# Class and Type Hierarchies: Extension, Constraining, and Roles

Jürgen Schlegelmilch*

**Abstract**

With object-orientation, we model the world with objects and group objects with similar properties into classes. There are then two ways to build up a hierarchy of classes: extension adds new properties to create a subclass, while constraining restricts the values of existing properties. Programming languages support only subclassing by extension, but databases offer also subclassing by constraining via selection views. However, constraining is considered not type-safe because an object may change to no longer meet the selection criteria, thus leaving the view and dropping its type; references of this type to this object will then become invalid.

We show that support for roles allows both modes to be combined into a database programming language. Classes defined by constraining are a special case of role classes, so supporting roles allows for constraining. Type-safety is achieved by using relationships instead of references.

**Keywords:** Object-oriented programming languages, object-oriented data models, extension of types, constraining classes, dynamic roles

## 1 Overview

Programs manipulate elements of domain models, and their data model offers the means to define them. Object-oriented data models (OODMs) describe these elements as objects and group objects with similar properties into classes. From already defined classes, we can derive subclasses in two ways: either by adding new properties (extension), or by placing constraints on existing properties. While extension is offered by almost all object-oriented programming languages (OOPLs), constraining is considered not type-safe and is therefore unsupported.

Object-oriented database systems (OODBSs), on the other hand, offer constraining in the form of selection queries, used as integrity constraints for derived classes: a set of objects can be defined by posing constraints on the elements of a larger set. However, current database programming languages do not regard such a set as a class.

We show that support for roles allows both extension and constraining to be supported in a database programming language without compromising type-safety. First, we give an informal introduction in Section 2 and review existing solutions in Section 3. Then we provide the necessary definitions in Section 4 and explain them using the running example in Section 5. In

*University of Rostock, computer science department, database research group, D-18051 Rostock, Germany; WWW: `http://www.informatik.uni-rostock.de/~schlegel/`

Section 6, we discuss possible solutions and present our approach. Section 7 shows how extension and constraining can be combined into the definition of a single class, and Section 8 concludes the presentation with a summary.

## 2 Introduction

Classes[1] group objects with common properties together. These commonalities can be found either in the structure, called the *type*, or in the properties, called the *state*, of the objects. So, a class has two aspects: a type, describing the structure of its objects, and a condition, describing the state of its objects. There are correspondingly two ways to derive new classes from existing ones:

**Extension** defines a class with a subtype, i.e. a type with more properties.

**Constraining** yields a class with a stronger condition.

In both cases, objects of the new class are also objects of the base class and can be used as such; this is called substitutability.

**Example 2.1:** Both extension and constraining are natural forms of class definitions: Given a class `PERSON` with attributes `name` and `address`, we can define a class `STUDENT` by extending `PERSON`s with a student ID number and a university. On the other hand, we can derive a class `NEW_YORKER` from `PERSON` by constraining the attribute `address` to contain `'New York'`. ◇

Methods are operations defined in classes; they are the only way to manipulate objects. Update methods may change the state of an object, and thus may leave the object in a state violating the class condition. There are two possible approaches to this problem of constraining:

- Always maintain the class condition, so objects are kept consistent. Methods that may leave an object inconsistent must be redefined.
- Move the object out of the class; this is called demigration.

**Example 2.2:** Suppose we have a method `move(new_address: STRING)` in class `PERSON` that updates the attribute `address` according to its argument. In class `NEW_YORKER`, we now have the choice:

- Redefine `move` so that its argument must contain `'New York'`.
- Let `move` migrate an object out of class `NEW_YORKER`; it will still be in class `PERSON`.

◇

Both approaches have drawbacks:

- Methods defined in the base class may have to be redefined in the derived class to respect the stronger class condition, thus possibly becoming incompatible with their original definition due to covariance. So, objects of the derived class are no longer substitutable for objects of the base class.

[1] We assume basic familiarity with object-oriented concepts, e.g. from [Mey97]. Most notions like class, type, base class, derived class etc. are defined formally in Section 4.

- Variables annotated with the derived class may refer to the object. Applying a method may remove the object from the class, so the variable is then either ill-typed, or holding a dangling reference, or must be set to a null value.

**Example 2.3:** If we have two variables annotated with types `PERSON` and `NEW_YORKER`, respectively, that are referring to the same object, then applying the method call `move('Washington')` to the first variable shows the problem:

If method `move` has been redefined in `NEW_YORKER` to not accept this argument, then `NEW_YORKER`s are no longer `PERSON`s because they cannot be used as such; both variables are then not allowed to refer to the same object, thus avoiding the problem.

If alternatively this application of `move` removes the object from `NEW_YORKER`, then the second variable must no longer refer to it, for example by setting it to the null value `void`. To do so, we have to know about all variables annotated with class `NEW_YORKER` that refer to this object. ◇

A database programming language has to offer a solution to this problem, or to drop support for class definition by constraining.

### Generalising the problem

In typical OOPLs, methods cannot change the structure of objects, so there is no similar problem with extension. Any method can be safely applied to an object without references becoming ill-typed. However, OODBSs hold objects for longer periods of time and have to reflect possible type-changes.

**Example 2.4:** A person may become a student by enrolling on a university, and eventually ceases to be a student when graduating. So, a method `enrol` should be applicable to objects of class `PERSON`, migrating them into the class `STUDENT`, and likewise a migrator `graduate` for `STUDENT`s to move them out again. If there is then any variable annotated with class `STUDENT` referring to such an object, it is left ill-typed. ◇

The classes that an object can acquire or drop dynamically are called *roles*, and OODMs supporting this kind of type-change are called *role models*. They include special methods called *migrators* (`enrol` and `graduate` in Example 2.4) to change the class and type of objects. It is clear that in these models the demigration problem shows up even without constraining. The only difference is that in role models migration is always performed explicitly while with constraining migration can be implicit as a side-effect of some updates.

