Coupling Aspect-Oriented and Adaptive Programming

Karl Lieberherr and David H. Lorenz

College of Computer & Information Science
Northeastern University
360 Huntington Avenue 161CN,
Boston, MA 02115, USA
{lieber,lorenz}@ccs.neu.edu
April 5, 2003

Abstract. Aspect-oriented programming (AOP) enables the programmer to modularize concerns that cut across other concerns. Adaptive programming (AP) enables the programmer to practice concern-shy programming. A program is concern-shy if it hides the details of a certain concern it cuts across, and as a result exhibits adaptive behavior. AP can be viewed as an instance of AOP in several ways, and vice-versa, AOP can be viewed as an instance of AP in several ways. In this chapter, we compare AOP and AP and discuss their integration. The benefits of the integration are: better support for ubiquitous traversal-related concerns and better support for shy aspect-oriented programming. We illustrate the coupling of AOP and AP by describing DJ, a hybrid tool of Demeter and Java, and by describing DAJ, a hybrid tool of Demeter and AspectJ.

1 Introduction

Aspect-oriented programming (AOP) and adaptive programming (AP) are closely related fields. Chronologically, AOP is a younger field than AP. Conceptually, however, the field of general AP methods is a branch of general AOP methods. Both AP and AOP deal with separation of concerns; both aspire a better modularization of otherwise crosscutting concerns, but their perspective is different. In this chapter we explain the close relationship between AOP and AP and lay out various ways in which the two can be combined.

1.1 An AOP perspective on AP

From an AOP perspective, AP is a form of concern-shy programming. A concern is something that the programmer cares about. A program is concern-shy if it relieves the programmer from the details a certain concern, to which the program can then adapt automatically.

The most practiced form of AP is structure-shy programming. Structure-shy programming relieves the programmer from the details of the structural concern by making the behavioral concern adaptive. Without AP, the behavior and the structure of the program are typically tangled. Changes to the structure (e.g., class graph) require changes to the code of the program (e.g., traversal of object structures). Structure-shy AP factors
out the structure concern into an explicit abstraction over the concrete class graph, and then expresses the program in terms of an abstraction over the traversal of object structures. The program becomes more robust to structural changes; and changes to the class graph do not necessarily entail changes to the code. The program is capable of adjusting itself to some changes in the class graph. The program is then said to be adaptive.

The best-known example of structure-shy AP is Demeter. The DemeterJ [18, 24] software allows the programmer to succinctly specify few traversal strategy [19] methods in place of the many methods that would be needed to traverse a complex object structure. These traversal methods are automatically generated at compile-time by a preprocessor. A traversal strategy describes a traversal at a high level, only referring to the minimal number of classes in the program’s object model: the root of the traversal, the target classes, and waypoints and constraints in between to restrict the traversal to follow only the desired set of paths. If the object model changes, often the traversal strategy doesn’t need to be changed; the traversal methods can simply be re-generated in accordance with the new model, and the behavior adapts to the new structure. Programming with traversal strategies is thus known as adaptive programming [16].

One can view an adaptive program as comprising of aspect. The traversal strategy is an aspect; the class-graph is an aspect; and the two are weaved together using the AP library to generate the working program.

1.2 An AP perspective on AOP

From an AP perspective, AOP is fundamentally a form of module-shy programming. Programming is module-shy if the modular structure of the program does not prevent concerns that cut across other concerns to be modularized. The aspect construct provides the programmer with the ability to express even concerns that crosscut the modular structure of the program.

Typically, there are two sources of programming concerns: the problem space and the program space. Problem-space concerns are concerns about the solution of the problem. Program-space concerns are concerns about the structure of the program. Program-space concerns cut across the program-space concerns.

Deciding on the right decomposition is the most crucial program-space concern that the programmer faces. It is a crucial concern because it affects all other problem-space concerns. A good decomposition can minimize scattering and tangling of problem-space concerns, while a poor decomposition can hinder modularization. This is known as “the tyranny of the dominant decomposition”.

AOP reduces the significance of the dominant decomposition by providing aspects that may crosscut the modular structure. AOP factors out the problem-space concerns into aspects, which are shy with respect to the modular structure of the program, rendering the decomposition program-space concern less important.

AOP modularizes problem-space concerns that cut across the program-space. Concerns no longer need to be expressed only within the program’s hierarchical modular decomposition. Concerns that do not fit into the decomposition and crosscut the modular structure can be expressed in aspects without making code within modules tangled. Thus we say that AOP is module-shy.
Shyness  Adaptive programming is about shy programming. AOP applies the concept of module-shyness. AP applies the concept of concern-shyness. AP enables the programmer to practice (structure) shy programming. Programming is shy if it hides details of other concerns (e.g., the structure) it cuts across.

The Law of Demeter has the flavor: A method should not rely on too much information about other classes/objects. Generalized to the concern level we get: A concern should not rely too much on other concerns: in other words, a concern should be shy of other concerns.

