Extending Object-Oriented Systems with Roles

GEORG GOTTLOB
Vienna University of Technology, Austria
and
MICHAEL SCHREFL and BRIGITTE RÖCK
University of Linz, Austria

In many class-based object-oriented systems the association between an instance and a class is exclusive and permanent. Therefore these systems have serious difficulties in representing objects taking on different roles over time. Such objects must be reclassified any time they evolve (e.g., if a person becomes a student and later an employee). Class hierarchies must be planned carefully and may grow exponentially if entities may take on several independent roles. The problem is even more severe for object-oriented databases than for common object-oriented programming. Databases store objects over longer periods, during which the represented entities evolve. This article shows how class-based object-oriented systems can be extended to handle evolving objects well. Class hierarchies are complemented by role hierarchies, whose nodes represent role types an object classified in the root may take on. At any point in time, an entity is represented by an instance of the root and an instance of every role type whose role it currently plays. In a natural way, the approach extends traditional object-oriented concepts, such as classification, object identity, specialization, inheritance, and polymorphism in a natural way. The practicability of the approach is demonstrated by an implementation in Smalltalk. Smalltalk was chosen because it is widely known, which is not true for any particular class-based object-oriented database programming language. Roles can be provided in Smalltalk by adding a few classes. There is no need to modify the semantics of Smalltalk itself. Role hierarchies are mapped transparently onto ordinary classes. The presented implementation can easily be ported to object-oriented database programming languages based on Smalltalk, such as Gemstone's OPAL.

Categories and Subject Descriptors: D.1.5 [**Programming Techniques**]: Object-Oriented Programming; D.2.10 [**Software Engineering**]: Design—*methodologies*; *representation*; D.3.3 [**Programming Languages**]: Language Constructs and Features; H.2 [**Database Management**]: General; H.2.1 [**Database Management**]: Logical Design—*data models*; H.2.3 [**Database Management**]: Languages—*database* (*persistent*) *programming languages*; H.2.8 [**Database Management**]: Database Applications

General Terms: Design, Languages

Authors' addresses: G. Gottlob, Department of Information Systems, Christian Doppler Laboratory of Expert Systems, Vienna University of Technology, Paniglgasse 16, A-1040 Wien, Austria; email: gottlob@dbai.tuwien.ac.at; M. Schrefl and B. Röck, Department of Information Systems, Data and Knowledge Engineering, University of Linz, Altenbergerstrasse 69, A-4040 Linz, Austria; email: schrefl@dke.uni-linz.ac.at.

Additional Key Words and Phrases: Delegation, inheritance, object-oriented databases, object specialization, roles

1. INTRODUCTION

Object-oriented systems allow the real world to be represented more directly than do conventional ones. Objects model the structure as well as the behavior of real-world entities by means of a set of variables and a set of methods, respectively.

The object-oriented approach uses two important abstraction principles for structuring designs: classification and generalization. *Classification* is an abstraction principle according to which similar objects are grouped into classes defining the structure and behavior of their instances. *Generalization* is an abstraction principle according to which classes are organized into a class hierarchy such that properties shared by several classes are abstracted out into a common superclass. *Specialization*, the inverse of generalization, is used to refine an existing class into more specific subclasses.

Class-based object-oriented systems, such as the object-oriented programming language Smalltalk [Goldberg and Robson 1989] and the object-oriented database programming language OPAL [Butterworth et al. 1991] represent real-world entities as instances of the most specific class in which they can be classified. The association between an instance and a class is exclusive and permanent. Therefore, this approach is appropriate only if the real-world entities to be modeled can be partitioned into a set of disjoint classes and never change their class. Classification hierarchies of plants, of animals, or of technical parts are good examples.

Many applications, however, are dynamic and encompass entities that evolve over time. Persons are the most illustrative example. A person may take on different roles, become a student, an employee, a department manager, and so forth. But not only persons evolve in time; so do office documents or products in production lines.

Representing and maintaining *evolving objects* using class hierarchies that require an object to be classified into a single class is a tedious task. Entities need to be reclassified whenever they evolve. An instance must be created of the class to which an entity belongs in the evolved state; relevant information from the instance representing the old state of the entity must be copied to the new instance; all references to the old instance must be reset to the new instance; and finally the obsolete instance must be deleted.

The problem is even more cumbersome if an entity can take on several roles independently. In order to support an exclusive classification of objects, a separate class must be defined for every possible combination of roles. These *intersection classes* are usually defined by means of multiple inheritance. Languages that do not support multiple inheritance have serious difficulties in handling evolving objects, as they stick to an exclusive classification hierarchy.

Evolving objects can be handled most naturally by applying specialization at the instance level rather than at the class level. A real-world entity is represented by several objects, each representing it in a particular role. The objects of one entity are organized in an object hierarchy in which more specialized objects inherit variable values and methods from more general ones, e.g., somebody being a student and employed will be represented by a person object, a student object, and an employee object corresponding to the birth certificate, the enrollment record, and the employment record in the real world. The student object and the employee object inherit name and birth date from the person object.

Such an approach is usually followed by classless object-oriented systems based on prototypes, in which instances inherit from each other. Whereas class-based systems can take advantage of the uniform structure of the instances of a class to store and access them efficiently, prototype-based systems cannot.

In this article, we show how evolving objects can be handled nicely in class-based systems as well. The key notion is that of a *role hierarchy*. A role hierarchy is a tree of special types, called *role types*. The root of this tree defines the time-invariant properties of an object. The other nodes represent properties (types) that the object may acquire and lose during its lifetime. The hierarchical organization reflects refinement, as will be explained later. At any point in time, an entity is represented by an instance of the root type and an instance of every role type whose role it currently plays. If an entity acquires a new role, a role-specific instance of the appropriate role type is created; if it abandons a role, the role-specific instance is destroyed. Role hierarchies extend traditional object-oriented concepts such as classification, object identity, specialization, inheritance, and polymorphism in a natural way. We discuss each of these concepts in detail in this article.

Consolidating the class-based and the prototype-based paradigm using *object specialization* (role hierarchies) has been proposed independently by Schrefl [1988] and Sciore [1989]. The proposed approaches differ in the degree of freedom in which objects may participate in object hierarchies. In Sciore's model an object hierarchy may be cloned from a *template hierarchy*, but it is potentially unique in structure and behavior. In Schrefl's model possible object hierarchies must be predefined by *role specialization classes* at the type level. Thus Sciore's model is biased toward the prototype-based approach and Schrefl's model toward the class-based approach. Whereas the former is more appropriate for experimental phases of system development, the latter is usually followed in database design.

More recently, Stein and Zdonik [1990] and Wieringa and Jonge [1991] have pointed out that objects may reference a particular role of an object and not only an object itself. Richardson and Schwarz [1991] have introduced the concept of *aspects* to support modeling of roles in strongly typed object-oriented database systems. They use a model in which type implementations and type interfaces are defined separately without any explicit relationship. A particular type implementation belongs to a type interface if it *conforms* to it. An aspect is defined by a special type implementation that extends the abstract data type of the base object. Any object conforming to the abstract data type of the base object may acquire the aspect, i.e., take on the role modeled by the aspect. The model proposed by Richardson and Schwarz [1991] does not consider inheritance and has no explicit operators for switching between roles.

The importance of roles during object-oriented analysis has been pointed out by Pernici [1990], Papazoglou [1991], and recently by Martin and Odell [1992]. Martin and Odell propose an implementation of roles using *object slicing*, in which a real-world entity is represented by a conceptual object and several implementation objects. Object slicing is essentially a special form of object specialization suggested in Schrefl [1988]. It corresponds to an object specialization hierarchy of height two, with the conceptual object being the root and the implementation objects its descendants.

The restriction that an object be associated with a single most-specific type was first levied in Iris [Fishman et al. 1987]. Iris allows an object to belong to several types and to gain or lose types during its lifetime. But it misses the possibility of role-specific behavior: real-world entities are represented by a single object; thus the entire set of types the object belongs to is visible in every context. Consequently, two roles of an object may not have different methods of the same name.

Independently, Albano et al. [1993] have developed an object data model with roles and have implemented this model in a new language called Fibonacci. At first sight, Albano's model and our model, which is an extension of our own previous work [Gottlob et al. 1990; Schrefl and Neuhold 1988], have several features in common. The most important ones are: (1) a role hierarchy can be associated to a root class, and an object in this class can play any role belonging to the hierarchy; (2) messages are resolved according to roles the object plays. Taking a closer look, the models exhibit significant differences, which will be explained in detail within this article. One such difference is the resolution mechanism for methods. We will argue why we consider our approach semantically more meaningful. Other major contributions of our model are the possibility of an object to appear repeatedly in the same type of role and the possibility to combine object classes and role types orthogonally. Similar features are not present in Fibonacci.

