AOP and metadata: The need
Signature-based pointcut

• Utilize join point signatures
  – Exploits inherent data associated with signature
  – Wildcards select a wide range of join points

• Work well in many cases
  – Logging and tracing
  – Profiling
  – Policy enforcement
  – Caching and pooling
  – Fault tolerance
  – Thread safety
  – Business rules
Metadata based pointcuts

• Utilize join point metadata
  – Often along with join point signature

• Needed in certain cases
  – Transaction management
  – Authentication
  – Authorization
  – Concurrency control
  – …
Conventional transaction management

```java
public class Account {

    ...

    public void credit(float amount) {
        UserTransaction ut = ...;
        try {
            ut.begin();
            ...
            business logic ...
            ut.commit()
        } catch (Exception ex) {
            ut.rollback();
            // rethrow after logging and wrapping
        }
    }
}
```
Conventional transaction management

```java
public void debit(float amount) throws InsufficientBalanceException {
    UserTransaction ut = ...;
    try {
        ut.begin();

        ... business logic ...

        ut.commit()
    } catch (Exception ex) {
        ut.rollback();
        // rethrow after logging and wrapping
    }
}
```
Conventional transaction management

```java
public float getBalance() {
    ... business logic ...
}
```
AOP transaction management

```java
public class Account {

    ...

    public void credit(float amount) {
        ... business logic ...
    }

    public void debit(float amount) throws InsufficientBalanceException {
        ... business logic ...
    }

    public float getBalance() {
        ... business logic ...
    }

}```
public aspect BankingTransactionManagement {
    public pointcut transactedOps() : ???;

    Object around() : transactedOps()
        && !cflowbelow(transactedOps()) {
            ...
            try {
                ut.begin();
                retval = proceed();
                ut.commit();
            } catch (Exception ex) {
                ut.rollback();
            }
            return retval;
        }
}
Reusable base aspect

```java
public abstract aspect TransactionManagement {
    public abstract pointcut transactedOps();

    Object around()
        : transactedOps() && !cflowbelow(transactedOps()) {
            ...
            try {
                ut.begin();
                retval = proceed();
                ut.commit();
            } catch (Exception ex) {
                ut.rollback();
            }
            return retval;
        }
}
```
public aspect BankingTransactionManagement extends TransactionManagement {

    public pointcut transactedOps() :
    ;

}
System-specific subaspects

```java
public aspect BankingTransactionManagement
    extends TransactionManagement {

    public pointcut transactedOps()
        : ???;
}
```

• **Issue**
  – Capturing operations with a transaction management need
public aspect BankingTransactionManagement 
    extends TransactionManagement {

    public pointcut transactedOps() 
        : execution(* Account.credit(..)) 
        || execution(* Account.debit(..)) 
        || ...;

}
System-specific subaspect

```java
class BankingTransactionManagement extends TransactionManagement {
    public pointcut transactedOps() {
        return execution(* Account.credit(..)) || execution(* Account.debit(..)) || ...;
    }
}
```

• Issue
  — Maintaining the method list in the pointcut definition
public aspect BankingTransactionManagement
    extends TransactionManagement {

    public pointcut transactedOps()
        : @annotation(Transactional);

}
public class Account {

    ...

    @Transactional public void credit(float amount) {
        ... business logic ...
    }

    @Transactional public void debit(float amount) throws InsufficientBalanceException {
        ... business logic ...
    }

    public float getBalance() {
        ... business logic ...
    }

}
Multidimensional interfaces using metadata
Tyranny of dominant signature

• Simple signature

    void credit(float amount);
    – Business concern is dominating the signature
    – Other concerns aren’t apparent

• Tangled signature

    void transactionalAuthorizedCredit(float amount);
    – Ugh!
    – All clients aware of all concerns
    – Changes affect all clients
    – Expressing values in crosscutting dimension even harder
      • transaction_requiredAuthorized_accountModificationCredit() 😞
Untangling using metadata

```java
@Transactional(Required)
@Authorized("accountModification")
public void credit(float amount);
```

- Each dimension expressed nicely
- Implementation need to understand only relevant dimensions
- Changed affect only the relevant concern
- Expressing values in crosscutting dimension trivial
Multidimensional signature with metadata

public void credit()
Multidimensional signature with metadata

Transaction Management

× (Credit, REQUIRED)

Business

@Transactional(REQUIRED)

public void credit()
Multidimensional signature with metadata

Transaction Management

Business

Authentication

(Credit, REQUIRED, “bankOperation”)

@Authenticated(“bankOperation”)

@Transactional(REQUIRED)

public void credit()
Annotated Account class

```java
public class Account {
    @Transactional(kind=Required)
    public void credit(float amount) {
        ...
    }

    @Transactional(kind=Required)
    public void debit(float amount) {
        ...
    }

