

CONCEPTUAL MODELING OF BUSINESS APPLICATIONS WITH DYNAMIC OBJECT ROLES*

KAZIMIERZ SUBIETA[†], ANDRZEJ JODŁOWSKI[‡], PIOTR HABELA[§], AND JACEK PŁODZIEN[¶]

Abstract. The paper discusses the concept of dynamic object roles as a facility for conceptual modeling and as a data structure to be implemented in object-oriented or object-relational database management systems. We show that the concept is useful for a real business case. We discuss advantages of the concept and known approaches to implement it. A new approach is proposed, which assumes that a role is a distinguished subobject of an object. A role inherits dynamically attribute values and methods of its parent object. Objects can be accessed by their names, as well as by the names of their roles. The role concept essentially changes the semantics of other notions of object-oriented models, such as classes, inheritance and substitutability. In the paper we discuss how dynamic roles could be involved into an object store and in an object definition language built in the spirit of the ODMG standard.

Key words. dynamic object role, object-oriented, conceptual modeling, database schema

1. Introduction. Dynamic object roles have had for several years the reputation of a notion on the brink of acceptance. There are many papers advocating the concept, see e.g. [1], [6], [7], [12], [14], [17], [19], [20], [21], [22], [25], [26], [29], [40], [41], [42]. On the other hand, many researchers consider applications of the concept not sufficiently wide to justify the extra complexity of conceptual modeling facilities. Moreover, the concept is neglected on the implementation side - as far as we know, none of popular object-oriented programming languages or database systems introduces it explicitly. Some authors assume a tradeoff, where the role concept is the subject of special design patterns [8], [12], [13], [30], applied both on the conceptual modeling and the implementation sides.

In our opinion, this view on the concept of dynamic roles should be revised. In this paper we show that the dynamic object roles are useful both for conceptual modeling and implementation. The concept could much facilitate such modeling tools as UML [38] and could be an important paradigm on object databases built e.g. in the spirit of ODMG [23]. The notion is already present (under another name, with specific semantics, and with some limitations) in the standard SQL:1999 [5].

The idea of dynamic object roles assumes that a real or abstract entity during its life can acquire and lose many roles without changing identity. The roles appear during the life of a given object, they can exist simultaneously, and they can disappear at every moment. For example, a certain person can at the same time be a student, a worker, a patient, a club member, etc., Fig. 1.1. Similarly, a building can be an office, a house, a magazine, etc.

The concept assumes that an object is associated with other objects (subobjects), which are modeling its roles. Object-roles cannot exist without their parent object. Deleting an object causes deleting all of its roles. Roles can exist simultaneously

*This work was partly supported by the EU 5th Framework project ICONS, IST-2001-32429.

[†]Polish-Japanese Institute of Information Technology, Warsaw, Poland; Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland (subieta@ipipan.waw.pl).

[‡]Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland (andrzejj@ipipan.waw.pl).

[§]Polish-Japanese Institute of Information Technology, Warsaw, Poland (phabela@pjwstk.edu.pl).

[¶]Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland; Warsaw School of Economics, Warsaw, Poland (jpl@ipipan.waw.pl).

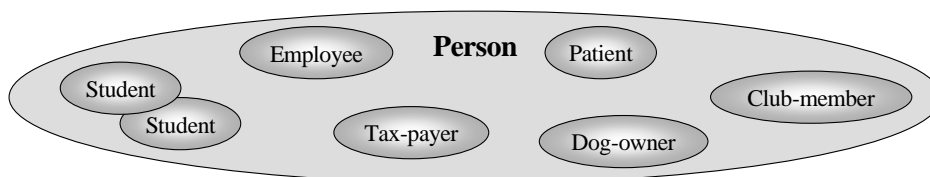


FIG. 1.1. Roles played by a person.

and independently. A role can have its own additional attributes and methods. Two roles can contain attributes and methods with the same names, and this does not lead to conflict. This is a fundamental difference in comparison to the concept of multiple inheritance. Relationships (associations) between objects can connect not only objects with objects, but also objects with roles and roles with roles. For example, a relationship *works_in* connects an *Employee* role with a *Company* object. This makes the referential semantics clean in comparison to the traditional object models. Roles can be further specialized as subroles, sub-subroles, etc.; e.g. *Club_Member* can be specialized by a role *Club_President*.

The role concept requires introducing composite objects with a special structure, semantics and generic operations. In this paper we describe the structure formally and present assumptions of a query/programming language supporting generic operations to process such structures. Our idea to deal with dynamic roles in a query language [16] is based on the stack-based approach (see e.g. [27], [28], [32]), probably the only current formalism, which can naturally adopt the concept. A version of roles [33] was implemented in the prototype system Loqis [36]. Currently we are working on a prototype of an object-oriented DBMS aiming intelligent content management for Web applications, where we intend to implement the ideas presented here.

The rest of the paper is organized as follows. Section 2 describes the real business case when the dynamic role concept was involved into a UML class diagram. Section 3 shortly presents the state-of-the-art concerning dynamic object roles. Section 4 introduces an object model with roles and discusses its differences with traditional object models introduced in programming languages and database management systems. Section 5 presents changes to popular design notations and object definition languages which are implied by roles. Section 6 presents our conclusion and future plans.

