Aspect Component Based Software Engineering[1]

Pedro J. Clemente and Juan Hernández
Quercus Software Engineering Group. http://quercusseg.unex.es
University of Extremadura. Spain

{jclemente, juanher}@unex.es

ABSTRACT

Component Based Software Engineering (CBSE) and Aspect Oriented Programming (AOP) are two disciplines of software engineering, which have been generating a great deal of interest in recent years. From the CBSE point of view, the building of applications becomes a process of assembling independent and reusable software modules called components. However, the necessary dependencies description among components and its latter implementation causes the appearance of crosscutting, a problem that Aspect Orientation resolves effectively. Aspect Oriented Programming allows programmers to express in a separate form the different aspects that intervene in an application which are composed adequately at a later stage. This paper analyses the problem of *crosscutting* which is produced during component development, and a component based development extension using Aspect Oriented techniques is proposed. This Component based Software Engineering (CBSE) extension has been named Aspect Component Software Engineering (ACBSE).

Keywords

Aspect-Oriented Software Development (AOSD), Component Based Software Engineering (CBSE)

1. INTRODUCTION

Component-Based Development is gaining recognition as the key technology for the construction of high-quality, evolvable, large software systems in timely and affordable manners. Constructing an application under this new setting involves the assembly/composition of prefabricated, reusable and independent pieces of software called components. A component should be able to be developed, acquired and incorporated into the system and composed with other components independently in time and space[1].

The ultimate goal, once again, is to be able to reduce developing costs and efforts, while improving the flexibility, reliability, and reusability of the final application due to the (re)use of software components already tested and validated. Component Oriented Programming aims at producing software components for a component market and for later composition (composers are third parties). This requires standards to allow independently created components to interoperate, and specifications that put the composer into the position to decide what can be composed under which conditions[1]. This approach moves organizations from application development to application assembly.

This situation has given birth to the existing commercial component models and platforms such as CCM, EJB or DCOM. However, most of the publicity surrounding these component models and platforms is oriented towards gaining the race way under between middleware architects and vendors to establish their products as standards for developing open distributed systems. Thus, whilst companies are focused on highlighting the benefits of software developing using the plug and play mechanism of their products, there is little or no discussion in the media of how to really design reusable, flexible and adaptable components.

In this sense, there are reasons in Component Based System (CBS) which cause a lack of reusability and adaptability: CBSE imposes a structure on the programs that makes it difficult to have different concerns well-modularized: code-tangling is inherent to CBSE programs[2;3]. The *uses statement* (like CCM statement) during component specification may be considered harmful. The purpose of these statements is on the specification of receptacles (i.e., a component reference in order to use the operations it provides). However, these references express an aggregation relation between components, thus establishing strong (and hard-coded) dependencies among components which make them difficult (or even impossible) to reuse, adapt and evolve.

Aspect-Oriented Software Development (AOSD) is an emerging technology that provides direct support for separating and weaving concerns that crosscut the functional components in a typical software system. Aspect Oriented Programming (AOP) has been created with the objective of allowing programmers to express separately the different concerns of an application, in order to be composed adequately at a later stage. The main characteristics of software developed with AOP are flexibility, adaptability and reusability of the elements used to compose the system [4]. AOP provides various mechanisms to express, adapt, isolate and reuse *crosscutting concerns* in the software development to obtain these main characteristics.

This paper focuses on the study of the current problems of Component Based Systems. On the one hand, each phase of CBD; that is specification, implementation, package, assembly and deployment, is revised. On the other hand, an Aspect Oriented Programming methodology to develop Component Based Systems business rules is proposed. The final software systems are composed using Aspect Oriented techniques as a “glue” among components, giving that the main advantages provided by aspect orientation to component-based systems.

The rest of the paper is as follows: in section 2, the problems arising during Component-based development will are identified. In section 3, our proposal is presented. Finally, we feature a set of conclusions.

[1] This project has been financed by CICYT, project number TIC02-04309-C02-01.

2. CURRENT CBS PROBLEMS

Currently, common component platforms like CORBA Component Model (CCM)[5] or Enterprise Java Beans (EJB)[6] are based in the idea of D'Souza in Catalysis[7]. This idea is simple: to build software systems using modules (components) like a builder builds a house, using independent modules. Each module has a specification and an implementation, and then, each is composed to build the final software. For this objective, the interfaces which a component provides and requires are used like the connectors in a "*lego piece*".

