Category Classes: Flexible Classification and Evolution in Object-Oriented Databases

Erik Odberg*
Department of Computer Science
Norwegian Institute of Technology

Abstract

Object-oriented databases (OODBs) are believed to more naturally reflect the behavior and organization of real world objects. However, OODBs are mostly concerned about only the *static* aspects of object modeling. While real world objects typically may be *multi-perspectived* and *evolve* over time by changing classification and behavior, contemporary OODB models typically regard objects as instances of classes in such a way that classification (and thus behavior) is fixed at the time of creation.

This paper introduces the notion of an object *role* to denote a particular *perspective* of an object, corresponding to a class for which it is an instance. Roles may be dynamically added and removed from objects to reflect the way real world objects classify and evolve over time, and simultaneously change behavior. A *category class* is a special class which is associated with a *predicate*, and in this way describe *constraints* on how objects may evolve, as well as how objects may *automatically* gain and discard roles based on various criteria.

1 Introduction and Motivation

Object-oriented data models (and consequently object-oriented databases (OODBs)) are believed to better and more naturally reflecting the behavior and organization of real world phenomena, incorporating more of the real world semantics. *Objects* separate externally visible behavior from internal representation and implementation. *Classes* abstract over commonalities between objects, defining *classifications* of objects. Class *hierarchies* impose an organization of classes describing a *conceptual specialization* (i.e. that instances of one class may also be regarded as instances of more general *superclasses*) and *inheritance* of properties (i.e. that instances of one class will also contain properties as defined for superclasses)[1].

However, most object models are only concerned about the *static* aspects of modeling, and not how objects may *evolve* over time. Objects are *created* as an instance of one class, which also serves to *classify* the object according to the predefined taxonomy, and to define its properties. Once created, objects *cannot* change classification or property possession. Naturally, this does not reflect very well the way the "real world" behaves: Real-world objects ("*phenomena*") may typically have *different* appearance (exhibited behavior) in different contexts of *expectation*, and may be perceived to *classify* differently (dependent on expectation). Phenomena are often highly *dynamic*, and *evolve* over time by *changing* classification and appearance.

The remaining part of this paper describes an *extension* to the traditional notion of object-orientation which naturally reflects the behavior of the "real world". Section 2 introduces the notion of an object *role* as a particular perspective of an object, and discusses how roles may be added to and removed from objects to allow objects to evolve. Section 3 defines a *category class* as a specialization of the

*Detailed address: Department of Computer Science, Norwegian Institute of Technology (NTH), N-7034 Trondheim-NTH, Norway. Phone: +47 73 594484. Fax: +47 73 594466. Email: eriko@idt.unit.no

[1]The *conceptual organization* and *property reuse* dimensions of subclassing reflect the *Scandinavian* and *American* school of OO thinking, respectively.

class construct which may define *restrictions* on or *enable* certain evolutions of objects. Section 4 compares the approach with related work, while Section 5 concludes the paper and presents directions for further work.

2 Objects with Roles

The fundamental object model is "traditional" in that it adopts C++ as the primary source of influence. *Objects* are created as instances of *classes*, which have an associated collection of *properties*. Properties may be *attributes* or *methods*, defined in a way which is similar to C++, and have a *visibility* which is either *public* (accessible by external clients of the object) or *private* (only accessible to method implementations for the class). The visibility is specified by keywords public and private, with public being default (in contrast to C++). A notion of *explicit relationships* is supported to provide symmetric associations (with cardinalities) between two classes, and with special functionality for navigating over these. A class may have one or more *superclasses* (defining the class as a *subclass* of these). The subclass *inherits* the properties of the superclasses (retaining the visibility mode[2]), and may define additional properties or *redefine* inherited ones. The subclass also defines a *specialization* relationship with its superclasses, and so that instances of one class are also instances of all superclasses (transitively). Note that a class Class is the implicit superclass of all classes for which no superclass is defined; in this way a schema will consist of *one* connected class hierarchy.

2.1 Object manipulation

Persistent objects are manipulated (created, modified, deleted) through a corresponding C++ *placeholder* object. Placeholder objects are instances of C++ *persistent classes*, which are generated from the schema specification. Each persistent C++ class directly corresponds to a schema class, having the same name and properties, and providing additional functionality for persistent manipulation. Placeholders are acquired from the database by *navigation* over database relationships, by *iteration* over a set of persistent objects (possibly acquired through an *associative query*), or as a result from a *method invocation*.

Objects created as instances of persistent classes will *not* automatically persist in the database. This must be explicitly requested by invocation of a function *MakePersist*, associated with each persistent class. In this way instances of persistent classes may be *both* persistent and non-persistent, but may smoothly interact within the same application process. Object modifications are also "committed" persistently by invocation of MakePersist.

2.2 Object roles

Each class an object may be regarded an instance of will be denoted a role of this object. Each object role reflects a particular *perspective* of the object, a context of behavior which may be referenced by clients. More important, roles (i.e. class memberships) reflect characteristics of objects which may be independently gained and lost: Objects may *evolve* over time by *changing* role possession (i.e. the classification and associated properties) dynamically. Some roles may reflect *specializations* of other roles (corresponding to superclasses), however an important characteristic of the model is the ability for objects to possess roles corresponding to *sibling* classes. Between two sibling classes there is neither a subclassing relationship, nor is there a common subclass of these classes. We say that objects may contain multiple *most-specific* roles, or that objects are *multi-perspectived*. This is in contrast with traditional object-oriented models where each object is an instance of *one* class

[2]This is the same as *public* inheritance in C++.

which is decided upon creation, and all "roles" correspond to the creation class and its superclasses. There is no ability to have object evolve or assume additional most-specific roles.

