Compare Object Orientated Programming in Go with Smalltalk

Daniel Voß

Introduction

Smalltalk is a pure object-oriented programming language and was designed and created in 1970s by Alan Kay, Dan Ingalls, Adele Goldberg, Ted Kaehler, Scott Wallace and Peter Deutsch. It influenced the development of following languages like Objective-C, Java and Ruby and so it also has some influence on the programming language Go.

Smalltalk is even more than a programming language it is an Interactive Programming Environment because it comes with its own Virtual Machine (VM) where the integrated development environment (IDE), the interpreter and all libraries necessary for the language are integrated. That means a project with Smalltalk is developed in its own VM like Squeak [sque] or Pharo [pharo].

In the following sections the language will be explained with the main aspect on object orientation and compared to the language Go. The code examples in this article are written in Pharo, a modern dialect of Smalltalk that implements the language very close to Smalltalk-80.

Syntax

The Syntax of Smalltalk aim to be near a normal but minimalized English sentence. There are only six keywords reserved in the language: *nil, true, false, self, super,* and *thisContext*. The rest like if else expressions and creation of classes are achieved by implementing it with the language itself.

Table 1. Table of syntax (adaptet from [syn08])

symbol	meaning
“blah blah”	Comment
$a	Character
‘String‘	String
:=	Assignment
.	Expression separator
^	Return
| x y |	Temporal variable declaration
#(1 2 3)	Literal array
{1. 2. 1+2}	Dynamic array
[…]	Blocks

In the static typed programming language Go every variable is has a data type. In the dynamically typed language Smalltalk - as it can be seen in the declaration example for temporary variable - variable do not have a data type. That means every object can be assigned to every variable without any cast.

Therefore, array also can store every object. For literal arrays the members must be constant at compile time. Dynamic array are defined at runtime (They only are available in newer dialects of Smalltalk).

Blocks are like Anonymous Functions in Go. A block is an object - like everything in Smalltalk - and can be assigned to a variable or passed as an argument and is executed with the message *value* (see Message).

Object Orientation Concept

In Go Object orientation can be achieved but it is not a must. Smalltalk is pure object oriented and therefore much stricter with rules for object orientation. The rules are: ([And11])

- Everything is an object.
- Every object is an instance of a class.
- Every class has a superclass.
- Everything happens by sending messages.
- Method lookup follows the inheritance chain.

These are the essential rules that will be deeper explained in the following sections.

Messages

In Smalltalk the communication between objects happens through messages. How an object reacts to a message is determined by the class of that object. Not all messages respond successful to all messages. How to respond to a message must be defined in the class or a subclass. When a message is send to a receiver another object is always returned.

There are three types of messages:

- unary message:

These are messages that don't require any argument and have the form receiver messageName. For example *5 sqrt* sends the message *sqrt* to the object *5*. An integer object *25* will be returned.

- binary messages

These messages take one argument and is used for operations, often arithmetic like: *3 * 4* this expression sends the message *** to the object *3* with the argument *4* and will return a new object *12*

- keyword messages

These messages require one or more arguments that are send to the receiver. For every argument is a keyword ending with *:*. For example *10 between: 5 and: 15* sends a messages with two arguments to the object *10* and it will returns true.

Messages can be send successively. Without brackets the sending priority is Unary > Binary > Keyword. Messages with the same priority are send from left to right. With only these three priorities the expression *3 + 2 * 4* returns *20* and not like in Go 11.

Messages can be compared to method calls in Go but the main difference is, that in Smalltalk the receiver choose how he responds to the message. In Go keywords for if-else statements and loops are reserved. Smalltalk does not have such keywords, and even that happens by sending messages to objects.

```
myValue < 17 ifTrue: [ blockTrue ] ifFalse: [ blockFalse ].
```

Here first the binary message < with the argument *17* is send to the object *myValue*. That will return an instance of the class *True* or *False*. Now the message is send to that object. Depending on the class of the object it will execute the first or second argument (first or second block).In a similar way loops are implemented, just by sending a message to an object that reacts to that message.

Classes

A class can be defined like:

```
Object subclass: #Person
    instanceVariableNames: 'surname age’
    classVariableNames: 'count'
    package: 'Peoples'
Person>> age
    ^ age
Person>> age: anObject
    age:= anObject
```

A class has instance variables and class variable. In the example were *surename* and *age* as instance variables defined and *count* as class varaible. Instance variables are unique for each instance of that class, class variables are shared by every instance.

The behaviour of the object from a class is defined by its methods. A method determines how an object reacts to a received message. In the example only two methods are implemented, the getter and setter for the instance variable *age*.

In Smalltalk it is not possible to declare the access of a variable or method like it is done in Go with upper or lower letter. In Smalltalk every method is public, and every variable is protected. It is protected because subclasses that inherit from a class have also access to all variables. Therefore, like done in the example getter and setter have to be defined to access them from outside or from their metaclass (see metaclass).

As pointed out before the creation is achieved with the language itself - else like Go that uses for the purpose of defining a class the keyword *struct* - because it can be read as sending a message with four arguments to the object called *Object*. This will create a unique instance of a metaclass for the defined class (see section Metaclass) which now can create instances of this defined class.

Inheritage and Polymorph

Every class inherits from a single superclass. That means Smalltalk has a clear class tree beginning with ProtoObject. Usually it inherit from *Object* or an subclass like:

```
Person subclass: #Teacher
    instanceVariableNames: 'subject'
    classVariableNames: ''
    package: 'Peoples'
Teacher>>age
    ^(age - 1)
```

Here a new subclass *Teacher* is created that inherits from *Person*. That means an instance of *Teacher* will have the same instance variable as an instance of *Person* and additional the variable *subject*. It will also share the same class variable.

The subclass can define new methods and override or expand existing methods.In the code example the method age is overritten. To expand a method, the old message just has to be send to the superclass with the keyword *super* and then add the new functionality.

When a message is send to an object it will first look for a method to handle the message.When no suitable method is fund the search continuous in the superclass and goes up the inheritance chain until a method is found.

Because Go decided against an inheritance tree like Smalltalk such functionality can be achieved by embedding.

Interfaces like known in Go are not necessary in Smalltalk because Smalltalk is dynamically typed and no type checked for arguments is done. A handed argument can be everything.

Metaclass

As pointed out before in Smalltalk everything in Smalltalk is an object and every object is an instance of a class. Because Smalltalk is consequence with that rule classes are object too. That means even classes are an instance of a class, called metaclass. By creating a class with *subclass:* - as described before - a hidden metaclass and a unique instance of it is created. That concept might be at first a bit unfamiliar for an Go programmer but it will be clearer and just a logic continuation of Smalltalks rules with the following example. Most of the time one does not even have to care about metaclasses.

By sending the message *new* to this instance of metaclass an instance of the class is returned. For the abouve example:

```
t1 := Teacher new.
```

will create a *Teacher* object and saves it in the variable *t1*. From this point of view, it is almost equal to Go’s New but what happens in the background differs. As New in Go is a keyword, the creation of an instance in Smalltalk is defined by the language just by sending the message *new* to an object -an instance of a metaclass -.

To completely fulfil the rule every metaclass is an instance of *Metaclass*. Even the metaclass of Metaclass is an instance of the class *Metaclass*. That relation is shown in illustration Figure 1 for the abouve generated teacher t1.

