

Implementing Aspectual Collaborations with AspectJ

Pengcheng Wu*

College of Computer Science
Northeastern University
360 Huntington Avenue, 161CN
Boston, MA 02115, USA
wupc@ccs.neu.edu

October 15, 2001

Abstract

Aspect-oriented Programming proposes a technique to implement cross-cutting concerns in a modular way, but most work has been concentrated on the class level. Aspectual Collaborations proposes an AOP technique on a higher level, i.e., on class collaboration level. This paper presents one implementation of Aspectual Collaborations that automatically translates the Aspectual Collaborations' descriptions to AspectJ code by which the cross-cutting collaboration concern is added to the actual classes in a modular way.

1 Introduction

Aspect-oriented Programming (AOP) has been proposed as a technique for improving separation of concerns in software [4]. Separation of concerns and modularity are at the heart of the programming process. Concerns are abstract conceptual units in which we decompose a given problem [6]. Without tool support for separating concerns, ad-hoc implementations of concerns may scatter over multiple classes

*Work supported by the National Science Foundation under grant number CCR-0098643 and by DARPA and BBN under contract number F33615-00-C-1694

and the code may tangle a lot, which makes the programs difficult to understand, maintain and evolve.

To implement any non-trivial concern, programmers usually have to design and implement many classes which collaborate with each other. And it is highly desirable that we can reuse some general concern implementations in different scenarios. AOP proposed the idea of implementing cross-cutting concerns at the class level. We believe the same idea should also make sense at a higher level, e.g., the collaboration of classes. Aspectual Collaborations (henceforth AC) satisfy this need. AC is a description of class collaboration which can be reused in many scenarios in which the actual classes/methods will play the roles of classes/methods in the collaboration. AC also provides the binding mechanism to express the role playing. Although the target language of AC is Java, the idea also applies to other Object-Oriented programming languages.

This paper is to present one of the implementations of AC undergoing at Northeastern University. We have implemented AC by translating the collaboration description and the binding into AspectJ code. To accomplish the translation, we use DemeterJ and DJ, which are adaptive programming tools in Java [1] [5]. The outline of this paper is as follows. In Sec. 2 we present the key ideas of AC and its syntax, Sec. 3 demonstrates the usefulness of AC by applying it to an example program, Sec. 4 discusses the implementation strategies, Sec.5 lists some related work, and Sec. 6 talks about conclusion and future work.

2 Key Ideas and Syntax of Aspectual Collaborations

The most important concepts of AC are *collaboration* and *adapter*. An AC consists of *zero* or *more* collaborations and *zero* or *more* adapters.

2.1 Collaboration

The collaboration part is just very similar to UML Collaboration [3], but we are using extended Java language syntax to represent the collaboration of classes. The objective of defining a collaboration in AC is to reuse it in users' scenarios. A collaboration consists of *zero* or *more* classes working together to achieve some functionalities, those classes are called *participant classes* in AC, whose roles are expected to be taken by actual classes during deployment. The methods of classes in the collaborations may or may not be fully implemented, they could be provided outside from the actual classes or they could somehow provide extensions to the functionalities of methods in actual classes. Collaboration provides the following ways to describe how classes or methods to be mapped to actual classes:

- By default, the fields and methods defined by a collaboration class will be introduced to the actual classes that will take its role when deployed. During deployment, appropriate name mappings between collaboration namespace and actual classes namespace will be done.
- A collaboration class may declare *expected* methods which don't have implementation and whose implementations are expected to be provided by the actual classes that will take its role.
- A collaboration class may define *enhance* methods whose implementations add more behavior to methods of the actual classes that will take its role.
- A collaboration class may be added to the base program as a whole new class with appropriate name mappings having been done.

Before we present the syntax of collaboration, let's take a look at an example of a collaboration description to get some intuition. In network software domain, propagating messages from a sender node to all of the receiver (could also be a sender) nodes that can be recursively reached from the sender is a very common requirement. Such kind of requirement also emerges in other domains, such as passing an administrative order to all of the staffs through an organization hierarchical structure, traversing a tree structure object to execute visitor methods on each node, etc.. So it does make sense to describe this propagation pattern as an AC to make it reusable. Figure 1 is an example conforming to the syntax of AC.