The *creation* of an object will allocate a collection of roles (for all classes the object is an instance), and simultaneously assign a unique object identifier (OID) to the new object. New roles may be added, according to the class hierarchy, and in this way "*extending*" the object downwards. *No* new OID is assigned upon role addition, it is still the *same* object. Any role(s) may also be removed from the object (without deleting the object or affecting the OID), and which will also have the impact that all roles corresponding to *subclasses* of the denoted role class are removed as well. The strong notion of identity means that no object set may contain multiple occurrences of the same object, even if different roles are regarded. Furthermore, *two* operators for checking the identity of object references are provided: Two placeholders are *object identical* if they correspond to the same (persistent) object, while placeholders are *object-role identical* if they correspond to the same object *and* role.

Figure 1 illustrates an example class hierarchy.

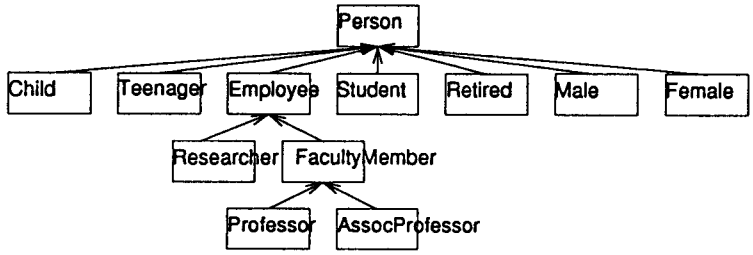

Figure 1: Class hierarchy

A *person*'s "life" is described by the way roles are gained and lost, possibly possessing many different (most-specific) roles simultaneously. Some roles may be automatically gained and lost (for instance Child and Teenager due to age), while some must be explicitly added (as for instance Student and Employee). Some roles are inherently incompatible, and may not be possessed at the same time (for instance Female and Male, or Retired and Employee). Other roles may be possessed simultaneously, as for instance a female teaching assistant which is an instance of Female, Employee and Student, and behaves in different ways in these three roles.

2.3 Object role manipulation

The introduction of the role notion have implications to the functionality provided for object manipulation.

2.3.1 Adding roles

New roles are added to persistent objects through the add operator, which is applicable to all placeholders:

```
Person *pers = ...;
Student *stud = pers add Student;
```

A role is always added as a *specialization* of some existing role, however may perfectly well be added as another "*branch*" of the object, and so that the object does not have to be an instance of a *single* most-specific class. For instance, the following example illustrates a *person* which is simultaneously also a *student* and an *employee*:

```
Employee *emp = stud add Employee;
```

It must be noted that there is *no* constraint on what is the class of the object which is the *basis* for the role addition: In principle *any* role may be added to any object, however Section 3 will show how restrictions may be imposed.

The add operator returns a pointer to a C++ placeholder object as an instance of the class corresponding to the added role. This means that the placeholders referenced by pers, stud and emp, which correspond to the *same* persistent object, need *not* be the same C++ objects[3]. Rather, they reference different *perspectives* of the same (persistent) object. Application programs must ensure, in order to avoid mutation conflicts, that objects are mutated through only *one* perspective (placeholder) at the time. That is, applications should preferably *synchronize* with the database (by invoking MakePersist) before the new role is added. The DBMS has no other means to ensure that multiple placeholders within the same process are maintained consistently wrt. mutations. Note that a role addition is not reflected persistently unless MakePersist is explicitly invoked.

If a role is added to an object which already *does* possess this role, the object is not affected by the addition. This means that an object may have only one role corresponding to each class. If a role is possessed by a persistent object, but not reflected by the actual placeholder, a *new* placeholder must be returned. This is because the result of the add operation is expected to be an instance of the added class. Consequently, for the following operation no new role is added, but another placeholder (which may be the same as for stud) is returned:

```
Student *stud2 = emp add Student;
```

It must be noted that some other approaches, notably Aspects [RS91] and Clovers [SZ89], allows for instantiating multiple roles of the *same* class. It is our position that this is *unnatural* modeling, and better reflected by the use of *relationships* between objects. Furthermore, there is an important problem wrt. *identification* of roles which reflect the same class.

A notion of a *role constructor* is provided for the initialization of new roles. Role constructors operate as ordinary constructors, but initialize *only* attributes directly associated (i.e. non-inherited) with the class of the role to be added. Zero or more role constructors may be associated with each class. If there is *no* visible role constructor defined for a particular class, this means that the class may *not* be added as a role to any existing object[4]:

```
enum {male, female} Sex;

class Person{
  char *name;
  enum Sex sex;
  int age;
  Person(char*, enum Sex, int);};

class Employee : Person {
  char *company;
  Employee(char*, char*, enum Sex, int);     // Ordinary constructor
  $Employee(char*);}                         // Role constructor
```

Similar to for object creation, role addition will thus take arguments as indicated by the role constructor:

```
Person *pers = new person(''Donald Duck'', male, 60);
Employee *emp = pers add employee (''Walt Disney Corp.'');
```

[3] In fact, in this case they *cannot* be, as it is not possible to have one C++ object be an instance of both Student and Employee, as there is no common subclass for these classes.

[4] As for ordinary C++ constructors, if *no* role constructor is explicitly specified, a non-argument role constructor will by default be available.

A role is always added as a *specialization* of some existing role, however may perfectly well be added as another "*branch*" of the object, and so that the object does not have to be an instance of a *single* most-specific class. For instance, the following example illustrates a *person* which is simultaneously also a *student* and an *employee*:

```
Employee *emp = stud add Employee;
```

It must be noted that there is *no* constraint on what is the class of the object which is the *basis* for the role addition: In principle *any* role may be added to any object, however Section 3 will show how restrictions may be imposed.