The major difference of our approaches, however, is in the motivation. Albano's important contribution is to show how, using Fibonacci, the role concept can be supported in a new language built from scratch. Using Smalltalk, we demonstrate how existing object-oriented languages can be easily extended with roles. We show that roles can be handled in Smalltalk by adding a few classes. There is no need to modify the semantics of Smalltalk itself. Our work also shows that there is no need to give up a favorite class-based object-oriented language because it does not directly support roles. We chose Smalltalk as a typical and widely known representative of class-based languages. Our findings, however, also apply to object-oriented database programming languages based on Smalltalk.

A concept closely related to roles is class migration. It refers to the change of the class of an object [Li and McLeod 1988]. Class migration is also important in schema evolution [Banerjee et al. 1987; Nguyen and Rieu 1989; Skarra and Zdonik 1986] if instances of an old version of a class must be migrated to a newer version.

The remainder of the article is organized as follows. We portray the characteristic features of the role concept in Section 2. In Section 3, we show that representing roles by ordinary class hierarchies is difficult. In Section 4, we introduce role hierarchies. We discuss the representation of roles in Smalltalk; we show how role hierarchies extend conventional object-oriented concepts in a natural way, and we demonstrate how role hierarchies support the characteristic features of the role concept in a very flexible way. In Section 5, we extend our discussion to roles in which entities may occur repeatedly. In Section 6, we discuss the combination of conventional class hierarchies and role hierarchies. We present the overall idea of the implementation in Section 7. The most important parts of code for extending Smalltalk with roles are given in Section 8, along with a conclusion and summarization of the main results.

2. CHARACTERISTIC FEATURES OF ROLES

Analyzing the dynamic nature of entities and their ability to play various roles leads to the following characteristic features of the role concept:

—*Various roles of an entity may share common structure and behavior*, e.g., the student role and the employee role of a person share his/her name and birth date.

—*Entities can acquire and abandon roles dynamically*, e.g., a person being an employee may be promoted to a department manager and may later be demoted if the managerial job is not performed satisfactorily.

—*Roles can be acquired and abandoned independently of each other*, e.g., a person can become a student independently of being an employee.

—*Entities exhibit role-specific behavior*, e.g., different phone numbers may apply to persons in their private, student, and employee roles.

—*Roles restrict access to a particular context*, e.g., an employee's salary is not accessible if the person is viewed in the student role.

—*Entities may occur repeatedly in the same type of role*, e.g., an employee may become a project manager of several projects. Each of these projects may require different skills and give different responsibilities.

3. CLASS HIERARCHIES

Representing entities and their roles using ordinary class hierarchies à la Smalltalk is difficult. Class hierarchies can only cope efficiently with sharing structure and behavior, but they have serious difficulties with the remaining characteristic features of roles:

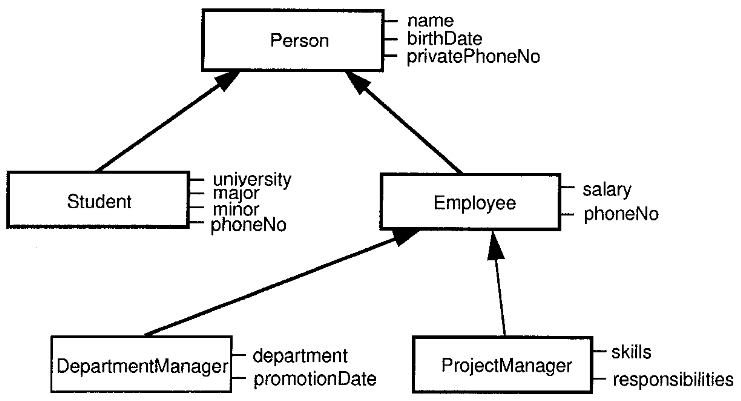

Fig. 1. A class hierarchy.

—*Class hierarchies support sharing of structure and behavior among several classes due to inheritance* . Instance variables and instance methods defined in some class are inherited by its subclasses. Figure 1 shows a class hierarchy, in which classes are depicted by rectangles and subclass relationships by solid arrows pointing from the subclass to the superclass. For simplicity, only instance variables of classes are depicted. Instance methods are not shown. The class hierarchy expresses that students and employees are persons and that department managers and project managers are employees. The class Person defines the three instance variables name, birthDate, and privatePhoneNo. These are inherited by the subclass Employee, which defines the two additional instance variables salary and phoneNo.

In a class hierarchy, every real-world entity is represented as an instance of the most specific class for which it qualifies. An instance stores a value for each instance variable defined in or inherited by its class. Figure 2 shows two instances of the class hierarchy of Figure 1. Mr. Miller is a person, but neither a student nor an employee. He is represented as an instance of class Person with instance variables name, birthDate, and privatePhoneNo. Mrs. Smith is an employee, but neither a department manager nor a project manager. She is represented as an instance of class Employee with instance variables name, birthDate, and privatePhoneNo, which are inherited by the class Employee from its superclass Person, and with instance variables salary and phoneNo defined at class Employee.

—*Tracking evolving objects is a tedious task.* If an entity acquires or abandons a role, it must be reclassified, e.g., promoting Mrs. Smith to a department manager involves the following steps (cf. Figure 3).

(1) An instance of class DepartmentManager must be created.
(2) The instance variables of the instance of Employee that represents Mrs. Smith must be copied into the corresponding instance variables of the new instance of DepartmentManager.

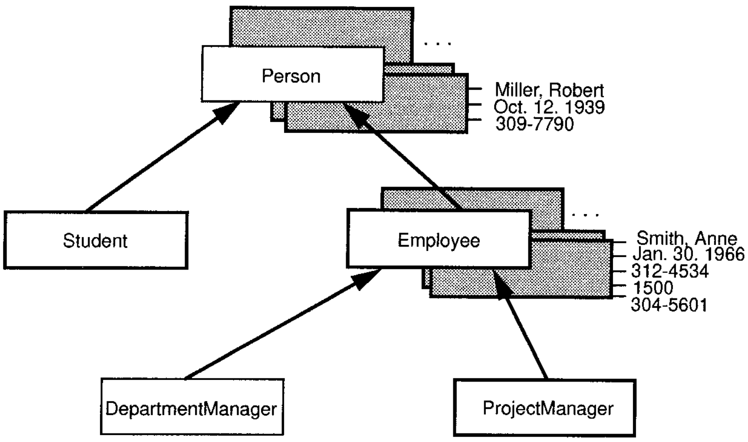

Fig. 2. An extension of a class hierarchy.

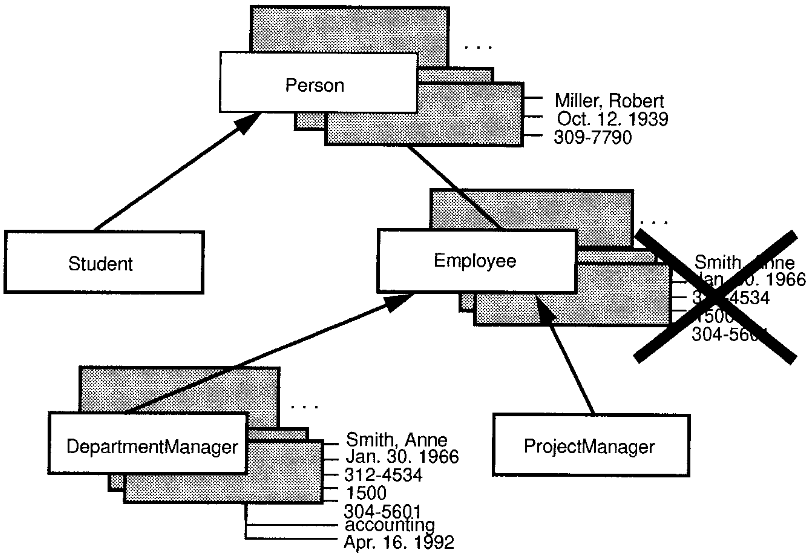

Fig. 3. Evolving an object in a class hierarchy.

(3) All references to the Employee instance of Mrs. Smith must be reset to the new DepartmentManager instance.

(4) The Employee instance of Mrs. Smith must be deleted.

Figure 4 shows the corresponding Smalltalk code. It should be self-explanatory, except for the message become. The message oldInstance become: newInstance is system defined. It switches all references pointing to the receiver (oldInstance) to the instance supplied as parameter (newInstance). Demoting Mrs. Smith to an employee is equally difficult. The steps necessary to reflect the demotion are similar to those described above for promoting Mrs. Smith.

```
" Create an instance of DepartmentManager representing Mrs. Smith "
depMgSmith <- DepartmentManager new.


" Copy instance variables "

depMgSmith name: (empSmith name).
depMgSmith birthDate: (empSmith birthDate).
depMgSmith privatePhoneNo: (empSmith privatePhoneNo).
depMgSmith salary: (empSmith salary).
depMgSmith phoneNo: (empSmith phoneNo).


" Reset all references to empSmith to depMgSmith "

empSmith become: depMgSmith.
```

Fig. 4. Evolving an object in Smalltalk.