Shyness can be defined in many different ways: An implementation of a concern $C_1$ is $C_2$-shy if:

- The $C_1$ implementation relies only on minimal information of $C_2$ implementations.
- The $C_1$ implementation can adapt to small changes in $C_2$ implementations.
- The $C_1$ implementation is loosely coupled with $C_2$ implementations.
- The $C_1$ implementation can work with $C_2^{11}, C_2^{12}, C_2^{13}, \ldots$, which are close or similar to $C_2$ implementations.

AP is concern-shy programming. A special case of concern-shy programming is structure-shy programming where we write concerns that are shy of a graph structure with nodes and edges. Examples of such graph structures are class graphs and call graphs.

An Adaptive View of Aspects  AOP contains multiple instances of AP. Shyness appears in different places in AOP. Observed in different levels. Aspects are module-shy. One can view aspects as adaptive pieces of code. Each aspect is concern-centric, and the aspect weaver puts the aspects together. We cannot eliminate scattering and tangling, but we can minimize it. Adaptiveness helps us address tangling and scattering. The weaver is a mechanism for addressing tangling and scattering. An aspect-shy program is an oblivious program.

2  The Roots of AOP and AP

Several research groups have been working on ideas surrounding AOP before the name AOP was coined. The work on Demeter/AP at Northeastern University is one such early instance of AOP, which deals with concerns that have scattered and tangled implementations.

Clean separation of concerns was the interest of the Demeter research group, even before AOP existed. The idea of separating structural and behavioral concerns was taken a step further in separating synchronization concerns in the form of synchronization patterns and separating marshaling concerns using adaptive parameter passing. (See the chapter by Crista Lopes in this book.)

AP as a name was introduced around 1991. An early definition of AOP from Northeastern without using the AOP terminology is in the AP book[16]. The main message was to minimize dependencies between concern implementations so that a large class of concern implementation modifications should have minimal impact on other concern implementations.
When the collaboration between Northeastern and Xerox started the name “adaptive programming” was replaced and the better term “AOP” introduced, along with a general definition and many more examples of AOP.

The new separate term is a very good one since it allows to distinguish two key ideas: “better modularization of concerns” (key idea behind AOP) and “concern-shyness” (key idea behind AP). Before the introduction of AOP, the concept of AP tried to fill both roles.

Given the new term of AOP, the definition of AP was reworked in terms of AOP and we call it here structure-shy AP. The reason was to avoid confusion and to have a clear distinction between AP and AOP:

**Definition 1.** Structure-shy AP is the special case of AOP where some of the building blocks are expressible in terms of graphs and where the other building blocks refer to the graphs using traversal strategies. A traversal strategy may be viewed as a partial specification of a graph pointing out a few cornerstone nodes and edges. A traversal strategy crosscuts the graphs it is intended for; only mentioning a few isolated nodes and edges.

The definition of structure-shy AP in terms of AOP is intentionally general. Traversal strategies may be viewed as regular expressions specifying a traversal through a graph. (Formally, a traversal strategy for a graph G is a subgraph of the transitive closure of G with some bypassing information added.) Since traversal strategies have many applications, it is likely that many incarnations of AOP will qualify as structure-shy AP.

For example, AspectJ uses static call graphs and their "instances" are dynamic call trees. In AspectJ, we can write a traversal strategy: "from jp to *" in the form `cflow(jp)` and this qualifies as an application of structure-shy AP.

### 2.1 Law of Demeter

An example of the overlap between AP and AOP is apparent in the Law of Demeter. The Law of Demeter [17] is a style rule for OOP whose goal is to reduce the behavioral dependencies between classes. Its primary form says that a method $M$ should only call methods (and access fields) on objects which are preferred suppliers: immediate parts on this, objects passed as arguments to $M$, objects which are created directly in $M$, and objects in global variables (in Java, public static fields). Limiting which methods can call which other methods keeps programmers from encoding too much information about the object model into a method, thus loosening the coupling between the structure concern and the behavior concern.

In order to write code that follows the Law of Demeter in an ideal way, methods whose ad-hoc implementation is scattered across several classes need to be cleanly localized. The result is a clean separation of various behavioral concerns from concerns about the structural information (class graph). Following the Law of Demeter, a programming style rule for loose coupling between the structure and behavior concerns, can result in a large number of small methods scattered throughout the program, which can make it hard to understand the high-level picture of what a program does. Adaptive programming with traversal strategies and adaptive visitors in DemeterJ avoids this problem while even better supporting this loose coupling of concerns.
The drawback of following the Law of Demeter is that it can result in a large number of small methods scattered throughout the program, which can make it hard to understand the high-level picture of what a program does. [29] studied three medium-sized object-oriented systems and found that in all three systems, 50% of the methods were less than 2 C++ statements or 4 Smalltalk lines long. The example in the next section also demonstrates this effect.

2.2 Coupling AOP and AP

AOP and AP have the same benefit list, namely understandability, maintainability, and reusability.

