Improving Your Code with POJOs, Dependency Injection, AOP, and O/R Mapping

Chris Richardson
Author of *POJOs in Action*
Chris Richardson Consulting, Inc

http://www.chrisrichardson.net
About Chris

- Grew up in England
- Live in Oakland, CA
- Over twenty years of software development experience
  - Building object-oriented software since 1986
  - Using Java since 1996
  - Using J2EE since 1999
- Author of POJOs in Action
- Speaker at JavaOne, JavaPolis, NFJS, JUGs, .....
- Chair of the eBIG Java SIG in Oakland (www.ebig.org)
- Run a consulting and training company that helps organizations build better software faster
Overall presentation goal

Show how dependency injection, aspect-oriented programming, object/relational mapping makes code easier to develop and maintain.
Agenda

- Tangled code, tight coupling and duplication
- Using dependency injection
- Simplifying code with aspects
- Simplifying DAOs with O/R mapping
Common pattern: Big Fat Service

- Services are **tightly coupled** to other components/infrastructure APIs
- Services contain a **tangle** of business logic and infrastructure logic
- Implementation of infrastructure concerns is **scattered/duplicated** throughout the service layer
- Code is difficult to: write, test and maintain
- Dies with the infrastructure frameworks
## Example banking application

### Transfer Money

#### Transfer Between Your Accounts

<table>
<thead>
<tr>
<th></th>
<th>SAVINGS</th>
<th>(Avail. balance = $1,155.98)</th>
<th>CHECKING</th>
<th>(Avail. balance = $140.90)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Transfer From Account</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Transfer To Account</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Amount</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Transfer Description</strong></td>
<td></td>
<td><em>Descriptions appear for checking, savings, money market or market rate accounts only.</em></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(optional)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Example design
Let’s walk through the code
The good news

- Code is relatively clean
- Database access logic is encapsulated by DAOs
- Other concerns, e.g. transaction management, are implemented by other classes

BUT
Procedural code

- Anemic Domain Model
  - AccountService = Business logic
  - Account and BankingTransaction = dumb data objects

- Code is more difficult to:
  - Develop
  - Understand
  - Maintain
  - Test

- Solution: Attend my other talk

```java
public class AccountServiceImpl
    implements AccountService {

    public BankingTransaction transfer(String
        fromAccountId, String toAccountId,
        double amount) {

        Account fromAccount =    accountDao.findAccount(fromAccountId);
        Account toAccount =
            accountDao.findAccount(toAccountId);
        double newBalance = fromAccount.getBalance() –
            amount;
        fromAccount.setBalance(newBalance);
        toAccount.setBalance(toAccount.getBalance() +
            amount);

        ...
    }
```
Tangled code

Every service method contains:

- Business logic
- Infrastructure logic

Violates Separation of Concerns (SOC):

- Increased complexity
- Testing is more difficult
- More difficult to develop

Naming clash: transaction
Duplicated code

```java
public class AccountServiceImpl implements AccountService {
    private Log logger = LogFactory.getLog(getClass());
    
    public BankingTransaction transfer(String fromAccountId, String toAccountId, double amount) {
        BankingSecurityManager.verifyCallerAuthorized(AccountService.class, "transfer");
        logger.debug("Entering AccountServiceImpl.transfer()");
        TransactionManager.getInstance().begin();
        AuditingManager.getInstance().audit(AccountService.class, "transfer",
                new Object[] { fromAccountId, toAccountId, amount });
        try {
            ... 
            TransactionManager.getInstance().commit();
            logger.debug("Leaving AccountServiceImpl.transfer()");
            return txn;
        } catch (RuntimeException e) {
            logger.debug("Exception thrown in AccountServiceImpl.transfer()");
            throw e;
        } catch (MoneyTransferException e) { 
            logger.debug("Exception thrown in AccountServiceImpl.transfer()");
            TransactionManager.getInstance().commit();
            throw e;
        } finally {
            TransactionManager.getInstance().rollbackIfNecessary();
        }
    }
    
    public void create(Account account) {
        BankingSecurityManager.verifyCallerAuthorized(AccountService.class, "create");
        logger.debug("Entering AccountServiceProceduralImpl.create()");
        TransactionManager.getInstance().begin();
        AuditingManager.getInstance().audit(AccountService.class, "create",
                new Object[] { account.getAccountId() });
        try {
            ... 
            TransactionManager.getInstance().commit();
            logger.debug("Leaving AccountServiceProceduralImpl.create()");
            throw e;
        } catch (RuntimeException e) {
            logger.debug("Exception thrown in AccountServiceProceduralImpl.create()");
            TransactionManager.getInstance().rollbackIfNecessary();
        }
    }
}
```

Violates Don’t Repeat Yourself (DRY)
Tightly coupled code

- **Service instantiates DAOs**
- **References to:**
  - Singletons classes
  - Static methods
- **Consequences:**
  - Difficult to unit test
  - Difficult to develop

```java
public class AccountServiceImpl implements AccountService {

    public AccountServiceImpl() {
        this.accountDao = new JdbcAccountDao();
        this.bankingTransactionDao = new JdbcBankingTransactionDao();
    }

    public BankingTransaction transfer(String fromAccountId, String toAccountId, double amount) {
        BankingSecurityManager.verifyCallerAuthorized(AccountService.class, "transfer");
        TransactionManager.getInstance().begin();
        ...
    }
}
```
Low-level, error-prone code

