

# Run-time coordination of components: design patterns vs. Component & aspect based platforms<sup>?</sup>

M. Pinto, M. Amor, L. Fuentes and J.M. Troya

Dpto. de Lenguajes y Ciencias de la Computación, Universidad de Málaga

Campus de Teatinos. E29071-Málaga (SPAIN)

e-mail: {pinto, pinilla, lff, troya}@lcc.uma.es

## 1. Introduction

Describing software architectures in terms of the interaction relationships between components brings us nearer to a compositional view. Although everybody agrees in that component oriented design represents a significant advance towards assembling systems by plugging off-the-shelf components (COTS), reducing application complexity, there is a lack of programming languages that support components and composition rules as language primitives.

As an alternative, component platforms (CPs) (e.g. CORBA, .NET, EJB, ...) provide a component abstraction and a composition mechanism, more or less dynamic, that support the coordination of distributed components. However, they do not care about the global configuration of the application, that is, the application architecture definition is not explicitly stated anywhere, but usually spread throughout component implementation modules.

Design patterns (DPs) and application frameworks (AFs) are currently widely accepted approaches to deal with this problem of an explicit definition absence of software architecture. DPs provide a customizable solution to a concrete design problem, as a set of few related classes that interact in a particular way [1][2]. The application of DPs concern the detailed design phase of the software life cycle, for instance in componentware DPs could be applicable to the internal design of a component.

On the other hand, AFs represent a step forward to a explicit description of software architectures, and are defined as a reusable design of all or part of a system, encapsulating a reusable, customizable software architecture as a collection of collaborating, extensible components [3]. AFs can be defined to aid in different levels of the software development process. At the middleware level, component-based AFs implement sophisticated component models and usually live on top of existing CPs. High-level composition models are really needed at the moment in order to endow both, component platforms with architectural abstractions and object-oriented languages with the assembly capabilities.

It is commonly agreed that the separation of concerns principle is a good approach to support adaptability and architectural evolution, representing one step further in the component-based software engineering (CBSE). This principle enforces the separation of different aspects of a system, that do not necessarily align with the functionality of components, such as synchronization, coordination, security, failure handling, ...

The preceding work of our current approach is MultiTEL [4], that offers a middleware framework that defines a new component model, that separates the coordination and computation issues, in two different entities, the *Connector* and the *Component*. The connection between components and connectors is dynamically established in MultiTEL at runtime, providing a

---

<sup>?</sup> This research was funded in part by the CICYT under grant TIC99-1083-C02-01, and also by the telecommunication organization “*Fundación Retevisión*”.

powerful mechanism for late binding among them. For instance, a change in a coordination protocol encapsulated inside connectors, can be simply achieved by replacing the appropriate connector, without modifying the computation involved.

MultiTEL is a working application<sup>1</sup> that also defines a component AF for the developing of multimedia applications for the Web. We have learned a lot of our experience in applying CBSE and Aspect-Oriented Programming (AOP) technologies to real world applications like these. First of all, the profits were the foreseeable, related with modularity, reusability, extensibility and scalability of MultiTEL derived applications, and at no so high overload cost. So, currently we are extending this model in order to encompass any other kind of aspects taken off from the Collaborative Virtual Environments (CVEs) domain (a new challenging research project). And secondly, we concluded that the separation of computation and coordination, or any other concern, should only be done when it is necessary and not should be mandatory as in MultiTEL.

Our current goal, is to define a new component and aspect platform, that support the dynamic attachment of any sort of aspects to components, performed according to some architectural information enclosed inside a new middleware platform [5]. In this paper we are going to focus on the coordination aspect, specially in how our approach provides applications with the facilities to resolve all kind of architectural mismatches arisen from the dynamic composition of components and aspects.