AP can benefit from AOP in that AP capabilities can be more easily implemented with an AOP language than with an OO language. AOP can also benefit from AP. AP helps by decoupling aspects from graph structures. Without AP, AOP would need to violate the DRY principle: Don’t Repeat Yourself.

More specifically, Demeter can help AspectJ by ease of programming in the following ways:

- Parameterize introductions with traversal specifications
- Make type patterns structure-shy
- Visitors are shorter than traversal advice
- Use traversal specifications in dynamic call graph in addition to object graph.

For example, we can use the AP library to improve the expressiveness of cflow-like pointcuts. So when we have a logging aspect in AspectJ:

```java
aspect Logging {  
  pointcut whatToLog(C c) : call (void foo()) && target(c);  
  before whatToLog(C c) { ... };  
}
```

which logs all calls to `foo()` on C-objects, we may want to log only calls to a subset of C-objects:

```java
aspect Logging {  
  pointcut whatToLog(A a, B b, C c) :  
    call (void foo()) && target(c) &&  
    reachable(from (call (* A.bar()) && target(a))  
      via (call (* B.bar()) && target(b))  
    to (call (void foo()) && target(c)));  
  before whatToLog(A a, B b, C c) { ... };  
}
```

Here `reachable` is a new pointcut designator:

```java
reachable(from jp1 via jp2 to jp3);  
```

describes the subset of join points `jp3` that are reachable from `jp2` and that are in turn reachable from `jp1`. 

```java
aspect Logging {  
  pointcut whatToLog(A a, B b, C c) :  
    call (void foo()) && target(c) &&  
    reachable(from (call (* A.bar()) && target(a))  
      via (call (* B.bar()) && target(b))  
    to (call (void foo()) && target(c)));  
  before whatToLog(A a, B b, C c) { ... };  
}
```

which logs all calls to `foo()` on C-objects, we may want to log only calls to a subset of C-objects:
reachable() can be viewed as a special case of the if pointcut. It expresses a traversal constraint in terms of the dynamic call graph.

A common way to achieve structure-shyness is to use predicates that select nodes and edges either using local information or using graph connectivity. Prominent examples are the AspectJ poincuts such as this, target, args, call, execution that select points in the call graph. Another prominent example are the traversal strategies in Demeter that select nodes and edges based on graph connectivity. The cfow construct of AspectJ falls into this category. And in Demeter we use traversal strategies to select nodes and edges in object graphs that are constrained by class graphs.

The following tree shows different forms and uses of concern-shyness.

```
closest
   concern-shy
   structure-shy
      local predicates
      call graph
      class graph
      connectivity-based predicates: strategies
      call graph
      class graph
```

3 Dynamic Adaptive Programming

DJ is a pure-Java package for adaptive programming that allows traversal strategies to be constructed and interpreted dynamically at run-time. Traversal strategies can be used with adaptive visitors or in a generic programming style by adapting them to the Java Collections framework. The DJ package makes heavy use of Java reflection and we give the highlights of this implementation.

With the addition of reflection to Java [9], it became possible to interpret a traversal strategy at runtime. DJ [22] is a pure-Java package that provides this capability. This makes it easier to add traversal strategies to a Java program, because there is no need to modify the compilation process to run the preprocessor or to convert the source code to the Demeter input language syntax. Instead traversal strategies can simply be expressed as Java strings in ordinary Java code, or even constructed dynamically from an external source not known at compile time.

3.1 Example

The example domain for this paper will be that of processing XML Schema definitions [5]. A simple task that one might want to implement is checking a schema for undefined types. This task involves two traversals of the object structure representing the schema definition: one to collect all the types defined in the schema, and one to check each type reference to see if it’s in the set of defined types. Figure 1 shows a UML class diagram that represents a small subset of the XML Schema definition language; figure 2 and the related listings show the Java code for these classes, along with the methods for these two traversals. The methods `getDefinedTypeNames`
and `getUndefinedTypeNames` on class `Schema` are the public interfaces to the traversals, and the methods `addDef` and `addUndef` do the actual traversal, building up the sets of type names.

Fig. 1. UML diagram for XML Schemas.

```
abstract class SchemaItem {
    void addDef(Set def) { }
    void addUndef(Set undef, Set def) { }
}

abstract class TypeDef extends SchemaItem {
    Attribute attrs[];
    void addDef(Set def) { }
}
```
import java.util.*;

class Schema {
    Attribute attrs[];
    SchemaItem items[];

    public Set getDefinedTypeNames() {  
        Set def = new HashSet();
        addDef(def);
        return def;
    }

    public Set getUndefinedTypeNames() {  
        Set undef = new HashSet();
        Set def = getDefinedTypeNames();
        addUndef(undef, def);
        return undef;
    }

    void addDef(Set def) {  
        for (int i = 0; i < items.length; i++)
            items[i].addDef(def);
    }

    void addUndef(Set undef, Set def) {  
        for (int i = 0; i < items.length; i++)
            items[i].addUndef(undef, def);
    }
}

