Applied Aspect-Oriented Programming

Brian Sletten
Bosatsu Consulting
brian@bosatsu.net

No Fluff Just Stuff
2007

Speaker Qualifications

- 14 years of software development experience
- Been working with Aspects for 7 years
- Have own software consulting company for design, mentoring, training and development
- Currently working in Semantic Web, AOP, Security and P2P domains

Agenda

- AOP Review
- Developmental Aspects
- Production Aspect
- Production Aspect Ideas
- Spring + AspectJ
- Static Introduction

AOP Review

Separation of Concerns

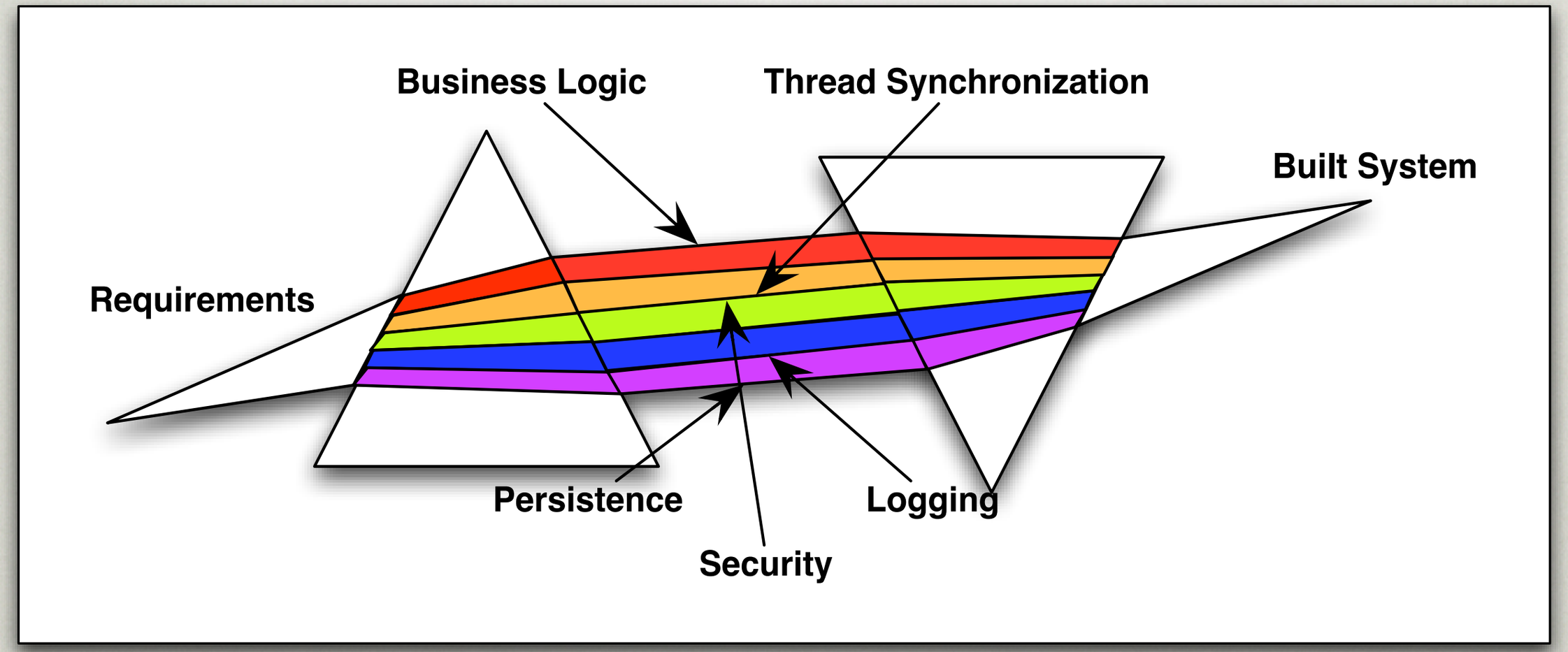
- Intellectual forebear to AOP
- Reduction of code coupling and tangling
- Flexibility and Reuse in Design
  - “Pay as You Go”

What is a Concern?

"...A specific **requirement** or **consideration** that must be addressed in order to satisfy the overall system goal..."

