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

ABSTRACT

OODINI 2.1: AN ENHANCED GRAPHICAL SCHEMA REPRESENTATION FOR OBJECT-ORIENTED DATABASES

by
Rajashekar Rao

The graphical representation of an object-oriented database (OODB) schema is useful for the designers and users of a database system. The purpose of my thesis was to enhance the existing version of OOdini, an interactive graphical tool for editing an 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 the ObjectMaker toolkit to design our own methodology using the ObjectMaker Extension Language.

OODINI 2.1: AN ENHANCED GRAPHICAL SCHEMA REPRESENTATION FOR OBJECT-ORIENTED DATABASE

by
Rajashekar Rao

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 Computer Science

Department of Computer and Information Science

October 1996

APPROVAL PAGE

OODINI 2.1: AN ENHANCED GRAPHICAL SCHEMA REPRESENTATION FOR OBJECT-ORIENTED DATABASES

Rajashekar Rao

Dr. Y. Perl, Thesis Advisor — Date
Professor of Computer and Information Science, NJIT

Dr. J. Geller, Committee Member — Date
Associate Professor of Computer and Information Science, NJIT

Dr. M. Halper, Committee Member — Date
Assistant Professor of Math and Computer Science, Kean College

BIOGRAPHICAL SKETCH

Author: Rajashekar Rao

Degree: Master of Science

Date: October 1996

Undergraduate and Graduate Education

- Master of Science in Computer Science
  New Jersey Institute of Technology, Newark, NJ, 1996

- Bachelor of Engineering in Computer Science
  University of Poona, Pune, India, 1994

Major Computer Science

ACKNOWLEDGEMENT

I would like to express my deepest appreciation to Dr. Y. Perl, who not only served as my research supervisor, providing valuable and countless resources, insight, and intuition, but also constantly gave me support , and encouragement. I would also like to thank Dr. J. Geller and Dr. M Halper for their valuable help and thorough insight into this project.

TABLE OF CONTENTS

TABLE OF CONTENTS
(Continued)

TABLE OF CONTENTS
(Continued)

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Introduction to OOdini

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. OOdini is a comprehensive graphical notation 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. A graphical schema editor called OOdini was developed to allow users to interactively create and manipulate OODB schemata. 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 - enough to support a diverse group of object-oriented data models. The graphical schema editor offers constraint-based editing of the OOdini schema representation, thus making OOdini an effective OODB graphical interface.

1.2 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 [37, 44]. 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 OdeView [3] and O_2[14] support the graphical

display. Hence, many OODBs such as OdeView [3] and O_2[14] support the graphical display of data. However, this type of graphical representation is not considered in this thesis. Our concern is a graphical representation of database schema which can be employed as a data definition language [13].

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 [6, 43] are invariably presented in a graphical form. Conceptual Graphs [42] and Conceptual Dependencies [36] both employ graphical formalisms. Even frames have been given pictorial forms [36].

In the database community, there are a number of data models which present schemata in diagrammatic fashion. Perhaps none of these is more prevalent than the Entity-Relationship model (ER) model [11, 15, 45]. In fact, this graphical language is often used as a diagramming device for other data models such as the relational (e.g. Schemadesign[9]). Another semantic data model with a graphical schema representation is Galileo [4], for which a schema editor Sidereus [5] has been built.

Other models which are readily depicted graphically include IFO [1], which is related to the functional model [41]. SNAP [7], developed by the originators of IFO, is a system which provides this graphical support. GOOD [21], 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 OdeView, Iris [15], O_2 and Ontos [34]. Unfortunately, the class hierarchy relates only a limited part of

the interrelations between classes. Kim [27] presents a notation he calls *schema graph* which captures the normal class hierarchy as well as the class-composition hierarchy. The Object-Oriented Entity-Relationship Model [20], 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[8]. 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 which complements the Object Modeling Technique (OMT) [38]. 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 [29]. These conventions constituted a portion of a larger graphical formalism which 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 variants.

In [26], 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.

CHAPTER 2

OODB GRAPHICAL SCHEMA REPRESENTATION

2.1 Introduction

2.1.1 Motivation

An object-oriented database (OODB) system, typically, is made up of a large number of object classes. Usually in the order of hundreds and sometimes in the order of thousands. The designer must insure that each class contains the attributes necessary to describe its objects and that the classes are connected with appropriate relationships. Relationships describe the *connectivity* between classes. They convey semantic information and allow the retrieval of remote data relevant to a given class.

Since it is the designer who decides the structure of the database, the above issues makes it mandatory that the designer has a solid grasp of the overall structure of the database. Besides, since the user of the database plays no role in deciding the way in which the system is built, the need for the database to have an organized and transparent structure becomes even more important.

This warrants a need to have a graphical language which can prove useful to both the system designer and the end user of the OODB. This graphical language should incorporate a wide variety of constructs to satisfy the most diverse object-oriented models. Also the graphical icons used to represent the various entities in this schemata should have a high mnemonic value.

2.1.2 General Approach

Objects and classes form the most prominent notions that characterize the OODB systems. A class can be regarded as a container for objects which are similar in their structure and semantics in the application. The four properties enlisted below can be used to describe, best, the structure and semantics of objects :

1. Attributes - contain values of a given data type.
2. Relationships - contain references to other classes.
3. Methods - specify operations which can be applied to instances of a given class.
4. Generic relationships - these are similar to relationships in that they are references to other classes; however, these are system-defined, while relationships are user-defined.

A major point of distinction between OODINI and other OODB graphical systems is that in the latter the relationships are viewed simply as pointer-type attributes while in OODINI the edges which represent relationships are labeled - which permits representation of various generic relations, relationships, and path methods. The basis for OODINI is a labeled, directed graph where the vertices represent classes and the ability to label classes allows us to represent different kinds of classes. The edges represent relationships.

2.2 Classes

An object class is represented as a rectangle. With this rectangle the attributes and the operations associated with the class are also represented. The representation of the object class *customer* with its corresponding attributes is shown in Figure 2.1. To

represent an essential attribute (i.e. its value can not be null) we add a small circle to the right of its name.

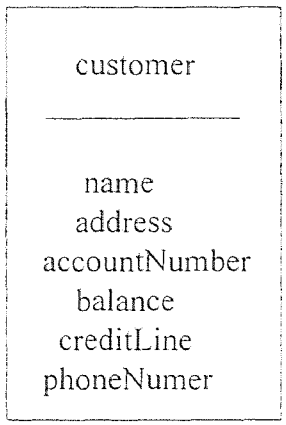

Figure 2.1: The class *customer* and its attributes

Besides the simple class, our system can represent composite classes obtained from other classes by two types of constructors :

1. the set constructor.
2. the tuple constructor.

The set constructor is used to obtain a class whose instances are sets of instances of another class. For example, the class *customers* of Figure 2.2 is obtained by applying the set constructor to the class *customer*. Such a class might have an instance representing the set of all customers who purchased a given product.

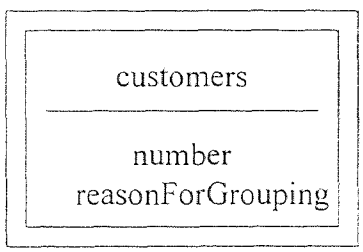

Figure 2.2: The class *customers*

The graphical representation of a set class is a rectangle with a double-framed ›order. The double-frame is used to convey the inherent multiplicity of sets, their non-tomic nature. Since each set class derives all its meaning when associated with a simple lass, it is represented by *socketing* the set class to the corresponding simple class. This s shown in Figure 2.3.

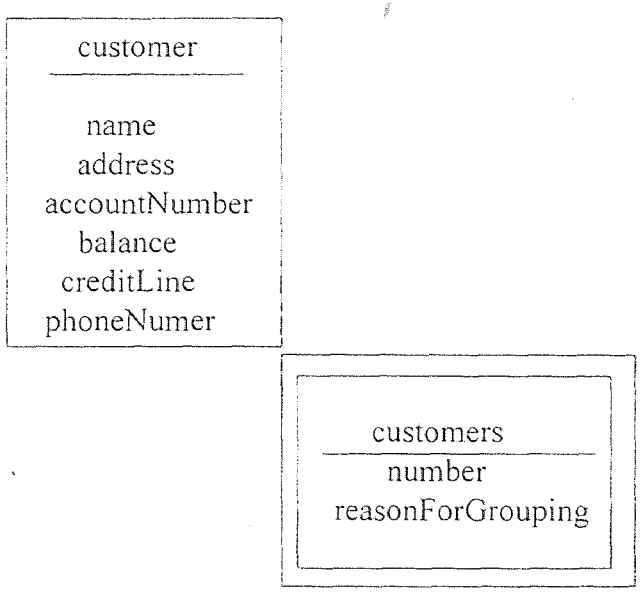

Figure 2.3: Diagrammatic representation of set class.

