ASPECT IPM: TOWARDS AN INCREMENTAL PROCESS MODEL BASED ON AOP FOR COMPONENT-BASED SYSTEMS

Alexandre Alvaro, Eduardo Santana de Almeida , Silvio Romero de Lemos Meira
Federal University of Pernambuco and C.E.S.A.R – Recife Center for Advanced Studies and Systems

Daniel Lucrédio, Vinicius Cardoso Garcia, Antonio Francisco do Prado
Computing Department – Federal University of São Carlos, Brazil

Keywords: Incremental Process Model, Reuse, Components, Aspect-Oriented Programming.

Abstract: In spite of recent and constant researches in the Component-Based Development area, there is still a lack for patterns, processes and methodologies that effectively support either the development "for reuse" and "with reuse". This paper presents Aspect IPM, a process model that integrates the concepts of component-based software engineering, frameworks, patterns, non-functional requirements and aspect-oriented programming. This process model is divided in two activities: Domain Engineering and Component-Based Development. An aspect-oriented non-functional requirements framework was built to give support in these two activities.

1 INTRODUCTION

One of the most compelling reasons for adopting component-based approaches is the premise of reuse. The idea is to build software from existing components primarily by assembling and replacing interoperable parts. The implications for reduced development time and improved product quality make this approach very attractive (Krueger, 1992). In this sense, one of the most important approaches for achieving reuse is the Component-Based Development (CBD).

In order to make reuse effective, it must be considered in all phases of the software development process (Krueger, 1992), (Jacobson, et al., 1997), (Heineman, et al., 2001), (Szyperski, 2002). Therefore, CBD must offer methods and techniques that support different activities, from the components identification and specification to their design and implementation in a component-oriented language. Besides, CBD must use interrelations among components already in existence, which has been previously tested, aiming to reduce the complexity and the development costs.

Another important point to reach these benefits is the Non-Functional Requirements (NFRs) treatment. Unfortunately, NFRs many times are not considered in components development activities. There are some reasons that can help us to understand why these requirements are not explicitly dealt with (Rosa, 2001). In particular, developers find it very difficult to deal with different NFRs because they often compete with each other and with the functional requirements. When proper treatment is not given to NFRs, these conflicts may lead to several problems, such as low modularity and code interlacement.

To help to overcome the difficulties of treating different NFRs, Aspect-Oriented Programming (AOP) (Kiczales, et. al., 1997) may be a good choice. With AOP it is possible to create, for the functional requirements, a group of components expressed in a programming language (e.g. Java), and for the NFRs, a group of aspects, related to the properties that affect the application's behaviour. Using AOP, the NFRs can be easily manipulated, without causing impact in the business code, since these codes are not interlaced in several units of the system. Besides, AOP can facilitate the reuse of the NFRs, requiring little or no modification, since they are isolated from the rest of the code.

In order to aid the Software Engineer in the integration of the NFRs into design decisions during the software development process, a framework may be used to represent, organize and analyze NFRs

Alvaro A., Santana de Almeida E., Romero de Lemos Meira S., Lucrédio D., Cardoso Garcia V. and Francisco do Prado A. (2005).
ASPECT IPM: TOWARDS AN INCREMENTAL PROCESS MODEL BASED ON AOP FOR COMPONENT-BASED SYSTEMS.
In *Proceedings of the Seventh International Conference on Enterprise Information Systems*, pages 226-232
DOI: 10.5220/0002514302260232

(Mylopoulos, et. al., 1992), increasing quality and reducing time and costs.

In this context, motivated by ideas of reuse, CBD, NFRs and AOP, this work proposes and evaluates Aspect IPM, an Incremental Process Model based on AOP for Component-Based Systems.

2 ASPECT IPM

In order to enable CBD with aspect-oriented support for NFRs, an Incremental Process Model (Aspect IPM) was defined. An overview of this process model can be found in (Alvaro, et al., 2004).

