Implementing Connectors for Adaptive Plug-and-Play Components with Java Inner Classes

Toby Elliott

The benefits of Adaptive Plug-and-Play Components are described in some detail by Mezini and Lieberherr [1]. In short, these components allow slices of behavior to be spread across an object model while keeping their code separate from the base classes that they are providing extra functionality to. This reduces the tangling of code and allows for easier maintenance of projects, especially ones with repetitive functions that cut across many classes in the application.

While the general approach to APPCs is well documented in their paper and variants on the theme are described in others, a specific, clean approach is missing. This paper describes one particular approach, using inner classes to unite the components with the objects they are to act upon. This is done by pre-compiling a “connector” through which calls to the class methods are executed.

Several factors affected the approach taken in this design. First and foremost was the idea that the connector should be doing all the work, and that a minimum of changes should be necessary in the component classes and no changes should be necessary in the base object classes. We cannot assume that we have access to the code in the original class (or even the component), and thus, their additional requirements should be minimized. Additionally, this approach allows us to make use of legacy Java code.

Secondly, this approach was designed to be flexible, at the cost of some speed enhancements. This was a conscious choice, and represents a common tradeoff.

Thirdly, the focus was placed on simplicity. By having a precompilation step, the emphasis is on generating most of the complex java code in the precompiler. This allows for fast generation of connectors and easy changes.

Finally, this approach is done in 100% Java. It requires no libraries beyond the standard ones, and has no dependencies on specialized code. It also does not manipulate the code of the component or application during the precompilation stage - it only links the two together.

Variables needed for Component/Connector Design

Let $M_{(i=1..x)}$ be the methods in the original class that are to have components wrapped around them. At this point the precompiler should have expanded all wildcards, so these methods may be stored in an array.

Let $P_{(j=1..y)}$ represent each operational method within the component. Note that Method(Object a) and Method (Object a, Object b) my both have the same name (for mapping purposes, but are treated as different methods within the component.

Let $C_{(k=1..z)}$ be the names of all the components to be joined to the application class.

Let *Depth* be the number of components between a component and the application class (this is a constant for each component).

With these variables, it is possible to build the connector, though knowing P is also helpful in creating the component. Note that there is a 0..1 relationship between M_i and P_j, and a 0..* relationship between P_j and M_i .

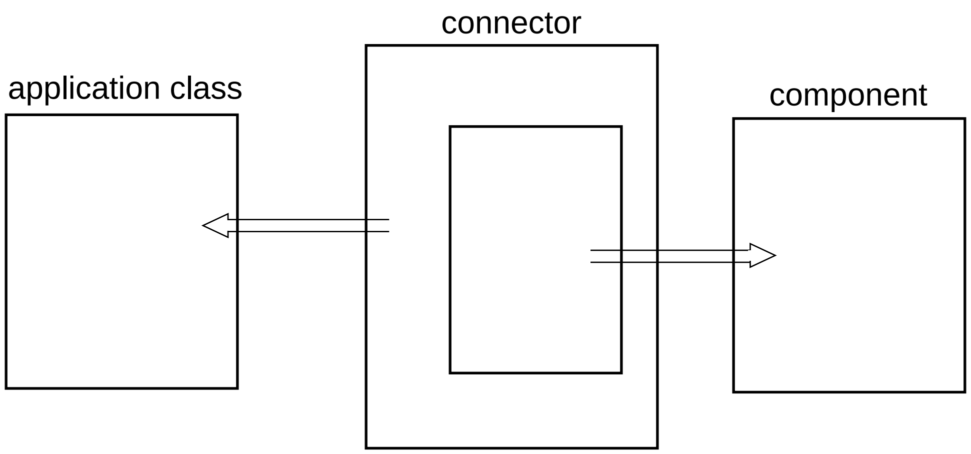

Figure 1: Inheritance diagram for the connector, showing how inner classes are used to join the appropriate application methods with the wrapping component

Connector model

Figure 1 shows the approach taken towards implementing the connector classes. By using the inner class (a concrete extension of the abstract component) within the conncteor class (itself an extension of the base class, or the next connector in the sequence), methods in the connector class have access to methods in the component. The abstracted sequence of calls relative to the outer and inner classes is shown in Figure 2.