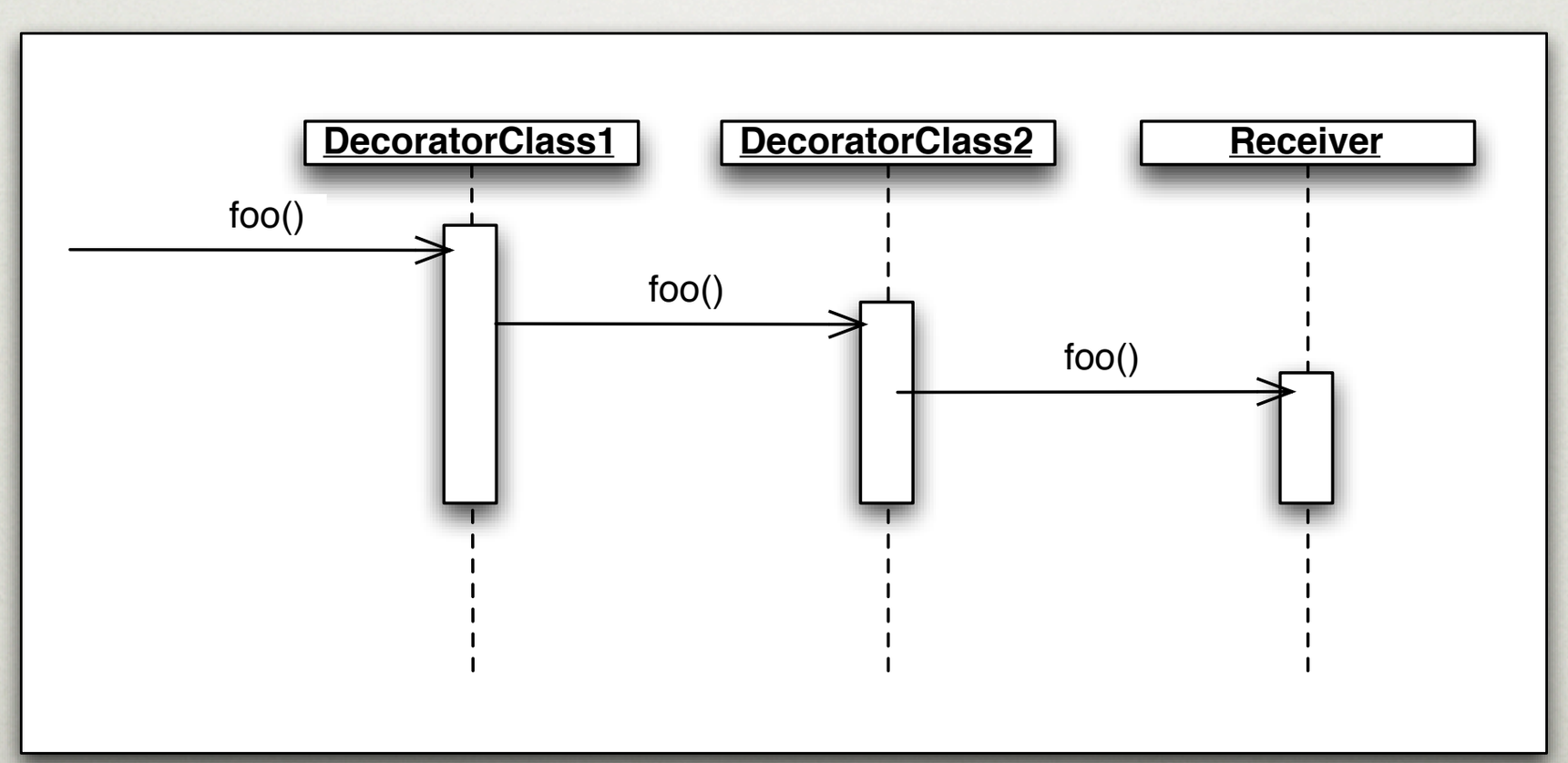
"AspectJ in Action"

Prism Metaphor for SOC



Interception

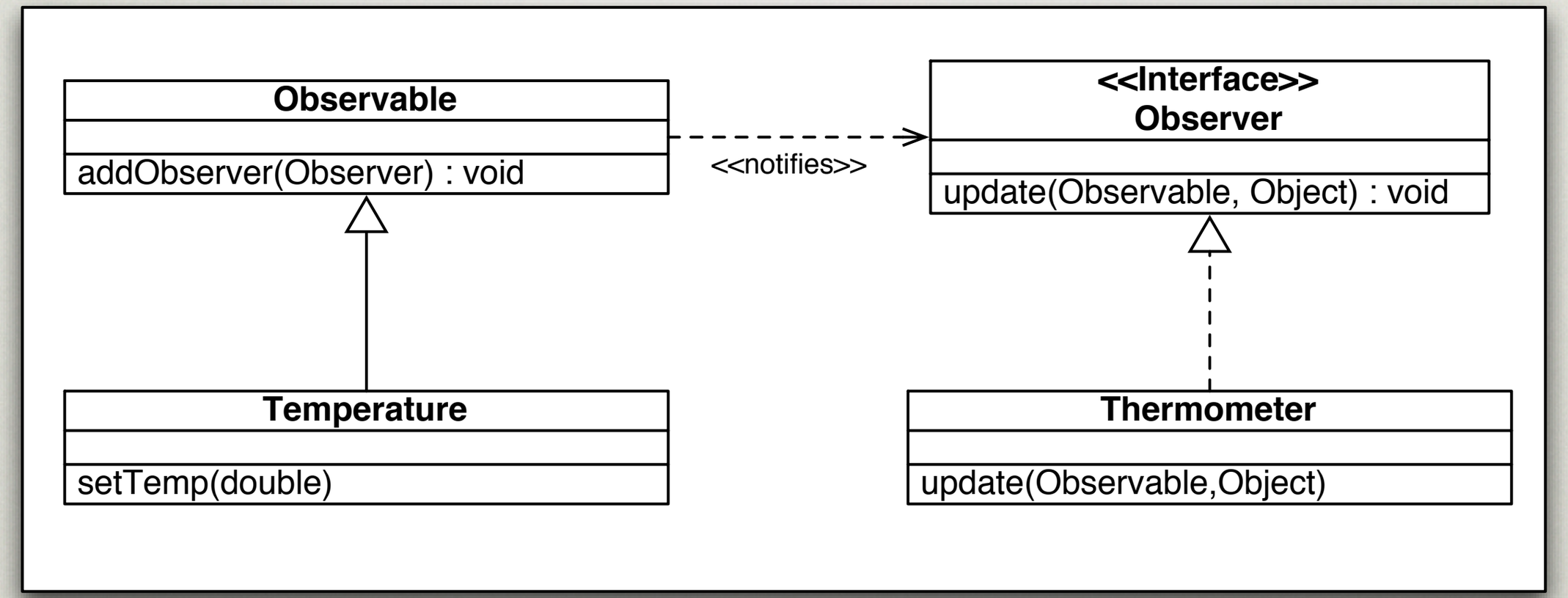
- "Catching" the call to the Receiver foo()
- Decorator Pattern is shown, but would be similar for Dynamic Proxies and Servlet Filters



"Problems" with Design Patterns

- Invasive, complex
- Requires foresight and planning to use
- Decorator works on instances, not classes

Observer Example



Complaint

- Why does Temperature have to extend Observable?
  - Inheritance is too strong of a relationship for this behavior!
  - Mixing Domain modeling with Application logic

"Problems" with Dynamic Proxies

- Requires the use of interfaces
- Works on instances
- "Wrapping" instances is an issue

What's Really Going On?

