Dynamic Weaving in CAM/DAOP: An Application Architecture Driven Approach

Lidia Fuentes
Dpto. Lenguajes y Ciencias de la Computación
University of Málaga, SPAIN
lff@lcc.uma.es

Mónica Pinto
Dpto. Lenguajes y Ciencias de la Computación
University of Málaga, SPAIN
pinto@lcc.uma.es

Pablo Sánchez
Dpto. Lenguajes y Ciencias de la Computación
University of Málaga, SPAIN
granpablo@lycos.es

ABSTRACT

Dynamic weaving is much more flexible than static weaving because the separation of concerns remains at runtime. This results in highly configurable and adaptable applications, since the rules that govern the weaving of aspects can evolve during the application execution, according to different criteria – i.e. user preferences, execution context, etc. In this paper we describe the dynamic weaving mechanism offered by CAM/DAOP, our own component and aspect platform. The most relevant feature is that the plugging of components and aspects is driven by the application architectural information, which is loaded into the internal structures of the CAM/DAOP platform to be consulted at runtime.

1. INTRODUCTION

During the last years, Aspect-Oriented Software Development (AOSD) [2] became a more and more consolidated software technology. Hundreds of new aspect-oriented approaches appeared that cope with the separation of concerns principle in different ways. In this paper, our main interest rely on the different weaving processes, which can be static (performed during compilation) or dynamic (performed at runtime).

In those approaches where the weaving process is static [5, 8, 10] the object/component and aspect code is mixed at compile-time. Static composition provides high performance, but separation of concerns is lost at runtime. Although they normally use introspection to provide reflective information about join points at runtime, the number and type of join points affected by an aspect cannot be modified after compilation.

Dynamic weaving is an interesting alternative to static weaving. It is much more flexible than static weaving because the separation of concerns remains at runtime, enabling, in some cases, the late binding between objects/components and aspects. Approaches that offer a dynamic weaving mechanism [17, 11] are mainly based on a reflection mechanism that offers the ability to modify the application semantics while the application is running. This adaptability is commonly achieved by implementing a Meta Object Protocol (MOP) as part of the language interpreter that specifies the way a program may be modified at runtime.

In this paper we describe the dynamic weaving mechanism of CAM/DAOP [13]. CAM (Component-Aspect Model) is a new component and aspect model that defines components and aspects as first-order entities. The underlying infrastructure supporting the CAM model is a Component-Aspect Platform (DAOP, a Dynamic Aspect-Oriented Platform) where the plugging of software aspects into components is performed at runtime.

One of the most relevant features of our approach is that the dynamic weaving among components and aspects is driven by the information about the software architecture of the application. Concretely, CAM/DAOP uses an XML-based architectural description language (DAOP-ADL) [14] to describe components and aspects, together with the composition rules (i.e. the declaration of aspect pointcuts) that govern the weaving of components and aspects. The platform weaving mechanism loads and consults this information at runtime to establish the connections among components and aspects. This is particularly useful because we make components and aspects much more reusable, isolating the dependencies between them in the platform internal structures. In addition, this information can be easily adapted at runtime, improving the flexibility and adaptability of the final application.

The complete description of CAM/DAOP and the DAOP-ADL language is beyond the scope of this paper and can be found in [13, 14]. In this paper we focus on describing the dynamic weaving mechanism offered by the DAOP platform, and the main advantages obtained from consulting the application architectural information at runtime. After this introduction, the paper is organized as follows. Next section compares other AOSD approaches offering dynamic weaving. Then, section 3 describes the architecture of the DAOP platform and section 4 the DAOP dynamic weaving mechanism. In order to cope with the limitations introduced by dynamic weaving, in section 5 we describe our approach to cope with these limitations in CAM/DAOP. Finally, we

present our main conclusions in section 6.

2. RELATED WORK

Table 1 contains a brief description of several AO frameworks and platforms providing dynamic composition. Regarding the main features of CAM/DAOP, we have analyzed the different works particularly interested in: (i) how they incorporate the component concepts; (ii) the separation (or not) of advice and pointcuts in isolated entities; allowing the reuse of aspects, (iii) the mechanisms to express pointcuts; with special interest in if they describe in some way architectural information (iv) if they use an invasive or non-invasive model, and (v) the mechanisms they use to perform dynamic weaving.

