# Aspect-Oriented Frameworks: The Design of Adaptable Operating Systems

P. Netinant[1], C. A. Constantinides[1], T. Elrad[1], and M. E. Fayad[2]

[1]Concurrent Research Group
Computer Science Department
Illinois Institute of Technology
Chicago, IL, U.S.A.
{netipan,elrad,conscon}@iit.edu

[2]Department of Computer Science and Engineering
University of Nebraska - Lincoln
Lincoln, NE, U.S.A.
fayad@cse.unl.edu

## ABSTRACT

With software systems 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. There is a general feeling that OOP promotes reuse and expandability by its very nature. This is a misconception as none of these issues is enforced. Rather, a software system must be specifically designed for reuse, expandability, and adaptability [4]. Operating systems are dominated in many aspects. Supporting separation of concerns and aspectual decomposition in the design of operating systems provides a number of benefits such as reusability, expandability, adaptability and reconfigurability. However, such support is difficult to accomplish. Aspect-Oriented Programming (AOP) [7] is a paradigm proposal that aims at separating components and aspects from the early stages of the software life cycle, and combines them together at the implementation phase. Besides, Aspect- Oriented Programming promotes the separation of the different aspects of components in the system into their natural form. However, Aspect-Oriented software engineering can be supported well if there is an operating system, which is built based on an aspect- oriented design. Therefore aspects can be created in applications, reused and adapted from the aspects provided by the operating systems. Object-Oriented Operating Systems treat aspects, components, and layers as a two dimensional models, which is not a good design model. Aspects in the operating system cannot be captured in the design and implementation. Two-dimensional models lead to inflexibility, limit possibilities for reuse and adaptability, and make it hard to understand and modify. The poster will show an Aspect-Oriented Framework [1, 8], which simplifies system design by expressing its design at a higher level of abstraction, for supporting the design of adaptable operating systems. A framework is more than a class hierarchy and it is a reusable to produce custom systems and applications [5]. Aspect-Oriented Framework is based on a three-dimensional design that consists of components, aspects, and layers.



OOPSLA 2000 Companion Minneapolis, Minnesota


Components consist of the basic functionality modules of the system. Aspects are the properties in the systems that cut across the components in the operating systems. Some aspects in operating systems such as synchronization, scheduling, fault-tolerance cut across, in horizontal and vertical, the basic functionality of the systems. Layers consist of the components and aspects. By separating aspects and components of the operating systems in every layer, we can provide a better generic design model of the operating systems. The framework uses design patterns [6]. The overall architecture is divided into two frameworks: Base Layer and Application Layer Framework. The poster will show The UML model of frameworks and how to maximize separation of aspects, components, and layers from each other. Our goal is to achieve a better design model and implementation of operating systems, in terms of reusability, adaptability, extensibility, and reconfigurability.

## 1. Problem: Cross-Cutting of Design Issues

The principle of separation concerns introduces a number of benefits, originally addressed by [3, 9]. These include better understanding, modifiability, extensibility, debugging of the system, and better reuse of the concern. The primary idea for operating system design has traditionally been based on functional decomposition where a problem is broken into sub-problems that are addressed relatively separately. Traditional languages and techniques have been supportive to functional decomposition. However, no decomposition technique has yet managed to address a complete separation of concerns. Further, certain properties in the system seem to cut across a number of functional components, making the system difficult to understand. Example properties include fault-tolerance, scheduling, synchronization, and distribution. The tangling of concerns in the designing system leads to increase the implementation dependency between functional components and aspects, which makes its source code difficult to understand, evolve, and maintain. Aspect-Oriented Programming [7] suggests that these properties should be addressed relatively separately at the analysis and design phases, and be combined with the main functionality of the system at the implementation phase.

We address a number of operating system design issues in the context of AOP and we discuss our approach in the context of the supporting aspectual decomposition for the designing of the operating systems. Our goals are to provide a clean separation of design concerns, flexibility, reusability, as well as to provide a methodology that would be practical to implement. We want to provide the design that aspects of the operating system can be

captured in both the design and implementation. We want to demonstrate that AOP can be used effectively in this area, as an alternative design approach.

## 2. The Aspect-Oriented Frameworks

Our observation suggests that AOP could support designers and programmers in cleanly separating components and aspects from each other on each layer. AOP can provide a mechanism that would make it possible to abstract and compose components and aspects to produce the overall system such as aspect moderator framework [2]. We argue that a cross-cutting property of the system should not be seen within a two-dimensional model, and it should not be treated as a single monolithic aspect.

