Domain Modeling in Self Yields Warped Hierarchies

Ellen Van Paesschen - Wolfgang De Meuter - Theo D'Hondt

Programming Technology Laboratory, Vrije Universiteit Brussel
Pleinlaan 2, 1050 Brussel, Belgium

ABSTRACT

Domain modeling can result in a hierarchical set-up in which the modeled entities follow the standard hierarchical taxonomies while the proper execution of the corresponding code demands the reversed hierarchy. Modeling roles and the identity problem are typical cases of these "warped" hierarchies, which are difficult to implement in class-based languages. In the prototype-based language Self, entities are modeled into hierarchies of traits, supporting multiple inheritance, dynamic parent sharing and copy-down techniques. This powerful cocktail of features allows building warped hierarchies in a straightforward and natural manner.

Keywords: Prototypes, multiple inheritance, dynamic delegation, traits, parent sharing, roles

1. INTRODUCTION

Since the advent of Simula, object-oriented languages are promoted as programming languages that facilitate modeling the real world and make it possible to create taxonomies from the entities that surround us. Indeed, many aspects of a problem domain are easily modeled in object-oriented languages: usually, the modeled entities correspond to an object or a class and taxonomies of entities give rise to class-hierarchies. This way of thinking is pretty straightforwardly applied in a prototype-based language like Self as well. The only difference is that one will replace classes and their hierarchies by traits objects and their hierarchies.

Nevertheless, there exists a significant hiatus in this story. During such a modeling process in Self, we experienced a number of occasions where this straight-forwarded approach gives rise to a hierarchical set-up in which the entities follow the standard hierarchical taxonomies but in which the corresponding code demands exactly the reverse version of this hierarchy. We discovered the existence of such *warped hierarchies* while doing *role modeling*, an activity which is known to be far from easy in a class-based language.[4] They also showed up in relation to fundamental and philosophical shortcomings: e.g. a mathematician would consider a circle as a special kind of ellipse, where both axes are equal, while an object-oriented modeler would rather define an ellipse as a descendant of circle.

Warped hierarchies cannot be implemented in class-based languages. However, this is perfectly feasible in Self, thanks to multiple inheritance, parent sharing and copy-down techniques. We will illustrate this using the circles/ellipses example and the role modeling case.

2. PROTOTYPE-BASED LANGUAGES

2.1. In General

In general, prototype-based languages (PBLs) can be considered object-oriented languages without classes. The most interesting feautures of a PBL are *creation ex nihilo*, *cloning*, *dynamic inheritance modification*, *delegation with late binding of self*, *dynamic parent modification*, and *traits objects**. Many PBLs have been designed in research labs. Examples are Self,[7] Agora,[2] Kevo[9] and NewtonScript.[8] A taxonomy can be found in.[3] We will elaborate on the PBL Self, since it is a textbook example of a PBL and moreover, includes a mature programming environment.

Send e-mail correspondence to {evpaessc,wdmeuter,tjdhondt}@vub.ac.be

*To avoid copying behavior every time an object is cloned, the SELF-group[11] introduced **traits objects**: storing the shared behavior in an object and let the cloned objects inherit from it, i.e. a kind of class-based programming in a PBL

2.2. Self

Self is closely related to the syntax and semantics of Smalltalk[5] but Self has no classes. Objects in Self are *created ex-nihilo* by putting slot names (together with a possible initial filler value for that slot) between vertical bars, separated by dots. The following code, for example, creates an ex-nihilo `myPoint`† object:

```
myPoint: (|parent* = traits clonable. x <- 3. y <- 4.
           addPoint: point = ((copy x: x + point x)
           y: y + point y)|)
```

Self visualizes its objects with outliners, cfr. figure 1. A slot marked with an asterisk is a parent slot and makes

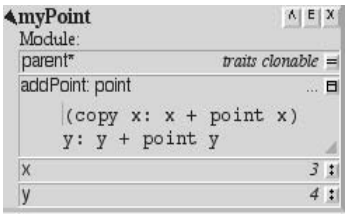

Figure 1. The self-contained `myPoint` object combines data and behavior

the child inherit all the slots of the parent slot. In this way, `myPoint` inherits (its behavior) from the *traits object* `clonable`‡, and has two data slots containing an `x` and a `y` coordinate. The remaining method slot contains a method for adding two points, by *cloning* `point` and initialize it with the added `x` and `y` coordinates.