PROSE [15] is an AO platform with dynamic composition, for using aspects with objects. Aspects can intercept points that are part of the internal behavior of objects. Its main contribution is that the platform weaves and unweaves aspects directly in the Java Virtual Machine (JVM), inserting the aspect advice directly into the native code generated by the just-in-time (JIT) compiler. In addition, pointcuts and advice are implemented in PROSE in the class representing the aspect, with the drawback of reducing the (re)use of the aspect advice.

Another similar approach is JAC [11], an AO framework that uses the reflexive API BCEL for adding aspects. Aspects in JAC are dynamically deployed and undeployed on top of running application objects using wrappers and aspect containers. JAC pointcuts are not specified as part of the aspect definition but in a third-party entity available at runtime, making aspects more reusable. In addition, JAC uses AspectComponent configuration files (.acc files) or XML files to configure externally the aspect evaluation rules (or pointcuts). It does not really define a component platform and its components cannot be considered software components in the CBSD sense. JAC distributed protocols are introduced as an extra mechanism to be able to distribute aspects in different hosts.

Another AO framework that performs dynamic composition of objects and aspects is AspectWerkz [3], which implements several weaving techniques. It offers static weaving like AspectJ [5], and other based on different mechanisms: JSR-163 JVMTI, hotswap, and bootclasspath. Using these technologies, aspects are composed with objects at runtime, by modifying objects byte code after class loading. Uses XML files to define pointcuts separately from aspect implementations. However, aspects in AspectWerkz are applied to objects and not to components, defining an invasive model. Its weaving mechanism completely relies on the Java technology.

JAsCo [17] is an aspect oriented implementation language that defines a new component model compatible with the JavaBeans component model. Aspects in JAsCo can be applied, adapted and removed at runtime. JAsCo introduces two concepts: aspect beans that encapsulate advice, and connectors that define pointcuts. Both advice and pointcuts can evolve separately, increasing the reuse of advice. Dynamic connector loading and unloading is possible in the JAsCo connector registry. However, JAsCo connectors have to be compiled, reducing their runtime adaptation. Aspects interception is performed only before or after a method execution, defining a non-invasive model, although an aspect can replace the normal execution of a method.

Table 1: Related Work in Dynamic AOSD Approaches

	PROSE	JAC	Aspect Werkz	JAsCo	Lasagne	AOP JBoss	CAM/ DAOP
I	No	No	No	Yes (Beans)	Yes	J2EE	Yes
II	No	Yes	Yes	Yes	Yes	Yes	Yes
III	No	.acc or XML files	XML files	No	Composition policy file	XML files	XML files
IV	Yes	Yes	Yes	No	No	Yes	No
V	JVM Class Loader/ HotSwap2	Wrappers + Container	HotSwap / JSR-163	Hotswap + Connector Registry	Decorator-like wrappers	Container	Middleware Layer

(I) CBSD concepts (II) Separation od advice and pointcuts (III) External Configuration of pointcuts (IV) Invasive model (V) Dynamism

Lasagne [18] defines a platform-independent architecture for dynamic customization of component-based distributed systems using decorator-like wrappers, according with a non-invasive model, that only intercept incoming/outcoming messages. In Lasagne, the composition logic is completely separated from the code of the components, and of the extensions as well, increasing their reuse. This information is specified in composition policy files that can be dynamically attached to the system. Lasagne composes extensions at the instance-level instead of at the class-level, giving Lasagne a runtime performance overhead, although the composition mechanism is much more flexible.

Finally, the JBoss AOP [1] framework is built on top of the JBoss J2EE application server, and tries to solve the limitation of providing just a set of built-in services. Aspect advice in JBoss AOP is implemented using interceptors, according with a invasiveness model, which is not required for components models. The JBoss AOP framework has the advantage that it separates advice and pointcuts in different entities, where pointcuts are configured using XML descriptor files.

There are other dynamic approaches not covered in this section due to the lack of space such as JMangler [6], Caesar [9], EAOP [4], Rapier-LOOM.NET [16] and Weave.NET [7], among others.

