Aspect Coordination for Web Applications in Java/AspectJ and Ruby/Aquarium

Arturo Zambrano*, Alejandro Alvarez*, Johan Fabry† and Silvia Gordillo* ‡
* *LIFIA, Facultad de Informática, Universidad Nacional de La Plata*
50 y 115, La Plata, Argentina
arturo,aalvarez,gordillo@lifia.info.unlp.edu.ar
† *PLEIAD Lab, Computer Science Department (DCC), Universidad de Chile*
Blanco Encalada 2120, Santiago, Chile
jfabry@dcc.uchile.cl
‡*CIC, Provincia de Buenos Aires*

***Abstract*—Aspects are independent modules that capture otherwise crosscutting behavior. There are mechanisms for enabling their coordination without compromising their independence, but the aspect oriented implementation platform highly impacts on the final result. In this work we analyze Java/AspectJ and Ruby/Aquarium for the implementation of such coordination mechanism. Performance, conceptual and coding trade-offs can be derived from the differences between these platforms. The balance between absolute and relative performance costs and the benefits of provided by each platform are not obvious. The results of this analysis is determining to choose the right underlying aspect platform for different scenarios.**

***Keywords*-aspect oriented programming, static languages, dynamic languages**

I. Introduction

In the application development process, it is common to find a set of concerns that affect many objects beyond the modules which constitute the natural units to define concerns functionality. They are called crosscutting concerns. A crosscutting concern is one that is spread over (cuts across) many of the modules of a system. These concerns often cannot be cleanly decomposed from the rest of the system, resulting in code difficult to develop and maintain [19]. Typical examples of crosscutting concerns are: persistence, synchronization, error handling. etc.

Aspect-Oriented Software Development (AOSD for short) [11] is one of many paradigms resulting from the effort to modularize crosscutting concerns. The goal of AOSD is to decouple these concerns, so that the system's modules can be easily maintained, evolved and seamlessly integrated.

AOSD introduces a new kind of module called *aspect*. An aspect augments the behavior of one or many modules (known as *base* application) by executing some behavior when determined execution points in the base application are reached. These execution points are known as *join points* in the AOSD jargon. Examples of joinpoints are: method calls, variable assignation, constructor call, etc.

An aspect is composed by *pointcuts* and *advices*. A pointcut is an expression denoting a set of joinpoints. Advices define the behavior of the aspect, and they are associated to a pointcut. When the base application is executed if a joinpoint *matches* a pointcut definition, the associated advice is executed. The method is said to be *advised*. Along the paper we will use the verb *to advise*, to indicate that behavior defined in an advice is added to method or an advice.

As the objective of AOSD is to modularize crosscutting concerns, aspects are defined separately from the rest of the application. The process that composes the behavior of the base application and the aspects is called *weaving*.

The goal behind using aspect orientation is to have independent aspects whose composed behavior provides the desired functionality when they are woven into the base application.

Despite of the desired decoupling between aspects, it is recognized that aspects interact in a number of ways [16]. Therefore, it is necessary to coordinate their behavior, while keeping them independent, and without compromising reusability. We have previously shown this can be achieved for context-aware systems using semantic aspectual information [22].

In this paper we extend this work, showing a scenario where aspect coordination is needed in the context of collaborative web applications. Then we analyse the trade-offs derived from applying our coordination approach using two different platforms for the implementation of such applications: Java [7]/AspectJ [10] and Ruby [13]/Aquarium [1]. Implementing aspect oriented systems in these platforms is substantially different. These differences make it necessary to perform an in-depth analysis of each feature needed for aspect coordination, evaluating the abilities of each platform.

In this work we report the results of such an analysis. For example, the ability of performing runtime unweaving in Ruby provides conceptual benefits over simulating it when using AspectJ [10]. On the other hand, runtime weaving is an expensive operation which takes 7000 times the time needed for a method call.

Some differences can be deduced from the fact that Java is strongly typed and compiled, and Ruby is dynamically typed and interpreted. At same time, many relevant consequences

for the development of aspect oriented systems are reported, particularly when aspect behavior needs to be coordinated at runtime using context information.

This paper is structured as follows: Sect. II presents the domain and motivates the need for aspect coordination. Sect. III shows our approach for aspect coordination and the peculiarities of its implementations in Java and Ruby. Sect. IV presents a comparison of both implementations from many perspectives. Finally, in Sect. V we conclude the work and present future work.

II. Concerns in Collaborative Web Applications

Collaborative web applications deal with a myriad of concerns. Building this kind of software involves concerns at different levels: from low level ones such as persistence, distribution, synchronization to more functional ones, such as sharing, conflict solving or activity awareness.

Consider for example the features present in the set of collaborative web applications known as Google Docs [6]. Google Docs enable collaboration for editing spreadsheets and text documents. These applications share a common set of features related to collaboration, for example: the discussion pane, versioning, user activity awareness, notification of changes, change logging, chat, etc.

Depending on the decomposition chosen, some of these concerns will be crosscutting. Therefore, many of these crosscutting concerns can be designed and implemented as aspects. We argue that such aspects must be coordinated in order to get the application working properly.

A. Running Example

For the rest of the paper we will consider a slightly modified version of the Google Docs text editor, which we describe below:

> A collaborative text editor allows multiple users to cooperate in writing documents. A document is created by a user, who can then decide to share it. A document has a quota of disk space in the server. When two (or more) users are working on the same document at the same time, each one must be aware of the activity of the other. Users can discuss about the document content in a right side chat widget. Discussions are saved associated to the corresponding version of the document where they took place. Users can set up notification rules, that is, upon certain events –in the document– email notifications are delivered. There is also a log of activities which register events associated to the document.

Collaborative web applications are complex. In this case, assuming an aspect oriented approach to the problem, we model the "pure" text editor as the base application, leaving aside, for a moment, the collaborative functionality. In this application, entities such as: the document and the users could be considered part of the base application. Typical operations for the document would be *open*, *save*, *close*, *add collaborator*, *remove collaborator*, *create a new version*, etc. Besides the objects in the main dimension there are number of crosscutting concerns related to collaboration (explained below). These concerns can be effectively modeled and implemented as aspects. For example, awareness has been modeled as an aspect by Torres et al. [18]. The personalization concern is also crosscutting, since it affects many elements in the application. In [21] we have shown how personalization can be introduced seemleasly by applying AOSD. Other concerns such as activity log and notification of change are crosscutting, since they need to be invoked in every single event that must be notified or logged.

A description of some of the collaboration concerns of this application follows:

Awareness: If the document is shared, each user must be aware of the activity of other online users. When a change is introduced by a user and the document is saved, all other users must receive the new version of the document. Other types of awareness include remote cursors, remote field of vision, remote selection or user presence [17].

Discuss: Discuss feature is enabled just when a document is shared, and when at least two collaborators are currently logged in. Discussion logs are saved along with the corresponding version of the document.

Notification Rules: There are a number of events in the life cycle of the document that a user can request to be notified of. Upon the occurrence of any of these events, email notifications are delivered according to configuration settings. Some of the events that can be notified are: a document was opened, there is a new collaborator, a collaborator left the document, etc.

Activity Log: Activity is registered in a log, which is associated to the different versions of the document. Such information includes events such as: collaborators joining or leaving the document, changes, open and close operations, invitations for collaboration delivered, etc.

Personalization: Personalization [3] changes somehow the content or links, in a user specific manner. Usually, personalization is based on the user profile. Building a user profile involves taking information from many sources about user activities and preferences. Then, adaptations are rendered and introduced into the content of the application.

B. Interactions between Collaborative Web Application Concerns

When collaboration concerns are designed and implemented as independent aspects it is necessary to coordinate their behavior so that the final desired functionality is achieved. This need is reinforced if adaptability to the context is desired. Context must be taken into account to select the aspects which should be active.

Next, we list some example scenarios where such coordination is needed:

Scenario 1: Notification Rules & Awareness: Lets suppose that *notification of change* rules and *awareness* are active. In this scenario two users have been collaborating (one of them using a slow connection), so both of them are aware of changes made by each other. In this case, it would be good to avoid sending the email notifications (*notifications of change* concern) to the user with the slow connection, because he is aware of the changes, information would be redundant and it would consume a scarce resource.

Scenario 2: Discuss & Personalization: As we explained before, personalization involves the customized presentation of content and links. In this scenario, lets suppose two users are colaborating, each one with personalized content based on his own user profile. When they start a discussion, it is desirable to switch off all the personalization functionality. This is needed in order to have all of them looking at the same content, without this precondition discussion could be difficult as they might be observing different content.

Scenario 3: Activity Log & Discuss: As we said, each document – and its associated data such as logs, discussions, etc. – has a quota of disk space. If the quota is being reached, some concerns may compete for the remaining space. For example, if *activity log* is working, and a discussion takes place, at the time of saving a new version of the document it is possible that activity information need to be discarded in favor of the discussion. This is because under this circumstance we privilege content generated by users (such as the discussion) instead of logging data. In this case the best choice would to be switch off the *activity log* to avoid gathering unnecessary data.

C. The Role of the Execution Context

Implementing this kind of behavior using aspect orientation is a challenge, since we would like to have each aspect independent and not coupled to others, but at the same time we want to keep them working cooperatively. In these cases, the key piece of information to get them working correctly is given by the execution context: who is collaborating, how much space left there is for data, what the users are doing, which aspects are running, etc. Using this information, it is possible to activate or deactivate the aspects in order to adapt application behavior to the context.

III. Aspect Coordination

In this section we describe the general approach for coordinating aspects and the particularities derived from its implementation in different platforms.

A. General Approach

In this section we briefly introduce the general approach for coordinating aspects explained in our previous work [22].

Aspects are enriched with metadata containing semantic information about them, as shown in Figure 1. We call this metadata *semantic labels*. Depending of the platform of choice, *semantic labels* can be expressed using annotations [8] (in the case of Java) or attributes (for .Net [12]), as separate file (for example using XML [5]), or as part of the documentation in the source code (in similar way as XDoclets [20]). This semantic information describes which functionality an aspect provides, how it affects the base application, what effects it causes on available resources, etc. Examples of such semantic labels in our domain are:

- A semantic label that describes a resource and the effect that the tagged aspect has on it. In our example, the *Activity Log* could be tagged as “Hard Disk Consumer”.
- A semantic label indicanting the role or functionality provided by an aspect: “Gathers User Content” could be attached to the *Discuss* aspect. The tag “Adapt User Content” can be applied to the *Personalization* aspect.

Having such semantic information allows us to write coordination logic for aspects based on their semantics and not coupled with syntactic details.

Coordination logic is expressed in rules, that indicate which aspect(s) must be (de)activated according to the current context. But, instead of referring them by their names, rules can use semantic labels. The condition of these rules evaluate the context, and the consequence performs actions on the aspects (for example deactivating an aspect which is not longer necessary). The combined use of labels and rules for controlling aspects, avoid the coupling between aspects themselves and between rules and aspects.

The key piece for deciding how the aspects must be coordinated (which ones need to activated or deactivated) is the context. Context for this application includes the state of resources (for example remaining space for a document and its data), who is collaborating, which actions users are performing (e.g. discussing), the connection type used by each user, which aspects are active, etc.

All this context information is continuously maintained at runtime and can be used to express conditions for our rules. After the context is changed, rules are evaluated so that just the aspects suitable for such context are active. Activation or deactivation of aspects can be implemented in different ways according to the platform (see Section IV-B).

Let see how our approach solves, for example, scenario 3. In this scenario we are running out of space for a document and the *Discuss* and *Activity Log* aspects are running. The metadata for *Discuss* states that it *consumes disk space* and that it *gathers user content*. On the other hand, semantic labels for *Activity Log* indicate that it *consumes disk space* and it *logs activities*.

Besides this semantic information, we have coordination rules that, based on the context, coordinate the aspects. For this example the rule is as follows:

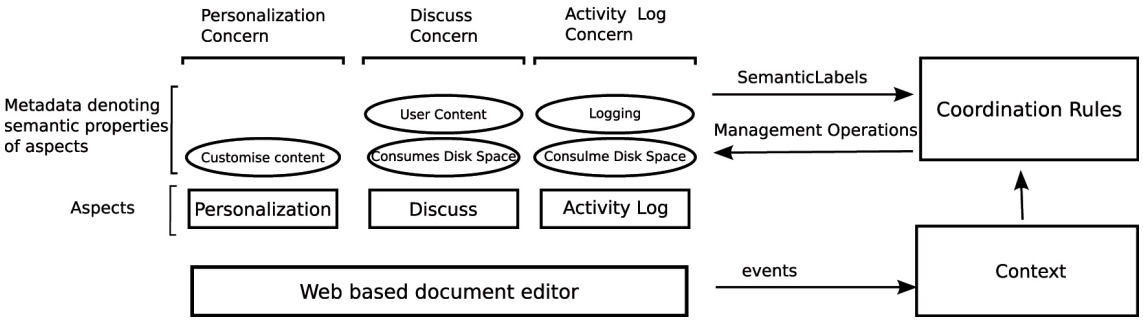


Figure 1: Aspect coordination based on aspect metadata and context evaluation.

If "running out of space" turn off aspects that "log activities" and "consume disk space"

This high level expression is translated into a proper rule for a rule engine. Upon a change in the context of a document, these rules are evaluated so aspects are coordinated according to current context. As a result, *Activity Log* will be turned off when we are running out of disk space.

B. Implementing Coordination in Java and AspectJ

In this section we will explain how our approach is implemented when the implementation platform is Java and AspectJ [10]. AspectJ is an extension of the Java programming language that supports the concepts introduced by aspect orientation.

We have implemented base entities (Document, User, etc) as objects. Collaboration crosscutting concerns have been implemented as aspects which have pointcuts bound to relevant methods in our base objects, for example: document open, close, save. etc.

Semantic labels are expressed using annotations [8]. They can be directly attached to aspects as they are coded. For example, our Activity Log aspect could be declared as shown in Listing 1, where it is annotated as disk space consumer and provider of logging functionality.

```
1  @ConsumeDisk
2  @Logging
3  public aspect ActivityLog ...
```

Code Listing 1: Aspect declaration including semantic labels.

Context is modelled as an object and there is a context instance per document. Context updates are performed by an aspect that captures all events that are relevant to our domain. To name just a few: document open, close, save, user start collaborating, user joins a document, user leaves a document.

After the context has been refreshed, coordination rules are evaluated. In order to evaluate rules we have selected Drools [4] as our rule engine. The context of the affected document is passed as a parameter each time it is needed to evaluate the rules. We also decided to add aspect instances as part of the context of a document, making it easier to manipulate them from the rules. Listing 2 shows an example of a coordination rule written using Drools syntax. This rule turns off aspects that consume disk and that provide logging functionality when 85% of the disk quota has been occupied by a document and its associated data.

```
1  rule "DiskQuotaLimit"
2   when
3     context : DocumentContext(occupiedDisk
4                     > QUOTA_LIMIT * 0.85)
5   then
6      context.turnOffAspects("ConsumeDisk",
7                                        "Logging")
8  end
```

Code Listing 2: Coordination rule using Drools syntax.

To provide aspect (de)activation, every execution of a collaboration aspect is controlled by another aspect which decides if the collaboration aspect advice is actually executed or not. Therefore, it is possible to skip such execution if context situation demands it. We contrast this in detail against the Ruby/Aquarium implementation in Section IV-B.

C. Implementing Coordination in Ruby and Aquarium

In this section we explain how our approach is implemented using Ruby [13], and the Aquarium framework [1] for the aspect oriented support. As the Ruby/Aquarium combination is not as widespread as Java/AspectJ, we show our implementation in more detail.

Ruby does not provide a built-in support for attaching information to code, therefore we developed a workaround. The implementation of semantic labels for Ruby is done by defining a set of global constants which are attached to a given class by calling a method. Listing 3 shows and aspect declaration, the `attach` method is called in line 2, with a parameter called `semanticLabels`.

```
1  class ActivityLog < WebAspect
2    attach :semanticLabels
3      => [ConsumeDisk, Logging]
```

Code Listing 3: Declaration of semantic labels in *Ruby*

This definition is captured at runtime by the *method missing* exception. By implementing the handler for the

method missing exception it is possible to capture and interpret any call that the interpreter could not match to a method signature. In our implementation for *method missing* (in the root of our aspect hierarchy), we first validate it is a semantic information definition. Second, the definition is processed and information is attached to the aspects.

We have modelled context information as an object containing the information regarding the document (such as disk quota available), deployed aspects, server and user context information. There is a context monitor aspect which captures relevant events and updates the context as necessary. After a context update it calls for rule evaluation. Listing 4 shows a simplified version of the ContextMonitor aspect. In this code, after calls to methods `open` and `save` of the class `Document` (lines 2-4), we get the context (line 5) and fire rule evaluation (line 6).

```
1 Aspect.new
2       :after,
3       :calls_to => ['open', 'save'],
4       :in_types => 'Document' do
5   |join_point, aDocument, *args|
6   context = createOrUpdateContextFor(aDocument)
7   RulesEngine.evaluateContext(context)
8 end
```

Code Listing 4: Aspect definition example using *Aquarium*

For the rules processing we chose the rule engine Ruleby [14]. Listing 5 shows a rule example where the disk quota available (for a given document context) is evaluated and according to the result, some aspects are deactivated. The rule is composed by an optional *symbol* that is a unique identifier for the rule (line 1, `DiskQuotaLimit`). In line 2 the parameter context is bound to a DocumentContext instance. The rule condition is presented in line 3. The consequence of the rule is represented by the block defined in lines 4 to 7.

```
1 rule :DiskQuotaLimit,
2   [DocumentContext, :context,
3    m.occupiedDisk > QUOTA_LIMIT * 0.85]
4 do |vars|
5  vars[:context].turnOffAspects([ConsumeDisk,
6                                 Logging])
7 end
```

Code Listing 5: Rule definition example using *Ruleby*

In this example, the consequence is to deactivate all aspects that have the semantic label "ConsumeDisk" and "Logging".

Activation and deactivation of aspects is done by weaving and unweaving the aspects. These operations are expensive but after unweaving an aspect, its associated overhead is removed from the base code. Turning on aspects is done by installing aspects on the required objects (in this case Document instances) or classes. System wide aspects, such as the aspect in charge of context updates, are installed at the class level, in this way they affect the behavior of all the existing instances. On the other hand, aspects that need to work on particular instances of Document are installed at the instance level. For example if *Activity Log* needs to be active for one document and disabled for other documents, the aspect is woven just into the instance of Document that needs logging.

IV. Comparison between Java and Ruby Implementations

In this section we recapitulate and compare pros and cons of the implementation on each platform.

A. Common Characteristics

There are a number of characteristics present in both implementation:

- Coordination rules can be modified at runtime: both implementations enable the modification of coordination rules without stopping the system for compilation. This is necessary in order to have real runtime adaptation in our systems.
- Rules can manipulate aspects using semantic labels. In both implementation we managed to avoid coupling rules to aspect names. Instead rules use semantic labels.
- Aspect instances are kept as part of the context. We found that aspects need to be included as part of the context, in order to provide the full picture of the system state to the rules.

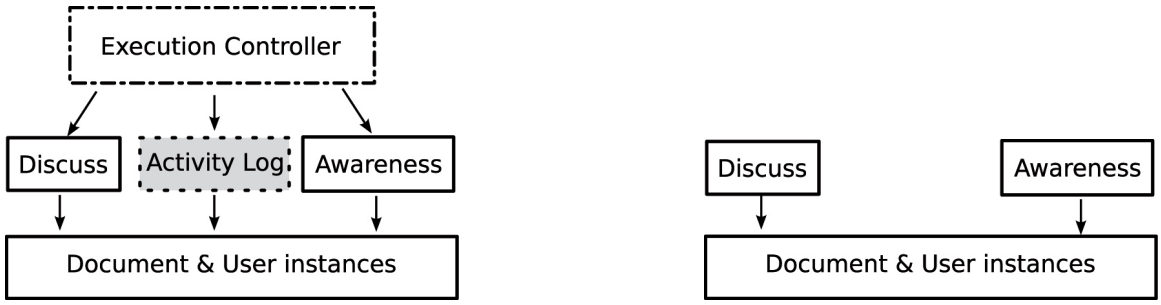
B. Dynamic vs Static approaches

Many of the implementation differences and difficulties are derived from the dynamic nature of Ruby/Aquarium compared to the static nature of Java/AspectJ. Here we explain how this impacts our implementations.

As we need to implement activation and deactivation of aspects at runtime and AspectJ does not provide runtime weaving and unweaving, we applied a workaround. It is based on deploying another aspect, called ExecutionController. It has an around advice that affects all the collaboration related aspects, allowing the execution of some aspects and preventing the execution of others. This workaround adds an extra overhead in order to keep track of which aspect instances must be active and which not. The ExecutionController aspect also imposes another overhead for checking the execution of each advice of our collaboration aspects.

For AOSD platforms where (un)weaving is possible at runtime such as Ruby/Aquarium, the solution is simpler; when an aspect needs to be deactivated it is unwoven from the application or the particular object. Weaving and unweaving are expensive operations, we analyze performance in Sect. IV-G.

Figure 2 shows how the *Activity Log* is deactivated in both implementations. For the AspectJ implementation (Fig. 2a) we can see that *Activity Log* is still working on the base application, but executions are skipped due to the



(a) AspectJ (b) Aquarium

Figure 2: ActivityLog aspect deactivated in AspectJ (left) and Aquarium (right). Arrows indicates advising.

ExecutionController aspect. We can see that for the Ruby based implementation (Fig. 2b) the aspect instance has been removed, therefore does not affect any element in the base application.

To sum up, AspectJ requires extra work to simulate aspect deactivation. On the other hand Aquarium provides full fledge runtime weaving and unweaving, making it ideal for scenarios where aspects need to be removed.

C. Aspect Instantiation

Aspect instantion mechanisms have an impact when the programmer wants to take control of some aspect instances, as it is our case with the coordination rules.

As stated in the AspectJ documentation [2]:

> ... aspect instances are automatically created to cut across programs.

This sentence is valid for the wide range of keywords that control aspect instantion (perthis, pertarget, percflow and percflowbelow). In all these instantion models, aspects are instantiated automatically. This makes it difficult the task of getting references to the aspects instances, which is necessary for our approach to work.

In our case we need to keep an association, in the context, of each aspect instance applied to a document. Getting such references is not trivial as AspectJ is seemingly not intended for this usage. AspectJ requires more complex code (when compared with Aquarium) to keep track of aspect instances.

On the other hand, in Ruby it is necessary to have the interpreter execute an extra instruction for performing the weaving of the aspect. This makes it easier to get the reference to the aspect instance, but it also asks for a careful startup process, to ensure all necessary aspects are woven by the time the application effectively runs.

Ruby is more appropriate when we need to keep explicit references to deployed aspect instances. If manipulation of aspects becomes a must, more work is needed to make AspectJ friendly in this respect. Research has been undertaken in this direction by Sakurai et al. [15].

D. Aspect Modelling

Aspects in Aquarium are instances of the `Aspect` class provided by the framework. That is, it is not necessary to declare a new class to have an aspect, as it is the case in AspectJ. In our approach it is however necessary to provide a class for attaching the semantic labels. Therefore, in the Aquarium implementation we had to define a class per aspect we want to manage. In this way, we have something to attach the semantic label to.

E. Using Around Advices

In the AspectJ implementation around advices cannot be used by collaboration aspects, because it could harm base functionality. This is because a collaboration aspect could be deactivated in some context, and in the case of an around advice this situation would lead to skipping the execution of the method being advised.

This is not a problem when using Ruby/Aquarium, since aspects containing an around advice can be unwoven, leaving the original joinpoint functional.

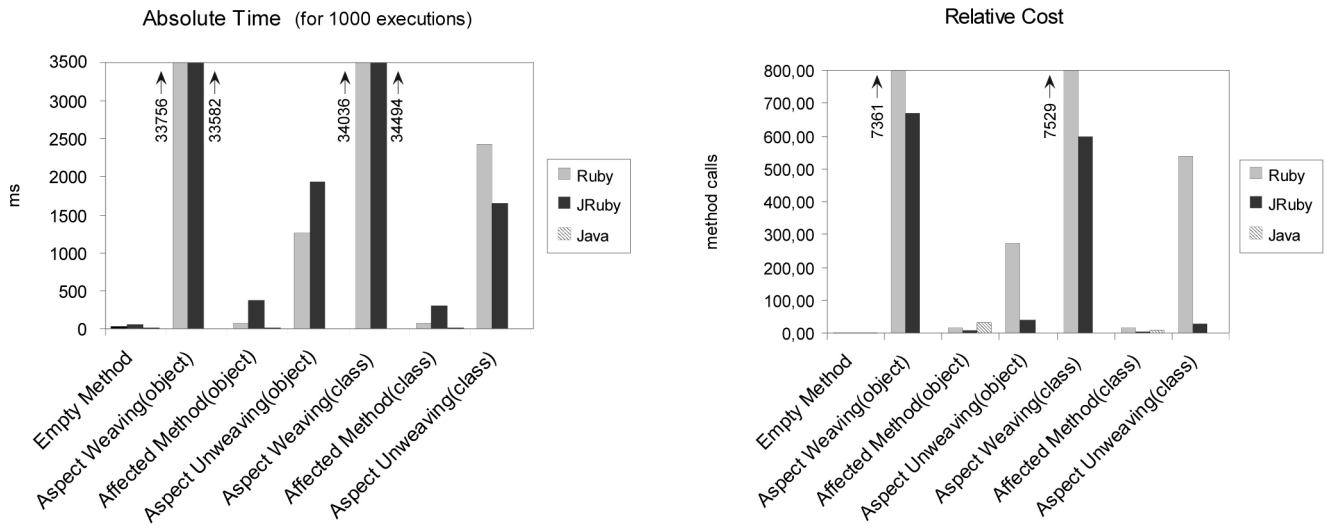
F. Expressing and Attaching Semantic Information

Annotation support provided by Java is well-suited for expressing semantic information. As annotations are types they can be type-checked. For example, it is possible to avoid adding unexpected semantic labels, because annotations must be defined beforehand. Annotations can be examined at compile time or accessed via a reflective API at runtime. We decided to use them at runtime, inspecting them when aspects are instantiated.

In contrast, Ruby does not provide any built-in mechanism for attaching information to code. In this case, it is necessary to go through a workaround defining constants and attaching them using the `method_missing` mechanism we described previously.

G. Performance

In this section we analyse the performance overhead imposed by our coordination approach in both implementations. As the mechanisms used for achieving the runtime adaptation of aspect behavior depend on the platform, each



(a) Absolute Time

(b) Relative Cost

Figure 3: Charts comparing absolute time and the relative cost of executing advised methods, weaving and unweaving.

platform must be analyzed separately. In the case of Ruby we also measured the time for our tests running on top of JRuby [9], the Java based Ruby interpreter. Henceforth we refer with Ruby to the Ruby native interpreter, and JRuby to its Java based counterpart.

The coordination approach is based on the idea of evaluating context and then deciding which aspects must run and which ones not. So, the overhead imposed is mainly due to the context monitoring, rule execution and (de)activation of aspects. Context monitoring is also done by aspects. Therefore we measured the overhead for executing advised methods, for performing (un)weaving operations (in the case of Ruby), for executing advised advices (in the case of AspectJ) and finally the time for executing the coordination rules in each platform.

To avoid the obvious differences between a byte-code precompiled language such as Java and an pure interpreted one as Ruby, we compared the overhead imposed against a relative unit, which is the time needed to perform a method call in each platform.

Table I: Compared execution times for Ruby/Aquarium (1000 executions)

Operation	Units	Milliseconds
Method Call	1	4,6
Affected method (object level)	15	67
Aspect Weaving (object level)	7360	33756
Aspect Unweaving (object level)	275	1250
Aspect Weaving (class level)	7529	34036
Aspect Unweaving (class level)	540	2440

Tables I and II show the time need to execute methods,

Table II: Compared execution times for JRuby/Aquarium (1000 executions)

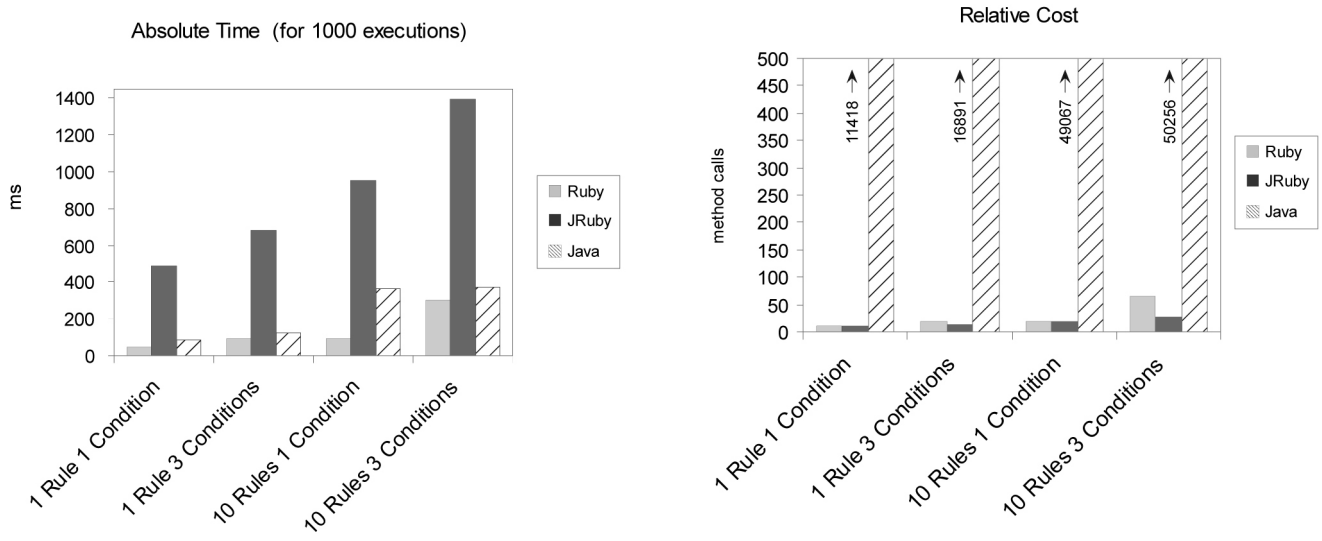
Operation	Units	Milliseconds
Method Call	1	50,2
Affected Method (object level)	7.28	365.6
Aspect Weaving (object level)	668.98	33582
Aspect Unweaving (object level)	38.43	1929
Aspect Weaving (class level)	596.79	34494
Aspect Unweaving (class level)	28.44	1643.8

advised methods (called *affected methods*), and (un)weaving for classes and objects. JRuby is slower than Ruby native interpreter in absolute time for (un) weaving (see Fig. 3a), but it is noticeable that the relative times for weaving and unweaving are better for JRuby, being almost 10 times faster than Ruby (see Fig. 3b). JRuby is also twice faster than Ruby for advised method calls, when comparing their relative times.

Table III: Compared execution times for Java/AspectJ (1000 executions)

Operation	Units	Milliseconds
Method Call	1	0,0074
Affected Method (Advised method + Execution Controller)	31	0,2314

Table III shows the comparative costs for methods and affected methods, which are advised methods whose advices



(a) Absolute Time

(b) Relative Cost

Figure 4: Charts comparing absolute time and the relative cost for rules execution.

Table IV: Compared execution times for Ruby/Ruleby

Operation	Units	Milliseconds
Method Call	1	4,6
1 Rule 1 Condition	10	45
1 Rule 3 Conditions	20	92
10 Rules 1 Condition	20	92
10 Rules 3 Conditions	66	304

Table V: Compared execution times for JRuby/Ruleby

Operation	Units	Milliseconds
Method Call	1	50,2
1 Rule 1 Condition	10	488
1 Rule 3 Conditions	14	685
10 Rules 1 Condition	19	954
10 Rules 3 Conditions	28	1392

are at the same time controlled by the ExecutionController[1]. In this case the relative time needed for executing an advised method call (using singleton aspects) is similar to JRuby/Aquarium. For the case when an aspect is instantiated in a per object basis (`perthis` keyword), the advised method doubles its execution time (16 units against 8 for aspect singletons). The advised advice for our per-object instantiated aspects takes almost 4 times the time needed for a regular advised method (31 units against 8).

Tables IV and V show the time needed for evaluating rules using the Ruleby engine on top of Ruby native interpreter and JRuby. The relative times in this case are quite similar.

Table VI shows the costs of invoking the rule-engine for the Java based rule engine. Drools is a full featured rule engine, which results to be slow in execution, specially when it is compared with regular times for method calls in Java (see Fig. 4b). But in terms of time, the performance of Drools is comparable to Ruleby running on Ruby and better than Ruleby running on JRuby (see Fig. 4a).

Table VI: Compared execution times for Java/DRools

Operation	Units	Milliseconds
Method Call	1	0,0074
1 Rule 1 Condition	11418	84.5
1 Rule 3 Conditions	16891	125
10 Rules 1 Condition	49067	363.1
10 Rules 3 Conditions	50256	371.9

In the case of Ruby/Aquarium an aspect can be completely (un)woven at runtime. As it is a costly operation, in cases where aspects need to be frequently (de)activated it would be better to use an AspectJ based solution (as it does not requires (un)weaving). On the other hand, if aspect activation cycles are stable and long enough, the Ruby based implementation is more appealing, since the overhead imposed by an aspects is completely removed after it is unwoven.

Considering the relative number we conclude that if aspects deactivation cycle is more frequent that 475 aspect executions, it is better to use AspectJ. This is because we need long periods (more than 475) of no (de)activation to amortize the cost of performing (un) weaving in Ruby.

