

Objects don't migrate!

Perspectives on Objects with Roles

Report IAI-TR-96-11
April 1996

Günter Kniessel
Institut für Informatik III
Universität Bonn
Römerstr. 164
D-53117 Bonn
Germany

e-mail: gk@cs.uni-bonn.de
phone: +49-228-73-4503
fax: +49-228-73-4382

Abstract

In this paper we extend previous role concepts for modelling structural and behavioural modification of objects during their life-time. Published approaches allow to add roles to an object but either treat these roles in isolation or have problems defining the interaction of all simultaneously existing roles of the same object.

The first contribution of our paper is the definition of a conceptual model introducing a semantics of "roles" as well as a semantics of "objects with roles". When talking about "objects with roles" we view a conceptual object as the hierarchy of all its simultaneously existing roles and define how these roles interact. Each role provides a different "perspective" of the full object. The behaviour of an object depends on its current roles and on the perspective from which it is regarded. Acquisition of new roles extends the interface of each perspective, but does not alter its behaviour with respect to previously available operations.

The second main result of our work is that roles can be expressed in terms of known object-oriented concepts, without recouring to role-specific extensions. We show how our conceptual model can be mapped to class-based, typed models extended by a single, well-known concept: object-based inheritance. In such models objects never change class but achieve the desired effect by changing their object-level inheritance relationships. Thus "objects don't migrate".

1. Introduction

Objects in real world are subject to ongoing structural and behavioural changes, which need to be adequately reflected in computational models. This is an especially challenging problem for database management systems, whose objects are typically long-lived and thus need repeated adaptation to changes in the real world. Whereas object-oriented databases opened the way for integrated modelling of structural and behavioural aspects of an entity, their strength is static classification, rather than dynamicity. For instance, they can express that a student *is* a person, but they do not capture the notion that a person may *become* a student, an employee, a parent, etc.

During its life-time an object can play many roles. The problem has early been recognised by researchers working on object-oriented data bases and database programming languages and various proposals for modelling changing behaviour have been published¹. *Role playing* has also been described as *object migration*, i.e. varying membership of a database object in a set of classes related by inheritance. Acquiring and abandoning roles is interpreted as acquiring and abandoning class membership while retaining one unique identity for an object.

However, it has been argued by [WRS91/94] that generally retaining the identity of an object during "migration" raises a *classification paradoxon*, which can best be explained by considering the problem of *counting* objects. In the object migration view, a person that travels each day by train would be represented by an instance of PERSON that repeatedly acquires and loses membership in the PASSENGER subclass. Since it always retains its object identifier the object would count as only one passenger. However, if we want to determine the number of tickets sold, the most frequented connections, etc. we need to count ten passengers where we counted one person. Then we have to give PASSENGER instances their own identifiers, different from each other and from PERSON identifiers. In this view, we may have different instances of different classes coincide in time and space, e.g. ten different PASSENGER instances "coincide" with the same PERSON instance.

In [Scio89], [WJS94], [WJS94] models were proposed in which each role of an object is itself an object with an own identity. A role object shares the properties of the object that plays this role (*the player*) by forwarding to the player all messages that it cannot answer itself. However, different sibling roles are not aware of each other and the player object is not aware of any of its roles. This raises an *identification paradoxon* since now accessing all properties of a conceptual object having many roles requires *a set* of object identifiers, contradicting the widely accepted definition of identifier ([KhCo86]), which requires that one unique identifier allows access to the whole object.

Even role models that retain one unique identity for an object do not provide a satisfactory solution. They either allow to access an object as a whole, but do not define the semantics of dynamic binding when different "most specialised" roles could potentially answer a message (e.g. [FBC+87]), or restrict access to an object to be always through one of its roles (e.g. [ScSw89], [RiSc91], [ABGO93], [LiDo94]). In the latter approaches, only a "slice" of the whole conceptual object can be viewed from the perspective of a role. In principle, they face the same problem as approaches that explicitly allow different object identifiers: there is no unique

¹ We shall comment in detail on related work in section 6.

identifier, that can be used to access the *whole* object. Instead, one needs to "navigate" through different roles of an object using a *set of implicit* identifiers, $\langle \text{objectId}, \text{role} \rangle$.

The main contribution of our paper with respect to role modelling is to show that these apparent contradictions can be reconciled. We present a model in which a conceptual object is regarded as the set of *all* its simultaneously existing roles. Each role provides a different "perspective" of the conceptual object. All perspectives provide access to the *full* conceptual object and thus share the same interface, but with possibly different behaviour. Acquisition of new roles extends the interface of *each* perspective, but does not alter its behaviour with respect to previously available operations. Besides of being perspective-dependent, object behaviour may also be context-dependent, in the sense that the static type of a variable is interpreted as a default perspective of that variable on the objects that it references. Classes can be related by role hierarchies and inheritance hierarchies, which can be orthogonally combined, making role playing inheritable and instances of role classes substitutable by instances of their subclasses.

Moreover, we show how our conceptual model can be mapped to a class-based, typed model essentially extended by a single concept: object-based inheritance. In this model, roles are represented by distinct objects that appear as a whole by mutually "inheriting" from each other. Objects never change class but extend their interface by changing their object-level inheritance relationships. Thus "objects don't migrate".

The paper is structured as follows. In the next section we illustrate the intended functionality of our role model on an example. We do not formalise the role model directly but define two variants of "object-based inheritance" (section 3), present their integration into a typed, class-based object-oriented model (section 4), and show how this model can express the intended role model (section 5). In section 6 we give a detailed account of related work, classifying previous approaches and showing how they can be expressed in our base object model.

2. Yet Another Role Model!?

In this section we shall informally describe our role model, illustrating its functionality on an example. In our view an entity of the real world can be described by one *essential role*, which is present as long as the entity exists, and a (possibly empty) set of *transient roles*¹, which it can acquire or abandon dynamically. An entity can play different roles simultaneously or in sequence. The most familiar essential role is probably "being a person". Some particular person, Sally, might become a student, cease to be a student, then become an employee, and later on a parent. Sometimes, she may become ill or unemployed for some period.

Whenever an entity interacts with other entities, one of its roles dominates the others, determining the perceivable behaviour at that moment. We equivalently say that an entity is always perceived from the *perspective* of one of its roles. Each perspective presents the *full* entity, including all the properties defined for any of its current roles. The dominant role is implicitly determined by the *context* or explicitly by *requests* from outside. For instance, when Sally goes to the doctor she will automatically behave in her role as patient. Even in her patient role, Sally will still be able to answer questions about her employer or her children. When her child comes to her, asking for a candy, she will temporarily switch to her role as parent, react accordingly, and then come back to her default role in the current context.

¹ Note that, although they are called *transient*, these roles "persistent" in the sense that they may be stored in the database and accessed like any other "persistent" object.

Roles can be unrelated to each other, or refinements of each other. More precisely, roles can be organised in a partial order having the essential role as greatest element. We call a role that is smaller (bigger) with respect to the partial order a *subrole* (*superrole*). Incomparable roles are called *sibling roles*. The semantics of the partial order is that every role, r , is a specialisation of its superroles in the sense that, when an entity is regarded from the perspective of r , the behaviour of r overrides the behaviour of any superrole. On the other hand, r can use methods of sibling roles or subroles, but cannot modify their semantics. For instance, when Sally becomes engaged in a conversation about education while in the doctor's waiting room (i.e. in her patient role), she will take the *same* opinion as in her role as parent.

Sometimes external requests addressed to an entity might not be specific enough to be answered at all, or to be answered as desired. In our opinion the requesting entity might prefer a general answer over no answer, and no answer (resp. being forced to state the request more precisely) over a useless one. E.g., when the doctor's receptionist asks a patient for his phone number she will in general get the private phone number of the person and might have to ask explicitly for the office phone number. However, she will have little interest in the phone number of the French restaurant that the patient has recently discovered in his gourmet role.

Graphical notation. To ease the extension of a database schema, each type of role should be independently specifiable as a class. Classes should be related by semantic relationships indicating whether a class specifies a dynamic specialisation (role) or a static specialisation (partition) of another class. According to these principles, we would like to represent Sally's situation as depicted in fig. 1.

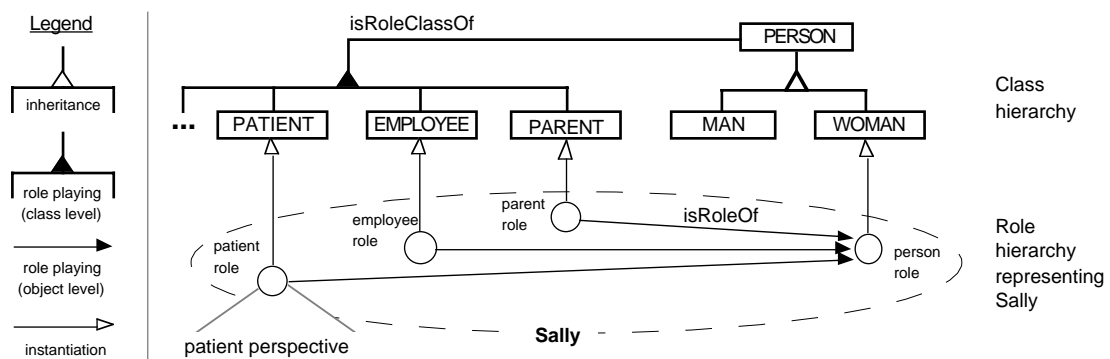


Fig. 1: Conceptual representation of combined inheritance and role hierarchy

Note how the class hierarchy statically specifies potential roles (PATIENT, EMPLOYEE, PARENT) of an essential concept (PERSON) by the *isRoleClassOf* relationship and at the same time specifies static partitions (MAN, WOMAN) of the same concept by inheritance. Since in our approach objects do not migrate along inheritance hierarchies, a person can *be* either a man or a woman, but can *become* a patient, employee, parent, etc. We can extend this design by adding new role classes or subclasses to any class, taking advantage of the orthogonal combination of inheritance and role playing: role playing is inheritable and instances of any subclass of a role class may be used in that role (e.g. a WOMAN instance in the PERSON role).

Why no dedicated role model? We could now proceed by formalising the sketched functionality as a dedicated role object model. However, we will follow another way, defining a more general model first and showing then how it can express the intended functionality. Stepping back a bit, this choice follows from our aims. In order to solve the classification paradoxon described in the introduction, we have to represent each role of a conceptual entity by a distinct object, as indicated in fig. 1. However, to make each of the role objects appear as the same con-

ceptual object, all role objects must "inherit" in some way from each other. This cannot be the standard, class based inheritance, since it would replicate attributes and would apply to all instances of a class, whereas role playing is an object-specific property. Thus, we need some form of "inheritance" between objects. However, we do not want to give up the benefits of classes, class-based inheritance and typing. Thus, we obviously need a model that integrates classes, types, and class-based as well as object-based inheritance. We will define such a model in the next two sections.

3. Delegation and Consultance

In this section we shall define two variants of object-based inheritance, *delegation* ([Lieb86], [UnSm87], [SmUn95], [Scio89], [Cham93]) and *consultance* ([KRC91]), used in class-free, object-based systems. In both cases an object, called the *child*, may have modifiable references to other objects, called its *parents*. Messages that cannot be executed by the message receiver are automatically forwarded to its parents. When a suitable method is found, it is executed, after binding its *self* parameter to the (initial) *message receiver*. This automatic forwarding with binding of *self* to the receiver is called *delegation* (cf. fig. 2). In contrast, automatic forwarding with binding of *self* to the object in which the method was found (the *method holder*) is called *consultance*¹.

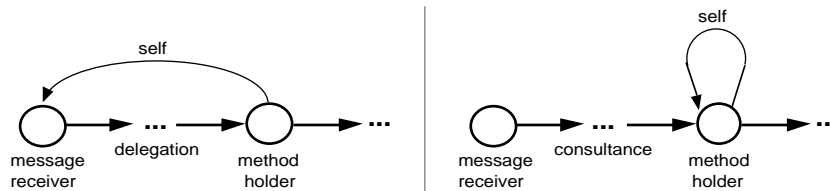


Fig. 2. Different effect of delegation and consultance on *self*

Intuitively, delegating a message means asking another object to do something *on behalf of the message receiver*, i.e. as the message receiver would do it, whereas consultance means asking another object to do something as *it* knows how.

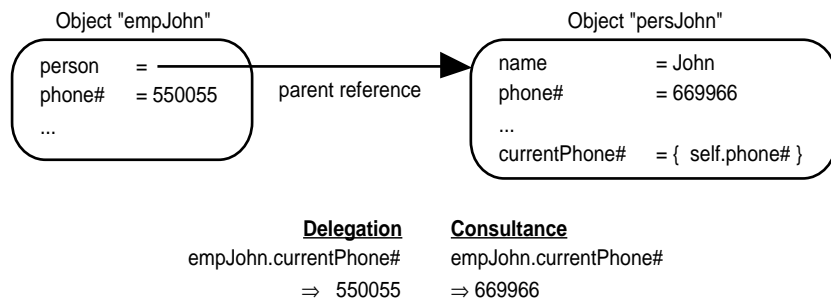


Fig. 3: Different result of the same message for consultance and delegation

Let us consider the example illustrated in fig. 3, where the object *empJohn* references its parent object *persJohn* in the attribute *person*. The message *empJohn.currentPhone#* cannot be answered by *empJohn*, since it contains no method for *currentPhone#*. Therefore the message is

¹ To avoid confusion, note that the term *delegation* is often used in a non-standard way in literature, denoting *consultance* ([LTP86], [GSR94]) or variants of consultance ([Stro87], [WJS94], [ABGO93]). Also the term *object-based inheritance* is often used in literature in a more restricted sense, mostly denoting *static* forwarding relationships between objects, e.g. static delegation in [Nier89] and static consultance in [HaNg87].

forwarded to *persJohn*. When the method for *currentPhone#* is found in *persJohn*, its *self* will be bound to the message receiver, *empJohn*, if delegation is used, and to the method holder, *persJohn*, if consultance is used. Thus, the subsequent message *self.phone#* will return John's office phone number in the first case, and his private phone number in the second case.

Note that in the case of consultance, the *phone#* method¹ of *empJohn* "overrides" the one of *persJohn* for the message *empJohn.phone#*, but not for the message *empJohn.currentPhone#*. With delegation we would always get the employee phone number. For delegation, overriding applies to explicit receiver messages, as well as to messages to *self*. For consultance it only has effect for explicit receiver messages. To stress this distinction we talk about *replacement* in connection with consultance and reserve *overriding* for delegation (and class-based inheritance).

Delegation, consultance, and roles. In section 2. we mentioned that subroles should appear as specialisations of their superroles. Thus they must "inherit" from their superroles such that their methods override methods of the superroles for all messages, including messages to *self*. This is exactly what has been introduced as *delegation*.

On the other hand, a role should "inherit" from subroles and sibling roles but must not specialise the "inherited" behaviour, i.e. methods of the role may replace but not override methods of sub- and sibling roles. This is exactly the effect of *consultance*.

Summarising, it appears that a natural way to achieve the functionality sketched in section 2 is to represent objects with roles by object hierarchies in which each object representing a role delegates to a superrole object and consults subrole objects. The interaction of sibling roles is indirect, via their common superrole. Note that, with respect to *self*, the effect of delegating a message to an object, *super*, which itself consults another object, *sibling*, is the same as directly delegating to *sibling*.

4. Darwin²: Combining Class-based and Object-based Inheritance

Having defined our notion of object-based inheritance we now proceed to show how it can be integrated in a typed, class-based environment. The following discussion is restricted³ to the aspects that are essential for understanding the representation of roles and the composition of independently developed role hierarchies described in the next two sections. We assume that the reader is familiar with the notions of class, instance, and class-based inheritance ([ABW+90], [Wegn89]).

Classes and Objects. Each object is a direct instance of one most specific class and an implicit instance of all its superclasses. Classes specify the *interface* of their instances, their *attributes* (variables), and the implementation of *methods* (operations) specified in the interface. All attributes, local variables, parameters and method bodies have a *statically declared type*.

¹ In most object-based ("prototype-based") systems there is no distinction between attributes and methods. Each attribute name can be used as a method to read its value.

² We regard our model as an evolution from two fundamentally different families of object-oriented systems to a new "species", that will also have its share in the "struggle for survival". We have therefore named it in honour of Sir Charles Darwin, who founded the theory of the evolution of species.

³ More details can be found in [Knie94, 96].

Inheritance and Delegation¹. Classes may (multiply) inherit from superclasses. In addition, objects may delegate to other objects referenced by their *delegation attributes*. Delegation attributes have to be declared in an object's class. If class C declares a delegation attribute of type T we say that C *delegates to* T and that

- C is a *declared child class* of T and of each of T 's subclasses, and
- T is a *declared parent class* of C and of each of C 's subclasses.

The *parent classes* of a class are the union of its declared parent classes and their subclasses. Please note that "class C delegates to class T " is just a shorthand for saying "class C defines that its instances delegate to instances of T ". E.g. in fig. 4, class C delegates to class D and therefore a C instance may delegate to a D_1 instance. Note how delegations allows sharing of the same parent object by different child objects: both B instances delegate to the same D_2 instance and (indirectly) consult the same E instance.

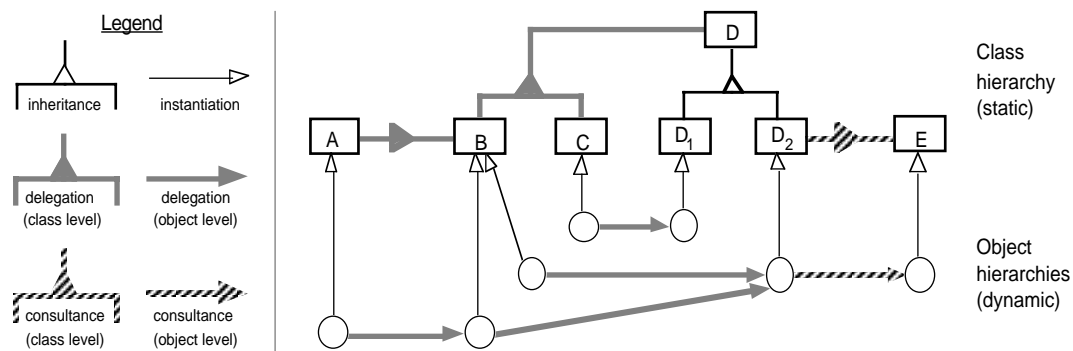


Fig. 4: Graphical notation and example of class hierarchy and corresponding object hierarchies

There are two essential differences between classes related by inheritance and delegation:

- delegation extends the interface, but not the structure (i.e. the set of attributes) defined by the child class and
- each direct instance of a child class is distinct from each direct instance of a parent class, e.g. in fig. 4, no instance of C is an instance of D (but every instance of D_1 is an instance of D).

Types. A *type* specifies an *interface*, i.e. a set of method signatures. In purely inheritance-based models, the type of an instance corresponds to the signature of the methods defined by its class (and its superclasses). Delegation has the effect of extending this interface by the interfaces defined for the *declared parent classes*. In the following, we shall use the term *type* in this extended sense. We require the type of a declared child class to be a *subtype* of each of its declared parent classes, with respect to the same subtyping condition that must hold between subclasses and superclasses. We depart from the standard condition that parameter types must be contravariant and result types covariant with respect to the corresponding method signature of a supertype ([CaWe85], [CCHO89]). Instead, we require non-variance, i.e. identical signatures for all methods with the same name.

¹ For ease of presentation, we shall only talk about delegation in the remainder of this section. However, everything said applies equally to consultancy, unless stated otherwise. All definitions explicitly made for delegation have their consultancy counterparts (e.g. consultancy attribute).

Mandatory and Optional Attributes. An attribute is called *mandatory* if it must always have a non-nil value, *optional* otherwise. This distinction is especially important for delegation attributes. The above extension of the type notion is statically safe only if delegation attributes are mandatory. Allowing a delegation attribute to be optional requires run-time checks for all messages that need to be delegated via this attribute (i.e. that are not defined locally). Note that this is a different kind of check than the usual run-time checks in purely inheritance-based systems: it tests that an object exists, *not* that it belongs to a certain class). Both kinds of check may be used in our model, due to the coexistence of subclassing and optional delegation attributes, which allow to statically capture the *potential* structure of dynamically evolving object hierarchies. Role modelling is a good example hereof (cf. section 5).

Cyclic Delegation. Delegation hierarchies may contain cycles, provided that at least one arc in the cycle corresponds to an optional attribute. Cyclic delegation is the reason why we use a non-variant subtyping rule. However, since contravariant parameter redefinition is seldom exploited in practice, we think that giving it up is a reasonable price for enabling cyclic delegation, which is the basic ingredient for a more powerful category of role models (cf. 6, 7.).

Static and Dynamic Delegation. Since a delegation attribute can reference any value that conforms to its declared type, assignment to a delegation attribute can be used to change the behaviour of an object at run-time by changing its parent object(s). If desired, we can restrict delegation to be static, by allowing assignments to selected attributes only in special constructor and destructor methods, i.e. at the beginning and end of an object's "life". Both variants are necessary for capturing role semantics. Dynamic delegation is required for modelling that an object may acquire and abandon roles repeatedly. Static delegation is required for modelling that a role of one object cannot become a role of another object.

Renaming. If different superclasses of a class C contain conflicting methods these methods are automatically renamed in the context of C by appending the suffix $as<Class>$ to the *original* method name. The $<Class>$ part of the suffix is the *direct* superclass from which the method is inherited. Renamed methods can be redefined in C . Then they override the corresponding original definitions. The same mechanism is applied to solve conflicts between methods in different declared parent classes. In that case the $<Class>$ part of the suffix is the *declared* parent class to which the method is delegated.

Note that no renaming is required for a method that is defined in a *superclass* and in a *parent* class. There is no conflict among such definitions since methods defined in an object's class (and its superclasses) are considered part of that object and thus *override* the corresponding methods from parent objects. A more precise definition of overriding is presented and discussed in the next section.

Overriding and Delegation¹. Let c be an instance of a class C , delegating to p , an instance of class P , and let DP be the nearest superclass of P that is a *declared parent class* of C , as illustrated in fig. 5. Then the methods of C and of C 's superclasses override the methods of P contained in the interface of DP . The additional methods introduced in P are only replaced (cf. 3.), but not overridden by methods from C .

¹ Note that this is the only aspect of our discussion that is *not* applicable to *consultance*. By definition, consultance does not allow overriding but just replacement (cf 3).

This rule is motivated by the desire to be able to integrate independently developed subclasses and child classes of the same class into one hierarchy, without fear of undesired side-effects. E.g. in fig. 5a, different definitions of method x were *independently introduced* in two classes, C and P , that may have been developed and compiled independently, knowing only DP but not each other. Therefore it is very unlikely that the different definitions of x have the same semantics and overriding would silently change the semantics of x relied upon by methods from P .

Consider for instance the message $c.b$. After unsuccessfully searching C , the message will be delegated to p , the method found in P will be executed, the message $self.x$ will be sent, and the search will start again in class C (because $self = c$). If methods from C were allowed to override *all* methods from P the local definition of x would replace the one from P . This could produce very obscure and hard to locate run-time errors during the execution of methods from P .

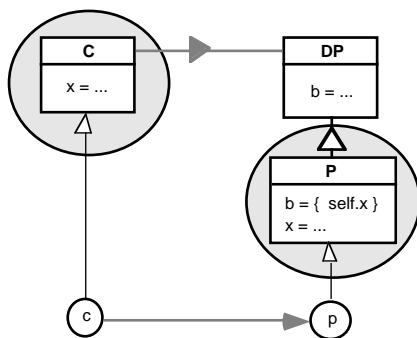


Fig. 5a: Independently introduced methods:
no overriding, just replacement (cf. 3)

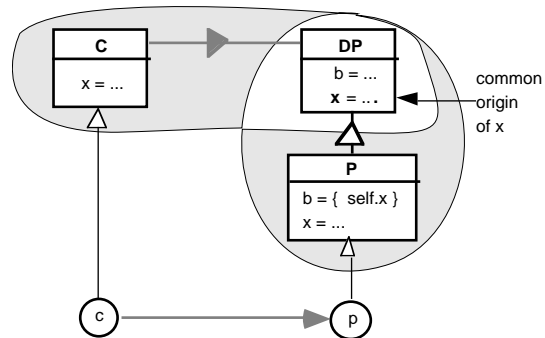


Fig. 5b) Methods with common origin:
overriding enabled

In contrast, in fig. 5b, method x was first introduced in class DP , and then *redefined* in C and P . Both redefinitions of x most likely have a compatible semantics, since they both modify the same original definition. Therefore overriding is allowed and necessary, to implement c as a specialisation of p .

Change-aware compilation. The overriding rule used in Darwin could also be stated as: "The methods of c override the methods of p that have the same name and a common *origin*." In example 5b the common origin of method x is the class DP .

Now suppose that fig. 5a illustrates our initial database schema whereas 5b is a latter one. After the addition of method x to class DP the independently introduced definitions of x in class C and P will *appear* to have a common origin in DP . Thus the message $c.b$ will silently change its semantics, since the definition for x from A will be executed instead of the one from P . Note that this is a more general instance of a problem that already occurs in purely inheritance-based systems, where a method added to a superclass (e.g. in a new release of a library) would silently be overridden by a previously existing, identically named method in a self-written subclass.

The problem arises if the compiler determines the origin of a method only by looking at the current state of the program. Therefore we use *change-aware compilation*: for every compiled source class, the compiler records the origin of every method of that class. It always compares the currently computed origins with the stored information from the previous compilation. If the origin appears to have moved "up" in the class hierarchy, the compiler conservatively assumes that the added definition has a different semantics from previously existing ones, since it is not guaranteed that the programmer of the changed class was aware of the existence of the same method in subclasses / child classes.

Such cases are treated by automatically renaming the added definition in the context of each subclass / child class that already possessed a definition of the method.. In our example, the definition of x added to class DP would be renamed in the context of class P and C , thus maintaining the initial semantics of the program.

Renaming versus Overriding. Obviously, renaming is not just a means to resolve multiple inheritance / delegation conflicts, but can also be used to statically enforce or avoid overriding. In essence, renaming provides static, fine-grained control over which method definitions are considered semantically compatible.

As shown in table 1, most of the problems raised by the integration of independently developed classes can be treated by a combination of automatic and manual renaming. Automatic renaming implements conservative default strategies, intended to avoid undesired interactions of independently developed / changed parts of a program. It is reported to the schema designer, who may decide to alter the generated renaming declarations if they do not match the intended semantics of the class. Thus, automatic renaming eases the joint use of different independently developed (super- / parent) classes by reducing the amount of "integration work" that has to be performed manually. However, renaming is a program transformation and can therefore express only decisions that can be made statically, by inspecting a program's code. It is therefore complemented by our special overriding rule, which decides dynamically¹ whether overriding may take place.

Problem	Treatment	
	default (automatic)	tuning (manual)
multiple inheritance conflict	renaming	change of renaming statements
multiple delegation conflict	renaming	change of renaming statements
avoid overriding between		
• class and super- /declared parent class	--- (overriding allowed)	insertion of renaming statement
• class and super- /declared parent class (after schema change)	renaming (no overriding)	deletion of renaming statement
• class and <i>potential</i> parent class	run-time check (no overriding)	---

Table 1: Treatment of multiplicity conflicts

Summarising, the main extension of the Darwin model with respect to purely inheritance-based, strongly typed models, is the introduction of a static, class-level declaration of dynamically modifiable "*inheritance*" relations between *individual instances*. Object-level inheritance can be either delegation ([Lieb86]) or consultance ([KRC91]). Both relations can be either mandatory or optional, allowing to statically specify fixed as well as potential structures of object hierarchies created at run-time. The existence of different kinds of class hierarchies (inheritance, declared delegation, declared consultance) enables independent modelling of different aspects of a concept. Renaming, change-aware compilation, and a special overriding rule allow automatic, semantics-preserving integration of independently developed classes.

¹ In [Knie94] it is shown that implementation of the overriding rule boils down to an integer comparison at run-time.

5. Back to Roles

In this section we show how easy it is to define the semantics of the role model sketched in section 2, using the features of the Darwin model. Mapping of the common features, classes and inheritance, is one-to-one, i.e.

- a) Every class in a role schema is part of the corresponding Darwin schema.
- b) If S isSubclassOf C in a role schema, then S isSubclassOf C in the corresponding Darwin schema.

We only have to define the Darwin counterparts for the class-level *isRoleClassOf* relation and the object-level *isRoleOf* relation of the role model:

- c) If R isRoleClassOf P in a role schema, then in the corresponding Darwin schema
 - R delegates to P via a mandatory, static delegation attribute, "playerP", added to R ,
 - P consults R via an optional, dynamic delegation attribute, "roleR", added to P .

If r and p are instances of the classes R and P , respectively, the above definition implies that the r isRoleOf p relation at the object level is represented by the corresponding *playerP* attribute in r and the *roleR* attribute in p . Fig. 6 shows the result of mapping the role schema illustrated in fig. 1, to a Darwin schema.