3. THE DAOP PLATFORM

The DAOP platform is a distributed component-aspect middleware platform for running applications conforming to the CAM model [13]. Figure 1 shows its architecture, which contains information about the services and facilities it offers to components and aspects (elements that appear above the DAOP Platform class in figure 1), together with the information the platform stores to provide such services (classes below the DAOP Platform class in figure 1).

Similar to other component platforms, the DAOP platform provides a set of common services to develop distributed applications, such as the instantiation of components (Com-

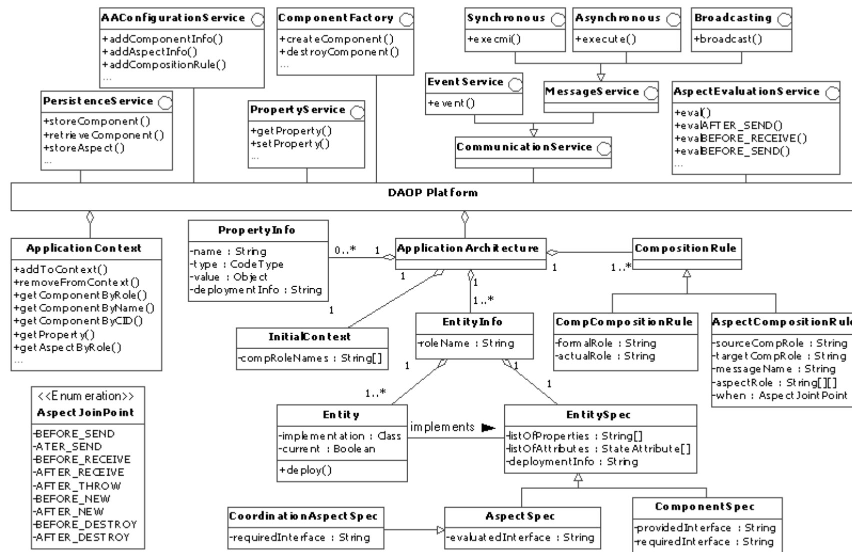

Figure 1: The DAOP Platform Architecture

ponentFactory interface), the communication of components (CommunicationService interface), the evaluation of aspects (AspectEvaluationService interface), the storage of properties (PropertyService interface), the persistence service (PersistenceService interface) and the dynamic adaptation of the application architecture (AAConfigurationServices interface).

Regarding the internal infrastructure of the DAOP platform its information is arranged basically in two objects. The ApplicationArchitecture object, which stores the architectural description of the application; and the ApplicationContext object, which holds the current list of component, aspect and property instances.

The DAOP model in figure 1 is a platform independent model that may be implemented using different middleware technologies, such as .NET, CORBA or Java/RMI. Currently, we have a Java implementation that uses Java/RMI as the base communication mechanism, and the reflective package to help us to implement dynamic composition.

Regarding other AOSD approaches the main advantage of CAM/DAOP is that it supports the development of peer-to-peer distributed applications, since DAOP is neither a client/server approach nor uses a central manager to be notified of messages and events that occur within the application. Instead, the DAOP platform is a distributed platform that do not need to define extra mechanisms to distribute aspects in different hosts. An application in DAOP is distributed among different hosts, where a local instance of the DAOP platform is running. These DAOP platform instances communicate among themselves, being possible for all the components and aspects in a DAOP application to communicate and collaborate amongst themselves. During the deployment of the application it is determined the number of instances that the DAOP platform creates for each aspect and how these instances are distributed. Also components are distributed through the different nodes of the DAOP application.

Other relevant advantage of DAOP is that components and aspects are plain code in the language where the platform was implemented. For instance, they are plain Java code in the current Java/RMI implementation. Neither the use of new constructions nor the generation of stubs and skeletons are needed in order to implement DAOP components and aspects. Only the use of the services offered by the DAOP platform are needed. Other advantages of CAM/DAOP are shared with other AOSD approaches, such as: (1) the application of aspects to components [1, 17, 18] instead of objects [3, 11, 15]; (2) the definition of a non-invasive model similar to most component-based approaches [17, 18], where it is not possible to intercept join points that are part of the internal behavior of a component. Instead, only the behavior exposed through the explicit interfaces of components can be intercepted, considering components as black-box entities, and (3) aspects are applied to components at runtime, and the information needed to perform the dynamic weaving of components and aspects is described using a declarative language such as in [3, 18] and is not hard coded as part of the application implementation classes.