As you can see from the collaboration example, three participant classes, namely *Data*, *Sender* and *Receiver* are involved. *Data* introduces sequencing concern into the classes playing its role, *Sender* and *Receiver* collaborates with each other to recursively propagate the *Data* object to all of the reachable *Receiver* objects. *Sender* introduces a new method *sendData(Data d)* into the classes playing its role and relies on the playing classes to provide *allReceivers()* implementation; *Receiver* enhances the behavior of methods of its playing classes by defining an *enhance* method *receiveData* in which some code will be executed after the execution of *enhanced* method. *expected*, *enhance* and *enhanced()* are all newly introduced AC keywords to Java syntax. Figure 2 shows a part of syntax of collaboration part of AC.

The non-terminal *JavaInterface* will derive the standard Java interface syntax which has more rules and is not listed here. The *ExtendedJavaClass* will derive the standard Java class syntax (not listed either) except the extensions we imposed on it. The extensions we made to standard Java class include two modifiers on method declarations, which are *expected* and *enhance*, and one fixed form of method, i.e.

```

collaboration Propagate {
    class Data {
        private int sequence;
        public void setSequence(int s) { sequence = s;}
        public int getSequence() {return sequence;}
    }

    class Sender {
        int sequence = 0;
        expected java.util.Iterator allReceivers();
        public void sendData(Data d) {
            d.setSequence(this.sequence ++);
            java.util.Iterator receivers = allReceivers();
            while(receivers.hasNext()) {
                Receiver receiver = (Receiver) receivers.next();
                receiver.receiveData(d);
            }
        }
    }
}

class Receiver {
    enhance void receiveData(Data d) {
        enhanced();
        // if the receiver is also a sender, propagate further
        if(this instanceof Sender)
            ((Sender)this).sendData(d);
    }
}
}

```

Figure 1: *Propagate Collaboration*

```

<Collaboration> ::= "collaboration" <CollaborationName> "{"
                    <Interfaces> <Classes> "}" .
<CollaborationName> ::= Identifier .
<Interfaces> ::= nil | <Interface> <Interfaces> .
<Interface> ::= <JavaInterface> .
<Classes> ::= nil | <Class> <Classes>
<Class> ::= <ExtendedJavaClass>

```

Figure 2: *Collaboration syntax*

enhanced(). As you can see from the propagate example, *expected* modifier specifies that the method doesn't have an implementation in the collaboration and expects the playing classes to provide their implementations to be used with other collaboration constituents; *enhance* modifier specifies that the behavior logic provided by the collaboration will be used to enhance the method implementations in playing classes, the *enhanced()* keyword is used to refer to the invocation of the original methods and it can appear at any place in the method body where a method invocation can take place.

2.2 Adapter

Adapter is the binding mechanism in AC to map the names in base program to the names in a collaboration. By mapping an actual class, say A , to a participant class, say A' , A will automatically have the newly introduced fields and methods of A' , some methods of A will be enhanced by methods of A' which have the *enhance* modifier, and A must provide an implementation for a method of A' which has the *expected* modifier. There is also such a situation that some participant classes in the collaboration are necessary to complete a collaborative task but users may not be able to find their counterparts in the base program, then *Adapter* provides the export mechanism to export those participant classes to the base program. Figure 3 is the syntax of *Adapter*. To save space, all non-terminal symbols which are not defined in the syntax are defined as *Identifier* by default.

ClassPlay is used to let the actual classes in base program to play the roles of participant classes in the collaboration; *ClassExport* is used to export the participant classes from the collaboration which have no images in base program. *FieldExport* enables users' choice of field names during deployment. Corresponding to the different ways of declaring a method in collaboration, *Adapter* also provides

```

<Adapter> ::= "adapter" <AdapterName> "{" <ParticipantGraphList>
           "}" .
<ParticipantGraphList> ::= nil
                        | <ParticipantGraph> ";"
                        <ParticipantGraphList> .
<ParticipantGraph> ::= <ClassPlay> | <ClassExport> .
<ClassPlay> ::= <QualifiedClassName> "PlayedBy" <ClassName>
              <FieldMethodMapList> .
<ClassExport> ::= "ClassExport" <QualifiedClassName> "As"
                <ClassName> .
<FieldMethodMapList> ::= nil | <FieldMethodMap>
                    <FieldMethodMapList> .
<FieldMethodMap> ::= <FieldExport>
                    | <MethodExport>
                    | <MethodProvide>
                    | <MethodEnhance> .
<FieldExport> ::= "FieldExport" <FieldName> "As"
                <FieldExportName> .
<MethodExport> ::= "MethodExport" <MethodName> "As"
                 <MethodExportName> .
<MethodProvide> ::= "MethodProvide" <MethodName> "With"
                  <MethodRealName> .
<MethodEnhance> ::= "MethodEnhance" <MethodRealName> "With"
                   <MethodName> .
<QualifiedClassName> = <CollaborationName> "." <ClassName> .

```

Figure 3: *Adapter syntax*

```

<AdaptedCollaboration> ::= "Collaborations" "{" <CollaborationList> "}"
                          "Adapters" "{" <AdapterList> "}"
<CollaborationList> ::= nil | <Collaboration> <CollaborationList>
<AdapterList> ::= nil | <Adapter> <AdapterList>

```

Figure 4: *Adapted Collaboration syntax*

three ways to do method mapping. *MethodExport* exports the regular methods in classes of the collaboration to their playing classes; *MethodProvide* lets the playing classes to provide their implementations for the methods declared as *expected* in collaboration; *MethodEnhance* specifies which methods in the playing classes will be enhanced by the *enhance* methods in classes of the collaboration. An example of *Adapter* will be shown in Sec. 3.

An *Adapter* is the representation of a scenario in which the playing classes will collaborate with each other in the way defined by the collaboration. To make the mappings reasonable, in one *Adapter*, the mapping from classes in collaboration to actual classes should be a function, namely a class in collaboration can have at most one image in actual classes. Otherwise, the mappings should be separated into different adapters.

2.3 Adapted Collaboration

In principle, collaborations and adapters can be written and compiled separately so that the collaborations can be reused many times. In this implementation, we simplify the problem by assuming collaboration will always be attached with some adapters, which is called *Adapted Collaboration*, and separate compilation will be provided in future work. In an adapted collaboration, there could be several collaborations in the collaboration part and several adapters in the adapter part. Figure 4 is the syntax of *Adapted Collaboration*.

3 An Example

Before we discuss our implementation strategies, let's take a look at an example to demonstrate the usefulness of AC and the way to use AC.

Figure 5 is the class diagram of a company management system. The company is organized as a tree structure. Every Employee except the top manager must have