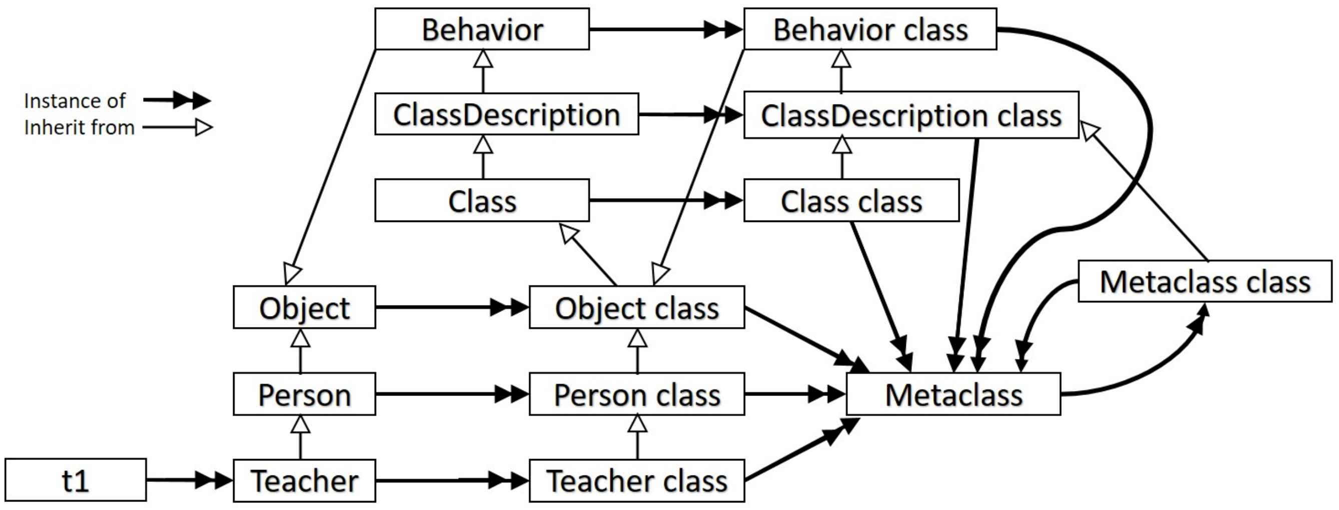

Figure 1:All instances realtion and inheritage relation for t1 (adaptet from [pharo] Chapter 13)

One can also change the behaviour of the instance of the metaclass. For that most of the common IDS have a button to switch to the “class side” view of a class. Here for example constructors can be implemented:

```
Person class>>initialize
    count := 0.
Person class>>new
count := count +1.
    ^super new.
Person class>>new: aAge
    | person |
    person := self new.
    person age: aAge.
    ^person
```

That code expression implements a constructor for a Person. Every time an instance of Person or Teacher is created the *count* variable is incremented. Initialize is executed the first time a *new* message is send for this object.

As it may at first view scare programmer new to that concept it shows that Smalltalk is a pure object orientated language and is very strict with its concept.

Conclusion

Smalltalk is an interesting programming language because of its strict object orientation and the messaging system. It was designed for educational use what can be seen because it is easy to learn with only a few keywords, close to the englisch language and only focuse on one way of programming - object oriantaion - where else Go might be harder to learn because its possibility to code in more ways.

For a Go or even Java or C++ programmer testing Smalltalk might change the way one think about object orientation. To see how strong that object orientation can be implemented and also learning about the beginning of the object orientation that might be implemented in ones favorite language.

There are also some problems with the programming language. The Go developers had a reason when not to implement large inheritance chain when looking back on existing lanuages.

Another problem is, that on common implementations of the language everything, including development and workspace runs in an VM. On the first view and for testing the lanuage out, that might be an advantage because everything is combine in one software - what also supports the purpose for education - but that makes it harder to include new libraries, share code and work on several PCs on one project.

One can learn a lot by testing out Smalltalk but it might be not the language for every bigger commercial project.

Literature

- [pharo] https://pharo.org/
- [sque] https://squeak.org/
- [And11] Pharo by Example. Andrew P. Black, Stéphane Ducasse, Oscar Nierstrasz and Damien Pollet. 2011 http://pharo.gforge.inria.fr/PBE1/PBE1.html
- [syn08] syntax-across-languages 2008 http://rigaux.org/language-study/syntax-across-languages-per-language/Smalltalk.html