Supporting Adaptability and Modularity of System Software

Paniti Netinant
Computer Science Department,
Bangkok University,
40/4 Rama IV Road, Bangkok 10110, Thailand
e-mail: paniti.n@bu.ac.th

Abstract: It is difficult to design system software to meet a better separation of concerns, which can provide a number of benefits such as adaptability, extensibility, and modularity in the design and implementation. During design, some aspectual properties, such as synchronization, scheduling, performance and fault tolerance, crosscut the basic functionalities of the system software. By separating functional components from the different aspectual components of the system software in the design, we can provide a better generic design model of system software. Aspect-Oriented Programming is a methodology that aims at separating components and aspects from the early stages of the software life cycle, and using techniques to combining them together at the implementation phase. In this paper we discuss an aspect-oriented framework that can simplify system software design and implementation by expressing it at a higher level of abstraction. Our work concentrates on how to achieve a higher separation of aspectual components, functional components, and layers from each other. Our goal is to achieve a better design model for implementing system software in terms of modularity, reusability and adaptability.

1. Introduction

A key principle in software design is modularity. It is a process that decides how to decompose the system into modules [8, 2]. A stable modularity of software system may increase when shifting focus from programming to design, supported by OOP. It allows for modeling the system, oriented concepts in a direct, natural way. Object-oriented design and implementation of systems are based on separation of policy and mechanism where the policy is how things should be. The mechanism is what the system wants to do. However, the separation of concerns cannot completely be achieved using object-orient design and implementation and it will increase the price of the design effort to understand the system.

Dijkstra [4] advocated the layer approach, which is a two-dimensional model, to lessen the design and implementation complexity of the system software. The layered approach divided the operating systems into layers and components. The components or functions of system software are vertically composed into these adjacent layers. Each layer has well-defined functionality and input-output interfaces with the two adjacent layers. The bottom layer interfaces with the machine hardware and the top layer interface with users (specifications and application software). The examples of the layer approach to building system software, such as THE operating system [4], the MULTICS systems [15].

In OOD and OOP, 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. Further more, 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. OOD and OOP 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 distributed systems become larger, the interaction of their components is becoming more complex. This interaction may limit comprehension, reuse, and adaptability make it difficult to validate correctness of the design and implementation of system software, and thus force reengineering of the system either to meet new requirements or to improve the performance. Certain system aspectual properties of the system do not localize well. They tend to crosscut groups of components or services (functions or methods) in the system. System aspectual properties tangle in components or services making the system difficult to understand, reuse and adapt. Code tangling destroys modularity of the design and implementation. Changing needs to understand and correctly identify both system aspectual properties and core service implementation of the component(s) or service(s). It is tightly coupled design and implementation between functional components and system aspectual properties.

The primary objective of object-oriented system designs is to partition the subsystem into a set of interconnectable collaborative objects or components. Each object implements an important entity of complete system software. Object-Oriented Design (OOD) and Programming (OOP) can help to partition complex system software into smaller, and simple objects. However, separation of concerns cannot be completely achieved. System aspectual properties, such as synchronization, tracing, and fault tolerance, cannot be captured and localized in both the design and implementation. Codes of these system aspectual properties are scattered among all objects in the system design and implementation. For system software such as operating systems, the interaction of their components becomes more complex. This interaction may limit reusability, adaptability, and make it difficult to validate the design and correctness of the system. As a result, re-engineering of these systems might be inevitable

to meet future requirements. Some system aspectual properties in object-oriented operating systems such as synchronization, scheduling, and fault tolerance crosscut the basic functionality of the system, such as the file system and process management as illustrated in Figure 1.

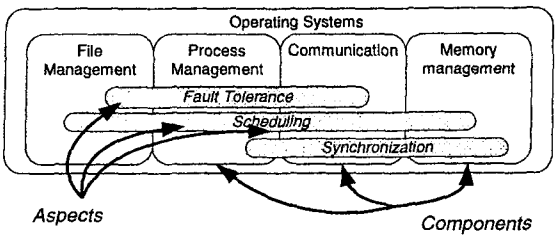

Figure 1. Fault tolerance and scheduling aspects cut across components in the operating systems such as file system, process management, and communication.

