Learning by Doing: Using an Extendible Language To Teach Object-Oriented Concepts

Dehla Sokenou[1] and Stephan Herrmann[2]

[1] GEBIT Solutions GmbH, Cicerostraße 37, D-10709 Berlin,
EMail: dehla.sokenou@gebit.de
[2] Technische Universität Berlin, Fakultät IV - Elektrotechnik und Informatik,
Institut für Softwaretechnik und Theoretische Informatik, Fachgebiet Softwaretechnik,
Sekr. FR 5-6, Franklinstr. 28/29, D-10587 Berlin,
EMail: stephan@cs.tu-berlin.de

Abstract. The paper describes our experiences in teaching object-oriented concepts independently from a given programming language in the context of a course for advanced students. Instead of either teaching only theoretical concepts or using a selected object-oriented programming language, we combine both views. A simple extendible language, Lua [Lua], is used to explain object-oriented concepts and their different specificities. The exercise for our students is to implement a simple object-oriented interpreter based on Lua. We have seen that implementing object-oriented concepts themselves facilitates understanding object-oriented concepts and learning new object-oriented languages. Furthermore, it has been shown that our approach is suitable for teaching new programming paradigms like aspect-oriented or role-based programming, too.

1 Motivation

When teaching object-oriented concepts, there are two main approaches. Firstly, concepts can be described without explicitly referring to any programming language. This approach is independent from the different flavours of object-oriented concepts found in different programming languages. In most cases, this kind of courses is very theoretical in nature but gives a good base for students to learn new object-oriented languages. Secondly, teaching can be focussed on the concepts of one specific object-oriented language. Here, one object-oriented programming language stands for a special interpretion of object-oriented programming. Advantage is the practical approach that is based on only one programming language for either explanation of concepts and implementation of exercises.

A typical example for different flavours of object-oriented programming is inheritance. When regarding programming languages like Java, C++, Eiffel, Smalltalk, CLOS or Self, each of them gives a special meaning to inheritance. Java allows only single inheritance and introduces the interface concept. C++ has multiple inheritance like Eiffel but a completely different technical realization

in terms of name clashes. In Eiffel, method signatures can be redefined, not only method bodies. In Smalltalk, classes are special objects, so we have metaclass inheritance. CLOS allows to redefine a method by adding program code before or after the super-class method, a concept related to aspect-oriented programming. Self is a language without classes using object-based inheritance instead.

In our course, we teach all these different flavours. Concepts are presented using examples from different object-oriented languages. But instead of using a simple object-oriented example as introductory programming exercise for our students, they first have to implement their own object-oriented interpreter based on Lua. The newly created interpreted programming language is called *LOS* (Lua Object System). After that, the students have to use their LOS implementation for other exercises, e.g. implementing an object-oriented system using design patterns.

The course is being taught in this form since 1999, which seems to be a long time in software engineering. To our own suprise, our approach is still up-to-date, because it can easily adapt to new developments in the field of programming languages. We have seen that also concepts beyond object-oriented programming like aspects and roles can be taught in the same way.

2 The Interpreter Kit

In this section, we give a short introduction to the Lua interpreter and its extendibility and explain how Lua can help to implement – and therefore understand – object-oriented concepts.

Lua is a simple imperative language with a small set of functionality. Some features facilitate the development of an object-oriented extension of Lua. Entities in Lua are polymorphic and dynamically typed. A special syntax allows to pass a hidden target reference to a function and to refer to this object with the special name *self* inside the function body. This syntax in a way anticipates object-oriented programming, but Lua is not an object-oriented language. It has neither classes nor inheritance nor any other object-oriented features.

The way to implement an extension of the normal Lua interpreter behaviour is the metatable mechanism, a powerful reflective language concept. We first explain this concept based on a short example.