## 3 Related work

To our knowledge, for constraining the problem was first presented in [ZM90]. In this paper, four properties of class hierarchies are shown to be incompatible:

**Mutability** Immutable objects may not change their state; this would disallow the method `move` and therefore the implicit migration.

**Substitutability** Objects may not be attached to variables declared for superclasses; this would disallow variables annotated with type `PERSON` to refer to `NEW_YORKER`s and avoid the polymorphic application of `move` with inappropriate arguments to objects of that class.

**Static type checking** At run-time, we can decide whether the object attached to a variable actually is a `NEW_YORKER` or not, and react accordingly to avoid type errors.

**Subclassing by constraining** This would disallow the definition of class `NEW_YORKER` by constraining the class `PERSON`.

According to [ZM90], one can only combine three of these properties into a single language. However, we will show in Subsection 6.1 that these properties are sufficiently local to narrow this restriction to single class hierarchies.

**Cecil** In the object-oriented programming language Cecil, so-called predicate classes [Cha93] are derived from base classes by constraining with a predicate. These classes have to satisfy some properties: predicate classes must not redefine common methods unless they are either ordered or disjoint, and all predicate classes of a set of base classes must partition these base classes and have the same set of methods. This ensures that objects in the base classes always have the same set of methods and only one implementation for each method, but also disallows arbitrary constraining and extension and is therefore not a general solution. The intended usage for predicate classes is to model state-dependent methods of their base classes. Thus, the predicates each describe a partition of the possible states in the base classes. [Cha93] provides a comparison of other approaches that use disjointness and coverage, all of them using explicit declarations instead of predicates and inference.

**Eiffel** The programming language Eiffel 3 [Mey92] offers class conditions and constraining, but no migration; thus the conditions are invariants. Derived classes may strengthen the class invariants of their base classes and have to redefine methods that may leave objects in inconsistent states. Objects of such classes are then no longer substitutable for objects of base classes. A set of rules called CAT rules extends the type check to prevent substitutions. Again, this solution leaves the programmer alone with the problem. Even worse, there is no syntactic difference between a derived class whose objects are substitutable, and other subclasses, and the compiler does not enforce the redefinition of inherited methods when the invariant has been strengthened.

**Fibonacci** The database programming language Fibonacci [AGO95] offers migrators, but no class conditions. Fibonacci allows objects to migrate into classes with a subtype, but not out of classes; because of substitutability this is type-safe, so variables do not have to be rechecked. Fibonacci can therefore use a static type check without compromising type-safety. On the other hand, this solution does not help the programmer since it makes modelling the application domain very hard.

**DOOR, BCOOL** The database object model DOOR [WCL96] and the functional object database language BCOOL [LS93] offer migrators that allow objects to gain and loose types freely. References that became ill-typed due to an object dropping a type are set to a null value. However, this requires to check the whole database for such references and therefore does not scale well.

**LOOM** The knowledge representation language LOOM [Bri93] offers class conditions, constraining, migration, mutability, and substitutability; it consequently drops static type checking.

Methods are not tightly bound to classes; rather, their applicability is defined by predicates called situations, making them more flexible and deferring the class membership test to run-time. LOOM is based on predicate logic, and the run-time system includes an inference mechanism that makes the applicability check quite powerful. Here, the programmer has all means to model the application domain closely, at the cost of possible run-time errors.

**Role models** Several role models have proposed solutions:

- In [GSR96], roles are themselves objects that are components of other objects, so application domain objects are represented by hierarchies of implementation objects. Migration is performed by manipulating the internal hierarchy of the object. The underlying language Smalltalk [GR83] supports only automatic memory management, so role objects are kept alive as long as there are references to them. The owning object may have dropped the role (migrated out of a class) long before.
- [RS91] introduces role objects just as in [GSR96] and calls them aspects; however, aspects may hide features of their base object and are therefore not substitutable for them. [RS91] proposes to disallow the deletion of aspects as long as there are references to them, without discussing an implementation.

**Database views** Views in OODBSs provide a means to define classes by constraining. However, most approaches [Mot96] do not address the demigration problem but concentrate on issues like positioning of derived classes in the class and type hierarchy, combining constraining with extension, and updatability of objects in derived classes. The latter ability will introduce the demigration problem into views.

**Unique references** Still another alternative to control the effects of object migration are unique references. In example 2.3, the alias problem has been shown to be one of the sources of the problems of constraining. Avoiding aliasing is therefore one way to minimise the problems, and unique references are the means to avoid aliasing [Hog91]. A unique reference is defined as a reference that is guaranteed to be unique, i.e. there is no other reference to the same object. However, unique references disallow the sharing of objects, and their domain of use is therefore restricted to hierarchic structures. Using unique references, we can keep the effects of demigration local: only the variable referencing the receiver of an update can become ill-typed, and this can be handled by a local exception handler.

# 4 Definitions

We now define an object model that supports constraining and migration.

## 4.1 Signatures, types and their hierarchies

Types are sets of operation signatures, where a signature consists of the method name, the number and types of the arguments, and the result type[2]. We require the method name to be

[2] Abstract data types also include a set of axioms describing relationships among the signatures; these are not relevant to type checking.

unique within a type as a means of identification, and we also require all signatures of a type to contain the type at least once. The implementation of a type consists of a sort and a function for each of its signatures; a sort is a set of attributes and its elements are tuples from the cartesian product of the attributes. An element of the type is an element of the sort.

Types form a hierarchy: if a type $T$ supports at least the operations of a type $U$, it is called a subtype of $U$ ($T \preceq_{type} U$). It depends on a corresponding hierarchy on signatures where subsignatures of a signature $s$ can safely handle argument lists intended for a call to $s$.