—*Class hierarchies must be planned carefully and may grow exponentially if entities can take on several roles* . A separate class must be defined for every possible combination of roles. In our example, a class Student_Employee must be defined to take into account that students may also be employees. A person being a student as well as an employee is represented by an instance of that class. Furthermore, a class Department_and_ProjectManager must be defined if a department manager may also be a project manager. Supporting every combination theoretically possible leads to an exponential number of classes. In our example, the class Student_ProjectManager must be defined to represent persons who are students and project managers. Similarly, the classes Student_DepartmentManager and Student_Department_and_Project-Manager need to be defined, although such combinations might occur seldom if ever. Careful planning is necessary to decide which odd cases a system must be able to handle. If the need for a new class combination is discovered only at a late stage of the software life cycle, an expensive modification of both the class hierarchy and the methods using these classes may become necessary.

Intersection classes, such as Student_Employee, are usually defined by means of multiple inheritance. The class Student_Employee is made a subclass of the classes Student and Employee, inheriting instance variables and instance methods from both. Naming conflicts must be resolved, if Student and Employee define variables or methods of the same name. In our example, Student and Employee define an instance variable phoneNo. One way to resolve the conflict is by renaming phoneNo of Employee to business-PhoneNo. Languages that do not support multiple inheritance, such as Smalltalk, can define intersection classes by inheriting only from one class and by copying instance variable and method definitions from the other. This solution, however, introduces redundancies that complicate later extensions and modifications.

—*Role-specific behavior is not supported by class hierarchies.* Role-specific behavior refers to the capability of an object to respond to a message from the perspective of its particular role. A person will provide a different

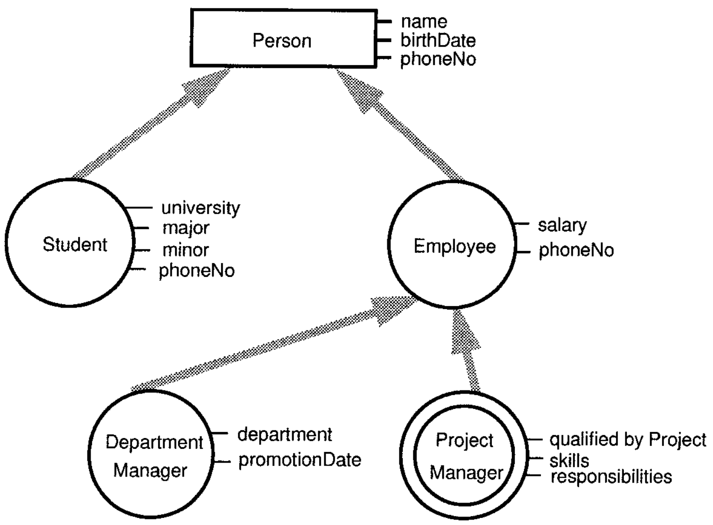

Fig. 5. A role hierarchy.

phone number depending on whether the context is private or business. Such context-dependent behavior is not supported by class hierarchies in Smalltalk.

—*Context-dependent access restrictions are not supported by class hierarchies.* As objects cannot be viewed in particular roles, the entire set of properties of an object is always accessible.

—*Class hierarchies cannot handle multiple occurrences of one entity in the same type of role.* An entity is usually represented only once as an instance of a class. If the entity is represented by several instances, however, the information that these instances belong to one real-world entity gets lost.

4. ROLE HIERARCHIES

Class hierarchies have serious difficulties in modeling evolving objects because they apply specialization and inheritance at the class level. When objects change their type dynamically, it is more appropriate to apply specialization and inheritance at the instance level. This is the case with role hierarchies. A role hierarchy consists of a tree of role types and defines how some kind of entity may evolve. At the schema level, a role hierarchy looks like a class hierarchy. Every role type defines a set of instance variables and a set of instance methods. The difference with respect to class hierarchies is that a subtype in a role hierarchy does not inherit the definitions of instance variables and instance methods from the supertype. As we will see later, inheritance is defined at the instance level rather than at the class level. Figure 5 shows a role hierarchy corresponding to the class hierarchy of Figure 1. Role types are depicted by circles. Every role type is connected by a shaded arrow to the class or role type whose objects

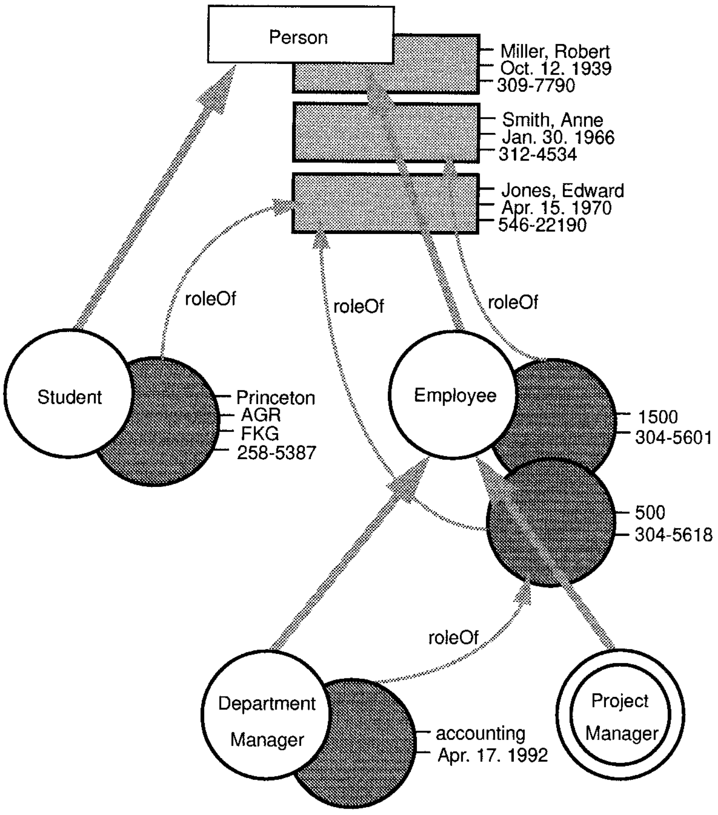

Fig. 6. An extension of a role hierarchy.

may take on that role. Double circles represent qualified roles, which are discussed later.

At the instance level, a real-world entity is represented as an instance of the root type and as an instance of every role type for which it currently qualifies. As definitions of instance variables are not inherited between role types, every instance of a role type stores only values of instance variables defined for that role. An instance of a subtype and an instance of the supertype that represents the same real-world entity are related by a roleOf relationship. Although a subtype in a role hierarchy does not inherit definitions of instance variables and instance methods from its supertype, every instance of the subtype inherits from its corresponding instance of the supertype. Every message not understood by the former is delegated to the latter. Figure 6 shows an extension of the role hierarchy of Figure 5. Mr. Miller is a person, but neither a student nor an employee. He is represented as an instance of Person with instance variables name, birthDate, and phoneNo. Mrs. Smith is an employee, but neither department manager nor a project manager. She is represented by an instance of Person with instance variables name, birthDate, and phoneNo, as well as by an instance of Employee with instance variables salary and phoneNo. The

class method of RoleType	purpose

`defRoleType: roleTypeName`
`instanceVariableNames: stringOfInstVarNames`
`classVariableNames: stringOfClassVarNames`
`poolDictionaries: stringOfPoolNames`
`category: categoryNameString`
`roleSuperType: nameOfRoleSuperType`
	`Define role type roleTypeName.`

`newRoleOf: anObject`	`Create new role of anObject.`

Fig. 7. Methods for defining role types and instances.

employee instance is linked by a roleOf relationship to the corresponding person instance. Mr. Jones is both student and department manager. He is represented by instances of Person, Student, Employee, and DepartmentManager.

Fibonacci [Albano et al. 1993] uses a different approach to cope with the many-faceted nature of real-world entities. There, a real-world entity is represented by a single object which is internally made of two components: (a) a set of blocks where data and methods are stored and (b) a dispatcher in charge of directing the messages to the appropriate method that produces the answer. Each block represents a role of the object; the dispatcher implements the binding of methods to messages. We will discuss this binding mechanism later. Fibonacci's approach is feasible when a new language is built from scratch. Our approach, which uses several objects to represent a real-world entity, is more appropriate when an existing language is to be extended with roles. It also is the better choice in a distributed system in which, for example, several roles of an entity may be stored at different sites, each using a different object identity (one per role). The latter problem has been addressed in a previous paper [Schrefl and Neuhold 1988].

4.1 Role Definition in Smalltalk

Role hierarchies can be represented in Smalltalk by two classes: ObjectWithRoles and RoleType.

A particular role hierarchy is represented as follows: the root of the hierarchy is represented by a subclass of ObjectWithRoles and every role type by a subclass of RoleType.

The class behavior of RoleType, which is summarized in Figure 7, provides messages for creating new role types and for defining role instances.

—*Defining new role types:* A particular role type can be defined by sending message defRoleType:instanceVariableNames:classVariableNames:poolDictionaries:category:roleSuperType: to class RoleType, where the class representing the supertype in the role hierarchy is provided as parameter in addition to the role-specific instance variables. The supertype must be a subclass of either a

```
ObjectWithRoles
      subclass: #Person
      instanceVariableNames: 'name birthDate phoneNo'
      classVariableNames: ''
      poolDictionaries: ''
      category: 'Example'.


RoleType
     defRoleType: #Employee
     instanceVariableNames: 'salary phoneNo'
     classVariableNames: ''
     poolDictionaries: ''
     category: 'Example'
     roleSuperType: #Person.


RoleType
     defRoleType: #Student
     instanceVariableNames: 'university major minor phoneNo'
     classVariableNames: ''
     poolDictionaries: ''
     category: 'Example'
     roleSuperType: #Person.
```

Fig. 8. Role type definition in extended Smalltalk.

RoleType or ObjectWithRoles. Figure 8 shows the definition of entity type Person and role types Employee and Student.

Creating role instances: An instance of a role type can be created using a message of type aRoleType newRoleOf:anObject, where anObject becomes the ancestor of the new role type instance. Figure 9 illustrates how the employee and student roles of Mrs. Smith can be defined.

The behavior for handling objects with roles and role instances, which is summarized in Figure 10, provides messages for deleting a role, for switching roles, and for checking whether some role instance is currently playing some other role.

—*Deleting a role:* Message aRoleObject abandon deletes the addressed role object. If an inner node in a role hierarchy of instances is abandoned, the subhierarchy having that node as root is abandoned, too.[1] Figure 9 gives an example.

—*Switching roles:* Three message signatures are provided in our Smalltalk extension for switching the behavioral context of an entity: Message anObject roleOf returns the ancestor of the receiver in the role hierarchy—e.g., message studentSmith roleOf retrieves Mrs. Smith as an instance of Person. Message anObject root returns the root of the receiver's role hierarchy—e.g., message employeeSmith root retrieves Mrs. Smith as Person. Message anObject as: aRoleType retrieves that instance of aRoleType that represents the same real-world entity as anObject. An

[1] Smalltalk has no explicit function to remove an object no longer being used. An object is reclaimed automatically when there are no references to it from other objects. The purpose of the message abandon is to remove all internal references to the receiver. Various internal references are established between objects in a role hierarchy to speed up switching between roles and testing for entity equivalence. This will be discussed later.

```
" Define root instance "

personSmith <- Person new.
personSmith name: 'Smith, Anne'.
personSmith phoneNo: '312-4534'.
personSmith birthDate: (Date newDay: 30 month: #jan year: 66).

" Define employee role "

empSmith <- Employee newRoleOf: personSmith.
empSmith salary: 1500.
empSmith phoneNo: '304-5601'.


" Define student role "

studentSmith <- Student newRoleOf: personSmith.
...


" Smith graduates "

studentSmith abandon.
```

Fig. 9. Role instance definition in extended Smalltalk.

instance method of ObjectWithRoles and RoleType	purpose
root	Retrieve root of role hierarchy.
roleOf	Retrieve ancestor in role hierarchy.
as: aRoleType	Switch to role aRoleType.
existsAs: aRoleType	Does receiver have role aRoleType?
entityEquiv: anObject	Does receiver represent the same real-world entity as anObject?
instance method of RoleType	**purpose**
abandon	Quit this role.

Fig. 10. Methods for handling roles and objects with roles.

error is raised if no such instance exists—e.g., message studentSmith as: Employee retrieves that instance of Employee representing Mrs. Smith.

—*Checking on role existence:* Message anObject existsAs: aRoleType checks whether an instance of aRoleType exists that represents the same real-world entity as anObject—e.g., message aStudent existsAs: Employee can be used for checking whether a given student is also an employee. Message anObject1 entityEquiv: anObject2 can be used for checking whether anObject1 and anObject2 represent the same real-world entity—e.g., message aStudent entityEquiv: anEmployee checks whether aStudent and anEmployee represent the same person.

The instance behavior of class ObjectWithRoles provides the same messages for switching roles and for checking on role existence as class RoleType. Thus, despite being represented by instances of different classes, the objects at the root and at the inner nodes of a role hierarchy conform to a common protocol.

Our inheritance mechanism is different from that of Fibonacci. We propose to delegate a message which cannot be handled by some role instance to the more general instance in the role hierarchy. Fibonacci's alternative resolution mechanism for methods first goes down the role hierarchy. The most specialized behaviors prevail; the dispatch table of an object is updated each time a new role is acquired. When among the methods to choose there is not a most specialized one, the most recently acquired one is chosen. For example, if the Person instance Edward Jones receives the messages phoneNo (cf. Figure 6), the phone number of his student role or the phone number of his employee role is returned, depending which role was acquired more recently.

We have deliberately avoided a simple priority-based upward inheritance mechanism from more special roles to more general roles. As pointed out in a previous paper by Schrefl and Neuhold [1988], selecting one subrole from several possible ones often gives undesired semantics. Consider, for example, two roles Entrepreneur and Employee of Person, each defining a method income. Now, a person's income might not be fully reflected in self-employment income or in employee income alone, nor in the latest acquired role. Rather, a person's income is an aggregate of the incomes of all roles. Aggregation is one choice, but not always the most meaningful one. Therefore this article identifies several frequently occurring semantic relationships between methods of corresponding objects (sibling roles), each giving rise to a different strategy for upward inheritance. A particular upward-inheritance strategy is chosen by specifying one of these predefined semantic relationships for each pair of corresponding methods in the database schema. Each of these inheritance strategies can be programmed using our methods for role switching and for checking role existence.

It should be noted that Fibonacci provides an alternative binding mechanism, called strict binding, which must be explicitly requested by a special operator when a message is passed to a role. If strict binding is requested, method lookup proceeds only upward in the role hierarchy. This avoids undesired upward inheritance.

4.2 Object-Oriented Concepts and Role Hierarchies

Classification, object identity, specialization, and polymorphism are the most prominent concepts of object-oriented systems. In the following, we show how role hierarchies extend them in a natural way.

—*Classification:* Classification is supported in two ways:

(1) *Entities are classified into a set of disjoint entity types represented by object classes.* Every entity type may be the root of a role hierarchy and defines the structure and behavior common to all roles of any entity of that type.

(2) *Entities are classified into a set of overlapping role types.* Every entity is represented by an instance of every role type it belongs to. A

```
anEmployee <- empSmith.
aStudent <- studentSmith.

" Checking for role identity "

aStudent == studentSmith.
    true

aStudent == anEmployee.
    false

" Checking for entity equivalence "

aStudent entityEquiv: studentSmith.
    true

aStudent entityEquiv: anEmployee.
    true
```

Fig. 11. Checking for role identity and entity equivalence.

role type defines the structure and behavior of an entity that is specific to the particular role.

—*Object identity:* The notion of object identity has to be revised in the realm of roles, because real-world entities are represented by several objects. More specific notions are needed:

(1) *Entity identity:* Every real-world entity is identified by a unique system-defined identifier. Entity identity is provided by the object identity of the root instance of a role hierarchy.

(2) *Role identity:* Every particular role of a real-world entity is identified by a unique system-defined identifier. Role identity is provided by the object identity of the instance representing the entity in that role. Figure 11 shows an example of checking for role identity.

(3) *Entity equivalence:* Two instances are *entity equivalent* if they correspond to the same real-world entity. This is the case if they have a common root instance. Message anObject1 entityEquiv: anObject2, introduced above, can be used to check for entity equivalence (cf. Figure 11).

—*Specialization and inheritance:* Specialization and inheritance apply at the instance level rather than at the class level—e.g., the instance representing Mrs. Smith as an Employee inherits from its corresponding instance of Person. Every message not understood by the former is delegated to the latter. Thus the instance representing Mrs. Smith as an Employee inherits the value of the property name from the instance representing Mrs. Smith as a Person (cf. Figure 12).

—*Polymorphism and behavioral contexts:* Polymorphism means that a single message selector may refer to different method code. Polymorphism has traditionally been used to handle a message differently for different kinds of entities. But messages sent to entities of the same kind are always handled the same way. This is different with roles. The same message sent to different yet entity-equivalent instances may yield different results. Every role type defines a separate behavioral context—

```
" empSmith inherits name from personSmith "

empSmith name.
   Smith, Anne
```

Fig. 12. Instance-level inheritance.

e.g., message phoneNo sent to the instance representing Mrs. Smith as Person will yield her private phone number. The same message sent to the instance representing Mrs. Smith as Employee will yield her business phone number (cf. Figure 13).

