date: May 1999

Undergraduate and Graduate Education:

- Master of Science in Electrical Engineering,
  New Jersey Institute of Technology, Newark, NJ, 1999

- Bachelor of Science in Computer Systems Engineering,
  Western Michigan University, Kalamazoo, MI, 1993

Major: Electrical Engineering

ACKNOWLEDGEMENT

I am grateful to Dr. Yehoshua Perl and Dr. Michael Halper for having given me this opportunity and guiding me through this project. I would also like to thank Dr. James Geller for his valuable help and thorough insight into this project.

TABLE OF CONTENTS

Chapter **Page**

1 INTRODUCTION TO OODINI 1

1.1 Introduction 1

1.2 Background 2

1.3 Motivation 2

1.4 Previous Work 3

1.5 General Approach 5

1.5.1 Classes and Attributes 7

1.5.2 Semantic Relationships 9

1.5.3 User-defined Relationships 10

1.5.4 Methods 16

1.6 OOdini 2.0: A Graphical Schema Editor 23

2 THE OODB PART RELATIONSHIP 26

2.1 Terminology and Notation 27

2.2 Definition of the Part Relationship 27

2.2.1 Exclusive and Shared Part Relationships 30

2.2.2 Single-/multi-valued Part Relationships 34

2.2.3 Dependent Part Relationship 36

2.2.4 Value Propagating Part Relationships 37

3 OWNERSHIP RELATIONSHIP 43

3.1 Definition of Ownership 43

3.2 Ownership as an OODB Semantic Relationship 45

3.2.1 Transactions and Inheritance 45

Chapter	Page

3.3 Formal Definition of the Ownership Relationship 47

3.3.1 Exclusive Dimension 48

3.3.2 Value Propagation Dimension 49

3.3.3 Additional Dimensions 50

4 THE ARCHITECTURE OF OBJECTMAKER 53

4.1 Components 53

4.1.1 Diagramming Tool 53

4.1.2 Repository Management 53

4.1.3 View Management 53

4.2 Levels of Functionality 54

4.2.1 Kernel 54

4.2.2 Support Layer 54

4.2.3 Schema Layer 54

4.2.4 Method Layer 55

4.3 Directories and Files 55

5 THE OBJECTMAKER EXTENSION LANGUAGE 56

5.1 What is the Extension Language 56

5.2 Why do we need the Extension Language 56

5.3 What can you do with the Extension Language 57

5.4 Nature of the Language 58

5.5 Rules 59

5.5.1 Rule Head 59

5.5.2 Rule Body 60

Chapter	Page

6 IMPLEMENTING A METHODOLOGY WITH OBJECTMAKER 61

6.1 Menu definition 61

6.2 Diagram Syntax Checking 64

7 CURRENT IMPLEMENTATION OF OODINI 2.0 67

7.1 Actual Implementation 67

7.2 Limitations of OOdini 2.0 67

CONCLUSION 70

APPENDIX A USER MANUAL FOR OODINI 2.0 71

A.1 File Menu 71

A.2 Edit Menu 73

A.3 View Menu 75

A.4 Database Menu 77

A.5 Tools Menu 78

A.6 Help Menu 79

APPENDIX B SOURCE FILES FOR OODINI 2.0 85

B.1 method.cfg file 85

B.2 ood2.mnu file 85

B.3 ood2.rul file 100

REFERENCES 106

TABLE OF FIGURES

Figure **Page**

1.1 Sample University Database 8

1.2 Regular Relationships and Regular Relationship Expansions 15

1.3 Classes and Attributes 17

1.4 Expansions of Classes and Attributes 18

1.5 Semantic Relationships and Methods 20

2.1 Part Relationships 28

2.2 Expansion of Part Relationships 40

3.1 Expansion of the Ownership Relationships 52

A.1 OOdini 2.0 Main Menu 80

A.2 Choosing various symbols using menu bar 81

A.3 Accessing a sub-menu 82

A.4 ObjectMaker Toolbars 83

A.5 ObjectMaker Class - Person 84

CHAPTER

1 INTRODUCTION TO OODINI

1.1 Introduction

The graphical representation of database schemata has been a useful tool for the designers and users of database systems. Such a tool is no longer viewed simply as a convenience, but as a necessity. Our research group at NJIT developed a comprehensive graphical notation called OOdini for the representation of OODB schemata. The OOdini notation is based on a set of mnemonic icons that can be composed in an incremental and intuitive way.

The group created a graphical schema editor called OOdini, which allows users to interactively create and manipulate OODB schemata. This version was implemented in the X Windows/OSF Motif environment. In this report, we will present the complete set of notations supported by OOdini and the new implementation of OOdini 2.0. The OOdini notation incorporates a wide variety of symbols including those for classes, attributes, methods, user-defined and constraint relationships, part-whole relationships, ownership relationships, and semantic relationships---enough to support a diverse group of object-oriented data models. The OOdini 2.0 system that creates and manipulates such graphical schemata is discussed. This system, besides offering constraint-based editing of the aforesaid schema representation, will provide for conversion into an abstract OODB schema language, thus making OOdini an effective OODB graphical interface.

In this chapter, a quick introduction to OOdini is given. Some background on object-oriented database modeling is covered, and the importance of interactive graphical schema is emphasized.

1.2 Background

The designer of an object-oriented database schema is faced with a number of challenges. The foremost among these is the need to create and organize a large number of object classes. The designer must ensure that each class contains the attributes and methods necessary to describe its objects. The classes must also be connected with the appropriate relationships, which convey semantic information and allow for the retrieval of relevant, remote data. To accomplish these tasks, the designer needs a solid grasp of the overall structure of the database.

1.3 Motivation

For the user of an OODB, the requirement of understanding the overall structure may be felt even more intensely. Having not built the database's conceptual model, the user must be provided with a means for becoming acquainted.

The problem of comprehending the structure of a database is not unique to OODBs, but due to the rich modeling capabilities of such systems, this task becomes more difficult. A medium-sized OODB typically comprises hundreds to thousands of classes. Remembering the names of just a few dozen of these and the interconnections between them may be almost impossible.

Consider, also, the need to devise path methods1, 2,3, which are used to retrieve information about a given class stored in other remote classes.

Usually a path method is composed of a sequence of relationships. The seemingly simple task of creating such a sequence can be complicated if the designer or user is only vaguely familiar with the `surrounding landscape.' In order to construct such methods effectively, a general view of the OODB is needed.

With all this in mind, our research group designed a graphical language for the representation of OODB conceptual schemata, a language useful to both schema designer and user alike. The language includes symbols for classes, attributes, methods, user-defined and constraint relationships, part-whole relationships, ownership relationships, and semantic relationships (such as role-of and category-of) — a wide enough variety to satisfy a diverse group of object-oriented data models.

1.4 Previous Work

One of the goals of the latest generation of database management systems (dbmss), including oodbs, is overcoming the problems of representing, storing, and manipulating highly complex data entities[4, 5.] among these are speech signals, cad/cam drawings, and images. invariably, these kinds of data require some form of graphical display. Hence, many oodbs such as ode[6, 7] and o_2[8] support the graphical display of data. however, this type of graphical representation is not the one considered in this paper. our concern is a graphical representation of the database schema, which can be employed as a data definition language[9].

The usefulness of the graphical representation of knowledge-base schemata has long been acknowledged. Early on, the knowledge representation community recognized the importance of graphical aides. Semantic Nets[10, 11] are invariably presented in a graphical form. Conceptual Graphs[12] and Conceptual Dependencies[13] both employ graphical formalisms. Even frames have been given a pictorial form[13].

In the database community, there are a number of data models that present schemata in diagrammatic fashion. Perhaps none of these is more prevalent than the Entity-Relationship (ER) model[14, 15, 16]. In fact, this graphical language is often used as a diagramming device for other data models such as the relational (e.g., Schemadesign[17]). Another semantic data model with a graphical schema representation is Galileo[18], for which a schema editor Sidereus[19] has been built.

Other models that are readily depicted graphically include IFO[20], which is related to the Functional Model[21]. SNAP[22], developed by the originators of IFO, is a system that provides this graphical support. GOOD[23], an object data model also related to the Functional Model, uses a graphical formalism as a basis for its definition.

Within the OODB community, some system designers have considered the graphical representation of the class hierarchy. Among these systems are Ode, Iris[24], O_2, and Ontos[25]. Unfortunately, the class hierarchy relates only a limited part of the interrelations between classes. Kim[26] presents a notation that he calls a schema graph that captures the normal class hierarchy as well as the class-composition hierarchy. The Object-Oriented Entity-Relationship Model[27], an object-oriented extension of the ER model, uses a diagram derived from the ER model. Of late, there has appeared a graphical representation language and editor for GemStone[28].

However, our representation accommodates a larger number of schema constructs in that we graphically represent methods, different generic relationships, and constraint relationships.

In the area of object-oriented modeling and design, there exists a graphical notation that complements the Object Modeling Technique (OMT) [29]. While not specifically aimed at object-oriented databases (but rather object-oriented systems in general), it can be employed to describe database schemata.

As with OODBs, object-oriented programming languages (OOPLs) can greatly benefit from graphical representations. The designers of Eiffel have introduced some graphical conventions in[30]. These conventions constituted a portion of a larger graphical formalism that was under development. As was alluded to by the author, the formalism will focus mainly on aspects unique to OOPLs, such as class preconditions, post-conditions, and invariants.

In[31], Kappel and Schrefl combine the approaches of both fields by presenting object/behavior diagrams for OODBs. Since they are presenting the object diagram in the context of behavior diagrams, they have chosen to represent class interconnections with symbols inside the class construct rather than with connecting arrows.

1.5 General Approach

Following[32], the characteristics of OODB systems are the notions of objects and classes. A class can be regarded as a container for objects that are similar in their structure and semantics in the application. To describe the structure and semantics of objects, the class uses four kinds of properties:

1. Attributes - values of a given data type.
2. User-defined relationships - named references to other classes. Note that we will drop the qualification and refer to these simply as relationships when there is no possibility of confusion (cf. semantic relationships below).
3. Methods - operations which can be applied to instances of a given class.
4. Semantic relationships - similar to relationships in that they are references to other classes; however, these are system-defined, while relationships are user-defined.

The basis for our graphical language is the labeled, directed graph, where both vertices and edges are labeled. The vertex labels allow us to represent the different kinds of classes (see Section 1.5.1 below). Similarly, the edge labels permit the representation of the various semantic and user-defined relationships, and path methods. In many OODBs, relationships are ``buried in pointers," i.e., viewed simply as pointer-type attributes. We follow the approach that emphasizes the importance of relationships. By clearly displaying these class interconnections, our representation provides better expressive power and enhanced readability.

In designing this language, we have taken into account the mnemonic value of the graphical icon. Historical precedents also influenced the choice of symbols. Certain symbols were chosen because of the close correspondence between some object-oriented concepts and those in earlier data models.

Another major factor was our desire to see the graphical representation used as a pencil-and-paper device. The task of constructing a large database schema is an arduous one. Advances often occur away from any computer workstation. The ability to quickly

jot down ideas on paper at such times is a great advantage. Also, some people prefer to do their own designing away from the computer. A notation that permits hand-written diagrams is bound to be of greater utility than one that does not (Witness the great popularity of the ER model). The simplicity of our symbols readily lends itself to this purpose.

The graphical schema representation presented herein has gone through several stages of modification as a result of its use in applications such as: building telecommunication schemata at Bellcore[50], modeling a university environment[33, 34], and modeling a purchasing department[35]. The current version, derived from these experiences, has proven to be both expressive and easily learnable.

The rest of this report is organized as follows. Classes and attributes are discussed in Section 1.5.1. Semantic relationships and User-defined relationships are considered in Sections 1.5.2 and 1.5.3 respectively, while part-whole relationships and ownership relationships are presented in and Section 3.1.

Methods are presented in Section 1.5.4, followed by the ObjectMaker Toolkit and Extension Language in 5.1.

1.5.1 Classes and Attributes

We follow the ER practice and represent an object class as a rectangle with its name printed inside. The attribute names reside inside the class in the form of a list Figure 1.1.

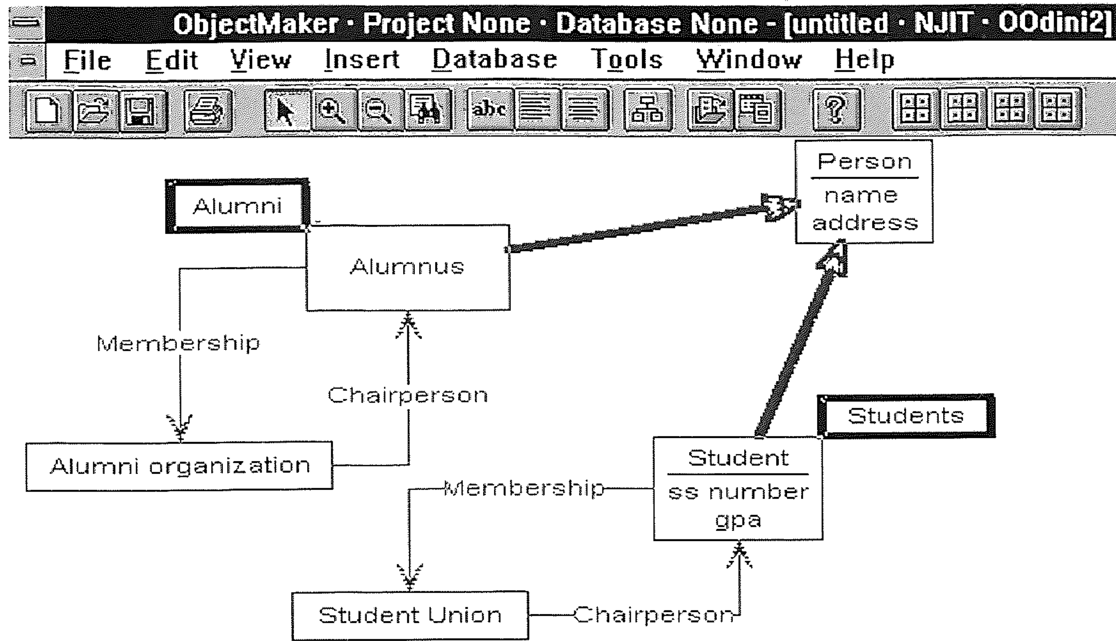


Figure 1.1 Sample University Database

An attribute can be further classified as essential, meaning that its value must be non-nil. In this case, a special character is added to the end of the attribute name inside the class.

In addition to a simple class, our system is capable of representing composite classes obtained from other classes by some type of constructor:

1. The set constructor.

The set constructor is used to obtain a class whose instances are sets of instances of another class. For example, the class alumni of Figure 1.1 is obtained by applying the set constructor to the class alumnus. Such a class might have an instance representing the set of all alumni who have made a donation to the school.

The graphical representation of a set class is a rectangle with a thick border. The thick border is used to convey the multiplicity of sets and their non-atomic nature.

To summarize, the symbol we use for an object class is a rectangle. Set classes are represented using rectangles with thick borders.

1.5.2 Semantic Relationships

As mentioned above, we use the term semantic relationship to refer to a connection between classes which, due to its generality and importance, is system-defined (or, in other words, is a modeling primitive of the system).

The most important semantic relationship is subclass (is_a), which enables the expression of specialization and the creation of a class hierarchy. This hierarchy normally forms the skeleton of an application, and its comprehension is essential to an overall

intuitive understanding. Thus, in any graphical representation, the hierarchy must be emphasized. For this reason, we have chosen to specify subclass as a heavy line directed from the specialized class (subclass) to the more general class (superclass). As we shall see, user-defined relationships are represented using thin arrows; therefore, subclass is duly highlighted, and its hierarchy is readily apparent on even the most cursory inspection. To further emphasize the hierarchy, we encourage the placement of a subclass below its superclass.

In the case where the subclass specialization is in a different context from that of the superclass, we call the generic relationship role-of[1, 2,3.] The graphical representation for role-of retains the heavy arrow of subclass; however, the line is not solid, but a dot-dash pattern. The mnemonic device employed here is borrowed from the world of maps. There, the boundary between any two territorial units, such as states or countries, is defined using a dot-dash pattern. In our case, we denote the crossing of the boundary between contexts.

Another semantic relationship that we represent is part-of, which is used to connect a part of a complex or assembled (real-world) object to its integral object. In the next chapter, the entire part-whole relationship will be discussed in great detail.

1.5.3 User-defined Relationships

A relationship is a named, user-defined connection directed from one class to another. Since it can be viewed as a pointer, we draw it as a labeled arrow from its class of definition to the target class. The arrow is thin as compared with the heavy arrows of the hierarchical generic relationships.

Often an application requires a relationship from a class A to a class B, as well as its converse. This situation is handled using a pair of arrows pointing in opposite directions. One should contrast this approach with the ER model, where a relationship is bi-directional and given an `existence' of its own, complete with its own attributes. In OODBs, a relationship is typically defined as a property of one class, acting as a reference to another class.

