Aspect Oriented Programming

Programming Languages Seminar

Presenter: Barış Aktemur
University of Illinois
18 Feb. 2004

Mostly taken from Bedir Tekinerdogan’s slides
Outline

- Introduction
- Problems
- Terminology
- Aspect-Oriented Programming Languages/Frameworks
  - Compositional Filters
  - AspectJ
  - Hyper/J
  - DemeterJ
- Conclusions
Introduction

- Evolution of Programming Languages
  - Assembly/Machine Languages
  - Formula Translation
  - Procedural Programming
  - Structured Programming
  - Functional Programming
  - Logic Programming
  - Programming with abstract data types

- Evolution of Software Design
  - Monolithic ---> Modular
Design Principles → Modularity

- **Abstraction**
  - Focus only on relevant properties

- **Decomposition**
  - Divide software into separately named and addressable modules

- **Encapsulation**
  - Group related things together.

- **Information Hiding**
  - Hide implementation details from the outside

- **Separation of Concerns**
  - Ensure that each module only deals with one concern
    - **Low Coupling**
      - aim for low coupling among the modules
    - **High Cohesion**
      - aim for high cohesion within one module
Separation of Concerns

Cohesion
- **Maximize** cohesion within a component
  - i.e. Cohesive component performs only **one concern/task**
  - required changes can be easily localized and will not propagate

Coupling
- Highly coupled components have many dependencies/interactions
- **Minimize** coupling between components
  - reduces complexity of interactions
  - reduces ‘ripple’ effect
Advantages of separation of concerns

- Understandability
- Maintainability
- Extensibility
- Reusability
- Adaptability

*Separation of Concerns directly supports quality factors.*

*Lack of Separation of Concerns negatively impacts quality factors.*
Example - Figure Editor

- A figure consists of several figure elements. A figure element is either a point or a line. Figures are drawn on Display. A point includes X and Y coordinates. A line is defined as two points.
Example - Figure Editor - Design

Components are
- Cohesive
- Loosely Coupled
- Have well-defined interfaces
  (abstraction, encapsulation)

Nice Modular Design!
Crosscutting Concern - Example

Notify ScreenManager if a figure element moves

Diagram:
- Display
  - Figure
    - FigureElement
      - Point
        - getX()
        - getY()
        - setX(int)
        - setY(int)
      - Line
        - getP1
        - setP1
        - setP2(Point)

DisplayTracking
Example: Display Tracking

```java
class DisplayTracker {
    static void updatePoint(Point p) {
        this.display(p);
        ....
    }
    static void updateLine(Line l) {
        this.display(l);
        ....
    }
}

class Point {
    void setX(int x) {
        DisplayTracker.updatePoint(this);
        this.x = x;
    }
}

class Line {
    void setP1(Point p1 { 
        DisplayTracker.updateLine(this);
        this.p1 = p1;
    }
}
```
Example - Tracing - Design

Trace the execution of all operations...

Display

Figure

FigureElement

Point
- getX()
- getY()
- setX(int)
- setY(int)

Line
- getP1
- setP1
- setP1(Point)
- setP2(Point)

Tracer
- traceEntry
- traceExit

Tracing
Example - Tracing

class Point {
    void setX(int x) {
        Tracer.traceEntry("Entry Point.set");
        _x = x;
        Tracer.traceExit("Exit Point.set");
    }
}

class Line {
    void setP1(Point p1 {
        Tracer.traceEntry("Entry Line.set");
        _p1 = p1;
        Tracer.traceExit("Exit Line.set");
    }

    void setP2(Point p2 {
        Tracer.traceEntry("Entry Line.set");
        _p2 = p2;
        Tracer.traceExit("Exit Line.set");
    }
}

class Tracer {
    static void traceEntry(String str) {
        System.out.println(str);
    }
    static void traceExit(String str) {
        System.out.println(str);
    }
}
Example – Tracing and Display Tracking

Display

Figure

* FigureElement

Point

g getX()
g getY()

setX(int)

setY(int)

Line

g getP1

setP1

setP1(Point)

setP2(Point)

Tracer

trace

Tracing

Display Tracker
Crosscutting, Scattering and Tangling

- **Crosscutting**
  - concern that *inherently* relates to multiple components.
  - results in scattered concern and tangled code
  - non-functional requirements likely to crosscut

- **Scattering**
  - Single concern affects multiple modules

- **Tangling**
  - multiple concerns are interleaved in a single module
Example of crosscutting concerns

- Synchronization
- Real-time constraints
- Error-checking
- Object interaction constraints
- Memory management
- Persistency
- Security
- Caching
- Logging
- Monitoring
- Testing
- Domain specific optimization
- ...

