Journal of Object Technology
Published by ETH Zurich, Chair of Software Engineering © Jot, 2010
Online at http://www.jot.fm

Formal Model and DSL for Separation of Concerns based on Views

Mehdi Adda[a] **Hamid Mcheick**[b] **Hafedh Mili**[c]

a. Département de mathématiques, informatique et génie, Université du Québec à Rimouski, Rimouski (Québec), Canada

b. Département d'informatique, Université du Québec à Chicoutimi, Chicoutimi (Québec), Canada

c. Département d'informatique, Université du Québec à Montréal, (Québec), Canada

Abstract The separation of concerns (SOC), as a conceptual tool, enables us to manage the complexity of software systems that we develop. The benefits of this paradigm, such as reuse, enhanced quality and adaptability, have been key drivers of its adoption. Modern software systems and applications take advantage of the technologies built around this paradigm, in which a client program can access different functional aspects (views) of the same domain. One of these SOC approaches is View-oriented Computing (VOC), which suffers from a formal model to canonically and consistently represent the different concepts of VOC as well to have the necessary background to formally verify the systems build on top of it. This paper describes a formal algebra-based model to describe different entities related to VOC. Especially, it introduces algebra and formalism associated with a Domain Specific Language (DSL) notation to illustrate the VOC paradigm.

Keywords Separation of Concerns; View-Oriented Computing; Formal model

Mehdi Adda, Hamid Mcheick, Hafedh Mili. Formal Model and DSL for Separation of Concerns based on Views. In Journal of Object Technology, vol. 9, no. 6, 2010, pages 25–50. Available at http://dx.doi.org/10.5381/jot.2010.9.6.a2

1 Introduction

Separation of concerns is a problem-solving idiom to break the complexity of a problem into loosely-coupled, easier to solve, subproblems. Underlying this idiom is the hope that the solutions to these subproblems can be composed relatively easily to yield a solution to the original problem. The history of programming languages may be seen as a perennial quest for modularisation boundaries that best map (back) to "natural modularisation boundaries" of requirements. Aspect Oriented Software Development (AOSD) methods are no different. However, most of the research on AOSD has focused on the semantics of aspects and aspect composition, i.e. the solution (software) domain, as opposed to the semantics of concerns and concern separation and composition, i.e. the problem (requirements) domain. The existing study cases have no algebra model to descibe the conceptual appropriateness of the AOSD techniques. This paper describes an algebra model of one of separation of concerns approaches: View-Oriented Computing (VOC).

In VOC, an application object consists of a core object, a variable set of functional slices (or views), reflecting the changing roles of the object during its lifetime. The set of views "attached" to an object determine the messages to which it can respond, and the way it responds to them. Viewpoints are generic views which can be parametrized per domain object. They abstract functional behavior in a domain-independent way, and are developed independently of the classes to which they apply.

In our approach (VOC) [Mcheick06, Mcheick07], clients require explicitly the activation, the deactivation, the attachment and the detachment of views, even if these views are distributed on different sites. In distributed view-oriented programming, different client sites may access different view combinations of the same object, known as interfaces. Therefore, the servers have to manage several interfaces requested by different client sites. Each view passes through several states: active, inactive, attach, detach–called view lifecycle [12]. In the approach we propose here the evolution of any object is extracted and abstracted to the Module level.

View-oriented programming (VOC) suffers also from a formal model to check up and verify the privacy of each client site. This paper introduces a cluster-based architecture, and an algebra model (as a formalism) to describe the different concepts of VOC.

The next section includes a brief overview of View-Oriented Programming. Section 3 describes an algebra model for VOC. An example, which illustrates our model, is given in section 4. We conclude in section 5.

2 View-Oriented Computing

The goal of this section is to introduce briefly and explain the VOC paradigm and explain some of its aspects. Among these aspects we may mention the enabling, disabling, activation, deactivation and cloning of views, etc. We set out to provide support for the following:

- enable client programs to access several functional areas or views simultaneously;
- support the addition and removal of views (functional slices) during run-time;
- support different interfaces during run-time;
- have a consistent and unencumbered protocol to address objects that support views;
- enable each client program to add functional slices based on its privileges.

More details are set forth in the following sub-sections.

2.1 VOC Architecture Overview

In VOC, enterprise software can be seen as a set of local or distributed software systems. Each system has one or more clusters that represent a kind-of aggregation of services to target different subjects (users). Thus, a cluster is the abstraction of the system to hide the details of software. A cluster is a set of functional slices (views) and it may be composed of views from different domain objects that let a transparency access to a system to different categories of subjects. In other words, a cluster is the generalisation of the view concept to enterprise systems (see Figure 1). It is important to understand the VOC because it provide support for the following: i) enable clients to access several views simultaneously, ii) suppport the addition or removal of views during runtime, and iii) have a consistent and unencumbered protocol to address objets that support views.