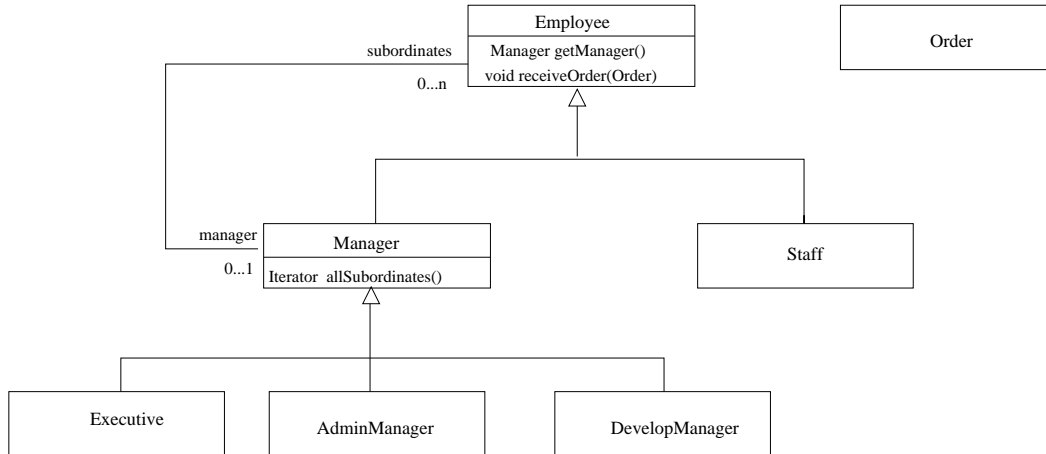


Figure 5: *Class Diagram of Company Management System*

one and can only have one Manager, a Manager may have *zero* or *more* subordinate Employee; The employee who is not a manager is called a Staff, and there are three kinds of different managers, which are Executive, AdminManager and DevelopManager. Employee may receive orders from its direct or indirect manager and execute the orders by method *receiveOrder(Order o)*. But currently the classes do not support the order propagation from a Manager to all of his subordinates (direct or indirect) through the management tree, which does match the propagate pattern well we mentioned earlier. Figure 6 shows the code to adapt the collaboration with the classes in the company management system using the *Adapted Collaboration*.

In the code, *Data* of Propagate collaboration will be played by *Order*, since we are going to propagate administrative orders in this example; *Sender* will be played by *Manager* (only Manager has the right to send out an Order to the Employee), *Manager* provides its *allSubordinates* method to replace *allReceivers* and the newly introduced method *sendData* is exported to *Manager* as *sendOrder*, so *Manager* gets this new functionality; the receiver in this situation could be a *Staff* object, or it could be another *Manager* object, so it has to be their super class, i.e., *Employee* to play *Receiver* here, the *receiveData* of *Receiver* then will enhance *receiveOrder* so that the orders can be propagated recursively along the tree structure.

We rely on AspectJ to introduce new fields and methods to existing classes and to impose new behavior logic to methods. In Sec. 4, we will show the generated AspectJ code from this example.

```

Collaborations {
  collaboration Propagate {
    // same as Figure 1.
  }
}
Adapters {
  adapter adapter1 {
    Propagate.Data PlayedBy Order;
    Propagate.Sender PlayedBy Manager
    MethodProvide allReceivers With allSubordinates
    MethodExport sendData As sendOrder;
    Propagate.Receiver PlayedBy Employee
    MethodEnhance receiveOrder With receiveData;
  }
}

```

Figure 6: *Adapt Collaboration Propagate with Company Management System's Classes*

4 Implementation Strategy

As mentioned earlier, our implementation is a source code level translation strategy, i.e., we try to translate the AC representation to AspectJ code, so the questions are:

- How should we do the parsing and name mapping on the AC so that the translated AspectJ code will refer to the correct class names, method names and field names in base program;
- Since the translation is automatic, how should we map different patterns of AC representation to their appropriate AspectJ code patterns.

We address the two questions separately.

4.1 Parsing and Name Mapping

We are using DemeterJ and DJ to handle with parsing and name mapping issues. DemeterJ is very powerful in generating parser from grammar representations so that we can use the parser to construct a syntax tree against the AC input stream.

DJ is a Java class library that simplifies navigation through object structures [5]. It supports traversal on object structures directed by traversal strategies and users can define the computation they want to execute during traversal in their visitor classes which are very like *Visitor pattern* [2]. We make use of these DJ's capabilities to do name mapping processing to generate AspectJ code while traversing the syntax trees constructed by the parser.

We implement a *TranslateVisitor* class in which *before* and *after* methods, whose signatures conform to the predefined DJ node visitor [7] and edge visitor [8] interfaces, are defined on all kinds of syntax tree nodes and edges between nodes that directly derive keywords, delimiters, literal values or identifiers. These methods will be executed automatically during traversal on the syntax tree if the type of the node or types of the end nodes of the edge being traversed match the methods' signatures. Depending on the nodes or the edges, the extended Java syntax introduced by AC will be translated to the appropriate AspectJ syntax and stored into persistent storage. The name mappings only occur on *Identifier* nodes or *Identifier* related edges, according to the mapping information collected from the *Adapters*.