...
Aspect-Oriented Software Development

- Provides better separation of concerns by explicitly considering crosscutting concerns (as well)
- Does this by providing explicit abstractions for representing crosscutting concerns, i.e. aspects and composing these into programs, i.e. aspect weaving or aspect composing.
- As such AOSD improves modularity
- and supports quality factors such as
  - maintainability
  - adaptability
  - reusability
  - understandability
  - ...
Basic AOP technologies

- Composition Filters
  - University of Twente, The Netherlands

- AspectJ
  - XEROX PARC, US

- DemeterJ/DJ
  - Northeastern University, US

- Multi-dimensional separation of Concerns/HyperJ
  - IBM TJ Watson Research Center, US
History of AOP languages

Scripts (Francez 81)

Reflection (Smith 81)

AI (semantic networks 79)

OO languages

MOP (1985)

CLOS-MOP

Law of Demeter (1988)

Sina interface predicates (1988)

Composition Filters (1992)

Adaptive programming (1992)

Crosscutting aspects (1996)

Composition Filters with superimposition (2001)

AspectJ (1997)

AspectJ (2000)

http://trese.cs.utwente.nl
AspectJ

- A general purpose AO programming language
  - just as Java is a general-purpose OO language
  - unlike examples in ECOOP’97 paper
    - domain specific languages for each aspect
- an integrated extension to Java
  - accepts all java programs as input
  - outputs .class files compatible with any JVM
  - integrated with tools
Example – Without AOP

```java
class Line {
    private Point _p1, _p2;
    Point getP1() { return _p1; }
    Point getP2() { return _p2; }

    void setP1(Point p1) {
        Tracer.traceEntry("entry setP1");
        _p1 = p1;
        Tracer.traceExit("exit setP1");
    }

    void setP2(Point p2) {
        Tracer.traceEntry("entry setP2");
        _p2 = p2;
        Tracer.traceExit("exit setP2");
    }
}

class Point {
    private int _x = 0, _y = 0;
    int getX() { return _x; }
    int getY() { return _y; }

    void setX(int x) {
        Tracer.traceEntry("entry setX");
        _x = x;
        Tracer.traceExit("exit setX");
    }

    void setY(int y) {
        Tracer.traceEntry("exit setY");
        _y = y;
        Tracer.traceExit("exit setY");
    }
}
class Tracer {
    static void traceEntry(String str)
    {
        System.out.println(str);
    }
    static void traceExit(String str)
    {
        System.out.println(str);
    }
}
```

Tangling Code
Scattered Concern
Example – With AOP

```java
class Line {
    private Point _p1, _p2;
    Point getP1() { return _p1; }
    Point getP2() { return _p2; }
    void setP1(Point p1) {
        _p1 = p1;
    }
    void setP2(Point p2) {
        _p2 = p2;
    }
}

class Point {
    private int _x = 0, _y = 0;
    int getX() { return _x; }
    int getY() { return _y; }
    void setX(int x) {
        _x = x;
    }
    void setY(int y) {
        _y = y;
    }
}
```

```java
aspect Tracing {
    pointcut traced():
        call(* Line.* ||
            call(* Point.*);
    before(): traced() {
        println("Entering:" +
            thisjopinpoint);
    }
    void println(String str)
    {<write to appropriate stream>}
}
```

Aspect is defined in a separate module
Crosscutting is localized
No scattering; No tangling
Improved modularity
Aspect Language Elements

- join point (JP) model
  - certain principled points in program execution such as method calls, field accesses, and object construction

- means of identifying JPs
  - picking out join points of interest (predicate)
  - **pointcuts**: set of join points

- means of specifying behavior at JPs
  - what happens
  - **advice** declarations
Modularizing Crosscutting

- Joinpoints: any well-defined point of execution in a program such as method calls, field accesses, and object construction
- Pointcut: predicate on joinpoints selecting a collection of joinpoints.

```
pointcut traced():
call(* Line.*) ||
call(* Point.*);
```
Joinpoints

- method call join points
  - when a method is called
- method reception join points
  - when an object receives a message
- method execution join points
  - when the body of code for an actual method executes
- field get joint point
  - when a field is accessed
- field set joint point
  - when a field is set
- exception handler execution join point
  - when an exception handler executes
- object creation join point
  - when an instance of a class is created
Some primitive pointcuts

