Providing Dynamic Adaptability in an Aspect-Oriented Framework

Constantinos A. Constantinides*, Therapon Skotiniotis†, Tzilla Elrad‡

1 Introduction

Recent work in AOP [6] and other technologies that support advanced separation of concerns [1, 7] has proven that there are numerous advantages when cross-cutting behavior is extracted and addressed separately from the main functionality.

The requirement of systems to evolve over time and to continuously adapt to a changing environment is realized by constructing open software systems. The Aspect Moderator is a framework extension to the object model, designed to support the encapsulation of the main functionality of an object along with its aspectual behavior, while further supporting reuse and adaptability [2]. In this paper we illustrate how aspectual behavior can be plugged into the system at run-time.

2 Architecture of the Aspect-Moderator Framework

Consider a Book class (Figure2) that implements a variation of the Readers-Writers protocol where authorized clients may change the value of a string variable, or read the value of this variable. To support the requirements of the protocol, methods read() and write() have to be associated with authorization, synchronization and scheduling constraints.

2.1 Initialization

The Aspect-Moderator is an extension to the object model. A proxy to the main functional component has a number of different responsibilities. During the initialization of the framework, the proxy (Figure 4) requests from a third party, the Factory (Figure 3) the creation of authentication (Figure 7), (Figure 8), synchronization (Figure 9), (Figure 10) and scheduling aspect instances (Figure 11), (Figure 12) associated with services Read and Write of the Book class. We can identify notable characteristics in the nature and the behavior of these aspects. Authentication is evaluated once and it always yields a result which is dependent on outside parameters (the IP of the client), but independent of the current state of the system. Synchronization depends closely on the current state of the system through a re-evaluation policy. Scheduling on the other hand, although dependent on the current state of the system, it does not perform a re-evaluation of any constraints.

Aspects are designed as first-class abstractions that implement the interface defined by methods precondition() and postaction() (Figure 5), (Figure 6). This way, we are able to define a "before" and "after" aspectual behavior for every method that must be associated with an aspect (Figure 13). Aspect objects

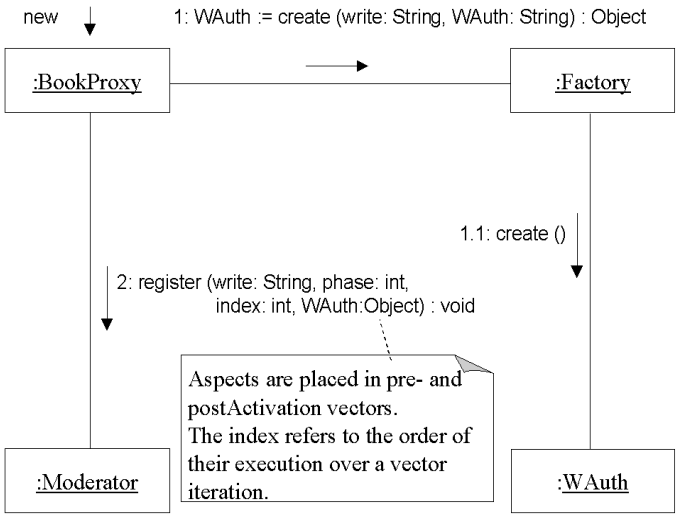

Figure 1: Aspect creation and registration.

```
public class Book {
  protected StringBuffer Message;
  Book() {Message = new StringBuffer("null");}
  public int write(String msg) {
   Message.replace(0, Message.length(), msg);
   return 1;
  }
  public String read() {return Message.toString();}
}
```

Figure 2: Implementation of Book.

```
public class Factory implements FactoryIF {

  public Object create(String methodId, String aspectId){
   if (methodId.equals("Write")
   && (aspectId.equals("Auth")))
     return new WAuth();
   // similarly for other aspect objects
  }
}
```

Figure 3: Implementation of Factory.

