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

Toby Elliott

Introduction

The benefits of Adaptive Plug-and-Play Components are described in some detail by Lieberherr, Mezini and Lorenz [1] [2]. The emphasis is on generating code that is modular and easily extensible, even when the source is not directly available. It is also focused on grouping similar functions together in their own space so that repeated use of the same functions across methods may be abstracted out and added to the code at a later time.

Components are a grouping of these like concerns, aspects of a program that may cross-cut the program, but all serve a centralized function. For example, access to a database may require many functions to lock and unlock the database before performing operations on it. Rather than have these functions contained within the code, they are abstracted out, then inserted as components, using connectors to reunite them in the code as needed. 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.

The ultimate goal of this is aspectual programming, a form of programming in which the code is separated out into many sections without a full knowledge of the structure of the application. At a later point, often compile- or even run-time, these components are mapped to participants whose functionality is extended by the code in the components.

This paper describes one particular approach to implementing ‘connectors’, the code that provides a framework for the components to cut across the application class and add the functionality of the components to their methods. Several approaches have been attempted for to do this. The one in this paper uses Java 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. It has advantages and disadvantages as will be seen when it is compared to several other approaches to building this framework. These advantages and disadvantages arise out of the design factors, which have a strong impact on the resulting approach.

Design Factors

Several factors affected the approach taken in this design. First and foremost is the idea that the connector should be doing all the work, and that no changes should be necessary in the base object classes. Ideally, there should also be no changes necessary in the component class. With the inner class method, this is not possible, though if specific guidelines are followed when creating the components, they can be used by the connectors without regard to application structure and thus are effectively unchanging.

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 is designed to be flexible, at the cost of some speed enhancements. This was a conscious choice, and represents a common tradeoff. The key flexibility, and one of the strengths of this model, is that it can support multiple differently-signatured application class methods with the same slice of code.

Thirdly, the focus is 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.

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 connector 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.

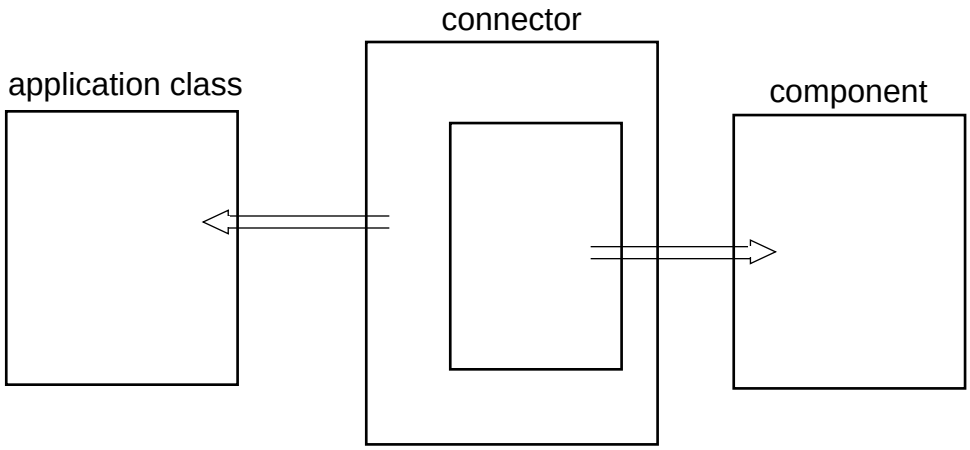

Figure 1: Inheritance diagram for the connector, showing how inner classes are used to join the appropriate application methods with the wrapping component

A separate connector will need to be generated for every application class that is to have a component attached to it. This is because Java will not allow multiple inheritance by the connector class.

An application class may receive multiple new behaviors by having more than one component associated with it. This is accomplished by chaining the connectors together, passing control flow through each of the components before (and after) arriving at the original method, as in Figure 2.

The abstracted sequence of calls relative to the outer and inner classes is shown in Figure 2.

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.

