2012 International Conference on System Engineering and Modeling (ICSEM 2012)
IPCSIT vol. 34 (2012) © (2012) IACSIT Press, Singapore

Building System Software Using Aspect Orientation Framework

Paniti Netinant

School of Science and Technology, Information Technology Department
Bangkok University, Thailand

Abstract. Building system software suffers from many of the assets and some of the weaknesses that are experienced by developer. Many accomplishments already achieved using software engineering approaches, but there are some significant works yet to do. Aspect-oriented approach has shown to be an effective means of capturing and collaborating system software design. We believe that aspect oriented approach can be used for system software development on several phases. In this paper, we explore what we have come to understand as crucial aspects of an extensible and adaptable model for system software. We present an extensible and adaptable technique in which system software is decomposed into a set of architectural abstractions. A design pattern is used to describe the separation of concerns among the components of the abstractions. The proposed model provides a formal methodology for the architectural design and specification of system software.

Keywords: Aspect Orientation, Adaptability, Extensibility, Framework, and System Software.

1. Introduction

One of the key characteristics of the system software needs and demands for faster adaptability and extensibility. Adaptable system software can be adapted with respect to software and hardware changes that will need to perform. Extensible system software can be extended with respect to requirements that will add in system software. Most software faults and descents are caused by changes in software [1]. However, the new features or requirements cannot be avoided. Such changes are often the result of the system software evolution and modifications in the underlying requirements of system software to meet these evolving needs. Certain refinements can be applied to traditional object-oriented analysis and design techniques. However, such refinements must not too complicate. Simplicity in the development is considered an important characteristic of a good ideal. To demonstrate the simplicity and practical of the adaptable and extensible model, a study of benchmark problems such as readers/writers problems usually use to implement in a system software design and implementation.

System software consists of splitting multiple concerns across over many components of the system. Systems are notorious of many crosscutting concerns such as synchronization, scheduling, fault tolerance, logging, and etc. We refer these crosscutting concerns as system properties. System property is an aspectual object or component. By supporting separation of concerns in the system software, we can provide a number of benefits such as easy to comprehension, reusability, extensibility, and adaptability for system software. In both the design and implementation of system software, the system designer has to consider how a number of system properties can be captured, and how a separation of concerns [5] will be addressed. Functional decomposition has so far been used based on the components and layering paradigm. In object-oriented programming, these dimensions are layers and components; included methods, objects and classes. Current programming languages and techniques have been supportive to functional decomposition. However, languages are specific domain. Furthermore; system software design has also been aligned with traditional functional decomposition techniques. No functional decomposition technique has yet managed to address a complete separation of concerns. Object-oriented programming seems to work well only if the problem can be described with relatively simple interfaces among objects. Unfortunately, this is not the case when we move

from sequential programming to concurrent and distributed programming. As a system becomes larger, the interaction of their components is becoming more complex. This collaboration may limit reuse, make it grim to validate the design and accuracy of system software, and thus force reengineering of these systems either to meet new requirements or to improve the system. Definite system properties of the system do not localize well. They tend to cut across over groups of components or services (functions or methods) in the system. System properties tangle in components or services making the system hard to adapt and extend. To avoid massive changes, changing needs to understand and correctly identify both system properties and core services of components. It is tightly couple design and implementation between components and system properties. In this paper we focus on adaptability and extensibility by proposing an adaptable and extensible model that is the basic of a framework for the system software development. This adaptable and extensible model, using the aspect-oriented techniques [3 and 4], provides a declarative way of developing, handling, and characterizing adaptable and extensible system software and represents a novel attempt to decompose and compose the system properties and components.

2. The Aspect-Oriented Framework

In this section we briefly describe the aspect-oriented framework [6 and 7]. We have designed the framework in order to support the development and deployment of adaptable and extensible system software. The overall framework architecture is composed of components.

Our framework is founded on aspect orientation, which is three dimensions of system design. The framework consists of components, aspects, and layers. Components consist of the modules that provide the basic functionality of the system such as the file system, communication, and process management, etc. Aspects are crosscutting system properties, and they can be a fault tolerance, synchronization, and scheduling, naming, etc. Layers consist of the components and system properties. In general, lower layers deal with a far shorter time scale. The lower the layer, the closer it is to the hardware. The higher layer deals with interaction with the user.

