Final project for com3362
Northeastern University, June 1999
Predrag Petkovic, predrag@ccs.neu.edu

Implementation of aspectual components

1. Introduction

This project is implementation attempt of the aspectual components in Java. This implementation of Aspectual Components is based on “event driven action” design. Main goal achieved by this implementation is simplicity of the deployment of the components into an existing application. This is achieved at the cost of low speed (extensive use of reflection) and non-generality. Implementation is based on two-layer model (application classes, and components). In this implementation we need source code for both application classes and component. We need to instrument them so they will be prepared for connection. Once we instrument application classes and components they do not need to be changed in order to connect them or disconnect them anywhere in our application. Original application classes should be written in usual Java style, following principles of Object Oriented Design, no special rules need to be respected. Components on the other side need to follow some rules, which will be stated later, but are also pure Java code and can be compiled before instrumentation.

2. Design

In order to connect application classes and components they need to communicate in some fashion. Design of this implementation is based on event model. In the event model some objects notify of some event, objects that are interested in listening to that type of event by firing events to them. Upon receiving event listener performs some action based on received event. Instrumentation of the original code (of both application classes and components) is made in order to enable them to notify all interested listeners of what method is invoked on them, by firing MethodEvents. Once they are instrumented the connector object is one that listens to the MethodEvents and based on the map decides what to do, what method to invoke. Map is an object that contains information for deployment of the component into application classes. Once application classes and components are instrumented they can be re-used in different mappings. Application classes and component classes communicate through the connector that receives MethodEvent from one side of connection, translate it to the other's side names, using map and invoke appropriate method.

Example: of connecting and then disconnecting component.

```
Connector  conn = Aspect.connect("deployment_file.dpl");
....
conn.disconnect();
```

3. Components

Component is one public abstract class that has several public inner classes representing participants. Component has to be written in one public class that doesn’t implement any interface. Participants have to be written as public inner classes of the component. Component normally should not have any other members (constructors, methods or fields). All non-public inner class will not be regarded as a component, neither will any public inner classes nested inside inner classes.

Aspectual component’s participants can have five types of methods:

- *before/after*
  Methods that are activated before/after invocation of application class methods.
- *expected*
  Methods that are expected to be find in application class, and are used inside participant’s code.
- *replaced*
  Methods from application classes are replaced by implementation provided by participant.
- *participant’s*
  Methods that are local to the participant.

All of them have to be public and expected methods have to be abstract too. They need to follow naming conventions in order to be recognized when component is mapped.

- *before* name starts with before_
- *after* name starts with after_
- *expected* name starts with expected_
- *replaced* name starts with replace_
- *participant's* do not need to follow any convention

Restrictions in writing components:

- No new instantiations of participants.
- Instead of using this, component writer should use getHost() method, that should be declared as

  public abstract Object getHost() .

- No behavior can be attached on one instance of the application class (e.g. counter that counts accesses to some object, counts accesses for all instance of a class together).
- Participant methods shouldn't be overloaded
  Following is forbiden:
  before_set(int i) {...}
  before_set(float f) {...},
  because mapping uses only methods names.
- There can not be mapping of constructors (for now, later will this also be implemented)

Example of component:

```
package components;
public class AutoReset {
  public abstract class DataToReset {
    public abstract void expected_reset();
    public void after_setOp() {
      if (++count >= 100) {
      expected_reset();
      count = 0;
      }
    }
    private int count = 0;
  }
}
```

4. Deployment

Deployment grammar:

Deployment = Component Applications "{" ParticipantMaps "}".
Component = "component" ComponentName.
Applications = "applications" ApplicationName { "," AppliactionName } ";".
ParticipantMaps = ParticipantMap { ParticipantMap } .
ParticipantMap = ParticipantName "->" ApplicationNameList "{" MethodMaps "}".
MethodMaps = MethodMap { MethodMap }.
MethodMap = ParticipantMethod "->" ApplicationMethods ";".
ApplicationMethods = MappedMethod { "," MappedMethod }.
MappedMethod = MappedMethodName ["*"] | "*"MappedMethodName.
ComponentName = Name.

ApplicationName = Name.
ApplicationNameList = Name { "," Name }.
Name = IDENTIFIER { "." IDENTIFIER }.
MappedMethodName = IDENTIFIER.
ParticipantMethod = IDENTIFIER.
ParticipantName = IDENTIFIER.