- Repeated boilerplate:
  - Opening connections
  - Preparing statements
  - Try/catch/finally for closing connections, etc
- Lots of code to write and debug
- Change a class ⇒ Change multiple SQL statements

```java
public class JdbcAccountDao implements AccountDao {
  public Account findAccount(String accountId) {
    Connection con = JdbcConnectionManager
                 .getInstance().getConnection();
    PreparedStatement ps = null;
    ResultSet rs = null;
    try {
      ps = con.prepareStatement(...);
      ...      return account;    } catch (SQLException e) {
throw new RuntimeException(e);
} finally {
      JdbcConnectionManager.getInstance()           .cleanUp(con, ps, rs);
    }
  }
}
```

Violates Don’t Repeat Yourself (DRY)
So what? It works!

- Code is difficult to change ⇒ can’t keep up with the needs of the business
- Bad code/obsolete frameworks ⇒ difficult to hire/retain good people
- It’s a downwards spiral
  - Bug fixes and enhancements aren’t done correctly
  - Design continues to degrade
Improving the code

- **Dependency injection**
  - Decouples components from one another and from the infrastructure code

- **Aspect-oriented programming**
  - Eliminates infrastructure code from services
  - Implements it one place
  - Ensures DRY SOCs

- **Object/relational mapping**
  - Simplifies DAO code

Use the POJO programming model
Agenda

- Tangled code, tight coupling and duplication
- **Using dependency injection**
- Simplifying code with aspects
- Simplifying DAOs with O/R mapping
Dependency injection

- Application components depend on:
  - One another
  - Infrastructure components
- Old way: components obtain dependencies:
  - Instantiation using new
  - Statics – singletons or static methods
  - Service Locator such as JNDI
- But these options result in:
  - Coupling
  - Increased complexity
- New way: Pass dependencies to component:
  - Setter injection
  - Constructor injection
Replace instantiation with injection

```java
public AccountServiceImpl(AccountDao accountDao, BankingTransactionDao bankingTransactionDao) {
    this.accountDAO = accountDao;
    this.bankingTransactionDAO = bankingTransactionDao;
}
```

```java
public BankingTransactionDao transfer(String toAccountId, double amount) {
    BankingSecurityManager.runAsUser("transfer");
    logger.debug("Entering transfer()");
    TransactionManager.getInstance().begin();
}
```
Replace singleton with dependency injection

```java
class AccountServiceImpl {
   public BankingTransaction transfer(String fromAccountId, String toAccountId, double amount) {
      TransactionManager.getInstance().begin();
      ...
   }
}

AccountServiceImpl(AccountDao accountDao, BankingTransactionDao bankingTransactionDao) {
   this.accountDAO = accountDao;
   this.bankingTransactionDAO = bankingTransactionDao;
   transactionManager = TransactionManager.getInstance();
}
```

```java
BankingTransaction transfer(String fromAccountId, String toAccountId, double amount) {
   TransactionManager transactionManager = TransactionManager.getInstance();
   transactionManager.begin();
   ...
}
```

```java
public AccountServiceImpl(AccountDao accountDao, BankingTransactionDao bankingTransactionDao, TransactionManager transactionManager) {
   this.accountDAO = accountDao;
   this.bankingTransactionDAO = bankingTransactionDao;
   this.transactionManager = transactionManager;
}
```
Replace static dependency with injection

```java
BankingSecurityManager.verifyCallerAuthorized(AccountService.class, "transfer");

public class BankingSecurityManagerWrapper {
    public void verifyCallerAuthorized(Class<?> targetType, String methodName) {
        BankingSecurityManager.verifyCallerAuthorized(targetType, methodName);
    }
}

public AccountServiceImpl(
    ... BankingSecurityManagerWrapper bankingSecurityWrapper) {
    this.bankingSecurityWrapper = bankingSecurityWrapper;
}
```
But what about the clients?

- Clients that instantiate services need to pass in dependencies
- But they could use dependency injection too!
- Ripples through the code ⇒ messy
- We could use a hand-written factory but that’s where Spring comes into play

```java
public class AccountServiceDelegate implements AccountService {
    public AccountServiceDelegate(AccountService service) {
        this.service = service;
    }
}
```

```java
public class SpringAccountServiceTests extends AbstractSpringTest {
    protected void onSetUp() throws Exception {
        service = new AccountServiceDelegate(            new AccountServiceImpl(                new JdbcAccountDao(),                new JdbcBankingTransactionDao(),            TransactionManager.getInstance(),            AuditingManager.getInstance(),            BankingSecurityManagerWrapper.getInstance()));
    }
```
Spring lightweight container

- Lightweight container = sophisticated factory for creating objects
- Spring bean = object created and managed by Spring
- You write XML that specifies how to:
  - Create objects
  - Initialize them using dependency injection
public class AccountServiceImpl ...

```java
public AccountServiceImpl(
    AccountDao accountDao, ...)
{
    this.accountDao = accountDao;
    ...
}
```

