Desarrollo de Software Orientado a Aspectos, DSOA 2006
Asociado a XV Jornadas de Ingeniería del Software y Bases de Datos
J. Araújo, J. Hernández, E. Navarro y M. Pinto (Eds)
Sitges (Barcelona), Octubre - 2006

AOP AND DYNAMIC COMPONENT ADAPTATION*

Javier Cámara[1]*, Carlos Canal[1], Javier Cubo[1] y Juan Manuel Murillo[2]

1: GISUM. Departamento De Lenguajes y Ciencias de la Computación
ETSI Informática
Universidad de Málaga
Campus de Teatinos 29071 – Málaga (España)
e-mail: {jcamara,canal,cubo}@lcc.uma.es, web: http://www.lcc.uma.es

2: Quercus Software Engineering Group. Departamento de Informática
Escuela Politécnica
Universidad de Extremadura
Avda. de la Universidad s/n 10071 - Cáceres (España)
e-mail: juanmamu@unex.es, web: http://quercusseg.unex.es

Keywords: Adaptation, CBSD, AOP, Framework, AspectJ

Abstract. *The use of Commercial-Off-The-Shelf components in the development of software systems is supposed to result in a reduction of development costs and time by promoting reusability. On the contrary, due to the Black-Box nature they exhibit, the use of COTS products may result in a burden for developers as continuous re-adaptation may be required along the software life-cycle or even at runtime if the nature of the system demands it. In this work we consider AOP techniques and mechanisms in order to alleviate this situation, proposing an AOP-based adaptation framework design.*

1. INTRODUCTION

When developing software systems, the use of Commercial-Off-The-Shelf (COTS) Components promotes reusability by incorporating the specific functionality they provide into larger systems. This is supposed to result in a reduction of both development costs and time. Although this would be the ideal scenario, the use of COTS has its drawbacks. Due to the Black-Box nature they exhibit, developers do not have any details about their

* This work has been supported by Spanish MCYT Project TIN2004-07943-C04-01.

implementation. This results in a lack of control over the component's functionality, performance and evolution. Since in most of the cases COTS products are not designed to interoperate with each other, they require a certain degree of adaptation which may have to be performed several times along the development process of the system, or repeatedly at runtime, if the nature of the system requires it (critical systems, pervasive computing, etc.). Software Adaptation [5], a field characterized by the modification and extension of application behaviour through highly dynamic runtime procedures, and the extensive use of software adaptors [20] (specific-purpose computational entities for the mediation and resolution of interoperability issues between components), can bring a new semi-automatic approach to these adaptation tasks. Interoperability issues between components can be classified in four different levels:

- Signature Level: Interface descriptions at this level specify the methods or services that an entity either offers or requires. These interfaces provide names, type of arguments and return values, or exception types. This kind of adaptation implies solving syntactical differences in method names, argument ordering and data conversion.

- Protocol Level: Interfaces at this level specify the protocol describing the interactive behaviour that a component follows, and also the behaviour that it expects from its environment. Indeed, mismatch may also occur at this protocol level, because of the ordering of exchanged messages and of blocking conditions. The kind of problems that can be addressed at this level is, for instance, compatibility of behaviour, that is, whether the components may deadlock or not when combined.

- Service Level: This level groups other sources of mismatch related with non-functional properties like temporal requirements, security, etc.

- Semantic Level: This level describes what the component actually does. Even if two components present perfectly matching signature interfaces, and also follow compatible protocols, the components must behave as expected.

Through the following sections, a description is given on how AOP [10] can provide an elegant and non-invasive way of "bridging the gap" between components by modifying and extending their behaviour without directly altering their code, as well as providing seamless message translation between components. In this paper, Section 2 discusses different approaches to adaptation through AOP. Although signature level is the state-of-the-art in adaptation (e.g. CORBA's IDL-based signature description), several proposals have been made in order to enhance component interfaces with a description of their concurrent behaviour [2, 4, 12], allowing automatic adaptor derivation in some circumstances [3]. Section 3 outlines the design of a dynamic adaptation management framework, and gives

some tips on the implementation of a prototype which makes use of AspectJ. Finally, section 4 presents some conclusions and open issues.