In every layered system, aspectual properties may need to be modified. By separating aspectual properties and functional components of operating systems in every layer, we can provide a better modularity model of operating system design and implementation, and better support of adaptability, reusability, and expandability.

Object-based technology has emerged in every field of computer science, including operating systems. Object-Oriented Programming (OOP) makes the system designers and programmers easier to design and understand interactions of a system as a collection of objects. Developer can model the entities of the system in the real world. Object-Oriented Programming helps to reduce the complexity of software system development. Object-Oriented based software has become increasingly developing and adopted for development of software system [19, 1]. Object-Oriented Programming is used and implemented in operating systems such as Clouds in [3], Chorus and COOL in [10], and Amoeba in [17].

The principle of separation of concerns lies at the heart of software development as it introduces a number of benefits, originally addressed by [5, 16]. These include better understanding, extensibility, adaptability [7] of the system, and better reuse of the concerns. Although these benefits have been well established, there is still no universally accepted methodology in order to guide a programmer to best achieve separation of concerns. Concerns are divided into system and application level. Operating systems consists of separating multiple *concerns* crosscutting many functional components of the system. Systems are notorious of many crosscutting concerns. We refer to these crosscutting concerns as *system aspectual properties*. Examples of system aspectual properties are synchronization, scheduling, performance, fault tolerance, logging, and etc. However, separation of concerns is difficult to accomplish. Some system aspectual properties in object-oriented operating systems crosscut the basic functionality of the system, such as file system and process management. Supporting separation of concerns in the operating systems makes the system better modularity, and can provide a number of benefits such as comprehension, reusability, extensibility and adaptability for system and application software. In both the design and implementation of the operating system, the system designer has to consider how a number of aspects can be captured in the design and implementation, and how a separation of concerns [16] will be addressed. Only supporting separation of concerns in the implementation makes the design and implementation inconsistent and it is unacceptable for software design and development. Functional decomposition has so far been used as well as achieved along two dimensions - based on components and layers.

Supporting separation of functional components and system aspectual properties in the design and implementation of operating systems makes the operating system better modularity, and can provide a number of benefits such as comprehension, reusability, extensibility and adaptability in both design and implementation. Aspect-Oriented Programming (AOP) [6, 9] is a paradigm proposal that aims at separating components and aspects. However, separating components and aspects should begin from the early stages of the software life cycle, including analysis and design, finally combines them together at the implementation phase.

We have developed the Open Layered Aspect-Oriented Systems (OL-AOS) using Aspect-Oriented Framework (AOF) for supporting the design and implementation of the system software. A framework represents the reusable and adaptable to both design and implementation of a family of system software, such as operating systems. The framework, represented as a collective set of collaborative abstract and concrete classes, enables separation of crosscutting concerns defined as system aspectual properties. An aspect-oriented framework simplifies the system design and implementation by expressing it at a better and higher level than what is provided by OOD and OOP. Our framework promises adaptability, reusability, and expandability of the design and implementation while separating functional components and system aspectual properties from each other in every layer of the system. In this paper we demonstrate an Aspect-Oriented Framework (CAL) that can be used for system software such as operating systems. We also show how the separation of system aspectual properties from functional components. Producers/Consumers problem is demonstrated using our framework. Our framework, which is based on aspect-oriented technology, is languages and architecture independence.