- The OO principles of Encapsulation and Modularity should be applied to design elements as well
- A Goal:
  - 1:1 mapping between a design concept and implementation

How Does Our Observer Do?

- 1 Design Concept (State change notification) : N Implementation Constructs
- Every class that wants this behavior has to be modified to include it
- Known as “Scattering”

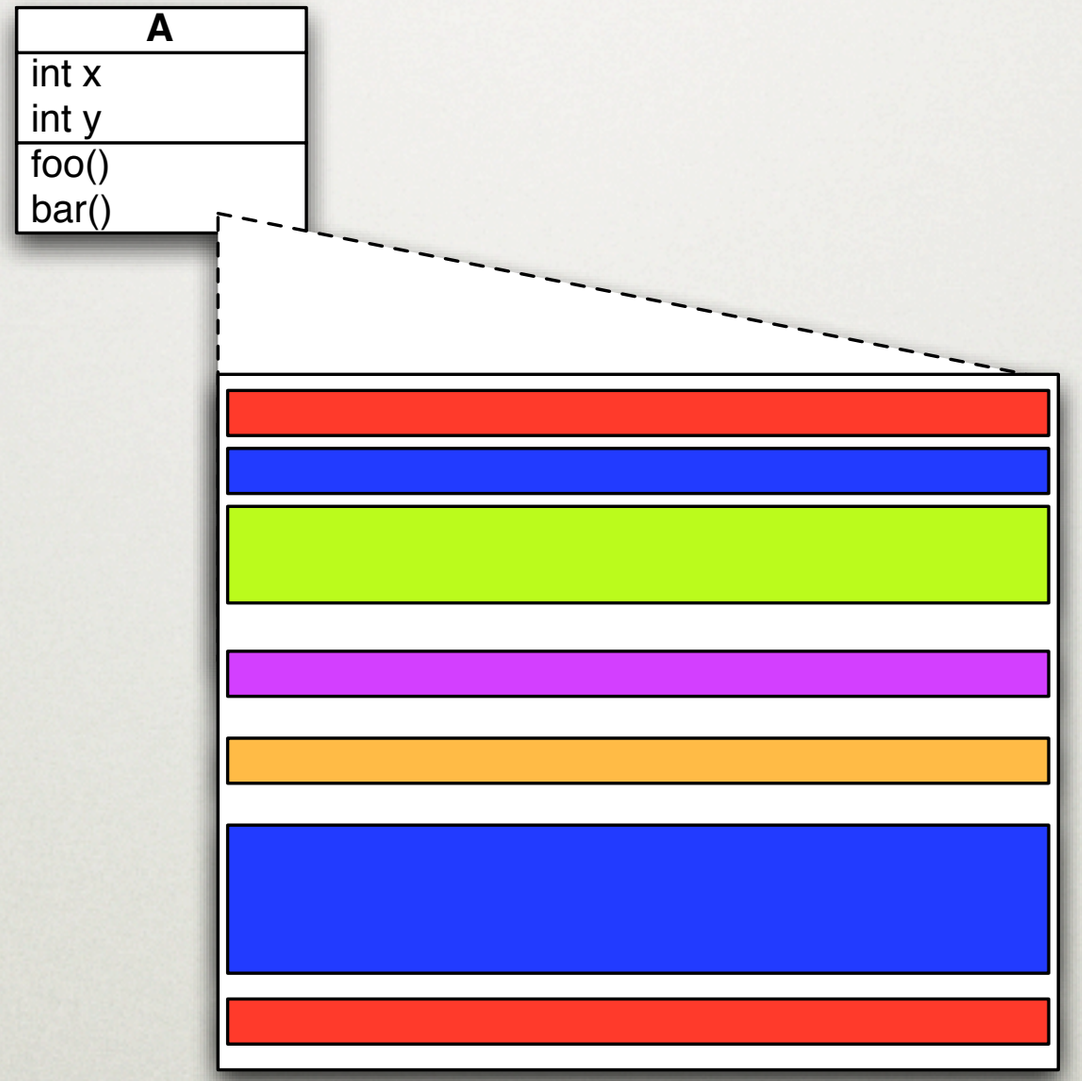
Adding Other Concerns

- In addition to State Change notification, we want
  - Thread Synchronization
  - Persistence
  - Caching

Inverting the Ratio

- N Design Concepts : 1 Implementation Construct
- Known as “Tangling”
  - Hard to change
  - Definitely not “Pay As You Go”