By adding the aspect dimension to a two dimensional model, system properties and functional components are divided from each other in every layer. It makes the system software design and implementation more modularity as well as makes it loosely coupled. Each layer has defined functionalities, system properties, and interfaces with the two adjacent layers. Each layer can be designed, implemented, and tested independently. The upper layer can reuse the layer beneath without knowing how the lower aspects or components are implemented. The upper layer does not have to build own system property components from scratch. However, new aspectual property components can be added to a layer without interfering with system property components or functional components in the layer underneath. It gives the system software easier extensibility and adaptability. Adding new system property components, which are orthogonal, requires no changes in functional components or system property components in other layers. Modifying a system property component needs no changes in system property components in the other layers. With current growth and rapid change, in technology and the features of system software, this architecture allows both functional components and system property components to be added into the system software more easily. The three-dimensional model makes it possible to manage both system property components and functional components in each layer.

By isolating the different system property of each component, we can separate functional components, system properties, and layers from each other (components from each other, system properties from each other, layers from each other, functional components from system properties, functional components in each layer, and system property in each layer). It would thus be possible to abstract and compose them to produce the overall system. This would result in the clarification of interaction and increased understanding of system properties of each functional component in the system. A high level of abstraction is easier to understand. Further, the reusability achieved by the higher level can use the lower level of the implementation not only to promote extensibility and refinement, but also to reduce cost and time in system development. A change in the implementation at a lower level would not result in a change at the higher level if the interface level has not been changed. Thus the design can achieve stability, consistency, and separation of concerns as well. A system property may have multiple domains. Some system properties (scheduling, synchronization, naming, and fault

tolerance, e.g.) are scattered among many components in the system with varying policies, different mechanisms, and possibly under different applications. To reduce the tangling of system properties in system software each system property can be considered and analyzed separately. For example, a system property of scheduling in file systems can be considered in different domains in each layer. This would separate policy from a system property of each layer. A system property interface would represent the general specifications needed to provide the abstraction. Further, a policy can be added or modified in each layer for each specific domain. This approach can support reusability to achieve adaptability.

One way of structuring system software is to decompose it into layers. Each layer is decomposed into its components. This decomposition of the system design both horizontally and vertically helps to deal with the complexity and reusability of system software. The layered architectural design decomposes a system into a set of horizontal layers where each layer provides an additional level of abstraction over the next lower layer and provides an interface for using the abstraction it represents to a higher-level layer. Every layer is decomposed into system components and system properties. System components and system properties are separated from each other. Changing either system components or system aspectual properties does not affect the other. The advantage of this decomposition is that system software tends to be easy to understand, adapt, extend, and maintain. Each layer can be understood, adapt, extend, and maintained individually without affecting other layers. However, it may be bad for performance and traceability because of using lower layer components.

The framework expresses a fundamental paradigm for structuring system software, a vertical composition of each layer where system components and system aspectual properties are composed into an abstraction of the layer. The framework structure can be described by the design pattern [2]. The framework uses a client-server model in which the server components (Functional Components and System Components) are composed by the Aspect Moderator and make their services available to clients. Clients access the server component services by sending requests to the Proxy component. The Proxy component intercepts a requesting message from clients and forwards the message to the Aspect Moderator component. The Aspect Moderator component locates and instantiates the composition rules defined by pointcut(s) – a collection of join points consists of join points between functional components and system property components. Figure 1 illustrates the modeling of the adaptable and extensible framework.

