Improving code with dependency injection, objects and aspects

Chris Richardson

Chris Richardson Consulting, Inc

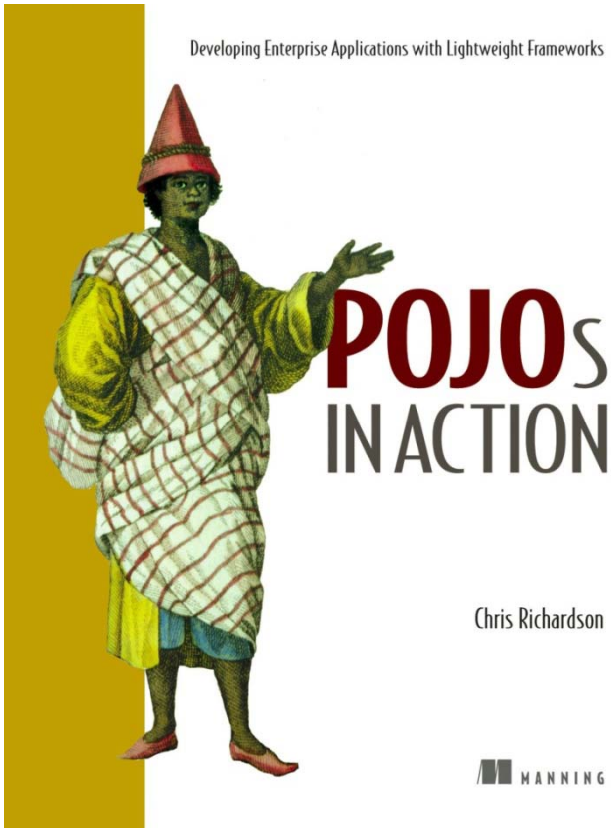
http://www.chrisrichardson.net

Overview

Use dependency injection, aspects and real objects to improve a design by reducing coupling and increasing modularity

(i.e. eliminate big fat services)

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

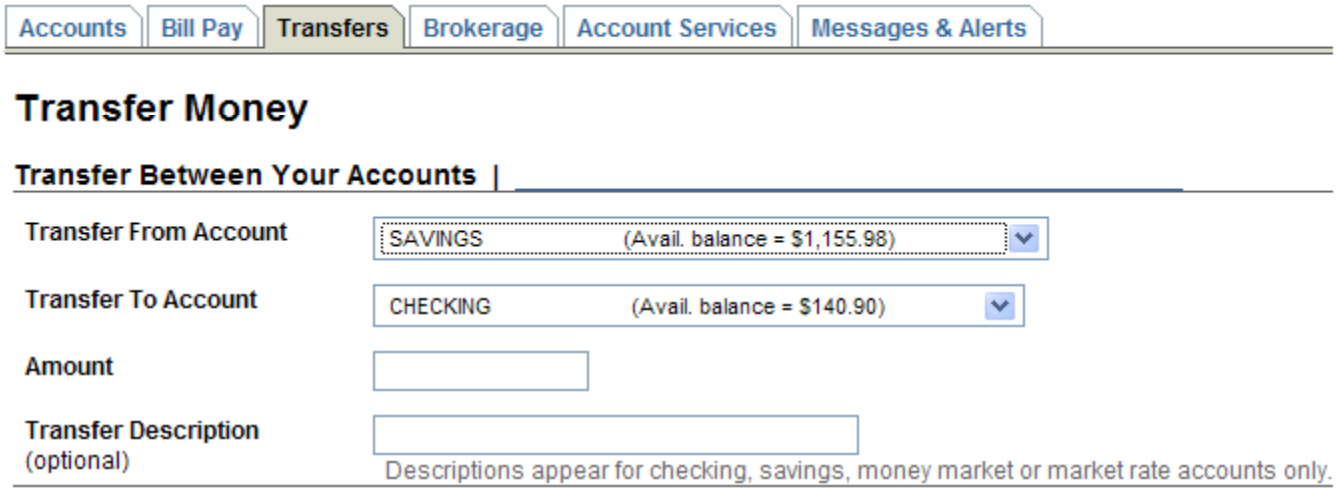
Agenda

- **Coupling, tangling and scattering**
- Using dependency injection
- Untangling code with aspects
- In search of real objects
- Cleaning up stinky procedural code