The ER model supports one-to-many or what we call multi-valued relationships. The object-oriented approach supports multi-valued relationships in two different ways. The first is a multi-valued relationship connection, which indicates that an instance of one class can be related to any number of instances of the class to which the relationship is directed. An example of this is the relationship between the classes section and student, where a given section can have many students. We have chosen to represent the multi-valued relationship as a dual-lined arrow. This choice emphasizes the multiplicity of the relationship, just as in the case of the set-constructed class (cf. Section 1.5.1).

The second alternative is to define a set class. In this case, creating a set class at the `many end' and directing an ordinary single-valued relationship to it captures the multi-valued relationship. Consider a related example: a student can be in many sections. Using the set alternative, we create a new class sections, defined with respect to section. We then create a single-valued relationship from student to sections. In this way, we have related a single student with many sections. Here, however, we are required to explicitly group the section objects into a set.

While the two approaches are basically equivalent, the usefulness of the set class alternative becomes apparent when trying to model relationships with cardinality

constraints. Assume that we are trying to model the interrelations between courses, instructors, and students. We first define the classes course, instructor, and student. Because there are a number of sections offered for each course, we also need a class section. Now assume the following constraints:

1. At most r sections of a given course can be offered in a semester.
2. An instructor may teach no more than s sections in a given semester.
3. A student may take at most t sections per semester.

We could model this situation by having relationships from each of the three classes course, instructor, and student to a set class sections, defined with respect to section. Sections would be given the attribute number to maintain the cardinality of an instance, as well as an attribute maximum that would hold the maximum cardinality. This latter attribute would be set at instantiation time to an appropriate value (e.g., to r if the set were to consist of the sections of a particular course). The method to add an instance of section to a given instance of sections would then check the current cardinality and deny any request that would violate the prescribed maximum.

These cardinality constraints could alternatively be enforced by each of the two classes course, instructor, and student individually. Placing two additional attributes, numberofSections and maxNumberofSections, in each of the three classes can do this. These attributes play the same roles that the attributes number and maximum did in the class sections above. Next, multi-valued relationships are established between each of the classes and section. Lastly, each class is equipped with methods to monitor the constraints.

There are a number of reasons why this is not as elegant a solution as the former. First, the multi-valued relationships do not convey information about the required cardinality constraints. The set alternative makes the structure of the model more meaningful — more semantic. Second, the cardinality constraint is really a characteristic of the set of sections associated, for example, with a given course, not of the course itself. Hence, this constraint should be defined as a property of sections rather than course. Finally, placing the two attributes and the corresponding `watchdog' method in sections, instead of repeating them three times eliminates redundant specifications.

Constraint relationships are those which impose additional semantic constraints on the participating classes. In general, a constraint relationship requires two aspects of definition; the static or state definition which imposes constraints on the database at any fixed instant of time; and the dynamic or transient definition which expresses the behavior that it implies in the context of change (i.e., the creation, deletion, and update semantics). The dynamic aspect of any constraint relationship is required to maintain the constraints imposed by the static aspect.

We have chosen to represent two types of constraint relationships, essential and dependent, though more could be added in the future. Both of these relationships are normally used to maintain referential integrity[9].

An essential relationship is one which must always refer to an existent object (i.e., which may not have a nil value). Its creation semantics is such that the referent class of the relationship must have instances before any instances of the source class can be created. The update semantics is; the relationship cannot be assigned a value of nil.

Finally, the deletion of an instance of the referent class is forbidden if there exist instances of the source class which refer to it.

To represent an essential relationship, we place a small circle behind the head of the arrow representing such a relationship. This symbol was chosen to maintain consistency with respect to the rest of the graphical representation, as essential attributes are also denoted by the addition of a circle at the end of the attribute name. Hence, adding a circle to an attribute name or relationship consistently expresses essentiality. In Figure 1.2, we see the essential relationship.

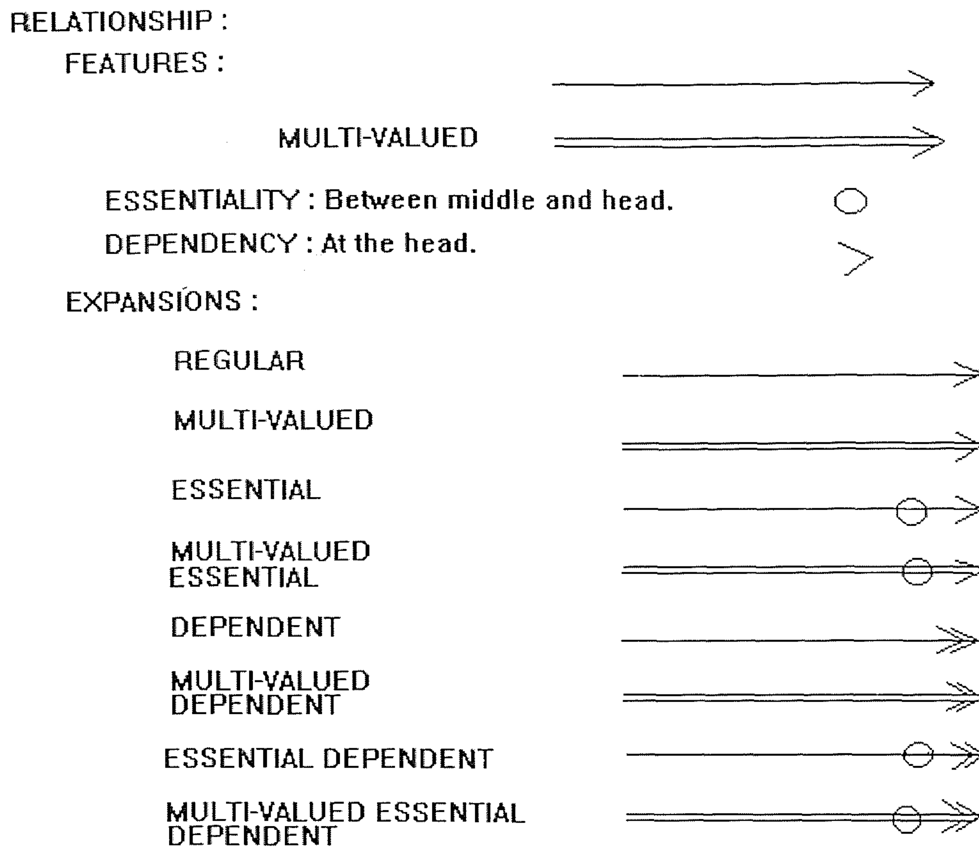


Figure 1.2 Regular Relationships and Regular Relationship Expansions

A dependent relationship is identical to an essential relationship except for the deletion semantics: Assume that the class A has a dependent relationship to class B; if an instance a of A refers to an instance b of B, and b is deleted, then a is also (automatically) deleted. Thus, the existence of an instance of A is dependent on the existence of an instance of B. We represent a dependent relationship as a double-headed arrow (either single-lined or dual-lined). The double head of the arrow emphasizes the ``stronger'' connectivity of this type of relationship. In Figure 1.2, we see the dependent relationship.

1.5.4 Methods

We distinguish between two types of methods in OODBs: path methods and local methods. As their name implies, local methods operate strictly locally to an object; i.e., no remotely accessed data is used in their operation. Local methods can be divided into selectors/mutators (also referred to as readers/writers) and derived attributes[29]. A selector (mutator) method simply reads (writes) a given attribute. Selectors and mutators do not require separate graphical representations. The symbol representing the attribute they operate on is sufficient.

Derived attributes are very similar to the selectors of attributes. These methods derive values from one or more attributes through some computation. An example of a derived attribute is the `available' method of the set class sections. This method computes the available `room' in a given set by subtracting the attribute number from the attribute maximum. Derived attributes require a unique symbol at the end of their name in the list. The reason for our choice is that a derived attribute can be viewed as a hybrid of an attribute and path method. In Figure 1.3, we see the derived attribute.

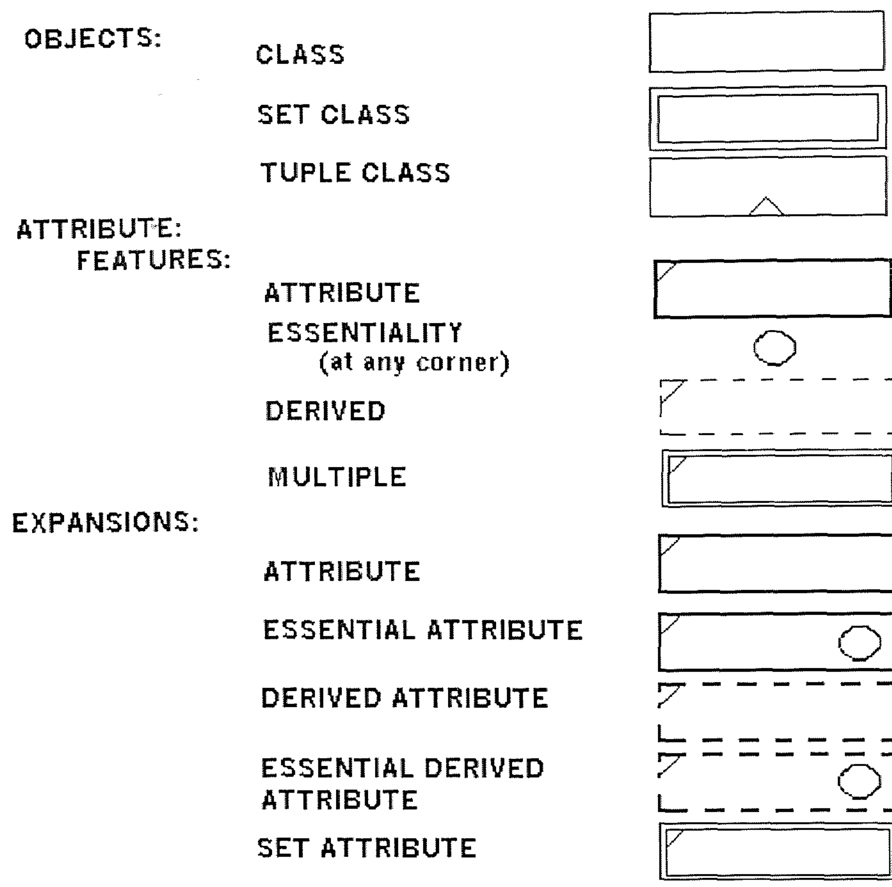
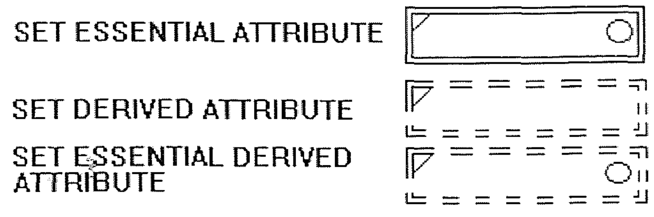
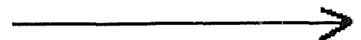


Figure 1.3 Classes and Attributes



SEMANTIC RELATIONSHIP :

SUBCLASS



ROLE-OF

METHOD :

PATH METHOD

ATTRIBUTE PATH METHOD

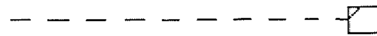


Figure 1.4 Expansions of Classes and Attributes

A path method is an operation (defined on a class) comprising a chain of classes connected by semantic relationships and/or user-defined relationships. In general, a path method can be concatenated with mathematical operations. However, these need not be represented graphically and are omitted from our discussion. The symbol employed for path methods is a dashed, thin-lined arrow pointing from the class defining the method to the remote class it accesses (i.e., ends in). The reason for the choice of this symbol is as follows. The function of a path method is similar to that of a relationship: Each is used to retrieve relevant information from another class. We therefore chose the thin arrow so as to make the symbol for a path method reminiscent of the representation of a relationship. However, there is a difference between relationships and methods. A relationship is a direct connection, while a method is an indirect connection established via a chain of connections.

In this sense, a method can be viewed as a composite construct, and so we once again employ the dashed-line convention (cf. The representation of the part-of relation). An attribute path method is a special case of a path method. It is an operation comprising of classes connected by semantic relationships and/or user-defined relationships that will end with an attribute or derived attribute. The symbol employed for attribute path methods is the same symbol as path methods except the class at the end of the chain will have a box on the outside that will point to the class that contains the attribute, instead of the usual arrowhead. In Figure 1.5, we see the attribute path method.

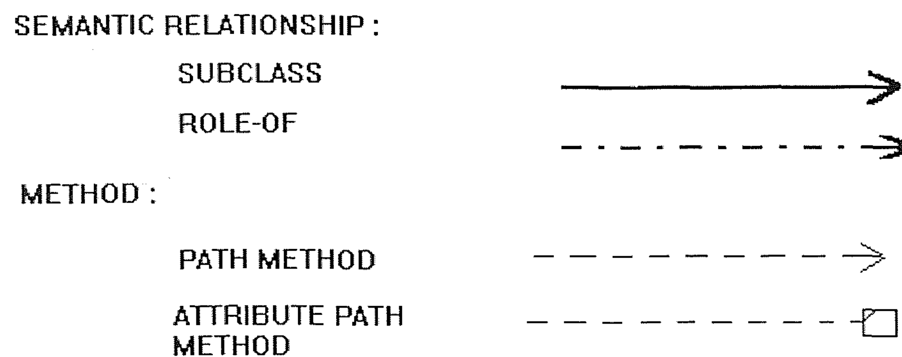


Figure 1.5 Semantic Relationships and Methods

As an example, consider a method called `get_courses' that would consist of the class instructor. This method returns the names of all the courses taught by a particular instructor. To accomplish this, it accesses the attribute name of course through the relationships path teaches, setof, is_offering_of. More specifically, it operates as follows. It starts by applying the relationship teaches to instructor yielding sections. Applying setof then gives a set of section. Next, applying is_offering_of yields a set of course. And, finally, applying the attribute name to each instance of this set produces the desired result; a set of course names. Since the desired data is stored as the attribute name of the class course, we represent this method as a dashed line pointing from instructor to the name attribute of class course.

There are actually two aspects of a path method that should be represented graphically. The first one, which we have just presented, displays the connection between the source class and target data item of the method. This aspect reflects the retrieval effect of the path method (i.e., what data it actually returns). As we mentioned above, this aspect functions similarly to a relationship, and hence was given a graphical symbol similar to the relationship icon.

The second aspect of a path method is the chain composed of classes that are connected by relationships. This aspect reflects the implementation of the retrieval mechanism. Clearly, it is a critical portion of the definition of a path method. Without it, one cannot judge if the method is semantically correct, i.e., whether it correctly retrieves the desired information.

Obviously, we would like a graphical representation of this second aspect as well. Since it is a chain composed of elements that are already represented graphically, it is natural to simply highlight those elements in some manner. For example, in the context of an editor program (such as the OOdini system), the elements could be emboldened or given some tiling pattern or color.

We note, however, that highlighting the chains of all path methods in a schema will leave much of the schema highlighted and render most chains unrecognizable due to overlaps. Therefore, this highlighting must be used sparingly. In this sense, it is similar to italicization in written natural language. If overused, it becomes confusing and ineffective. In fact, it has been our experience that a designer or user is not interested in the chains of all path methods simultaneously. Typically, one wants to concentrate on the implementation of a single method. In such cases, only the method of interest would be highlighted. If one wants to determine all the available methods, the retrieval aspect represented by the dashed line is sufficient. Therefore, we view the highlighted aspect of the representation of a path method as optional. If employed, it should be restricted to a small number of methods.

The representation of a path method that is a branching method, i.e., that accesses data from two or more remote locations, remains a topic for further research.

For a summary of all the graphical symbols that we have introduced, see Figure1.1, 1.2, 1.3, 1.4, 1.5, 1.6, and 1.7. Note that text to the left is descriptive material and is not part of the symbols.

1.6 OOdini 2.0: A Graphical Schema Editor

We are in the final stages of the development of a software system called OOdini 2.0 (Object-Oriented di agrams, N ew Jersey I nstitute of Technology) which allows the user to interactively manipulate the graphical schema representation for OODBs defined above. The system is being implemented on a Sun 4/20 workstation and also on an IBM Personal Computer. ObjectMaker Tool Developer's Kit is providing Graphical and interface supports. OOdini's main screen can be seen in Figure A.1.

OOdini is a constraint-based graphical editor designed specifically for the schema representation we have presented above. We note that it does not include the features of a general graphical-constraint toolkit such as Garnet[36]. To see what we mean by constraint-based, consider the representation of a relationship. In particular, consider a relationship emanating from a class and left dangling, that is, unattached at its other end. Clearly, such a construction is meaningless. So, during input OOdini 2.0 will mark such a diagram with an anchor on the dangling end. This representation will alert the user that the diagram is not drawn properly. Moreover, if at a later time one of these classes moves, the relationship is automatically moved relative to it. In this way, OOdini guarantees that the integrity of the schema diagram is always maintained, and it relieves the user of a lot of tedious manipulation.

