1 Modeling Aspects with AP&P Components

In this section, we outline the elements of APPC construct and show how APPCs can be used to model aspects. We will start with a small set of APPC concepts, illustrate the use of these concepts in modeling aspects and then inttroduce the rest of the concepts by presenting more sophisticated examples. In order to facilitate the comparison, a good part of the examples in this section are taken, eventually slightly modified, from the tutorial on *AspectJ* [**?**], which is the first extension of Java designed and implemented to bridge the gap between object and aspectual decomposition in Java.

1.1 AP&P Components as a Language Construct

An AP&P component is a module construct for object-oriented languages on top of classes that serves to capture behavior that affects several classes. However, the captured behavior is written in terms of an ideal class graph and not to a concrete application. An AP&P component consists of the following elements:

- A set of *participants*. A participant is a formal class (or a class in an ideal class graph, ICG), which stands at the same level as a formal argument in a procedure declaration I needs to be later bound to classes in a concrete class graph, CCG.
- Instead of type constraints to be satisfied by the CCG classes, a participant definition merely lists a set of operation signatures, preceded by the keyword *expected*.
- Expected operations are used in the appc-specific definition of the participant. This definition includes:
  - (a) "reimplementations" of the expected operations, denoted by the keyword *replace*.
  - (b) definitions of new operations – all operations in the definition of a participant that are declared public.
  - (c) new field definitions that would extend the structure of the CCG classes to be bound to this participant.
  - (e) In order to implements replacements or new functionality, a participant definition might include "local" definitions (operations as well as fields). These are denoted by the same keywords as in Java: *private*, *protected*.

The key idea behind APPCs as a construct for reconciling object-based and aspect-based decomposition of the systems is to add a new dimension in the organization of object-oriented software, as illustrated in Fig. 1. The software is first decomposed based on aspects which are symbolically represented by rectangle areas with different grey scales in Fig. 1. Each aspect definition has its own object-based decomposition. Different data types involved in an aspect definition are represented by different shapes in Fig. 1. Aspect definitions are based on the assumption that other aspects or/and the concrete application will supply part of the functionality of the involved data which is then modified/extended by the aspect. The expected interface of aspect definitions is represented by the transparent (colorless) shapes on the left hand side of all but one colored area in Fig. 1.

The left most colored area in Fig. 1 represents a concrete application. It is self-contained in that it does not expect anything to be supplied from the outside and provides the basic implementation of the application functionality. It is generally much richer in data type definitions than the aspect definitions. The modifications and/or extensions of the expected interface by aspect definitions are illustrated by the shape towers on the right hand side of each grey scale area: there are generally more shapes of each kind on the right hand side than on the left hand side. Colorless shapes on the left hand side are replaced by colored shapes on the right hand side. In addition, some new shapes are added to each tower.

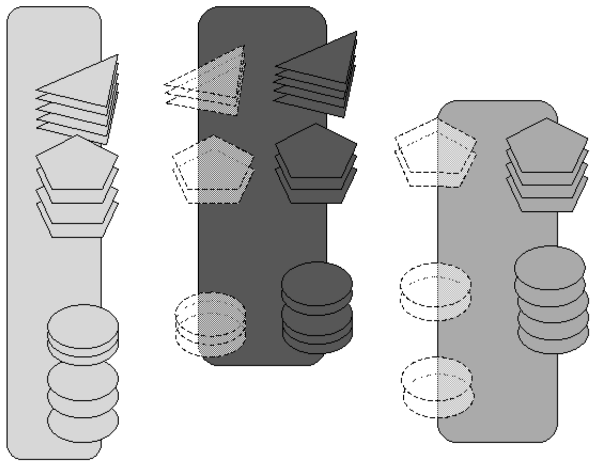

Figure 1: Aspect Decomposition with APPCs

For illutrating the concepts so far, we start with a very simple example. The aspect considered here is a trivial one: instrumenting access operations on data types. A description of the aspect in informal English would be: "For any data type in an application, say DataToAccess, any read access operation, AnyType readOp() defined for DataToAccess, and any invocation of this operation on an instance of DataToAccess, dataInstance, display Read access on <string representation of dataInstance>. Sure enough, the description is reusable in different contexts, since it does not make any commitment about its application context. It is obvious that an aspect definition language that allows to delay binding aspects to their deployment contexts is desirable. This is in fact the key point about using APPCs to model aspects.

