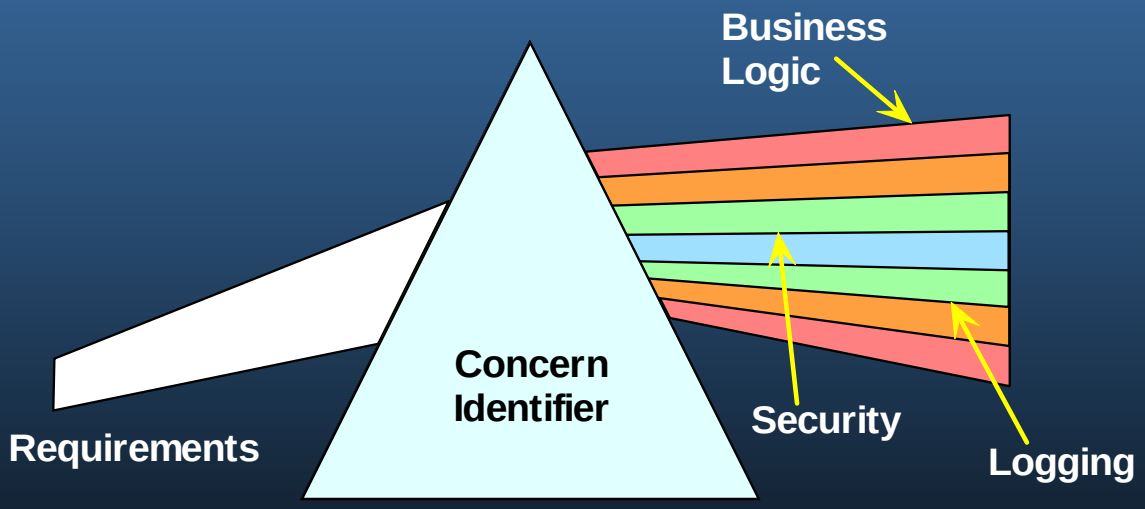
Aspect Oriented Programming

Todd A. Whittaker
Franklin University
whittakt@franklin.edu

What is AOP?

- Addresses *crosscutting* concerns
  - Requirements analysis leads to identification of *concerns* in a software system.
    - Primary concerns are issues within the problem domain. i.e. `InvoiceProcessor`, `Item`, `Shipper`, `ReorderProcessor`, etc. classes that are central to solving the problems of inventory control.
    - Crosscutting concerns are the important (even critical) issues that cross typical OO class boundaries. i.e. logging, transaction control, authorization and authentication

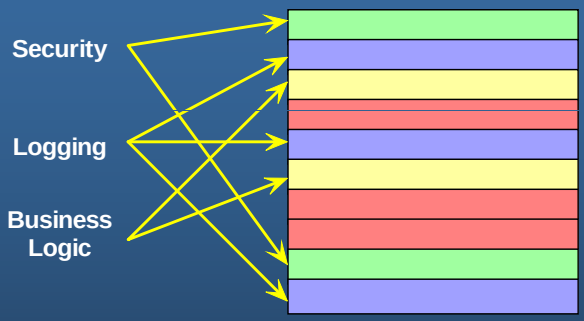
What is AOP?

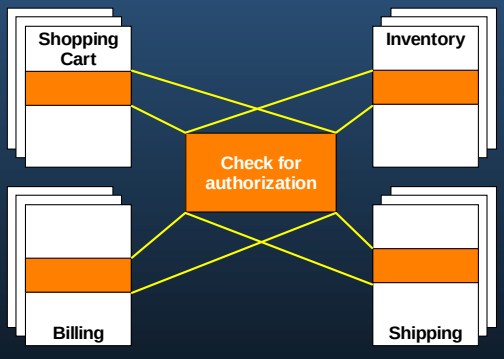


(Laddad, 2003, p. 9)

What is AOP?

- Non-AOP implementation of crosscutting concerns leads to *code tangling* and *code scattering*.
  - Tangling: concerns are interwoven with each other in a module.
  - Scattering: concerns are dispersed over many modules.
  - Typical design problem of high-coupling and low cohesion.





