ODDESSY: AN OBJECT-ORIENTED DATABASE DESIGN SYSTEM

Jim Diederich & Jack Milton

Department of Mathematics
University of California, Davis

Abstract. A prototype database design system is presented. The system is object oriented in two ways. First, a uniform object-message paradigm is used for the design, definition, and manipulation language. Second, the structure of the underlying system is object oriented and is being implemented in Smalltalk-80.

Introduction. A current topic which has generated much interest in database research is the use of object oriented concepts in database management and design systems[1]. The exact meaning of the term object oriented is not commonly agreed upon, however. For example, several systems have been developed to aid in designing databases which are termed object oriented. Their object orientation seems to revolve around the treatment of elements in the domain of the database application, such as entities and relationships, as objects which are graphically displayed and manipulated by the designer. However, these systems are implemented in conventional programming languages such as C and Pascal and do not use the object-message pair and supporting environment found in an object oriented language such as Smalltalk-80[2].

We are currently in the process of developing and prototyping a database design system, ODDESSY (Object-oriented Database DESign SYstem), which is object oriented in several respects. First, it is being implemented using Smalltalk-80. But more importantly, the object-message paradigm is used in new and novel ways: (1) to create a simple and uniform language as a database design, definition, and manipulation language, (2) to support multiple approaches to data modeling, (3) to allow, through extensions to the system, incorporation of new concepts such as abstract data types[3,4], and frames[5], and (4) to support rule-based inference. Eventually, our system will support the full range of design activities at the conceptual, logical and physical levels.

Some systems are based on a single data model such as the Semantic Data Model[6] (SDM), one of its related models, or the Entity-Relationship Model[7] (ER Model). Our design system will allow more flexiblilty by supporting several models both in terms of the style the designer can adopt and the underlying information which is captured in the design. In particular, major features of the SDM, the Structural Model[8], and the ER Model are incorporated into our system. We see no apriori restrictions on the modelling ODDESSY can support and anticipate that ODDESSY can be used for modelling object-oriented databases[9].

The Architecture. The architecture of the ODDESSY system consists of four levels, with each level dependent on the previous ones.

The top level is the **model level**. It consists of the interactive interface and the language used for creating and specifying the design. The primary purpose of this level is to capture the conceptual design of the database. The designer will use a combination of mouse, keyboard, and multiple windows in the course of using the system which will reduce the input required and eliminate the need to remember specific syntax or commands, Figure 1. The second level is the **object level** which is a repository of the objects created in the course of producing the database design. The third level is the **production level**. It is here that rules and procedures are created and employed to transform the conceptual design into a logical design. The rules are conditioned on the current state of objects in the object level and their actions modify existing objects and create new ones. Our initial goal is to transform the conceptual model into normalized relations. In its current state, rules are used to generate functional dependencies which in turn produce third normal form relations[10]. The last level is the **mapping level** which will map the logical design onto a specific RDBMS. This paper will focus on the first three levels.

The Model Level: Metabases. Metabases are dictionaries used to store and retrieve objects by name. Metabases are also used to catalogue objects by type. This will serve several purposes. It will aid in deriving information about objects. Examples will be given in the next section. It will facilitate the design process since once the name of the object is entered into a metabase it need not be reentered elsewhere, since it can thereafter be selected using a pointer. Also, metabases can be used to browse through the design.

In creating a design, the designer may have only partially formulated notions about the application domain. Initially, this might simply involve identifying things in the application domain. For example, 'employee', 'department', 'captain', 'ship', 'name', 'date', and 'end-of-week-sales-report' may be items of interest. These items will initially be stored as objects in a metabase called the **Generic metabase.** Eventually the designer will take some of these items and refine their meaning so that more of their semantics can be

captured. Such items will migrate to other metabases. Items in the Generic metabase which are refined by specifying their attributes or properties will be moved to the **Entity metabase.** Naturally, items can be entered directly into the Entity metabase without having to belong to the Generic metabase first. Ultimately, the Generic metabase will consist of objects that are atomic in that they have simple structures such as 'name', 'date', 'number'. Other metabases are the **Query metabase** for objects which are queries, a **Relations metabase** for normalized relations, a **View metabase** for views, a **Dependency metabase** for functional and multivalued dependencies, and a **Datatypes metabase** for various datatypes including user defined data types. In this paper we will concentrate on the Generic, Entity, and Query metabases.

The Model Level Language: Objects and Messages. In Smalltalk each object has its own data or local memory called **instance variables.** The only way to modify or retrieve the values of the instance variables of an object is to send the object a message it understands. For instance, if "card" is an object which has instance variables of "suit" and "rank", then sending "card" a message as in

card suit: 'diamond' rank: '9'.

modifies its suit and rank. Likewise, sending "card" a message as in

card rank

retrieves its rank. Each such message is implemented in a procedure called a **method,** which is part of the protocol for a given class of objects. Each method in turn consists of objects and messages sent to them. (Note that the same name can be used for instance variables and for messages and the names of the messages here are of our own choosing.)