The code in the ShowReadAccess appc below is actually as free of details about concrete deployment contexts as the informal English description above is. It simply states that whatever classes in an application will be assigned to implement (play) the single participant in the aspect definition, DataToAccess, and whatever methods in these classes will be assigned to implement the interface expected from DataToAccess, readOp in this case, the latter methods will be replaced by implementations that first print out a message and then execute the original methods (see the replace statement below).

```
ShowAccesses package;

  appc ShowReadAccesses {

    participant DataToAccess {
      expected Object readOp();

      replace Object readOp() {
        System.out.println("Read access on " + this.toString())
        return expected.read_op();
      }
    }
 }
```

...

The meaning of the pseudo-variable this is the same as in a Java program. In the definition above, it denotes the application instance being instrumented. If there are no ambiguities, there is no need to explicitly refer to the pseudo-variable this. Any invocation without an explicit receiver is considered to be a self-invocation. A method for this invocation is first looked-up in the aspect-specific definition of the participant. If no method is found there, the class in the application assigned to play the participant role is searched for.

Now, given a concrete application the programmer needs to deploy the aspect with the application by assigning classes in the application to aspect participants and by specifying how the interface expected by the aspect is implemented in terms of the application. Specifying a mapping for the expected interface of an aspect occurs apart of the definition of an aspect by means of so-called *connector* constructs. In abstract terms, the deployment process is schematically presented in Fig. 2. The connector is symbolically represented by the two colored rectangle between the application and the aspect. It transforms a part of the application definition into the expected interface of the appc. The result of the deployment is equivalent to the set of shape towers on the right hand side of Fig. 2. Partial definitions from the application and the aspect are mixed together into the final definition of each object.

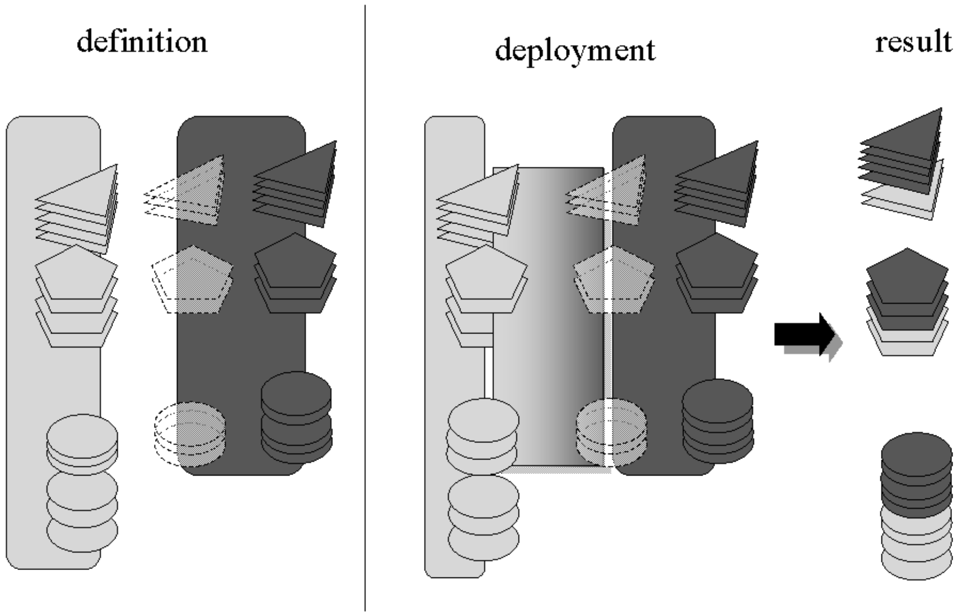

Figure 2: Deploying Aspects with Applications

For illustration consider the trivial example of an application that models graphical shapes (the Shapes package below), and a simple deployment scenario in which we are interested in instrumenting only read accesses on point objects.

```
Shapes package;

class Point {
  private int _x = 0;
  private int _y = 0;

  void set(int x, int y) { _x = x; _y = y;}
```

```
    void setX(int x) { _x = x; }
    void setY(int y) { _y = y; }
    int getX(){ return _x; }
    int getY(){ return _y; }
  }

  class Line { ... }

  class Rectangle {... }
  ...
```

