

Dynamic Weaving for Building Reconfigurable Software Systems

FAISAL AKKAWI
Akkawi@cs.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@charlie.cns.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 reusability and ensure flexibility of the software system. In this position paper we present the dynamic Weaver Framework (DWF), which is an aspect-oriented framework that supports the dynamic attachment and detachment of aspects to components at run-time, 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 the software system. 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 reconfigurability of both the functional and aspectual properties of the system software.

Keywords: Dynamic Weaving Framework, reconfigurability, dynamic Adaptability.

Introduction

Aspect-oriented software design [3,4,5,6] research 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] we presented the aspect Moderator Framework (AMF) 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 application to adapt to changes at run time, because components and aspects are

independent from each other's and they are weaved 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 about the number and type of aspects they are effected 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.

Another kind of features, like debugging, security, and logging, that may require to be activated or deactivated during runtime can benefit from reconfigrability where different security measures need to be introduced or new pointcut added.

Design for change in order to adapt gracefully to the evolving requirements can be farther supported when we address both the behavioral and structural properties as being the 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 definition and implementation used in the program. There are a number of patterns that are documented in the software pattern literature that show how to a certain degree these patterns can be used in order to support static and dynamic adaptability of the software system, but these patterns once implemented and compiled will be difficult to alter at runtime

Dynamic Weaver Framework

Recognizing crosscutting concerns when building software system 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 the figure 1 we illustrate the micro-architectural elements in the dynamic weaver framework.

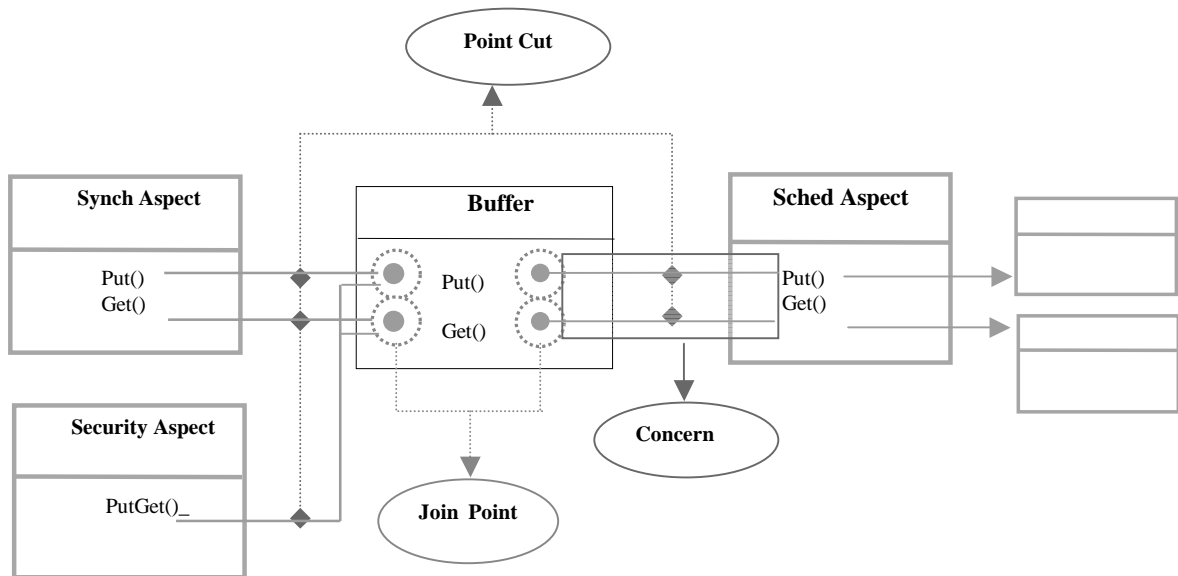


Figure 1: architectural elements in the dynamic weaver framework.

Using behavioral pattern, proxy, to control access to the surrogated object is problematic since proxies are static, once compiled we can't change them during 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 point cuts. 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 points methods. The semantics of aspect, point cut, and advice are similar to the ones sited in AspectJ [8].

```
Class Buffer implements BufferIF {  
  
    Put() { ..}  
    Get() { ..}  
}
```

The Aspect Weaver Framework uses the DynamicProxy class that is part of the J2SE in order to weave classes and their perspective aspects at runtime. The AspectWeaver intercept the message to the component and redirect it to the AspectRepository. AspectRepository keeps the information about the aspect(s) (e.g. scheduling, synchronization, security...) to apply and the order in which they have to be executed. And in order to isolate the aspects creation and destruction from the actual evaluation, we deploy the strategy design pattern that allows us to add the hooks for these aspects to the aspect repository. The DWF has a loose coupling between component and aspects, because component and aspect do not have direct reference between them. Let us consider an example by which we like to add the debugging concern into the bounded buffer example to show how the framework works.

