AOSA - Betriebssystemkomponenten und der Aspektmoderatoransatz

Results obtained by researchers in the aspect-oriented programming are promoting the aim to export these ideas to whole software development system.

Operating system code is quite complex and some of this complexity is caused by modularity problems. Aspect-oriented programming is being explored in nowadays research as a means of dealing with this kind of complexity.

Aspect-oriented programming (AOP) has put forth the idea that some concerns are inherently crosscutting by their very nature they are present in more than one module. Such concerns are being called aspects of the system. The goal of work in AOP is to make it possible to modularize the implementation of aspects by developing mecha nisms that explicitly support crosscutting structure. Several applications have successfully used AOP to structure issues such as synchronization and performance optimization.
Supporting separation of concerns in the design of operating systems can provide a number of benefits such as reusability, extensibility and reconfiguration.

Operating Systems Design issues

The commercialization of operating systems has resulted in their design gaining more importance. Operating systems are constantly extended for improvements as well as to support new features and hardware. In order to support this, reusability and adaptability of system software during design is crucial.

Operating system design issues can be divided into hardware-oriented and software-oriented. Hardware oriented issues include :

- physical networks
- communications protocol design
- hardware measures
- physical clock synchronization
- storage
- system components

On the other hand, software-oriented issues include

- distributed algorithms
- naming
- resource allocation, distributed operating systems, system integration, reliability, tools and languages
- real-time systems
- performance measurement

Issues like stability, reusability, adaptability and reconfigurability are included in the design and implementation of operating systems.

Problems

Separation of concerns is at the heart of software development, and although its benefits have been well established, the core problem remains how to achieve it. There is still no universally accepted methodology in order to guide a programmer to achieve it. The system designer has to consider how a number of aspects in the system can be captured, and how a separation of concerns will be addressed.

Object Oriented Programming (OOP) works well only if the problem at hand can be described with relatively simple interface among objects. Unfortunately, this is not the case when we move from sequential programming to concurrent and distributed programming.

Functional decomposition is being used as one of the techniques but it has so far been achieved along one dimension, based on the underlying paradigm. In functional decomposition a problem is broken down into sub-problems that can be addressed relatively independently. In OOP, this dimension is a component hierarchy that includes methods, objects and classes. Current programming languages and techniques do support functional decomposition. Moreover operating system design has also been based on traditional functional decomposition techniques. However, no functional decomposition technique has yet managed to address a complete separation of concerns. Object Oriented Programming (OOP) works well only in simple scenarios where a problem can be described with relatively simple interface among objects.

Unfortunately, this is not the case when we move from sequential programming to concurrent and distributed programming. With the increase in the size of distributed systems, the interactions between their components become more and more complex. Consequences of this interaction may be limited reuse and it makes difficult to validate the design and correctness of software systems. Also force reengineering of these systems may be needed in order to meet future requirements.

Certain properties of the systems do not localize well. Rather, they tend to cut across groups of functional components, making the system difficult to understand. In the OOP paradigm, these properties are not localized to objects. These properties do not arise randomly, but tend to constitute emergent entities, arising during execution. Example properties include synchronization, scheduling, resource allocation, performance optimizations, failure handling, persistence, communication, replication, coordination, memory management, and real-time constraints. Manually programming these properties into the system's functionality using current languages or methodologies results in these properties being spread throughout the code. Aspects were originally introduced by Kiczales and are defined as system properties that tend to cut across functional components, increasing their interdependencies, and resulting in what is being described as "the code-tangling problem." This code tangling makes the source code difficult to develop, understand and evolve by destroying modularity and reducing software quality.

The core complexity is that concurrent and distributed systems manifest over more than one dimension. Features such as scheduling, synchronization, fault tolerance, security, testing and verifications are all expressed in such a way that they tend to cut across groups of objects as shown in figure 1.