4. DAOP DYNAMIC WEAVING

In this section we will explain the main features of the DAOP dynamic weaving mechanism. The weaving in DAOP is dynamic since components and aspects remain as separate entities during the application execution. By aspect weaving we mean the execution of the corresponding aspect advice when a join point is intercepted by the DAOP platform.

Even though the weaving mechanism in DAOP is based on the interception of messages and events, there is an important difference between DAOP and traditional component platforms, such as CORBA, CCM/CORBA and EJB/J2EE. Whereas these platforms offer a concrete number of services that cannot be extended by users, in our approach it is possible to separate any crosscutting property. The difference can be found in how both approaches manage these properties. In our approach the provision of these properties

does not rely on the platform provider, like in CORBA, CCM/CORBA and EJB/J2EE. Instead, components and aspects in DAOP are first-class entities that coexist at the application level. Consequently, using CAM/DAOP application's developers decides how to divide the application functionality into components and aspects, with no limit to the kinds of aspects.

In the rest of this section we will give specific details about the DAOP dynamic weaving mechanism. For that, we will use the example showed in figure 2.

This example is taken from a virtual office application we have developed. In this application users (User component) join a shared space (VirtualOffice component) to collaborate with other users. The figure shows that the user is authenticated before joining the office, by evaluating the Authentication aspect before the *join* message is sent by the User component (BEFORE_SEND join point). Additionally, when the user leaves the office, the Persistence aspect is evaluated after the *leave* message is received by the VirtualOffice component (AFTER_RECEIVE join point), to make persistence the status of the office for that user. This status can be restored the next time the user joins the office, by evaluating the Persistence aspect after the *join* message is received by the VirtualOffice component (AFTER_RECEIVE join point).

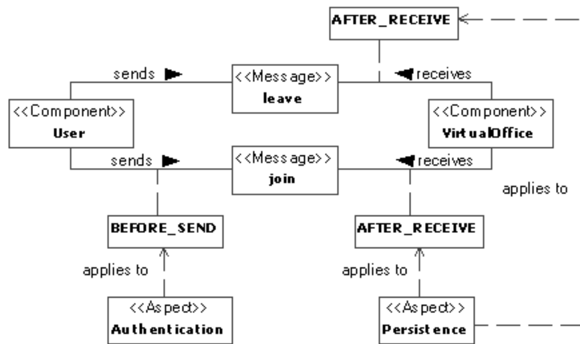

Figure 2: A CAM/DAOP Example

We have used the CAM model to design the example in figure 2. Although the CAM model has not been described in this paper, in order to understand the example we only need to know that the CAM model describes components (with stereotype ⟨⟨Component⟩⟩), aspects (with stereotype ⟨⟨Aspect⟩⟩) and the composition among them, expressed in terms of applies to relationships. These relationships describes, in this example, which aspects are applied when a message (with stereotype ⟨⟨Message⟩⟩) is sent or received. The join points are the same described in the previous section.

From the point of view of the application developer, the main step in the development of a CAM/DAOP application – i.e. related to the dynamic weaving mechanism, is the description of the application architecture. From the point of view of the platform, the main step is the use of the information about the application architecture to perform the dynamic weaving of components and aspects.

4.1 Description of the Application Architecture

Step 1: Describe the application architecture information, including the declaration of aspect's pointcuts

As we mentioned in the introduction, an important feature of our approach is that the DAOP platform stores a description of the application architecture (AA), which is consulted by the platform to perform component and aspect instantiation and dynamic weaving. This information is specified during the design and architecture phases using the DAOP-ADL [14] language, an XML-based architectural description language. Then, during the execution, when an instance of the DAOP platform is created, the XML document is parsed and the structure of the CAM/DAOP application is stored in the ApplicationArchitecture class and its subclasses (see figure 1).