The Aspect IPM is divided in two activities. In the first activity, **Domain Engineering (DE)**, the problem domain requirements are identified and organized in a series of reusable information. Software components are specified, designed – including the NFRs design, which were performed through aspect-oriented NFRs framework support – and tested, being then stored into a repository. In the next activity, **Component Based Development (CBD),** the software engineer may build applications that reuse these components, consulting the repository to find and reuse them. Still, the aspect-oriented NFRs framework aids the software engineer in the application development.

2.1 Domain Engineering (DE)

Domain Engineering (DE) has been one of the most used approaches to enable reuse-based development (Griss, et al., 1998). It involves the identification and development of reusable assets within an application or a domain (Griss, 1997).

The software development based on DE allows the production of applications through accumulated knowledge (i.e. activity of collecting, organizing, and storing past experience in building systems) in a particular domain, as well as providing an adequate means of reusing these assets (i.e. retrieval, dissemination, adaptation, assembly, and so on) when building new systems (Jacobson, et al., 1997).

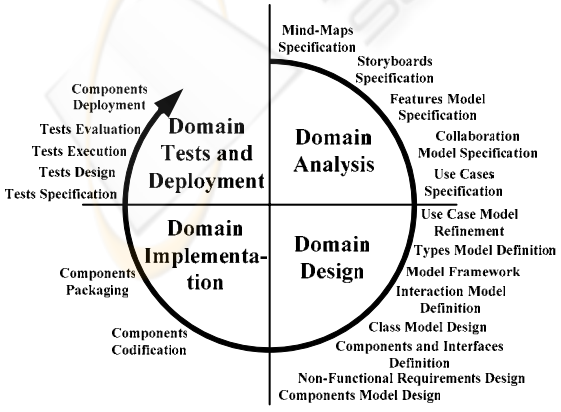

Figure 1: Domain Engineering Activity

In Aspect IPM, the Domain Engineering is performed in four phases: Domain Analysis, Domain Design, Domain Implementation and Domain Tests and Deployment, according to Figure 1. Each phase contains some steps that aid the Software Engineer in the DE. A detailed description of each phase of the Domain Engineering activity is presented next.

2.1.1 Domain Analysis

In this first phase, emphasis is placed on understanding the problem domain and specifying "*what*" the components must do to solve the problem. All the possible information, including any kind of textual specifications, such as informal requirements descriptions and interviews transcriptions, is identified and organized to become more reusable in new developments (Prieto-Diaz, 1990), creating a well defined reuse infra-structure that allows the specification and the implementation of applications inside a defined scope (Arango, 1988). As shown the Figure 1, some techniques and models are used in this step in order to modeling, in a high abstraction level, the domain problem.

2.1.2 Domain Design

In this second phase, the software engineer refines the specifications from the previous steps, aiming to obtain the components specifications, but without worrying with the implementation details. As shown the Figure 1, some techniques and models are used to improve the problem understanding, looking for a solution for the problem.

Until this moment, the components with their interfaces were specified and the components contain only specifications about the functional requirements. Next, the NFRs must be specified. To facilitate this task, an Aspect-Oriented NFRs Framework was constructed. Next, this framework is presented.

2.1.2.1 Aspect-Oriented Non-Functional Requirements Framework

Given the problems related to NFRs, described in Section 1, an Aspect-Oriented NFRs Framework was constructed for dealing with NFRs in the software development process, reducing time and costs, and helping to achieve the main quality

attributes wanted by the customer, such as reliability and performance.

Another motivation is the need to avoid the code interlacement between functional and non-functional requirements. To accomplish that, the Separation Of Concerns (SOC) (Kiczales, et. al., 1997) principles was applied, through AOP, in the framework development.

The constructed framework deals with six NFRs: exception handling, distribution, persistence, fault tolerance, caching and security. Although there are others important requirements that are not considered in this work, such as concurrence control and load balance, these six are quite common in most of the software systems, being therefore essential to a NFRs framework.

This framework is still under construction, and some packages are not fully implemented. More information can be found in (Alvaro, et. al., 2003).

