Implementing Producers/Consumers Problem Using Aspect-Oriented Framework

Paniti Netinant[1, 2] and Tzilla Elrad[2]

[1]Computer Science Department
School of Science
Bangkok University
Bangkok, Thailand
netipan@iit.edu

[2]Concurrent Programming Research Group
Computer Science Department
Illinois Institute of Technology
Chicago, IL, U.S.A.
elrad@iit.edu

ABSTRACT

For 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. Supporting separation of concerns in the design and implementation of operating systems can provide a number of benefits such as comprehension, reusability, extensibility and adaptability in both design and implementation. However, in order to maximize these benefits, such a support is difficult to accomplish. System aspectual properties define as crosscutting concerns of many components of the system. Examples of system aspectual properties are synchronization, scheduling, performance, fault tolerance and etc. Aspect-Oriented Programming 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. In this paper we demonstrate an Aspect-Oriented Framework (ACL) that can be used for system software such as operating systems. We also show how the separation of system aspectual properties from components. Producers/Consumers problem is demonstrated using our framework. Our framework, which is based on aspect-oriented technology as well as language and architecture independence, is a three-dimensional model consists of aspects, components, and layers.

Keyword— Adaptability, Aspect-Oriented Programming, Framework, Operating Systems, Reusability.

1. Issues in the Design of Operating Systems

The principle of separation of concerns lies at the heart of software development as it introduces a number of benefits, originally addressed by [8, 2]. These include better understanding, extensibility, adaptability [3] 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 components of the system. Systems are notorious of many crosscutting concerns such as synchronization, scheduling, fault tolerance, logging, and etc. We refer to these crosscutting concerns as *system aspectual properties*. Supporting separation of concerns in the system 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, and how a separation of concerns [8] will be addressed. Functional decomposition has so far been used as well as achieved along two dimensions - based on the components and layering paradigm. In 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, operating system 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 (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 reuse, make it difficult to validate the design and correctness of operating systems, and thus force reengineering of these systems either to meet new requirements or to improve the system. 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 and adapt. Changing needs to understand and correctly identify both system aspectual properties and core service implementation of the component or service. It is tightly couple design and implementation between components and system aspectual properties.

2. System Aspectual Properties in the Operating Systems

System aspectual properties are, for instances, mutual exclusion, scheduling, synchronization, fault tolerance, security, load balancing, performance measurement, testing, verifications and etc. They are all expressed in such a way that tends to cut across groups of components or services. This tangling code of system aspectual properties results increasing of code dependencies between components and properties of the system. 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). AOS aims at language and architecture independence, where components and system aspectual properties are separately decomposed in both design and implementation. These properties can be reused and adapted in the application later. Finally, components and system aspectual properties are combined together at run-time. We distinguish between components and aspects in the design of systems. System aspectual properties are defined as properties of the system that do not necessarily align with the system's components or services but tend to cut across 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 1. Inter-dependency defines as a system aspectual property that crosscuts between many components or services, as illustrated in Figure 2.

Although not bound to OOP, Aspect-Oriented Programming (AOP) [4, 5] is a paradigm proposal that retains 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 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 components are combined together, forming the overall system.

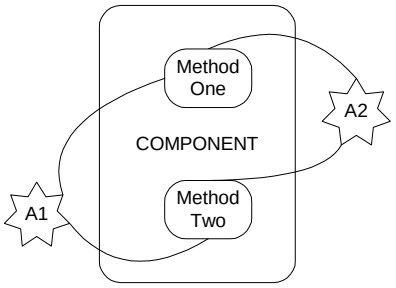

Figure 1. Intra-Dependency

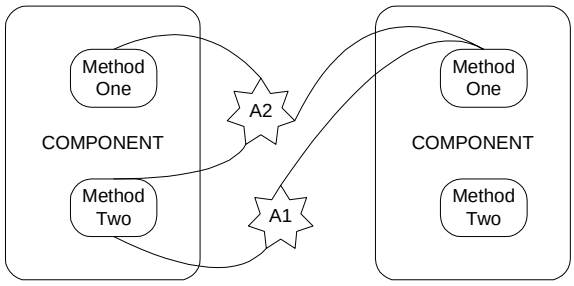

Figure 2. Inter-Dependency

In this paper we have shown system design and implementation based on system aspectual decomposition in the context of the aspectual decomposition in the design of operating systems. Our approach is an aspect-oriented framework [6, 7]. Compared with what has so far been able to be supported by traditional approaches, our goals are to provide a better design and implementation for operating systems, better flexibility, higher reusability and adaptability, as well as to provide a technique that would be practical.