As with most software systems built on top of a windows environment, OOdini relies heavily on the mouse for interaction with the user. The keyboard is required occasionally in response to a dialog box to input textual data, such as the name of a class. The interaction with OOdini during schema creation follows a regular pattern: The user selects a symbol, such as a class or a relationship, from either the menu or the palette and

then proceeds to add any number of instances of that symbol to the schema. The status bar at the bottom of the screen will inform the user of what the current symbol of insertion is. Also, the palette will indent the current symbol of insertion. When finished with this `current' symbol, the user chooses another and further expands the schema. This continues until the schema is complete, at which time the user can request that it be saved to disk or printed. Of course, the system also provides facilities for modifying the schema. Such features include interactive movement of schema objects using the mouse; interactive labeling of schema objects using the keyboard, as well as interactive deletion, again carried out with the mouse. We emphasize that all modifications of the schema are constraint-based. For example, the movement of a class always entails the movement of all its associated graphical objects (e.g., attributes, relationships, set classes, and so on). Likewise, the deletion of a class propagates into the deletion of those associated objects.

OOdini manages a large drawing canvas, allowing the designer to create very large schemata. This is a very important characteristic of the system since OODBs typically comprise many hundreds of classes. A tool that provides but a single `sheet' becomes totally worthless for such applications. Scrollbars are provided to allow the user to reposition the current working window (in the ordinary graphics sense) of the canvas. Using the scrollbars, the user can readily pan left and right, or up and down.

While it is possible for the user to quickly navigate to and view any portion of the canvas, the current working area presents only a small fraction of the entire schema. It is normally not possible to display a schema of substantial size in its entirety with a reasonable magnification. To give the user the possibility of viewing the schema `globally,' we provide a zooming mechanism. The zooming feature is under the view

menu on the toolbar. The Zoom feature has 5 levels of zooming. The user can zoom in or zoom out. The current level of zoom is indicated on the status bar at the bottom of the screen. This feature of OOdini 2.0 is particularly useful when it comes to rapidly moving between distant regions of the schema. Of course, the scrollbars could do the same thing, but using them is more tedious; the destination area is not in full view, and it is likely that the user will end up doing some scrollbar `oscillating' during the search. In general, it is expected that the scrollbars will be used to make fine position changes to the current working region, while the roadmap will be employed for large jumps.

To maintain conformance with other standard windows type systems, the main system screen is laid out with the preferred menu bar and work area arrangement. (See [37] for further details.) At the uppermost portion of the main window is the menu bar which contains the normal array of entries, `File,' `Edit,' `View,' `Insert,' `Database,' `Tools,' `Window,' and `Help.'

For more detailed information on the menu bar, please see appendix A at the end of this paper.

CHAPTER

2 THE OODB PART RELATIONSHIP

If OODB systems are to fulfill their expectations in different areas, it is imperative that they support aggregation by including a part-whole relationship as a built-in modeling primitive. By such a relationship we mean a connection between two object classes that provides more than just a common name like "part-of". Rather, it must capture accepted real world, part-whole semantics by imposing limitations on the interactions between the instances of the participating classes and by providing them with additional functionality befitting parts and wholes.

Our part model has as its foundation a part-whole semantic relationship that encompasses the following:

- Constraints that impose appropriate "part-whole" restrictions on the state of the database and the various part transactions (like "add-part and "remove-part").
- Dependency between part and wholes.
- Inheritances of properties, both from part to whole and vice versa.

Because there exists a wide range of part-whole semantics, we organize the above into four characteristic dimensions: (a) exclusiveness, (b) multiplicity, (c) dependency, and (d) inheritance. Each of these dimensions can take on a number of different values, giving flexibility to an application developer or schema designer, who simply declares

the desired semantics by choosing the appropriate values. The OODB system then automatically ensures that the chosen semantics is obeyed during the entire lifetime of the database.

2.1 Terminology and Notation

Following[38], we will refer to a `part' as a meronym (the prefix mero-, from the Greek meros, meaning part). A whole object will be called a holonym (holo- meaning whole). A part's class is a meronymic class, whereas that of a whole is a holonymic class. For example, if classes chapter and book are in a part-whole configuration and chapter c is part of book b, then c is a meronym and b is a holonym. Chapter and book are the meronymic and holonymic classes, respectively.

2.2 Definition of the Part Relationship

In this section, we present a formal definition of a part relationship between a pair of OODB classes. This relationship is described formally as a quintuple comprising a relation between the extensions of the participating classes, and four `characteristic' dimensions: (1) exclusiveness, (2) cardinality, (3) dependency, and (4) value propagation. The first of these addresses the issue of how parts may be distributed among wholes. The next is concerned with the way parts of the same kind are collected together to form wholes. The third dimension deals with the dependency semantics, i.e., how the deletion of a holonym or meronym affects its counterpart in the part-whole configuration. The final dimension addresses the issue of propagating relevant data across the part

relationship from the whole to the part, or vice versa, leading to the definition of derived attributes.

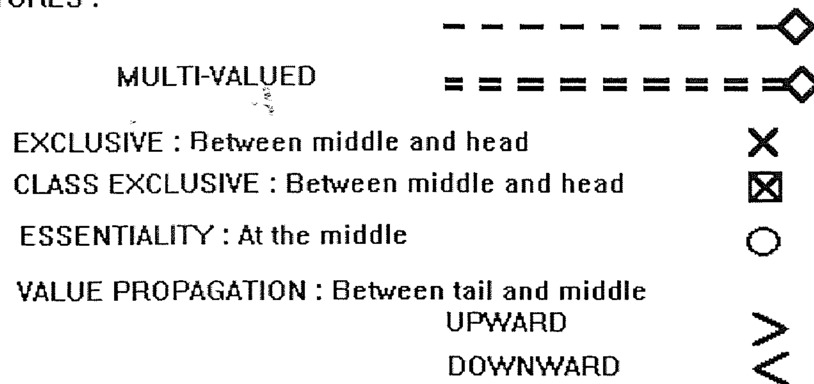


Figure 2.1 Part Relationships

For a class C, let E(C) denote the extension of C (i.e., the set of all its instances). Also, let $\Pi(C)$ denote the set of all properties of C. A part relationship between a meronymic class B and holonymic class A (written $P_{B,A}$) is defined as the following quintuple:

$$P_{B,A} = \langle \Diamond, \chi, \kappa, \delta, (\pi_{Bi}, \pi_{Aj}) \rangle$$

where $\Diamond$ is a relation from E(B) to E(A). The pair (b, a) $\in \Diamond$ indicates that the instance b of class B is part of the instance a of class A. We will ordinarily express this fact in an infix expression as b $\Diamond$ a. At times, the relation $\Diamond$ may carry a subscript to distinguish between multiple part relationships. For example, if we have a part relationship $P_{B1,A}$ $P_{B2,A}$ between the class B_1 B_2 and A, then we would write the constituent relation as $\Diamond_1$ ($\Diamond_2$).

The next three elements χ, κ, and δ represent the values of the exclusiveness, cardinality, and dependency characteristic dimensions, respectively. Their values are as follows:

Note that the value of delta may be nil indicating the lack of any dependency. The last dimension, the value propagation dimension, comprises a pair of sets π_{Bi} and π_{Aj}, where $\pi_{Bi} \subset \Pi(B)$, $\pi_{Aj} \subset \Pi(A)$, and $\pi_{Bi} \cap \pi_{Aj} = \varnothing$.

Complete accounts of each dimension will be given in subsequent subsections, where formal descriptions are provided. To accomplish this, we will need the following two definitions. Assume that there exists a part relationship $P_{B,A}$.

Definition :

$$\forall\, a \in E(A), \text{ let } M_{\Diamond(a)} = \{b \mid b \in E(B) \wedge b \Diamond a\}$$

$M_{\Diamond(a)}$ is called the meronym set of a with respect to the part relationship $P_{B,A}$, i.e., the set of instances of B which are parts of a.

Definition :

$$\forall\ b \in E(B), \text{ let } H_{\Diamond(b)} = \{a \mid a \in E(A) \wedge b \Diamond a\rangle$$

$H_{\Diamond(b)}$ is called the holonym set of b with respect to the part relationship $P_{B,A}$, i.e., the set of instances of A of which b is a part.

The graphical symbol for the part relationship serves as the basis for a rich set of symbols that denote the various semantics of the part relationship. The symbol is a bold, dashed line connecting the meronymic and holonymic classes. A diamond head at one end of the line indicates the holonymic class. See Figure 2.1. The symbol carries a mnemonic device: "the pieces of the line indicate the parts of the object".

Also, the bold line serves to highlight the part relationship, making its hierarchy clearly recognizable as a backbone of the graphical schema.

2.2.1 Exclusive and Shared Part Relationships

Part relationships in general can be divided along the lines of exclusive and shared[39,40].

An exclusive part relationship enforces the restriction that a given meronym can be a component of only a single holonym. In other words, the holonym is the sole owner of the meronym. Of all the part relationships we will introduce, the exclusive relationship is perhaps the most intuitive because part modeling is most often associated with physical assemblies such as cars, bridges, and buildings, things that one can "go out and kick"[41]. For such items, the exclusiveness restriction is quite natural: Two cars cannot share the same engine.

While no two cars can share an engine, it is also the case that a car and, say, an airplane cannot share one either. Therefore, the exclusive part relationship between the classes engine and car must have ramifications for the entire database topology, restricting not only "part" references from cars to engines but from objects of other classes to engines as well. There are times, however, when we would like to confine the exclusive reference restriction to a single holonymic class.

Consider a computer science publication database which contains scholarly journals and books (and, in particular, books which are compilations of articles). If we were to diagram this database, we would use the generic part relationship symbol to indicate that class article is in a part relationship with both journal and compilation (the latter being a subclass of book). Ordinarily, different journals do not contain the same article. Therefore, it is sensible to impose this constraint on the database. However, an article can appear as part of some compilation (a common

practice in the area), and so we do not want the exclusiveness constraint between article and journal to have any implications on the relationship between article and compilation.

For this reason, we distinguish between two types of exclusiveness, global exclusiveness and class exclusiveness. An exclusive part relationship, such as the one between engine and car, which affects the entire database topology will be referred to as a global exclusive part relationship. This kind can be found in a number of existing systems (e.g.,[39,40]), where it is simply called the exclusive part relationship. We too will usually drop "global" and just call it exclusive. The class exclusive part relationship is one that only enforces the exclusiveness constraint on the relationship between the

participating classes, as between article and journal. Both the exclusive and class exclusive relationships will be given formal definitions and their own graphical representations below.

Part relationships that are not exclusive are called shared[39]. A shared part relationship puts no restrictions on the number of holonyms that a given meronym can be part of, allowing the meronym to be freely shared. The part relationship between article and compilation in the example discussed above is shared. The same article can be included in any number of compilations.

To formalize the definitions of these three part relationships, we place different constraints on the cardinality of the holonym sets of the parts. It is interesting to note that the resulting theoretical ordering with respect to the extent of the constraints (namely, shared, followed by class exclusive, and then exclusive) is exactly opposite to the "intuitive" ordering that we used to introduce these part-relationships.

Definition :

$B \Diamond A$ is a shared part relationship iff there exist no constraints on $| H_{\Diamond(b)} |$, $\forall\, b \in E(B)$

In particular, the cardinality of such sets must be allowed to be greater than one. Because of this lack of constraints, we use the generic graphical symbol to represent the shared relationship. (Actually, our generic symbol denotes a single-valued, shared part relationship. Sharing may also be combined with multi-valuedness that is discussed in the next section.)

Definition :

$B \Diamond A$ is a class exclusive part relationship iff $\forall\, b \in E(B)$, $| H_{\Diamond(b)} | \leq 1$

In other words, it is class exclusive iff $\Diamond$ is a partial function from E(B) to E(A).

To express the fact that a part relationship is class exclusive, we add a rectangle enclosing an "X" directly behind the diamond head of the generic symbol. The "X" is derived from "eXclusive." The rectangle, our symbol for a class, brings to mind "class." An example of the class exclusive part relationship can be seen in Figure 2.2.

The global exclusive part relationship between a pair of classes B and A is equivalent to a class exclusive relationship if it is the only one that B participates in. Therefore, assume that B is in part relationships $\Diamond_1, \Diamond_2, \ldots, \Diamond_n$ with classes $A_1, A_2, .., A_n$, respectively.

Definition :

The part relationship $\Diamond_i$ is global exclusive (exclusive, for short) iff

$$\forall\, b \in E(B), \text{ if } \exists\, a \in E(A_i) \text{ such that } b \Diamond_i a, \text{ then } \forall\, j \neq i, |H_{\Diamond j(b)}| = 0.$$

In other words, if an instance of B is part of an instance of A_i, then it cannot be part of any other object in the database. Thus, the part relationship $\Diamond_i$ affects the entire database topology; in particular, it puts constraints on all the other part relationships that B participates in as the meronymic class.

The graphical symbol used to represent the exclusive relationship is the generic symbol with an ``X" placed behind the diamond head (Figure 2.2). Here, the ``X" should be read as the global eXclusiveness constraint.

2.2.2 Single-/multi-valued Part Relationships

The holonyms in a part relationship may have a single part from the meronymic class or they may have many. To accommodate these situations, we introduce a number of single-/multi-valued part relationships, defined in terms of constraints on the $M_{\Diamond}$'s.

The generic part symbol aptly expresses the single-valuedness of this part relationship as it is a single-lined connection (Figure 2.2). The mnemonic here is "single line equals single part". This is in contrast to the multi-valued part symbol where a dual line is employed to convey multiplicity (Figure 2.2). The multi-valued relationship is defined presently.

We note that according to our definitions the characteristics of exclusive/sharing and single-/multi-valuedness are completely independent of each other and can be freely mixed and matched to form such part relationships as the single-valued, shared; single-valued, class exclusive; multi-valued, exclusive; etc. (In fact, as we noted earlier, the generic symbol in fig3b is actually the single-valued, shared part relationship.) Because of this orthogonality, we demonstrated the graphical symbols for the exclusive/shared variations without any regard to single-/multi- valuedness. Likewise, in this section, we will illustrate the graphical symbols without regard to exclusiveness/sharing.

Pictorially, the range-restriction is shown as a numerical range alongside the dual-lined symbol of the multi-valued part relationship. Note that even though we are using parentheses, the range is interpreted to include both endpoints. The upper or lower bounds of a part relationship may be omitted for an "m or greater" or "0 to n" interpretation. Graphically, a dash replaces the omitted bound.

Definition :

The fixed-cardinality part relationship is a range-restricted part relationship with m = n. The value of m is said to be the multiplicity of the relationship.

If only engines with six cylinders are of interest in the database, then a fixed-cardinality relationship of multiplicity six would be used to model this. The notation for this relationship is similar to that for the range-restricted, except that the upper and lower bounds are consolidated into a single number.

Definition :

The essential part relationship is fixed-cardinality relationship of multiplicity 1.

If we wish to require that all cars in our database have an existent engine, we would employ an essential part relationship to model this. Since the essential relationship is not actually multi-valued, we drop the dual line from its symbol. We also forgo the "1" in parentheses and instead place a circle directly on the broken line (Figure 2.2). This representation follows the convention we introduced in[42] where we indicated the essentiality of attributes and "ordinary" relationships using a circle. (This notation is also consistent with that used in[43].)

Definition:

The multi-valued, essential part relationship is a range-restricted part relationship with lower bound 1 and no upper bound.

We also use the circle for this relationship, but here we maintain the dual line.

2.2.3 Dependent Part Relationship

A part relationship can be endowed with different forms of dependency as specified by the domain of the third characteristic dimension:

part-to-whole, whole-to-part, nil

The third value indicates that the part relationship lacks any dependency semantics.

Dependency semantics is often desired when modeling with parts[39], especially when the holonyms comprise numerous meronyms. Such a scenario is illustrated in a CAD drawing along with its many parts. Having the parts deleted automatically on the deletion of the drawing alleviates the burden of deleting them manually. As is meant to be conveyed by the picture, such a process can be tedious and time-consuming. For this reason, we include part-to-whole dependency in our model.

There are some part-whole configurations where the part acts as a defining element, without whose existence the whole becomes insubstantial. Consider, for example, that without its frame, a bicycle may be seen as nothing more than a collection of "spare" parts. Therefore, it makes sense to propagate the deletion of a frame into the deletion of its bicycle. We refer to this as whole-to-part dependency.

To be more precise about the two types of dependency, we define part relationships that exhibit these characteristics in the following. There, we use the notation del(x) to denote the application of a method to delete the instance x[39]

Definition :

The part relationship $P_{B,A}$ is part-to-whole dependent (i.e., δ = part-to-whole) if

$$\forall\, a \in E(A)\ del(a) \rightarrow \forall\, b \in E(B) \text{ such that } b \Diamond a\ \wedge H_{\Diamond(b)} = \{a\},\ del(b)$$

Definition :

The part relationship $P_{B,A}$ is whole-to-part dependent (i.e., δ = whole-to-part) if

$$\forall\ b \in E(B)\ del(b) \rightarrow \forall\ a \in E(A) \text{ such that } b \lozenge a\ \wedge M_{\lozenge(a)} = \{b\},\ del(a).$$

If the value of δ is nil, then neither of the above deletion semantics is applicable. It will be noted that in both cases, the condition requiring that the independent deleted item (e.g., a in the case of part-to whole dependency) be the only existing referent implies a "multivalued" deletion semantics in that the deletion is not propagated until the set of referents on which a given object depends becomes empty[39]).

