Journal of Universal Computer Science, vol. 13, no. 6 (2007), 786-805
submitted: 19/1/07, accepted: 22/3/07, appeared: 28/6/07 © J.UCS

RE-AspectLua - Achieving Reuse in AspectLua

Thaís Batista
(Federal University of Rio Grande do Norte, Brazil
thais@ufrnet.br)

Maurício Vieira
(Federal University of Rio Grande do Norte, Brazil
mauricio.vieira@gmail.com)

Abstract: AspectLua is a Lua-based dynamic aspect-oriented language that follows the original AspectJ concepts. It suffers from the same problem of AspectJ-like languages with regard to limitations in terms of aspect reusability, modularity and heterogeneous interaction. In this paper we propose RE-AspectLua, a new version of AspectLua that combines aspect interfaces with abstract joinpoints and the use of a connection language, the Lua language, to instantiate, at application composition time, the abstract joinpoints. Thus, the connection language defines the composition of reusable aspects with base code. Using these concepts RE-AspectLua intends to break away from the syntactically manifest coding of aspects in which joinpoints are hard-coded into aspects, thereby promoting general reusability and the heterogeneous composition of an aspect with different base codes. In order to illustrate RE-AspectLua concepts we present two case studies.

Keywords: AOP, Lua, dynamic aspects, reusability, heterogeneity
Categories: D.2.3, D.3.3

1 Introduction

Aspect-Oriented Programming (AOP) [kiczales 97] has emerged to modularize elements that cut accross the basic decomposition modules of a system and that traditional object-oriented programming cannot modularize. In order to modularize such crosscutting concerns AOP introduces a new abstraction named *aspect*. The modularization of these concepts aims to reduce development costs and to improve comprehensibility, reusability and adaptability. AspectJ [Kiczales et al. 01] was a pioneer aspect-oriented language that introduced a set of concepts: *joinpoints*, *advice*, *pointcuts* and *intertype declarations*. The main problem of traditional AOP approaches following AspectJ original concepts is that they promote a new modularity mechanism but the internal aspect code contains a direct reference to the element that the aspect crosscuts. The first drawback of this approach is that it limits the aspect reuse in different context. As the aspect internal code is tightly associated with the element that it crosscuts, it cannot be reused in other context, with other elements. The second drawback is that the aspect is coupled to details of the base code that the base coder programmer is free to change. The third drawback, also related to the first one, is that the way that the aspect composes with other elements is also fixed within the aspect code. Ideally, an aspect should compose with different elements, in a different way. This heterogeneous composition ability is essential to promote aspect reusability. Reusability can be defined as the ability to reuse the aspect behaviour (advice) in different compositions. Heterogeneity is the ability to use the aspect behaviour in a different way for each composition where the aspect is applied.

AspectLua [Cacho et al. 05] is an AOP language based on the Lua programming language [Ierusalimschy 06]. Lua is an interpreted and dynamically typed language. These features introduce a different style for aspect-oriented programming where dynamism is a key issue, weaving is done at runtime and both the basic elements and aspects can be inserted into and removed from the application at runtime. In addition, the Lua philosophy is to be simple and small and AspectLua keeps this philosophy. AspectLua is built upon a meta-object protocol, LuaMOP [Fernandes and Batista 04] that provides an abstraction over the reflective features of Lua and allows application methods and variables to be affected by the aspect definition. AspectLua follows AspectJ concepts and, as a consequence, presents the aforementioned reusability limitations.

In order to promote the reusability of aspects and the ability to support heterogeneous composition, in this work we defined important abstractions useful to a better separation of concerns and we implement these abstraction in a new version of AspectLua, named *RE-AspectLua*. In RE-AspectLua aspects are defined at *aspect specification time* and their instantiation is defined later at *application composition time*. An aspect is defined by a set of *aspect interfaces*. Each aspect interface specifies an abstract join point and an advice. Abstract join points instantiation is defined at application composition time via a *connection language*. The connection language is the Lua language itself since it is a scripting language. The fact of using a scripting language as the connection language introduces a great flexibility to the aspect composition process as the glue code can contain conditional statements to decide, at runtime, which composition must be defined. In addition, as Lua is a dynamically typed and interpreted language, it allows the dynamic connection and even the adaptation of the aspect connection at runtime. This is a new application domain of scripting languages: as a glue language between aspects and classes.

The use of a scripting language to support the composition between aspects and advised code also benefit adaptability and evolution. The independence of an aspect from the advised code allows the evolution of both the aspect and the base code. In addition, the composition language can define glue code that address possible composition mismatches between the two codes. Furthermore, the use of an interpreted language improves even more adaptability support by allowing adaptation at runtime.

Some recent work also address some limitations of the AspectJ-based AOP [Aracic et al. 05], [Gal et al. 03], [McDirmid and Hsieh 03], [Suvée et al. 05], [Suvée at al. 03], [Sullivan et al. 05]. However, none of them is based on an interpreted language and provides the set of AOP features supported by AspectLua (see [Cacho et al. 05]). The related work in terms of AOP languages built on top of scripting languages [Dechow 04], [Bryant and Feldt 02], [Hirschfeld 03], are based on AspectJ concepts and none of them promotes reusability of aspects that is the main goal of RE-AspectLua.

Although some researchers do not associate the use of AOP with scripting languages because, in general, such languages are not intended to write large and complex software systems, we argue that the benefits of AOP target not only large and complex software systems but it also has an important role in embedded systems where the problem of composition is even harder. This type of system needs to maintain the application code small. Thus, separation of concerns is essential and AOP is a good technique to manage crosscutting concerns in embedded systems [Sztipanovits et al. 02].

This work is organized as follows. Section 2 presents some background concepts about Lua, AspectLua and LuaMOP. It also introduces a running case study that is used through the paper. Section 3 presents our proposal to support reuse and heterogeneous interaction in AOP and how they are supported by RE-AspectLua. Section 4 re-examines the case study previously introduced and also presents a new case study that exploits the versatile support of a scripting language to define heterogeneous aspectual composition. Both the case studies illustrate the concepts of RE-AspectLua. Section 5 discusses related work. Section 6 contains the final remarks.