Tangling in Action



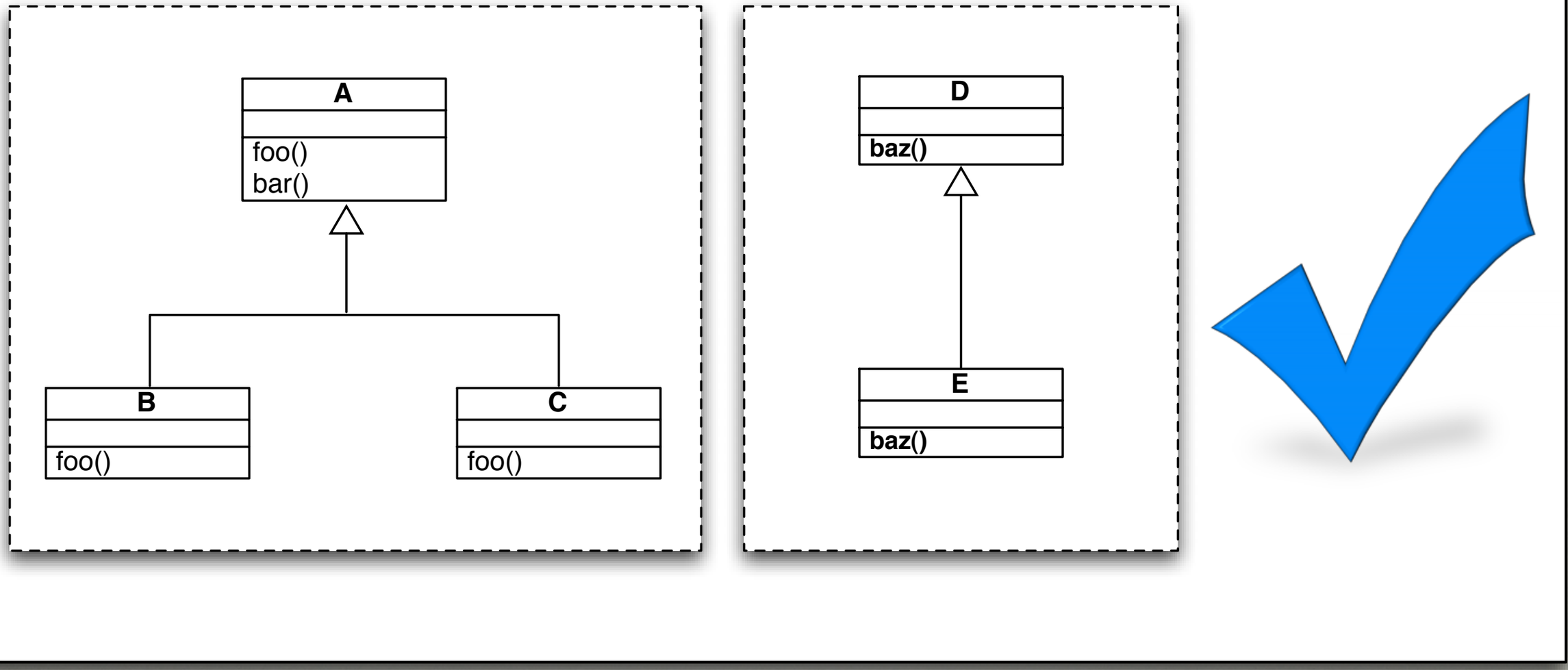
Logging and Tracing

- AOP Zealots always talk about logging and tracing because they are extreme cases of the whacked-out ratio
- 1:1000 or more!

Out With the OOLD?

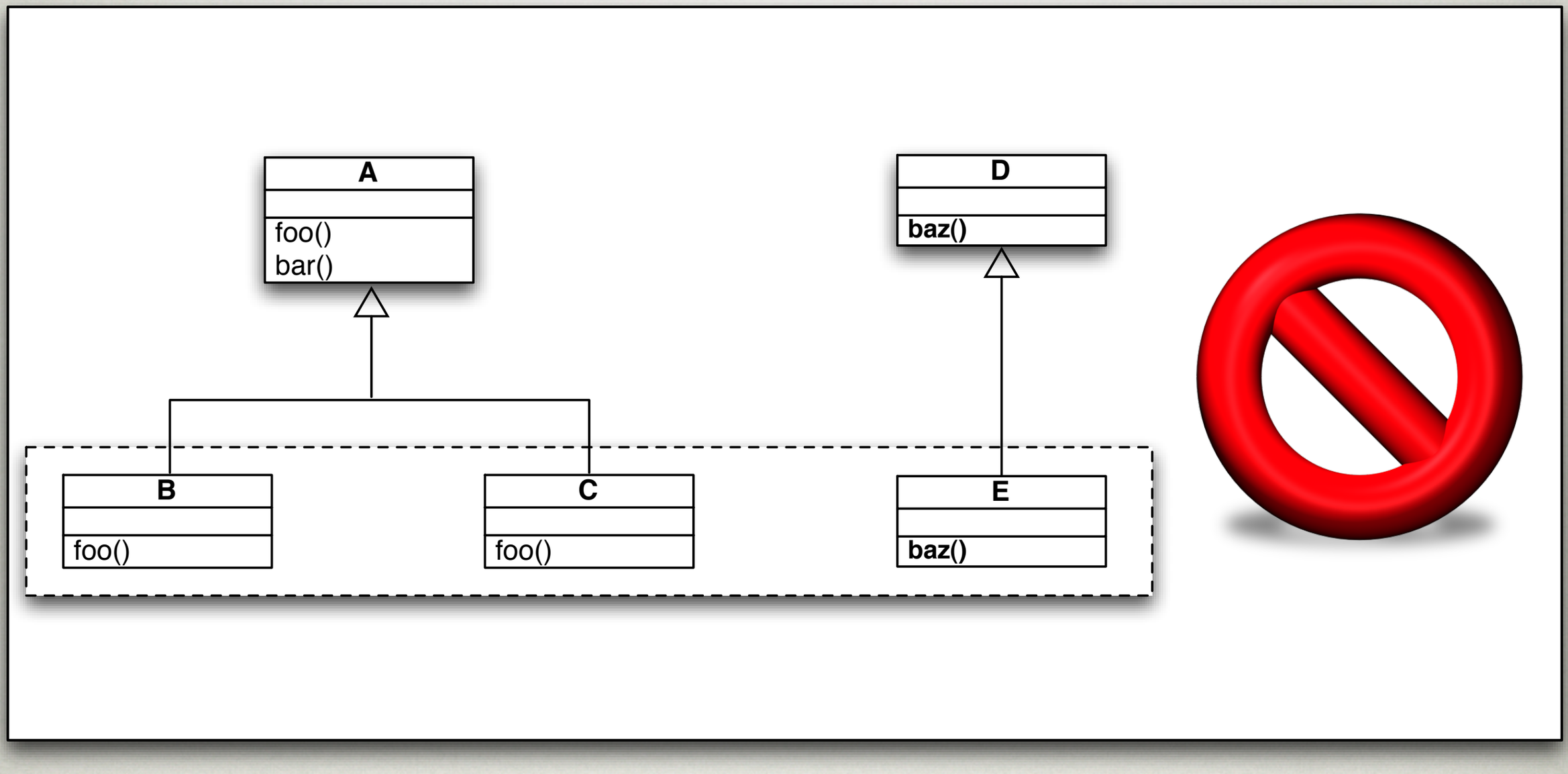
- Is AOP just a conspiracy by technical book publishers to force an overhaul of our libraries?