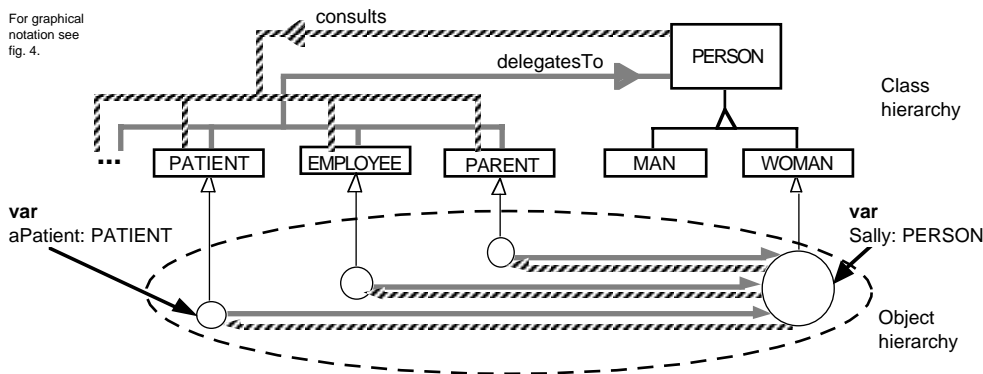


Fig. 6: Darwin representation of the role schema from fig. 1 ("Sally's example").

The above mapping precisely defines the semantics of the notions introduced in section 2. One conceptual entity is represented as a virtual object made up by a hierarchy of mutually inheriting physical objects, each representing a different role. The identifier of each of the physical objects acts as the perspective of the corresponding role on the virtual object. The identifier of the conceptual object is the identifier of the object representing the essential role. In fig. 6 for instance, the identifier of the PATIENT object acts as the PATIENT perspective on the conceptual object "Sally". The identifier of "Sally" is the identifier of its WOMAN object. The standard equality of object identifiers tests for identical roles. The correspondence of different object identifiers to the same conceptual entity is checked by the message,

obj.conceptuallyIdenticalTo(otherObj),

which tests the identity of the essential roles of *obj* and *otherObj*. Thus our model supports both notions of counting described in the introduction, solving the discussed classification and identification paradoxa.

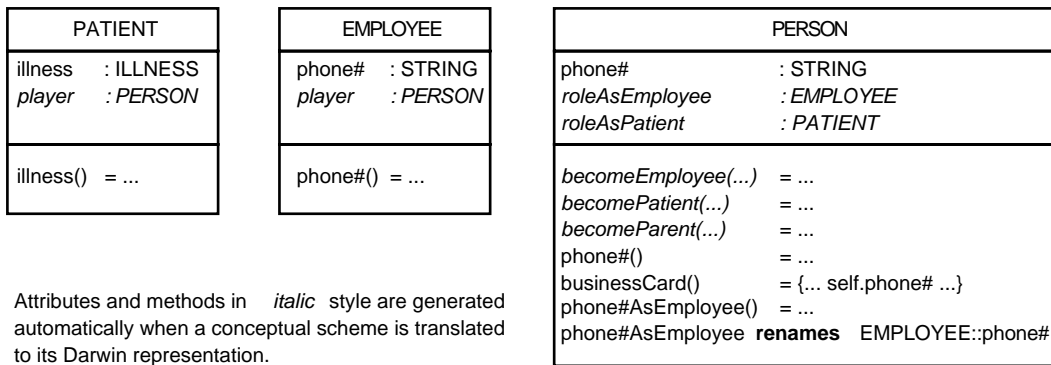


Fig. 7: Details of class definitions for classes from fig. 6.

Darwin's typing rules imply that a perspective presents a *partial* view of a conceptual object, if only messages that are statically safe for the corresponding role are allowed, and a *full* view of the object, if run-time checking for existence of roles (non-nil optional "role" attributes) is allowed. Supporting the second position is easy, since provision of run-time checking and / or exception handling mechanisms is needed anyway for correct treatment of role dropping ([FBC+87], [Zdon90], [ABGO93]). Furthermore, use of implicitly defined *combinatorial types* ([Knie96]) can significantly reduce the amount of run-time type-checking. Note that the statically safe interpretation coincides with the restricted interpretation of perspectives in most published role models, while the second interpretation is an extension over all known approaches (cf. section 7).

For each of the messages visible from the *perspective* of a given role, its behaviour is determined by the definition of consultance, delegation, overriding, and replacement. For instance, the class hierarchy from fig. 6 and the class definitions illustrated in fig 7, produce the behaviour described in the comments of the following application program fragment:

```

var Sally : PERSON;           // "Sally" has static Type "PERSON"
var aPatient : PATIENT;      // "aPatient" has static Type "PATIENT"
...
Sally.becomeParent(...);    // "becomeRole()" is an automatically generated method of each player class.
Sally.becomePatient(...);   // If the receiver object does not yet have the role Role, it adds a corresponding
                             // parent object and initializes it with the provided parameter values.
aPatient := Sally;          // Run-time check. A reference to the PATIENT role of Sally is assigned to the
                             // variable "aPatient". From now on, Sally is implicitly viewed in her PATIENT role.
                             // The situation after this assignment is illustrated in fig. 6.
aPatient.illness();         // Statically safe. Solved locally in PATIENT role.
aPatient.phone#();         // Statically safe. Delegated to PERSON role.
aPatient.phone#AsEmployee;  // Statically safe. Delegated to PERSON role.
                             // Calls renamed method that has been redefined for PERSON role.
aPatient.AsEmployee.businessCard(); // Run-time check. Now, aPatient is explicitly viewed in his EMPLOYEE role.
                             // Delegated from EMPLOYEE to PERSON role. Returns the employee's phone#!

```

Note that the fact that a person might have two office phone numbers, an official one for business issues, and another one for private issues, is modelled by renaming and redefining in the person role the `phone#` method of the employee role. Thus the person role and all the non-employee roles delegating to it will use the "private" office phone number when calling `phone#AsEmployee`. If the "official" office phone-number is desired, then the employee

view can be explicitly requested, using the automatically generated *role casting method* ([ABGO93]) `asEmployee`. Note however, that a message like

```
aPatient.AsEmployee.phone#
```

is *not* statically bound to the method defined in the `EMPLOYEE` class. If the object that represents the employee role is an instance of a subclass of `EMPLOYEE`, dynamic binding will select its specific method.

We close this section without discussing role acquisition, role dropping, role inspection, and role casting. In all these aspects we share the view of [ABGO93] and provide corresponding operations, whose implementation is straightforward in the presented model.

6. Related Work

Roles had already been proposed as extensions of the network data model ([BaDa77]) and have since then received much attention in the context of the relational data model and semantic data models as a means for modelling structural aspects and relationships among entities (e.g. [BBMP95]). The advent of object-oriented data models, focusing on integration of data and operations, allowed to additionally capture the intimate relation between role changes and behavioural changes. There are two main aspects in this field: definition of the *message passing semantics* of objects with roles in a given state of the world, and definition of legal *migration patterns* ([Su 91]), i.e. sequences of legal role changes. Our paper focuses on the first aspect, providing a reference model, Darwin, in which existing approaches can be expressed, compared, and extended. We give an overview of the variations of message passing semantics found in literature and classify known approaches (7.1), before discussing related work in detail (7.2). The relation of Darwin to migration patterns is discussed in section 7.3.

6.1. Message Passing Semantics: A Classification

Roles of objects and objects with roles. Regarding message passing semantics, published papers reflect two contrasting attitudes, which we call "*roles of objects*" and "*objects with roles*". In the former ones, objects may acquire new roles but the behaviour that can be perceived through previously *existing* references to the extended object does *not* change. These approaches represent distinct, independent, *external perspectives on* an object, also called *aspects* ([RiSc91]) or *views* ([ScSw89]). In contrast, in the "*objects with roles*" approaches, the behaviour perceivable through existing object references changes when roles change, reflecting the interpretation of roles as *integral parts of* an object. The perceivable change may be either a modification of the behaviour of already available operations (specialised behaviour) or a modification of the set of available operations (new behaviour), or both. Table 2 summarises the possible alternatives.

Interfaces and dynamic binding. The message passing semantics supported by a particular role model is determined by its handling of interfaces and dynamic binding. In all typed approaches published to date *interfaces are fixed*. Some untyped approaches ([Scio89], [GSR94]) allow the interface of a role to be extended by the interface of its current *player*, but not by the interface of sibling or descendant roles. Since a role of one player cannot sensibly become a role of another player¹ we also count these approaches as supporting only fixed interfaces. The

¹ However, in [Scio89] there is no way to prevent arbitrary changes of player objects.

work of [Perni90] is the only one that appears to support variable interfaces. However, there is no notion of overriding in that model and dynamic binding is replaced by parallel execution of all selectable methods from all available roles. To the best of our knowledge, our approach is the first one that allows variable interfaces in a typed language without restricting dynamic binding.

		Interface	
		visible through an object reference	
		fixed	variable
Dynamic binding with highest priority for ...	receiver role	A) fixed set of operations with fixed behaviour	B2) extensible set of operations with fixed behaviour
	most specialized subrole of receiver role	B1) fixed set of operations with specializable behaviour	B3) extensible set of operations with specializable behaviour

Table 2: Variants of "objects with roles" (A) and "roles of objects" (B1 to B3) resulting from different ways of dynamic binding and interface handling.

With respect to *dynamic binding* published papers diverge on the question whether methods in subroles override methods in the message receiving role. One class of approaches give highest priority to the message receiving role, another one to its most specialised subrole. The first alternative guarantees fixed behaviour of an object seen in a certain role, whereas the second allows to express that the existing behaviour in a given role may be specialised when subroles are acquired. Each of the existing approaches only support one of these semantics. In contrast, both semantics can be expressed in Darwin. An example of B2 semantics (cf. table 2) and of its mapping to Darwin is our reference role model (chapter 2 and 6). Modelling of B1 semantics (specialisable behaviour) in Darwin is shown in the discussion of Fibonacci (cf. 7.2). B3 semantics is a combination of B1 and B2 and can be expressed accordingly. Therefore our object model allows to define variants of the *isRoleClassOf* and *isRoleOf* relations with different dynamic binding semantics and to let the schema designer choose the appropriate relation for each pair of role classes in his application.

