Dynamic Weaving for Building Reconfigurable Software Systems

JAGDISH LAKHANI
lakhjag@iit.edu
Computer Science Dept.
Illinois Institute of Technology
Chicago, IL 60616

FAISAL AKKAWI
akkawif@iit.edu
Computer Science Dept.
Illinois Institute of Technology
Chicago, IL 60616

ATEF BADER
abader@lucent.com
Lucent Technologies
Naperville, IL 60655

TZILLA ELRAD
elrad@iit.edu
Computer Science Dept.
Illinois Institute of Technology
Chicago, IL 60616

Abstract

Aspect-oriented technology is a programming paradigm that provides the user with the ability to modularize the representation of crosscutting concerns in order to maximize the reusability and ensure the flexibility of software system. In this position paper we present the Dynamic Weaver Framework (DWF), which is an aspect-oriented framework that supports the dynamic attachment of aspects to components at run-time and also the dynamic detachment of aspects from components. As well as the capability to add and remove aspects and pointcuts during runtime. This capability is the prime factor that enables us to support reconfigurability of software systems. The need to adapt to environmental changes and cope gracefully with the challenges that may have an impact on performance degradation, safety and liveness properties of the running system requires recongfigurability of both the functional and aspectual properties of the system software.

.

Keywords: Dynamic Weaver Framework, reconfigurability, dynamic adaptability.

Introduction

Aspect-oriented software design research [3,4,5,6] has stressed the need to address crosscutting concerns earlier in the design phase in order to avoid the associated code-tangling phenomenon and its undesirable implications.

Software systems go through cycles by which new requirements are introduced that may necessitate changes to their behavioral and structural properties. Some of these changes require invasive modifications. The visitor pattern [7] may reduce the effects of the invasive changes but can't eliminate them. Similarly, few of the structural and behavioral patterns [7], like decorator, adaptor, and proxy patterns, may help reduce the effect of the non-invasive changes but can't eliminate them. In [2] the authors presented the Aspect Moderator Framework, which is based on a static proxy that can provide weaving of predefined aspects at runtime, but once compiled we can't change them. DWF enables applications to adapt to changes at run time, because components and aspects are independent from each other's and they are woven at runtime. The component and the aspect in DWF must have a predefined interface, but the users are free to change the class

implementation at runtime. Also, components have no knowledge of the number and type of aspects they are affected by. So we can change the number and the type of aspects associated with a component at runtime by addPointcut() in the aspect class as shown in Figure 2. Other kinds of features, like debugging, security, and logging, that may require be activating or deactivating during runtime can benefit from reconfigurability so that different security measures can be introduced or a new Pointcut added.

Design for change in order to adapt gracefully to the evolving requirements can be farther supported when we consider both the behavioral and structural properties as core elements when addressing dynamic adaptability. While the behavioral property represents the ability to add or alter the behavior of methods in a program, the structural property represents the ability to alter class definitions and the implementation used in the program. There are a number of patterns that are documented in the software pattern literature that show how these patterns can be used to support static and dynamic adaptability of software systems, but these patterns once implemented and compiled will be difficult to alter at runtime

Dynamic Weaver Framework

Recognizing crosscutting concerns when building software systems is crucial to guarantee design and code reuse. And identifying the micro-architectural elements that constitute the skeleton of the crosscutting concerns is even more important for the design and reusability of these concerns. In Figure 1 we illustrate the micro-architectural elements of the dynamic weaver framework.