**Definition 4.1 (Signature hierarchy):**
Let $S = n_s : s_1 \times \cdots \times s_k \rightarrow s_r$ and $T = n_t : t_1 \times \cdots \times t_l \rightarrow t_r$ be signatures.

$$S \prec_{sig} T \iff$$

$$\begin{array}{lll} & n_s = n_t \wedge k = l & \text{(a)} \\ \wedge & s_r \preceq_{type} t_r & \text{(b)} \\ \wedge & \forall_{i=1\ldots k}\, t_i \preceq_{type} s_i & \text{(c)} \end{array}$$

Thus, the names and number of arguments must be equal (a), the type of the the result may vary with the hierarchy (b), and the types of the arguments may vary against the hierarchy (c). This relation on signatures is called *contravariance* [Cas95]. □

The signature hierarchy allows types to not only add new operation signatures but also change those they have in common with any supertype. Thus, the subtype relationship is defined as follows:

**Definition 4.2 (Type hierarchy):**
Let $T = \{t_i | i \in I\}$ and $U = \{u_j | j \in J\}$ be types with signatures $t_i$ and $u_j$, respectively, for some finite index sets $I, J$.

$$T \preceq_{type} U \iff \forall u_j \exists t_i : t_i \preceq_{sig} u_j$$

where $\preceq_{sig}$ already assumes $T \preceq_{type} U$. □

Note that a subtype may add new signatures arbitrarily; this is called type *extension*.

Variables are annotated with types; they may only refer to objects having that type. A variable is called *polymorphic* if objects with different types can be attached to it. The subtype hierarchy allows to statically check attachments to polymorphic variables: since elements of a subtype support all operations of their supertypes, an attachment is type-safe if the type of the source has a subtype of the type the target variable is annotated with.

## 4.2 Objects and classes

OODMs are built around the notion of objects: an object is an immutable identity and has an associated state. A class $C$ consists of a domain $\mathsf{dom}(C)$ of possible objects, a type $\mathsf{type}(C)$ describing the structure of their (local) state, and a condition $\mathsf{cond}(C)$; the extent $\mathsf{ext}(C) \subseteq \mathsf{dom}(C)$ of a class is the set of its existing objects. A class maps objects of its extent to elements of its type, defining the local state of the objects.

$\mathsf{cond}(C)$ is a term of some predicate logic relating the results of method applications on the state of an object of the class. An element of the type is a valid state for an object of the class if it satisfies the condition.

Objects are manipulated by sending messages to them. Valid messages cause the implementation associated to a matching signature to be executed. This run-time matching is called method lookup or late binding. It allows the type of the first argument, the object, to vary with the type hierarchy, thus in Definition 4.1:

$$s_1 \preceq_{type} t_1$$

without compromising type-safety with static type checks.

Method implementations may change the state of the object; conceptually, they associate a new state with the object (that is, its identity). So, a class divides the operations of its type into selectors that leave the object unchanged, and modifiers that may change the state. Modifiers that do not take the state of the object as an argument are called constructors because they can be used to build the first state of the object after its creation with `new`. *Migrators* are special modifiers that move objects into and out of class extents; their result type is usually different from the type of the class. Modifiers that are not migrators must have the type of their class as result type. A class is considered a *role class* iff there are migrators for it.

Finally, we note that the changes caused by modifiers are visible only via selectors:

**Definition 4.3 (Modifier for a selector):**
A modifier $m$ is called a *modifier for selector s*

$$\iff \exists \text{object } o, \text{values } v_i \colon s(o) \neq s(m(o, v_i))$$

So, applying $m$ to $o$ causes a visible change in the state of $o$. □

## 4.3 Subclasses

Objects can be in (the extent of) many classes; the resulting subset hierarchy is called the class hierarchy. Classes have to be placed into this hierarchy using a binary relation $\prec_{class}$ among classes:

**Definition 4.4 (subclass):**
Let $A, B_i$ be some classes. If $A \prec_{class} B_i$ holds, then $A$ is called a *subclass* of each $B_i$ ($B_i$ a *superclass* of $A$), and $\mathsf{ext}(A) \subseteq \mathsf{dom}(A) = \bigcap_i \mathsf{ext}(B_i)$. Therefore, objects in $\mathsf{ext}(A)$ have both $\mathsf{type}(A)$ and all $\mathsf{type}(B_i)$; in general, the *global type* of an object is the union of the types of all classes it is in, which is a subtype of the type of any such class. □

We require that no conflicts occur among the signatures in the union; conflicts among implementations are resolved according to the class hierarchy (see [Sch96b] for details). For a class, we call the union of the types of its superclasses its *inherited type*. Superclasses are often called *base classes* in the context of class definitions.

Now we can define some notions to talk about classes in hierarchies:

**Definition 4.5 (direct subclass):**
A class $C$ is called a *direct subclass* of a class $E$ if

$$C \prec_{class} E \wedge \neg \exists D : C \prec_{class} D \prec_{class} E$$

□

Migrators can move objects only into direct subclasses. The result type of migrators (in this stricter sense) must be the type associated with the target class, while that of demigrators must be the empty type. Also, demigrators of constrained and derived classes must not take arguments besides the object that has to be demigrated because they have to be called implicitly.

After placing a new class into the class hierarchy with $\prec_{class}$, we can define its local type and condition:

- Specifying a type results in type extension.
- Specifying a condition can make the class a constrained class.

Note that objects of a superclass are not automatically objects of a subclass; they have to be migrated explicitly. Views in object-oriented databases [Mot96] and predicate classes in Cecil [Cha93] define classes where objects of superclasses migrate automatically if they meet the condition; because of substitutability migration into a class is type-safe. We call a class *derived* iff objects migrate into this class implicitly. A derived class is always defined by a query and therefore a constrained class.