(Laddad, 2003, p. 16, 17)

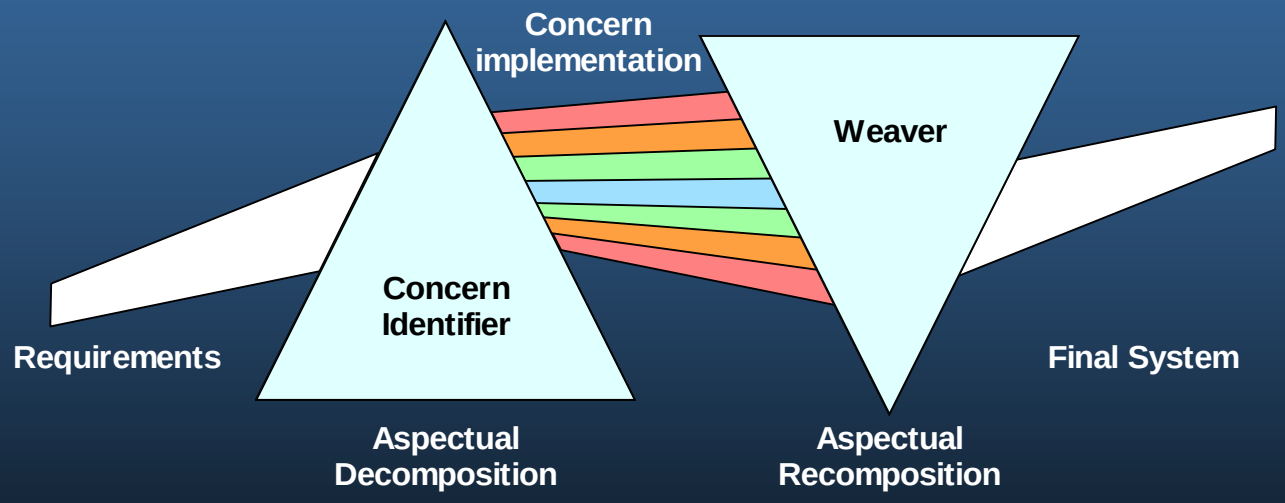
What is AOP?

- Implications of tangling and scattering on software design (Laddad, 2002)
  - Poor traceability: simultaneous coding of many concerns in a module breaks linkage between the requirement and its implementation.
  - Lower productivity: developer is paying too much attention to peripheral issues rather than the business logic.
  - Less code reuse: cut-and-paste code between modules is the lowest form of reuse.
  - Harder refactoring: changing requirements means touching many modules for a single concern.

What is AOP?

- The AOP-style solution
  - Three phase implementation (Laddad, 2003)
    - Aspectual decomposition: based on requirements, extract concerns, identifying them as primary and crosscutting.
    - Concern implementation: code each concern *separately* – primary (OO), crosscutting (AO).
    - Aspectual recomposition: tools *weave* the separately implemented code together into a final instrumented software system.

What is AOP?



(Laddad, 2003, p. 22)

What is AOP?

- Complementary to OOP as OOP was to procedural programming
  - Not a replacement for OOP, rather a superset of OO. i.e. every valid Java program is also a valid AspectJ program.
- Complementary to the XP process as well
  - YAGNI "You Aren't Gonna Need It": Always implement things when you *actually* need them, never when you just *foresee* that you need them (C2.com, 2004).

AOP Terminology

- Common terms in AOP (Gradecki, 2003)
  - join point: a well-defined location within the primary code where a concern can crosscut.
    - Method calls, method execution, constructor calls, constructor execution, field access (read or write), exception handler execution, etc.
  - Pointcut: a set of join points.
    - Can be named or anonymous
    - When a pointcut is matched, advice can be executed.