The add operator returns a pointer to a C++ placeholder object as an instance of the class corresponding to the added role. This means that the placeholders referenced by pers, stud and emp, which correspond to the *same* persistent object, need *not* be the same C++ objects[3]. Rather, they reference different *perspectives* of the same (persistent) object. Application programs must ensure, in order to avoid mutation conflicts, that objects are mutated through only *one* perspective (placeholder) at the time. That is, applications should preferably *synchronize* with the database (by invoking MakePersist) before the new role is added. The DBMS has no other means to ensure that multiple placeholders within the same process are maintained consistently wrt. mutations. Note that a role addition is not reflected persistently unless MakePersist is explicitly invoked.

If a role is added to an object which already *does* possess this role, the object is not affected by the addition. This means that an object may have only one role corresponding to each class. If a role is possessed by a persistent object, but not reflected by the actual placeholder, a *new* placeholder must be returned. This is because the result of the add operation is expected to be an instance of the added class. Consequently, for the following operation no new role is added, but another placeholder (which may be the same as for stud) is returned:

```
Student *stud2 = emp add Student;
```

It must be noted that some other approaches, notably Aspects [RS91] and Clovers [SZ89], allows for instantiating multiple roles of the *same* class. It is our position that this is *unnatural* modeling, and better reflected by the use of *relationships* between objects. Furthermore, there is an important problem wrt. *identification* of roles which reflect the same class.

A notion of a *role constructor* is provided for the initialization of new roles. Role constructors operate as ordinary constructors, but initialize *only* attributes directly associated (i.e. non-inherited) with the class of the role to be added. Zero or more role constructors may be associated with each class. If there is *no* visible role constructor defined for a particular class, this means that the class may *not* be added as a role to any existing object[4]:

```
enum (male, female) Sex;

class Person{
  char *name;
  enum Sex sex;
  int age;
  Person(char*, enum Sex, int);};

class Employee : Person {
  char *company;
  Employee(char*, char*, enum Sex, int);     // Ordinary constructor
  $Employee(char*);}                         // Role constructor
```

Similar to for object creation, role addition will thus take arguments as indicated by the role constructor:

```
Person *pers = new person(''Donald Duck'', male, 60);
Employee *emp = pers add employee (''Walt Disney Corp.'');
```

[3] In fact, in this case they *cannot* be, as it is not possible to have one C++ object be an instance of both Student and Employee, as there is no common subclass for these classes.

[4] As for ordinary C++ constructors, if *no* role constructor is explicitly specified, a non-argument role constructor will by default be available.

2.3.2 Removing roles

Roles may also be *removed* from objects, using the rem operator:

```
Employee *emp = ...;
Person *pers = emp rem Employee;
```

The rem operator removes the role corresponding to the indicated class from the object (provided it is contained), as well as all contained roles corresponding to subclasses. A pointer to the same object as an instance of the immediate *superclass* of the removed class is returned[5]. In general, this means that *another* placeholder (another C++ pointer) will be returned. MakePersist must be invoked to have the role removal to be persistently reflected. Note that it is possible to remove a role from an object even if the *placeholder* is not an instance of the corresponding persistent class, as long as the role is possessed persistently:

```
Employee *emp = ...;
Person *pers = emp rem Student;
```

A notion of a *role destructor* is provided, similarly to ordinary destructors, to "clean up" data structures when some role is removed from an object. More interestingly, if there is *no* public role destructor associated with a class, the corresponding role may not be removed from the object by any application program.

2.3.3 Reference coercion

Object references may be *coerced* to another class, in the same way as in C++. In this way the same (persistent) object may be inspected through another role and thus exhibiting itself differently. Coercions may be performed *upwards* (to a superclass, and thus restrict the set of available properties and/or change bindings for properties overridden by subclasses), *downwards* (to a subclass, and thus assuming *additional* properties), or to a *sibling* class (a class which is neither a superclass or subclass, and thus may provide a completely different set of properties). The following are some natural coercions which may originate using the class hierarchy of Figure 1:

```
Employee *emp = ...;
Person *pers = (Person*)emp;        // Upwards coercion
Professor *prof = (Professor*)emp;  // Downwards coercion
Student *stud = (Student*)emp;      // Sibling coercion
```

The coercion will return a pointer to a C++ placeholder object which is generally *different* from the original, as they reflect the same object as an instance of *different* persistent classes. Note that if the object does not *have* the role corresponding to the coercion, an exception will be raised (and the return pointer is invalid). Naturally, this may only be checked at runtime.

2.3.4 Placeholder construction

When a placeholder is constructed to reflect a persistent counterpart, it will normally be created as an instance of the persistent class corresponding to the most-specific class of the persistent object (rather than the class of the *reference*). However, according to the model of roles an object may have *multiple* most-specific classes. This cannot be expressed within C++, and thus the placeholder objects (reflecting references to persistent objects) must be created as instances of a *unique* persistent class. Different possibilities exist to decide about this class: It may the same as the class of the *reference*, but this will not work correctly in the presence of virtual functions. Rather, an object should be returned as an instance of *some* most-specific class for which the persistent object is an

[5] If the removed class have *multiple* immediate superclasses, the rem operator will return a pointer to an instance of the class which is the most-specific (unique) common superclass of these classes.

instance. This most-specific class may be selected arbitrarily by the system, or there may be some priority based on for instance which role was first (or last) added. Another possibility is to have application programs denote the class of the placeholder object by explicit coercion.

