Associations and Roles in Object-Oriented Modeling*

Wesley W. Chu and Guogen Zhang
{wwc, gzhang}@cs.ucla.edu

Computer Science Department
University of California, Los Angeles, CA 90095

Abstract. We present an extended ER model with entity, role, and association as the basic constructs for object-oriented modeling. The purpose of the constructs is to support object evolution and extension for long lived objects. A class hierarchy consists of a static part and a dynamic part. The static part is a classification of entity classes, while the dynamic part is the role classes played by entities. The interaction among objects are captured with association classes. Based on the observation that entities play roles in association with other entities, we provide a unified view on roles in associations and roles as an extension to objects. The proposed modeling constructs help developers better understand the interrelationship among entities, thus result in flexible implementations for dynamic systems.

1 Introduction

Object-oriented systems provide more constructs than relational systems to model and support application semantics. However, in the current object-oriented models there is a lack of direct association support and a lack of systematic support for dynamic object evolution. In this paper, we introduce an entity-role-association modeling framework. *Entity* classes will be used to capture the static ISA hierarchy of the fundamental objects being modeled in the system. *Role* classes will be used to capture the ISA hierarchy of roles that entities can dynamically assume throughout their lifetime in the system. Further, the *association* class construct is introduced to explicitly model the relationships entities can participate in as they evolve through various roles during their lifetime. Altogether, our entity-role-association model provides a useful extension to OO modeling that supports dynamic object evolution. In our model, all the instances of entity classes, roles classes, and association classes are objects.

Relationships among entities is one of the fundamental constructs in semantic modeling as indicated in the original Entity-Relationship (ER) model [Chen76] and later extensions [TYF86, HK87, PM88]. The relationships in the original ER can be viewed as refined into different relationships in later extensions, such

* This research is supported in part by DARPA contracts No. 30602-94-C-0207, No. N66001-97-C-8601, and US Air Force contract No. F30602-96-1-0255.

as ISA relationship and others. *Association* is the term commonly used to refer to the interaction among objects [Rat97]. The associated objects are the *roles* of the association. In object-oriented systems, objects interact with each other by sending and reacting to messages. Real world entities are usually modeled by objects, and the state and other properties of associations often have to be encapsulated into the role objects involved. However, the properties of interactions, or associations, can be captured and represented in separate "association objects".

Tanzer [Tan95] presented a critique on the current practice of using regular instance variables for associations. The main problems of using regular classes and pointer/reference instance variables for associations are: (1) For classes with existing objects, it is not easy to insert an instance variable for a new association. (2) Constructing association instances using regular classes for polymorphic associations needs typecase testing on the roles. (3) Using direct multiple dispatching [Ing86] or the visitor pattern [GHJV95] to achieve multiple polymorphism for associations is a complex scheme. Further, it requires the methods to be added into role classes, thus increases code dependencies among modules, and is only partially extensible. In this paper, we propose *association class* construct for complex association modeling and support, and provide implementation schemes to resolve these issues.

We have used the role concept within the context of associations above. Let us now introduce another role concept in *object with roles* for dynamic aspects of objects [GSR96, RS91, Pern90, Alba93, WCL97]. In traditional class-based object-oriented systems, an object is uniquely represented by an instance of the most specific class in the class hierarchy that the object qualifies. Sometimes the static ISA relationship results in complex class hierarchies, and causes problems for object evolution. *Object with roles* is a way to remedy these problems. Under this extension, an object has an instance corresponding to the base (role) class, and a set of role objects that the object is currently playing. An object can acquire roles or drop roles dynamically. A role implicitly inherits the properties of its player. Object with roles extension provides object-oriented systems with more flexibility and expressive power.

For example, a person in a university can be a student. He can also be an employee if he is a TA or RA. The traditional class hierarchy is shown in Fig. 1a. If John was only a student at the beginning, he was represented by an instance of "`Student`" class. But later he became a TA, and had to be represented with an instance of "`Student-Employee`" class, while keeping the same object identity, which is a problem. An instance of "`Student`" cannot become an instance of "`Student-Employee`" without change of its object identifier. If we create a new instance, all the references to the old object will have to be changed to the new instance, which is very costly if not impossible[2]. If both instances are kept, there will be some redundancy, causing potential inconsistency. We call this *object reclassification anomaly.* Using objects with roles, "`Person`" can be an entity class, and "`Student`" and "`Employee`" are two role classes of "`Person`"

[2] Some systems use `becomes` to support this. But a new object needs to be created.