First we implement the BufferIF

```
Class Buffer implements BufferIF {  
  
    Put() { ..}  
    Get() { ..}  
}
```

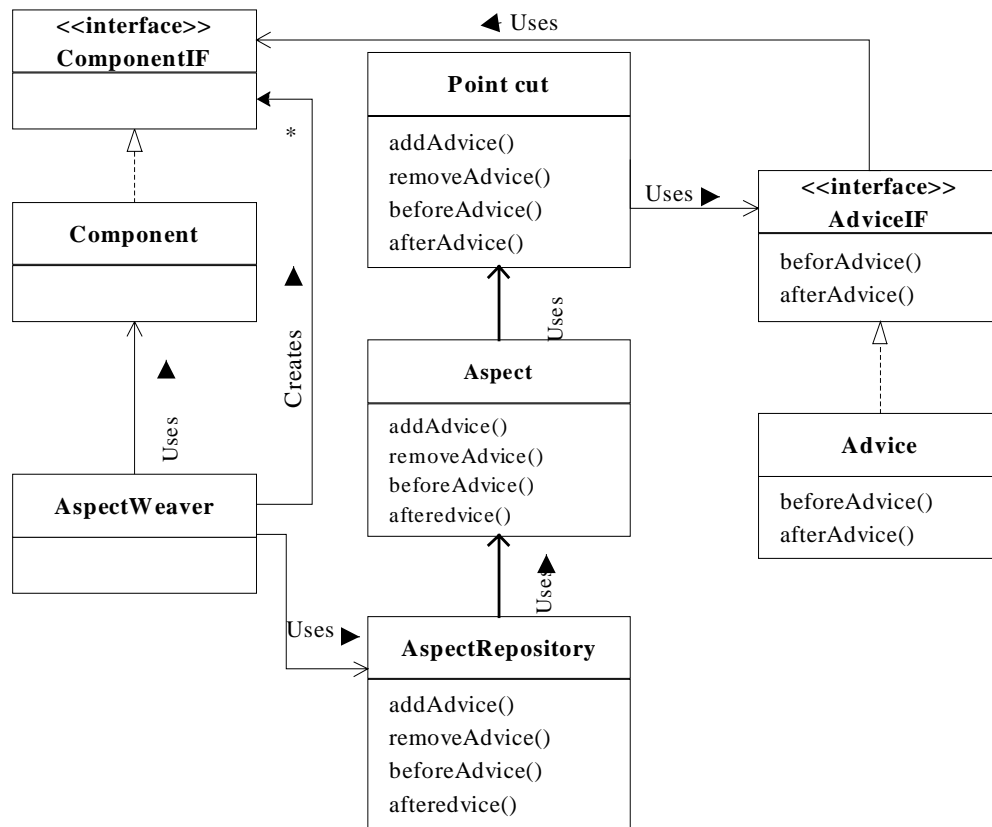


Figure 2. Dynamic Weaver Framework

Then we create the BufferDebugProxy

```

public class BufferDebugProxy implements java.lang.reflect.InvocationHandler {

    private AspectRepository aspect_repository;
    private Object obj;

    public static Object newInstance(Object obj, AspectRepository aspect_repository) {
        return java.lang.reflect.Proxy.newProxyInstance(
            obj.getClass().getClassLoader(),
            obj.getClass().getInterfaces(),
            new BufferDebugProxy(obj, aspect_repository));
    }

    private BufferDebugProxy(Object obj, AspectRepository aspect_repository) {
        this.aspect_repository = aspect_repository;
        this.obj = obj;
    }

    public Object invoke(Object proxy, Method m, Object[] args)
        throws Throwable
    {
        Object result;
        try {
            aspect_repository.abefore_dvice (obj, m);
        }
    }
}
  
```

```

        result = m.invoke(obj, args);
    } catch (InvocationTargetException e) {
        throw e.getTargetException();
    } catch (Exception e) {
        throw new RuntimeException("unexpected invocation exception: " +
            e.getMessage());
    } finally {
        aspect_repository.after_advice(obj, m);
    }
    return result;
}
}

```

The aspect repository class is responsible to maintain a list of aspects that are published and registered by other classes.

```

class Aspect_Repository implements Aspect_RepositoryIF{
private Aspect aspects[];

void before_advice (Object obj, Method m)
{
    for(int i=0; aspects.length; i++)
        aspect.before_advice(obj, m)
}
...
}

```

```

class Aspect implements AspectIF {

pointcut pointcut[];

void before_advice (Object obj, Method m){
    pointcut[m].before_advice(object)
}
...
}

```

```

class Pointcut implements PointcutIF {

Advice advice[];

void before_advice (Object obj, Method m){
    advice[m].before_advice(obj, m)
}
...
}

```

```

class Advice implements AdviceIF {

void before_advice (Object obj, Method m){
    System.out.println("before method " + m.getName());
}
...
}

```

Dynamic Weaving and Reconfigurability

Aspect weaver framework has a number of advantages; it provides the capability to add and remove aspects as well as pointcuts during runtime. This capability is the prime factor that enables us to support reconfigurability in order to adapt to environment changes and cope gracefully with the challenges that may have an impact on performance degradation and safety and liveness property 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 require adding a new point cut or add join point to a list of methods that comprise the point cut. An example of this is the addition of new method *gget()* to the point cut 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 woven, altered or removed dynamically. This approach is a step forward to automate 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 the software systems.

References

- [1] <http://www.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-Marc Loingtier, 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 Canonical Models*. Position paper in ECOOP '97 workshop on Aspect-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, Minneapolis, 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. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm & W. Griswold, *An Overview of AspectJ*, whitepaper at <http://aspectj.org>, 2001.