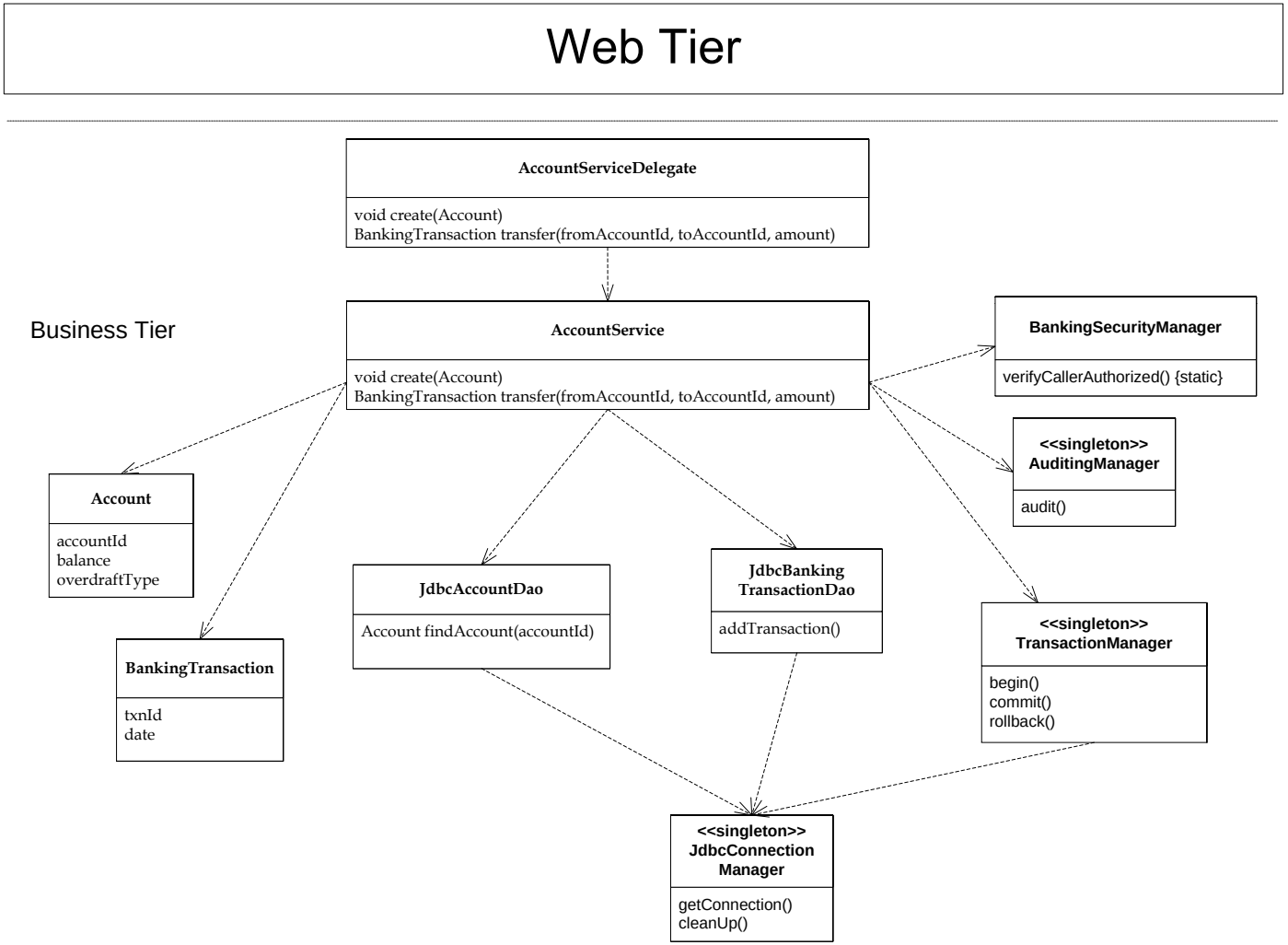
Common pattern: Big Fat Service

- Components are **tightly coupled** to one another and the 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



Example design



Demo

- Let’s walk through the code

The good news

- Code is relatively clean
- Database access logic is encapsulated by DAOs
- Other concerns such as transaction management are implemented by other classes

BUT

Tightly coupled code

- Service instantiates DAOs
- References to:
  - Singletons classes
  - Static methods
- Problems:
  - Unit testing is difficult
  - Change infrastructure ⇒ change code

```
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();

...
}
```

Tangled business logic

- Anemic Domain Model
  - AccountService = Business logic
  - Account and BankingTransaction = dumb data objects
- Fat services implement multiple features
- Violates Separation of Concerns (SOC)
- Increased complexity
- Code is more difficult to:
  - Develop
  - Understand
  - Maintain
  - Test

```
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 business logic and infrastructure

- Every service method contains:
  - Business logic
  - Infrastructure logic
- Violates Separation of Concerns (SOC):
  - Increased complexity
  - Testing is more difficult
  - More difficult to develop
  - Change infrastructure ⇒ change code
- Naming clash: transaction!

```
public class AccountServiceImpl implements AccountService {

 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 {
    Account fromAccount = accountDao.findAccount(fromAccountId);
    Account toAccount = accountDao.findAccount(toAccountId);
    double newBalance = fromAccount.getBalance() - amount;
    switch (fromAccount.getOverdraftPolicy()) {
    case Account.NEVER:
     if (newBalance < 0)
      throw new MoneyTransferException("Insufficient funds");
     break;
    case Account.ALLOWED:
     Calendar then = Calendar.getInstance();
     then.setTime(fromAccount.getDateOpened());
     Calendar now = Calendar.getInstance();

     double yearsOpened = now.get(Calendar.YEAR) - then.get(Calendar.YEAR);
     int monthsOpened = now.get(Calendar.MONTH) - then.get(Calendar.MONTH);
     if (monthsOpened < 0) {
      yearsOpened--;
      monthsOpened += 12;
     }
     yearsOpened = yearsOpened + (monthsOpened / 12.0);
     if (yearsOpened < fromAccount.getRequiredYearsOpen()
        || newBalance < fromAccount.getLimit())
      throw new MoneyTransferException("Limit exceeded");
     break;
    default:
     throw new MoneyTransferException("Unknown overdraft type: "
        + fromAccount.getOverdraftPolicy());

    }
    fromAccount.setBalance(newBalance);
    toAccount.setBalance(toAccount.getBalance() + amount);

    accountDao.saveAccount(fromAccount);
    accountDao.saveAccount(toAccount);

    TransferTransaction txn = new TransferTransaction(fromAccount, toAccount,
       amount, new Date());
    bankingTransactionDao.addTransaction(txn);

    TransactionManager.getInstance().commit();

    logger.debug("Leaving AccountServiceImpl.transfer()");
    return txn;
   } catch (RuntimeException e) {
    logger.debug(
       "Exception thrown in AccountServiceImpl.transfer()",
       e);
    throw e;
   } catch (MoneyTransferException e) {
    logger.debug(
       "Exception thrown in AccountServiceImpl.transfer()",
       e);
    TransactionManager.getInstance().commit();
    throw e;
   } finally {
    TransactionManager.getInstance().rollbackIfNecessary();
   }
  } }
```

Infrastructure

Business Logic

Infrastructure

Scattered implementations

