Concepts in Aspect-Oriented Programming

COMP6209: Program Generation

Julian Rathke
jr2@ecs.soton.ac.uk
From macros to aspects

- Macros and aspects are code-based transformations
- Macros use explicit invocation
- Aspects use implicit invocation

```java
public void service(Request request, Response response) {
    DEBUG_ENTER("service", "New request " + request);
    try {
        DEBUG_PING("service", 1);
        request.setContextManager(this);
        request.setResponse(response);
        response.setRequest(request);
        // wrong request - parsing error
        int status=response.getStatus();
        if (status < 400) status=processRequest(request);
        if (status==0) status=authenticate(request, response);
        if (status==0) status=authorize(request, response);
        if (status==0) {
            request.getWrapper().handleRequest(request, response);
        } else {
            // something went wrong
            handleError(request, response, null, status);
        }
    } catch (Throwable t) {
        handleError(request, response, t, 0);
    }
    DEBUG_PING("service", 2);
    try {
        response.finish();
        request.recycle();
        response.recycle();
    } catch( Throwable ex ) {
        if(debug>0) DEBUG_LOG("Error closing request " + ex);
    }
    DEBUG_LOG("Done with request " + request);
    DEBUG_EXIT("service");
    DEBUG_PING("service", 3);
    return;
}
```

Each macro must be invoked explicitly.
Lecture Goals

• introduce and define basic concepts behind and in aspect-oriented programming
  – cross-cutting concerns
  – join points
  – point cut descriptors, advice, and aspects
  – aspect weaving
Separation of Concerns

Definition: a **concern** is a *specific requirement or consideration that must be addressed in order to satisfy the overall system goal.*

- functionality
- efficiency
- reliability
- user interface
- …

Atomic transaction design for databases:

- transaction management
- data management
- adaptation of transaction protocols
- concurrency control
- failure management

• software design is “separation of concerns” (Dijkstra, 1974)

• each concern represented as separate code unit
Well localized concerns

XML parsing in org.apache.tomcat

- red shows relevant lines of code
- nicely localized in one code unit (i.e., module)
Crosscutting concerns and the “Tyranny of the Dominant Decomposition”

Definition: a concern **cross-cuts** a design if its implementation spans multiple code units.
- induces high coupling between units
- induces **tangled code** (i.e., low cohesion)

• cross-cutting concerns …
  - are typical for large systems
  - cannot be untangled simply by a system re-design

• Tyranny of the Dominant Decomposition:
  “A program can be modularized in only one way at a time, and the many kinds of concerns that do not align with that modularization end up scattered across many modules and tangled with one another.”
Crosscutting concerns and the "Tyranny of the Dominant Decomposition"

Definition: a concern **cross-cuts** a design if its implementation spans multiple code units.
- induces high coupling between units
- induces **tangled code** (i.e., low cohesion)

- cross-cutting concerns …
  - are typical for large systems
  - cannot be untangled simply by a system re-design

- Tyranny of the Dominant Decomposition
  - design favors one concern over others
    > sort CD collection first by artist, then by genre
  - re-design to separate one concern tangles another
    > re-sort CD collection first by genre, then by artist
Cross-cutting concerns

logging in org.apache.tomcat

- red shows relevant lines of code
- not in just one code unit
- not even in a small number of places
Cross-cutting concerns (cont’d)

Closer look shows code is
• distributed over many modules
• distributed over many locations in each module
Cross-cutting concerns are common.

• General programming
  – logging
  – undo / history / caching
  – error checking
    > null pointer, range, …
  – exception handling
  – memory management
  – persistency

• Distribution
  – synchronization
  – transaction management
  – remote invocation
  – load balancing
  – replication
  – failure handling

• Networking
  – network transparency
  – security
  – monitoring
  – quality of service

• Architecture-specific
  – support for EJB

• UI preferences
Tangling and Scattering
Wouldn’t it be nice if …
Fundamental Idea of Aspect-Oriented Programming

- Isolate cross-cutting concerns as separate code units called *aspects*
- Apply code generation techniques to distribute the aspects over the base code
Join points describe where or when cross-cutting concerns materialize.

Definition: a join point is a point in the program execution where a cross-cutting concern might intervene.