To express the dependency in our graphical schema representation, an arrowhead facing in the direction of the dependency (i.e., against the direction of the deletion propagation) is placed immediately behind the diamond head. See Figure 2.2.

2.2.4 Value Propagating Part Relationships

We now define two part relationships that support upward and downward value propagation[44,40]. Value propagation refers to the flow of a data value across the part relationship. As a modeling tool, it is useful for expressing certain functional dependencies between integral objects and their parts. As an example, a car may be modeled such that its age is equal to the age of its frame. In other words, the attribute age of class car would be defined to be identical to the attribute age of class frame, which is a meronymic class in relation to car. In such a case, instead of storing the value of age at both classes, the value should be stored at frame and propagated upward through the part relationship to car as needed. In this way, age need not be stored multiple times, and its value is guaranteed to be the same at both car and frame.

In the discussion below, we follow[45] in defining the properties of a class [namely, its attributes, relationships, and (reader) methods] as functions that map instances of the class into values of an associated type. For example, if the attribute name is defined for class person, then name is a function that maps instances of person into "nameType" (e.g., "string").

The requirement that ◊ have a converse that is a partial function insures that F_π is well defined. In a realization of this relationship, the public interface of the class A would be augmented with a message to retrieve the value of F_π for a given instance. This message would be identical to the message used to retrieve property π from an instance of B. As an illustration, consider the above example where age is propagated from frame to car. The interface of car would be augmented with a message "age" which for any instance of car would retrieve the value of the function F_{age} defined as

Definition :

$$F_{age(c)} = \{ \text{ age(r), \&if } \exists \text{ r} \in \text{E(frame) s.t. r} \lozenge \text{ c undefined, \& otherwise } \}$$

The upward propagating part relationship is represented graphically by placing the name of the property being propagated in parentheses alongside the generic symbol. An upward-pointing arrowhead is written in front of the parentheses to indicate the direction of the propagation (Figure 2.2).

The value propagation mechanism could be defined such that all the properties of the meronymic class are made available to the holonymic class. We have chosen to concentrate on a single property because the propagation of all properties is ordinarily not meaningful in the context of a part relationship. A holonym does not normally require

many of its part's properties. We can, of course, extend the definition to a set of properties.

The downward propagating part relationship is used in the case where a data value of the whole determines something about its parts. For example, in the real world, if a filing cabinet is composed of steel, then its drawers are probably composed of steel, too. In general, we could opt to model drawers such that they are always composed of the same material as their cabinets. We stress that within our part model, such an arrangement would not represent a default (see, e.g.,[13]), but rather a definitive modeling decision requiring all drawers to obtain their material make-up from their filing cabinets.

The definition of the downward propagating relationship is analogous to that of its upward propagating counterpart. Due to lack of space, we omit the actual specification. We do point out that the graphical symbol used is identical to that for the upward propagation except that the prepended arrowhead points downward (See Figure 2.2).

PART WHOLE RELATIONSHIP :

EXPANSIONS :

SHARED

MULTI-VALUED

EXCLUSIVE

MULTI-VALUED EXCLUSIVE

CLASS EXCLUSIVE

MULTI-VALUED CLASS EXCLUSIVE

ESSENTIAL

MULTI-VALUED ESSENTIAL

DEPENDENT (UPWARD)

MULTI-VALUED DEPENDENT (UPWARD)

DEPENDENT (DOWNWARD)

MULTI-VALUED DEPENDENT (DOWNWARD)

Figure 2.2 Expansion of Part Relationships

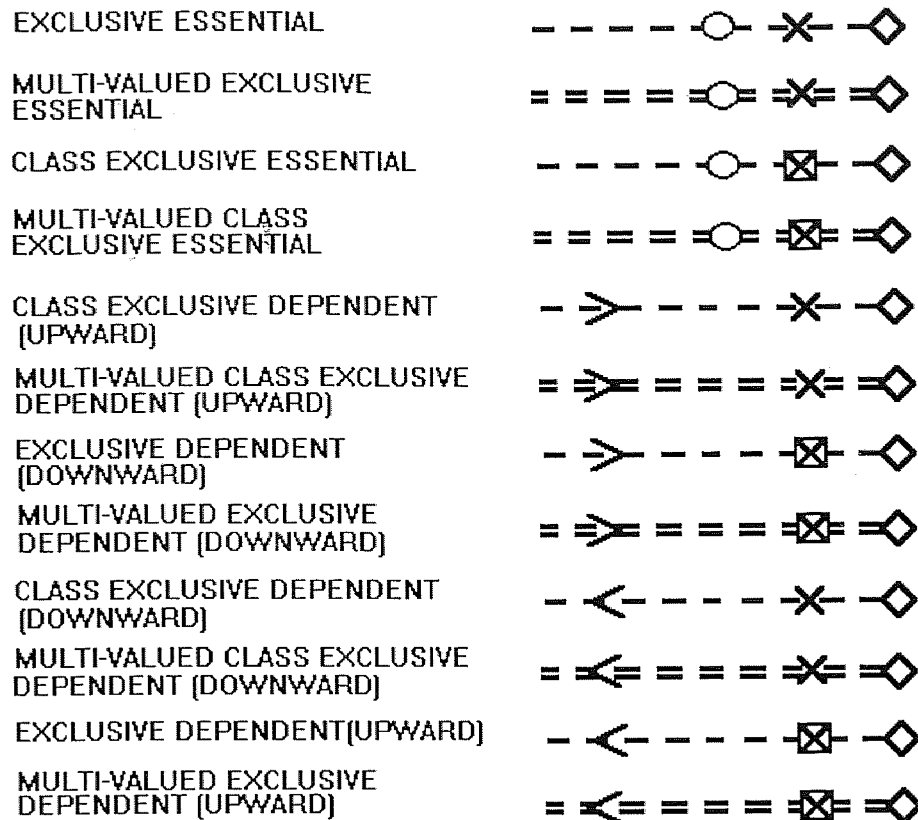


Figure 2.2 Expansion of Part Relationships (cont'd)

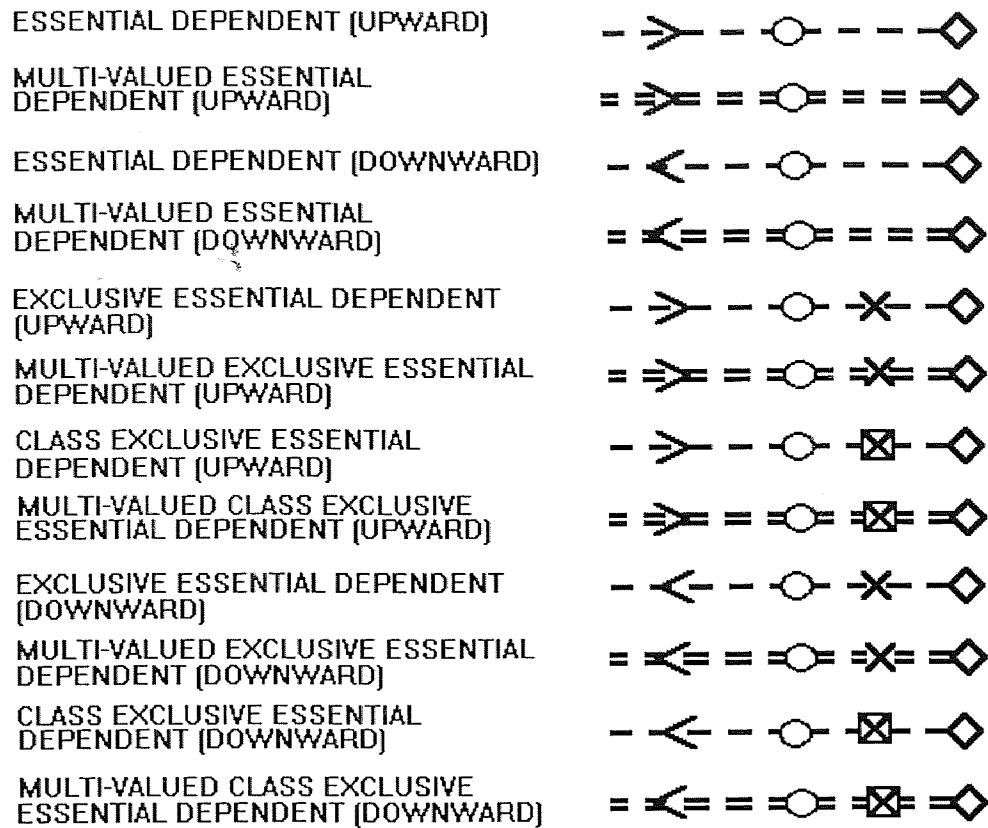


Figure 2.2 Expansion of Part Relationships (cont'd)

CHAPTER

3 OWNERSHIP RELATIONSHIP

Ownership is a very important relationship in the business world. It is endowed with rich semantics with respect to the owner and the property that is owned. As used in the corporate world, ownership can exhibit a hierarchical structure. For example, one company can own other companies.

Because of its complexity, modeling ownership in the context of a database system can be an extremely difficult task. In our model, we introduce an "ownership" relationship model that can be integrated into an Object Oriented Database (OODB) system. The use of this relationship greatly facilitates the problem of modeling real-world ownership and of enforcing its associated constraints.

3.1 Definition of Ownership

When we describe a state of "ownership", we must, in general, include the following three features:

- The owner,
- The property that is owned, and
- The characteristic of the relationships between the two.

According to Webster's Dictionary, ownership is defined as follows:

- The state or fact of being an owner.

- Proprietorship; Legal right of property; Legal or just claim or title (to something); in law, the right to use for one's own advantage some property.

The owner referred to above can, by law, be a natural person, a corporation, or an organization. The latter two are, in general, referred to as legal entities. Under the law, legal entities are vested with certain powers, some of which are also held by natural persons. Others, like the power to exist in perpetuity, are unique to legal entities. In our databases, we see that Jim as a natural person owns his business. The Chrysler Corporation as a legal entity owns Dodge.

Ownership of an item is often distributed among persons and legal entities. E.g., Jim and David together own a business, and a business bank account. Also, the Eagle Corporation is a joint venture of Chrysler and Mitsubishi. We describe such a situation as joint ownership. It is legitimate for a person and a company to jointly own a property. The ownership need not be divided into equal portions. Stock holdings partition the ownership of a public company into various percentages.

In law, property means rights which one has in anything subject to ownership, whether it is mobile or immobile, tangible or intangible, visible or invisible. Ownership is used synonymously with rights in property. Thus, a person is said to be the owner of a property if he has certain rights in it. The term ownership is often used to indicate that one has the "highest rights" in a property, but it may be used even when one does not have all the rights. Thus, we say that a person is an owner of the house even though he has rented it to a tenant who has exclusive rights to the use of the house during the term of the lease.

A property can be classified as real, intellectual, or personal. A real property refers to the rights that one has in land or things closely related to it. An intellectual property is the rights held on an idea (e.g., the design of an invention) or a creative work (such as a musical composition or a novel). For such property, the rights apply to a potentiality — no claim is made on any tangible item. Copyrights and patents are the ordinary forms of intellectual property. Personal property encompasses everything that is not a real or intellectual property. As an example, Jim's business resides in a building that is his real property. One characteristic of the ownership relationship itself centers on the existence of a legal document that verifies the owner's rights to a property. A copyright owner, e.g., is granted a legal certificate giving him exclusive rights to possess, make, publish, and sell copies of his intellectual productions, and to authorize others to do so. In contrast, the owner of a household item does not have a legal document to support his ownership, but he has the right to use it as he pleases. We call ownership of the former kind documented and ownership of the latter kind undocumented. So, Jim's patent is documented, while his ownership of a toaster oven is undocumented.

As a final distinction, some kinds of ownership are acquired by operation of law, and we call it a de jure ownership. While some others are not, and are called de facto ownership.

3.2 Ownership as an OODB Semantic Relationship

3.2.1 Transactions and Inheritance

As noted above, the most crucial aspects of ownership are the constraints that it imposes on its related transactions such as sale and lease. Certain transactions can be applied to

specific kinds of ownership, while others cannot. For example in the case of exclusive ownership, the owner can sell his belonging without restriction (and thus the transaction "sale" can be applied freely). While for joint ownership an owner can not sell the property without the consent of the other owners (so the use of "sale" must be controlled). When a person has accepted an offer to sell his house, he cannot accept another offer, even though he is still the owner, until that time when the first offer becomes invalid. We call the ownership of this kind action-limited. Similarly, when one has bought a stock option, the ownership of it may expire after a certain period of time if it is not exercised. In this case, we say that the ownership is time-limited. Likewise, when one has an ownership of some property like a car or a house, it cannot be sold without its proper documentation.

Aside from the transactions, the ownership relationship plays a vital role in more accurately modeling various application domains via its inheritance mechanism, which allows values of certain attributes to be propagated across it. Consider that to calculate Chrysler's profit for 1995, the profits of Dodge, Plymouth, and Jeep must be added together. Furthermore, the profits of Dodge must take into account the profit of Eagle. In such an example, a value propagation between properties and owners is required.

From the above we see that to properly support transactions and inheritance with respect to ownership, we need to explicitly model the different characteristics (which we call the dimensions) of the ownership relationship. The investigation has revealed six important dimensions.

3.3 Formal Definition of the Ownership Relationship

Let E(C) denote the extension of a class C, i.e., the set of all its instances. The ownership relationship between a property class B and an owner class A (denoted OB,A) is defined as the following septuple:

$$OB,A = \langle \Omega BA, \lambda, \beta, \alpha, \chi, \delta, \nu \rangle$$

where ΩBA is a relation from E(B) to E(A). The pair $(b,a) \in \Omega BA$ indicates that the instance b of class B is the property of (i.e., is owned by) the instance a of class A. We will ordinarily express this fact as $b\Omega BA$. The remaining elements of the septuple are the six characteristic dimensions, whose names are Legality, Documentation, Limitation, Exclusiveness, Dependency, and Value Propagation, respectively. For each, we list its domain in the following:

$\lambda \in$ de jure, de facto

$\beta \in$ registration-docum'ted, transfer-docum'ted, undocum'ted

$\alpha \in$ action-limited, time-limited, action&time-limited, unlimited

$\chi \in$ exclusive, free-joint, percentage-joint, global-percentage-joint

$\delta \in$ owner-to-property, nil

$\nu \in$ up, down, upTrans, downTrans, up&down, nil

The values of both δ and ν may be nil, indicating that the particular characteristic (dependency or value propagation) is inapplicable.

3.3.1 Exclusive Dimension

Ownership can be classified as exclusive or joint. In other words, a property may be owned by one owner or jointly owned by several owners. The formal definition for the exclusive ownership relationship follows :

Definition :

For the ownership relationship $O_{B,A}$, χ = *exclusive* implies that

$$\forall\ b \in E(B),\ |N_{\Omega BA}(b)| \leq 1.$$

To represent this graphically, we add an **X** to the dotted arrow (Figure 3.1).

Those ownership relationships that are not exclusive are referred to as joint, in which case a property may be either jointly owned freely, i.e., there is no explicit partition of the rights of the joint owners in the property. For example, a joint bank account is freely shared by a couple — we call this free joint. It can be jointly owned such that each owner takes a certain percentage of the ownership. For example, husband and wife each own 50% of their house — we call this percentage joint. We call the case where all owners have the same percentage equal joint [46,47]

Graphically, a plain dotted arrow indicates free joint (See Figure 3.1). Percentage joint and equal joint are denoted by labels of **P** and =, respectively (See Figure 3.1).

Definition :

For the relationship $O_{B,A}$, χ = *free-joint* implies that $\forall$ b $\in$ E(B), $O_{B,A}$ does not impose any constraints on $|N_{\Omega BA}(b)|$. Each instance *b* may have any number of owners.

Definition :

For the ownership relationship $O_{B,A}$, χ = *percentage joint* implies that $\forall$ b $\in$ E(B), each of its owners *a* has an associated number $p_{b,a}$ ($0 < p_{b,a} \leq 100$) indicating *a*'s

percentage of ownership of *b*. The percentages $p_{b,a}$ associated with all the owners of *b* must total 100%.

Above definition defines the percentage joint ownership relationship when the property class has only one associated owner class. At times, the ownership of an object may be distributed among owners from different classes. This case is defined as follows.

Definition :

The ownership relationships $O_{B,A1}$, $O_{B,A2}$, ⋯ , $O_{B,An}$ are *global percentage joint* if $\forall$ b $\in$ E(B), each of its owners (regardless of their classes) own percentages of *b* totaling 100%.

3.3.2 Value Propagation Dimension

There are times when a certain feature of a property is naturally assimilated as a feature of its owner, or vice versa. E.g., the address of a person may be modeled as the address of his house rather than as an intrinsic attribute of the person. The value of address, rather than being duplicated, should be stored solely with the house and propagated upward on demand. Address, in this sense, is a derived attribute of person.

Definition :

Let π_B:E(B) $\rightarrow$ π be an attribute of *B*. The ownership relationship $O_{B,A}$ is said to be *invariant upward propagating* if it defines a property π_B on the class *A* such that the value of π_B for an instance $a \in E(A)$ is identically the value of π_B for that $b \in E(B)$ which is owned by *a*.

For example, if the property address were propagated from the class house to the class person, then the ownership relationship would define the property address on class. Thus, the address of a person is identically that of the house that he or she owns. Invariant propagation in the other direction is defined analogously (see[48]).

Definition :

Let $\pi_B : E(B) \rightarrow \pi$ be an attribute of B. The ownership relationship $O_{B,A}$ is said to be *transformational upward propagating* if it defines a property π_B on the class A such that the value of π_B for an instance $a \in E(A)$ is derived by applying some transformation collectively to the values of π_B for all $b \in E(B)$ such that b is owned by a.

Here, instead of being identical to a value at a single property object, the value of the propagated attribute is derived through a transformation of values from many owned objects.

3.3.3 Additional Dimensions

The dependency dimension (see[48,49]) regulates the semantics of deletion of ownership class A or property class B. It defines when deletion of one should cause the deletion of the other. Ownership can be either documented, or undocumented. Documented ownership always has a supporting legal document, while undocumented ownership does not.

Some kinds of ownership are acquired "by operation of law," i.e., through a formal legal procedure. We call such ownership de jure. Others are not, and are called de facto. These are the values for the legality dimension. Ownership is often used to indicate the "highest rights," but it may be used when one does not have all the rights. In

other words, ownership may be limited in some aspects. For example, if the owner of a house has accepted an offer to sell that house to someone, then he cannot sell it to some other person, even though he is still the owner, unless the offer becomes invalid.

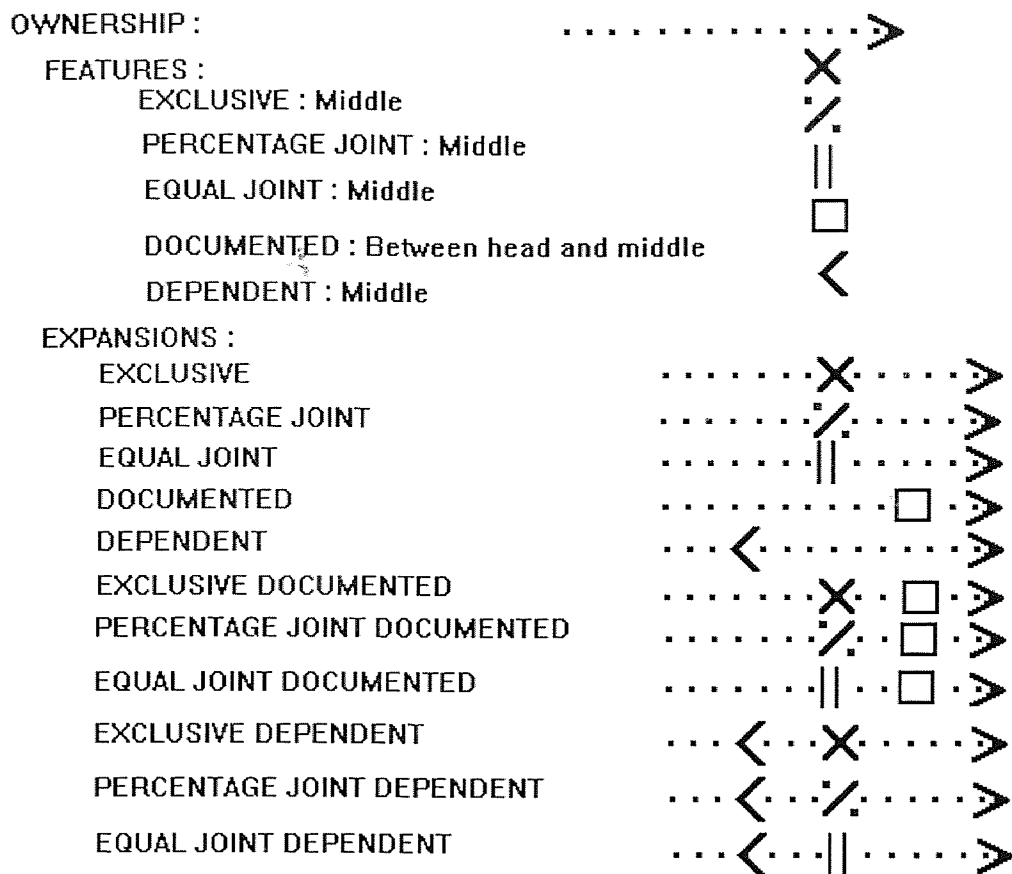


Figure 3.1 Expansion of the Ownership Relationships

CHAPTER

4 THE ARCHITECTURE OF OBJECTMAKER

4.1 Components

ObjectMaker provides the following functional components.

4.1.1 Diagramming Tool

Diagramming tool provides support for building many types of notation, performs basic syntax checking, derived diagram creation (e.g., subdiagram), and map semantics of diagrams to the repository.

4.1.2 Repository Management

The ObjectMaker Repository is relational in its schema definition and storage capabilities, but provides navigational access facilities in addition to the usual mechanisms of sets and cursors. The schema consists of a variety of record types, with two types of fields per record : text and link. The link fields provide the basic facility for representing complex concepts and navigating among them.

4.1.3 View Management

The View mechanism provides access to repository information through display windows that can be set in one of three modes : search, which allows specifying criteria for selecting records to display (in a QBE-like way); table, which shows all selected records,

one per row ; and form, which shows the fields of one record and allows records to be added, deleted, or modified.

4.2 Levels of Functionality

4.2.1 Kernel

This layer is provided by Mark V as a set of executable, and should be considered immutable. It provides the Extension Language interpreter, primitive predicates, the drawing engine and the repository and view management facilities. It interacts with Extension Language programs through primitives and callbacks.

4.2.2 Support Layer

This layer is provided as a set of Extension Language files (encrypted for the end user, plain text user for the TDK user). It provides higher-level support for defining and managing operations for various methods and notations; in many cases, it provides a declarative way to specify relations and transformations. The support layer is neutral with respect to repository schemata, diagramming notations, and methods.

4.2.3 Schema Layer

The Extension Language files in this layer provide a specific schema for object storage, schema-specific view definitions, and other schema- related information. It is possible for users to interact with the tool entirely at the schema layer, independent of particular methods. This layer is delivered with the Mark V standard schema, but may be tailored by the TDK developer.

4.2.4 Method Layer

This layer contains Extension Language files that provide support for methods and their associated notations, for creating and editing diagrams, generating repository information from them, and supporting method-specific views of the resulting records.

4.3 Directories and Files

The Extension Language's files that are part of ObjectMaker are stored in the context directory. Its useful for a developer to study them, both to see what facilities are available in the various layers and as a source of Extension Language predicates to learn from.

CHAPTER

5 THE OBJECTMAKER EXTENSION LANGUAGE

5.1 What is the Extension Language

The Extension Language is a definition and programming language that specifies the external behavior of ObjectMaker. It is used to define all layers of functionality above the Kernel. It can also be used by the TDK developer to personalize ObjectMaker into a special purpose tool, either using or replacing the layers supplied by Mark V.

The language is interpretive; the Kernel includes an interpreter for the language plus primitive constructs to interface with the internal functionality of ObjectMaker.

5.2 Why do we need the Extension Language

Mark V has developed the Extension Language in order to allow ObjectMaker's behavior to be defined and customized by Mark V and its customers. Extension Language allows users to customize ObjectMaker, and therefore adapt it to their work situations. Additionally, the Extension Language predicates provide the capability for inter-tool integration, allowing ObjectMaker to be operated by, and control the operation of, other programs using the windows DDE and OLE protocols (capabilities for message passing and coordinating applications' work on shared documents), Unix's RPC mechanism, or other platform-supported communication protocols. The Extension Language allows users to specify the "binding" of keyboard and pointer inputs to language-driven actions.

5.3 What can you do with the Extension Language

With Extension Language predicates, you can customize ObjectMaker's interface, as well as its behavior. For example, you can customize ObjectMaker's menu entries, and the actions that are performed when these entries are selected. If you want a certain menu item or a certain behavior when that menu item is selected you can modify the particular rule that controls that aspect of ObjectMaker. You can also customize how ObjectMaker retrieves information and displays it in diagrams, text, and code. If you want a certain processing routine performed when ObjectMaker accesses data from the underlying semantic repository, or if you want to implement a certain pre-and post-conditions to accessing data in forms and tables, or pre- and post- conditions to graphic editing, you can modify the particular rules that control that aspect of ObjectMaker.

You can specify how an object on your diagram relates to entities in the semantic repository. In ObjectMaker, each elementary diagram object is matched (by its shape, its pen and other style flags) to rules that determine its semantic use. So, if you want to modify the semantic behavior of an object on your diagram, you can modify the particular rule that controls that aspect of ObjectMaker.

Additionally, for a complex object consisting of multiple shapes, a predicate could check the objects in a given neighborhood.

You can specify what ObjectMaker should do when a view of part of the repository is requested. View Generation for tabular information is accomplished by the forms and table view facility, which is itself controlled by predicates. As a textual screen display is being prepared, these predicates prepare a search specification (which specifies a set of objects in the semantic repository to be viewed), a view specification (a definition

of the format and rules for the view's appearance and behavior), and default values to assign newly created records. All of these actions may be customized.

5.4 Nature of the Language

The Extension Language is a rewrite language. This means that programs in the language are texts containing references to stored definitions. In operation, a text is scanned (left to right) for these references; when one is encountered, it is replaced by its definition. The result of this operation is a completely scanned text. In the ObjectMaker Extension Language, the stored definitions are called *rules*, and consist of two parts: a *head* which is matched against references in the scanned text (called invocations), and a *body* or *tail*, which is the text to be substituted for the invocation.

The power of the language derives from several features of this process:

1. The invocations are not simple words to be substituted, but may contain parameters thus allowing one rule to match many different invocations. The parameters may be used in the body of the rule, thus allowing what gets substituted to vary, depending on the parameters in the invocation.
2. Invocations may be nested, and will be replaced "inside out", thus, the result of an invocation may become a parameter to another invocation.
3. The body of a rule may also contain invocations. When a body replaces an invocation, scanning normally resumes from the beginning of the replacement, so that these invocations will be seen and replaced in their turn.

4. Some invocations may refer to primitives, which are defined in the kernel rather than as rules. They may turn replacement text and also produce side effects, such as popping up a dialog box.

5. The kernel may initiate Extension Language processing under certain circumstances. Depending on the resulting test, certain actions may be taken.

5.5 Rules

5.5.1 Rule Head

Rule head consists of a pattern. For readability, and to avoid ambiguity in matching invocations, we use the convention that it should look similar to a typical function or subroutine call in procedural languages : a name(italics) (which should consist of alphanumeric characters plus dash, underscore or number sign(#), character), optionally followed by parameters in parentheses. Another reason for adhering to this syntax is that, in the future, we may restrict the allowable syntax to permit efficient compilation of the Extension Language. Again by convention, we refer to a set of rules with the same name as predicates(italics).

Parameters may contain the following two patterns matched characters : "*" to match any string, "?" to match any one character. Parameters may be named; the syntax for this is name = value. It is not necessary to explicitly represent parameter names in a head or, for that matter express the number of parameters. For example, the head foo(*) will match an invocation with the name foo and anything at all in the parentheses.

5.5.2 Rule Body

The body of a rule contains a mixture of plain text, embedded expressions, and parameter references. Expressions are strings delimited by angle brackets and may be arbitrarily nested. An expression may have the form of a language-defined expression, or may be an invocation (a reference to a rule). In the latter case, it should have the same form as a rule head, except that its parameters will be taken literally.

Parameter references request substitution of text from invocation parameters, and take the form <ref>, where ref is either an integer n, requesting substitution of the nth parameter, or a name, requesting substitution of the named parameter. If the invocation does not contain at least n parameters, or a parameter with a given name, a null string is substituted.

CHAPTER

6 IMPLEMENTING A METHODOLOGY WITH OBJECTMAKER

In this chapter, we discuss the components that are needed to be created to provide support for the methodology that we have discussed in this report: the diagramming notations, repository definitions and view specifications that allow users to create and maintain method-related data.

6.1 Menu definition

The methods and notations supported by ObjectMaker are stored in directories under context/methods. Each directory corresponds to a method; it contains a file with the extension .mnu for each notation. By convention, this file contains only menu specifications; syntax and semantic rules are stored in files with the same base name and the extension .rul. The last file needed when adding a new method is the file method.cfg. It should be created in the method directory to describe the method and its notations. This file consists of a single association list with method information and a sublist giving information for each menu. It must have no comments or other extraneous matters. Also, names and descriptive text must not contain any characters that might confuse the Extension Language scanner (i.e. commas, angle brackets, or unpaired parenthesis). Optional attributes may be omitted or may contain any information, subject to the above restrictions. Other named attributes may be added to the list for descriptive purposes, and will be ignored. For the method .cfg file of OOdini 2.0, see Appendix B.

The next file that needs to be created is the .mnu file. ObjectMaker menus consists of the following components: a menu bar structure common to all diagram types, menu items specific to individual diagram types, accelerator (shortcut) keys, and a palette menu.

Most diagram notations supported by ObjectMaker will have the same items on the top level menu bar: "File," "Edit," "View," "Insert," "Database," "Tools," Window," and "Help." However, most notations differ from each other in the definition of the supported icons, the products that can be generated (such as code-generation options), and possibly others (such as type-specific toggles or palette menus). Accordingly, MarkV Systems has provided an easy way to use the common menu definitions for the shared common menu functions, and optionally the ability to add submenus for the type-specific items. Each of the menus includes a corresponding submenu_of_menu invocation that can be used to add items for a particular notation.

The icons that represent a particular notation are mostly "localized" in the "Insert" pull-down menu.

In addition , menu files may define a palette menu or a shortcut accelerator key for some menu items, actions, etc. Theses key assignments appear in the declaration of the menu items and as supplementary accelerator definitions. The palette definitions appear as a separate definition in the menu file (See Appendix B).

To show how menus and their items are defined, we'll begin with a simple example. Every menu file will contain a pair of rules that define the "Insert" menu; the following is part of the actual .mnu file for OOdini 2.0 that defines a sub-menu entry for a class under the menu entry classes:

method_menus ::= item(&Insert,,<menu_of_icons>,);

menu_of_icons ::= menu(

item(Classes,, menu(

item(Class,, RECTANGLE(flags= solid,,,Class,Insert a class,Insert),);

The first rule defines an item on the menu bar(it's invoked by the support layer rule that defines the menu bar). If warranted, additional method-specific menus could be added here.

"Menu" is a key-word enclosing a list of entries. The top level menu is the menu bar; the second-level menus (sub-lists) are drop-down menu panels; lower-level menus (sub-lists) are walking menus. Menus can be nested to a reasonable depth (certainly deeper than good user interface principles would permit). In this example, "Classes" will appear on the menu with a right-pointing arrowhead. Selecting this item would reveal the one-item nested menu item of Class.

The following is the format of defining a menu item:

item(prompt-left, prompt-right, action, accelerator, status bar message #1, status bar message #2, status bar message #3)

defines a menu item, in which prompt-left and prompt-right are the text strings that appear on the menu(left- and right-justified, respectively), action defines what is to be done when the item is selected, and accelerator defines a keystroke combination to be used to achieve the same effect as choosing the item from the menu. When an accelerator is defined, a representation for it will appear on the right side of the menu entry next to the prompt-right, if the later is present. The status bar messages will appear on the status

bar at the bottom of the screen to inform the user what type of item is selected, what type of action is being performed on that item and what type of action is being performed on the schema.

6.2 Diagram Syntax Checking

As described earlier in this report, there are several occasions when the support layer calls appropriate predicates to check the user's drawing activities. The ".rul" file corresponding to the notation's ".mnu" file contains the rules for checking the legality of diagramming operations for the notation. We discuss here the predicates commonly provided for such checks.

The basic predicate, icon_type, is used in several contexts. It returns, for a given icon, a "syntactic type" which is used in legality checks as well as in mapping icons to the repository. In its simplest form, it's a context free mapping from an icon's shape and style to an expression that is defined. A more complex form takes a handle to the icon, which may be used to navigate around its neighborhood in the diagram when the type can't be determined from the shape and style alone. For example, a box may be of a different type depending on whether it's nested in another or is at the top level.