Our aim is to confine all the design solutions that face the dynamic composition problems inside a single middleware object. Likewise, there are plenty of design patterns that already provide solutions for solving these kinds of problems. For instance, the Abstract Factory [1] pattern's goal is to improve the evolution of a specific part of the system architecture, and the Mediator [1] pattern's goal is to decouple components reducing the number of interconnections. Nevertheless, believing that both approaches, middleware platform or design patterns, are at the same level is really wrong. But surprisingly, lots of people actually consider that defining a new coordination model is not worthy because DPs already give solutions for everything, but we do not agree. Although, the pattern approach provides a system of patterns capable of describing and documenting large-scale applications [6], how to apply DPs relies on users, so different users may produce different resulting systems, some of them good, some of them bad.

High level component models implemented as middleware component platforms, as the one proposed in this paper, should provide architectural abstractions and composition mechanisms, being the suitable site where to solve dynamic composition problems and architecture evolution. Applying DPs instead, has some drawbacks that we are going to show in this paper.

First and foremost, DPs are a solution driven by the user, where this user must have knowledge for each problem about the appropriate design pattern that is often not very well documented and sometimes difficult to understand and use. Most solutions to complex systems require the combination of several DPs, but DPs catalogues normally document patterns individually. Furthermore, sometimes the object or component that acts as a client of a DP-based component should be aware of this, that is, client components implementations must include specific code to instantiate adequately target components designed following a DP, making heavier the architectural evolution problem in the client site. Some glue strategies proposed as DP, such as the adaptor, mediator, bridge,... patterns have to be applied every time they are needed, overloading and complicating the resulting architecture with mediator classes. DPs make available really good design solutions of the most recurrent problems when you program in an object-oriented language (Visual C++, Java, ...), but middleware solutions are needed anyway and preferable to DPs.

By this way, we are going to show how a component and aspect based platform give answers to all these dynamic composition problems, providing also a programming interface to set the

---

<sup>1</sup> Web site: <http://www.lcc.uma.es/~lff/MultiTEL/>

architectural information of an application, making the application architecture explicit in the source code. The architecture consists of a set of components, a set of aspects and a set of static or dynamic links that may be established among them during application execution. Thus, in the current approach the coordination aspect will only be applied provided that the architectural information contains a link between a component and a concrete coordination aspect.

## 2. Our proposal: An Aspect-Oriented Middleware Framework

The purpose of our work is the construction of CVEs derived from an Aspect-Oriented Framework (AOF), where components and aspects are developed in the same general-purpose language and are composed dynamically at runtime through a middleware layer. The main goals of middleware component frameworks are making explicit application architecture definition, loose coupling between components and the possibility of COTS integration. In our approach, that defines two different entities, components and aspects, this implies: 1) components and aspects are first order entities that must exist at runtime, 2) the model detaches components and aspects interfaces from the final implementation classes, 3) the components and aspects implementation might be modified at runtime without client code recompilation and without changing the abstract definition of the software architecture 4) the components do not have direct references among them, 5) the components have no knowledge about the aspects they are affected by, 6) the number and type of aspects that are applicable to a component might change dynamically, 7) components involved in a collaboration do not need to “know” each other, 8) components interfaces can be adapted according to the necessities derived from the collaboration with COTS.

In some situations, our approach does not trail exactly an existing pattern but achieve the same benefits avoiding their liabilities. In this paper we are going to focus on the *VirtualEnvironmentSite* (*VESite*) component within the middleware layer and on the *Coordination* aspect of the framework. The *VESite* component represents a user in the environment and stores the architecture of the system, maintaining all needed information to dynamically compose components and aspects at runtime. The *VESite* component behavior is optionally complemented with the *Coordination* aspect that allows the composition among components, which do not have any knowledge among them or which interfaces do not conform to each other.

