

Department of Computer Science

PURDUE
UNIVERSITY

CS505: Distributed Systems

Lecture 12: Separation of Concerns

Outline

- ▶ **Separation of concerns and aspect-oriented programming**
- ▶ **AspectJ intro**

Application Logic vs Distribution

- ▶ **Distribution is hard to deal with**
- ▶ **Intuition**
 - Functional behavior, i.e., application logic, is main aspect of program
 - Interaction behavior is another one
- ▶ **Hypothesis: distribution is orthogonal to application logic**
 - E.g., 2-tier / 3-tier architecture, reliable or unreliable communication, is “implementation detail”

Meta-Object Protocols

► Popular model

- Every object is assigned a meta-object -- a “dual”
 - Assignment can be made lazily
 - Meta-objects can be “chained”
- Meta-object controls everything that “goes in and out of object”
- Cf. *Behavioral/computational reflection* [Maes’87]

► Implementations

- Early ones with proxies
 - “Self problem”, “encapsulation problem”
- Deeper integration (runtime/compiler)

Aspect-Oriented Programming

- ▶ **Reflective approach to concurrency**
 - cf. [Briot *et al.*'98]
- ▶ **With well-defined interfaces**
 - More than only pre-/post-invocation actions
- ▶ **And an underlying methodology**

AOP Model

▶ By generalizing, every program involves various *aspects*, e.g.

- Synchronization/concurrency
- Persistence
- Security
- Replication
- Logging
- Debugging
- ...

▶ These “cut across” application: *crosscutting*

Crosscutting

▶ Goal

- Aspects can be dealt with by programmer in isolation each
- Can be “plugged” into application

▶ Aspects are rarely orthogonal

- Developer must identify conflicts
 - Crosscutting concerns
 - “Aspects are well-modularized crosscutting concerns”
- Aspects are *weaved*

Variations

▶ *Static* AOP

- Similar to compile-time meta-object protocol
- Aspects are defined like classes, types, etc.
- Weaving occurs at compilation

▶ *Dynamic* AOP

- New aspects can be added on the go
- Weaving occurs at runtime
- Challenge: consistency at transition, “predict” weaving points without high overhead