For example, here are a couple of definitions corresponding to the OOdini methodology from the OOdini 2.0 .rul file:

icon_type(rectangle,solid)::=regular_class;

icon_type(rectangle,thick_skt_*_0%)::=set_class;

icon_type(arc,solid,arrow_none,arrow_none,arrow)::=regular_relationship;

icon_type(arc,solid,arrow_none,arrow_none,arrow_o)::=essential_relationship;

icon_type(arc,solid,arrow_none,arrow_none,arrow_double)::=dependent_relatio nship;

icon_type(arc,solid_double,arrow_none,arrow_none,arrow)::=multi-value_relationship;

icon_type(arc,solid_double,arrow_none,arrow_none,arrow_o)::=mv_essential_r elationship;

icon_type(arc,solid_double,arrow_none,arrow_none,arrow_double)::=mv_depen dent_relationship;

The "style" for an arc is actually four parameters: pen style (solid in the example) followed by the "decorations" at the tail, middle, and head of the arc.

The predicate arc_check is called twice during the drawing of an arc from one node to another. Once when starting the arc, to see if it is legal to begin an arc at the "from" node, and once at the finish of the operation, to see if an arc can terminate at the "to" node. An example :

arc_check(regular_class, regular)::= regular_class;*

arc_check(regular_class, essential_relationship)::=regular_class|set_class;

The parameters and tail of an arc_check are icon_types (or patterns matching them); the first example rule may be read "an arc of regular type (i.e. one whose icon_type begins with regular) may begin and end at a regular class". The tail of the

second rule is an example of an "or" pattern; the meaning here that an essential relationship may end either at a regular class or a set class.

The node_parent predicate is called when a node is created or moved inside another (i.e. the set class in the OOdini methodology). It has the form:

node_parent(child-type, parent-type)::nesting;

where nesting may be a socket, nested_or_socket, or a nested. These values tell whether the child icon can be nested (float freely within the parent), socketed (be restricted to the border of the parent)or either.

CHAPTER

7 CURRENT IMPLEMENTATION OF OODINI 2.0

7.1 Actual Implementation

As explained earlier, there are 3 main components needed for implementation of OOdini 2.0. The 3 components ; *the method.cfg, menu and rule files* are referenced in APPENDIX B.

The specific limitations that were discovered during the implementation are discussed in the next section.

7.2 Limitations of OOdini 2.0

There were many limitations that were encountered with the TDK used during the development of OOdini 2.0. The primary limitation was the non-availability of desired icons and adornments, as specified by the OOdini graphical representation. This led to alternative representations, and in some cases the icons were not supported at all.

Another limitation was the non-availability of the double framed icon. Because of this a set class could not be represented. A *set class* is represented by a double framed rectangle as shown in Figure 1.3.

The TDK provided very little support when it came to adornments. Although the TDK allowed every relationship to have at most 3 adornments : one at the head, one in the middle, and one at the tail, the adornments could only be one from the set of pre-defined(in the TDK adornment library) adornments. One could not design his/her own

adornment. This makes it impossible to represent relationships such as the *percentage joint relationship* that needs to have a percentage sign as an adornment. In certain cases, such as that of ;the documented relationship, one needed a rectangle as an adornment on the relationship. Although the ObjectMaker TDK supports both the rectangle and a regular relationship, it still can't be used. This is because ObjectMaker does not provide any way in which we can group both these concepts(the rectangle and a regular relationship together.

All icons that need to represent *part whole relationships* have a diamond shaped head. The TDK only supports regular arrow heads (those with '>' at the head). This made impossible to represent any of the *part whole relationships*.

Besides, in cases such as the *tuple class* see Figure 1.3, we require an entirely unsupported icon. The *tuple class* is represented by a rectangle with a small triangle attached to its bottom width. Since such icons are not supported the *tuple class* was not implemented in OOdini 2.0.

Many of OOdini 2.0's enhancements seem difficult because of the inability of the TDK to provide a way by which a developer can design icons in any desired manner. An example of such a requirement could be the regular class, which needs to be designed as a set of drawers to hold individual attributes. This is required to model path methods that end in an attribute. This feature was impossible to implement in OOdini 2.0 because of a lack of TDK support. Besides, even if such a regular class was diagrammatically possible, ObjectMaker did not have rules strong enough in their specifications to allow such a feature. By not having such rules to control the feature, the tool cannot validate

user design(i.e. it might allow other relationships to be associated with an attribute instead of the class).

CONCLUSION

In this report, we have presented a graphical schema representation language for OODBs. This language captures a full range of OODB constructs including classes, attributes, methods, user-defined, and semantic relationships such as specialization relationships, constraint relationships, part-whole relationships, and ownership relationships. The language has been employed successfully in a number of large applications and has proven to be expressive and intuitive. In this regard, further work such as a full-scale, subjective evaluation study is needed.

We have also introduced OOdini, a constraint-based graphical editor designed specifically for the creation and manipulation of our schema representation. A graphical schema representation can greatly facilitate the design of OODBs and can serve as a means for orienting users of such systems. The database community has recognized the need for a standard for declarative query languages7. We believe that there is a similar need for OODB graphical schema languages. Perhaps ours can serve as a step toward this end. As a final example, we refer the reader to Figure 1.1, which shows an excerpt from the schema of a university database we have built.

APPENDIX A USER MANUAL FOR OODINI 2.0

Listed below are the menu commands and an appropriate definition of the command. The File entry drops down a menu giving the user access to a number of disk storage and retrieval commands. OOdini 2.0 also gives the user the option of selecting output to a printer or setting up the page for printout. The menu also gives the user a choice for exiting the OOdini program entirely.

A.1 File Menu

New

Create a new, untitled, diagram. You will be prompted to save any changes to the current diagram. Then you will have the option of specifying a new method and/or notation.

Open

Open an existing diagram. You will be prompted to save any changes to the current diagram. If the current diagram is untitled, the Save As dialog is displayed to allow you to name the diagram before saving. The Open dialog initially points to the current Project directory, if one has been opened. The method/notation with which the diagram was saved will be automatically reloaded with the diagram.

Close

Close the current diagram. You will be prompted to save any changes. If the diagram is untitled, the Save As dialog is displayed to allow you to name the diagram before saving. A new, untitled, diagram is automatically opened in the current method.

Save

Save the current diagram using the current file and path name. If the diagram is untitled, the Save As dialog is displayed to allow you to name the diagram before saving.

Note: Save operations brand the diagram with the method/notation that is current at the time of the Save operation, so that the next time the diagram is opened, the method/notation is automatically loaded for you.

Save As

Save the current diagram with a new file and/or path name.

Print

Print the current diagram.

Print Full Page

Print the current diagram using default page settings.

Print Setup

Specify standard printer setup options as found in your system environment.

Diagram History

Open a diagram that was previously opened during the current ObjectMaker session. View the selected diagram in the current window or in a newly spawned session. If you open a diagram in the current window, you will be prompted to save any changes.

Exit

Exit from ObjectMaker. You will be prompted to save any changes. If the current diagram is untitled, the Save As dialog is displayed to allow you to name the diagram before saving.

A.2 Edit Menu

The Edit menu provides the user with, among other things, a search feature for tracking down a desired class. This function positions the system's current working window around a given class. It gives the user yet another method of repositioning the working window. This feature should aid both browsers and designers in their navigation through the schema.

Undo

Undo the last diagram edit action. Shortcut: Click the right mouse button. Does not undo semantic database operations.

Select All

Select all of the graphic elements in the diagram.

Delete

Delete one or more boxes or arrows. If action is selected first, click on box or arrow to be deleted. If the box or arrow is already selected, selecting the Delete action will complete the deletion process. (Shortcut - use the Delete key on the keyboard, then click on item to be deleted.) Deleting Mapped Icons Deleting a diagram shape that has been mapped to a semantic database causes ObjectMaker to ask if the related database record should also be deleted.

Clicking on No will cause the semantic database record to remain even though the diagram box is deleted. This is a logical action if the deleted object appears on other diagrams mapped to this semantic database. Clicking on Yes will cause both the semantic database record and the box to be deleted. If there are any associated shapes

(arrows, children, etc.) you will be asked if the semantic database record should be deleted for each of these associated records.

Clicking on Cancel will cancel the entire operation. Choosing Undo after the completion of the delete operation will restore the diagram but will not restore the semantic database. This feature allows you to edit and remap the boxes without having to redraw them.

Duplicate

Duplicate the selected graphics, including any nested boxes and labels. Select the box(es) to be duplicated, then choose Duplicate from the Edit menu. Place the cursor over the selected box (or one of the boxes within the group) and hold the left mouse button down while dragging the duplicate to its new location. During dragging operations, boxes are depicted as dotted or dashed rectangles. Release the mouse button to finish.

Note: Duplicating a group of boxes doesn't "group" or attach them to one another.

Move

Move or resize the selected box. ObjectMaker will default to the Move command at the completion of most edit actions. Moving a box will also move all nested boxes and attached arrows and labels. Place the cursor over shape to be moved, hold left mouse button down and drag to new location. To resize a box, place one of the edge or corner cursors at the edge or corner of the box you wish to resize, hold left mouse button down and drag to the proper location in order to obtain the new size.

Label Menu

Resize Box to Label: Resize the selected box to fit its label.

Resize Box: Resize a box. This command allows manual resizing of a box from anywhere within that box. Select the command, place the cursor anywhere in the box, press and hold the left mouse button and drag until the box is the desired size. Alternate: Simply position the cursor at an edge or corner of the box, press the left mouse button and drag until the box is the desired size.

Resize Nested: Resize a box that contains nested boxes. This command is similar to Resize Box except that when the parent is resized, any nested or socketed boxes are resized at the same proportion as the parent.

Make Construct

Create a reusable graphic construct.

Insert Construct

Insert an existing graphic construct into the current diagram.

Move Attached

Move boxes that have been attached to each other. The graphic will remain attached after the move, even if moved outside of its parent.

A.3 View Menu

Through the View menu, OOdini allows the user to Zoom in or Zoom out. OOdini can thus serve as a schema browser or OODB orientation device.

Zoom In

Zooms into (increase zoom level of) drawing. Choose Zoom In, then click at the point in the diagram you wish to be the center of the zoomed image. ObjectMaker supports 5 levels of zoom.

Zoom Out

Zoom out (decrease zoom level) of drawing. Choose Zoom Out, then click at the point in the diagram you wish to be the center of the zoomed image. ObjectMaker supports 5 levels of zoom.

Refresh Window

Refresh the window display. This command is used to clean up the drawing area after editing operations that have left stray artifacts.

Center Diagram

Center the diagram in the window without changing the current zoom level.

Zoom to Fit

Zoom in or out of the current diagram so that as much of it as possible will fit within the window. As we have discussed earlier that there are five zoom levels available to the user, in the ObjectMaker. Note that very large diagrams may not fit even at zoom level 1.

Center on Cursor

Center the window on a selected point in the diagram. Choose Center on Cursor, move the cursor to the spot on the diagram you wish to be in the center of the screen, and click the left mouse button. Provides fast, direct method of panning large diagrams or diagrams being viewed at high zoom levels. Alternately: place the cursor at the desired spot and press F10.

Find Graphic

Center the window on the first graphic whose label matches the specified text string. You may use wild card characters * and ?.

Show Grid

Toggle the visibility of the alignment grid. Grid matches value set using Alignment Grid command except when grid would be too dense. Only box shapes, anchors, and bends are snapped to grid, and they are centered on the nearest grid point. Labels, arrows, notes and other diagram constructs are not snapped to grid.

Show Anchors

Toggle the visibility of anchors, internal direction indicators on arrows, and spline outlines.

A.4 Database Menu

New

Create a new semantic database and open it. Prompts for a new database name (8 characters maximum). The dialog will default to the current Project directory if one has been opened. Each database consists of five files. For example, assuming the name of the database is named "mydata", the following files will be created:

mydata.tbl (the database file itself)

mydata.nxd (the database index)

mydata.lok (the database lock file)

mydata.nd0 (a copy of the index file as of the last database Save action)

mydata.tb0 (a copy of the database contents as of the last database Save action).

The ND0 and TB0 files are maintained so you can revert the database to the previously saved state.

Open

Open a semantic database file. The path will default to the current Project directory, if a Project has been opened.

Save

Save changes made to the current semantic database.

Save As

Save the current semantic database with another name.

Edit Record for Graphic

Edit the record in the current semantic database for the selected graphic.

Map Entire Diagram

Map an entire diagram into the current semantic database. Creates optional log file that is stored in the same directory as the diagram. The log file details, which boxes and arrows were mapped and any anomalies detected.

Map Selected Graphic

Map the selected graphic into the current semantic database. Also may create a log file of mapping activity results. Select the Map Selected Graphic command from the Database menu and then click on the shape you wish to map.

A.5 Tools Menu

Note that some items may not be available with all methods or notations.

Trace Feature

Trace the execution of Extension Language commands. The Message Control dialogue box is used to turn the trace feature on and off, set the trace filters and set file

names and paths. This command should only be used in consultation with Mark V technical support personnel.

A.6 Help Menu

Contents

Provides access to on-line help. A standard Windows Help window is displayed with the top-level index for ObjectMaker on-line help.

About ObjectMaker

Displays a window which provides various information about the current status of ObjectMaker including:

License information

Version, Code: (shows the version and build date of the ObjectMaker executable)

The number of Nodes (boxes, bends and anchors) and Arcs (arrows) contained in the current diagram. The maximum number of Nodes and Arcs allowed on a single diagram is controlled by the Nodes, Arcs and Notes values in the Setup Preferences dialog (Tools/Installation menu).

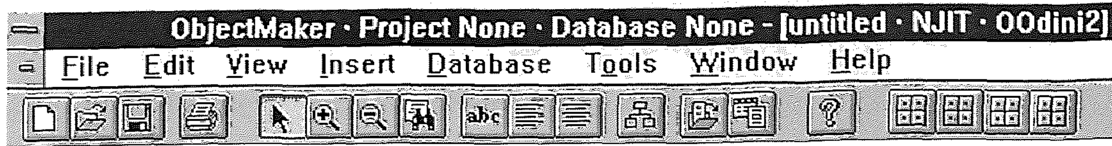


Figure A.1 OOdini 2.0 Main Menu

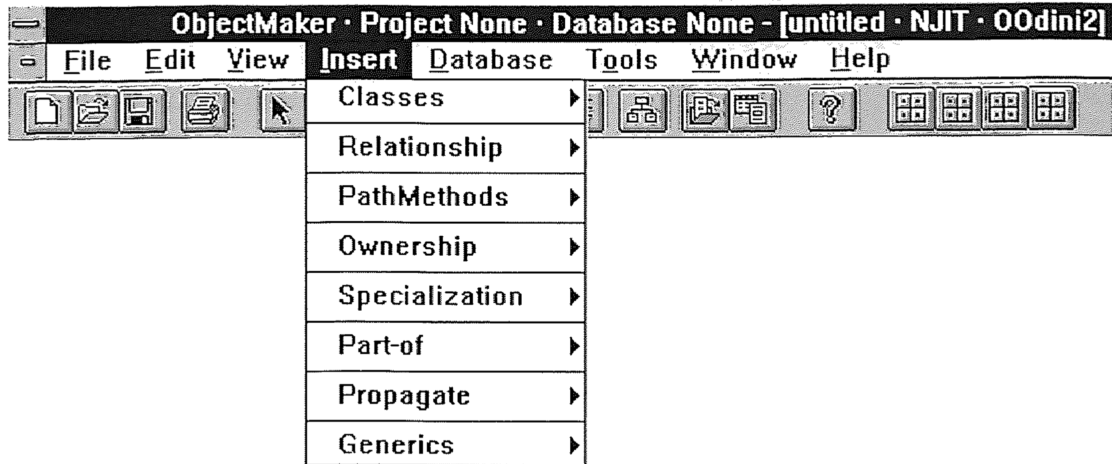


Figure A.2 Choosing various symbols using menu bar

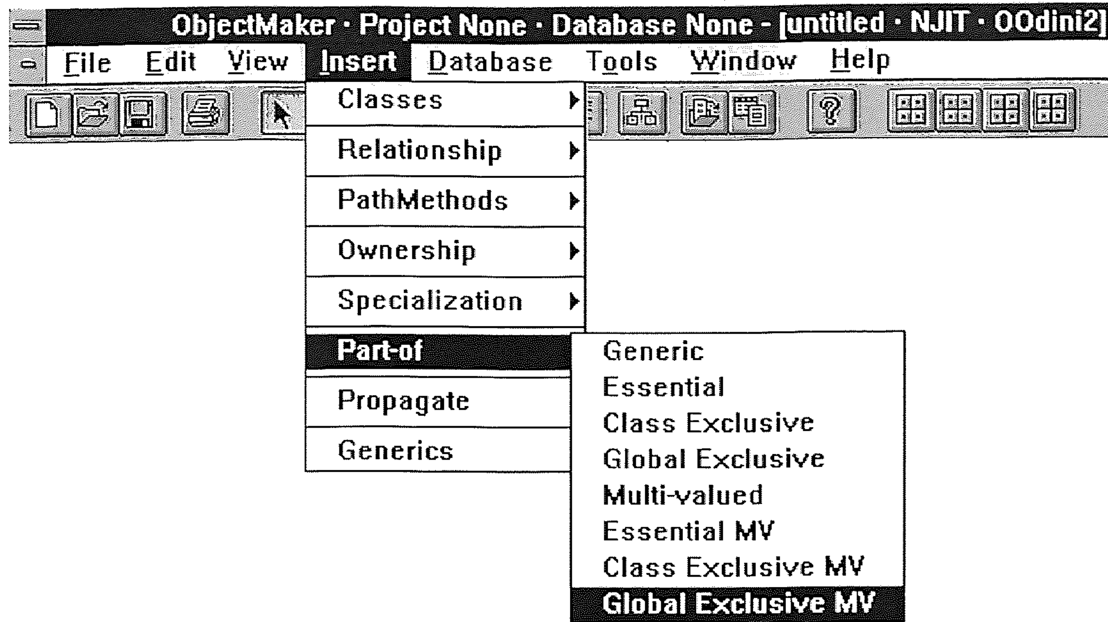
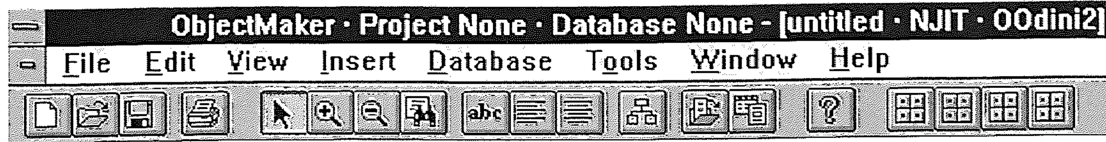