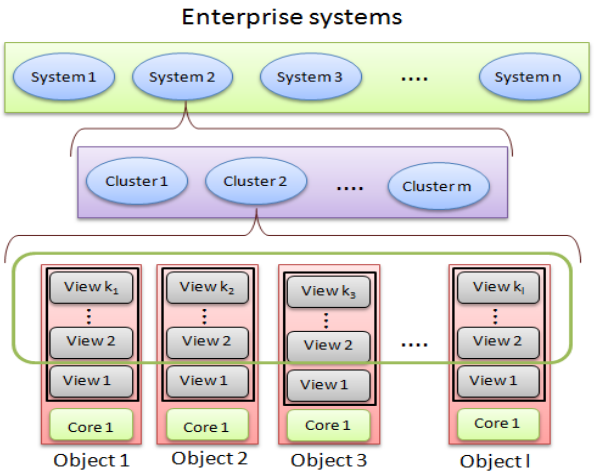

Figure 1: VOC architecture overview

In Section 4, we illustrate the cluster-based VOC using a bank account example. The system of this example, where we show how views are assigned to different clusters, have two different clsuters. Each of them contains a set of fonctionalities to which a category of users have access to. As we show it, the access control is managed by means of roles.

2.2 Separation of concerns

The separation of concerns technique is a general problem solving heuristic that consists of solving a problem by addressing its constraints, first separately, and then combining the partial solutions with the expectation that, 1) they be composable, and 2) the resulting solution is nearly optimal. For this heuristic to yield satisfactory results, the concerns that we are trying to treat separately must be fairly independent, to start with, so that they don't interfere with each other. Further, the problem solving activity itself needs to yield solutions that are composable.

In this section, we define the separation of concerns problem for the case of software. In this case, the "problem" is a set of requirements, and the "problem solving" process is the software development process. Characterizing the software development process is given in this section.

2.2.1 A transformational view of software development

Software development is a complex activity involving a variety of skills and a variety of conceptual and formal tools. For the purposes of reasoning about software development—and perhaps automating some of its steps— researchers and practitioners alike have found it useful to view software development as the process of going from specifications of what is to be done (requirements), to precise specifications of how it is to be done (implementation). Dasgupta identified two kinds of requirements in any design problem, empirical requirements, which specify externally observable or empirically determinable qualities that are desired of the artefacts, and conceptual requirements, which specify adherence to a particular style [Dasgupta91]. For the case of software, there are two kinds of externally observable qualities, functionality—the what—on one hand, and run-time behaviour—the how, including performance, and the like. Accordingly, we see three major categories of requirements for software development:

1. Requirements of functionality.
2. Run-time requirements.
3. Requirements on the software artifacts.

These correspond closely to the categories of architectural qualities identified by [Bass03]. Describing a program using an executable specification language may be seen as performing a first step of the design process, i.e. ensuring functionality. Later steps can worry about run-time behaviour and artefact quality.

2.2.2 Aspect-Oriented Programming (AOP)

Aspect-oriented programming recognizes that the programming languages that we use do not support all of the abstraction boundaries in our domain models and design processes. Underlying. AOP is the observation that what starts out as fairly distinct concerns at the requirements level, or at the design requirements level (non-functional requirements) end us tangled in the final program code because of the lack of support, both at the design process level, and at the programming language level, for keeping these concerns separate. With aspect-oriented programming, these concerns may be packaged as aspects, which can be woven into "any" application that has those concerns [Kiczales97].

Aspect-oriented programming requires three ingredients:

1. A general purpose programming language for defining the core functionalities of software components,

2. An aspect language for writing aspects, i.e. code modules that address a specific concern and that cross-cut various components in the general-purpose language, and
3. An aspect weaver, which is a pre-processor that "weaves" or "injects" aspects into the base software components to yield vanilla flavor components, coded in the general purpose programming language.

The output of the aspect weaver is next fed into regular programming toolkit (compiler, linker, etc.) to yield the application.

2.2.3 Subject-Oriented Programming (SOP)

Subject-oriented programming views object oriented applications as the composition of several application slices representing separate functional domains or add-ons (features) to existing functional domains. Such a slice is called a subject and consists of a self-contained, declaration-wise, object-oriented program, with its own class hierarchy [Harrison93].

Consider the example of a company that manages a set of trucks to make merchandise deliveries. A particular truck means different things to different departments: it is a production resource that comes in limited supply and that needs to be scheduled efficiently. It is also an amortizable asset to the accounting people who write-off the depreciation on its purchase value each year. It may also been seen as some piece of equipment that needs regular (scheduled) and on demand maintenance. It is conceivable that different people will have developed each one of these applications—and perhaps some purchased off-the-shelf—with its own definition of the class Truck.

2.2.4 Runtime aspects

Allowing an object to change its behaviour in non-predictable ways during runtime is problematic. First, we have to make that behaviour somehow available on-demand. As we saw in sections above, all of AOP, and SOP, integrate the various concerns/views during coding time by instantiating the appropriate classes, but the set of roles/subjects available to an object remain constant throughout the lifetime of the object. Additional constructs may be used to make those behaviours available on demand, as we show below.

In terms of invoking the proper combination of behaviours, depending on which roles/concerns are embodied in a given object, we have to implement some runtime dispatching method which will direct a message request to the method (or

combination of methods) that is available to it at that time. This can be implemented in one of several ways, including:

1. With typed languages, a combined interface that embodies all the potentially activated/attached concerns dispatches to specific private methods, depending on which concerns/roles are currently activated/available. This may be the approach used with C++ based or Java-based implementations,
2. With dynamically typed, reflexive languages, we can modify the default method dispatching mechanism to direct a method call to the appropriate method combination. This method is appropriate for Smalltalk and Ruby.

2.3 Viewpoint concept

It has been our experience that in business information systems, the roles played by domain objects often correspond to generic business processes, and do not depend on the business domain. For example, regardless of what product or service an organization sells, an employee will always be considered both as a resource, to be assigned to various production tasks, as well as a "recurrent cost". Lots of the earlier information engineering frameworks (see e.g. [Carlson79]) and the new wave of analysis patterns are based on this reality [Coad00].

Viewpoints are to views, what genericity (parametric polymorphism) is to classes. In other words, a viewpoint is a template that describes the behaviour and the state of views. A view of a viewpoint is called instance of the viewpoint. Thus a viewpoint is a reusable component that can be adapted to different domains.

The instantiation is a process that generates from the reusable component by means of transformation process into a concrete form, based on parameters.

Using our model of view programming, the different roles that an application object can play will be represented by views. When those roles correspond to different business processes, then the logic of the code of the views should be reusable across business domains. For example, for the accounting department, a machine tool, a computer, or a truck, are all pieces of equipment whose acquisition price can be amortized over time to reflect both wear & tear and obsolescence. Hence, we should find a way of coding the behaviour of "amortizable assets" generically, and reuse it across business entities. We propose a kind of a template for functional roles/views that is parameterized by those elements of the interface of the core object that are required by the functional role. This template, called viewpoint, can then be instantiated for different types of assets.

In addition to supporting the reuse of views, the use of view templates, or viewpoints, enables us to decouple the development of views from that of core

objects. For instance, because views refer to core objects, without the generation from viewpoints, not only do we have to hand-code specific versions of views for each core object, but we have to have the definition of those core objects available beforehand (see Figure 2). Practically, this means that the "core objects" teams and the "viewpoints" teams within an organization can proceed independently from each other, removing some of the bottlenecks from the development of interdepartmental information systems. It could also mean that viewpoints may be purchased from third party software vendors.

2.4 VOC systems evolution and experience capitalization

In software engineering, refactoring is an important phase of the life cycle of products which also includes their evolution, recovery, maintenance, and retirement [Mens02, Mens05, Mens08, Van03]. In VOC, we mostly encounter situations where core objects are extended with many views as well as with a dynamic evolution of core objects and views. In fact, a core object or a view may be extended at runtime with new behavior and state as the related domain evolves [Mcheick09]. The capability of VOC to add and remove funtional area is useful to fill the gap between new business expectations and old planned features. However, this has a price; if the dynamic evolution of a VOC system is pushed too far (core objects and views with growing number of attributes and methods), the system will be less trivial to manage and experience and knowledge gained from this evolution is mostly localized in the object itself (i.e. in its core and views) and may not be generalized in order to be applied elsewhere to extend viewpoints, views, classes, and objects.

Thus, we propose an approach by which the evolution of any object may be extracted and abstracted to the Module level. By doing so, the same evolution may be applied back to other core objects or views without reinventing the wheel again and again. Technically speaking, we are bringing module capabilities, offered by dynamic languages such as ruby [12, 13], to be dynamically included in an inheritance chain of a class or an object in OOP that classes and objects lack, to the VOC paradigm. In other words, the behavior and state of a core object or a view may be extracted and turned into modules that in turn may be used to extend existing core objects or views, or simply used as basic components to create new viewpoints, classes, and objects.

It is noteworthy that VOC experience capitalization is different from pure view cloning where a view is created from an existing one. Indeed, the former technique goes beyond the copying process performed when cloning a view. It is an abstraction approach that may be defined as a reverse instantiation relationship (generalization).

Since the experience capitalization is aimed to developers of VOC systems, we provided in the VOC formalism and prototype with an operation named modulify, that accepts parameters to indicate which properties, methods will be included in the generated module (see the second subsection of Section 3).

3 VOC algebra and formalism

3.1 VOC concepts formulation

Our goal is to formalize the VOC paradigm. The formalization creates the foundations upon which one can build, extend the paradigm, and also be able to easily reason with and handle existing components. We present below a VOC algebra based on the mathematical set theory. Later on, we use this algebra as a formal tool and show proof of concept.

3.1.1 Universe of properties

$U_p = \{p_1, p_2, p_3, \ldots\}$ where p_i is an attribute (property).

An attributed p is formally represented by the triplet: $p =< n, d, v >$, where n is the name of the attribute, d the domain of definition of this attribute, and v the value of the attribute.

3.1.2 Universe of methods

$U_m = \{m_1, m_2, m_3, \ldots\}$ where m_i is an attribute.

In our system, a method is represented by a sextuplet of input parameters, output parameters, read only attributes and write attributes. The formalization of the representation of a method m_i is as follows:

$m =< name, core, params_{in}, params_{out}, p_r, p_w >$. With $name$ the name of the method, $core$ is the body of m and $(params_{in}, params_{out}, p_r, p_w) \in (U_p)^4$ s.t.:

1. $params_{in}$: is a set of input parameters;
2. $params_{out}$: is a set of output parameters;
3. p_r: is a set of read only attributes: attributes whose values are used in the method core but not overridden;
4. p_w: is a set of write attributes: attributes that are eventually overridden in the method.

3.1.3 Universe of objects in the classical object oriented programming paradigm

We refer to the universe of objects of the classical OOP paradigm as U_b and each object of this universe is represented by a quintuplet as follows:

$$U_b = \{b | \ b = < n, m_{pr}, \ m_{pc}, p >, s.t. n \ is \ a \ string, (m_{pc}, m_{pr}) \subseteq (U_m)^3 \ and \ \ p \subseteq U_p \ \}$$

Where:

1. n : is the name of the base object o;
2. m_{pc} : is as set of public methods of the object;
3. m_{pr} : is as set of methods that are not accessible outside of the object (private methods);
4. p : is as set of object properties.

Note: *The name n has to be unique inside the universe U_b, and a dot (.) is used to reference a component of a given object.*

For instance, public and private methods of an object b are respectively referenced by $b.m_{pc}$ and. Also, we will refer to $b.m_{pr} \cup b.m_{pub}$ simply as $b.M$.

3.1.4 Viewpoints description language

A viewpoint is composed of a set of variables and a set of methods. While the variables are identical to those of objects in OOP, the methods are slightly different. The difference does not reside in the contract/functionality a method offers but in how it is integrated in and interacts with the already existing methods. This integration and interaction is enforced by a set of requirements.

In order to formalize the description of viewpoints, we propose a description language endowed with syntactical and semantic constructs. While the syntactic part provides the characteristics and features of viewpoint's structure, the semantic part associates to them a meaning under the VOC domain.

Syntax of viewpoint

A viewpoint is composed of a set of variables and a set of methods. The variables are properties that extend the state of a core object and the methods are procedures and functions intended to extend the behaviour of an existing core

object. Thus, the viewpoint universe, denoted U_{VP}, is composed of triplets as follows:

$$U_{VP} = \{vp \mid vp = < P, M, CTS, \text{status} > \; s.t. \; P \subseteq U_p, M \subseteq U_m\} \text{ and } CTS \subseteq U_{ctr}$$

Where U_{ctr} is the universe of constraints materializing concern requirements that may be associated to a method, and status may have one of the four values: $\{enabled, disabled, activated, deactivated\}$. A requirement is composed of three parts: (1) a target, (2) a type, and (3) a core. The target selects methods from the viewpoint which are concerned by the requirements. The core is where constraints on the selected methods are defined. In the core we may find method matchers that filter methods from the core object, and operations that let us define how the method will be integrated with existing core methods which is limited to four different kinds of integration*: execute before, execute after, execute around and replace*. We also find in the core the list of methods/attributes required by the targeted methods. Furthermore, a method matcher is composed of the following sub-matchers: (1) parameters-in matcher, (2) parameters-out matchers, and (3) method name matcher. As their respective names suggest it, each of those matchers is intended to identify one part of the essential parts of a method which are: input parameters, outputs parameters and the method name.

Note: *A method definition in a viewpoint may have an empty set of requirements.*

We illustrate the syntax of a viewpoint through an example written in an internal DSL:

```
viewpoint m_loan_vp(extended_object_core alias co) do
  var penalties as Double
  var new_tax   as Double

  def change_due_date(date as Date)
    co.due_date = date
  end

  def calculate_interest as Double()
    interest = co.calculate_interest
    # apply new tax
    return interest -= new_tax*interest/100
  end

  requirements_all do
```

```
   with_method change_due_date do
     #     scope of the method
     type :public
     # check if co (core object) contains a
     # variable due_date of type Date
     verify_presence_of :name => co.due_date as var%Date
     # not chained with other core
     chain :none
   end

with_method calculate_interest do

    type :public
     verify_presence_of :name => co.calculate_interest
           as method
           {
            :in_params => :none
            :out_params => interest as var%Double
           }
     # check if co contains a variable named
     # interest of type Double
     verify_presence_of :name => co.interest as var%Double
    when :replace
   end # end with_method
  end # end requirements
end # end viewpoint
```

Figure 2. Example of a viewpoint written in a Ruby-like domain specific language.

The code of Figure 2 that represents a viewpoint declaration, m_loan_vp, which is composed of three complementary types of declaration statements respectively related to: (1) variables, (2) methods, and (3) requirements.

New variables are declared using the reserved keyword `var`. For example, in `m_loan_vp`, we have declared two variables: penalties and new_tax. These variables will be available in the views that are generated from `m_loan_vp`.

Methods, however, are declared using the delimiters def and end. Input parameters of the method are declared between parentheses while the output parameter, is eventually declared using the keyword as followed with the data type of that parameter.

In this example, we also illustrated the declaration of requirements as expected by the newly added methods. This is done using the `requirements_all do ... end` clause. As it is noticed in Figure 2, this clause accepts the declaration of the requirements of a method in one unique bloc delimitated by `with_method name do ... end`; where name is the name of the current method. Inside this bloc, we may

have tree constructs: the first one used to verified the presence of a variable or a method in the core object to be extended (performed at runtime) by means of `verify_presence_of` clause where regular expressions-like/string matchers may be used to design the method or methods that are targeted by the requirements. The second construct is used to declare how the current method chains with existing methods from the core object using predefined constants: `:before, :after, :around, :replace,` and `:none` if no chaining is required. The third construct, named type, defines the type of the given method and may have one of the following three predefined values: *public* for a public method and *private:* for a private method. It is to notice that the second method (`calculate_interest`) of `m_loan_vp` viewpoint is replacing an existing method in the core object.

Semantic of viewpoint

The meaning of a viewpoint resides in its interpretation composed of a set of views. This interpretation is based on two elements: (1) the interpretation domain which consists of all views, and (2) an interpretation function which associates for each viewpoint a set of elements within the interpretation domain such as defined below.

Universe of views in the VOC paradigm

In VOC, a view is composed of the same components as a core object in OOP plus a status.

$$\mathrm{U_o} = \{\mathrm{o} \mid \mathrm{o} = < C, V, E, A, v > \; s.t. \; C \in \mathrm{U_b}, \; V \subseteq (U_b), \mathrm{E} \subseteq V\}$$

Where C is the core of the object o, V the set of all views of o, E the set of enabled views ($status = enabled$), A the set of activated views (a view has to be enabled in order to be activated $A \subseteq E$), and v is a hash-key representing the current version of the object o. For the sake of concision, we will use $v.status$ to refere to the status of the view v.

It is noteworthy that the version number of an object is computed by a function, say θ, from the following elements:

- The name of the core object C;
- The different names of the views in V;
- The different names of the views in E;
- The different names of the views in A.

In the remaining of this paper we will use a dot (.) to make reference the different components of an object. For example, the core object of the object o, is referenced by $o.C$.

Instantiation relationship (Interpretation function)

Given a viewpoint vp from U_{VP} and an object o from U_o. A view v from $o.E$ ($v \in o.E$) is instance of vp denoted by $v \ll vp$, if and only if the following conditions are satisfied:

- $\forall\, m' \in vp.M: \exists\, m'' \in (v.m_{pc} \cup v.m_{pr})$ such that $m'.core = m''.core$; More specifically:
  - if $m'.type =' public'$ than $m'' \in v.m_{spc}$;
  - if $m'.type =' private'$ than $m'' \in v.m_{pr}$;
- $\forall\, p' \in vp.P : \exists\, p'' \in v.p$ such that: $p' = p''$;
- $\forall\, cts \in vp.CTS$:
  - $\exists\, m \in vp.M$ such that: $m \subseteq cts.target$;
  - $\forall\, r \in cts.core$: $r.target \subseteq vp.M, r.matchers \subseteq o.C.M$ and $r.required \subseteq (o.C.M \cup o.C.P)$;

Figure 3 presents the `o_bank_loan` core object and Figure 4 presents the `m_loan_v` view that is instantiated from the viewpoint of Figure 2.

```
o_bank_loan = Object.new

o_bank_loan.instance_eval do
  var due_date      as Date
  var loan_date     as Date
  var interest      as Double
  var amount        as Double
  var deposit       as Double
  var interest_rate as Double

  def calculate_interest as Double()
   return interest
  end

  def set_loan_amount(amount as Double)
   this.amount = amount
  end
end
```

Figure 3. Bank Loan Core object

```
view m_loan_v extend m_loan through m_loan_vp do
  var penalties as Double
  var new_tax   as Double

  def change_due_date(date as Date)
    co.due_date = date
  end

  def calculate_interest as Double()
    interest = m_loan.calculate_interest
    return interest -= new_tax*interest/100
  end
end # end view
```

Figure 4. A Loan Manager View.

The core object `o_bank_loan` is a standard object that contains five properties (`due_date, loan_date, interest, amount, deposit, interest_rate`) and two methods (`calculate_interest, set_loan_amount`). This object is then extended by the `view m_loan_v` of Figure 4. As it can be noticed from this figure, the keyword `extend` is used to indicate the core object extended and the keyword `through` to indicate the viewpoint that used to instantiate the given view.

3.1.5 View Clustering

In VOC, a cluster is a collection of views related to different objects of a system. Formally, a cluster, denoted by f, is represented by $f = \{A | A =< o, \{v_i | v_i \in o.V\} > s.t.\ o \in U_o\}$.

It is noteworthy that: (1) an element A of a cluster f is referenced by $f.A$; (2) object and views of a view in A are respectively referenced to by $A.o$ and $A.V$.

For the sake of simplicity, a couple $< o, V >$ of a cluster f is referenced to by $f[i]$ where i is the position of the given couple when they are lexicographically ordered inside the cluster f. We have also the following concepts:

a. $|f|_o$: Number of objects for which at least one view is contained in f;

b. $|f|$: Number of views contained in f.

3.1.6 Subject Space

In VOC, a subject represents an entity that is attempting access any resource of a given system. This can be a user, a process, etc. We represent a subject by u which belongs to the universe of all subjects $\mathrm{U_u}$.

3.2 VOC operations

3.2.1 Object-level operations

Given an object $o = < C, V, E, A, v >$ the operations that can be performed on this object are presented below.

Adding a view

Adding a view $v_3$to the object o consists at modifying the sets $o.V$. The operation is represented by: $addView(o, v) = < C, V \cup \{v\}, E, A, \theta(C, V \cup \{v\}, E, A) >$ and the operation set the status of v to $disabled(v.status = disabled)$. This operation is valid if and only if the following conditions are satisfied:

- $o \in U_o$;
- $v \in U_b$;
- $\forall v_i \in o.V : v_i.n \neq v.n.$

Otherwise, the operation is not valid.