From the point of view of creating a design, definition, and manipulation language, which we will often simply refer to as a design language when we mean all three, objects and messages provide a very flexible approach. For one thing, the design language does not have to be parsed and compiled. Any valid message can be sent to any existing object. In a sense there is minimal formal structure or syntax for the design language. One is not confronted with filling in all of the required slots, as is the case with standard data definition languages, in order to compile the schema. A design language based on objects and messages allows working with incomplete stages in the design and compiling its current state. Furthermore, the design language can be extended by defining new messages, without concern for an underlying parser and compiler. The object-message paradigm facilitates all of this

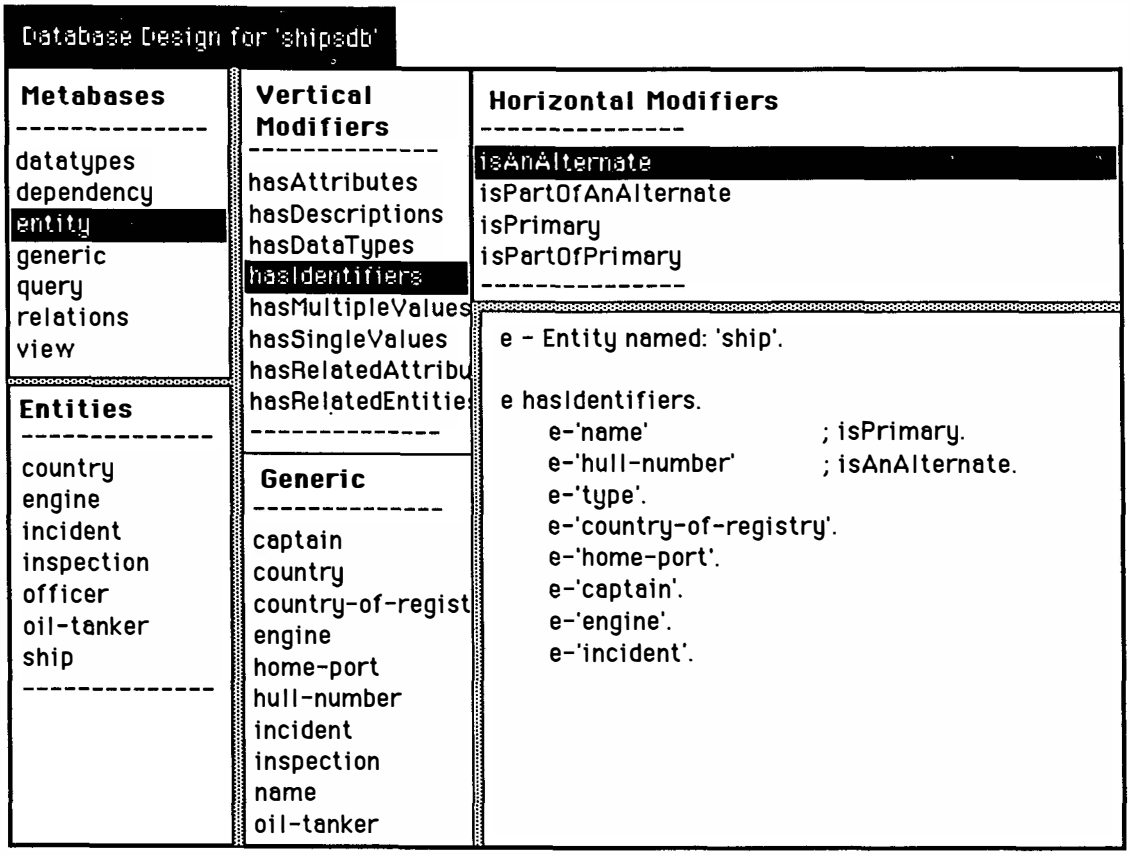

Figure 1.

because in some sense, the language is self-compiling in that new messages can be added by creating new methods for them. Moreover, parts of the design specification which are not completed do not affect those which are, since the missing parts only reflect messages which have not been sent, while they do not invalidate messages which have been. Dealing with incomplete information will be the responsibility of the production system rules.

The use of the object-message paradigm for the design language produces an essentially text based design language. A number of current design systems[11,12,13] are graphics based, using geometric figures to represent entities, relationships, attributes and even queries[11]. Some feel that text based languages tend to be formal and undesirable[12]. However, it appears that in graphics based systems handling a large number of entities and specifying queries graphically remain problems. They serve well as long as the complexity of the graphical information is minimized. Supporting complexity by graphical means would require powerful scaling techniques. Also, it is not clear how readily the graphics can be extended to support extensions to the design system since the tools typically have a fixed set of primitives with no means of easily defining additional primitives to support enhanced functionality of the system.

It is interesting to note on the other hand that the Smalltalk-80 System is basically textual even though it supports quite complex interrelationships of classes, methods, and messages. Some systems support both[14]. We adopt an approach similar to Smalltalk's in that our design language is text based, apart from the windowing interface. Naturally, we intend to incorporate some graphical aids through use of the high level graphical primitives provided in Smalltalk.