The deployment of ShowReadAccesses with the class Point is specified by the ShowReadAccessConn1 connector below.

```
Deployment package;

import ShowAccesses.*;
import Shapes.*;

connector ShowReadAccessesConn1 {
   Point is DataToAccess with {readOp = get*};
}

...
```

The connector in the example above is simple. It contains only one implement statement. In general, a connector consists of several scopes, where each scope contains several implements statements. A simplified grammar of the struture of connector is given in Fig. **??**. The class-to-participant mapping in an implements statement is in general many to one, as it will be illustrated in the following. An implement statement has one or more mapping clauses for the expected interfaces associated with it. In the example above the mappping is done based on pattern matching on operation names, with the pattern being specified by the regular expression get*. In general, this mapping can also take a more "programmatic" form, i.e., an implementation of the expected signature is given by means of composing functionality provided by the application. Later in this section, we will present more sophisticated examples that give a more complete picture of the structure of connectors.

For the moment being, it should be noted that specifying deployment details apart of an aspect definition, make the aspect reusable with other classes without changing its definition. Rather, what is needed is the definition of a new connector. For illustration, assume that after having defined ShowReadAccessesConn1, we realize that read accesses on lines need also be instrumented as well. For this purpose the ShowReadAccessesConn2 connector is defined below as an extension of ShowReadAccessesConn1.

```
connector ShowReadAccessesConn2 extends ShowReadAccessesConn1 {
   Line is DataToAccess with {readOp = get*};
  }

...
```

On the other side, if we knew from the very beginning that whatever classes are included in the application, we would like to instrument all read access operations, we would have defined ShowReadAccessesConn3, rather than ShowReadAccessesConn1. If this was the case, no modification of the aspect or extension of the connectors would be needed, if we extend the application with a new data type, say Circle.

```
connector ShowReadAccessesConn3 {
```

```
    Shapes.* is DataToAccess with {
       readOp = set*;
     }
}
```

An aspect can be defined that affects several operation categories of the same class in different ways. For illustration, assume that in addition to read accesses, we would like to instrument write accesses. This is expressed by the ShowReadWriteAccesses component below. The aspect is then deployed with the shape package by ShowReadWriteAccessesConn1.

```
ShowAccesses package;

appc ShowReadWriteAccesses {

  participant DataToAccess {
     expected Object readOp();
     expected void writeOp(Object[] args);

     replace Object readOp() {
        System.out.println("Read " + displayString());
        return expected.read_op();
      }

     replace void writeOp(Object[] args) {
        System.out.println("Write " + displayString());
        expected.write_op(Object[] args);
      }

     private String displayString() {
       return "access on " + this.toString();
     }

   }
 }

...

Deployment package;

import ShowAccesses.*;
import Shapes.*;

connector ShowReadWriteAccessesConn1 {
    Shapes.* is DataToAccess with {
       readOp = set*;
       writeOp = get*;
  }
}
```

Assuming that ShowReadAccesses was already defined, we could also define ShowReadWriteAccesses as an extension of ShowReadAccesses, and the corresponding connector as an extension of the ShowReadAccessesConn3 connector, thus supporting reuse at the level of both aspect and connector definitions.

```
appc ShowReadWriteAccesses extends ShowReadAccesses {

  participant DataToAccess {
```

```
    expected void writeOp(Object[] args);

    replace void writeOp(Object[] args) {
      System.out.println("Write access on " + this.toString());
      expected.writeOp(Object[] args);
    }

  }
}

connector ShowReadWriteAccessesConn2 extends ShowReadAccessesConn3 {
  Shapes.* is DataToAccess with {
    writeOp = get*;
  }
```