Perspectives. The "roles of objects" approaches are motivated by the desire to represent different *independent* perspectives on one object. There are two prerequisites for modelling independent perspectives: the existence of *simultaneous sibling roles* and of selective *references to individual roles*.

Simultaneous sibling roles can be expressed either by one object that is an instance of different most specific classes or by different objects that are each instances of one most specific class. In the latter case each object represents one role and the corresponding *object identifier (oid)* can be used as a reference to that role. If, on the other hand, objects are multiple instances, there is only one oid for the whole object, i.e. all its roles. Therefore, when using object identifiers as object references there is no way to selectively refer to different roles. The first approach that postulated one unique identity for all the roles of an object but, nevertheless, allowed references to specific roles ("aspects") was described in [RiSc91]. The Melampus data model replaced the traditional view that references are object identifiers by a more fine-grained notion that allows references to specific "chunks" of structure and behaviour *within* an object. The interaction of the different notions of objects and references is summarised in table 3. The field labelled "---" is a theoretically possible but useless combination: when the different roles of a conceptual entity are represented by separately referenceable, different objects there is no use for an additional "subatomic" level of reference refinement.

		Simultaneous sibling roles by	
		Multiple instantiation, single object (object ≈ set of roles)	Single instantiation, multiple objects (object ≈ role)
References	oid	fixed, unique perspective	multiple perspectives
	oid + role	multiple perspectives	---

Table 3: Relationship between "perspectives", object references, and different ways to achieve simultaneous sibling roles

Perspectives and contexts. The static type of an object reference may be seen as determining an implicit, default perspective on an object in a certain context. In different contexts the same object reference may be interpreted differently, leading to context-specific behaviour of an object. This holds for all kinds of systems, whether they allow explicit multiple perspectives or not. For instance, context-dependent behaviour is achieved by the "preferred class dispatching rule" of [BeGu95], in a system based on objects that are multiple instances and are referenced by one unique oid. However, systems that allow references to different roles, may explicitly switch the perspective. Thus they offer the programmer more control on the behaviour of objects, allowing to explicitly request a specific behaviour instead of the default one in a given context. Corresponding predefined operation are, e.g. *role casting* in [ABGO93] and *role switching* in [GSR94].

		Simultaneous sibling roles					
		no		yes			
Multiple Perspectives	no	A/a	B2/a	A/b	B2/b	essential role (of) receiver object	Dynamic binding with highest priority for ...
		B1/a	B3/a	B1/b	B3/b		
	yes	d)		A/c	B2/c	receiver role	
				B1/c	B3/c	most specialized subrole of receiver role	
		fixed	variable	fixed	variable		
		Interface visible through an object reference					

Proposals in "interesting" categories

A/c) [Scio89], [ScSw89], [Zdon90]¹, [RiSc91], [GSR94], [WJS94]².

B1/a) [ACO85], [StZd89], [Zdon90]¹, [WJS94]²

B1/b) [FDC+87], [ABG091], [BeGu95].

B1/c) [VeCa93], [ABGO93], Darwin

B2/a,b,c).No approaches known yet.

B3/a,b,c).No approaches known yet.

Each semantics can be expressed in Darwin.
Examples are shown in section 6 (B2/c semantics) and section 7.2 (A/c and B1/c semantics)

Table 4: Overview of related work, classified with respect to message passing semantics.

The categories A, B1, B2, B3 introduced in table 2 are refined by the categories a, b, c, d, resulting from the classification with respect to availability of simultaneous sibling roles and multiple perspectives.

¹ [Zdon90] discusses "global type changes" with category B1/a semantics and "local type changes" with category A/c semantics.

² [WJS94] discusses "static partitions" with category A/a semantics, "dynamic partitions" with category B1/a semantics and "roles" with category A/c semantics.

Classification. Let us now classify our own and previous work according to the semantic criteria introduced above. Table 4 gives an overview of all approaches that described their message passing semantics in enough detail in order to determine their membership in exactly one category. Not all semantic categories shown in the table are possible *and* meaningful. Category *A/a*, for instance, is not interesting for the purpose of our discussion, since it corresponds to the "traditional" definitions of object-oriented programming, which have no notion of "roles", "aspects", "perspectives", or "views". Category *A/b* is possible but not meaningful: if one can only reference the object but not its roles, the object cannot specialise its behaviour, and does not extend its interface, addition of roles has no noticeable effect. Category *d* is impossible because multiple perspectives depend on existence of *simultaneous* sibling roles.

I'm grateful for any comments on the proposed classification
and for suggestions on how to classify other approaches
(e.g. [Pern90], [Papa91], [MaOd92], [NRE92], [NgRi92], [Cham93], [LiDo94])

6.2. Message Passing Semantics: A closer look

In this section we shall discuss in more depth the work referenced in the above overview table. Of the ten "interesting" semantic categories, those allowing to view an object from multiple perspectives are obviously more expressive than those that support only one fixed perspective. Therefore (and for lack of space) we shall focus our discussion in the following on approaches in the four c) categories, which support multiple perspectives and also cover the distinction between "roles of objects" (*A/c*) and the different interpretations of "objects with roles" (*B1/c*, *B2/c*, and *B3/c*). A notable exception, which will also be discussed, is [BeGu90], which achieves context-dependent behaviour without using an explicit notion of perspectives.

Roles of objects – Category *A/c*

The "roles of objects" approaches focus on the semantics of *roles*, viewing each role (and its superroles) separately from each other (sub- and sibling) role simultaneously played by an object. These approaches are motivated by the desire to represent different independent aspects / views of one object. All references are to a specific role. The only way for a programmer to "see" all properties of a conceptual entity at a given time is to explicitly manage a set of references to different roles of the same object.

Single object, multiple interfaces, no inheritance

Some "roles of objects" approaches represent one conceptual entity by *one* object that maintains its identity when changing roles. In these approaches references to roles are different from object identifiers.

Views. [ShSw89] allow one object to have multiple, independent interfaces and to be alternatively regarded through one of them, providing different behavioural perspectives ("*views*") of an object. Although it preserves the identity of the object, each view allows to access only a part the object. There is no notion of inheritance or subtyping between interfaces, and no dynamic binding. There is also no provision for acquiring new interfaces.

Aspects. Dynamic acquisition of new interfaces is provided by the *aspects* mechanism of the Melampus data model ([RiSc91]). However, there is still no reuse of aspects by inheritance, and as a side-effect no dynamic binding. Also there are no restrictions on aspect acquisitions. An

aspect may hide methods of its base aspects, resulting in extensions of an object that do not conform to (i.e. are a subtype of) its previous type.

Multiple objects, single instantiation, inheritance

Most "roles of objects" approaches represent conceptual objects by a *set* of physical objects. Each physical object represents one role of the conceptual object and is an instance of one most specific class. Messages that cannot be answered by the message receiving role (object) are forwarded to its superroles (superobjects).

Sciore. The Vision database system of [Scio89] has a purely prototype- and delegation-based data model with no notion of class and type. Classes are simulated by "prototype" objects delegating to the other prototype objects representing superclasses. Instantiation is simulated by cloning (copying) a prototype and all its parent prototypes. Thus, a conceptual object is *always* split into subobjects, e.g. a woman would be represented by a female object delegating to a person object, instead of only one WOMAN instance (cf. first row of table 5). Due to the integration of delegation with classes and inheritance the Darwin-based modelling of the same message passing semantics requires no artificial splitting of objects (e.g. Sally as a person is represented by exactly one object). This better expresses the application semantics, since delegation is only used if there is a potential for role change or sharing of common roles. Also the system is more efficient, since object hierarchies are smaller, speeding dynamic binding of delegated messages.

Gottlob & al. The model of [GSR94] is implemented in Smalltalk ([GR89]), based on the ability to create classes at run-time and to treat messages as first-class objects. Consultance is implemented by redefining the `doesNotUnderstand: aMsg` method such that it forwards `aMsg` to an object referenced by an instance variable. Each role class is dynamically created as a subclass of the predefined class `RoleType`. As a consequence of this implementation and of Smalltalk's single inheritance role classes cannot inherit from other classes. Thus, there is no way to express that different role classes have a common structure. For instance, the example illustrated in fig. 8 could be expressed in Darwin but not in the model described in [GSR94]. The example models that only women can be mothers and only man can be fathers. If `PARENT` were modelled as a role of `PERSON` (as required by [GSR94]) any combination of role subclasses, e.g. female fathers and male mothers, would be allowed at run-time. Alternatively, one would have to manually duplicate the `PARENT` and `PERSON` properties in their respective subclasses, contradicting the two main goals of object-orientation, good conceptual modelling and reuse.

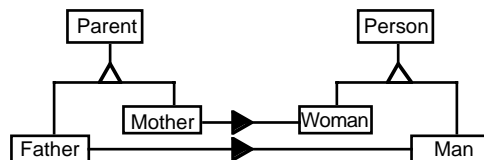


Fig 8: Role classes must have superclasses in order to express common features that are not roles.

In order to model situations like the one depicted in figure 9a [GSR94] introduced the notion of *qualified roles*. The fact that a project manager can manage different projects is represented by letting the same person have two project manager roles, which are instances of the same class. In order to distinguish the different instances the class must define a special instance variable named *qualifier*, which at run-time must have a unique value in every instance. We did not include this concept into our reference role model, because, in our opinion, it needlessly

complicates the model. Figure 9b illustrates how we alternatively recommend to model the intended semantics. The general rule is that one object always has one role of one type and multiplicity is modelled by an association with a corresponding cardinality. In fig. 9b, for instance, the project manager *pm* manages two projects, *p1* and *p2*. We would use different project manager objects only to express that the behaviour as a manager of one group of projects is different from the behaviour as a manager of another group of projects. But then each project manager object would be an instance of a different class, implementing a different project manager behaviour. This alternative does not work well for non-binary relationships. E.g. if an employee can work for different companies and has different positions in each company it is better to represent EMPLOYEE as a qualified role class and specifically link each of the different employee roles of the same person to their specific companies and jobs.