The tuple constructor is used for association purposes, i.e., to gather a group of lasses together. As a typical example, consider a ternary relation. Sometimes the nformation expressed in a ternary relation cannot be captured by three binary relations ›etween the pairs of classes. In an OODB, the tuple constructor is used to form a class :omprising the three classes of interest. A concrete example of this situation is the class *hipment*, which is defined to be a triple composed of *supplier*, *product* and *department*. The graphical construct of a tuple class is a rectangle with a triangle at the bottom.

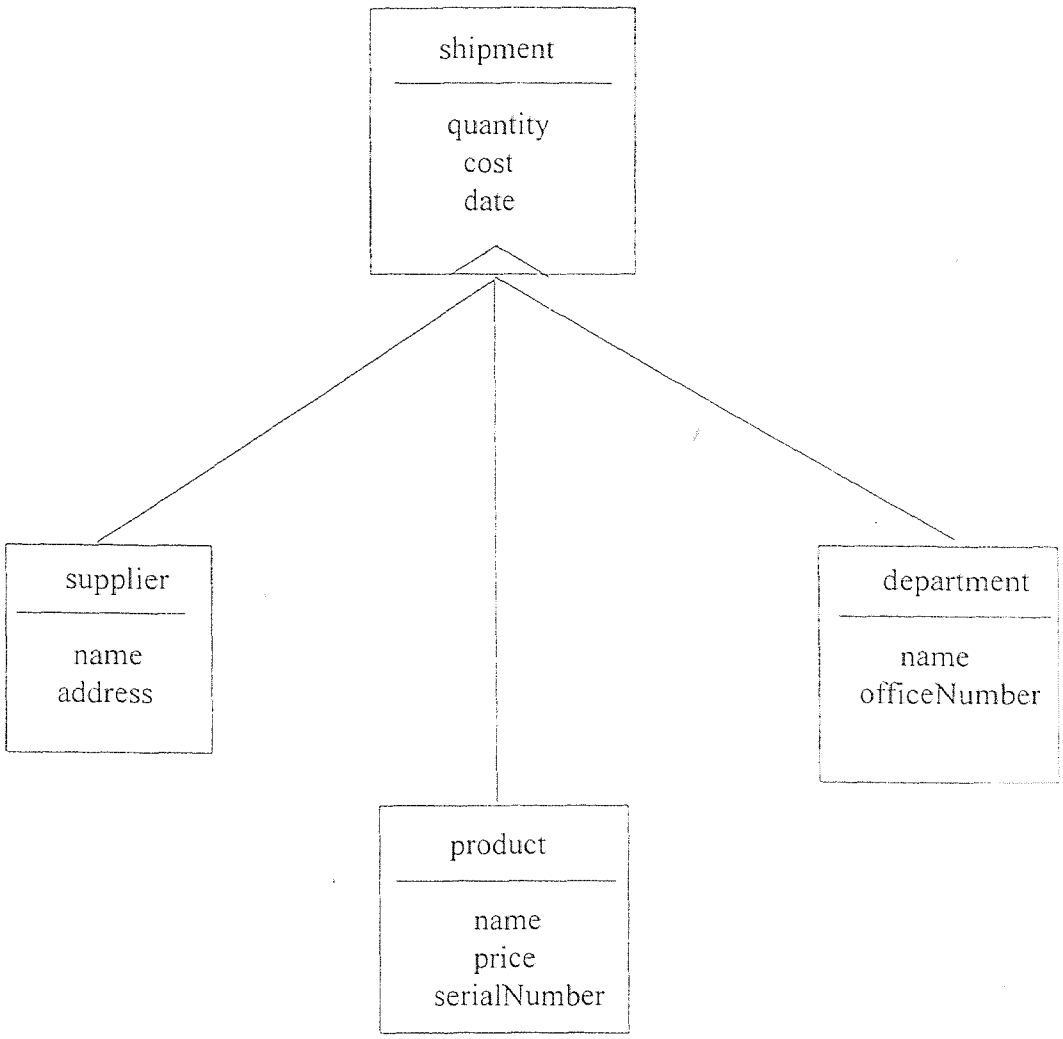

Figure 2.4: A tuple class

2.3 Generic Relationships

Generic relationships are system-defined connections between classes which bear a prime property of generality. The most important among these relationships is that of *subclass* (*is_a*) which enables us to express specialization and create a hierarchy of classes. In certain situations, it might seem convienient to load the *subclass* relationship with further semantics regarding context, to refer to the *categoryof* relationship* . The hierarchy

* In the present discussion both the relationships will not be distinguished.

strongly reflects the structural layout of the application and thus it is very essential in order to get an overall intuitive understanding. Since all normal relationships are shown with a thin line, the subclass relationship is shown with a thick line directed from a more specialized (subclass) to a more general (superclass). This is done in order to make apparent the hierarchy even on cursory inspection. To further emphasize the hierarchy, it is encouraged to place the subclass below its superclass.

In the case where the *subclass* specialization is in a different context from that of the superclass, the relationship is called *roleof*. The graphical representation for *roleof* retains the directed, heavy line feature of the subclass; however, the line is not solid, but a dash-dot pattern. The mnemonic device employed here is borrowed from a feature typical in maps. In maps, the boundary between any two territorial units, such as states and countries, is defined using a dot-dash pattern. Figure 2.5 presents a specialization hierarchy, including *subclass* and *roleof*.

Partof is another relationship which is used to connect a part of a complex or assembled (real-world) object to its integral object. Extensive use of this relationship is made in computer graphics. The graphical representation of the *partof* relationship is a thick, broken line directed from the part to the whole. The *partof* relationship is represented as a thick broken directed line to maintain consistency with the other hierarchical relations, *subclass* and *roleof*, which are represented as thick lines too. As in the case of the *subclass* hierarchy, the schema designer is encouraged to maintain the "parent" part and its descendents in a top-down spatial relationship in the picture.

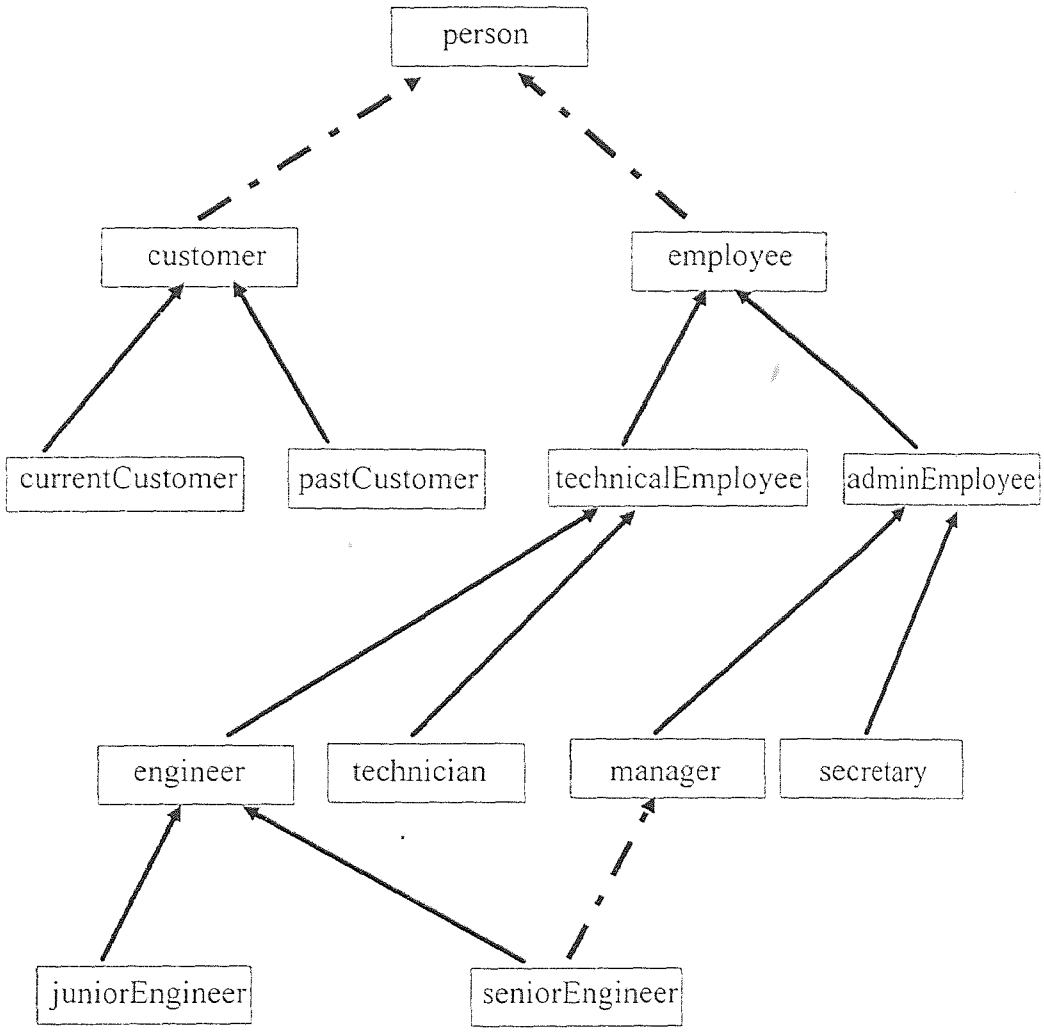

Figure 2.5: A specialization hierarchy

The relationships discussed in this section are the *setof* and its converse *memberof* relation. A class A is in a *setof* relationship with class B if the instances of A are sets of instances of B. Conversely, B is in a *memberof* relationship with A. In contrast to the other generic relationships, the *setof* and *memberof* imply no hierarchy. The graphical representation involves drawing the two participating classes so that they touch at one of

their corners. The set class is drawn with a double-framed box. The reason for the representation is that besides saving space in the picture, the four sides of each rectangle remain accessible from a graphical standpoint In Figure 2.6 the classes section and sections are in *memberof / setof* configuration.

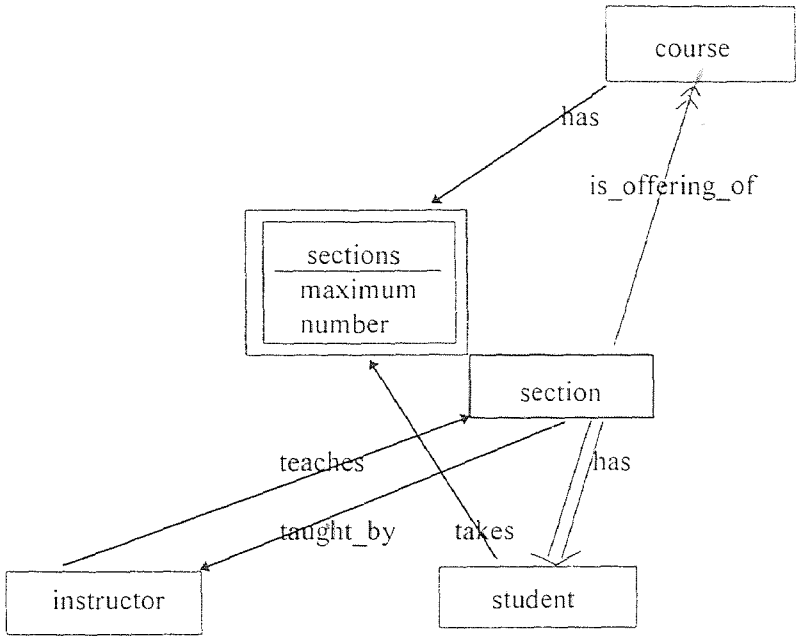

Figure 2.6: The section-student example.

2.4 Relationships

Relationships are user-defined connections between classes. A relationship can be viewed as a pointer to another class. It is thus drawn as an arrow from one class to another. This arrow is a regular one as compared with the heavy arrow of the hierarchical relations. Accompanying the arrow is the name of the relationship.

If the situation warrants a relationship from class A to class B and its converse, it is handled by drawing a pair of labeled arrows in opposite directions between class A and class B. This approach is in contrast to the approach used in ER diagrams wherein a relationship is bi-directional and given an "existence" of its own, complete with its own attributes.

The ER models supports one-to-many relations which in OODINI is called multi-valued relationship. The graphical representation of a multi-valued relationship is a dual-lined arrow. This representation is used to emphasize the multiplicity of the relationship just as in the case of set class. An example of this is the relationship between section and student, where a given section can have many students (see Figure 2.6).

Constraint relationships are those which impose additional semantic constraints on the participating classes. Two aspects that are involved in the complete definition of a constraint relationships are :

- the static definition or state definition which imposes constraints on the database at any fixed instant of time.
- the dynamic and transient definition which expresses the behavior that it implies in the context of change (i.e. creation, deletion and update semantics).

The dynamic aspect of any constraint relationship is required to maintain the constraints imposed by the static aspect.

The two constraint relationships that are represented are as follows:

- *Essential relationships* which must always refer to an existent object. The creation semantics are such that the referent class of the relationship always must have

instances before the source class can be created. The update semantics insists that the relationship cannot be a nil value. The deletion of an instance of the referent class is forbidden if there exist instances of the source class which refer to it.

Essential relationships are represented by placing a small circle on the arrow body. This representation is chosen to maintain consistency since essential attributes are also represented by the addition of a circle.

Figure 2.7 illustrates the above points. It reads as "Working in a department is essential to an employee ". In other words if there is no *department*, then no *employee* can be hired (created) for it. If a *department* is abolished then all *employees* belonging to that department must be transferred to an other department or been fired.

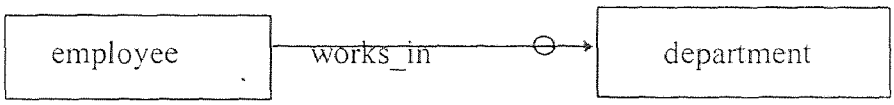

Figure 2.7: An Essential relationship.

A *dependent relationship* is identical to an *essential relationship* except for the case of deletions. The deletion semantics is as follows : Assume that the class A has a dependent relationship to a class B; if an instance *a* of A refers to an instance *b* of B, and if *b* is deleted then *a* get deleted automatically. Thus, the existence of A is dependent on B. It is represented by an double headed arrow to suggest "stronger" connectivity of the relationship. Figure 2.8 shows the graphical icons used to describe the various aspects of relationships.

Figure 2.9 shows a *dependent relationship is_offering_of* from section to course. In other words if a course is deleted then all its sections get deleted automatically.

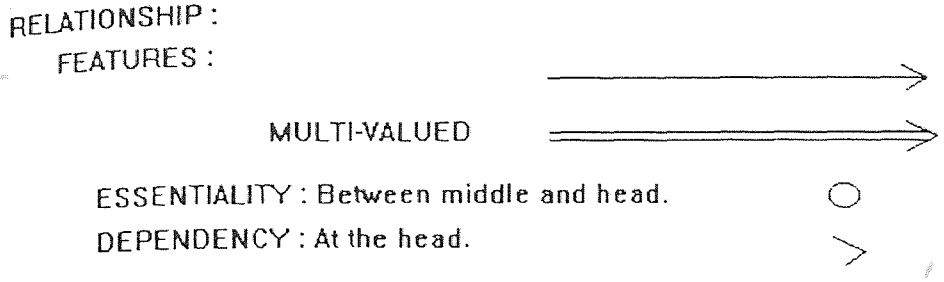

Figure 2.8: Relationships

2.5 Methods

There are two types of methods defined in OODBs. They are :

- path methods
- local methods

Local methods operate locally on the object. Local methods can be divided into selectors mutators (also referred to as reader / writers) and derived attributes. A selector (mutator) method simply reads (writes) a given attribute. Selectors and mutators do not require a separate graphical representation. 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.

A path method is an operation (defined on a class) comprising a chain of classes connected by generic relationships; this chain might end with an attribute or derived attribute. The symbol employed is a dashed thin line arrow pointing from the class

defining the method to the remote data item (e.g. a class or an attribute) it accesses (i.e. ends in). The reason for this representation is that the function of a path method is similar to the function of a relationship: Each is used to retrieve information which is relevant to its own class and is stored in another. The thin arrow is chosen so as to make the symbol for a path reminiscent of the representation of the 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.