We now have to define when a class is considered a constrained class. First, we allow the condition of a class to contain signatures of the types of its superclasses. This can be used to either restrict the corresponding state, or to relate the new local state to that defined in superclasses; these conditions involving selectors of the new type are not considered constraining.

**Definition 4.6 (constrained class):**
Let $\mathsf{closure}_C(t)$ denote the transitive closure of a term $t$ with respect to $\mathsf{type}(C)$, i.e. $t$ enriched with all transitive comparisons[3] that involve signatures from $\mathsf{type}(C)$, and $\pi[S](t)$ the projection of $t$ on signatures in $S$, i.e. $t$ with all minimal subterms removed that contained a signature not in $S$. We call a class $E$ *constrained* iff its condition is strictly stronger on its inherited type than the condition of a superclass $C$:

$$\begin{aligned} E \text{ constrained} \iff \exists C, D\colon \quad & E \prec_{class} D \prec_{class} C \\ \wedge \quad & \pi[\mathsf{type}(C)](\mathsf{closure}_C(\mathsf{cond}(D))) \\ & \quad \not\Rightarrow \pi[\mathsf{type}(C)](\mathsf{closure}_C(\mathsf{cond}(E))) \end{aligned}$$

We call $E$ *directly constrained from* $D$ iff this class $D$ is a direct superclass of $E$. □

Applying an update method to an object of a constrained class can leave the object in a state violating the class' condition; it must consequently demigrate from that class and all its subclasses. It is therefore sufficient to check only the conditions of directly constrained classes, in order to find out from which classes an object may have to demigrate.

## 4.4 Classes as types

Classes can be used as types in most OOPLs. This has two aspects:

1. Variables annotated with a class may only refer to objects in the extent of the class. In databases, this is called referential integrity; in programming languages, variables bound to objects not in their class are called dangling references.

[3] Note that we need information which user-defined signatures denote transitive relations. For ADTs, transitivity can be expressed in the axioms.

2. Messages sent to the object attached to a variable are executed against the state of the object[4]. So, the state of objects attached to variables annotated with a class must be of a subtype of the type of that class.

Since subclasses are subsets and have (global) subtypes (both in the non-strict sense), it is type-safe to bind objects of a class to any variable annotated with a superclass; this is called substitutability.

# 5 The running example

Here is the introductory example with all the concepts introduced so far:

**Example 5.1 (Class PERSON):**
The following class[5] models Persons:

```
class PERSON
  creation birth;                                  -- allow make as constructor
  feature {NONE}
    STRING name, address;                                            -- the sort
  feature
    birth(a_name, an_address:STRING)                               -- constructor
    get_name:STRING                                            -- first selector
    get_address:STRING                                        -- second selector
    move(new_address:STRING)                                          -- modifier
  end
```

`name` and `address` are private features of the class, thus defining the sort of its type, all the others are public and constitute `type`(`PERSON`):

$$\{\begin{array}{l} \texttt{birth} : \texttt{STRING} \times \texttt{STRING} \rightarrow \texttt{PERSON} \\ \texttt{get\_name} : \texttt{PERSON} \rightarrow \texttt{STRING}, \\ \texttt{get\_address} : \texttt{PERSON} \rightarrow \texttt{STRING}, \\ \texttt{move} : \texttt{PERSON} \times \texttt{STRING} \rightarrow \texttt{PERSON} \quad \} \end{array}$$

The first signature `birth` is a constructor, the second and third `get_name` and `get_address` are selectors, and the last `move` is a modifier. ◇

From this base class, we can define the subclass `STUDENT` by type extension:

**Example 5.2 (Class STUDENT):**

```
class STUDENT inherit PERSON
  creation enrol
  feature
    student_id:INTEGER
    enrol(a_person:PERSON)
  end
```

[4] except for copying and assignment, of course.
[5] We use an Eiffel 3-like notation

The inherit clause defines STUDENT $\prec_{class}$ PERSON, and type(STUDENT) is

$$\{ \quad \mathtt{enrol} : \mathtt{PERSON} \rightarrow \mathtt{STUDENT} \\ \mathtt{student\_id} : \mathtt{STUDENT} \rightarrow \mathtt{INTEGER} \quad \}$$

The method student_id is a selector, and the modifier enrol should be a migrator. Thanks to ext(STUDENT) $\subseteq$ ext(PERSON), students also have a name and an address; in OOPLs, this is called *inheritance*. ◇

In Eiffel 3, the method enrol does not migrate its receiver into class STUDENT; instead the receiver object remains unchanged, and the method result is a new object. In our model, enrol is a migrator that inserts the receiver into ext(STUDENT) and initialises the attribute student_id appropriately. Eiffel has no syntax for demigrators (or destructors, as they are called in e.g. C++ [ES92]), so we cannot define graduate; it would have the signature

$$\mathtt{graduate} : \mathtt{STUDENT} \rightarrow \emptyset$$

indicating that the properties of STUDENTS are lost for an object after its demigration. Unlike C++ destructors, a demigrator does not delete its argument object but instead moves it out of a class extent; if it is still in other classes, it will survive the operation.

The class NEW_YORKER is an example for a constrained class: it should hold all objects of class PERSON with an address containing 'New York'. The following class definition tries to capture this constraint:

**Example 5.3 (Class NEW_YORKER):**

```
class NEW_YORKER inherit PERSON
  invariant get_address.contains('New York')
  end
```

The type associated with class NEW_YORKER is the empty set since it does not define new features. We will examine the combination of constraining with extension in Section 7. ◇

However, as the Eiffel 3 keyword invariant implies, no object of this class may move out of New York. The modifier move as inherited from (the type of) class PERSON may not be called with arguments that violate the invariant, rather than migrating the object out of class NEW_YORKER. This restriction is implicit in Eiffel 3 and only checked when assertion checking is enabled.