OO Is Good For



Object Abstractions that fit well
with Class-based decomposition

OO Is Not As Good For



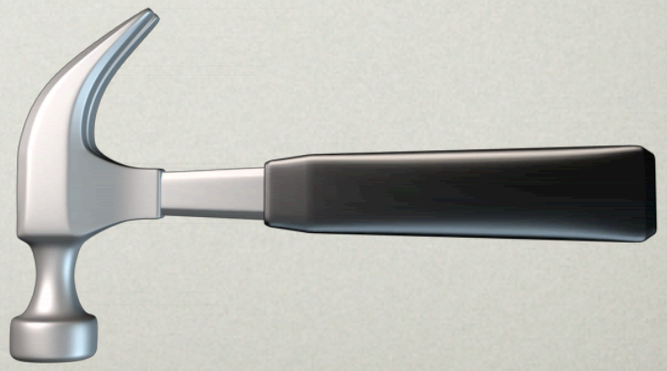
Concern Abstraction that does not follow Class-based decomposition

Cross-Cutting Concerns

- Concerns that don't fit nicely in Class-based decomposition are called "Cross-Cutting" concerns
- They apply across arbitrary portions of class hierarchies

Tyranny of Dominant Decomposition

- Phrase comes from the Hyper/J team
- Languages like Java support classes, interfaces and packages but not "features"
- "When All You Have is a Class..."



What is an Aspect?

- An Aspect is a unit of modularization for cross-cutting concern
- An attempt to maintain the 1:1 ratio for design concept to implementation construct

Benefits of AOP

- Where possible, each design concept has a simple, clear implementation
- Modules are minimally coupled
- Better chance for reuse
  - “Pay As You Go”

How Do You "Do AOP"?

- Remember the Prism Metaphor
- Separately modularized concerns are re-woven to implement a particular system
  - Compile time or Runtime
- Aspects can be Development or Production-oriented

What is AspectJ?

- A tool to support AOP in Java developed at Xerox PARC, now maintained as part of Eclipse project
- Aspects look like classes
  - Woven against other source code
- Requires separate compiler but will run on any JVM with runtime support

Warning : Jargon Alert

Join Points

- Any identifiable point in the control flow of a program
  - Method calls (**caller** side)
  - Method execution (**callee** side)
  - Accessing an instance variable
  - A constructor

Pointcuts

- Expressions that select some set of join points and their context
  - arguments
  - object being called
  - return values
  - method signature

Advice

- Pieces of code that are associated with one or more pointcuts
- Executed when a selected join point is reached
  - **before** advice runs before join point
  - **after** advice runs after join point
  - **around** advice runs around join point

How Does AspectJ Work?

- Concerns are implemented in aspects
  - *Pointcut Designators* describe how the aspect is to be woven in to a codebase
  - *Join Points* are the hooks upon which *Advice* is "hung"

Development Aspects

Design By Contract

- Bertrand Meyer argues that Unit-Tested components in isolation are not sufficient for quality software
- The interaction needs to be explicit and demonstrable
- Components must abide by a "contract"

Contract

- Pre-Conditions and Post-Conditions to verify class invariants
- Example
  - Transformation of a Shape instance does not invalidate properties of the concrete instance (i.e. Circle, Square, etc.)

Applying the Contract

- How many methods does it apply to?
- How many classes?
- Is it debug only or should it propagate into production?
  - Want to avoid unnecessary checks if it is debug only

Java "Conditional Compilation"

```
public static final boolean ENFORCE_CONTRACT = false;
.
.
public void transform() {
     if( ENFORCE_CONTRACT ) {
          checkPreCondition();
     }
    .
    .
    .
    if( ENFORCE_CONTRACT ) {
          checkPostCondition();
    }
}
```

Problems With This Approach

- Scattering
- Forgetting to add to a new method that the contract applies to
- Forgetting to do both checks
- Have to modify code to release it
  - CM Burden

Interception-Based

- Dynamic Proxies or Decorator Pattern
  - help avoid CM Burden
    - modify a property file instead of code
  - help avoid the need to remember pairs
- Decorator does not provide a “catch-all” so let’s try Dynamic Proxies

Dynamic Proxy Version

```
public Object invoke(Object proxy, Method m, Object[] args)
        throws Throwable
{
    Object result = null;

    try {
        checkPreCondition();
        result = m.invoke(obj, args);
    } catch (InvocationTargetException e) {
        throw e.getTargetException();
    } catch (Exception e) {
    } finally {
        checkPostCondition();
    }
    return result;
}
```

What's Wrong With This Approach

- What if the contract doesn't apply to all methods?
- Applies based on type, not signature
- Requires interface
- What if reference does not use a Factory method?

