Implementing APPCs with Java Beans (preleminary thoughts)

(By Mira, Oct. 30, 1998)

1. Overview

Fig. 1 shows (a) the programming constructs available to the programmer when constructing software with APPCs, (b) their definition time, (c) the respective Java constructs generated by the APPC compiler, and (d) how these generated Java constructs relate to each other. The programming level constructs are presented elsewhere [Mezini & Lieberherr 98]. In the following, we will discuss how these programming level constructs get translated into a network of Java beans.

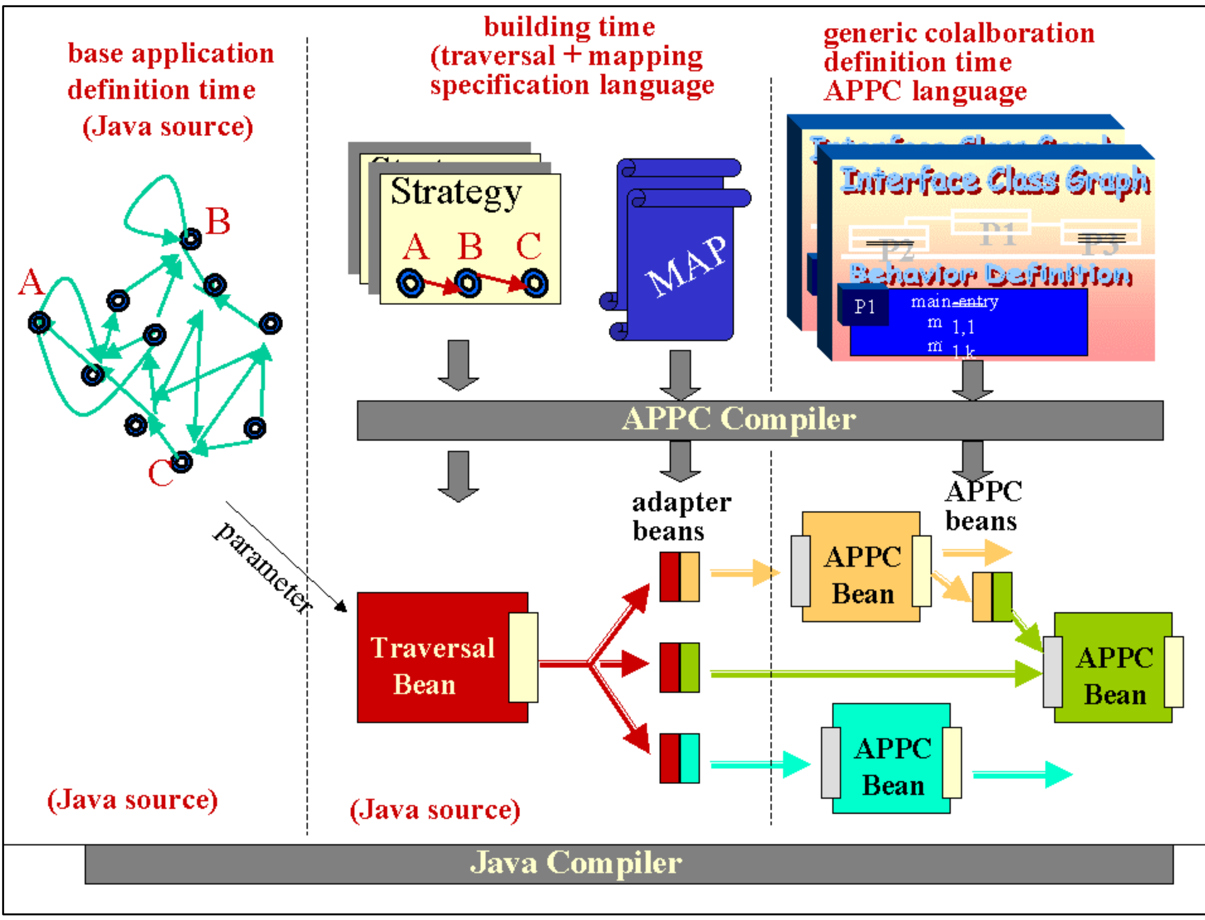


Figure 1

The discussion is illustrated by a simple example: we model two simple functionalities over a recursive containment structure, as defined by the composite pattern [Gamma & al 94].

