Role Modeling in SelfSync with Warped Hierarchies

Ellen Van Paesschen
Programming Technology Lab

Wolfgang De Meuter
Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel

Maja D'Hondt
System Software Engineering Lab

Email: {evpaessc,wdmeuter,mjdhondt}@vub.ac.be

Abstract

In object-oriented software engineering roles are considered both classifications and instances. To reduce the gap between the conceptual modeling of roles and a corresponding implementation, we propose a new role modeling concept based on warped inheritance hierarchies. We integrated this new modeling concept in our prototype-based object-oriented round-trip engineering environment SelfSync. In this way it is possible to model roles in an Extended Entity-Relationship diagram while the corresponding implementation objects are automatically created and synchronized with the conceptual model. We apply constraint enforcement during the lifetime of role objects, based on dependency and role combinations.

Introduction

There exist various view points on roles in object-oriented software engineering. In this paper, we merge the perspectives on roles as *dynamic multiple classifications* of objects, and as *instances to be adjoined* to the objects that perform the role.

The role as a *modeling concept* cannot be emulated by any of the better established conceptual or object-oriented modeling constructs. The challenge of defining a suitable role modeling concept is to integrate it into existing modeling frameworks causing as little redefinition as necessary, while capturing as much of its semantics as possible (Steimann 2000).

Due to the fact that roles are considered both specializations and generalizations of the entities performing the role (Steimann 2000), the role modeling concept can be mapped to the *subtype-supertype paradox* (Cockburn 1999). A typical example of this problem is the circle-ellipse case where state and behavior do not follow the same specialization/generalization hierarchical setup.

We introduce the modeling concept of *warped inheritance hierarchies* (Paesschen, Meuter, & D'Hondt 2004) to alleviate the difficulties during the conceptual modeling of roles, caused by the subtype-supertype paradox. These warped hierarchies are based on a separation between state and behavior inheritance in the prototype-based language Self (Ungar & Smith 1987). Implementing warped hierarchies applies parent sharing, multiple inheritance and dynamic parent modification. Moreover, using prototypes and delegation brings significant advantages for modeling roles: there is no difference between roles as classifications and as instances, and roles can be added and removed dynamically.

The warped hierarchy modeling concept was integrated in the Extended Entity-Relationship model (Chen 1976) of our round-trip environment *SelfSync* (Paesschen, Meuter, & D'Hondt 2005; Paesschen, D'Hondt, & Meuter 2005) that was built on top of Self. In this way, roles are modeled in SelfSync while a corresponding implementation is automatically created that combines both generalization and specialization between roles and the objects that perform them.

This paper is structured as follows. We start with mapping conceptual modeling of roles to the subtype-supertype paradox. Next we introduce the concept of warped inheritance hierarchies in the prototype-based language Self. We describe how these warped hierarchies were integrated in our round-trip engineering environment SelfSync. Our approach is evaluated based on a number of characteristics (Steimann 2000) shared by most role-related research and we provide a comparison to related approaches. Finally, a conclusion is presented.

The Subtype-Supertype Paradox in Role Modeling

In this section we relate the conceptual model of roles in general to the *subtype-supertype paradox* that deals with a reverse specialization setup between state and behavior.

Usually, domain modeling concepts can easily be mapped to object-oriented programming languages. The state-of-the-art in both modeling and programming languages is dominated by the class-based paradigm. Furthermore, the current standard for modeling is the Unified Modeling Language (Fowler & Scott 2000) which is targetted to classes as well. During conceptual modeling modeled entities correspond to a class and taxonomies of entities give rise to class-hierarchies. However, there exists a number of occasions where this straightforward approach results in a setup in which the entities follow the standard hierarchical taxonomies but in which the corresponding implementation demands exactly the reverse hierarchy.

For example, from a real world (domain model) perspective, a circle really *is-a* kind of ellipse with its *radius* being both the major semi-axis *a* as well as the minor semi-axis *b* used in ellipses. Therefore, circles should be implemented as specializations of ellipses. In a class-based language the `circle` type is implemented with inheritance: as a subclass of the `ellipse` type. However, this results in inefficient code since a `circle` will not use all instance variables inherited from `ellipse` because both its axes are equal by definition. In general there exist two main difficulties. First, the state of `circle` is less specialized than the state of `ellipse` (i.e. contains less attributes) while the behavior of `circle` is more specialized than the behavior of `ellipse` (i.e. contains more methods). Second, circles can receive messages intended for ellipses, transforming them dynamically into ellipses, and vice versa. For instance, when a `circle` receives a `stretch` message that largens the width of an `ellipse`, a `circle` would become an `ellipse` but be of class `circle`.