2 Aspect-Oriented Programming

This section presents the background of this work. Section 2.1 presents the main concepts of Lua and AspectLua. Section 2.2 presents LuaMOP, the meta object infrastructure used by AspectLua.

2.1 Basic Concepts

AspectLua [Cacho et al 05] is an aspect-oriented programming language based on the Lua language. AspectLua implementation uses LuaMOP, a meta-object protocol that allows the dynamic weaving of aspects and components. Lua is dynamically typed, which means that variables are not bound to types but each value has an associated type. Lua syntax and control structures are similar to those of Pascal. Some non-conventional features of Lua: (i) functions are first-class values; (ii) Lua tables are the main data structuring facility in Lua. Tables implement associative arrays, are dynamically created objects, and can be indexed by any value in the language, except nil. Tables may grow dynamically. Lua offers reflective facilities and metatables are the main reflective abstraction in Lua. Metatables allow modification of the behavior of a table. More details about Lua can be found in [Ierusalimschy 06].

AspectLua combines a set of features to make AOP easier and powerful: (i) insertion and removal of aspects at runtime, (ii) the definition of precedence order among aspects, (iii) the possibility of using wildcards, (iv) the possibility of associating aspect with undeclared elements (*anticipated join points*), and (v) a dynamic weaving process via a meta-object protocol.

AspectLua follows AspectJ philosophy and, as a consequence, presents the same problems related to lack of reusing support. In particular, AspectLua supports the following set of join points:

- *call* for method invocations;
- *callone* for the specification of aspects that must be executed only once;
- *introduction* to add functions in objects (intertype declarations);
- *get* and *set* join points to capture operations on variables.

AspectLua aspect also defines an advice to be executed when the set of join points specified is reached.

```
a = Aspect:new()
a:aspect( {name = 'aspect-name'},
          {pointcutname = 'pointcut-name',
           designator = 'designator-type',
           list = {'some-class.method1()','anotherclass.*'},
          {type ='advice-type', action = advice} )
function advice()
 -- do something
end
```

Figure 1 : Metatable definition

Figure 1 illustrates the syntax of aspects definition in AspectLua:

- the first aspect parameter is its name.
- the second parameter is a Lua table that defines the elements of the join points: its name, its designator type (*call*, *callone*, *introduction*, *get* or *set*), and functions or variables that must be intercepted. The *designator* field indicates the join point type. The *list* field contains the functions or variables that will be intercepted. The '*' wildcard can be used. For instance, *another-class.** means that the aspect must be applied to all methods of a class *another-class*.
- the third parameter is a Lua table that defines the elements of the advice: the type (*after*, *before* or *around*), and the action that is executed when a join point is reached. In the example, the action is the function declared with the name *advice*.

As the aspect defines an explicit association with the affected components, aspect reusing is not possible. Also, the heterogeneous composition is limited in AspectLua. To illustrate the lack of aspect reusing in AspectLua, let us consider a simple banking application. Suppose that a client wishes to register the access to the bank component (Figure 2). It has two operations: *deposit* and *cash*.

```
Bank = {balance = 0}
function Bank:deposit(amount)
   self.balance = self.balance + amount
 end
function Bank:cash(amount)
   self.balance = self.balance - amount
end
```

Figure 2: Bank Component

Figure 3 shows two aspects, written in AspectLua, that affect the *bank* component. These two aspects are needed to log methods of the Bank component.

laspect_before aspect (lines 1-6) determines that the *cash* method is preceded by the *logbalance* advice. The second aspect (lines 8-13) is used to advise *deposit* and *cash* just after they are called.

```
1 a = Aspect:new()
2 a:aspect({name = 'laspect_before'},
3          {pointcutname = 'logged_methods',
4           designator = 'call',
5           list = {'Bank.cash'},
6          {type ='before', action = logbalance} )
7
8 b = Aspect:new()
9 b:aspect( {name = 'laspect_after'},
10          {pointcutname = 'logged_methods',
11           designator = 'call',
12           list = {'Bank.cash','Bank.deposit'},
13          {type ='after', action = logbalance} )
14
15 function logbalance(self)
16    print ('Balance is now: ', self.balance)
17 end
```

Figure 3: Bank Component Logging

There is a clear lack of reuse illustrated by the example in Figure 3. The aspect code is directly bound to its pointcuts, thus the aspect must be directly changed if one wishes to affect other joinpoints, and the heterogeneous interaction, i.e. aspect acting in a different way (before and after), can be achieved only by code duplication.

AspectLua provides the concept of *anticipated joinpoints*. Anticipated joinpoints are join points related to elements that do not exist at the *aspect specification time*. This joinpoint is useful to avoid the need of loading an application code before loading the aspect code that contains a joinpoint to the application. This facility is useful to allow lazy loading at the *aspect execution time*. More details can be found in [Cacho et al. 05].

2.2 LuaMOP

The Aspect weaving process used in AspectLua is supported by LuaMOP. LuaMOP is a meta-object protocol that supports the creation of a meta-representation to each element that composes the Lua runtime environment: variables, functions, tables, userdata and so on. Each element is represented by a meta-class that provides a set of methods to query and to modify the behavior of each element of the base class. They are organized in a hierarchical way where *MetaObject* is the base meta-class (Figure 4). Derived from this meta-class are *MetaVariables*, *MetaFunctions*, *MetaCoroutine*, *MetaTable*, and *MetaUserData* meta-class. Furthermore, LuaMOP also provides a *Monitor* class to monitor the occurrence of events in the Lua runtime environment.