To integrate the NFRs, the packages of the framework are added in an incremental way. First the functional requirements are designed. Once the component is functionally complete, if there are any NFRs to be added, each one is designed and added to the component, generating new component versions until it is complete, as shows Figure 2.

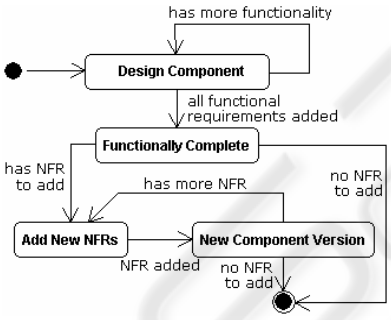

Figure 2: Adding NFRs.

Once all the components are designed, with the functional and NFRs defined, it is necessary to describe them, in order to provide a good basis for their reuse. The objective is to provide to the software engineer enough information so that the components are correctly reused. This includes some information, such as the name of the responsible engineers and developers, the key words to aid in component search, information related to the modifications occurred in the component, the NFRs integrated with the component, etc. All these information is stored together with the component in the repository. After all components are described and implemented, the domain may be then implemented in an executable language.

2.1.3 Domain Implementation

In this phase, the software engineer defines the programming language to perform the components codification, and the distribution technology – if required. The code related to the functional and non-functional requirements is separately constructed. Since Aspect IPM is based on AOP, it is easy to integrate these different-purpose codes.

Figure 3 shows an example of a component that implements distribution and exception handling NFRs, to deal with operations related to a generic customer. To implement the aspects, an aspect-oriented language, such as AspectJ (Eclipse Project, 2003), was used. According to Figure 3, the *Customer* component involves aspects that isolate the distribution technology and the exception treatment. Before, an important point to consider is the AOP notation presents in the Figure 3 (and in Figure 7). As not exists a well-define notation to adopt in literature (Clarke, et. al., 2002), we will represent aspects through UML stereotypes. However, our research group is defining an AOP notation for UML that will be aids the software engineer in this kind of representation.

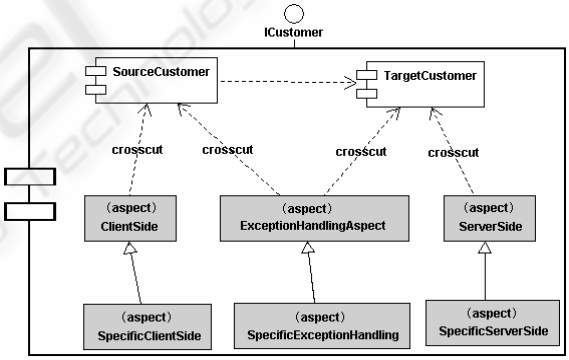

Figure 3: Customer Component

Using AOP it is possible to modularize the NFRs, keeping their code separated from the functional code. It is also possible to introduce changes without modifying the source code. For example, to change from CORBA to RMI, it is only necessary to develop a new aspect and integrate it with the functional code. This integration is automatically performed, through the process called *weaving* (Kiczales, et. al., 1997), shown in Figure 4.

The weaver is a compiler that, given the functional code (*SourceCustomer*) and a group of aspects (*Distribution* and *Exception Handling* aspects), a version of the component with these aspects is generated.

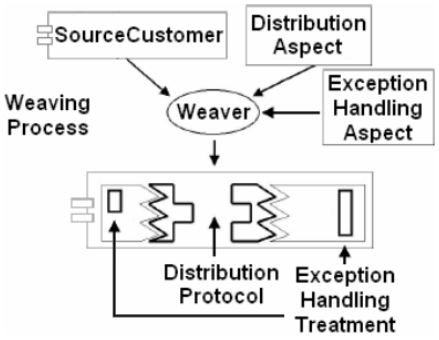

Figure 4: *Weaving* process

2.1.4 Domain Tests and Deployment

Testing is an important concern to assure software quality. However, in this case, we will take in consideration the fact that aspects affect the component behavior. In order to determine whether an aspect is hostile or not, there needs to be some concept of what constitutes correct behavior for a component. Any aspect that interferes with the correctness of an underlying component can then be deemed hostile.

