Combatting the paucity of paradigms in current OOP teaching

Theo D'Hondt – Isabel Michiels
{tjdhondt+imichiel}@vub.ac.be
Programming Technology Lab - Vrije Universiteit Brussel

Abstract

If Java has succeeded in putting OOP on the map as the generalized approach to software development, it has also greatly reduced the natural diversity of programming languages that make computer science such an exciting field. This has led to a significant impoverishment of current academic offering, which in turn reflects badly on the capacity of future computer science professionals to react to the evolution in technology that undoubtedly lies in store for them. In this paper we advocate the (re)inclusion of the study of multiple paradigms in our academic curricula, in particular object flavoured ones. The results of an interesting experience in this context are proposed as a first step in this direction.

Introduction

The overwhelming success of Java as the premier programming language for the internet has turned its particular flavour of object oriented programming into the most widely accepted paradigm in use today. It is not surprising that the resulting demand for competent Java programmers has resulted in a tremendous pressure on the educational system to conform to this situation. In many cases, computer science curricula have been changed to promote Java as the first language that an undergraduate student is confronted with; in some cases this choice permeates the curriculum to the extent that other approaches to programming have largely disappeared. Even some very highly respected academic institutions have succumbed to this trend and the effect is becoming measurable: whereas past generations of computer science graduates were generally familiar with a wide range of programming paradigms, this is no longer a fact.

Even when we limit our scope to the world of objects, there seems to exist a general disregard for non static typing. Languages like Smalltalk and certainly Common Lisp/CLOS only occur in small pockets of professionals still in possession of the necessary skills and motivation. At a higher resolution, we see that alternatives to class based inheritance are largely ignored and in many cases even unknown. Java has effectively constrained programming to an object oriented, class based, statically typed, garbage collected style.

Even major conferences such as ECOOP are subjected to this evolution; it suffices to take stock of the tutorial content over the past years to notice this. Even within the technical track, Java related concerns seem to dominate, even at a more formal level . A surprising number of researchers are being starved away from their original interests and are turning to topics that are easier to find funding for.

We question this evolution: we deplore this shortsightedness, which is essentially driven by concerns of a commercial nature, concerns that as likely as not will be obsolete on fairly short

notice. We deplore that a large body of knowledge in the field of programming languages is no longer an active part in the expertise of the next generation of computer science graduates. We are more than ever convinced that engineering software starts with competence in engineering languages.

Experiences

An extremely recent monograph published by *Springer* [1] illustrates our point. This book is an erudite rendition of the various flavours of object oriented programming and their embedding into other paradigms. Unfortunately, it is unreadable to many current computer science undergraduates because it presupposes something, call it an historical and cultural background, which they no longer possess.

Our conjecture is that one should consider something like [1] as a basis for a high-level undergraduate course on the variations of the object oriented paradigm, but *not on its own*. In the current state of affairs, students need to be acclimatized through a very targeted program to (re)assert his or her consciousness of programming language culture.

Since two years now, we have been involved in the development of a *European Master in Object Oriented Software Engineering* (EMOOSE) [2]. This graduate program was launched under the auspices of the European Commission (see: ALFA); it offers a degree issued jointly by the *Vrije Universiteit Brussel* [3] and the *École des Mines de Nantes* [4]. Initially, because of the ALFA context, EMOOSE received a mix of French, Belgian and Latin-American students; currently, this mix is growing more varied.

There was a strong need in this program to establish a common ground for such an internationally diverse body of students. In particular, the requirement for an historical and cultural background as mentioned above, was felt to be an issue. A one week crash course was devised to remedy this; the content of this course is the main topic of the present communication.

Pico: a *tweakable* language implementation for teaching

We were strongly inspired by [6] to establish a self-contained technological framework for uniting all of the relevant notions and concepts. Skirting any really formal approach to syntax and semantics of programming languages, we nevertheless felt strongly about exposing students to a rigorous treatise of the matter of building a consistent language processor. The notion of *metacircularity*, brought to a pedagogical apex in [6], was the basic inspiration. At the source, we adopted *Pico* [5] as our universe of discourse. Pico is an extremely simple and portable virtual machine, with a very intuitive syntactical front end. The Pico virtual machine has a number of advanced qualities not really relevant to this discussion; it is a continuation based stack machine which boasts reflective features that make it ideal for supporting truly mobile code.

The basic Pico implementation is an 8000 line ANSI C framework that incorporates a fully self contained computation/storage model. It is documented by a 500 line metcircular implementation which closely mimics the C version. The evaluation engine consists of a number of semantic routines which attribute a simple abstract grammar.

```
makeBag()::
  { root: void;
    register(Item)::
      { traverse(Node, Action(New))::
          if(is_void(Node),
            Action([Item, 1, void, void]),
            if(Node[1]=Item,
              Node[2]:= Node[2]+1,
              if(Node[1]>Item,
                traverse(Node[3], Node[3]:= New),
                traverse(Node[4], Node[4]:= New))));
        traverse(root, root:= New);
        display(Item, " ok") };
    member(Item)::
      { traverse(Node)::
          if(is_void(Node),
            0,
            if(Node[1]=Item,
              Node[2],
              if(Node([1]>Item,
                traverse(Node[2]),
                traverse(Node[3])))));
        traverse(root) };
    capture() }
```

One can infer most of the properties of the Pico language from the example above. It is a simple, dynamically typed language with automatic memory management very similar to Scheme; Pico is properly tail recursive and it is based on first-class functions which are implemented as closures because of static scoping. Computational control is likewise first class, accessible using continuations and a `call-with-current-continuation`-like native operation. Similarly to Scheme, Pico can be used as a functional programming language, but it does implement destructive assignment should this is required.

A number of differences are worth mentioning:

- canonical function and operator notation
- tables rather than pairs/lists for data composition and argument lists
- Church booleans
- named lambda expressions
- call-by-name in addition to call-by-value argument binding
- no special forms or macro's
- first class environments called *dictionaries*
- first class computational state

This language implementation is extensively used in teaching at the undergraduate level. It is available as a portable and self-contained component that comes with a frontend for MacOS, Windows and Linux. It is sufficiently compact to have been ported to PalmOS.

Pico is *tweakable* in the sense that moving from the original language to a derived language is very simple: it requires prototyping the new features in the metacircular specification and then moving these out to the actual virtual machine. Because every aspect is first-class, including the computational state of the virtual machine, it has been used to experiment with

code mobility and migrating objects [7]. This is currently the most sophisticated Pico tweak.

Because of its flexibility, and because of prior encouraging results, it was decided to use the Pico language framework as the technological *equalizer* in the EMOOSE program. Two successive promotions of master's students confirm the aptness of this choice.

Modeling objects and object paradigms

The example on the previous page illustrates the abstraction features of Pico: `makeBag` is actually a *generator* function that can be used to instantiate bags:

```
aBag: makeBag()
:<dictionary>
```

The second line is in fact a response from the Pico evaluator: the bag is actually a dictionary, i.e. an instance of the Pico namespace implementation. A call to the native function `capture` is responsible for *freezing* the current dictionary.

Dictionaries are actually stacks of name-value bindings, implemented as simple linear lists. A qualification syntax allows us to evaluate an expression with respect to a given dictionary:

```
aBag.register("red")
:red ok
aBag.register("green")
:green ok
aBag.register("red")
:red ok
aBag.member("red")
:2
```

Using namespaces or *dictionaries* in this case, to support objects is a well known technique. Our objective was to use equally simple and basic language constructs to model other aspects of the object paradigm. The idea is to appeal to an existing intuition to reason about concepts that are not always immediately obvious.

In the context of the EMOOSE program, students were *shown* a prototype based object model expressed in this medium and then asked to *build* a class based model. Issues such as delegation, self and super reference, overriding and shadowing, information hiding, code sharing and reentrancy, and much more were covered. Most of the feedback was very positive; students felt that their grasp of otherwise vague and abstract notions was significantly improved; moreover their new capacity to experiment with variations in the object paradigm gave them a much better understanding of it.

Let us investigate some of the elements from this approach. This may shed some light on *why* the approach proved to be successful; also it might inspire the reader to adopt similar techniques in his or her educational mission.