(Fig. 1b). Thus John can be represented by a `Person` object with a `Student` role at first. He acquired the employee (TA) role when he became a TA later. Multiple inheritance is also avoided in this case.

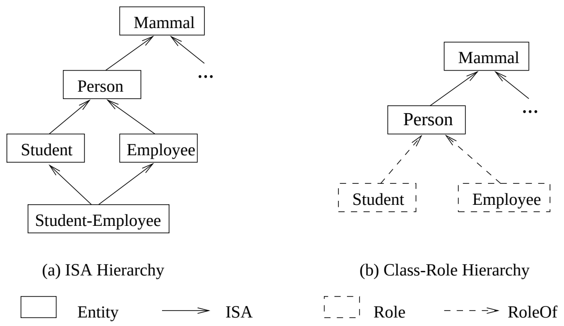

Fig. 1. ISA hierarchy and Class-Role hierarchy

Current research on *object with roles* and associations does not connect these concepts together. Based on the observation that an entity can play a role in association with other entities, we propose to unify the role concepts in *object with roles* and in *associations*. Thus roles can be unified both conceptually and in implementation under the entity-role-association scheme.

The paper is organized as follows. In section 2, we give an overview on modeling using entity-role-association. In section 3, we discuss association classes and their features. In section 4, we discuss the relationships of associations related to roles. Then in section 5, we discuss the relationships between players and roles. An application example is presented in section 6. And finally, a comparison of other related work is presented in section 7.

2 Modeling with Entities, Roles, and Associations

In our approach, the ISA relationship is subdivided into subclassing and role-playing, the entity classes constitute a static classification hierarchy while the role classes are dynamic aspects the entities can assume. The roles are played by objects, which can be entities or other roles.

The entity classes are those that reflect the static aspects of the real-world objects. The relevant entity classes are classified into a class hierarchy, in which classes have the partial order ISA relationship.

Role classes capture the temporal and evolutionary aspects of the real-world objects. The role classes themselves may also have class hierarchies to factorize the common properties with the ISA relationship. The semantics of ISA relationship for role classes are the same as the regular class inheritance (subclassing). For example, `Undergraduate` ISA `Student`, and `Graduate` student ISA `Student`. The player classes and role classes are related through RoleOf relationships, e.g.

`Student` is RoleOf `Person`, `ProjectManager` is RoleOf `Employee` as shown in Fig. 2.

Objects assume roles in their associations with other objects. Associations may have attributes for states and a set of methods, collectively called *properties*, in addition to the roles involved. Association classes are used to capture the properties of associations among objects with specific roles. The number of roles participating in an association is called the *arity* of the association. Association classes among the same set of classes may form a hierarchy.

Fig. 2a shows the student-employee example with associations for `Taking` and `Offering` courses. Note that `Student` and `Employee` are both roles of `Person` and roles in associations `Taking` and `Offering`. It also shows that a `ProjectManager` is a role of an `Employee`. Fig. 2b shows an example of medical image feature relationships for the tumor in a patient's brain, where a microlesion evolves to macrolesion and its spatial relationship with the lateral ventricle changes from disjoint to invading.