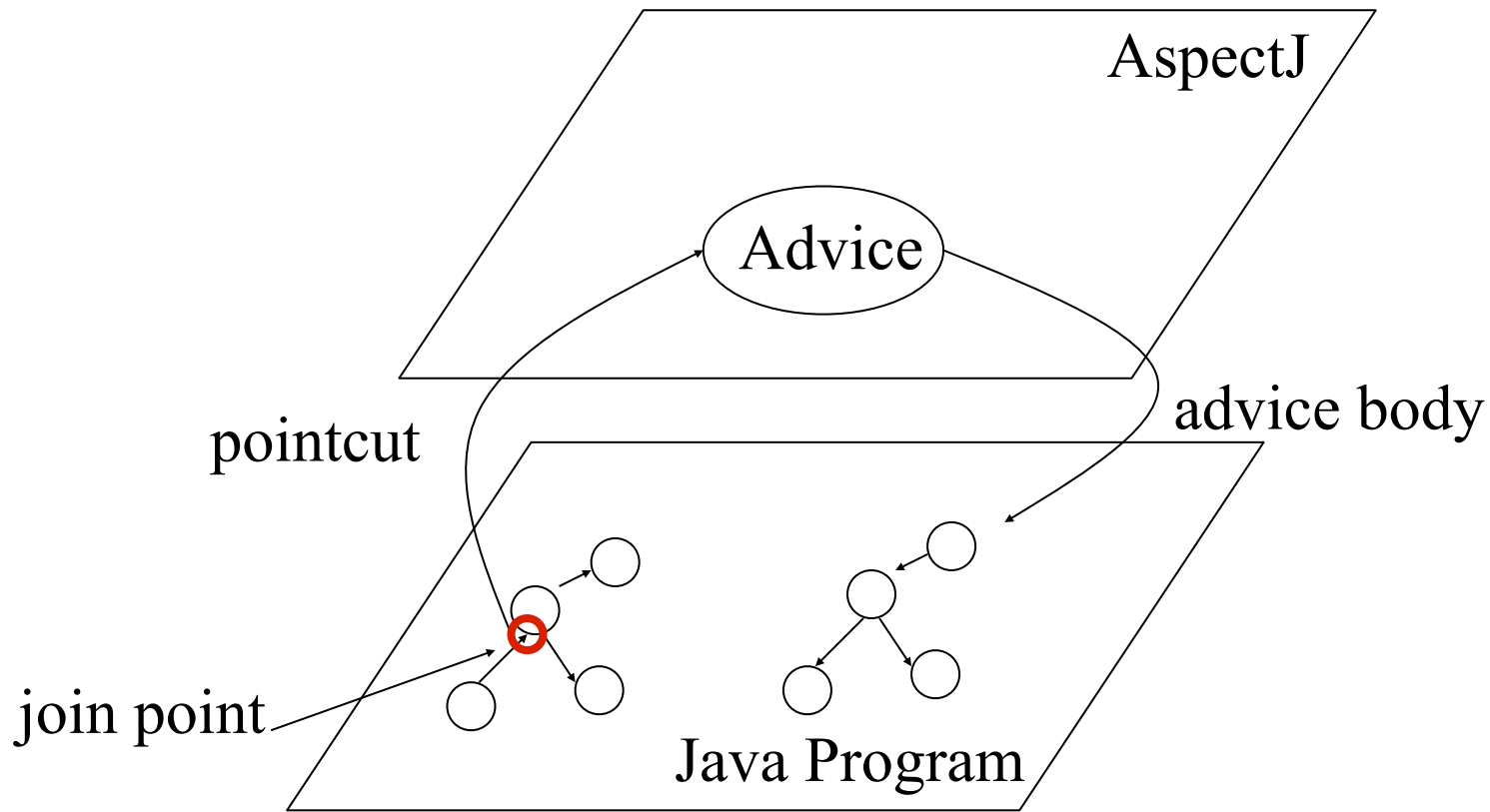
AspectJ

▶ Freely available at <http://eclipse.org/aspectj>

▶ Concepts

- *Join points*
 - Elementary points in main code for (inter)action with aspects
- *Pointcuts*
 - Describes what triggers an aspect
- *Advice*
 - Description of what to perform for pointcuts and when
- *Inter-type declarations*
 - Additions to classes (class hierarchies)
- *Aspects*
 - Define what pointcuts and advices apply to

Highlevel View of AspectJ



Join Points

▶ Transition points or boundaries

- Method call
- Class boundary
- Method execution
- Access or modification of member variable
- Exception handlers
- Static and dynamic initialization

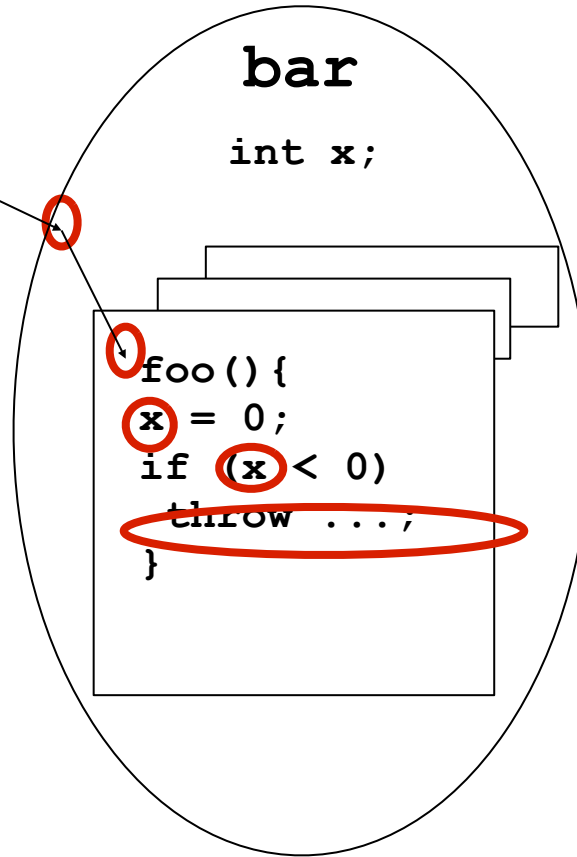
▶ Examples

- Method invocations, e.g., `void Hello.print()`
- Field accesses, e.g., `String Hello.message`

▶ Not declared individually in AspectJ

Graphical View of Join Points

`bar1.foo()`



Specifying Join Points with Designators

- ▶ `call(method signature)`
- ▶ `handler(exception name)`
- ▶ `cflow(joinpoint designator)`: including call itself
- ▶ `cflowbelow(joinpoint designator)`: not including call
- ▶ `this(type name)`: caller
- ▶ `target(type name)`: call-site (type)
- ▶ `within(class name)`: call-site (lexical)
- ▶ `execution(method signature)`
- ▶ `get(field signature), set(field signature)`
- ▶ `initialization(signature),
staticinitialization(type name)`

Designators with Wildcards

- ▶ `call(* foo())`
- ▶ `call(public bar.*(..))`
- ▶ `call(void foo(..))`
- ▶ `call(* *(..))`
- ▶ `call(*.new(int, int))`
- ▶ `handler(File*Exception)`

Pointcuts

- ▶ Set of join points

```
pointcut name(args) : pointcut_designators;
```

- ▶ Treats pointcut designators as predicates

- ▶ Support for pattern matching, e.g.,

- *

- ||

- &&

- !