2. Dynamic Object Roles - a Business Case. In October 2001 we started a project for a Polish governmental institution dealing with regulations and investigations of the capital market. The investigations concern various forms of data mining, making summary reports and checking legality of investors' operations on the market. The general assumption of the system is that it should record all past and present

information that could be relevant for the investigations. Typical entities recognized in the business domain were as follows: person, broker, investment advisor, organizational unit, company, stock instrument, share, stock session, stock transaction, stock order, brokering house, document, legal regulation, etc. Trying to develop a class model in UML we have faced three difficulties:

1. Some objects have many specializations at the same time. For instance, some person can be at the same time an employee and a broker. This leads to multiple inheritance.
2. Some objects have many specializations of the same type. For instance, a person can be a member of many boards of directors at the same time.
3. Some objects could have specializations dependent on time. For instance, a person was a broker, but now he is a director of a company. Moreover, a person can be several times a broker, at different time and brokering houses.

Similar problems, mostly related to recording historical information, have occurred with other entities, such as institutions, companies and documents. We have concluded that for data environment dependent on historical data, the classical inheritance concept, as assumed for instance in UML, is not fully adequate. The design that we have developed is illustrated in Fig. 2.1.

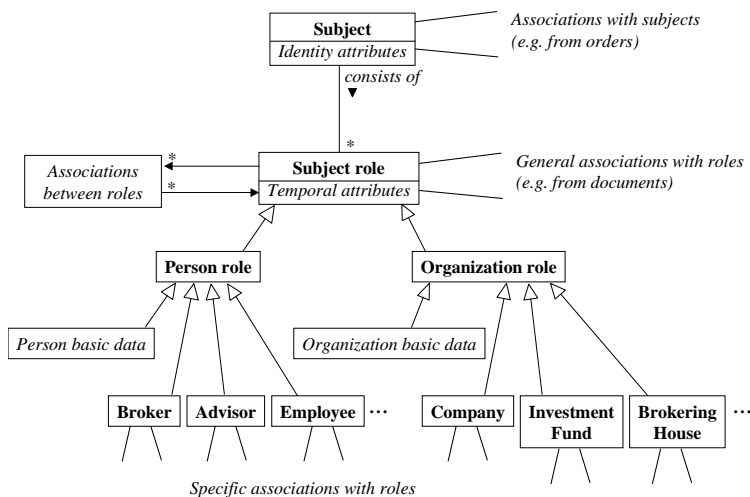


FIG. 2.1. A part of a UML diagram reflecting subject's roles.

During the design we have assumed that each subject (person or organization) itself is described by some attributes (for instance, by a number) that are unique and invariant for the subject's entire life. All other pieces of information about a subject can be changed. A change means recording a new piece of information rather than overwriting an old piece of information. To this purpose we introduce a class *Subject role*, with temporal attributes. A subject consists of many subject roles. A subject role is described by attributes such as "beginning date", "terminating date" and "is historical?". Subject roles can be associated through a special class, to record e.g. the information that some person is/was an employee of some organization. Subject roles are then specialized to *Person role* and *Organization role*. Then, the *Person role* is further specialized to *Broker*, *Advisor*, *Employee*, etc.; similarly for the *Organization role*. In this way we are able to record any current and historical information; for example, the information that somebody is currently *Broker* and

Employee, but previously he was *Advisor*, then *Broker*, then *Advisor* again, etc.

As usual, subject specializations, such as *Broker* and *Advisor*, should inherit basic attributes of a person. However, *Person basic data* and *Organization basic data* can vary during the time; for instance, a person can change the address, and a company can change the name. As a firm requirement, we must record information on all such changes. A simple solution is to consider *Person basic data* and *Organization basic data* as regular roles. If anything is changed in such a role, then the role is "frozen" (with proper temporal data written to the *Subject role*); then a copy of this role is created and modified accordingly to the new subject's state.

Note that in this case the classical inheritance is non-applicable. We cannot directly record the information that e.g. an *Employee* is a *Person*. Instead, we have implicit inheritance through temporal attributes: an *Employee* "inherits" from a *Person* if the time periods for both are overlapping. Unfortunately, this could be inconvenient in cases when temporal data are unknown or imprecise. For example, we know that Smith was a broker but we do not know when. Hence *Broker* should inherit from *Person basic data*, but we are unable to record this information (or must record it in some artificial way). Another disadvantage of the design is complexity, especially after mapping it to a relational DBMS. Very complex relational structures have implied very complex SQL queries. We have concluded that such cases are poorly treated in UML and cause difficulties during implementation. The only radical cure is to introduce dynamic object roles both on the level of UML class diagrams and on the level of data structures implemented in object-oriented or object-relational DBMS.

3. Dynamic Object Roles - State-of-the-Art. The idea of dynamically changing object roles was proposed for the network database model by Bachman and Daya [6]. During the era of the relational model and relational systems the concept was not considered in the context of databases because it did not fit well with the relational ideology. The interest to dynamic object roles has increased after computer professionals have realized the meaning of conceptual modeling in software construction. In effect, object-orientedness has been popularized in various domains of information technology. Together with object-orientedness dynamic roles are considered more frequently.

The concept often appears in papers devoted to object design, programming languages and databases, sometimes under other names, in different contexts and with various semantics. The classical object-oriented model assumes that each object is associated with its most specific class. A deviation from this rule can be treated as a certain variant of the role concept. In particular, the Iris system [11] supports many types for a single object. A similar proposal can be found in [7]. The dynamic role notion appears in papers related to modeling office information systems [26], computer aided manufacturing [41], workflow management [17], multimedia [42], semantic modeling [29], [31], [35], and object modeling [25]. These papers propose to take advantage of dynamic roles for various dynamic properties, such as object migration, schema evolution, conceptual object clustering, creation of several views for one object, and others.