*Loyola University Chicago, cac@cs.luc.edu
†Illinois Institute of Technology, skotthe@charlie.iit.edu
‡Illinois Institute of Technology, elrad@iit.edu

```
public class BProxy {
  Moderator moderator;
  Factory factory;
  static Book myBook = new Book();
  BProxy (Moderator moderator,Factory factory){
    this.moderator=moderator;
    this.factory=factory;
    Object WAuth=factory.create("Write","Auth");
    // similarly for all initial aspects
    // register aspects for preactivation
    // first authorization
    moderator.register("Write","Auth",0,0,WAuth);
    moderator.register("Read""Auth",0,0,RAuth);
    // then synchronization and scheduling
    // register aspects for postactivation
    // first scheduling
    moderator.register("Write","Sched",1,0,WSched);
    moderator.register("Read","Sched",1,0,RSched);
    // then synchronization and authentication
  }
  ...
}
```

Figure 4: Implementation of BProxy, creation and registration of aspects on instanciation.

```
public abstract class Aspect {
 static int ActiveReaders = 0;
 static int ActiveWriters = 0;
 static int WaitingReaders = 0;
 static int WaitingWriters = 0;
 public abstract int precondition();
 public abstract void postaction();
 // methods to manipulate data
}
```

Figure 5: Implementation of Aspect.

are created with the deployment of the Factory Method pattern [5]. The interaction and coordination between the main functionality and aspects is monitored by a special object, the Moderator. In order for the moderator to be able to coordinate this interaction, it maintains references to the aspect objects in a number of dynamic data structures (vectors). Upon aspect creation, the proxy registers all aspect objects with the Moderator (Figure 1).

Consider the aspectual behavior of service Write. The registration of all Write-related aspects inside the moderator is crucial to support the control of the interdependencies between the three aspects. The semantics of the system dictate that authorization has to be executed first and as a result, the authentication aspect is stored in the first position of wPreActivationVector. Similarly the synchronization aspect is stored in the second position, and scheduling is stored in the third position. The registration of the Read-related aspects follows a similar manner (Figure 14). As postactivation defines the "after" aspectual behavior of both Write and Read, the order of registration of aspects inside wPostActivationVector and rPostActivationvector will be different: the scheduling aspect

```
public interface AuthIF {
 public int precondition(int ip);
 public void postaction();
 static final int MaxAuthClients = 10;
}
```

Figure 6: Implementation of AuthIF interface.

```
public class RAuth implements AuthIF {
 protected int AuthClients = 0;
 public int precondition() {return 1;}
 public int precondition(int ip) {
   if ((ip % 10) == 0){return Moderator.ABORT;}
   else if (AuthClients < MaxAuthClients) {
       AuthClients++;
       return Moderator.RESUME;
    }
   else {return Moderator.BLOCKED;}
 }
 public void postaction() {AuthClients--;}
}
```

Figure 7: Implementation of the RAuth aspect.

```
public class WAuth implements AuthIF {
 protected int AuthClients = 0;
 public int precondition() {return 1;}
 public int precondition(int ip) {
  if  ((ip % 10) > 0) {return Moderator.ABORT;}
  else if (AuthClients < MaxAuthClients) {
    AuthClients++;
    return Moderator.RESUME;
  }
  else {return Moderator.BLOCKED;}
 }
 public void postaction() {AuthClients--;}
}
```

Figure 8: Implementation of the WAuth aspect.

```
public class RSync extends Aspect {
 public int precondition() {
  if ((getAWriters() == 0) && (getWWriters() == 0)){
    decrWReaders();
    incrAReaders();
    return Moderator.RESUME;
  }
  else {return Moderator.BLOCKED;}
 }
 public void postaction() {decrAReaders();}
}
```

Figure 9: Implementation of the RSync aspect.

```
public class WSync extends Aspect {
 public int precondition() {
  if  ((getAReaders() == 0) && (getAWriters() == 0)) {
    if(getWWriters() > 0 ) decrWWriters();
    incrAWriters();
    return Moderator.RESUME;
  }
  else {
    incrWWriters();
    return Moderator.BLOCKED;}
  }
  public void postaction() {decrAWriters();}
}
```

Figure 10: Implementation of the WSync aspect.

```
public class RSched extends Aspect {
  public int precondition() {return Moderator.RESUME;}
  public void postaction() {
    if ((getWWriters() > 0) && (getAReaders() == 0))
      synchronized (Moderator.wPreActivationVector){
        Moderator.wPreActivationVector.notify();
    }
  }
}
```

Figure 11: Implementation of the RSched aspect.

will be placed in the first position, with the synchronization and authentication to follow in this particular sequence.
As the placement of aspect references during their registration defines their order of execution both during the before and after behavior, the framework provides support of non-orthogonal aspects.