2. AOP AND COMPONENT ADAPTATION

Applying AOP to adaptation is not a new idea [1, 16]. If we consider for instance, [13], focused on the evolution of data models, we can observe that this work deals with the problems of structural and behavioural consistency arising after data model evolution. While structural consistency addresses the problem of accessing objects whose definition is no longer accessible after evolution, behavioural consistency refers to the problem of legacy applications having invalid references and method calls. The proposal of the authors is to encapsulate into aspects the adaptation code to access the evolved model, thus managing a more flexible result than those provided by approaches based on conventional class versioning. Other proposals, such as [9], deal with the adaptation of non-functional concerns. These concerns are given the shape of an aspect. The same idea is applied in [14], where it is shown how aspect oriented techniques can help adaptation in the context of pervasive computing environments. Again the idea is aspectizing those facets of the system which could be adapted. Similarly, [8] is focused on the Adaptive Object Model (AOM) architectural style, which supports adaptable systems not being adaptable itself. Using aspect oriented techniques the authors provide an adaptable AOM.
In [6], some suggestions to make joint points models more open are proposed, in order to provide aspect oriented programming languages with a better support for adaptation. In [15], the Iguana/J architecture and programming model to support unanticipated dynamic adaptation is presented. Each functional class is associated with a set of adaptation classes which contain the adaptation code. The association is also specified in separated entities achieving improved flexibility.
In the following sections, a new approach is outlined in which the main focus is put into using aspects for the implementation of the adaptation framework itself, rather than for aspectizing some facets of the system beforehand. Moreover, adaptation code is encapsulated into aspects, although not in a static way. On the contrary, aspects act as interpreters of the design information gathered from the components and as coordinators of the interaction between them. This adds a dynamic component to this approach which allows a degree of flexibility not available in previous proposals.

3. ADAPTATION MANAGEMENT FRAMEWORK

If we want to adapt components at runtime, we require their signature and protocol information in order to solve potential mismatches between them. The information available about the components in different runtime platforms often proves to be insufficient for dynamic adaptation, so we use techniques for the incorporation of metadata into components [7] to enrich interface descriptions. Specifically, we have used Java annotations [11], incorporating protocol related information, as well as required interfaces to obtain services from peer components. These annotations are readable at runtime through reflection. Having that information available, it will be used in the

production of a *mapping* or correspondence between component interfaces, which specifies the translation of method invocations between components. However, the construction of such mappings is not addressed in this paper.
Once all the correspondences between the interfaces have been established, these are used by the adaptation management framework which exploits the mechanisms provided by AOP for the translation of messages. This framework mainly performs three different tasks, which correspond to three functional modules or *managers*:

- Interface Manager: It gathers information about the components' interfaces.
- Adaptor Manager: Derives adaptors using the algorithm presented in [3] for the interaction between the components making use of the aforementioned *mappings*.
- Coordination Manager: Coordinates the interaction between components, translating the messages based on the description of the adaptors previously derived.

These tasks are implemented as aspects, separating coordination from concerns such as adaptor generation, or interface description management. In order to illustrate the proposal, we will make use of AspectJ, the most extended AOP tool currently available.

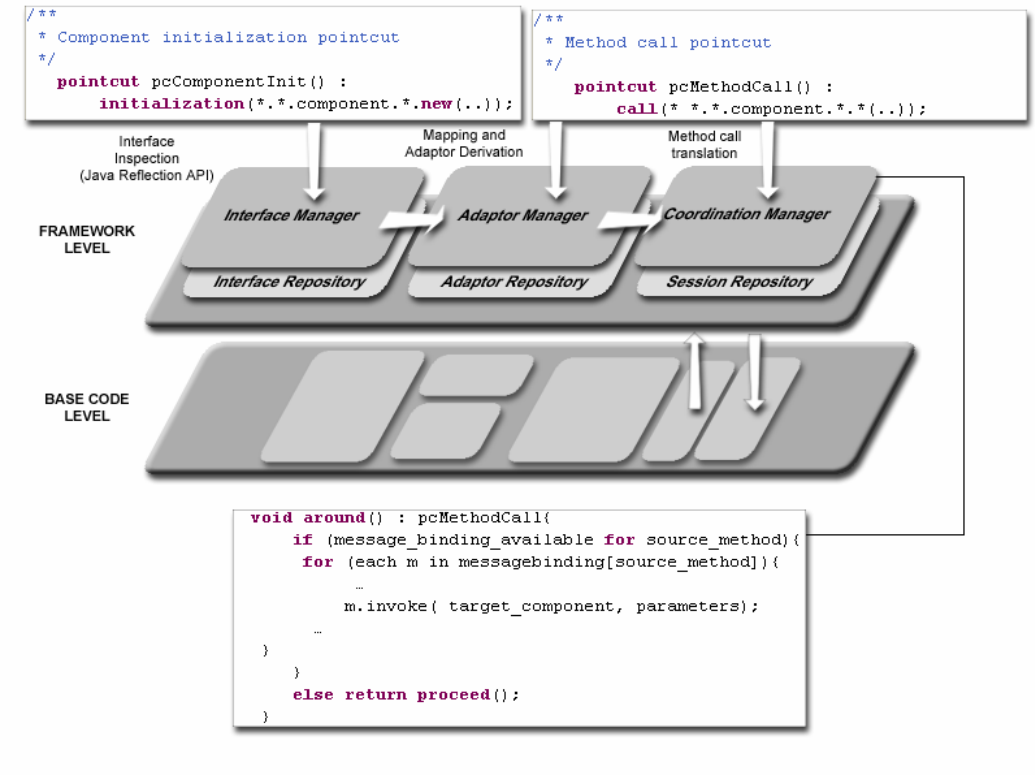

Figure 1. Framework architecture and key AspectJ definitions illustrating its operation.