In a popular approach roles are represented informally through already existing notions of the conceptual modeling by using design patterns [8], [12], [13], [19], [30]. The design pattern *decorator* [13] is considered in [12] as a good mapping of dynamic roles. The pattern allows one to insert additional functionality to a class without subclassing. In more general setting roles corresponds (to some extent) to Aspect-Oriented Programming [18], the separation of concerns principle [9] and the

Subject-Oriented Programming [15]. There are features showing that AOP and SOP have conceptual similarities with dynamic roles. Another approach introduces the role concept as an explicit notion of conceptual modeling and a database feature orthogonal to other features. Such an approach has been implemented in Aspects [29] and Fibonacci [1], [2], [4]. A feature called "subtable" has some correspondence to a dynamic role in the emerging standard SQL:1999 [5] (formerly SQL3).

In Fibonacci it is assumed that a role does not have its own behavior. The support for contextual object behavior was introduced first time in the Clovers system [37], in the proposal presented in [40], in Multiple View in the Smalltalk context [14], in the ORM model [34], and in Aspects [29]. In these approaches, however, the roles do not possess their own classes and they do not support inheritance among roles. There is also no ability to move a role from an object to another one. The proposals do not define operators supporting the programmer to switch from one object role to another one.

The proposals can be distinguished depending on their attitude to strong static typing and depending on whether introduced classes have first-class or second-class citizenship. In [31] classes are first-class citizens (so-called *prototypes*). ORM [34] is a similar proposal, but classes have the second-class citizenship. Strong static typing requires the second-class citizenship of roles' types.

Some papers (e.g. [40]) pay attention to the fact that in case of dynamic roles a unique object identifier becomes problematic. Indeed, for consistent referential semantics (e.g. for implementing relationships among object and roles) each object role should possess its own unique identifier. Hence, having the unique *identity*, an object can have many *identifiers*. This is an essential semantic novelty in comparison to the classical object model.

Multiple interfaces, which are a feature of Java and Microsoft's COM/DCOM, have some associations with dynamic object roles. The programmer is able to define interfaces in such a way that a single interface represents a single object role. However, multiple interfaces do not support all dynamic roles' features, in particular, do not deal with the inherent dynamic roles' property to be inserted to (removed from) an object at run time.

4. Object Store Model with Dynamic Roles. We assume that an object can contain many sub-objects called roles. The subobjects can be inserted and removed at run time, as in [1], [29]. Roles cannot exist without their parent objects. Deletion of an object causes deletion of all its roles. Roles can possess different types and can exist simultaneously and independently. A role can possess its own attributes and behavior (i.e. methods, rules, event processing, etc.). Identical names in two or more roles of different types do not imply any semantic dependency between corresponding properties. For example, a person can play simultaneously the role of an employee of a research institute with the attribute *Salary*, and the role of an employee of a service company with the attribute *Salary* too. These two attributes may not share any other feature, including values, types, semantics and business ontologies. A role dynamically "imports" attributes (values) and behavior from its super-roles, in particular, from its parent object.

Basic features of the store model with dynamic roles are the following:

1. An object has one main role *Person* and any number of specializing roles (for instance, *Employee* and *Student*).
2. Each role has its own name, which is used to bind the role from a program or a query. For instance, objects can be bound through name *Person*, *Employee* or

Student. Each binding returns the identifier of a proper role (or the identifiers of proper roles in case of multi-valued bindings).

3. Each role is encapsulated: its properties are not seen from other roles unless it is explicitly stated through a dynamic inheritance link. For instance, a role *Employee* imports all properties of its parent role *Person*, thus the properties of a *Person* object (e.g. Name="Brown", BirthYear=1975) are available from the *Employee* role. However, properties of a *Student* role are not directly available from the *Employee* role.

4. Each role is connected to its own class. Classes contain invariant properties of corresponding roles, in particular, names, attributes and their types and methods.

4.1. Links among objects and roles. Providing each object has a single main role, links join object roles rather than objects. For example, a link *works_in* joins a main role *Company* with a role *Employee*. A similar link *studies_at* joins a role *Student* with a main role *School*. If such a link leads to *Employee*, it indirectly leads to *Person*, because the role *Employee* imports the properties of its parent object *Person*. After accessing the object via such a link, the properties of the role *Student* remain invisible.

The ability to create links between roles is an important quality for analysis and design methodologies and notations (such as OMTUML). Links must lead to parts of objects, not to entire objects. To model this situation the methodologies recommend the notion of aggregation/composition. Such an approach implicitly assumes that e.g. an *Employee* is a part of a *Person* on the similar principle as an *Engine* is a part of a *Car*. Although the approach achieves the goal (e.g. we can connect the relationship *works_in* directly to the *Employee* sub-object of *Person*), it obviously misuses the concept of aggregation, which normally is provided for modeling the „whole-part” situations. Design patterns [8], [12], [13], [19], [30], which are proposed to deal with dynamic roles, offer a limited solution. They map the conceptual structures with roles onto structures without roles, but the resulting data structures must be processed by classical programming constructs, which usually need not (and are not) structured in a way reflecting the initial design concept. Thus this initial idea of the designer is deformed, which may result in various anomalies e.g. during software maintenance. For instance, the reverse mapping from the code into conceptual structures with roles may become problematic. In our opinion, the only radical cure for these drawbacks are dynamic roles explicitly introduced within design methodologies and within implementation environments.

4.2. Dynamic Roles vs. Classical Object-Oriented Models. Below we list several features, which make the concept of dynamic roles different in comparison to the classical object-oriented concepts.

1. **Multiple inheritance:** Because roles are encapsulated there is no name conflict even if the super classes would have different properties with the same name. There is no need for *EmployeeStudentClass*, which inherits both from *EmployeeClass* and *StudentClass*.

2. **Repeating inheritance:** An object can have two or more roles with the same name; for instance, *Brown* can be an employee in two companies, with different *Salary* and *Job*. Such a feature cannot be expressed by the traditional inheritance or multi-inheritance concepts.

3. **Multiple-aspect inheritance:** A class can be specialized according to many aspects. For example, a vehicle can be specialized according to environment (ground, water, air) and/or according to a drive (horse, motor, jet, etc.). Some

modeling tools (e.g. UML) cover this feature, but it is neglected in object-oriented programming and database models. One-aspect inheritance makes problems with conceptual modeling and usually requires multiple inheritance. Roles avoid problems with this feature.

4. **Variants** (unions): This feature, introduced e.g. in C++, CORBA and ODMG object models, leads to a lot of semantic and implementation problems. Some professionals argue that it is unnecessary, as it could be substituted by specialized classes. However, if a given class can possess many properties with variants, then modeling this situation by specialized classes leads to the combinatorial explosion of classes (e.g. for 5 properties with binary variants - 32 specialized classes). Dynamic object roles avoid this problem. Each branch of a variant can be considered a role of an object.

5. **Object migration**: Roles may appear and disappear at run time without changing identifiers of other roles. In terms of classical object models it means that an object can change its classes without changing its identity. This feature can hardly be available in classical object models, especially in models where binding objects is static.

6. **Referential consistency**: In the presented model relationships are connected to roles, not to the entire objects; thus, e.g. it is impossible to refer to *Salary* and *Job* of *Smith* when one navigates to its object from the object *School*. In classical object-oriented models this consistency is enforced by strong typing, but is problematic if the typing is weak.

7. **Overriding**: Properties of a super-role can be overridden by properties of a sub-role. The ability of overriding is extended in comparison to the classical object models: not only methods but also attributes (with values) can be overridden.

8. **Binding**: An object can be bound by the name of any of its roles, but the binding returns the identifier of a role rather than the identifier of the object. By definition, the binding is dynamic, because in a general case during compilation it is impossible to decide that a particular object has a role with a given name.

9. **Typing**: A role must be associated with a name, because this is the only feature allowing the programmer to distinguish a role from another one. Hence, the role name is a property of its type (unlike classical programming languages, where a type usually does not determine the name of a corresponding object/variable). Because an object is seen through the names of its roles, it has as many types as it has different names for roles.

10. **Subtyping**: It can be defined as usual; for instance, the *Employee* type is defined with the use of the *Person* type. However, there is no sense to introduce the *StudentEmployee* type. Due to encapsulated roles, properties of a *Student* object and properties of an *Employee* object are not mixed up within a single structure.

11. **Substitutability**: Since names of roles are determined within types, it makes little sense to say, e.g. that the *Employee* type can be used in all places, where the *Person* type can be used. Thus the substitutability principle must be at least reformulated.

12. **Temporal properties**: As shown in Section 2, dynamic object roles are enormously useful for temporal databases, as roles can represent any past facts concerning objects, e.g. the employment history through many *Employee* roles within one *Person* object. Without roles, historical objects present a hard design problem, especially if one wants to avoid redundancy, preserve reuse of unchanged properties through standard inheritance, and avoid changing objects' identifiers.

13. **Dynamic inheritance:** The *EmployeeClass* do not inherits statically from the *PersonClass*. Instead, an *Employee* role inherits dynamically all the properties of its *Person* super-role, thus indirectly inherits properties of the *PersonClass*.

14. **Aspects of objects and heterogeneous collections.** A big problem with classical database object models, for instance ODMG [23], is that an object belongs to at most one collection. This is contradictory with both multiple inheritance and substitutability. For instance, we can include a *StudentEmployee* object into the extent *Students*, but we cannot include it at the same time into the extent *Employees* (and vice versa). This violates substitutability and leads to inconsistent processing. Dynamic roles have a natural ability to model heterogeneous collections: an object is automatically included into as many collections as the types of roles it contains.

15. **Aspect-Oriented Programming.** AOP [18] makes it possible to encapsulate cross-cutting concerns within separate modules, for example, such concerns as: history of changes, security and privacy rules, visualization, synchronization, etc. As follows from the previous feature, dynamic object roles have conceptual similarities with AOP or can be considered as a technical facility supporting AOP.

16. **Metadata support.** Metadata are a particular case of crosscutting concerns. Meta-information, as assumed e.g. in Dublin Core [10] or W3C RDF [39] (such as authorship, validity, legal status, ownership, coding, etc.) can be implemented as dynamic roles of information objects.

As follows from the above, dynamic object roles have the potential to create new powerful qualities, which are difficult or impossible to achieve in the classical object model.

4.3. Dynamic Roles - a Formal Model of an Object Store. For simplicity and for making the semantics clean we assume object relativism (i.e. each property of an object is an object too) and consider all properties of the store (including classes) first-class citizens. The store models a program/database *state* thus does not involve types, which we consider a checking utility rather than a "materialized" property of the state.

Formally, an object is a triple $\langle i, n, v \rangle$, where i is a unique internal object identifier, n is an external object name, and v is an object value. The value can be atomic (e.g. "Doe"), can be a reference to another object, or can be a set of objects. Classes and methods are objects too. A store model with explicitly defined roles is defined as a 6-tuple $\langle O, C, R, CC, OC, OO \rangle$, where:

1. O is a collection of (nested) objects,
2. C is a collection of classes,
3. R is a collection of root identifiers (of objects being entries of the store),
4. CC is a binary relation determining inheritance relationship,
5. OC is a binary relation determining membership of objects within classes.
6. OO is a binary relation determining inheritance relationship between roles.