Figure A.3 Accessing a sub-menu



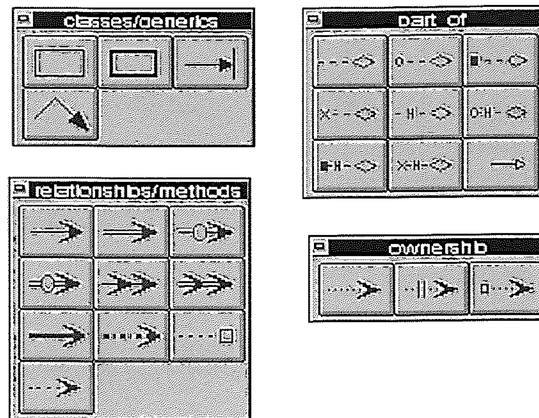


Figure A.4 ObjectMaker Toolbars

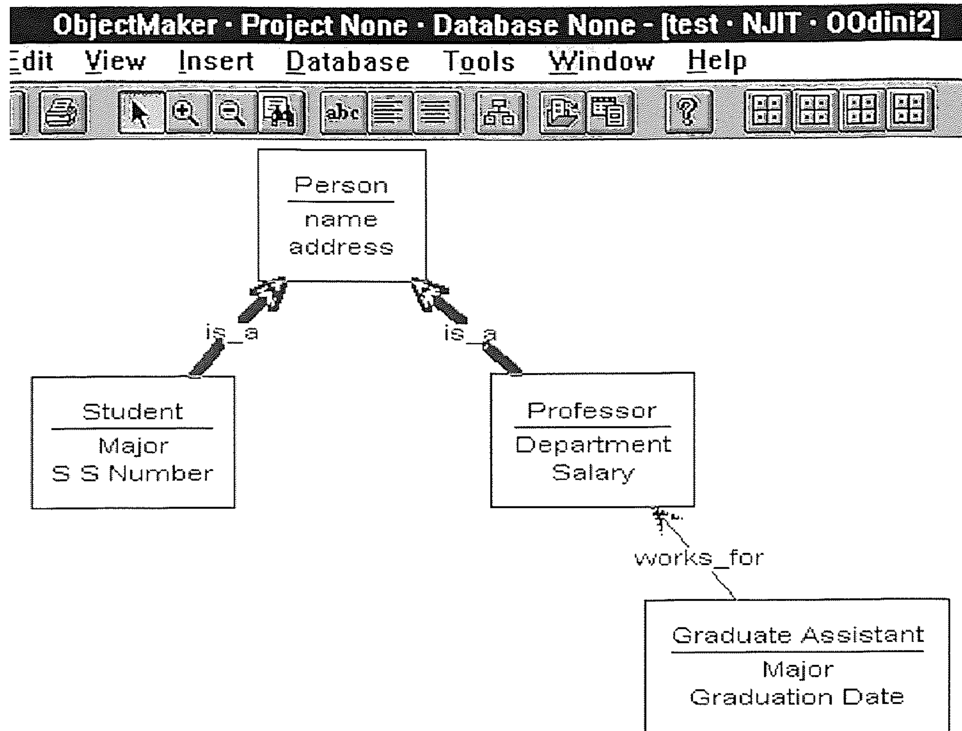


Figure A.5 ObjectMaker Class - Person

APPENDIX B SOURCE FILES FOR OODINI 2.0

B.1 method.cfg file

```
(name=OOdini2,
 desc=,
 date=,
 author=,
 name=OOdini2,
menus=(
(name=NJIT,
desc=,
date=,
author=,
file=ood2.mnu),
),
)
```

B.2 ood2.mnu file

```
ood2_menu_version ::= 1.1.alpha;
!bitmap mbomshp   ::=  ood2shp.bmp;
!bitmap mbomagg ::= ood2shp2.bmp;
!bitmap mbomgen ::=  ood2shp3.bmp;
!bitmap mbombina ::= ood2shp4.bmp;
```

```
!bitmap mbgeneric  ::= generics.bmp;


- use compound label edit

menu_of_label ::= menu(

item(&New,,            extl(comp_label_edit), immed act L),

        item(&Edit,,           extl(comp_label_edit), immed act L),

        item(Re&center,,       LABEL_CENTER,),

        item(&Grab,,           LABEL_POSITION,),

        item(&Flush Left,,     LABEL_JUSTIFICATION,),

        accl(,,                CLOSE_EDIT,    ESCAPE),

        submenu(label)

        );


dispatch_edit(ec) ::=

  putup_label("Class Definition",(Name,Attributes,Operations));

dispatch_edit(ecd) ::=

  putup_label("Set Class Definition",(Name,Attributes,Operations));

dispatch_edit(ecra0) ::=

  putup_label("Regular", (Name)) ;

dispatch_edit(ecra1) ::=

  putup_label("Multi-value", (Name)) ;

 dispatch_edit(ecra2) ::=

  putup_label("Essential", (Name)) ;
```

```
dispatch_edit(ecra3) ::=
  putup_label("MV Essential", (Name)) ;
dispatch_edit(ecra4) ::=
  putup_label("Dependent", (Name)) ;
dispatch_edit(ecra5) ::=
  putup_label("MV Dependent", (Name)) ;
dispatch_edit(scra0) ::=
  putup_label("Subclass", (Name)) ;
dispatch_edit(scra1) ::=
  putup_label("Role-of", (Name)) ;
dispatch_edit(ecrpp) ::=
  putup_label("Propagate", (Name)) ;
dispatch_edit(pcra0) ::=
  putup_label("Generic", (Name)) ;
dispatch_edit(pcra1) ::=
  putup_label("Essential", (Name)) ;
dispatch_edit(pcra2) ::=
  putup_label("Class Exclusive", (Name)) ;
dispatch_edit(pcra3) ::=
  putup_label("Global Exclusive", (Name)) ;
dispatch_edit(pcra4) ::=
  putup_label("Multi-value", (Name)) ;
dispatch_edit(pcra5) ::=
```

```
  putup_label("Essential MV", (Name)) ;
dispatch_edit(pcra6) ::=
  putup_label("Class Exclusive MV", (Name)) ;
dispatch_edit(pcra7) ::=
  putup_label("Global Exclusive MV", (Name)) ;
dispatch_edit(op0) ::=
  putup_label("Regular", (Name)) ;
dispatch_edit(op1) ::=
  putup_label("Disjoint", (Name)) ;
dispatch_edit(op2) ::=
  putup_label("Documented", (Name)) ;


dispatch_edit(pm0) ::=
  putup_label("Path Method", (Name)) ;
dispatch_edit(pm1) ::=
  putup_label("Attribute Path Method", (Name)) ;


- for snip code generation
menu_of_codegen ::=menu(subomaux);


method_menus ::= item(&Insert,,menu_of_icons,);
method_has_toolbar ::= yes;
method_toolbar_extension ::=
```

```
separator,
separator,
drag_anywhere,

bmpitem(stdimage'25,classes/generics,SHOW_CONTROL(palette=classes/generics),,Classes/Generics,"Classes/Generics palette","menu: tools"),drag_anywhere,

bmpitem(stdimage'26,relationships/methods,SHOW_CONTROL(palette=relationships/methods),,Relationships/Methods,"Relationships palette"),drag_anywhere,

bmpitem(stdimage'26,ownership,SHOW_CONTROL(palette=ownership),,Ownership Relationships,"Ownership Relationships palette"),drag_anywhere,

bmpitem(stdimage'26,part_of,SHOW_CONTROL(palette=part_of),,Part_of Relationships,"Part_of Relationships palette"),;

method_palettes ::=
  disable,
  classes/generics= palette(
        title= "classes/generics",
        columns= 3,
        width= 30,height= 20,
        bmpitem(mbomshp'1,,use(om1)),
```

```
        bmpitem(mbomshp'2,,use(om2)),
        bmpitem(mbgeneric'1,,use(gen1)),
        bmpitem(mbgeneric'2,,use(gen2)),


            ),
  disable,
  relationships/methods= palette(
        title= "relationships/methods",
        columns= 3,
        width= 30,height= 20,
bmpitem(mbombina'1,,use(omb1)),
bmpitem(mbombina'2,,use(omb2)),
bmpitem(mbombina'3,,use(omb3)),
bmpitem(mbombina'4,,use(omb4)),
bmpitem(mbombina'5,,use(omb5)),
bmpitem(mbombina'6,,use(omb6)),
bmpitem(mbombina'7,,use(omb7)),
bmpitem(mbombina'8,,use(omb8)),
bmpitem(mbombina'15,,use(om4)),
bmpitem(mbombina'16,,use(om3)),


        ),
  disable,
```

```
ownership= palette(
        title= "ownership",
        columns= 3,
        width= 30,height= 20,
        bmpitem(mbomgen'1,,use(om5)),
        bmpitem(mbomgen'2,,use(om6)),
        bmpitem(mbomgen'3,,use(om7)),
        ),
disable,
part_of= palette(
        title= "part_of",
        columns= 3,
        width= 30,height= 20,
bmpitem(mbomagg'1,,use(omb9)),
bmpitem(mbomagg'2,,use(omb10)),
bmpitem(mbomagg'3,,use(omb11)),
bmpitem(mbomagg'4,,use(omb12)),
bmpitem(mbomagg'5,,use(omb13)),
bmpitem(mbomagg'6,,use(omb14)),
bmpitem(mbomagg'7,,use(omb15)),
bmpitem(mbomagg'8,,use(omb16)),
bmpitem(mbombina'18,,use(omb18)),
        )
```

```
;

menu_of_icons ::= menu(

item(Classes,, menu(
        om1= item(Class,,    RECTANGLE(flags= solid,code= prop concept
ec\73),,Class,Insert a class,Insert),

        om2= item(Set Class,,    RECTANGLE(flags= (thick,skt_outside),code=
prop concept ecd\73),,Set Class, Insert a Set Class,Insert)
    ),),

        separator,
item(Relationship,,menu(
        omb1= item(Regular, ,ARC(head= ARROW,tail= ARROW_NONE,code=
prop concept ecra0\73),,Regular, Insert a Regular Relationship,Drawing arrows),

        omb2= item(Multi-valued,
,ARC(flags= SOLID_DOUBLE,head= ARROW,tail= ARROW_NONE,code=
prop concept ecra1\73),,Multi-Value, Insert a Multi-Value Relationship,Drawing arrows),
```

omb3= item(Essential, ,ARC(head= ARROW_O,tail= ARROW_NONE,code= prop concept ecra2\73),,Essential, Insert a Essential Relationship,Drawing arrows),

omb4= item(MV Essential, ,ARC(flags= SOLID_DOUBLE,head= ARROW_O,tail= ARROW_NONE,code= prop concept ecra3\73),,MV Essential, Insert a MV Essential Relationship,Drawing arrows),

omb5= item(Dependent, ,ARC(head= ARROW_DOUBLE,tail= ARROW_NONE,code= prop concept ecra4\73),,Dependent, Insert a Dependent Relationship,Drawing arrows),

omb6= item(MV Dependent, ,ARC(flags= SOLID_DOUBLE,head= ARROW_DOUBLE,tail= ARROW_NONE,code= prop concept ecra5\73),,MV Dependent, Insert a MV Dependent Relationship,Drawing arrows)

),),

separator,

item(PathMethods,,menu(

om3= item(Path Method, , ARC(flags= DASH,head= ARROW,tail= ARROW_NONE,code=prop concept pm0\73),,Path Method, Insert a Path Method, Drawing arrows),

om4= item(Attribute Path Method, , ARC(flags= DASH,head= ARROW_SQUARE,tail= ARROW_NONE,code=prop concept pm1\73),,Attribute Path Method, Insert a Path Method, Drawing arrows)

),),

separator,

item(Ownership,,menu(

om5= item(Regular, , ARC(flags= DOT,head= ARROW,tail= ARROW_NONE,code=prop concept op0\73),,Regular Ownership, Insert an Ownership, Drawing arrows),

om6= item(Equal Joint, , ARC(flags= DOT,head= ARROW,middle= CROSS_DOUBLE,tail= ARROW_NONE,code=prop concept op1\73),,Equal Joint Ownership, Insert an Equal Joint Ownership, Drawing arrows),

om7= item(Documented, , ARC(flags= DOT,head= ARROW,tail= ARROW_SQUARE,code=prop concept op2\73),,Documented Ownership, Insert a Documented Ownership, Drawing arrows)

),),

separator,

item(Specialization,,menu(

omb7= item(Subclass, ,ARC(flags= THICK,head= ARROW,tail= ARROW_NONE,code= prop concept scra0\73),,Subclass, Insert a Subclass Realtionship,Drawing arrows),

omb8= item(Role-of, ,ARC(flags= (DASH_DOT,THICK),head= ARROW,tail= ARROW_NONE,code= prop concept scra1\73),, Role-of, Insert a Role-of Relationship,Drawing arrows)

),),

separator,

item(Part-of, ,menu(

omb9= item(Generic,,ARC(flags= DASH,head= ARROW_DIAMOND,tail= ARROW_NONE,code= prop concept pcra0\73),,Generic,Insert a Generic Relation,Drawing arrows),

omb10= item(Essential,,ARC(flags= DASH,head= ARROW_DIAMOND,tail= O_EMPTY,code= prop concept pcra1\73),, Essential, Insert an Essential Relation, Drawing arrows),

omb11= item(Class Exclusive,,ARC(flags= DASH,head= ARROW_DIAMOND,tail= ARROW_SQUARE_FILL,code= prop concept pcra2\73),, Class Exclusive, Insert a Class Exclusive Relation, Drawing arrows),

omb12= item(Global Exclusive,,ARC(flags= DASH,head= ARROW_DIAMOND,tail= ARROW_FULL_X,code= prop concept pcra3\73),, Global Exclusive, insert a Global Exclusive Relation, Drawing arrows),

omb13= item(Multi-valued,,ARC(flags= DASH,head= ARROW_DIAMOND,Middle= CROSS_DOUBLE,tail= ARROW_NONE,code= prop concept pcra4\73),, Multi-Valued, Insert a MV Relation, Drawing arrows),

omb14= item(Essential MV,,ARC(flags= DASH,head= ARROW_DIAMOND,middle= CROSS_DOUBLE,tail= O_EMPTY,code= prop concept pcra5\73),, Essential MV, Insert an Essential MV Relation, Drawing arrows),

omb15= item(Class Exclusive MV,,ARC(flags= DASH,head= ARROW_DIAMOND,middle= CROSS_DOUBLE,tail= ARROW_SQUARE_FILL,code= prop concept pcra6\73),, Class Exclusive MV, Insert an Class Exclusive MV Relation, Drawing arrows),

omb16= item(Global Exclusive MV,,ARC(flags= DASH,head= ARROW_DIAMOND,middle= CROSS_DOUBLE,tail= ARROW_FULL_X,code= prop

concept pcra7\73),, Global Exclusive MV, Insert an Global Exclusive MV Relation, Drawing arrows)

```
        ),)
                separator,
        item(Propagate, ,menu(

                omb18= item(Propagate, ,MSG_SIMPLE(flags= (df_in,solid),code= prop concept ecrpp\73),,Propagate operation,Must be parented by an association instance,Insert)
            ),),
            separator,

        item(Generics,, menu(

                gen1= item(Anchor,, ANCHOR(flags= solid),,anchor,Place an anchor for all types of arcs,anchoring arrows),
                gen2= item(Bend,, BEND,,Bend,Insert a bend to any type of arc,inserting bends into arrows)
            ),),
        );
```

```
submenu_of_toggles ::=
    item(Rotate,,          ROTATE,),
        item(In/Out Mode,,     SOCKET_GENDER,  immed act F),

;

submenu_of_help ::=
  separator,

  item(Help for &Rumbaugh,, HELP(file=
rumbaugh.hlp,key=contents),),
  ;

submenu_of_export ::=
  item(Export &Table Format,, immed EXTL(mth_export),),
  item(Export &Page Format,, immed EXTL(mth2_export),),
  ;

- snip code generation stuff

subomaux       ::=
```

```
separator,
- ties into rom2snip.cmd
item(Generate Object Pseudocode,, immed


command_file(file= rom2snip.cmd
),),
item(Generate Managed Object C++,, immed command_file(file=snip2cxx.cmd),
),;
- uncomment next line for no right button popup menu
- do_popoup_menu ::= ;
do_popup_menu ::=
   diagram'popupmenu := diag_popup_menu
   (SHOW_CONTROL(menu= noname))
   ;


-  This file imports the following:


!include ::= menubar.rul;
!include ::= ood2.rul;


- menubar.rul forces the menu bar to be precompiled "last" after all
- rules are available
```