### Spring code example

```xml
<beans>

<bean id="AccountService"
    class="AccountServiceImpl">
    <constructor-arg ref="accountDao"/>
    ...  
</bean>

<bean id="accountDao"
    class="JdbcAccountDao">
    ...
</bean>

</beans>
```

public class JdbcAccountDao implements AccountDao {

```java
...  
}
```
Using Spring dependency injection

```xml
<beans>
  <bean id="AccountServiceDelegate"
       class="net.chris...client.AccountServiceDelegate">
    <constructor-arg ref="AccountService"/>
  </bean>

  <bean id="AccountService"
       class="net.chris...domain.AccountServiceImpl">
    <constructor-arg ref="accountDao"/>
    <constructor-arg ref="bankingTransactionDao"/>
    <constructor-arg ref="transactionManager"/>
    <constructor-arg ref="auditingManager"/>
    <constructor-arg ref="bankingSecurityManagerWrapper"/>
  </bean>

  <bean id="accountDao"
       class="net.chris...domain.jdbc.JdbcAccountDao"/>

  <bean id="bankingTransactionDao"
       class="net.chris...domain.jdbc.JdbcBankingTransactionDao"/>

  <bean id="transactionManager" factory-method="getInstance"
       class="net.chris...infrastructure.TransactionManager"/>

  <bean id="auditingManager" factory-method="getInstance"
       class="net.chris...infrastructure.AuditingManager"/>

  <bean id="bankingSecurityManagerWrapper"
       class="net.chris...infrastructure.BankingSecurityManagerWrapper"/>
</beans>

ApplicationContext ctx =
    new ClassPathXmlApplicationContext("appCtx/banking-service.xml");

service = (AccountService) ctx
    .getBean("AccountServiceDelegate");
```
Eliminating Java singletons

- Spring beans are singletons (by default)
- Spring can instantiate classes such as the TransactionManager (if all of its client’s use Spring)

```java
public class TransactionManager {
    public TransactionManager() {
    }
    public void begin() {...}
}

<beans>
....
<bean id="transactionManager"
     factory-method="getInstance"
     class="net.chrisrichardson.bankingExample.infrastructure.TransactionManager"/>
<bean id="auditingManager"
     factory-method="getInstance"
     class="net.chrisrichardson.bankingExample.infrastructure.AuditingManager"/>
</beans>
```
Simplifying DAOs

```java
public class JdbcAccountDao implements AccountDao {
    public void addAccount(Account account) {
        Connection con = JdbcConnectionManager.getInstance().getConnection();
        ...
    }
}

public class JdbcAccountDao implements AccountDao {
    public JdbcAccountDao(JdbcConnectionManager connectionManager) {
        this.connectionManager = connectionManager;
    }
    public void addAccount(Account account) {
        Connection con = connectionManager.getConnection();
        ...
    }
}<bean id="accountDao"
class="net.chris...JdbcAccountDao">
    <constructor-arg ref="jdbcConnectionManager"/>
</bean>

<bean id="jdbcConnectionManager"
class="net.chrisr...JdbcConnectionManager">
    <constructor-arg ref="dataSource" />
</bean>

<bean id="dataSource"
class="org.apache...dbcp.BasicDataSource">
    <property name="driverClassName" value="org.hsqldb.jdbcDriver" />
    <property name="url" value="jdbc:hsqldb:memory:mydatabase;shutdown=true" />
    <property name="username" value="sa" />
    <property name="password" value="sa" />
</bean>
```
Revised design

Web Tier

Business Tier

Account

AccountServiceDelegate

AccountService

AccountDao

Account

AccountDao

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account

AccountDao

Account
Fast unit testing example

Create mock dependencies and inject them

```java
public class AccountServiceImplMockTests extends MockObjectTestCase {

    private AccountDao accountDao;
    private BankingTransactionDao bankingTransactionDao;
    private TransactionManager transactionManager;

    protected void setUp() throws Exception {
        accountDao = mock(AccountDao.class);
        bankingTransactionDao = mock(BankingTransactionDao.class);
        transactionManager = mock(TransactionManager.class);

        service = new AccountServiceImpl(accountDao, bankingTransactionDao, transactionManager, auditingManager, bankingSecurityWrapper);
    }

    public void testTransfer_normal() throws MoneyTransferException {
        checking(new Expectations() {{
            one(accountDao).findAccount("fromAccountId"); will(returnValue(fromAccount));
            one(accountDao).findAccount("toAccountId"); will(returnValue(toAccount));
            one(transactionManager).begin();
            ...
        }});

        TransferTransaction result = (TransferTransaction) service.transfer("fromAccountId", "toAccountId", 15.0);

        assertEquals(15.0, fromAccount.getBalance());
        assertEquals(85.0, toAccount.getBalance());

        verify();
    }
}
```
Using Spring beans in an application

- **Web application**
  - ApplicationContext created on startup
  - Web components can call AppCtx.getBean()
  - Some frameworks can automatically inject Spring beans into web components