Consider a situation where a function call is made on a table[3] (e.g. `a(2)`, where `a` is a table with `a = {value=3}`. The plain Lua interpreter generates an error method because tables cannot be handled like functions. But Lua allows to define a function which will be called instead of the normal error behaviour. With two lines of code, we can define how to handle function calls on tables:

```
1   function mult(x, y) return x.value * y end
2   setmetatable(a, {__call=mult})
```

[3] Lua tables are associative arrays.

Line 1 defines a function `mult` that multiplies two number values. In line 2, the function `mult` is defined as the new standard behaviour of a function call on table `a`, thus `a(2)` will be a legal expression evaluating to `6`.

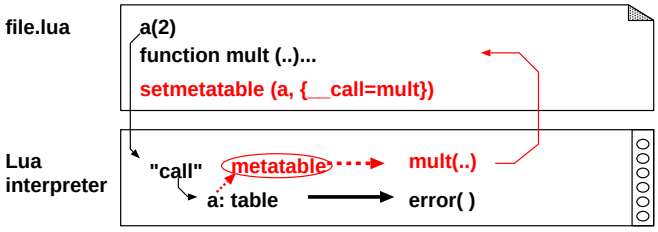

Fig. 1. Lua metatable mechanism

The situation is illustrated in Figure 1. Instead of calling the built-in error function, the user-defined function `mult` is called.

```
1   function findField (object, field)
2      if field == "parent" then          -- avoid loops
3         return nil
4      end
5      local p = object.parent            -- access parent object
6      if type(p) == "table" then         -- check if parent is a table
7         return p[field]                 -- (this may call findField again)
8      else
9         return nil
10     end
11  end
12
13  ObjectMetatable = {}                  -- metatable
14  ObjectMetatable.__index = findField   -- bind index access to find field
```

Fig. 2. Prototype-based language

As said before, Lua tables are associative arrays. Any value can be assigned to any key. Even a function can be a value. We use Lua tables to define objects. Attributes and functions are stored in a Lua table and define the object's features and behaviour. To define new objects, we can use other objects as prototypes. The prototype object becomes a parent of the new object. We define a metatable (`ObjectMetatable` in lines 13-14, see Figure 2), which gives special behaviour to all objects as follows: The attributes and functions of an object are found in the object itself or the object's parents (lines 1-11). Not shown in the example is

the creation of new objects based on a prototype object. The main duty of the method to create new objects is to set the metatable to the created objects (see also Figure 3, line 18). This is the first step to our own object-oriented language LOS and implements a prototype-based language.

When extending the simple prototype-based language with classes, we can reuse parts of the implementation of the prototype-based language. The class-based variant presented here consists of typed objects and attributes. A complete listing of the class-based LOS language is given in Figure 3.

The function `Class` (lines 1-11) creates a new class with the given attributes and methods. Methods of the parent class were copied to the new class. The function `new` (lines 15-20) creates a new object and assigns the metatable `Object-Metatable`. The function `findField` in lines 22-35 (including function `find-Method`) is similar to the one of the prototype-based language. It distinguishes between attributes that are found in the object itself and methods that are found in the object's class. If a value is assigned to an object, a dynamic type check is performed by the functions `setField` and `isTypeOK` in lines 37-56.

A short program using the class-based language implemented in Figure 3 is shown in Figure 4. A class `PERSON` and a subclass `ARTIST` and some objects of these types are defined.

The syntax presented in this example is a compromise between two conflicting requirements. First, the Lua parser cannot be modified, so any LOS program must use legal Lua syntax. On the other hand, the syntax should reflect the concepts to be taught in a convenient way, in order to support object-oriented thinking. Lua supports some syntactic sugar, which to a certain degree allows to write LOS programs such that they actually look like object-oriented programs.

The given examples of LOS variants show that classes can be special types of objects. Prototype-based and class-based languages have many similarities like finding features along the inheritance hierarchy. For students, the implementation of both examples is a way to understand the differences but also the commonalities in different kinds of inheritance.

The lesson learned is that different object-oriented concepts can co-exist in the same language and can have the same underlying technique but different characteristics.