In this context, building applications are based on a process to compose/assemble *plug&play* components. Therefore, building an application requires the following phases: on the one hand, the description and implementation of *plug&play* components is needed (*specification and implementation CBD phases*). On the other hand, a process to interconnect and deploy components is required (*package, assembly and deployment CBD phases*). Initially, CBD methodology increases the quality of software by providing flexibility, adaptability and reusability through the assembly/composition of independent software components.

However, individual components are not as reusable and adaptable as may appear in a first place because the *crosscutting* phenomenon arises in a actual way, as we explain in the following paragraphs.

2.1 CBS Adaptability and Reusability

From the *adaptability* point of view, a system must be adapted when new requirements appear. This means that changes in the business rules can be applied to systems already built with minimal changes. Let us proceed to analyze how we can adapt the functionality of CBS to new requirements. First, a business rule is a process in a software system; therefore, business rule changes introduce software systems changes. These system changes can include ones in *functional* or *non-functional* properties.

Updating Non-functional properties. Common component platforms offer a container to manipulate the *non functional* system properties like security, persistence, distribution, etc. The container properties facilitate the development of components, because the containers offer common services for all components[5;6]. The container configuration can be changed during the *package phase.* During this phase the developer can specify various kind of policies for each component. For example, if we are developing a system using CCM, the security, transactions, and persistence for each component can be configured in *Package Phase*[5].

Updating Functional properties. The container can not be used to change a functional system behavior. CBS functional behaviors are specified by business rules which describe the interconnection among components. This interconnection among components will form the final system. Besides, business rules change for each system, for each domain, etc. Business rules respond to the specific problem to be solved, and they establish the specification and interconnection components in the design phase [8].

When a component A declares that uses the services offered by other component B, that declaration affects both the specification and implementation of A. This is due to the fact that the *uses statement* expresses an aggregation relation between A and B. Furthermore, in the implementation of A, direct calls to B methods appear and they are hard-coded. Consequently, changes in business rules involve updating both the specification and implementation of software components. In addition, specification changes affect to package, assembly and deployment phases of CBD.

For example, let us assume that we have developed a CBS controlling a market. Then, a new business rule for clients' authentication is required in our system. This rule describes an aspect called the *Authentication Aspect.* Suppose that the *Authentication Aspect* is implemented in a specific component (Authentication Component). In order to introduce this new aspect (it is implemented by Authentication Component) in the system we have to perform the following steps:

- First, to identify the components affected by the Authentication Aspect.
- Second, the specifications and implementations of the identified components should be updated, as it has been shown above.
- Finally, the software architecture of the system should be updated; that is to say, package, assembly and deployment phases must be reviewed.

This means that component systems are not easily adaptable to new requirements, because the introductions of new requirements involve changes in the all phases of the component life cycle, namely specification, implementation, package, assembly and deployment.

This situation happens frequently because business rules evolve and, continuously the software systems need to implement new business rules. Consequently, new mechanisms for developing more adaptable and reusable component based systems are needed, and this is the main contribution of this paper, which it is explained below.

3. ACBSE: BUILDING CBS USING AOP

In this section a new component based development methodology is presented. This methodology combines the principles of Component Based Software Engineering and the flexibility, adaptability and reusability characteristic are provided by Aspect Orientation. This methodology is called ACBSE (Aspect Component Based Software Engineering).

In the following paragraphs we are going to express the changes necessary to apply aspect oriented programming in each component based system development phase (design and specification, implementation, package, assembly and deployment).

3.1 System Design and Components Specification Phases

During the component-based system specification, the interfaces that a component provides and requires must be described. For example, in the specification of a CCM[5] component, the interfaces that it provides (facets) and those that it requires (receptacles) are described. We will focus our attention on the dependencies introduced by the *uses clause*, that is to say, the interfaces it requires from other components. There are two alternatives to define the dependencies between components: doing it during the specification phase, or leaving it for subsequent phases.