Tweaking object related concepts

It is impossible to condense the details of a week long seminar into these few pages. However it is conceivable to get a feeling for the approach, i.e. *tweaking* the abstract grammar and the

```
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());
        clone() };
    clone() }
```

semantic routines of an existing virtual machine. So let us explore a number of concepts related to the object paradigm using the proposed approach; these are generally introduced in superficial terms, but it this instance will become much more tangible.

object prototype: actually a dictionary, the result of an anonymous cloning operation:

```
S: Stack(10)
:<dictionary>
```

object cloning: application of a native `clone` function; anonymous cloning `clone()` captures the current object; a specific object is cloned by specifying it as an argument (as in `clone(aDict)`):

```
T: clone(S)
:<dictionary>
```

object state: the mutable variables in an object (`n`, `T` and `t` in the `Stack` example); these are private, i.e. invisible outside the object. Cloning implies a *deep copy* of the object state

object behaviour: the immutable variables in an object (`empty`, `full`, `push`, `pop` and `make-Protected` in the `Stack` example); these are public, i.e. visible outside the object using the qualification syntax. Cloning implies a *shallow copy* of the object behaviour

method: a public variable bound to a function; methods are *not* closures, they are named lambda expressions without any reference to an environment. Functions used in a first class mode are implicitly converted to closures

message passing: application of an object's method using the object as dictionary:

```
S.push('alpha')
:<dictionary>
S.push('omega')
:<dictionary>
S.pop()
:alpha
```

mixins: methods that return a clone (`makeProtected` in the `Stack` example); mixins implement inheritance:

```
R: S.makeProtected()
:<dictionary>
R.push('delta')
:<dictionary>
display(R.pop(), ' ', R.pop())
:delta alpha
```

super reference: this syntactical construct is expressed by leaving out the qualifier in a qualification (`.push(x)` in the `Stack` example); it is implemented as a variation of dictionary lookup

self reference: this is implemented by a call to the native `this` function; it is implemented to capture either the current dictionary or the most recent qualifier value

overriding: this is performed by using homonyms; earlier definitions are hidden according to standard scoping rules

Note that *state* and *behaviour* are used in the historical sense; they can refer to any kind of value. Moreover, since scoping is no longer purely static (a method is evaluated in the context of a receiver), the notion of late binding is extended to variables:

```
number(p):
  { add(q):: number(p+q);
    show():: display(p);
    makeRat(u,v)::
      { p:[u,v];
        add(q)::makeRat(p[1]+p[2]*q, p[2]);
        clone() };
    clone() }
:<closure number>
z:number(5)
:<dictionary>
w:z.add(3)
:<dictionary>
w.show()
:8
r: z.makeRat(1,2)
r.show()
:<dictionary>
r:=r.add(1)
:<dictionary>
:[3, 2]
```

The essential tweak for our prototype based variation of the original Pico virtual machine rests with the dictionary model. We had to introduce a namespace based on frames that also makes the explicit distinction between mutable and immutable variables. A clone operator was immediately obvious as support for object prototyping, instantiation and refinement. The idea to factor out the dictionary from the original closure based representation of functions in order to obtain methods seemed to be a foregone conclusion.

This was the main hurdle for the EMOOSE students to take; once the approach was clear to them they managed a similar experiment for a class based tweak with surprising ease.

Conclusions

In the previous pages we have reported on an experiment to bring a very varied group of computer science graduates up to speed in the OO programming paradigm. The approach using a tweakable virtual machine as a technological support proved to be extremely efficient. In a single sweep, students acquired insight in both concepts and implementation related to objects.

Using well known concepts such as functions, closures, name spaces, scoping etc. and *without* having to resort to a formal approach based on lambda or other calculi, it is possible to describe the semantical finesse of the object paradigm. Moreover, this approach is well within the reach of any computer science graduate and he or she can experiment to his or her heart's content with variations in the paradigm.

It is our conjecture that a similar but simplified approach can be applied successfully at the undergraduate level.

References

[1] *The interpretation of object oriented languages*
Ian Craig
Springer Verlag (2000)

[2] http://www.emn.fr/MSc/

[3] http://www.vub.ac.be/english/

[4] http://www.emn.fr/international/Welcome.html

[5] http://pico.vub.ac.be

[6] *Structure and Interpretation of Programming Languages*
H. A. Abelson, G. J. Sussman
The MIT Press (1996)

[7] *Location Transparent Routing in Mobile Agent Systems - Merging name Lookups with Routing*
W. Van Belle, K. Verelst, T. D'Hondt
Proceedings of the 7th IEEE Workshop on Future Trends of Distributed Computing Systems (1999)