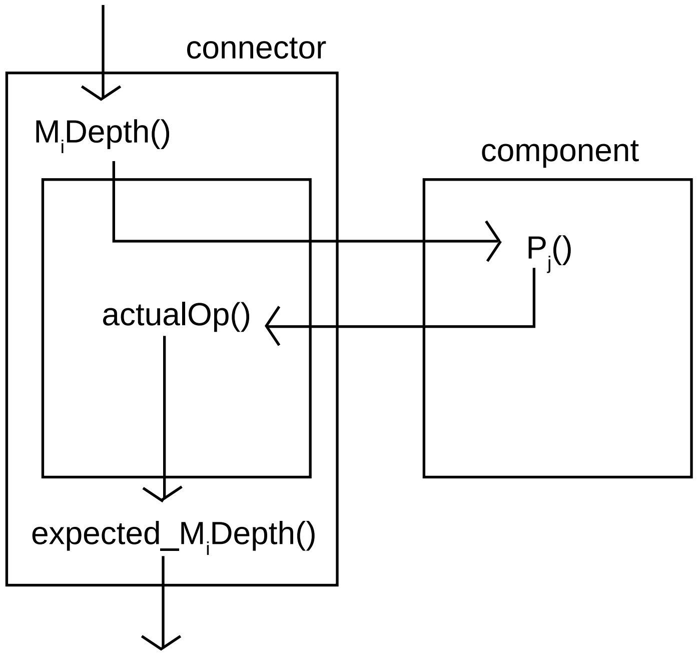

Figure 2: Method progression through the connector and component classes.

Parameters and Return Types

One of the frequent limitations of APPCs is that they bound the allowable mappings from a class method to a single 1-to-1 mapping with a component method. This seems a needless limitation: if you are locking access to a database while you perform the code in a method, you should not be limited to only methods of the same structure. The inner class approach allows us to break out of this problem by mapping the operations to the appropriate identically named component method. Thus, push* will not have a problem with both push(Object a) and push(Object a, Object b). The component also provides the flexibility to handle these methods slightly differently if it is so desired.

The Precompilation Instructions

The specific syntax for the instructions to the precompiler can be left up to the implementor. It should contain the following:

1) A list of all components to be added to the structure, in a way that allows these components to be parsed in a specific and consistent order. This is essential so as to avoid side effects that might otherwise occur in the chaining process.

2) For each component, a map of the component methods to the methods to be extended in the original class. Wildcard characters may be used for many-to-one method manipulations. These mappings will be used to make sure that the appropriate methods are wrapped with the components.

3) A participant may also contain overridden methods, or methods defined for that participant (often referred to as "expected methods". While these are not covered in this paper, any such methods can be dropped into the inner class of the component without modification.

A sample precompilation instruction file is included below.

```
LockComponent -> CountComponent -> OriginalClass

LockComponent
{
    push* -> pushOp;
    pop* -> popOp;
}
CountComponent
{
    push* -> pushOp;
    pop* -> popOp;
}
```

Structure of the component

Components should be public and abstract. An abstract method actualOp should be defined with the same parameters and return type as the original method for each P_j. Within the component methods, when it comes time to call the original method, actualOp (which has been defined in the connector class) is called. See Code1 for an example.

For each operation to be performed by the component, a method corresponding to each permutation of return type and parameters that may use that particular mapped method must be defined. These methods should be concrete and contain the code to be executed (including the actualOp call described above. The class may contain any shared variables that may be accessed from all methods in the class, as well as those defined locally to a method.

```
public abstract void actualOp(Object passed);
public void pushOp(Object passed)
{
        count++;
        System.out.println(count);
        actualOp(passed);
}
```

Code 1: Sample Coomponent Method

One advantage to this approach to APPCs rather than a more straightforward use of "before" and "after" methods is that it allows for operations that wrap themselves around the code. For example

```
public void ComponentMethod(Object passed)
{
    if (condition)
    {
        actualOp(passed);
    }
}
```

Building the Connector

The connector is entirely built during the precompilation stage. Connectors cannot be built in separate sessions, as they rely on knowing the order in which they are chained in order to keep them in sequence, rather than in parallel.