AOP Terminology

- Advice: code to be executed when a pointcut is matched.
  - Can be executed *before*, *around*, or *after* a particular join point.
  - Analogous to event-driven database triggers.
- Aspect: container holding point cuts & advice
  - Analogous to a class holding methods and attributes.
- Weaver: the tool that instruments the primary concerns with the advice based on matched pointcuts.

AOP Terminology

Object Oriented	Aspect Oriented
Class – code unit that encapsulates methods and attributes.	Aspect – code unit that encapsulates pointcuts, advice, and attributes.
Method signatures – define the entry points for the execution of method bodies.	Pointcut – define the set of entry points (triggers) in which advice is executed.
Method bodies – implementations of the primary concerns.	Advice – implementations of the cross cutting concerns.
Compiler – converts source code into object code.	Weaver – instruments code (source or object) with advice.

Weaving Types

- When can advice be inserted?
  - Compile-time: source code is instrumented before compilation. (AspectC++)
  - Link-time: object code (byte code) is instrumented after compilation (AspectJ)
  - Load-time: specialized class loaders instrument code (AspectWerkz)
  - Run-time: virtual machine instruments or application framework intercepts loaded code (JBossAOP, XWork).

Defining Pointcuts

- Calling methods & constructors (Laddad, 2002)
  - Advice is inserted after argument evaluation, but before calling. Access to caller's context.

Pointcut	Description
`call (public void MyClass.myMethod(String))`	Call to `myMethod()` in `MyClass` taking a `String` argument, returning `void`, and `public` access
`call (* MyClass.myMethod*(..))`	Call to any method with name starting in "`myMethod`" in `MyClass`
`call (MyClass.new(..))`	Call to any `MyClass`' constructor with any arguments
`call (MyClass+.new(..))`	Call to any `MyClass` or its subclass's constructor. (Subclass indicated by use of '+' wildcard)
`call (public * com.mycompany..*.*(..))`	All `public` methods in all classes in any package with `com.mycompany` the root package

Defining Pointcuts

- Execution of methods & constructors
  - Advice is inserted in the method or constructor body itself. Access to callee's context. Replace **`call`** with **`execution`**.
- Field access – read or write

Pointcut	Description
`get (PrintStream System.out)`	Execution of read-access to field out of type `PrintStream` in `System` class
`set (int MyClass.x)`	Execution of write-access to field `x` of type `int` in `MyClass`

(Laddad, 2002)

Defining Pointcuts

- Lexically based pointcuts
  - Keyword `within` captures all join points in a class, and `withincode` captures all join points in a particular method.
- Control flow based pointcuts

Pointcut	Description
`cflow (call (* MyClass.myMethod(..))`	All the join points in control flow of call to any `myMethod()` in `MyClass` including call to the specified method itself
`cflowbelow (call (* MyClass.myMethod(..))`	All the join points in control flow of call to any `myMethod()` in `MyClass` excluding call to the specified method itself

(Laddad, 2002)

Defining Pointcuts

- Context capturing pointcuts
  - Can attach names to the types to capture the variables for use inside the associated advice.

Pointcut	Description
`this (JComponent+)`	All the join points where `this` is `instanceof JComponent`
`target (MyClass)`	All the join points where the object on which the method is called is of type MyClass
`args (String,..,int)`	All the join points where the first argument is of `String` type and the last argument is of `int` type
`args (RemoteException)`	All the join points where the type of argument or exception handler type is `RemoteException`

(Laddad, 2002)

Defining Pointcuts

- Pointcuts and logical operators
  - Can be combined with `&&`, `||`, and `!`
    - Ex: Capture `public` operations on `CreditCardProcessor` which take arguments of `CreditCard` and `Money`. (Laddad, 2002)

```
pointcut cardProc(CreditCard card, Money amount):
   execution (public * CreditCardProcessor.*(..))
   && args (card, amount);
```

Advices

- Advice executes when a pointcut matches.
  - Three instrumentation points
    - `before`: executes just prior to the join point.
    - `after`: executes just after the join point.
    - `around`: executes before and/or after, with the operation taking place via a call to `proceed()`.