2. APPCs

. . .

3. Running Example

3.1 Summing and Capacity Checking over a Composite Pattern Structure

Describe the pattern and both functionalities

3.1 APPCs for Capacity Checking

Let's write APPCs for the capacity checking example. Note, I have implemented Summing directly for recursive containment hierarchies, instead of implementing it for flat hierarchies and having the Initial (or Difference) component as it has been the case up to now. I think writting the summing functionality specifically for recursive structures is more natural: it accounts for the fact that we are working with a recursive structure. Also, we use traversals in the implementation of the APPC.

```
APPC Summing {

    ICG:  Item =  Simple | Composite.
          Composite = 0..* Item.
          Simple = <summableValue> float.

    Behavior {
        float total = 0;
        from Composite to Simple {

            Composite {
                    float myTotal;
                    void before {myTotal = total};
                    float after {return total - myTotal};
                } // Composite
             Simple {
                    void at {total += summableValue}
                 } // Simple
            } // traversal
      } // Behavior
    } // Summing


APPC LimitCheckingDelta {

      ICG: Composite = <limit> float <actual> float.

       Behavior {
            Composite {
                boolean at {
                   if (limit > actual) return true    else return false;
                          }
```

```
                    } // Composite
               } //Behavior

          } // LimitCheckingDelta
```

Some notes on the above APPC based implementation:

- The ICG lays down the ***„Input Ports „*** of the APPC, i.e. what needs to be connected to other outputs – **summableValue** in this case. There are two alternatives here:

    1. We have syntax for explicitly denoting the inputs, e.g. a keyword ***expected*** before the declaration of <summableValue>. This will explicitly distinguish them from links between nodes in the ICG (in this particular example there are no links, but in general we will have libks, e.g. <item> in the Pricing APPC.

    2. We make the silent convention that any variable in the ICG with a type that is not one of the nodes of the ICG is something expected, an input. This seems preferable to me since it requires less syntax. This is the assumption made in the implementation above.

- How to specify the ***„Output Ports „***of the APPC? In the previous version of this document, we had syntax for that – the keyword ***provided*** was put before the definition of Composite::after. The other alternative is to make the silent convention that everything returned is an output. This is preferrable because of less syntax (Karl also preferred this one). That's why, I have removed in this ***provided*** version. The compiler will take care that the result of the **after** entry at Composite in both APPCs above will be provided to evrybody outside that wants to hear these return values.

- **at** - means I don't care when exactly, can be before or after. Could remove it and work only with before and after. Later you will find a note suggesting that when connecting interfaces at building time it might be quite useful to have **at** besides **before** and **after.**