In some OODBSs, we can define a view DB_NEW_YORKER:

**Example 5.4 (View DB_NEW_YORKER):**
Using the query language OQL of the ODMG database standard [CBB+97], we can select all persons that have an address containing 'New York':

```
define DB_NEW_YORKER as
  select p from Person p
  where p.address like '%New York%'
```

This assumes the name of the extent of class PERSON to be Person, and attaches the name DB_NEW_YORKER the query. ◇

However, in the ODMG standard, this defines only a set of objects of class PERSON, rather than a new derived class. The reason is the data model of the standard which was developed by combining the data models of three OOPLs, so the standard model inherits their restrictions: classes only define the structure of objects, but are not sets of objects. Constraining is therefore unsupported in this model.

# 6 Solutions

In Section 2 (Example 2.3), we have seen that constraining is dangerous because an update may require demigration of an object, and most current OOPLs are lacking support for migration. It follows that adding support for object migration allows for constraining. However, the new problem is not easier to solve than the old one. So, before discussing object migration in general in Subsection 6.2, we offer a solution for constraining only.

## 6.1 Constraining reconsidered

To avoid the problem of constraining, [MD94] proposes to disallow the definition of subclasses in this way altogether, in favour of mutability, substitutability and static type checking. However, this decision is not appropriate for many application domains. For example, in mathematics all objects are immutable and constraining is common, so we would rather drop mutability. In fact, [MD94] is too pessimistic: all four properties can be combined into a single language, although not in a single class definition.

Mutability and constraining are mutually exclusive, if we want to retain static type checking and substitutability. However, based on Definition 4.6 we can push the choice between constraining and mutability into the class definition:

**Lemma 6.1:** A modifier for a selector $s$ is a migrator for any constrained subclass with an invariant involving $s$. Therefore, we have the choice:

- If there is a modifier $m$ for selector $s$, we disallow the definition of constrained subclasses with invariants involving $s$ since in any such subclass $m$ would be a migrator.
- If a class is constrained by an invariant involving selector $s$, we disallow the definition of modifiers for $s$ for the same reasons.

So, a class is either mutable or constrained with respect to selector $s$. $\square$

This policy avoids implicit object migration caused by updates, and makes constraining practicable without excluding mutability, substitutability and static type checking from the whole language. However, it still disallows many class definitions where constraining would be natural.

**Example 6.1:** Class `PERSON` is mutable with respect to selector `get_address` because of the modifier `move`. Therefore, we are not allowed to define the class `NEW_YORKER`. This requires the programmer to manually make sure that `PERSON`s are really `NEW_YORKER`s where they should be. $\diamond$

## 6.2 Managing object migration

While avoiding migration as described in Subsection 6.1 looks like a solution, it is generally preferable to handle it because of the benefits in modelling power. There are several proposals for role models but none handles the demigration problem satisfactorily (see Section 3). We found two ways to cope with demigration:

1. disallow the annotation of variables with constrained classes, so no variable can become ill-typed because of a demigration, or
2. modify ill-typed variables after a demigration, by taking advantage of a mechanism for general relationships.

The first solution is very limiting, but can be made practical with suitable support; this is discussed in Subsection 6.2.1. The second one offers a general solution, but adds some overhead; we present its details in Subsection 6.2.2.

### 6.2.1 No variables of constrained classes

The demigration of an object from a class will leave variables annotated with that class that reference this object ill-typed. If there are no variables annotated with a constrained class, they trivially cannot become ill-typed. This is the approach taken by most object-oriented database systems: even though they support selection views, they do not regard them as classes, and consequently one cannot annotate variables with them.

However, even if we accept these sets of objects as classes, this solution makes constrained classes less useful because there is no way to access their local features. A dynamic type check facility can help here:

Type guards like in Oberon [WR92] can help to simulate local variables of constrained classes, because they provide a dynamic type test. A type guard controls a block by narrowing the type of a variable in that block: if the object bound to the variable does not conform to the type, the block is skipped. So, if the block is executed, it can safely assume that the object bound to that variable has the required type, which can be that of a constrained class. However, the object may not migrate out of that class within the scope of the block, so updates are not possible, except for the very last statement in the block. Due to late binding, it is hard to predict which update methods can safely be used, and due to aliases, even method applications to objects attached to other variables might really effect the object in question. Thus, the controlled block may only contain calls to selectors, plus an optional last call to a modifier on the constrained variable.

**Example 6.2:** In Example 2.3, no variable may be annotated with class `NEW_YORKER`. If we want to access parts specific to `NEW_YORKER`s, for example the club they visit, we have to use a type guard:

```
local p:  PERSON
with (p is NEW_YORKER)
  do
    -- p has type NEW_YORKER in this block
    System.out.println(p.club)
  end
```

◇

Note that this restriction is not necessary for role classes that are not constrained because with them demigration is explicit. If you do not call a demigrator, directly or indirectly, then you can annotate local variables with role classes. Since it is possible to determine statically whether a relevant demigrator is in the closure of called methods, we can apply a static check. Because of late binding, we have to consider all implementations of methods in all subclasses when building the closure.

### 6.2.2 Using a relationship mechanism

After a demigration, some variables may be left ill-typed. To avoid type errors due to such dangling references, it is necessary to either redirect them, or set them to a null value. However, this amounts to browsing the whole set of objects of the current program (or even worse, the database, in case of a database programming language) for such references, plus local variables in methods up the call chain. This is clearly undesirable, and should be avoided.

Fortunately, some object-oriented database models offer a relationship mechanism that helps managing this task. Relationships describe relations between objects that are navigable in all directions, thus allowing to find any object holding a reference to a given one. Several relationship mechanisms have been proposed in the literature [Rum87, DG90, AGO91] including one for the OODB standard ODMG 2.0 [CBB$^+$97]; we use the one presented in [Sch96a].