To ensure correct behavior, a component needs to be checked conforms its pre- and post-conditions to establish the range of acceptable behavior. When an aspect intercept the component execution flow, the component needs to test its pre- and post-conditions before the message returns, looking for to ensure that the component behavior still holds.

In order to reach this objective, the component tests are performed at two levels: individual components with their aspects and consistency within a domain.

i. Within a domain, each component with their aspects is individually tested by building tests application around the component that allows the component's and aspect's functionalities to be exercised. Thus, whatever time that the aspect intercept or returns the execution flow to a component is verified the pre- and post-conditions in order to analyze if the component behavior was kept.

ii. Then, tests to verify the internal consistency between components of a domain are accomplished.

After the domain tests, the last step is to release the component to the repository. This task has to assure that the component is packaged in a form to be used on applications development in the future, together with all the information needed for reuse.

In this moment, the components are ready to be deployed in an execution environment. In the second activity of the Aspect IPM, the software engineer may develop applications, reusing the components that were built in this first activity.

2.2 Component-Based Development (CBD)

This second activity guides the software engineering during the construction of an application that reuses the components developed in the first activity. Figure 5 presents the phases for CBD activity. It starts with the application requirements and proceeds with the normal life cycle development. The application is constructed in an incremental way, with the functionalities being added during the iterations of the process. To implement these functionalities, the software engineer consults the repository, which contains the constructed components of the problem domain. Next, each phase is briefly discussed.

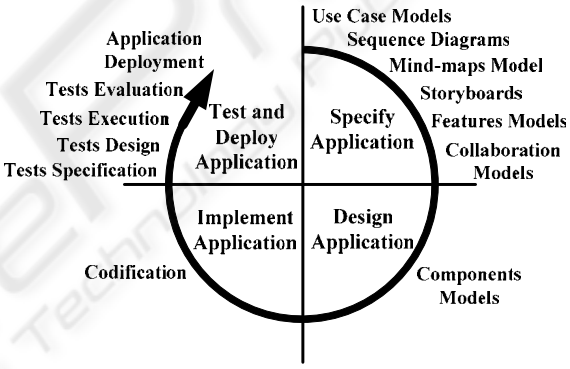

Figure 5: Application Development.

2.2.1 Specify Application

In this first phase, specifications are created in order to understand the problem and "*what*" should be made to solve the problem. Use case models, sequence diagrams and informal textual descriptions are created to understand the application requirements.

To aid in this task, the domain models that were built in the first activity of Aspect IPM can be reused. Mind maps, features model, storyboards, collaboration models and use case models contain much of the knowledge related to the domain, and constitute an important information basis for the software engineer. These models may be reused and modified to help in specifying the application.

Once the application is specified, the software engineer moves to the next step, where the design is performed.

2.2.2 Design Application

In this second phase, emphasis is put on "*how*" to solve the problem that was defined in the previous phase. Here, the components of the domain can be reused to perform the design. Figure 6 shows how the components are integrated into the design.

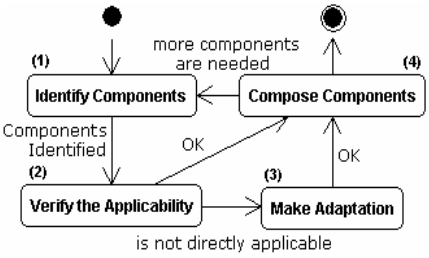

Figure 6: Analyse the appropriated components.

In the first step **(1)**, the software engineer searches for available components that are likely to be used in the application being designed. Several techniques can be used to aid in the search for components, ranging from mechanisms based on key words or introspection, to more complex mechanisms based on *ontologies* (Braga, 2000) and software agents (Ye, et. al., 2002).