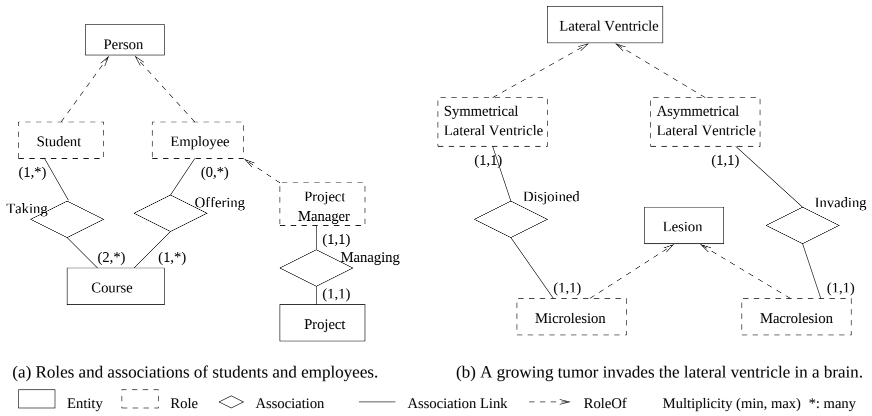

Fig. 2. Examples modeled by entity-role-association constructs.

The basic integrity constraints related to associations are:

1. Referential integrity: the role objects referenced in an association must exist and have the right types. This is a constraint on individual instances.
2. Multiplicity: the number of association instances of the same class in which each role can participate. The multiplicity of each role is specified with a (min, max) pair (as shown in Fig. 2), meaning that a role has to participate in the associations at least min times and at most max times. The multiplicity is a constraint on collections of instances.

Based on the properties, associations can be divided into different kinds: (1) Simple associations: A simple association is a simple link that does not have any particular properties of its own. An example of a simple association is *PartOf* relationship. (2) Complex associations: A complex association is one that has its own properties. Instances of associations are used to capture the interrelations among objects. (3) Polymorphic associations: Polymorphic associations are the same kind associations, but for different role classes they have different properties. These associations classes constitute a class hierarchy. For an association instance and a given message to it, the behavior depends on the types of roles involved in the association. For instance, a borrower borrowing a loan item from a library is modeled by `Borrowing` association class, and `loan_period()` is a method of the association. `Loan_period()` depends on the borrower classes and the loan item classes.

The entity-role-association scheme provides flexible basic model constructs. Other high-level objects can be built on top of these basic constructs, e.g. by grouping related objects. As a result, it can be used to specify dynamic and evolving OO systems.

3 Association Classes and Their Features

We first propose a scheme for the association class definition, then discuss the object-oriented characteristics for the association classes. To focus on the association construct and its properties, we ignore the RoleOf relationship on the role class side in this section. We use a simple library model as shown in Fig. 3 to illustrate association features.

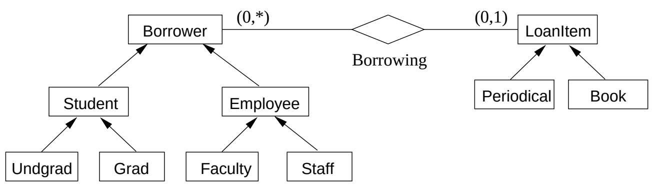

Fig. 3. A simple library model with association features.

An association class can be defined with its role classes, instance variables, and methods. For example, Fig. 4 shows for the `Borrowing` association class between `Borrower` and `LoanItem` for the library example in Fig. 3. The role classes in the order they appear in the association class definition constitute the *tuple of role classes* for the association.

All the association classes in a hierarchy for the "same" type of associations have the same name. For instance, if we want to capture the specialties of the association in different cases. For example, the "same" methods `loan_period` for `Borrowing[Borrower, Periodical]` and `Borrowing[Faculty, Book]` are different, and they would be defined as association classes as follows:

```
association Borrowing[Borrower, LoanItem]{...};
```

```
// association class name and the tuple of role classes:
association Borrowing[Borrower, LoanItem]
{   // normal attributes
    Date    loanDate;
    Date    dueDate;
    Date    returnDate;
    // methods
    virtual int loan_period() { ... }
};
```

Fig. 4. An association class definition for the simple library model example.

```
association Borrowing[Borrower, Periodical]:
              Borrowing[Borrower, LoanItem]{...};
association Borrowing[Faculty, Book]:
              Borrowing[Borrower, LoanItem]{...};
```

We now discuss the characteristics of object-orientation for the association classes. The major differences between association classes and regular classes are on the classification and polymorphism.