**Definition 6.2 (Relationship):**
A *relationship* $R$ consists of a relation schema, a condition, and an exception policy. The relation schema is a set of attributes, some of them of class types (references). Like a class, a relationship has an extent $\mathsf{ext}(R)$ which is a relation over the given schema. Each tuple in the relation describes a *link* between the objects in the object-valued attributes. The condition describes valid tuples, with the special case of cardinality constraints that restrict the number of tuples. The exception policy specifies the behaviour in case of integrity violations; possible reactions are removal of the offending tuples or abort of the transaction. □

The relationship mechanism introduced here is presented in more detail in [Sch96a]; a persistence specification based on it is presented in [Sch96c].

Object-oriented systems prefer an object-centred view on the world, and relationships follow this preference by offering a per-object view on their extent. All objects referenced from an attribute of a relation form a derived class (defined by selection); in this class, we can define methods to select the objects related to a given one, thus giving the illusion of a simple reference. Such methods can be defined for any class of objects referenced from attributes of the relation, allowing navigation in all directions. It is therefore possible to find all links that an object participates in, by simply selecting tuples from relations. These relations are generally much smaller than the set of all objects.

Relationships also allow to react flexibly on integrity violations like the dangling reference problem shown in Section 2. Both properties together make implicit migrations harmless: Suppose an object participating in a relationship migrates out of the class that the relationship assumes. This constitutes an integrity violation, and the exception policy of the relationship is checked:

- The standard behaviour is to remove all referencing tuples. This is equivalent to setting referencing variables to a null value, and thus avoids type errors.
- If this is inappropriate, we can take advantage of database semantics and specify to abort the transaction. This rolls back the change that caused the demigration, and is suitable whenever the object must be in that class as long as the link exists.

The suitable policy depends on the application domain:

**Example 6.3:** Consider a library in New York and its customers. The library may not want customers to move away if they still have books, so it specifies the relationship `BORROWED_BOOKS` with attributes `the_book: BOOK` and `the_customer: NEW_YORKER` and chooses the abort policy. Each tuple in the extent of `BORROWED_BOOKS` describes who has borrowed which book. If a customer tries to move away from New York and still has a book from the library, the demigration will cause an integrity violation check on `BORROWED_BOOKS` and consequently an abort of the transaction: the customer must not leave New York with the book.

On the other hand, the New York clubs mentioned in Subsection 6.2.1 will have to let members leave them and therefore choose the removal policy for their relationship `MEMBERSHIP`. If a member migrates out of class `NEW_YORKER`, the integrity check will remove the tuple with the dangling reference. So, the object will simply cease to be a club member. ◇

Relationships are powerful but introduce some overhead. If the tuples are really stored in a relation, then updates are simple but navigational access may be slower. If the tuples are stored distributed in instance variables in the related objects, then updates require consistent changes in all objects [The95] but navigational access is fast.

### 6.2.3 The Perfect Mix

The second solution is powerful enough to cover all aspects of demigration, but using relationships adds some overhead, so we combine both solutions: For inter-object references, relationships must be used, while we avoid their use for simple local variables in methods. For these, constrained classes still cannot be used as types; this is no improvement over existing programming languages.

As a result, we are able to support demigration, whether implicit or explicit, allowing both constrained classes as well as general roles. There is no restriction on the use of these concepts in the data model, and only local variables in methods are restricted to non-constrained classes.

We note that inter-object references are sufficient if they are annotated with non-constrained classes, so demanding relationships for all links between objects is a bit of an overdose. However, relationships have a number of other advantages over references (see [Sch96a, Rum87, AGO91] for details), and using only one concept in the data model avoids confusion. Also, modern object-oriented analysis and design models like OMT [RBE$^+$91] and UML [HW97] model links between objects exclusively with relationships.

## 7 Combining extension and constraining

Defining classes by constraining is uninteresting if these classes cannot have additional local state or methods. It is therefore necessary to check how extension and constraining can be combined. Classes can be extended in three ways:

- local state to hold additional information
- new methods to manipulate the new state, or to offer functionality that only applies to objects of the constrained class
- new implementations for inherited methods, typically in the form of additional actions

We will now examine each of these ways.

## 7.1 Extension by local state

Adding local state in a constrained class is possible, but the corresponding selectors can only be accessed if the object is known to be in the class defining them (or a subclass). This is discussed in the next Subsection 7.2. The state itself, i.e. the element of the sort and its attributes, is directly accessible only in implementations of methods of the class.

Each object of a class is mapped to its local state, as specified by the modifier of this class that was last applied to the object. It follows that constructors and migrators of a class determine the first local state; this is called *initialisation*.

Because objects migrate implicitly into derived classes, there is no way to initialise local state in these classes by arguments. The modifier that caused the migration cannot initialise them because it is defined in a superclass. It follows that the derived class must have a parameter-less constructor; the initial state can thus only be derived from the current state of the object (and related objects). There is no similar requirement for constrained or role classes.

**Example 7.1:** In the derived class `DB_NEW_YORKER` from Example 5.4, we can define a new attribute `since_when_in_town` to record the arrival date, and a selector `years_in_town` to calculate how long someone resides in New York. The migrator will then need to initialise `since_when_in_town` to the current date, when an object migrates into the derived class due to a call to the modifier `move`. ◇

## 7.2 Extension by methods and implementations

For new methods, there is no initialisation problem because implementations depend on classes, not on individual objects. But due to static type checking, a method can only be applied to objects that are known to be in a class with a type that contains this method. Variables are annotated with types so that only objects with conforming types can be attached to them. Since we disallow the annotation of variables with constrained classes, selectors of these classes can only be accessed via relationship links, inside the scope of a type guard, or, thanks to late binding, in other methods of these classes.