AspectJ to the Rescue

- Contract defined as an Aspect
- Use of before() and after() advice
- Can be compiled in or out without modifying code or property files
  - As of AspectJ 1.2 can be woven at runtime!
- Does not require interface or particular type; use any arbitrary set of methods

Example:

Contract Aspect

```java
/*
 * Created on Sep 10, 2004
 */
package nfjs.appliedaop.contract;

/**
 * @author brian
 */
public class BusinessThing {
    public void doSomething() {
        System.out.println("Doing something");
    }

    public void doSomethingElse() {
        System.out.println("Doing something else");
    }

    public void doSomethingUnrelated() {
        System.out.println("Doing something unrelated");
    }
}
```

```java
/*
 * Created on Sep 10, 2004
 *
 * Enforce a contract in arbitrary and modularized ways.
 */
package nfjs.appliedaop.contract;


/**
 * @author brian
 */

public aspect ContractAspect {
    pointcut enforceContract() : execution ( * BusinessThing.doSomething() ) ||
        execution( * BusinessThing.doSomethingElse(..) );

    before() : enforceContract() {
        System.out.println("Asserting a pre-condition");
    }

    after() : enforceContract() {
        System.out.println("Asserting a post-condition");
    }
}
```

Example: Basic AspectJ Syntax

```java
/*
 * Created on Sep 8, 2004
 */

package nfjs.appliedaop.basic;

/**
 * @author brian
 */

public class Foo {
    public void foo() {
        System.out.println("foo");
    }

    public void boo() {
        System.out.println("boo");
    }

    public void zoo( int i ) {
        System.out.println("zoo: " + i );
    }

    public int hoo( int i ) {
        System.out.println("hoo: " + i );
        return i;
    }
}
```

```aspectj
/*
 * Created on Sep 8, 2004
 */
package nfjs.appliedaop.basic;


/**
 * @author brian
 */

public aspect BasicAspect {

    pointcut callAll() : call( void Foo.*(..) );
    pointcut executeFoo() : execution( void *.foo(..) );
    pointcut callInt( int j ) : call( * Foo.*(int) ) && args( j ) ;

    before() : callAll() {
        System.out.println( "Before Call JoinPoint" );
    }

    before() : executeFoo() {
        System.out.println( "Before Execution JoinPoint" );
    }

    before( int j ) : callInt( j ) {
        System.out.println("Before Call JoinPoint with int args: " + j );
    }
}
```

Controlled Access

- As an Architect, you make decisions and set policies by which you *HOPE* the developers abide
- How do you enforce this?
  - grep and code inspections might work but who has that kind of time?

Controlled Access Examples

- No use of System.out or System.err
- No use of public variables
- No direct access to JDBC classes without first going through a Facade

Example:
Controlled Access Warning Aspect

```java
/*
 * Created on Sep 9, 2004
 *
 * A Controlled Object. We don't want developers to access it
 * directly, so please don't.
 *
 */
package nfjs.appliedaop.controlled;

/**
 * @author brian
 */

public class ControlledObject {
    public void something() {
        System.out.println( "Something" );
    }

    public void somethingElse() {
        System.out.println( "Something Else" );
    }
}
```

```java
/*
 * Created on Sep 9, 2004
 *
 * A Facade to a service that we'd like all of our developers to
 * use for calling ControlledObject methods.
 *
 */
package nfjs.appliedaop.controlled;

/**
 * @author brian
 *
 */
public class PreferredService {
    private ControlledObject co = new ControlledObject();

    public void doSomething() {
        co.something();
    }

    public void doSomethingElse() {
        co.somethingElse();
    }
}
```

```
/*
 * Created on Sep 9, 2004
 */
package nfjs.appliedaop.controlled;


/**
 * @author brian
 */
public abstract aspect ControlledAccessAspect {
    abstract pointcut callControlled() ;

    before() : callControlled()
    {
        System.out.println("Look, Buddy, we asked nicely! Don't call this directly");
    }
}
```

```
/*
 * Created on Sep 9, 2004
 *
 */
package nfjs.appliedaop.controlled;


/**
 * @author brian
 */
public abstract aspect PreferredServiceControlledAccessAspect
                        extends ControlledAccessAspect {
    pointcut callControlled() : call( * ControlledObject.*(..) )
                        && !within(PreferredService);
}
```

```
/*
 * Created on Sep 9, 2004
 *
 * Our documentation in ControlledObject wasn't enough. Let's put one some
 * more pressure by introducing a warning.
 */

package nfjs.appliedaop.controlled;

/**
 * @author brian
 */

public aspect CompileWarningControlledAccessAspect
                extends PreferredServiceControlledAccessAspect {
    declare warning : callControlled() : "We're warning you, use the PreferredService";
}
```

```
/*
 * Created on Sep 9, 2004
 */
package nfjs.appliedaop.controlled;

/**
 * @author brian
 */
public aspect NoFoolinAroundControlledAccessAspect
                    extends PreferredServiceControlledAccessAspect
{
    before() : callControlled() {
        throw new IllegalStateException( "That's it!" );
    }
}
```

Swing Thread Safety

- You are handed a non-trivial Swing application with multiple threads and a fixed price contract for new features
- While familiarizing yourself with the code, you uncover a non-threadsafe Swing component modification
- Do you cry or yawn?