We distinguish *objects* and *roles*. Objects may consist of many roles, and a role belongs to a single object. An object has exactly one *main role*. The name of this main role should reflect semantics of the entire object. Deleting this role implies deleting the entire object. Any role can *inherit dynamically* from another role within the same object; inheritance among roles in different objects is forbidden. The inheritance is based on the same rule as inheritance of class properties. A role, which inherits, is called a *sub-role*; an inherited role is called a *super-role*. The relation OO defines two functional aspects. On the one hand, the relation determines which roles are inherited by other roles. On the other hand, the relation OO fixes the semantics of manipulating

objects with roles. In particular, copying an object implies "isomorphic" copying of all of its roles, and deleting an object implies deleting all of its roles. Deleting a role implies recursive deleting all of its sub-roles. The relation *OO* is a pure hierarchy.

```

O – Objects (roles):
< i1, Person , { < i2, Name, "Doe" >, < i3, BirthYear, 1948 > } >
< i4, Person , { < i5, Name, "Brown" >, < i6, BirthYear, 1975 > } >
< i7, Person , { < i8, Name, "Smith" >, < i9, BirthYear, 1951 > } >
< i13, Employee , { < i14, Salary, 2500 >, < i15, works_in, i127 > } >
< i16, Employee , { < i17, Salary, 1500 >, < i18, works_in, i128 > } >
< i19, Student , { < i20, StudentNo, 223344 >, < i21, Faculty, "Physics" > } >
.....
C - Classes:
< i40, PersonClass , { < i41, Age, (...code of the method Age...) >,
...other properties of the class PersonClass... } >
< i50, EmployeeClass , { < i51, ChangeSalary, (...code of the method ChangeSalary...) >,
< i52, NetSalary, (...code of the method NetSalary...) >,
...other properties of the class EmployeeClass... } >
< i60, StudentClass , { < i61, AvgScore, (...code of the method AvgScore...) >,
...other properties of the class StudentClass ... } >
.....
R – Root identifiers:
i1, i4, i7, i13, i16, i19, ...
CC - Inheritance relationships between classes:
Empty.
OC - Membership of roles in classes:
< i1, i40 >, < i4, i40 >, < i7, i40 >, < i13, i50 >, < i16, i50 >, < i19, i60 >, ...
OO – Inheritance between roles:
< i13, i4 >, < i16, i7 >, < i19, i7 >, ...
    
```

FIG. 4.1. An example state of the object store in the object model with roles.

In Fig. 4.1 we present an example of the object store with roles and in Fig. 4.2 we present the same object store graphically. The component *CC* - the relation determining inheritance between classes - is empty in this case. Each role directly inherits from its class. A role inherits the properties of its super-roles, hence indirectly inherits the properties of the classes that these super-roles belong to.

The model does not imply problems with multiple inheritance, because no name conflict is possible. The model also clearly shows the reason for multiple inheritance anomalies in classical object models: they are caused by the fact that properties of different classes (perhaps incompatible) are not encapsulated, but mixed up in a single environment.

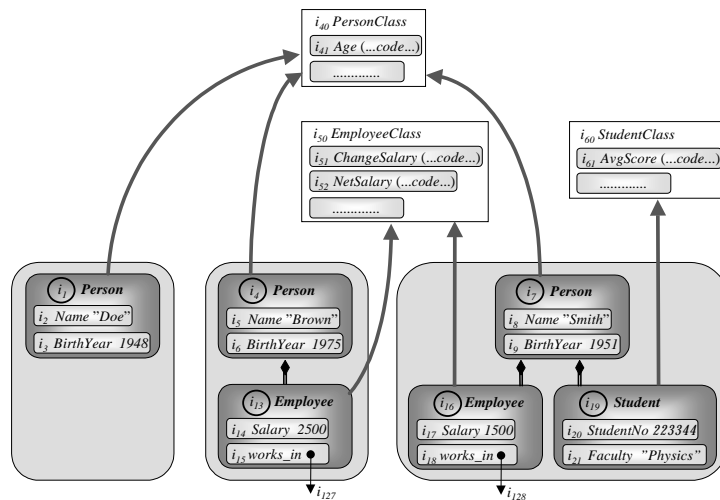


FIG. 4.2. Graphical representation of the object store from Fig. 3.

Note that the identifier of each role belongs to root identifiers R , which present starting points for binding objects. Hence name *Person* is bound to i_1, i_4, i_7 . The binding concerns exactly the *Person* roles; other roles are invisible. Similarly, name *Employee* is bound to i_{13} and i_{16} , but after this binding the corresponding roles *Person* (i_4 for i_{13} and i_7 for i_{16}) become visible, according to the *OO* relation. Thus, for instance, *Employee.Name* and *Employee.Age* are correct expressions. Similarly when name *Student* is bound.

The model is consistent concerning references. For example, when name *Person* is bound then references *works_in* are unavailable, because the roles *Employee* are invisible. This property holds independently on whether the system is strongly typed or untyped.

5. Specification of Dynamic Roles in Database Schemata. We adopt the syntax of ODMG ODL (based on IDL CORBA [24]), however, we will be more precise concerning defined concepts and semantics.