Removing a view

Removing a view v from the object o consists at modifying the set $o.V$. This operation is represented by: $removeView(o, v) = < C, V - \{v\}, E, A, \theta(C, V - \{v\}, E, A) >$. This operation is valid if and only if the following conditions are satisfied:

- $o \in U_o$;
- $v \in o.V$;
- $\forall v_i \in o.A : v_i.n \neq v.n$;
- $\forall v_i \in o.E : v_i.n \neq v.n.$

Otherwise, the operation is not valid.

Enabling a view

Enabling a view v in the object o consists at modifying the set $o.E$. The operation is represented by: $enableView(o,v) = < C, V,\ E \cup \{v\}, A, \theta(C, V, E \cup \{v\}, A) >$ and the operation sets the status of v to $enabled$ $(v.status = enabled)$. Enabling a view is valid only if:

- $o \in U_o$;
- $v \in o.V$ and $v \notin o.E$;
- $v.status \neq enabled$.

Disabling a view

Disabling the view $v$in the object o consists at modifying the sets $o.E$ and $o.V$. The operation is represented by: $disableView(o,v) = < C, V,\ E - \{v\}, A, \theta(C, V, E - \{v\}, A) >$ where the status of the view v is set to $disabled$. This operation is valid if and only if the following conditions are satisfied:

- $o \in U_o$;
- $v \in o.E$ and $v.status = enabled$;
- $v \notin o.A$ and $v.status \neq activated$.

Activating a view

Activating the view $v$in the object o consists at modifying the sets $o.E$ and $o.V$. The operation is represented by: $activateView(o,v) = < C, V,\ E, A \cup \{v\}, \theta(C, V, E, A \cup \{v\}) >$. This operation sets the status of the view and v to and $activated$ $(v.status = activated)$ and is valid if and only if the following conditions are satisfied:

- $o \in U_o$;
- $v \in o.E$ and $v.status = enabled$;
- $v \notin o.A$ and $v.status \neq activated$.

Deactivating a view

Deactivating the view v in the object o consists at modifying the sets $o.E$ and $o.V$. The operation is represented by: $deactivateView(o,v) = < C, V,\ E, A - \{v\}, \theta(C, V, E, A - \{v\}) >$. This operation sets the status of and v to and $deactivated$ $(v.status = dectivated)$ and is valid if and only if the following conditions are satisfied:

- $o \in U_o$;
- $v \in o.A$ and $v.status = activated$.

Cloning a view

Cloning the view v in the object o1 to the object o2 consists at creating a view in o2 that is identical (except the name) to v. The operation represented by: $cloneView$, is valid if and only if the following conditions are satisfied:

- $(o_1, o_2) \in (U_o)^2$;
- $v \in o_1.V$;
- $\forall v' \in o_2.V : v'.n \neq name$.

Once the operation is performed under the conditions above, we will have: $v'' = < name, v.m_{pc}, v.m_{pr}, v.p, v.status > and\ v'' \in o_2.V$. It is noteworthy that a view may be cloned to the same object.

Abstracting views and core objects

The operation that "modelifies" a view or part of it, consists at extracting methods and attributes from it and creating a new module. This operation, represented by: $modelify(o, v) = < ModuleName, atts, mths >$ is valid if and only if the following conditions are satisfied:

- $o \in U_o$;
- $v \in o.V$;
- $\forall p \in atts: p \in v.P$;
- $\forall m \in mths: m \in (v.m_{pc} \cup v.m_{pr})$.

It is to notice that the second parameter of modelify may be the core object of the object represented by the first parameter. Furthermore, another version of modelify is provided; this version accepts a hash as a third parameter. The hash may have only one cay with one of the two values: {'except','only'}. If the key value is 'except', the data associated to this key, which is an array of methods and attributes, will be excluded from attributes and methods that will make its way to the generated module. Else (key value is 'only'), the associated array contains the methods and attributes of the view/object that will be included in the generated module.

3.3 Cluster-level operations

3.3.1 Adding a view

Given a cluster f from U_f, and an object o from U_o, adding a view v of o to f is possible if and only if the following condition is satisfied: $\nexists A \in f : \exists v' \in A.V \; s.t. \; (o.v = A.o.v) \wedge (v'.n = v.n)$.

This operation is represented by $addViewToFacet(f, o, v)$.

3.3.2 Removing a view

Given a cluster f from U_f, and an object o from U_o, removing a view v from f, represented by $removeViewFromFacet(f, o, v)$, is possible if and only if the following conditions are satisfied:

- $\exists A \in f \; s.t.$:
- $A.o.v = o.v$;
- $\exists v' \in A.V \; s.t. \; v'.n = v.n$.

3.4 System-level operations

In our context, a system, represented by $s \in U_s$ where U_s is the universe of all systems, is composed of a set of clusters ($s = \{f \mid f \in U_f\}$).

System level operations are the operations that let us add and remove clusters to/from the given system. Those operations are defined below.

3.4.1 Adding a cluster

Given a system s from U_s and a cluster f from U_f, and an object o from U_o, adding a cluster f to s is possible if and only if the following condition is satisfied: $\forall f' \in s : f \neq f'$.

3.4.2 Removing a cluster

Given a system s from U_s and a cluster f from U_f, and an object o from U_o, removing a cluster f from s is possible if and only if $f \in s$.