4. APPC-Beans: Hand-Compiled Code

4.1 Summing APPC

Given the Summing APPC, the compiler will produce something like the following. Please note that I haven't strictly followed Java syntax. It's just for giving you an idea. And I want to give you an idea of what I am thinking about as soon as possible – don't want to wait for getting the syntax right. Will polish it further to get clean Java syntax. (Wouldnt complain if you do it ☺)

```
package Summing {

class InputSimpleEvent {                                              // Note 1

        float summableValue;
        void setSummableValue(float val)
                        { summableValue = val; }
        float getSummableValue()
                        {return summableValue;}
}  // InputSimpleEvent

class OutputCompositeEvent {                                          // Note 2

         float sum;
         float getSum() {return sum;}
         void setSum(float value) {sum = value;}
   }
                                                                      // Note 3

class SummingBean extends TraversalAPPCBean implements InputEventListener {

         Stack nodePath = new Stack();                                // Note 4
         float total = 0;

         void beforeComposite() {                                     // Note 5
                       nodePath.push (new Float(total));
            }

          void atSimple(InputSimpleEvent ev) {
                        total += ev.getSummableValue(); }

           void afterComposite() {                                    // Note 6
                     OutputCompositeEvent compEv =  new
                               OutputCompositeEvent(total - ((Float)nodePath.pop()).getFloat());
                      listeners.afterComposite(compEv);
              }
     } // SummingBean

  interface InputEventListener extends Global.OutputEventListener{    // Note 9
            void beforeComposite();
            void atSimple(InputSimpleEvent);
            void afterComposite();
        }

  interface OutputEventListener {
            void afterComposite(OutputCompositeEvent);
        }
   } // package Summing


package Global {
```

```
class APPCBean implements GenericTraversalEventListener, OutputEventListener {

    MapObject map;      // will be initialized at building & mapping time

    private Vector listeners = new Vector();

    void traversalEvent(TraversalEventObject obj) { }                    // Note 7

    public void addOutputEventListener(OutputEventListener  list) {      // Note 10

            let flag be the result of checking whether it is feasible to
            do the connection based on the input event interface
            of list and map;

            if (flag = OK ) listeners.addElement(list);
      }

    public void removeOutputEventListener(OutputEventListener  list) {
            listeners.removeElement(list); }
  }

 interface OutputEventListener { };                                       // Note 8

 interface GenericTraversalEventListener {

    void traversalEvent(TraversalEventObject);
  }

class TraversalEventObject {

    . . .
  }

class TraversalBean {

    private Vector appcListeners = new Vector();

    void addGenericTraversalEventListener(GenericTraversalEventListener) {
            . . . check compatibility . . . }

    void removeGenericTraversalEventListener(GenericTraversalEventListener) {
            . . .}


    . . .
 }

} //Global
```

Notes:

1. **SimpleInputEvent** is the event object expected to be passed along the **atSimple(SimpleInputEvent)** event that is being listened at by the SummingBean (SummingBean implements **atSimple**). The implementation of this event is extracted from the expected interface of the Summing APPC. We know that at simple we expect an event that transmits **summableValue. Note** : in this version I have removed the instance variable **node** of type **Simple** from the definition of SimpleInputEvent. All what Summing APPC is interested in, is a certain summable value of CCG classes that implements ICG's Simple. This value will be provided wrapped in a **SimpleInputEvent**. In general may be sevral values that are wrapped together (see details on LimitCheckingDelta). More details about who creates and initializes those parameter objects.

2. For the same reasons as in 1., I have removed the instance variable node from the definition of OutputCompositeEvent. If another APPC needs some information from the CCG object, it will bind one of its inputs to the CCG class(es) and an adapter will be created as a result responsible for retrieving this information from the CCG object passed by the traversal and passing it in the form of ICG objects further. More on that below

3. The class InputCompositeEvent has been removed. When at a CCG node corresponding to ICG's Composite, we don't need any additional information except for the fact that we are at such a node (before or after it).

4. Since the Summing APPC declares a local variable for each composite and the ICG is about a recursive structure, the compiler deduces that it will use a stack for storing the values of the local variables in the internal composite nodes in a containement path. This is what nodePath is about

5. The parameter of the type **InputCompositeEvent** has been removed– see 3

6. **InputCompositeEvent** parameter has been removed – see 3. Beside that we don't pass the CCG node corresponding to ICG's Composite further, as we did before.

7. The **GenericTraversalEventsListener** interface is part of a global package. This is a very Generic listener interface we might want to use simply for fooling builder tools. The appc bean (by being a subclass of **APPCBean**) pretends to implement the **GenericTraversalEventsListener** interface simply for fooling the builder so that the latter would allow the connection to CCG traversals. During this connection, however, the system will perform extensive retrospection in order to match the output interface of the traversals to the real input interfaces of the APPCs (the „real" input interface of the Summing interface is in facts **InputEventListener** and not **GenericTraversalEventListener**. This matching behind the scenes will be partly the responsibility of the traversal beans and partly the responsibility of adapter beans that are created and initialized at connection (or building) time. At least one of them will check. Currently, we let it open, where the checking does exactly take place and put some pseudo-code for checking in both. (Need to decide)

8. It seems like we are loosing all type checking here. Again, this is just for fooling the builder. The packages extend this interface and extensive checking should happen at building time to make sure that interfaces really agree on their signatures while adapting their names. This is done by the register functionality in **APPCBean**.

9. The compiler extracts the input and output interfaces of the beans so that this information is immediatelly available at building time for retrospection.

