Context-oriented Programming

Willian Ver Valem Paiva

1 Introduction

The purpose of this study is to analyze the article *Context-oriented Programming* (COP) [1] and to show how it is viewed and used today by the community. For that purpose, a deep analysis of the article, and of other works related to the subject, is done, with the objective of well understanding the program paradigm proposed by COP.

The contextual information is becoming the core of many contemporary applications, and the need for a support for context-awareness on mainstream languages has become considerable. As the tools offered by mainstream languages are somehow cumbersome, COP brings to the table this paradigm that treats of context explicitly. For that purpose, COP uses dynamic representations of layers as *first-class entities* that can be activated and deactivated in arbitrary places of the code.

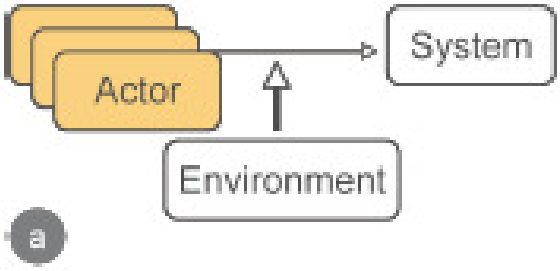
The article also introduces some extensions of programming languages that have being adapted to COP :

- ContextL, a Common-lisp extension
- ContextS, a Squeak/Smalltalk extension
- ContextJ, a Java extension [2]

2 Context definition

To better define a context, COP strips it down into 3 subdivisions which are *actor*, *system* and *environment*.

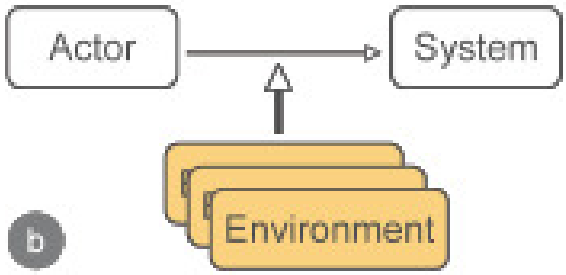
2.1 Actor



The actor is an entity which talks with the system in such terms as: function calls, messages, and any other means that demand a behavior from the system. Actor-dependent behavior variants are different ways of making representations of a system or part of a system. It can be for example the creation of different charts or graphs based on a statistical resource. The actor is the element which defines how information given or extracted from the system will be represented, in which form. It is part of the elements which generates the context-dependent behavior of the system.

The system behavior differs not only according to the different requests made by the actor but also according to a same request made by different actors.

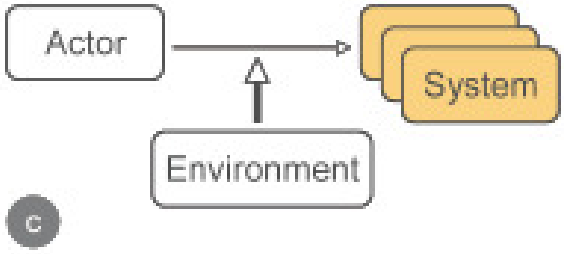
2.2 Environment



The environment represents everything external to the relationship between *actor* and *system* which needs to be taken into consideration by the system. Some examples may be given by GPS systems, temperature sensors, accelerometer, etc . . .

A system's response to an actor's demand can be adjusted to take properties of the computational environment into account.

2.3 System



The system is a computational entity that provides some behaviors upon request. The scale of the system is not taken into consideration here. The constituents of a system may for be example methods, objects, subsystems.

The system behavior can differ depending on its current state, on information or dependencies to other system's parts, or subsystems. An example of system-dependent behavior could be a notification for the change of system parts. On the system, the determining element that has influence is the system context itself, and not any type of activity from the actors.

3 Structure

Context-oriented programming is supported by the following essential language properties :

- means to specify *behavioral variations*
- means to group variations into *layers*
- dynamic *activation* and *deactivation* of layers based on context,
- means to explicitly and dynamically control the *scope* of layers

[1]

3.1 behavioral variations

Variations consists in the addition, removal or modification of a behavior. Those variations can be expressed as partial definitions of modules in the underlying programming model such as procedures or classes, but not only; variations can also be expressed as edits, wrappers or transformations.

3.2 Activation and Deactivation

Possibility to activate or deactivate at run-time the layers bulking context-dependent behavioral variations based on the current context.

3.3 Scoping

Offers the control over the scope of layers that are *activated* or *deactivated*. These variations cam be simultaneously active or not, within different scopes of the same running application.

3.4 Layers

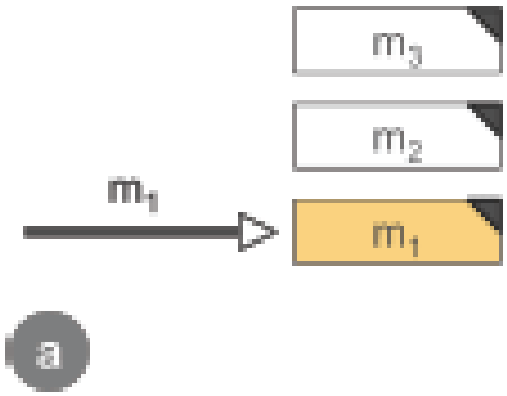
Layers are first-class entities that group related context-dependent behavioral variations, which are being composed in reaction to contextual information.

According to the current context, layers can be activated or deactivated. COP limits itself to the activation and deactivation of layers at run-time, and leave to the application the modeling of the context and the providing of what contextual information is relevant, since solutions using object-oriented abstraction are sufficient for the task. [3]

4 Multi-dimensional message dispatch

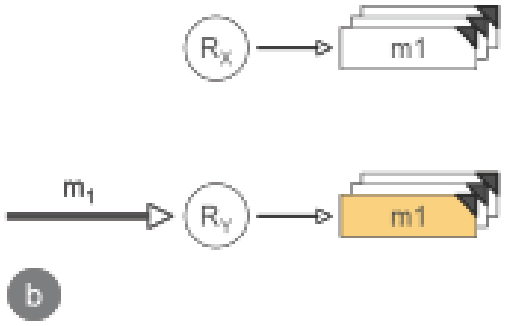
Multi-dimensional message dispatch [4] will be used to present COP ; but it does not mean that COP is limited to scenario. The following analogy will help understand that COP is the next step in the chain : procedural programming, object-oriented, and subjective programming.

4.1 One-dimensional dispatch



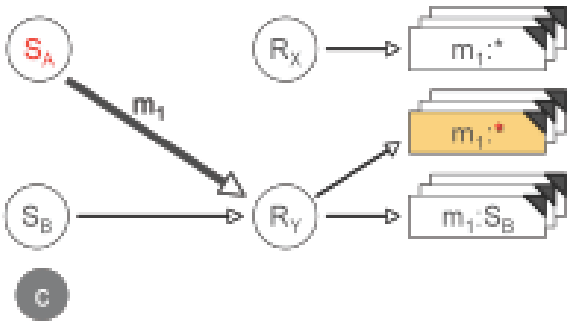
Also known as *procedural programming* it offers only one dimension [4]. Procedures calls or names are directly linked to its implementation. As in the image above the call to *m1* can only invoke *m1*.