3.5 Partial order over the universe of clusters U_f

We are defining a generalization relationship among clusters which is a partial order over the universe of clusters. This relationship is based on low level component generalization: generalization relationship among views.

3.5.1 View generalisation relationship

Generalization relationship among views organize views into hierarchies. Given two views v_1 and v_2, v_2 is said more general than v_1, denoted $v_1 <_v v_2$, if and only if the following conditions are satisfied:

- $v_1.m_{pc} \subseteq v_2.m_{pc}$;
- $v_1.m_{pr} \subseteq v_2.m_{pr}$;
- $v_1.p \subseteq v_2.p$.

If the above conditions are satisfied, we also say that v_1 is more specific than v_2.

The generalization relationship $<_v$ is a partial order on the universe ${U_b}^p$.

3.5.2 Cluster-based generalization relationship

Cluster generalization relationship is based on the generalization relationship of active views included in clusters.

Given two clusters f_1 and f_2 from U_f and $k = |f_2|_o$, f_2 is more general than f_1, denoted $f_1 <_f f_2$, if there exist k sets $\{ I_i | I_i \subseteq f_2[i].V \; with \; i \in [1..k]\}$ and k subjective functions $S_\psi =< \psi_1, \psi_2, \dots, \psi_k >$ such that:

- $\forall \psi_i \in S_\psi$:
  a. $\psi_i : I_i \rightarrow f_1[i].V$;
  b. $\forall v \in I_i : \psi_i(v) <_v v, with \; i \in [1..k]\}$.

Generalization relationship among clusters let us organizing clusters into hierarchies.

4 Running example

Hereafter, we illustrate cluster-based VOC using a bank account example which is a canonical example in the object-oriented programming literature.

Figure 5 presents two objects *account_o* and *bank_loan_o* each with a set of methods and attributes dispatched into views: the *account_o* object is extended with two views (*c_account*_v and m_*account*_v) and the *bank_loan_o* object is extended with two views (*c_loan_v* and *m_loan_v*). Those views are than assigned to clusters: *Customer Cluster* and *Manager Cluster*. Those clusters are then composed with two permission modes (deny and grant) to form four permissions. Finally, those permissions are assigned to user via two roles: *Manager* role and *Customer* role (see Figure 5).

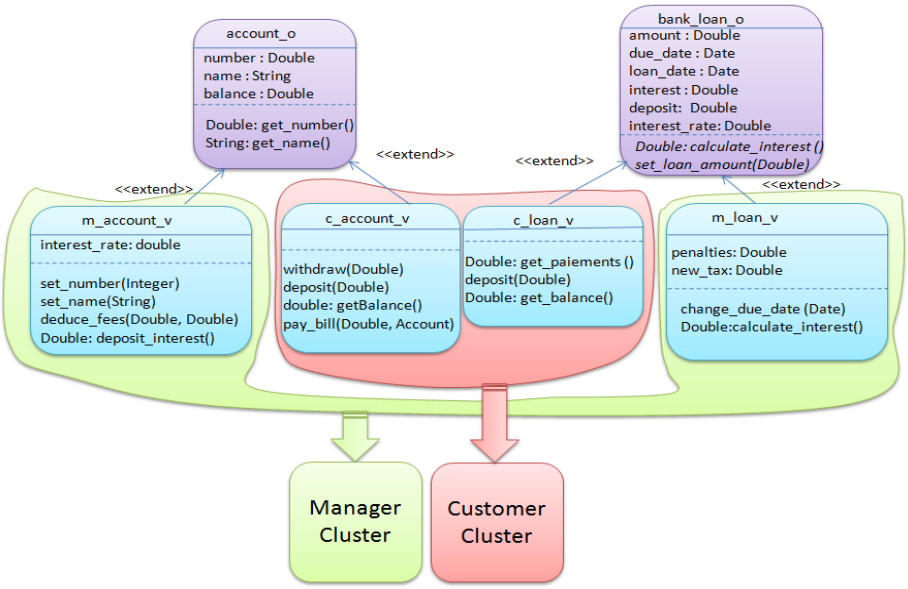

Figure 5: Bank account and Loan example.

To implement cluster based VOC (C-VOC) policy, a Ruby DSL (Domain Specific Language [Consel04, Thomas09]) is used. The concepts behind this implementation are used to illustrate the example presented in Figure 5.

```
# clusters
cluster ManagerCluster do
  :Account  =>{MAccount}
  :BankLoan =>{MLoan}
end
cluster CustomerCluster do
  :Account  =>{CAccount}
  :BankLoan =>{CLoan}
end
# permissions
permission MFGrant do
  :Cluster => ManagerCluster
  :Mode  => Grant
end
permission MFDeny do
  :Cluster => ManagerCluster
  :Mode  => Deny
end
permission CFGrant do
  :Cluster => CustomerCluster
  :Mode  => Grant
end
permission CFDeny do
  :Cluster => CustomerCluster
  :Mode  => Deny
end
# roles
role Customer do
  :permissions =>[MFDeny,CFGrant]
end
role Manager do
  :permissions =>[MFGrant,CFGrant]
end
# assigning roles to subjects
  subject1.assign{Manager, Customer}
  subject2.assign{Customer}
# activating a role
  subject1.activate{Manager}
# invoke a method role
  subject1.setName(‘‘Toto’’)
# deactivating a role
  subject1.deactivate{Customer}
```

Figure 6: Cluster Attribution to users.