    @Transactional(kind=None)
    public float getBalance() {
        ...
    }
}
```
public class Account {

    public void credit(float amount) {
        ...
    }

    public void debit(float amount) {
        ...
    }

    public float getBalance() {
        ...
    }

}
public class Account {
    @Transactional(kind=Required)
    public void credit(float amount) {
        ...
    }
    
    @Transactional(kind=Required)
    public void debit(float amount) {
        ...
    }
    
    @Transactional(kind=None)
    public float getBalance() {
        ...
    }
}
Annotated Account class:
Transaction management perspective

```java
public class Account {
    @Transactional(kind=Required)
    * *.*(..) {
        ...
    }

    @Transactional(kind=Required)
    * *.*(..) {
        ...
    }

    @Transactional(kind=None)
    * *.*(..) {
        ...
    }
}
```
Metadata-fortified AOP

• **Concern interface**
  – Projection of a program element on a dimension

• **Each dimension**
  – Maps to a concern
  – Business concerns
    • Implemented by classes
  – Crosscutting concerns
    • Implemented by aspects
  – Coupling between classes and aspects limited to metadata
Metadata and AOP best practices
Metadata to capture join points vs. current mechanism

• The upside
  – Easy way to capture certain crosscutting concerns
  – Limits collaboration between aspect and classes to just annotations

• The downside
  – Collaboration from classes is needed
  – Overuse may obscure AOP’s obliviousness property
    • Especially when a little extra design effort may obviate the need for annotations
Guidelines in using metadata

• Don’t use when you can do without
  – Implicit data available in join point signature often suffice
    • All RMI operations
      
      ```
      execution(* Remote+.*(..) throws RemoteException);
      
      • All thread safe Swing calls
        ```call`(void JComponent.revalidate())
        ```call`(void JComponent.repaint(..))
        ```call`(void add*Listener(EventListener))
        ```call`(void remove*Listener(EventListener));
      ```
  – Bad idea for certain cases
    • @Trace
    • @Profile
Guidelines in using metadata

• Employ aspect inheritance
  – Push down decision
  – Multiple subaspects combined with earlier guideline

• Use annotation defined for other purposes
  – EJB, EMF, Hibernate
Guidelines in using metadata

• Use abstract annotation types
  – do not dictate implementation
    • @ReadOnly, not @ReadLock
    • @Transactional, not @JTATransactional
    • @Timing, not @WaitCursor
    • @Idempotent, not @RetryOnFailure
Meta-guideline

• Wisdom comes with experience!
• Start with something
  – Refactoring is often your best friend
Metadata-fortified AOP: Effect on obliviousness

- Metadata is in eye of beholder!
  - Inherent data may be metadata
    - Is exception specification inherent data or metadata?
    - Public? Private? …
    - Type specification?
  - Metadata may be inherent data
    - @OrderProcessing?
    - @Persistent?
- If guidelines are followed, obliviousness is preserved
Design Evolution with Metadata-fortified AOP
Naïve aspect implementation

<<aspect>>
BankingTxMgmt
pointcut transactedOps() : execution(…)

Customer
void setAddress()
...

Account
void credit()
...

captures
Extracting the base aspect

<<aspect>>
TransactionManagement

pointcut transactedOps()

<<aspect>>
BankingTxMgmt

pointcut transactedOps() :
execution(…)

captures

Customer

void setAddress()

...

Account

void credit()

...

March 2005
Copyright © Ramnivas Laddad (ramnivas@aspectivity.com)
Using participant pattern: direct participation

```
<<aspect>>
TransactionManagement

pointcut transactedOps()
```

Account

void credit()

...<aspect>>
Participant

Customer

void setAddress()

...<aspect>>
Participant

participates

Participates
Metadata-driven crosscutting

```
<<aspect>>
TransactionManagement
pointcut transactedOps()

<<aspect>>
MetadataDrivenTxMgmt
pointcut transactedOps()
: @annotation(Transactional)

Customer
@Transactional
void setAddress()
...

Account
@Transactional
void credit()
...
Using participant pattern: indirect participations

```java
declare @annotation : *
    Account.credit(..) || Account.debit(..)
: @Transactional(Required);
```

Or

```java
declare @annotation : *
    @PurchaseActivity Customer.*(..)
: @Transactional(Required);
```

---

![Diagram](https://example.com/diagram.png)
Summary

• AOP is a powerful programming methodology
  – Fortifying it with metadata makes it more powerful

• Metadata and AOP combinations is synergistic
  – AOP can crosscut based metadata
  – Metadata get a principled consumer and supplier

• Projection onto multidimensional concern space is a systematic approach to view metadata

• Watch out for overuse of metadata undermining AOP principles
For More Information

• Ramnivas Laddad, Metadata and AOP: A Perfect Match
  – Part of AOP@Work series on IBM developerWorks

• AOP@Work series
  – http://www.ibm.com/developerworks/views/java/libraryview.jsp?search_by=AOP@work: