Seoul Journal of Business
Volume 3, Number 1

Object-Oriented Development: Patterns, Interfaces, and Update Semantics*

Sangkyu Rho
Seoul National University
Salvatore T. March
University of Minnesota

Abstract

Object-Orientation has the potential to significantly improve information system development productivity through the reuse of code and design components. To achieve this potential, object patterns having similar update semantics must be recognized and standard interfaces developed for them. We propose six such patterns and demonstrate their benefits by developing standard interface templates for each. A developer can analyze the patterns in an object model and quickly configure interfaces and update methods for an information system using these templates. These templates are implemented as meta-classes in Smalltalk as part of SOODAS, a Semantic Object-Oriented Data Access System, which also supports Entity-Relationship semantics and provides a set-level query language.

I. Introduction

Object-Orientation has the potential to significantly improve information system development productivity through the reuse of code and design components. However, its touted benefits have not been fully realized due to difficulties in building reusable objects, determining which objects can be

* Professor Rho was partially supported by the Institute of Management Research, Seoul National University.

reused, and learning how to reuse those objects [Curtis, 1989; Izakowitz and Kaufman, 1995].

Design patterns are a promising technique for achieving a high level of design and code reuse [Budinsky et al., 1996; Gamma et al., 1995; Pree, 1995; Schmidt et al., 1996]. A design pattern systematically describes a time-tested solution to a recurring design problem. A design pattern also describes the applicability, trade-offs, and consequences of the solution. It may also illustrate how to implement the solution in a programming language such as Smalltalk or C++. Therefore, it provides systems developers with a reference of proven design solutions and guidelines for implementing them.

We apply the concept of design patterns to the development of user interfaces and update programs for a conceptual object model. Object patterns having similar update semantics are identified and standard interface templates developed. We propose six such patterns: Unconstrained Independent, Constrained Independent, Parent-Child, Full Intersection, Subtype, and Cyclic Constraint. Although not exhaustive, the use of these patterns facilitates the learning of object modeling and results in semantically richer object models. It also significantly reduce development efforts by facilitating the reuse of design and code components. We demonstrate their benefits by developing standard interface templates for each. A developer can analyze the patterns in an object model and quickly configure interfaces and update methods for an information system using these templates. These templates are implemented as Smalltalk meta-classes in SOODAS, a Semantic Object-Oriented Data Access System, which also supports ER semantics and provides a set-level query language [March and Rho, 1997].

The remainder of this paper is organized as follows. The next section discusses prior research on design patterns. The following section describes the six patterns using an example object model. The next sections present the meta-classes used to implement them and illustrate the use of these classes for developing update programs. The final section summarizes this research and presents directions for future research.

II. Prior Research

There has been increasing interests in the use of design patterns and

frameworks in object-oriented systems development as techniques to improve productivity. A design pattern systematically describes a general solution to a recurring design problem [Alexander, 1979; Alexander et al., 1977], while a framework is, "a set of cooperating classes that make up a reusable design for a specific class of software" [Gamma, et. al., 1995, p. 26].

Coad [1992] explores the concept of patterns in the context of object-oriented analysis and design. An object-oriented pattern is a building block for object-oriented development which consists of a set of classes and their relationships. Most of his patterns describe how a set of classes can be used to build conceptual object models when certain conditions hold.

Johnson [1992] applies patterns in documenting HotDraw, a framework for implementing various kinds of graphic editors. He describes, in an informal way, how to use patterns to describe a framework. The goal is to help users understand the framework and effectively apply the patterns it contains.

Gamma et al. [1993, 1995] propose design patterns as a mechanism to express object-oriented designs. Their patterns describe solutions to recurring design problems, the rationale behind each solution, hints regarding implementation, and code examples in C++/Smalltalk. They view these patterns as "reusable micro-architecture that contribute to an overall system architecture" and as "a common vocabulary for design." Schmidt [1995] describes his experience in applying these design patterns to the development of commercial communications software. He predicts that patterns will become integrated with frameworks to form "systems of patterns" which will ultimately form pattern language extensions to object oriented programming languages (OOPLs).

Budinsky et al. [1996] argues that the implementation of such design patterns requires significant effort because they must be programmed each time they are applied and because there may be many trade-offs to consider in their application. To address this problem they developed a tool that creates class declarations and definitions that implement the patterns identified by Gamma, et. al. [1993, 1995]. Their tool requires application specific information and choices for applicable design trade-offs.

In this research, we study conceptual object models to identify object patterns having similar update semantics and requiring similar interfaces. We develop a framework for implementing these patterns within SOODAS, a Smalltalk-based Semantic Object-Oriented Data Access System. Our framework is different from most of GUI application development tools such as Microsoft Access and Oracle Designer/2000 in that applications are created by subclassing a set of

meta-classes rather than by generating codes. The modification of applications is easier since applications are modified by overloading methods rather than by modifying generated codes. This also facilitates the evolutionary improvement of applications since the improvement of the meta-classes will be reflected in applications developed using the tool. The next section describes our object patterns. The following section describes our framework and illustrates its use.

Ⅲ. Object Patterns

Common among object representations, SOODAS [March and Rho, 1997] supports five basic modeling concepts: class (entity[1]), attribute, binary relationship, external identifier, and subclass (subtype). In object models using these concepts we identify six recurring patterns for developing user interfaces: Unconstrained Independent, Constrained Independent, Parent-Child, Full Intersection, Subtype, and Cyclic Constraint. As Coad [1992] points out, these patterns were identified by carefully observing many object-oriented development projects rather than by applying theories or logic. They were chosen simply because they occur frequently in object modeling. Although not exhaustive, these patterns can significantly reduce development efforts by facilitating the reuse of design and code components. For each pattern, we present an example, describing its update semantics and user interface.

1. Unconstrained Independent (Referent)

Pattern: This is a single class pattern where instances of the class exist (can be added and modified) independently of any other instances in the system. The class has only attributes (variables) in its identifier(s) and all of its relationships have maximum cardinality of 1 for itself and minimum cardinality 0 for the other class in the relationship. Its relationships may have minimum cardinality of 0 or 1 for itself and maximum cardinality of 1 or many for the other class. Instances of the class often serve as *referents* for one to many relationships (have minimum cardinality 1 for this class). If so, deletion actions must be specified.

Example: In Figure 1, Department is an Unconstrained Independent (Referent)

---

1) SOODAS supports Entity-Relationship semantics, hence we use the terms Entity and Class interchangeably.

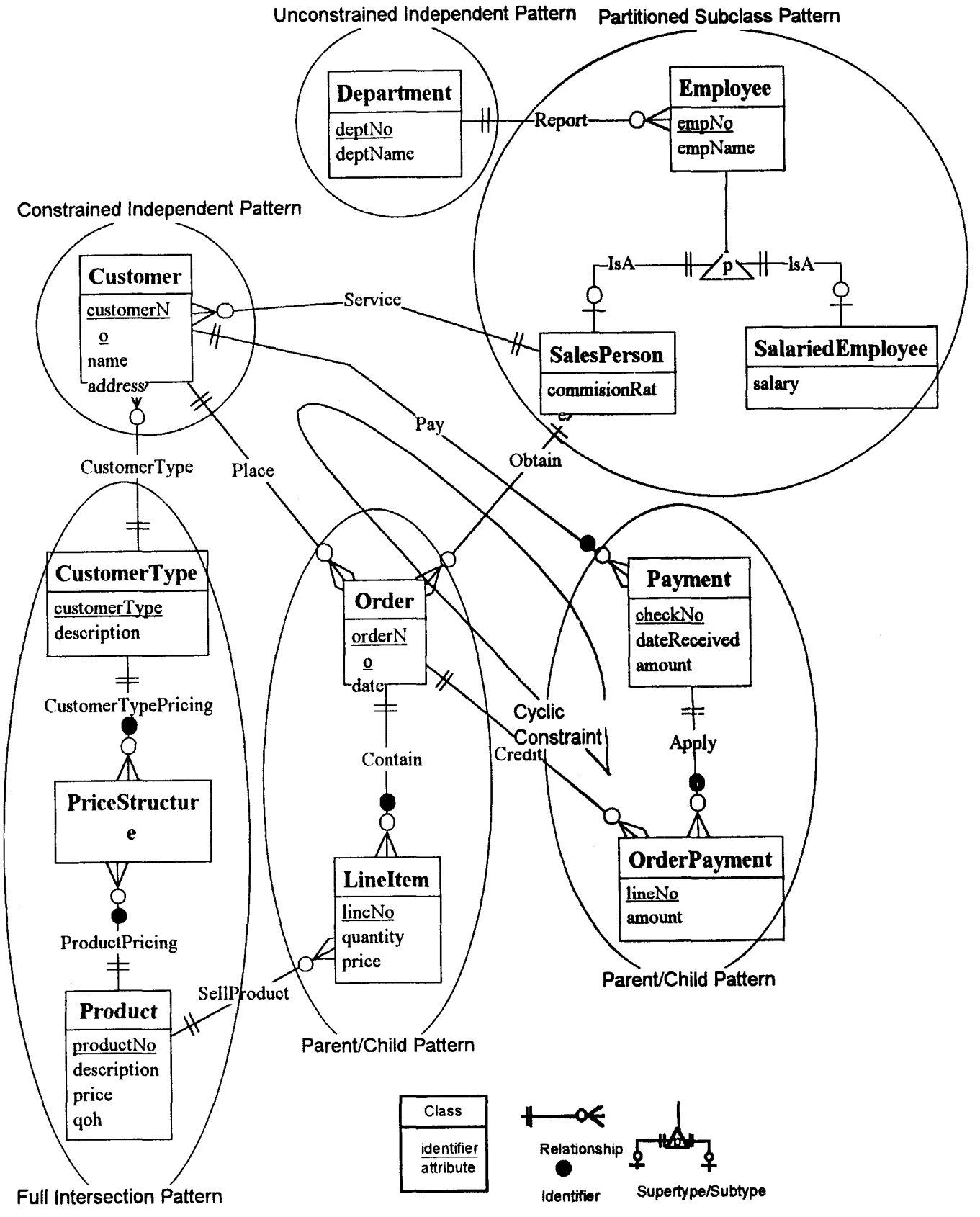


Figure 1. An Example Object Model with Patterns

class. It has only attributes (variables) in its identifier, deptNo (denoted by the fact that it is underlined). Its only relationship, Reports, has maximum cardinality 1 for itself and minimum cardinality 0 for Employee, the related class.

Update Semantics: Department instances are added and maintained independently of any other instances in the system. Its instances serve as referents for Employee instances in the Report relationship, that is, the minimum cardinality is 1 on Department. If related Employee instances exist then the deletion action must be specified as either (a) remove related instances (cascade delete) or (b) disallow.

Interface: Classes conforming to this pattern can be maintained using an interface such as that shown in Figure 2. The display window lists all instances in a default format (which can be changed as desired) and contains buttons to add, delete, find, and edit instances. The update window is used to add or edit instances.

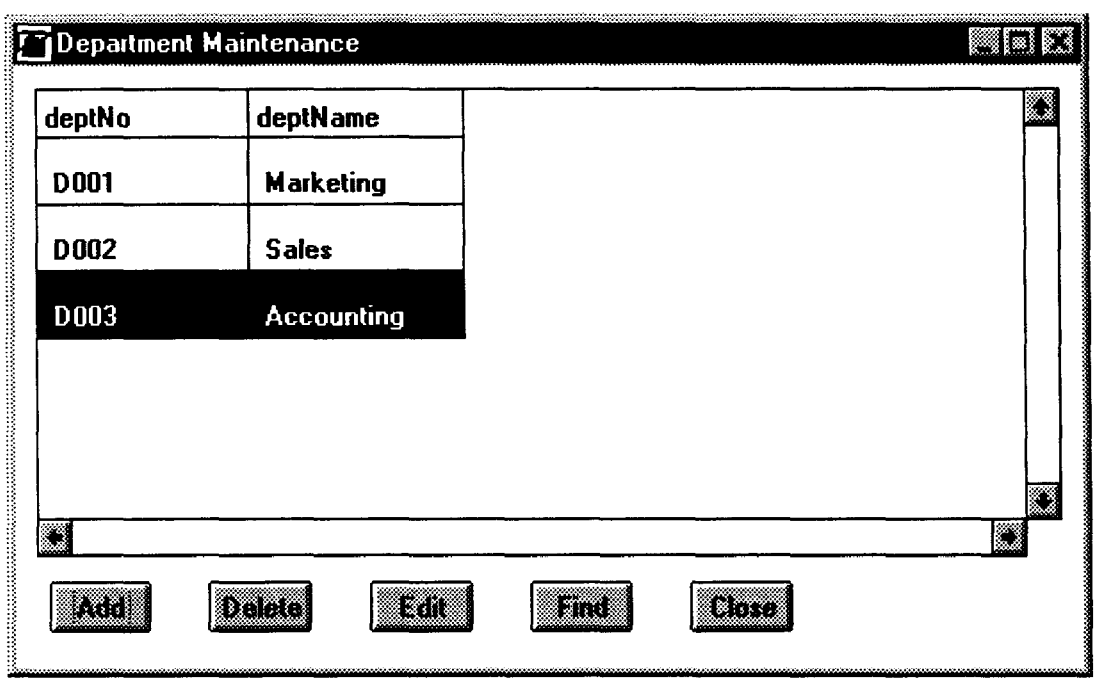
2. Constrained Independent (Referential)

Pattern: This is a single class pattern where instances of the class reference instances of at least one other class. The class has only attributes (variables) in its identifier(s) and has at least one relationship with minimum cardinality 0 for itself and a maximum cardinality 1 for the other class. If the minimum cardinality for the other is 1, then referential integrity constraint(s) exist -- instances of this class cannot exist without the referenced instance(s) of the other class(es).

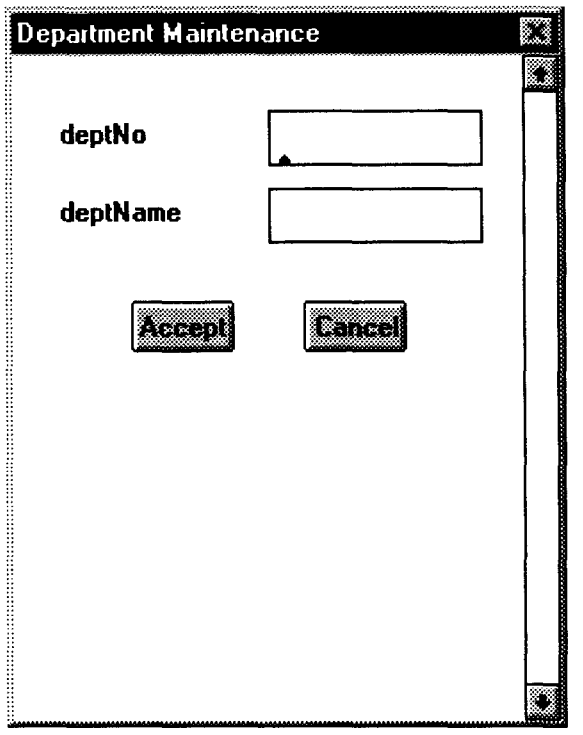
Example: In Figure 1, Customer is a Constrained Independent (Referential) object. It has only attributes (variables) in its identifier, customerNo. It participates in two relationships, Services and CustomerType, with minimum cardinality 0 for itself and 1 for SalesPerson and CustomerType, respectively. That is, Customer has referential integrity constraints with SalesPerson and CustomerType. A Customer instance cannot exist without a related SalesPerson instance and a related CustomerType instance. Customer participates in additional relationships, Pay and Place, for which it is the referent class.

Update Semantics: Insert or maintenance operations that violate referential integrity constraints are disallowed. Other update semantics are the same as those of the Unconstrained Independent pattern.

Interface: Classes conforming to this pattern can be maintained using a single object display window as in Figure 2.a., and a constrained update window such



a. Default Display Window for Department



b. Default Update Window for Department

Figure 2. Default Interface for Department (Unconstrained Independent Pattern)

as one shown in Figure 3. The update window includes a widget (e.g., combo box) for maintaining each many-one relationship. It displays the legal values for

the relationship using the familiar "foreign key" concept, i.e., identifiers of the class on the other side. Widgets maintaining relationships with referential integrity constraints must display identifiers from the related class and require the user to select one. Widgets maintaining relationships without referential integrity constraints include null for instances that do not participate in the relationship.

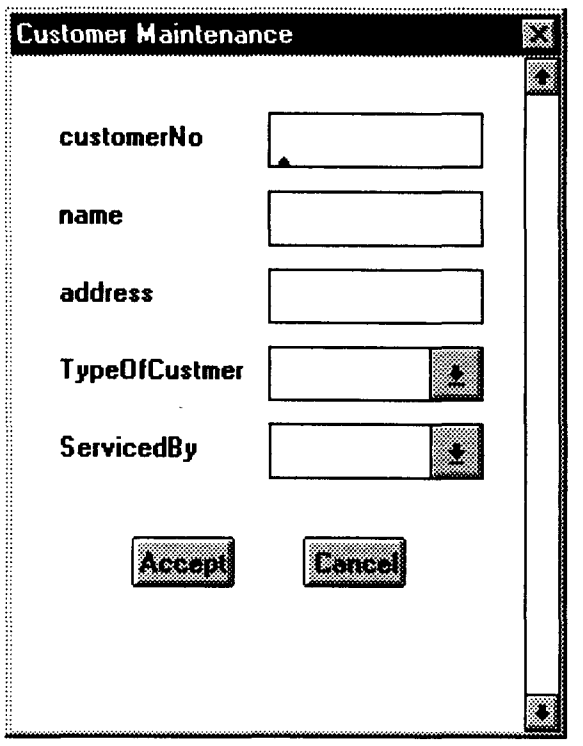


Figure 3. Default Update Window for Customer (Constrained Independent Pattern)

3. Parent-Child

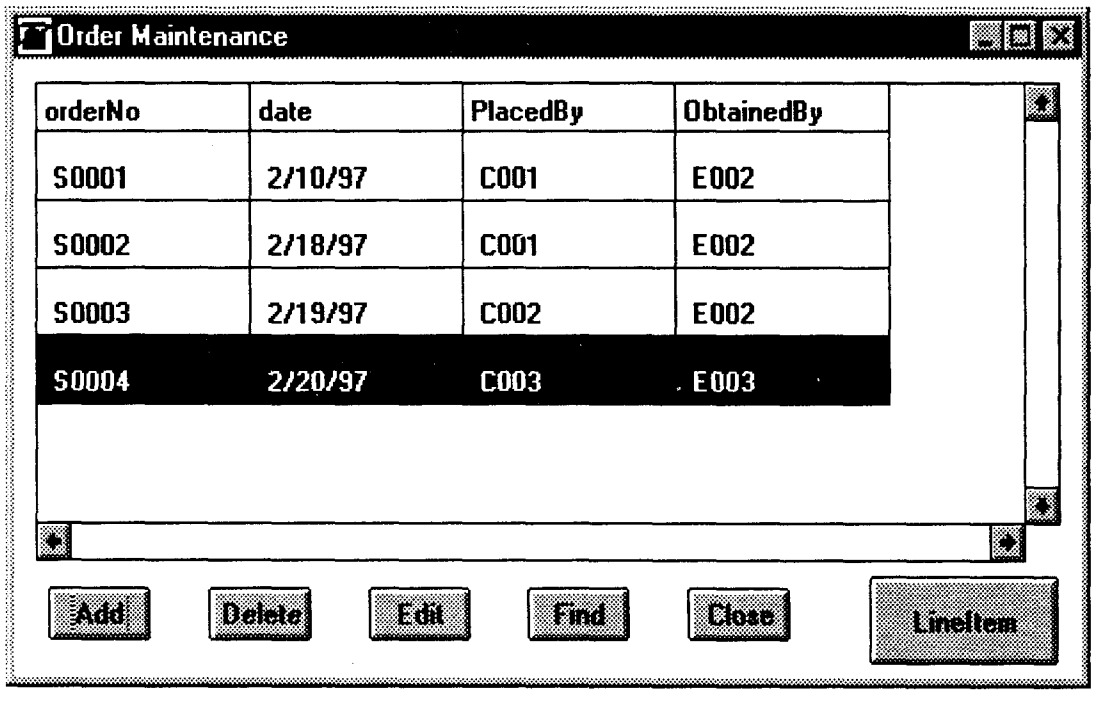
Pattern: This pattern involves two classes designated Parent and Child. The Parent has a one to many relationship with the Child with a minimum and maximum cardinality of 1 on the Parent side. The relationship is part of an identifier of the Child. Typical instances of the Child are thought of as being a "part of" a Parent instance. Unlike referential integrity constraints, where the referent instance can be changed, a Child instance must remain with a single Parent instance. Both Parent and Child may participate in other relationships. The Parent-Child pattern is similar to the concept of a composite object, an object with a hierarchy of exclusive component objects [Kim et al., 1987].

Example: In Figure 1, Order and LineItem form a Parent-Child pattern. There

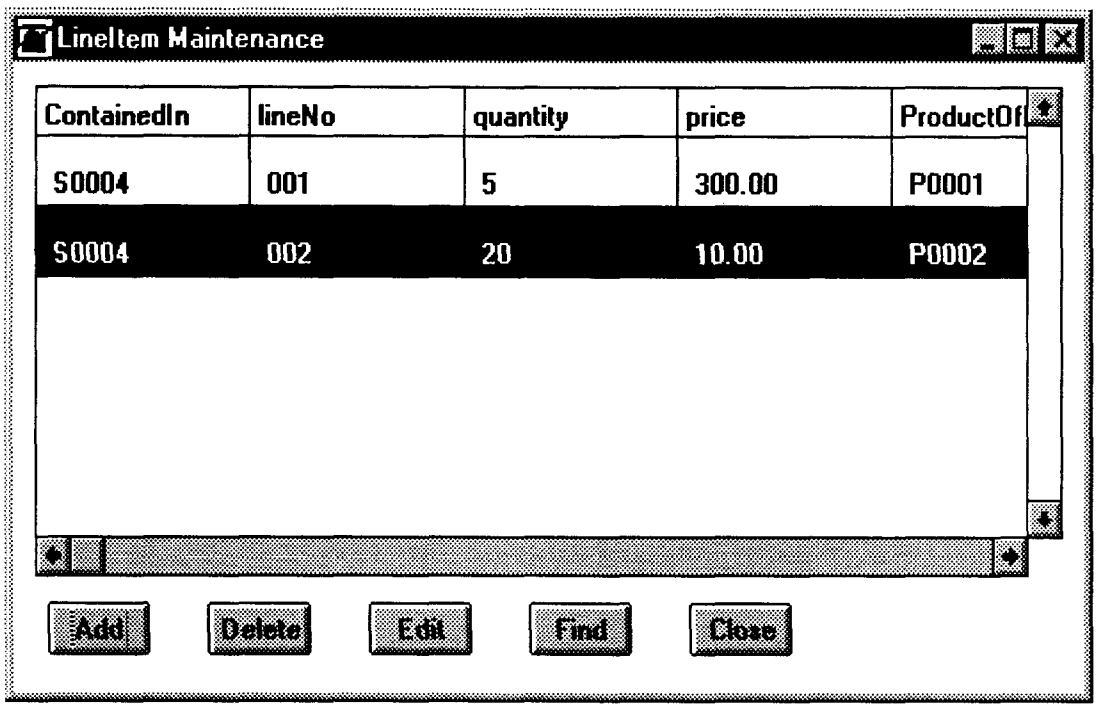
is a one to many relationship between them, Contain, which is part of the identifier of LineItem (denoted by the dot on this relationship near the LineItem class). LineItem instances are thought of as being part of a single Order instance. The combination of an Order instance and its associated LineItem instances form a composite object. Both Order and LineItem participate in other relationships.

Update Semantics: Children instances are inserted through the parent instance and cannot exist without the parent instance. Children instances are cascade deleted when the parent instance is deleted.

Interface: Classes conforming to this pattern can be maintained using an integrated interface for both classes with the child instances represented as the detail for the parent instance. As illustrated in Figure 4, the interface for Order contains a button to open the interface for LineItem. When the interface for LineItem is open, it lists the instances related to the selected Order instance. Any LineItem instances added using this interface are automatically related to the currently selected Order instance. LineItem instances cannot be moved from one Order instance to another. When an Order instance is deleted, its related LineItem instances are also deleted. The update windows for Order and LineItem each contain widgets (e.g., combo boxes) for their other many to one relationships (Place for Order and SellProduct for LineItem).



a. Default Display Window for Order



b. Default Display Window for LineItem

Figure 4. Default Interface for Order-LineItem (Parent-Child Pattern)

4. Full Intersection

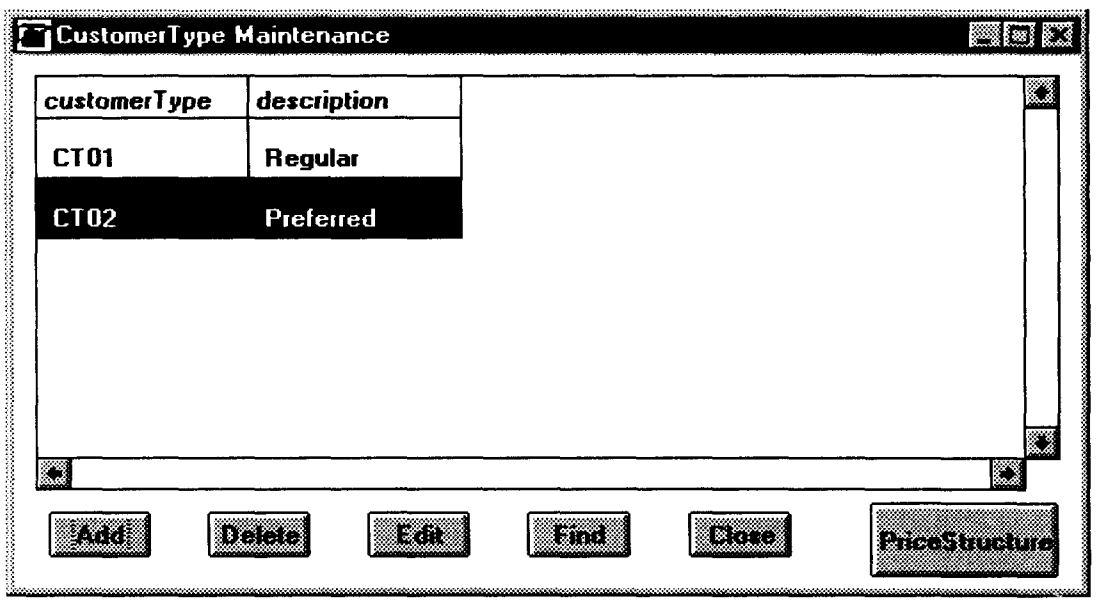
Pattern: This pattern involves three classes: 2 Entry classes and 1 Full Intersection class. Each Entry class has a one to many relationship with the Full Intersection class having minimum cardinality 0 on the Full Intersection side and minimum cardinality 1 on the Entry class side. The relationships with the Entry classes together form the identifier of the Full Intersection class. The Full Intersection class has one instance for each unique pair of instances of the two Entry classes.

Example: In Figure 1, CustomerType, PriceStructure and Product form a Full Intersection pattern. CustomerType and Product are the Entry classes. PriceStructure is the Full Intersection class. PriceStructure is identified by the relationships with CustomerType and Product. For each combination of Product and CustomerType instances, there must be an instance of PriceStructure which defines the discount rate for that Product instance and CustomerType.

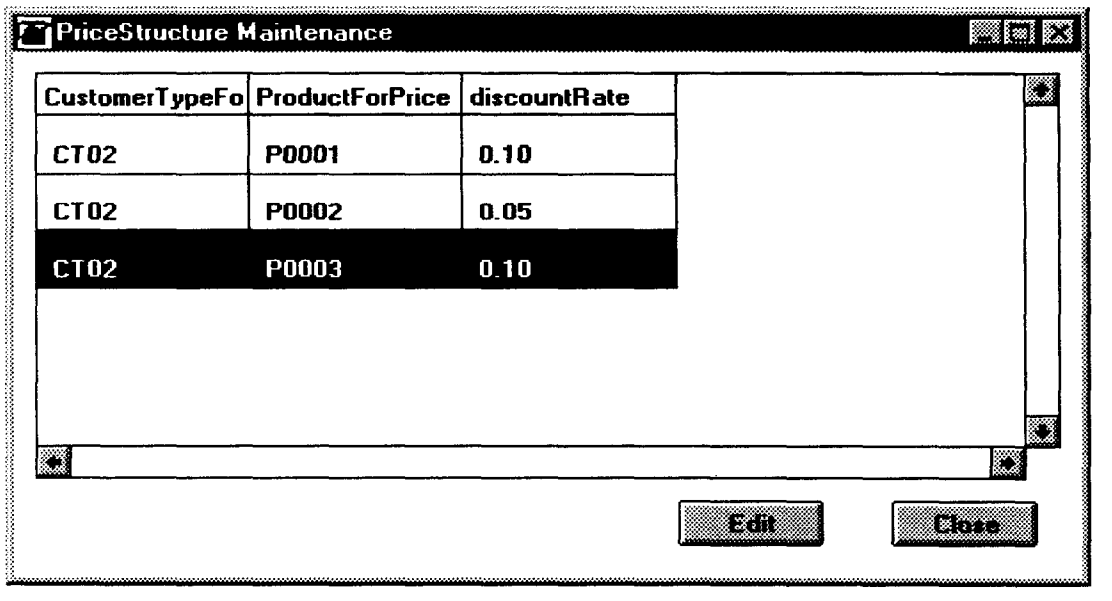
Update Semantics: Inserting an instance of either Entry class requires the insertion of an instance of the Intersection class for each existing instance of the other Entry class. Deleting an instance of either Entry class requires the deletion of all related Intersection class instances. Instances of the Intersection class can

only be added or deleted through an Entry class instance.

Interface: Classes conforming to this pattern can be maintained using an interface similar to that for Parent-Child patterns, with the Intersection class instances represented as the detail for each Entry class. However, when an instance of an Entry class is added, the Intersection class instances related to it for all instances of the other Entry class are also added. When an instance of an Entry class is deleted, all Intersection class instances related to it must be deleted. Figure 5



a. Default Display Window for CustomeType



b. Default Display Window for PriceStructure

Figure 5. Default Interface for CustomerType-PriceStructure-Product (Full Intersection Pattern)

illustrates an interface for the Full Intersection pattern for PriceStructure through CustomerType (the interface through Product is similar). The interface for CustomerType (and Product) are similar to the standard interface for Parent classes having a button for PriceStructure which opens its display window. PriceStructure has an instance for each pair of Product and CustomerType instances. Instances related to the selected CustomerType instance (or the selected Product instance if the Product display window is used) are included in the display window. When a new Product instance is added, an instance of PriceStructure is added for each instance of CustomerType. Similarly, when a new CustomerType is added, an instance of PriceStructure is added for each instance of Product. If a Product or CustomerType instance is deleted, all related PriceStructure instances are deleted as well. Since PriceStructure instances can only be added and deleted via their related CustomerType and Product instances, this window does not contain add or delete buttons.

This pattern can be generalized to intersection objects with more than two entry objects, i.e., classes representing full n-ary relationships.

5. Subclasses

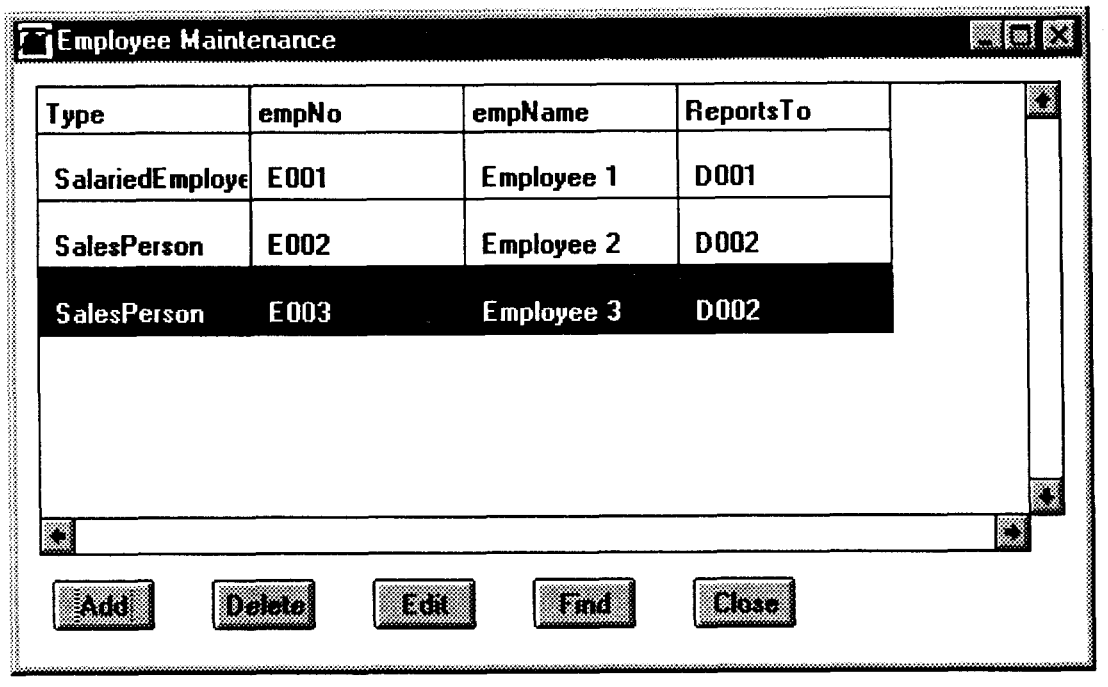
5.1 Partitioned Subclass

Pattern: This pattern involves a class having subclasses such that each instance of the class must be in exactly one of its subclasses.