- ▶ Nesting with `cflow` (<pointcut>)

- ▶ Allows parameter passing from pointcut to the advice

- `this`, `target`, and `args` are used to “publish” attributes of pointcut

Examples

```
▶ pointcut MemberRead() : call(*  
  *get*(..)) || call(* *Get*(..));
```

```
▶ pointcut MemberReadOrWrite():  
  MemberRead() || call(* *set*(..)) ||  
  call(* *Set*(..));
```

```
▶ pointcut A1(int t2, String m1): !  
  target(t2) && (call(* c*.foo*(..)) || *  
  get(m1));
```

Advice

▶ Pointcuts don't do anything

- Advice associate pointcuts with actions, and how to perform them

▶ Advice forms

- `before(param) : pointcut(param) {body}`
- `after(param) : pointcut(param) {body}`
- `after(param) returning []: pointcut(param) {body}`
- `after(param) throwing []: pointcut(param) {body}`
- `type around(param) [throws typelist] :
pointcut(param) {body}`
- `[]` stands for `<type object>`

▶ Parameter passing permitted from pointcuts to advice with

- `args ()` : arguments of call
- `this ()` : executing object (call-site)
- `target ()` : called object (callee)

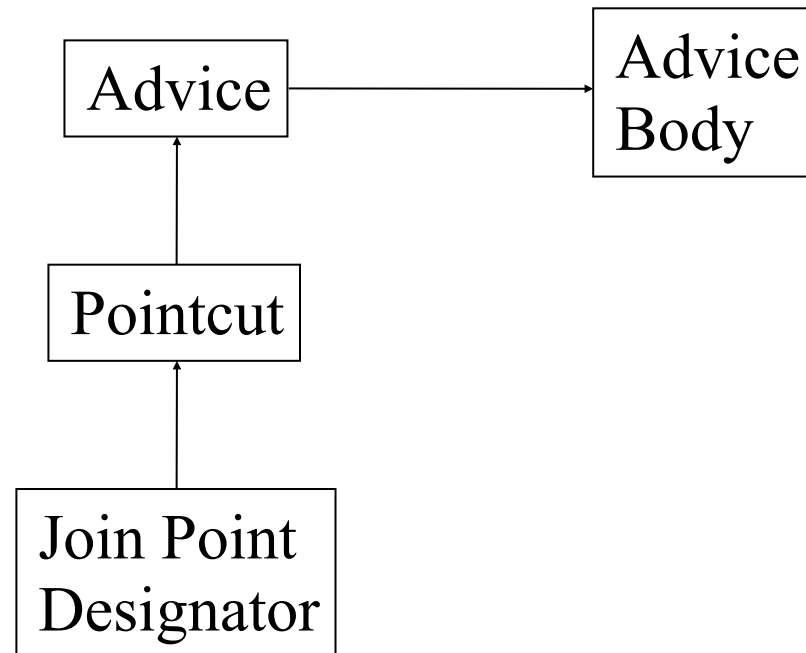
Examples

▶ `before () : MemberRead () { ... }`

▶ `after () : MemberReadOrWrite () { .. }`

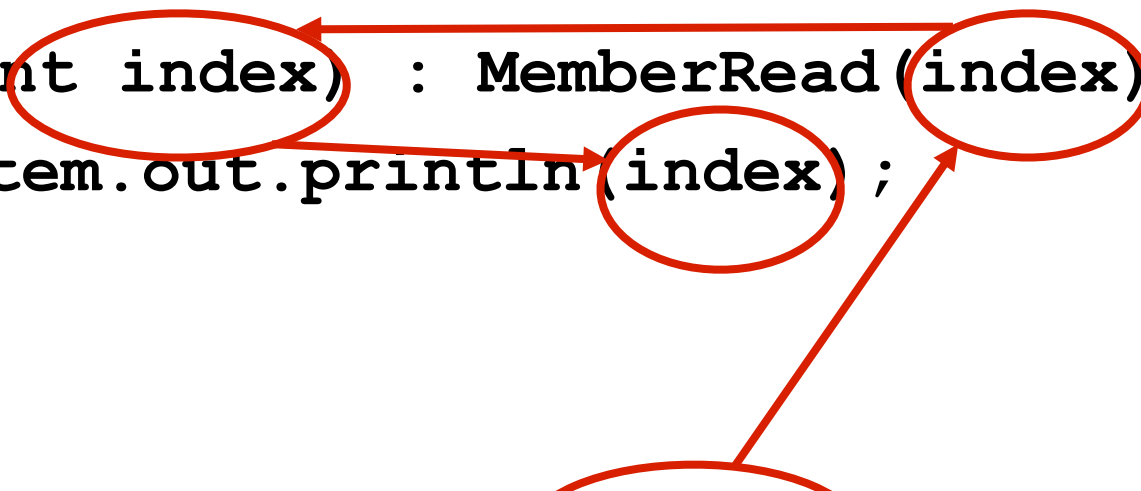
▶ `before () : MemberRead () && within (Lock) { .. }`