Different join point models:

- **static**: join points described by code patterns
- **dynamic**: join points described by control flow
- **event-based**:
  - “whenever condition C arises, perform action A” (Filman, 2005)
- AOP languages often have a mixed (static/dynamic) join point model
Join points describe where or when cross-cutting concerns materialize. (cont’d)
Join points describe *where* or *when* cross-cutting concerns materialize. (cont’d)

```java
package FigureEditor;
...

class Line implements FigureElement {
    private Point p1, p2;
    Point getP1() { return p1; };
    Point getP2() { return p2; };
    void setP1(Point p1) { this.p1=p1; }
    void setP2(Point p2) { this.p2=p2; }
    void moveBy(int dx, int dy) {...};
};

class Point implements FigureElement {
    private int x=0, y=0;
    int getX() { return x; }
    int getY() { return y; }
    void setX(int x) { this.x=x; }
    void setY(int y) { this.y=y; }
    void moveBy(int dx, int dy) {...};
};
```

(Palo Alto Research Center.)
Join points describe *where* or *when* cross-cutting concerns materialize. (cont’d)

Dynamic join point model in AspectJ:

```java
package FigureEditor;
...

class Line implements FigureElement {
    private Point p1, p2;
    Point getP1() { return p1; };
    Point getP2() { return p2; };
    void setP1(Point p1) { this.p1=p1; }
    void setP2(Point p2) { this.p2=p2; }
    void moveBy(int dx, int dy) {...};
};

class Point implements FigureElement {
    private int x=0, y=0;
    int getX() { return x; }
    int getY() { return y; }
    void setX(int x) { this.x=x; }
    void setY(int y) { this.y=y; }
    void moveBy(int dx, int dy) {...};
};
```

(Palo Alto Research Center.)
Pointcut descriptors formalize and name sets of join points.

- join points are a *semantic* concept, pointcut descriptors are a *syntactic* construct
- use pointcut patterns for description
  - basic pointcut descriptors for different join point kinds
    - method *call* or *execution*
    - class / object *initialization*
    - field access (*get* / *set*)
    - exception handling (*handler*)
Pointcut descriptors formalize and name sets of join points.

- join points are a *semantic* concept, pointcut descriptors are a *syntactic* construct
- use pointcut patterns for description
  - primitive pointcut descriptors for different join points
  - descriptors for static and dynamic join points
    > class containment (**within**)
    > control flow graph reachability (**cflow**)
Pointcut descriptors formalize and name sets of join points.

- Join points are a *semantic* concept, pointcut descriptors are a *syntactic* construct.
- Use pointcut patterns for description
  - Primitive pointcut descriptors for different join points
  - Descriptors for static and dynamic join points
  - Combinators (\(\neg\), \&\& and \(||\))

Pointcut name combinator (lexical) wildcard (syntactic) wildcard

```c
pointcut move():
call(void * .set*(..)) ||
call(void * .moveBy(int,int));
```

"all calls to any set-method or to the moveBy-method"
Advice is code that belongs to cross-cutting concerns.

• i.e., code will be executed at join points
• advice is a syntactic construct
• different relations between advice and base code:
  – based on relative position of advice
    > advice runs adjacent to base code (before / after)
    > advice replaces base code (around / proceed)
  – based on success of base code
    > “modifiers” to advice
      > normal (returning) or failure (throwing)
• careful design of language interface between base code and advice (see AspectJ lectures)
Aspects package pointcuts and advice into a syntactic unit.

- cross-cutting concerns are a *semantic* concept, aspects are a *syntactic* construct

```java
aspect Undo {
    pointcut move():
        call(void *.set*(..)) ||
        call(void *.moveBy(int,int));

    after() returning: move() {
        <code here runs after each move>
    }
}
```
Aspect modularity cuts across class modularity.

- **Point**
  - `getX()`
  - `getY()`
  - `setX(int)`
  - `setY(int)`
  - `moveBy(int, int)`

- **Line**
  - `getP1()`
  - `getP2()`
  - `setP1(Point)`
  - `setP2(Point)`
  - `moveBy(int, int)`

- **Figure**
  - `makePoint(..)`
  - `makeLine(..)`

- **FigureElement**
  - `moveBy(int, int)`

- **Display**

The diagram illustrates the relationships and operations between the classes and their methods, highlighting the modularity and undo functionality.
Suggested further reading


• The AspectJ Team: *The AspectJ Language*.  
  – original AspectJ documentation, both available via http://www.eclipse.org/aspectj/doc/released/

  – from the horse’s mouth…