### 2.1 The *VirtualEnvironment* Component

The *VESite* component determines the *system's configuration* and performs the composition between components and aspects at runtime. This component maintains the architecture of the application defined in terms of the components and aspects that can be instantiated, and the architectural constraints for composing them. Also it maintains the *application context*, that consists of the references of all the components and aspects instantiated at each moment for a user. This context will change dynamically as the components and aspects instances are created or destroyed during the execution of the application. The design guides followed to implement this component promote the following desired properties in dynamic systems:

1. *Independence between components and aspects*: In our proposal, *components* and *aspects* are independent first order entities that are weaved at runtime as said by the architectural information stored in the *VESite*. In addition, components have no knowledge about the aspects they are affected by and the number and type of aspects that are applicable to a component can change dynamically. This means that when a source component sends a message to a target component, the middleware layer intercepts the message and applies the matching output and input aspects that the architecture definition of the *VESite* states. Consequently, developers implement components without having in mind input and output

aspects and vice versa. Thus in this approach the task of the software architect is to register aspects, components and their composition constraints in the *VESite*. For each component and each method in a component, the *VESite* stores the information about the sort of aspects to apply (e.g. authentication, multiple views, ...) and the order in which they have to be applied. In addition, it differentiates between *entry* aspects (input) that are applied before the execution of a method, and *exit* aspects (output) that are applied after execution. Regarding design patterns, the Mediator and the Adapter [1] pattern's main goal is to keep components from referring to each other explicitly, encapsulating a cooperative behavior in an intermediary class. As well as the adapter is used to redefine a method name, a sort of coordination aspect may state that after intercepting some message it should be resend with a different name. Also, the Interception pattern [7] tries to solve the same problem. The difference between both approaches is that aspects can be seen as additional behaviors that are attached before or after the execution of a component method, but the DPs mentioned above must be explicitly added by the user for each component of the system, which is so tedious.

2. *Components and aspects interface and implementation separation*: Each component and aspect instantiated in a service is registered in the *VESite*, providing its identifier, an interface and an implementation. Components and aspects in the framework must conform a predefined interface, but developers can provide different implementations, and even change the implementation class at runtime. During application execution the *VESite* component provides an interface for creating components and aspects providing only the identifier, hiding implementation details to client code. The goal of the Abstract Factory pattern is similar, but when the user instantiates a concrete factory, it determines the implementation objects, that cannot be changed dynamically. In order to obtain this functionality the Abstract Factory pattern must be combined with the Bridge [1] pattern, that decouples an interface from its implementation that can be selected or switched at runtime. Another disadvantage of applying the Abstract Factory pattern is that it is not so easy to add new products to the factory because the interface fixes the set of products that can be created. In our approach this is not a problem because the methods to create components and aspects are unique and the different components or aspects implementations that must be instantiated are indicated by its identifier as a parameter.
3. *Loose coupling between components*: In our framework, components do not have direct references among them. They have a reference (for example named *vesite*) to the local *VESite* and all the communication between components is performed through this *VESite* component. When a component identified as *c1* wants to send the message *message(args)* to another component named *c2*, it executes *vesite.execute(c1,c2,message,{args})*. The *VESite* checks within its *context application* which component instance plays the *c2* architectural role and it invokes the *message(args)* method using reflexive programming. The Mediator pattern provides the same functionality defining an object that encapsulates how a set of objects interact and decoupling them by keeping objects from referring to each other explicitly. The difference is that using DPs it is necessary to apply the pattern several times implementing a Mediator object for each set of collaborating components, while using our approach the middleware layer controls the interactions of all framework's components. In addition, the user that instantiates an application does not need to know how the middleware layer is designed and implemented, he/she just registers the components and aspects in the *VESite* component. Nevertheless, both the framework and the middleware layer implementations follow some well-known design patterns or a combination of them.

## 2.2 The Coordination Aspect

The *Coordination* aspect is one of the aspects that can be applied in our framework. The other ones are related with the CVE application domain such as *awareness*, *persistence*, *authentication* or *multiple views* aspects. However, the *Coordination* aspect is singular because it complements the *VESite* composition model with richer and flexible composition rules. In addition, this aspect provides the separation of concerns between *computation* and *coordination* as MultiTEL does. The main difference with MultiTEL approach is that MultiTEL forces to make the separation of *computation* and *coordination* and in this new proposal this separation is optional. The reason is that we have found that this separation is not always necessary and complicates unnecessarily the implementation in some cases.

