Dynamic Adaptive Programming in Java

Jianjun Zhao
School of Software,
Shanghai Jiao Tong University

Some slides adapted from Prof. Karl Lieberherr’s talk
Overview

- Adaptive Programming
- Dynamic Adaptive Programming in Java (DJ)

Karl Lieberherr (Northeast University)
The Law of Demeter

- **A method should only talk to its friends**
  - Arguments
  - Part Objects
  - Objects created by it
  - Global variables
- **Problem:**
  - This leads to many very small methods scattered throughout the system
  - Makes it hard to understand what the program is doing
- **Traversal strategies are the solution to this problem**
- **Demeter = Greek Goddess of Agriculture (grow software from small blocks)**
Adaptive Programming: Demeter

- Adaptive Programming: references to other objects are replaced by traversal strategies for the class graph
- Methods become less brittle with regard to changes in the class structure
Adaptive Programming: Demeter (cont.)

- Instance of AOP [Lieberherr92]
- Aspects are traversal strategies
- Separate the program text and the class structure
  - Program is independent of class graph
- Accomplish tasks by traversals
  - Specification for what parts of received object should be traversed
  - Code fragments for what to execute when specific object types are encountered
Object Traversals

• The heart of Adaptive Programming is object traversals
• Traversal code is tedious to write
• Traversal code should not obstruct the view of the “real” program logic
AP Example: UML Class Diagram

```
BusRoute
  -------
     |   |   |
     v v v
BusList
  -------
     |   |   |
     v v v
Bus

BusStopList

BusStop

PersonList

Person

busStops
buses
0..*

waiting

0..*
```
Collaborating Classes

Find all persons waiting at any bus stop on a bus route

BusRoute \rightarrow BusStopList

BusList \rightarrow Bus

BusStop \rightarrow PersonList

OO solution: one method for each red class
class BusRoute {
    BusStopList busstops;
    void printWaitingPassengers () {
        busstops->printWaitingPassengers (); }
}
class BusStopList {
    BusStop stops[];
    void printWaitingPassengers () {
        for (int i = 0; i < stops.length; i++)
            stops[i].printWaitingPassengers (); }
Java Solution (cont.)

class BusStop {
    PersonList waiting;
    void printWaitingPassengers () {
        waiting.print ();
    }
}

class PersonList {
    Person people[];
    void print () {
        for (int i = 0; i < people.length; i++) people[i].print ();
    }
}

class Person {
    String name;
    void print () { System.stdout.println (name); }
}
Demeter Approach

• Devise a traversal strategy
• Specify code for different types of objects reached on a traversal

• Example: code prints name if object is a Person
• Independent of class graph
Traversals Strategy

First try: from BusRoute to Person

Diagram:
- BusRoute
- BusList
- Bus
- Person
- BusStopList
- BusStop
- PersonList

Relationships:
- BusRoute → BusList
- BusList → Bus
- Bus → Person
- BusStopList → BusStop
- BusStop → PersonList
- busStops
- buses
- waiting
- passengers
- 0..*
Traversing Strategy

From BusRoute through BusStop to Person

- BusRoute
  - buses
  - busStops
  - BusList
  - passengers
  - Bus
  - 0..*
- BusStopList
  - 0..*
- Waiting
- PersonList
  - Person
  - 0..*
strategy: from BusRoute through BusStop to Person

```java
BusRoute {
    traversal waitingPersons(PersonVisitor) {
        through BusStop to Person; } // from is implicit
    void printWaitingPersons() // traversal/visitor weaving instr.
        = waitingPersons(PrintPersonVisitor);
PrintPersonVisitor {
    before Person () < do printing >}
```
Robustness of Strategy

from BusRoute bypassing Bus to Person

Diagram:
- BusRoute
  - buses
  - villages
- VillageList
- BusList
  - 0..*
- Village
  - 0..*
- buses
- persons
  - waiting
- PersonList
  - 0..*
- BusStop
  - busStops
  - 0..*
- Person
  - 0..*
Filter out noise in class diagram

- only three out of seven classes are mentioned in traversal strategy!

from BusRoute through BusStop to Person

replaces traversal methods for the classes
BusRoute VillageList Village BusStopList BusStop PersonList Person
DJ

• DJ is a Java package to interpret traversal strategies at runtime
• Traversal strategy is specified in special language
• Generates an object graph using Java reflection
• Allows to traverse classes when their source code is not available
• Allows the traversal strategy to be specified at runtime
DJ

- Allows the retrieval one or more objects from the object graph according to a traversal strategy
- Allows the traversal an object graph according to the traversal strategy and to specify a visitor to be executed before and after each node that is traversed
- Allows traversals to be treated as Lists. You can iterate through that list through the iterator’s previous and next methods → Can be used with the Java Collections Framework
Basic Traversal Example [1]

```java
import edu.neu.ccs.demeter.dj.*;

class Main {
    public static void main(String[] args) {
        // constructed by reflection from the classes in the default package
        ClassGraph cg = new ClassGraph();
        // construct some object
        A a = new A(new B(new D()), new C());

        Strategy sg = new Strategy("from A to D");
        TraversalGraph tg = new TraversalGraph(sg, cg);
    }
}
```
Basic Traversal Example [2]

tg.traverse(a, new Visitor() {
    public void start() { System.out.println("begin"); }
    public void finish() { System.out.println("end"); }

    public void before(A o) { System.out.println("before A"); }
    public void after(A o) { System.out.println("after A"); }

    public void before(D o) { System.out.println("before D"); }
    public void after(D o) { System.out.println("after D"); }
});
System.out.println("Traversal Graph for from A to D");
System.out.println(tg);
Adaptive Visitors

• DJ uses adaptive visitors
  – Before and after methods are executed before and after the traversal of a matching object
  – No traversal behavior has to be specified in the visitors
  – No need for “accept” methods in the classes to be traversed
  – Not necessary to specify methods for every class to be traversed
Traversals as Lists

- The `asList` method of the class `graph` allows a traversal to be treated as a list
- Allows previous, next
- Changes made to the List "write through" to object structure
- Can be used anywhere a List can be used (e.g. with sort, reverse or shuffle of the Java Collection Framework)
- Traversal implemented as a background thread
Fetch and Gather

- The ClassGraph methods fetch and gather allow the retrieval of a single target (or a set of targets) specified in a traversal strategy
Contributions of DJ

• DJ is a tool that makes Adaptive Programming available under Java
• Does not need generators and integrates into IDEs
• No steep learning curve, easy to use
• Makes Adaptive Programming available for the “common programmer”
• Contribution to AP is similar to the contribution of AspectJ to Aspect Oriented Programming
Conclusion

• General doubts about the usefulness of the adaptive programming approach aside

• DJ allows the use of a special aspect language to describe traversals
  – This contributes to better readability and maintainability of the source, because the traversal oriented concerns are concentrated in one place

• It is relatively easy to use and integrates seamlessly into standard development environments (because it is only a library)

• The major disadvantage is the performance
  – Due to the heavy use of reflection, it is 25-30 times slower than custom generated static code
References

• Doug Orleans, Karl Lieberherr. DJ: Dynamic Adaptive Programming in Java. REFLECTION 2001

• http://www.ccs.neu.edu/research/demeter/DJ/