4.3 Flexibility of Role Hierarchies

We have seen that representing roles using ordinary class hierarchies is difficult. In the following, we show how role hierarchies support the characteristic features of the role concept in a very flexible way:

—*Role hierarchies support sharing of information between different roles:* Properties shared by several roles are represented in a common ancestor in the role hierarchy. In contrast to class hierarchies, sharing is utilized at the instance level rather than at the class level. This facilitates modeling the remaining features of roles.

—*Tracking evolving objects is easy with role hierarchies:* If an entity acquires a new role, a new instance of the respective role type is created. If an entity abandons a role, its instance of the respective role type is deleted—e.g., promoting employee Mrs. Smith to a department manager involves a single step. An instance of DepartmentManager is created and linked by a roleOf relationship to its corresponding instance of Employee (cf. Figure 14). Similarly, demoting Mrs. Smith thereafter involves a single step. Figure 15 shows the Smalltalk code for promoting Mrs. Smith to department manager and the code for demoting Mrs. Smith thereafter.

—*Several independent roles of an entity can be modeled without using multiple inheritance:* Class hierarchies require every entity to be represented as an instance of a single class. If an entity takes on several roles, each of which is represented by a separate class, they must be combined into a single class using multiple inheritance. Role hierarchies represent an entity by an instance of every role type for which it qualifies. Several roles are combined at the instance level rather than at the class level. Hence, multiple inheritance can be avoided—e.g., if Mr. Jones is a student and an employee, he will be represented by an instance of Person, by an instance of Student, and by an instance of Employee (cf. Figure 6).

—*Role hierarchies support modeling role-specific behavior:* Every instance in a role hierarchy represents a separate behavioral context of an entity. Figure 16 demonstrates the effect of the messages for switching roles introduced earlier.

—*Accessing an entity is restricted to the role in which it is currently viewed:* An instance of a role type can answer every message defined at this type or defined at any ancestor in the role hierarchy. But it does not

```
" Smith as person returns her private phone number "

personSmith phoneNo.
   312-4534


" Smith as an employee returns her business phone number "

empSmith phoneNo.
   304-5601
```

Fig. 13. Behavioral contexts.

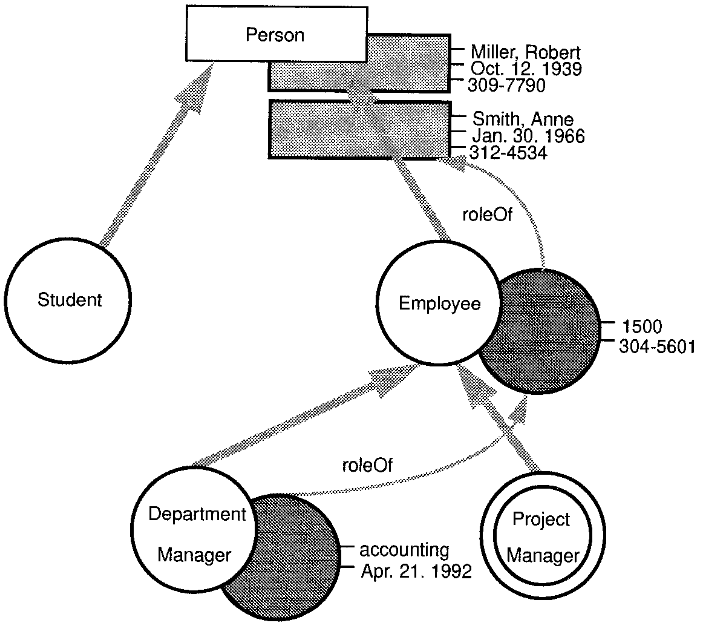

Fig. 14. Evolving an object in a role hierarchy.

```
" Promoting Mrs. Smith "

depMgSmith <- DepartmentManager newRoleOf: empSmith.
depMgSmith promotionDate: today.
depMgSmith department: accounting.

...

" Demoting Mrs. Smith "

depMgSmith abandon.
```

Fig. 15. Evolving an object in extended Smalltalk.

understand messages defined at sibling nodes in the role hierarchy—e.g., the message studentSmith salary will fail, whereas the message (studentSmith as: Employee) salary will return the salary of Mrs. Smith as an employee.

—*An entity may have multiple occurrences of the same type of role:* Such a role is called a *qualified role*. Qualified roles are discussed in detail below.

```
" Selecting phone number of employee Smith "

empSmith phoneNo.
    304-5601


" The roleOf-message returns the ancestor in a role hierarchy.
  Thus the following message returns the private phone number
  of employee Smith "

empSmith roleOf phoneNo.
    312-4534


" The root-message returns the root instance in a role hierarchy "

personSmith <- studentSmith root.
depMgSmith root phoneNo.
    312-4534


" Checking whether Mrs. Smith exists as an employee and, if so,
  selecting her phone number "

(personSmith existsAs: Employee)
    ifTrue: [(personSmith as: Employee) phoneNo].
        304-5601
```

Fig. 16. Switching behavioral contexts.

Fibonacci, in which a real-world entity is represented by a single yet multifaceted object, does not support qualified roles. In the following section we show that this kind of multiple instantiation is easy to implement if a real-world entity is represented by several model-world objects.

5. QUALIFIED ROLES

A role type defines a *qualified role* if a real-world entity may have several occurrences of that role. Occurrences of the same entity are distinguished by the value of a special instance variable named *qualifier*. Note that one could also allow a nonqualified role to be instantiated multiple times by the same entity. However, this would make it more difficult to identify a specific occurrence of that role and to apply a general operator for role switching, as no common discriminating attribute is known.

In our example, ProjectManager is a qualified role. The qualifier takes on the Project supervised.

Figure 17 depicts an extension of the qualified role ProjectManager. Mrs. Smith, who manages two projects, "CAD/CAM" and "ooDB," is represented by two instances of ProjectManager. Each instance models Mrs. Smith as project manager of one of these two projects. Mrs. Smith has different responsibilities in each of them. She is responsible for "reuse management" in the former and for "quality assurance" in the latter.

Qualified role types can be made available in Smalltalk by a predefined subclass of RoleType, QualifiedRoleType. The class behavior of Qualified RoleType, which is summarized in Figure 18, provides messages for defining new qualified role types and for creating instances of qualified role types.

—*Defining new qualified role types:* A particular qualified role type can be defined by sending message defQualifiedRoleType:instanceVariableNames:

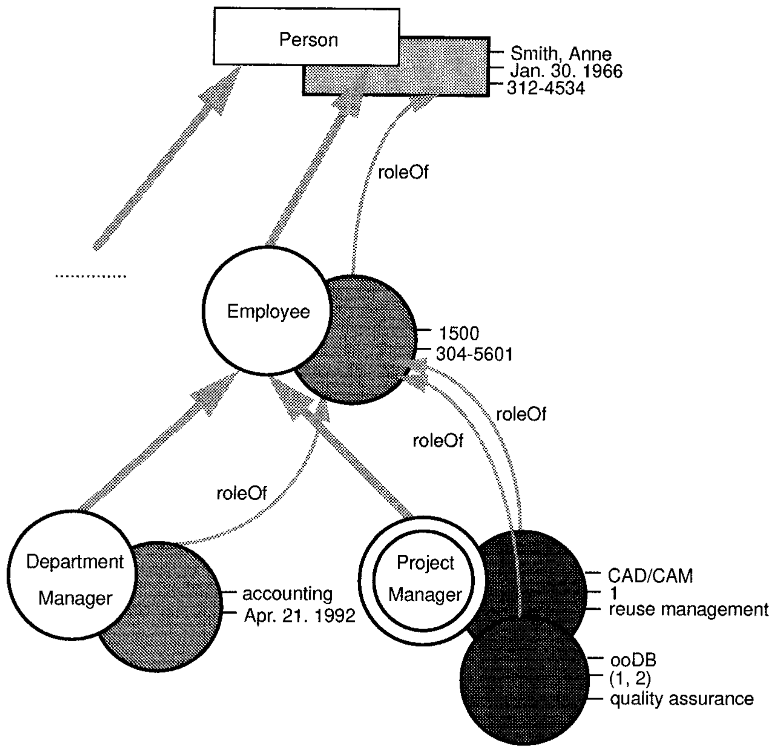

Fig. 17. An extension of a qualified role type.

class method of QualifiedRoleType	purpose

`defQualifiedRoleType: roleTypeName`
`instanceVariableNames: stringOfInstVarNames`
`classVariableNames: stringOfClassVarNames`
`poolDictionaries: stringOfPoolNames`
`category: categoryNameString`
`roleSuperType: nameOfRoleSuperType`
`classOfQualifyingObj: aClass`
	`Define qualified role type roleTypeName.`

`newRoleOf: anObject qualifiedBy: qualifyingObj`	`Create new qualified role of anObject.`

Fig. 18. Methods for defining qualified role types and instances.

classVariableNames:poolDictionaries:category:roleSuperType:classOfQualifyingObj: to class QualifiedRoleType, where the class representing the supertype in the role hierarchy, nameOfRoleSuperType, and the class whose instances are used as qualifiers, classOfQualifyingObj, must be provided as parameters in addition to the role-specific instance variables and instance methods. Figure 19 illustrates the definition of the qualified role type ProjectManager.