2.2 Method Invocation

During method invocation (Figure 15), the proxy has a further responsibility to intercept and manipulate incoming messages, by guarding its corresponding methods within a preactivation and a postactivation phase. The result of this evaluation may cause the service to be activated, cause the caller to wait, or abort the invocation. Both preactivation and postactivation methods are implemented in the Moderator class. Consider the reception of a write message that will initially be intercepted by the proxy component which will in turn execute its preactivation phase by calling the corresponding preactivation method in the Moderator to evaluate the aspectual behavior of this service.
The Moderator will, in turn, iterate through the preactivation vector associated with service Write (wPreActivationVector). As aspects were stored in a vector structure in a semantically correct order, the Moderator is essentially oblivious about its requested evaluation as it will attempt to execute the precondition method of each aspect reference in the stored sequence. As a result, it will first evaluate the precondition() method of its authentication aspect and similarly the precondition of synchronization scheduling aspects associated with method write()

```
public class WSched extends Aspect {
 public int precondition() {
   if  (getWWriters() == 0) {return Moderator.RESUME;}
   else return Moderator.BLOCKED;
   }
 public void postaction() {
   if (getWWriters() > 0)
     synchronized (Moderator.wPreActivationVector){
       Moderator.wPreActivationVector.notify();
     }
   else if (getWReaders() > 0)
     synchronized (Moderator.rPreActivationVector){
       Moderator.rPreActivationVector.notify();
     }
 }
}
```

Figure 12: Implementation of the WSched aspect.

```
public class BProxy{
...
public int write(String msg,int ip) {
if(moderator.preActivation("Write",ip)==Moderator.RESUME){
    int result = myBook.write(msg);
    moderator.postActivation("Write");
    return result;
}
else return 0;
}
public String read(int ip) {
if(moderator.preActivation("Read",ip)==Moderator.RESUME){
    String item = myBook.read();
    moderator.postActivation("Read");
    return item;
}
else return "null";
}
 ...
}
```

Figure 13: Implementation of BProxy, enforcing preactivation and postactivation.

```
public class Moderator implements ModeratorIF {
public static Vector wPreActivationVector=new Vector(0);
public static Vector wPostActivationVector=new Vector(0);
public static Vector rPreActivationVector=new Vector(0);
public static Vector rPostActivationVector=new Vector(0);
// implementation of preActivation() and postActivation()
// implementation of register()
}
```

Figure 14: Implementation of Moderator.

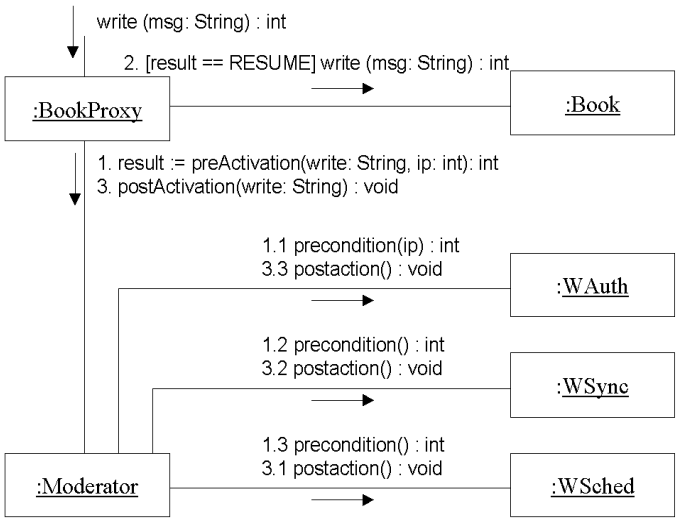

Figure 15: Method invocation.

(Figure 16). Upon successful return of the preactivation() method, the proxy will call the actual method in the main functional component. Once execution is complete, the proxy will initiate the postactivation phase for Write, by delegating responsibility to the Moderator to evaluate the postactivation() method for that service. During this phase, the Moderator will iterate through the postactivation vector for Write (wPostActivationVector) while calling the postaction() method of all registered aspects in sequence: scheduling, synchronization, and authentication (Figure 17). The class diagram of the framework is illustrated in (Figure 18)

3 Dynamic Adaptability

As requirements change and slightly different problems arise over time, a system must be able to grow and change [3]. Incremental adaptability means coping with changing requirements without modifying previously defined software components and as a result we avoid an endless cycle of re-engineering. [4]. Dynamic adaptability ensures that run-time requirements can be met without the need to re-engineer or halt the running system in order to ensure proper reconfiguration. This can be advantageous in mission-critical systems or in on-line applications.