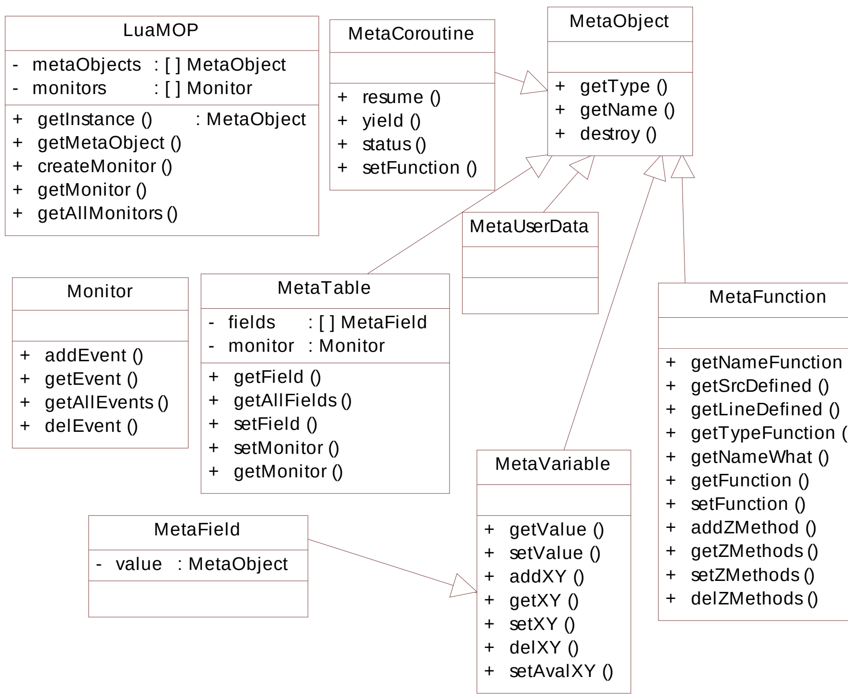

Figure 4: LuaMOP class diagram. X should be replaced by Pre and Pos, Y should be replaced by Get and Set and Z should be replaced by Pre, Pos or Wrap

The meta-representation provided by LuaMOP is created via the invocation of the *getInstance(instance)* method. This method returns the meta-object corresponding to the object with name or reference described by the instance parameter. This meta-object is an instance of a meta-class described above. For each meta-class there are methods that describe it and that support changing the behavior of a meta-object. Thus, *getType()* and *getName()* methods can be invoked by all meta-classes, since these methods are part of the MetaObject meta-class. These methods return, respectively, the meta-object type and name. The *destroy()* method is used to disconnect the meta-object from the base object and to destroy the meta-object. The *getInstance(instance)* method can also be invoked, using as an input parameter a non-determined name. For instance: *getInstance("string.*")* returns a Lua table with meta-objects that represent the functions of the string package.

For the sake of brevity, only some *MetaFunction* methods are commented here, an extensive explanation of LuaMOP can be found in [Cacho et al. 05]. The *MetaFunction* meta-class offers the *addPreMethod*, *addPosMethod*, and *addWrapMethod* methods. These methods define the place where the behavior is added: Pre(before), Pos(after), and wrap the execution of a function. An example of the use of these functions is illustrated in Figure 5.

```
1 function reglog(self,value)
2    print("Deposited Value: ",value)
3 end
4 metafun = LuaMOP:getInstance("Account.deposit")
5 metafun:addPosMethod(reglog)
6 Account:deposit(10)
```

Figure 5: LuaMOP example with addPosMethod [Cacho et al. 05]

The meta-object is obtained at line 4. At line 5, the *addPostMethod* method is invoked to add the *reglog* function defined from line 1 to 3. When the *deposit* method is executed (line 6), the LuaMOP mechanisms automatically invoke the *reglog* method.

LuaMOP functionality goes beyond the provision of a meta-representation. It can also capture events from the runtime execution environment. A *Monitor* is created to handle events related to elements that have not yet been declared in the application. This facility is used to support the anticipated join points strategy (see [Cacho et al. 05]).

The integration of these properties in an aspect environment is straightforward. For instance, the definition of the advice mechanism for a *call* join point is illustrated in Figure 6**.**

```
1 function AspectDefinition:defineCall(id)
2    local aspect = AspectDefinition.aspectList[id]
3    local metaobject = LuaMOP:getInstance(aspect.pointcut.list)
4    if (aspect.advice.type == 'before') then
5        metaobject:addPreMethod (aspect.advice.action)
6    elseif (aspect.advice.type =='around') then
7        metaobject:addWrapMethod (aspect.advice.action)
8    else
9        metaobject:addPosMethod (aspect.advice.action)
10   end
11   aspect.mob = metaobject
12 end
```

Figure 6: Aspect Definition using LuaMOP features in AspectLua

For each aspect declared in AspectLua, it only suffices to define the advice method as a *pre*, *wrap*, or *pos* method to the metaobject representing the join point (Figure 6, lines 4-10), depending on the type of aspect that can be *before*, *around* or *after* (Figure 1, line 6).

3 Reusability and Heterogeneity Improvement in AOP

In this section we discuss the features of aspect-oriented programming languages needed to benefit aspect reuse, context independence and heterogeneous interaction and then we introduce RE-AspectLua, a solution for these problems written in the Lua language. Section 3.1 addresses the lack of reusability and heterogeneity in AOP languages, presents a solution, the concepts of *aspect interfaces* and *abstract join points*, and discusses how a connection language may help in supporting reusability and heterogeneity in AOP languages. Section 3.2 presents how RE-AspectLua supports the features discussed in section 3.1 in order to promote aspect reuse and heterogeneous behavior.

3.1 Abstractions for Aspect Reuse

There are two essential features that aspect-oriented programming languages must include to benefit from aspect reuse, context independence and heterogeneous interaction:

Abstract Join Points. Abstract join points [Lieberherr et al. 99] are declarations of join points that are not bound to the base element at specification time. The definition of the real instance is done later, at application composition time. In this way, abstract join points specify generic and context independent aspects. Different applications can reuse the aspect and instantiate the abstract join points with different elements. Thus, this strategy promotes aspect reuse and allows the heterogeneous interaction of the aspect with different components. There is a lot of work [Aracic et al. 05], [Suvée et al. 03], [Hermann and Mezini 01] that uses this concept, including the more recent version of AspectJ.