Virtual method invocations are bound to implementation *dynamically* (late binding), on the basis of the class of the *object* rather than the *reference* to the object. As there need not *be* a unique most-specific class of the (persistent) object, virtual binding will be based on the *placeholder* object, in this way taking advantage of C++ runtime provisions. This means that virtual binding is dependent on the strategy for placeholder object construction. Indeed, virtual binding is an *inherent* problem of the model, given the traditional interpretation. A more "natural" solution, may be to have virtual method implementations take into consideration which *subclasses* the (persistent) object is an instance of, and perform a computation based on this knowledge. In this way special "*combination methods*" may be implemented in terms of methods of multiple most-specific classes, or possibly explicitly *selecting* between these.

3 Category Classes

A **category class** is a specialization of the class construct with special abilities to *restrict* or *enable* object membership in the class through the association with a *predicate*. For *ordinary* classes, there are *no* restrictions on object creation and deletion, or role addition and removal, apart from the possible inhibition described by invisible (role) constructors or destructors. Fundamentally, any role may be added (explicitly) to any object, and any role possessed may be removed. Membership in category classes, however, depends on satisfaction of the category class predicate. For a *manual* category class, the explicit addition of the corresponding role to some object will only succeed if the predicate is satisfied at the time of addition. An *automatic* category class is different in that satisfaction of the predicate will *automatically* make the role be added to the object, however it will be *lost* if the predicate is later dis-satisfied.

In most respects category classes are similar to ordinary classes. They are organized in the same class hierarchy, and thus define *inheritance* and *substitutability* relationships with superclasses. Category classes associate with *properties* as ordinary classes, and object variables may reference instances of category classes (through the placeholder) just like instances of other classes. Category classes and ordinary classes are different variations of the *same* abstraction mechanism, with an additional ability for category classes to *restrict* or *enable* membership in powerful ways[6]. Category classes express special knowledge about phenomena evolution and classification in the domain of discourse, making the schema able to include *more* aspects of real-world semantics.

3.1 Category class definition

The general syntax of a category class definition goes as follows:

```
class catclass [:<superclasses>] [ WHEN <predicate> [ON <candclass>] ]
                                 [ AND|OR]  [ IF   <predicate> [ON <candclass>] ]
      {
        // class properties
      };
```

A category class (catclass) may have one or more *superclasses*, which may be ordinary or category classes. *Manual category classes* are defined by an IF expression, for which <predicate> describes a restriction to be satisfied to permit the explicit addition (using the add operator) of the corresponding role to some object. *Automatic category classes* are defined by a WHEN expression, for which <predicate> defines the criterion for the corresponding role *automatically* to be added to some object. IF and WHEN expressions may also be *combined* (using logical operators AND or OR), as will be explained below.

[6] An ordinary class may thus be regarded as a manual category class with an empty (i.e. *True*) predicate.

Candidate classes

ON expressions define the *candidate class* of the particular category class. Only objects which possess the corresponding role (is an instance of the candidate class) are valid candidates for membership in the category class. Furthermore, category class predicates may only reference properties defined for the candidate class (or superclasses of this). The specification of a candidate class is optional; if *no* candidate class is defined, the immediate superclass of the category class will apply by default. If the category class has *multiple* superclasses, the lowest common superclass will be the effective candidate class. Most often, the candidate class is a *superclass* of the category class, and so that membership in the category class will reflect a *specialization* of the candidate class role. A candidate class may also be a *sibling* class of the category class, and so that a corresponding role addition may add an additional most-specific role of the object. A candidate class which is a *subclass* of the category class is meaningless.

Predicates

Category class predicates may reference *two* aspects of objects; the *properties* defined for the candidate class, and the *role possession* of the object. Both public and private attributes may be referenced as part of the predicate specification. While *clients* of objects may only access the *public* part of class definitions, category class predicates are regarded a part of the *definition* of the class and thus may reference internal representation as well. *Method-based* predicates are permitted, but must be used with care as method invocations may have undesirable *side-effects* (Cfr. for instance [Odb92]). Moreover, automatic category class predicates are (at least conceptually) *continuously* evaluated for all candidate objects, which means that side-affect methods may have arbitrary affects on objects.

Property-based predicates are specified using ordinary C++ comparison operators and boolean operators && (*and*), || (*or*) and ! (*not*), referencing the state of objects as well as *constant* values. Predicates are *evaluated* in the context of objects which are instances of the candidate class, with predicate attributes bound to the particular object. *Role-based* predicates specify requirements on the object possession of roles. Role possession is denoted through the *name* of the class, and using C++ boolean operators any predicate over role possession may be defined to *constrain* (for manual) or *enable* (for automatic) membership in the category class. Attribute- and role-based expressions may also be *combined* in the same predicate. Examples of the different kinds of predicates are shown below.

3.2 Manual category classes

Manual category classes are defined through an IF expression, giving a predicate which must satisfied to permit the explicit addition of the corresponding role to an object (which must also possess the role corresponding to the candidate class). Furthermore, for all *superclasses* of the category class, the object must either already possess the corresponding role (be a member), or addition must be permitted (i.e. predicates possibly associated must be satisfied). This means that *multiple* roles may be added in one go. If for some reason role addition is not permitted, the operation fails and an error code is set. If addition succeeds, the role is contained until the object is deleted, the role is explicitly removed, or some automatic superclass predicate is no longer satisfied (Cfr. below). However, there is no requirement that *manual* category class predicates are satisfied *after* a successful addition.