- **call(Signature)**
  - picks out method or constructor call based on Signature

- **execution(Signature)**
  - picks out a method or constructor execution join point based on Signature

- **get(Signature)**
  - picks out a field get join point based on Signature

- **set(Signature)**
  - picks out a field set join point based on Signature

- **handles(TypePattern)**
  - picks out an exception handler of any of the Throwable types of TypePattern

- **instanceOf(ClassName)**
  - picks out join points of currently executing objects of class ClassName

- **within(ClassName)**
  - picks out join points that are in code contained in ClassName

- **withinCode(Signature)**
  - picks out join points within the member defined by methor or constructor (Signature)

- **cflow(pointcut)**
  - picks out all the join points in the control flow of the join points picked out by the pointcut
Advice

- Piece of code that attaches to a pointcut and thus injects behavior at all joinpoints selected by that pointcut.

- example:
  
  ```
  before (args): pointcut
  { Body }
  ```

  where `before` represents a before advice type (see next slide).

- Can take parameters with pointcuts
Advice Types

Advice code executes

- **before**, code is injected before the joinpoint
  
  \[before\ (args)\): \text{pointcut}
  
  \{ Body \}

- **after**, code is injected after the joinpoint
  
  \[after\ (args)\): \text{pointcut}
  
  \{ Body \}

- **around**, code is injected around (in place of) code from joinpoint
  
  \[ReturnType\ around\ (args)\): \text{pointcut}
  
  \{ Body \}
Aspect

- A modular unit of cross-cutting behavior.
- Like a class, can have methods, fields, initializers.
- can be abstract, inherit from classes and abstract aspects and implement interfaces.
- encapsulates pointcuts and advices
- can introduce new methods / fields to a class
Example - AspectJ

class Line {
    private Point _p1, _p2;

    Point getP1() { return _p1; }
    Point getP2() { return _p2; }

    void setP1(Point p1) {
        _p1 = p1;
    }
    void setP2(Point p2) {
        _p2 = p2;
    }
}

class Point {
    private int _x = 0, _y = 0;

    int getX() { return _x; }
    int getY() { return _y; }

    void setX(int x) {
        _x = x;
    }
    void setY(int y) {
        _y = y;
    }
}

aspect Tracing {

    pointcut traced():
        call(* Line.* ||
            call(* Point.*);

    before(): traced() {
        println("Entering:" +
            thisjopinpoint);
    }

    after(): traced() {
        println("Exit:" +
            thisjopinpoint);
    }

    void println(String str)
        {/*write to appropriate stream*/}
}

aspect
Code Weaving

- Before compile-time (pre-processor)
- During compile-time
- After compile-time
- At load time
- At run-time
Example - AspectJ

```java
aspect MoveTracking {
    private static boolean _flag = false;

    public static boolean testAndClear() {
        boolean result = _flag;
        _flag = false;
        return result;
    }

    pointcut moves():
        receptions(void Line.setP1(Point)) ||
        receptions(void Line.setP2(Point));

    static after(): moves() {
        _flag = true;
    }
}
```
Law Of Demeter

- Each unit should only have limited knowledge about other units: only about units “closely” related to the current unit.
  - “Each unit should only talk to its friends.”
  - “Don’t talk to strangers.”
- Goal: Reduce behavioral dependencies between classes.
- Loose coupling
Applying LoD

- A method must be able to traverse links to obtain its neighbors and must be able to call operations on them.
- But it should not traverse a second link from the neighbor to a third class.
- Methods should communicate only with preferred suppliers:
  - immediate parts on this
  - objects passed as arguments to method
  - objects which are directly created in method
  - objects in global variables
  - No other calls allowed

---→ Scattering
Solution is Adaptive Programming

- Encapsulate operation into one place thereby avoiding scattering
- Specify traversal over (graph) structure in a succinct way thereby reducing tangling.
- Navigation strategy
Use of Visitors

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

// define strategy

String strategy = "from BusRoute through BusStop to Person"

class BusRoute {
    // define class graph
    static Classgraph cg = new ClassGraph();
    int printCountWaitingPersons() { // traversal/visitor weaving
        // define visitor
        Visitor v = new Visitor()
        public void before(Person host) { r++;
        public void start() { r = 0;
        ...
    }
    cg.traverse(this, strategy, v);
    ...
}
```