Connection Language. Connectors are commonly used in component-based languages to connect components in order to compose the final system [Szyperski 02]. In a similar way, connection languages can be used to support the connection between aspects and components. Some works use connectors to yield component-aspect composition [McDirmid and Hsieh 03, Suvée et al. 03, Suvée et. al 05]. The use of such a mechanism, associated to abstract join points, is an interesting solution to reduce dependence relationships between aspects and components and to improve the flexibility of the interaction between them.

Aspect Interface. In order to address context independence, we borrow the concept of *aspect interface* defined in [Chavez et al. 05] to model aspects at architectural level. A connection language is used to define the instantiation of the join points and the definition of the advice activation time. Thus, the connection language, also named configuration language, makes it possible the independence of the aspect in relation to the affected components. In this way, in the specification, the aspect does not determine the code that it affects.

3.1.1 Aspect Interfaces

At the programming language level, we propose the concept of *aspect interface*. Aspect interfaces are contract definitions of aspectual functionalities established at specification time. The aspect interface defines *refinements*. Refinements contain (i) the definition of the abstract join point (the elements that will be affected by the aspect); (ii) the definition of action to be taken (the advice) when the join points are reached. This set of join points and the advice type (*before*, *after*, or *around*) are defined by the connection language at *application composition time*.

The aspect interface abstract syntax is BNF is illustrated in Figure 7.

```
1 <AspectInterfaceDeclaration> ::=  <AspectInterfaceName> "="
                                    <AspectInterfaceInstantiation>
2 <AspectInterfaceName> ::=  <identifier>
3 <AspectInterfaceInstantiation> ::= "AspectInterface:new()"
4 <RefinementDeclaration> ::=  <AspectInterfaceName> ":"
                                          <RefinementInstantiation>
7 <RefinementInstantiation> ::= "refinement({name = '"<identifier> "'},"
                               "          {refine = '" <identifier> "',"
                               "           action = " <identifier> "})"
```

Figure 7: BNF Syntax of Aspect Interfaces

The BNF syntax defines the declaration of aspect interfaces, and its refinements. An aspect interface has a identifier and is instantiated by the use of *AspectInterface:new()*. It may have one or more refinements. The refinements are declared for each aspect interface, and must define its name (*name*), the abstract pointcut name (*refine*) and the method to act as advice for the abstract join point (*action*).

3.1.2 Connection Language

In order to express the relationships between an aspect and base elements, the use of connectors, as we previously discussed in section, can give a proper support to promote the aspect independence in relation to the usage context.

Scripting languages have been used to support the interconnection of elements in component-based systems [Batista 00]. It acts as a configuration language that defines the relationship between the components. Scripts can also be used to adapt the component interconnection when interfaces are not compatible and, in this case, are called *glue code*.

In a similar way, scripting languages also act as connection language between aspects and components. In this case, the scripting language must instantiate the abstract join points. It cannot break the component interface contract and it must define the precedence between aspects that act on the same join point. In addition, the inherent flexibility of most scripting languages allows the definition of complex relationships between aspects and advised code such as (i) the conditional removal of an aspect behavior in the presence of other aspect, (ii) higher semantics by enabling complex aspect protocols, (iii) the definition of glue code addressing compositional mismatches.

The use of a scripting language as a connection language gives more flexibility that the use of connectors. In general connectors have a fixed structure that imposes a predefined way of expressing the composition. In contrast, with a scripting language the composition code can include several commands to coordinate the composition.

3.2 RE-AspectLua

RE-AspectLua includes the abstractions aforementioned. An aspect in RE-AspectLua is a component that contains a set of aspect interfaces (*AspectInterface*). An aspect interface can have one or more definitions of refinements. Refinements are declarations of abstract join points and advice (Figure 8).

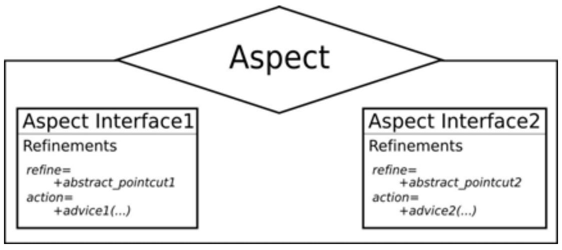

Figure 8: Aspect Model

In RE-AspectLua, common aspects are also called **aspectual components** by acting like a component with aspectual behaviour.

3.2.1 Aspect Definition

```
1  aspectA = Aspect:new( {name = "Aspect A"} )
2  aspectB = Aspect:new( {name = "Aspect B"} )
3
4  ai1 = AspectInterface:new()
5  ai1:refinement( {name = 'interface1'},
6                 {refine = 'abstractpointA',
7                  action = advice1} )
8
9  ai2 = AspectInterface:new()
10 ai2:refinement( {name = 'interface2'},
11                 {refine = 'abstractpointB',
12                  action = advice2} )
13
14 aspectA:interface(ai1)
15 aspectA:interface(ai2)
16
17 aspectB:interface(ai1)
```

Figure 9: Aspect Definition in RE-AspectLua

Figure 9 shows an example with two aspect interfaces declared: *ai1* and *ai2*. The *ai1* interface (line 4-7) defines a refinement that declares a behavior *advice1* method) to be executed when a set of abstract join points *abstractpointA* is reached. The moment when the method will be executed (*before*, *after* or *around*), and the exact join points that the *refine* declaration represents are defined at application composition time, using a connection language. The *ai2* interface (lines 9-12) is similar to the *ai1* interface. However, it defines another behavior to be executed when the set of abstract join points *abstractpointB* is reached.

In RE-AspectLua, the aspectual components can share definitions of aspect interfaces. In the example of Figure 9, the *aspectA* aspect has two interfaces, while *aspectB* has just the first interface declared. The two aspects have the same aspect interface but interact in a different way with the application. This illustrates the heterogeneity interaction ability of RE-AspectLua.

RE-AspectLua offers "syntax sugar" to reduce the code needed to define an aspect interface and to associate it to an aspect. Figure 10 presents the syntax sugar used to define aspect interfaces and aspects.

```
1 ai1 = AspectInterface:new_refinement( {name = 'interface1'},
2                                       {refine = 'abstractPointA',
3                                        action = advice1} )
4 ai2 = AspectInterface:new_refinement( {name = 'interface2'},
5                                      {refine = 'abstractPointB',
6                                       action = advice2} )
7
8 aspectA = aspect ({ai1, ai2})
```

Figure 10: Syntax Sugar for the Definition of Aspects in RE-AspectLua

Figure 10 (lines 1-3) illustrates the creation of the same *interface1* of Figure 9. The *aspectA* aspect is created containing the two interfaces *ai1* and *ai2* (line 8). In this case the aspect name is not explicitly defined.

3.2.2 Aspect Instantiation

```
1 int1 = aspectA:get_interface{'interface1'}
2 int1.abstractpointA =
3  {designator = 'call',
4   pointcut-list = {'some-class:method1()'},
5   type = 'before'}
6
7 int2 = aspectA:get_interface{'interface2'}
8 int2.abstractpointB =
9  {designator = 'callone',
10   pointcut-list = {'another-class.*'},
11   type = 'after'}
```

Figure 11: Instantiating the interfaces of aspectA

The interaction and precedence relationships of the aspects are defined also via scripts of the connection language. In RE-AspectLua, the language used to connect the elements is the Lua language itself with a library that allows the instantiation of join points and the definition of the relationships between aspects. This library offers the *get_interface* method, to dynamically get an aspect interface by its name, and the *bind_refinement* method to bind a refinement by instantiating concrete join points for it.

Figures 11 and 12 illustrate the definition of the aspectual connection to the aspects defined in Figure 9.

Figure 11 shows how the connection language is used to declare the instantiation of aspectA's aspect interfaces. First, the *ai1* aspect interface is retrieved by a *get_interface()* call on *aspectA* (line 1). Then the *abstractpointA* refinement has its instantiation defined: the *advice1* method (Figure 9, line 7) will be executed just before *some-class:method1()* call. In this language it is possible to quantify the join points using the wildcard '*' as can be seen in the declaration of the join points of the *abstractpointB* refinement (Figure 11, line 8).

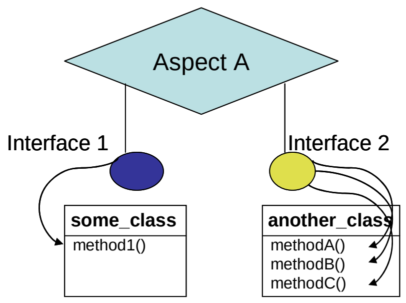

Figure 12: AspectA model

```
int3 = aspectB:
  get_interface{'interface1'}
int3.abstractpointA =
 {designator = 'execution',
  pointcut-list = 'third_class.*',
  type = 'before'}
```

Figure 13: Instantiating the interfaces of aspectB

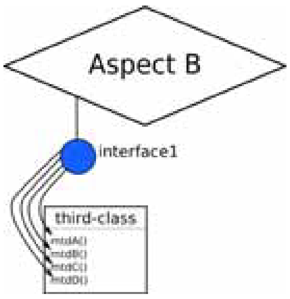

Figure 14: AspectB model

Another crosscutting relationship is illustrated in Figures 13 and 14. The *interface1* aspect interface in *aspectB* has a set of join points that intercepts the executions of all methods of the *third-class* component and executes the *advice1* advice (defined in the aspect interface *ai1* at Figure 9, line 7).

3.3 Application Development in RE-AspectLua

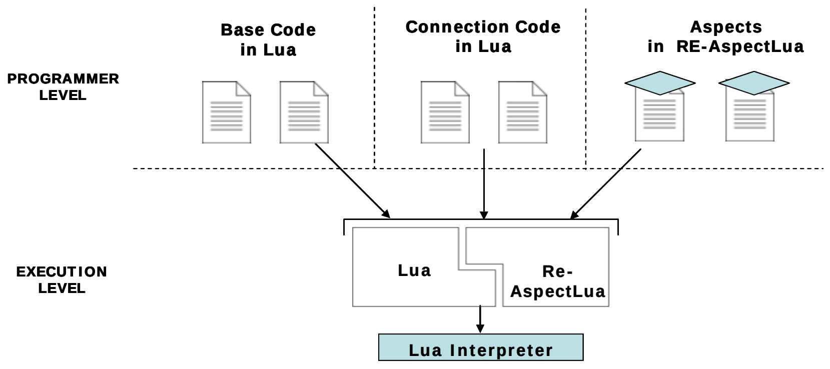

Figure 15: Application Development in RE-AspectLua

Figure 15 illustrates the application development process in RE-AspectLua. The programmer is responsible for the development of : (i) the base code that implement the main functionality of the application in Lua; (ii) the aspect code written in RE-AspectLua; (iii) the connection code in Lua that defines the combination of base code and aspects.

Lua does not offer support for aspect-oriented programming. In order to support it RE-AspectLua extends the language by including support for reusable aspects. Due to the Lua reflexive features it is not necessary to modify the Lua interpreter to support RE-AspectLua. AspectLua exploits the powerful of Lua tables and uses them to the definition of aspects, aspects interfaces, abstract join points and advices. These elements are defined using the Lua features and no special commands are needed.

The two codes are executed by the Lua interpreter that invokes RE-AspectLua when executing commands associated with aspect-oriented programming. This is done transparently for the programmer and supported by the reflexive facilities offered by Lua. In contrast to the traditional aspect-oriented programming, in this approach there is no need of a special compiler to join aspect with the functional modules, the interpreter does this mixing at runtime. Thus, it is possible to support dynamic reconfiguration – new base code and aspects can be selected dynamically according to runtime conditions.

4 Case Studies

4.1 A Banking Application

A simple banking application is shown to illustrate advices in RE-AspectLua. AspectLua does not offer a suitable solution for this case unless by some code duplication. This section shows how RE-AspectLua handles it.

Figure 16 contains a RE-AspectLua generic aspect code for logging bank transactions. The *logaspect* aspect is declared at line 3. The *logged_methods* refinement is associated to the *logging* aspect interface at lines 5-7; and the interface is linked to the *logaspect*. The advice code is at lines 10-12.

```
1 require 'REAspectLua'
2
3 logaspect = Aspect:new( {name = "logaspect"} )
4 aint = AspectInterface:new ()
5 aint:refinement ( {name = 'logging'},
6                  {refine = 'logged_methods',
7                   action = logbalance} )
8 logaspect:interface(aint)
9
10 function logbalance(self)
11    print ('Balance is now: ', self.balance)
12 end
```

Figure 16: RE-AspectLua Aspect Declaration

In order to instantiate the *logged_methods* refinement, a connection script is needed. Figure 18 shows multiple refinement instantiation. First, the aspect and the aspect interfaces are retrieved by their names (lines 1-2). Then, the refinement is instantiated (or bound) 3 times. The *logbalance* advice is executed after the *deposit* method of the *Bank* component (lines 4-6); and before and after the invocation of the *cash* method.

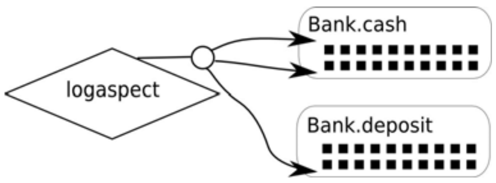

Figure 17: Instantiating the Interface of logaspect

```
1 laspect = Aspect:get("logaspect")
2 log_int = laspect:get_interface('logging')
3
4 log_int:bind_refinement("logged_methods", { designator = 'call',
5                                     pointcut_list = {'Bank.deposit'},
6                                         type = 'after'})
7 log_int:bind_refinement("logged_methods", { designator = 'call',
8                                         pointcut_list = {'Bank.cash'},
9                                         type = 'after'})
10 log_int:bind_refinement("logged_methods", { designator = 'call',
11                                        pointcut_list = {'Bank.cash'},
12                                        type = 'before'})
```

Figure 18: Instantiation Script

The pure AspectLua implementation of this functionality would need more than one aspect due to lack of reuse and heterogeneity abilities of AspectLua (discussion in section 2.1, Figure 3).

4.2 A Petroleum Fields Monitoring System

A *Petroleum Fields Monitoring System* has the main goal of registering the individual characteristics associated with the operational process of oil extraction in each automated well of a petroleum field. Based on the collected information, it provides descriptive and behavioral results in order to make possible an efficient monitoring process.

The system is composed of a *Central Monitor* that receives information from the *fields* and sends commands to them. CM performs an automatic data sampling that is a periodic task that accesses the monitoring data collected in every *well* of a field.

Figure 19 illustrates the class diagram of the system. The *CentralMonitor* class requires data from the *Field* class that, in turn, collect data from the *Well* class.

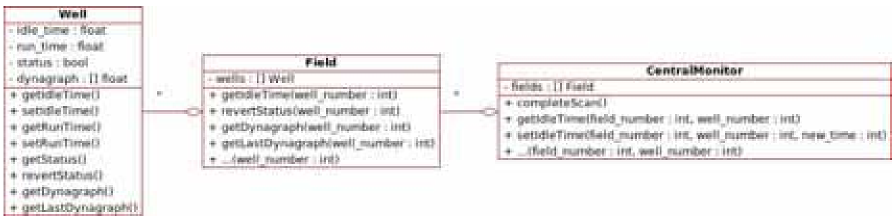
Well
- idle_time : float
- run_time : float
- status : bool
- dynagraph : [] float
+ getIdleTime()
+ setIdleTime()
+ getRunTime()
+ setRunTime()
+ getStatus()
+ revertStatus()
+ getDynagraph()
+ getLastDynagraph()
*
Field
- wells : [] Well
+ getIdleTime(well_number : int)
+ revertStatus(well_number : int)
+ getDynagraph(well_number : int)
+ getLastDynagraph(well_number : int)
+ ...(well_number : int)
*
CentralMonitor
- fields : [] Field
+ completeScan()
+ getIdleTime(field_number : int, well_number : int)
+ setIdleTime(field_number : int, well_number : int, new_time : int)
+ ...(field_number : int, well_number : int)

Figure 19: Monitoring System Class Diagram

Two crosscutting functionalities are essential to this system: (i) persistence; (ii) security. The system must store in a database the information collected from the field and the well. It must also support user authentication and different classes of users may have different views of the system. These two functionalities are implemented by the following generic aspect in RE-AspectLua: *PersistenceAspect* and *SecurityAspect*. Figure 20 shows a graphical view, using AsideML notation [Chavez et al. 05], the interaction of these generic aspects when applied in the system.

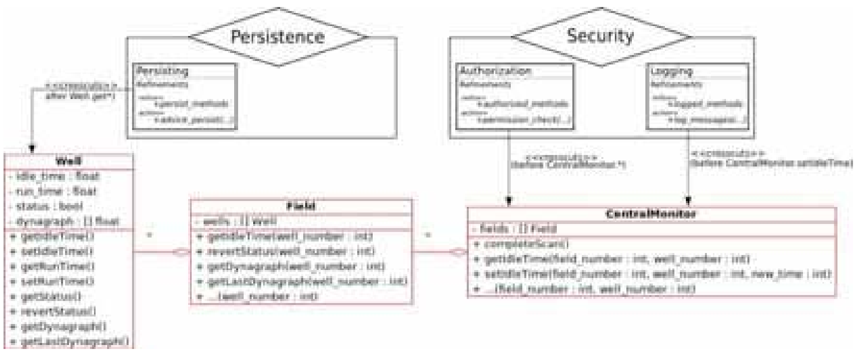

Figure 20: Persistence and Security Aspects

PersistenceAspect contains an aspect interface (*persisting*) (Figure 21) that defines a refinement that implements persistence via the *advicepersist* advice (lines 3-8). Note that it is a generic aspect that can be applied in this case study or in any other that requires persistence.

```
function advice_persist(self, ...)
   parameters = DBUtils:getInstance().getSQLInfo(self, ...)
   -- generic code for SQL insertion data into tables
end

p_asp = Aspect:new({name = 'PersistenceAspect'})
p_int = AspectInterface:new({name = 'persisting'})
p_int:refinement ({refine = 'persist_methods',
                  action = advice_persist})
p_asp:interface(p_int)
```