Example

Simple bank account class

```
package org.thewhittakers.banking;
public class Account implements Loggable {
    private double balance;
    private String owner;

    public Account(String owner, double initialBalance) {
        this.setOwner(owner);
        this.credit(initialBalance);
    }

    public void credit(double amount) {
        this.balance += amount;
    }

    public void debit(double amount) {
        this.balance -= amount;
    }

    public void transferTo(Account other, double amount) {
        this.debit(amount);
        other.credit(amount);
    } // less interesting items removed.
}
```

Example

Adding pre-condition checking

```
package org.thewhittakers.banking;

public aspect AccountConstraintsAspect {
    pointcut preventNegativeAmounts(Account account, double amount)
        : (execution(* Account.credit(double))
          || execution(* Account.debit(double)))
          && this(account) && args(amount);

    pointcut preventOverdraft(Account account, double amount)
        : execution(* Account.debit(double))
          && this(account) && args(amount);


    before(Account account, double amount):
        preventNegativeAmounts(account, amount) {
        if (amount < 0)
            throw new RuntimeException("Negative amounts not permitted");
    }

    before(Account account, double amount): preventOverdraft(account, amount) {
        if (account.getBalance() < amount)
            throw new RuntimeException("Insufficient funds");
    }
}
```

Example

Adding indented logging (Laddad, 2003, p. 171)

```
package org.thewhittakers.banking;

public abstract aspect IndentedLoggingAspect {
    protected int indentationLevel = 0;
    protected abstract pointcut loggedOperations();

    before() : loggedOperations() {
        ++this.indentationLevel;
    }

    after() : loggedOperations() {
        --this.indentationLevel;
    }

    before() : call(* java.io.PrintStream.println(..))
        && within(IndentedLoggingAspect+) {
        for (int i=0, spaces = this.indentationLevel<<2; i<spaces; ++i)
        {
            System.out.print(" ");
        }
    }
}
```

Example

Opt-in logging (AspectJ.org, 2004)

```
package org.thewhittakers.banking;

import org.aspectj.lang.*;

public aspect OptInIndentedLoggingAspect extends IndentedLoggingAspect
{
    protected pointcut loggedOperations()
        : this(Loggable)
          && (execution(* Loggable+.*(..))
              || execution(Loggable+.new(..)));

    before() : loggedOperations() {
        Signature sig = thisJoinPointStaticPart.getSignature();
        System.out.println("Entering [“
            + sig.getDeclaringType().getName()
            + "." + sig.getName() + "]");
    }
}
```

Conclusion

- AOP is an evolutionary step
  - Does not supplant OOP, but enhances it.
    - Decomposes the system into primary and crosscutting concerns which map more directly into requirements.
    - Increases comprehension of the system by reducing tangling and scattering.
    - Joinpoints, pointcuts, and advice are used to instrument primary concerns with crosscutting concerns.

Questions

- Questions? Comments? Experiences?

References

AspectJ.org. (2004). *AspectJ Sample Code*. Retrieved May 11, 2004, from the AspectJ documentation: http://dev.eclipse.org/viewcvs/indextech.cgi/~checkout~/aspectj-home/sample-code.html

C2.com. (2004). *You Arent Gonna Need It*. Retrieved May 11, 2004, from the Extreme Programming Wiki: http://xp.c2.com/YouArentGonnaNeedIt.html.

Gradecki, J., & Lesiecki, N. (2003). *Mastering AspectJ: Aspect-Oriented Programming in Java*. Indianapolis, IN: Wiley Publishing.

Laddad, R. (2002). *I want my AOP! Part 1*. Retrieved May 11, 2004, from JavaWorld: http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect_p.html

Laddad, R. (2002). *I want my AOP! Part 2*. Retrieved May 11, 2004, from JavaWorld: http://www.javaworld.com/javaworld/jw-03-2002/jw-0301-aspect2_p.html

Laddad, R. (2003). *AspectJ in Action: Practical Aspect-Oriented Programming*. Greenwich, CT: Manning Publications.