[1]In the tables and charts we use the terms *affected methods* to denote those methods that are advised by the collaborative behavior. In the case of the Ruby/Aquarium implementation it is equivalent to an advised method, but in the case of AspectJ, an affected method measure also considers the existence of the ExecutionController.

V. Conclusion and Future Work

In this work we have compared the implementation of our aspect coordination approach in two different platforms: Java/AspectJ and Ruby/Aquarium.

From the comparison of these implementations we have shown that Ruby provides more appropriate support for our approach, as it is easy to manipulate aspect instances. Furthermore, its ability of performing (un)weaving at runtime is suitable for implementing the notion of aspect (de)activation we need. The downside of Ruby is its performance: weaving is notably slow, even compared to other operations in Ruby, such as method calls.

In contrast, the Java/AspectJ implementation is faster in absolute and relative terms, but the automatic instantiation mechanism provided by AspectJ makes difficult the task of getting the aspects instances we need to manipulate. Furthermore, in the AspectJ implementation there is always an extra overhead corresponding to our aspect (de)activation infrastructure. However, the base methods affected by the collaboration aspects and our coordination infrastructure run notably faster than their Ruby based versions.

From the conceptual point of view Ruby/Aquarium also offers the advantage of allowing the programmer to use `around` advices, which are not allowing in our approach while using AspectJ.

To sum up, Ruby/Aquarium is a great choice if aspect (de) activations happens no so frequently, since unweaving removes the whole overhead associated with aspects.

As future work we plan to evaluate this coordination mechanisms for other domains, and possibly other platforms for its implementation. Our approach works now as a workaround that enables to control de execution of the aspects based on the context information. Therefore, we need to study if proposed mechanism can be integrated into the aspect language or platform. This would give the aspect programmer some tools for coping with aspect interactions.

References

[1] Aquarium framework. http://aquarium.rubyforge.org/.

[2] AspectJ documentation: Language Semantics. http://www.eclipse.org/aspectj/doc/released/progguide/semantics-aspects.html.

[3] J. Blom. Personalization - a taxonomy. In *CHI 2000 Workshop on Designing Interactive Systems for 1-to-1 Ecommerce*, 2000.

[4] Drools rule engine. http://www.jboss.org/drools.

[5] B. DuCharme. *XML: the annotated specification*. The Charles F. Goldfarb series on open information management. Prentice-Hall PTR, Upper Saddle River, NJ 07458, USA, 1999.

[6] Google docs. http://docs.google.com/.

[7] The Java language specification. http://java.sun.com/docs/books/jls/third_edition/html/j3TOC.html.

[8] JCP. A metadata facility for the Java programming language, 2004. http://www.jcp.org/en/jsr/detail?id=175.

[9] JRuby pure Java implementation of the Ruby language. http://www.jruby.org/.

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. pages 327–353. Springer-Verlag, 2001.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[12] MSDN. C # language specification - attribute specification. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/csspec/html/vclrfcsharpspec_17_2.asp.

[13] Ruby programming language. http://www.ruby-lang.org/en/.

[14] Ruleby a Ruby rule-engine. http://rubyforge.org/projects/ruleby/.

[15] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Association aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 16–25, New York, NY, USA, 2004. ACM.

[16] F. Sanen, E. Truyen, B. D. Win, W. Joosen, N. Loughran, G. Coulson, A. Rashid, A. Nedos, A. Jackson, and S. Clarke. Study on interaction issues. Technical Report AOSD-Europe Deliverable D44, AOSD-Europe-KUL-7, Katholieke Universiteit Leuven, 28 February 2006 2006.

[17] T. Schummer and S. Lukosch. *Patterns for Computer-Mediated Interaction (Wiley Software Patterns Series)*. Wiley, August 2007.

[18] D. Torres, A. Fernández, G. Rossi, and S. E. Gordillo. Fostering groupware tailorability through separation of concerns. In J. M. Haake, S. F. Ochoa, and A. Cechich, editors, *CRIWG*, volume 4715 of *Lecture Notes in Computer Science*, pages 143–156. Springer, 2007.

[19] J. Viega and J. Voas. Can aspect-oriented programming lead to more reliable software? *IEEE Softw.*, 17(6):19–21, 2000.

[20] C. Walls, N. Richards, and R. Oberg. *XDoclet in Action (In Action series)*. Manning Publications Co., Greenwich, CT, USA, 2003.

[21] A. Zambrano, L. Polasek, and S. Gordillo. Decoupling personalization aspects in mobile applications. In *HCI related papers of Interaccin 2004*, pages 29–40. Springer Netherlands, 2005.

[22] A. Zambrano, T. Vera, and S. Gordillo. Solving aspectual semantic conflicts in resource aware systems. In W. Cazzola, S. Chiba, Y. Coady, and G. Saake, editors, *Third ECOOP Workshop on Reflection, AOP and Metadata for Software Evolution*, Nantes, France, 2006.