Self implements a *delegation* mechanism that respects the *late binding of* `self`. Next to *dynamic inheritance and parent modification*, this delegation mechanism also supports *parent sharing*, i.e. when two or more child objects share the same parent object. This kind of sharing is typical for all PBLs. *Child sharing* (multiple inheritance), on the other hand, when two or more parent objects share the same child object, is a specific feature of Self. When modeling knowledge these two inheritance features are constantly combined.

3. MULTIPLE INHERITANCE IN SELF

When modeling a data type in Self, the data (specific for each "instance" of this data type) is contained in a prototype while the behavior (shared by all objects of this data type) is typically gathered in a traits object. All prototypes inherit their behavior from the traits object, which in his turn often inherits from `traits clonable`:

```
traits myPoint = (|parent* = traits clonable.
                 addPoint: point = ((copy x: x + point x)
                 y: y + point y)|)

myPoint = (|parent* = traits myPoint. x .  y|)
```

The graphical representation is illustrated in figure 2. To obtain *a* point, we clone the `myPoint` prototype and set the x and y coordinates.

```
(myPoint copy x: 1) y: 2.
(myPoint copy x: 3) y: 4.
```

†We use the name `myPoint` since Self already implements a `point object`

‡Most concrete not-unique objects in the SELF world are descendants of the top-level traits object `traits clonable`.

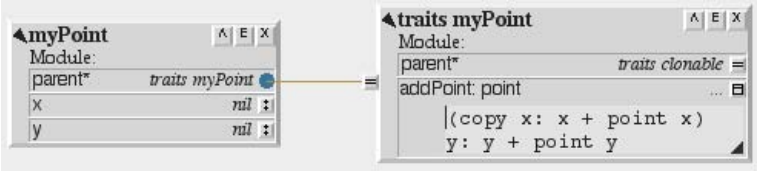

Figure 2. The `myPoint` prototype inherits its behavior from `traits myPoint`

Both points now share the `traits myPoint` object since they both contain a copy of the `parent*` pointer of the prototypical `myPoint`, i.e. the most common form of parent sharing.

When we want to create for example a *coloured* point, data and behavior are to be inherited from a normal point. First, a prototypical `coloured point` is created that inherits its behavior from a corresponding `traits coloured point` object. Naturally, the `traits coloured point` inherit behavior from the `traits myPoint`, since the behavior of a coloured point will be a specialization of a normal point's behavior. On the other hand, the `coloured point` prototype can inherit the coordinates of the normal `myPoint`, and extend them with an extra slot to contain the colour, see figure 3. Remark that this multiple inheritance structure is a diamond. Imagine a

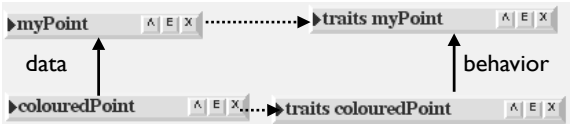

Figure 3. `colouredPoint` inherits data and behavior from `MyPoint`

method `m` in `traits myPoint` that is overridden in `traits coloured point`. When we now send the message `m` to a `coloured point` we get a *name collision*: the method lookup algorithm finds `m` in `traits coloured point` (overriding method) but also in `traits myPoint` (original method) via the data inheritance link with `myPoint`. The early version of Self solved this ambiguity with obscure language mechanisms like *prioritized* parents or the *tie-breaker sender path* rule, which proved to be rather unsatisfying. In the current version of Self we have to resolve ambiguous methods manually by adding a *directed resend* in `coloured Point`. Calling `m = (traits colouredPoint.m)` would invoke the overridden method while `m = (traits myPoint.m)` would return the original method. But then we violate the principle of traits-based inheritance, since we add shared behavior in a prototype in stead of into the corresponding traits object.