The language LOS is used to implement normal object-oriented programming exercises. It is a great success for students to see that their own implementation of an object-oriented language can be used like any other programming language.

3 Experiences With LOS

As we have seen at the end of our course, students are able to differentiate between general object-oriented concepts and their technical realizations in different programming languages. Learning and using another object-oriented programming language than LOS is not difficult for them, like other exercises in

```
 1  function Class (name, parent)                -- define a new class
 2     local new = {_name=name}
 3     if parent then
 4        table.foreach (parent, function (i,m)
 5                  if type(m) == "function" then -- find all functions in parent
 6                     new[i] = m                 -- and copy them to new
 7                  end
 8           end)
 9     end
10     _G[name] = new
11  end
12
13  ObjectMetatable = {}                         -- metatable for objects
14
15  function new(class)                          -- define new objects
16     local newobject = {_attributes={}}
17     rawset(newobject, "_class", class)
18     setmetatable(newobject, ObjectMetatable)
19     return newobject
20  end
21
22  function findField(object, field)            -- find field (attribute or method)
23     local attributes = rawget(object, "_attributes")
24     local fieldValue = attributes[field]
25     if fieldValue ~= nil then
26        return fieldValue
27     end
28     local class = rawget(object, "_class")
29     local method = findMethod(class, field)
30     return method
31  end
32
33  function findMethod(class, method)           -- find method
34     return rawget(class, method)
35  end
36
37  function setField(object, fieldName, value)    -- sets attributes (type-safe)
38     local fieldType = object._class.Attributes[fieldName]
39     if fieldType == nil then
40        error("Assigning undeclared field "..fieldName)
41     end
42     if not isTypeOK(fieldType, value) then
43        error("Assigning incompatible value to field "..fieldName)
44     end
45     object._attributes[fieldName] = value
46  end
47
48  function isTypeOK(fieldType, value)           -- type check
49     if fieldType == type(value) then
50        return true
51     end
52     if getmetatable(value) == ObjectMetatable then
53        return value._class == fieldType
54     end
55     return false
56  end
57
58  ObjectMetatable.__index = findField           -- bind new behaviour for objects
59  ObjectMetatable.__newindex = setField
```

Fig. 3. Class-based language

```
1   Class('PERSON')                        -- class PERSON
2   PERSON.Attributes = {
3      name = String,
4      age  = Number,
5   }
6   function PERSON:print()
7      io.write("Name: " .. self.name .. ", Age: " .. self.age)
8   end
9
10  Class('ARTIST', PERSON)                -- class ARTIST
11  ARTIST.Attributes = {
12     pseudonym = String
13  }
14  function ARTIST:printExtended()
15     self:print()
16     io.write(" Pseudonym: " .. self.pseudonym)
17  end
18
19  person1 = new(PERSON); person1.name = "Alice"; person1.age = 24
20  person2 = new(PERSON); person2.name = "Bob"; person2.age = 30
21  person3 = new(ARTIST); person3.name = "Fred"; person3.age = 18
22  person3.pseudonym = "Wallace"
23  person1:print(); person2:print(); person3:printExtended()
```

Fig. 4. Example LOS program

this course – and subsequent classes – have shown. In these exercises, languages like Java, Eiffel or Object Teams are used.

We use Lua in our course since 1999. Students knowledge has changed over time. In 1999, most of the students were not familiar with object-oriented concepts when starting our course. Only a few of them knew Java or C++. To date, most of our students are experienced in a popular language like Java. But focussed on Java, they are surprised how many different flavours exist in object-oriented programming. Normally, more students apply for this course than we can accommodate. Feedback from our students support the impression of a successfull teaching approach for object-oriented concepts. Most of the students use their own LOS implementation for further exercises instead of our sample implementation. This experience of first implementing a programming language and then writing programs in this new language is fascinating for students. It allows students to better understand the evolving nature of programming languages, rather than taking today's languages as irrevocable laws.