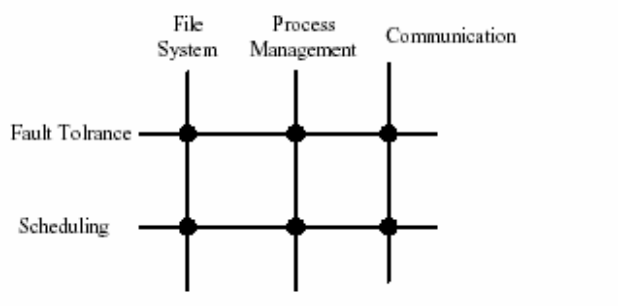

Figure 1: Aspects like fault tolerance, scheduling cut across functional components like file system, process management etc.

This tangling of concerns results in an increase of the dependencies between functional components that makes their source code difficult to understand, develop, and maintain. As a result, simple object interfaces are violated and the traditional OOP benefits no longer hold.

Aspect-oriented Programming

Aspect-oriented programming is one of the approaches to resolve this issue of code tangling and separation of concerns. Components and aspects are distinguished in concurrent systems to address their multi-dimensional structure. Aspects are defined as properties of a system that do not necessarily align with the system's functional components but tend to cut across groups of functional components, increasing their interdependencies, and thus affecting the quality of the software. Aspect-oriented programming is not bound to OOP but it does retain the advantages of OOP and aims at achieving a better separation of concerns. AOP suggests that from the early stages of the software life cycle aspects should be addressed relatively separately from the functional components. As a result, aspectual decomposition manages to achieve a two-dimensional separation of concerns. At the implementation phase, aspects and components should be combined together, forming the overall system. Mainly aspect-oriented programming is based of following steps:

- Identify concerns that *cut across* class / object boundaries
- Write code (an aspect) that encapsulates that concern
- Define a weave that specifies how the aspect will be weaved into existing code

During the last years, framework technology has consolidated as a suitable technology for the design and implementation of complex distributed systems. A framework is a reusable design of all or part of a system that is represented by a set of abstract classes and the way their instances interact. Aspect-oriented frameworks should be able to provide a cleaner separation of design concerns compared with what has so far been able to be supported by traditional approaches, better flexibility and higher reusability, as well as to provide a technique that would be practical to implement. The framework approach being discussed here in this presentation is based on aspectual system decomposition and is also language independent.

Architectural Issues in Aspect-oriented programming

There are number of different issues related with current AOP architectures and a number of these are being described here:

Language:

There are mainly three approaches for the implementation of aspects:

1) using libraries,
2) using domain-specific aspect languages and
3) designing language extensions for the support of aspects.

Language extensions seem to have a number of advantages over domain-specific languages like they are more scalable, and also they allow for the reuse of the compiler infrastructure and language implementation.

There have been suggestions of having appropriate languages for the expression of aspects based on the arguments that current languages do not provide the right abstraction for the description of aspects and also it is argued that aspect languages make aspect code more concise and easier to understand. If aspects are expressed in domain-specific languages, one needs an aspect language for every type of aspect and an automatic weaver tool would implement one (or more) aspect languages. Proposals of specific languages for the support and implementation of AOP include COOL, RIDL , D2-AL, IL and TyRuBa . Proposals for extensions to general-purpose languages include AspectJ which extends the Java language, AspectC++ etc.

Static and Dynamic weaving

As been discussed before that systems are decomposed into aspects and functional components. Now one main issue in the proposals for supporting an Aspect-Oriented Software Architecture (AOSA) resides in the way in which aspects are weaved across the functional components of the system. There are mainly two approaches in this regard:

? Static weaving

- Dynamic weaving

Static weaving is implemented in automatic weavers. Aspects reference the classes of those objects whose behavioural additions describe, and define at which join points additions should be made. Static weaving means to modify the source code of a class by inserting aspect-specific statements at join points. In other words, aspect code is inlined into classes. The result is highly optimized woven code whose execution speed is comparable to that of code written without AOSA. There are certain problems associated with this approach as well like it makes difficult to later identify aspect-specific statements in woven code. As a consequence, adapting or replacing aspects dynamically during run-time can be time consuming or not possible at all. Currently, automatic weaving technology cannot support aspects in a dynamic environment.