The Model Level Language: Modifiers and Layers. In creating a design language using objects and messages we have goals which appear to be in conflict. One goal is to allow sufficient expressiveness in the language to capture the manifold characteristics of the various database design objects. Another is to make the language sufficiently uniform that it can easily be used to express conditions of rules used in the production level. Also, we wish to have a single language for data definition and manipulation. In order to achieve these goals we restrict our language to two message types which we call **vertical** and **horizontal modifiers.** This terminology is derived in part from the physical structure of the textual layout. (Designers who wish to use ODDESSY as is will not need to learn about its underlying object-message framework or language. Those who wish to modify ODDESSY to suit their own purposes will need learn some elements of the Smalltalk-80 language and environment.)

A vertical modifier is a message sent to an object which sets the context for defining or refining the meaning of the object in terms of other objects.

A horizontal modifier refines the meaning of individual objects aggregated under a vertical modifier.

In Figure 2, the first line creates an instance of the class Entity, named 'ship', and identifies it with the temporary variable "e". The message "hasAttributes" is a vertical modifier which refines the meaning of 'ship' by setting the context for specifying other objects which will serve as attributes of 'ship'. These are specified in the messages following the "hasAttributes" message. The attributes of the entity are created by sending the entity a message of the form "- aString", where "-" is the message selector and aString is the name of the attribute object. These are shown in the "vertical" aggregation of attributes in Figure 2.

```
e <- Entity named: 'ship'.

e hasAttributes.

   e-'name'.
   e-'hull-number'.
   e-'type'.
   e-'country-of-registry'.
   e-'home-port'.
   e-'captain'.
   e-'engine'.
   e-'incident'.
```

Figure 2

To create the entries in Figure 2, the designer works with a window of the type shown in Figure 1. The vertical modifier "hasAttributes" can be selected from a subview and the names of the attributes can be entered in an entry window or selected from a current metabase in another subview using the mouse. When the entry in Figure 2 is accepted by the designer, an entity named 'ship' will be entered into the Entity metabase and each attribute object, 'name', 'hull-number', and so forth will be added to the Generic metabase if not currently in any other metabase. (By "accepted" it is meant that the designer compiles the Smalltalk code in Figure 2 by selecting an 'accept' from a pop-up menu.) It should be emphasized that attributes of 'ship' have the names 'ship-name', 'ship-hull-number', 'ship-type', and so forth. These are represented by the messages e-'name', e-'hull-number', etc.. Thus an attribute of an entity is determined by two objects, one is the entity itself, such as 'ship', and the other is the attribute object, such as 'hull-number'. Giving independent identity to an attribute object used to form an attribute of an entity facilitates the formulation of rules to distinguish cases when attributes of different entities might be the same, e.g., 'oil-tanker-hull-number' is the same as 'ship-hull-number', or different, e.g., 'department-number' is not the same as 'employee-number'.

This has the flavor of that of the Semantic Data Model in that entities in SDM are allowed to be attributes of other entities and the designer is free to list for each entity or class all of the properties which distinguish that class. This eliminates the need to decide which objects are entities and which are attributes or decide where the proper placement of an attribute must be, as in the ER Model. It also tends to make the subsequent stages of the design process more of a refinement of the earlier stages rather than a reformulation. However, unlike the SDM, our system does not require the designer to specify classes and subclasses for each entity and it does not require any other conditions on the class or its member attributes in order to accept a new entity. Classes in ODDESSY will be implicit as discussed in a subsequent

section. Also, because classes do not have to be specified, the designer is free to follow a more traditional approach typically used with the ER Model and may distinguish between attributes and entities if s/he so desires, as illustrated in a subsequent example.

A **horizontal modifier** is a message sent to an attribute of an entity to refine its meaning in the context of some vertical modifier. Figure 3 illustrates the use of the horizontal modifiers "isPrimary" and "isAnAlternate" in the context of the vertical modifier "hasIdentifiers". (Attributes which are not modified by a horizontal modifier are still listed to show the full entity. In some cases we will surpress the display of attributes without horizontal modifiers.) Figure 1 illustrates the case when two attributes form part of an alternate identifier or key for an entity.

```
e <- Entity named: 'ship'.

e hasIdentifiers.

   e-'name'                ; isPrimary.
   e-'hull-number'         ; isAnAlternate.
   e-'type'.
   e-'country-of-registry'.
   e-'home-port'.
   e-'captain'.
   e-'engine'.
   e-'incident'.
```

Figure 3

The combination of a vertical modifier and its set of horizontal modifiers is called a **layer** because for each object, such as an entity, the object is the sum of its layers. It can be examined and extended through them. Some additional layers needed for specifying entities are shown in Figure 4. These will be discussed next. Figure 5 illustrates the typical use of the layer "hasMultipleValues". A ship can have between 2 and 10 engines and be involved in any number of incidents. This is similar in style to the SDM. However, the designer is not forced to adopt this approach and can use a more traditional one similar to that used in the ER Model, or can even mix styles. For example, in Figure 6(a), the first instance, 'products', 'warehouses', and 'inventory-items', have been created using the ER Model approach in that attributes are treated as atomic and m:n relationships are expressed through the third entity, 'inventory-item', which relates products and where they are warehoused. In Figure 6(b) the designer has listed all properties of products as s/he identifies them. Then using the vertical modifier "hasMultipleValues" s/he refines the meaning of product by specifying, using the horizontal modifiers, that 'warehouse', 'quantity-onhand', and 'bckord-quantity' are multivalued and that the values of the latter two depend on that of the first.