3. An Aspect-Oriented Framework for Operating Systems

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 [1]. We argue that system aspectual properties of the operating system 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.

Our proposed framework (ACL) is based on system aspectual decomposition of crosscutting concerns in operating system design and implementation. ACL framework consists of two frameworks: Based Layer and Application Layer Framework. In this paper, we show how producers/consumers problem can be implement in the based layer framework. A system aspectual property is implemented in SystemAspect class, while a component of the system is implemented as Component class. Alike AspectJ [9], our framework uses *PointCut*, *Precondition*, and *Advice*. AspectModerator object, where the point cut is defined, combines both system aspectual properties and components together at run-time. Pointcut is defined collections of join points, where system aspectual properties will be altered and executed in the program flow. Every aspectual property could identify and implement precondition. Precondition is defined a set of conditions or requirements that must be hold in order to be executed an aspect. Advice is defined collections of methods for each aspectual property that should be executed at

join points. Advice could be either *before* or *after*. Before advice could be implemented as *blocking* or *non-blocking*. Before advice executes when join point is reached, before the component executed, and if the precondition is hold. After advice executes after the component at the join point executes.

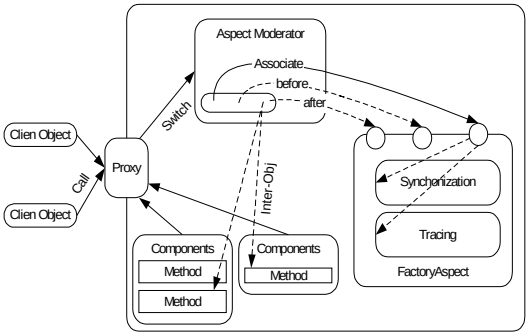

Figure 3. PointCut Defines Inter-dependency

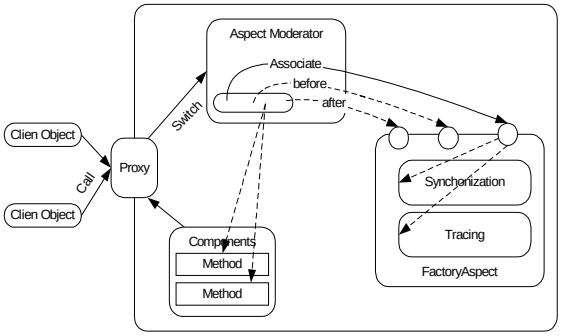

Figure 4. PointCut Defines Intra-dependency

Every aspectual property will define advice methods. Figure 3 and 4 are illustrated the execution model of a pointcut in the ACL framework based on inter-dependency and intra-dependency.

4. Example of The Producers/Consumers Problem

To show and prove our approach, we present the producers-consumers example in C++ running on Window 2000.

```
#ifndef Func H
#define Func_H

class FileBuffer
{
public:
    FileBuffer() { in = 0; out = 0;};
    int ReadFile();
    int WriteFile(const int &);

protected:

    DWORD getThreadId()
          { return m_ThreadId; };
    DWORD m_ThreadId;
        int iBuffer[20];
        int in;
        int out;
};

int FileBuffer::ReadFile()
{
        int value;

        value = iBuffer[out++];
        if (out==20) out = 0;

        return value;
}
int FileBuffer::WriteFile(const int &value)
{
        iBuffer[in++] = value;
        if (in == 20) in = 0;

        return 0;
}
#endif
```