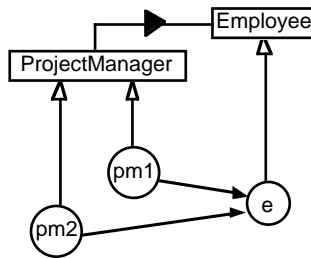


Fig. 9a: "Qualified roles": a manager of different projects is represented by different manager objects (The continuous dark arrows are "player" references.)

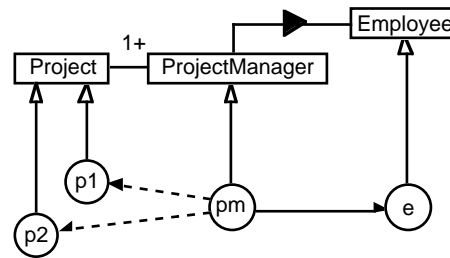


Fig 9b: Alternative representation: one manager associated to different projects (Dashed arrows are "normal" object references.)

Wieringa & al. The approach of [WRS94] models roles using a forwarding mechanism that is a very restricted, compile-time version of consultance. If no method for the message receiver.*msg* is defined in the static type of receiver, then the parser replaces the receiver by its parent object whose static type has a method definition. In the example from fig. 10, a message *g.msg* would be replaced by *g.parent.parent.msg* if *msg* is defined in class PERSON but not in class STUDENT and GRADUATED. This compile-time replacement has the effect of statically binding *msg* to its definition in the PERSON role. Note that this is slightly more general as static binding to the definition in the PERSON class: if class MAN redefines *msg*, then the redefined version would be executed. However, redefinitions of *msg* in subclasses of GRADUATED and STUDENT (e.g. in UNIVERSITY) would be ignored. Thus in [WRS94] the parent/player attributes are not semantically distinct at run-time from any other attributes. This is illustrated by normal object references (dashed arrows) in fig. 10 and table 5.

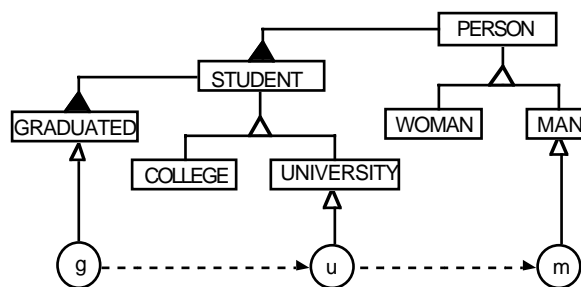


Fig. 10: Compile-time delegation / consultance statically binds messages to a specific role.

Using Darwin as a common framework, the following table shows how much of the semantics of Sally's example (fig. 1) would be expressible in the discussed "roles of objects" approaches that support dynamic binding, resp. reuse of roles by inheritance / delegation. Class names are abbreviated by their first letters. In the column "original representation" we show the original semantics, *not* the original notation. In order to simplify the presentation and ease comparison

the various original notations have been translated to the notation introduced in fig. 1 and 4. In addition, "normal" object references are indicated by dashed arrows.

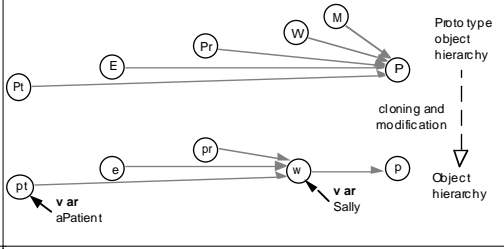
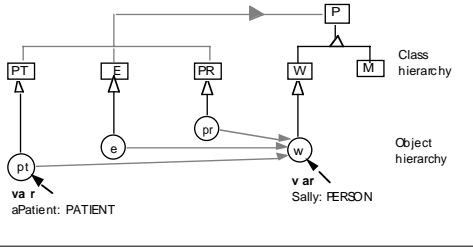
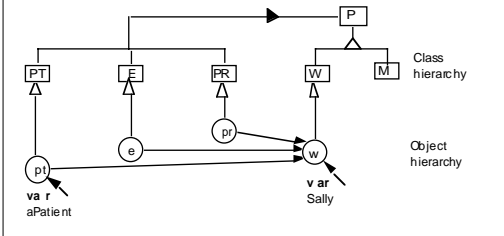
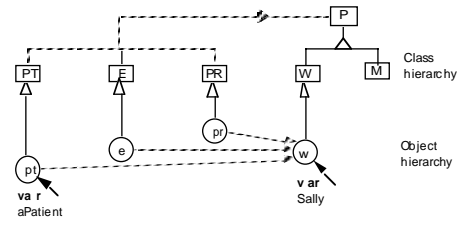
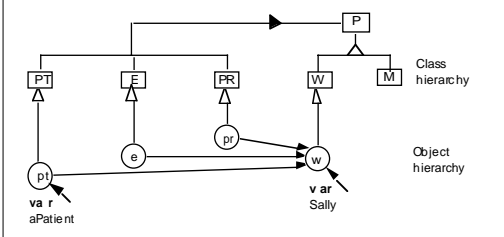
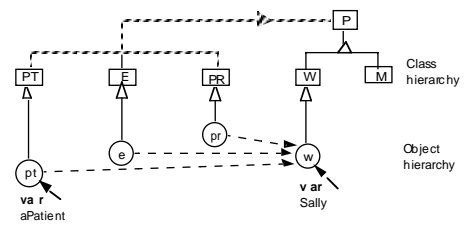
Roles of objects (single instantiation, multiple objects)		
Approach	Original representation of Sally's example	Semantically equivalent Darwin representation
[Scio89]: prototype-based dynamic delegation, untyped		
[GSR94]: class-based dynamic consultance, untyped		
[WJS94]: class-based compile-time consultance, typed		

Table 5: Existing "roles of objects" approaches that support dynamic binding and their different semantics illustrated by their mapping to Darwin

Objects with roles

The only variant of "objects with roles" that has been considered in literature corresponds to category *B1* in table 2, i.e. objects that have a fixed interface but specialise their behaviour when subroles are added. The approaches of this category that allow multiple sibling roles to be played simultaneously – category *B1/c* in table 4 – are based on multiple instantiation resp. multiple type membership.

Category *B1/b*: Fixed interface, specializable behaviour, simultaneous sibling roles, no perspectives

Fishman & al. Iris ([FDC+87]) was apparently the first system that gave up the single most specific instantiation restriction. Iris objects may acquire and lose types retaining their identity. Methods in most specialised types override methods in more general types, but there is no criterion for deciding which method to select if no unique most specialised type that contains a selectable method exists. There is no perspective-or context-dependent behaviour.

Bertino & Guerrini. In a recent paper [BeGu95] compared different options for dynamic binding of messages to instances of multiple most specific classes in the context of the deductive, object-oriented database language Chimera ([CeMa93]). Their *preferred class*

approach is based on defining a total order among classes, prioritising based on the static type of an object, which induces a total order among the run-time types / classes of an object that is exploited for method dispatch. As an alternative the *argument specificity approach* was considered, a method inspired by techniques for multi-method dispatch¹. Each of these alternatives has its particular strength and deficiencies. In particular, the argument specificity approach does not allow context-dependent behaviour, is only partially applicable, and not type-safe. The preferred class approach remedies these drawbacks, but is too coarse grained, since defining an ordering on *classes*, *all* methods from a higher priority class will be preferred to methods from lower priority classes. It is not possible to define, e.g. that for method *m* the implementation from class *A* will be preferred whereas for method *n* the implementation from class *B* will be preferred. In contrast, our approach allows fine-grained, method-specific resolution of multiplicity by renaming, allows context- and perspective-dependent behaviour, is generally applicable, and type-safe (partially using run-time checks).

Category B1/c: Fixed interface, specializable behaviour, perspectives

We know of only two approaches that combine "perspectives" with "objects with (specialisable) roles". The probably best-known one, used in the interactive database programming language Fibonacci ([ABGO93]), will be discussed in this section. The approach of [VeCa93] has (appears to have) the same message passing semantics. It extends the work done on Fibonacci by treatment of dynamic aspects of role changes and will therefore be discussed in the section "7.3 Migration patterns".

Fibonacci is the successor of Galileo ([ACO85]) and Nuovo Galileo ([ABGO91]). In Fibonacci objects can be regarded from the perspective of each of their current roles and each role, *r*, may use methods defined for its subroles, *if the methods had already been defined for r*. Thus subroles do not extend the interface of their superroles. Their use by superroles is restricted to specialise already existing behaviour.

When there are different most specific subroles, the method to answer a message is selected from the one acquired last². Making method dispatch dependent on the role acquisition history introduces a significant degree of nondeterminism and was therefore not included in our reference role model. However, if this behaviour is desired, it can be expressed in Darwin by a variant of consultance that we call *forced consultance*. Whereas consultance *only* forwards messages for locally *undefined* methods, forced consultance *always* forwards received messages, ignoring local definitions. Fibonacci's message interpretation mechanism can be expressed by letting subroles delegate to superroles and each superrole forcedly consult its last acquired subrole (cf. fig. 11). Whenever a new subrole is added, the consultance attribute of each of its superroles has to be redirected towards the new role. E.g., in fig. 11 the consultance attribute of object *p* references the object *e*, which represents the newly acquired employee subrole. Before the acquisition of *e* the consultance attribute referenced the student subrole, *s*. This change of

¹ Note that, unlike in [NRE92] and [NgRi92] for instance, the argument specificity approach is an adaptation of criteria for multi-method dispatch to a data model in which methods are defined *within*, not outside of classes.

² In order to ensure type safety, the type of the last acquired role must be a subtype of the types of all other roles added simultaneously.

consultance relationships is reflected in Fibonacci by dynamic restructuring of object-specific¹ dispatch tables.

Objects with roles, variant B1 c) (fixed interface, specialisable behaviour, different perspectives)		
Approach	Original example	Semantically equivalent Darwin representation
[ABGO93]: type-based, dynamic dis- patch table re- structuring		

Fig. 11: Fibonacci's semantics expressed in Darwin.