The traversal order on the AC syntax tree is as follows: the outer loop is iterating over the adapters, after the translator gets all the mapping information from one adapter, it then enters the inner loop which iterates on the classes in the collaboration adapted by the adapter and traverses on each class with the *TranslateVisitor* object plus the mapping information. After the outer loop exits, the translation is finished.

4.2 Mapping AC to AspectJ patterns

¹ What are the aspects in AC? The aspects are the implementations of functionality extensions to actual classes by playing corresponding roles in the collaborations. It is possible that an actual class may play multiple roles in one adapter or in multiple adapters. To organize the aspects more clearly, our implementation constructs an aspect for each actual class playing one role in an adapter, e.g., if a class plays two roles in one adapter or one role in two adapters, two aspects will be generated. Corresponding to the ways a class/method can be mapped to its counterpart in an adapter, we currently use two AspectJ syntax structures to implement the mapping.

- For the newly introduced fields and methods without *enhance* and *expected* modifiers, we just make use of *introduction* syntax of AspectJ to introduce them into the playing class with appropriate name mappings.

¹This implementation is based on AspectJ v0.7 whose syntax is different from that of the current 1.0 version, but the ideas apply.

- For each method having the *enhance* modifier, we will define an *around* advice on the *receptions* or *calls* (if the playing method is *static*) joint points of the playing method; all the keyword *enhanced()* in the body of the method will be replaced to *this.JointPoint.runNext()*. There is a restriction on this kind of mapping, namely, the *static* methods in the collaboration can be played by both static and non-static methods in actual classes, while the *non-static* methods can only be played by non-static methods in actual classes ².
- For each method having the *expected* modifier, we just map its all occurrences of invocation to the provided method name.

Since no aspect has internal state information, all the generated aspects' *of* clause are *of eachJVM()*.

During mapping, to avoid the necessity of building a symbol table, we assume that all the identifiers that have to be mapped have the unique name in the scope of the collaboration. Figure 7 shows the generated AspectJ code from the AC description of Figure 6.

Besides the functionality extension, AC may also impose new type information on the playing classes, e.g, the collaboration may state that *Sender* extends *Receiver*, then the playing classes of *Sender* should also extend the playing classes of *Receiver*. Our implementation works fine in this issue as long as it doesn't introduce multiple inheritance relationship between playing classes, which is not supported by Java and AspectJ .

5 Related Work

Another implementation of AC is undergoing as well[6]. This implementation is not a source code level translation approach, instead it focus on combining compiled Java *.class* files which makes the approach rely only on Java.

As the precursor of AC, *Adaptive Plug&Play*(AP&P) components [9] are designed to facilitate the construction of complex software by making the collaboration explicit and support the evolution of both structure and behavior. The new staff AC adds to AP&P components is AC's mechanism to enhance the behavior of playing classes.

Composition Patterns is another approach to capture reusable patterns of cross-cutting behaviour at the design level[11]. It specifies the design of a cross-cutting

²To simplify the implementation, we assume that in the body of *enhance* method in a collaboration, it always uses *this.v* to refer to the non-static fields or methods and uses *ClassName.v* to refer to the static fields or methods.

```

aspect __Generatedadapter1PropagateData of eachJVM() {
  introduction Order {
    private int sequence;
    public void setSequence(int s) {
      sequence=s;
    }
    public int getSequence() {
      return sequence;
    }
  }
}

aspect __Generatedadapter1PropagateSender of eachJVM() {
  introduction Manager {
    int sequence=0;
    public void sendOrder(Order d) {
      d.setSequence(this.sequence ++);
      java.util.Iterator receivers = allSubordinates();
      while(receivers.hasNext()) {
        Employee receiver = (Employee) receivers.next();
        receiver.receiveOrder(d);
      }
    }
  }
}

aspect __Generatedadapter1PropagateReceiver of eachJVM() {
  introduction Employee {
  }
  around(Employee __obj,Order d) returns void : instanceof(__obj)
    && receptions( void receiveOrder(d)) {
    thisJoinPoint.runNext(__obj,d);
    if(__obj instanceof Manager)
      ((Manager)__obj).sendOrder(d);
  }
}

```

Figure 7: *Generated code for the Company Management System Example*

requirement independently from any design it may potentially cross-cut, and how that design may be re-used wherever it may be required. Instead of using textual form of representation of a design, it uses an extended UML diagram representation. In [10], the mapping from Composition Patterns to AspectJ and Hyper/J is presented. It is not clear if the mapping is automatical or manual.