In the second step **(2)**, the software engineer verifies the applicability of the components, since not all the components recovered in the first step can be applied to the situation. Consequently, they have to be analyzed to allow an effective reutilization. If the component is directly applicable, it goes to the last step **(4)**, without any adaptation.

Normally the components stored in the repository require some adaptations or customizations **(3)**. In this step, the software engineer identifies and performs the necessary changes in the components.

The last step **(4)** consists in the components composition. After the first three steps are accomplished, the software engineer defines the dependencies and establishes the inter-relationship between the components. If needed, the software engineer may return to the first step to identify more components to compose the application.

After the existing components are identified and composed, the software engineer completes the design with new application-specific components. As happens in the first activity of Aspect IPM, the NFR Framework may be used to design the NFRs.

The components model of Figure 7 shows the architecture of an application to maintain a record of customers. The reused elements are shaded, and new elements are represented in white. The *Customer* component (in the left), which already implements the distribution NFR, was completed with database persistence, using the Persistence package of the NFRs framework (in the right). The *PersistObjAspectCustomer* aspect was created to store the customer data. *ServletAddCustomer*, *ServletDeleteCustomer*, and *ServletConsultCustomers* are responsible for obtaining data from the user, via Web, and passing them to the *Customer* component.

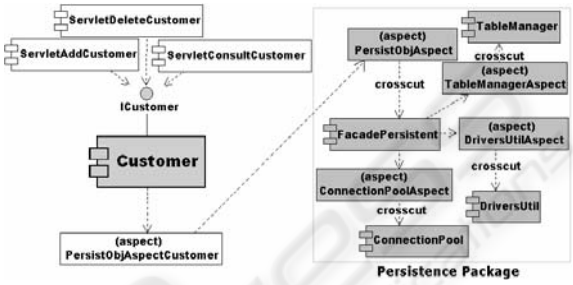

Figure 7: Design of Customer Application

2.2.3 Implement Application

At last, based on the design, the software engineer creates the code of the application. In order to accomplish this, the weaving process described in section 2.1.4 is used. In this case, software engineer builds the *servlets* and weaves them in the structure shown in Figure 7. The *servlet* can not just compiling (e.g. java compiler), in another words, when using AOP, every new components built must be integrated to the system using the weaving process because if just compiled with another compiler, all join points previously defined wouldn't be understandable by these new components. For example, the *ServletAddCustomer* needs to initialize the *SourceCustomer* component, though its interface, and after its initialization, the distribution aspect crosscuts the component to register its interface in the naming service, as already explained in section 2.1.3. Only then the *ServletAddCustomer* could use the services available.

2.2.4 Application Test and Deployment

As happens in domain test, the application test is performed in two levels: individual components modified by the software engineer in the Design Application phase and application development reusing components and aspects.

i. If a component is modified, it is individually tested by building a tests application around the component that allows its functionality to be exercised. Still, its necessary perform again the tests to verify if the component behavior was kept (i.e.

check if the aspect does not affect the component behavior). For this, the first test step of the Domain Test phase is performed again.

ii. The last form of testing is at the level of the application. The application is submitted to a long series of tests before taken into production. Here, in the case of the software engineer choose to use some components and aspects of the NFRs framework is necessary to do the test looking for maintain the application behavior. However, in this case, the software engineer chooses only reuse the persistence framework and, thus, does not affect the behavior of the application development.

Tests help to verify the quality of the new parts inserted in the components, analysing if the component behaviour was kept, and to assure that the constructed application fulfils its requirements.

At the end, the application can be deployed. According to the component model that was used, this process may involve several issues, such as distribution, database access, installation, reconfiguration and adaptation of the components. An intelligent deployment tool may be used to manage, package, and deploy the component-based application, obtaining better performance and result.

3 CONCLUSION AND FUTURE WORKS

The main contribution of this work is to propose an Incremental Process Model based on AOP for Component-Based Systems, providing a high reutilization degree, guiding the software engineer during the development and reuse of software components.