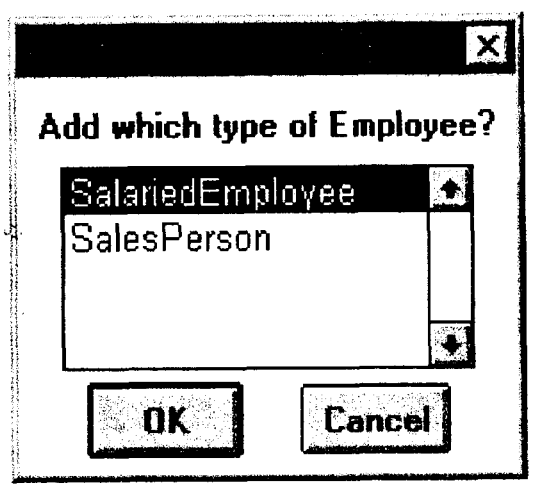
Examples: In Figure 1, Employee and its subclasses form a Partitioned Subclass pattern, denoted by the p inside the subclass triangle. This indicates that each employee is either a salaried employee or a salesperson and not both. In the Smalltalk vocabulary, Employee is an *abstract* class, i.e., it contains no instances. All of its instances are in one of its subclasses, SalariedEmployee or SalesPerson which are said to *partition* their superclass, Employee.

Update Semantics: Inserting an instance of the class requires the instance to be assigned to exactly one of its subclasses.

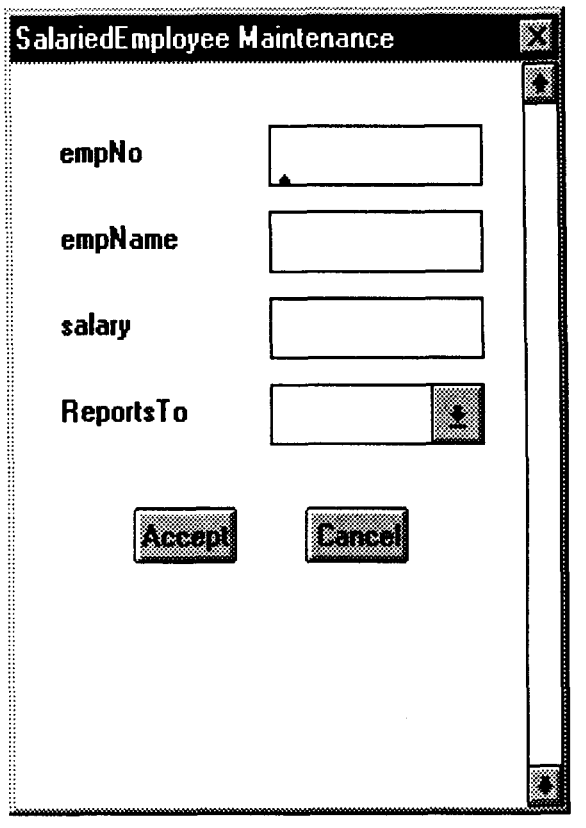
Interface: The display window for the class includes all instances of its subclasses. It has an additional column denoting the subclass of each instance (Figure 6.a). Furthermore, when an instance is added, a display list is opened which requires the user to choose the subclass to which the instance belongs (Figure 6.b). The update window for each subclass includes all attributes and many side relationships of the class as well as those of the subclass (Figure 6.c).



a. Default Display Window for Employee



b. Employee Type Display List



c. Default Update Window for SalariedEmployee

Figure 6. Default Interface for Employee (Partitioned Subclass Pattern)

5.2 Exclusive Subclass

Pattern: This pattern also involves a class with subclasses. It is similar to partitioned subclasses, however, when an instance is added, it can be assigned to the class *or* to one of its subclasses.

Example: There is no example in Figure 1, however, if the p in the subclass triangle were replaced with an e (exclusive), this would indicate that an employee can be a salaried employee or a salesperson or neither, i.e., that SalariedEmployee and SalesPerson form an Exclusive Subclass pattern with Employee.

Update Semantics: When an instance of the class is inserted, it may be assigned to the class or to one of its subclasses.

Interface: Similar to the Partitioned Subclass pattern, however, when an instance is added, the display list requires the users to choose the class or one of its subclasses for the instance.

6. Cyclic Constraint

Pattern: This pattern involves a set of classes involved in a constrained relationship loop (or cycle). A constrained relationship loop has one class in which the relationship paths in each direction are a set of many (and/or one) to one relationships that, for each instance of the class, converge at a single instance of the class at the end of each path. Such patterns exist, for example, when the Child in a Parent-Child pattern is in a time dependent relationship loop through the Parent. In this case the path through the Parent represents a transitive dependency specified at a point in time prior to the creation of instances in the relationship loop representing the basic functional dependencies. As a result, the set of legal instances for the other relationship of the Child in the relationship loop are restricted to those reachable through the Parent.

Example: In Figure 1, there is a Cyclic Constraint pattern involving the Parent-Child pattern Payment and OrderPayment. When a customer makes a payment, an instance of Payment is created and related to that customer via the relationship Pay. At a later point in time that payment must be allocated to orders placed by that same customer. Instances of OrderPayment contain the allocation of Payment instances to Order instances. The cyclic constraint specifies that Payment instances can only be allocated to Order instances that are related to the Customer instance to which the Payment instance is related. Using these classes and their defined relationships, this constraint can be specified as follows

(where anOrderPayment is any instance of OrderPayment):

anOrderPayment Apply Pay = anOrderPayment Credit Place.

Where Apply, Pay, Credit, and Place are messages that traverse relationships. In SOODAS, the corresponding methods are automatically generated when the database is defined [March and Rho, 1996, 1997].

Update Semantics: From an update perspective, the set of instances to which an instance can be related must be restricted to those conforming to the given constraint. The list of Order instances to which an Order-Payment instance can be related, for example, must be restricted to those that are related to the Customer instance related to the Payment instance to which the OrderPayment instance is related.

Interface: Since relationships are maintained via widgets (combo boxes), the update window for a class designated to have a Cyclic Constraint for a relationship will generate choices for the relationship that conform to the specified constraint. Cyclic Constraints are specified as a series of relationships resulting in the set of legal values for the relationship to which it applies. The legal choices for the Credit relationship would be specified as the traversal of the following relationships: Apply Pay Place.

Ⅳ. A Framework for Object Patterns

To support the object patterns described in the previous section, a set of meta-classes are defined. They are: *EntityInterface, ParentInterface, EntryInterface, DependentInterface, ChildInterface,* and *IntersectionInterface*. They have been implemented in Smalltalk as part of SOODAS, a Semantic Object-Oriented Data Access System [March and Rho, 1996, 1997]. They generate the types of display and update windows illustrated in Figures 2 through 7. SOODAS has four meta-classes: *EntityObject*, which implements Entities as subclasses of itself (instance variables are used to represent attributes), *Relationship* which defines and maintains relationships, *PermanentObject* which provides persistence (*EntityObject* and *Relationship* are subclasses of it), and *QueryNode* which supports multi-class querying.