- **Testing**
  - Tests instantiate application context
  - Call getBean()
  - Better: Use AbstractDependencyInjectionSpringContextTests for dependency injection into tests

```xml
<web>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>appCtx/banking-service.xml</param-value>
  </context-param>
</web>
```

```java
ApplicationContext ctx = WebApplicationContextUtils.getWebApplicationContext(ServletContext)
AccountService service = (AccountService) ctx.getBean("AccountServiceDelegate");
```

```java
public class SpringAccountServiceTests extends AbstractDependencyInjectionSpringContextTests {
  private AccountService service;
  
  @Override
  protected String[] getConfigLocations() {
    return new String[] { "appCtx/banking-service.xml" };  
  }

  public void setAccountServiceDelegate(AccountService service) {
    this.service = service;
  }
}
```
Demo

- Let’s walk through the revised code
Dependency injection into entities

- Domain model entities need to access DAOs etc
- But they are created by the application or by Hibernate – not Spring
- Passing DAOs as method parameters from services clutters the code
- Spring 2 provides AspectJ-based dependency injection into entities
- AspectJ changes constructors to make them invoke Spring

```java
@Configurable("pendingOrder")
public class PendingOrder {

    private RestaurantRepository restaurantRepository;

    public void setRestaurantRepository(RestaurantRepository restaurantRepository) {
        this.restaurantRepository = restaurantRepository;
    }

    public void updateDeliveryInfo(...) {
        restaurantRepository.isRestaurantAvailable(...);
    }
}
```

```xml
<bean id="pendingOrder" lazy-init="true">
    <property name="restaurantRepository" ref="restaurantRepository"/>
</bean>
```

```xml
<aop:spring-configured/>
```
Benefits of dependency injection

- Simplifies code
- Promotes loose coupling
- Makes testing easier
Agenda

- Tangled code, tight coupling and duplication
- Using dependency injection
- **Simplifying code with aspects**
- Simplifying DAOs with O/R mapping
Crosscutting concerns

- Every service method:
  - Manages transactions
  - Logs entries and exits
  - Performs security checks
  - Audit logs

- Tangled and duplicated code

- OO does not enable us to write this code in one place
Aspect-Oriented Programming (AOP)

- **AOP**
  - enables the modular implementation of crosscutting concerns
  - *i.e.* eliminates duplicate code

- **Aspect**
  - Module that implements a crosscutting concern
  - Collection of pointcuts and advice

- **Join point**
  - Something that happens during program execution
  - *e.g.* execution of public service method

- **Pointcut**
  - Specification of a set of join points
  - *e.g.* All public service methods

- **Advice**
  - Code to execute at the join points specified by a pointcut
  - *e.g.* manage transactions, perform authorization check
Spring AOP

- Spring AOP = simple, effective AOP implementation
- Lightweight container can wrap objects with proxies
- Proxy masquerades as original object
- Proxy executes extra advice:
  - Before invoking original method
  - After invoking original method
  - Instead of original method
Transaction Management Aspect

@Aspect
public class TransactionManagementAspect {
    private TransactionManager transactionManager;

    public TransactionManagementAspect(TransactionManager transactionManager) {
        this.transactionManager = transactionManager;
    }

    @Pointcut("execution(public * net.chrisrichardson..*Service.*(..))")
    private void serviceCall() {
    }

    @Around("serviceCall()")
    public Object manageTransaction(ProceedingJoinPoint jp) throws Throwable {
        transactionManager.begin();
        try {
            Object result = jp.proceed();
            transactionManager.commit();
            return result;
        } catch (MoneyTransferException e) {
            transactionManager.commit();
            throw e;
        } finally {
            transactionManager.rollbackIfNecessary();
        }
    }
}

class AccountServiceImpl {
    public BankingTransaction transfer(String fromAccountId, String toAccountId, double amount) {
        transactionManager.begin();
        try {
            ...
            transactionManager.commit();
        } catch (MoneyTransferException e) {
            transactionManager.commit();
            throw e;
        } finally {
            transactionManager.rollbackIfNecessary();
        }
    }
}
Spring configuration

```xml
<beans>
  
  <aop:aspectj-autoproxy />

  <bean id="transactionManagementAspect"
       class="net.chrisrichardson.bankingExample.infrastructure.aspects.TransactionManagementAspect">
    <constructor-arg ref="transactionManager" />
  </bean>

</beans>
```
public class AccountServiceImpl ...

    private Log logger =
        LogFactory.getLog(getClass());

    public BankingTransaction transfer(
        String fromAccountId,     
        String toAccountId, double amount) {

        try {
            logger.debug("Entering
AccountServiceImpl.transfer()"sl);

            try {
                logger.debug("Leaving
AccountServiceImpl.transfer()"sl);
            } catch (RuntimeException e) {
                logger.debug("Exception thrown in
AccountServiceImpl.transfer()", e);
                throw e;
            }
        }
    }
Auditing Aspect

```java
@Aspect
public class AuditingAspect {
    private AuditingManager auditingManager;

    public AuditingAspect(AuditingManager auditingManager) {
        this.auditingManager = auditingManager;
    }

    @Pointcut("execution(public * net.chrisrichardson..*Service.*(..))")
    private void serviceCall() {
    }

    @Before("serviceCall()")
    public void doSecurityCheck(JoinPoint jp) throws Throwable {
        auditingManager.audit(AccountService.class, "transfer", new Object[] {
            fromAccountId, toAccountId, amount });
    }
}
```

```java
public class AccountServiceImpl ...

public BankingTransaction transfer(String fromAccountId, String toAccountId, double amount) {
    ...
    auditingManager.audit(AccountService.class, "transfer", new Object[] {
        fromAccountId, toAccountId, amount });
}
```
public class AccountServiceImpl ... 