Self avoids this problem by performing a `copy-down` of the `myPoint` prototype: this mechanism for data inheritance copies (some of) the slots of the receiver into a new object, ensuring that changes (adding/removing slots) to the receiver are propagated to all copied-down children. Next, we override the `parent*` pointer with the `traits colouredPoint` object. In this way, `colouredPoint` inherits all the data of `point` except for its parent: this implies that there are no name collision when `traits coloured point` override methods of `traits myPoint` In fact, `copy-down` allows a kind of class-based programming: copy-down can be considered as creating a subclass. The `colouredPoint` and `myPoint` inheritance structure is illustrated in figure 4. The complete Self code for the literal point objects can be found in Appendix A.

4. WARPED SELF INHERITANCE HIERARCHIES

It is our experience that modeling domains in Self often results in a rather classical object organisation, differing little from a class-based set-up. However, we found two examples where the transition from domain model notation to code notation gives rise to warped inheritance hierarchies, namely the *identity problem* of circles and ellipses and *role modeling*.

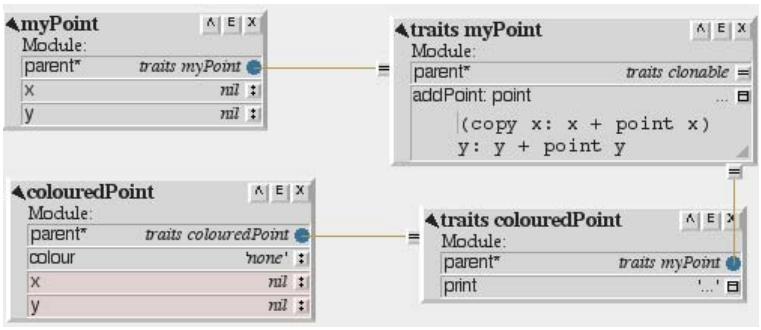

Figure 4. `colouredPoint` inherits data from `myPoint`, `traits colouredPoint` inherits behavior from `traits myPoint`

4.1. Is a circle an ellipse?

Although not many OO-programmers are aware of it, from the *real world* (domain model) point-of-view, a circle really *is-a* kind of ellipse (with major semi-axis a = minor semi-axis b = radius) and thus the code should see circles as specializations of ellipses. In a class-based language the circle type *can* be implemented as a subclass of the ellipse type, resulting in inefficient code since circle will not use all instance variables inherited from ellipse. The difficulty is mainly caused by the fact that the data of circle is less specialized than ellipse's data while the behavior of circle is more specialized than ellipse's behavior. An extra problem in this context, is that circles can receive messages intended for ellipses, transforming them dynamically into ellipses, and vice versa. E.g. 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"!

Thanks to the separation of data and behavior inheritance, and dynamic modification of parents, Self allows us to model the identity example with warped hierarchies. We let `ellipse` inherit data from `circle` (since it extends it with an extra slot for a major semi-axis value), while `traits circle` inherit from `traits ellipse`, see figure 5. As mentioned in the previous section, the diamond set-up can be broken by defining `ellipse` as