The default is that all attributes are single valued until specified as multivalued. However, to indicate that an 'employee' belongs to exactly one 'department' the layer "hasSingleValues" "exactlyOne" can be used.

Vertical modifier	**Horizontal modifiers**
hasAttributes	
hasIdentifiers	**isPrimary**
	isPartOfPrimary
	isAnAlternate
	isPartOfAnAlternate: anInteger
hasMultipleValues	**atLeast:** anInteger **upTo:** anInteger (or **'any'**)
	atLeast: anInteger **upTo:** anInteger (or **"any"**) **forEach:** anAttributeName
hasSingleValues	**exactlyOne.**
	upToOne.
hasRelatedEntities	**isa:** anEntityName
hasRelatedAttributes	**sameAs:** anAttributeName **ofEntity:** anEntityName
hasDescriptions: aString	**description:** aString
hasDataTypes	**character:** anInteger
	integer: anInteger
	boolean
	type: aUserDefinedType

Figure 4

```
e <- Entity named: 'ship'.

e hasMultipleValues.
   e-'name'.
   e-'hull-number'.
   e-'type'.
   e-'country-of-registry'.
   e-'home-port'.
   e-'captain'      .
   e-'engine'       atLeast: 2 ; upTo: 10.
   e-'incident'     atLeast: 0 ; upTo: 'any'.
```

Figure 5

Figure 7 illustrates the use of the layer "hasRelatedEntities". One fundamental element of the SDM is the value class of an attribute. Each member attribute of a class in the SDM takes its value in class. For example, for classes named EMPLOYEE and DEPARTMENT in SDM if Employee-

department is a member attribute of EMPLOYEE, then the values class of Employee-department is the class DEPARTMENT. In ODDESSY this would be represented as shown in Figure 7(a) where 'employee-department' is specified as a 'department' and 'employee-manager' is declared as an 'employee'. It is not required that 'department' be an entity at the time that 'employee-department' is specified as a 'department'. Entity attributes can also be specified indirectly as entities, as illustrated in the layer to be discussed next.

```
e <- Entity named: 'product'.

e hasAttributes.
  e-'number'.
  e-'description'.
  e-'price'.
  e-'shipping-weight'
  e-'quantity-in-case'
  e-'stocking-class'

e <- Entity named: 'warehouse'.

e hasAttributes.
  e-'code'.
  e-'city'.
  e-'type'.

e <- Entity named: 'inventory-item'.

e hasAttributes.
  e-'number'.
  e-'code'.
  e-'quantity-onhand'.
  e-'bckord-quantity'.
```

(a)

```
e <- Entity named: 'product'.

e hasMultipleValues.
  e-'number'.
  e-'description'.
  e-'price'.
  e-'warehouse'          atLeast: 1 ; upTo: 5
  e-'quantity-onhand'    atLeast: 1 ; upTo: 1
                         forEach: 'warehouse'.
  e-'shipping-weight'
  e-'quantity-in-case'.
  e-'stocking-class'.
  e-'bckord-quantity'    atLeast: 1 upTo: 1
                         forEach: 'warehouse'.
```

(b)

Figure 6

The layer "hasRelatedAttributes" is used to indicate that attributes of one or more entities are the same. If the layer in Figure 7(b) is added to the layers in Figure 6(a), then 'product-number' and 'inventory-item-number' are declared to be the same as are 'inventory-item-code' and 'warehouse-code'. The "hasDescriptions" layer is used to enter textual descriptions of entities and attributes as seen in Figure 8.

```
e <- Entity named: 'employee'.

e hasRelatedEntities.

  e-'number'.
  e-'name'.
  e-'address'.
  e-'departement' isa: 'department'.
  e-'manager'     isa: 'employee'.
```

(a)

```
e <- Entity named: 'inventory-item'

e hasRelatedAttributes.

  e-'number' sameAs:'number'; ofEntity:
                              'product'.
  e-'code'   sameAs:'code';   ofEntity:
                              'warehouse'.

  e-'quantity-onhand'.
  e-'bckord-quantity'.
```

(b)

Figure 7

```
e <- Entity named: 'ship'.

e hasDescriptions: 'all ships with potentially dangerous
                    cargoes that may enter U.S. waters'.

  e-'name'.
  e-'hull-number'.
  e-'type'          description: 'the kind of ship such as
                                  merchant ship'.
  e-'country-of-registry'.
  e-'home-port'.
  e-'captain'.
  e-'engine'.
  e-'incident'.
```