Dynamic weaving facilitates incremental weaving and makes debugging easier. An example where this would be beneficial is given by where a load balancing aspect could replace the load distribution strategy woven before with a better one depending on the current load of managed servers. Architectures that make use of reflective technologies allow dynamic weaving.

Ideally, an implementation should support both static and dynamic weaving.

Code Transformation

Another issue is whether there is source-to-source code transformation, from separate constructs to an intermingled source code. Technologies that rely on automatic weavers produce code transformation. Examples include D, AspectJ, D2-AL and IL. Reflective technologies will typically not have code transformation.

Level of weaving

A difference in the AOSA proposals is in the level of weaving. The level of weaving defines the point up to which one manages to achieve separation of concerns in the software system.

There are mainly two levels:

- Pre-compile
- Compile-time

Pre-compile is having two phases of compilation: one for the weaver to produce 'woven' (intermingled source) code, and another for the final compilation into an executable code.

Compile-time is a level of weaving where the intermingled code exists only at the binary level. An example of compile-level weaving architecture is the Aspect Moderator Framework which is the core idea behind Aspect-oriented framework which will be

discussed in the coming sections. The level of weaving can also be run-time, as in the case of reflective architectures. As a result, the interpretation of Figure 2 will depend on the level on weaving in discussion.

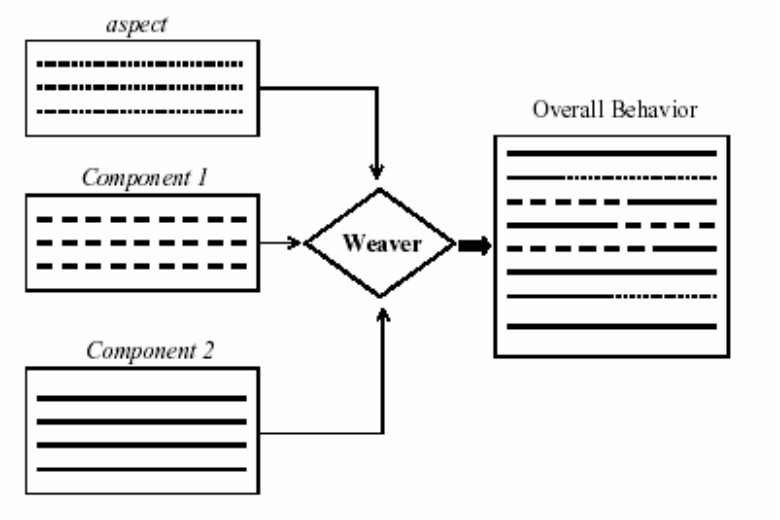

Figure 2 : Aspect weaver (weaving aspects and components together)

Open versus Closed Implementations

There are tradeoffs between using a language and adopting an open language. A language is ready to program but it is limited to the facilities that it provides. Clearly, opening a language can be considered a risky approach, as the semantics of the extension mechanisms should balance openness with protection and security.

Aspect Moderator Framework

Aspect Moderator framework has been extended to develop Aspect-oriented framework architecture. Aspect Moderator framework is based on idea of a concurrent (shared) object as being decomposed into a set of abstractions that form a cluster of cooperating objects: a functional behaviour, synchronization, and scheduling. The behaviour of a concurrent object can be reused or extended. Synchronization and scheduling are being viewed as aspects, and the focus is on the relationships between these abstractions within the cluster. Responsibility of an automatic weaver has been shifted to an object, the aspect moderator that would coordinate aspects and components together (Figure 3). In this framework, a proxy object controls access to the functionality class. Aspects are created using the factory method pattern. The proxy would use the aspect factory in order to create aspects, and it would also use the moderator object to evaluate the aspects for every method of the functionality class. Before invocation, the proxy would call the moderator to evaluate the aspect(s) associated with this invocation. It is believed that this approach provides the flexibility to the programmer to retain the definition of aspects by

using current programming languages. In this framework, the semantic interaction between the components and the aspects is cleanly defined. Part of the semantics is the order of activation of the aspects.