—*Creating qualified role type instances:* The method newRoleOf: anObject qualifiedBy: qualifyingObj is predefined for creating new qualified role

```
QualifiedRoleType
        defQualifiedRoleType: #ProjectManager
        instanceVariableNames: 'skills responsibilities'
        classVariableNames: ''
        poolDictionaries: ''
        category: ''
        roleSuperType: #Employee
        classOfQualifyingObj: #Project.
```

Fig. 19. Defining a qualified role type.

instances. Figure 20 illustrates the use of this message. Employee Smith is made manager of two projects.

For handling qualified roles uniformly, the instance behavior of ObjectWithRoles and the instance behavior of RoleType are extended by methods for checking role existence and for switching roles in the realm of qualified roles. The semantics of the messages existsAs: and as: are redefined (cf. Figure 10), and two new messages, existsAs:of: and as:of:, are introduced (cf. Figure 21).

(1) Message anObject existsAs: aQualifiedRoleType returns true if at least one occurrence of aQualifiedRoleType exists for anObject.
(2) Similarly, the message anObject as: aQualifiedRoleType retrieves a set of instances that model the occurrences of aQualifiedRoleType for anObject.
(3) Message anObject existsAs: aQualifiedRoleType of: qualifyingObj checks whether anObject exists as instance of aQualifiedRoleType that is qualified by qualifyingObj.
(4) Message anObject as: aQualifiedRoleType of: qualifyingObj switches to the instance of aQualifiedRoleType that is entity equivalent to anObject and qualified by qualifyingObj.

QualifiedRoleType inherits methods to handle these messages from RoleType and defines an additional method qualifier returning the qualifier of the role.

6. COMBINING CLASS AND ROLE HIERARCHIES

Class hierarchies and role hierarchies serve different purposes. Class hierarchies provide a classification of entities into a set of disjoint entity types. Role hierarchies define different behavioral contexts for entities of the same type.

A class hierarchy can coexist with several role hierarchies. Every class in the class hierarchy may function as the root of a role hierarchy. Roles defined at a nonleaf node of the class hierarchy are inherited by this node's subclasses.

The possibility to combine class hierarchies and role hierarchies orthogonally is a feature unique to our model. Every class in a class hierarchy may be the root of a role hierarchy and every role type the root of a class hierarchy, i.e., roles may be subclassed. Furthermore, the ability to inherit role hierarchies along a class hierarchy like methods and instance vari-

```
oodbProj <- Project new name: 'ooDB'.
...
oodbProjMgSmith <- ProjectManager newRoleOf: empSmith qualifiedBy: oodbProj.
...
cadcamProj <- Project new name: 'CAD/CAM'.
...
cadcamProjMgSmith <- ProjectManager newRoleOf: empSmith qualifiedBy: cadcamProj
```

Fig. 20. Defining a qualified instance type.

instance method of ObjectWithRoles and RoleType	purpose
`as: aQualifiedRoleType of: qualifyingObj`	Switch to role aQualifiedRoleType qualified by qualifyingObj.
`existsAs: aQualifiedRoleType of: qualifyingObj`	Does receiver have role aQualifiedRoleType qualified by qualifyingObj?
instance method of QualifiedRoleType	**purpose**
`qualifier`	Return qualifier of a qualified role.

Fig. 21. Additional methods for handling qualified roles.

ables is an important contribution of our approach. Again, as we will see, this feature is very useful and easy to implement.

Figure 22 shows one class and two role hierarchies. The class hierarchy consists of three object classes: LegalEntity, Person, and Company. The classes LegalEntity and Person are roots of role hierarchies. The first consists of a single role, the qualified role Customer. The second corresponds to the role hierarchy of Figure 5. The role Customer defined in the class LegalEntity is inherited by the subclasses of LegalEntity, Person, and Company. This means that every instance of Person may take on the role Customer as well as every role defined in the role hierarchy of Person (cf. Figure 23).

Classes and roles can be combined orthogonally. Classes may have subclasses and may function as the root of role hierarchies, e.g., the class LegalEntity has subclasses Person and Company and is role supertype of role type Customer. Roles may have subroles and subclasses. We have seen an example of the former. Role type Employee has subroles DepartmentManager and ProjectManager. We get an example of the latter by subclassing role type Student to ForeignStudent (cf. Figure 22). This means that the student role of a person is classified into either Student or ForeignStudent. For foreign students, the country is recorded next to the university, major, minor, and phone number (cf. Figure 23). The latter instance variables are inherited by role type ForeignStudent at the class level from role type Student.

To distinguish subclasses of role types from role types and ordinary object classes, they are depicted by ovals (cf. ForeignStudent in Figure 22).

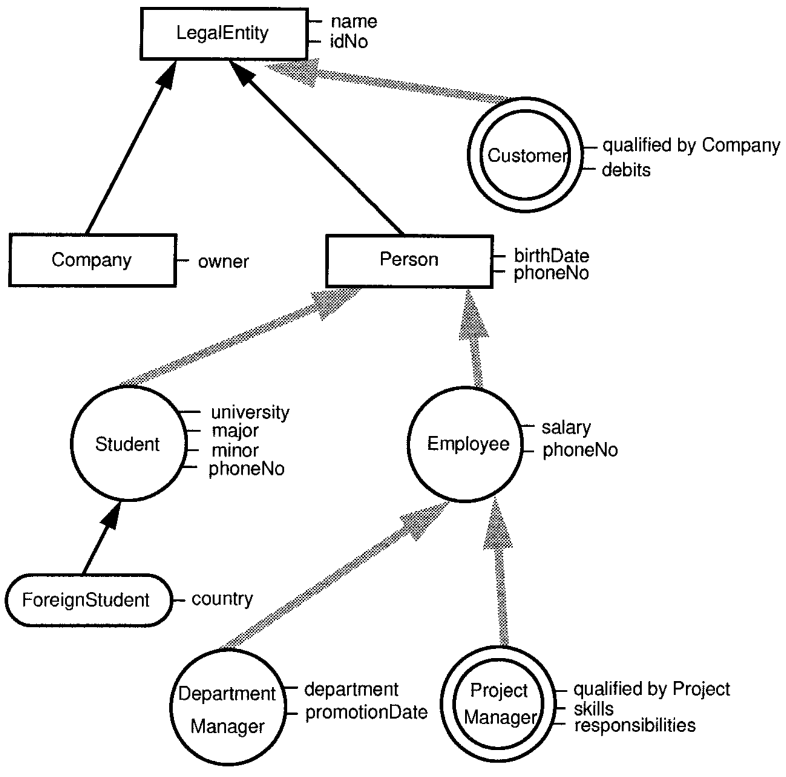

Fig. 22. A class and role hierarchy.

7. THE IMPLEMENTATION

Roles can be made available in Smalltalk by adding three classes: ObjectWithRoles, RoleType, and QualifiedRoleType. The public interfaces of these classes have already been introduced above. In this section we present the overall idea of their implementation. The reader interested in details is referred to the Figures 24–26, which show the instance protocol of the above classes in Smalltalk. The entire code is available per email/ftp from the authors.