Figure 8

The last layer in Figure 4, "hasDataTypes" is used to specify the data types of the attributes of an entity, Figure 9. Note that for attributes which have been modified with 'isa' or 'sameAs' modifiers, their data types are derived in the first case by the data type of the primary identifier of the related entity and for the second case by the data type of the related attribute. Figure 9(b) illustrates that entities may also be data types. For example, if an entity 'coordinate' has been defined with attributes 'x-coordinate' and 'y-coordinate', then the data type of 'coordinates' is a 'coordinate'. This also could have been handled by specifying that 'coordinates' is multivalued

and "isa: 'coordinate'". The only difference between these is that if the data type is not declared, then the data type is taken to be that of the primary key rather than the entire attribute itself. Data types can also be specified by the user. These will reside in the DataTypes metabase with messages and methods developed to support them.

```
e <- Entity named: 'ship'.

e hasDataTypes.
   e-'name'                  character: 20.
   e-'hull-number'           integer: 10.
   e-'type'                  character: 2.
   e-'country-of-registry'.
   e-'home-port'             character: 3.
   e-'captain'.
   e-'engine'.
   e-'incident'.
```

(a)

```
e <- Entity named: 'pad'.

e hasDataTypes.
   e-'serial-number'  character: 10.
   e-'coordinates'    type:'coordinate'.
   e-'size'           type:'derive-pad-size'
```

(b)

Figure 9

Queries in ODDESSY. Our objective in creating a query language is to provide the designer with a language in which to express queries to allow performance analysis at the logical level including computations of transport volumes[15]. To support this, layers would have to be added to entities which specify their cardinality and selectivities on their attributes. We will not discuss these layers. To make the query language essentially the same as the design language new layers are introduced to express the fundamental constituents of a query, i.e., selections, projections, and joins.

Since complex queries are easier to express by nesting queries or decomposing queries into query fragments, queries can be used in certain horizontal modifiers. This allows expression of queries requiring transitive closures. We will examine some of the fundamental layers for expressing queries. A full complement of layers for queries remains to be developed so we present examples for purposes of illustration. For convenience, in the examples shown here, queries will not be displayed in terms of their layers, but will be framed as new layers of existing entities. To illustrate these layers, the entities in Figure 10 will be used. They are expressed in a normalized form, except for 'subpart' which is multivalued, with the relationship of supplier-part expressed as the third relation. Figure 11 illustrates various selection layers. In Figure 11(a) all 'London' suppliers are selected. In 11(b) all suppliers from 'Paris' or 'Rome' are selected and in 11(c) the selection is from the User. In [11], queries are expressed as subclasses. One can take the dual point of view and define subclasses as queries. Thus the queries and Figure 11(a) and (b) can be considered as defining subclasses. Figure 12 shows the projection of fields from a query in a projections layer.

```
e <- Entity named: 'part'.

e hasAttributes.
   e-'number'.
   e-'price'.
   e-'description'.
   e-'subpart'.

e <- Entity named: 'supplier'.

e hasAttributes.
   e-'number'.
   e-'city'.
   e-'status'.

e <- Entity named: 'supplier-part'.

e hasAttributes.
   e-'supplier-number'.
   e-'part-number'.
   e-'quantity'.
```

Figure 10

```
e <- Entity named: 'supplier'.

e hasSelections        inQuery:'London-suppliers'.
   e-'number'.
   e-'city'            equals: 'London'.
   e-'status'.
```

(a)

```
e hasSelections        inQuery:'Paris-Rome-suppliers'.
   e-'number'.
   e-'city'            equals: 'Paris';
                       orEquals: 'Rome'.
   e-'status'.
```

(b)

```
e hasSelections        inQuery: 'supplier-by-city'.
   e-'number'.
   e-'city'            ; fromUser.
   e-'status'.
```

(c)

Figure 11

```
e hasOutput            inQuery: 'supplier-by-city'.
   e-'number'                    ; output.
   e-'city'            ; output.
   e-'status'.
```

Figure 12

Figure 13 illustrates queries involving joins. The selection of a supplier comes from a user. The parts are those that match the supplier specified through the supplier-part entity and includes 'part-number' and 'part-price' since they are output. These queries can also be expressed if the entities are not normalized and are similar in form. As our last example, in Figure 14 we illustrate the query which locates all 'subparts' of a given 'part' and uses the output from another query, in this case its own output, to join tuples. Not shown is that 'subpart' has been specified as being a 'part'. The user enters a 'part-number'. Any tuple with this 'part-number' will have as output a 'subpart' which becomes the input to the query.

```
e <-  Entity named: 'supplier'.

e hasSelections      inQuery:'all-parts-for-supplier'.
  e-'number'              ; fromUser.
  e-'city'.
  e-'status'.

e hasMatches         inQuery:'all-parts-for-supplier'.
  e-'number'         equals: 'supplier-number';
                     inEntity: 'supplier-part'.
  e-'city'.
  e-'status'.

e <-  Entity named: 'part'.

e hasMatches         inQuery:'all-parts-for-supplier'.
  e-'number'         equals: 'part-number';
                     inEntity: 'supplier-part'.
  e-'price'.
  e-'description'.
  e-'subpart'.

e hasOutPut          inQuery:'all-parts-for-supplier'.
  e-'number'              ; output.
  e-'price'          ; output.
  e-'description'.
  e-'subpart'.
```

Figure 13

```
e <-  Entity named: 'part'.

e  hasSelections.
            inQuery:'all-subparts'.
  e-'number' ;        fromUser;
                      orEquals: 'subpart';
                      fromQuery: 'all-subparts'.
  e-'price'.
  e-'description'.
  e-'subpart'      .

e hasOutput
            inQuery:'all-subparts'.
  e-'number'       .
  e-'price'.
  e-'description'.
  e-'subpart'              ; output.
```

Figure 14

The Object Level. The objects, which receive the messages in the model level and which are specified by layers, reside at the object level in dictionaries called metabases. Each object consists of two basic constituents which are also objects. The first constituent, called a **surrogate**[16], serves as an identifier of the object and is shareable with other objects. Surrogates have the same structure for all objects in the design system and are a subclass of the general class of Object. A surrogate has instance variables for the object's name, it's metatype, i.e., the metabase to which it belongs, and it's database, for purposes of database integration.