1.2 AP&P Components versus Frameworks

1.3 Composing Aspects

The example discussed so far illustrated the basic elements of the AP&P Component language for aspect definition, deployment, and reuse. In the following, we show a few more details related to aspect definition and concentrate on how aspects can be composed. The first example illustrates how aspects can be composed by being deployed simultaneously with the same application. For illustrating this case, two new aspects are defined below. The first aspect is about logging new instances in a system, called NewInstanceLogging below. The NewInstanceLogging component illustrates the definition/initialization and use of aspect related state which is shared by all application classes that will be assigned to participant roles of the aspect for a certain deployment. For instance, equally how many application classes will be assigned to be DataToLog during a deployment, they all will share the logObject variable defined in NewInstanceLogging.

```
appc NewInstanceLogging {

  participant DataToLog {

    expected public DataToLog(Object[] args);

    replace public DataToLog(Object[] args) {
      expected.DataToLog(args);
      long time = System.currentTimeMillis();
      try {
        String class = this.class.getName() + " ";
        logObject.writeBytes ("New instance of " + class + at " " + time + " " \n);
      } catch (IOException e) {System.out.println(e.toString());}
    }
  }

  protected static DataOutputStream logObject = null;

  public static init() {
      try {logObject = new DataOutputStream(new FileOutputStream(log));}
      catch (IOException e) {System.out.println(e.toString());}
    }
}
```

The second aspect is about automatically resetting values of certain data types after a certain number of accesses on that data has occurred. This second aspect, called AutoReset, is defined below It introduces a new instance variable to each application object that will be mapped to the DataToReset participant role in the component, called count. It is an instance variable, since it is declared as non-static within the aspect-specific definition of DataToReset. The AutoReset component illustrates another feature of AP&P Components: there are two kinds of expected operations. The first kind, represented by setOp, includes operations which are expected in order to be replaced by the aspect (expected operations preceded by the keyword replaced). Operations of the second kind, represented by reset are expected in order to be used by the aspect-specific definition of the participants. Expected operations of the first kind constitute the so-called *modification interface* between aspects and application. This corresponds to the concept of joint points in *AspectJ*. Expected operations of the second case constitute what we call *usage interface* between aspects and application.

```
appc AutoReset {

  participant DataToReset {
    expected void setOp(Object[] args);
    expected void reset();

    protected int count = 0;

    replace void setOp(Object[] args) {
     if (++count >= 100 ) {
        expected.setOp(args);
        count = 0;
        reset();
      }
    }
  }
}
```

Let us now consider how these aspects are composed by being deployed with the Shapes application, we have been using so far. Assume that we want (a) all three aspects to apply to points, (b) ShowReadWriteAccesses and NewInstanceLogging to apply to Line, and (c) only NewInstanceLogging to apply to Rectangles. This is expressed by the connector CompositionConn1 below. This connector illustrates that besides specifying mappings for expected operations by means of pattern matchings on method names, the deployment programmer can also directly implement an operation in the expected interface in terms of operations and/or state defined in the application. This is illustrated by the definition of reset() in CompositionConn1, i.e., in this case the mapping is "programmatic" rather than declarative.

```
connector CompositionConn1 {
    {Line, Point} is DataToAccess with {
          readOp = set*;
          writeOp = get*;
       };

   Point is DataToReset with {
         resetOp = set*;
         void reset() {
            x = 0;
            y = 0; }
      };

   Shapes.* is DataToLog;
```

```
}
```

The composition structure expressed by CompositionConn1 is symbolically represented in Fig. 3. The composed deployment is presented on the left hand side of Fig. 3. On the right hand side the result of the deployment is shown by the towers of shapes, representing the resulting weaved code for points (represented by triangles in the figure), lines (represented by xxx in the figure), respectively rectangles (represented by circles in the figure).