Manual category classes are most commonly used to define *constraints* on object evolution, and thus implicitly define valid patterns of migration. Addition of a particular role to some object requires that it fulfills certain characteristics in terms of state and/or that it does/does not already possess a certain combination of roles. Manual category classes generally associate with additional properties, so that new attributes are typically initialized by a role constructor.

For instance no *person* may become a *student* if not old enough:

```
class Student : Person IF age >= 17 ON Person {
  ...};
```

This predicate is attribute-based, but manual category class predicates may also be based on the possession of roles. For instance, all full or associate *professors* may be a *PhD promoter* (however does not have to), while other *faculty members* may not:

```
class PhDPromoter : FacultyMember IF Professor || AssocProfessor
  ... };
```

Note that PhDPromoter is neither a subclass nor superclass of Professor or AssocProfessor.

Manual category classes may also be defined on the basis of predicates referencing *both* attributes and role possession. For instance, to qualify for professorship you must have earned a PhD degree and have more than 25 publications, or have been employed at the university (as a *faculty member*) more than 30 years:

```
class PhD : Person { ... };

class Employee : Person {
  ...
  int employed_years;};

class Professor: FacultyMember IF (PhD && publications > 25) ||
                                  employed_years > 30 {
  ... };
```

Note that, as Professor is a *manual* category class, promotion is not guaranteed even if the predicate is satisfied[7].

3.3 Automatic category classes

Automatic category classes are defined by WHEN expressions, which define a criterion for objects to *automatically* assume the role corresponding to the category class. Logically, all objects which possess the candidate class role (i.e. is an instance of the candidate class) are (at least *conceptually*) *continuously* evaluated for possible predicate satisfaction. If the predicate is satisfied, and roles corresponding to superclasses either *are* possessed, or addition is permitted, the category class role (and possibly "missing" superclass roles) are assumed by the object. If at some time the predicate for the class, or for some automatic category superclass, is no longer satisfied the role will become *invisible*. The same applies if the candidate class role is lost. However, if the role possession is later re-enabled the *same* role (with the same state) will *reappear*; the role is *not* persistently removed. The same also applies for any *subclasses* of the automatic category class as well: Upon dis-satisfaction of an automatic category superclass predicate subclass roles will become invisible (superclass membership is a prerequisite for membership in subclasses), but will *reappear* upon predicate re-satisfaction. Note that, in contrast, for *manual* category classes predicates are *only* evaluated upon role addition, and upon role removal all roles (including for subclasses) are persistently removed.

In some cases, an object may satisfy the predicate of *multiple* automatic category classes, although membership in these may be *mutually exclusive*. In this case predicates will be *conflicting*, and membership in either may prohibit membership in others. The system will make an arbitrary choice of class membership in these cases. Note that the system is not able to detect that such a situation may possibly arise, and thus no warning may be given at schema definition time.

While any (category) class may associate with both methods and attributes, automatic category classes often tend *not* to add attributes in addition to those inherited from superclasses. The reason is that automatic category class roles are *implicitly* added, and thus only non-argument role constructors may

[7] Which means promotion should probably not be expected for just waiting passively for 30 years ...

be invoked for the initialization of the role. More commonly, however, automatic category classes define new *methods*, or redefine inherited ones. In this way special behavior may be associated with objects as a member of automatic category classes, and for which the association is dependent on predicate satisfaction. Automatic category classes are mostly concerned about the *membership* (extent) dimension and for which flexible specification is possible. This also reflects the typical *use* of automatic category classes, described below.

3.3.1 Using automatic category classes

Automatic category classes may be used in different ways. Typically they are used to describe one or more *partitions* of a superclass based on values for attributes of the superclass, and so that each partition class contain a *subset* of the objects in the superclass. Partition classes may be *disjoint* or *overlapping*, and may be *complete* or *non-complete* partitions. Specification of completeness and disjointness is left to the schema designer, and is dependent on the ability to define predicates which reflect the semantics of the partition. The following examples specify partition classes for Person, inherently constituting a complete and disjoint partition of Person.

```
class Child : Person WHEN age < 13 {
  ... };

class Teenager : Person WHEN 13 <= age < 20 {
  ... };

class Adult : Person WHEN  age >= 20 {
  ... };
```

Adding the following means that the partition is (potentially) no longer disjoint:

```
class Retired : Person WHEN age >= 67  {
  ... };
```

Adding the following implies there will be *multiple* partitions of Person:

```
class Male   : Person WHEN sex = male { ... };
class Female : Person WHEN sex = female { ... };
```

Automatic category classes may also be applied to define various *propagations* of class membership, on the basis of different criteria. These category classes are typically defined by *role-based* predicates, and so that possession of a special combination of roles implies automatic assumption of another role. Consider the following example:

```
class GroupLeader : Employee WHEN Professor {
  ... };
```

As a general rule, every *full professor* is entitled to her own group, in which the *professor* automatically becomes the *group leader*. However, the class GroupLeader is *not* a specialization of Professor: The fact that all *group leaders* are *professors* (and vice versa) is just another *role* that inherently is played by *professors*. GroupLeader and Professor are *sibling* classes in the class hierarchy; both are subclasses of Employee.