The constructors for association instances require a tuple of role instances that corresponds to the tuple of role classes declared in the association class. It is feasible that the constructors of the association classes are polymorphic, based on the role classes. With polymorphic constructors, it is unnecessary to do typecase testing on roles when constructing association objects. It puts more restrictive classification limit on the association instances.

We shall present two other major features – inheritance and polymorphism in the following sections.

3.1 Inheritance of Association Classes

One usage of inheritance is to specialize a class while allowing the subclass to inherit common properties from its superclass. Association class inheritance has its own features.

Implicit Inheritance. An association class defined for some role classes in class hierarchies is applicable to tuples of their corresponding subclasses.

For the library example (Fig. 3), because `[Undgrad, Book]` $\prec$ `[Borrower, LoanItem]`, the association class `Borrowing[Borrower, LoanItem]` defined in Fig. 4 is applicable to tuple `[Undgrad, Book]`. Similarly, the same holds for tuple `[Grad, Periodical]`.

Explicit Inheritance. The explicit inheritance of association classes is mainly used to override the implicitly inherited association class properties, therefore the following special rules apply to the explicit association class inheritance: (1) The subclass must have the same name as the superclass. (2) The subclass has the same arity as the superclass. (3) The tuple of role classes in the subclass are more specific than that of the superclass.

The following example shows how a subclass can be defined.

```
association Borrowing[Undgrad, Book] :   // subclass
            Borrowing[Student, LoanItem] // superclass
{   ...
    virtual int loan_period() { ... }    // overriding
};
```

3.2 Multiple Polymorphism of Messages on Associations

Given a message on an association instance, there are the following two message dispatching cases:

1. If the method is explicitly defined on the association class of the instance, then the defined method is invoked.
2. If the method is not explicitly defined on the corresponding class of the instance, then the implicitly inherited method will be invoked.

The method invoked depends on the role class tuple of the association instance, thus is *multiply polymorphic*. To determine a unique method to invoke for a message, the explicitly defined association methods must satisfy the unambiguity requirement. For the library example, if we have `loan_period()` explicitly defined on `Borrowing[Borrower, Book]` and on `Borrowing[Undgrad, LoanItem]`, to avoid ambiguity, `loan_period()` must be explicitly defined on `Borrowing[Undgrad, Book]`. An algorithm can help check if this requirement is met.

For any polymorphic methods that satisfy the unambiguity requirement, we can determine a unique and most specific method to invoke for a message.

4 The Relationship Between Associations and Roles

When implementing associations, depending on the requirements and constraints, there are two alternatives: intrinsic associations and extrinsic associations. Intrinsic associations provide a direct navigation path from role instances to the association instances, while extrinsic associations allow new associations defined on existing objects. Both alternatives can be realized as extensions atop "traditional" OO systems.

4.1 Intrinsic and Extrinsic Associations

In the intrinsic associations (Fig. 5a), roles are tightly coupled with the associations they participate in. An instance variable within each of the role classes is used for each association class, either single-valued or set-valued, depending on the multiplicity of the role in the association. It is similar to the traditional instance variable approach except that the instance variable now refers to the association objects.

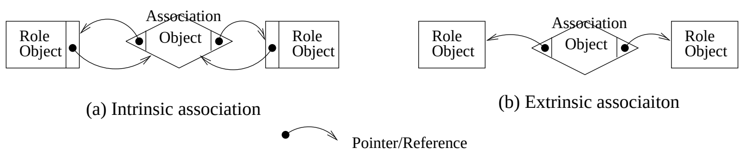

Fig. 5. Intrinsic and extrinsic binary association instances with roles.

If the role classes have already been defined and we want to add more associations but do not want to make changes to the role classes, or do not want to add additional variables and methods to role classes for sparse associations, we can use the extrinsic association (Fig. 5b). In such associations, no variables are used within role classes for associations, instead, a collection of all association instances is maintained, and the relationships from role objects to association objects is recovered by explicit join between role objects and the association instance collection. We call a collection of all the instances of the same association class an *association extent*.