With this language, firstly, all the components and aspects that could be instantiated in the application are described; each of them identified by its role name. Role names are identifiers that we use in CAM/DAOP to uniquely identify both components and aspects. These role names are architectural names that are used for component-aspect composition and interaction, allowing loosely coupled communication among them – i.e. no hard-coded references need to be used for exchanging information, but just a role name identifying the source and the target of a message.

Then, the aspect composition rules, – ie. pointcuts, defining when and how to apply aspects to components are described, where components and aspects are referred by their corresponding role names. In this example, we have two components with role names "user " and "office", and two aspects with role names "authentication" and "persistence". Aspect composition rules are stored in the AspectCompositionRule class in figure 1.

There are three kinds of aspect composition rules. In the following code, we have expressed these rules in EBNF in order to better explain which kind of information can be provided to define pointcuts and, even more important, which kind of information can be adapted to modify component and aspect weaving at runtime.

The first kind of rule describes which aspects are applied when components communicate by sending messages:

```
1 <message_pc>   ::= <message_jp> <message_description>
                     '{'<aspect_composition>'}'
2 <message_jp>   ::= BEFORE_SEND | AFTER_SEND | BEFORE_RECEIVE |
                     AFTER_RECEIVE | AFTER_THROW
3 <message_description> ::= <source> <target> <message>
4 <message>      ::= [<TYPE>] <Ident> '(' (<TYPE>)* ')'
5 <source>       ::= <rolename> [<message>]
6 <target>       ::= <rolename>
7 <aspect_composition> ::= '{' sequential_aspects '}' |
                          '{' sequential_aspects '}'
                          <aspect_composition>
8 <sequential_aspects> ::= (<aspect>)+
```

Notice that the description of the intercepted message (see line 3) includes the source and the target components, identified by their role names. In order to make the definition

of pointcuts much more flexible, the source of a message includes not only the role name of the source component but, optionally, the definition of the method from which that message was sent (see line 5).

The aspects to be evaluated are described using a bi-dimensional array of strings with the format $\{\{A_1\},\{A_2\}, \{\{A_3,A_4\}\}$ (see lines 7 and 8) where every A_i is an aspect role name. This bi-dimensional structure allows us to specify two kinds of aspect evaluation: sequential evaluation and parallel evaluation. Aspects enclosed in the outer brackets, for instance A_1 and A_2, are evaluated sequentially. On the other hand, aspects in the inner brackets, for instance A_3 and A_4, will be evaluated concurrently.

Coming back to our example there are three pointcut declarations that correspond with this kind of rule:

```
1 BEFORE_SEND   user * office join(String)  {{authentication}}
2 AFTER_RECEIVE user * office join(String)  {{persistence}}
3 AFTER_RECEIVE user * office leave(String) {{persistence}}
```

which can be interpreted as follows: (1) *before sending* the message *join(String)* from the component with role name *user* to the component with role name *office* the aspect with role name *authentication* must be evaluated. The wildcard "*" indicates that this rule is applicable independently of the method in the source component from which the *join(String)* message is sent; (2) *after receiving* the same message with the same source and target components the aspect with role name *persistence* has to be evaluated, and finally (3) *after receiving* the *leave(String)* message, being the source and target components the same, the aspect with role name *persistence* is applied once again.

There are other kind of rule to describe which aspects are applied when components communicate by throwing events:

```
 9 <event_pc>     ::= <event_jp> <event_description>
                      '{' <aspect_composition> '}'
10 <event_jp>     ::= SEND_EVENT
11 <event_description> ::= <source> <message>
```

This kind of rule only differs from the previous one in that the description of events (see line 11) does not include its target.

