Intersecting Classes and Prototypes

Wolfgang De Meuter, Theo D'Hondt, and Jessie Dedecker
{wdmeuter,tjdhondt,jededeck}@vub.ac.be

Programming Technology Lab
Department of Computer Science
Vrije Universiteit Brussel
Pleinlaan 2 - 1050 Brussels - Belgium
Fax: +32 2 629 35 25

Abstract. The object-oriented programming language design space consists of class-based and prototype-based languages. Both language families have been shown to posses many advantages but also several disadvantages with respect to software construction. Hybrid languages featuring both prototype-based and class-based mechanisms have been proposed as a solution. Unfortunately these languages not only unify the advantages but also the disadvantages of both families. In this paper we propose a more intersectional point of view and propose a language that inherits the advantages but shuns the disadvantages of both families.

1 Introduction

Object-oriented programming languages (OOPLs) can be classified in two big language families: the prototype-based languages (PBLs) and the class-based-languages (CBLs). While CBLs are related to the Aristotelian point of view that the world we live in consists of ideal ideas and instances of those ideal ideas, PBLs are a reflection of Wittgenstein's philosophy that it is more correct to think about our world as consisting of objects that have a "family resemblance" with "representative prototypes" [1].

In CBLs, classes (representing ideal ideas) contain the knowledge of a program. Objects are created by *instantiating* a class and classes can share behaviour by means of *inheritance*. In PBLs on the other hand, objects are created *ex-nihilo* (e.g. by putting a number of fields and methods between parenthesis), or by *cloning* other objects. Sharing can be accomplished by establishing an "inheritance relation" between objects. This relation is usually referred to as *delegation*. Examples of CBLs are Java [2], C++ [3] and Smalltalk [4]. Examples of PBLs are Self [5], Agora [6], Kevo [7] and NewtonScript [8].

Some people think that the controversy between CBLs and PBLs has become obsolete with the advent of Java as "the" standard object-oriented language. However, although academic interest in object-oriented language design *did* decrease right after Java's popularization, we believe this was temporal. Recent workshops such as "Feyerabend" at OOPSLA'01, OOPSLA'02, ECOOP'02 and "The Inheritance workshop" at ECOOP'02 show that language design is still

alive and kicking. More importantly, we believe that PBLs are back into business. Proof of this are recent publications that show the relevance of prototypes in components [9], in agent technology [10] and the fact that Self has been released for the new OS for the Mac (see [11]). As argued by Cardelli in the mid nineties (with Obliq [12]), prototypes will gain interest especially in the field of mobile computing where agents can roam networks in an unpredictable way.

We begin this extended abstract by reviewing (see section 2) the major drawbacks researchers have attributed to CBLs, and why PBLs are an answer to them. However, as we will see in section 2.3, PBLs also have a number of disadvantages. One reaction to avoid the shortcomings one gets by choosing a language from either family is to design new languages that contain both class-based and prototype-based characteristics. Such languages are often called *hybrid languages* such as Hybrid [13] and more recently JavaScript 2.0 [14][1]. Unfortunately, these hybrid languages are based on a union between the language feature sets of CBLs and PBLs. As a result, they unify advantages, but also the disadvantages related to the families. Furthermore, they tend to be large languages with non trivial combinations between prototype-based and class-based features.

This paper presents a tiny language, called *Pic%* combining the prototype and class-based characteristics in an intersection, thereby combining their advantages but shunning their drawbacks.

2 PBLs vs. CBLs

PBLs were invented in the late seventies by the AI community[2] for knowledge representation purposes. But we had to wait for Lieberman's influential paper [15] in order for PBLs to make it to the object-oriented community. Many PBLs have been designed in research labs. Examples are Self [16], Agora [6], Kevo [7] and NewtonScript [8]. A taxonomy can be found in [17]. In section 2.2 we summarize how PBLs solve many problems associated with CBLs. Section 2.3 lists a number of drawbacks of PBLs related to the absence of classes. First we briefly review the most important characteristics of PBLs.

2.1 Prototype-based languages

A PBL is often defined as "an OOPL without classes" and to a large extent this is true. In PBLs, objects are created *ex-nihilo* (i.e. out of the blue, by listing a number of fields and methods) and/or by *cloning* an existing object. In the latter case, the cloned object is called the *prototype* of the clone.

Whereas CBL allow classes to share each others fields and methods by means of inheritance, this feature is absent in PBL. PBLs achieve the same effect by

[1] JavaScript1.x is one of the most widely used PBLs. However its PBL features are poorly documented and the language is a very poor example of language design. This is even worse with JavaScript2.0 which was further enriched with classes.

[2] In the AI community, PBLs were known as *frame-based languages*.

a (highly dynamic) mechanism usually called *delegation* or *object-based inheritance*. The idea is that one object can dynamically decide another object to be its parent. Every message that cannot be handled by the object itself is (automatically) delegated to the parent object. An important remark to make here is that this happens (exactly as between inheriting classes in a CBL) with *late binding of self* as illustrated in figure 1a. Messages send to the `self` (or `this`) pseudo variable in the parent will "come back" to the object that originally received the message. This is a crucial feature of delegation as originally defined by Lieberman [15] and most PBLs implement it. The "delegation" described in the famous design patterns book [18] is depicted in 1b. This might be called "message forwarding" but it does not meet the definition of delegation in PBLs.

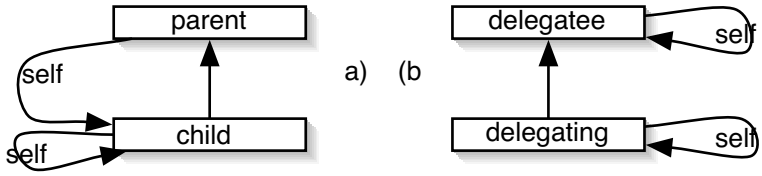

Fig. 1. Delegation vs. Messsage Forwarding

A problem with "naive prototype-based thinking" is that, when cloning objects, not only their state but also their behaviour gets copied. This means that an object with 10 methods will give rise to 50 methods when cloned 5 times, or at least 50 pointers all pointing to the same methods. A solution to this problem invented by the Self-group [19] is a programming technique called *traits objects*. The idea is to store the 10 methods in one object (called a traits object) and to let the 5 clones inherit them from that object. All 5 objects contain their own state variables but share the behaviour of the traits in the shared parent.

Although traits objects are but a programming idiom and are not an enforced language feature, we have to face the fact that the traits technique *actually* boils down to writing class-based programs in a PBL: the traits objects play the role of classes and the instances refer to the 'class' by means of a 'parent pointer' instead of an 'instance-of pointer'. Aside from some technical details this is the same as class-based programming. Therefore, Dony [17] argues that it is better to speak about object-centred programming instead of prototype-based programming. Object-centred programming refers to the act of writing programs that do not use these "class-centered techniques" too much but instead put the object as a "design entity" in a central position again.

2.2 Drawbacks of CBLs and Class-centered Programming

Over the years, several researchers have been drawing our attention to the numerous drawbacks of CBLs, which we can divide into what we might call *language*

theoretical drawbacks and *philosophical drawbacks*. The philosophical drawbacks are not really a consequence of particular language features, but rather related to the way of thinking CBLs impose. They basically boil down to the fact that modeling the world with a statically determined classification doesn't work in general, and, that humans prefer to think in terms of concrete objects rather than in terms of abstractions. We refer to Taivalsaari's excellent overview article [1] for the philosophical background of PBLs.. The language theoretical drawbacks of CBL are drawbacks related to language design issues and interest us more. We merely summarize the analysis of [1]:

- A PBL system really has nothing but objects, also on the second sight. In uniform CBLs where everything is an object, one easily ends up with very complex concepts such as meta classes and "the double hierarchy" in Smalltalk. In Java, classes are objects that belong to the class `Class` and `Class` is an object that is its own class.
- Bugs in constructors can lead to unmeaningfully initialized objects. In PBLs, fields in clones always have a meaningful initial value, namely the value of the field of the object cloned.
- In CBLs it is not possible to individualize the behaviour of objects unless one makes extra subclasses. E.g. in Self, we can put a "halt"-statement in a method of one object. This will only affect that object.
- PBLs support singleton objects (such as True, False and singletons [18]) for which a class needs to be constructed in CBLs. Smalltalk has the classes `True` and `False` with `True` and `False` as only instances.
- In PBLs two or more objects can inherit (by delegation) from the same object such that changes in the parent done by one child are also "felt" by the other children. This powerful mechanism, called *parent sharing*, allows to define views on objects. Simulating this in a CBL leads to clumsy indirections.
- Classes play no less than eleven roles [20], a.o. object generation, description of representation and behaviour of objects, taxonomization of objects, code reuse, modularity, encapsulation and visibility definition and types definition. Of course by removing classes a lot of these roles will be taken over by objects. But in that case we know objects fullfil *all* the roles without arbitrarily deferring a few of them to classes.
- Since all object manipulation is totally decoupled from classes, a PBL allows for re-introducing classes for the sole purpose of classification. In StripeTalk [21] e.g., a programmer can assign "attributes" (called stripes) to objects and classify objects based on the stripes.

2.3 Drawbacks of PBLs and Object-centred Programming

PBLs have been shown to be a solution to most of the problems outlined in the previous section. However, PBLs have their problems as well, most of them related to the absence of a feature playing a role previously played by classes :

- The construction of certain objects requires a certain **construction plan** to be executed (e.g. building up a GUI). In CBLs, it is possible to formalize

this plan as a class constructor. In PBLs there is no such thing: objects are created by cloning other objects and it requires a lot of discipline from programmers to make sure certain procedures are followed.
- The **prototype corruption problem** was put forward by Gunther Blashek [22]. Prototypes (e.g. the empty `String` prototype) can be inadvertently modified which might affect future clones. This can lead to subtle bugs.
- PBLs suffer from a **re-entrancy problem**. As explained in section 2, naively copying prototypes would result in (pointers to) methods being needlessly copied[3]. As explained, the problem can be circumvented using the traits technique, but this is sometimes problematic as the traits technique heavily interferes with the inheritance hierarchy. In Self this doesn't pose real problems because of Self's multiple inheritance (which is also problematic, see [16]): objects inherit from their "real" parent *and* from their traits parent. In PBLs without multiple inheritance this is not possible.
- Some concepts are **inherently abstract**. E.g. in order to describe a stack one *has* to go to the abstract level. When writing the code for push and pop, one writes code for all possible stacks (empty stacks, full stacks,...) and hence one is *by definition* working on the "class" level of abstraction. The problem is that "stack" is an inherently abstract concept. Rather than talking about one particular stack, the code talks about stacks in general.

All these problems can be solved by re-introducing classes but this automatically implies re-introducing the problems outlined in section 2.2. But they can also be solved without classes: in the following sections we introduce Pic%[4], a tiny OOPL which we consider to be an intersection between CBL and PBL. Pic% is an object-oriented extension of a procedural language called Pico. We first sketch Pico and in section 4 we introduce Pic%. Section 5 explains why Pic% is a good intersection between CBLs and PBLs.

3 Pico: A Short Overview

Pico was designed (a.o.) as a vehicle for teaching language concepts and interpreters to our sophomores. A full explanation of Pico is beyond the scope of this text but can be found on the internet [23]. The following implementation of the quicksort algorithm gives a general flavor of the language:

```
QuickSort(V, Low, High):
  { Left: Low;
    Right: High;
    Pivot: V[(Left + Right) // 2];
    until(Left > Right,
          { while(V[Left] < Pivot, Left:= Left+1);
            while(V[Right] > Pivot, Right:= Right-1);
```

[3] In CBLs these methods are store in a method table in the class.
[4] Read: Pic-oo

```
            if(not(Left > Right),
               { Swap(V, Left, Right);
                 Left:= Left+1; Right:= Right-1 },
               false) });
      if(Low < Right, QuickSort(V, Low, Right), false);
      if(High > Left, QuickSort(V, Left, High), false) }
```

Pico is a dynamically interpreted language that is heavily based on Scheme [24]. All Pico values are first class: basic values, functions and tables can be passed around as arguments, returned from functions, act as value for an assignment and so on. The major differences with Scheme are

- Pico uses ordinary function application notation and infix operator notation.
- Since Pico is equipped with an efficient memory manager, we no longer have to restrict ourselves to the fixed-size storage model originally associated to Scheme. Hence, instead of dotted pairs, Pico features variable sized tables.
- Pico functions always carry a name.
- While Scheme always uses eager evaluation, Pico allows for the definition of functions of which some parameters use lazy evaluation. This removed the need for special forms, while still allowing for a uniform notation (see the usage of `if`).