3.1. Interface Manager

Stores interface descriptions in an interface repository in order to use them later for mapping generation. Upon initialization of the component *c* of class *C*, the manager checks for the existence of an entry for *C* in the repository, and if it does not exist, it creates one for it. Each one of these entries is a set of information containing method signatures (both offered and required, as well as protocol-related information. While the initialization of the component is acknowledged through the `Initialization` pointcut designator (see Figure 1), its interface is inspected making use of classes `Class` and `Method` of the Java Reflection API.

3.2. Adaptor Manager

Once a component of class *S* joins the context, it may perform one or several method invocations to other components. Every time one of these invocations is made, the manager captures it and checks if it is the first one consigned to a target component of class *T*. If that is the case, a *mapping* is produced between the source and target component classes. The manager can alternatively incorporate a *mapping* established manually beforehand by the developer, or produce one through an inference engine based on pre-agreed ontologies explicitly defining resources, preconditions, and effects of processes, as well as domain related properties and relationships [17]. This latter approach is required for automatic runtime adaptation (in pervasive computing environments, for instance), while the former can be used to support the process of software evolution. The incorporation of this inference engine is still an issue to be tackled, although of most importance, since it provides the system with a machine-interpretable description of the semantics of the components. This enables the use of techniques traditionally used in AI in order to infer relevant properties from the components and adapt them, establishing an appropriate correspondence between the interfaces.

Based on the *mapping*, an adaptor is automatically generated making use of the algorithm described in [7].This adaptor is stored in a repository and it will be used for interaction management between any pair of components of classes (*S*, *T*). Once generated, these adaptors will allow both syntactical and protocol adaptation, storing method invocations whenever required for a delayed delivery.

In order to monitor the invocations between components we make use of a second pointcut definition using the `call` pointcut designator, which is used both by this adaptor manager, and the coordination manager (Figure 1).

3.3. Coordination Manager

By Monitoring and translating method invocations between components, it performs the actual adaptation. Each time a component s_i sends a message to a component t_i, the manager translates it making use of the already available adaptor for (*S*, *T*) stored in the repository. A repository for session information is established in this manager in order to

store specific information about the state of the components and their interaction. For each pair of interacting components (s_i, t_i), a session is created in the repository the first time s_i sends a message to t_i. This session information is updated if necessary with each method invocation between components. Session information will be publicly available to the mechanisms in the coordination manager since the state of some interactions between components may influence that of others.
In order to perform method invocation translation, we apply an `around` advice to the method invocation, proceeding with it if there is no correspondence established for that method, or else substituting it with the call(s) specified in the *mapping* using the Java Reflection API (Figure 1).

3.4. Putting everything together

It is worth mentioning that since multiple aspects are present in the system, pieces of advice in the different aspects corresponding to each of the managers, may apply to a single join point. When this situation is given, the order in which advices are applied to the join point must be explicitly defined. This is the case of method invocation, which is used both by the adaptor and the coordination managers (see Figure 1). In order to observe this order, AspectJ uses precedence rules to determine the sequence in which advices are applied. Aspects with higher precedence execute their before advice on a join point before the ones with lower precedence. When the method of a component is invoked, the sequence to follow is: **(a)** the adaptor manager checks if an adaptor needs to be generated. **(b)** The coordination manager checks if a session entry must be created, and **(c)** the coordination manager translates the message and updates session information. This translation is driven by the previously generated mapping and implemented through the join point model provided by AOP. This provides an elegant and non-invasive way of performing message translation.

4. CONCLUSIONS AND OPEN ISSUES

In this paper, a proposal has been outlined for Aspect-Oriented Dynamic Component Adaptation. In order to test this approach, a prototype of the framework is being developed in AspectJ in order to measure the appropriateness of representative AOP facilities for component adaptation. Although the platform does not support dynamic weaving, it is capable of performing load-time weaving, which is enough in order to test the first stage of our approach. So far, only the signature and protocol levels have been tackled, and further study has to be performed related to mapping generation in order to provide suitable techniques for the semantic level as well (automatic mapping generation). The nature of this problem is not related to the AOP mechanisms which have been used to lay out the foundations of the adaptation framework. Thus, all this functionality will be packed into the inference engine since all the semantic level concerns are deeply interwoven with the process of mapping generation.