Consider the introduction of a new requirement that dictates the keeping of track of requesting and terminating writers and readers. This logging information is kept in class Logged (Figure 19) and the logging behavior for each service (Read and Write) is captured in two new logging aspect objects (Figure 20), (Figure 21). We would like to support the fact that this new requirement will be triggered at run-time. In order to be able to do so, the proxy Figure(23) may instruct the Factory Figure(22) to create logging aspects which will be registered into the corresponding preactivation and postactivation vectors in the Moderator. As a vector is a dynamic data structure, an object element may be placed in a given position without affecting existing elements. Once aspect registration takes place, preactivation and postactivation phases will encounter a new aspect to evaluate.

```
public int preActivation(String methodId, int ip) {
 int result = this.ERROR;
 if (methodId.equals("Write")){
  synchronized (wPreActivationVector){
   for(int i=0;i<wPreActivationVector.size(); i++) {
    synchronized(this) {
     try{
       Object obj = wPreActivationVector.elementAt(i);
       Class c = obj.getClass();
       Method m = c.getMethod(
                "precondition",new Class[]{int.class});
       Integer tempInt = (Integer)
         m.invoke(obj,new Object[]{new Integer(ip)});
       result = tempInt.intValue();
     }catch (Exception exc){
      try{
        Object obj = wPreActivationVector.elementAt(i);
        Class c = obj.getClass();
        Method m = c.getMethod("precondition",null);
        Integer tempInt =(Integer)m.invoke(obj,null);
        result = tempInt.intValue();
      }catch (Exception exc2){// error handling
      }
    }
  } //synch this
  while (result == BLOCKED) {
   try {
    wPreActivationVector.wait();
     synchronized(this) {
      try{
        Object obj = wPreActivationVector.elementAt(i);
        Class c = obj.getClass();
        Method m = c.getMethod(
      "precondition",new Class[]{int.class});
        Integer tempInt=(Integer)
             m.invoke(obj,new Object[]{new Integer(ip)});
        result = tempInt.intValue();
       }catch (Exception exc){
        try{
         Object obj = wPreActivationVector.elementAt(i);
         Class c = obj.getClass();
         if (c.getName().equalsIgnoreCase("WAuth"))
          result = 1;
         else{
          Method m = c.getMethod("precondition",null);
          Integer tempInt =(Integer)m.invoke(obj,null);
          result = tempInt.intValue();
         }
        }catch (Exception exc2){// error handling
        }
      }
     }
    } catch (Exception exception) {
       return Moderator.ABORT;
    }
  }
  if (result == ABORT) return result;
 }
}
}
// similarly for read
 return result;
}
```

Figure 16: Implementation of PreActivation.

```
public void postActivation(String methodId) {
 if (methodId.equals("Write")) {
   synchronized (rPreActivationVector) {
     for(int i=0;i<wPostActivationVector.size(); i++) {
       synchronized(this) {
        try{
          Object obj=wPostActivationVector.elementAt(i);
          Class c = obj.getClass();
          Method m = c.getMethod("postaction",null);
          m.invoke(obj,null);
        }catch (Exception exc2){
            exc2.printStackTrace();
            System.out.println("Reflection Error");
        }
       }
     }
     rPreActivationVector.notify();
  }
 }
// similarly for read
}
```

Figure 17: Implementation of PostActivation.

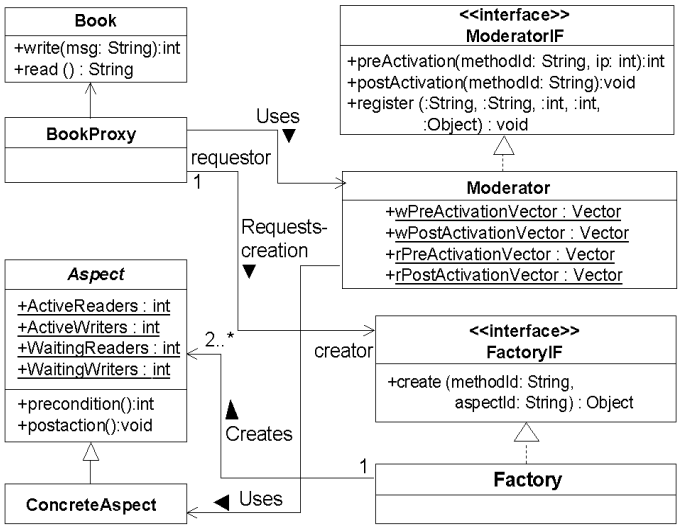

Figure 18: Class diagram of the Aspect Moderator framework.

```
public abstract class Logged extends Aspect {
 public static int RequestingWriters = 0;
 public static int RequestingReaders = 0;
 public static int TerminatingWriters = 0;
 public static int TerminatingReaders = 0;
 public int precondition() {return 1;}
 public abstract void postaction();
}
```

