JAsCoAP: Adaptive Programming for Component-Based Software Engineering.

Wim Vanderperren
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussel, Belgium
+32 2 629 29 62
wvdperre@vub.ac.be

Davy Suvée
Vrije Universiteit Brussel
Pleinlaan 2
1050 Brussel, Belgium
+32 2 629 29 65
dsuvee@vub.ac.be

1. INTRODUCTION

Adaptive Programming [3] aims at providing support for a very different kind of crosscutting concern than the ones tackled by classical aspect-oriented approaches. When an operation involves a set of cooperating classes, one can either localize the operation in one class or split up its logic among the set of involved classes. Localizing the operation in one class causes hard-coded information about structural relationships, this way violating the Law of Demeter [1]. Spreading the operation among the set of involved classes conforms to the Law of Demeter, but gives raise to crosscutting concerns which obstruct evolution. In order to cleanly encapsulate an operation that involves several cooperating classes, Adaptive Programming introduces adaptive visitors that allow visiting the objects contained within an application, without explicitly specifying the structural relationships among these objects.

JAsCo [6] on the other hand, is an aspect-oriented extension for Java which is especially tailored to be employed in the context of Component-Based Software Engineering (CBSE). CBSE advocates low coupling between components and high cohesion of single components [7]. The JAsCo language introduces two new entities, namely *aspect beans* and *connectors*. An aspect bean allows describing crosscutting concerns independently of concrete component types and APIs. JAsCo connectors on the other hand are used for deploying one or more aspect beans within the concrete application at hand. JAsCo connectors also allow managing the combined aspectual behavior of the instantiated aspect beans in a fine grained manner.

Although Adaptive Programming is originally designed for Object-Oriented Software Engineering, its ideas can also be reused within a Component-Based context. Currently available Adaptive Programming realizations, such as DJ [4], DemeterJ [2] or DAJ [5] however are not very suitable to be employed within CBSE. Even though adaptive visitors are independent of the architecture of the application at hand, they still refer to specific component types and APIs, rendering a visitor not as reusable as required by CBSE. To this end, we propose to recuperate the context-independency idea promoted by JAsCo within adaptive programming, this way making adaptive visitors suitable to be employed within CBSE. Furthermore, currently available Adaptive Programming realizations provide little support for specifying complex combinations among several collaborating adaptive visitors in order to execute their behavior simultaneously. Also here, the JAsCo ideas are able to contribute, as JAsCo allows expressing complex combinations among independently specified aspect beans.

In the next section, we show how the ideas of Adaptive Programming and JAsCo can be combined in order to make Adaptive Programming fit into the Component-Based world. We illustrate how adaptive visitors can be implemented by means of JAsCo aspect beans in order to improve their reusability. Afterwards, we demonstrate how the behavior of several adaptive visitors can be combined making using of JAsCo precedence and combination strategies. Finally, we present our conclusions.

2. JASCOAP

2.1 Aspect beans as adaptive visitors

An adaptive visitor is very similar to a set of related advices as it is able to group several before, after and around methods that need to be executed whenever a corresponding component type is visited. Therefore, it seems natural to employ a regular JAsCo aspect bean as a kind of abstract and loosely coupled adaptive visitor. Figure 1 illustrates the implementation of the *DataStorePersistence* aspect bean, which allows capturing an incremental backup of data objects.

```
1  class DataStorePersistence {
2
3    hook Backup {
4
5      Backup(triggeringmethod(..args)) {
6        execute(triggeringmethod);
7      }
8
9      isApplicable() {
10       //returns true if changed since last visit
11     }
12
13     before() {
14       ObjectOutputStream writer = …
15       writer.writeObject(getDataMethod(calledobject));
16     }
17
18     public abstract Object getDataMethod(Object c);
19   }
20
21 }
```

Figure 1: DataStorePersistence AspectBean that specifies a reusable backup aspect

The aspect bean contains one hook, the *Backup* hook (line 3 till 19). The constructor of this hook (line 5 till 7) specifies that the behavior of the hook should be performed whenever the concrete method, bound to the *triggeringmethod* abstract method parameter, is executed (line 6). An *isApplicable* method is employed (line 9 till 11) which returns true if the state of the object, on which *triggingmethod* is executed, has changed since it was last visited. The before method (line 13 till 16) serializes the visited object to file using the abstract method *getDataMethod*. This abstract method (line 18) is responsible for fetching the data from the current object related to the method call and needs to be implemented in the connector. Notice that the *DataStorePersistence* aspect bean does not refer to specific component types and APIs. As a result, the aspect bean remains completely independent and reusable.

```
1  traversalconnector BackupTraversal(
2    "from system.Root to *") {
3
4    DataStorePersistence.Backup hook = new
5      DataStorePersistence.Backup(visiting DataStore) {
6
7        public void getDataMethod(Object obj) {
8          return (DataStore)obj,getData();
9        }
10   };
11
12   hook.before();
13
14 }
```

Figure 2: BackupTraversal traversal connector.