10. Having the parameter of the add/remove listener methods be an empty interface means that any APPC can be a listener of any other APPC. This will allow appc beans to be put together as far as the builder tool is concerned. Interface checking does, however, happen at connection time when we have the map instance variable of an appc bean initialized. This is done by the within the add/remove methods.

4.2 LimitCheckingDelta APPC

Let us now consider what the compiler will produce for the second APPC.

```
package LimitCheckingDelta {

  class InputCompositeEvent {

      float actual;
      float limit;
      float getLimit() {return limit;}
      float getActual() {return actual;}
      void setLimit(float);
      void setActual(float);
    }

  class OutputCompositeEvent {
      boolean overdrawn;
      float overdrawn() {return overdrawn;}
    }

class LimitCheckingDeltaBean extends TraversalAPPCBean implements
                                                           InputEventListener {

     void atComposite(InputCompositeEvent compEv) {
          float limit = compEv.getLimit();
          float actual = compEv.getActual();
          boolean overdrawn = true;
          if (limit >= actual) overdrawn = false;
          OutputCompositeEvent outputEv = new
                        OutputCompositeEvent(overdrawn);
          listeners.atComposite(outputEv);
        }

    } // LimitCheckingDelta Bean
```

```
interface InputEventListener extends Global.OutputEventListener{
        void atComposite(InputCompositeEvent);
    }

interface OutputEventListener {
        void atComposite(OutputCompositeEvent);
    }

} // package LimitCheckingDelta
```

/ Note

Notes:

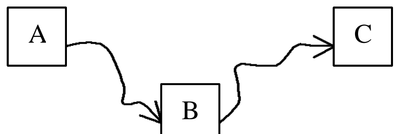
The **InputEventListener** of the **LimitCheckingDelta** states that **LimitCheckingDeltaBean** hears at **atComposite** events. That is, the input interface of **LimitCheckingDeltaBean** is compatible with output interfaces that declare either a **beforeComposite** or an **afterComposite** event. What if an output interface has both? The default connection in this case is **afterComposite** . It seems to me that having **at** is useful. If we didn't have **at** in the limit checking example, we would have to replace **Composte::at** in the **LimitCheckingDelta** with **Composite::after**. This sounds not very intuitive to me, because this would suggest that there is a constraint as when to compare the two values we expect from **Composite** nodes – after having visited the node. When I tightly couple the existence of **LimitCheckingDelta** with the **Summing** scenario we have here, this constraint does indeed make sense.

However, if I forget about the fact that we are going to use LimitCheckingDelta with **Summing**, and rather consider the **LimitCheckingDelta** APPC in isolation, it can be decribed as follows: „when at a **Composite** node, this APPC simply compares two values that are expected to be provided by the composite node. There is no restriction there that the comparison should happen after traversing a **Composite** node's substructure. One could for instance use **LimitCheckingDelta** also for the following functionality: *„traverse an object structure and when at nodes of a certain type – to be bound to Composite - compare two values and make further traversing dependent on the result of this comparison"*. Based on the discussion above, it makes sense to have **at** and be able to connect it to either **before** or **after**.

5. Traversal Beans and Traversal Objects

Given a concrete CCG and a traversal over this CCG, the compiler will create a traversal bean, **TB**. A traversal bean is an instance of a predefined class, called **TraversalBean**, defined in the „global package" of the system shown together with the generated code for the **Summing** APPC. Traversal beans encapsulate the code for traversing an object structure. They factor the traversal code generated from a traversal specification out in a single place (class) rather than having it spread arond all classes in the strategy graph. The code in a traversal bean not only does traversing but also fires events at each step during this traversing. I can roughly think of two alternatives for implementing the traversal beans:

1. Have the class **TraversalBean** that implements the default listener register functionality, and generate a subclass of **TraversalBean** for each strategy specification **S**. E.g. assuming **s = from A to C**