A Pico expression is always evaluated in the context of an environment which is a linked list of name-value associations. Values can be numbers, texts, tables and functions. The environment is organized according to the well-known lexical scoping rules. Values no longer bound to a name in some environment get automatically garbage collected.

Table 1 shows the most frequently used syntactic constructions in Pico. New names are added to the 'current' environment using the colon notation. These names are associated to primitive values, tables or functions. This constitutes the second row of the table. Using names, indexing tables and calling functions make up the first row. The bottom row shows how variables are given a new value, how tables our updated and how functions can get a new body. Notice that the curly braces as used in the quicksort teaser are mere syntactic sugar that is replaced by the parser to a call to the (lazily evaluated) `begin` function.

Table 1. Pico Basic Syntax

reference	`nam`	`nam[exp]`	`nam(`$\mathtt{exp}_1$`, ... ,`$\mathtt{exp}_n$`)`
definition	`nam: exp`	`nam[exp1]: exp`$_2$	`nam(`$\mathtt{exp}_1$`, ... ,`$\mathtt{exp}_n$`): exp`
assignment	`nam:= exp`	`nam[`$\mathtt{exp}_1$`]:= exp`$_2$	`name(`$\mathtt{exp}_1$`, ... ,`$\mathtt{exp}_n$`):= exp`

4 Pic%, an Intersection of PBLs and CBLs

Pic% is a prototype-based extension of Pico. The concepts of Pic% where heavily influenced by our previous research on Agora [6] of which we "proved" in [25]

that it is an intersection of CBLs and PBLs in terms of denotational semantics. But Pic% is much simpler than Agora. The basic ideas we had to introduce in order to turn Pico into an OOPL are:

- The `capture()` primitive returns, at any point in the execution, a reference to the current environment. These captured environments are Pic%'s objects.
- The qualified dot operator is used to send messages. If `e` is a captured environment (i.e. an object), then `e.nam(`e_1`, ... ,`e_n`)` searches `nam` in that environment and invokes the function associated with it. The search starts, as always in Pico and Pic%, at the "bottom" of the environment and proceeds upward in the linked list of bindings. This corresponds to method lookup.

The following code excerpt shows how these features enable object-oriented programming (more about the double-colons later):

```
counter(n):
  { incr():: n:= n+1;
    decr():: n:=n-1;
    capture() }
```

Upon evaluation of this code, an expression of the kind `c:counter(10)` will create a new object with two methods `incr` and `decr` because the result from calling `counter` is a reference to the environment of execution that was built up to the point where `capture` is called. `counter` is a *constructor function* because each call generates an object, i.e. an extension of the lexical environment of the constructor[5]. Messages can be send to objects using expressions like `c.incr()`.

Inheritance and overriding in our model come for free as illustrated by the following code excerpt:

```
counter(n):
  { incr():: n:= n+1;
    decr():: n:=n-1;
    super: void;
    protect(limit)::
      { incr()::
          if(n=limit,error("overflow"),super.incr()) };
        decr()::
          if(n=-limit,error("underflow"),super.decr()) };
        capture() }
    super:= capture() }
```

Having constructed a counter object `c`, the message `c.protect(4)` will return an object whose parent is the receiver of the message, i.e. `c`. This is because objects are nothing but environments: upon entering `protect`, the 'current environment' is gradually extended with new increment and decrement methods and the 'new current environment' is captured again. Hence the result is an environment that

[5] In Scheme terminology, this is called a *closure*.

contains the four methods and the `super` variable. Inheritance and overriding come for free as Pico name lookup proceeds 'upwardly' in the environment. A method like `protect`, that generates extensions of its receiver, is called a *mixin-method*. Mixin-methods were a built-in language feature of Agora [6].