4.2 Two-dimensional dispatch



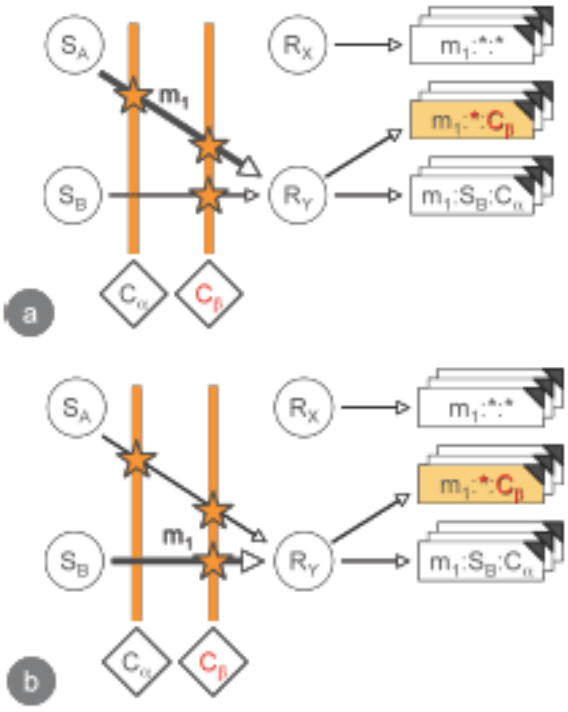
This refers to object-oriented programming. In this paradigm another dimension is added to the message dispatch in relation to procedural programming [4]. In addition to procedural programming, the message dispatch takes into consideration the receiver of the message to decide whose implementation will be used. The figure above has two receivers *Rx* and *Ry*, and each one maps to its respective implementation of *m1*. A *m1* call will be directed to an implementation depending on which receiver gets the message.

4.3 Three-dimensional dispatch



As introduced by Smith and Ungar [4], subjective programming expands object-oriented programming by adding yet another dimension to the message dispatch. In this paradigm the implementation is not only selected by its name and receiver, but also by its sender. On the figure above, *Ry* has two implementations of *m1* the *m1:** that is a general implementation and *m1:SB* which is an implementation destined to receive messages sent from *SB*. In the example, the sender is *SA* what makes the message being directed to *m1:**.

4.4 Four-dimensional dispatch



By adding one more dimension to the message dispatch ("*context*"), context-oriented language takes subjective programming a step further, by taking the context into consideration when selecting a method's implementation.

For example on the figures above we have two possibilities. As already seen in subjective programming which takes into consideration the senders *SA* and *SB*, in this situation, two new elements *Ca* and *Cb* are added ; those are the context. And sender *SA* is affected by both contexts *Ca* and *Cb* ; and *SB* is only affected by the context *Cb*. For that reason when the call is made from sender *SA* in the "*figure a*" the implementation to be used is the *m1:*:Cb* as its signature fits the call received "*m1:SA:Ca:Cb*". On the second figure, "*figure b*", the sender is *SB* ; in this case the call signature is *m1:SB:Cb* which also fits for the method *m1:*:Cb*.

5 Examples

In this section, we will present brief examples of contextJ [1] [2] to show context-oriented programming in action.

1. At the time when this text is being written, contextJ is not being maintained any more, but of its features can be found on its successor *jcop* https://www.hpi.uni-potsdam.de/hirschfeld/trac/Cop/wiki/JCop

Obs: the contextJ code showed on the following samples are a proof of concept. To achieve such functionalities a rework on the Java from the compiler to the virtual machine itself would be necessary.

Here are presented at first the definition of two classes ; one to represent a Person that has a name, address and employer ; and a second class to represent an Employer that also has a name and address. Both classes have the *toString* method implemented. This method just prints the name of the object like = Name: name =. On the *Person class* two layers are created *Address* and *Employment* which are the *first-class* entities, used, in this case, to override the method *toString*.

Note the use of *proceed*, used to ensure that the original implementation of the method is called, in a way similar to that of the *super* in Java.

```
class Person{

    private String name;
    private String address;
    private Employer employer;

    Person(String newName,
           String newAddress,
           Employer newEmployer){

        this.name = newName;
        this.address = newAddress;
        this.employer = newEmployer;

    }

    String toString(){
        return "Name: "+name;
    }

    layer Address {
        string toString(){
            return proceed()+
                "; address: "+
                address;
        }
    }

    layer Employment {
        string toString(){
            return proceed()+
                "; [Employer] "+
                employer;
        }
    }

}

class Employer{
    private String name;
    private String address;

    Employer(String newName,
             String newAddress){

        this.name = newName;
        this.address = newAddress;
    }

    String toString(){
        return "Name: "+name;
    }

    layer Address {
        string toString(){
            return proceed()+
                "; address: "+
                address;
        }
    }

}
}
```