Our studies have shown that it is possible to incrementally add these common NFRs into components and applications, through a framework. However, for other kinds of NFRs, such as performance issues and concurrency control, this may not be true. In future works, we intend to treat the NFRs from the beginning of the requirements elicitation. We believe that the features model may be a good choice. This may help to determine whether or not it is possible to adopt an incremental approach for adding any kind of NFRs.

Finally, tools to facilitate the usage of Aspect IPM are currently being developed. This includes tools to facilitate the search and recovery of software artifacts, aiming to minimize the effort of locating reusable assets (Lucrédio, 2004).

REFERENCES

C.W. Krueger, 1992. Software Reuse, In *ACM Computing Surveys*, v. 24, no. 02, June, pp. 131-183.

I. Jacobson; M. Griss; P. Jonsson, 1997. *Software Reuse: Architecture, Process and Organization for Business Success*. Addison-Wesley-Longman.

G.T. Heineman; W.T. Councill, 2001. *Component-Based Software Engineering, Putting the Pieces Together*. Addison-Wesley. USA.

C. Szyperski, 2002. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley. USA.

N.C. Rosa, 2001. *Nfi: An Architecture-Based Approach for Treating Non-Functional Properties of Dynamic Distributed Systems*. PhD Thesis, Federal University of Pernambuco (UFPE), Brazil.

G. Kiczales; J. Lamping; A. Mendhekar; C. Maeda; C.V. Lopes; J.M. Loingtier; J. Irwin, 1997. Aspect-Oriented Programming (AOP). In *11th European Conference on Object-Oriented Programming (ECOOP)* - LNCS, Springer-Verlag, Finland.

J. Mylopoulos; L. Chung; B. Nixon, 1992. Representing and Using Non-Functional Requirements: A Process Oriented Approach. In *IEEE Transaction on Software Engineering*, v. 18, no. 6, June, pp. 483-497.

A. Alvaro; D. Lucrédio; A.F. Prado; E.S. Almeida, 2004. Towards an Incremental Process Model based on AOP for Distributed Component-Based Software Development. In: *International Symposium on Distributed Objects and Applications (DOA)*, Poster Session, Cyprus. Lecture Notes in Computer Science (LNCS), Springer-Verlag.

M. Griss; J. Favaro; M. D'Alessandro, 1998. Integrating Feature Modeling with RSEB. In *5th International Conference on Software Reuse (ICSR)*, IEEE Press., Canada, June.

M. Griss, 1997. *Domain Engineering and Variability in the Reuse-Driven Software Engineering Business*. Fusion Newsletter, February.

R. Prieto-Diaz, 1990. Domain Analysis: An Introduction. In *ACM SIGSOFT Software Engineering Notes*. v. 15, no. 2, April, pp. 47-54.

G. Arango, 1988. *Domain Engineering for Software Reuse*. Ph.D. Thesis, Department of Information and Computer Science, University of California, Irvine.

A. Alvaro; D. Lucrédio; E.S. Almeida; A.F. Prado; L.C. Trevelin, 2003. A Framework of Components to Non-Functional Aspects. (*in portuguese*). In *Third Component-Based Development Workshop*, São Carlos, Brazil.

S. Clarke; R. Walker, 2002. Towards a Standard Design Language for AOSD. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD)*, ACM Press, NY.

Eclipse Projects, 2003. *AspectJ – Aspect-Oriented Programing (AOP)*. Available in http://eclipse.org/aspectj Consulted in November.

R. Braga, 2000. *Search and Recover of Components in Software Reuse Environments*.(*in portuguese*) PhD. Thesis, Federal University of Rio de Janeiro, Brazil.

Y. Ye; G. Fischer, 2002. Supporting Reuse by Delivering Task-Relevant and Personalized Information. In *24th International Conference on Software Engineering (ICSE)*, USA.

D. Lucrédio, 2004. *Extension of MVCASE tool with Remote Services for Storage and Recovery of Software Artifacts (in portuguese)*, On going MSc. Dissertation, Federal University of São Carlos, Brazil.