public BankingTransaction transfer(
    String fromAccountId,
    String toAccountId, double amount) {
    ... 
    public BankingTransaction transfer(String
    fromAccountId, String toAccountId,
    double amount) throws
    MoneyTransferException {
    ... 
    bankingSecurityWrapper.verifyCallerAuthorized( 
        AccountService.class, 
        "transfer");
    ... 

@Aspect
public class SecurityAspect {
    private BankingSecurityManagerWrapper 
        bankingSecurityWrapper;
    ... 
    public SecurityAspect(BankingSecurityManagerWrapper 
        bankingSecurityWrapper) {
        this.bankingSecurityWrapper = bankingSecurityWrapper;
    }
    @Pointcut("execution(public * 
        net.chrisrichardson..*Service.*(..))")
    private void serviceCall() {
    }
    @Before("serviceCall()")
    public void doSecurityCheck(JoinPoint jp) throws Throwable 
    {
        bankingSecurityWrapper.verifyCallerAuthorized(jp.getTarget() 
            .getClass(), jp.getSignature().getName());
    }
} 

In pictures

JDK/CGLib proxy

Transaction
Retry Aspect

Security Aspect

Transaction Management Aspect

Logging Aspect

Auditing Aspect

AccountService

Colorado Software Summit: October 21 – 26, 2007

© Copyright 2007, Chris Richardson

Slide 43
public class AccountServiceImpl implements AccountService {

public AccountServiceImpl (AccountDao accountDao, BankingTransactionDao bankingTransactionDao) {
    this.accountDao = accountDao;
    this.bankingTransactionDao = bankingTransactionDao;
}

public BankingTransaction transfer(String fromAccountId, String toAccountId, double amount) throws MoneyTransferException {

    Account fromAccount = accountDao.findAccount(fromAccountId);
    Account toAccount = accountDao.findAccount(toAccountId);
    assert amount > 0;
    double newBalance = fromAccount.getBalance() - amount;
    switch (fromAccount.getOverdraftPolicy()) {
    case Account.NEVER:
        if (newBalance < 0)
...
    ...}
Simpler mock object test

public class AccountServiceImplMockTests extends MockObjectTestCase {

    public void testTransfer_normal() throws MoneyTransferException {
        checking(new Expectations() {
            {                
                one(accountDao).findAccount("fromAccountId");
                will(returnValue(fromAccount));
                one(accountDao).findAccount("toAccountId");
                will(returnValue(toAccount));
                one(accountDao).saveAccount(fromAccount);
                one(accountDao).saveAccount(toAccount);
                one(bankingTransactionDao).addTransaction(
                    with(instanceOf(TransferTransaction.class)));
            }
        });

        TransferTransaction result = (TransferTransaction) service.transfer(
            "fromAccountId", "toAccountId", 15.0);
        ...
    }
}
public class AccountServiceDelegate {

    private static final int MAX_RETRIES = 2;

    public BankingTransaction transfer(String fromAccountId, String toAccountId, double amount) throws MoneyTransferException {
        int retryCount = 0;
        while (true) {
            try {
                return service.transfer(fromAccountId, toAccountId, amount);
            } catch (ConcurrencyFailureException e) {
                if (retryCount++ > MAX_RETRIES)
                    throw e;
            }
        }
    }
}

@Aspect
public class TransactionRetryAspect {

    private Log logger = LogFactory.getLog(getClass());
    private static final int MAX_RETRIES = 2;

    @Pointcut("execution(public * net.chrisrichardson..*Service.*(..))")
    private void serviceCall() {
    }

    @Around("serviceCall()")
    public Object retryTransaction(ProceedingJoinPoint jp) throws Throwable {
        int retryCount = 0;
        logger.debug("entering transaction retry");
        while (true) {
            try {
                Object result = jp.proceed();
                logger.debug("leaving transaction retry");
                return result;
            } catch (ConcurrencyFailureException e) {
                if (retryCount++ > MAX_RETRIES)
                    throw e;
                logger.debug("retrying transaction");
            }
        }
    }
}