Both approaches have their advantages and disadvantages:

- If the dependencies of a component are described during the specification, they belong to that component and, therefore, they must be maintained by all implementations of that component specification [8]. With this alternative, the component provides a clear and concise idea of its behavior. However, the use of this component is quite limited. For example it is possible that in a specific framework the handling of some of the dependencies is unnecessary or, even worse, the introduction of new dependencies becomes a difficult task.
- If the dependencies between components are not represented during the specification phase. As an advantage, the components can be easily adapted to the requirements of each context. However, CBD phases concerns system architecture (package, assembly and assembly) should be reviewed to apply the component dependencies.

We propose a new way to specify component dependencies avoiding the crosscutting at the implementation phase. The different implementations of a component specification do not need to implement those dependencies.

Component dependencies are classified as *intrinsic* and *non-intrinsic*[9], since not all the dependencies have the same degree of dependency regarding to the component:

- A *non-intrinsic* dependency is when its use depends on the framework or the context in which a component is to be used. That is, if we delete a dependency from the component description, the component maintains its initial functionality without those facilities that provide the deleted dependency.
- An *intrinsic* dependency is when its description and use is vital for the component itself. In others words, if this dependency is deleted then the component loses its meaning.

3.2 System Implementation Phase

The implementation phase allows for implementing the component functionality. The component developer should use only the *intrinsic* dependencies, and in the component implementation there are many calls to methods to *intrinsic* interfaces. This means that each component only implements the basic business rules.

Throughout the implementation phase of the components, each component implements the interfaces it provides, as well as all the methods needed to carry out its functionality. During the implementation of these methods, dependencies can be used in the component implementation but it can only use *intrinsic* component dependencies. However, all those dependencies defined as *non intrinsic* dependencies are applied throughout the package phase of software components. Therefore, *crosscutting* is not being introduced in the implementation of the component due to *non-intrinsic* dependencies.

3.3 System Package Phase

During the *Package Phase*, XML descriptors (for example, Component Descriptor in CCM) are used to describe the component properties which form a part of the component system. In this XML description each component identifies the interconnections with other components; that is to say, the system architecture is described through the connection among interfaces which are provided or required by components.

The *Package Phase* allows us to apply the *non intrinsic* dependencies on the new component based system. The steps are the following:

- First, the *non intrinsic* dependencies must be defined during *System Design Phase*[9] using a graphic representation like UML.
- Second, the dependencies which have been described in UML are translated to XML Component Descriptor Specification.
- Third, this XML Component Descriptor Specification describes the *non-intrinsic* dependencies for each component. This means identifying the new business rules or new dependencies in new contexts or new domains.
- Finally, the information that a specific component describes in its XML Component descriptor is pre-processed in order to recompile the component code, and add the restrictions and dependencies that are specified in this XML Component Descriptor. These dependencies are expressed as aspect implementations through a generic aspect-oriented programming language, such as AspectJ[10], AspectC++[11], etc.

Why is the packaging phase the most suitable one to describe when and where the new business rules or dependencies should be applied? One of the principal advantages of our proposal is the flexibility that is obtained in the component design, precisely due to the fact that the dependencies with other components are not expressed in the component implementation, but indeed are once the component has been implemented according to the necessities of the context.

3.4 System Assembly and Deployment Phase

At this moment of the lifecycle of the CBD we have defined the component interfaces, the *intrinsic* and *non intrinsic* dependencies and the mode to interconnect the components to obtain the final system (previous section). However, given that we are designing component-based systems, which are possibly distributed, we must also represent the location of every one of the components that form the final system.

During the packaging phase a description of when and how the dependencies that are defined in the specification phase must be applied is provided. However, the location of the component that implements or provides an interface is still not specified. The UML diagrams for assembly and deployment of component-based systems[9] are translated to XML Assembly Descriptor.

An XML Assembly Descriptor (similar to CCM Assembly Descriptor) permits us to connect the component implementations, which are part of our application. It is made up of a set of component descriptors and property descriptors. That is to say, it allows for the description of components and component instances which make up a final application. This descriptor permits us to specify the location of the components or instances.

The objective of this XML descriptor is to generate a specific properties file. This file can be read for all components and the rest of the components along the net can be located. This file can use various types of locations: URL, IOR, NameService, etc.