Now, to use those layers, contextJ provides *with* and *without* for their activation and deactivation, example:

```
Employer vub = new Employer("VUB","1050 Brussel");

Person somePersom = new Person("Pascal Costanza",
                               "1000 Brussel",
                                vub);

with(Address){

    with(Employment){
        System.out.println(somePersom);
    }

}
```

Note the possibility to chain and nest the layers activation to get the desired result :

```
Output:
 Name: Pascal Costanza; Address: 1000 Brussel;
 [Employer] Name: VUB; Address: 1050 Brussel
```

When looking at this example, it is possible to see the similarity of the layers with the design pattern *decorator*, where you can nest functionalities into the original object. Somehow, the layers approach presented by COP has its elegance, and simplifies the code. But as the layers have to be declared on the class, it gets behind *decorator* since a new decoration can be created at run-time. On the other hand, the advantage of COP over the *decorator* is the use of *with* and *without* that makes it easier and simple the control of the activation and deactivation of a functionality.

6 Related Work

Context-oriented programming share a bit of the design with *Aspect-oriented programming* (AOP) [5] and *Feature-oriented programming* (FOP) [6].

Aspect-oriented programming paradigm introduces the means to modulate *crosscutting concerns* and decrease code scattering. On the other hand, feature-oriented programming also tackles the *crosscutting concerns*.

In their own way, both paradigms engage on diminishing the cumbersomeness of dealing with *crosscutting concerns* in the actual scenario of object-oriented programming.

When looking at FOP and the mixing layers approach [7], it has a noticeable similarity to the layers on COP, but the difference among those paradigms lies in the fact that FOP is focused on the compile-time and COP introduces a dynamic activation and deactivation of layers at run-time.

7 Discussion

As was made clear, how to deal with *crosscutting concern* has being a dilemma for the programming community and many techniques to deal with such a riddle have been elaborated.

Each technique has been accepted differently by the community. This goes from some bad techniques such the anti-pattern *Grand central station*, where every significant method of function passes through a monstrous subroutine, to the elaborate and well thought solutions such as many of the creational design patterns, namely *decorator* and *compositor*. And it even went to, as COP shows it, the complete rethinking of the paradigm used to program, and pushing further from object-oriented, with the aim to make it possible to deal with *crosscutting concerns* in a cleaner and more elegant way.

What COP brings to the table has a full potential for being a game change on the software architecture design, but yet it has not been really embraced and accepted. On the other hand, Aspect-oriented programming had a large acceptance, and has even been integrated in many of the mainstream languages like *C++*, *Java*, *Scala* and some others. It has also been integrated into some big well known frameworks like *Spring* for java.

We can then conclude that, even if COP has shown some qualities and value, it does not seem mature enough to be integrated in mainstream programming.

References

[1] Robert Hirschfeld, Pascal Costanza, and Oscar Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3), 2008.

[2] Malte Appeltauer, Robert Hirschfeld, Michael Haupt, and Hidehiko Masuhara. Contextj: Context-oriented programming with java. *Information and Media Technologies*, 6(2):399–419, 2011.

[3] Anind K Dey. Understanding and using context. *Personal and ubiquitous computing*, 5(1):4–7, 2001.

[4] Randall B Smith and David Ungar. A simple and unifying approach to subjective objects. *TAPOS*, 2(3):161–178, 1996.

[5] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European conference on object-oriented programming*, pages 220–242. Springer, 1997.

[6] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In *European Conference on Object-Oriented Programming*, pages 419–443. Springer, 1997.

[7] Yannis Smaragdakis and Don Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):215–255, 2002.