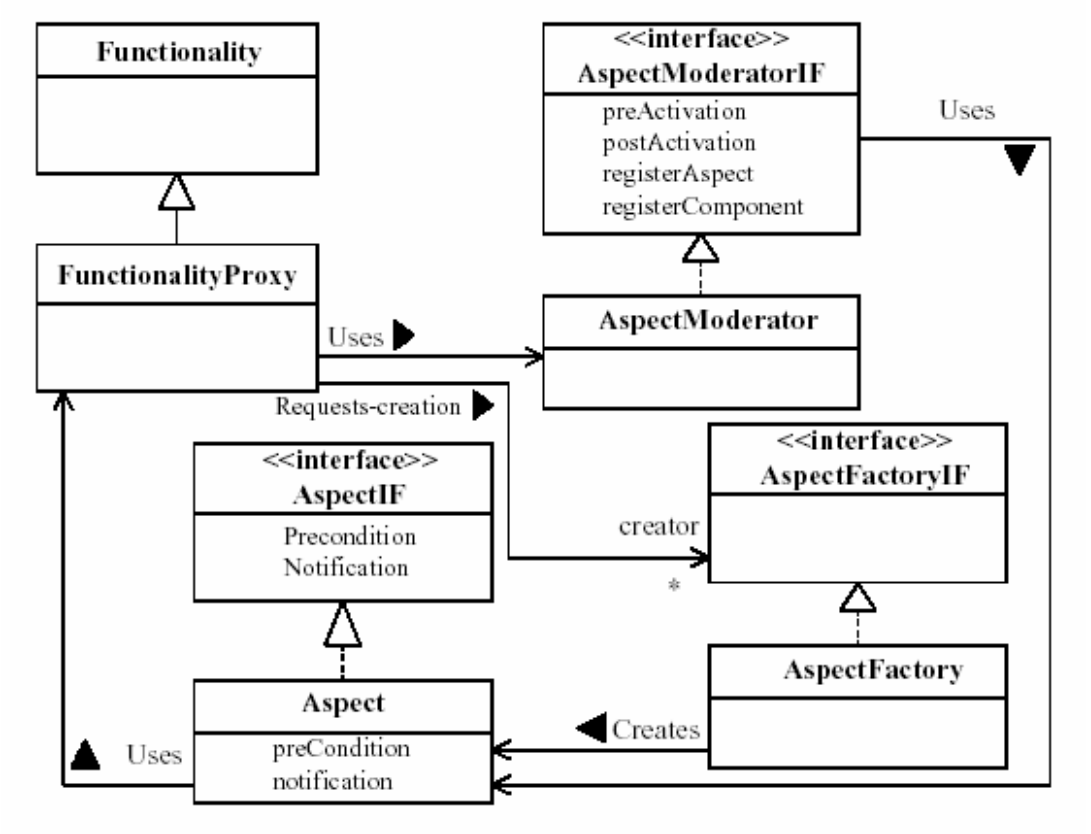

Figure 3: Aspect Moderator Framework.