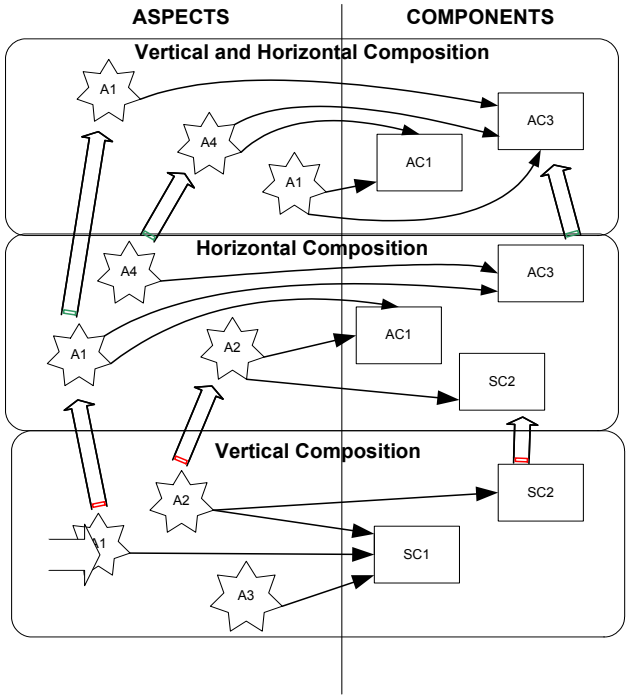

Fig. 1 An Architectural View of the Aspect-Oriented Framework

The framework expresses a fundamental paradigm for structuring system software, a vertical composition of each layer where system components and system aspectual properties are composed into an abstraction of the layer. The framework structure can be described by the design pattern [2]. The framework

uses a client-server model in which the server components (Functional Components and System Components) are composed by the Aspect Moderator and make their services available to clients. Clients access the server component services by sending requests to the Proxy component. The Proxy component intercepts a requesting message from clients and forwards the message to the Aspect Moderator component. The Aspect Moderator component locates and instantiates the composition rules defined by pointcut(s) – a collection of join points consists of join points between functional components and system property components. Figure 3 illustrates the modeling of the adaptable and extensible framework.

3. Characteristic Of The Framework

The framework supports both vertical and horizontal reusability. Reusable assets in the framework can be found in the vertical composition, where the upper neighbor layer can reuse a functional component or an aspectual property component from the lower layer. There are two levels of reuse in the aspect-oriented framework: Inter-layer reuse: reuse of functional components or aspectual property components from the lower layer, such as using an aspectual component derived from an abstract aspect. Intra-layer reuse: reuse of functional components or aspectual property components from the same layer, such as using an aspectual component to solve another problem. The aspect-oriented framework provides a better way to reuse both design and implementation code. Both inter and intra-layer reuse can be divided into three levels of reuse in the aspect-oriented framework as follows: Functional component: Reuse of functional component(s), such as reuse or redefinition of the functional component. System property component:

Reuse of an aspectual property component, such as reuse or redefinition of the aspectual property component.

Reuse of a framework provides a set of classes that manifest as an abstract design and implementation for solutions to a set of related problem.

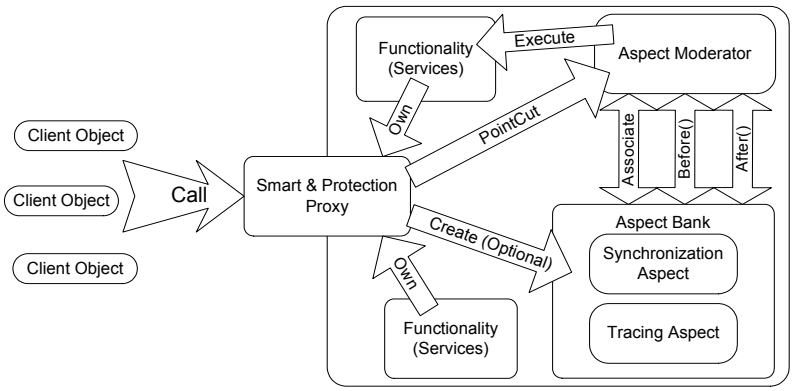

Fig. 2 The Model of the Adaptable and Extensible Framework

The aspect-oriented framework supports both vertical and horizontal compositions. Functional and aspectual property components in the framework can be composed vertically or horizontally. In vertical composition, the upper layer can use the lower functional or aspectual property components from the lower layer. In horizontal composition, functional and aspectual property components in the particular layer only use to be composed. The framework is based on system aspectual decomposition of crosscutting concerns in operating system design and implementation. The framework consists of two frameworks: the Base Layer and the Application Layer Framework. A system aspectual property is implemented in the SystemAspect class, while a component of the system is implemented as a Component class. The framework uses PointCut, Precondition, and Advice. The AspectModerator class, where the point cut is defined, combines both system aspectual properties and components together at runtime. Pointcuts are defined collections of join points, where system aspectual properties will be altered and executed in the program flow. Every aspectual property can identify and implement preconditions. A precondition is defined a set of conditions or requirements that must hold in order that an aspect may be executed. Advice is a defined collection of methods for each aspectual property that should be executed at join points. Advice can be either before or after advice. Before advice can be implemented as blocking or non-blocking. Before advice is executed when the join point is reached, before