Whenever the messages in a layer are sent to an object, such as the layers illustrated in Figures 3-9, the effect is to store the layer in the allLayers dictionary of the object. Because layers are uniform and layered objects have a uniform structure the following benefits are realized: (1) creation of new modifiers is straightforward in that the code for thier methods is essentially boiler-plate; (2) retrieval of information from the layers is uniform; (3) the conditions of the rules required in the production level have a uniform structure; and (4) objects can migrate from one metabase to another, such as a Generic object becoming an Entity.

The second constituent of an object is called the **substance** which consists of the following instance variables:

theLayer contains the name of the vertical modifier corresponding to the current layer.

theCurrentAttribute contains an attribute surrogate currently of interest.

theAttributes is an ordered collection of attribute surrogates of the object.

attributeOf is a set of surrogates for which this object is an attribute.

modifierIn is a set of entity surrogates for which this object appears in a horizontal modifier.

allLayers is a dictionary indexed by the vertical modifiers of this object. At each vertical modifier, the storage structure parallels the structure of the layer in that it contains the attribute surrogates and their horizontal modifiers as seen in Figures 3-9.

The Production Level. The model and object levels support the interaction and storage of information about the current state of the design process. At the production level information is derived based on information captured in the model level. As the name of this level suggests it is used in a manner similar to a production system[17]. The production level is structured according to protocols[18] for carrying out specific tasks. Within the protocols are rules of the form

$$C_1 , C_2 , \ldots , C_n \rightarrow A$$

where the C_i are conditions and A is an action modifying the design database by modifying existing layers or creating new layers. At this stage of development there are no metarules for

rule selection or conflict resolution. The best way to handle rules in an object oriented system is currently being investigated. Our main thrust at this point in developing a prototype is to allow the designer to produce normalized relations based on a few layers for each entity. The normalization routines currently handle through third normal form[10]. We will illustrate some of the rules for creating functional dependencies from the layers after conditions are discussed in more detail.

The conditions reflect the structure of the layers in the specification and the actions create or modify layers. The standard "if - then -else" condition in Smalltalk has the form

(<conditions>) ifTrue: [<action block>]
ifFalse:[<action block>].

where the conditions are each enclosed in parentheses and separated by boolean connectors "&" and "|". A **not** following a condition negates it. (Only one of the messages ifTrue: and ifFalse: is required.) The basic constituent used in forming conditions has one of the following forms:

<object> <vertical modifier> at: <object>
<object> <vertical modifier> at: <object> is: <horizontal modifier>
<object> <vertical modifier> at: <object> for: <horizontal modifer>

The first two forms return boolean values and the third returns the value of the horizontal modifier for those modifiers which take arguments. Using names rather than the objects themselves for purposes of illustration, we give the following examples:

'ship' hasMultipleValues at: 'engine'.

'ship' hasIdentifiers at: 'name' is: #isPrimary.

'ship' hasMultipleValues at: 'engine' for: #atLeast.

In the first example, the condition checks if 'engine' is a multivalued attribute of 'ship'. In the second it checks if 'name' is a primary identifier of 'ship'. In the third, a value is returned which specifies the constraint on the minimum number of engines a ship must have. The following is an example of conditions in a rule used to create functional dependencies:

(anEntity hasIdentifier at: anAttribute is: #isPrimary) not
&
(anEntity hasIdentifier at: anAttribute is: #isPartOfPrimary) not
&
((anEntity hasSingleValues at: anAttribute) |
(anEntity hasMultipleValues at: anAttribute) not)
&
(anEntity hasRelatedAttributes at: anAttribute)

These conditions determine if an attribute of an entity is single valued, is not part of the primary identifier, and has related attributes in another entity. If so, functional dependencies are created between the related attributtes.

The Production Level: Data Models and Derived Information. In one respect data modelling in ODDESSY is constrained by the fact that information is specified in layers. In another respect it is less constrained than other models in that additional layers can be added in order to capture more of the semantics of the application domain than might be supported by a single data modelling approach. It is the rules in the production level used in conjunction with the layers which dictates which model or models can be supported by ODDESSY. With just the few levels in Figure 4 much of the information captured by other models such as the SDM, the Structural Model, and the ER Model is explicitly or implicitly captured in ODDESSY. Some information, such as an attribute having multiple values, is explicitly represented in the layers. Other information can be derived from the layers by rules. Implicit information need not remain so. Additional layers can be derived from existing layers. The benefit comes from minimizing what the designer has to specify using a minimum number of layers. The remainder of the specification is derived. The remainder of this section illustrates this.