The traversal subclass generated for S would look as follows:

```
class S-TraversalBean {

    void traversal(A aObject) {
        -  fire traversalEvent(event object telling: „starting“);
        -  fire traversalEvent(event object telling: „before aObject of type A“);
        -  fire traversalEvent(event object telling:
                   „planning to go over edge (aObject:A, bObject:B)“;
        -  fire traversalEvent(event object telling:
                          „before edge (aObject:A, bObject:B)“);
        -  traverse(aObject.b);
        -  fire an event object telling that we are after edge (A, B);
        -  fire traversalEvent(event object telling: „after A“);
    }

    void traverse(B bObject) {
        . . .
        . . .
        traverse(bObject.c);
        . . .
        . . .
    }

    void traverse(B bObject) {
        . . .
        . . .
        fire a finish event
    }

}
```

2. Have a single class **TraversalBean** that implements the following
   a) Store the computed traversal graph in an instance variable
   b) Given this graph and a source object traverse the object structure starting at source according to the graph

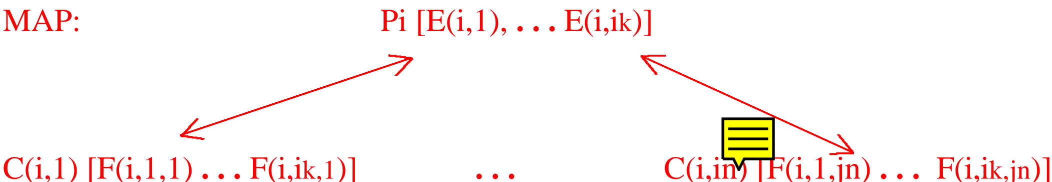
The first alternative generates a lot of code, but is expected to be more efficient; for the second alternative the opposite is true. Instead of generateing a subclass of **TraversalBean**, a new instance of it will be created for each traversal in the second case. We can ignore for the moment how we implement traversal beans. What matters for the discussion here is the fact that a **TB** will fire the standard (generic) event **traversalEventTraversalEventObject travObj)** in each traversing step.

TraversalEventObject is also a predefined type (global package) – the superclass of eventually several types, such as **NodeEvent**, **EdgeEvent**, ... (others?). For the moment, we ignore details about the implementation of **TraversalEventObject** as well. We simply assume that traversal event objects encapsulate the entire information about the traversal state that might be needed by the APPCs, such as e.g., the node, or edge object the traversal is at,

whether the traversal intends to go over an edge, whether it is before, or after this node or edge, the class of the node being visited, etc.

5. Adapter Beans

Now, if the traversals fire only generic events, who fires specific events such as **atSimple(InputSimpleEvent), afterComposite(),** and **beforeComposite()** events that are listened at by the **SummingBean?** Let us consider the problem first in more general terms. Given an APPC that is connected to a traversal over an CCG, say **APPCNameBean** (contained in a package **APPCName** that resulted from the compilation of **APPCName**) and **TB** respectively, we need to adapt the output event interfaces of **TB** to match the input interface of **APPCNameBean.** This is the responsibility of an adapter bean, **APPCNameAB**, which will be automatically created as the result of specifying the ICG-to-CCG mappings by the programmer. Assume that all mapping information is available in **MAP**:



APPCNameAB hears **traversalEventTraversalEventObject travObj)** from **TB** and fires events in the real input interface of **APPCNameBean.** Which event should be fired is decided based on the information encapsulated in **travObj** as well as the information in **MAP. MAP** will somehow be visible to APPCNameAB (an instance available will point to the structure storing **MAP**). An outline of the adaption process is given in the following pseudo-code for the **traversalEventTraversalEventObject travObj)** method in the class **AdapterBean**.

class **AdapterBean** {
 . . .
 void **traversalEventTraversalEventObject travObj)** {

⇒ let **C** be the set of all classes in CCG that are mapped to a participant in ICG (lookup **MAP**);

⇒ let **nodeClass** = **travObj.getNodeClass()** be the class of the object being traversed;

⇒ if (**C.includes**(**nodeClass**)) {

⇒ let **Part** be the participant in **APPCName's ICG** mapped to **nodeClass;**

⇒ let **Before-After** be the string that indicates the state of traversing **nodeClassObj** (whether we are before or after), which is gained by calling **travObj.getStatus();**

⇒ let **messName** be the message name created according to the syntactic pattern **Before-After** + **Part;**

⇒ if (**messName** is included in **APPCName.InputEventInterface) {**

- let **nodeClassObj** = **travObj.getNode**() be the object being traversed;
- let **ICGPartObj = new APPCName.InputPartInterface()** be a new **ICG (Part)object;**
- let **EXP** be the set of inputs expected from the class playing **Part.**
- let **FUNC** be the set of functions in **nodeClass** mapped to **EXP**;
- for all **F** in **FUNC** do {
  - let **Val** be the result of invoking **F** on **nodeClassObj;**
  - invoke **setEXP(Val)** on **ICGPartObj;** }

⇒ invoke **messName** on **APPCNameBean** with **ICGPartObj** as a parameter;

}

. . .