A Smalltalk class can have class and instance variables and class and instance methods. Furthermore, a Smalltalk class can have class-instance variables, that is class variables with unique values in each of its subclasses. The class hierarchy for the SOODAS interface system is shown in Figure 7. Model and

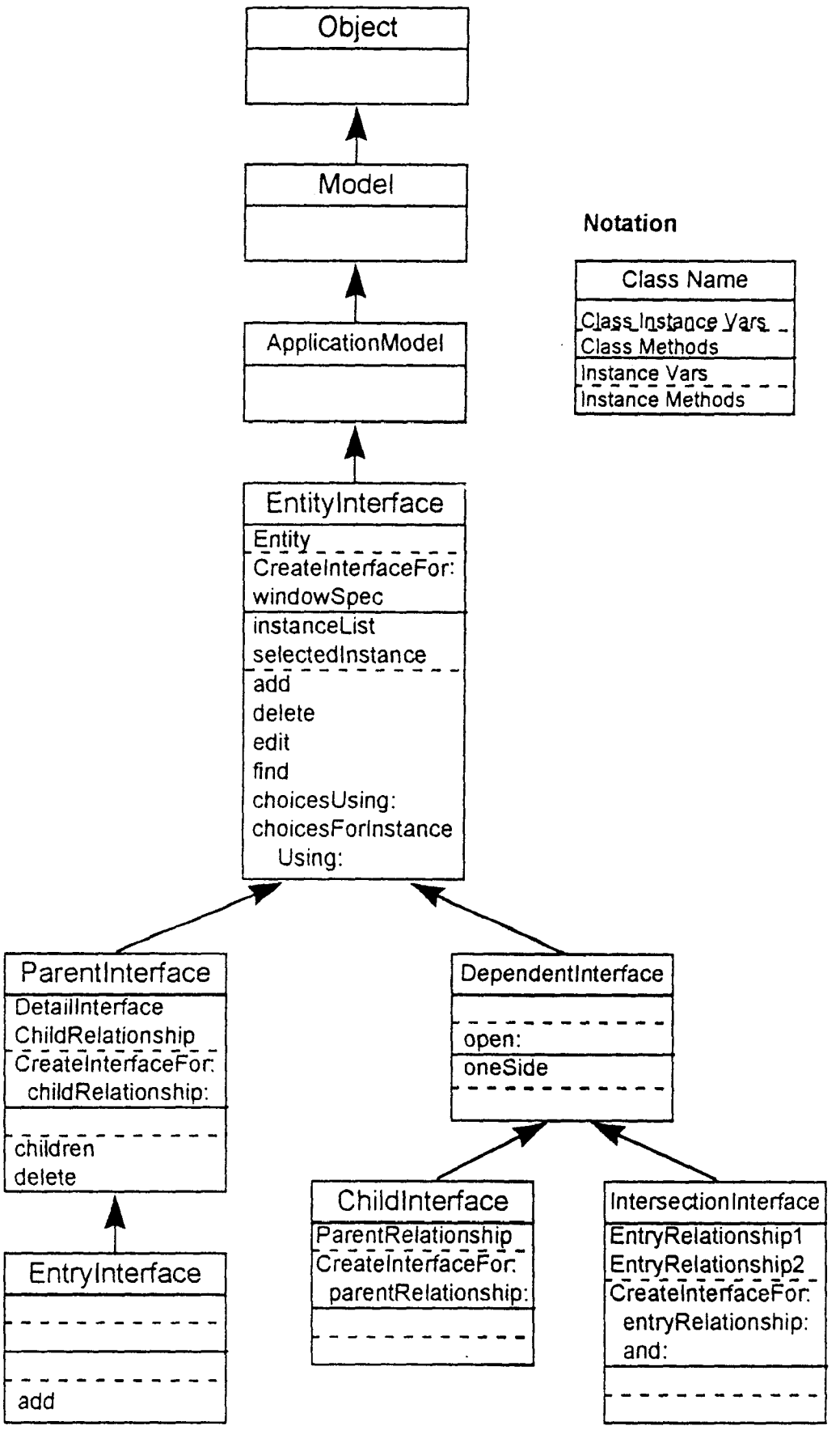


Figure 7. Meta-Classes

ApplicationModel are system classes that provide the engine behind the Model-View-Controller (MVC) architecture used to manage user interfaces [Goldberg and Robson, 1989; VisualWorks, 1994].

EntityInterface is a subclass of ApplicationModel. It has methods to generate interface classes for Unconstrained and Constrained Independent patterns and for Partitioned and Constrained Subclass patterns as subclasses of itself. It also has methods to define and enforce cyclic constraints.

ParentInterface is a subclass of *EntityInterface*. It has methods to generate interface classes for Parent-Child patterns. The Parent interface class is generated as a subclass of *ParentInterface*. The Child interface class is generated as a subclass of *ChildInterface* (see below). *ParentInterface* also provides methods to maintain parent-child relationships including cascade delete of Child instances.

EntryInterface is a subclass of *ParentInterface*. It has methods to generate interface classes for Full Intersection patterns. The Entry interface classes are generated as subclasses of *EntryInterface*. The Intersection interface class is generated as a subclass of *IntersectionInterface* (see below). *EntryInterface* provides a standard interface for Entry classes and creates (cascade deletes) instances of the Intersection class when an Entry class instance is added (deleted).

ChildInterface and *IntersectionInterface* provide interfaces for Child classes and Intersection classes, respectively. They are both defined as subclasses of *DependentInterface* which provides methods to integrate the Child/Intersection class with its Parent/Entry class(es). Interface classes for Child and Intersection classes are defined as subclasses of *ChildInterface and IntersectionInterface*, respectively.

The following subsections describe the class and instance variables and methods of each of these meta-classes. For brevity, we exclude accessor and assignment methods.

1. EntityInterface

EntityInterface has one class-instance variable, **Entity** which contains the class updated in this interface. It has two class methods. **windowSpec** answers the template for the display window (illustrated in Figure 2.a). **CreateInterfaceFor**: aClass creates a subclass of itself named <aClass>Interface. aClass, the parameter of the message, is the class to be maintained via the generated windows. It generates and stores the definition of the update window (illustrated in Figure 3) in a class method of <aClass>Interface named <aClass>Spec. It

contains an input box for each instance variable of aClass and a combo box for each one to one relationships and for each one to many relationship in which it participates on the many side. Thus it generates a standard interface window corresponding to the familiar "foreign key" notion in relational systems (although SOODAS uses the Relationship class rather than foreign keys or embedded objects to represent relationships).

EntityInterface has two instance variables. **instanceList** contains a value holder on the display list for the window. Thus it defines the list of instances to be displayed. **selectedInstance** contains a value holder on the instance selected from the display list. It specifies the instance to be acted upon (added, deleted, edited).

EntityInterface has six instance methods. Four, **add, delete, find** and **edit** correspond to action buttons in the display window. Two, **choices-Using: and choicesForInstanceUsing:**, are used in combo boxes for relationships. **add** checks to see if subtypes must be specified (for Partitioned Subclass and Exclusive Subclass patterns), creates a new instance, and opens the appropriate update window. **delete** removes the instance specified in **selectedInstance**. **find** requests an identifier, finds the instance with that identifier, and stores it in **selectedInstance**. **edit** opens the appropriate update window, displaying the instance specified in **selectedInstance**. **choicesUsing:** relName answers the selection list for the unconstrained relationship named relName. This includes all instances of the related class. **choicesForInstanceUsing:** aString answers the selection list for the cyclic constraint specified in the parameter aString. It applies the relationship methods specified in aString to the instance specified in **selectedInstance**, answering the set of legal instances for the relationship.

To illustrate these concepts, consider the result of the following Smalltalk code segment:

```
EntityInterface CreateInterfaceFor: Department.
```

First, *EntityInterface* creates a subclass of itself named DepartmentInterface. It assigns Department to its class-instance variable, **Entity**. It generates a class method named DepartmentSpec containing the specification of the update window for Department (Figure 2.b).

2. ParentInterface and EntryInterface

ParentInterface is a subclass of EntityInterface and thus inherits its class and instance variables and methods. It has two additional class-instance variables:

DetailInterface which holds the interface class of the Child and **ChildRelationship** which holds the Parent-Child relationship. The same relationship is held in the class-instance variable **ParentRelationship** in the interface class of the Child (see *ChildInterface* below). It has one class method, **CreateInterfaceFor**: aClass **childRelationship**: relName, which creates a subclass of *ParentInterface* for aClass, named <aClass>Interface, where aClass is the Parent class. It generates and stores the definition of the update window in a class method of <aClass>Interface named <aClass>Spec. Finally, it sends a **CreateInterfaceFor:** aChildClass to **ChildInterface** (see below) where aChildClass is related to aClass via the relationship relName (the resulting <aChildClass> Interface is held in **DetailInterface**).