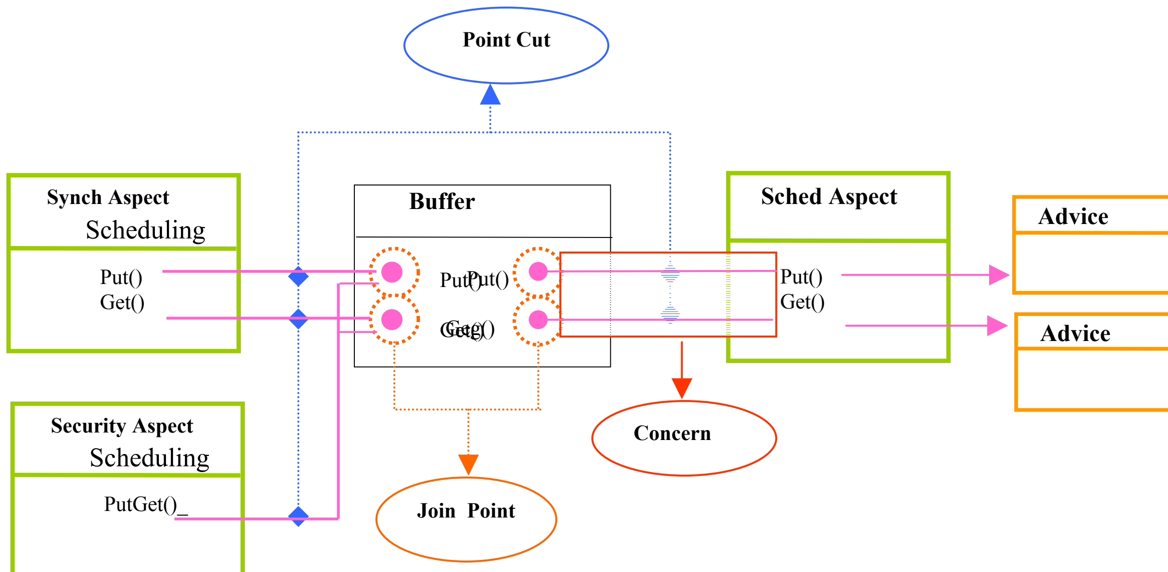

Figure 1: Architectural Elements in the Dynamic Weaver Framework.

Using behavioral pattern, proxy, to control access to the surrogated object is problematic since proxies are static, once compiled they can't change at runtime. Since in most of the cases we end up re-implementing each method in the super class or interface and add the

control access code to it, it turns out that proxies are not reusable. Ideally we wish to make aspects generic; but using a proxy for the logging aspect, for example, would require us to add it the hard way for each method call

Our DWF takes advantage of the dynamic proxy capability in Java J2SE[1]. The framework structure is depicted in the class diagram figure 2. Each class uses dynamic proxy class, which represents the aspect weaver class, from the java J2SE. The dynamic proxy class is responsible for creating that proxy object of the initiator object. And each proxy has a number of aspects, and each aspect has a number of pointcuts. The join point in our framework is the method call. Each point cut has an advice class that has two methods beforeAdvice() and afterAdvice() that will be executed when control reaches the join point methods. The semantics of aspect, pointcut, and advice are similar to the ones cited in AspectJ [8].

```
Class ReaderWriter implements ReaderWriterIF {

        Read() { ..}
        Write() { ..}
}
```

The Aspect Weaver Framework uses the DynamicProxy class that is part of the J2SE in order to weave classes and their prospective aspects at runtime. The AspectWeaver intercepts the message to the component and redirects it to the AspectRepository. The AspectRepository keeps the information about the aspect(s) (e.g. scheduling, synchronization, security…) to be applied and the order in which they have to be executed. The DWF has a loose coupling between components and aspects, because the components and the aspects no longer have direct reference between them. Let us consider an example in which we want to add the debugging concern into the bounded buffer example to show how the framework works.

First we implement the ReaderWriterIF


```
Class ReaderWriter implements ReaderWriterIF {

        Read() { ..}
        Write() { ..}
}
```