In our view *classes* are implementation units storing invariant properties of their members (objects). The invariant properties are usually reduced to names and types of member's attributes, and methods, which can be executed on the member. Other kinds of invariant properties include: a name assigned to a class member, specification of events/exceptions, implementation of reactions to events/exceptions, implementation of integrity constraints concerning the member, specification of exported properties of objects (public properties), specification of imported properties (active and passive side effects), etc. *Interfaces* allow the programmer to treat objects as black boxes, thus should bear all the information, which is necessary to deal with objects. A *type* is the main component of an interface presenting constraints on the object's structure and constraints on the contexts in which objects can be used. Multiple interfaces to a single object are possible (as in Java and COM/DCOM). A role of an object is presented through its interface. Each interface represents an object's role; there are no other interfaces.

5.1. A Sample Construction of an Object Schema with Dynamic Roles.

We will follow the top-down approach of the object-orientedness in which classes are reusable units, which can be further specialized by subclasses. In terms of dynamic roles, a class after development can be closed for modifications, but it can be further extended by new, ad hoc defined dynamic roles. Role classes correspond to subclasses of a given class.

A class is an implementation unit registered in the object store by a system or by a database administrator. Each class has a unique name (*AbstractPersonClass*, *PersonClass*, *StudentClass*, etc.). The registration of a class means the following actions:

1. The full code of a class is introduced as a single object to the object store. Methods and other properties of the class are introduced as subobjects of this object;
2. Meta-information concerning the class (class name, class location, ownership, comments, etc.) is introduced to the corresponding structure of the catalog;
3. The *entire interface* to a class is introduced to the catalog. The interface specifies all public properties of the class and is used to build specialized interfaces to class properties. A class interface can be introduced manually by a person registering the class, or automatically, by a special registration utility. It can use special keywords and language constructs in the class code, such as keyword "public", signatures of methods, etc.

Interfaces are defined and identified independently of classes, but they refer to the names of registered classes (more precisely, to the names of their entire interfaces). The approach makes it possible to define several interfaces to a single class.

Any properties relevant to collections of objects being members of a class (class methods and class attributes in the C++ terminology) have to be stored in a separate class (so called a *power-set-of* class), whose members are collections of objects (e.g. *PersonCollectionClass*).

Classes are specified by interfaces, which (as in IDL CORBA and ODL) present specification of public class properties. Interfaces contain all typing information (types of attributes and signatures of methods), but may also contain information irrelevant to types (e.g security rules). Interfaces are the subject of inheritance relationships, which can be one of two kinds. "Static" inheritance concerns properties of a class. "Dynamic" inheritance concerns properties of a super-role (in Fig. 5.1 - a double-line with a black diamond end). Usually the inheritance among interfaces corresponds to the inheritance among corresponding classes. However, this is not mandatory and may depend on implementation of classes and the conceptual view on corresponding interfaces.

The relationship between *PersonCollectionInterface* and *PersonInterface* (shown in Fig. 5.1 as a dashed double-line arrow) determines the dependency between a power-set-of class and a class of its elements. Power-set-of classes do not constitute a special kind of classes - they follow ordinary rules but their members are collections of objects.

All the lines, arrows and boxes in Fig. 5.1 are visual comments - the same information is explicitly present inside interfaces. The syntax presented in Fig. 5.1 is an ad hoc extension of the ODL syntax. The differences to the ODL semantics are the following:

1. Each interface is named and is associated with a class name. These names do not determine the names of the corresponding objects. Some interfaces (e.g. *PersonInterface*, *EmployeeInterface*) define names of corresponding objects. Interfaces need not to determine such a name (e.g. *PersonCollectionClass*).

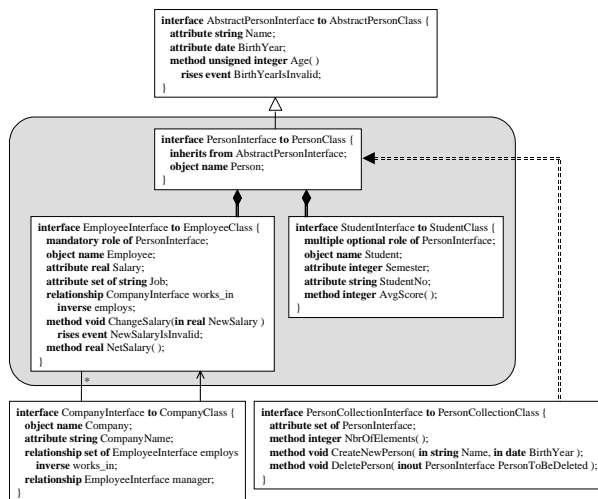


FIG. 5.1. Interfaces to database objects with roles.

2. An interface to a role can be associated with a super-role interface. A role can

be mandatory optional, and/or multiple. A mandatory role means that its super-role must possess it. Usually roles are optional. A multiple role may have many instances within an object.

3. As in ODL (and unlike UML and the CORBA Relationships Service), relationships are binary and have no attributes. In our opinion, more powerful relationships lead to clumsy programming options. Relationships can be bidirectional (with the **inverse** option, as for *works_in/employs*) or directed (as for *manager*). Independently of the **inverse** option, the system is responsible for keeping referential integrity of relationships thus no dangling references can appear. Unlike ODL, defining attributes with values being references to objects/roles is not allowed - such situations must be determined as directed relationships.

4. The interface *PersonCollectionInterface* has an attribute being a set of objects *Person*. The name for this attribute is undefined here, because the definition is present in the *PersonInterface*. An interface like the *PersonCollectionInterface* can define more such attributes, what makes it possible to define heterogeneous collections. Such a feature is available in the ODMG standard; however, the semantics is not consistent [3].