Fig. 2. Java code.
for (int i = 0; i < attrs.length; i++)
    attrs[i].addDef(def);
}

class Attribute {
    String name;
    String value;
    void addDef(Set def) {
        if (name.equals("name")
            def.add(value);
    }
    void addUndef(Set undef, Set def) {
        if (name.equals("type")
            && !def.contains(value))
            undef.add(value);
    }
}

class SimpleType extends TypeDef {
}

class ComplexType extends TypeDef {
    SequenceGroup content;
    AttributeDecl adecls[];
    void addDef(Set def) {
        super.addDef(def);
        content.addDef(def);
    }
    void addUndef(Set undef, Set def) {
        content.addUndef(undef, def);
        for (int i = 0; i < adecls.length; i++)
            adecls[i].addUndef(undef, def);
    }
}

class SequenceGroup {
    Attribute attrs[];
    ElementDecl edecls[];
    void addDef(Set def) {
        for (int i = 0; i < edecls.length; i++)
            edecls[i].addDef(def);
    }
    void addUndef(Set undef, Set def) {
        for (int i = 0; i < edecls.length; i++)
            edecls[i].addUndef(undef, def);
    }
}
abstract class Decl extends SchemaItem {
    Attribute attrs[];
    void addUndef(Set undef, Set def) {
        for (int i = 0; i < attrs.length; i++)
            attrs[i].addUndef(undef, def);
    }
}

class AttributeDecl extends Decl {
}

class ElementDecl extends Decl {
    TypeDef typeDef;
    void addDef(Set def) {
        if (typeDef != null)
            typeDef.addDef(def);
    }
    void addUndef(Set undef, Set def) {
        if (typeDef != null)
            typeDef.addUndef(undef, def);
    }
}

Note that the Law of Demeter is strictly followed: each method only refers to fields defined on the same class. However, the overall algorithm is lost in the noise of all the traversal methods. The actual functional behavior is split between the Schema and Attribute classes. Moreover, even though each method only refers to local fields, deciding whether to traverse a field requires knowledge of the overall class structure: for example, in SequenceGroup, the addDef method only needs to traverse the edecls field because an element declaration may include a type definition; if the object model were extended so that an attribute declaration could also include a type definition, the addDef method in ComplexType would have to be changed to traverse the adecls field, even though nothing about ComplexType itself changed.

Another way of implementing this example would be to use the Visitor design pattern [7], by creating two classes TypeDefVisitor and DeclVisitor, moving the traversal methods into visit methods on those classes, and making subclasses overriding visit(Attribute) to perform the behavior of checking for defined and undefined types. While this would eliminate the scattering of traversal methods across the class structure, the same set of traversal methods would need to be written, and they would still need to be modified when the object model changes.

3.2 Adaptive programming with DJ

DJ is a library of classes that make traversals like the previous example much easier to define, understand, and maintain. Figure 3 shows an alternate implementation of the Schema class that defines the two traversals succinctly using the ClassGraph and Visitor classes from the edu.neu.ccs.demeter.dj package.
import edu.neu.ccs.demeter.dj.*;

class Schema {
  Attribute attrs[];
  SchemaItem items[];

  static final ClassGraph cg = new ClassGraph();

  public Set getDefinedTypeNames() {
    final Set def = new HashSet();
    cg.traverse(this,
        "from Schema via ->TypeDef,attrs,* to Attribute",
        new Visitor() {
          void before(Attribute host) {
            if (host.name.equals("name"))
              def.add(host.value);
          }
        });
    return def;
  }

  public Set getUndefinedTypeNames() {
    final Set def = getDefinedTypeNames();
    final Set undef = new HashSet();
    cg.traverse(this,
        "from Schema via ->Decl,attrs,* to Attribute",
        new Visitor() {
          void before(Attribute host) {
            if (host.name.equals("type")
                && !def.contains(host.value))
              undef.add(host.value);
          }
        });
    return undef;
  }
}
A `ClassGraph` object is a simplified representation of a UML [2] class diagram; its nodes are types (classes and primitive types) and its edges are (uni-directional) associations and (bi-directional) generalizations.

The default `ClassGraph` constructor builds a graph object using reflection from all the classes in the default package; a string containing a package name can be provided as a constructor argument to build a class graph from another package. The methods `addPackage(String pkgname)` and `addClass(Class cl)` can be used to add other packages and classes to a class graph object. A traversal is done by calling the `traverse` method on a `ClassGraph` object. It takes three arguments: the root of the object structure to be traversed; a string specifying the traversal strategy to be used; and an adaptive visitor object describing what to do at points in the traversal.

A traversal strategy specifies the end points of the traversal, using the `from` keyword for the source and the `to` keyword for the target(s). In between, any number of constraints can be specified with `via` or `bypassing`. The two traversals in figure 3 traverse from `Schema` to `Attribute`; in other words, they visit attributes in a schema, because type names appear in attribute values for both definitions and references. They differ in their constraints: to find the names of types defined by the schema, the first traversal only looks at attributes of type definitions (`TypeDef` objects); to find the names of types referenced by the schema, the second traversal only looks at attributes of declarations (`Decl` objects). The `->TypeDef, attrs,*` syntax is a pattern specifying the set of association edges whose source is class `TypeDef` and whose label (field name) is `attrs`; the asterisk means that an edge in the set can have any target type.