When an application is developed, the *Coordination* aspect is applied to components when at least one of the following conditions is necessary:

1. *Components involved in a collaboration do not need to know each other*: In our approach we split this design solution in two different objects, the *VESite* and the *Coordination* aspect. The *Coordination* aspect can encapsulate collaboration protocols among components where a component do not know which other components participate in the collaboration. This *Coordination* aspect functionality complements the *VESite* functionality. While the latter avoid direct references between components identifying them through a name, the former allow that the components do not have any kind of references. They just send a message and the *Coordination* aspect will decide which component or components have to receive it. Again, the Mediator pattern provides a design solution. However, it is necessary to have a deep knowledge of the problem to implement the pattern functionality efficiently, especially when the pattern implementation guidelines do not align with our necessities.
2. *Use of COTS*: Components interfaces may be adapted according to the further functionality resultant from the collaboration with COTS. This adaptation may imply some changes in the method names, the number and order of parameters and type of parameters. Another important applicability of this aspect is trying to solve one of the liabilities of using DPs. Often, when a component is implemented following a DP, the client code must be aware of the component implementation details. For instance, using the Role Object [8] pattern, when a client component needs to use a specific component role it has to interact with the role protocol specified by this pattern, to check if the target component can play that role, add the role and get a reference to it. This decreases a lot the reusability of the framework's components. In order to avoid it, the *Coordination* aspect can be applied to any component implemented following the Role Object pattern. In this case, the *Coordination* aspect is in charge of dynamically adding a role when it is necessary and managing the different role references, transparently to the client code. The Adapter pattern design is suitable here, because it converts the interface of a class into another interface allowing components with incompatible interfaces to work together. But plugging COTS transparently into a partially existing solution is not a straightforward task and the DPs fail providing a solution because it is necessary to apply and implement the same design pattern over and over, for each component.

## 3. Conclusions

We have seen that combining the Abstract Factory, the Adapter, the Bridge and the Mediator patterns it is possible to implement a dynamic environment. The main drawback is that developers need to understand the patterns very well to be able to combine them successfully, taking into consideration all the domain specific requirements. Sometimes, only part of the functionality of a

DP is required and not always the implementation guidelines that documents the pattern fit our necessities. This makes very difficult the implementation of a reusable, extensible and adaptable solution using just patterns.

By the other hand, we have shown that a middleware framework is a better approach than just DPs because it applies the same recurrent design solutions than patterns but providing a built-in single mechanism to do it. Furthermore, the middleware layer provides the software architecture abstraction, that lacks in CPs. The development of dynamic and distributed applications as CVEs is lessened to the implementation of some new components and aspects, setting the architectural constraints, putting the desired implementation class names (or CORBA IDLs, ...) and letting to change implementation class names dynamically by components, aspects or even the final user under controlled conditions.

## 4. References

- [1] E. Gamma, R. Helm, R. Johnson, J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.
- [2] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerland, M. Stal, "Pattern-Oriented Software Architecture: A System of Patterns", Wiley&Sons, 1996.
- [3] M. Fayad, D. Schmidt, "Object-Oriented Application Frameworks", Communication of the ACM, 40 (10), October, 1997.
- [4] L. Fuentes, J.M. Troya, "Coordinating Distributed Components on the Web: an Integrated Development Environment", Software Practice and Experience, 31, March 2001.
- [5] M. Pinto, M. Amor, L. Fuentes, J.M. Troya, "Collaborative Virtual Environment Development: An Aspect-Oriented Approach", Next Publication in Proceedings of DDMA'01, 2001.
- [6] F. Buschmann, et al. "Pattern-Oriented Software Architecture - A System of Patterns", Wiley and Sons Ltd., 1996.
- [7] D.C. Schmidt, M. Stal, H. Rohnert, F. Buschmann, "Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects", Wiley&Sons, 2000
- [8] D. Baumer, D. Riehle, W. Siberski, M. Wulf, "Role Object", In Patterns Languages of Program Design 4. Addison-Wesley, 2000.