Finally, the last kind of rule describes which aspects are applied when components are instantiated or eliminated from the system:

```
12 <component_pc> ::= <component_jp> <component_description>
                      '{' <aspect_composition> '}'
13 <component_jp> ::= BEFORE_NEW | AFTER_NEW | BEFORE_DESTROY |
                      AFTER_DESTROY
14 <component_description> ::= <source> <rolename>
```

The main advantages of describing the AA information as done in CAM/DAOP are the following:

Advantages

1. *The moment of the evaluation (when creating/destroying components and when sending and/or receiving messages and events), the kind of evaluation (sequential or parallel), and the information about which components are affected by aspects is not hard-coded as part of the component or aspect implementations. Instead, this information is taken out of components and aspects and stored in the* ApplicationArchitecture *class inside the DAOP platform, achieving a higher degree of component and aspect independence.*
2. *Pointcut declarations are not distributed through the different aspects in the application. Instead they are centralized in the document describing the AA. In consequence, designers and programmers are able to comprehend the structure of the application, facilitating the understanding and evolution of final applications.*

4.2 Dynamic Weaving of Components and Aspects

Step 2: At runtime, the DAOP platform consults the AA information that was previously stored in its internal structures

During the deployment phase, the document describing the AA is stored together with all the other application resources, i.e. component and aspect implementations, images, etc. Later at runtime, when a user join an application, the information about its AA is downloaded as part of the application specific applet, is de-serialized at the user site and the information stored in the platform.

Finally, this information is used by the platform to perform the dynamic plugging of components and aspects. This occurs when components create or destroy other components using the corresponding methods of the ComponentFactory interface (see figure 1). Also, it occurs when components communicate between them using the component communication primitives offered by the platform (see the CommunicationService interface and all its subinterfaces in figure 1). As in other component platforms (e.g., CORBA), DAOP allows components to send synchronous and asynchronous messages, as well as to broadcast a message to several target components. DAOP also allows components to throw events to other components.

By intercepting the throwing of events, CAM/DAOP provides a join point that occurs within the execution of a component method, similar to the *around* join point in other approaches like AspectJ or PROSE. The difference with these white-box approaches is that the DAOP platform can only intercept the points that the component makes visible throughout the throwing of events, considering it still a black-box component. The handling of events is resolved at runtime by a *coordination aspect*, which is out of the scope of this paper [12].

Coming back to our example, figure 3 describes how components and aspects are dynamically plugged when the component with role name user sends the join(String) message to the component with role name office; using for that the DAOP execmi() communication primitive (step 1 in figure 3).

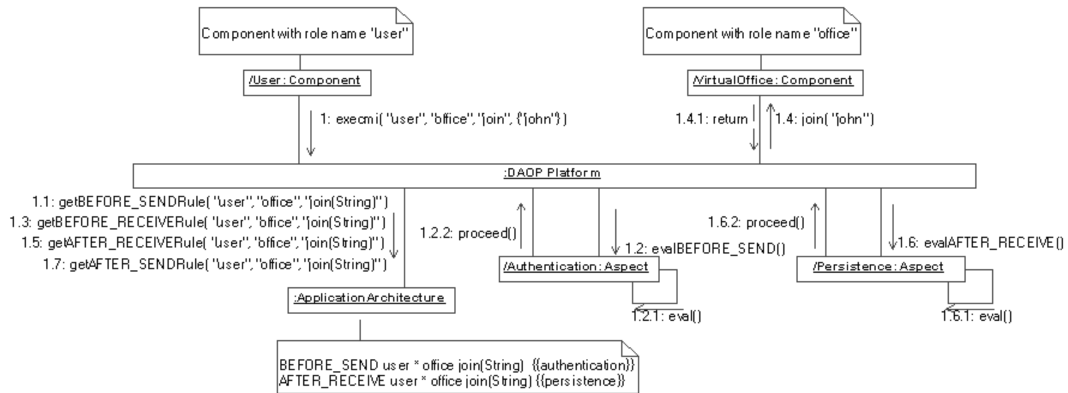

Figure 3: Dynamic Weaving Mechanism

At this moment, the DAOP platform takes the control and, by each intercepted join point, consults the information about the aspect pointcuts, which are stored in the :ApplicationArchitecture object (steps 1.1, 1.3, 1.5 and 1.7). As specified by these pointcuts, the platform invokes the evalBEFORE_SEND() method in the Authentication aspect (steps 1.2, 1.2.1, and 1.2.2), then it invokes the join("john") method in the target component (steps 1.4 and 1.4.1) and, finally, it invokes the evalAFTER_RECEIVE() method in the Persistence aspect (steps 1.6, 1.6.1 and 1.6.2).