5.2. Declarations of Data Structures. The schema presented in Fig. 5.1 maps interdependencies among interfaces, but it does not determine which objects are currently stored in the database. Defining stored objects is the main function of a database schema, as this information is necessary for the application programmer. In ODMG it is implicitly assumed that some interfaces determine stored objects. Besides the user can use an explicit definition of a stored data structure (*extent*) associated with an ODMG class. The CORBA standard has no clauses determining stored data structures, but it is implicitly assumed that each IDL interface can be used to access CORBA objects, which are created in a way dependent on particular object implementation. An interface definition is coupled with the definition of stored data structures in relational systems. This is materialized in the SQL "create table" statement.

Lack of separation interfaces/types from declarations/creations of data structures makes problems with the following cases:

1. One would like to define an abstract interface (to inherit from it), which has no direct member objects;
2. One would like to define an interface to an attribute, to a sub-attribute, etc. i.e. to an object which does not belong to the "root" objects;
3. One would like to define an interface to a single object (which does not participate in a collection);
4. One would like to define an interface to a module (an entity encapsulating classes, interfaces, types, etc.) and - independently - interfaces to objects stored within the module.

The emerging SQL 1999 standard abandons such a solution and assumes that a table type (i.e. an interface, in another terminology) can be defined independently from declarations/creations of stored tables. This is a comeback to the tradition of programming languages. Such an approach supports *reuse* and *encapsulation*: a single definition (perhaps complex, opaque and closed) can be reused to create many data structures. Thus, declarations of interfaces, as shown in Fig. 5.1, are not declarations of stored data structures. They present another information to be stored in the database catalog. In Fig. 5.2 we have declared:

1. A single object named *President* (accessed by the *AbstractPersonInterface* -

no roles).

2. Two objects named *YoungPersons* and *OldPersons* (being collections of objects *Person* accessed by the *PersonInterface*; objects *Person* can be specialized by roles).

3. A set of objects *Company*, named *Companies*, without a corresponding power-set-of class thus without an interface to the collection. Thus it is implicitly assumed that such a collection is accessible through a *generic interface*, which need not be defined explicitly by the designer or programmer. In our approach any declared structure can be served by a query language, which presents such a generic interface.

The presented idea concerns the conceptual view on objects, roles, their classes and types.

```

AbstractPersonInterface President;
PersonCollectionInterface YoungPersons;
PersonCollectionInterface OldPersons;
set of CompanyInterface Companies;

```

FIG. 5.2. *Declarations of data structures stored in the database.*

6. Summary and Conclusion. We have presented an object model with dynamic roles, which we consider an alternative to the classical database object models, such as the ODMG object model. Dynamic roles can support conceptual models of many applications and, in comparison to classical object models, do not lead to anomalies and limitations of multiple, repeating or multi-aspect inheritance. We have shown that the classical concepts of the object-orientedness, such as object identity, polymorphism and overriding can be smoothly incorporated into the model. An advantage of the model with dynamic roles is conceptual clarity concerning the level of object store and the mechanisms of data naming, scope control and binding names. The model leads to some new concepts for a schema definition language and metadata management; this issue requires further research.

The model with dynamic roles and with a corresponding query/programming language is currently being implemented as a repository for intelligent content management systems for distributed Web applications, in particular, as an enhanced XML repository.

REFERENCES

- [1] A. ALBANO, R. BERGAMINI, G. GHELLI, AND R. ORSINI, *An Object Data Model with Roles*, Proc. VLDB Conf. (1999), pp. 39–51.