Elizabeth Kendall [12] also explores the implementation of *Role Model* with Aspect-oriented Programming.

6 Conclusion and Future work

This paper has presented one of the implementations of Aspectual Collaborations undergoing at Northeastern University. We presented key concepts of Aspectual Collaborations and its syntax and demonstrated its usefulness by an example. Our approach implemented a source code level translator from textual form representation of Aspectual Collaborations to AspectJ code to reuse the collaboration in the actual scenarios, which supports implementing cross-cutting concerns at the collaboration level.

There are still following issues to be addressed as future work:

- To save the parsing work and to make the collaboration more reusable, we need to support the separate compilation of *Collaboration* and *Adapter*. The predefined collaboration should be written and compiled independently and stored as an intermediate form, so that the adapter can import it to avoid multiple times of parsing.
- Besides the syntax checking on the collaboration and adapter, we would also like to add some more sophisticated semantic checking to make the adaption safer. For example, by examining the code of collaboration *Propagate*, we can conclude that there is an assumption that the playing class of *Sender* should be a sub-class of the playing class of *Receiver*. This assumption should be explicitly stated in one part of the collaboration as a pre-condition so that we can check the relationship between classes before adaption.
- More ways of describing collaborations and adapters should be developed to make them more flexible and more powerful.

Needless to say, we will keep track of the development of AspectJ to make our implementation be based on the most updated AspectJ version.

7 Acknowledgement

The author wishes to thank Karl Lieberherr for his direction on this work and valuable feedback on this paper.

References

- [1] Demeter home page. <http://www.ccs.neu.edu/research/demeter/>.
- [2] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [3] Martin Fowler with Kendall Scott . *UML Distilled Second Edition* . Addison-Wesley, 1999 .
- [4] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mike Kersten, Jeffrey Palm, and William Griswold. An Overview of AspectJ. In Jorgen Knudsen, editor, *European Conference on Object-Oriented Programming*, Budapest, 2001. Springer Verlag.
- [5] Doug Orleans and Karl Lieberherr. DJ: Dynamic Adaptive Programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of Cross-cutting Concerns* , Kyoto, Japan, September 2001. Springer Verlag. 8 pages.
- [6] Karl Lieberherr, Johan Ovinger, Mira Mezini and David Lorenz . *Modular Programming with Aspectual Collaborations*. Technical Report NU-CCS-2001-04, College of Computer Science, Northeastern University, Boston, MA. <http://www.ccs.neu.edu/research/demeter/papers/publications.html> .
- [7] DJ Manual. <http://www.ccs.neu.edu/research/demeter/DJ>.
- [8] DJ Edge Visitor and ContextVisitor Manual. <http://www.ccs.neu.edu/home/wupc/html/alpha.html>.
- [9] Mira Mezini and Karl Lieberherr . *Adaptive Plug-and-Play Components for Evolutionary Software Development*. Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices, Oct., 1998 .
- [10] Siobhan Clarke and Robert Walker . *Mapping Composition Patterns to AspectJ and Hyper/J*. ICSE 2001, Workshop on Advanced Separation of Concerns in Software Engineering, 2001.

- [11] Siobhan Clarke and Robert Walker . *Composition Patterns: An Approach to Designing Reusable Aspects.* in *Proceedings of the 23rd International Conference on Software Engineerin*, May, 2001.
- [12] Elizabeth A. Kendall . *Role Model Designs and Implementations with Aspect-oriented Programming.* in *Proceedings of the 1999 ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 1999 .