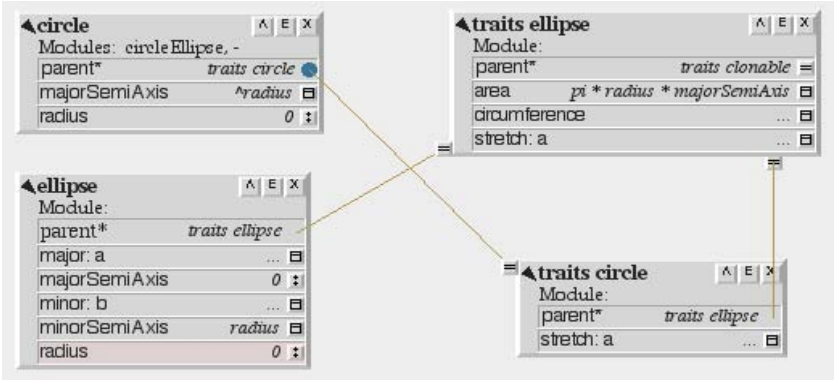

Figure 5. Warped hierarchy of `circle` and `ellipse`

a `copy-down` of `circle` and assigning the `parent*` pointer to `traits ellipse`. Thanks to the late binding of the `self` variable, the correct data is accessed when executing methods (e.g, `area`, `circumference`) - and thus polymorphism is ensured. When a circle is stretched to an ellipse, we add all the slots of the `ellipse` prototype into the circle, thereby overriding the `parent*` pointer from `traits circle` to `traits ellipse`. Vice versa, an ellipse whose major semi-axis is stretched to the same value as its minor semi-axis, becomes a circle, by removing all slots that were not copied-down from circle and by overriding the `parent*` pointer from `traits ellipse` to `traits circle`. In this way, objects seem to change the prototypes they were cloned from dynamically.

4.2. Role Modeling

The roles a person can perform are on one hand subtypes of a person: e.g. an engineer *is-a* kind of person. On the other hand, when a role type inherits from person, how will we - in a class-based language - model that this person can perform other roles? E.g. when both engineer and manager are subclassed from person and we want to model a person that is both manager and engineer. When we instantiate the manager class, the engineer class will be invisible and vice versa. Creating combination classes is not feasible: imagine the difficulties when a person can change dynamically between a large set of roles[4]! Alternatively, roles are often modeled with aggregation: 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.[4]

The real difference with the previous example lies in the fact that roles can be added or removed *dynamically*, and that a person can have *multiple* roles implementing the *same* method. Simply warping the data hierarchy between a person and its roles will not be sufficient.

Therefore, we implemented `receiver createDataparent:parent`[§] as a reverse of the `copy-down` method: in stead of copying down the data from the receiver into a new child object, the data of the parent is copied down into the receiver. *We now create dynamically parents in stead of children.* Due to the dynamic character of the derived types, we also provided a `receiver remove Dataparent:parent` that removes all copied-down data from the receiver.

Consider a `person` prototype that inherits from `traits person`, and a set of role prototypes (e.g. `manager`, `engineer`) inheriting from their traits (e.g. `traits manager`, `traits engineer`), that in their turn all inherit from `traits person`. A `person` that dynamically starts performing a role is implemented by dynamically adding this role's prototype as a data parent to the `person` prototype. Next, we remove[¶] the person's `parent*` link to `traits person` since these are already inherited via the role data parent. Due to the multiple inheritance in Self we can add as many roles as we like, cfr. figure 6. When a `person` dynamically stops performing a role, we

Figure 6. Warped hierarchy of `person` and two roles

remove the data parent. When there are no more role data parent we make the `traits person` visible again. In fact, the *desired behavior is added or removed dynamically.*

To ensure polymorphism we need to intercept the dynamic diamond that is implemented by a person that inherits from two role data parents whose traits both inherit from `traits person`. More specifically, when two roles of a person both override the same method in their traits, sending the corresponding message to person will cause a VM ambiguity error. Our approach depends on the way the methods should be combined from the view point of person. E.g. when we send the message `pay` to `person`, he should get payed for *all* the roles he performs . Therefore, we implemented `delegateMethod:selector` that *sequentially resends the message to all the data parents*, i.e. the roles, of person. However, it is possible that we only want to invoke a specific method, defined in the role in whose context we currently see the person. E.g. when we send the message `lunch` to `person`, she

[§]Meta-programming methods heavily use the technique of Self mirrors: an object is reflected on by means of a mirror; manipulating the mirror results in manipulating the object

[¶]We simply make a plain slot from this parent slot

might simulate the specific behavior to have lunch with her best friend and not, for example, with her boss and some clients of the company she works for. In that case, we suggest to turn on/off the parent visibility of the desired behavior, i.e. (temporarily) changing the parent slots, that point to the traits of currently non-desired roles, to normal slots. In this way we maintain the illusion that we are dealing with one person performing various roles.

