CHAPTER 6

Coupling Aspect-Oriented and Adaptive Programming

Karl Lieberherr and David H. Lorenz

Adaptive Programming (AP) is a programming technique that has grown out of the need to write programs that are more flexible [13]. The Law of Demeter (LoD) [new 15] and a specialization, the Law of Demeter for Concerns (LoDC) [14] have been proposed as style rules that promote flexibility. The Law of Demeter for concerns is a style rule for programming that says: Talk only to your friends that share your current concerns. In the context of an object-oriented system, the friends of a method are the

preferred supplier objects: the argument objects, including this, the immediate parts of this and newly created objects in the method. A concern is any issue the designer is concerned with, such as a use case or the synchronization or logging policy for the system. The Law of Demeter is the first part of LoDC: Talk only to your friends, i.e., each method should only send messages to preferred supplier objects. LoD implies LoD. Following the LoD induces Adaptive Programming and following the LoDC induces Aspect-Oriented Programming (AOP) []. Adaptive Programming is technique to cope with the many small methods that need to be written when following the LoD [29].

AP can be viewed as a special case of AOP, and vice-versa. In this chapter we examine the close relationship between AP and AOP and discuss their integration. The integration of AP and AOP produces better support for ubiquitous traversal-related concerns and better support for *concern-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.

6.1. INTRODUCTION

Aspect-oriented programming (AOP) and adaptive programming (AP) are closely related. Conceptually, AP techniques are a subset of general AOP techniques, but chronologically, AP appeared first. Both AP and AOP deal with separation of concerns and both aspire to better modularize otherwise crosscutting concerns. But their perspective is different. AOP enables the programmer to modularize crosscutting concerns. AP enables the programmer to practice concern-shy programming.

6.1.1. Concern-shy programming

A program is *concern-shy* if it abstracts away from some concerns it cuts across. Shy programming builds on the observation that traditional black-box composition is not abstract (disciplined) enough. Conventional black-box composition isolates the implementation from the interface, allowing the implementation to evolve independently of its interface. However, interfaces also evolve [20]. Interfaces become strongly coupled to their clients unless

programmers use considerable self-discipline in coding. This discipline of programming abstracted away from certain parts of the interface is referred to as *shy programming;* shy programming lets the program recover from (or adapt to) interface changes. This principle is similar to the shyness metaphor in the Law of Demeter (LoD [13]; see Section 6.2.3) that argues that since structure evolves over time, it is best to restrict communication to just a subset of the visible objects. The AP principle, contrasted with the principle of black-box abstraction, is stated in Table 6-1.

Table 6-1 Black-box versus AP

Black-box principle: The representation of an object can be changed without affecting clients.

AP principle: The interface of an object can be changed within certain parameters without affecting clients.

6.1.2. Structure-shy programming

Current implementations of AP apply concern-shy programming primarily to structure. Structure-shy programming provides a special-purpose embedded

language that controls the traversal of objects in complex object-graph structures. Because knowledge of the graph structure is confined to small, specific statements in this special-purpose language, programs that use only these statements are immune to most effects of changing the underlying object relationships (i.e., changes to the class graph). For this reason, we sometimes use structure-shy programming as a synonym for AP. We have created several implementations of this technology. The most recent ones, DJ and DAJ, respectively use the reflective mechanisms in Java and the aspect-oriented mechanisms in AspectJ, to search the graph.

The next section presents shyness in human behavior as a motivating metaphor for adaptive behavior in structure-shy programming. Section 6.3 presents DJ, and Section 6.4 presents DAJ. Related works are described in Section 6.5.

6.2. SHYNESS AS A METAPHOR FOR ADAPTIVE BEHAVIOR

Knowledge relationships among software components can be understood by analogy to human organizations. Humans create hierarchies—such as managers, group leaders, and individual workers (**Figure 6-1**)—so that changes in knowledge state follow restricted paths and not everyone has to know about all changes. We relate those restrictions to shyness in human behavior.

This *shyness* has three elements:

- **(a)** Minimizing information sharing,
- **(b)** Minimizing the number of communication channels, and
- **(c)** Reducing the scattering of communications.

The structural alternative is organizational bedlam: having arbitrary people dependent on the work structure of arbitrary other people, requiring

considerable routing and analysis with each change in process or organization.