The above example illustrates a fundamental problem in class-based software modeling: conceptual subtypes are not always implementation subtypes. This is refered to as the subtype-supertype paradox.

Another instance of this problem is the conceptual modeling of roles (Fowler 1997). Consider the following example. In a company, a person can (at most) have two roles: either that of a salesman or of an engineer, and at the same time, a manager role. In a class-based design, this setup is modeled as illustrated in figure 1. In this UML class diagram, multiple classification is used to express the two possible roles of a person, while dynamic classification denotes the mutual exclusiveness of the salesman and the engineer roles.

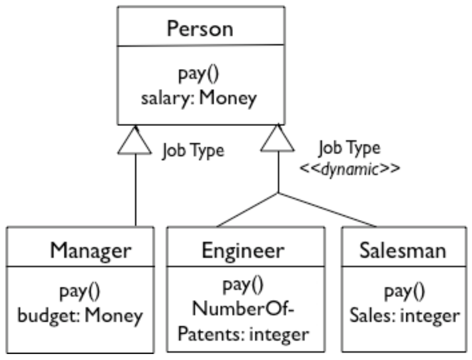

Figure 1: A person and the roles manager, salesman and engineer

Conceptually, the roles a person can perform are subtypes of `person`: an `engineer` is-a kind of `person`. More specifically the behavior of a role type is more specialized than that of the `person` type, e.g. a `pay` method in `engineer` is more specialized than the `pay` method in `person`.

However, also in role modeling conceptual subtypes are not always implementation subtypes, especially when a person can perform multiple roles. When for example both `engineer` and `manager` are subclassed from `person` how to model a person that is both manager and engineer? Instantiating the `manager` class causes the `engineer` class to be invisible and vice versa. Creating combination classes is not feasible: persons can often change dynamically between a large set of roles. Alternatively, roles are modeled with aggregation (Fowler 1997): a set of roles is held by an instance variable in the `person` class. By delegating the messages of `person` to its roles, polymorphism is simulated.

Some of the problems described above can be solved when we consider a person that performs multiple roles as an (implementation) subtype of these roles: the state of a person is extended with the attributes of the roles it performs. For example when a person becomes an engineer it is extended with the state (attributes) of the `engineer` type. When this person-that-is-an-engineer later becomes also a manager, all attributes of the `manager` type are added into this object. However reversing the subclass hierarchy is not an option since this would mean that it is no longer possible to override the behavior of a person by the more specialized behavior of a specific role.

Therefore roles are considered both subtypes (at the level of behavior) and supertypes (at the level of state) (Steimann 2000). Mapping roles to a class-based setup cannot be done straightforwardly since classes contain both state and behavior and they both have to follow the inheritance hierarchy of the class.

Additionally roles add an extra difficulty to the subtype-supertype problem since they can be added or removed dynamically, and a person can have multiple roles implementing overloaded method.

Warped Hierarchies in Self

In this section we discuss a solution to the subtype-supertype paradox by mapping conceptual subtypes to a corresponding implementation based on *warped hierarchies*. As opposed to class-based languages, the *prototype-based language Self* is suitable thanks to *multiple* inheritance and the separation between state inheritance and behavior inheritance.

A Prototype-based Programming Environment

In general, prototype-based languages (PBLs) (Lieberman 1986) can be considered object-oriented languages without classes. Self (Ungar & Smith 1987; Smith & Ungar 1995) is closely related to the syntax and semantics of Smalltalk (Goldberg & Robson 1983) but Self has no classes. Objects in Self are either created ex-nihilo or cloned from a prototype. Self implements a delegation mechanism (Lieberman 1986) that respects the late binding of the `self` variable. In other words method lookup is recursively delegated to all parents and the appropriate method is called in the context of the original receiver. Dynamic inheritance allows Self objects to change their parents at run-time. As do most PBLs, Self supports parent sharing (Chambers *et al.* 1991). Finally, a specific feature of Self is child sharing (multiple inheritance) that occurs when two or more parent objects share the same child object. When modeling knowledge parent and child sharing are constantly combined.

Multiple Inheritance in Self

When modeling an object in Self, the state is contained in a *prototype* while the behavior (shared by all objects of this type) is typically gathered in a *traits* object (Smith & Ungar 1995) that stores shared behavior, to be inherited by the prototype and all its clones. Prototypes and clones inherit from a traits object through parent pointers. In this way copying behavior every time an object is cloned is avoided because it resides in a single shared traits object: i.e. a kind of highly dynamic class-based programming in a PBL.

Analoguously, inheritance is separated between state and behavior: 1) the child prototype inherits from the parent prototype (state inheritance), and 2) the child's traits object inherits from the parent's traits object (behavior inheritance). Since both child and parent prototype inherit from their respective traits objects (behavior inheritance), this setup results in a multiple inheritance structure that is a "diamond" [1], see figure 2. Self avoids ambiguity caused by overloaded

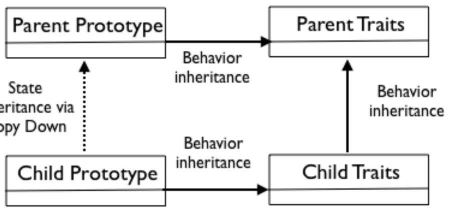

Figure 2: Separation between state and behavior inheritance in Self

methods by performing state inheritance via a `copy-down` (Self-website 2003) of the parent prototype. This mechanism is usually applied at cloning time: a new clone (the child prototype) is created and (part of) the state of the receiver (the parent prototype) is copied into it. A copy-down link ensures that changes to the parent prototype are propagated to the child prototype, similar to subclassing. E.g. when, at any point in its life time, a new attribute is added in the parent object, it is also added to the child prototype. For more details, including an elaborated example we refer to (Paesschen, Meuter, & D'Hondt 2004).

Warped Hierarchies

We propose a solution to tackle the subtype-supertype paradox with two separate "warped" specialization hierarchies: one for state and its reverse for behavior. Thanks to the separation of state and behavior in prototypes and traits, Self allows us to model these hierarchies and implement them with state and behavior inheritance, and dynamic parent modification.

For example, initially the `ellipse` prototype is created by copying down the `circle` prototype through copy-down (and extended with an extra attribute for a major semi-axis value), while the traits of a circle inherits the traits of an ellipse through behavior inheritance, with parent pointers, see figure 3. More specifically, we reversed the direction of the state inheritance hierarchy between the `circle` and `ellipse` prototypes. Thanks to the late binding of the

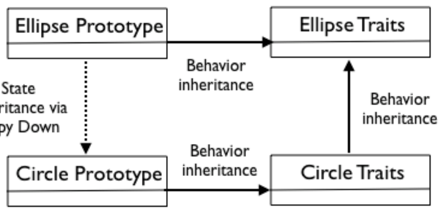

Figure 3: Warped inheritance hierarchies of circle and ellipse

`self` variable, the correct state is accessed when executing methods (e.g, `area`, `circumference`) - and thus polymorphism is ensured.

To tackle the problem of dynamic type changes, we combine dynamic copy-down and its reverse[2], mechanism together with dynamic parent modification.

For example, when a clone of circle is stretched to an ellipse, dynamically[3] the state of the `ellipse` prototype (i.e., the one additional attribute to contain the extra axis value) is copied down into this circle object that now inherits the traits of an ellipse instead of the traits of circle.

Vice versa, an ellipse clone whose major semi-axis is stretched to the same value as its minor semi-axis, becomes a circle clone. The state that is not copied down from the `circle` prototype is removed from its prototype and the inherited traits of ellipse are replaced by the traits of circle.

For roles, the situation is similar but slightly more complicated. Based on the example in the previous section, the initial setup contains a `person` prototype that inherits from `traits person`, and a set of role prototypes (such as `manager`, `engineer`) inheriting from their traits (such as `traits manager`, `traits engineer`), that in their turn all inherit from `traits person`, see figure 4.

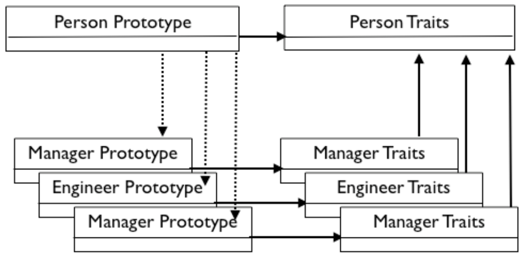

Figure 4: Warped inheritance hierarchies of person and roles

A person (i.e., a clone of the `person` prototype) that (dynamically) starts performing a role is implemented by dy-

[1]Whenever an object C inherits from both objects A and B, whereby A and B inherit from a common object T, one speaks about a diamond.

[2]Removing the attributes that were copied down from an argument prototype from the reciever.

[3]With the help of Self's reflective meta-programming *mirror* mechanism.

namically copying down the state of this role's prototype into the `person` prototype. Next, we remove the inheritance link to the person's traits since this behavior is already inherited via the role prototype. Due to multiple inheritance in Self an arbitrary number of roles can be added dynamically. When a person (dynamically) stops performing a role, the copied down state of the role prototype is removed from the person object. When there are no more roles left, the person object again inherits the traits of `person`.

To ensure polymorphism the dynamic multiple inheritance diamond in a `person` object that inherits state from two or more role prototypes needs to be intercepted. The `person` object then inherits the traits of all these roles that in their turn inherit from `traits person`. When two roles override the same method in their traits, sending the corresponding message to `person` causes ambiguity. One possibility is to combine the overloaded methods from the view point of `person`. When we send a message such as `pay` to `person`, she should get payed for each role. We automatically and sequentially resend the message to the traits of the roles she performs.

We also support role-specific behavior, defined in the role in whose context the person currently is viewed. For instance, sending the message `lunch` to `person`, might result in the specific behavior of having lunch with a friend and not, for example, with the boss and some clients of a company. To achieve this the inheritance link to the desired behavior (traits of a role) is temporarily (dynamically) switched on or off.

Alternative approaches include disambiguating techniques based on lattices that prioritize certain ambiguous methods (Burger 2005) and select the most suitable method to be called.

SelfSync

In this section we describe how the concept of warped hierarchies to model roles is integrated in our round-trip engineering environment SelfSync. SelfSync is an extension of Self that allows modeling Extended Entity-Relation diagrams while a corresponding implementation is automatically created and subsequently synchronized with the diagrams and vice versa.

A Two-Phased Approach

SelfSync supports a two-phased approach, which we present in detail in (Paesschen, Meuter, & D'Hondt 2005; Paesschen, D'Hondt, & Meuter 2005) and briefly summarize here. In the first *active modeling* phase a user draws an Extended Entity-Relationship (EER) diagram while corresponding Self objects — prototypes and traits — are automatically created. In reality, the Self objects *are* the modeled entities: drawing a new EER entity automatically results in a graphical EER entity view being created on a new Self object. Hence, we support incremental and continuous synchronization *per entity* and *per object*: changes to an EER entity are in fact changes to a view on one object and thus automatically propagated to the object via Self's reflection mechanism. Similarly, changes to an object are automatically propagated to the corresponding EER entity. View-dependent information, such as relationships constraints in the EER diagram and method bodies in the Self objects, is preserved during changes and subsequent synchronization.

The second phase of our approach is an *interactive prototyping* process[4]. This phase allows a user to clone and further initialize the prototypes and traits created in the previous phase into ready-to-use objects. As we explain below an interactive conversation with the programmer will make objects adhere to the structure and relationship constraints imposed by the EER model.

The Role Modeling Concept

We extended SelfSync with a modeling concept to model roles that automatically creates warped hierarchies. Therefore we added a new kind of relationship to the EER diagram format denoting that the one entity can play the role of the other entity[5].

In the active modeling phase the example diagram described in the first section was drawn in SelfSync, as illustrated in figure 5. As a result, corresponding implementation

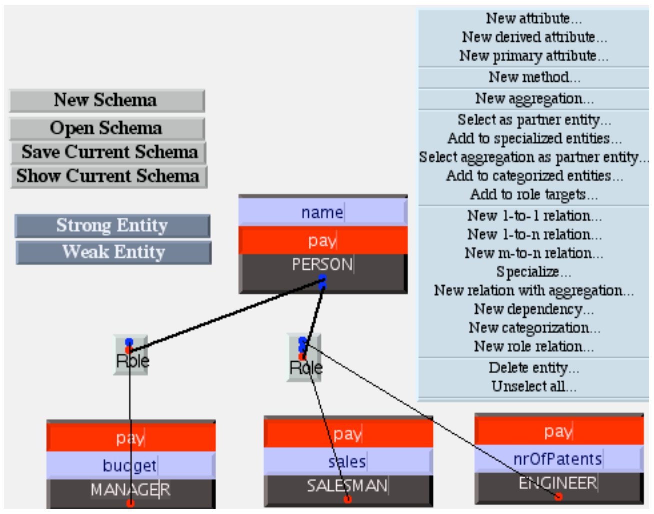

Figure 5: Role modeling example in SelfSync

objects are automatically created and synchronized during modeling. The setup of the implementation objects `person` and `manager` are shown in figure 6. Notice that there is behavior inheritance between the traits of a manager and the traits of a person. The state inheritance is realized in the next phase.

The interactive prototyping phase for this example consists of two possible cases: 1) creating a new person object and letting it perform different roles that are allowed to be

[4]Note that the term *prototyping* is used here to denote the activity of instantiating and initializing a program into a ready-to-use, running system.

[5]SelfSync also supports roles as named places, i.e. the labels on relationships between the entities in the EER diagram. For more details we refer to (Paesschen, D'Hondt, & Meuter 2005).

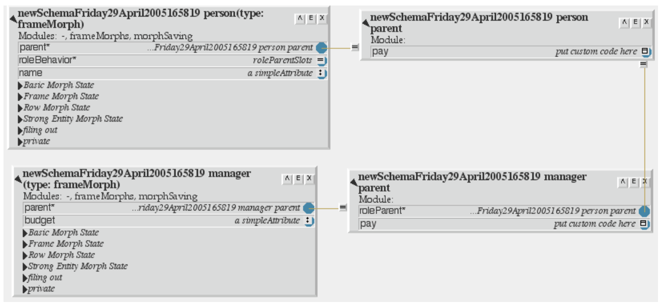
```
newSchemaFriday29April2005165819 person(type: frameMorph)
Modules: -, frameMorphs, morphSaving
parent*          ...Friday29April2005165819 person parent
roleBehavior*    roleParentSlots
name             a simpleAttribute
Basic Morph State
Frame Morph State
Row Morph State
Strong Entity Morph State
filing out
private

newSchemaFriday29April2005165819 person parent
Module:
pay              put custom code here

newSchemaFriday29April2005165819 manager (type: frameMorph)
Modules: -, frameMorphs, morphSaving
parent*          ...riday29April2005165819 manager parent
budget           a simpleAttribute
Basic Morph State
Frame Morph State
Row Morph State
Strong Entity Morph State
filing out
private

newSchemaFriday29April2005165819 manager parent
Module:
roleParent*      ...Friday29April2005165819 person parent
pay              put custom code here
```

Figure 6: Automatically created implementation objects for person and manager

combined and 2) simply creating a new stand-alone role object that is dependent of an existing person object.

We first focus on the first case while the second one is discussed later on. When creating a new person object, the user will be asked for each set of allowed roles (as dictated by the diagram), whether to let this object perform one of the roles. In the case of our example, a new person object can perform at most two roles: 1) the role of a manager (or not) and 2) the role of an engineer or a salesman or neither. For example we can create a person that is both a manager and a salesman, see figure 7. Based on the warped hierar-

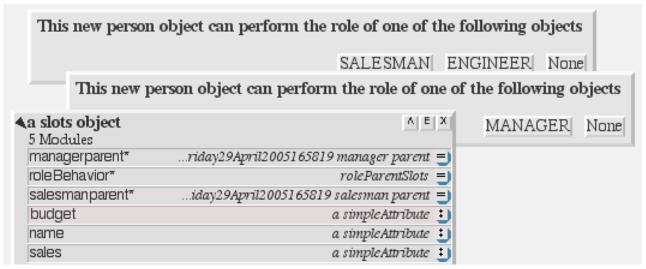
```
This new person object can perform the role of one of the following objects
SALESMAN  ENGINEER  None
This new person object can perform the role of one of the following objects
MANAGER  None

a slots object
5 Modules
managerparent*   ...riday29April2005165819 manager parent
roleBehavior*    roleParentSlots
salesmanparent*  ...iday29April2005165819 salesman parent
budget           a simpleAttribute
name             a simpleAttribute
sales            a simpleAttribute
```

Figure 7: Interactive creation of a salesman-manager person object

chies, this new person object automatically inherits the state of both the `salesman` and the `manager` prototypes (reversed state inheritance), and inherits the behavior of both `traits manager` and `traits salesman` (real-world behavior inheritance).

During the lifetime of this new person object roles can be added or removed dynamically. For example, we can dynamically delete the manager role from the salesman-manager object. In that case, automatically, the copied-down state from the `manager` prototype is deleted from the `person` prototype, and the behavior inheritance of `traits manager` is removed, see figure 8.

Constraint Enforcement

Role Combinations We enforce the allowed combination of roles at two levels. First, during the first case of interactive prototyping, the creation of a new person object is guided based on how we modeled the different roles in the

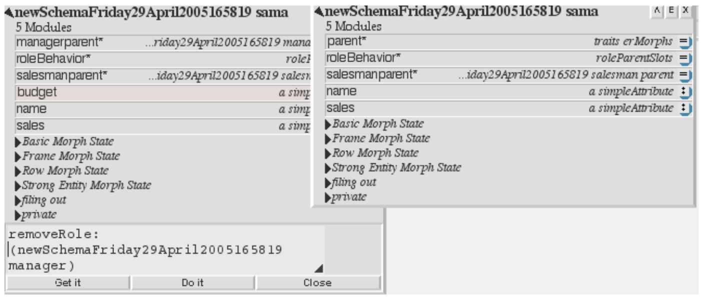
```
newSchemaFriday29April2005165819 sama
5 Modules
managerparent*   ...riday29April2005165819 mana
roleBehavior*    roleP
salesmanparent*  ...iday29April2005165819 salesn
budget           a simp
name             a simp
sales            a simp
Basic Morph State
Frame Morph State
Row Morph State
Strong Entity Morph State
filing out
private
removeRole:
(newSchemaFriday29April2005165819
manager)
Get it   Do it   Close

newSchemaFriday29April2005165819 sama
5 Modules
parent*          traits erMorphs
roleBehavior*    roleParentSlots
salesmanparent*  ...iday29April2005165819 salesman parent
name             a simpleAttribute
sales            a simpleAttribute
Basic Morph State
Frame Morph State
Row Morph State
Strong Entity Morph State
filing out
private
```

Figure 8: Removing the manager role from the salesman-manager person object

EER diagram. In our example, we explicitly modeled two classifications: one for a manager role and one for either a salesman or an engineer role. Based on this knowledge, for each classification we are forced to select at most one of the allowed roles. This implies that newly created objects always satisfy these exclusiveness constraints.

Second, during the lifetime of person objects, it is possible that certain roles are dynamically added thereby possibly violating one of the exclusiveness constraints. Therefore we provide a mechanism that automatically checks whether adding a new role is allowed based on the roles the person object already performs. When an illegal role is added dynamically, SelfSync automatically generates a warning.

Lifetime Dependency The second case of interactive prototyping is to create one new stand-alone role object, for example a manager object. During its lifetime it is likely that this object will - in one way or another - be associated with a person object (without state inheritance). Intuitively we can state that a role object is dependent of the person associated with it: when the person object is deleted, all its roles should follow. Therefore, when a person prototype is deleted, SelfSync automatically deletes all the role prototypes associated with it. Notice that in the first case of interactive prototyping where a person object inherits the state of its roles, this enforcement is trivial.

Discussion

In this section we discuss the consequences of our choice of object-oriented paradigm, inheritance mechanism and typing system.

Using prototypes merges the views of roles as dynamic classifications and as instances to be adjoined to the objects performing them. In class-based languages, a role has to be either a class or an instance, while with prototypes all roles – prototypes as well as clones – are objects. In this way the same role object can be used as both a modeled entity and a run-time implementation object.

With delegation control remains in the receiver that forwards method lookup. In this way, messages sent to an object that performs roles are found in one of the traits of the roles it inherits at run-time (i.e. dynamic inheritance). Next the method is called in the context of the receiver. With (static) inheritance used in class-based languages an object can only perform the role it is assigned at creation time.

Next to prototypes and delegation, dynamic typing is a third factor that facilitates role modeling: objects can start or stop performing an arbitrary number of roles at any point in time. In statically typed languages it is necessary at creation time to know which roles an object will perform at run-time.