More complex propagation specifications may be defined using C++ logical operators, e.g. to define new classes as *intersections*, *unions* or *differences* between others. Due to the notion of object roles there is *no* special restrictions wrt. the placement of such classes in hierarchy[8]. For instance, a *temporary employee* is any *employee* which is not a *faculty member* or a *technical/administrative person*;

[8] COCOON [SS91], for instance, requires that an intersection class is a subclass and a union class a superclass of the base classes.

```
class TempEmployee : Employee WHEN !(FacultyMember || TechnAdm);
```

Property and role-based predicates may be *combined*, and so that membership in the category class is based on *both* property evaluation and role possession. This may be convenient to model *conditional* propagation. For instance, a *researcher* is automatically promoted to a *senior researcher* after ten years of employment, provided she has a *PhD* degree:

```
class Researcher : Employee { ... };

class SeniorResearcher : Researcher WHEN employed_years > 10 && PhD {
  ...};
```

Two-way propagation is useful when two classes, which are *sibling* classes, should contain the *same* objects[9]. For instance, for the *employees* at some university, all *lecturers* are *faculty members*, and vice versa:

```
class Lecturer : Employee WHEN FacultyMember OR IF ... {
  ... };

class FacultyMember : Employee WHEN Lecturer OR IF ... {
  ... };
```

Lecturer and FacultyMember are *sibling* classes in the class hierarchy, so that any *person* possessing either of these roles will also (automatically) possess the other (and generally behave in different ways in the different roles). The definition of classes Lecturer and FacultyMember also includes an IF expression, ensuring that roles corresponding to these classes may be *explicitly* added as well; otherwise there will be *no* way for any object to become a *lecturer* or *faculty member*. The combination of manual and automatic predicate expressions is elaborated upon in the next section.

3.4 Manual and automatic category classes

Category classes which are *both* manual and automatic may have membership criteria defined by a *conjunction* (AND) or *disjunction* (OR) between IF and WHEN expressions (associated with the same or different candidate classes[10]):

```
class class2 : class1 WHEN pred1 ON classX AND IF pred2 ON classY {
  ... };

class class3 : class1 WHEN pred3 ON classX OR IF pred4 ON classY {
  ... };
```

class2 is defined by *conjunction*, and the corresponding role may be assumed by an object upon *explicit* addition only, and only in the case that *both* predicates pred1 and pred2 (defined on the candidate class classX and classY, respectively) are satisfied. The role will be removed from the object by *explicit* request (using the rem operator, and irrespective of predicate evaluation), or *implicitly* if pred1 is no longer satisfied. In the latter case the role may *reappear* upon possible re-satisfaction of the predicate.

Conjunction-based category classes are most interesting in cases where the role must be *explicitly* added, although may also be *automatically* lost. For instance, only *persons* which are older than eighteen may become an *employee* (assuming this is the law for some particular work). When an *employee* retires, the *employee* role is automatically lost.

```
class Employee : Person WHEN !Retired AND IF age >= 18 {
  ... };
```

[9] If one class is a *subclass* of the other, propagation into the superclass will be *implicit* due to substitutability.
[10] Most frequently they associate with the same.

This is one example on the modeling of a *transition* for which the assumption of one role necessarily implies that some other (which does not reflect a superclass) must be lost. Another examples models how an *associate professor* role is revoked when promoted to a *full professor* (Professor is not a subclass of AssocProfessor):

```
class AssocProfessor WHEN !Professor AND IF True {
  ... };
```

class3 is defined by *disjunction*, and the corresponding role is added to some object *either* implicitly by the satisfaction of predicate pred3 *or* upon explicit request (provided that predicate pred4 is satisfied). *Removal* of the role from the object is dependent on the way it was added. If the role has been implicitly added (through the satisfaction of pred3) and pred3 is no longer satisfied, the role is removed from the object. If role removal is explicitly requested, the role is removed unless pred3 is satisfied. Note that it is impossible to remove explicitly a role which have been implicitly added. Furthermore, implicitly removed roles may *reappear* upon some future re-satisfaction of the pred3 predicate.

An example of a disjunction-based manual and automatic category class is the class Retired used above. In Norway, a *person* generally retires at 67, but may continue working until 70 when she *must* retire[11]:

```
class Retired : Person WHEN age >= 70 OR IF age >= 67 {
  ... };
```

3.5 Disjoint Predicates

In many cases we find that there are natural restrictions on role possession, and which are, using category classes, directly associated with each class. However, frequently we find that the restriction is more naturally associated with a complete *collection* of classes, and so that possession of a role corresponding to either of these *inhibits* the possession of others from the same collection. For instance, possession of roles corresponding to classes Child, Employee, Retired and Dead is inherently disjoint, and may be specified as follows[12]:

```
class Child : Person IF !(Employee || Retired || Dead) { ... };
class Employee : Person IF !(Child || Retired || Dead) { ... };
class Retired : Person IF !(Child || Employee || Dead) { ... };
class Dead : Person IF !(Child || Employee || Retired) { ... };
```

As a *simplified* specification of this relationship between classes, the notion of a *disjoint predicate* is introduced. Disjoint predicates specify that no object may be a member of more than *one* class in a specified collection at the same time. If an object already is an instance of *one* of the indicated classes (or a subclass of this class), it is *not* possible to add (explicitly or implicitly) any of the other classes, unless the former class is eventually removed from the object. Using disjoint predicates, the definitions above may be rewritten as:

```
disjoint(Child, Employee, Retired, Dead);
```

Disjoint predicates may range over *both* manual and automatic category classes, but most frequently apply for manual category classes as they define inherent *restrictions* on object evolution and role combination. Automatic category classes are *enabling* in nature, and thus definitions easily may *conflict* with disjoint predicates[13]. Furthermore, disjoint predicates are solely *role based*, and thus cannot reference properties of classes participating in the predicate. Disjoint predicates represent a convenient shorthand specification mechanism, however may always be rewritten in terms of equivalent restrictions directly associated with the classes involved.

[11] In reality, rules for retirement are more complex.