The first stage is to define the class relationships. The format is:

```
public class AComponent extends BComponent
```

where A Component and BComponent are sequential in the list of all components to be added to the structure. This ensures that super.*() calls will eventually bubble up to the original method while traversing the components in the correct order.

The final line in the connector method (excluding the inner class) is just an instantiation of the inner class:

```
public InnerClassName c = new InnerClassName();
```

Each entry M_i is given two methods in the connector:

public $M_{i\text{-return-type}}$ M_iDepth ($M_{i\text{-parameters}}$) { [return] c.setInner(c.inner i).P_j($M_{i\text{-parameters}}$);}
public $M_{i\text{-return-type}}$ expected_M_iDepth ($M_{i\text{-parameters}}$) { [return] super. M_i(Depth +1)($M_{i\text{-parameters}}$); }

Note that the returns are not necessary if the return type of M_i is void. A sample is shown in Code 2.

```
public void pushOn2(Object passed) { c.setInner(c.inner1).pushOp(passed); }
public void expected_pushOn2(Object passed) { super.pushOn(passed); }
```

Code 2: Sample Direction Methods in the Connector

The first method is the entry point into the class to be called by the method (thus, the first one can be named as the original expected method) or by the super.* call in an extending class. Most of the work is done in this method as it moves the correct actualOp method into position, then calls the component method associated with M_i from within the inner class.

The second method is the exit method for the class, passing control up to the superclass. Appending the depth to the method name is necessary to avoid namespace conflicts. If a method does not have that indication, it overrides its own superclass method and ends up calling itself in an infinite loop.

The Inner Class

Each connector class has an inner class which allows access to the component methods associated with that connection. The classes can be lengthy, but follow the same pattern.

The class declaration extends the abstract connector class:

```
public class InnerClassName extends ComponentPackage.ComponentName
```

There is only one method to an inner class, structured as follows:

```
public InnerClassName setInner(Inner inner)
{
    this.inner = inner;
    return this;
}
```

This inner class contains several classes of its own. The first part of the inner class is the interface (Inner) to the remaining classes within the inner class. This interface contains the method declarations for all actualOps corresponding to P (see Code 3 for a sample interface. It should be observed that these interface methods should have a 1-to-1 correspondence with the actualOps declared in the component.

```
interface Inner
{
        public void actualOp(Object passed);
        public void actualOp(Object passed, Object passed2);
        public Object actualOp();
}
```

Code 3: Sample interface inside an inner class

The need of these “inner-inner” classes derives from the desire to support different return types and parameters associated with the same component object (though they are technically different objects in the component, they are mapped as a group and may be regarded as such). By not knowing what type or parameter is associated with an operation, we are forced to delay this decision until run-time, when an appropriate decision can be made. However, in the interim, we are required to implement all possibilities, which is done through these inner classes.

For each method M_i, an implemeting class is declared (Inner*i*). Obviously, only one of the actualOp methods will be called by M_i, but since the rest are abstract, they all must be declared. Thus, the actualOp that corresponds to the actualOp being called by the component method is defined and the remainder are set to {} or {return null;}

The body of the one non-empty operation is :

[return] OuterClassName.this.expected_ $M_i(M_{i\text{-parameters}})$;

Finally, an instance of each of these classes is generated:

```
public Inner inneri = new Inneri();
```

Putting It All Together: Tracing a Sequence Through a Connector.

1) M_i*Depth* is called from outside the class

2) c.Inner is set to Inner*i* and the component method is invoked.

3) P_j (the component method associated with the method M_i) is invoked through c.

4) Control passes to the component that executes its code. At some point, it invokes actualOp

5) actualOp invokes expected_M_i*Depth*

6) M_i*Depth* passes control to the superclass.

Limitations and Potential Future Extensions.

Not defined in the sequence above, though easy to implement, is the concept that certain methods may not need to trace through all components. For example, while you might want to log all method calls, you may only want to lock a database in certain situations. The best approach to dealing with this situation involves defining all methods in each connector class, but having ones that do not use the particular component simply call their superclass directly, e.g.

```
public void pushOn2(Object passed) { super.pushOn3(passed); }
```

Clearly, no inner-inner class needs to be defined to hold the actualOp for this method, nor is an expected_ method needed.