Deployment restrictions:

- application class can be mapped only to one participant
- participant's *expected* method can be mapped to only one application method
- participant's *expected* method has to be mapped
- mapped application and participant methods have to be compatible

Examples of compatibility:

Participant Method	Application Method
void before_setOp()	void setX(int), Object set(Integer, String)
Object expected_method(String)	String toString()
Integer replace_get()	Object getSome(int)

Example of deployment file:

```
component components.AutoReset;

applications application.Point;

{

      DataToReset -> application.Point {

            setOp -> set*;
            reset -> reset;
      }
}
```

5. Implementation details

Instrumentation of aspectual components and application classes is made using parser written in JavaCC. Key role in instrumentation of application classes play Invoker class. One instance of Invoker is added to every application class. Every public method of application class is transformed into private ("hidden") method. New public method only passes MethodEvent, which contain arguments and hidden method, to the Invoker and retrieve result from it. Invoker upon receiving MethodEvent fires before event, then invoke method and fires after event.

```
      class Invoker {

            ... // register/unregister methods for Before/After listeners

            void receiveMethodEvent(MethodEvent event) {

                  try (fireBefore()) {

                        invokeMethod(evt);
                  } catch (ArroundException e) {}
```

```
                fireAfter();

            }

        }
```

Connection between application and component is established when connector register itself as a listener to the Invoker (precisely it register at application class that delegate call to the Invoker). AspectConnector is connection between application and component. It takes a map object, establish connection and using map translate events that receives from application class to call to the participant's methods and vice verses.

Before connection is established connector needs to have map, upon which it is going to work. In order to get the map, we need to read deployment file. Deployment file is parsed using DelpoymentParser, which is also created by JavaCC. DeploymentParser returns ComponentMap object that contains map structure represented by strings (ParticipantMap, MethodMap and ApplicationMap are parts of the returned ComponentMap). Now the ComponentMap need to be checked with the component and application classes. The checking is done inside `getMap()` method of the Mapper. It checks the deployment and return Map that now contains Method and Class objects, as opposed to ComponentMap that contains their names.

All this is done just in one call:

```
Connector conn = Aspect.connect("deployment_file.dpl");
```

Aspect connector implements Connector interface that has only one method `disconnect()` that allows to disconnect component from application.

- Instrumentation of components

Example: (Original code was given earlier in the text.)
Generated code:

```
public class AutoReset implements Component {

      public class DataToReset {

            public void expected_reset() {

                  connector.receiveMethodEvent( new MethodEvent( getHost(),
                  MethodEvent.NORMAL, resetMethod, new Object[]{}); }
            private int count = 0;
            public void after_setOp() {
                  if (++count >= 0) { expected_reset(); count = 0; }
            }
      }
      private Method resetMethod;
      { try {
            resetMethod =
      AutoReset.DataToReset.class.getMethod("expected_reset", new Class[]{});
      } catch(Exception e) {} }

      public DataToReset DataToReset; // an instance for every participant
      private Connector connector;
      public void initComponent(Connector c) {
            connector = c;
            DataToReset = new DataToReset();
      }
      public Object getHost() {
            return connector.getHost();
      }
}
```

- Instrumentation of application classes

Example:
Original code :

```
public class Point {

      public int getX() { return x; }

      public int getY() { return y; }

      public void setX(int i) { x = i; }

      public void setY(int i) { y = i; }
      private int x = 0;
      private int y = 0;
}
```

Generated code (part):

```
public class Point {
      public static void addBeforeListener(BeforeListener l) {
            invoker.addBeforeListener(l);}

      public static void removeBeforeListener(BeforeListener l) {

            invoker.removeBeforeListener(l); }
      public static void addAfterListener(AfterListener l) {
            invoker.addAfterListener(l);  }
      public static void removeAfterListener(AfterListener l) {
            invoker.removeAfterListener(l); }
      private static Invoker invoker = new Invoker(Point.class);

      private static Method getXMethod;
      .....
      static { try {
            getXMethod = Point.class.getMethod("hidden_getX",new Class[] {});
            .....
      }     }
      public int getX() {
            invoker.receiveMethodEvent( new MethodEvent(this,MethodEvent.NORMAL,
                                    getXMethod, new Object[]{}));
            return ((Integer)invoker.getReturnValue()).intValue();
      }
      public int hidden_getX() {
            return x;
      }
      .....
}
```