[12] Other restrictions may also be associated with these category classes, but are left out here for brevity.

[13] The class Child above may, however, typically be an automatic category class.

4 Comparison

The notion of an object role is not new, and various approaches to providing more support for multi-perspectived objects and flexible object evolution have been described i literature. In most traditional object models a multi-perspectived nature of objects may only be described through *multiple inheritance*, defining special "*intersection classes*". However, these classes are *constructed* abstractions which need not reflect any natural abstraction from the real world. Furthermore, a *single* behavioral context is imposed, when multiple independent contexts are more natural, and there is a possibility for a *combinatorial explosion* of intersection classes to model all possible combinations. McAllester [MZ86] introduce a notion of *boolean classes* to alleviate this problem. Clovers [SZ89] is particularly concerned about the ability for objects to possess multiple independent perspectives, how these may be added, and how they are independently referenced. However, while new roles (or leaves in the "*clover*") may be added, they cannot be removed. Consequently, Clovers may hardly be regarded to properly support evolving objects. OORASS [RAB+92] is more concerned about the role as the fundamental concept, describing patterns of communication between roles and how instances of different types may play one role.

Other approaches are more concerned about the flexible classification of objects through predicate (over object state) satisfaction. In this way objects may implicitly evolve, on the basis of *intensional* descriptions. Many approaches to OODB *views* have been proposed, for which objects may assume membership in *virtual* classes based on the satisfaction of a *selection* predicate. However, virtual (view) classes often cannot be managed in the same way as ordinary classes. COCOON [SS91] is one example, and will be described below. ECR [EWH85] provides flexible capabilities to define how instances of a class may be specialized into subclasses, and how these may define disjoint/non-disjoint and complete/non-complete partitions of a superclass. [Cha93] presents a notion of *predicate classes*, which are similar to ordinary classes, but membership may only be assumed automatically by predicate satisfaction. Special restrictions on class membership combination may be imposed[14]. Predicate classes are similar to our automatic category classes, however are mostly concerned about the possibility for state-specific object *specializations* for dispatching purposes (and assumption of additional attributes).

Some approaches are concerned about object evolution on explicit (*extensional*) request. Aspects [RS91] allows for arbitrary addition and removal of special "*aspects*" (chunks of state and behavior) to existing objects. Multiple aspects may be added to objects, which thus may behave in a multi-perspectived manner. The aspect definition is regarded as an *extension* to some ordinary class, but will *not* inherit strictly from this class. This means that aspects do *not* integrate properly with the ordinary class hierarchy, with the possible implication that an *employee* aspect addition to a *person* may no longer be acceptable as a *person*. Iris [FBC+87] also allows for arbitrary addition and removal of types dynamically, however there is only *one* context of behavior, and thus no notion of role. [Zdo90] allows for roles to be added and removed, providing special abilities to specify that some roles are *not* removable ("essential type"), and that some roles may only be acquired upon creation ("exclusionary type"). Based on these, other restrictions on evolution may be specified, although in an awkward and unnatural way. [Ara89] defines *conversions* (change class) and *enhancements* (add class) to objects through special functions associated with the source class of the migration, and which define *all* valid migration patterns of objects.

Finally, [Vel93, SS91] provide abilities for *both* implicit and explicit evolution (classification) of objects. [Vel93] allows object to be specialized/generalized (to a subclass/superclass) upon the occurrence of an *event* (a method invocation), provided that an associated *assertion* is satisfied. As both these are *optional*, a transition may be completely *implicit* (no event), or *explicit* (no assertion). *Life cycles* (sequences of valid event occurrences) may be defined to *restrict* valid patterns of evolution. To model the fact that instances of *different* classes may play the same role, a transition to one class may originate from *different* source classes. However, in this way the destination class (denoted a *phase*) will behave *differently* from ordinary classes wrt. inheritance and

[14] These restrictions are motivated by the need to ensure that some objects may *not* be a member of multiple, most-specific classes when a binding conflict may occur.

substitutability. That is, the notion of a phase is a (partially) *different* abstraction mechanism from the class. COCOON [SS91, SLT91] describes an OODB *view* mechanism which distinguishes type (interface) and class (collection of objects with the same type). These are organized in different (but often correlating) hierarchies for property inheritance and subsetting, respectively. *Classes* may be associated with a *predicate* (over the state), stating *necessary* (corresponding to our *manual*) or *necessary and sufficient* (corresponding to our *automatic*, and reflecting a select query) conditions on membership. In this way objects may dynamically evolve (by gaining/losing class membership) implicitly or upon explicit request, which may also imply the assumption of more type. However, these types are *virtual* (i.e. *derived* from other types), and thus objects *cannot* assume new state. COCOON also provides special set-theoretic operators for the definition of view classes, with special rules for how these are to be placed in the class hierarchy). No arbitrary *propagation* of membership into *sibling* classes are, however, possible. Furthermore, no *combination* of manual and automatic specifications (which was found to be very useful in Section 3.4) may be given. COCOON is primarily a *view* approach, and thus more concerned about how virtual classes and types are to be located in the class/type hierarchies, and how they map onto base classes/types.

5 Conclusions, Contributions and Further Work

A notion of an *object role*, describing a perspective of an object as an instance of a particular class, has been presented. Roles may dynamically be added or removed from objects, according to how real-world objects evolve and exhibit themselves through multiple perspectives. *Category classes* describe *constraints* on valid evolution patterns and combination of roles, as well as defining how the possession of a role may be automatically *enabled*. Finally, category classes constitute a powerful vehicle for *conceptual modeling*, with flexible means for object classification. While the model has been presented in a *database* context, the important aspects have more general applicability as a powerful *modeling* framework or part of a programming language.