In order to deploy the *DataStorePersistence* aspect bean as an adaptive visitor, a new kind of connector is introduced: a *traversal connector*. A traversal connector instantiates one or more aspect beans as adaptive visitors onto a traversal strategy. Figure 2 illustrates a traversal connector that instantiates the *DataStorePersistence* aspect bean (line 4 till 10) upon the "*from system.Root to **" traversal strategy (line 1 till 2). The *visiting* keyword allows declaring on which specific type of objects, encountered during the traversal, the behavior of the hook needs to be performed, in this case, *DataStore* objects. As a result, the object structure of an application is traversed as specified by the traversal strategy "*from system.Root to **" and the *before* advice of the *Backup* hook is triggered each time a *DataStore* object is encountered. Likewise to a regular JAsCo connector, the *getDataMethod* abstract method is implemented in order to fetch the data from the visited *DataStore* objects (line 7 till 9).

```
1  public void backup(system.Root mySystemRoot) {
2
3    Connector myBackup = BackupTraversal.getConnector();
4    myBackup.traverse(mySystemRoot);
5
6  }
```

Figure 3: Invoking the BackupTraversal connector.

Traversal strategies need to be invoked explicitly in order to start the traversal. Figure 3 illustrates how the traversal specified in the *BackupTraversal* connector is explicitly invoked (line 4).

2.2 Combinations among Adaptive Visitors

Currently available Adaptive Programming realizations provide little support for specifying complex combinations between several collaborating adaptive visitors. The precedence and combination strategies offered by the JAsCo connector language can however be employed when adaptive visitors are implemented as aspect beans.

```
1  traversalconnector BackupFileLoggerTraversal (
2    "from system.Root to *") {
3
4    DataStorePersistence.Backup hook1 = …
5    Logger.FileLogger hook2 = …
6
7    logger.before();
8    backup.before();
9    addCombinationStrategy(new TwinComb(hook1,hook2));
10
11 }
```

Figure 4: Connector with explicit combinations.

Figure 4 illustrates the implementation of a traversal connector that instantiates the *Backup* hook and the *FileLogger* hook, which logs backup actions, upon the same traversal strategy. The *BackupFileLoggerTraversal* traversal connector allows to explicitly control the precedence of both hooks when they visit the same object. In this case, the before behavior method of the *logger* hook is triggered prior to the before behavior method of the *backup* hook (line 7 till 8). In addition, a log should only be kept of those objects that have been saved to file. For these specific kinds of interactions between hooks, combination strategies can be employed. A combination strategy is a kind of filter which acts on the list of applicable hooks. In this case, the *TwinComb* strategy specifies that the behavior of the *FileLogger* hook should only be performed, if the behavior of the *BackUp* hook was also executed.

3. CONCLUSIONS

This paper illustrates how the ideas behind Adaptive Programming and JAsCo can be combined in order to make Adaptive Programming fit into the Component-Based world. Adaptive visitors are described by means of traditional, context-independent JAsCo aspect beans which are deployed making use of JAsCo traversal connectors. As a result, adaptive visitors, implemented as aspect beans, are now truly reusable as no specific component types and APIs are hard coded into the visitor itself. In addition, the behavior of several adaptive visitors, implemented as aspect beans, can easily be combined into one traversal connector in order to visit the same traversal strategy simultaneously. JAsCo precedence and combination strategies can be employed here in order to describe complex interactions between several adaptive visitors applied upon the same traversal strategy.

4. REFERENCES

[1] Lieberherr, K. and Holland, I. *Assuring Good Style for Object-Oriented Programs.* IEEE Software, pages 38-48., September 1989.

[2] Lieberherr, K and Orleans, D. *Preventive Program Maintenance in Demeter/Java*. In Proceedings of International Conference of Software Engineering (ICSE), pp. 604-605, 1997.

[3] Lieberherr, K., Orleans, D. and Ovlinger, J. *Aspect-Oriented Programming with Adaptive Methods.* Communications of the ACM, Vol. 44, No. 10, October 2001.

[4] Orleans, D. and Lieberherr, K. DJ: *Dynamic Adaptive Programming in Java.* In Proceedings of Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns, Kyoto, Japan, September 2001.

[5] Sung J. and Lieberherr, K. DAJ: *A Case Study of Extending AspectJ.* Northeastern University Technical Report NU-CCS-02-16, 2002. Available at: http://www.ccs.neu.edu/research/demeter/biblio/DAJ1.html

[6] Suvee, D., Vanderperren, W. and Jonckers, V. *JAsCo: an Aspect-Oriented approach tailored for Component Based Software Development.* In Proceedings of the second International Conference on Aspect-Oriented Software Development. Boston, USA, March 2003.

[7] Szyperski, C. *Component Software - Beyond Object-Oriented Programming.* Addison-Wesley / ACM Press, ISBN 0-201-17888-5, 1998.