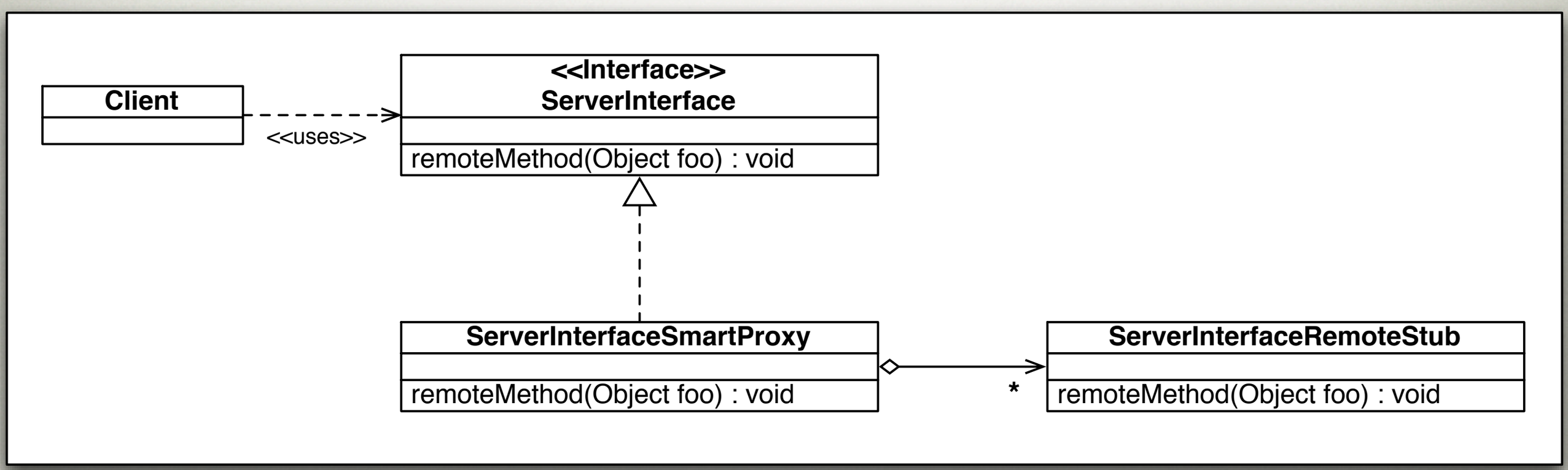
Example : Swing Thread Checker Aspect

Production Aspect

Smart Proxy

- The first time I ever felt the need for AOP was in a distributed computing system
- To support dynamic failover, we put a level of indirection in so that the “client” could recover from a temporary failure
  - On a remote exception, failover to another server instance

Smart Proxy


Client
<<uses>>
<<Interface>>
ServerInterface
remoteMethod(Object foo) : void
ServerInterfaceSmartProxy
remoteMethod(Object foo) : void
*
ServerInterfaceRemoteStub
remoteMethod(Object foo) : void

Problems With This Approach

- Worked well enough, but it was a pain
- Hand-coded per interface
- Changes in the interface required changes to the proxies
- Failover logic was scattered all over

Example: Smart Proxy Aspect

Production Aspect Ideas

Forcing Homogenous Exceptions For a Layer

- If you are building a service or a subsystem you probably want to control what kind of exceptions get thrown to your clients
- You can manually wrap every method, but that involves scattering and a rigid policy
- AspectJ can be used to catch all Exceptions thrown and convert them to something you want to expose

Example : Homogenous Exceptions Aspect

Modular Synchronization

- I once had a client with a Swing app that shared access to a JDBC Connection via multiple threads
- Discovered a deadlock in the Oracle JDBC driver
- Re-architecting wasn't an option

Decorator Synchronization

- Ended up implementing a Decorator that used Reader/Writer locks to avoid the problem
- Wrapped instances of Connection were returned
  - Don't leak out unwrapped versions!

AOP Synchronization

- Modularized
- Can support different synchronization policies in different circumstances

Spring + AspectJ

Spring AspectJ Support

- AspectJ pointcut designators for method interception
- **Note**: not field interception or static introduction
- Some special handling on **`this`**/**`target`** pointcuts due to Spring's Proxy-based implementation

- **`execution`**
- **`within`**
- **`this`**
- **`target`**
- **`args`**
- **`@syntax`**

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
    "http://www.springframework.org/dtd/spring-beans.dtd">

<beans>
    <bean id="mya"
        class="net.bosatsu.spring.A">
        <property name="greeting">
            <value>Hola</value>
        </property>
    </bean>

    <bean id="mywrappeda" class="org.springframework.aop.framework.ProxyFactoryBean">
        <property name="proxyInterfaces">
            <list><value>net.bosatsu.spring.IA</value></list>
        </property>
        <property name="interceptorNames">
            <list><value>logger</value></list>
        </property>
        <property name="target">
            <ref bean="mya"/>
        </property>
    </bean>

    <bean id="logger" class="net.bosatsu.spring.LoggingAdvice">
    </bean>
</beans>
```

```java
package net.bosatsu.spring;

import java.lang.reflect.Method;

import org.springframework.aop.MethodBeforeAdvice;

public class LoggingAdvice implements MethodBeforeAdvice
{
    public void before( Method method, Object [] args, Object target ) throws Throwable
    {
        IA a = ( IA ) target;

        System.out.println("A is about to say: " + a.getGreeting() );
    }
}
```

```xml
<?xml version="1.0" encoding="UTF-8"?>
    <beans xmlns="http://www.springframework.org/schema/beans"
           xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
           xmlns:aop="http://www.springframework.org/schema/aop"
           xsi:schemaLocation="http://www.springframework.org/schema/beans
                               http://www.springframework.org/schema/beans/spring-beans.xsd
                               http://www.springframework.org/schema/aop
                               http://www.springframework.org/schema/aop/spring-aop.xsd">
    <aop:config>
        <aop:aspect id="sayHello" ref="logger">
            <aop:pointcut id="mypc" expression="execution(* sayHello(..)) and target(bean)"/>
            <aop:before pointcut-ref="mypc" method="announce" arg-names="bean" />
        </aop:aspect>
    </aop:config>

    <bean id="mya" class="net.bosatsu.spring.A">
        <property name="greeting">
            <value>Güten Tag</value>
        </property>
    </bean>

    <bean id="logger" class="net.bosatsu.spring.LoggingPOJO" />

</beans>
```

```
package net.bosatsu.spring;

public class LoggingPOJO {
    public void announce( Object o ) {
        IA a = ( IA ) o;
        System.out.println( "I could say something about the fact "
            + "that someone is about to say: "+ a.getGreeting() );
    }
}
```

Just a POJO!