(1) Class ObjectWithRoles defines the behavior and structure of objects that may take on roles, i.e., that may be roots of role hierarchies. For selecting a particular role of an object efficiently, the root object keeps a dictionary of role occurrences, roleDictionary, indexed by role type. In case of a qualified role, the dictionary entry contains not a single object, but a set of objects representing all occurrences of the root object in that particular role. If some role is currently not played by an object, no entry will appear for that role in the role dictionary. Internal methods—which are not supposed to be called by user-defined methods—initRoleDictionary, recordNewRole:, and recordNewQualifiedRole: are used by the external methods newRoleOf: and newRoleOf:qualifiedBy:

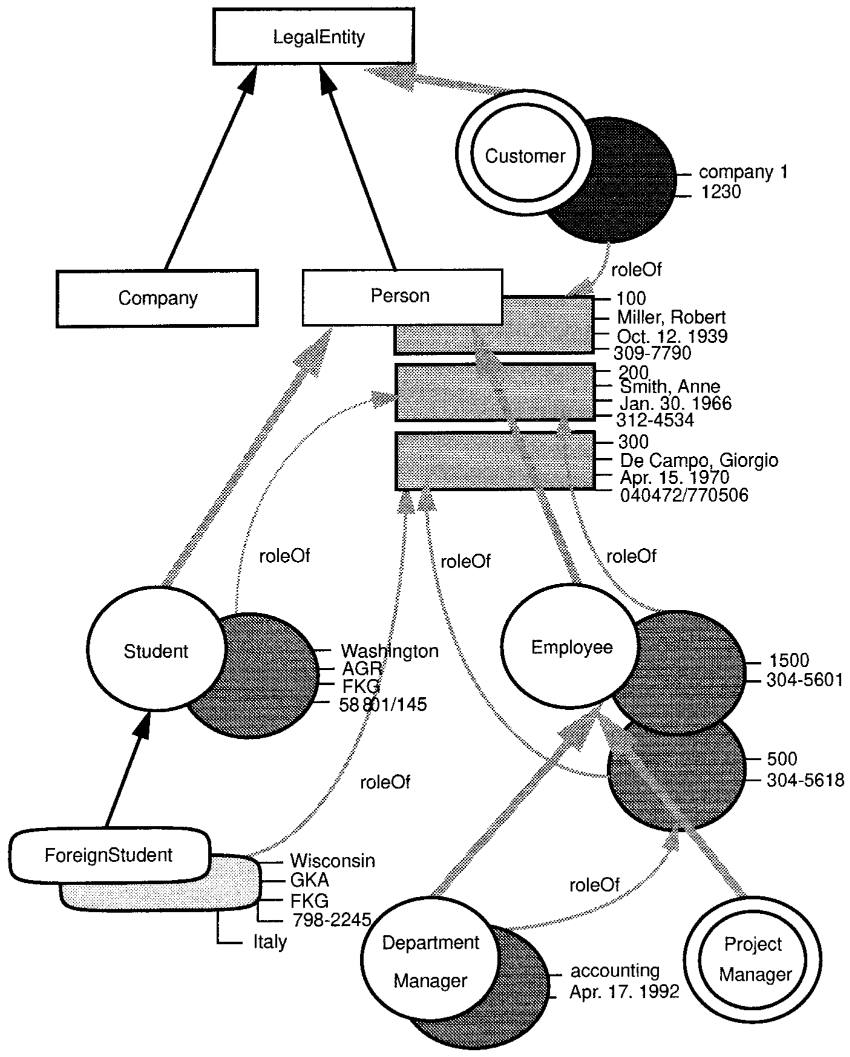

Fig. 23. An extension of a class and role hierarchy.

for initializing the role dictionary and for recording new roles or qualified roles, respectively. Similarly, internal methods cancelRoleAndSubroles: and cancelQualifiedRoleAndSubroles: are used by the external methods abandon of RoleType and abandon of QualifiedRoleType for removing an entire subtree in the role hierarchy. These methods again make use of internal methods cancelRole: and cancelQualifiedRole:, which remove a single entry from the role dictionary.

(2) The instance protocol of class RoleType defines the structure and behavior of objects at inner nodes of a role hierarchy. Two references are kept, one to the ancestor in the role hierarchy and one to the root.

```
Object subclass: #ObjectWithRoles
     instanceVariableNames: 'roleDictionary'
     classVariableNames: ''
     poolDictionaries: ''
     category: 'Roles'

ObjectWithRoles methodsFor: 'accessing'

root
     "return the root of the role hierarchy"
     ^self

ObjectWithRoles methodsFor: 'switching roles'

as: aRoleType
     "retrieve the receiver's entity-equivalent instance of aRoleTye"
     ^roleDictionary at: (aRoleType  roleTypeName)
               ifAbsent: [self error: 'role does not exist']

as: aQualifiedRoleType of: qualifyingObj
     "retrieve the receiver's entity-equivalent instance of aQualifiedRoleType qualified
      by qualifyingObj"
     ^((self as: aQualifiedRoleType)
          detect: [:o | o qualifier == qualifyingObj]
               ifNone: [self error: 'qualified role does not exist'])

ObjectWithRoles methodsFor: 'testing for role existence'

existsAs: aRoleType
     "test whether an instance of aRoleType exists that is entity-equivalent to the receiver"
     ^roleDictionary includesKey: (aRoleType roleTypeName)

existsAs: aQualifiedRoleType of: qualifyingObj
     "test whether an entity-equivalent instance of aQualifiedRoleType, qualified by
      qualifyingObj, exists"
     ^(roleDictionary includesKey: (aQualifiedRoleType roleTypeName)) and:
          [((roleDictionary at: (aQualifiedRoleType roleTypeName))
               detect: [:o | o qualifier == qualifyingObj] ifNone: []) notNil]

ObjectWithRoles methodsFor: 'testing for entity equivalence'

entityEquiv: anObject
     "test if the receiver and anObject represent the same  real-world entity"
       ^self root == (anObject root)

ObjectWithRoles methodsFor: 'internal'
     "no implementation is shown for internal methods"

cancelQualifiedRole: aQualifiedRoleObject
     "remove a qualified role object from the set of roles in the receiver's roleDictionary"

cancelQualifiedRoleAndSubroles: aQualifiedRoleObject
     "remove a qualified role object and its sub-objects from the receiver's roleDictionary"

cancelRole: aRoleObject
     "remove a role object from the receiver's roleDictionary"

cancelRoleAndSubroles: aRoleObject
     "remove a role object and its sub-objects from the receiver's roleDictionary"

initRoleDictionary
     "initialization of instance variable roleDictionary"

recordNewQualifiedRole: aQualifiedRoleObject
     "insert a new qualified role object into the receiver's roleDictionary"

recordNewRole: aRoleObject
     "insert a new role object into the receiver's roleDictionary"
```

Fig. 24. Instance protocol of class ObjectWithRoles.

```
Object subclass: #RoleType
     instanceVariableNames: 'roleOf root'
     classVariableNames: ''
     poolDictionaries: ''
     category: 'Roles'

RoleType methodsFor: 'accessing'

roleOf
     "get ancestor"
     ^roleOf

root
     "get root"
     ^root

RoleType methodsFor: 'abandoning roles'

abandon
     "quit this role"
     root cancelRoleAndSubroles: self

RoleType methodsFor: 'switching roles'

as: aRoleType
     "retrieve entity-equivalent instance of aRoleType"
     ^root as: aRoleType

as: aQualifiedRoleType of: qualifyingObj
     "retrieve entity-equivalent instance, qualified by qualifyingObj, of aQualifiedRoleType
     ^root as: aQualifiedRoleType of: qualifyingObj

RoleType methodsFor: 'testing for entity equivalence'

entityEquiv: anObject
     "check if the receiver and anObject represent the same real-world entity"
     ^root == (anObject root)

RoleType methodsFor: 'testing for role existence'

existsAs: aRoleType
     "check whether an entity-equivalent instance of aRoleType exists"
     ^root existsAs: aRoleType

existsAs: aQualifiedRoleType of: qualifyingObj
     "check whether an entity-equivalent instance of aQualifiedRoleType qualified by
      qualifyingObj exists"
     ^root existsAs: aQualifiedRoleType of: qualifyingObj

RoleType methodsFor: 'inheritance'

doesNotUnderstand: aMessage
     "delegate every message not understood to the ancestor"
     ^roleOf perform: (aMessage selector) withArguments: (aMessage arguments)

RoleType methodsFor: 'internal'

init: ancestor
     "initialize references"
     roleOf <- ancestor.
     root <- ancestor root.
     root recordNewRole: self

nullify
     "set all references to nil"
     roleOf <- nil.
     root <- nil
```

Fig. 25. Instance protocol of class RoleType.

— The reference to the ancestor roleOf is mainly used for inheritance. If a message is not understood by a particular role object, it is delegated to its ancestor in the role hierarchy: in Smalltalk, every

```
RoleType subclass: #QualifiedRoleType
     instanceVariableNames: 'qualifier'
     classVariableNames: ''
     poolDictionaries: ''
     category: 'Roles'


QualifiedRoleType methodsFor: 'accessing'

qualifier
     "return the value of the instance variable qualifier"
     ^qualifier

QualifiedRoleType methodsFor: 'role abandoning'

abandon
     "quit this role"
     root cancelQualifiedRoleAndSubroles: self

QualifiedRoleType methodsFor: 'internal'

init: ancestor
     "not applicable to instances of QualifiedRoleType"
     self shouldNotImplement

init: ancestor qualifiedBy: qualifyingObj
     "initialize the receiver's references in the role hierarchy"
     roleOf <- ancestor.
     root <- ancestor root.
     qualifier <- qualifyingObj.
     root recordNewQualifiedRole: self
```

Fig. 26. Instance protocol of class QualifiedRoleType.

message not understood by the receiver is taken care of by the predefined method doesNotUnderstand:. This method is overridden at RoleType such that every message not understood by a role object is re-sent to its ancestor in the role hierarchy.