**Example 7.2:** To access the selector `years_in_town` introduced in Example 7.1, we either need a link from another object, e.g. from a club via the relationship `MEMBERSHIP` (see Example 6.3), or a type guard similar to that in Example 6.2. In the implementation of `years_in_town`, we could access any method defined (or inherited) in class `DB_NEW_YORKER` without such hassle because late binding will execute this implementation only for objects in that class. ◇

Adding implementations for inherited methods can be done in two ways, depending on the language support:

**conventional languages:** redefine inherited methods

**event-based languages:** associate additional actions to events

Both approaches are described in the following subsections.

### 7.2.1 Redefiniting methods

Conventional OOPLs map method calls to function executions, and let subclasses redefine this mapping. This is achieved by supplying an implementation for an inherited method. Due to late binding, such an implementation will completely replace the one in superclasses. As a consequence, it is not possible to simply add an action; the new implementation has to explicitly call the inherited implementation to achieve that effect. This approach is more flexible than the event-based one, but introduces problems with late binding.

Late binding will execute the implementation of the most specific class an object belongs to. Now, we allow objects to be in several classes simultaneously, and there is often not a unique highest lower bound for any set of classes that an object belongs to, and even if there is, the object does not necessarily belong to that class. [Sch96b] presents an algorithm that adds conflict resolution classes to a given class hierarchy to make late binding unambiguous even in the presence of role classes.

All role models have this method lookup problem if they support late binding. However, constrained classes make the situation even worse: the relative position of constrained classes in the class hierarchy is undecidable in general [HS91a, HS91b] because of the constraining predicates, so we have to assume that constrained classes with the same base classes are incomparable siblings. We can provide means to place them explicitly in the class hierarchy so the programmer can decide. Once the hierarchy is unambiguous, we can apply the conflict resolution algorithm presented in [Sch96b].

**Example 7.3:** If we have derived classes `DB_NEW_YORKER` and `DB_BOSTONER` as defined in Example 5.4, we can declare them to be disjoint. The conflict resolution then does not have to solve conflicts between implementations in these classes.

Now assume a derived class `LIBRARY_CUSTOMER` that we can define by projecting the relationship `BORROWED_BOOK` from Example 6.3 on its attribute `the_customer`. We can infer that it is a subclass of `DB_NEW_YORKER`, so there is also no method lookup conflict possible.

If, on the other hand, we define the class of `CLUB_MEMBER`s by projecting the relationship `MEMBERSHIP` from Example 6.3 accordingly, it is not clear whether it is a subclass of the class `LIBRARY_CUSTOMER` or not, or vice versa. If there are implementations for the same method, we have to either order them, or introduce a subclass for their intersection, as described in detail in [Sch96b]. ◇

### 7.2.2 The event-based approach

OOPLs based on events are able to associate several actions in different classes with an event. If an event happens, all associated actions are preformed in parallel. If the event is equivalent to a selector, all results have to be combined into a net result by means of an aggregation function (see [GSR96, HSJ+94] for examples of such languages). In constrained classes, you can associate an action with an event that is already handled by superclasses; since events are very similar to method calls, this action can be seen as an extension of the implementation of event.

**Example 7.4:** If we assume that our model is event-based, we can add an action to the method/event `move` in class `STUDENT` to let the university know about the new address.

```
class STUDENT inherit PERSON
                                          -- all the features as in Example 5.2
```

```
feature
  get_university:UNIVERSITY                      -- queries a relationship
  move(new_address:STRING)
    is also                                              -- invented syntax
      get_university.notify(new_address)
    end
end
```

If modifier `move` is applied to an object of class `STUDENT`, both actions from classes `PERSON` and `STUDENT` will be executed, causing the change of address and notification of the university about it. ◇

This approach is simple and safe but limited to pure additions of actions. There is no way to modify actions defined in superclasses, or to optimize them by replacing them with algorithms that can take advantage of properties of the subclass. Also, the aggregation of partial results may not be suitable to calculate the net result, but there is no way to process intermediate results by different aggregation functions. On the other hand, this approach is well suited to support constrained classes because it does not require the classes to be explicitly placed in the class hierarchy.

## 8 Conclusion

Subclassing by constraining is an important modelling concept that lacks support in current OODMs. Besides giving guidelines how to use constraining safely in OOPL data models, we have shown that role support is sufficient to allow unrestricted constraining; role models are a well accepted concept in OODMs, so we can concentrate on them. Our solution is based on another accepted concept, namely relationships. Using relationship links instead of references helps to find ill-typed variables efficiently and handle invalid links flexibly. Finally, we have shown that constraining and extension can be safely combined under acceptable restrictions.

## References


[AGO91] Antonio Albano, Giorgio Ghelli, and Renzo Orsini. A relationship mechanism for a strongly typed object-oriented database programming language. In *Proceedings of the 17th International Conference on Very Large Data Bases*, pages 565–575, September 1991.

[AGO95] Antonio Albano, Giorgio Ghelli, and Renzo Orsini. Fibonacci: A programming language for object databases. *The VLDB Journal*, 4(3):403–444, July 1995.

[Bri93] David Bril. LOOM Reference Manual Version 2.0. Technical report, University of Southern California, December 1993.

[Cas95] Giuseppe Castagna. Covariance and contravariance: Conflict without a cause. *ACM Transactions on Programming Languages and Systems*, 17(3):431–447, May 1995.

[CBB+97] R.G.G. Cattell, Douglas Barry, Dirk Bartels, Mark Berler, Jeff Eastman, Sophie Gamerman, David Jordan, Adam Springer, Henry Strickland, and Drew Wade, editors. *The Object Database Standard, ODMG 2.0*. Morgan Kaufmann, San Mateo, CA, 1997.