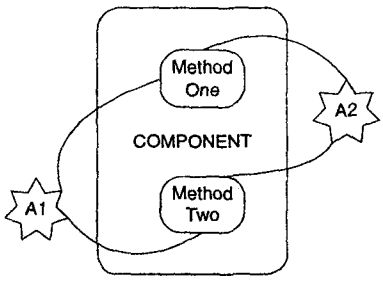

Figure 2. Intra-Dependency

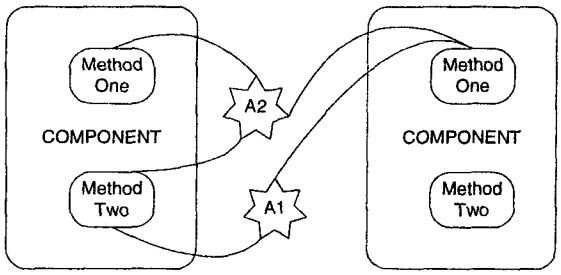

Figure 3. Inter-Dependency

2. System Aspectual Properties in the Systems Software

System aspectual properties are, for instances, mutual exclusion, scheduling, synchronization, fault tolerance, logging, tracing, security, load balancing, performance measurement, testing, verifications and etc. System software must juggle the implementation of a large variety of often interfering and crosscutting concerns. They are all expressed in such a way that tends to crosscut groups of functional components or services. This tangling design and implementation code of system aspectual properties results increasing of code dependencies between functional components and aspectual properties of the system. That code is not usually segregate into neat modules, but instead interacts and tangles with the underlying functionality. System developers want to be able to evolve this code easily as new services are demanded, new devices are invented, and new protocols are implemented. It makes their source code difficult to understand, reuse, adapt, and maintain. One current attempt to resolve this issue is the Aspect-Oriented System (AOS) [12]. AOS aims at language and architecture independence, where functional components and aspectual properties are separately decomposed in both design and implementation. These properties can be captured in the design and implementation, reused, and adapted in the application software later. Finally, functional components and system aspectual properties are combined together at run-time. We distinguish between functional components and aspects in the design of systems. System aspectual properties are defined as properties of the system that do not necessarily align with functional components or services but tend to crosscut groups of functional components, increasing either inter-dependency or intra-dependency, and thus affecting the quality of the software. Intra-dependency defines as a system aspectual property that crosscuts between many services (functionalities or methods) in the same components, as illustrated in Figure 2. Inter-dependency defines as a system aspectual property that crosscuts between many components or services, as illustrated in Figure 3.

Although not bound to OOP, Aspect-Oriented Software Development (AOSD) is a paradigm proposal that retains the advantages of OOP and aims at achieving a better separation of concerns. AOSD suggests that from the early stages of the software life cycle aspects should be addressed relatively separately from the components. As a result, aspectual decomposition manages to achieve a better design and implementation for both operating system and application. At the implementation phase, aspectual properties and functional components are combined together, forming the overall system.

In this paper we have shown the system design and implementation based on system aspectual decomposition in the context of the aspectual decomposition for the design and implementation of system software. Our approach is an aspect-oriented framework [13, 14]. Compared with what has so far been able to be supported by traditional approaches, our goals are to provide a better modularity, flexibility, higher reusability and adaptability, for the design and implementation of system software as well as to provide a technique that would be practical.

3. An Aspect-Oriented Framework for System Software

Our observation suggests that an Aspect-Oriented Systems (AOS) that uses Aspect-Oriented Framework could support designers and programmers in cleanly separating components and system aspectual properties from each other. Our framework is based on Aspect-Oriented techniques and layered approach [4]. We argue that system aspectual properties of the system software should be excluded from the system components or services if there is a possibility to often change it, and it should not be treated as a single monolithic aspect.

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 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 it's 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 aspectual properties. System components and system aspectual 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 and maintain. Each layer can be understood and maintained individually without affecting other layers. However, it may be bad for 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 uses a client-server model in which the server components (Functional Components and System Aspectual 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 Aspect Moderator component. The Aspect Moderator component locates and instantiates the composition rules defined by pointcut(s) – where consist of join points between functional components and system aspectual components.

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 system software design and implementation. The framework consists of two frameworks: The Based 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. Alike AspectJ [18], our framework uses *PointCut, Precondition*, and *Advice*.

4. Conclusion

In this paper, we stressed the importance of the better separation of concerns within the context of adaptability and modularity for system software. We discussed how aspect-oriented technique provide an alternative to system software design and implementation, and show how our approach can be achieved separation of crosscutting concerns in the design and implementation of system software. Our work concentrates on the decomposition of system aspectual properties, which crosscut functional components in the system software and our goal is to achieve a better adaptability and modularity for design and implementation of system software while supporting separation the crosscutting concerns in every layer. Our design framework provides an adaptable model that allows for open languages and architectures where new aspectual properties and components can be easily manageable and added without invasive changes or modifications. In application, system aspectual properties could be reused and redefined from the system layer preventing the re-engineering of all aspectual properties and functional components. The framework approach is promising, as it seems to be able to address a large number of system and application aspectual properties and functional components. The advantage of decomposing of functional components and aspectual properties makes the design and implementation of system software better modularity as well as is to promote comprehension, reusability, adaptability, manageability, and extensibility of both functional components and aspectual properties in the system software easily. In the future, the framework will be extended and demonstrated for distributed object environment.