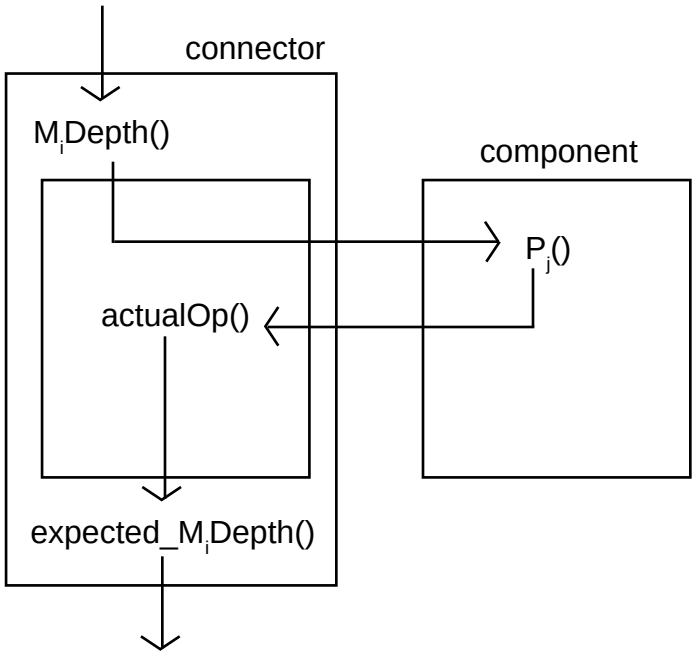

Figure 2: Method progression through the connector and component classes. Control is passed into the connector by calling *M*$_i$*Depth*(), which executes the mapped component code within it's inner class. The component code calls actualOp(), which has been dynamically defined in the inner class and calls the appropriate expected_*M*$_i$*Depth*() in the outer class. expected_*M*$_i$*Depth*() makes a call to the appropriate function in the superclass, which may represent the original method, or possibly another connector in the sequence.

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;
}
```

In this example, the first line refers to the “chaining” order of the components. Clearly, the order in which the components are executed is important. For example, the positioning of a counting component may come up with very different answers when placed before or after a component that may determine that the operation should not continue.

The remainder of the examples show the mapping in each of the components of component operations to the original methods. The wildcard allows multiple methods in the application class to be mapped to the same component operation, allowing logical clustering of methods.

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);
    }
}
```

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 .

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 $M_{i\text{-return-type}}$ $M_i Depth$ ($M_{i\text{-parameters}}$) { [return] c.setInner(c.inner i).P_j($M_{i\text{-parameters}}$);}
public $M_{i\text{-return-type}}$ expected_$M_i Depth$ ($M_{i\text{-parameters}}$) { [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 *j* possibilities, which is done through these inner classes. The *j* possibilities are determined by the number of operational methods (combinations of parameters and return types for each mapped method, as determined in the precompilation variables above).

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 (where *i* represents the number of methods that are to have aspects of the components wrapped around them, and *k* is the number of different components in the chain.

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.

Comparions to Other Precompilation Approaches

Several other approaches to generating connectors in pure Java (using a precompilation phase) have been proposed. They share much of the same philosophy, seeking to avoid modification of the application class or component, but have different strengths and weaknesses.

The proposal by Mezini and Lieberherr [1] requires that the source code be made available for recompilation. It does, however contain the essential elements of the overriding approach - it defines collaborative behavior through superclassing the component. This approach also places more emphasis on the class-mapping approach and maintaining control flow through use of visitor classes, true to it's roots in Demeter/Java. While this method does allow for complex object mapping and applica-

tion redefinition, it is unwieldly for the application of simple components, a strength of the process described in this paper.

Another proposal, by Mezini, Lieberherr and Seiter [3] also adopts the inner class approach as an easily patterned way to generate complex connector objects. As a result, their approach is quite similar to the approach taken in this paper. However, they defer more behaviour from the component into the connector, preferring a more complex implementation of the connector. This does provide them with the ability to interleave applications more effectively, a potential bonus when working with collaborating classes. By trying to keep most of the coding behavior in the components, the inner class method in this paper keeps a tighter rein on how much the connector can modify the behavior of the components. This is a weakness of this paper's approach: while direct access to the component is explicitly not necessary, it is frequently desirable in order to execute more complex object mappings.

A different approach is taken by Predrag Petkovic. His proposal does not involve inner classes at all, and is based more closely on the Demeter/Java approach, defining before and after methods to be wrapped around the called method. Obviously, this approach has a more difficult time handling components that nest their original call inside loops or block structures, and it requires more complex coding at the component level, but it makes the connectors substantially simpler to write. Indeed, it might even be possible to avoid a precompilation step with this approach, as the connector may not prove difficult to write. Predrag's proposal shares the least philosophically with this paper.

Finally, the most radical approach is taken by Johan Ovlinger. He adopts the use of a 'thunk', a wrapper around the original method that allows for generic access to itself at the appropriate moment while still generecizing the method so that it can be passed around without concern for parameters and return types. This approach is both powerful and flexible. While a totally different approach, the underlying principles are actually closest to those in this paper, most notably support for multiple return types and parameters. The code generated by his approach is shorter (not having to do a different piece of code for each return type/parameter combination). As a tradeoff, though, it is generally more difficult to read, and uses a form of reflection, which the approach described in this paper sought to avoid. Benchmarking these two approaches would be an excellent way to determine the viablility of reflection as an approach to be taken by the connectors.

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.

References

[1] Mira Mezini and Karl Lieberherr. Adaptive Plug-and-Play Components for Evolutionary Software Development.

[2] Karl Lieberherr, David Lorenz and Mira Mezini. Programming with Aspectual Components.

[3] Karl Lieberherr, Mira Mezini and Linda Seiter. Inner Classes for Component-Based Programming.

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);
    }
}
```