Although the current approach may suffice the requirements to perform dynamic adaptation in

simple cases, it is necessary to explore other alternatives in further work in order to scale up the problem to more complex scenarios. An interesting possibility to explore is the implementation of adaptors directly by means of aspects which are generated, applied and removed at runtime as required. This approach increases the complexity of the infrastructure required for execution, demanding some non-trivial modifications to it, such as the inclusion and integration of new functionality (specifically runtime aspect code generation and compilation –some platforms already perform runtime aspect (un)weaving [18, 19] –. This far more sophisticated approach will improve dramatically the flexibility of adaptation.

REFERENCES

[1] Aksit M.,Tekinerdo Gan B., Bergmans L. Achieving Adaptability through Separation and Composition of Concerns. Muhlhauser M., Ed., *Special Issues in Object-Oriented Programming*, dpunkt, 1996, p. 12–23.

[2] Allen R. and Garlan D. A formal basis for architectural connection. ACM Trans. on *Software Engineering and Methodology*, 6(3), 1997, p. 213–49.

[3] Bracciali, A., Brogi, A., Canal, C.: A formal approach to component adaptation. *The Journal of Systems and Software*. Special issue on Automated Component-Based Software Engineering 74, 2005, p. 45-54.

[4] Canal, C., Fuentes, L., Pimentel, E., Troya, J.M., Vallecillo, A.: Adding roles to CORBA objects. *IEEE Transactions on Software Engineering* 29 (2003), p. 242–260.

[5] Canal, C., Murillo, J.M. and Poizat, P. Software Adaptation. in L'objet, 12(1):9-31, 2006. Special Issue on Coordination and Adaptation Techniques for Software Entities. L'Objet, vol. 12, no. 1, Hermes, 2006, p. 9-31.

[6] Cazzola W., Pini S., Ancona M. Evolving Pointcut Definition to Get Software Evolution. Cazzola et al. in *Reflection, AOP, and Meta-Data for Software Evolution, report num. Research report C-196, 2004*, Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology, p. 83–90.

[7] Cazzola, W., Pini, S. and Ancona, M. The Role of Design Information in Software Evolution. In *Proceedings of the 2nd ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution* (RAM-SE'05).

[8] Dantas A., Borba P., Yoder J., Johnson R. Using Aspects to Make Adaptive Object-Models adaptable. Cazzola et al. in *Reflection, AOP, and Meta-Data for Software Evolution, report num. Research report C-196, 2004*, Dept. of Mathematical and Computing Sciences, Tokyo Institute of Technology , p. 9–20.

[9] David P.-C., Ledoux T. Towards a Framework for Self-Adaptative Component-Based Applications. *Distributed Applications and Interoperable Systems (DAIS)*, vol. 2893 of *Lecture Notes in Computer Science*, Springer, 2003, p. 1–14.

[10] Filman, Robert E., Friedman, Daniel P. Aspect-Oriented Programming Is Quantification and Obliviousness. In Mehmet Aksit, Siobhán Clarke, Tzilla Elrad, and Robert E. Filman, eds., *Aspect-Oriented Software Development*. Addison-Wesley, 2004.

[11] Islam Chisty, M. An Introduction to Java Annotations Developer.com's Gamelan, 2005
[12] Magee J., Kramer J., and Giannakopoulou D. Behaviour analysis of software architectures. *In Software Architecture, pages* 35-49. Kluwer, 1999.
[13] Rashid A., Sawyer, P., Pulvermueller, E. A Flexible Approach for Instance Adaptation during Class Versioning, *Objects and Databases*, vol. 1944 of *Lecture Notes in Computer Science*, Berlin, 2000, Springer, p. 101–113.
[14] Rashid A., Kortuem G. Adaptation as an Aspect in Pervasive Computing. *Workshop on Building Software for Pervasive Computing at OOPSLA*, 2004.
[15] Redmond B., Cahill V. Supporting Unanticipated Dynamic Adaptation of Application Behaviour. *Object-Oriented Programming (ECOOP)*, vol. 2374 of *Lecture Notes in Computer Science*, Springer, 2002, p. 205–230.
[16] Sánchez F., Hernández, J., Murillo, J. M., Pedraza E. Run-time adaptability of synchronization policies in concurrent object-oriented languages. *Workshop on Aspect Oriented Programming at ECOOP (AOP)*, June 1998.
[17] "OWL-S: Semantic Markup for Web Services", The OWL Services Coalition (2004), http://www.daml.org/services.
[18] Pinto, M.: CAM/DAOP: Component and Aspect Based Model and Platform, PhD thesis. Dpto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga (2004) Available in Spanish.
[19] Popovici, A., Frei, A., Alonso, G.: A proactive middleware platform for mobile computing. In: *4th ACM/IFIP/USENIX International Middleware Conference*, Rio de Janeiro, Brazil (2003).
[20] Yellin, D.M., Strom, R.E.: Protocol specification and component adaptors. *ACM Transactions on Programming Languages and Systems* 19(2), 1997.