Figure 21: Persistence Aspect

Figure 22 contains the composition code indicating that after (*type* = *'after'*) invoking the *get* method from the Well component (*pointcut_list* = *'Well.get*'*), the results are intercepted and stored in a database according to the *persisting_methods* refinement defined by the aspect code.

```
p_int:bind_refinement('persist_methods',{designator = 'call',
                                        pointcut_list = 'Well.get*',
                                        type = 'after'})
```

Figure 22: Binding Code of the Persistence Aspect

The security aspect contains two aspect interfaces, *Security* and *Logging* (Figure 23). The *Security* interface contains a refinement to user authorization. The *Logging* interface contains a refinement to register the invocation of some methods that need to be monitored.

```
function permission_check(self, ...)
   su = SecurityUtils:getInstance()
   if su:authorized(enviroment:user,self) then
       proceed(self, unpack(arg))
   else
       su:security_exception("Access denied. Unauthorized role")
   end
end

security_asp = Aspect:new({name = 'SecurityAspect'})
security_int = AspectInterface:new()
security_ref = security_int:refinement ({name =
                                          'security_interceptor'},
                                        {refine =
                                          'authorized_methods',
                                        action = permission_check})
security_asp:interface(security_int)
```

Figure 23: The Security Aspect

The *permissioncheck* advice consults the *SecurityUtils* component to verify if the user has the permission to execute the method.

The *logging* refinement contains the *logmessages* advice to log the invocations to some methods. It must log the invocation whenever a trainne change the *idletime* but it is not necessary to log the invocation if engineering do the same action. This behavior is defined in the connection language as this is a typical information of the context.

The connection between the security aspect and the base code is defined by the script illustrated in Figure 24. The abstract pointcut *authorizedmethods* is bound to all invocations of all methods of the *CentralMonitor* class. The declaration *before* specifies that all methods are intercepted before their execution and the security aspect checks if the user has the permission to invoke the method.

The connection code contains a conditional statement (lines 7-12) to verify if the logging aspect must be executed. The user identity is verified and if it is not a engineering that is trying to change the *idletime*, the invocation is logged.

```
1 s_int = security_asp:get_interface('security_interface')
2 s_int:bind_refinement('authorized_methods',{designator = 'call',
3                             pointcut_list =  CentralMonitor.*',
4                             type = 'before'})
5
6 l_int = security_asp:get_interface('logging_interface')
7 if enviroment.user.getRole() != 'engineering' then
8     l_int:bind_refinement('logged_methods',
9                         {designator = 'call',
10                         pointcut_list = 'CentralMonitor.setIdleTime',
11                          type = 'after'})
12 end
```

Figure 24: Connection code of the Security Aspect

This flexible scenario is possible due to the decoupling between the aspect specification and the connection with the base code and also because the use of a scripting language to specify the composition. Without the connection language, the aspect would include the conditional test that is specific to the context of this system. As a consequence reusability in other context would be impossible.

5 Related Work

[Sullivan et al 05] addresses the problem of decoupling aspects from advised code by defining an interface, based on design rules, that base code implements and that aspects depend upon. While this strategy modularizes both aspect and base code, aspects reusability is not a central issue. Our approach aims to modularize the aspect code and make aspects reusable. In order to support reusability we use abstracts join points and a connection language.

AspectLagoona [Gal et al. 03] defines aspects as component-like modules containing advices. The pointcut language is the method definition in component interface. The aspect is defined for component interface, not for component implementation, crosscutting all implementations. This way, there is low independence of aspect from the component, leading to limited reuse of aspect. RE-AspectLua allows context independence, providing more reusable aspects than AspectLagoona.

Jiazzi [McDirmid and Hsieh 03] supports component developments for Java. Components are linked by a connection language. The connection language does not allow advice-like behaviour, although it permits that some crosscutting concerns are better modularized by the use of open classes and open signatures. New code statements and a post-compilation phase are needed in Jiazzi. RE-AspectLua also defines aspect interaction by using a connection language, but it does not require any further step to bind aspect behaviour into component code.

JasCo [Suvée et al. 03] provides aspects in JavaBeans component model. Aspectbeans are defined containing advice associated to abstract pointcuts. Special connector entities instantiate the abstract pointcuts, supporting a high level of reuse. RE-AspectLua is inspired on JAsCo, by providing abstract pointcuts to be instantiated later, at the instantiation phase. RE-AspectLua uses Lua as connection language, in consequence dynamic advice implementation is simpler than in JAsCo, where all JavaBeans must be traced in the execution environment.

CaesarJ [Aracic et al. 05] integrates components and aspects with mixin composition of family classes. This way, CaesarJ does not use a connection language to link aspect to components. A wrapper mechanism is used to adhere crosscutting structure and behavior to components of different family classes. RE-AspectLua does not use a wrapper mechanism, but refinements in aspect interface, to link crosscutting behavior to base components.

LAC -- Lua Aspectual Components [Hermann and Mezini 01] -- is a Lua extension whose main goal is to support the idea of Aspectual Components (AC) [Lieberherr et al.99]. It defines an explicit connector module to bind the participants. In contrast, RE-AspectLua is more flexible as it uses the Lua scripting capabilities in the definition of the connection between aspects and base elements. This flexibility allows the definition of complex aspect composition.

6 Final Remarks

The abstractions to the modularization of the crosscutting concerns at traditional aspect-oriented languages are limited in terms of reusability and heterogeneous interaction capability. In this paper we use the concepts of *aspect interface* and *connection language*, commonly used in component-based systems, in order to promote aspects reuse. Thus, we adopt the idea that AOP must follow fundamental concepts of information hiding modularity. We also propose the use of a scripting language, the Lua language, to the definition of the interaction between aspects and base elements. Thus, we suggest a new application domain of scripting languages: as a glue language between aspects and classes.