As an example, consider the method "get_courses" of the class *instructor* in Figure 2.9. 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 generic relationship path *teaches, setof,* and *is_offering_of.*

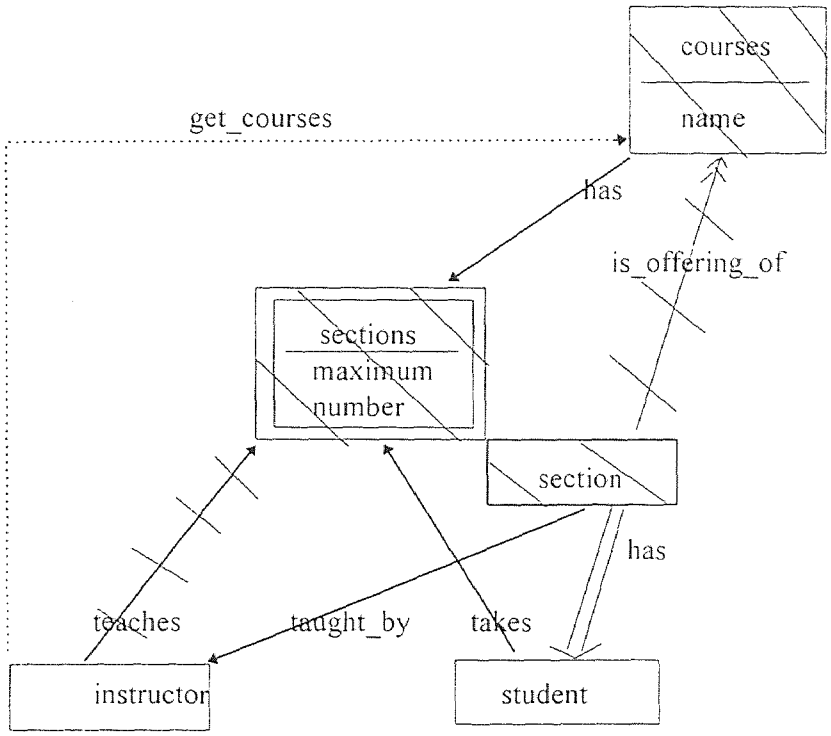

Figure 2.9: The section-student example with the "*get_courses*" method

2.6 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.

The part model has as its foundation in 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 parts and wholes.
- Inheritance 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, 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.

2.6.1 Terminology and Notation

In the following sections, 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.

2.6.2 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 partwhole 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.

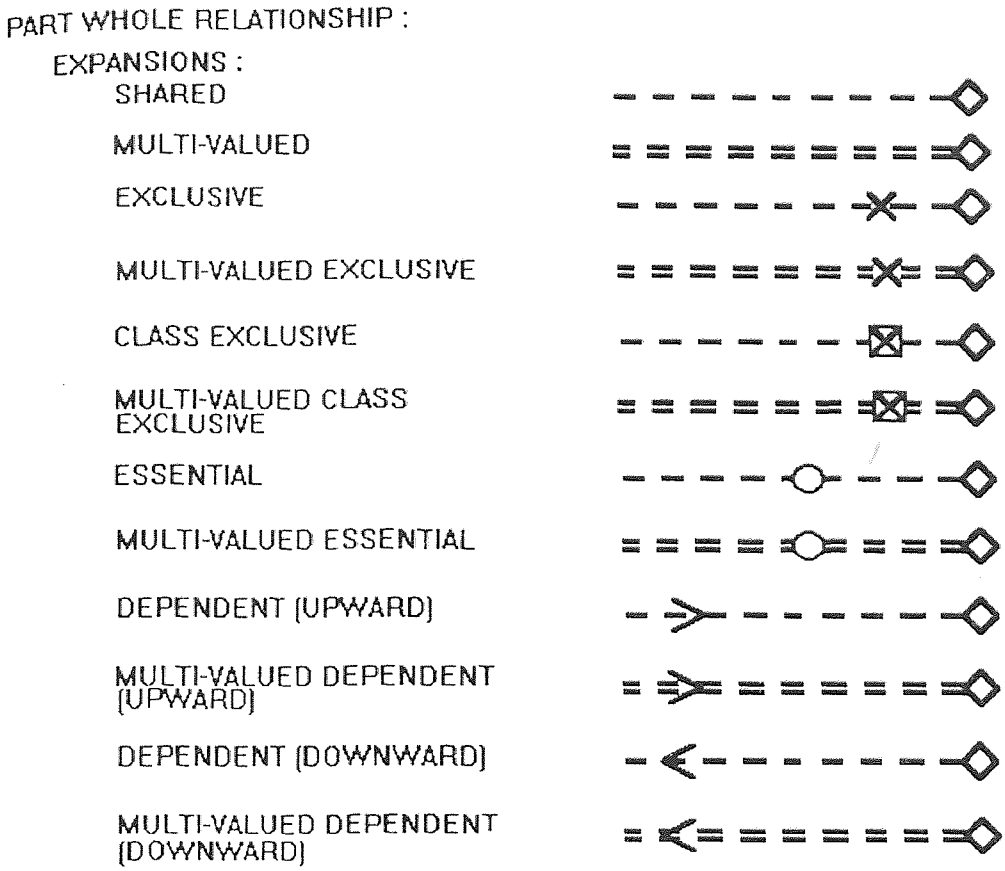

Figure 2.10: Part-Whole Relationship

2.6.3 Exclusive and Shared Part Relationships

Part relationships in general can be divided along the lines of *exclusive* and *shared*. 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. 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, the same 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, 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 which only enforces the exclusiveness constraint on the relationship between the participating

classes, as between *article* and *journal*. Both the exclusive and class exclusive relationships have a formal definitions and their own graphical representations below.

Part relationships which are not exclusive are called shared. 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.

2.6.4 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, especially when the holonyms comprise numerous meronyms.

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 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 2.10.

2.6.5 Value Propagating Part Relationships

We now define two part relationships which support upward and downward value propagation. 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*.

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 2.10).

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, 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. The graphical symbol used is identical to that for the upward propagation except that the prepended arrowhead points downward (Figure 2.10).

2.6.6 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. The generic part symbol aptly expresses the single-valuedness of this part relationship as it is a single-lined connection. 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. 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. 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 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 "O to n" interpretation. Graphically, a dash replaces the omitted bound.

2.7 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.

2.7.1 Definition of Ownership

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

1. The owner,
2. the property that is owned, and
3. the characteristics of the relationships between the two.

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

1. The state or fact of being an owner.
2. 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. For example, Jim as a natural person own 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 be 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 which is his real property. One characteristic of the ownership relationship itself centers around 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 ownerships 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.

2.7.2 Ownership as an OODB Semantic Relationship

2.7.2.1 Transactions and Inheritance: 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. 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.

2.7.3 Formal Definition of the Ownership Relationship

2.7.3.1 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 :

To represent this graphically, we add an X to the dotted arrow to denote eXclusive (See Figure 2.11).

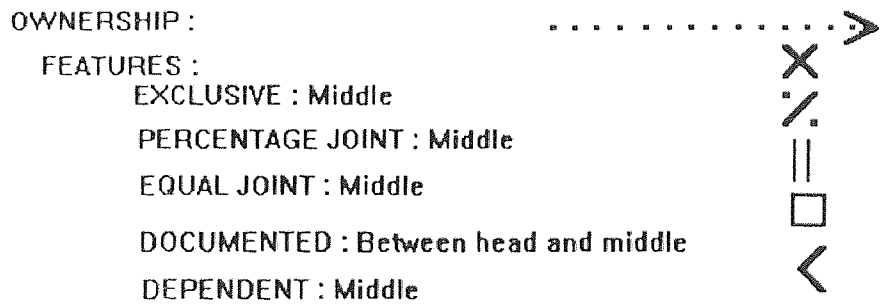

Figure 2.11: Ownership Relationships

Those ownership relationships which 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 (e.g., a joint bank account is freely shared by a couple—we call this free joint), or jointly owned such that each owner takes a certain percentage of the ownership (e.g., 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.

In our graphical notation, a plain dotted arrow indicates free joint. Percentage joint and equal joint are denoted by labels of P and =, respectively (See Figure 2.11).

2.7.3.2 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.

2.7.3.3 Additional Dimensions: The dependency dimension 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. Combinations of various ownership relationships appear in Figure 2.12.