the component is executed, if the precondition holds. After advice is executed after the component at the join point is executed. Every aspectual property will define advice methods. Figures 2 illustrate the execution model of a pointcut in the framework based on inter-dependency and intra-dependency.

4. Smooth of Adaptability and Extensibility

One important aspect of the framework is that it can separate functional components and system property components (system and application depending on the layer of a framework). A client object calls the services from the servers through a proxy object, rather than having services called directly from a client object. Then the framework creates necessary objects and calls the appropriate system properties to perform a specific service. In other words, the framework is like a mixer that combines and coordinates the crosscutting concerns of a specific service. The rules are used to combine and coordinate a functional component (a service of the system) defined by the pointcuts. Thus, for a particular system or application, it can adapt the generic functional components or system property components defined in the framework. The framework supports adaptability and extensibility in either of two ways:

Extensibility: Derive new components from the framework: They can be either functional components or system property components.

Adaptability: Instantiate and compose existing classes: They can be either functional components or system property components.

5. Conclusion

In this paper, we tense the importance of the better split-up of concerns within the context of an adaptable and extensible using aspect-oriented framework. Using the readers/writers benchmark problem, we show how this technique could provide an alternative to system software design and implementation, and show how our approach can be achieved separation of concerns crosscutting in the system. Our work concentrates on the decomposition of system properties crosscutting functional components in the systems implementation of system software to isolate the crosscutting concerns.

Our framework provides an invasive adaptable model that allows for minimal changes as well as it is open languages in object-oriented programming. Our goal is to achieve a better design and architectures where new system property and components can be easily manageable and added without invasive changes or modifications. The framework approach is promising, as it seems to be able to address a large number of system software and system property components. The advantage of decomposing of functional components and system property in every layer is to promote reusability, adaptability, manageability, and extensibility of both components and system property in system software easier without interfering each other. In the future, the framework will be extended and demonstrated for distributed object environment.

6. Acknowledgments

This research project was funded a partial financial support of Bangkok University, Thailand. We would like to express our gratitude for kindness of supporting.

7. References

[1] Fayad, M. and Altman, A. “An introduction to software stability,” Communications of ACM, Vol.44 (9), 95-98 (2001).

[2] Gamma, E., Helm, R., Johnson, R., and Vlissides, J., “Design Patterns: Elements of Reusable Object-Oriented Software,” Addison-Wesley, Reading, MA, (1995).

[3] Kiczales, G., Lamping, J., Mendhekar, J.,Maeda, C., Lopes, C., Loingtier, J., and Irwin, J. “Aspect-Oriented Programming,” In M. Aksit and S. Matsuoka, editors. Proceedings of the 11th European Conference on Object-Oriented Programming, Lecture Notes in Computer Science number 1241, Springer Verlag, Berlin, 220-242 (1997).

[4] Lopes, C., Tekinerdogan, B., Meuter, W., and Kiczales, C. “Aspect-Oriented Programming,” In M. Aksit and

S.Matsuoka, editors, Proceedings of the 12th European Conference on Object-Oriented Programming ECOOP'98, Springer Verlag (1998).

[5] Parnas, D. "On the Criteria to be Used in Decomposing Systems into Modules," Communications of ACM, Vol. 15(12), 1053-1058 (1972).

[6] Netinant, P. "Extensibility Aspect-Oriented Framework to Build Agent-Based System Software," Proceedings of the International Conference on Software Engineering and Data Engineering (SEDE 2006), Los Angeles, California, USA (2006).

[7] Netinant, P., Elrad, T., and Fayad, M. "A Layered Approach to Building Open Aspect-Oriented Systems," Communications of ACM, Vol. 44(10), 83-85 (2001).