Traversal strategy interpretation is done as described in [19], with a few modifications whose details will be presented in a future paper. The general idea is that at each object in the traversal, those associations (including inherited associations) which can possibly lead to a target object (subject to any constraints specified in the traversal strategy) are traversed sequentially; if an object is encountered which has no possible path leading to a target object, the traversal returns to the previous step in the traversal. For example, in our XML Schema example, the `items` field of `Schema` contains an array of `SchemaItem` objects; this array may contain `TypeDef` objects, since `TypeDef` is a subclass of `SchemaItem`, so the elements of the array are traversed as part of the `getDefinedTypes` traversal. However, some of the elements may be `AttributeDecl` objects, and there is no possible path to a `TypeDef` object; if one of these elements is encountered in the array, it is simply skipped over. The `adecls` field of `ComplexType` is never traversed at all, since it can only contain an array of `AttributeDecl` objects. Note that if the `adecls` field were a `Vector` instead of an array, it could contain objects of any type, and so DJ would have to traverse it in case one of its elements were a `TypeDef` object or some other object that could lead to a `TypeDef`. If parametric polymorphism is added to Java, such as that proposed in [3], this problem will be easier to avoid: the type of `adecls` could be `List<AttributeDecl>` and DJ would know it could avoid it.

An adaptive visitor class is a subtype of the `Visitor` class in the DJ package; it implements the Adaptive Visitor pattern described in [16, pp. 426-427]. The Adaptive Visitor pattern differs from the Visitor pattern as presented in [7] in two ways: only
a minimal set of methods needs to be defined, namely those describing the functional
behavior to be performed at points along the traversal, rather than one method each
for every class in the traversal; and no accept methods need to be defined, nor does
traversal behavior need to be defined in the visitor methods. These two differences result
in a unit of behavior that can adapt both to changes in the object model and changes in
the traversal.

During a traversal with adaptive visitor $V$, when an object $o$ of type $T$ is reached in
the traversal, if there is a method on $V$ named before whose parameter is type $T$, that
method is called with $o$ as the argument. Then, each field on the object is traversed if
needed. Finally, before returning to the previous object, if there is a method on $V$ named
after whose parameter is type $T$, that method is called with $o$ as the argument. The
Visitor subclasses defined inline in figure 3 only define one before method each,
which is executed at Attribute objects, the end point of the traversal.

DJ also provides support for generic programming [23]: the asList method on
ClassGraph adapts an object structure and a traversal strategy into a List, part
of Java's Collections framework [10]. The object structure is viewed as a collection
of objects whose type is the target of the traversal strategy; the collection’s iterator
performs the traversal incrementally with each call to next. Figure 4 shows how to
rewrite the previous example using asList.

DJ also has edge visitor methods that get executed whenever certain edges in the
object graph are traversed (so far, visitor method execution depended only on the class
of the object being traversed).

An edge has the form $n_{l}(source, target)$ or $n_{l}(source, l, target)$. In
the first case the name of the part-of edge is fixed (1), in the second case it is variable.
$n$ is either before or after. around is only available in an alpha version of DJ.

For a part-of edge, the following signatures are matched, in order:

1. $n_{l}(S, T)$
2. $n_{l}(S, Object)$
3. $n_{l}(Object, T)$
4. $n_{l}(Object, Object)$
5. $n(S, String, T)$
6. $n(S, String, Object)$
7. $n(Object, String, T)$
8. $n(Object, String, Object)$

where $n$ starts with before or after. $S$ is the source type of the edge. $l$ is
the label of the edge, and $T$ is the target type of the edge. For example, if an edge
$\rightarrow Employee, salary, Currency$ is traversed, then first the visitor method
before_salary(Employee, Currency)
is invoked, if it exists, followed by
before_salary(Employee, Object),
etc.
import edu.neu.ccs.demeter.dj.*;

class Schema {
    Attribute attrs[];
    SchemaItem items[];
    static final ClassGraph cg = new ClassGraph();
    public Set getDefinedTypeNames() {
        final Set def = new HashSet();        
        List typeDefAttributes =
        cg.asList(this,
            "from Schema via ->TypeDef,attrs,* to Attribute");
        Iterator it = typeDefAttributes.iterator();
        while (it.hasNext()) {
            Attribute attr = (Attribute) it.next();
            if (attr.name.equals("name"))
                def.add(attr.value);
        }
        return def;
    }
    public Set getUndefinedTypeNames() {
        final Set def = getDefinedTypeNames();
        final Set undef = new HashSet();
        List declAttributes =
        cg.asList(this,
            "from Schema via ->Decl,attrs,* to Attribute");
        Iterator it = declAttributes.iterator();
        while (it.hasNext()) {
            Attribute attr = (Attribute) it.next();
            if (attr.name.equals("type")
                && !def.contains(attr.value))
                undef.add(attr.value);
        }
        return undef;
    }
}

Fig. 4. Using the collection adaptor asList.
The expressions 1. through 8. are pointcut designators for execution join points of traversals. For example, \( n.l(S, \text{Object}) \) selects all join points during a traversal where we traverse a part-of edge called \( l \) starting from an object of type \( S \) (bound in the first argument) and going to any kind of object (bound to the second argument). Those DJ pointcut designators can only select join points in traversals while AspectJ has a much richer set of join points. The DJ pointcut designators can be simulated in AspectJ using pointcuts such as \( \text{this}, \text{target}, \text{args} \) and \( \text{call} \).

### 3.3 Implementation Highlights

In this section we present some highlights of the implementation of DJ and some examples of interesting uses.

When the `ClassGraph` constructor is called, it creates a graph object containing reflective information about all the classes in a package. In Java, however, there is no way to get a list of all classes in a package; packages are just namespaces, not containers. Moreover, the JVM only knows about classes that have already been loaded, and it only loads classes when they are referenced. Since a class graph might be constructed before many of the classes in the package have been referenced, the constructor has to discover classes some other way: it searches the class path (provided by the JVM as `System.getProperty("java.class.path")`) for all `.class` files in subdirectories corresponding to the package name. For each class file that is found, it calls `Class.forName()` with the class name, which causes the JVM to load the class if it hasn’t already been loaded. If there are classes that need to be added to a class graph that do not exist as `.class` files in the class path, for example if they are loaded from the network or constructed dynamically, they will need to be added explicitly by calling `addClass()`.

A class graph may also be created from another class graph \( G \) and a traversal strategy \( S \), forming the subgraph of classes and edges in \( G \) that would be traversed according to \( S \). This can be used to remove unwanted paths from a class graph, such as backlinks, rather than having to add bypassing constraints to every traversal strategy.

The `traverse` method on `ClassGraph` is implemented in a two-stage process: first, a _traversal graph_ is computed from the class graph and the traversal strategy (which itself is converted into a _strategy graph_, whose nodes are the classes mentioned in the traversal strategy and whose edges each have constraints attached to that leg of the traversal); then, the object structure is traversed, using information from the traversal graph to decide where to go next at each step, and visitor methods are invoked as needed. The traversal graph computation takes time proportional to the product of the number of edges in the class graph and the number of edges in the strategy graph; since the same traversal strategy is often reused multiple times with the same class graph, the traversal graph can be saved and reused without needing to be recomputed every time. The `TraversalGraph` class has a constructor that takes a traversal strategy and a `ClassGraph` object, as well as methods `traverse` and `asList`. The traversal computation algorithm is also available as a separate package, the AP Library [25].

At each step in a traversal, the fields and methods of the current object, as well as methods on the visitor object, are inspected and invoked by reflection. Some of this reflective overhead could be avoided by generating a new class (at run-time) that invokes...
the appropriate fields and methods directly; this is planned for a future addition to DJ. Other applications of partial evaluation to speed up the traversal may be possible as well.

The implementation of \texttt{asList} is somewhat trickier than regular traversal: the list iterator must return in the middle of the traversal whenever a target object is reached, and then resume where it left off when \texttt{next} is called again. An earlier version created an ad-hoc continuation-like object that was saved and restored at each iteration, but this was error-prone and not very efficient; the current version uses a separate Java thread as a coroutine, suspending and resuming at each iteration. An additional provided method \texttt{gather} can be used to copy all the target objects into an \texttt{ArrayList}, which is faster still, but the list returned by \texttt{asList} has the advantage that calls to \texttt{set} on the iterator can replace target objects in the original object structure.

Java’s reflection system, unlike other meta-object protocols [12], has no mechanism for \textit{intercession}: there is no way to make a new subclass of \texttt{Class} that behaves differently for certain meta-operations such as method invocation. However, DJ’s \texttt{Visitor} class does allow a limited form of intercession. It has the method \texttt{before(Object obj, Class cl)} (and corresponding \texttt{after}), which is invoked by the \texttt{ClassGraph.traverse} method at each traversal step; it looks for a method named \texttt{before} with a single parameter whose type is the class represented by \texttt{cl}, and invokes it with \texttt{obj} as argument. This method can be overridden by a subclass to perform more dynamic behavior based on the refined class object of the object being traversed. Figure 5 shows a simple pretty-printing visitor that uses this technique, along with a method on class \texttt{Schema} that uses it.

Note that the \texttt{XMLPrinter} visitor class is generic, in that it makes no mention of any of the XML Schema classes but is parameterized by a mapping of classes to element names.

4 Static Adaptive Programming

DJ shows how AP is best conceptually integrated with Java. Basically, we made the concepts of AP available as Java classes: \texttt{ClassGraph}, \texttt{Strategy} and \texttt{Visitor}.

It is interesting to see how AP is best integrated with AspectJ and this is achieved by the DAJ project [26, 28]. The goals of the AspectJ integration are:

1. keep it as simple as possible to make it easy to use by AspectJ programmers and
2. improve the performance over the Java integration (DJ)
3. make it easy to define domain-specific languages to provide structure-shy representations of objects (as an optional feature).

In DAJ we want structure-shy representation of objects as well as behavior. We focus on structure-shy behavior because structure-shy object representations are described in [16].

4.1 Strategy graph intersection

While in DJ we may work with any number of class graph views, using a strategy to define each view, in DAJ we work only with one main class graph. But this is not a
public XMLPrinter(Map map) {
  this.map = map;
}

Map map;
String indent = "";

public void before(Object obj, Class cl) {
  String elementName = (String) map.get(cl);
  if (elementName != null) {
    System.out.println(indent + "<" + elementName + ">");
    indent += " ";
  }
}

public void after(Object obj, Class cl) {
  String elementName = (String) map.get(cl);
  if (elementName != null) {
    indent = indent.substring(2);
    System.out.println(indent + "</" + elementName + ">");
  }
}

class Schema {
...
void print() {
  Map map = new HashMap();
  map.put(Schema.class, "schema");
  map.put(SimpleType.class, "simpleType");
  map.put(ComplexType.class, "complexType");
  map.put(ComplexTypeContent.class, "complexContent");
  map.put(SequenceGroup.class, "sequence");
  map.put(ElementDecl.class, "element");
  map.put(AttributeDecl.class, "attribute");
  Visitor v = new XMLPrinter(map);
  XSD.cg.traverse(this, "from Schema to *");
}
}

Fig. 5. Using visitor method intercession.
restriction to the expressiveness of DAJ; we compensate the lack of multiple class graph views by making the strategy language more expressive. We add a strategy intersection capability so that we can define a strategy eachFile as follows:

```
declare strategy: down:
    "from * bypassing -> *,parent,* to *";
declare strategy: eachFile:
    "intersect(from CompoundFile to File, down)";
```

down is a strategy that selects only the down links in a recursive data structure by bypassing all parent links. eachFile is a strategy that reaches all File-objects reachable from a CompoundFile-object, but only following down links.

To get the equivalent of

```
cg.traverse(o, whereToGo, whatAndWhenToDo)
```

in DAJ, we introduce a second kind of declaration, called a traversal declaration. It defines a new method using the strategy whereToGo and the class of whatAndWhenToDo:

```
WhatAndWhenToDo
```

```
declare traversal:
    void someName(): whereToGo (WhatAndWhenToDo);
```

### 4.2 Visitor classes

WhatAndWhenToDo is a Java identifier naming a class (declared elsewhere) containing visitor methods which are invoked during the traversal. Arguments to the traversal will be passed to the constructor of the visitor. There are five kinds of visitor methods:

- **void start()** is invoked at the beginning of the traversal.
- **void before(ClassName)** is invoked when an object of the given class is encountered during the traversal, before its fields are traversed.
- **void after(ClassName)** is invoked when an object of the given class is encountered during the traversal, after its fields have been traversed.
- **void finish()** is invoked at the end of the traversal, that is, after all the fields of the root object have been traversed.
- **Object getReturnValue()** is invoked at the end of the traversal, and its value is returned as the result of the traversal (suitably cast to the traversal’s return type).

In the future, all the capabilities in DemeterJ will be added to DAJ. This means that we will have around methods in DAJ visitors.

In the current implementation of DAJ, we have the restriction that traversal and strategy declarations must be put into separate .trv files. This is a small inconvenience and has the advantage that the AspectJ compiler does not need to be modified. After strategy and traversal declarations have been added to AspectJ, it also makes sense to add a new pointcut designator to AspectJ: **traversal(s)** for a traversal strategy **s**.
It selects all join points in the traversal defined by \( s \) and can be freely combined with other pointcut designators.

The implementation of DAJ translates the .cd files to class definitions with parsing methods using the ANTLR tools [27] and it translates the .trv files to AspectJ introductions defining the appropriate traversal methods using the AP Library [20, 25]. It then weaves all the AspectJ files together.

We conclude with a simple aspect that defines two methods `eachFile` and `findDirectory`.

```java
aspect FileSystemTraversals {
    declare strategy: eachFile:
        "intersect(from CompoundFile to File, down)";

    declare traversal:
        void listAll(): eachFile(FileLister);

    declare traversal:
        void findDirectory(Ident target):
            "intersect(from CompoundFile to CompoundFile, down)"
            (DirFinder);

    declare strategy: down:
        "from * bypassing -> *,parent,* to *";
}
```

DAJ has achieved the goals mentioned at the beginning. It is easy for AspectJ programmers to use DAJ: only two new declarations need to be learned, namely strategy and traversal declarations. The implementation is an order of magnitude faster than DJ and class dictionaries have been added as an optional feature.