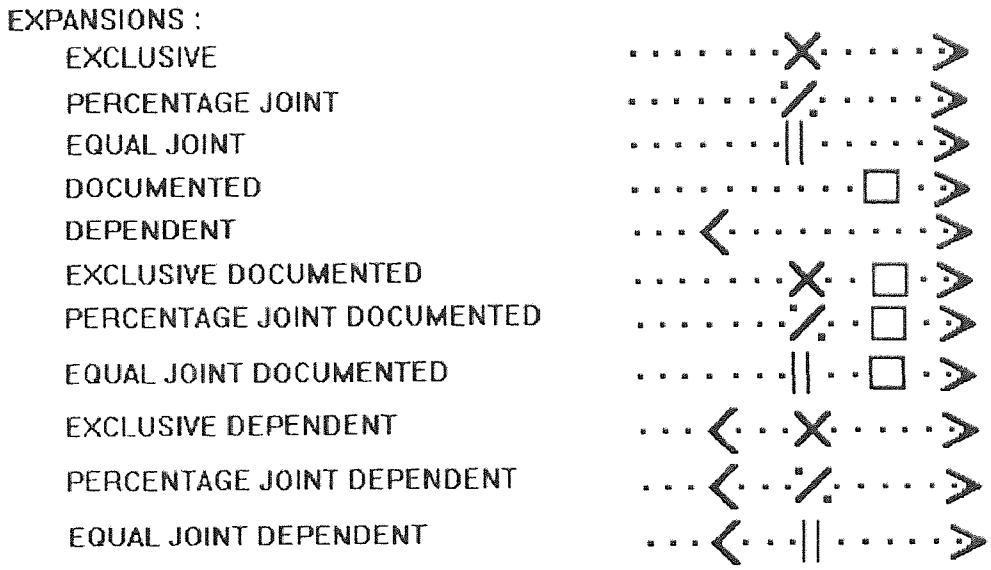

Figure 2.12: Expansion of the Ownership Relationships

CHAPTER 3

THE ARCHITECTURE OF ObjectMaker

3.1 Components

ObjectMaker provides the following functional components.

3.1.1 Diagramming Tool

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

3.1.2 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.

3.1.3 View Management

The View mechanism provides access to repository information through display windows that can be set to be 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.

3.2 Levels of Functionality

3.2.1 Kernel

This layer is provided by Mark V as a set of executables, 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.

3.2.2 Support Layer

This layer is provided as a set of Extension Language files (encrypted for the end user, plain text 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.

3.2.3 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.

3.2.4 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.

3.3 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.

3.4 The ObjectMaker Extension Language

3.4.1 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.

3.4.2 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. The 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.

3.4.3 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 Object Maker, 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.

3.5 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 the completely scanned text. In the ObjectMaker Extension Language, the stored definitions are called rules, and consists 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 :

- 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.

- Invocations may be nested, and will be replaced "inside out", thus, the result of an invocation may become a parameter to another invocation.

- 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.

- 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.

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

3.5.1 Rules

3.5.1.1 Rule Head: A 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 parantheses. 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 header.

3.5.1.2 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 n^{th}

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.

3.6 Implementing Support for a Methodology with ObjectMaker

In this section, we discuss the components that need 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.

3.6.1 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 is 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 matter. Also, names and descriptive text must not contain any characters that might confuse the Extension Language scanner, for example 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 2.1, see Appendix A.

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 include 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 shortcut accelerator keys for some menu items, actions, etc. These 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.

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 2.1 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 should 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), the action defines what is to be done when the item is selected, and the 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.

3.6.2 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 (e.g., 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 00dini methodology from the 00dini 2.1 .rul file:

*icon_type(rectangle, solid) ::=regular_class; icon_type(rectangle, thick_skt_ *_O);*

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

icon_ type (arc, solid, arrow_ none, arrow_none, arrow_ double)

::=dependent_relationship;

icon_t ype(arc, solid_ double, arrow_none, arrow_none, arrow)::=

multi_value_relationship;

icon_ type(arc,solid_ double, arrow_ none, arrow_none, arrow_ o)::=

mv_essential_relationship;

icon_type(arc, solid-double, arrow-none, arrow-none, arrow-double)

::=mv_dependent_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 4

OODINI 2 SPECIFICATIONS

OODINI 2.1 was designed to be an interactive tool to manipulate graphical schemata discussed in Chapter 2.

The following features were to be supported by the tool.

⇒ OODINI 2.1 should be a constraint based graphical editor specifically designed for the representation discussed in Chapter 2. By *constraint based* we mean that the integrity of the schema representations should always be maintained. E.g. Consider a relationship emanating from a class and left dangling, that is, left unattached at its other end. Clearly such a construction is meaningless. So, during input OODINI 2.1 will mark such a diagram as 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 the classes moves, the relationship is automatically moved relative to it.

⇒ OODINI 2.1 will manage 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. Scrollbars should be provided to allow the user to reposition the current working window (in the ordinary graphics sense) of the canvas.

The tool should be able to generate relevant C++ code for any schema represented by it. The code generated need not be complete in all respects in a way that it can be

compiled but it should have all relevant classes, each with its attributes and methods supported by it. The code should also incorporate all relationships between these classes which are represented in the schema. The user may need to integrate the code before compiling. The classes should be in the *orthodox canonical class form*. This means that the concrete data types follow a specific form using class members to augment the C++ compiler's type system so the compiler can generate efficient and safe code for arbitrarily complex abstractions.

⇒ Presented the source code the tool should *reverse engineer* the input to regenerate the schema which would result in generation of the code.

⇒ Since it is mandatory for the tool to support all icons used by the OODINI representation to represent the various classes, relationships and methods. It would be very convenient for the user if he/she is provided with a toolbar which displays the various icons supported by the OODINI methodology. Without such a facility the user would have to go through several sub-menus before he/she can get to the desired item.

⇒ A path method is a sequence of relationships which enable us to retrieve or update distant information. The OODINI icon for path method is a broken line thin arrow from the source class to the target class. The structure of the path method consists of a sequence of relationships starting at the source class and ending at the target class. The tool should have an option of clicking on a path method icon to highlight the corresponding path of relationships.

⇒ A path method can be sequence of relationships ending with an attribute. In such cases we want the path method icon to point to the drawer containing this attribute (in which case the icon for a class will have a chest of drawers to represent the attributes). In the drawer representation of a class a circle drawn alongside the attribute qualifies it as essential.

CHAPTER 5

PHASE 1 OF OODINI 2.0 : A REVIEW

In phase1 OODINI 2.0 was essentially built to support as many icons as there could be using the TDK extension language of ObjectMaker. Since the older versions of TDK did not support the concept of user-defined icons nor the concept of "*writing from rules*", it was impossible for this version of OODINI to support most of the icons including those of *set class* and the *tuple class*.

This phase essentially concentrated on putting schema validating rules in place. A very important requirement was that of the set class which was supposed to be "*socketed*" to its corresponding class. This was implemented using the *skt_outside* option. The code which supported this feature is as follows :

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

om2 => item (Set Class ,, BITMAP_ICON (image => set, flags=>(skt_outside), code => prop concept ecd<\73>),, Set Class , Insert a set class , Insert),

Another important concept which was put in place was the relationship validation. Each relationship made sense only when associating two valid items. E.g. A multivalued

relationship makes sense only between two regular classes and is nonsense when relating a regular class with a set class and vice versa. Such validation rules are present in the ".rul" file. The synatx for this is as follows :

-- ec = regular class

icon_type (rectangle, solid) ::= ec;

-- ecra = multivalued relationship

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

-- ecra2 is a valid arc between ec and ec

arc_check (ec, ecra2) ::= ec;

CHAPTER 6

PHASE 2 DIFFICULTIES DUE TO TDK PREVIOUS RELEASE

The main limitation in the TDK used during the development of OODINI 2 was the non-availability of desired icons and ardornments as specified by the OODINI graphical schema representation. This led to alternative representations and in a few cases the icon was just not supported.

Main among these limitations 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 2.3

The TDK provided very little support when it came to adornments. Although the TDK allowed every relationship to have at most 3 adornments : one each at the head, middle and tail, the adornments could only be one from the set of adornments already supported by the TDK adornment library. One cannot design his own adornment. This makes it impossible to represent relationships such as the *percentage joint relationship* which 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 both the rectangle and a regular relationship are supported by the ObjectMaker TDK, 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 it impossible to represent any of the *part whole relationships*.