ParentInterface has two additional instance methods. **children** opens the display window for the Child instances related to the selected Parent instance (e.g., used when the LineItem button is selected (Figure 4)). **delete** overrides *EntityInterface* delete method. It checks deletion constraints for the selected parent instance and cascade deletes children instances.

EntrytInterface is a subclass of *ParentInterface* and has one additional instance method, **add** which overrides *EntityInterface* **add** method as necessary for the Full Intersection pattern.

3. DependentInterface

DependentInterface is a subclass of *EntityInterface* and the superclass of *ChildInterface* and *IntersectionInterface*. It has one class method **open:** anInstance which creates an instance of itself, assigns anInstance, the Parent or Entry class instance, to its instance variable, **oneSide**, and opens its update window. *ChildInterface* has one class-instance variable, **ParentRelationship** corresponding to **ChildRelationship** in the Parent class.

IntersectionInterface has two class-instance variables, **EntryRelationship1** and **EntryRelationship2**, each holding one of the Entry relationships of a Full Intersection pattern. It has one class method, **CreateInterfaceFor:** aClass **entryRelationship:** relName1 **and:** relName2 which creates a subclass of itself named <aClass>Interface where aClass is the Inter- section class. It sends a **CreateInterfaceFor:** message to *EntryInterface* for each of the Entry classes related to aClass via relName1 and relName2.

V. Illustration

Update interfaces are generated by identifying patterns in the object model and sending messages to the appropriate meta-classes. This section illustrates how update interfaces for the object model of Figure 1 are generated. The database definition is shown in **Appendix 1**. The Interface classes are illustrated in Figure 8.

The interface for Department, an Unconstrained Independent (Referent) pattern, is generated using:

```
EntityInterface CreateInterfaceFor: Department.
```

EntityInterface generates a subclass of itself named DepartmentInterface which is capable of managing the interface windows illustrated in Figure 2. It uses methods inherited from *EntityInterface* to do so. It opens the Department display window when sent the message **open** (inherited from ApplicationModel). It opens the Department update widow, specified in DepartmentSpec, when the add or edit buttons are selected (using the **add** or **edit** methods inherited from *EntityInterface*).

The interface for Customer, a Constrained Independent (Referential) pattern is generated using:

```
EntityInterface CreateInterfaceFor: Customer.
```

EntityInterface generates CustomerInterface. Its class method CustomerSpec defines the Customer update window. It includes a combo box for each many to one relationship as shown in Figure 3. Since the minimum cardinality of those relationships is 1, the choices in the combo box are restricted to the identifier values of the related class.

The interfaces for Order-LineItem and Payment-OrderPayment, Parent-Child patterns, are generated using:

```
ParentInterface CreateInterfaceFor: Order childRelationship: 'Contain'.
ParentInterface CreateInterfaceFor: Payment childRelationship: 'Apply'.
```

ParentInterface generates interface classes for Parents (OrderInterface and PaymentInterface) as subclasses of itself and requests *ChildInterface* to generate interface classes for Children (LineItemInterface, and OrderPaymentInterface). The

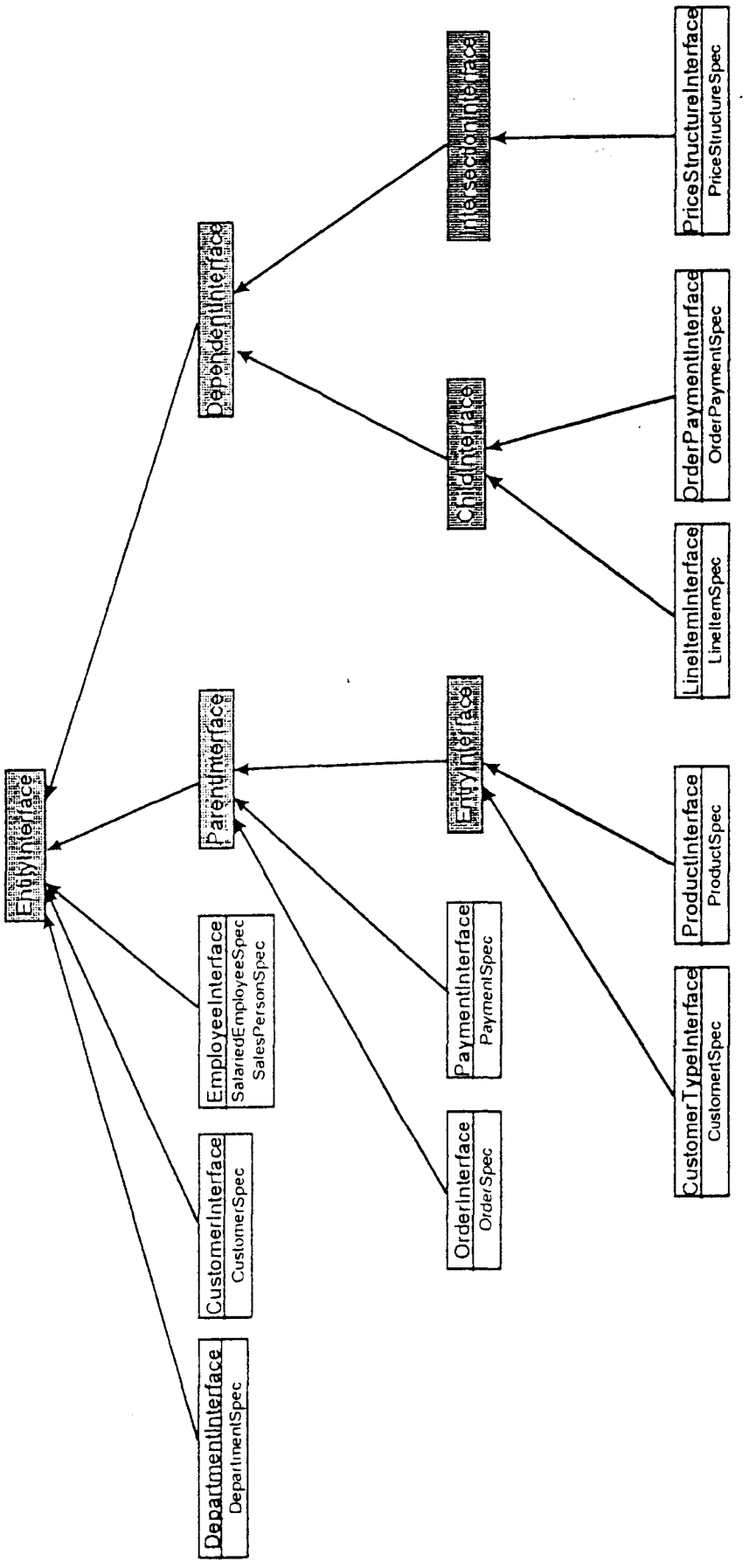


Figure 8. Interface Class Hierarchy

display windows for Order and LineItem are illustrated in Figure 4.

The interfaces for CustomerType-PriceStructure-Product, a Full Intersection pattern are generated using:

IntersectionInterface CreateInterfaceFor: PriceStructure entryRelationship: 'CustomerTypePricing' and: 'ProductPricing'.

IntersectionInterface generates PriceStructureInterface as a subclass of itself and requests *EntryInterface* to generate CustomerTypeInterface and ProductInterface. The display windows for CustomerType and PriceStructure are illustrated in Figure 5.

Employee and its subclasses form a Partitioned Subclass pattern -- SalariedEmployee and SalesPerson partition Employee. This requirement is defined in the database definition (**Appendix 1**) by sending the message,

Employee Partition: true.

Having specified constraint for Employee, its update interface is generated using:

EntityInterface CreateInterfaceFor: Employee.

EntityInterface generates EmployeeInterface which contains the update window definitions for both SalariedEmployee and SalesPerson. Its display window has an additional column for the class (Figure 6.a). When an employee is added, its class must be selected from the display list (Figure 6.b). Then the update window for the selected class is open (Figure 6.c).

OrderPayment has a Cyclic Constraint specified as:

OrderPaymentInterface CyclicConstraintFor: 'Credit' relationshipPath: 'Apply Pay Place'.

OrderPaymentInterface inherits this method from *EntityInterface* (through *DependentInterface* and *ChildInerface*). The list of values displayed in the combo box for Credit in the update window specified in OrderPaymentSpec (a generated class method of OrderPaymentInerface) is restricted to orders placed by the customer who made the related payment.

Ⅵ. Summary and Future Research

We describe six recurring patterns in conceptual object models each having

similar update semantics and requiring similar update interfaces. These are: Unconstrained Independent, Constrained Independent, Parent-Child, Full Intersection, Subtype, and Cyclic Constraint. We develop a framework, i.e., a set of meta-classes, for implementing these patterns within a Semantic Object Oriented Data Access System (SOODAS). Once a developer recognizes a pattern, display and update windows can be easily configured by sending messages to the appropriate meta-class.

Future research will progress in several directions. First, additional patterns will be identified through the application of our current patterns in the development of real object systems. Although our patterns are widely applicable and general, we anticipate that there are other useful patterns that should be identified and supported by our framework. Second, the current framework will be refined to include additional interface options such as multiple instance update windows, for example, a single update window in which a Parent (or Entry) instance and all of its related Child (Intersection) instances can be maintained. Third, our patterns must be evaluated empirically. One of the claimed advantages of Object Orientation is development efficiency and code re-usability. A key concern, and one for empirical research, is the impact of our patterns on system development effort and maintainability. Preliminary evidence is currently being gathered by using the patterns as the implementation environment for teaching OO development. Finally, our patterns can be applied to OOPL's other than Smalltalk. Currently, a framework that supports the patterns is being developed in JAVA.

References

Alexander, C., *The Timeless Way of Building*, Oxford University Press, New York, 1979.

Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., and Angel, S., *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press, New York, 1977.

Budinsky, F. J., Finnie, M. A., Vlissides, J. M., and Yu, P. S., "Automatic Code Generation from Design Patterns," *IBM Systems Journal*, Vol. 35, No. 2, 1996.

Coad, P., "Object-Oriented Patterns," *Communications of the ACM*, Vol. 35, No. 9, September 1992, pp. 152-159.

Coad, P., North, D., and Mayfield, M., *Object Models: Strategies, Patterns, and Applications*, Yourdon Press, Englewood Cliffs, NJ, 1995.

Curtis, B., "Cognitive Issues in Reusing Software Artifacts," in Biggerstaff, T. J. and Perlis, A. J. (eds), *Software Reusability, Volume II*, Addison-Wesley, 1989, pp. 269-287.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J., "Design Patterns: Abstraction and Reuse of Object-Oriented Design," *Proceedings of ECOOP '93 Conference*, 1993.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Publishing Co., Reading, MA, 1995.

Goldberg, A. and Robson, D., *Smalltalk-80*, Addison-Wesley, Reading, MA, 1989.

Isakowitz, T. and Kauffman, R. J., "Supporting Search for Reusable Software Objects," *IEEE Transactions on Software Engineering*, Vol. 22, No. 6, June 1996, pp. 407-423.

Johnson, R., "Documenting Frameworks Using Patterns," *Proceedings of OOPSLA '92*, Vancouver, Canada, October 1992, pp. 63-76.

Kim, W., Banerjee, J., Chou, H.-T., "Composite Object Support in an Object-Oriented Database System," *Proceedings of OOPSLA '87*, October 4-8, 1987, pp.118-125.

March, S. T. and Rho, S., "Object Support for Entity Relationship Semantics," *Proceedings of Workshop on Information Technologies and Systems*, December 1996, pp. 1-10.

March, S. T. and Rho, S., "SOODAS: A Semantic Object-Oriented Data Access System," under journal review.

Pree, W., *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, Publishing Company, Wokingham, England, 1995.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorenson, W., *Object-Oriented Modeling and Design*, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1991.

Schmidt, D. C., "Using Design Patterns to Develop Reusable Object-Oriented Communication Software," *Communications of the ACM*, Vol. 38, No. 10, October 1995, pp. 65-74.

Schmidt, D., Fayad, M., and Johnson, R., "Software Patterns," *Communications of the ACM*, Vol. 39, No. 10, October 1996, pp. 36-40.

VisualWorks, *User's Guide*, ParcPlace Systems, Inc., 1994.

Appendix . SOODAS Script to Implement the Object Model of Figure 1.

"Define the entities."

EntityObject CreateEntity: #CustomerType attributes: 'customerType description' under: 'OrderEntry'.

EntityObject CreateEntity: #Customer attributes: 'customerNo name address' under: 'OrderEntry'.

EntityObject CreateEntity: #Order attributes: 'orderNo date' under: 'OrderEntry'.

EntityObject CreateEntity: #LineItem attributes: 'lineNo quantity price' under: 'OrderEntry'.

EntityObject CreateEntity: #Product attributes: 'productNo description price qoh' under: 'OrderEntry'.

EntityObject CreateEntity: #PriceStructure attributes: 'discountRate' under: 'OrderEntry'.

EntityObject CreateEntity: #Department attributes: 'deptNo deptName' under: 'OrderEntry'.

EntityObject CreateEntity: #Employee attributes: 'empNo empName' under: 'OrderEntry'.

EntityObject CreateEntity: #Payment attributes: 'checkNo dateReceived amount' under: 'OrderEntry'.

EntityObject CreateEntity: #OrderPayment attributes: 'lineNo amount' under: 'OrderEntry'.

" Define the subtypes. "

Employee CreateSubtype: #SalariedEmployee attributes: 'salary'.

Employee CreateSubtype: #SalesPerson attributes: 'commisionRate'.

Employee Partition: true.

" Define the relationships. "

Relationship new: CustomerType and: PriceStructure withMin: 1 andMin: 0 withMax: 1 andMax: Many named: 'CustomerTypePricing' accessBy: 'PricesForCustomerType' inverselyBy: 'Customer-TypeForPrice'.

Relationship new: Product and: PriceStructure withMin: 1 andMin: 0 withMax: 1 andMax: Many named: 'ProductPricing' accessBy: 'PricesForProduct' inverselyBy: 'ProductForPrice'.

Relationship new: CustomerType and: Customer withMin: 1 andMin: 0 withMax: 1 andMax: Many named: 'CustomerType' accessBy: 'CustomersOfType' inverselyBy: 'TypeOfCustmer'.

Relationship new: Product and: LineItem withMin: 1 andMin: 0 withMax: 1 andMax: Many named: 'SellProduct' accessBy: LineItemsOfProduct' inverselyBy: 'ProductOfLineItem'.

Relationship new: Customer and: Order withMin: 1 andMin: 0 withMax: 1 andMax: Many named: 'Place' accessBy: 'Places' inverselyBy: 'PlacedBy'.

Relationship new: Order and: LineItem withMin: 1 andMin: 0 withMax: 1 andMax: Many named: 'Contain' accessBy: 'Contains' inverselyBy: 'ContainedIn'.

Relationship new: Department and: Employee withMin: 1 andMin: 0 withMax: 1 andMax: Many

named: 'Report' accessBy: 'Employs' inverselyBy: 'ReportsTo'.

Relationship new: SalesPerson and: Order withMin: 1 andMin: 0 withMax: 1 andMax: Many named: 'Obtain' accessBy: 'Obtains' inverselyBy: 'ObtainedBy'.

Relationship new: SalesPerson and: Customer withMin: 1 andMin: 0 withMax: 1 andMax: Many named: 'Service' accessBy: 'Services' inverselyBy: 'ServicedBy'.

Relationship new: Customer and: Payment withMin: 1 andMin: 0 withMax: 1 andMax: Many named: 'Pay' accessBy: 'Pays' inverselyBy: 'PaidBy'.

Relationship new: Payment and: OrderPayment withMin: 1 andMin: 0 withMax: 1 andMax: Many named: 'Apply' accessBy: 'AppliedTo' inverselyBy: 'AppliedFrom'.

Relationship new: Order and: OrderPayment withMin: 1 andMin: 0 withMax: 1 andMax: Many named: 'Credit' accessBy: 'CreditedBy' inverselyBy: 'Credits'.

" Define the external identifiers. "

CustomerType Identifier: 'customerType'.

Product Identifier: 'productNo'.

PriceStructure Identifier: 'CustomerTypeForPrice ProductForPrice'.

Customer Identifier: 'customerNo'.

Order Identifier: 'orderNo'.

LineItem Identifier: 'ContainedIn lineNo'.

Department Identifier: 'deptNo'.

Employee Identifier: 'empNo'.

Payment Identifier: 'PaidBy checkNo'.

OrderPayment Identifier: 'AppliedFrom lineNo'.