A sequential object is comprised of functionality control and shared data. Access to this shared data is controlled by synchronization and scheduling abstractions. Synchronization controls enable or disable method invocations for selection. The synchronization abstraction is composed of guards and post-actions. Every method is executed within preActivation and postActivation phases, which are defined by the AspectModerator class. The preActivation will evaluate the precondition of the method's corresponding synchronization aspect. During the precondition phase, guards will validate the synchronization constraints of the invoked method, returning RESUME upon success. After successfully executing the precondition phase, the AspectModerator will return a RESUME to the FunctionalityProxy that will activate the method in the sequential object (Functionality). The completion of the method execution will initiate a call by the FunctionalityProxy to the AspectModerator's postActivation phase. During

postActivation, there is a call to the notification method of the aspect. During notification, synchronization variables are updated upon method completion. Activation order of the aspects is the most important part in order to verify the semantics of the system. Synchronization has to be verified before scheduling. A possible reverse in the order of activation may violate the semantics. There are other issues that might also be involved. If authentication is introduced to a shared object for example, it must be handled before synchronization.

Aspect-oriented Framework

Focus of this approach is that aspect-oriented software architecture (AOSA) that uses aspect-oriented frameworks could support designers and programmers in cleanly separating components and aspects from each other in different layers. AOSA can provide a mechanism that would make it possible to abstract and compose components and aspects to produce the overall system. The argument in this proposal is 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. Visualization of a three-dimensional model for system design is being suggested. By adding the aspect dimension, aspects can be captured in the design. Three dimension model for system design consists of :

- Components – basic functionality modules
- Aspects – cross-cutting entities
- Layers – components and aspects decomposed into more manageable sub-problems

Components, aspects and layers are separated from each other by separating the different aspects of each component (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 is then possible to abstract and compose them to produce the overall system. This results 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 aspects (scheduling, synchronization, naming, and fault tolerance e.g.) are scattered among many components in the system with varying policies, different mechanism, and possibly under different application. Each aspect is needed to be considered and analyzed separately to reduce the tangling of aspects in an operating system. For example, the aspect of scheduling in the file system can be considered in different domains in each layer. It would separate a 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 a specific domain. This approach can support a high degree of reusability.

Architecture of the Framework

As has been told before, the proposed aspect-oriented design framework for operating systems is an extended model of the Aspect Moderator Framework. 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.

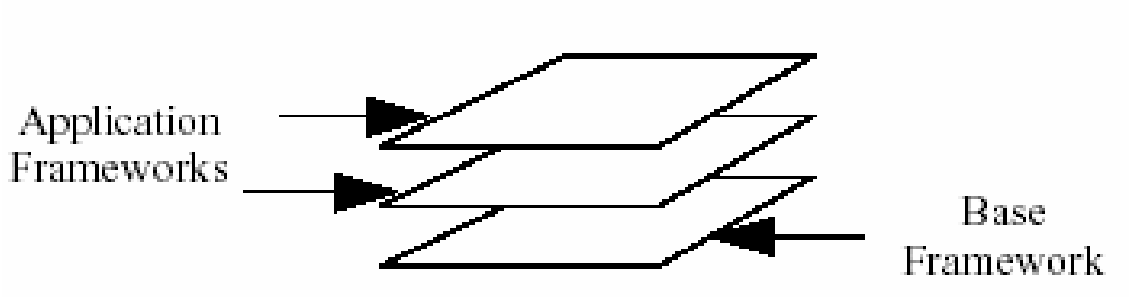

Figure 4 – Aspect-oriented Design Framework

The Base Framework corresponds to the system layer. There can be more than one application frameworks on the upper layer.

The framework uses design patterns. Aspects are created using the Abstract Factory and the Bridge patterns. The Abstract Factory isolate aspects from implementation classes. 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 smartprotection 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 aspect 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.

Execution Flow in Base Framework

When the request arrives at the system, the smart-protection proxy forwards this request to the AspectModerator object in order to identify whether this is a request to create an aspect or to invoke a method.