Besides, in cases such as the *tuple class* (see figure 2.4) 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 supported by OODINI 2.0.

Many of OODINI 2.0s' 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 which 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 specification to allow such a feature. By not having such rules to govern 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.

CHAPTER 7

IMPROVEMENTS IN TDK 4.0 AND NEW OPTIONS OF OODINI 2.0

The most important of the imporvements made to the TDK was the support to rules. This has been reflected in the latest release of TDK version 4.0. With the ability to program using rules allows the user to implement icons according to their requirement. When written using rules, each of these icons are treated as if they were like any other supported icon. This makes writing rules to validate the semantics of the schema very trivial.

7.1 General Draw Icons

Icons can now be of type "ICON", which requires a parameter "image" (e.g., ICON(image=>a-variable-name)), where the variable-name is either of type BITMAP or DRAWING. Types can be changed at run time by the diagram technique (substituting bitmaps or drawings, animating them, etc).

Bitmaps are loaded by declaring them in a rules file such as !bitmap

name-of-variable ::= file-name;

Drawings are loaded by !drawing name-of-variable ::= drawing-spec; or at run-time by variable ::= <!drawing(drawing-spec)>. There are three types of drawings, a DRAWN_ICON, DRAWN_ROTATBLE_ICON, and a DRAWN_DATAFLOW. The

dataflow is presented by an arc, as with other dataflows, whereas the drawn icon is owned by the diagram canvas or another node.

The drawing begins with the pen in the default node color, the style default (whatever thickness is specifies on creation or by dialog), all lines hit (for socket and line interception).

7.1.1 Bitmap Icons

Bitmap icons are monochrome or color bitmaps, which can are scalable icons. They are specified in extension language files, such as ".mnu" files for a methodology.

To specify a bitmap in a loaded extension language file :

!bitmap name ::= file-path-of-.bmp-file;

7.1.2 General Icons

General icons are drawings consisting of line and arc segments, plus optionally participating bitmaps. They are specified in extension languages files, such as ".mnu" files for a methodology, or at run time.

To specify in a loaded extension languages file:

!drawing name ::= (drawing elements);

or to specify at run time

<name::=<!drawing(drawing elements)>>

The drawing assumes there are pen-drawn lines, and that when the pen is down a line can be "hit". To be hit means that an arrow will terminate when reaching the closest outermost hittable line. A line which can be hit also can hold a socket (including line anchors).

Drawings are by default rotatable and scalable when drawn. The drawing begins with the pen in the default node color, the style default (whatever thickness is specified on creation or by dialog), all lines can be hit (for socket and line interception).

7.2 Extension Language Support

TDK 4.0 has another feature that allows the user to have unsupported adornments at the head or tail of the icon. The script that supports this feature is

omb ::= item (..... , ARC (..... , head_image => head, ... , tail_image => tail .....) .....) ;

where both *head* and *tail* are defined using the rules as

!bitmap head (.....,

.....,

);

and

!bitmap tail (..... ,

..... ,

);

The advantage of using this method is that besides getting a user-defined icon one can write rules to validate the use of such an icon. Rules to govern these icons are exactly the same as those for normal arcs.

CHAPTER 8

OPEN DEVELOPMENT PROBLEMS AND DIFFICULTIES

8.1 Adornment Shortcomings

Although ObjectMaker allows the user to customize new icons according to our needs, it still leaves certain issues and aspects untouched. One of these is the fact that a certain icon might need an adornment in the middle of the icon body .e.g. Consider a *part-whole* relationship :- *class exclusive essential* . It has a graphical representation which is as shown in Figure 8.1

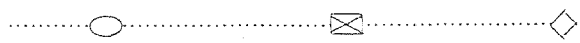

Figure 8.1: Class Exclusive Essential Relationship

This icon can only be obtained by writing from rules (since none of the adornments needed for the icon are supported directly by ObjectMaker). But even writing from rules doesn't support adornments at the middle of the icon. It only supports adornments at the head and tail of the icon.

This problem surfaces for every icon which has two or more adornments.

8.2 Code Generation

This section deals with the feature of code generation which is considered very important for any tool used to model object-oriented systems. Without this feature the tool remains as a mere 'schema viewing' tool.

The issue of code generation is a broad subject. It can be better dealt by subdividing the topic of discussion into two sections :

1. code generation from schema diagrams
2. reverse engineering code to generate the schema diagram.

8.2.1 Code Generation from Schema Diagrams

Object-oriented systems use one or more methodologies to represent their schema. Each methodology has its own interpretation for different icons and relationships. More than often a user designs a system also needs to code the design. C++, SmallTalk are a few of the popular languages used to code such object-oriented models. A tool which only supports schema drawing just validates user design but does not assist the users in any way when it comes to implementation of the design.

The code generation feature is getting increasingly popular with designing tools. The user has a clear advantage with such packages that the software generates code for the design. It should be noted that the code which is generated can not in any respect be classified as complete. It would in most cases be individual objects which the user may have to integrate and patch up before this code can be compiled and be used.

ObjectMaker supports code generation only for the Booch Methodology.

8.2.2 Reverse Engineering Code to Generate the Schema Diagram

Reverse engineering is the process of examining a program's source code to recover information about its schema design. To reverse engineer a program:

1. You analyze source files containing C++ code.
2. You direct the software package to export design information extracted from the source code.
3. You use the tool to view and manipulate the reverse-engineered model file directly.

This feature, that of reverse engineering the code to generate the diagram, is very powerful when it comes to redesigning object-oriented systems that have only code available.

This feature is not supported by ObjectMaker.

APPENDIX A

METHOD.CFG

```
(name=>OOdini2Icon ,

 desc=>,
 date=>,
 author=>,
 name=>OOdini2 ,

 menus=>(

   (name=>NJIT,
    desc=>,
    date=>,
    author=>,
    file=>ood2.mnu ),

   ),
)
```

APPENDIX B

THE RULE FILE

```
ood2_rule_version ::= 1.0.alpha;

--OBJECTS
-- ec= regular class,
-- ecd= set class
-- ect= tuple 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
-- ecra6=Essential Dependent
-- ecra7=MV Essential Dependent

icon_type(arc,solid,arrow_drawn,arrow_drawn,arrow_drawn)::=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;
icon_type(arc,solid,arrow_none,arrow_none,arrow_double)::=ecra6;
icon_type(arc,solid_double,arrow_none,arrow_none,arrow_double)::=ecra7;


-- 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 Exclusive 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;
icon_type(arc,dash,o_empty,cross,arrow_diamond)::=pcra8;


--Ownership
-- op0=Regular
-- op1=Equal Joint
-- op2=Documented
-- op3=Exclusive
-- op4=Percentage Joint
-- op5=Equal Joint Documented
-- op6=Exclusive Documented
-- op7=Dependent
-- op8=Equal Joint Dependent
-- op9=Exclusive Dependent

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_none,arrow_square,arrow)::=op2;
icon_type(arc,dot,arrow_full_x,arrow_none,arrow)::=op3;
icon_type(arc,dot,arrow_none,arrow_square,arrow)::=op4;
icon_type(arc,dot,arrow_none,arrow_square,arrow)::=op5;
icon_type(arc,dot,arrow_none,arrow_square,arrow)::=op6;
icon_type(arc,dot,arrow_none,arrow_square,arrow)::=op7;
icon_type(arc,dot,arrow_none,arrow_square,arrow)::=op8;
icon_type(arc,dot,arrow_none,arrow_square,arrow)::=op9;
```

```
--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;


-- Various Arc_checks

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


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;


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

APPENDIX C

THE MENU FILE

```
ood2_menu_version ::= 1.1.alpha;
!preexecute <bmpdir:=<CommonDir><dirsep>bitmaps<filesep>>;


--****THESE .BMP FILES CAN BE EDITED USING PAINTBRUSH****

!bitmap mbomshp  ::= <MenuDir><dirsep>ood2shp.bmp;
!bitmap mbomagg ::= <MenuDir><dirsep>ood2shp2.bmp;
!bitmap mbomgen ::= <MenuDir><dirsep>ood2shp3.bmp;
!bitmap mbombina ::= <MenuDir><dirsep>ood2shp4.bmp;
!bitmap mbown ::= <MenuDir><dirsep>own.bmp;
!bitmap mbprwh1 ::= <MenuDir><dirsep>parwh1.bmp;
!bitmap mbprwh2 ::= <MenuDir><dirsep>parwh2.bmp;
!bitmap mbgeneric  ::= <BmpDir>generics.bmp;