We can delete the delegate class!
public BankingTransaction transfer(String fromAccountId, String toAccountId, double amount) {
    Account fromAccount = accountDao.findAccountWithOverdraftPolicy(fromAccountId);
    Account toAccount = accountDao.findAccount(toAccountId);
    assert amount > 0;
    double newBalance = fromAccount.getBalance() - amount;
    switch (fromAccount.getOverdraftPolicy().getOverdraftPolicyType()) {
        case OverdraftPolicy.NEVER:
            if (newBalance < 0) throw new MoneyTransferException("Insufficient funds");
            break;
        case OverdraftPolicy.ALLOWED:
            Calendar now = Calendar.getInstance();
            Calendar now = Calendar.getInstance();
            do {
                if (now.get(Calendar.YEAR) - now.get(Calendar.YEAR) <= 1) {
                    now.add(Calendar.MONTH, now.get(Calendar.MONTH));
                } else {
                    break;
                }
            } while (now.get(Calendar.MONTH) < 13);
    }

    auditingManager.audit(new Target().WithName().getTarget().getSignature());
    return 78;
}

public void doAuditing(@JoinPoint jp) throws Throwable {
    auditingManager.audit(jp.getTarget().getClass(), jp.getSignature()
        .getName(), jp.getArgs());
}

public int getOrder() {
    return 78;
}

// AccountServiceProceduralImpl.transfer(String, String, double)
Demo

- Step through the code
Spring provided aspects

- Spring framework provides important infrastructure aspects
  - Transaction Management
    - TransactionInterceptor
    - PlatformTransactionManager
  - Spring Security *a.k.a.* Acegi Security
    - MethodSecurityInterceptor
Spring transaction management

1. call transfer()
2. call invoke()
3. begin transaction
4. begin transaction
5. call transfer()
6. transfer() returns
7. commit transaction
8. commit transaction
9. invoke() returns
10. transfer() returns

Transaction management API
(JDBC, Hibernate, JDO, JTA, ...)

Some web component
AOP Proxy
Transaction Interceptor
AccountService
Transaction Manager
PlatformTransactionManager Hierarchy

Transaction Interceptor
invoke()

<<interface>>
Platform Transaction Manager
getTransaction()
commit()
rollback()

DataSource Transaction Manager
Manages local transactions

Hibernate Transaction Manager

JtaTransaction Manager
Manages global/JTA transactions

...
DataSourceTransactionManager

- Manages JDBC Connections
  - Opens and closes JDBC connection
  - Stores Connection in a ThreadLocal
- Manages transactions
  - Connection.setAutoCommit(false)
  - Connection.commit()
  - Connection.rollback()
public class JdbcConnectionManager {
    ....
    public Connection getConnection() {
        logger.debug("getting connection");
        return DataSourceUtils.getConnection(dataSource);
    }

    private void closeConnection(Connection con) {
        if (con != null) {
            logger.debug("releasing connection");
            DataSourceUtils.releaseConnection(con, dataSource);
        }
    }
}

• Delete homegrown TransactionManager and TransactionManagementAspect
Spring bean definitions

```xml
<aop:config>
  <aop:pointcut id="serviceCall"
      expression="execution(public * net.chrisrichardson..*Service.*(..))" />
  <aop:advisor advice-ref="txAdvice" pointcut-ref="serviceCall"/>
</aop:config>

<tx:advice id="txAdvice" transaction-manager="transactionManager">
  <tx:attributes>
    <tx:method name="*"
      no-rollback-for="net.chrisrichardson.bankingExample.domain.MoneyTransferException" />
  </tx:attributes>
</tx:advice>

<bean id="transactionManager"
  class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
  <property name="dataSource" ref="dataSource" />
</bean>
```
Using Aspects in the Domain Model

- Spring AOP works well for the service layer
- But it has limitations:
  - Objects must be created by Spring
  - Can only intercept calls from outside
  - Only efficient when method calls are expensive
- Inappropriate for domain model crosscutting concerns:
  - e.g. tracking changes to fields of domain objects
Introduction to AspectJ

- What is AspectJ
  - Adds aspects to the Java language
  - Superset of the Java language

- History
  - Originally created at Xerox PARC
  - Now an Eclipse project

- Uses byte-code weaving
  - Advice inserted into application code
  - Done at either compile-time or load-time
  - Incredibly powerful: *e.g.* intercept field sets and gets
Change tracking problem – old way

```java
public class Foo {

    private Map<String, ChangeInfo> lastChangedBy
        = new HashMap<String, ChangeInfo>();

    public void noteChanged(String who, String fieldName) {
        lastChangedBy.put(fieldName, new ChangeInfo(who, new Date()));
    }

    public Map<String, ChangeInfo> getLastChangedBy() {
        return lastChangedBy;
    }

    private int x;
    private int y;

    public void setX(int newX) {
        noteChanged(determineCallerIdentity(), "x");
        this.x = x;
    }
}
```

• Put in a base class
• Unless you run into single-inheritance restriction

• Call noteChanged() whenever a field value is changed.
• Tangled code
• Error prone – too easy to forget
Change tracking problem – AOP way

```java
@Observable
public class Foo {

    @Watch
    private int x;

    private int y;

    public void setX(int newX) {
        this.x = x;
    }

    Now it's a simple POJO again
```
public aspect ChangeTrackingAspect {

    declare parents: (@Observable *) implements Trackable;

    private Map<String, ChangeInfo> Trackable.lastChangedBy = new HashMap<String, ChangeInfo>();

    private void Trackable.noteChanged(String who, String fieldName) {
        lastChangedBy.put(fieldName, new ChangeInfo(who, new Date()));
    }

    public Map<String, ChangeInfo> Trackable.getLastChangedBy() {
        return lastChangedBy;
    }

    ...