For example, we can maintain the collection for the `Borrowing` associations, which can be defined or generated like this:

```
Multiplicity multi[] = {MANY, MANY};
extent<Borrowing>  borrowing(multi);
               // for a many-to-many association Borrowing
```

An association extent is a special collection class with integrity enforcement. The association extent also needs to support associative search, join with roles, and probably aggregate functions.

The advantages for the extrinsic association are flexibility in adding new associations to existing classes, and centralized integrity maintenance for each association class. Extrinsic associations also reduce code dependencies among modules using the same roles. Polymorphism can be utilized without adding methods into existing classes.

4.2 Interdependencies Between Roles and Associations

The referential integrity requires that the roles referenced in associations exist. Therefore when an association instance is to be created, the roles either exist or should be created at the same time. When a role is deleted, the deletion is cascaded to all the associations that the role is involved. An application may require that the deletion of a role be prohibited if there are still existing associations involving the role.

On the other hand, a role may exist without participating in any associations, that is, in the multiplicity of the role $min = 0$. For example, the borrower role for a student belongs to this case, because a student is a borrower even he is not currently borrowing any books. A role may be required to be involved in associations for the role to exist, i.e. $min > 0$ for the role. A project manager role in Fig. 2 belongs to this case, since a project manager is an employee who manages a project.

Since there are references to roles in associations, the navigation from an association to roles is always direct. However, navigation from a role to the association may not always be possible. Depending on the implementation scheme, the navigation from a role to associations can be direct (intrinsic associations) or indirect (extrinsic associations with extents).

5 The Relationship Between Players and Roles

A player of a role can be an entity or another role, which is ultimately played by an entity. Depending on the situation and functionality requirements, a role may be incorporated into an entity as a base role, and an entity may or may not be aware of some or all of the roles it is currently playing. An entity may be only shared by the extrinsic roles for redundancy elimination.

A base role is a role played by an entity for its entire lifetime in the system. Thus, if the entity is not playing the base roles, it would not be included into the system. Therefore there is no need to separate the base roles from the entity. For example, `Employee` is usually considered as a base role and is not separated from `Person` in a company personnel system.

Often it is unnecessary for the player to keep track of all the roles it is playing. Thus, there are links only from the roles to the player. The roles cannot be accessed directly from their player. We call these *extrinsic roles* to the player (Fig. 6a). The typical supporting scheme for extrinsic roles is the inheritance by delegation [Taiv96]. That is, messages that are not part of direct interface of the role are forwarded (delegated) to the player. The user of the role object treats the role instance the same way as the role class inherits the player class in subclassing. For instance, a student is a member of the student body, but he is also a member of a club. The club member role can be modeled by an extrinsic role.

When a player needs to know the specific roles that it is playing, the roles are *intrinsic*, and the behavior of the player depends on these roles. The roles may be only referenced by the player, i.e. unidirectional from the player to the roles. In this case, roles are just part of states of the player (Fig. 6b). When the relationship between the player and roles are bidirectional (Fig. 6c), in addition to being accessed from inside the player object, the intrinsic roles can also be referenced by other objects. For example, the properties of an employee and his project manager role (Fig 2a) are dependent on each other, therefore, a bidirectional intrinsic role for project manager should be used (as in Fig 6c).

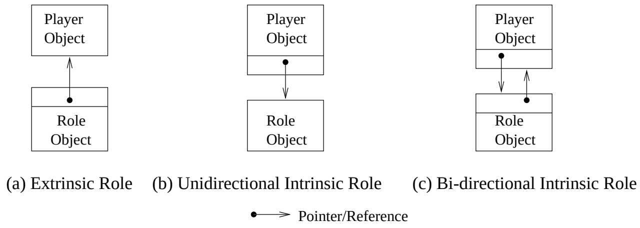

Fig. 6. Extrinsic and intrinsic roles.

The relationships between players and roles are either one-to-one or one-to-many, with roles dependent of players, i.e. a role cannot exist without a player. Since role classes cannot be instantiated without a player, players always exist first. An entity can play multiple roles at the same time, and play different sets of roles at different times.

The dependencies among players, roles, and associations have to be defined precisely in the system model to achieve system integrity.

6 An Application Example

Let us use our experience in implementation of our CoBase [Chu96] project as an example to illustrate the usefulness of the association class and role class. CoBase is a cooperative database interface which provides query relaxation (approximate answers), associative querying (provides relevant answers that the user does not explicitly ask for), and explanation that describes the relaxation process and its reasoning. A portion of its classes and associations are shown in Fig. 7, where `RelaxEngine` is for Relaxation Engine, `AssocEngine` is for Association Engine, and `ExplanEngine` is for Explanation Engine. These engines all work on a uniform internal query representation (QueryRep). Part of QueryRep component classes are shown in the left-hand side of Fig. 7.

There is a large class hierarchy for representing operands in SQL query conditions with our cooperative SQL (CoSQL) [CID96] extension. Engine modules of the system manipulate on a query representation (QueryRep) to achieve their functionalities. The CoBase system is developed incrementally. In traditional object-oriented way, many functions would have to be added gradually into the operand class hierarchy. It would cause endless recompilations of all the code and a fat interface that is the union of relatively independent functions for different modules. If we extend functions outside of the QueryRep, we have to use a long list of typecase tests to discriminate the types of the operands. Polymorphism is important in eliminating these long lists of cases.

To reduce the dependencies among `RelaxEngine`, `AssocEngine`, and `ExplanEngine`, we used the following mechanisms:

1. Extrinsic role classes for `QueryCond` classes are used for roles associated with `RelaxEngine` and `AssocEngine` to keep their states. Extrinsic roles make

their usage of `QueryCond` in `RelaxEngine` and `AssocEngine` independent of each other.
2. Extrinsic roles are used for the conditions that need to be explained in `ExplanEngine`. As a result, there are only dependencies of `ExplanEngine` on `RelaxEngine` and `AssocEngine`, but not the other direction.
3. Intrinsic associations are used for the associations on the `RelaxEngine`, `AssocEngine`, and `ExplanEngine`, so that the manipulation of these associations is direct from these respective engines.

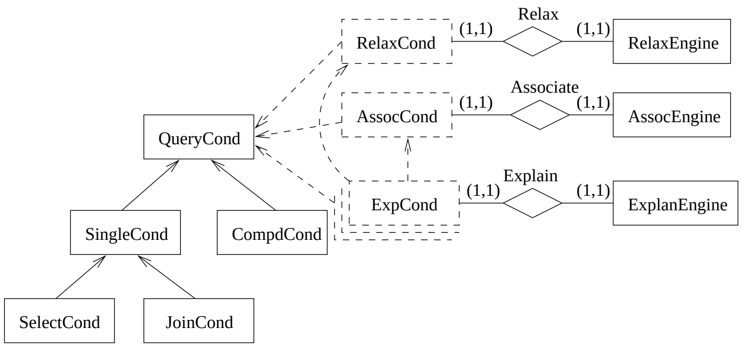

Fig. 7. Illustration of associations and roles in part of the CoBase system.

Our experience has revealed the association and role constructs not only increase expressive power, but more importantly, greatly simplify system development and maintenance.

7 Related Work

The association construct proposed in this paper is related to semantic data modeling, object modeling technique(OMT), multi-methods in functional OO languages, and traditional relational database integrity constraints.

The importance of supporting explicit relationships in object-oriented systems has long been recognized [Ditt90, Rumb87, Tan95]. There have been also several attempts in supporting relationships in object-oriented databases [AGO91, Brat91, Catt96, DG91, NQZ91]. In [Rumb87], Rumbaugh proposed relation as a construct to support associations. A relation is a collection of associations, but associations are not objects and cannot have methods. The relationship support proposed in [Brat91] is similar on this aspect. In [AGO91], an association is a set of distinct bindings, each of which is like a value tuple. Their association is more like our association extent, or an extension to relations for relationships in relational databases. The major difference is that our association also emphasizes the properties of individual association instances. ODMG-93 [Catt96] has a `relationship` prefix for instance variables that are for associations to maintain integrity constraints, but no special construct for associations. An association concept in OMT is proposed by Rumbaugh et al.[Rumb91] and used in object relationship modeling and translation into regular instance variables. These latter two proposals suffer some of the problems as pointed out in the introduction of this paper.

Our work is similar to those in semantic data modeling[Chen76, HK87, PM88, TYF86], which focuses on structural aspects of representing semantics in different constructs. However, we introduced behavior modeling with object-oriented constructs.

Multi-methods in OO languages[BK86, Cham92, MHH91] provide multiple polymorphism and is a well-studied topic for multiple polymorphism and dispatching. But they are not as well organized and have been criticized as not being object-oriented, since they are not tied with any objects. Association class construct makes multiple polymorphism a natural property of association instances, an extension to the current object-oriented model.

Object-role modeling [Bron95, SD96] allows bottom-up analysis, based on local relationship properties. Then the roles can be integrated into classes. Object with roles extension to object-oriented systems [GSR96, RS91, Pern90, Alba93, WCL97] aims at solving the object evolution problem, thus provides more flexibility to the class-based systems. We are unifying them under the entity-role-association scheme, combining the benefits from both.

8 Conclusions

We have proposed entity-role-association as basic constructs for supporting flexible applications and studied the properties and relationships of these constructs.

The ISA relationships among classes are subdivided into ISA and RoleOf relationships. ISA is used for static classification and RoleOf is for dynamic role-playing and object evolution.

A new association class construct is proposed. Its object-oriented features are presented. ISA relationship in association class hierarchy has its special features. Different implementation scheme of associations meet different needs. The intrinsic association allows easy referential integrity maintenance, while the extrinsic association allows easy extension for existing classes and objects. The extrinsic association also allows polymorphism without adding methods into role classes, which can improve the code dependencies among modules.

The role concepts from *object with roles* and *associations* are unified based on the observation that entities play roles in association with other entities. The roles are classified into extrinsic roles and intrinsic roles. Extrinsic roles provide easy extension to existing objects, while intrinsic roles allow players to adapt their behaviors based on the roles they are playing. The different role classification allow developers to choose suitable roles in their implementation.

The entity-role-association scheme supports bottom-up analysis and easy integration of submodels. The proposed constructs not only provide clear semantics, clear interdependencies, and guidance for the implementation with current

OO techniques, also can be supported as language constructs. Our experience shows that our proposed model provides useful software structure, and also results in easy software maintenance.

Acknowledgments

The authors would like to thank Brad Perry and Chih-Cheng Hsu of UCLA for their insightful comments during the writing of the paper.

References

[AGO91] Albano, A., Ghelli, G., and Orsini, R., A relationship mechanism for strongly typed object-oriented database programming language, in *Proceedings of the 17th VLDB conference*, Barcelona, Sept. 1991, pp.565-575.

[Alba93] Albano, A. et al. An object data model with roles, *Proc. 19th VLDB Conf.*, Dublin, Ireland, 1993, pp. 39-51.

[BK86] Bobrew, D. G., and Kahn, K. et al. CommonLoops: Merging lisp and object-oriented programming, In *Proceedings of ACM OOPSLA '86*, pp. 17-29.

[Booc94] Booch, G., *Object-oriented analysis and design with applications*, Benjamn/Cummings, Redwood City, CA, 1994.

[Brat91] Bratsberg, S. E., FOOD: Supporting explicit relations in a fully object-oriented database, in *Object-oriented databases: Analysis, design & construction (DS-4)*, Proceedings of the IFIP TC2/WG 2.6 Working Conference, Windermere, UK, July 1990, pp. 123-140.

[Bron95] Bronts, G.H.W.M. et al., A unifying object role modelling theory, *Information Systems*, Vol. 20, No. 3, 1995, pp. 213-235.

[Catt96] Cattell, R. G. G. (Ed.), *The object database standard: ODMG-93 Release 1.2*, Morgan Kauffmann Publishers, Inc. San Francisco, California, 1996.

[Cham92] Chambers, C., Object-oriented multi-methods in Cecil, in *Proceedings of ECOOP '92*, LNCS 615, pp. 33-56.

[Chen76] Chen, P. P., The entity-relationship model: toward a unified view of data, *ACM TODS* Vol.1, No.1, March, 1976, pp.9-36.