-- Arcs for Ownerships...begin

!drawing percs ::= (
        PenStyle(solid),
        MoveTo(-8,8),
        LineTo(16,16),
        MoveTo(-8,-8),
        MoveTo(-5,6),
        LineTo(1,-1),
        MoveTo(-1,1),
        MoveTo(5,-6),
        MoveTo(5,-6),
        LineTo(-1,1),
        );

!drawing rarrs ::= (
        PenStyle(solid),
        MoveTo(0,4),
        LineTo(8,8),
        MoveTo(-8,-8),
        LineTo(-8,8),
```

```
        );

--End Ownership Arc_icons


-- Set class Icon

!drawing set ::= (
        PenStyle(stop_rotate),
      PenStyle(solid_double),
        MoveTo(20,20),
        LineTo(0,-40),
        LineTo(-80,0),
        LineTo(0,40),
      LineTo(80,0),
        );


-- Tuple Class Icon

!drawing tup ::= (
        PenStyle(stop_rotate),
      PenStyle(solid_double),
        MoveTo(20,20),
        LineTo(0,-40),
        LineTo(-80,0),
        LineTo(0,40),
      LineTo(80,0),
        MoveTo( -50, -40 ),
        LineTo( 10,10 ),
        LineTo(10,-10),
        );


-- 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)>
```

```
);


--****THIS IS WHERE THE LABELS ARE DEFINED FOR EACH NODE OR
ARC****

dispatch_edit(ec) ::=
  <putup_label("Class Definition",(Name,Attributes,Operations))>;
dispatch_edit(ecd) ::=
  <putup_label("Set Class Definition",(Name,Attributes,Operations))>;
dispatch_edit(ect) ::=
  <putup_label("Tuple 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(ecra6) ::=
   <putup_label("Essential Dependent", (Name))>;
dispatch_edit(ecra7) ::=
   <putup_label("MV Essential 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(pcra8) ::=
  <putup_label("Exclusive Essential",(Name))>;
dispatch_edit(op0) ::=
  <putup_label("Regular",(Name))>;
dispatch_edit(op1) ::=
  <putup_label("Equal Joint",(Name))>;
dispatch_edit(op2) ::=
  <putup_label("Documented",(Name))>;
dispatch_edit(op3) ::=
  <putup_label("Exclusive",(Name))>;
dispatch_edit(op4) ::=
  <putup_label("Percentage Joint",(Name))>;
dispatch_edit(op5) ::=
  <putup_label("Equal Joint Documented",(Name))>;
dispatch_edit(op6) ::=
  <putup_label("Exclusive Documented",(Name))>;
dispatch_edit(op7) ::=
  <putup_label("Dependent",(Name))>;
dispatch_edit(op8) ::=
  <putup_label("Equal Joint Dependent",(Name))>;
dispatch_edit(op9) ::=
  <putup_label("Exclusive Dependent",(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),,Cl
asses/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(mbown'10,,use(ot1)),
        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)),
bmpitem(mbown'15,,use(omb17)),
```

```
bmpitem(mbown'16,,use(omb19)),
        ),
  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)),
        bmpitem(mbown'1,,use(om8)),
        bmpitem(mbown'2,,use(om9)),
        bmpitem(mbown'3,,use(om10)),
        bmpitem(mbown'4,,use(om12)),
        bmpitem(mbown'5,,use(om11)),
        bmpitem(mbown'7,,use(om14)),
        bmpitem(mbown'9,,use(om13)),

        ),
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(mbprwh2'1,,use(omb20)),
bmpitem(mbombina'18,,use(omb18)),
        );

<xa(NEW_NODE(icon=>rectangle,flags=>(thick,skt_straddle),parent=><pParentID>,co
de=>"prop concept ecoval;", width=>4,xpos=>20, ypos=>32,flags=>sktstraddle))>

<xa(NEW_NODE(icon=>anchor,flags=>(thick,skt_straddle),parent=><pParentID>,code
=>"prop concept ecoval;", width=>4,xpos=>10, ypos=>20,flags=>sktstraddle))>

<xa(NEW_NODE(icon=>rectangle,flags=>(thick,skt_straddle),parent=><pParentID>,co
de=>"prop concept ecoval;", width=>4,xpos=>50, ypos=>20,flags=>sktstraddle))>
```

```
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,, BITMAP_ICON(image=>set, flags=>(skt_outside),
code=>prop concept ecd<\73>),, Set Class, Insert a Set Class, Insert),


         ot1=>item(Tuple Class,,      BITMAP_ICON(image=>tup,
location=>NODELBL_INSIDE,code=>prop concept ect<\73>),,Tuple 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,cod
e=>prop concept ecra5<\73>),,MV Dependent, Insert a MV Dependent
Relationship,Drawing arrows),
```

```
        omb17=> item(Essential Dependent, , Arc(flags=>SOLID,
head=>ARROW_DOUBLE_O, tail=>ARROW_NONE, code=>prop concept
ecra6<\73>),, Essential Dependent, Insert an Essential Depenedent Relationship,
Drawing arrows),

        omb19=> item(MV Essential Dependent, , Arc(flags=>SOLID_DOUBLE,
head=>ARROW_DOUBLE_O, tail=>ARROW_NONE, code=>prop concept
ecra7<\73>),, MV Essential Dependent, Insert an MV Essential Depenedent
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,c
ode=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),


        om8=>item(Exclusive, ,
ARC(flags=>dot,head=>ARROW,tail=>ARROW_FULL_X,code=prop concept
op3<\73>),,Exclusive Ownership, Insert an Ownership, Drawing arrows),
```

```
   om9=>item(Percentege Joint,, ARC(flags=>dot,  head=>arrow_drawn,
head_image=>rarrs, tail=>arrow_drawn, tail_image=>percs, code=>prop concept
op4<\73>), immed mth H, Percentage Joint Relationship, Insert an percentage joint
relationship, drawing arrows),

        om10=>item(Equal Joint Documented, , ARC(flags=>dot,head=>ARROW,
middle=>cross_double, tail=>ARROW_SQUARE,code=prop concept op5<\73>),,Equal
Joint Documented Ownership, Insert a Equal Joint Documented Ownership, Drawing
arrows),

        om11=>item(Exclusive Documented, , ARC(flags=>dot,head=>ARROW,
middle=>cross, tail=>ARROW_SQUARE,code=prop concept op6<\73>),,Exclusive
Documented Ownership, Insert a Exclusive Documented Ownership, Drawing arrows),

        om12=>item(Dependent,, ARC(flags=>dot,  head=>arrow_drawn,
head_image=>rarrs,  tail=>arrow_drawn, tail_image=>rarrs code=>prop concept
op7<\73>), immed mth H, Dependent Relationship, Insert a Dependent relationship,
drawing arrows),

        om13=>item(Equal Joint Dependent,, ARC(flags=>dot,  head=>arrow_drawn,
head_image=>rarrs, middle=>cross_double, tail=>arrow_drawn, tail_image=>rarrs
code=>prop concept op8<\73>), immed mth H, Equal Joint Dependent Relationship,
Insert a Equal Joint Dependent relationship, drawing arrows),

        om14=>item(Exclusive Dependent,, ARC(flags=>dot,  head=>arrow_drawn,
head_image=>rarrs, middle=>cross, tail=>arrow_drawn, tail_image=>rarrs code=>prop
concept op9<\73>), immed mth H, Exclusive Dependent Relationship, Insert a Exclusive
Dependent relationship, 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 Reltionship,Drawing arrows),

      omb8=>item(Role-of,
,ARC(flags=>(DASH_DOT),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),

omb20=>item(Exclusive Essential,, Arc(flags=>DASH, head=>ARROW_DIAMOND, middle=>cross, tail=>O_EMPTY, code=>prop concept pcra8<\73>),, Exclusive Essential, Insert an Exclusive essential 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=><inidir><dirsep>help<filesep>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=><CntxDir><DirSep>common<DirSep>om2snip<FileSep>rom2snip
.cmd
),),
item(Generate Managed Object C++,, immed
command_file(file=><CntxDir><DirSep>common<DirSep>om2snip<FileSep>snip2cxx.
cmd),
),
;

-- uncomment next line for no right button popup menu
-- do_popoup_menu ::= ;
do_popup_menu ::=
   <--<.diagram'popupmenu := diag_popup_menu>>
   <--<xa(SHOW_CONTROL(menu=>noname))>>
   ;

--  This file imports the following:

!include ::= <CommonDir><DirSep>menus<FileSep>menubar.rul;
!include ::= <MenuDir><FileSep>ood2.rul;

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

REFERENCES

1. S. Abiteboul and R. Hull IFO: A formal semantic database model. *ACM Trans. Database Syst.*, 12(4):525-565, 1987.

2. R. Agrawal and N. H. Gehani. ODE (Object Database and Environments):The language and data model. In *Proc. 1989 ACM SIGMOD Int'l Conference on Management of Data*, pp 36-45, Portant, OR, May 1989, ACM.