Parameter Information Flow



Example

```
before (int index) : MemberRead (index) {  
    System.out.println(index);  
}
```



```
pointcut MemberRead (int index) :  
    args (index) &&  
    (call (*get* (..)) || call (*Get* (..)));
```

Inter-type Declarations / Introduction

- ▶ **Static changes/additions to classes**
 - Reflecting logic handled by aspects

- ▶ **Forms**
 - Add fields to a class
 - Add methods to a class
 - Inheritance and interface specification

Examples

▶ `public int foo.bar(int x);`

▶ `private int foo.counter;`

▶ declare parents: `mammal extends animal;`

▶ declare parents: `MyThread implements MyThreadInterface;`

Compiler Errors and Warnings

- ▶ `declare error: pointcut : message;`
 - Outputs error at compilation and aborts
- ▶ `declare warning: pointcut : message;`
 - Outputs warning at compilation and aborts
- ▶ **Note**
 - Pointcut must be statically determinable, i.e., no `cflow` etc.

Aspects

- ▶ **Aspects wrap up pointcuts, advice, and inter-type declarations**
 - Represent unit of “concern”

- ▶ **Declared similarly to classes**
 - Can have local fields, methods
 - Can be declared abstract

Ordering and Association

▶ `declare precedence : aspect* ;`

- Provides ordering of application

▶ Inheritance among aspects

- Aspects can `extend` (inherit from) aspects
 - Thus possible to write subaspects e.g. for subclasses
- Derived aspect takes precedence

▶ By default one aspect instance

- Hence the need for inter-type declarations?
- Can be changed
 - `pertarget` (*pointcut*)
 - `perthis` (*pointcut*)

Reflection in AspectJ

▶ `thisJoinPoint`

- Similar to `this` in Java

▶ Access point to introspection

- `JoinPoint` interface
- Able to find various information about the join point that triggered the advice

Example

- ▶ `thisJoinPoint.toString()`
- ▶ `thisJoinPoint.getArgs()`

```
aspect TraceNonStaticMethods {  
  
    before(Point p): target(p) && call(* *(..)) {  
        System.out.println("Entering " + thisJoinPoint + " in " + p);  
    }  
}
```

Example: Readers/Writers

```
aspect RegistryReaderWriterSynchronizing
  of pertarget(readers() || writers()) {

  // internal variables
  protected int activeReaders, activeWriters, waitingReaders,
    waitingWriters;

  // procedures
  protected synchronized void beforeRead() {
    ++waitingReaders;
    while (!(waitingWriters == 0 && activeWriters == 0)) {
      try { wait(); } catch (InterruptedException ex) {}
    }
    --waitingReaders;
    ++activeReaders;
  }
  protected synchronized void afterRead() {...}
  protected synchronized void beforeWrite() {...}
  protected synchronized void afterWrite() {...}

  ...
}
```

Example (cont'd)

```
...

// pointcuts
pointcut readers():
    calls(Vector Registry.elementsNear(int, int));
pointcut writers():
    calls(void Registry.add(FigureElement)) ||
    calls(void Registry.remove(FigureElement));

// advices
before(): readers() { beforeRead(); }
after(): readers() { afterRead(); }
before(): writers() { beforeWrite(); }
after(): writers() { afterWrite(); }
}
```

Evaluation

▶ In practice?

- Often more complex scenarios than in toy examples
- Inheritance issues to be dealt with in aspects rather than in main code
- Distribution (failures) are hard to handle automatically [Kienzle&Guerraoui'02]
- ...

▶ Distributed aspects

- Several proposals, e.g. AWED, DADO, DJCutter [Nishizawa 2004]
- Usually add *remote* joinpoints
 - Logical and physical location

References

- ▶ ***Concepts and Experiments in Computational Reflection.*** P. Maes. OOPSLA 1988, 147-155.
- ▶ ***The Art of the Metaobject Protocol.*** G. Kiczales, J. des Rivières and D.G. Bobrow. MIT Press, 1991.
- ▶ ***An Overview of AspectJ.*** G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W.G. Griswold. ECOOP 2001, 327-353.
<http://www.eclipse.org/aspectj/doc/released/progguide/starting-aspectj.html>
- ▶ ***AOP: Does it Make Sense? The Case of Concurrency and Failures.*** J. Kienzle and R. Guerraoui, ECOOP 2002, 37-61.
- ▶ ***Remote Pointcut: A Language Construct for Distributed AOP.*** N. Nishizawa. AOSD 2004, 7-15.