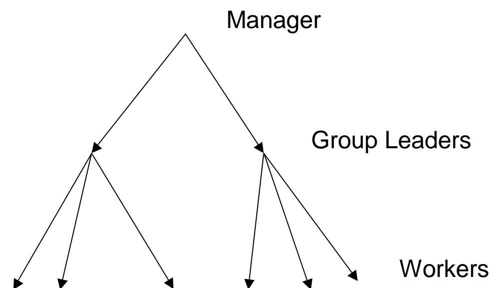

Figure 6-1 Manager, Group leaders, and Workers

Better software design mimics the shyness of good organizations rather than the disorderliness of bad ones. In software systems, an important locus of shared knowledge is the class graph. It is better to have components that rely only on local information in the class graph than global knowledge of its structure. It is better to have *shy* workers and components that minimize the external knowledge of their internal behavior than ones that become trapped into the *doing* of something a particular way because others rely not just on the output but also the process.

Better organizations also minimize communications channels, so that not everyone is bothered by the chatter of routine activities. Similarly, better software systems follow a parallel in the Law of Demeter, which states that an object should “talk” only to closely related objects. The closely related object for a worker is the group leader and not the manager.

Finally, better organizations restructure information so that higher managers deal with summarized and coalesced versions of the raw facts. This allows high-level managers to better understand the many projects they are responsible for and to optimize with respect to the overall system behavior. The orders of high-level managers affect many worker projects. Each order cuts across one or more projects and it is important that those orders are not too tied to the details in the projects. Otherwise, orders are brittle and the managers would have to spend too much time writing and revising them. The same point applies to programming: instead of sending an object a lot of individual requests, it is better to send one complex request that can be understood as a whole and that is not tied to many details in the object. Consider an SQL query as another example of a complex request.

The SQL query abstracts away from the physical database organization but the query optimizer will use it. Therefore, it is more efficient to send one more complex SQL query to a database rather then 50 very simple SQL queries. The query optimizer of the database system can coordinate the responses to the simple queries and the overall response time will be better.

6.2.1. How Shyness Relates to AP

The arguments for AP parallel those for structured organizations. Pure information hiding does not hide enough. Information hiding makes all public interfaces available, but minimizing sharing (a) asserts that only an abstraction of those interfaces should be visible at higher levels. In AP, only high-level information about the class graph is visible at the (adaptive) programming level. This shields the program from many changes to the class graph in the same way as the manager is shielded from many of the changes in the workers' projects. The role of the group leader is played by the glue code that maps high-level information to low-level information and vice-versa (Table 6-2).

Table 6-2. The correspondence between management and programs

Worker	Group Leader	Manager
Class graph	Glue code	Adaptive program

Following the LoD leads to many small methods that scatter and duplicate class graph information. AP allows to minimize the number of communication channels (b) while nevertheless following the LoD.

AP is related to complex requests by offering a declarative way of navigating through objects and executing operations along the way. This is better than sending many small data accessing requests and allows for optimal generation of correct traversal code that contains accidental details from the class graph.

6.2.2. How Shyness Relates to AOP

Aspect-Oriented Programming is about abstractions that cut across multiple modules. It can best achieve its potential if it follows the same principles as AP. Good AOP minimizes information sharing (a) when aspects are only

loosely coupled to base programs. Often good aspect systems provide glue code that maps the aspect to the detailed usage context. AOP is related to complex requests (c) by observing that an aspect is a complex request to modify the execution of a program. These relationships are illustrated in Table 6-3.

Table 6-3. The relationship between management and AOP

Worker	Group Leader	Management
Call graph	Glue code	Aspect

6.2.3. Law of Demeter

An example of the overlap between AP and AOP is apparent in the Law of Demeter. The Law of Demeter [14] is a style rule for OOP whose goal is to reduce the behavioral dependencies between classes. Its primary form asserts 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 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.

To obey the Law of Demeter, 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 can result in a large number of small methods scattered throughout the program. A study of three medium-sized object-oriented systems found that in all three 50% of the methods were fewer than 2 C++ statements or 4 Smalltalk lines long [28]. This 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 providing even better support for loose coupling of concerns.