Although the model outlined so far forms an extremely simple PBL (that already solves a few of the problems of section 2.3 thanks to the constructor function technique), it also misses a few crucial things such as super sends with late binding of self and cloning. Furthermore the biggest problem of PBLs (the re-entrancy problem) remains unsolved unless we turn to the clumsy traits objects technique. The key to solving this problem lies in redesigning the Pico environment model in such a way that every environment consists of 2 parallel lists of bindings as shown in figure 2. One list is called the variable part (all the names defined with `:`) and the other list is called the constant part (all names defined with `::`). This gives us a very consistent object model as shown by the following arguments:

- We decided to take the constant part to be the interface of the object. Hence, messages sent with the dot operator will always be looked for in the constant part. Once a method is found, it is executed in the context of the entire object. Inside a method, the receiving object can be accessed by called `this`. Hence, at any point, `this()` returns the current object. "Function calls" that are not qualified are implicit sends to `this()` such that calling a constructor function like `counter(4)` is merely a self-send to the "root environment".
- Every object understands `clone`, a function standardly installed in the constant (i.e. public) part of the root environment. Hence, every object inherits `clone`. The entire point of objects being implemented as two parallel lists of bindings and of the constant list to be the public part, is that `clone` will never be copy the static part of its receiver. Hence, by definition, the `clone()` method makes a new object consisting of the constant part of its receiver and a clone of the variable part of its receiver.
- Finally, a new dot syntax without receiver is introduced. "Dotted messages" without a receiver are send to the immediate parent object. Hence, `.m()` corresponds to `super.m()` in Java. The difference between this technique and the one with the `super` variable shown above is that the receiverless dot does not change the meaning of `this()` such that we have super sends with late binding.

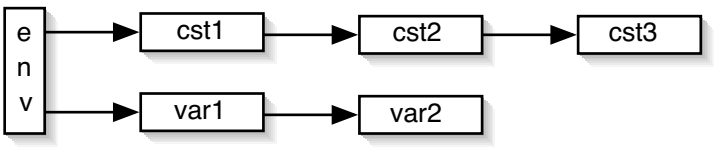

Fig. 2. Pic% Environment/Object Structure

Some Pic% Programming Patterns

– The following example shows the programming style favored by Pic%.

```
Stack(n):
  { T[n]: void;
    t: 0;
    empty():: t=0;
    full():: t=n;
    push(x)::
      { T[t:=t+1]:=x;
        this() };
    pop()::
      { x: T[t];
        t:=t-1; x };
    makeProtected()::
      { push(x)::
          if(full(),
            error("overflow"),
            .push(x) };
        pop()::
          if(empty(),
            error("underflow"),
            .pop()};
        capture() };
   capture() }
```

– A nice pattern in Pic% is the implementation strategy for singleton objects. This is easily accomplished by having constructors that override `clone()` so it does nothing and that replace themselves by their single generated object:

```
constructor(args):{
   ... ;
   clone():: this();
   constructor:=capture()}
```

– In [26] we explain a "feature" of Pic% that we did not touch upon in this paper, namely the first class manipulation of methods. This is particularly useful in a model like ours where all inheritance is accomplished by mixin-methods. Since this model requires all the potential extensions to reside in mixin-methods *in* objects, extension "from the outside" would not be possible if it wasn't for the possibility to assign (mixin-)methods.
– The fact that the constants of an object are never cloned give us a powerful language feature called *cloning families* inside the implementation of Kevo [7]. We believe cloning families might be a good abstraction to think about the notion of replicas in a distributed system. we are currently investigating this in a version of Pic% that uses objects as mobile agents [10].

5 Evaluation and Epilog

We now validate the claim that Pic% is a clean intersection of CBLs and PBLs. First, Pic% features nothing but objects. (Constructor) functions are mere methods in an object and calling a constructor function is an implicit send to `this()`. Hence, we did not reintroduce the problems associated to classes. Second, consider the drawbacks outlined in section 2.3 one by one:

- The **construction plan** problem was about the need for formalized construction plans. This need is clearly covered in Pic% by the constructor functions that generate objects using `capture()`. Constructor functions in Pic% are exactly the same as constructors in CBLs.
- The **prototype corruption** problem is solved by our proposal because constructor functions and overriding `clone` are two powerful techniques to *ensure* that fresh objects (constructed with a function or with `clone`) are always correctly initialized.
- A solution to the **re-entrancy problem** is the most important technical contribution Pic% makes to the field. By structuring objects as two dictionaries of which only one gets copied by the cloning operation, we make sure that all clones of the same prototype share the constant fields of that prototype. This "hidden traits" object does *not* interfere with the inheritance hierarchy in the way we discussed in section 2.3.
- The fact that some notions (like stacks) are **inherently abstract** is trivially covered because of the way constructor functions simulate classes.