In order to perform this evaluation, DAOP aspects must implement the AspectEvaluationService interface (see figure 1). This is mandatory since, as described above, the platform will invoke one of the methods in this interface to evaluate an aspect – i.e. to execute the aspect advice. There is a method in the AspectEvaluationService interface for each CAM/DAOP join points. All these join points are enumerated in the AspectJoinPoint enumeration class shown in figure 1. Additionally, CAM/DAOP also provides the eval() method, which is a general method that allows to make aspects completely independent from the intercepted join point. For instance, this is useful for a *Trace* aspect that records information about the application execution. Its behavior is always the same, independently of the join point where the aspect is evaluated.

The main advantages of the CAM/DAOP weaving mechanism are the following:

Advantages

1. *We close the usual "gap", or loss of information, between design and implementation levels, since exactly the same information generated during the description of the application architecture is then used at runtime*
2. *This is a real "runtime" weaving mechanism, not a "load time" one*
3. *During the execution it is possible to adapt the behavior of the application by adding, removing or modifying the information about the application architecture. The platform will assure that this information remains consistent if changes are performed by some user*

4.3 Dynamic adaptability of the Application Behavior

Step 3: Optionally, the application behavior can be adapted at runtime by modifying the information about the application architecture stored in the DAOP platform

Finally, if required by the application, the information about the application architecture that is stored in the internal structures of the DAOP platform can be adapted at runtime in order to modify the behavior of the application. This can be done without stopping or recompiling the application. In the Java implementation this also means that classes do not need to be loaded again with the Java class loader. New components and aspects can be incorporated to the application, or existing ones removed from it. Also, aspect composition rules can be modified, added or removed from the description of the application architecture, changing how the plugging of components and aspects is performed.

In order to do that the DAOP platform offers the AAConfigurationService interface (see figure 1). Currently, we are developing a system tool that will support any kind of application that desire to adapt its application architecture. Additionally, applications can incorporate their own tools to adapt it according to their specific necessities. In both cases, the AAConfigurationService interface must be used.

Advantages

1. *Final applications are more adaptable and evolvable*

5. IMPROVING PERFORMANCE: COMBINING STATIC AND DYNAMIC WEAVING

The main drawback of dynamic approaches is that in many cases static weaving is faster and therefore offers better performance than dynamic weaving. In this sense, in CAM/DAOP the use of reflection and the dynamic composition mechanism may introduce some overhead at runtime.

The performance study we have performed with CAM/DAOP has been using our Java/RMI implementation. Currently,

we have a virtual office application[1], and after one year of evaluation we can state that the performance is satisfactory. Even when Java poses significant drawbacks related to efficiency, we find that the overload of dynamic evaluation of aspects is not so critical in distributed systems based on Java. For instance, component and aspect creation through the platform takes 30 ms, and the time to incorporate the evaluation of the aspect at runtime is insignificant (around 20 ms). Comparing this evaluation time with the time spent loading a web page from the same host, they are insignificant.

Nevertheless, and taking into account that there are aspects that might not need to be adapted at runtime, we are now developing a alternative static composition mechanism using the Java BCEL API. One of the goals of this extension is to be able to study these run-time overheads to check if they are really relevant to the application performance and how we are able to reduce the overhead in some situations. Extending the DAOP-ADL language to specify whether an aspect must be woven into components statically or dynamically, this tool manipulates component class files to weave static aspects at compile time. Additionally, it generates a new DAOP-ADL document which only contains the declaration of pointcuts for dynamic aspects. Therefore, dynamic aspects will be weaved at runtime as described previously.

Using this approach we sacrifice the independence of aspects and components at implementation level, in order to increase the application's performance. Independence is still maintained at design and architectural levels.

6. CONCLUSIONS AND FUTURE WORK

In this paper we have presented the dynamic weaving mechanism of CAM/DAOP. The main contribution of this mechanism is that it performs the plugging of components and aspects using the information about the application architecture. This information is described during the architecture description phase using the DAOP-ADL language, and includes the definition of aspect's pointcuts. During the application instantiation, pointcuts are loaded in the internal structures of the DAOP platform, to be consulted at runtime. This have two important advantages: (1) aspects are more reusable in different contexts since they do not include the pointcut declaration, and (2) we bridge the gap between design and implementation, since the weaving of components and aspects is performed according to exactly the same information that was provided during the design and architecture phases.