[Cha93] Craig Chambers. Predicate Classes. In Oscar Nierstrasz, editor, *Proceedings of the ECOOP '93 European Conference on Object-oriented Programming*, number 707 in Lecture Notes in Computer Science, pages 268–296, Kaiserslautern, Germany, July 1993. Springer Verlag.

[DG90] Oscar Diaz and P. M. D. Gray. Semantic-rich User-defined Relationships as a Main Constructor in Object Oriented Databases. In *Conf. on Object-Oriented Databases*, Windermere, July 1990.

[ES92] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, Reading, 1992.

[FJLS96] Burkhard Freitag, Cliff B. Jones, Christian Lengauer, and Hans-Jörg Schek, editors. *Object-Orientation with Parallelism and Persistence*. Kluwer Academic Publishers, 1996. ISBN 0-7923-9770-3.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Reading, MA, 1983.

[GSR96] Georg Gottlob, Michael Schrefl, and Brigitte Röck. Extending object-oriented systems with roles. *ACM Transactions on Information Systems*, 14(3):268–296, July 1996.

[Hog91] John Hogg. Islands: Aliasing Protection in Object-Oriented Languages. In *Proceedings of the OOPSLA '91 Conference on Object-oriented Programming Systems, Languages and Applications*, pages 271–285, November 1991. Published as ACM SIGPLAN Notices, volume 26, number 11.

[HS91a] Andreas Heuer and Peter Sander. Classifying object-oriented query results in a class/type lattice. In *Proceedings of the 3rd Symposium on Mathematical Fundamentals of Database and Knowledge Base Systems, Rostock, MFDBS 91*, volume 495 of *Lecture Notes in Computer Science*, pages 14–28, Berlin, May 1991. Springer-Verlag.

[HS91b] Andreas Heuer and Marc H. Scholl. Principles of object-oriented query languages. In Hans-Jürgen Appelrath, editor, *Proceedings GI-Fachtagung "Datenbanksysteme für Büro, Technik und Wissenschaft", Kaiserslautern*, volume 270 of *Informatik-Fachberichte*, pages 178–197, Berlin, 1991. Springer-Verlag.

[HSJ+94] Torsten Hartmann, Gunter Saake, Ralf Jungclaus, Peter Hartel, and Jan Kusch. Revised Version of the Modelling Language TROLL (Version 2.0). Informatik-Bericht 94–03, Technische Universität Braunschweig, 1994.

[HW97] Paul Harmon and Mark Watson. *Understanding UML: the developer's guide*. Morgan Kaufmann Publishers, Los Altos, CA 94022, USA, November 1997.

[LS93] Christian Laasch and Marc H. Scholl. A functional object database language. In Catriel Beeri, Atsushi Ohori, and Dennis E. Shasha, editors, *Proceedings of the 4th International Workshop on Database Programming Languages*, Workshops in Computing, pages 136–156. Springer Verlag, August 1993.

[MD94] Nelson Mattos and Linda G. DeMichiel. Recent design trade-offs in SQL3. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 23(4):84–89, December 1994.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, New York, N.Y., 1992.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, Englewood Cliffs, NJ 07632, USA, second edition, 1997.

[Mot96] Renate Motschnig-Pitrik. Requirements and Comparison of View Mechanisms for Object-Oriented Databases. *Information Systems*, 21(3):229–252, 1996.

[RBE+91] James E. Rumbaugh, Micheal R. Blaha, F. Eddy, William E. Lorensen, and William J. Premerlani. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ, 1991.

[RS91] Joel Richardson and Peter Schwarz. Aspects: Extending Objects to Support Multiple, Independent Roles. In *ACM SIGMOD Record*, pages 298–307, May 1991.

[Rum87] James E. Rumbaugh. Relations as Semantic Constructs in an Object-Oriented Language. In *Proceedings of the ACM Conference on Object-Oriented Programming: Systems, Languages and Applications (OOPSLA)*, pages 466–481, 1987.

[Sch96a] Jürgen Schlegelmilch. An Advanced Relationship Mechanism for Object-Oriented Databases. Technical Report 19/1996, University of Rostock, Computer Science Dept., 1996.

[Sch96b] Jürgen Schlegelmilch. Conflict Resolution using Derived Classes. In Dilip Patel, Yuan Sun, and Shushma Patel, editors, *Proceedings of the 3rd International Conference on Object-Oriented Information Systems (OOIS'96), London, UK*, pages 267–279, Berlin, December 1996. Springer Verlag.

[Sch96c] Jürgen Schlegelmilch. Role-based Persistence. In Burkhard Freitag, Clifford B. Jones, Christian Lengauer, and Hans-Jörg Schek, editors, *[FJLS96]*, pages 175–195. Kluwer Academic Publishers, 1996.

[The95] Sven Thelemann. Assertion of Consistency Within a Complex Object Database Using a Relationship Construct. In Micheal P. Papazoglou, editor, *Proceedings of the 14th International Conference on Object-Oriented and Entity-Relationship Modeling, Gold Coast, Australia*, volume 1021 of *Lecture Notes in Computer Science*, pages 32–43, Berlin, December 1995. Springer Verlag.

[WCL96] Raymond K. Wong, H. Lewis Chau, and Frederick H. Lochovsky. DOOR: A Dynamic Oject-Oriented Data Model with Roles. Technical Report HKUST-CS96-12, The Hong Kong University of Science and Technology, Department of Computer Science, Clear Water Bay, Kowloon, Hong Kong, 1996.

[WR92] Niklaus Wirth and Martin Reiser. *Programming in Oberon - Steps Beyond Pascal and Modula*. Addison-Wesley, first edition, 1992. source code from the book is available at: ftp://ftp.inf.ethz.ch/pub/software/Oberon/Books/PinOberon/.

[ZM90] Stanley B. Zdonik and David Maier, editors. *Readings in Object-Oriented Database Systems*. Morgan Kaufmann Publishers, San Mateo, CA, 1990.