B.3 ood2.rul file

```
ood2_rule_version ::= 1.0.alpha;


- ec= regular class, ecd= set class

icon_type(rectangle,solid)::=ec;

icon_type(rectangle,thick_skt_*_0%)::=ecd;

node_parent(ecd,ec)::=socket;


- RELATIONSHIPS

- ecra0=regular, ecra1=Multi-value, ecra2=Essential, ecra3=MV Essential

- ecra4=Dependent, ecra5=MV Dependent

icon_type(arc,solid,arrow_none,arrow_none,arrow)::=ecra0;

icon_type(arc,solid,arrow_none,arrow_none,arrow_o)::=ecra2;

icon_type(arc,solid,arrow_none,arrow_none,arrow_double)::=ecra4;

icon_type(arc,solid_double,arrow_none,arrow_none,arrow)::=ecra1;

icon_type(arc,solid_double,arrow_none,arrow_none,arrow_o)::=ecra3;

icon_type(arc,solid_double,arrow_none,arrow_none,arrow_double)::=ecra5;


-scra0=Subclass, scra1=Role-of

icon_type(arc,thick,arrow_none,arrow_none,arrow)::=scra0;

icon_type(arc,(dash_dot,thick),arrow_none,arrow_none,arrow)::=scra1;


-Part-Of
```

-pcra0=Generic, pcra1=Essential, pcra2=Class Exclusive,

-pcra3=Global Exclusive, pcra4=Multi-Value, pcra5=Essential MV

-pcra6=Class Exlusive MV, pcra7=Global Exclusive MV

icon_type(arc,dash,arrow_none,arrow_none,arrow_diamond)::=pcra0;

icon_type(arc,dash,o_empty,arrow_none,arrow_diamond)::=pcra1;

icon_type(arc,dash,arrow_full_x,arrow_none,arrow_diamond)::=pcra3;

icon_type(arc,dash,arrow_square_fill,arrow_none,arrow_diamond)::=pcra2;

icon_type(arc,dash,arrow_none,cross_double,arrow_diamond)::=pcra4;

icon_type(arc,dash,o_empty,cross_double,arrow_diamond)::=pcra5;

icon_type(arc,dash,arrow_square_fill,cross_double,arrow_diamond)::=pcra6;

icon_type(arc,dash,arrow_full_x,cross_double,arrow_diamond)::=pcra7;

-Ownership

-op0=Regular, op1=Equal Joint, op2=Documented

icon_type(arc,dot,arrow_none,arrow_none,arrow)::=op0;

icon_type(arc,dot,arrow_none,cross_double,arrow)::=op1;

icon_type(arc,dot,arrow_square,arrow_none,arrow)::=op2;

-Path Methods

-pm0=path method, pm1=Attribute Path Method

icon_type(arc,dash,arrow_none,arrow_none,arrow)::=pm0;

icon_type(arc,dash,arrow_square,arrow_none,arrow)::=pm1;

```
-arc_check(ec,ecra0)::=ec;

-arc_check(ec,ecra2)::=ec;

-arc_check(ec,ecra4)::=ec;

-arc_check(ec,ecra1)::=ec;

-arc_check(ec,ecra3)::=ec;

-arc_check(ec,ecra5)::=ec;

-arc_check(ec,scra0)::=ec;

-arc_check(ec,scra1)::=ec;


-arc_check(ec,pcra0)::=ec;

-arc_check(ec,pcra1)::=ec;

-arc_check(ec,pcra3)::=ec;

-arc_check(ec,pcra2)::=ec;

-arc_check(ec,pcra4)::=ec;

-arc_check(ec,pcra5)::=ec;

-arc_check(ec,pcra6)::=ec;

-arc_check(ec,pcra7)::=ec;

-arc_check(ec,pm0)::=ec;

-arc_check(ec,pm1)::=ec;


-arc_check(ecd,ecra0)::=ecd;
```

```
-arc_check(ecd,ecra2)::=ecd;

-arc_check(ecd,ecra4)::=ecd;

-arc_check(ecd,ecra1)::=ecd;

-arc_check(ecd,ecra3)::=ecd;

-arc_check(ecd,ecra5)::=ecd;

-arc_check(ecd,scra0)::=ecd;

-arc_check(ecd,scra1)::=ecd;



-arc_check(ecd,pcra0)::=ecd;

-arc_check(ecd,pcra1)::=ecd;

-arc_check(ecd,pcra3)::=ecd;

-arc_check(ecd,pcra2)::=ecd;

-arc_check(ecd,pcra4)::=ecd;

-arc_check(ecd,pcra5)::=ecd;

-arc_check(ecd,pcra6)::=ecd;

-arc_check(ecd,pcra7)::=ecd;

-arc_check(ecd,pm0)::=ecd;

-arc_check(ecd,pm1)::=ecd;


-arc_check(ecd,pcra0)::=ec;

-arc_check(ecd,pcra1)::=ec;

-arc_check(ecd,pcra3)::=ec;
```

```
-arc_check(ecd,pcra2)::=ec;

-arc_check(ecd,pcra4)::=ec;

-arc_check(ecd,pcra5)::=ec;

-arc_check(ecd,pcra6)::=ec;

-arc_check(ecd,pcra7)::=ec;


arc_check(ec,ecra*)::=ec|ecd;

arc_check(ecd,ecra*)::=ec|ecd;


arc_check(ec,pcra*)::=ec|ecd;

arc_check(ecd,pcra*)::=ec|ecd;


arc_check(ec,scra*)::=ec|ecd;

arc_check(ecd,scra*)::=ec|ecd;


arc_check(ec,pm*)::=ec|ecd;

arc_check(ecd,pm*)::=ec|ecd;


arc_check(ec,op*)::=ec|ecd;

arc_check(ecd,op*)::=ec|ecd;


-arc_check(ec,pcra*)::=ecd;

-arc_check(ecd,pcra*)::=ec;
```

```
-arc_check(ec,scra*)::=ecd;
-arc_check(ecd,scra*)::=ec;

-arc_check(ec,pm*)::=ecd;
-arc_check(ecd,pm*)::=ec;

-arc_check(ec,op*)::=ecd;
-arc_check(ecd,op*)::=ec;

-arc_check(ec,op*)::=ec;
-arc_check(ecd,op*)::=ecd;

dataflow_parent(ecrpp,ecra*) ::= ok;
```

REFERENCES

1. E. J. Neuhold and M. Schrefl. Dynamic Derivation of Personalized Views. *Proc. 14th Int'l Conference on Very Large Databases.*, Long Beach, CA, 1988.

2. M. Schrefl and E. J. Neuhold. A Knowledge-based Approach to Overcome Structural Differences in Object-Oriented Database Integration. In *Proc. IFIP Working Conference on the Role of AI in Database and Information Systems,* Guangzhou, China, 1988. North Holland.

3. M. Schrefl and E. J. Neuhold. Object Class Definition by Generalization using Upward Inheritance In *Proc. 4th Int'l Conference on Data Engineering*, pages 4-13, Los Angeles, CA, Feb.1988.

4. Lawrence A. Rowe and Michael Stonebraker. The Design of POSTGRES. *In Proc. 1986 ACM SIGMOD Conference on Management of Data*, Washington, D.C., May 1986.

5. Michael Stonebraker et al. Third-Generation Database System Manifesto. *SIGMOD Record*,19(3): 31-44, Sept. 1990.

6. R. Agrawal and N. H. Gehani. ODE (Object Database and Environment): The Language and Data Model. In *Proc. 1989 ACM SIGMOD Int'l Conference on Management of Data,* pages 36- 45, Portland, OR May 1989. ACM.

7. R. Agrawal and N. H. Gehani and J. Srinivasan. OdeView: The Graphical Interface to Ode. In Hector Garcia-Molina and H. V. Jagadesh, editors, *Proc. 1990 ACM SIGMOD Int'l Conference on Management of Data*, pages 34-43, Atlantic City, NJ, May 1990. ACM.

8. O. Deux et al. The Story of O_2. *IEEE Trans. Knowledge and Data Eng.,* 2(1):91-108, 1990.

9. C. J. Date. *An Introduction to Database Systems*, volume 1. Addison-Wesley Publishing Co., Inc., Reading, MA, fourth edition, 1986.

10. Ronald J. Brachman and Hector J. Levesque, editors. *Readings in Knowledge Representation*, Morgan Kaufmann Publishers, Inc., Mountain View, CA, 1985.

11. J. F. Sowa. *Principles of Semantic Networks, Explorations in the Representation of Knowledge.* Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1991.

12. J. F. Sowa. *Conceptual Structures, Information Processing in Mind and Machine.* Addison-Wesley Publishing Co., Inc., Reading, MA, 1984.

13. Elaine Rich and Kevin Knight. *Artificial Intelligence*. McGraw-Hill, Inc., New York, NY, second edition, 1991.

14. Peter Pin-Shan Chen. The Entity-Relationship Model: Toward a Unified View of Data. *ACM Trans. Database Syst.,* 1(1):9-36, 1976.

15. Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. The Benjamin/Cummings Publishing Co., Inc., New York, NY, 1989.

16. Jeffrey D. Ullman. *Principles of Database Systems*. Computer Science Press, Rockville, MD, second edition, 1982.

17. R. G. G. Cattell and T. R. Rogers. Entity-Relationship Database User Interfaces. In M. Stonebraker, editor, *Readings in Database Systems*, pages 359-368, San Mateo, CA, 1988.

18. A. Albano and L. Cardelli and R. Orsini. Galileo: A Strongly Typed, Interactive Conceptual Language. *ACM Trans. Database Syst.,* 10(2): 230-260, 1985.

19. A. Albano et al. An Overview of Sidereus: A Graphical Database Schema Editor for Galileo. In *Proc. EDBT '88,* pages 567-571, Venice, Italy, Mar. 1988.

20. S. Abiteboul and R. Hull. IFO: A Formal Semantic Database Mode. *ACM Trans. Database Syst.,* 12(4): 525-565, 1987.

21. David W. Shipman. The Functional Data Model and the Data Language DAPLEX. *ACM Trans. Database Syst.,* 6(1): 140-173, 1981.

22. Daniel Bryce and Richard Hull. SNAP: A Graphics-based Schema Manager. In *Proc. Int'l Conference on Data Engineering,* 1986.

23. Marc Gyssens and Jan Paredaens and Dirk van Gucht. Graph-Oriented Object Model for Database End-User Interfaces. In Hector Garcia-Molina and H. V. Jagadesh, editors, *Proc. 1990 ACM SIGMOD Int'l Conference on Management of Data*, pages 24-33, Atlantic City, NJ, May 1990. ACM.

24. D. H. Fishman et al. Overview of the Iris DBMS. In Won Kim and Frederick H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*, pages 219-250. ACM Press, New York, NY, 1989.

25. Ontologic, Inc., Burlington, MA. ONTOS 2.01 documentation, 1991.

26. Won Kim. A Model of Queries for Object-Oriented Databases, In *Proc. 15th VLDB*, pages 423-432, 1989

27. Kevin Gorman and Joobin Choobineh. The Object-Oriented Entity-Relationship Model (OOERM). *Journal of Management Information Systems*, 7(3): 41-65, 1991.

28. P. Butterworth and A. Otis and J. Stein. The GemStone Object Database Management System. *Commun. ACM*, 34(10): 64-77, Oct. 1991.

29. James Rumbaugh and Michael Blaha and William Premerlani and Frederick Eddy and William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

30. Bertrand Meyer. Tools for the New Culture: Lessons from the Design of the Libraries. *Commun. ACM,* 33(9): 68-88, Sept. 1990.

31. Gerti Kappel and Michael Schrefl. Object/Behavior Diagrams. *In Proc. 7th Int'l Conference on Data Eng.*, pages 530-539, Kobe, Japan, Apr. 1991.

32. Peter Wegner. An Object-Oriented Classification Paradigm. In Schiver and Wegner, editors, *Research Directions in Object-Oriented Programming*. MIT Press, 1987.

33. Hungkway Chao and Veena Prakash Teli. Development of a University Database using the Dual Model of Object-Oriented Knowledge Bases. Master's thesis, Department of Computer and Information Science, NJIT, Newark, NJ, 1990.

34. E. Neuhold and Y. Perl and J. Geller and V. Turau. Separating Structural and Semantic Elements in Object-Oriented Knowledge Bases. In *Proc. of the Advanced Database System Symposium*, pages 67-74, Kyoto, Japan, 1989.

35. J. Geller and Y. Perl and E. Neuhold. Structural Schema Integration in Heterogeneous Multi-Database Systems using the Dual Model. In *Proc. First Int'l Workshop on Interoperability in Multidatabase Systems*, Los Alamitos, CA, 1991. IEEE Computer Society Press.

36. Brad A. Myers et al. Garnet, Comprehensize Support for Graphical, Highly Interactive User Interfaces. *Computer*, 23(11): 71-85, Nov. 1990.

37. Open Software Foundation. *OSF/Motif Style Guide*. Prentice Hall, Englewood Cliffs, NJ, 1990.

38. M. E. Winston and R. Chaffin and D. J. Herrmann. A Taxonomy of Part-Whole Relations. *Cognitive Science*, 11(4): 417-444, 1987.

39. Won Kim and Elisa Bertino and Jorge F. Garza. Composite Objects Revisited. In *Proc. 1989 ACM SIGMOD Int'l Conference on the Management of Data*, pages 337-347, Portland, OR, June 1989.

40. G. T. Nguyen and D. Rieu. Representing Design Objects. In J. Gero, editor, *AI in Design '91*. Butterworth-Heinemann Ltd., 1991.

41. P. Coad and E. Yourdon. *Object-Oriented Analysis*. Yourdon Press Computing Series. Prentice Hall, Englewood Cliffs, NJ, second edition, 1991.

42. Michael Halper and James Geller and Yehoshua Perl and Erich J. Neuhold. A Graphical Schema Representation for Object-Oriented Databases. In R. Cooper, editor, *Interfaces to Database Systems,* pages 282-307. Springer-Verlag, London, 1993.

43. D. Woelk and W. Kim and W. Luther. An Object-Oriented Approach to Multimedia Databases. In *Proc. ACM SIGMOD Int'l Conference on Management of Data*, pages 311-325, Washington, D.C., May 1986.

44. Stanley B. Zdonik and David Maier. Fundamentals of Object-Oriented Databases. In S. B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, pages 1-32, Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.

45. Michael Halper and James Geller and Yehoshua Perl. An OODB ``Part'' Relationship Model. In Y. Yesha, editor, *Proc. ISMM 1st Int'l Conference on Information and Knowledge Management*, pages 602-611, Baltimore, MD, Nov. 1992.

46. Michael Halper and James Geller and Yehoshua Perl. Value Propagation in OODB Part Hierarchies. In B. Bhargava and T. Finin and Y. Yesha, editors, *Proc. ISMM/ACM 2nd Int'l Conference on Information and Knowledge Management*, pages 606-614, Washington, DC, Nov. 1993.

47. M. Halper and Y. Perl and O. Yang and J. Geller. Modeling Business Applications with the OODB Ownership Relationship. In R. S. Freedman, editor, *Proc. 3rd Int'l Conf. on AI Applications on Wall St.*, pages 2-10, New York, NY, June 1995.

48. O. Yang and M. Halper and J. Geller and Y. Perl. The OODB Ownership Relationship. *In Proc. Int'l Conf. on Object-Oriented Information Systems (OOIS'94),* pages 389-403,London, UK, Dec. 1994.

49. J. Geller and Y. Perl and P. Cannata and A. Sheth and E. Neuhold. A Case Study of Structural Integration. In Y. Yesha, editor, Proc. 1st Int'l Conference on Information and Knowledge Management, pages 102-111, Baltimore, MD, Nov. 1992