Another consideration when using this approach is that methods are dynamically allocated in the inner class during run-time. As a result, in a multithreaded environment, race conditions do exist. It should be possible to use semaphores to delay activation of a new method until the old one has been invoked, but this may incur a performance penalty.

It is worth noting that each inner-inner class requires a further Java .class file that must be defined. Since there are *i* inner-inner classes and *k* inner classes, *k*i* .class files are generated.

Finally, one major obstacle, not unique to this approach to APPCs, is that method calls in the original class must be redirected through the component. Obviously, this is a one-time only change, but it is not always the case that access to the original code is possible.

Conclusions

The inner class approach represents a flexible and simple approach to defining Aspectual Plug-and-Play components. Use of a precompiler ensures that most of the complicated programming takes place “behind the scenes”, leaving the developer free to work on the methods and components, where the true effects of the program are generated.

Appendix

Full code for a sample set of connectors is given below. It assumes the presence of a stack module that supports the operations pushOn, pushTwice, pushTwo and popOff. The sample connector precompiler instructions is the same as the one given in the example.

```
package CountAPPC;

public abstract class CountComponent
{
	int count = 0;

	// classes to be defined by the extending inner class
	public abstract Object get_host();

	public abstract void actualOp(Object passed);
	public abstract void actualOp(Object passed, Object passed2);
	public abstract Object actualOp();

	// real component methods
	public void pushOp(Object passed)
	{
		count++;
		System.out.println("current count: " + count);
		actualOp(passed);
	}
	public void pushOp(Object passed, Object passed2)
	{
		count++;
		System.out.println("current count: " + count);
		actualOp(passed, passed2);
	}
	public Object popOp()
	{
		count--;
		System.out.println("current count: " + count);
		return actualOp();
	}

}
```

```
package LockAPPC;

public abstract class LockComponent
{
	boolean lock;

	// classes to be defined by the extending inner class
	public abstract Object get_host();

	public abstract void actualOp(Object passed);
	public abstract void actualOp(Object passed, Object passed2);
	public abstract Object actualOp();

	// real component methods
	public void pushOp(Object passed)
	{
		if (!lock)
		{
			System.out.println("Locking");
			actualOp(passed);
			System.out.println("Unlocking");
		}
	}
	public void pushOp(Object passed, Object passed2)
	{
		if (!lock)
		{
			System.out.println("Locking");
			actualOp(passed, passed2);
			System.out.println("Unlocking");
		}
	}
	public Object popOp()
	{
		if (!lock)
		{
			System.out.println("Locking");
			Object pop = actualOp();
			System.out.println("Unlocking");
			return pop;
		}
		return null;
	}

}
public class CountConnector extends  LockConnector
{
	//inner class to glom on the componenet.
	public class CountBase extends CountAPPC.CountComponent
	{
		public Inner inner;
		public Inner inner1 = new Inner1();
		public Inner inner2 = new Inner2();
		public Inner inner3 = new Inner3();
		public Inner inner4 = new Inner4();

		public Object get_host() { return CountConnector.this; }

		public void actualOp(Object passed) { inner.actualOp(passed); }
		public void actualOp(Object passed, Object passed2)
				{inner.actualOp(passed, passed2); }
		public Object actualOp() { return inner.actualOp(); }

		interface Inner
		{
			public void actualOp(Object passed);
			public void actualOp(Object passed, Object passed2);
			public Object actualOp();
		}
		public class Inner1 implements Inner
		{
			public void actualOp(Object passed)
			{
				CountConnector.this.expected_pushOn2(passed);
			}
			public void actualOp(Object passed, Object passed2){}
			public Object actualOp()
			{
				return null;
			}
		}
```

```
        public class Inner2 implements Inner
        {
            public void actualOp(Object passed)
            {
                CountConnector.this.expected_pushTwice2(passed);
            }
            public void actualOp(Object passed, Object passed2){}
            public Object actualOp() { return null; }
        }
        public class Inner3 implements Inner
        {
            public void actualOp(Object passed){}
            public void actualOp(Object passed, Object passed2){}
            public Object actualOp()
            {
                return CountConnector.this.expected_popOff2();
            }
        }
        public class Inner4 implements Inner
        {
            public void actualOp(Object passed){}