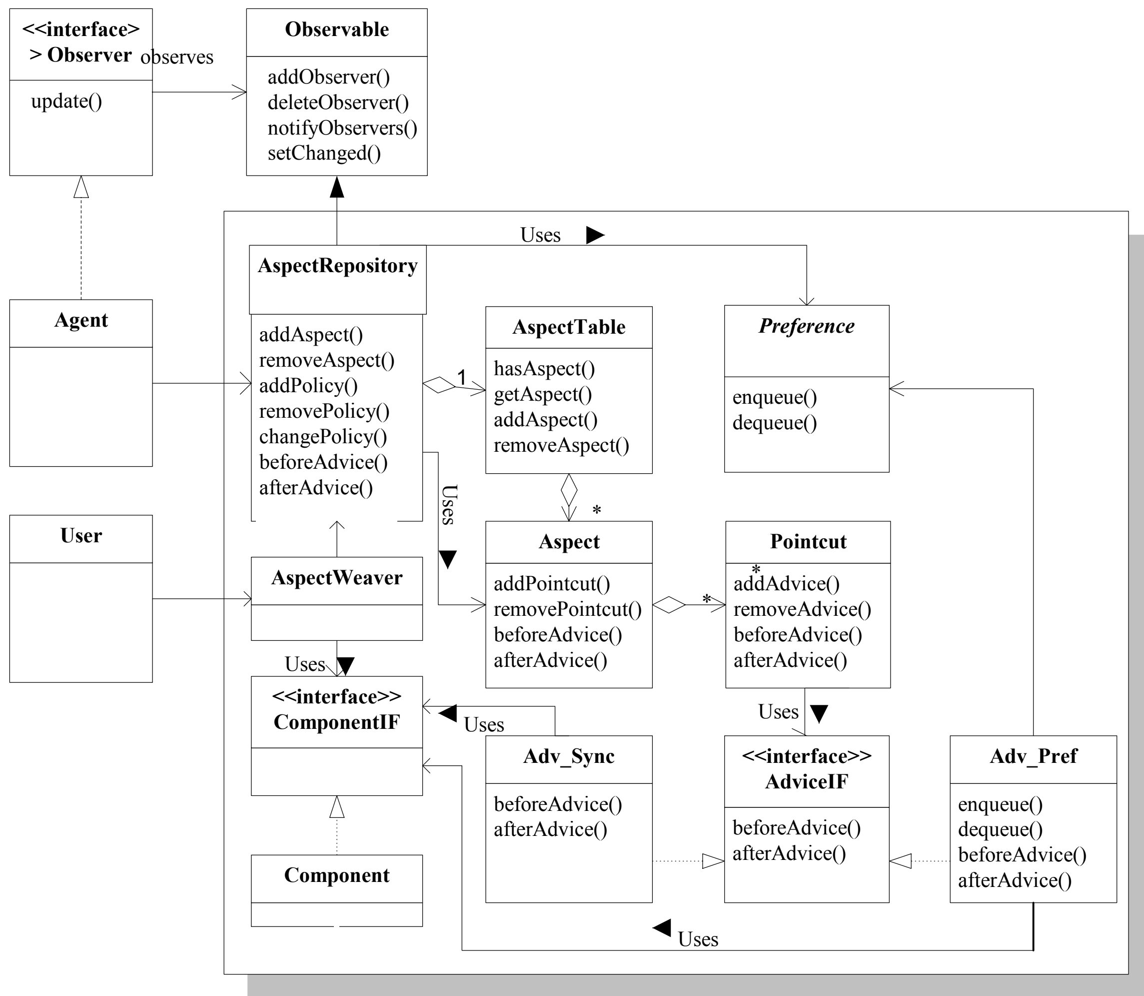

Figure 2. Dynamic Weaver Framework

Then we create the AspectWeaver. The AspectWeaver is the dynamic proxy, which directly interacts with the clients (e.g., readers and writers) and does the actual weaving of aspects at running. All the access from the clients to the bounded buffer will be accomplished through this dynamic proxy. Figure 3 shows the java code for AspectWeaver. Whenever a client calls a method of the bounded buffer, the proxy executes the invoke() method in the AspectWeaver. Inside the invoke(), AspectRepository's beforeAdvice() is called. If the call is successful, the actual operation on buffer is performed by m.invoke() as shown in the code. After this call is completed, AspectRepository's afterAdvice() method is called.

```
public class AspectWeaver implements InvocationHandler {
    private AspectRepository _ar;
    private Object _buf;

    public static Object newInstance(Object buf, AspectRepository ar) {
        return Proxy.newProxyInstance(buf.getClass().getClassLoader(),
                                      buf.getClass().getInterfaces(),
                                      new AspectWeaver(buf, ar));
    }

    private AspectWeaver(Object buf, AspectRepository ar) {
        _ar = ar;
        _buf = buf;
    }

    public Object invoke(Object proxy, Method m, Object[] args)
        throws Throwable {

        Object result = null;
        try {
                _ar.beforeAdvice(_buf, m, args);
                result = m.invoke(_buf, args);
                _ar.afterAdvice(_buf, m, args, result);
        } catch (InvocationTargetException e) {
            throw e.getTargetException();
        }
        return result;
    }
}
Figure3
```