- [2] A. ALBANO, G. GHELLI, AND R. ORSINI, *Fibonacci: A Programming Language for Object Databases*, VLDB Journal, 4(3) (1995), pp. 403–444.
- [3] S. ALAGIC, *The ODMG Object Model: Does it Make Sense?*, Proc. OOPSLA Conf. (1997), pp. 253–270.
- [4] A. ALBANO, G. ANTOGNONI, AND G. GHELLI, *View Operations on Objects with Roles for a Statically Typed Database Language*, TKDE, 12(4) (2000), pp. 548–567.
- [5] J. MELTON (ED.), *Database Language SQL*, American National Standards Institute (ANSI) Database Committee (X3H2), September 1999.
- [6] C. BACHMAN AND M. DAYA, *The Role Concept in Data Models*, Proc. VLDB Conf. (1977), pp. 464–476.
- [7] E. BERTINO AND G. GUERRINI, *Objects with Multiple Most Specific Classes*, Proc. ECOOP Conf. (1995), Springer LNCS 952, pp. 102–126.
- [8] D. BÄUMER, D. RIEHLE, W. SIBERSKI, AND M. WULF, *Role Object*, in Pattern Languages of Program Design 4, N. Harrison, B. Foote, and H. Rohnert, eds., Addison-Wesley, 2000, pp. 15–32.
- [9] E. W. DIJKSTRA, *A Discipline of Programming*, Englewood Cliffs, NJ. Prentice Hall, 1976.
- [10] *Dublin Core Metadata Initiative*, <http://dublincore.org>.
- [11] D. FISHMAN ET AL, *IRIS: An Object-Oriented Database Management System*, ACM Transactions on Office Information Systems, 5(1) (1987), pp. 48–69.
- [12] M. FOWLER, *Dealing with Roles*, (1997), <http://www.martinfowler.com/apsupp/roles.pdf>.
- [13] E. GAMMA, R. HELM, R. JOHNSON, AND J. VLISSIDES, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- [14] G. GOTTLÖB, M. SCHREPL, AND B. ROCK, *Extending Object-Oriented Systems With Roles*, ACM Transactions on Information Systems (1996), pp. 268–296.
- [15] W. HARRISON AND A. OSSHER, *Subject-Oriented Programming (A Critique of Pure Objects)*, Proc. ECOOP Conf. (1993), ACM SIGPLAN Notices 28(10), pp. 411–428.
- [16] A. JODŁOWSKI, P. HABELA, J. PŁODZIEN, AND K. SUBIETA, *Objects and Roles in the Stack-Based Approach*, Proc. DEXA Conf. (2002), to appear.
- [17] G. KAPPEL, S. RAUSCH-SCHOTT, AND W. RETSCHITZEGGER, *A Framework for Workflow Management Systems Based on Objects, Rules and Roles*, ACM Computing Surveys 32 (2000), p. 27.
- [18] G. KICZALES, J. LAMPING, A. MENDHEKAR, C. MAEDA, C. LOPES, J. LOINGTIER, AND J. IRWIN, *Aspect-Oriented Programming*, Proc. ECOOP Conf. (1997), Springer LNCS 1241, pp. 22–242.
- [19] B. B. KRISTENSEN AND K. ØSTERBYE, *Roles: Conceptual Abstraction Theory and Practical Language Issues*, Theory and Practice of Object Systems, 2(3) (1996), pp. 143–160.
- [20] B. B. KRISTENSEN, *Object-Oriented Modeling with Roles*, Proc. OOIS Conf. (1995), pp. 57–71.
- [21] Q. LI AND F. H. LOCHOVSKY, *ADOME: An Advanced Object Modeling Environment*, IEEE Transactions on Knowledge and Data Engineering, 10(2) (1998), pp. 255–276.
- [22] Q. LI AND R. K. WONG, *Multifaceted Object Modeling with Roles: A Comprehensive Approach*, Information Sciences 117(3-4) (1999), pp. 243–266.
- [23] R. G. G. CATTEL AND D. K. BARRY (EDS.), *Object Data Management Group: The Object Database Standard ODMG, Release 3.0*, Morgan Kaufmann, 2000.
- [24] *Object Management Group: OMG CORBA/IIOP Specifications*, 2002, <http://www.omg.org/technology/documents/corba.spec.catalog.htm>.
- [25] M. PAPAIOGLOU, *Roles: A Methodology for Representing Multifaced Objects*, Proc. DEXA Conf. (1991), pp. 7–12.
- [26] B. PERNICI, *Objects with Roles*, Proc. IEEE/ACM Conf. on Office Information Systems (1990).
- [27] J. PŁODZIEN AND A. KRAKEN, *Object Query Optimization through Detecting Independent Subqueries*, Information Systems 25(8) (2000), pp. 467–490.
- [28] J. PŁODZIEN, *Optimization Methods in Object Query Languages*, Ph.D. thesis, Institute of Computer Science, Polish Academy of Sciences, Warsaw, Poland, 2000.
- [29] J. RICHARDSON AND P. SCHWARZ, *Aspects: Extending Objects to Support Multiple, Independent Roles*, Proc. SIGMOD Conf. (1991), pp. 298–307.
- [30] D. RIEHLE AND T. GROSS, *Role Model Based Framework Design and Integration*, Proc. OOPSLA Conf. (1998), ACM Press, pp. 117–133.
- [31] E. SCIORE, *Object Specialization*, ACM Transactions on Information Systems 7(1) (1989), pp. 103–122.
- [32] K. SUBIETA, Y. KAMBAYASHI, AND J. LESZCZYŁOWSKI, *Procedures in Object-Oriented Query Languages*, Proc. VLDB Conf. (1995) pp. 182–193.
- [33] K. SUBIETA, F. MATTHES, J. W. SCHMIDT, A. RUDLOFF, AND I. WETZEL, *Viewers: A Data-World Analogue of Procedure Calls*, Proc. VLDB Conf. (1993), pp. 269–277.

- [34] M. SCHREFL AND E. NEUHOLD, *Object Class Definition by Generalization Using Upward Inheritance*, Proc. ICDE Conf. (1988), pp. 4–13.
- [35] L. A. STEIN, *Delegation is Inheritance*, Proc. OOPSLA Conf. (1987), pp. 138–146.
- [36] K. SUBIETA, *LOQIS: The Object-Oriented Database Programming System*, Proc. Intl. East/West Database Workshop (1990), Springer LNCS 504, pp. 403–421.
- [37] L. STEIN AND S. ZDONIK, *Clovers: The Dynamic Behavior of Type and Instances*, Technical Report CS-89-42, Brown University, 1989.
- [38] *Object Management Group: Unified Modeling Language (UML) Specification. Version 1.4*, 2001, <http://www.omg.org/>.
- [39] *Resource Description Framework (RDF)*, WWW Consortium (W3C), 2002, <http://www.w3.org/RDF/>.
- [40] R. WIERINGA AND W. D. JONGE, *The Identification of Objects and Roles - Object Identifiers Revisited*, Technical Report IR-267, Faculty of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, 1991.
- [41] R. K. WONG AND Q. LI, *Manufacturing Systems Modeling with Roles. A Comprehensive Approach*, Proc. IFIP WG2.6 Sixth Working Conference on Database Semantics (DS-6) (1995), pp. 461–478.
- [42] R. K. WONG, *Heterogeneous and Multifaceted Multimedia Objects in DOOR/MM: A Role-Based Approach with Views*, Journal of Parallel and Distributed Computing 56(3) (1999), pp. 251–271.