Evaluation

Steimann (Steimann 2000) defines 15 - sometimes conflicting - characteristics based on the leading literature on roles and categorizes a large set of existing role-oriented approaches to them. We compare the role modeling concept in SelfSync to the same characteristics:

1. *A role comes with its own properties and behaviour.* Yes, SelfSync allows roles to be modeled as stand-alone entities that are automatically mapped to a prototype and a corresponding traits object in Self.
2. *Roles depend on relationships.* Yes, a modeled role is linked to the object performing the role, through a specific role relation added to the EER model in SelfSync.
3. *An object may play different roles simultaneously*. Yes, see the warped hierarchies concept.
4. *An object may play the same role several times, simultaneously.* Possibly, SelfSync can easily be extended with an aliasing mechanism, in such a way that state inheritance of the same role object results in different states. For example, in our example, copying-down `budget1` and `budget2` when a person performs twice the role of manager. Alternatively the state of the roles can be copied into different slots of person implying that the homonymous attributes are accessed with a prefix, e.g. `manager1 budget`.
5. *An object may acquire and abandon roles dynamically.* Yes, see the warped hierarchies concept.
6. *The sequence in which roles may be acquired and relinquished can be subject to restrictions.* Yes, SelfSync enforces constraints, possibly about allowed role combinations.
7. *Objects of unrelated types can play the same role.* Yes this is possible in SelfSync.
8. *Roles can play roles.* Yes, this is possible and in a straightforward manner, as opposed to class-based languages where this characteristic is complicated by the distinction between classes and instances. In SelfSync all roles are objects.
9. *A role can be transferred from one object to another.* Yes, SelfSync automatically adds the state and the behavior of a role into the object that performs the role. This state and behavior can be removed and transferred automatically to another object.
10. *The state of an object can be role-specific, suggesting that each role played by an object should be viewed as a separate instance of the object.* No, SelfSync gathers the state of all roles in the same object.
11. *Features of an object can be role-specific: attributes and behaviour of an object may be overloaded on a by-role basis.* Yes, we can address specific behavior of a certain role due to dynamic parent modification. But we cannot overload attributes when we copy all attributes in the same object. Alternatively we can copy the overloaded attributes and methods in separate slots, see item 4.
12. *Roles restrict access.* No, also caused by the fact that Self's encapsulation is not enforced: visibility declarations have merely a documentational purpose.
13. *Different roles may share structure and behaviour.* Yes, we can model generalization between different roles in the EER model of SelfSync.
14. *An object and its roles share identity : "a role is a mask that an object can wear".* Yes, see the warped hierarchies concept.
15. *An object and its roles have different identities, related to the counting problem.* This is not covered in our approach.

As shown, SelfSync's warped hierarchies implement a full-fledged role modeling concept. Its expressiveness is comparable to most other role-oriented approaches. We succeeded in integrating a new role modeling concept in our object-oriented prototype-based modeling environment SelfSync that moreover provides automatic support for the mapping between roles at the conceptual level and their corresponding implementation, without suffering from the complications caused by the subtype-supertype problem. Hence, SelfSync is a round-trip engineering tool in which implementation objects as well as their corresponding EER diagram are continuously synchronized by a bidirectional active link, even when these objects are the subjects of dynamically changing roles.

Related Work

There exist various approaches that handle the paradoxical situation that roles are both super- and subtypes. For an in-depth discussion and more related approaches we refer to (Steimann 2000). We summarize the four approaches that are most relevant to our work.

The category concept (Elmasri, Weeldreyer, & Hevner 1985) concept is defined as the subset of the union of a number of roles (types). As in our approach the Entity-Relationship diagram was extended: relationships are not defined on entity types, but on categories.

In (Bock & Odell 1998) roles are considered temporal specializations: statically, a manager is a specialization of a person. However, when a particular person object becomes a manager, its type is changed from person to the subtype employee thereby inheriting all aspects of its new role. In this way reversed specializations, similar to warped hierarchies, are realized temporarily.

(Snoeck & Dedene 1996) also separate between static and dynamic type hierarchies: state sharing, behaviour sharing, as in Self, and subset hierarchies are combined into a new specialization modeling concept.

In (Jodlowski *et al.* 2004) delegation is used to implement dynamic roles that "import" state and behavior from their parent objects.