— The root reference is kept to speed up testing on entity equivalence and switching between roles. Testing on entity equivalence of two role objects is done efficiently by comparing their root references. Switching between roles is done efficiently using the role dictionary of the root object.

Internal methods init: and nullify are used for initializing and removing these references.

(3) The class protocol of RoleType provides method newRoleOf: for creating a new role object and method defRoleType:instanceVariableNames:classVariableNames: poolDictionaries:category:roleSuperType: for creating a subclass representing a particular role type. The subclass is created with two initialized class variables, RoleTypeName and RoleSuperType, holding the name of the class and the name of the ancestor class in

the role hierarchy and with the access methods for these class variables, roleTypeName and roleSuperType.[2]

(4) Class QualifiedRoleType, which is a subclass of RoleType, defines the additional structure and behavior of qualified roles. The instance variable, qualifier, stores a reference to the qualifying object. Qualified roles are initialized by internal method init:qualifiedBy: rather than by method init: inherited from RoleType.

(5) The class protocol of QualifiedRoleType replaces methods newRoleOf: and defRoleType:instanceVariableNames:classVariableNames:poolDictionaries:category:roleSuperType: inherited from RoleType class by methods newRoleOf:qualifiedBy: and defQualifiedRoleType:instanceVariableNames:classVariableNames:poolDictionaries:category:roleSuperType:classOfQualifyingObj:, which also take the qualifier and the class of the qualifying object into account.

The current implementation has the following restrictions:

(1) Every subrole of a qualified role must be a qualified role, too.

(2) The qualifier of a qualified role must be unique for a given role type and a given entity.

(3) Only names of role types and not subclasses of role types can be used as the names of behavioral contexts in the messages as, existsAs:, as:of:, and existsAs:of:. Note that this is no severe restriction, as the role type always provides a unique name for the behavioral context of an entity, regardless whether the entity-equivalent role instance is a direct instance of the role type or any one of its subclasses—e.g., the message aPerson as: Student will retrieve a foreign student, if the person is a foreign student.

8. SUMMARY AND CONCLUSION

We have shown how class-based object-oriented systems can be extended with roles. In particular, we have shown how roles can easily be implemented as an additional feature in Smalltalk. No changes to the Smalltalk system itself are necessary. Only a few classes need to be added to the predefined Smalltalk class hierarchy.

Our work complements the excellent work of Albano et al. [1993], who have independently worked on the role concept and shown how a role mechanism can be included in a new strongly typed object-oriented database language called Fibonacci. We have pointed out that our approach goes beyond their work in important aspects. Furthermore, in relating our approaches, we have discovered that important features which are not present in Fibonacci, such as multiple instantiation of roles or combining class and role hierarchies, can easily be provided in our approach, which

[2] The access methods cannot be defined once at class RoleType and be inherited to its subclasses, as in Smalltalk class variables accessed by a class method are bound statically to the class implementing the method.

represents a real-world entity by several model-world objects. This difference did not seem to be very significant at first.

Roles in class-based object-oriented systems alleviate the user from the tedious task of modeling multiple, independent roles of an entity using ordinary class hierarchies. Role hierarchies in class-based systems support the concept of roles in a very flexible way:

(1) Different roles of an entity may share common information.
(2) Evolving entities can be tracked easily.
(3) Several independent roles of an entity can be represented without multiple inheritance.
(4) Entities may exhibit role-specific behavior.
(5) Access to an entity is restricted to the role in which it is currently viewed.
(6) An entity may occur several times in the same type of role.

Furthermore, we have shown how role hierarchies extend traditional object-oriented concepts, such as classification, object identity, specialization and inheritance, polymorphism, and behavioral contexts, in a natural way.

While the concept of roles is suitable for common object-oriented programming, it is even more important when data are persistent. In practice, our concepts are already part of a design environment for object-oriented databases [Kappel and Schrefl 1991].

A formal description of the concept of roles using evolving algebras is given in Gottlob et al. [1990].

Future work concerns the specification of allowed sequences of role acquisitions by real-world entities. A good starting point for such an extension is the work of Su [1991].

ACKNOWLEDGMENTS

We like to thank Joachim Hans Fröhlich, Markus Stumptner, and Stefan Vieweg for their valuable comments. We are grateful to Bob Bach for patiently polishing our English.

REFERENCES

ALBANO, A., BERGAMINI, R., GHELLI, G., AND ORSINI, R. 1993. An object data model with roles. In *Proceedings of the International Conference on Very Large Data Bases.* VLDB Endowment Press, Saratoga, Calif., 39–51.

BANERJEE, J., KIM, W., KIM, K. J., AND KORTH, H. 1987. Semantics and implementation of schema evolution in object-oriented databases. In *Proceedings of the ACM SIGMOD Conference.* ACM, New York, 311–322.

BUTTERWORTH, P., OTIS, A., AND STEIN, J. 1991. The Gemstone object database management system. *Commun. ACM 34,* 10 (Oct.), 65–77.

FISHMAN, D. H., BEECH, D., CATE, H. P., CHOW, E. C., CONNORS, T., MAHBOD, B., NEIMAT, M.-A., RYAN, T. A., SHAN, M.-C., DAVIS, D. W., DERRETT, N., HOCH, C. G., KEPT, W., AND LYNGBÆK, P. 1987. Iris: An object-oriented database management system. *ACM Trans. Off. Inf. Syst. 5,* 1, 48–69.

GOLDBERG, A. AND ROBSON, D. 1989. *Smalltalk—80 The Language*. Addison-Wesley, Reading, Mass.

GOTTLOB, G., KAPPEL, G., AND SCHREFL, M. 1990. Semantics of object-oriented data models—The evolving algebra approach. In *Next Generation System Technology,* J. W. Schmidt and A. A. Stogny, Eds. Lecture Notes in Computer Science, vol. 504. Springer-Verlag, Berlin, 144–160.

KAPPEL, G. AND SCHREFL, M. 1991. Object/behavior diagrams. In *Proceedings of the 7th International Conference on Data Engineering*. IEEE Computer Society Press, Los Alamitos, Calif., 530–539.

KLAS, W., NEUHOLD, E. J., AND SCHREFL, M. 1988. On an object-oriented data model for a knowledge base. In *Research into Network and Distributed Applications*. North Holland, Amsterdam.

LI, Q. AND MCLEOD, D. 1988. Object flavor evolution in an object-oriented database system. In *Proceedings of the ACM Conference on Office Information Systems*. ACM, New York, 265–275.

MARTIN, J. AND ODELL, J. J. 1992. *Object-Oriented Analysis and Design*. Prentice-Hall, Englewood Cliffs, N.J.

NGUYEN, G. T. AND RIEU, D. 1989. Schema evolution in object-oriented database systems. *Data Knowl. Eng. 4,* 43–67.

PAPAZOGLOU, M. P. 1991. A methodology for representing multifaced objects. In *Proceedings of the International Conference on Database and Expert Systems Application*. 7–12.

PERNICI, B. 1990. Objects with roles. In *Proceedings of the ACM Conference on Office Information Systems*. ACM, New York, 205–215.

RICHARDSON, J. AND SCHWARZ, P. 1991. Aspects: Extending objects to support multiple, independent roles. In *Proceedings of the ACM SIGMOD Conference*. ACM, New York, 298–307.

SCHREFL, M. 1988. Object-oriented database integration. Ph.D. thesis, Technische Univ. Wien, Austria.

SCHREFL, M. AND NEUHOLD, E. J. 1988. Object class definition by generalization using upward inheritance. In *Proceedings of the IEEE 4th International Conference on Data Engineering*. IEEE, New York, 4–13.

SCIORE, E. 1989. Object specialization. *ACM Trans. Inf. Syst. 7,* 2, 103–122.

SKARRA, A. H. AND ZDONIK, S. B. 1986. The management of changing types in an object-oriented database. In *Proceedings of the International Conference on OOPSLA*. ACM, New York, 483–495.

STEIN, L. A. AND ZDONIK, S. B. 1990. Clovers: The dynamic behavior of types and instances. Tech. Rep., Brown Univ., Providence, R.I.

SU, J. 1991. Dynamic constraints and object migration. In *Proceedings of the 17th International Conference on Very Large Data Bases*. VLDB Endowment Press, Saratoga, Calif., 233–242.

WIERINGA, R. AND JONGE, W. D. 1991. The identification of objects and roles. Tech. Rep., Faculty of Mathematics and Computer Science, Vrije Univ., Amsterdam.

Received February 1993; accepted May 1995