

An Example of Using Collaborator and Adapters to Reuse a Synchronization Pattern

Ken Anderson
 BBN Technologies
kanderson@bbn.com

Abstract

In the AIRES Project - Aspects in Real-Time Embedded Systems, BBN Technologies and Northeastern University have teamed to investigate using Aspect Oriented Programming (AOP) techniques in the domain of real-time embedded systems. This work will build on two existing systems. BBN's Quality Objects (QuO) technology adds systemic-aspects to distributed objects systems, such as CORBA and RMI. Northeastern's DemeterJ adds functional-aspects to the Java programming language. The AIRES project will concentrate on composing multiple aspects into real-time applications, which has well defined properties. The properties will be analyzed for conflicts at compile time and code will be generated to check and control the properties at run-time.

In this position paper we describe one example of using AOP techniques to reuse a synchronization pattern. The example describes the pattern using Collaborators and Adapters and estimates the reduction in code for a large existing library (Java Swing). This work is sponsored by DARPA under contract no. F33615-00-C-1694.

Swing EventListener example

One facet of aspect-composition is to reuse the same pattern for several class-graphs. This is in contrast to AspectJ, where an aspect is a "distributed" type whose implementation is described in terms of several other types (classes). Our approach adds an extra level of indirection to decouple the definition of the aspect's pattern from its binding to a specific group of classes. Collaborators define the pattern and Adapters bind it to a specific class graph. Several Collaborators can be combined in the same object as Adapters overlay them onto a class graph.

We plan to use Collaborator and Adapter techniques in both Demeter for programming functional-aspects (business logic) and in QuO SDL (Structural Description Languages) to generate smart stubs and skeletons which manage conflicts between clients and objects at runtime.

The following example shows the benefit of using Collaborators and Adapters for code reuse, even without using their composition properties.

Collaborator

This collaboration describes a synchronization aspect, namely a variant of the EventListener pattern used extensively in Java's Swing GUI library.

```
collaboration EventListener {
  participant Publisher {
    // Additional fields.
    EventListenerList list = new EventListenerList();

    // Additional methods.
    public void addListener(Class c, Listener x) {
      list.add(c, x); }
}
```

```

public void removeListener(Class c, Listener x) {
    list.remove(c, Listener x); }

// Expected methods.
expect public void changeOp(...);

// Behavior to be added to an expected method.
replace public void changeOp(...) {
    expected();
    Listener[] ls = list.getListeners(Event.class);
    Event e = null;
    for(int i = 0; i < ls.length; i++) {
        if (e == null) e = new Event(this); // Lazy create.
        ls[i].handleEvent(e); }}

participant Listener {
    public void handleEvent(Event e); }
participant Event;
}

```

This collaboration has three participants (or roles):

- **Publisher** - A class, such as `AbstractButton`, that will publish an event to any of its listeners, whenever an event occurs, ie. when the button is pressed.
- **Listener** - A class, such as `ActionListener`, that can subscribe to receive events by calling `addListener()`.
- **Event** - A class of object passed from a publisher to a listener describing the event that has occurred.

The publisher has a field named "list" that contains an `EventListenerList` that manages the listeners. The publisher is also expected to have a method that matches the signature of the `changeOp()` method. The collaboration describes additional behavior that must be added to that method. Specifically, after the expected behavior of the method is performed, all listeners for the event are notified.

Adapters

An adapter is used to connect (or map) a collaboration to a specific class graph. One class-valued variable may be mapped to many classes and one method-valued variable may be mapped to many methods. By using several adapters a collaboration can be used in different ways. For example, here are two adapters that correspond roughly to Swing's `ActionListener` and `PropertyChangeListener`:

```

adapter ActionPerformed {
    Publisher = public class AbstractButton {
        changeOp(...) = fireActionPerformed(...);
    }
    Event = public class ActionEvent;
    Listener = Interface ActionListener extends EventListener {
        handleEvent() = actionPerformed();
    }
}

```

In the `ActionPerformed` adapter, the `Publisher` role is played by the class `AbstractButton`. The `changeOp()` role is assigned to its `fireActionPerformed()` method.

```

adapter PropertyChanged {
    Publisher = public class JMenuItem {
        changeOp(...) = set*(...);
    }
}

```

```

    }
    Event = public class PropertyChangedEvent;
    Listener = interface PropertyChangedListener extends EventListener {
        handleEvent() = propertyChanged();
    }
}

```

In the `PropertyChanged` adapter, the `changeOp()` role is played by each of the `set*()` methods of `JMenuItem`. Also, the `Listener` interfaces are automatically generated.

Impact

We estimate the impact of using this Collaborator and Adapter pattern from the following statistics from the Swing Library:

```

506 .java files
225,000 Lines of code and documentation.
21 Listener interfaces ~ 200+ lines.
73 uses of EventListenerList ~ 1,400 lines of code.
25 documentation errors due to cut and paste.

```

James Gosling wrote the original version of the `EventListenerList` class. Unfortunately, the pattern of use of the class can only be described in comments. His comments show that the event should be constructed only when necessary, in a lazy fashion. This pattern was then copied by cut and paste 73 times leading to several variations. Here's an example:

```

/**
 * Notifies all listeners that have registered interest for
 * notification on this event type. The event instance
 * is lazily created using the parameters passed into
 * the fire method. The listener list is processed in last to
 * first order.
 * @see EventListenerList*/
protected void fireActionPerformed() {
    // Guaranteed to return a non-null array
    Object[] listeners = listenerList.getListenerList();
    ActionEvent e = new ActionEvent(this, ...);
    // Process the listeners last to first, notifying
    // those that are interested in this event
    for (int i = listeners.length-2; i>=0; i-=2) {
        if (listeners[i]==ActionListener.class) {
            ((ActionListener)listeners[i+1]).actionPerformed(e);
        }
    }
}
}

```

The documentation suggests that the method follows Gosling's pattern of lazy event creation. However, the code does not follow the pattern. It simply creates the event each time. Thus the documentation is in error. This error is repeated 25 times out of 73 uses of this pattern.

Conclusion

Collaborators and Adapters will enable the analysis of conflicts when different aspects are woven together. The example showed that a substantial amount of code can be reused without adding additional complexity. A side benefit of this reuse is more robust, maintainable code and accurate documentation.

References

Karl Lieberherr and David Lorenz and Mira Mezini, Programming with Aspectual Components, College of Computer Science, Northeastern University {NU-CCS-99-01}, March 1999.

Mira Mezini and Karl Lieberherr, Adaptive Plug-and-Play Components for Evolutionary Software Development, Object-Oriented Programming Systems, Languages and Applications Conference, Special Issue of SIGPLAN Notices, 33, 10, October 1998, p. 97-116.