The aspect repository class is responsible for maintaining a list of aspects that are published and registered by other classes, as shown in Figure 4.

```
public class AspectRepository extends Observable
  implements AspectRepositoryIF {
  public static final int ABORT  = 1, BLOCK  = 2, RESUME = 3;
  private AspectTable aspectTbl = new AspectTable();
  .
  .
  public synchronized void beforeAdvice(Object obj, Method m, Object[] args) {
    Task task = new Task(m.getName());
    int i, rc;
    Aspect a;
    Preference policyAdvice;
    boolean mustWait = false;

    for(i = 0; i < aspectTbl.size(); i++) {
      a = aspectTbl.getAspect(i);
      mustWait = false;
      if (occupied)
        mustWait = true;
      else {
        rc = a.beforeAdvice(obj, m, args);
        if (BLOCK == rc)
          mustWait = true;
      }
```

```
      while (mustWait) {
        policyAdvice = (Preference)
          aspectTbl.getAspect("SCHED").getPointcut("DEFAULT").getAdvice(0);
        policyAdvice.enqueue(task);
        try { this.wait(); }
        catch(InterruptedException iex) { /* Ignore iex */ }
        policyAdvice.dequeue(task);
        if (task.equals(taskToBeAwakened)) {

          mustWait = false; // Exit this while loop and
          i = -1; // Force all the aspects to be checked again.
        }
      }
    }
    .
    .
  }
.
.
}
Figur 4
```

The AspectRepository contains the AspectTable, which has all the registered aspects in it. Initially, the table is empty, i.e. no aspect is registered. Adding and removing advices of certain aspect can be done by calling the corresponding methods of the AspectTable. As shown in Figure 5.

```
public class AspectTable {
  private Vector _aspects = new Vector();

  public int size() { .. }
  public boolean hasAspect(String aspect_name) { .. }
  public Aspect getAspect(String aspect_name) { .. }
  public Aspect getAspect(int index) { .. }
  public Aspect addAspect(String aspect_name) { .. }
  public boolean addAspect(String aspect_name, String pointcut_name,
                          String method_name, AdviceIF advice) {
    .
    .
  }
  public void removeAspect(String aspect_name, String pointcut_name,
                          String method_name) {
    .
    .
  }
  .
  .
}

Figure 5
```

AspectTable can contain multiple Aspect objects. Each Aspect can contain multiple Pointcuts. The method addPointcut() is used for adding new advice. The removePointcut() method removes an advice from a given Aspect. The beforeAdvice() and afterAdvice() methods are invoked inside the AspectRepository's corresponding methods as shown in figure 6, each Aspect object has a number of Pointcut objects and a name for itself. The three arguments passed in the beforeAdvice() method are a reference to the shared object, the Method object, and the array of arguments to the specific shared object's method begin called. All these arguments are to be used inside the current beforeAdvice() method or to be passed on to the Pointcut.beforeAdvice() method. Most of the time, not all of these arguments are used. Nevertheless, they need to be passed to the beforeAdvice() method because some of the advices may utilize those information. For example, a logging advice needs to know all the data passed to the shared object. The afterAdvice() method has all the same parameters as those of beforeAdvice() method. Additionally, it has an argument defined as Object result. This represents the result of a specific shared method's call.

```
public class Aspect {
  private String _aspect_name;
  public Hashtable _pointcuts = new Hashtable();
  .
  .
  public int beforeAdvice(Object obj, Method m, Object[] args){
    if (_pointcuts.containsKey("DEFAULT")) {
      Pointcut p = (Pointcut) _pointcuts.get("DEFAULT");
      return p.beforeAdvice(obj, m, args);
    }
    return AspectRepository.RESUME;
  }

  public Object afterAdvice(Object obj, Method m, Object[] args, Object result){
    if (_pointcuts.containsKey("DEFAULT")) {
      Pointcut p = (Pointcut) _pointcuts.get("DEFAULT");
      return p.afterAdvice(obj, m, args, result);
    }
    return null;
  }
}
```

Figure 6

An Aspect can contain multiple Pointcut objects. And each Pointcut can contain multiple Advices. The method addAdvice() is used for adding a new advice. The removeAdvice() method removes an advice from a given Pointcut. The beforeAdvice() and afterAdvice() methods are invoked by the corresponding Aspect's same-named methods. Class Pointcut is shown in Figure 7. Three arguments passed in the beforeAdvice() method are a reference to the shared object, the Method object, and the array of arguments to the specific shared object's method being called. All these arguments are to be passed on to the advice's beforeAdvice() method. Four arguments passed in the afterAdvice() method will also be passed on to the advice's afterAdvice().

```
public class Pointcut {
  private String _pointcut_name;
  public Hashtable _advices = new Hashtable();
  .
  .
  public int beforeAdvice(Object obj, Method m, Object[] args){
    AdviceIF adv = (AdviceIF) _advices.get(m.getName());
    if (adv == null)
      return AspectRepository.RESUME;
    return adv.beforeAdvice(obj, m, args);
  }

  public Object afterAdvice(Object obj, Method m, Object[] args, Object result){
    AdviceIF adv = (AdviceIF) _advices.get(m.getName());
    if (adv != null)
      return adv.afterAdvice(obj, m, args, result);
    return null;
  }
}
```