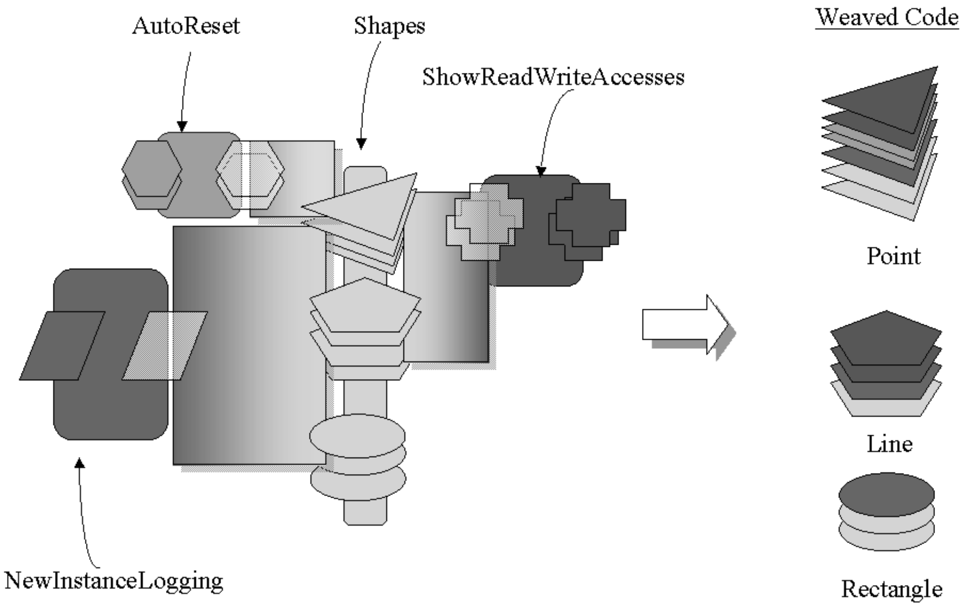

Figure 3: Deploying Several Aspects with the Same Application

Specifying connections apart of the definitions of the individual aspects, makes aspect definitions more reusable. For instance, we can later add new connection statements to specify additional "joint points" between an application and aspects, as illustrated by the CompositionConn2 connector below, which deployes the AutoReset aspect also with lines.

```
connector CompositionConn2 extends CompositionConn1 {

   Line is DataToReset with {
     resetOp = set*;
      void reset() { init();}
      };
}
```

Besides adding new connection statements, one can also modify existing ones. This is illustrated by the connector CompositionConn3 below. It redefines the connection CompositionConn1's statement for Point, to distinguish between the reset policy for operations that change both coordinates simultaneously and operations that change only one of the coordinates. Three different cases are distinguished by specifying three different *connection clauses*. This connection statement overrides the respective connection statement for the class Point, in CompositionConn1.

```
connector CompositionConn3 extends CompositionConn1 {

   Point is DataToReset with {
      { resetOp = set;
```

```
        void reset () {
            x = 0; y = 0; }
      }

      { resetOp = setX;
        void reset () { x = 0;}
      }

      {
        resetOp = setY;
        void reset () { y = 0;}
      }
  };

}
```

In the example considered so far, the aspects to be composed are independent form each other. The connector CompositionConn1 is equivalent to specifying three independent connectors (actually this is what Fig. 3 suggests). The order of deploying the individual aspects being composed is not important. In the following, we will illustrate how the expected interface of an aspect can be partly implemented by the application and partly connected to the provided interface of another aspect after the latter has been itself deployed with classes in the concrete application. Consider first the following two components.

```
aspect package;

appc DataWithCounter {

   participant Counter {
        void reset();
        void inc();
        void dec();
     }
   participant DataStructure {

        protected Counter counter;
        expected void initCounter();
            // the method for initializing the count field is let to be bound
            // at deployment time, when the concrete type of Counter is known

        expected void make_empty();
        expected void push(Object a);
        expected void pop();

        replace void make_empty() {
                counter.reset();
                expected.empty();
              }
        replace void push(Object a) {
              counter.inc();
              expected.push(Object a);
            }
        replace void pop() {
                counter.dec();
                expected.pop();
              }
```

```
    }
}

appc DataWithLock {

    participant Data {

        Lock lock;
        expected void initLock();
        expected AnyType method_to_wrap(Object[] args);

        replace AnyType method_to_wrap(Object[] args) {
          if (lock.is_unlocked()) {
                lock.lock();
                expected.method_to_wrap(Object[] args);
                lock.unlock(); }
          }
       }
   participant Lock {
      expected void lock();
      expected void unlock();
      expected boolean is_unlocked();
    }
}
```

The first aspect adds functionality for counting the elements included in a container data type, such stacks, queues, etc. The second aspect adds functionality for synchronizing invocations of certain operations on a given data type by means of a lock. Now assume the following application, which provides implementations for data types, such as stacks, queues, counters, locks, etc. We can now model stack and queue objects that count their elements and the operations of which are synchronized by means of a lock by specifying the connectors in the deployment package below. This kind of composition is schematically presented in Fig. 4.

```
appl package;

class StackImpl {
   String s = new String();
   void empty() {s = "";}
   void push(char a) {s = String.valueOf(a).concat(s);}
   void pop() {s = s.substring(1); }
   char top() { return (s.charAt(0));}
 }

class QueueImpl {...}

class CounterImpl {
   int i = 0;
   void resetImpl() {i=0;};
   ...
 }

class LockImpl {
   boolean l = true;
   boolean is_unlocked() {return l;}
   ...
```

```
}


deployment package;

connector addCounter {
  Counter = CounterImpl is DataWithCounter.Counter;

  Stack = StackImpl is DataWithCounter.DataStructure with {
      void initCounter() {counter = new Counter();}
      void push(Object obj) {push(((String) obj).charValue());}
      Object top() {return (Object) String.valueOf(top());}
      ...
   };

  Queue = QueueImpl implements DataWithCounter.DataStructure with {
     ... };
 }

connector addLock {
  addCounter.Stack is DataWithLock.Data with {
     method_to_wrap = {pop, push, top, make_empty}
   };

  addCounter.Queue is DataWithLock.Data with { ... };

  LockImpl is DataWithLock.Lock;
}
```

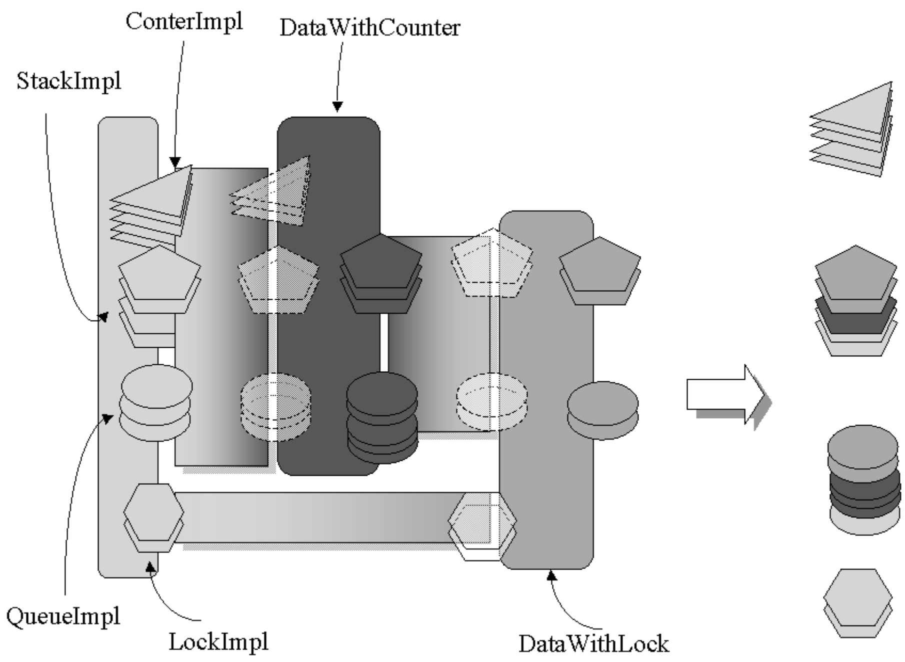

Figure 4: Deploying Several Aspects with the Same Application (2)

Alternatively, one can create composed aspects prior to deployment with any concrete application by connecting part of the expected interface of one aspect to part of the provided interface of another. The remaining unbound parts of the expected interfaces are connected to appliciations during deployment. Creating composed aspects prior to deployment is illustrated by the following code. First, DataWithStack is composed with DataWithLock into a new aspect, DataWithCounter&Lock. Later DataWithCounter&Lock get deployed with the concrete application. The process of creating DataWithCounter&Lock is schematically presented in Fig. 5.

```
aspect package;
...
appc DataWithCounter&Lock {
   participant Data = DataWithCounter.DataStructure is DataWithLock.Data with {
        method-to-wrap = {make_empty, pop, top, push}};

   participant Counter = DataWithCounter.Counter;

   participant Lock = DataWithLock.Lock;
}

...

deployment package;

conector addCounter&Lock {

  CounterImpl is DataWithCounter&Lock.Counter with {
      ...};

  StackImpl is DataWithCounter&Lock.Data with {
       void make_empty() {empty();}
       void initCounter() {
          counter = new Counter();
         }
      void push(Object obj) {push(((String) obj).charValue());}
      ...
    };

  QueueImpl is DataWithCounter&Lock.Data with { ...};
  LockImpl is DataWithCounter&Lock.Lock with { ...};
}
```