5. CONCLUSION

PBLs, especially Self, are a suitable medium for modeling knowledge, with powerful inheritance mechanisms which outrank the class-based ones. We experienced the phenomena of warped hierarchies and implemented a technique, that profits from the separated data and behavior inheritance in Self, and intercepts the dangers of multiple inheritance in this context. We have the "gut feelling" that these warped hierarchies are one of the fundamental "missing links" in the transformation process that leads domain models to code.

REFERENCES

1. G. Blashek, *Object-Oriented Programming with Prototypes.*, Springer Verlag, 1994
2. W. De Meuter, *Agora: The Scheme of Object-Orientation, or, the Simplest MOP in the World.* In J. Noble, A. Taivalsaari, I. Moore, eds.: *Prototype-based Programming: Concepts, Languages and Applications*, 1998
3. C. Dony, J. Malenfant, D. Bardou,"Classifying Prototype-based Programming Languages.", In J. Noble, A. Taivalsaari, I. Moore, eds.: *Prototype-based Programming: Concepts, Languages and Applications*, 1998
4. M. Fowler, *Dealing with Roles.*, Collected papers from the PLoP '97 and EuroPLoP '97 Conference, Technical Report wucs-97-34, Washington University Department of Computer Science; http://www2.awl.com/cseng/titles/0-201-89542-0/apsupp/roles2-1.html, 1997
5. A. Goldberg, D. Robson, *Smalltalk-80: The Language and Its Implementation.*, Addison-Wesley, 1983
6. H. Lieberman, "Using prototypical objects to implement shared behavior in object oriented systems.", In N. Meyrowitz, ed.: *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA).*, Volume 22, 214 - 223, 1987
7. R. Smith, D., Ungar, "Programming as an Experience: The inspiration for Self.", In J. Noble, A. Taivalsaari, I. Moore, eds.: *Prototype-based Programming: Concepts, Languages and Applications*, 1998
8. W. Smith, "NewtonScript: Prototypes on the Palm.", In J. Noble, A. Taivalsaari, I. Moore, eds.: *Prototype-based Programming: Concepts, Languages and Applications*, 1998
9. A. Taivalsaari, *A Critical View of Inheritance and Reusability in Object-oriented Programming.* PhD thesis, University of Jyvaskyla, Finland, 1993
10. D. Ungar, R. Smith, *Self: The Power of Simplicity.*, In: Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Volume 22, ACM Press, 1987
11. D. Ungar, C. Chambers, B. Chang, U. Holzle, *Organizing programs without classes.*, Lisp and Symbolic Computation 4, 223 - 242, 1991
12. *Self Home Page*, http://research.sun.com/research/self/.

APPENDIX A. SELF CODE

A.1. myPoint objects

```
globals _AddSlots: (|myPoint|).
traits _AddSlots:(|myPoint|).

myPoint: (|parent* = traits clonable. x <- 3. y <- 4.
           addPoint: point = ((copy x: x + point x)
           y: y + point y)|).

traits myPoint: (|parent* = traits clonable.
                 addPoint: point = ((copy x: x + point x)
```

```
                y: y + point y)|).

myPoint: (|parent* = traits myPoint. x .  y|).
```

A.2. colouredPoint objects

```
globals _AddSlots: (|colouredPoint|).
traits _AddSlots:(|colouredPoint|).

traits colouredPoint: (|parent* = traits myPoint.
                        print = ('...')|).

colouredPoint: (((myPoint _Mirror) createSubclass) reflectee)
                      _AddSlots: (|parent* = traits colouredPoint.
                                  colour <- 'none'|).
```