Figure7

Actual behavior of each aspect is provided by an object whose interface is defined by AdviceIF. They will be woven at runtime by the dynamic proxy, i.e. AspectWeaver. The interface for Advice is shown in Figure 8.

```
public interface AdviceIF {
    public int beforeAdvice(Object obj, Method m, Object[] args);
    public Object afterAdvice(Object obj, Method m, Object[] args, Object result);
}
```

Figure 8

Shown below is the implementation of the synchronization advice for the bounded buffer's read() method. The beforeAdvice() method first checks for the number of writers in the system and appropriately it returns one of the integer constants defined in the AspectRepository; RESUME, BLOCK, and ABORT.

```
public class Adv_SyncRead implements AdviceIF {
  public int beforeAdvice(Object obj, Method m, Object[] args) {
      if (numW>0)
      {
          System.out.println(Thread.currentThread().getName() + " gets blocked.");
           return AspectRepository.BLOCK;
      }
      else
      {
          System.out.println(Thread.currentThread().getName() + " gets dispatched.");
           return AspectRepository.RESUME;
      }
  }

  public Object afterAdvice(Object obj, Method m, Object[] args, Object result) {
    return null;
  }
}
```

Dynamic Weaving and Reconfigurability

The Dynamic Weaver Framework has a number of advantages; it provides the capability to add and remove aspects as well as pointcuts at runtime. This capability is the prime factor that enables us to support reconfigurability in order to adapt to environment changes and cope gracefully with any challenges that may have an impact on performance degradation and safety and liveness properties of the running system. Changes to the software systems may affect the structural or behavioral properties. Although the Visitor pattern [7] may reduce the severity of the invasive changes and the Decorator and Proxy pattern may support the engineering of the non-invasive changes, these patterns offer very little to support the design and implementation of the crosscutting concerns, especially in the cases where we desire to alter, add, or remove these crosscutting concerns at run time from the running system. The DWF represents the solution for all of these problems that are hard to anticipate fully during the design phase.

For instance, we may need to add a new pointcut or add a join point to a list of methods that comprise the point cut. An example of this is the addition of a new method *gget()* to the pointcut of the synchronization aspect, or altering the advice class for a certain joint point, like changing the scheduling point cut for the method get, to service requests based on the priority of the thread.

Conclusion

In this paper we presented an approach by which aspects and components can be weaved, altered or removed dynamically. This approach is a step toward automating the weaving process at runtime. Recognizing the micro-architectural elements of crosscutting concerns during the design phase is essential to ensure the reusability and reconfigurability of the resulting software system, and the DWF is an attempt to meet these desirable properties when crafting software systems.

References

[1] http://ww.java.sun.com/jdk1.3.

[2] Constantinos Constantinides, Atef Bader, and Tzilla Elrad. A Framework to Address a Two-Dimensional Composition of Concerns. Position paper to the OOPSLA ’99 First Workshop on Multi-Dimensional Separation of Concerns in Object-Oriented Systems.

[3] Gregor Kiczales, John Lamping,Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-MarcLoingtier, and John Irwin. Aspect-Oriented Programming. In *Proceedings of ECOOP ’97*. LNCS 1241.Springer-Verlag, pp. 220-242. 1997.

[4] Bedir Tekinerdogan and Mehmet Aksit. *Deriving Design Aspects from CanonicalModels*. Position paper in ECOOP ’97 workshop on As-pect- Oriented Programming.

[5] Harold Ossher and Peri Tarr. Multi-Dimensional Separation of Concerns in Hyperspace.Position paper at the ECOOP '99 workshop on Aspect-Oriented Programming.

[6] Robert E. Filman and Daniel P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In Workshop on Advance Separation of Concerns, OOPALA, Minneapoils, USA, 2000.

[7] E. Gamma, R. Helm, R. Johnson & J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley, 1994.

[8] The aspectj web site. Http://www/aspectj.org/.