Some default values are assumed when an entity is defined in the layer "hasAttributes". All attributes are initially assumed to be single valued. Consequently, a message such as

(anEntity hasSingleValues at: anAttribute)

will be true if multiple values have not been specified for the attribute.

When an object A, such as 'department', becomes an attribute of an entity E, such as 'employee', and if A is in the Entity metabase, the system will automatically generate an 'isa:' modifier, such as "isa: 'department', in Figure 7(a), subject to confirmation by the designer. If an attribute A is not an entity at the time entity E is accepted, but subsequently becomes an entity, then the layer "hasRelatedEntities" of E will be updated to reflect this fact.

A feature in the SDM is that member attributes may be specified as "may not be null". In ODDESSY, this is implicit for attributes which are identifiers for entities. Also, any single valued attribute with the horizontal modifier "exactlyOne" is presumed to be non-null.

In SDM an attribute can be specified as matching an attribute in another class. For example, for the class SHIPS the member attribute Captain is declared to

match: Officer of ASSIGNMENTS on Ship.

In ODDESSY this can be deduced by a rule which states that if entity E has attribute A which "isa:" attribute B of entity F, and if entity E is also an attribute of entity F, then then the value of E as an attribute matches the value of E as an entity.

In SDM a member attribute may be specified as the inverse of an attribute of another entity. For example, if SHIP has a member attribute 'Country-of-registry' and if COUNTRY has a member attribute 'Ships-registered-here' then these attributes

can be declared as inverses of one another. That is, if you take all of the instances of ships which have the same country of registry, then this set will match the values of the attribute of ships registered in that country for COUNTRY, and vice versa. Again in ODDESSY, this will be implicit in that if two entities each have each other as attributes, then an inverse relationship is presumed. For example, if 'employee' and 'department' are entities and attributes of each other, it is presumed that the set of all employees for a department is the inverse of the set of all employees who have that department value for their attribute department.

Perhaps the single most important feature of SDM is the support of classes and subclasses. Entities in ODDESSY correspond to class definitions in SDM. Some subclasses in ODDESSY are defined implicitly. For example, if the primary identifier of one entity is the same as the primary identifier of another entity, then the former entity is presumed to be a subclass of the latter. For example, if 'employee-number' is the primary identifier of the entity 'employee' and if 'manager' is an entity with attribute 'manager-number' specified as the "sameAs" in the layer "hasRelatedAttributes" and both are primary identifiers, then 'manager' will be presumed to be a subclass of 'employee'. Likewise if attribute A of an entity E is also an entity and is the primary key of E, then E will be presumed to be a subclass of A.

In the ER Model entities and relationships among entities are the principal constructs. A relationship is like an entity in that it may have attributes, but a relationship's existence depends on the existence of other entities. In the Structural Model, relations and connections play analogous roles, except that relations are assumed to be normalized into Boyce-Codd normal form. Also, connections carry additional semantics regarding the existence requirements on entities. In ODDESSY, relationships are implicit and are derived. For example, if faculty have attributes of 'course-TA', 'thesis-advisee', and 'graduate-reader' which are all specified as "isa: 'graduate-student'", then each of these represent a relationship between faculty and graduate students. For these to have attributes, they can either be declared as entities or can be specified as attributes of 'faculty' using the modifier "atLeast: upTo: forEach: 'course-TA' " and so forth.

The Structural and ER models capture information on cardinality of relationships which are 1-1, 1-n, and m-n. In ODDESSY, these may be implicit or explicit. If an attribute A of an entity E1 is not multivalued and A is an entity E2, then this represents either a 1-1 or 1-n relationship. If in addition E2 has E1 as an attribute which is not multivalued, then it is 1-1. On the other hand if E1 is a multivalued attribute of E2 or E1 is not an attribute of E2, then it is presumed to be 1-n. For example, if 'department' is a single valued attribute of 'employee' and 'employee' is a multivalued attribute of 'department' or is not an attribute, then there is a 1-n relationship between 'department' and 'employee'. The same would hold if instead of A being the entity E2, A is the primary identifier of E2. The m-n case holds when entities E1 and E2 are multivalued attributes of each other or if E3 has primary keys consisting of the keys of E1 and E2 such as in Figure 6(a) where 'inventory-item' is an m-n relationship between 'product' and 'warehouse'.

Dependency constraints concern whether entities exist independently or not and are classified as partial, total, and no dependency. For example, if every employee must belong to a department and all departments must have employees, the dependency is total. If departments can exist without employees, but not conversely, then it is partial from employee to department. Thus a department cannot be deleted while it has employees. If both can exist independently, then there is no dependency. The horizontal modifiers for the layers "hasSingleValues" and "hasMultipleValues" can be used to infer dependencies. For example, a partial dependency from E2 to E1 arises when "atLeast:1" is specified for E1 as an attribute of E2. A total dependency arises when the reverse also holds.