3.5 ACBSE advantages

- **Reusability.** The component is not recoded when the components are used in other domains or contexts, because the component implementation can be adapted to new business rules by changing the *non intrinsic* dependencies. Then these components can be coupled with others components.
- **Adaptability.** Programmers are offered the possibility of modifying the component descriptor by altering the final component functionality.
- **Scalability.** The system can be easily scalable because we obtain new component implementations and new component specifications. Then these new components with their *intrinsic* and *non intrinsic* dependencies can be used to compose new systems.
- **Compressibility.** Developing a new system is based on following a set of structured phases (Design and Specification, Implementation, Package, Assembly and Deployment). All information about the system is stored by using the common schemas (UML or XML). Besides, the interconnection code is generated by XML translation.

4. CONCLUSIONS

In this paper we have presented a joined CBSE and AOP proposal in which two of the recent tendencies in software system development are united. We have expanded the life cycle of a component-based system through techniques of aspect-oriented programming with the aim of making good use of the advantages of both tendencies and obtaining more flexible, adaptable and reusable software systems.

In a component based system, the business rules establish and determine the components specification and their relations. However, these relations or dependencies provoke the appearance of *crosscutting* as we have seen in this paper.

Therefore we have detached every one of the stages in the component based development. Every one of these stages is expanded so that a new description model of dependencies between components, which are materialized during the system composition phase, is implanted.

These interconnection descriptions in XML permit us to save time and cost, due to the fact that almost the entire code necessary is generated automatically. Finally, it should be emphasized that currently the interconnection between components is totally transparent to the programmer.

This ACBSE methodology has been developed with success in the CORBA Component Model domain [12].

5. REFERENCES

[1] Szyperski, C., *Component Software: Beyond Object-Oriented Programming*, Addison-Wesley, 1998.

[2] Duclos, F., Estublier, J., and Morat, P., "Describing and using non functional aspects in component based applications," *Proceedings of the 1st international conference on Aspect-oriented software development* Enschede, The Netherlands: ACM Press, 2002, pp. 65-75.

[3] Lieberherr, K. J., Lorez, D., and Mezini, M. *Programming with Aspectual Components*. 1999. Technical Report, NU-CCS-99-01, Northeastern University.

[4] Kiczales, G. *Aspect-Oriented Programming*. 1997. Procedings of ECOOP, Springer Verang. LNCS 1241.

[5] Object Management Group (OMG). *Specification of Corba Component Model (CCM)*. 1999. Web Site: http://www.omg.org/cgi-bin/doc?orbos/99-07-01.

[6] Sun Microsystems, *Enterprise JavaBeans (EJB) Specification 2.1*, Web site: http://java.sun.com, 2003.

[7] D'Souza, D. *Objects, Components and Frameworks with UML*. 2000. Web site: http://http://www.trireme.u-net.com/catalysis/.

[8] Chessman, J. and Daniels, J., *UML Components: A Simple Process for Specifying Component-Based Software*, Addison-Wesley, 2001.

[9] Clemente, P. J., Sánchez, F., and Perez, M. A. *Modelling with UML Component-based and Aspect Oriented Programming Systems*. 8-6-2002. Seventh International Workshop on Component-Oriented Programming(WCOP 2002) at European Conference on Object Oriented Programming (ECOOP). Málaga, Spain. Web Site Download: http://www.research.microsoft.com/%7Ecszypers/events/wcop2002/.

[10] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. *An Overview of AspectJ*. 2001. Proceedings of ECOOP, Springer Verang. LNCS 2072.

[11] Gal, A., hroeder-Preikschat, W., and Spinczyk, O. *AspectC++: Language Proposal and Prototype Implementation*. 2001. OOPSLA. Web Site.http://www.cs.ubc.ca/\~kdvolder/Workshops/OOPSLA2001/submissions/17-gal.pdf.

[12] Clemente, P. J., Hernández, J., Murillo, J. M., Perez, M. A., and Sánchez, F. *AspectCCM: An aspect-oriented extension of the Corba Component Model*. 2002. EUROMICRO Conference. Euromicro Component Based Software Engineering Track. Dortmund, Germany, IEEE Press.