"Forced consultance" is represented by consultance links with hollow heads ().

Note how the specialisation semantics is statically captured by the class-level definition of an optional "forced consultance" attribute in every role class², together with automatically created role acquisition / deletion methods, which manipulate the attribute at run-time. By our type rules instances of child classes may be used where instances of declared parent classes are expected, e.g. the consultance attribute of *p* may reference an EMPLOYEE instance instead of a PERSON instance. However, only PERSON messages will be accepted by the type-checker.

The above mapping shows that regarding *self*-reference semantics we share the position taken in Fibonacci, despite differences in terminology³. Also, the ability to provide different implementations for the same role type is present in both approaches. In Fibonacci it is achieved by providing different constructor functions for objects of the same type, in our model it is achieved by subclassing role classes.

In [ABGO93] the problem of extending an object with independent roles in a sound and sensible way has been described as the essential motivation for one of the primary design choices, "*cousin role independence*".

"Suppose that a type Person has two different subtypes Student and Employee, and that both of them add a property PersonalCode to the supertype. The two personal codes have unrelated semantics, and maybe even a different type. Let john be created as a Person and later on extended, first to Student with code 100200 and then to Employee with code 'jhn698'. In a language with late binding and with-

¹ Fibonacci's dispatch tables are actually "role-acquisition-history-specific". The implementation optimizes storage consumption by letting all objects that have acquired the same roles in the same order share *one* dispatch table.

² The figure is only intended for illustrating the principle of the mapping. In practice, the consultance attribute and the corresponding methods do not need to be manually defined in every role class, as suggested by the illustration. They are defined only once in the class *RoleWithSpecializableBehaviour* and may be inherited by every role class that should have B1/c semantics (cf table 4).

³ Fibonacci's notion of type-level inheritance corresponds to our notion of delegation and Fibonacci's notion of delegation corresponds to forced consultance. In our terminology the "cousin roles" of Fibonacci are called "sibling roles".

out roles, `johnAsStudent` answers 'jhn698' to a message `personalCode`, or `johnAsEmployee` answers 100200, because the objects always exhibit a uniform behaviour. This is both a semantic error and a type-level error."

The above problem is solved in Fibonacci, by interpreting roles as perspectives. In Fibonacci, `johnAsStudent` correctly answers 100200 and `johnAsEmployee` correctly answers 'jhn698'. However, the message `john.personalCode` is disallowed, and its possible semantics is left open. In our role model (chapter 2 + 6) the message is legal (subject to a run-time check for role availability) and its semantics is well-defined. The two definitions of `PersonalCode` will automatically be recognised to be semantically different, due to their different origin. The semantics of `PersonalCode` visible to `PERSON` instances will always be the one of the role *class* added first to the schema (independent of the role acquisition history of individual objects). The other definition will be accessible for `PERSON` instances in a renamed form, e.g. `PersonalCodeAsEmployee`, if the `EMPLOYEE` role class has been added after the `STUDENT` role class. The automated mechanisms ensure that joint use of independently developed classes does not change their semantics and leads to no type-level errors. However, the schema designer is free to explicitly specify other strategies (e.g. like the ones of [ScNe88]), by changing the definition of the automatically generated methods and renaming specifications.

In [ABGO93] the creators of Fibonacci motivate why they consider that "cousin role independence" excludes inheritance between cousin roles.

"The message interpretation mechanism, ensures, in a word, that there is neither interference nor inheritance between cousins. This is very important, since in general when an object is extended with two cousin roles (e.g. a Person with Student and Employee), if the same method is defined in all the three roles, the two cousins can specialise it with two subtypes T' and T" of the type T assigned by the father to that method, but there is no subtype relation between T' and T", which implies that inheritance between cousins would be unsound not only with respect to the modelling principles, but also with respect to the language typing rules."

Regarding typing, we do not face the above-mentioned problem, since our type rules require *non-variance*, i.e. equality of the signatures of common methods in player classes and role classes. Regarding modelling principles, we think that providing "holistic" perspectives of an object, which include all the properties of all its current roles, is often a more faithful modelling of reality than restricting perspectives to present only an excerpt of an object. Moreover, it solves the classification / identification paradoxon described in the introduction. We give the programmer the freedom to decide what he needs in a given context. With strictly static type-checking a perspective presents only a part of an object. With run-time checking for role availability an holistic view is possible.

6.3. Legal Role Histories

In real life there are many constraints on the roles that an object may play. For instance,

- (1) a person may not be male *and* female, neither simultaneously, nor in sequence,
- (2) only male persons can become fathers and only female persons can become mothers,
- (3) a person that once became a parent cannot ever abandon this role.

Although most researchers have pointed out the significance of expressing such constraints, only a few tackle the problem and the approaches in this area are very heterogeneous. They range from explicit general constraint management ([Su90], [BeGu95]) over constraints in the form of Eiffel-style conditions and invariants ([VeCa93]) to fixed sets of predefined, role-specific constraints expressed by special semantic relationships between objects ([Papa90]) or special built-in operations ([LiDo94]). The other extreme is marked by the Vision database system of [Scio89], where objects are not constrained in any way and can change their structure and behaviour at run-time in unpredictable and inconsistent ways.

In most approaches we are aware of there is no distinction between role-playing and inheritance (e.g. [BeGu95], [VeCa93]) or role-playing and subtyping (e.g. [ABGO93]). Therefore explicit constraints are the only way to express restrictions on role changes. Integrating constraints as first-class citizens into a language is certainly the most general solution to the problem at hand. Unfortunately, explicitly specified constraints cannot be used for disambiguating method dispatch – at least it is unclear how this could be done in a convenient way.

In contrast, by distinguishing between static, class-level inheritance and dynamic, object-level delegation, Darwin uses two conceptual hierarchies, which unambiguously define the message passing semantics and at the same time implicitly express simple constraints. For instance, constraints like (1) can be expressed by making MAN and WOMAN subclasses, *not* role classes, of PERSON (c.f. fig. 1). The integration of delegation with classes and types, allows to statically constrain the semantically meaningful object hierarchies at run-time. Modelling of constraint (2) is shown in fig. 8. For expressing condition (3) and other, more complex conditions, additional machinery is required. However, this is an issue that is orthogonal to the variants of message passing semantics discussed in this paper. Any of the approaches followed in literature could be integrated into our framework.

7. Conclusions

In this paper we have pointed out that most of the problems of modelling dynamic change of behaviour implied by changing roles of an object can best be tackled in the framework of an object-oriented data model, that has *no* role modelling specific properties. We have defined the Darwin model, which integrates the well-known notions of type, class, instance, and inheritance with another notion already known for about ten years but never used in the context of typed, class-based systems: object-based dynamic inheritance. Furthermore, we have analysed the ingredients that make up the message passing semantics of a role model and have defined corresponding semantic categories. Using the Darwin model as a common framework, we have been able to express and compare previous role models from different categories, stating precisely their distinctions with respect to message passing semantics. Our classification defines a set of further categories which have not yet been proposed but which can be expressed with the concepts of Darwin.

This added power was demonstrated by defining a new role model, which solves the apparent classification / identification paradoxon present in previous approaches. The proposed role model appears to be the first one that allows unrestricted coexistence of the notions of *perspectives*, *roles of objects* and *objects with roles*. Each role can either present a restricted perspective of an object or a global perspective that includes all the properties available to all the roles of an object at a given moment. In both cases the behaviour that can be perceived from the perspective of different roles can still be different due to dynamic binding.

The only aspect that seems to require going beyond types, classes and inheritance is modelling of legal role acquisition patterns. Although Darwin can implicitly express simple constraints, more powerful solutions must still be explored, and might require explicit constraint management as a further extension.

The extension of traditional object-oriented models by object-based inheritance is more than just a vehicle for expressing the semantics of a variety of role models and extending the state of the art in this domain. It is a general model for sharing in object-oriented systems, which can be used for modelling flexible solutions in many other domains of increasing significance to database technology (e.g. versioned objects [Katz90]). At the same time it has an efficient implementation that exploits the state of the art implementation techniques for strongly typed class-based object-oriented languages. Thus it promotes the easy extension of a system with high-level conceptual modelling concepts by simple mappings to a powerful base language and the independent optimisation of the base language implementation. Therefore we suggest extensions along the lines of Darwin as a means to achieve maximum expressiveness of a language without the need to burden its compiler and run-time system with many special purpose concepts. We are following this approach by implementing all of the described role modelling functionality as a high-level graphical schema design / CASE tool that will automatically generate corresponding "Darwin code". Independently, we are implementing an extension of the language Java ([GoMc95], [SUN95]) that conforms to the Darwin model, adapting the implementation techniques from [Knie94] to the special architecture of the Java environment.

Acknowledgements

We gratefully acknowledge the support of all who helped this work mature. Giorgio Ghelli, Giovanna Guerrini, Brigitte Röck and Michael Schrefl have kindly helped me understand their approaches. This paper has also benefited from many discussions with Wolfgang Reddig, Thomas Lemke, Rainer Manthey, Pascal Costanza and Matthias Schickel. Thanks are due to Pascal and Matthias also for their implementation efforts.

References

[ABW+90]

M. Atkinson, F. Bancilhon, D. De Witt, K. Dittrich, D. Maier, S. Zdonik: "The OO DB System Manifesto". In [DOOD 90], 40-57.

[ACO85]

A. Albano, L. Cardelli, R. Orsini: "Galileo: A Strongly Typed, Interactive Conceptual Language". In *ACM Transactions of Database Systems*, Vol. 10, No. 2, pp. 230-260, 1985. Also in: S.B. Zdonik, D. Maier (Eds.): "*Readings in Object-Oriented Database Systems*", Morgan Kaufmann, San Mateo, CA, pp. 147-161, 1990.