The source code of Figure 6, shows a partial view of how the C-VOP attributes clusters to different users via the concept of roles that abstracts permissions on clusters (grand, deny). The choice to implement the model as an internal DSL, is mainly motivated by the need to reflect in the model the domain business by the fundamental domain concepts (clusters, views, ...) at a higher level of abstraction.

5 Conclusion

Our work addresses the problem of VOC modeling and formalisation using algebraic operators and first order logic. The model we propose offers different functional aspect/views to different client programs where each object offers dynamically a set of views. Clients can call any function and get the right answer dynamically. Indeed, views are code fragments, which provide the implementation of different functionalities for the same object domain and theses views can be used as a units for distribution to improve performance issues. In this paper we also showed via an example the key concepts of our model with a flexible and straightforward DSL.

We need in the future more investigations to validate the proposed model and test its performance, scalability and stability with real word projects. Also, we plan to offer integrate versioning and transaction management aspects as well as a fine-grained end-user security enforcement strategy.

References

[Cousel04] Charles Consel, "*From a Program Family to a Domain-Specific Language*" book "*Domain-Specific Program Generation*", ISBN978-3-540-22119-7, pages 19-29, Springer Berlin/Heidelberg, 2004.

[Carlson79] Walter Carlson, "*Business Information Analysis and Integration Technique (BIAIT)- The New Horizon*", DATABASE, Spring 1979, pp. 3-9.

[Coad00] Peter Coad, Eric Lefebvre, and Jeff De Luca. "*Java Modeling In Color With UML: Enterprise Components and Process*". Prentice-Hall, 2000

[Bass03] L. Bass, P. Clements & R. Kazman, "*Software Architecture in Practice*", Addison-Wesley, 2003.

[Dasgupta91] S. Dasgupta, "*The Nature of Design Problems*", in Design Theory and Computer Science, Cambridge University Press, 1991, pp. 13-35.

[Harrison93] W. Harrison and H. Ossher, "Subject-oriented programming: a critique of pure objects," in Proc. of OOPSLA'93, pp. 411-428.

[Kiczales97] G. Kiczales et al., "Aspect-Oriented Programming," in Proc. of ECOOP 97, Springer-Verlag LNCS 1241.

[Mcheick06] H. Mcheick, "*Distribution d'objets en utilisant les techniques de développement orientées aspect*", Ph.D Thesis, 273 pages, Université de Montréal, Québec, Canada, 2006.

[Mcheick07] H. Mcheick, H. Mili, H. Msheik, A. Sioud, and A. Bouzouane, "*AspectGC: an Aspect Garbage Collection for Object Lifecycle Management*", Proceedings of Third International Conference on Intelligent Computing and Information Systems (ICICIS07), Sponsord by ACM SIGART and SIGMIS, pp.150-157, 15-18 mars 2007, Ain Shams University, Cairo, Egypt.

[Mcheick09] Mcheick, H., M. Adda, H. Mili, and M. Badri, "*Dynamic Object Behaviours in Separation of Concerns Based Systems*" ,International Conference on Software Engineering Research and Practice (SERP'2009), Las Vegas (USA), 2009.

[Mens02] T. Mens and M. Wermelinger, "*Separation of concerns for software evolution*", *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 14, no. 5, pp. 311-315, 2002.

[Mens05] T. Mens, M. Wermelinger, S. Ducasse, S. Demeyer, R. Hirschfeld, M. Jazayeri, "*Challenges in software evolution*", Eighth International Workshop on Principles of Software Evolution, 5-6 Sept. 2005 pp. 13-22, 2005.

[Mens08] T. Mens and T. Tourwé, "*Evolution issues in aspect-oriented programming*", Software Evolution Book, ISBN 978-3-540-76439-7, Springer Berlin Heidelberg, pp. 203-232, 2008.

[Van03] F. Van Rysselberghe and S. Demeyer, "*Reconstruction of successful software evolution using clone detection*", Proceedings of Sixth International Workshop on Principles of Software Evolution, pp. 126-130, 2003.

[Thomas09] Dave Thomas, Chad Fowler, and Andy Hunt, "*Programming Ruby 1.9: The Pragmatic Programmers' Guide*", The Pragmatic Programmers, LLC, ISBN: 9781934356081, 2009.

About the authors

Mehdi Adda, professor of computer science at the University of Quebec at Rimouski, Rimouski, Canada. His principal research interests lie in the fields of Software and Web engineering, data mining and knowledge discovery, Aspect oriented programming and distributed computing, Web personnalization and recommendation. Contact him at adda@ieee.org or mehdi_adda@uqar.qc.ca.

Hamid Mcheick, professor of computer science at the University of Quebec at Chicoutimi, Chcoutimi, Canada. Professor Mcheick is interested in evolution and distributed object and aspect oriented applications.

Hafedh Mili, professor of computer science at the University of Quebec at Montreal, Montreal, Canada. Throughout his academic career, he worked on a variety of subjects, starting with knowledge representation, object-oriented software engineering, aspect-oriented development, service-oriented computing, and business process engineering.