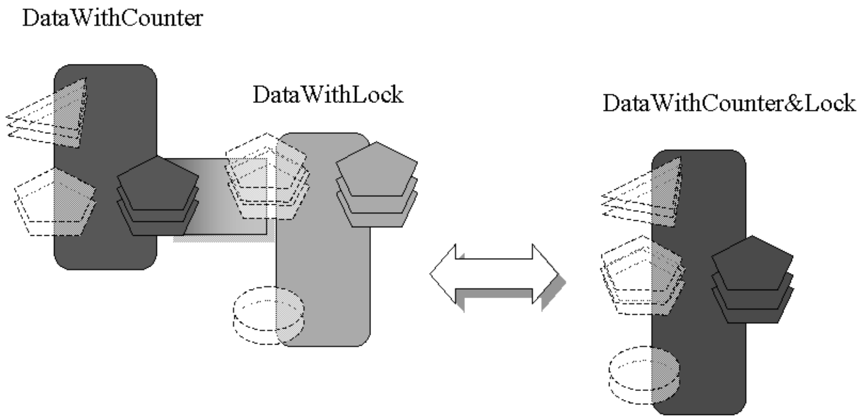

Figure 5: Deploying Several Aspects with the Same Application (3)

1.4 Defining New Behavior: The Publisher-Subscriber Aspect

In the examples considered so far, there has always been a single deployement of an aspect to a certain application. In the following we will illustrate that due to separate deployment specification, an aspect can be multiply deployed with the same application, each deployment with its own mappings. For illustration, a publisher/subscriber protocol aspect is modeled in the appc below.

```
appc PublisherSubscriberProtocol {

   participant Publisher {
       expected  void changeOp(Object[]  args);

       protected Vector subscribers = new Vector();

       public void attach(Subscriber subsc) {
           subscribers.addElement(subsc);}
       public void detach(Subscriber subsc) {
           subscribers.removeElement(subsc);}

       replace void changeOp() {
          expected.changeOp();
          for (int i = 0; i < subscribers.size(); i++) {
            ((Subscriber)subscribers.elementAt(i)).update(this);}
        }

   participant Subscriber {
      expected void subUpdate(Publisher publ);
      protected Publisher publ;

      public void update(Publisher aPubl) {
         publ = aPubl;
         expected.subUpdate(publ);}
      }
   }
}
```

In addition to replacing existing methods with enhancements, PublisherSubscriberProtocol defines new fields and methods that are involved in the protocol and establishes the structural relationship between the protocol participants. Note that the protocol in PublisherSubscriber-Protocol is defined for two hypothetical participants and not in terms of particular classes. As such, it can be applied to different publisher/subscribers pairs, as illustrated by the following two connectors. In the first case, the publisher role is played by the class Point, while the subscriber role is played by a simple class, called ChangePrinter, shown below. Establishing the publisher/subscriber protocol between Point and ChangePrinter is realized by the connector PubSubConn1.

```
Output package;

class ChangePrinter {
  void public printR() {
     System.out.println("Printer: " + this.toString() +
                  " read access has occurred ..." + \n);
   }
  void public printW() {
     System.out.println("Printer: " + this.toString() +
```

```
                    "␣write␣access␣has␣occurred␣..." + \n);
  }
  void public notifyChange() { System.out.println("CHANGE␣..." );
  }
}

...

Deployment package;
...

connector PubSubConn1 {
  Point is Publisher with
     { changeOp = {set*, get*};}
  ChangePrinter is Subscriber with {
      void subUpdate(Publisher publ) {
         notifyChange();
         System.out.println("on␣point␣object␣" + ((Point) publ).toString());
      }
  }
}
```

In a similar way, the following code, illustrates the deployment of the protocol defined by the PublisherSubscriberProtocol aspect with another application: this time a game package including a class for the TicTacToe game and a class for the graphical interface, GUI, with the latter being an aggregate of two other classes, BoardDisplay and StatusDisplay, as shown in Fig. **??**.

```
connector PubSubConn2 {

  TicTacToe is Publisher with {
    changeOp = {startGame, newPlayer, putMark, endGame}};

  {BoardDisplay, StatusDisplay} is Subscriber with {
       void update(Publisher publ) {
          setGame((Game) publ);
          repaint();
        }
     };
}
```

Both deployments of the publisher/subscriber aspect shown so far, result in the standard publisher/subscriber protocol, i.e., the protocol with one publisher being subscribed by a single set of subscribers, uniformly interested in any change in the publisher's state. One can also realize modified versions of the standard protocol by simply defining new connectors. For instance, the connectors, PubSubConn3 and PubSubConn4, deploy the protocol with the same classes as PubSubConn1, but now two different connection clauses distinguish the notification operations that are invoked on the subscribers depending on which access operations are invoked on the publisher. This corresponds to a variation on the publisher/subscriber pattern, where there are several lists of subscribers for one publisher, distinguished by the events they are interested in.

```
connector PubSubConn3 {

  Point is Publisher with { changeOp = set*;}

  ChangePrinter is Subscriber with {
      void subUpdate(Publisher publ) {
```

```
            printW();
            System.out.println("on point object " + ((Point) publ).toString());
            }
        }
}

connector PubSubConn4 {

  Point is Publisher with { changeOp = get*;}

  ChangePrinter is Subscriber with {
        void subUpdate(Publisher publ) {
          printR();
          System.out.println("on point object " + ((Point) publ).toString());
        }
     }
 }
```

The sets of operations of Point that are mapped to different notification operations of the subscriber participant need not be disjoint. For instance, we may want to distinguish between set operations that affect the x-coordinate, respectively, the y-coordinate of a point. The set(int, int), however, will then fall in both categories. This is expressed by the connectors PubSubConn3.1 and PubSubConn3.2 below.

```
connector PubSubConn3.1 {

  Point is Publisher with { changeOp = {set, setX};}

  ChangePrinter is Subscriber with {
       void subUpdate(Publisher publ) {
          printW();
          System.out.println("on the X coordinate of
                 point object " + ((Point) publ).toString());
          }
     }
}

connector PubSubConn3.2 {
  Point is Publisher with { changeOp = {set, setY}; }
  ChangePrinter is Subscriber with {
      void subUpdate(Publisher publ) {
          printW();
          System.out.println("on the Y coordinate of
                 point object " + ((Point) publ).toString());
          }
   }
}
```

1.5 Activating/Deactivating Aspects

There are two possible levels of applying aspects: object versus class level, which correspond to two different operations in the interface of connectors. Plugging an aspect into an instance of a class is realized by invoking adapt(ClassForParticipant instance) on the corresponding connector. On the contrary, all instances of a class that plays a participant role in an aspect according to the connector c are automatically adapted to their role in the program scope between the pair of invocations c.activate() and c.deactivate(). Object level application is illustrated by the client class (client in the sense that it uses Point ChangePrinter and PublisherSubscriberProtocol) Client1, while class level application is illustrated by Client2 in the following.

```
class Client1 {
  . . .
  public static void main(String[] args) {
     . . .
     Point p = new Point();
     Point p1 = new Point();
     p = PubSubConn3.adapt(p);
     p1 = PubSubConn3.adapt(p1);

     ChangePrinter ch = new ChangePrinter();
     ChangePrinter ch1 = new ChangePrinter();

     PubSubConn3.addSubscriber(ch, p);

     p.setX(12);
     // will print
     //    Printer ChangePrinter1678: write access has occurred ...
     //    in point object Point3578

     p1.setY(26);
     // nothing will be printed

     PubSubConn3.addSubscriber(ch1, p1);
      p1.setY(2);
     // will print:
     //    Printer ChangePrinter1802: write access has occurred ...
     //    in point object Point7382
     ...
     PubSubConn3.remove(p); // p will not react as a publisher anymore
     PubSubConn3.removeSubscriber(ch1, p1);
     ...

   }
  ...
}


class Client2 {
  . . .
  public static void main(String[] args) {
    . . .
    ShowReadWriteAccessesConn2.activate();
    Point p1 = new Point(14, 6)
    Point p2 = new Point(10, 5);
    Line l = new Line(p1, p2);
    p1.setX(10);
    // W will be printed on the screen
    Point p3 = l.getSource();
    // R will be printed on the screen
    ...
    ShowReadWriteAccessesConn2.deactivate();
    p3.getY(10);
```

```
    // nothing will be printed out
    ...
  }
}
```

1.6 Comparison with AspectJ

- less tangled aspects
  - aspects easier to understand
  - aspects easier to reuse
  - aspects easier to modify
- better modularity by separating aspect definition from aspect deployment
- Application and aspects could be even developed by different programmers.
-