Pic% has both PBL features (cloning) and CBL features (constructor functions). Nonetheless it does not inherit the problems from either language family. Furthermore we did not reintroduce classes as constructor functions are nothing but methods which in their turn reside in some environment (read: object). We therefore claim this tiny little language is a clean intersection of both families.

References

1. Taivalsaari, A.: Classes vs. Prototypes: Some Philosophical and Historical Observations. In Noble, J., Taivalsaari, A., Moore, I., eds.: Prototype-based Programming: Concepts, Languages and Applications. (1998)
2. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java^{TM} Language Specification. 2nd edn. Addison-Wesley (2000)
3. Stroustrup, B.: The C++ Programming Language. 3 edn. Addison-Wesley, Reading, Mass. (1997)
4. Goldberg, A., Robson, D.: Smalltalk-80: The Language and Its Implementation. Addison-Wesley (1983)
5. Ungar, D., Smith, R.: 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)
6. De Meuter, W.: Agora: The Scheme of Object-Orientation, or, the Simplest MOP in the World. In Noble, J., Taivalsaari, A., Moore, I., eds.: Prototype-based Programming: Concepts, Languages and Applications. (1998)

7. Taivalsaari, A.: A Critical View of Inheritance and Reusability in Object-oriented Programming. PhD thesis, University of Jyvaskyla, Finland (1993)
8. Smith, W.: NewtonScript: Prototypes on the Palm. In Noble, J., Taivalsaari, A., Moore, I., eds.: Prototype-based Programming: Concepts, Languages and Applications. (1998)
9. Zenger, M.: Type-safe prototype-based component evolution. In: Proceedings ECOOP 2002. Volume 2374 of LNCS., Springer Verlag (2002)
10. Van Belle, W., D'Hondt, T.: Agent mobility and reification of computational state: An experiment in migration. Lecture Notes in Computer Science **1887** (2001)
11. Self Home Page: `http://research.sun.com/research/self/`.
12. Cardelli, L.: A language with distributed scope. Computing Systems **8** (1995)
13. Lieberman, H., Stein, L., Ungar, D.: Treaty of orlando. In: Addendum to the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). Volume 23., New York, NY, ACM Press (1988) 43–44
14. ECMAScript Edition 4: `http://www.mozilla.org/js/language/es4/index.html`.
15. Lieberman, H.: 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 22. (1987) 214–223
16. Smith, R., Ungar, D.: Programming as an Experience: The inspiration for Self. In Noble, J., Taivalsaari, A., Moore, I., eds.: Prototype-based Programming: Concepts, Languages and Applications. (1998)
17. Dony, C., Malenfant, J., Bardou, D.: Classifying Prototype-based Programming Languages. In Noble, J., Taivalsaari, A., Moore, I., eds.: Prototype-based Programming: Concepts, Languages and Applications. (1998)
18. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley (1994)
19. Ungar, D., Chambers, C., Chang, B., Hölzle, U.: Organizing programs without classes. Lisp and Symbolic Computation **4** (1991) 223–242
20. Bracha, G., Lindstrom, G.: Modularity meets inheritance. In: Proceedings of IEEE Computer Society International Conference on Computer languages. (1992)
21. Green, T., Borning, A., O'Shea, T., Minoughan, M., Smith, R.: The Stripetalk Papers: Understandability as a Language Design Issue in Object-Oriented Programming Systems. In Noble, J., Taivalsaari, A., Moore, I., eds.: Prototype-based Programming: Concepts, Languages and Applications. (1998)
22. Blashek, G.: Object-Oriented Programming with Prototypes. Springer Verlag (1994)
23. Pico Home Page: `http://pico.vub.ac.be/`.
24. Abelson, H., Sussman, G., J., S.: Structure and Interpretation of Computer Programs. 2nd edn. Electrical Engineering and Computer Science Series. MIT Press, Cambridge, MA (1996)
25. Steyaert, P., De Meuter, W.: A marriage of class-and object-based inheritance without unwanted children. Lecture Notes in Computer Science **952** (1995)
26. D'Hondt, T., De Meuter, W.: Of first-class methods and dynamic scope. In: Proceedings of LMO, France, to appear. (2003)