}

<u>Notes</u>:

1. The pseudo-code above assumes that one class in CCG plays the role of a single participant in the ICG. In general **Part** can be a set. However, the pseudo-code above will also work with **Part** being a set.

2. The pseudo-code implicitly assumes that there are only node events. Need to be extended in an equivalent way for edge events.

3. Don't like the fact that reflective facilities (invoke) are being used. Alternatives??

6. Example

Adapters in the Running Example

Summing

When we connect the **summableValue** declared in the **ICG** of the **Summing APPC** to operations of **<SimpleClass>** - where **<SimpleClass>** is a class variable that denotes any CCG class that will be mapped to ICG's **Simple**, an adapter bean, **SummingAB**, will be created between the traversal bean and the instance of **SummingBean** we are connecting to the traversal, **SummingTB**. **SummingAB** hears the standard traversal event **traversalEvent(TraversalEventObject travObj)**. Assuming that **travObj.getNodeClass()** is in **<SimpleClass>,** and that **travObj.getNodeClass() = node** than **SummingAB** :

1. invokes the method mapped to **summableValue** on the **node** to get the value to be summed,

2. initializes a new **Summing.InputSimpleEvent** from the Summing package and fires **atSimple(Summing.InputSimpleEvent)** which is heared by SummingBean with the intialized InputSimpleEvent as a parameter.

. . . continue with a concrete CCG ...

CapacityCheckingDelta

The siple wrapper class **InputCompositeEvent** is built by the compiler based on the ICG definition of the **Composite**. in the **LimitCheckingDelta** APPC. This wrapper class will be used by adapter beans. An adapter bean will be created when **LimitCheckingDelta** Bean (resulting from the compilation of the LimitCheckingDelta APPC above) is connected with the **Summing** Bean and a CCG. During this connection we will map

1. **Composite** in both **LimitCheckingDelta** and **Summing** APPC beans to the same concrete classes in CCG; the class variable **<CompositeClass>** is used below to denote those concrete classes.

2. **actual** to the output of the **Composite:after** method of **Summing**, and **limit** to methods in CCG. As the result an adapter bean, AB, will be automatically created between the CCG and Summing bean on one side and LimitCheckingDelta bean on the other side.

AB will hear **after<CompositeClass>** events from the traversal bean (at is per default mapped to after) and **afterComposite(Summing.OutputCompositeEvent)** event from the **Summing** bean. <CompositeClass> can be any of the classes in CCG that are mapped to **Composite** in ICG. In response to **after<CompositeClass> AB** will

1. Create a new instance of **LimitCheckingDelta.InputCompositeEvent**, **lc-icev**

2. Invoke the method connected to **limit** on **node** and store the result in the variable **limit** of **lc-iecv**.

3. Store the created **lc-ecv** in an internal stack.

In response to **afterComposite(Summing.OutputCompositeEvent)** AB will

1. Get the value of the **myTotal** instance variable from the event parameter.

2. Pop **lc-iecv** from the stack.

3. Store the value of **myTotal** in the actual instance variable of **lc-iecv**.

4. Fire **atComposite(lc-iecv)** to LimitCheckingDelta bean.

7. Register Functionality

This will do the whole interface compatibility checking - is a kind of high level type checking operating at the level of independently compiled components. This will be an intelligent type checker in the sense that interfaces does not necessarily need to aggree on the method names. Names will be matched by the adapters anyway.

. . . to be continued . . . with

- **the register functionality of the traversal beans.,**
- **etc.**

8. Beans as Filters for Transforming CCG Paths to ICG Links

. . . to be continued . . .