The role modeling concepts in the approaches described above provide suitable alternatives for warped hierarchies. However, to the best of our knowledge, none of them is integrated in an object-oriented modeling environment that supports automatic synchronization between the modeled roles and a corresponding implementation.

Conclusion

Role modeling is a specific instance of the subtype-supertype paradox where the state of objects is more general while the behavior of these objects is more specific. We were able to map this subtype-supertype problem to warped hierarchies in the language Self, thanks to the separation between state and behavior inheritance, and dynamic parent modification.

We extended our prototype-based round-trip engineering environment SelfSync with a role modeling concept at the level of the Extended Entity-Relationship model. In this way modeling roles automatically results in a corresponding implementation of warped hierarchies. On these implementation objects we enforce constraints about allowed role combinations and life-time dependencies of role objects.

References

Bock, C., and Odell, J. 1998. A more complete model of relations and their implementation: Roles. *JOOP* 11(2):51–54.

Burger, T. 2005. Formalism for the systems with roles. In Stefan, J., ed., *ISIM 2005 - Information System Implementation and Modeling. Eight International Conference, Hradec Nad Moravici, Czech Republic, April 2005, Proceedings*. MARQ.

Chambers, C.; Ungar, D.; Chang, B.-W.; and Holzle, U. 1991. Parents are shared parts of objects: Inheritance and encapsulation in SELF. *Lisp and Symbolic Computation* 4(3):0–.

Chen, P. P. 1976. The entity-relationship model - toward a unified view of data. *ACM Trans. Database Syst.* 1(1):9–36.

Cockburn, A. 1999. Constructive deconstruction of subtyping. *Humans and Technology (online article at http://alistair.cockburn.us/crystal/articles/cdos/constructive desconstructionofsubtyping.htm)*.

Elmasri, R.; Weeldreyer, J.; and Hevner, A. 1985. The category concept: an extension to the entity-relationship model. *Data Knowl. Eng.* 1(1):75–116.

Fowler, M., and Scott, K. 2000. *UML distilled (2nd ed.): a brief guide to the standard object modeling language*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Fowler, M. 1997. Dealing with roles. Technical report, Department of Computer Science, Washington University.

Goldberg, A., and Robson, D. 1983. *Smalltalk-80: the language and its implementation*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Jodlowski, A.; Habela, P.; Plodzien, J.; and Subieta, K. 2004. Dynamic object roles – adjusting the notion for flexible modeling. In *IDEAS*, 449–456.

Lieberman, H. 1986. Using prototypical objects to implement shared behavior in object-oriented systems. In Meyrowitz, N., ed., *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 21-11, 214–223. New York, NY: ACM Press.

Paesschen, E. V.; D'Hondt, M.; and Meuter, W. D. 2005. Rapid prototyping of extended entity relationship models. In Stefan, J., ed., *ISIM 2005 - Information System Implementation and Modeling. Eight International Conference, Hradec Nad Moravici, Czech Republic, April 2005, Proceedings*, 194–209. MARQ.

Paesschen, E. V.; Meuter, W. D.; and D'Hondt, T. 2004. Domain modeling in self yields warped hierarchies. In Malenfant, J., and Ostvold, B. M., eds., *ECOOP 2004 Workshop Reader: ECOOP 2004 Workshops, Oslo, Norway, June 14-18, 2004, Final Reports*, volume 3344 of *Lecture Notes in Computer Science*, p. 101. Springer-Verlag.

Paesschen, E. V.; Meuter, W. D.; and D'Hondt, M. 2005. Selfsync: a dynamic round-trip engineering environment. In *Proceedings of the ACM/IEEE 8th International Conference on Model-Driven Engineering Languages and Systems (MoDELS'05), October 2-7, Montego Bay, Jamaica*.

Self-website. 2003. Website: http://research.sun.com/self/.

Smith, R. B., and Ungar, D. 1995. Programming as an experience: The inspiration for self. *Lecture Notes in Computer Science* 952:303–??

Snoeck, M., and Dedene, G. 1996. Generalization/specialization and role in object oriented conceptual modeling. *Data Knowl. Eng.* 19(2):171–195.

Steimann, F. 2000. A radical revision of UML's role concept. In Evans, A.; Kent, S.; and Selic, B., eds., *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, 194–209. Springer.

Ungar, D., and Smith, R. B. 1987. Self: The power of simplicity. In *OOPSLA*, 227–242.