[AGO92]

A. Albano, G. Ghelli, R. Orsini: "Objects for a Database Programming Language". In P. Kannelakis, J.W. Schmidt (Eds.): *Proceedings of the 3rd International Workshop on Database Programming Languages*, Morgan Kauffman Publishers, San Mateo, CA, pp. 236-256, 1992.

[ABGO93]

A. Albano, R. Bergamini, G. Ghelli, R. Orsini: "An Object Data Model with Roles". In *Proceedings of the 19th VLDB Conference*, Dublin, Ireland, 1993, pp. 39-51.

[BaDa77]

C.W. Bachman, M. Daya: "The role concept in data models". In "*Proceedings of the 3rd International Conference on VLDB*", 1977, pp. 464-476.

[BaBu90]

François Bancilhon, Peter Buneman (Eds.): "*Advances in Database Programming Languages*", ACM Press (Frontier Series), 1990.[BBMP95]

G.H.W. M. Bronts, S.J. Brouwer, C.L.J. Martens, H.A. Proper: "A Unifying Object Role Modelling Theory". In *Information Systems*, 20, 3 (May 1995), pp. 213-235.

[CaWe85]

L. Cardelli, P. Wegner: "On understanding types, data abstraction, and polymorphism". *ACM Computing Surveys*, 17, 4 (Dec. 1985), pp. 471-522.

[CCHO89a]

P. S. Canning, W. R. Cook, W. L. Hill, W. G. Olthoff: "Interfaces for Strongly-Typed Object-Oriented Programming". In [OOPSLA 89], pp. 457-468.

[CeMa94]

Stefano Ceri, Rainer Manthey: "Chimera: A Model and Language for Active DOOD Systems". In: [EdKa94], pp. 9-21.

[Cham93]

C. Chambers: "Predicate classes". In [ECOOP93], pp. 268-296.

[Cham93]

C. Chambers: "The Cecil Language: Specification and Rationale". Dept. of Computer Science and Engineering, Univ. of Washington, TR 93-03-05, March 1993 (available from ftp.cs.washington.edu).

[CUL89]

C. Chambers, D. Ungar, E. Lee: "An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language Based on Prototypes". In [OOPSLA 89], pp. 49-70.

[ECOOP 92]

Proceedings of European Conference on Object Oriented Programming, Utrecht, 1992, Springer Verlag, LNCS 615, 1992.

[ECOOP 93]

O. Nierstrasz (Ed.): *Proceedings of European Conference on Object Oriented Programming*, Kaiserslautern 1993, Springer Verlag, LNCS 707, 1993.

[ECOOP 95]

Walther Olthoff (Ed.): *"ECOOP '95 – Object-Oriented Programming"*, *Proceedings of 9th European Conference Århus, Denmark, August 1995*, Springer Verlag, LNCS 952, 1995.

[EdKa94]

Johann Eder, Leonid Kalinichenko: *"Extending Information System Technology"*, *Proceedings of Second International East-West Database Workshop*, Sept. 1994.

[FBC+87]

D.H. Fishman, D. Beech, H.P. Cate, E.C. Chow, T. Connors, J. W. Davis, N. Derett, C.G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M.A. Neimat, T.A. Ryan, M.C. Shan: "Iris: An object-oriented database management system." *ACM Transactions on Office Information Systems*, **5**(1):48-69, January 1987.

[FAC+89]

D.H. Fishman, J. Annevelink, E.C. Chow, T. Connors, J. W. Davis, W. Hasan, C.G. Hoch, W. Kent, S. Leichner, P. Lyngbaek, B. Mahbod, M.A. Neimat, T. Risch, M.C. Shan, W.K. Wilkinson: "Overview of the Iris DBMS." In [KiLo89], pp. 219-250.

[GoMc95]

James Gosling, Henry McGilton: "The Java Language Environment -- A White Paper". Document of Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, CA 94043, U.S.A, available electronically from <http://java.sun.com/whitePaper/java-whitepaper-1.html>.

[GSR94]

G. Gottlob, M. Schrefl, B. Röck: "Extending Information Systems with Roles". To appear in *ACM Transactions on Information Systems*.

[HaNg87]

Brent Hailpern, Van Nguyen: "A Model for Object-Based Inheritance". In [ShWe87], pp. 147-164.

[Katz90]

Randy H. Katz: "Towards a Unified Framework for Version Modeling in Engineering Databases". *ACM Computing Surveys*, **22**, 4 (December 1990), pp. 375-395.

[KhCo86]

S.N. Khoshafian, G.P. Copeland: "Object identity". In *Object-Oriented Programming Systems, Languages and Applications*, pp. 406-416, 1986. SIGPLAN Notices 22 (12)

[KiLo89]

W. Kim, F.H. Lochovsky (Eds.): *"Object-oriented Concepts, Databases, and Applications"*. ACM Press, Addison-Wesley 1989.

[Knie94]

G. Kniesel: "Implementation of Dynamic Delegation in Strongly Typed Inheritance-Based Systems". *Technical report IAI-TR-94-3, Oct. 1994, University of Bonn, Germany*.

[Knie96]

G. Kniesel: "A Uniform Model of Sharing for Object-Oriented Programming". Ph.D.-thesis, University of Bonn, 1996 (in preparation).

[KRC91]

G. Kniessel, M. Rohen, A.B. Cremers: "A Management System for Distributed Knowledge Base Applications". In W. Brauer, D. Hernández (eds.): "Verteilte Künstliche Intelligenz und Kooperatives Arbeiten" (Distributed Artificial Intelligence and Cooperative Work). Springer-Verlag 1991, pp. 65-76.

[Nier89]

O. Nierstrasz: "A Survey of Object-Oriented Concepts". In [KiLo89], pp. 3-22.

[NRE92]

G.T. Nguyen, D. Rieu, J. Escamilla: "An Object Model for Engineering Design". In [ECOOP 92], pp. 233-251.

[NgRi92]

G.T. Nguyen, D. Rieu: "Multiple Object Representations". In Proceedings of the 20th ACM Computer Science Conference, Kansas City, 1992.

[OOPSLA 87]

N. Meyrowitz (Ed.): "OOPSLA '87". Object-Oriented Programming: Systems, Languages and Applications - Conference Proceedings, Oct. 1987, Orlando, Florida. Special issue of *SIGPLAN Notices* 22 (12), December 1987, ACM Press.

[OOPSLA 89]

N. Meyrowitz (Ed.): "OOPSLA '89 Conference Proceedings. Object-Oriented Programming: Systems, Languages and Applications, Oct. 1989, New Orleans, Louisiana. *SIFPLAN Notices* 24 (10) Oct. 1989, ACM Press.

[Papa91]

M.P. Papazoglou: "Roles: A Methodology for Representing Multifaceted Objects. In *Proceedings of the International Conference on Database and Expert Systems Applications*, Springer Verlag 1991, pp. 7-12.

[Pern90]

Barbara Pernici: "Objects with Roles". In Proceedings of the Int. Conf. on Office Information Systems, Boston (Ma), ACM Press, 1990.

[RiSc91]

J. Richardson, P. Schwarz: "Aspects: Extending Objects to Support Multiple, Independent Roles." In J. Clifford, R. King (Eds.): *Proc. of the ACM SIGMOD International Conference on Management of Data*, 1991, pp- 298-307.

[ScNe88]

M. Schrefl, E.J. Neuhold: "Object Class Definitions by Generalisation Using Upward Inheritance". In *Proceedings of the IEEE Fourth international Conference on Data Engineering*, 4-13.

[Scio89]

E. Sciore: "Object Specialization". *ACM Transaction on Information Systems*, 7(2), 1989, pp. 103-122.

[ShSw89]

J. J. Shilling, P.F. Sweeney: "Three Steps to Views: Extending the Object-Oriented Paradigm". In [OOPSLA 89], pp. 353-362.

[ShWe87]

B. Shriver, P. Wegner (Eds.): "*Research Directions in Object-Oriented Programming*". MIT Press, Computer Systems Series, Cambridge, MA, 1987.

- [SmUn95]
Randall B. Smith, David Ungar: "Programming as an Experience: The Inspiration for Self". In [ECOOP 95], pp. 303-330.
- [StZd89]
L.A. Stein, S.B. Zdonik: "Clovers: The Dynamic Behaviour of Types and Instances". Brown University Technical Report CS-89-42, Nov. 1989.
- [Stro87]
B. Stroustrup: "Multiple Inheritance for C++". In *Proceedings of European Unix User's Group Conference*, Helsinki, May 1987.
- [Su 91]
J. Su: "Dynamic Constraints and Object Migration". In *Proceedings of 17th International Conference of Very Large Data Bases*, Barcelona, pp. 233-242.
- [SUN95]
Sun Microsystems, Inc, 2550 Garcia Avenue, Mountain View, CA 94043, U.S.A: "The Java Language Specification". Available electronically from <ftp://ftp.javasoft.com/docs/javaspec.ps.tar.Z> andps.zip.
- [UnSm87]
D. Ungar, R.B. Smith: "Self: The Power of Simplicity". In [OOPSLA 87], pp. 227-242.
- [VeCa93]
Amândio Vaz Velho, Rogério Carapuça: "From Entity-Relationship Models to Role-Attribute Models". In R.A. Elmasri, V. Kouramajian, B. Thalheim (Eds.): "*Entity-Relationship Approach - ER '93, 12th International Conference on the Entity-Relationship Approach*" Arlington, Texas, Dec. 1993, Springer Verlag, LNCS 823, pp. 257-270.
- [Wegn89]
P. Wegner: "Concepts and paradigms of Object-Oriented Programming" (Expansion of OOPSLA 89 keynote talk). *OOPS Messenger*, 1,1 (August 1990), pp. 7-87.
- [WJS94]
R. Wieringa, W. de Jonge, P. Spruit: "Roles and Dynamic Subclasses: A Modal Logic Approach". In *ECOOP 94 Proceedings*, Springer-Verlag, LNCS 821, 1994.
- [Zdon90]
Zdonik: "Object-Oriented Type Evolution". In: [Bancilhon & Buneman 90], pp. 277-288.