            public void actualOp(Object passed, Object passed2)
            {
                CountConnector.this.expected_pushTwo2(passed, passed2);
            }
            public Object actualOp() { return null; }
        }

        public CountBase setInner(Inner inner)
        {
            this.inner = inner;
            return this;
        }
    }

    public CountBase c = new CountBase();

    public void expected_pushOn2(Object passed) { super.pushOn(passed); }
    public void expected_pushTwice2(Object passed) { super.pushTwice(passed); }
    public void expected_pushTwo2(Object passed, Object passed2) { super.pushTwo(passed, passed2); }
    public Object expected_popOff2() { return super.popOff(); }

    // overridden new methods

    public void pushOn2(Object passed) { c.setInner(c.inner1).pushOp(passed);  }
    public void pushTwice2(Object passed) { c.setInner(c.inner2).pushOp(passed); }
    public void pushTwo2(Object passed, Object passed2) { c.setInner(c.inner4).pushOp(passed, passed2); }
    public Object popOff2() { return c.setInner(c.inner3).popOp(); }

}
```

```
public class LockConnector extends Host.MyStack
{
    //inner class to glom on the componenet.
    public class LockBase extends LockAPPC.LockComponent
    {
        public Inner inner;
        public Inner inner1 = new Inner1();
        public Inner inner2 = new Inner2();
        public Inner inner3 = new Inner3();
        public Inner inner4 = new Inner4();

        public Object get_host() { return LockConnector.this; }
        public void actualOp(Object passed) { inner.actualOp(passed); }
        public void actualOp(Object passed, Object passed2) { inner.actualOp(passed, passed2); }
        public Object actualOp() { return inner.actualOp(); }

        interface Inner
        {
            public void actualOp(Object passed);
            public void actualOp(Object passed, Object passed2);
            public Object actualOp();
        }
        public class Inner1 implements Inner
        {
            public void actualOp(Object passed)
            {
                LockConnector.this.expected_pushOn(passed);
            }
            public void actualOp(Object passed, Object passed2){}
            public Object actualOp()
            {
                return null;
            }
        }
        public class Inner2 implements Inner
        {
            public void actualOp(Object passed)
            {
                LockConnector.this.expected_pushTwice(passed);
            }
            public void actualOp(Object passed, Object passed2){}
            public Object actualOp() { return null; }
        }
        public class Inner3 implements Inner
        {
            public void actualOp(Object passed){}
            public void actualOp(Object passed, Object passed2){}
            public Object actualOp()
            {
                return LockConnector.this.expected_popOff();
            }
        }
        public class Inner4 implements Inner
        {
            public void actualOp(Object passed){}
            public void actualOp(Object passed, Object passed2)
            {
                LockConnector.this.expected_pushTwo(passed, passed2);
            }
            public Object actualOp() { return null; }
        }
        public LockBase setInner(Inner inner)
        {
            this.inner = inner;
            return this;
        }
    }

    public LockBase c = new LockBase();

    public void expected_pushOn(Object passed) { super.pushOn2(passed); }
    public void expected_pushTwice(Object passed) { super.pushTwice2(passed); }
    public void expected_pushTwo(Object passed, Object passed2)
                { super.pushTwo2(passed, passed2); }
    public Object expected_popOff() { return super.popOff2(); }

    // overridden new methods

    public void pushOn(Object passed) { c.setInner(c.inner1).pushOp(passed);  }
    public void pushTwice(Object passed) { c.setInner(c.inner2).pushOp(passed); }
    public void pushTwo(Object passed, Object passed2)
                { c.setInner(c.inner4).pushOp(passed, passed2); }
    public Object popOff() { return c.setInner(c.inner3).popOp(); }
}


public class Main
{
    static public void main(String argv[])
    {
        LockConnector stack1 = new LockConnector();
        LockConnector stack2 = new LockConnector();
        stack1.pushOn(new Integer(5));
        stack2.pushOn(new Integer(2));
        stack2.pushTwice(new Integer(2));
        stack1.pushTwo(new Integer(4), new Integer(3));
        Integer number1 = (Integer)stack1.popOff();
        Integer number2 = (Integer)stack2.popOff();
        Integer number3 = (Integer)stack2.popOff();
        Integer number4 = (Integer)stack2.popOff();
        System.out.println(number1 + " " + number4);
    }
}
```