The major contributions of the approach relate to the way *flexible classification*, *object evolution* and the *multi-perspectived nature* of objects are smoothly integrated within an object model based on C++. This is achieved by *retaining* the same abstraction mechanism (the class) as the basis for classification and addition/removal of roles. In this way the importance of the class hierarchy as an organization of real-world knowledge (conceptual specialization and property inheritance) is not affected by the added modeling power. The approach *integrates* multiple classification mechanisms: *Class-based* classification is defined by ordinary classes, and *set-based* classification may be defined *extensionally* (by manual category classes) and *intensionally* (by automatic category classes). Membership in different classes may be regarded *independently*, as different *perspectives* of the same object. A particularly useful provision is the ability to *propagate* objects (possibly conditionally) into other classes, i.e. acquiring new roles on the basis of some particular (combination of) role(s) possessed. The approach allows objects to *evolve* over time with special facilities to *restrict* and *enable* object evolution based on *both* the state and role possession of the object. In this way objects may explicitly and implicitly gain and lose roles and properties dynamically, *without* compromising the class hierarchy as the conceptual organization of real-world knowledge.

A prototype OODB, incorporating the notions of object role and category classes, is being implemented to demonstrate the applicability of the ideas. Preliminary results are promising. However, the primary focus of our work is within the area of *schema versioning*, and how different schema versions may contain different versions of the same class. [Odb94] will show how the evolutionary and multi-perspectived nature of *objects* have many traits which are similar to the evolutionary and multi-perspectived nature of *classes*, and how many principles of management are the same.

Acknowledgments Svein Erik Bratsberg and Reidar Conradi are acknowledged for comments and discussions.

References

[Ara89] Constantin Arapis. Type Conversion and Enhancement in Object-Oriented Systems. In *D. Tsichritzis: Object Oriented Development*, pages 191–205. Centre Universitaire d'Informatique, Université de Genève, July 1989.

[Cha93] Craig Chambers. Predicate Classes. In *ECOOP '93. European Conference on Object-Oriented Programming, Kaiserslautern, Germany*, July 1993.

[EWH85] R. Elmasri, J. Weeldreyer, and A. Hevner. The Category Concept: An Extension to the Entity-Relationship Model. *International Journal of Data & Knowledge Engineering*, 1, May 1985.

[FBC+87] D.H. Fishman, D. Beech, H.P. Cate, E.C. Chow, T. Conners, J.W. Davis, N. Derrett, C.G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M.A. Neimat, T.A. Ryan, and M.C. Shan. Iris: An Object-oriented Database Management System. *ACM Transactions on Database Systems*, January 1987. Also in [ZM90].

[MZ86] David McAllester and Ramin Zabih. Boolean Classes. In *Proceedings of the Conference on Object-Oriented Systems, Languages and Applications (OOPSLA), Portland, Oregon, USA*, pages 417–423, September 1986.

[Odb92] Erik Odberg. What "What" is and isn't: On Query Languages for Object-Oriented Databases. Or: Closing the Gap - Again. In *TOOLS USA '92 (Technology of Object-Oriented Languages and Systems), Santa Barbara, California, USA*, August 1992.

[Odb94] Erik Odberg. *MultiPerspectives: Object Evolution and Schema Modification Management in Object-Oriented Databases*. PhD thesis, Department of Computer Science, Norwegian Institute of Technology, 1994. In preparation.

[RAB+92] Trygve Reenskaug, Egil P. Andersen, Arne Jørgen Berre, Anne Hurlen, Anton Landmark, Odd Arild Lehne, Else Nordhagen, Eirik Næss-Ulseth, Gro Oftedal, Anne Lise Skaar, and Pål Stenslet. OORASS: Seamless Support for the Creation and Maintenance of Object-Oriented Systems. *Journal of Object-Oriented Programming*, 5(6):27–41, October 1992.

[RS91] Joel Richardson and Peter Schwarz. Aspects: Extending Objects to Support Multiple, Independent Roles. In *Proceedings of ACM/SIGMOD (Management of Data), Denver, Colorado*, pages 298–307, 1991.

[SLT91] Marc H. Scholl, Christian Laasch, and Markus Tresch. Updatable Views in Object-Oriented Databases. In *Proceedings of the Second International Conference on Deductive and Object-Oriented Databases (DOOD91), Munich, Germany, December 16-18, 1991*, pages 189–207, December 1991.

[SS91] Marc H. Scholl and H.-J. Schek. Supporting Views in Object-Oriented Databases. *IEEE Data Engineering Bulletin*, 14(2), June 1991.

[SZ89] Lynn Andrea Stein and Stan Zdonik. Clovers: The Dynamic Behavior of Types and Instances. Technical report, Brown University, Department of Computer Science, November 1989. Technical Report No. CS-89-42.

[Vel93] Amandio de Jesus C. Vaz Velho. From Entity-Relationship Models to Role-Attribute Models. In *Proceedings of the 12th International Conference on the Entity-Relationship Approach, Arlington, Texas*, December 1993.

[Zdo90] Stanley B. Zdonik. Object-Oriented Type Evolution. In *François Bancilhon and Peter Buneman (Eds.): Advances in Database Programming Languages*, chapter 16, pages 277–288. Addison-Wesley, 1990.

[ZM90] Stanley B. Zdonik and David Maier, editors. *Readings in Object-Oriented Database Systems*. The Morgan Kaufman series in Data Management Systems. Morgan Kaufman, 1990. ISBN 0-55860-000-0. ISSN 1046-1698.