In the Structural Model, various types of connections are defined between relations. Connections contain information needed for inserting and deleting tuples and are useful in integrating databases[19]. For example, a reference connection prohibits deletion of a department which is referenced by some employee, but requires an employee to belong to a department. This can be captured by specifying that department is exactlyOne as an attribute of employee. If this were not specified it would be presumed that employees did not have to belong to departments and an ownership connection existed between a department and its employees.

We also briefly mention that in many situations designs might not be complete when the rule base is used. Thus when a particular action is required the prototype containing the rule should either use default values and rely on the information present or should prompt for the missing information. This can provide the same kind of flexibility found in object-oriented programming or database systems where objects need not be completely specified in prototyping a system[9].

Advantages of the Object Oriented Approach. Some advantages of using objects and messages for the design language have already been discussed. There are other advantages in using an object oriented system such as Smalltalk-80 which are not immediately available in standard programming languages. Smalltalk-80 has a large number of predefined classes and messages including classes for many types of data structures and classes supporting interactive computing. One author[20] has called this concept a "software Integrated Circuit", i.e, a unit (class) can be delivered to a system builder "off the shelf" with a specified structure and functionality. Objects in a class can be created and used directly, or its functionality can be passed on and modified through subclasses. Software IC's should facilitate add-on interfaces and functionality to existing systems without complete rewrites and encourage protyping new systems. This is important for rapid prototyping a system in order to study its characteristics, as certainly will be necessary, because any complex database design system will require considerable experimentation and modification. Also the very sophisticated modeless environment of Smalltalk-80 with its extensive cross referencing of objects and messages and the modularity enforced by working with objects and messages facilitates, even encourages, making changes and experimenting.

Conclusion. We have presented an experimental prototype object oriented database design system. Several new concepts have been introduced in working within an object oriented system. These include the notions of metabases, vertical and horizontal modifiers, layers, and an architecture which is tailored to the development of an object oriented production system.

References.

[1] Database Engineering special issue on Object-Oriented Systems, vol. 8, no. 4, 1985.

[2] A. Goldberg and D Robson: "Smalltalk-80: The Language and its Implementation"; Addison-Wesley Pub., Reading, Mass, 1983.

[3] J. Ong, D. Fogg, and M. Stonebraker: "Implementation of Data Abstraction in the Relational Database System INGRES"; SIGMOD Record, vol. 14, no. 1, Mar. 1984.

[4] M. Stonebraker and L.A. Rowe: "The Design of POSTGRES"; Dept. of EE and CS, UC Berkeley, 94720.

[5] G. Wiederhold and R. Cherubini: "KSYS: An Architecture for Knowledge Bases"; Dept. of CS, Stanford University.

[6] M. Hammer and D. McLeod: "Date Description with SDM: A Semantic Data Model"; ACM TODS, Vol. 6, No. 3, Sept. 1981, pp. 351-386.

[7] P. Chen: "The Entity-Relationship Model:Towards a Unified View of Data"; ACM TODS, Vol. 1, No. 1, Mar. 1976, pp. 9-36.

[8] G. Wiederhold and R. ElMasri: "The Structural Model for Database Design"; Proc. Int. Conf. Entity-Relationship Approach, Los Angeles,Dec.,1979, pp. 247-267.

[9] D. Maier, J. Stein, A. Otis, and A. Purdy: "Development of an Object-Oriented DBMS"; Proceedings of the First Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, September 1986, pp. 472-482.

[10] J. Diederich and J. Milton: "New Methods and Fast Algorithms for Database Normalization"; submitted for publication.

[11] D. Bryce and R. Hull: "SNAP: A Graphics-based Schema Manager"; Proc. Int. Conf. Data Eng., Los Angeles, Feb. 1986, pp. 151-164.

[12] K.J. Goldman, S.A. Goldman, P.C. Kanellakis, and S.B. Zdonik: "ISIS: Interface for a semantic information System"; Proc. ACM SIGMOD Int. Conf. on the Management of Data, Austin, Texas, 1985, pp. 328-324.

[13] D. Reiner, M. Brodie, G. Brown, M. Friedell, D. Kramlich, J. Lehman, and A. Rosenthal: "The Database Design Evaluation Workbench (DDEW) Project at CCA"; Database Engineering, vol. 7, no. 4, Dec. 1984, pp. 16-21.

[14] Orr, Ken: Database Research Seminar, Stanford University, Spring, 1986.

[15] T.J. Teorey and J.P. Fry: "Design of Database Structures"; Prentice-Hall, Englewood Cliffs, N.J., 07632, 1982.

[16] N. Derrett, W. Kent, and P. Lyngbaek: "Some Aspects of Operations in an Object-Oriented Database"; Database Engineering special issue on Object-Oriented Systems, vol.8, no.4,1985, pp.66-74.

[17] J. McDermott: "R1: A Rule-based Configurer of Computer Systems"; Artificial Intelligence, vol. 19, Sep. 1982, pp. 39-88.

[18] J. S. Aikens: "Prototypical Knowledge for Expert Systems"; Artificial Intelligence 20, 1983, pp. 163-210.

[19] R. ElMasri and G. Wiederhold: "Data Model Integration Using the Structural Model"; Proc. ACM SIGMOD Conf., ACM, May 1979, pp. 319-326.

[20] Cox, Brad: " Object Oriented Programming"; Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.