6.3. REFLECTIVE ADAPTIVE PROGRAMMING WITH DJ

Consider the processing of XML Schema definitions [4]. **Figure 6-2** is an example of an XML schema taken from an aspect-oriented web development system [11]. The schema consists of a sequence of items. Some are type definitions and some are declarations. Verifying that all types used in the schema are either standard or locally defined is a typical consistency check.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:ddd="http://webjinn.org">
 <xsd:element name="structure">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="fields" type="Fields"/>
    <xsd:element name="comment" type="xsd:string"/>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
 <xsd:complexType name="Fields">
  <xsd:sequence>
   <xsd:element name="field" type="Field"/>
  </xsd:sequence>
 </xsd:complexType>
 <xsd:complexType name="Field">
  <xsd:sequence>
   <xsd:element name="attribute" type="Attribute"/>
  </xsd:sequence>
  <xsd:attribute name="name" type="xsd:NMTOKEN"/>
 </xsd:complexType>
 <xsd:complexType name="Attribute">
  <xsd:attribute name="name" type="xsd:NMTOKEN"/>
  <xsd:attribute name="value" type="xsd:string"/>
 </xsd:complexType>
</xsd:schema>
```

Figure 6-2 An example of an XML schema

Figure 6-3 shows a UML class diagram that represents a small subset of the XML Schema definition language. A simple algorithm for checking a schema for undefined types involves two traversals of the object structure representing the schema definition: one to collect all the types defined in the schema, and another to check each type reference to see if it is in the set of defined types.

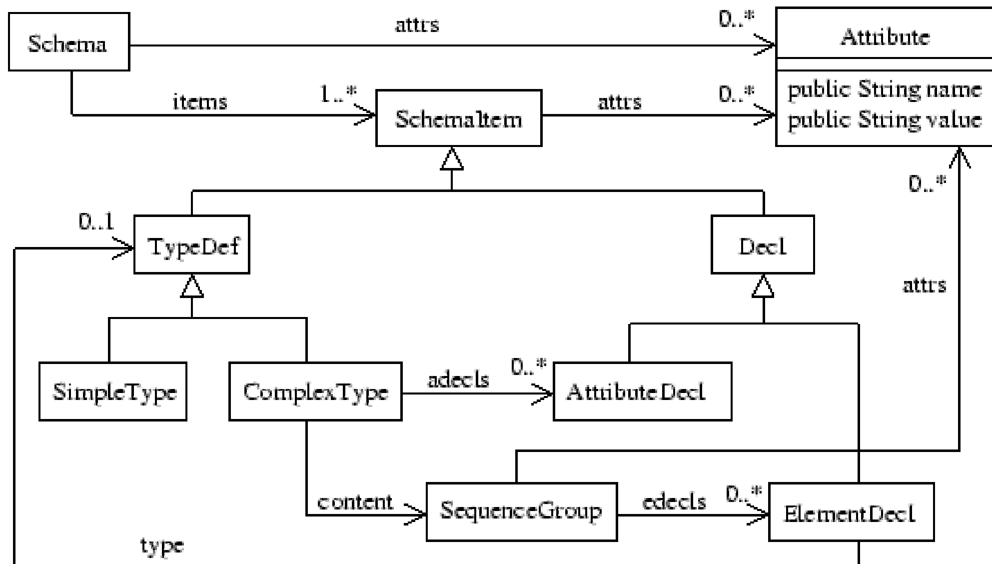

Figure 6-3 Class hierarch for representing XML schema

The actual traversal behavior, however, is quite complex. The functional behavior must be split between the `Schema`, the `Attribute`, and the other classes in order to comply with the Law of Demeter. Moreover, deciding whether to traverse a field requires knowledge of the overall class structure: for example, in `SequenceGroup`, the finding type definitions traversal method only needs to traverse the `edecls` field because an `ElementDecl` element declaration may include a `TypeDef` type definition; if the object model were extended so that an `AttributeDecl` attribute declaration could also include a type definition, the traversal method in `ComplexType` would have to be changed to traverse the `adecls` field, even though nothing about `ComplexType` itself changed.

DJ is a library of classes that make traversals like collecting all defined types and verifying all type references much easier to define, understand, and maintain. **Figure 6- 4** shows an implementation of the Schema class that defines the two traversals succinctly using the `ClassGraph` and `Visitor` classes from the `dj` package.

```
import edu.neu.ccs.demeter.dj.ClassGraph;
import edu.neu.ccs.demeter.dj.Visitor;
```

```
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;
  }
}
```

Figure 6- 4. Using traverse from the DJ library

A `ClassGraph` instance is a simplified representation of a UML [1] class diagram; its nodes are types (classes and primitive types) and its edges are (unidirectional) 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. Calling the traverse method on a ClassGraph object does a traversal. 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 objects. In between, any number of constraints can be specified with via or bypassing. The two traversals in **Figure 6- 4** 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.

Traversals of object structures is done efficiently [15]: for each object in the traversal, associations which can possibly lead to a target object are traversed sequentially (including inherited associations, subject to any constraints specified in the traversal strategy). If an object that has no possible path leading to a target object is encountered, the traversal omits searching it. For example, in the 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`, the elements of the array are traversed as part of the `getDefinedTypeNames` traversal. However, some of the elements may be `AttributeDecl` objects; there is no possible path to a `TypeDef` object. If one of these elements is encountered in the array, it is skipped. The `adecls` field of `ComplexType` is never traversed, 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`. Using the parametric polymorphism of Java 1.5 [2], this problem can be avoided: the type of `adecls` could be `List<AttributeDecl>` and DJ would know it could avoid it.

The anonymous inner visitor class is a subtype of the `Visitor` class in the `DJ` package. During a traversal with a visitor of type *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 6- 4** only define one `before` method each, which is executed at `Attribute` objects, the end point of the traversal.

```
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;
  }
}
```

Figure 6-5 Using the collection adaptor asList

DJ also provides support for generic programming [21]: the `asList` method on `ClassGraph` adapts an object structure and a traversal strategy into a `List`, part of Java's Collections framework [8]. 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 6-5** 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, in our examples visitor method execution depended only on the class of the object being traversed.)

An edge has the form `n l(source, target)` or `n(source, l, target)`. In the first case the name of the part-of edge is fixed (`l`), 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 signatures are matched in the order shown in **Figure 6-6**, 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 "`->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.

```
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)
```

Figure 6-6 The order of signature matching

From an AOP perspective, expressions 1 through 8 in **Figure 6-6** are pointcut designators for execution join points of traversals. For example, `n l(S, 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).

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 `this`, `target`, `args` and `call`.

6.4.1. 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. However, in Java 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 must be added explicitly by calling `addClass()`.

A class graph may also be created from another class graph and a traversal strategy, forming the subgraph of classes and edges in that would be traversed according to that pair. 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 class `TraversalGraph` 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 [22].

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. Applications of pluggable reflection [20] combined with partial evaluation to speed up the traversal may be possible as well. The implementation of `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 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 method, `gather,` can be used to copy all the target objects into an `ArrayList.` This is faster still, but the list returned by `asList` has the advantage that calls to `set` on the iterator can replace target objects in the original object structure.

Java's reflection system, unlike many other meta-object protocols [9], has no mechanism for intercession: there is no way to make a new subclass of `Class` that behaves differently for certain meta-operations such as method invocation [20]. However, DJ's `Visitor` class does allow a limited form of intercession. It has the method `before(Object obj, Class cl)` (and corresponding `after`), which is invoked by the `ClassGraph.traverse` method at each traversal step. It looks for a method named `before` with a single parameter whose type is the class represented by `cl`, and invokes it with `obj` as argument. This method can be overridden by a subclass to perform more dynamic behavior based on the reified class object of the object being traversed.

6.4. ASPECTUAL ADAPTIVE PROGRAMMING WITH DAJ

DJ exemplifies how AP is conceptually integrated with Java. DJ makes the concepts of AP available as Java classes: `ClassGraph`, `Strategy` and `Visitor`. It is interesting to see how AP is integrated with AspectJ. This has been the objective of DAJ project [24, 27].

DAJ achieves a couple of goals. First, it is easy for AspectJ programmers to use AP. Only two new declarations need to be learned to use DAJ, namely strategy and traversal declarations. Second, it improves the performance of AP. The implementation is an order of magnitude faster than DJ and class dictionaries have been added as an optional feature.

6.4.1. Strategy graph intersection

While DJ works with any number of class graph views, using a strategy to define each view, DAJ works with only one main class graph. Making the strategy language more expressive, particularly by adding a strategy intersection capability, compensates for this restriction. For example, we can define a strategy `eachFile` as

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

where strategy `down` selects only the down links in a recursive data structure by bypassing all parent links. Strategy `eachFile` 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, a second kind of declaration, called a traversal declaration, is introduced. It defines a new method using the strategy `whereToGo` and the class of `whatAndWhenToDo`:

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

6.4.2. 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.

Having added strategy and traversal declarations 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 class dictionary files to class definitions with parsing methods using the ANTLR tools [26] and it translates the strategy and traversal declaration files to AspectJ introductions defining the appropriate traversal methods using the AP Library [17, 22]. It then weaves all the AspectJ files together.

In the current implementation of DAJ, we have the restriction that traversal and strategy declarations must be put into separate files. This is a small inconvenience but has the advantage that the AspectJ compiler does not need modification.

6.5. RELATED WORK

The notion of shyness is linked to the notion of quantification [5]. AOP is quantification because an aspect works with an entire family of base programs. DJ is closely related to DemeterJ [23], 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++ [12, 18] 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 over DemeterJ. One is the ability to traverse classes for which the programmer does not have source code, or is not able or willing to modify the source code. For example, one might traverse parts of Java's Swing library of GUI widgets. 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. Another new feature of DJ 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 [29] 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 reflective visitor-beans [19] and `Walkabout` classes [25]. However, neither of those provide a language for customizing traversals. Java Object Query Language (OQL), a binding of OQL from ODMG 2.0 [3] to Java, treats query specifications much like DJ treats traversal strategy specifications [7]. 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:

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

6.6. CONCLUSION

AOP and AP have the same benefit list, namely understandability, maintainability, and reusability. The key idea behind AOP is "better

modularization of concerns"; the key idea behind AP is "concern-shyness." Before the introduction of AOP, AP tried to fill both roles.

Traversal specifications can be applied to graphs other that the object graph. In particular, traversals can be applied to the dynamic call graph in AspectJ. In fact, AspectJ already uses structure-shy regular expressions to specify traversals. 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 programming.