```
package net.bosatsu.spring;

public class LoggingPOJO {
    public void announce( Object o ) {
        IA a = ( IA ) o;
        System.out.println( "I could say something about the fact "
            + "that someone is about to say: "+ a.getGreeting() );
    }
}
```

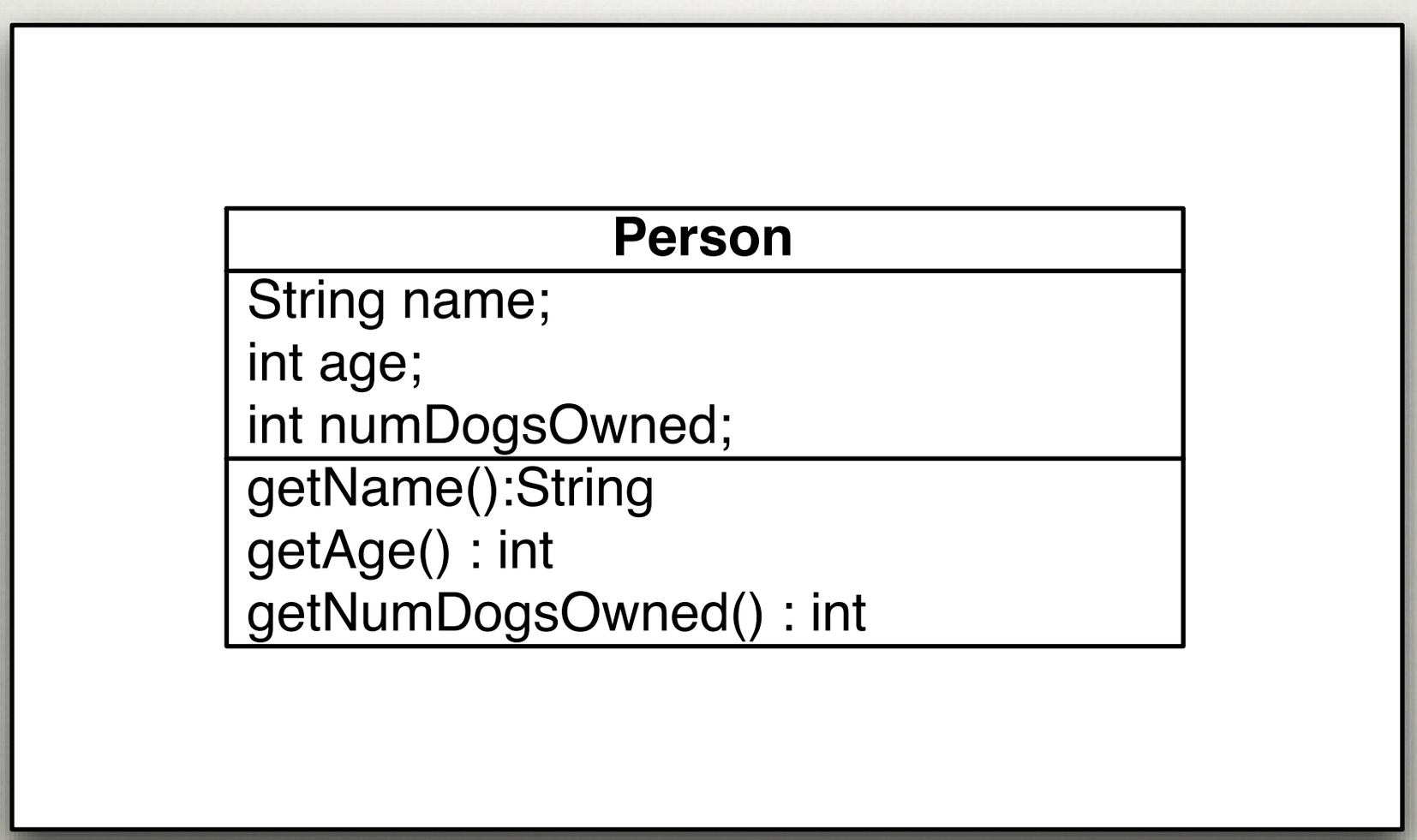
Static AOP

Static AOP

- We have mostly been talking about dynamic AOP usage
- Separation of concerns can have static benefits as well

Domain Modeling

- What's potentially wrong this?



Mixed-Role Cohesion

- Dog Ownership and Personness are separate concepts
- Person class is encumbered with Dog Ownership
  - Not necessarily bad, but can you imagine someone who doesn't own a dog?

Example : Domain Modeling Aspect

Conclusions

- AOP
  - is not a fad; is not academic navel-gazing
  - does not replace OO abstraction modeling
  - is about modularizing and separating concerns
  - is about more than logging and tracing!

References

- Colyer, Adrian, http://jroller.com/page/colyer
- Colyer, Adrian et al, “Eclipse AspectJ : Aspect-Oriented Programming w/ AspectJ and the Eclipse AspectJ Development Tools”, Addison-Wesley, 2004.
- Laddad, Ramnivas, “AspectJ in Action”, Manning, 2003.
- Miles, Russ, “AspectJ Cookbook”, O’Reilly, 2004.
- Page-Jones, Meilir, “Fundamentals of Object-Oriented Design in UML”, Addison-Wesley, 2000.

Please Write Your Reviews

Feedback/Questions: brian@bosatsu.net

Slides:
http://www.bosatsu.net/talks/AppliedAOP.pdf

Examples:
http://www.bosatsu.net/talks/examples/AppliedAOP-Examples.zip