**Adds the Trackable interface to all classes annotated with @Observable**

**Adds these members to all classes that implement the Trackable interface**
Tracking field sets

```java
private SecurityInfoProvider securityInfoProvider;

pointcut fieldChange(Trackable trackable, Object newValue) :
    set(@Watch * Trackable+.*) && args(newValue) && target(trackable);

after(Trackable trackable, Object newValue) returning() :
    fieldChange(trackable, newValue) {
        FieldSignature signature =
            (FieldSignature)thisJoinPointStaticPart.getSignature();
        String name = signature.getField().getName();
        String who = provider.getUser();
        trackable.noteChanged(who, name);
    }
```

```xml
<bean id="changeTracker"
        class="net.chrisrichardson.aopexamples.simple.ChangeTrackingAspect"
        factory-method="aspectOf">
    <property name="provider" ref="securityInfoProvider"/>
</bean>

<bean id="securityInfoProvider"
        class="net.chrisrichardson.aopexamples.simple.SecurityInfoProvider"/>
```

```java
Foo foo = new Foo();
foo.setX(1);
foo.setY(2);
Trackable trackable = foo;
```
Benefits of AOP

- Incredibly powerful
  - Modularizes crosscutting concerns
  - Simplifies application code
  - Decouples application code from infrastructure

- Two options:
  - Spring AOP – simple but less powerful
  - AspectJ – powerful but with a price
Drawbacks of AOP

- Cost of using AspectJ
  - Compile-time weaving – changes build
  - Load-time weaving – increases startup time
- Not everyone’s idea of simplicity
  - Code no longer explicitly says what to do
Agenda

- Tangled code, tight coupling and duplication
- Using dependency injection
- Simplifying code with aspects
- Simplifying DAOs with O/R mapping
Improving the DAO code

- Low-level, error-prone code
- Lots of repeated, boilerplate code
- Not DRY, *e.g.* add new field ⇒ change multiple places
- Not portable
- Not change tracking
- All data accesses are explicit

```java
public Account findAccount(String accountId) {
    Connection con =
        connectionManager.getConnection();
    PreparedStatement ps = null;
    ResultSet rs = null;
    try {
        ps = con.prepareStatement("SELECT * FROM BANK_ACCOUNT WHERE accountId = ?");
        ps.setString(1, accountId);
        rs = ps.executeQuery();
        Account account =
            new Account(rs.getInt("ACCOUNT_ID"),
                        rs.getString("accountId"),
                        ...)
        return account;
    } catch (SQLException e) {
        throw new RuntimeException(e);
    } finally {
        connectionManager.cleanUp(con, ps, rs);
    }
}
```
Using Spring JDBC

- **Benefits:**
  - Eliminates the boilerplate
  - Less code
  - Less error-prone
  - Portable error handling

- **But**
  - Not portable
  - Not DRY
  - No change tracking
  - ...

```java
public Account findAccount(String accountId) {
    return (Account) jdbcTemplate.queryForObject("SELECT * FROM BANK_ACCOUNT WHERE accountId = ?",
        new Object[] { accountId },
        new RowMapper() {
            public Object mapRow(ResultSet rs,...) throws SQLException {
                return new Account(rs.getInt("ACCOUNT_ID"),
                    rs.getString("accountId"),
                    rs.getDouble("BALANCE"),
                    ...);
            }
        });
}
```
Using an ORM framework

- Use an object/relational framework:
  - Developer writes metadata mapping the domain model to the database schema
  - Application manipulates objects
  - ORM framework generates SQL statements
- Hibernate
  - Very popular open-source project
- EJB 3/Java Persistence API (JPA)
  - Multiple implementations
  - Hibernate, Toplink, Open JPA/Kodo
Defining the mapping

```xml
<hibernate-mapping package="net.chrisrichardson.bankingExample.domain"
    default-access="field">
  <class name="Account" table="BANK_ACCOUNT">
    <id name="id" column="ACCOUNT_ID">
      <generator class="native" />
    </id>
    <property name="balance" column="BALANCE" />
    <property name="accountId" />
    <property name="dateOpened" type="timestamp" />
    <many-to-one name="overdraftPolicy" cascade="all" />
  </class>

  <class name="OverdraftPolicy" table="OVERDRAFT_POLICY">
    <id name="id" column="OVERDRAFT_POLICY_ID">
      ...
    </id>
    <property name="overdraftPolicyType" />
    <property name="requiredYearsOpen" />
    <property name="limit" />
  </class>

  ...
</hibernate-mapping>
```
public class HibernateAccountDao implements AccountDao {
    private HibernateTemplate hibernateTemplate;

    public HibernateAccountDao(HibernateTemplate template) {
        this.hibernateTemplate = template;
    }

    public void addAccount(Account account) {
        hibernateTemplate.save(account);
    }

    public Account findAccount(final String accountId) {
        return (Account) DataAccessUtils.uniqueResult(hibernateTemplate
            .findByNamedQueryAndNamedParam("Account.findAccountByAccountId",
            "accountId", accountId));
    }

    public Account findAccountWithOverdraftPolicy(String accountId) {
        return (Account) DataAccessUtils.uniqueResult(hibernateTemplate
            .findByNamedQueryAndNamedParam("Account.findAccountByAccountIdWithOverdraftPolicy",
            "accountId", accountId));
    }
}
Define the queries

```xml
<hibernate-mapping package="net.chrisrichardson.bankingExample.domain"
default-access="field">

... 

<query name="Account.findAccountByAccountId">
  from Account where accountId = :accountId
</query>

<query name="Account.findAccountByAccountIdWithOverdraftPolicy">
  from Account a
  a inner join fetch a.overdraftPolicy
  where a.accountId = :accountId
</query>

</hibernate-mapping>
```
public class AccountServiceImpl implements AccountService {

    public BankingTransaction transfer(String fromAccountId, String toAccountId, double amount) {
        ...
        fromAccount.setBalance(newBalance);
        toAccount.setBalance(toAccount.getBalance() + amount);
        