Since 2001, we have extended our approach to other concepts beyond object-oriented programming. In 2001, the LOS variant implemented by our students included a role concept. An aspect-oriented LOS variant was implemented in 2004 – we had a LOS variant with before and after method inheritance already

in 2000.

The flexibility of Lua is also used in our research projects. Often, new language features are explored by implementing a prototype using Lua (see for example [Her00]). As a result of these research activities, the language family Object Teams has been developed – firstly as an interpreted prototype in Lua [HM01] and lastly as a compiler resulting in ObjectTeams/Java [Obj].

This close relation between teaching and research is another advantage of our course. Many of our students decide to write their diploma thesis in the field of object-oriented languages and beyond or work in one of our research projects in our group.

4 Conclusion

Our experience has shown that object-oriented concepts are better understood by students if they implement these concepts themselves. In a fast changing discipline like software engineering, students must be capable to quickly learn new concepts and programming languages. In our course, different flavours of object-oriented programming and beyond are taught as well as their technical realizations. This means flexibility and helps students to understand new programming concepts and paradigms and learn new programming languages in their future work.

The implementation of object-oriented concepts in an own interpreter helps to see the relationships between different concepts. Students see how combining existing concepts and the mechanisms behind them allows to build new programming languages. Often, combinations of techniques and concepts are not randomly chosen but relate to each other – for example, the interface concept in Java is motivated by the single inheritance concept.

An exercise that asks for the implementation of a type system in LOS is given after the basic implementation exercise. This helps students to distingish between the execution of programs and the constraints over the set of correct programs.

Lua is a suitable language for implementing an interpreter for one's own object-oriented language. Lua is easy to understand and to extend. There are of course concepts that cannot easily be realized by using the interpreter approach, but usually require the static analysis performed by a compiler. Examples are genericity and overloading.

Other candidates for implementing the exercises are Ruby [Rub] and Scheme [Dyb03].

Ruby is already an object-oriented language but extendible like Lua. Because of the object-oriented nature of Ruby, concepts are already implemented that are not present in plain Lua. In one of our research projects, a Lua implementation of a new programming language was followed by a prototype implementation in Ruby [Vei02] that includes a lot more features than the Lua prototype and

additionally opens for access to are large number of program libraries. For our course, we consider important that the language used for the exercises is not by itself an object-oriented language. For that reason, we don't use Ruby in the context of the course.

We know of other teaching groups using Scheme instead of Lua for implementing an object-oriented language. Scheme is extendible, too, but has a functional programming background. We find that Lua as an imperative language is closer related to main-stream object-oriented programming. The advantage of this is twofold: First, students find it easier to implement the language extension in an imperative style. Secondly, the resulting language feels quite similar to standard imperative object-oriented languages. In the end, students are just proud to implement programs using the language they have just developed.

References

[Dyb03] R. Kent Dybvig. *The Scheme Programming Language: 3rd Edition.* MIT Press, 2003.

[Her00] Stephan Herrmann. Lua/P – A Repository Language For Flexible Software Engineering Environments. In *2nd International Symposium on Constructing Software Engineering Tools (CoSET), 22th International Conference on Software Engineering (ICSE)*, Limerick, Ireland, 2000.

[HM01] Stephan Herrmann and Mira Mezini. Combining Composition Styles in the Evolvable Language LAC. In *Workshop on Advanced Separation of Concerns in Software Engineering, 23th International Conference on Software Engineering (ICSE)*, Toronto, Canada, 2001.

[Lua] Lua Homepage. http://www.lua.org/.

[Obj] Object Teams Homepage. http://www.objectteams.org.

[Rub] Ruby Homepage. http://www.ruby-lang.org/.

[Vei02] Matthias Veit. Evaluierung modularer Softwareentwicklung mit Object Teams am Beispiel eines Projektmanagementsystems (*german*). Master's thesis, Technische Universität Berlin, Germany, 2002.