Initialization Phase

- Proxy forward request for aspect creation to AspectModerator object to find out if this aspect does not already exists.
- After verification proxy will call Aspectfactory to create the interface definition and the class definition of that aspect.
- Proxy will register both with AspectModerator

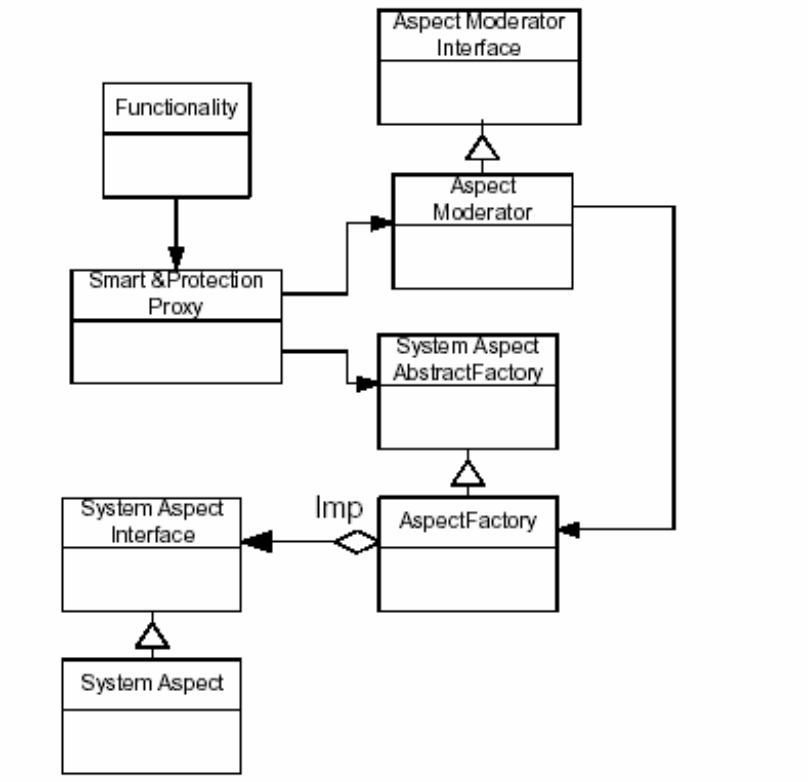

Figure 5: Base Framework

Invocation Phase

- Proxy checks whether an aspect that describes method‘s constraints is already registered with AspectModerator object
- AspectModerator will validate the constraints of the invocation method
- AspectModerator will activate the method of the aspect object and return control to the proxy.

Execution Flow in Application Framework

In the same way like in base framework when a request arrives at the proxy first it finds out if this is the request for aspect creation or method invocation.

Initialization Phase

- Proxy recognizes if request is for aspect creation or method invocation
- Checks if aspect is registered with AspectModerator and which aspects in lower layer are included in the Application layer
- If aspect not registered then call Aspectfactory to create one and register with AspectModerator

Invocation Phase

- Proxy will check register at the AspectModerator. In case of no reference it will look up the lower layer.
- In case requested aspect not registered in neither layer, error is returned
- After successful checking, the AspectModerator will validate the constraints of the method that is invoked and return control to proxy.