We highlight the importance of splitting the aspect-oriented development in two phases: (i) *aspect specification time* where reusable aspects are defined in an independent way of its target application; (ii) *application composition time* where the composition of aspects and base code are defined. These two phases are essential to promote the independence of aspects from the base code and to allow the heterogeneity interaction between an aspect and different base codes.

Although the concept of *abstract join points* is commonly found in some works, we use this concept in conjunction with a dynamically typed and interpreted connection language, that adds a great deal of flexibility to the composition of the reusable aspect and the base code. AspectJ supports the concept of abstract pointcuts however, the flexibility provided by using abstract pointcuts with a connection language is not addressed in AspectJ.

The concepts of abstract join points, aspect interface and connection language are instantiated in the definition of a new version of AspectLua, RE-AspectLua, that promotes reusability, context independence and a better organization of the heterogeneous interaction between aspects and base elements.

The two case studies presented in this paper show that RE-AspectLua enriches the modularization approach of AOP to allow a more effective reuse and heterogeneous interaction. In addition, we could illustrate the importance of using a scripting language, at application composition time, to select different compositions between aspects and base code according to runtime conditions.

Acknowledgements

This work has been partially supported by the Brazilian National Agency of under grant 2001.7999-5 for Mauricio B. C. Vieira.

References

[Aracic et al. 05] Aracic, I., Gaiunas, V., Mezini, M., and Ostermann, K.: “An Overview of CaesarJ”. Technical Report TUD-ST-2005-01, Darmstadt University of Technology, Darmstadt, Germany. (2005)

[Batista 00] Batista, T. V.: “LuaSpace: Um Ambiente para Reconfiguração Dinâmica de Aplicações baseadas em Componentes”, PhD Thesis, Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio), Rio de Janeiro. (2000)

[Bryant et al. 02] Bryant, A. and Feldt, R.: “AspectR – Simple Aspect-Oriented Programming in Ruby”. http://aspectr.sf.net/. Last visualization in May 2007. (2002)

[Cacho et al. 05] Cacho, N., Batista, T. and Fernandes, F.: “AspectLua – A Dynamic AOP approach”, In: Journal of Universal Computer Society (J.UCS), 11(7):1177-1197. (2005)

[Chavez 05] Chavez, C., Garcia, A., Kulesza, U., Sant’Anna, C. and Lucena, C.: “Taming Heterogeneous Aspects with Crosscutting Interfaces”. In SBES2005: Proceedings of the XIX Brazilian Symposium on Software Engineering, , Uberlândia, Brasil, (2005), 216-231.

[Dechow 04] Dechow, D.R.: “Advanced Separation of Concerns for Dynamic, LightWeight Languages”. In 5th Generative Programming and Component Engineering. (2004)

[Fernandes et al. 04] Fernandes, F. and Batista, T.: “A Dynamic Approach to Combine Components and Aspects”, In: Proceedings of the XVIII Brazilian Symposium on Software Engineering (SBES), Brasilia, Brazil, (2004), 102-112.

[Gal et al. 03] Gal, A., Franz, M., Beuche, D.: “Learning from components: Fitting AOP for Systems Software”, In Proceedings of the AOSD 2003 Workshop on Aspects, Components, and Patterns for Infrastructure Software, Boston, MA, USA. (2003)

[Herrmann et al. 01] Herrmann, S. and Mezini, M. Combining Composition Styles in the Evolvable Language LAC. In Workshop on Advanced Separation of Concerns in Software Engineering. (2001)

[Hirschfeld et al. 03] Hirschfeld, R.: AspectS – Aspect-Oriented Programming with Squeak. In NODe’02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services and Applications for a Networked Worls, (2003), 216-232, London, UK. Springer-Verlag.

[Ierusalimsky et al. 06] Ierusalimsky, R.: Programming in Lua. Lua.org, Rio de Janeiro, Brazil. (2006)

[Kiczales et al. 97] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J., Irwin, J.: “Aspect-oriented programming”, In: ECOOP’97 — European Conference on Object-Oriented Programming”, Proceedings of ECOOP´97. Springer-Verlag, Finland. (1997)

[Kiczales et al. 01] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: “An Overview of AspectJ”, In: ECOOP’2001 — European Conference on Object-Oriented Programming, Budapest, Hungary. (2001)

[Lierberherr et al. 99] Lierberherr, K., Lorenz, D., Mezini M.: “Programming with Aspectual Components”, In: Technical Report NU-CCS99 –01, Notheastern University. (1999)

[McDirmid et al. 03] McDirmid, S. and Hsieh, W. C.: “Aspect-Oriented Programming with Jiazzi”, In AOSD’03: Proceedings of the 2nd International Conference on Aspect-Oriented Software Development, pp 70-79, New York, NY, USA, ACM Press. (2003)

[Suvée et al. 03] Suvée, D., Vanderperren, W., and Jonckers, V.: “Towards a Symbiosis Between Aspect-Oriented and Component-Based Software Development”, In Proceedings of the SCI 2003 International Conference, , Orlando, USA, (2003), 442-447.

[Suvée et al. 05] Suvée, D. Vanderperren, W., Wagelaar, D., and Joncker, V. There are no Aspects. In Electronic Notes in Theorical Computer Science, vol. 114, Elsevier Science, (2005), 153-174.

[Sztipanovits et al. 02] Sztipanovits, J. and Karsai, G. Generative Programming for Embedded Systems. In PPDP 02: Proceedings of the 4th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming, , New York. ACM Press, (2002), 180-190.

[Sullivan et al. 05] Sullivan, G., Griswold, Y. et al: “Information Hiding Interfaces for Aspect-Oriented Design. Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering, (2005), 166 – 175.

[Szyperski 02] Szyperski, C. Component Software: Beyond Object-Oriented Programming. Addison-Wesley Longman Publishing Co., Inc., Boston, USA, (2002).