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

4. A. Albano, L. Cardelli, and R. Orsini. Galileo: A strongly typed, interactive conceptual language. *ACM Trans. Database Syst.*, 10(2):230-260, 1985.

5. A. Albano et al. An overview of Sidereus: A graphical database editor for galileo. In *Proc. EDBT '88*, pp 567-571, Venice, Italy, Mar. 1988.

6. R. J. Branchman and H. J. Levesque, editors. *Readings in Knowledge Representation.* Morgan Kauffmann Publishers, Inc., Mountain View, CA, 1985.

7. D. Bryce and R. Hull. SNAP: A graphics-based schema manager. In *Proc. Int'l Conference on Data Engineering*, 1986.

8. P. Butterworth, A. Otis, and J. Stein. The GemStone object database management systems. *Commun. ACM*, 34(10):64-77, Oct. 1991.

9. R. G. G. Cattell and T. R. Rogers. Entity-Relationship database user interfaces. In M. Stonebraker, editor, *Readings in Database Systems*, pp 359-368. Morgan Kaufmann Publishers, Inc/, San Mateo, CA, 1988.

10. H. Chao and V. P. Teli. Development of a university database using the Dual Model of object-oriented knowledge bases. Master's thesis, NJIT, Newark, NJ, 1990.

11. P. P. S. Chen The Entity-Relationship Model: Towards a unified view of data. *ACM Trans. Database Syst.*, 1(1):9-36, 1976.

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

13. C. J. Date. *An Introduction to Database Systems*, volume 1. Addision-Welsey Publishing Co., Inc., Reading, MA, fourth edition, 1986.

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

15. R. Elmasari and S. B. Navathe. *Fundamentals of Database Systems*. The Benjamin/Cummins Publishing Co., Inc., New York, NY, 1989.

16. D. H. Fishman et al. Overview of the IRIS DBMS. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases and Applications*, pp 219-250. ACM Press, New York, NY, 1989.

17. J. Geller. *A Knowledge Representation Theory for Natural Language Graphics*. PhD thesis, SUNY Buffalo CS Department, 1988. Tech. Report 88-15.

18. J. Geller, Y. Perl, P. Cannata, 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*, pp 102-111, Baltimore, MD, Nov. 1992.

19. J. Geller, Y. Perl and E. Neuhold. Structural schema integration in hetrogenous multi-database systems using the Dual Model. In *Proc. First Int'l Workshop on Interoperability in Multidatabase Systems*, pp 200-203, Los Alamitos, CA, 1991. IEEE Computer Society Press.

20. K. Gorman and J. Choobineh. The Object-Oriented Entity-Relationship Model (OOERM). *Journal of Managemant Information Systems*, 7(3):41-65, 1991.

21. M. Gyssens, J. Paredaens, D. van Gucht. A graph-oriented object model for database end-user interfaces. In H. Garcia-Molina and H. V. Jagadish, editors, *Proc 1990 ACM SIGMOD Int'l Conference on Management of Data*, pp 24-33, Atalnatic City, NJ, May 1990. ACM.

22. M. Halper, J. Geller, and Y. Perl. An OODB "part" relationship model. In Y. Yesha, editor, *Proc. ISMM Ist Int'l Confernce of Information and Knowledge Management*, pp 602-611, Baltimore, MD, Nov. 1992.

23. M. Halper, J. Geller, and Y. Perl. Value propogation in OODB part hierarchies. In B. Bhargava, T. Finin, and Y. Yesha, editors, *Proc. ISMM/ACM 2nd Int'l Conference on Information and Knowledge Management*, pp 606-614, Washington, DC, Nov. 1993.

24. M. Halper, J. Geller, Y. Perl, and E. J. Neuhold. A graphical schema representation for object-oriented databases. In R. Cooper, editor, *Interfaces to Database Systems*, pp 282-307. Springer-Verlag, London, 1993.

25. M. Halper, Y. Perl, O. Yang, and J. Geller. Modeling business applications with the OODB ownership relationship. In R. S. Freedman, editor, *Proc. 3rd Int'l Conference on AI applications on Wall St.*, pp 2-10, New York, NY, June 1995.

26. G. Kappel and M. Schrefl. Object/Behaviour diagrams. In *Proc. 7th Int'l Conference on Data Eng.*, pp 530-539, Kobe, Japan, Apr. 1991.

27. W. Kim. A model of queries for object-oriented databases. In *Proc. 15th VLDB*, pp 423-432, 1989.

28. W. Kim, E. Bertino, and J. F. Garza. Composite objects revisited. In *Proc. 1989 ACM SIGMOD Int'l Conference on Management of Data*, pp 337-347, Portland, OR, June 1989.

29. B. Meyers. Tools for the new culture. Lessons from the design of the Eiffel libraries *Comm. ACM*, 33(9):68-88, Sept 1990.

30. B. A. Myers et al. Garnet, comprehensive support for graphical, highly interactive user interfaces. *Computer*, 23(11):71-85, Nov 1990.

31. E. Neuhold, Y. Perl, J. Geller, and V. Turau. Seperating structural and semantic elements in object-oriented knowledge bases. In *Proc. of the Advanced Database System Symposium*, page 67-74, Kyoto, Japan, 1989.

32. E. J. Neuhold and M. Schrefl. Dynamic derivation of personalized views. In *Proc. 14th Int'l Conference on Very large Databases*, Long Beach, CA, 1988.

33. G. T. Nguyen and D. Rieu. Representing design objects. In J. Gero, editor, *AI in Design '91*. Butterworth-Heinemann Ltd., London, England, 1991.

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

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

36. E. Rich and K. Knight. *Artificial Intelligence*. McGraw-Hill, Inc., New York, NY, second edition, 1991.

37. L. A. Rowe and M. Stonebraker. The design of POSTGRES. In *Proc. 1986 ACM SIGMOD Conference of Management of Data*, Washington, D.C., May 1986.

38. J. Raumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design*. Prentence Hall, Englewood Cliffs, NJ, 1991.

39. M. Schrefl and E. J. Neuhold. A knowledge-based approach to overcome structural differences in object-oriented databases integration. In *Proc. IFIP Working Conference on the Role of AI in Database and Information Systems*, Guangzhou, China, 1988. North Holland.

40. M. Schrefl and E. J. Neuhold. Object class definition by generalization using upward inheritance. In *Proc. 4th Int'l Conference on Data Engineering*, page 4-13, Los Angeles, CA Feb. 1988.

41. D. W. Shipman. The Fuctional Data Model and the data language DAPLEX. *ACM Trans. Database Syst.*, 6910:140-173, 1981.

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

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

44. M. Stonebraker et al. Third-generation database system manifesto. *SIGMOD Record*, 19(3):31-44, Sept. 1990.

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

46. P. Wegner. An object-oriented classification paradigm. In Schiver and Wegner, editors, *Research Directions in Object-Oriented Programming.* MIT Press, 1987.

47. M. E. Winston, R. Chaffin, and D. J. Herrmann. A taxonomy of part-whole relations. *Cognitive Sciences*, 11(4):417-444, 1987.

48. D. Woelk, W. Kim, and W. Luther. An object-oriented approach to multimedia databases. In *Proc. ACM SIGMOD Int'l Conference on Management of Data*, pp 311-325, Washington, D.C., May 1986.

49. O.Yang, M. Halper, J.Geller, and Y. Perl. The OODB ownership relationship. In *Proc. Int'l Conference on Object-Oriented Information Systems (OOIS '94)*, pp 389-403, London, UK, Dec. 1994.

50. S. B. Zdonik and D. Maier. Fundamentals of object-oriented databases. In S. B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*, pp 1-32. Morgan Kaufmann Publishers, Inc., San Mateo, CA, 1990.