Our proposed framework is based on a three-dimensional system design that consists of components, aspects, and layers. 1). Components consist of the basic functionality modules of the system such as the file system, communication, and process management etc. 2). Aspects are cross-cutting entities, and they include fault tolerance, synchronization, and scheduling, naming etc. 3). Layers consist of the components and aspects decomposed into a number of more manageable sub-problems. In general, lower layers deal with a far shorter time scale. The lower the layer, the closer the hardware is. The higher layer deals with interaction with the user.

By separating the different aspects of each component, we can separate components, aspects, and layers from each other (components from each other, aspects from each other, layers from each other, components from aspects, components in each layer, and aspects 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 aspects of each component in the system. 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. An aspect might have multiple domains. Some aspect (scheduling, synchronization, naming, and fault tolerance e.g.) is scattered among many components in the system with varying policies, different mechanism, and possibly under different application. To reduce tangling of aspects in an operating system each aspect can be considered and analyzed separately. For example, an aspect of scheduling in the file systems can be considered in different domains in each layer. It would separate policy from an aspect of each layer. Aspects would represent the general specifications needed to provide the abstraction. Further, a policy can be added or modified in each layer to specific domain. This approach can support reusability.

The overall framework architecture is divided into two frameworks based on two layers: a base framework on the low layer and an application framework on the upper layer. The Base Framework corresponds to the system layer. On the upper layer we may have more than one application frameworks.

In this framework, aspects are created using the Abstract Factory and the Bridge patterns. The Abstract Factory would isolate aspects from implementation classes because the factory encapsulates the responsibility and the process of creating aspect objects. The class of concrete aspect appears only once in a functionality, where it's instantiated. The framework promotes consistency when an aspect is modified. The Bridge pattern avoids a permanent binding between an abstraction and its implementation. An example where this would be beneficial is when an implementation concern must be selected or switched at run-time. This way, different aspect abstractions and implementations can be combined and extended independently. This implementation is still useful when a change in the implementation of a class must not affect its existing components. As a result, a class need not be recompiled, but just re-linked. This approach supports polymorphism, and manages to avoid proliferation. Changing the implementation of an aspect abstraction should have no impact on functionality either. A smart-protection proxy controls access to the aspects and allows additional housekeeping tasks when an aspect is accessed.

In the application framework, the Adapter pattern allows the aspect factory to either convert the interface of an existing aspect (super aspect or aspects in the lower layers) into another interface functionality expect or to create a new aspect. Ideally, a new aspect should reuse an existing aspect to create new aspects, when it could be used. The upper layer can redefine existing aspects and override them.

The general architecture of the framework promotes reusability (the upper layer can reuse aspects from the lower layer), extensibility, and ensures adaptability of aspects and components because both are designed and implemented relatively separately from each other. Aspects in the application framework can be extended and redefined by aspects provided by the layer to meet new requirements. A new aspect can be added in both system layer and application layer without interfering with aspects or components in the other layer.

## REFERENCES


[1] Constantinides, C.A., A. Bader, T. Elrad, M.E. Fayad, and P. Netinant. Designing an Aspect-Oriented Framework in an Object-Oriented Environment. *ACM Computing Surveys*, Submitted for publication in March 2000.

[2] Constantinides C.A., A Bader, and T. Elrad. A Framework to Adress a Two-Dimensional Separation of Concens. Position paper to the OOPSLA'99 Workshop on Multidimentional Separation of Concerns, Denver, CO, November 1999, np.

[3] Dijkstra, Edsger W., A Discipline of Programming, Prentice-Hall, 1976.

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

[5] Fayad, M.E., W. Pree, and D.S. Hamu. Achieving Bottom-Line improvements with Enterprise Frameworks. Position paper to the OOPSLA'99 First Workshop on Enterprise Frameworks, Denver, CO, November 1999, np.

[6] Gamma E., R. Helm, R. Johnson, and J. Vlissides. Design Pattern: Elements of Reusable Object-Oriented Software. Reading, MA: Addison-Wesley, 1995.

[7] Kiczales G., J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.M. Loingtier, and J. Irwin. Aspect-Oriented Programming. *ACM Computing Surveys*, Vol. 28, No. 4es, Articles No.154, December 1996, np.

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

[9] Parnas, D. On the criteria to be used in decomposing systems into modules. *Communications of ACM*, Vol. 15, No. 12, December 1972, pp.1053-1058.