        /// NOT NEEDED
        // accountDao.saveAccount(fromAccount);
        // accountDao.saveAccount(toAccount);
        ...
    }
}
Spring bean definitions

```xml
<bean id="accountDao"
    class="net.chrisrichardson.bankingExample.domain.hibernate.HibernateAccountDao">
    <constructor-arg ref="hibernateTemplate" />
</bean>

<bean id="hibernateTemplate"
    class="org.springframework.orm.hibernate3.HibernateTemplate">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>

<bean id="transactionManager"
    class="org.springframework.orm.hibernate3.HibernateTransactionManager">
    <property name="sessionFactory" ref="sessionFactory" />
</bean>

<bean id="sessionFactory"
    class="org.springframework.orm.hibernate3.LocalSessionFactoryBean">
    <property name="dataSource" ref="dataSource" />
    <property name="mappingLocations" value="HibernateBankingExample.hbm.xml"/>
    ...
</bean>
```
Demo

- Walk through the code
- Step through the execution of the transfer() method
ORM framework features 1

- **Declarative mapping**
  - Map classes to tables; fields to columns; relationships to foreign keys and join tables
  - Either XML or annotations

- **Manages object identity**
  - Only one copy of an object per PK
  - Maintains consistency

- **Supports navigation between objects**
  - Application navigates relationships
  - ORM framework loads objects behind the scenes

- **Tracks changes to objects**
  - Detects which objects have changed
  - Automatically updates the database
**ORM framework features 2**

- ** CRUD API, e.g. Hibernate has:**
  - Session – CRUD API, manages object identity
  - SessionFactory – creates Session

- **Query language**
  - Retrieve objects satisfying search criteria
  - *e.g.* Hibernate Query

- **Transaction management**
  - Manual transaction management
  - Rarely call directly – used by Spring
  - *e.g.* Hibernate Transaction interface

- **Detached objects**
  - Detach persistent objects from the DB
  - Eliminates use of DTOs
  - Supports edit-style use cases
Lazy loading
- Provide the illusion that objects are in memory
- But loading all objects would be inefficient
  ⇒ load an object when it is first accessed

Eager loading
- Loading objects one at a time can be inefficient
  ⇒ load multiple objects per-select statement

Caching
- Database often the performance bottleneck
  ⇒ cache objects in memory whenever you can
- Easy for read-only objects
- Optimistic locking and cache invalidation for changing objects
O/R mapping benefits

- **Improved productivity:**
  - High-level object-oriented API
  - Less Java code to write
  - No SQL to write

- **Improved performance**
  - Sophisticated caching
  - Lazy loading
  - Eager loading

- **Improved maintainability**
  - A lot less code to write

- **Improved portability**
  - ORM framework generates database-specific SQL for you
When to use O/R mapping

- Consider using when the application:
  - Reads a few objects, modifies them, and writes them back
  - Doesn’t use stored procedures (much)

- Consider an alternative approach when:
  - Simple data retrieval ⇒ no need for objects
  - Lots of stored procedures ⇒ nothing to map to
  - Relational-style bulk updates ⇒ let the database do that
  - Some database-specific features ⇒ not supported by ORM framework

- You can always use ORM for most data accesses and JDBC for the rest
## Before and after metrics

<table>
<thead>
<tr>
<th>Metric</th>
<th>Original</th>
<th>Improved</th>
</tr>
</thead>
<tbody>
<tr>
<td>AccountService (Method LOC)</td>
<td>77 (56+19+2)</td>
<td>37 (34+1+2)</td>
</tr>
<tr>
<td>Account Service (Complexity)</td>
<td>4 (9+2+1)</td>
<td>3 (7+1+1)</td>
</tr>
<tr>
<td>AccountDao (Method LOC)</td>
<td>65 (23+21+21)</td>
<td>4 (1+3)</td>
</tr>
<tr>
<td>AccountDao (Complexity)</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>
Summary

- Dependency injection
- Aspect-oriented Programming
- Object/relational mapping

- Improved SOC
- DRY code
- Simpler code
- Improved maintainability
- Easier to develop and test
- Lets you focus on the core problem
For more information

Buy my book 😊

Send email: chris@chrisrichardson.net

Visit my website: http://www.chrisrichardson.net

Talk to me about consulting and training

Download Project Track, ORMUnit, etc.

http://code.google.com/p/projecttrack/
http://code.google.com/p/aridpojos
http://code.google.com/p/ormunit
http://code.google.com/p/umangite