Generally, 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 in that those AP abilities can help in decoupling aspects from graph structures. For example, type patterns are complex pointcut descriptors, which can be made structure-shy by using a traversal specification. Similarly, parameterizing introductions with traversal specifications can help express complex introductions in AspectJ via a single structure-shy expression. Hence, the premise is that both AP is a variety of AOP and AOP is a variety of AP.

In this chapter we discussed how AOP and AP could be coupled. "Shyness" is an important concept not only for AP but also for AOP. When a module M_1 keeps only limited information about another module M_2, we say that M_1 is M_2-shy. If aspects are not "shy" their usefulness greatly decreases. There are three facets to shyness: (1) sharing only limited information, (2) communicating only to a few other modules, and (3) avoiding scattered communications. 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, using interpreting traversal strategies at run-time and 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 excellent for writing an

object-form Law-of-Demeter checker that is so structure-shy that it works with any legal Java program [15].

References

1. BOOCH, G., RUMBAUGH, J., AND JACOBSON, I. 1999. *The Unified Modeling Language user guide*. Addison Wesley, Reading, Massachusetts.

2. BRACHA, G., ODERSKY, M., STOUTAMIRE, D., AND WADLER, P. 1998. Making the future safe for the past: Adding genericity to the Java programming language. In *13th Conf. Object-oriented Programming, Systems, Languages, and Applications (OOPSLA),* (Vancouver). ACM, 183–200.

3. CATTELL, R. G. G., BARRY, D. K., BARTELS, D., BERLER, M., EASTMAN, J., GAMERMAN, S., JORDAN, D., SPRINGER, A., STRICKLAND, H., AND WADE, D. 1997. *The Object Database Standard: ODMG 2.0*. Morgan Kaufmann, San Mateo, California.

4. FALLSIDE, D. C. 2000. XML Schema part 0: Primer. Tech. rep., W3C. Oct. http://www.w3.org/TR/xmlschema-0/.

5. FILMAN, R. E. AND FRIEDMAN, D. P. 2000. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns (OOPSLA),* (Minneapolis). http://trese.cs.utwente.nl/Workshops/OOPSLA2000/papers/filman.pdf

6. GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Massachusetts.

7. HARRISON, C. 1994. AQL: An adaptive query language. M.S. thesis, Northeastern University. http://www.ccs.neu.edu/home/lieber/theses-index.html.

8. JAVASOFT. Collections framework overview. http://java.sun.com/products/jdk/1.2/docs/guide/collections/overview.html.

9. KICZALES, G., DES RIVIERES, J., AND BOBROW, D. G. 1991. *The Art of the Metaobject Protocol*. MIT Press, Cambridge, Massachusetts.

10. KICZALES, G., LAMPING, J., MENDHEKAR, A., MAEDA, C., LOPES, C., LOINGTIER, J.-M., AND IRWIN, J. 1997. Aspect-oriented programming. In *ECOOP'97 Object-Oriented Programming, 11th European Conference,* M. Akşit and S. Matsuoka, Eds. LNCS, vol. 1241. Springer-Verlag, Berlin, 220–242.

11. KOJARSKI, S., LIEBERHERR, K., LORENZ, D. H., AND HIRSCHFELD, R. 2003. Aspectual reflection. In *Software engineering Properties of Languages for Aspect Technologies (SPLAT, AOSD),* (Boston). http://www.daimi.au.dk/~eernst/splat03/papers/Sergei_Kojarski.ps

12. LIEBERHERR, K. J. 1992. Component enhancement: An adaptive reusability mechanism for groups of collaborating classes. In *Information Processing '92, 12th World Computer Congress* (Madrid, Spain), J. van Leeuwen, Ed. Elsevier, 179–185.

13. LIEBERHERR, K. J. 1996. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston.

14. LIEBERHERR, K. J. 2004, Controlling the Complexity of SoftwareDesigns, *International Conference on Software Engineering (ICSE), Edinburgh, Scotland.*

15. LIEBERHERR, K. J. AND HOLLAND, I. 1989. Assuring good style for object-oriented programs. *IEEE Software 6*, 5 (Sept.), 38–48.

16. LIEBERHERR, K. J., LORENZ, D. H., AND WU, P. 2003. A case for statically executable advice: Checking the law of Demeter with AspectJ. In *2nd Int' Conf. Aspect-Oriented Software Development (AOSD),* (Boston), M. Akşit, Ed. ACM, 40–49.

17. LIEBERHERR, K. J. AND PATT-SHAMIR, B. 1997. Traversals of object structures: Specification and efficient implementation. Tech. Rep. NU-CCS-97-15, College of Computer Science, Northeastern University, Boston, MA. Sept. http://www.ccs.neu.edu/research/demeter/AP-Library/.

18. LIEBERHERR, K. J., PATT-SHAMIR, B., AND ORLEANS, D. Traversals of object structures: Specification and efficient implementation. *ACM Transactions on Programming Languages and Systems, vol. 26, no. 2, pages 370-412, 2004*.

19. LIEBERHERR, K. J., SILVA-LEPE, I., AND XIAO, C. 1994. Adaptive object-oriented programming using graph-based customization. *Comm. ACM 37*, 5 (May), 94–101.

20. LORENZ, D. H. 1998. Visitor beans: An aspect-oriented pattern. In *Workshop on Aspect Oriented Programming (ECOOP),* (Brussels). http://trese.cs.utwente.nl/aop-ecoop98/papers/Lorenz.pdf

21. LORENZ, D. H. AND VLISSIDES, J. 2003. Pluggable reflection: Decoupling meta-interface and implementation. In *Int'l Conf. Software Engingeering (ICSE),* (Portland, Oregon). ACM, 3–13.

22. MUSSER, D. R. AND STEPANOV, A. A. 1994. Algorithm-oriented generic libraries. *Software Practice and Experience 24*, 7 (July), 623–642.

23. ORLEANS, D. AND LIEBERHERR, K. 1999. AP library: The core algorithms of AP. Tech. rep., Northeastern University. May. http://www.ccs.neu.edu/research/demeter/APLibrary/.

24. ORLEANS, D. AND LIEBERHERR, K. 2001. DemeterJ. Tech. rep., Northeastern University. http://www.ccs.neu.edu/research/demeter/DemeterJava/.

25. ORLEANS, D. AND LIEBERHERR, K. 2003. DAJ: Demeter in AspectJ. Tech. rep., Northeastern University. Jan. http://www.ccs.neu.edu/research/demeter/DAJ/.

26. PALSBERG, J. AND JAY, C. B. 1998. The essence of the visitor pattern. In *22nd Int'l Computer Software and Applications Conference (COMPSAC),* (Vienna). IEEE, 9–15.

27. PARR, T. AND QUONG, R. 1995. ANTLR: A predicated-LL(k) parser generation. *Software Practice and Experience 25*, 7, 789–810.

28. SUNG, J. 2002. Aspectual concepts. M.S. thesis, Northeastern University. http://www.ccs.neu.edu/home/lieber/theses-index.html.

29. WILDE, N. AND HUITT, R. 1992. Maintenance support for object-oriented programs. *IEEE Transactions on Software Engineering 18*, 12 (Dec.), 1038–1044.

30. YODER, J. W. AND RAZAVI, R. 2000. Metadata and adaptive object-models. In *ECOOP 2000 Workshop Reader*. LNCS, vol. 1964. Springer-Verlag, Berlin, 104–112.