5 Related Work

The notion of adaptiveness and shyness is linked to the notion of quantification in [6]. Aspect-Oriented Programming is quantification because an aspect works with an entire family of base programs.

DJ is closely related to DemeterJ [24], a preprocessing tool that takes a class dictionary file (containing a textual representation of a UML class diagram, with syntax directives for parsing and printing object structures) and some behavior files (containing regular Java methods to be attached to the classes in the class dictionary, plus traversal method specifications, visitor methods, and adaptive methods that connect a traversal with a visitor class) and generates plain Java code for those classes with traversal methods attached along with a parser and some custom visitors such as for printing, copying, or comparing object structures. Demeter/C++ [15, 21] is a predecessor of DemeterJ with similar capabilities. DJ shares the same traversal strategy language and traversal graph algorithms as DemeterJ, but does no code generation and is a pure-Java library.

Besides being easier to use with existing Java code, DJ has a few other advantages compared to DemeterJ. One is the ability to traverse classes for which the programmer
DJ can traverse public accessor methods, or may even use private methods and fields if the JVM’s security manager allows reflective access to private parts (which is often the case outside of applets). Another feature of DJ which does not exist in DemeterJ is the ability to work with subgraphs of a class graph; in DemeterJ, all traversals are computed in the context of the whole class graph defined in the class dictionary, but in DJ you can create new class graphs by selecting a subgraph with a traversal strategy. In addition, DJ allows components to be more generic, by taking class graphs, traversal strategies, or classes to be visited as run-time parameters. These latter two advantages are due to the reification of concepts which only exist at compile-time in DemeterJ as first class objects in DJ.

An Adaptive Object-Model [30] is an object model that is interpreted at run-time. If an object model is changed, the system changes its behavior. Java’s object model can’t be changed at run-time (other than dynamic class loading) but DJ interprets the object model when doing traversals.

DJ’s Visitor class is similar to the reflective visitor described in [1] and the Walkabout class described in [11]. However, neither of these allows for customized traversals.

Java OQL, is the binding of OQL (Object Query Language) from ODMG 2.0 [4] to Java, treats query specifications much like DJ treats traversal strategy specifications. An OQLQuery object can be constructed from a string describing a query; the query can then be executed by calling the execute() method on the OQLQuery object. Queries are either compiled dynamically at run-time or interpreted. An example of a query is:

```java
OQLQuery query = new OQLQuery
    ("select p.getSpouse from p in persons");
Set spouses = (Set) query.execute();
```

For an adaptive version of OQL, see [8].

DJ has some connections with aspect-oriented programming (AOP) [13]. An adaptive visitor is a specialized aspect: it says what behavior should happen at certain principled points in the execution of a traversal. A traversal strategy can also be considered an aspect: it adds crosscutting behavior whose implementation would ordinarily require scattering methods across the class structure. More details about the aspectual nature of DJ are in [14].

One is the ability to write visitor methods that get executed whenever certain edges in the class graph are executed (currently, visitor method execution depends only on the class of the object being traversed).

6 Conclusion

In this chapter we discussed how AOP and AP are coupled. It turns out that ”shyness” is an important concept in AOP. If aspects are not ”shy” their usefulness greatly decreases.
We have presented DJ, a pure-Java library supporting dynamic adaptive programming. DJ makes it easier to follow the Law of Demeter, loosening the coupling between the structure and behavior concerns and adapting to changes in the object model. It is more flexible and dynamic than the preprocessing approach taken by DemeterJ, by interpreting traversal strategies at run-time and using reflection to traverse object structures with adaptive visitors.

Expression of pointcuts at a higher level of abstraction is an important issue in AOP. Traversal-strategy based pointcuts show one interesting way how this can be accomplished.

AOP, specifically AspectJ, has developed a good model for expressing sets of join points in a call graph in a structure-shy way. Indeed, AspectJ is an excellent language for expressing an object form Law of Demeter checker which is a program that is so structure-shy that it works with any legal Java program.

References

7. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
27. Terence Parr and Russell Quong.
A On the Horizon

What’s next? To be done.

A.1 Components and aspects
A.2 Aspectual polymorphism
A.3 Better join point models (higher level)
A.4 Statically executable advice
A.5 Modules and aspects combined
A.6 Contracts and aspects
A.7 Integrating methods and advice into a simpler aspect model
A.8 Composition of aspects

Composing a visitor with a traversal, the visitor cannot violated the strategy. Functional visitors.

A.9 Distributed systems
A.10 ?
A.11 Debug

How to make AspectJ easier to debug. Paint you program with an aspect. Then that aspect helps you find the place in the program where a change is needed. Using aspects to facilitate code navigation. The role of aspects in the IDE.

A.12 Constraining aspects

The visitor is supposed to advice A visitor is an example of a restricted aspect or a constraints aspects. It can only advice traversal code and only in such a way that the advise does not violate the strategy. It is a controlled aspect. How to do similar things for AspectJ?