Additionally, aspect's pointcuts can be added, modified or deleted at runtime without stopping or (re)compiling the application. The dynamic weaving mechanism offered by the platform allows that in order to automatically adapt the application behavior, only the information about the application architecture was modified, using for that the corresponding service of the DAOP platform.

7. ACKNOWLEDGMENTS

This work is supported by European Commission grant IST-2-004349: European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe), 2004-2008.

[1] see http://150.214.108.46/CoopTEL

8. REFERENCES

[1] The aspect oriented programming and jboss tutorial. http://www.onjava.com/pub/a/onjava/2003/05/28/aop_jboss.html.

[2] Aspect-oriented software development web site. http://aosd.net.

[3] Aspectwerkz web page. http://aspectwerkz.codehaus.org/.

[4] Event-based aspect-oriented programming. http://www.emn.fr/x-info/eaop/tool.html.

[5] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An Overview of AspectJ. In *Proc. of ECOOP'01*, pages 327–355, Budapest, Hungary, 18-22 June 2001. Springer-Verlag.

[6] G. Kniesel, P. Costanza, and M. Austermann. Jmangler - a powerful back-end for aspect-oriented programming. In R. Filman, T. Elrad, S. Clarke, and M. Aksit, editors, *Aspect-oriented Software Development.* Prentice Hall, 2004. To appear.

[7] D. Lafferty and V. Cahill. Language-independent aspect-oriented programming. In *Proc. of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 1–12, California, USA, 2003.

[8] K. Lieberherr, D. H. Lorenz, and J. Ovlinger. Aspectual collaborations: Combining modules and aspects. *The Computer Journal*, 46(5):542–565, Sept. 2003.

[9] M. Mezini and K. Ostermann. Conquering aspects with caesar. In *Proc. of the Second International conference on AOSD*, pages 90–100, Boston, MA, 17-21 March 2003. ACM Press.

[10] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In M. Aksit, editor, *Software Architectures and Component Technology.* Kluwer Academic Publishers, 2001.

[11] R. Pawlack, L. Seinturier, L. Duchien, and G. Florin. JAC: A flexible and efficient framework for AOP in Java. In *Proc. of Reflection'01*, Kyoto, Japon, 25-28 September 2001. Springer-Verlag.

[12] M. Pinto, L. Fuentes, M. Fayad, and J. M. Troya. Separation of coordination in a dynamic aspect-oriented framework. In *Proc. of the First International Conference on AOSD*, pages 134–140, Enschede, The Netherlands, 22-26 April 2002. ACM Press.

[13] M. Pinto, L. Fuentes, and J. M. Troya. A dynamic component and aspect platform. *The Computer Journal*, Accepted for Publication.

[14] M. Pinto, L. Fuentes, and J. M. Troya. DAOP-ADL: An architecture description language for dynamic component and aspect-based development. In *Proc. of the Second International Conference on GPCE*, pages 118–137, Erfurt, Germany, 22-25 September 2003. Springer-Verlag.

[15] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *Proc. of the First International Conference on AOSD*, pages 141–147, Enschede, The Netherlands, 22-26 April 2002. ACM Press.

[16] W. Schult and P. Trger. Loom.net - an aspect weaving tool. In *Proc. of the ECOOP 2003 Workshop on Aspect-Oriented Programming*, Darmstadt, June 2003.

[17] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: An aspect- oriented approach tailored for component based software development. In *Proc. of the Second International conference on AOSD*, pages 21–29, Boston, MA, 17-21 March 2003. ACM Press.

[18] E. Truyen, B. Vanhaute, W. Joosen, P. Verbaeten, and B. N. Joergensen. Dynamic and selective combination of extensions in component-based applications. In *Proc. of the 23rd International Conference on Software Engineering*, pages 233–242, Toronto, Canada, 12-19 May 2001. IEEE Computer Society.