References

[1] Cox, B. J. *Object-Oriented Programming: An Evolutionary Approach*, Second Edition, Addison-Wesley, Reading, MA, 1991.

[2] Crowley, C. *Operating Systems: A Design-Oriented Approach*, Irwin, Chicago, IL, 1997.

[3] Dasgupta, P. LeBlanc Jr., R., Ahamad, M., and Ramachandran U. The Clouds Distributed Operating System. *IEEE Computer*, Vol. 24, No.11, pp. 34-44, 1991.

[4] Dijkstra, Edsger W. The Structure of THE Multiprogramming System. *Communications of ACM*, Vol. 26, No. 1, pp. 49-52, January 1983.

[5] Dijkstra, Edsger W. *A Discipline of Programming*. Englandwood Cliff, NJ: Prentice-Hall, 1976.

[6] Elrad, T., R. E. Fieldman, and A. Bader. Aspect-Oriented Programming, *Communications of ACM*, Vol. 44, No. 10, pp. 29-32, October 2001.

[7] Fayad, M. E., and Cline, M. Aspect of Software Adaptability. *Communications of ACM*, Vol. 39, No. 10, pp.58-59, 1996.

[8] Galli, D. L. *Distributed Operating Systems: Concepts & Practice*, Prentice Hall, Upper Saddle River, NY, 2000.

[9] Kiczales, G., J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In M. Aksit and S. Matsuoka, editors. *Proceedings of the 11th European Conference on Object-Oriented Programming ECOOP 1997*, number 1241 in Lecture Notes in Computer Science, Springer Verlag, Berlin, pp. 220-242, Finland, June 9-13 1997.

[10] Lea, R., Jacquemot, C., and Pillevesse, E. COOL: System Support for Distributed Programming, *Communications of ACM*, Vol. 36, No. 9, pp. 37-46, 1993.

[11] Lopes, C., B. Tekinerdogan, W. de Meuter, and G. Kiczales. Aspect-Oriented Programming. In M. Aksit and S.Matsuoka, editors, *Proceedings of the 12th European Conference on Object-Oriented Programming ECOOP 1998*, Springer Verlag, 1998.

[12] Netinant, P., and T. Elrad. Building an Open Layered Aspect-Oriented Systems. *Communications of ACM*, Vol. 44, No. 10, pp. 83-85, October 2001.

[13] Netinant, P., and T. Elrad. Implementing Producers/Consumers Problem Using Aspect-Oriented Framework. *Proceedings of 4rd Workshop on Object-Orientation and Operating Systems ECOOP-OOOWS 2001*, pp. 11-16, Budapest, Hungary, June 2001.

[14] Netinant, P., C. A. Constantinides, T. Elrad, M. E. Fayad. Supporting Aspectual Decomposition in the Design of Adaptable Operating Systems Using Aspect-Oriented Frameworks. *Proceedings of 3rd Workshop on Object-Orientation and Operating Systems ECOOP-OOOWS 2000*, pp. 36-46, Sophia Antipolis, France, June 2000.

[15] Organick, E. *The Multics System*, MIT Press, Cambridge, MA, 1972.

[16] Parnas, D. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of ACM*, Vol. 15, No. 12, pp.1053-1058, December 1972.

[17] Tanenbaum, S. A. *Distributed Operating Systems*, Englewood Cliffs, NJ: Prentice Hall, 1995.

[18] The AspectJ Primer, in WebPages at http://www.aspectj.org, The AspectJ Team.

[19] Wegner, P. Dimensions of Object-Oriented Modeling, *IEEE Computer*, Vol. 25, No. 10, pp. 12-21, 1992.