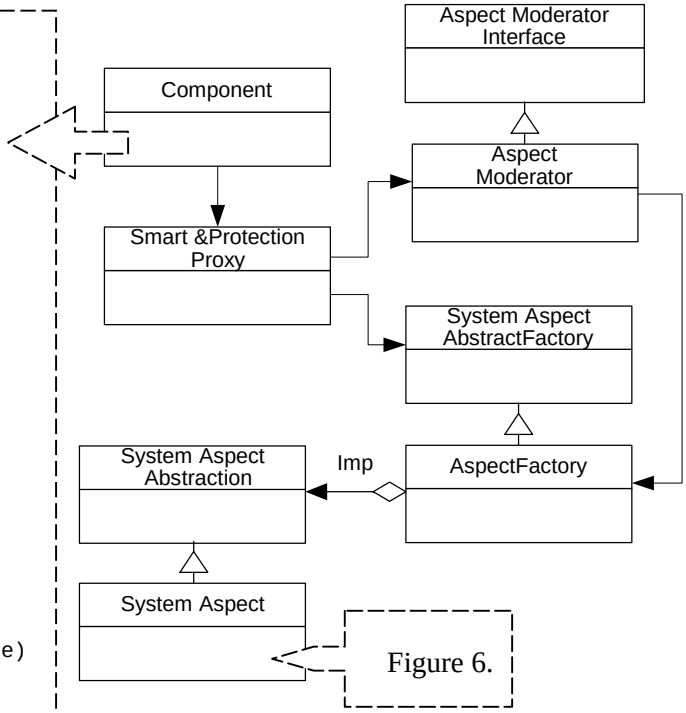

Figure 5. Base Layer Framework

The framework is promising the separation of system aspectual property, which is synchronization, from the component- FileBuffer class. The pointcut is defined the join point between FileBuffer component and Synchronization aspect in the AspectModerator class. Adding the new system aspectual property, such as tracing aspect, only requires join point of the pointcut between Tracing aspect and the component in AspectModerator to be defined.

```
SynchronizationAspectOne::SynchronizationAspectOne()
{
        ---Initial values and queues---
}

bool SynchronizationAspectOne::precondition( const char * sFuncName)
{
        if ((char *) sFuncName == "GET") {
                return ((iBuffer > 0) && (!no_put));
        }
    else
                if ((char *) sFuncName == "PUT") {
                return ((iBuffer < 20) && (!no_put));
        }
        else return FALSE;
}

void SynchronizationAspectOne::before(const char * sFuncName)
{

        BOOL  bBlock = FALSE;

        if ((char *) sFuncName == "GET") {
                switch (precondition(sFuncName)) {
                case FALSE:
                        WaitForSingleObject(cSemaphore, INFINITE);
                        WaitForSingleObject(Mutex, INFINITE);
                        break;
                case TRUE:
                        WaitForSingleObject(Mutex, INFINITE);
                        break;
                }
        }
        else if ((char *) sFuncName == "PUT") {
                        switch (precondition(sFuncName)) {
                        case FALSE:
                                WaitForSingleObject(pSemaphore, INFINITE);
                                WaitForSingleObject(Mutex, INFINITE);
                                break;
                        case TRUE:
                                WaitForSingleObject(Mutex, INFINITE);
                                no_put = TRUE;
                                break;
                        }
        }
}

void SynchronizationAspectOne::after(const char * sFuncName)
{

        if ((char *) sFuncName == "GET") {
                iBuffer--;
                ReleaseMutex(Mutex);
        }
        else if ((char *) sFuncName == "PUT") {
                        iBuffer++;
                        no_put = FALSE;
                        ReleaseSemaphore(cSemaphore, 1, NULL);
                        ReleaseSemaphore(pSemaphore, 1, NULL);
                        ReleaseMutex(Mutex);
        }
}
```

Figure 6. Implementation of Synchronization Aspect

5. Conclusion

In this paper, we stressed the importance of the better separation of concerns within the context of an Aspect-Oriented Frameworks. We discussed how this technique could provide an alternative to operating system design and implementation, and show how our approach can be achieved separation of crosscutting concerns of systems. Our work concentrates on the decomposition of system aspectual properties crosscutting components in systems and our goal is to achieve a better design and implementation of operating systems to separate the crosscutting concerns. Our design framework provides an adaptable model that allows for open languages and architectures where new aspects 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 aspects and components. The framework approach is promising, as it seems to be able to address a large number of system and application aspects and components. The advantage of decomposing of functional components and aspects in every layer is to promote reusability, adaptability, manageability, and extensibility of both components and aspects in system and application software easier without interfering each other. In the future, the framework will be extended and demonstrated for distributed object environment.

6. References

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

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

[3] Fayad, M. E., M. Cline. Aspect of Software Adaptabil-ity. Communications of ACM, Vol. 39, No. 10, pp.58-59, 1996.

[4] 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, number 1241 in Lecture Notes in Computer Science, pp.220-242, Finland, June 9-13 1997. ECCOP'97, Springer Verlag, Berlin.

[5] Lopes C., B. Tekinerdogan, W. de Meuter, and G. Kic-zales. Aspect-Oriented Programming. In M. Aksit and S.Matsuoka, editors, Proceedings of the 12th European Conference on Object-Oriented Programming EC-COP'98, Springer Verlag, 1998.

[6] 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.

[7] 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.

[8] 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.

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