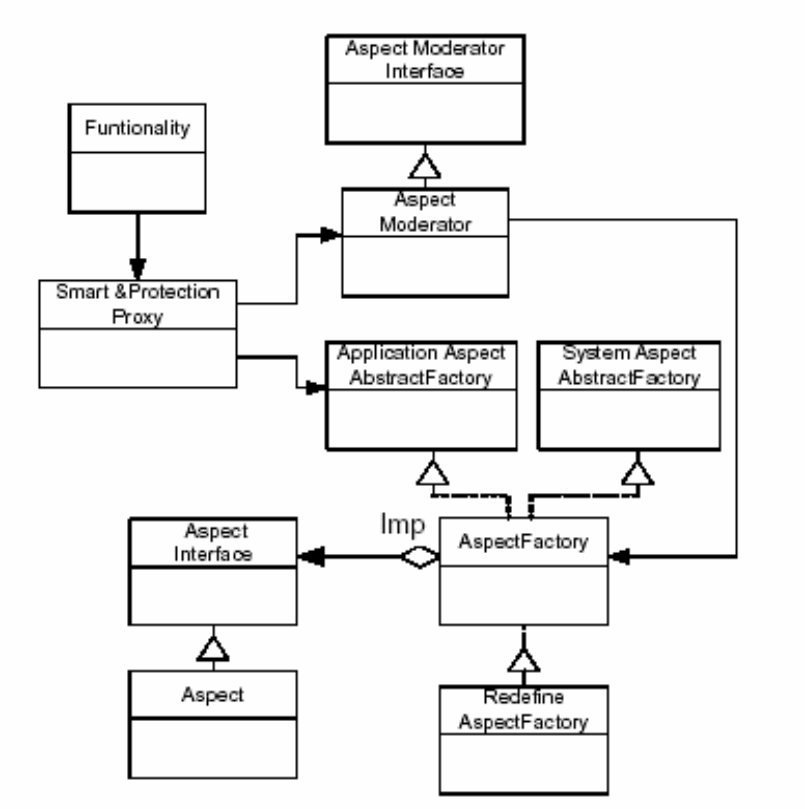

Figure 5: Application Framework

Framework Overview

The Aspect-Oriented Design Framework is a three dimensional model and it consist of a collection of aspects, and which can provide an abstraction in the operating system to support a number of components in the upper levels. Components form the main functionality of the operating system. Layers are divided into three levels

- Lower level – OS that provides reusable primitives for intermediate and upper levels
- Intermediate level – system programming or interface definition
- Upper level – application and programming level

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 other layer. The Aspect- Moderator in both frameworks need not be modified when a new aspect is introduced.

Advantages from Framework

Reusability

The upper aspects or components can use the lower aspects or components without knowing the internal details. Information hiding promotes either component or aspect modifiability and simplifies the perception of the upper level. If the implementation of lower aspect is changed to improve performance or to add new features, for example, provided the aspect interface (intermediate level) remains constant, the upper level aspect does not need to be changed.

Polymorphism

The reusability achieved in this framework not only save time but also helps avoid unnecessary proliferation of functions. It reduces problems once system becomes operational if a proven and debugged high quality of software is used in the development. Polymorphism enables us to provide a generality of aspect to handle a wide variety of policies. It also makes it easy to add new capabilities to an aspect. It helps to deal with complexity and redundancy in the system, and is particularly effective for implementing layered software systems. When fundamental aspects of the system such as scheduling, synchronization, and fault tolerance are created, these are defined as superaspects. When creating a new aspect, it is to either inherit from or override its super-aspect rather than being re-written. This new aspect is refereed to as a sub-aspect.

Reconfigurability

Aspects should be able to switch to strategies best suited to the environment or in other words they should be able to reconfigure themselves to appropriate policies for improving their performances. An aspect abstract is a super-aspect provided by the system and it can be reused and redefined by sub-aspects in the upper levels. The sole purpose of an abstract aspect is to provide an appropriate super-aspect from which other aspects could inherit, override or redefine implementation. Process management can redefine scheduling by round robin. Communication component might need FCFS. A scheduling of a file system should be capable to reconfigure to an appropriate policy for a better performance.

Summary and Conclusion

Operating system design should be based upon three dimensional model view including aspects being treated completely separately from functional components. Complete separation of concerns is at the heart of software quality. Functional components and aspects are designed relatively separately from each other. Framework provides an adaptable model that allows for open language. Interactions of newly added aspects are defined by contracts that binds a new aspect to the rest of the system rather than having to re-engineer the whole system.

References

? Netinant P., C. A. Constantinides, T. Elrad, and M. E. Fayad, Supporting the Design of Adaptable Operating Systems Using Aspect-Oriented Frameworks. Proceedings of the International Conference of Parallel and Distributed Processing Techniques and Applications (PDPTA), pp.271-278, Las Vegas, NV, June 2000.

? C. A. Constantinides, T. Elrad, and M. E. Fayad, Netinant P., Designing an aspect-oriented framework in object-oriented environment,ACM Computing surveys, March 2000