Figure 19: Implementation of Logged.

```
public class RLog extends Logged {
  public int precondition() {
    RequestingReaders++;
    System.out.println(
            "Requesting reads: "+RequestingReaders);
    return Moderator.RESUME;
  }
  public void postaction() {
    TerminatingReaders++;
    System.out.println(
    "Terminating reads:"+TerminatingReaders);
  }
}
```

Figure 20: Implementation of the RLog aspect.

```
public class WLog extends Logged {
  public int precondition() {
    RequestingWriters++;
    System.out.println(
            "Requesting writes: "+RequestingWriters);
    return Moderator.RESUME;
  }
  public void postaction() {
    TerminatingWriters++;
    System.out.println(
          "Terminating writes: "+TerminatingWriters);
  }
}
```

Figure 21: Implementation of the WLog aspect.

```
public class Factory implements FactoryIF {
  ...
  public Object create(String methodId,String aspectId,
  String path,String classname) {
    try{
      ClassLoader loader=
   new URLClassLoader(new URL[]{new URL(path)});
      Class c = loader.loadClass(classname);
      Object o = c.newInstance();
      return o;
    }catch(Exception exc){// error handling
    }
  }
}
```

Figure 22: Implementation of Factory, reflectively loading an aspect given its name and path.

```
public class BProxy{
  ...
public void plugandplay(String methodId,String aspectId,
                    int phase,int index,String path,
String classname){
    Object aspect =
factory.create(methodId,aspectId,path,classname);
    moderator.register(methodId, phase, index, aspect);
}
}
```

Figure 23: Implementation of PBroxy, creating and registering the new aspect.

4 Conclusion

The Aspect-Moderator is a framework extension to the object model that targets the design and development of open concurrent software systems. It supports a high level of separation of concerns that is retained during all development stages. The object's aspectual behavior is not "hard wired" to its main functionality but rather the binding of aspect objects and the main functional component is performed at run time. Aspects are created as first-class abstractions and much like functional components they can be reused and extended. The framework provides a clean separation of concerns in both design and implementation. This separation allows for high-level of reusability and adaptability. Design-level reuse is achieved through the use of design patterns and interfaces. Further, run-time adaptability is achieved by the creation and registration of aspect objects into a number of dynamic data structures maintained by the Moderator object.

This particular implementation of the Moderator class uses reflection to get the type of an object reference from inside a vector. As a result, no extra code (either in an invasive manner or through subclassing) is required in order to support the requirement of dynamic adaptability. Through the use of reflection we can further support the creation and registration of any aspect reference which abides to an interface that defines a precondition() and a postaction(). We also support the freedom for the developer to dictate the order of aspect evaluation during the preactivation and postactivation phases. This way, the Aspect Moderator framework can further support non-orthogonal aspects.

References

[1] Wakita K. Bosch J. Bergmans L. Aksit, M. and A. Yonezawa. Abstracting object interactions using composition filters. In Nierstrasz O. Guerraoui, R. and M. Riveill, editors, *ObjectBased Distributed Programming*, volume 791 of *LNCS*, pages 152–184. SpringerVerlag, 1993.

[2] C. A. Constantinides and T. Elrad. On the requirements for concurrent software architectures to support advanced separation of concerns. *Proceedings of OOPSLA 2000 Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, October 2000.

[3] M. Fayad and M. P. Cline. Aspects of software adaptability. *Communications of the ACM*, 39(10):58–59, October 1996.

[4] M. Fayad and D. Hamu. *Object-Oriented Enterprise Frameworks*. Wiley and Sons, 2000.

[5] Helm R. Johnson R. Gamma, E. and Vlissides J. *Design Patterns; Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, Inc., 1995.

[6] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *11th Europeen Conference on Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[7] William Harrison Peri Tarr, Harold Ossher and Jr. Stanley M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the International Conference on Software Engineering (ICSE 21)*, May 1999.