[Chu96] Chu, W. W., Yang, H., Chiang, K., Minock, M. Chow, G., Larson, C., CoBase: A Scalable and Extensible Cooperative Information System, *Journal of Intelligent Information Systems*, 1996.

[CID96] CoBase Internal Document, CoSQL Specification Report, CoBase Research Lab., Computer Science Department, UCLA, 1996.

[DG91] Diaz, O. and Gray, P. M. D., Semantic-rich user-defined relationship as a main constructor in object-oriented databases, in *Object-oriented databases: Analysis, design & construction (DS-4)*, Proceedings of the IFIP TC2/WG 2.6 Working Conference, Windermere, UK, July 1990, pp. 207-224.

[Ditt90] Dittrich, K. R., Object-oriented database systems: the next miles of the marathon, *Information Systems*, Vol. 15, No. 1, pp. 161-167, 1990.

[GHJV95] Gamma, E., Helm, R., Johnson, R. and Vlissides, J., *Design patterns: elements of reusable object-oriented software*, Addison-Wesley, Reading, MA, 1995, pp.331-344.

[GSR96] Gottlob, G., Schrefl, M. and Röck, B., Extending object-oriented systems with roles, *ACM TOIS*, Vol. 14, No. 3, July 1996, pp.268-296.

[HK87] Hull, R. and King, R., Semantic database modeling: survey, applications, and research issues, *ACM Computing Surveys*, Vol. 19, No.3, Sept. 1987, pp. 201-260.

[Ing86] Ingalls, D. H. H., A simple technique for handling multiple polymorphism, in *OOPSLA '86 Proceedings*, Sept. 1986, pp. 347-349.

[MHH91] Mugridge, W. B., Hammer, J. and Hosking, J. G., Multi-Methods in a statically-typed programming language, in *Proceedings of ECOOP '91*, pp.307-324.

[NQZ91] Nassif, R., Qiu, Y., and Zhu, J., Extending the object-oriented paradigm to support relationships and constraints, in *Object-oriented databases: Analysis, design & construction (DS-4)*, Proceedings of the IFIP TC2/WG 2.6 Working Conference, Windermere, UK, July 1990, pp. 305-330.

[Pern90] Pernici, B., Objects with Roles, *Proc. ACM Conf. on Office Information Systems*, 1990, pp.205-215.

[PM88] Peckham, J. and Maryanski, F., Semantic data models, *ACM Computing Surveys*, Vol.20, No. 3, Sept. 1988, pp. 153-189.

[Rat97] Rational Software Corporation, UML semantics v1.0, Jan. 1997.

[RS91] Richardson, J. and Schwarz, P., Aspects: Extending objects to support multiple, independent roles, *Proc. ACM SIGMOD '91 Conf.*, May 1991, pp. 198-307.

[Rumb87] Rumbaugh, J., Relations as semantic constructs in an object-oriented language, in *OOPSLA '87 Proceedings*, Oct. 1987, pp. 466-481.

[Rumb91] Rumbaugh, J. et al., *Object-oriented modeling and design*, Prentice-hall, Englewood Cliffs, New Jersy, 1991.

[SD96] Snoeck, M. and Dedene, G., Generalization/specialization and role in object oriented conceptual modeling, *Data & Knowledge Engineering*, Vol. 19, 1996, pp. 171-195.

[Taiv96] Taivalsaari, A., On the Notion of Inheritance. *ACM Computing Surveys*, Sept. 1996, Vol. 28, No. 3, pp.438-479.

[Tan95] Tanzer, C. , Remarks on object-oriented modeling of associations, *JOOP*, Vol. 7, No. 9, Feb. 1995, pp. 43-46.

[TYF86] Teorey, T. J., Yang, D. and Fry J. P., A logical design methodology for relational databases using the extended entity-relationship model, *ACM Computing Surveys*, Vol. 18, No.2, June 1986, pp.197-222.

[WCL97] Wong, R. K., Chau, H. L., and Lochovsky, F. H., A data model and semantics of objects withdynamic roles. In *13th IEEE International Conference on Data Engineering*, April, 1997.