```
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()",  e);
     throw e;
   } catch (MoneyTransferException e) {
     logger.debug("Exception thrown in AccountServiceImpl.transfer()", e);
     TransactionManager.getInstance().commit();
     throw e;
   } finally {
     TransactionManager.getInstance().rollbackIfNecessary();
   }
 }
```

- Auditing, transaction management, security, logging, ...
- Violates Don't Repeat Yourself (DRY)
- Similar code in every service method
- Potential need to change multiple places

Improving the code

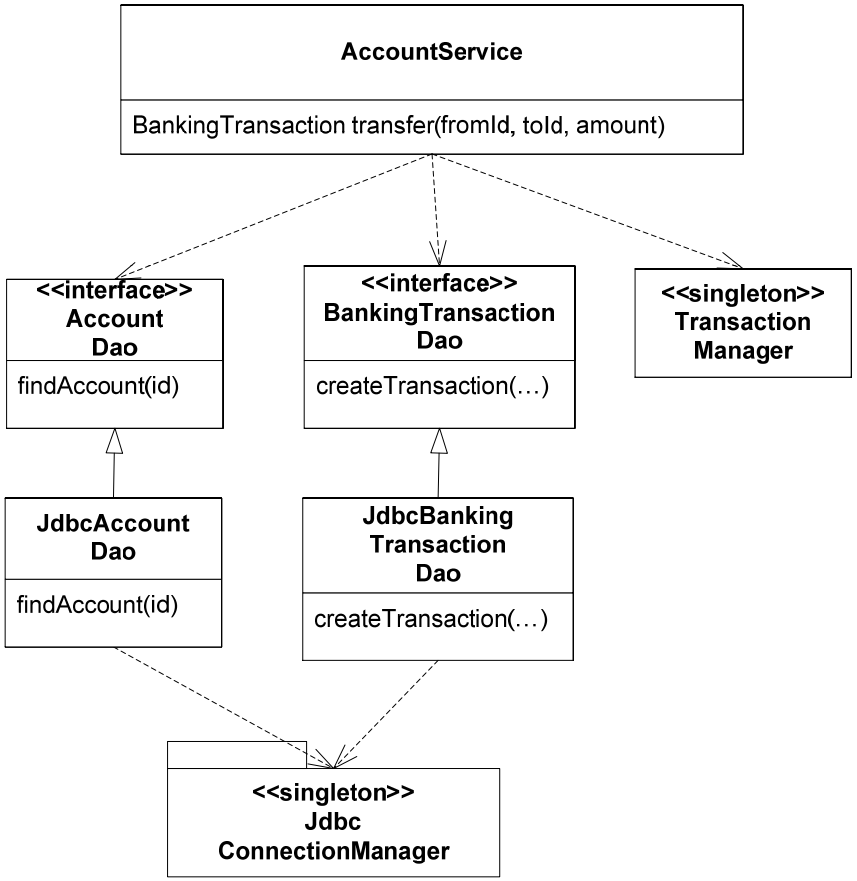
- Dependency injection
  - Decouples components from one another and from the infrastructure code
- Aspect-oriented programming
  - Eliminates infrastructure code from services
  - Implements it one place
  - Ensures DRY SOCs
- Real objects
  - Eliminates those fat services
  - Promotes SOCs and DRY

Use the POJO programming model

Agenda

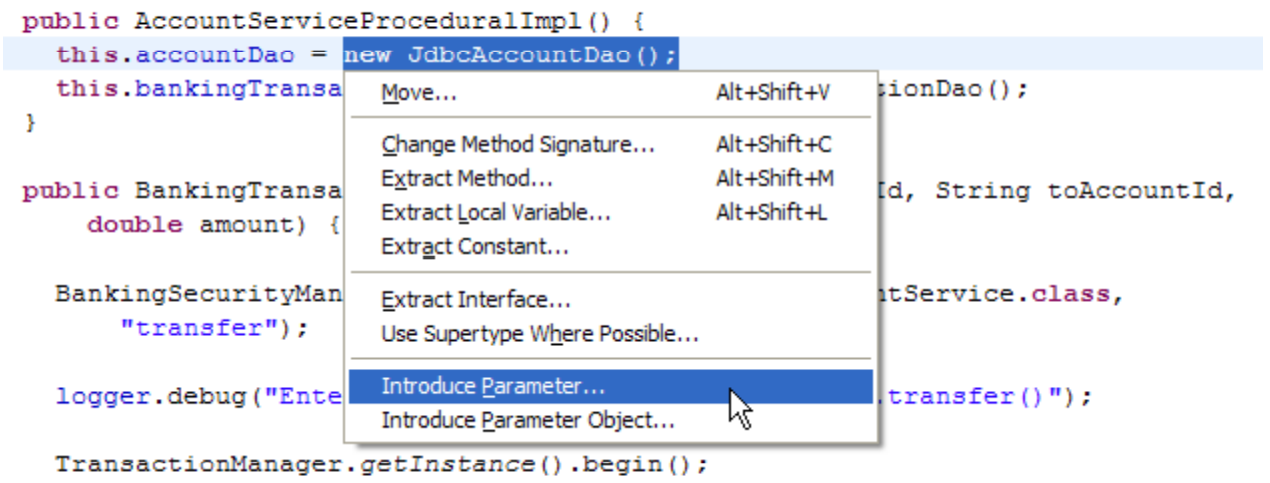
- Coupling, tangling and scattering
- **Using dependency injection**
- Untangling code with aspects
- In search of real objects
- Cleaning up stinky procedural code

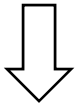
Dependency injection



- 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
- Better way: Pass dependencies to component:
  - Setter injection
  - Constructor injection

Replace instantiation with injection

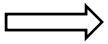




```
public AccountServiceImpl(AccountDao accountDao,
    BankingTransactionDao bankingTransactionDao) {
   this.accountDAO = accountDao;
   this.bankingTransactionDAO = bankingTransactionDao;
}
```

Eliminating other kinds of statics

```
class AccountServiceImpl ...

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

 TransactionManager.getInstance().begin();
...
}
```



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

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

```
public AccountServiceImpl(
      ...
     BankingSecurityManagerWrapper bankingSecurityWrapper) {
     ...
     this.bankingSecurityWrapper = bankingSecurityWrapper;
 }
```

The end result

```
public class AccountServiceProceduralImpl implements AccountService {

public AccountServiceProceduralImpl(AccountDao accountDao,
     BankingTransactionDao bankingTransactionDao,
     TransactionManager transactionManager, AuditingManager auditingManager,
     BankingSecurityManagerWrapper bankingSecurityWrapper) {
   this.accountDao = accountDao;
   this.bankingTransactionDao = bankingTransactionDao;
   this.transactionManager = transactionManager;
   this.auditingManager = auditingManager;
   this.bankingSecurityWrapper = bankingSecurityWrapper;
  }

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

   bankingSecurityWrapper.verifyCallerAuthorized(AccountService.class,
      "transfer");
   logger.debug("Entering AccountServiceProceduralImpl.transfer()");
   transactionManager.begin();
   auditingManager.audit(AccountService.class, "transfer", new Object[] {
      fromAccountId, toAccountId, amount });
...
}
```

We could introduce interfaces to further reduce coupling
(but that code will go away shortly)

But what about the clients?

- Components are no longer self-contained: code that instantiate them needs to pass dependencies
- Clients 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

```
public class AccountServiceDelegate implements AccountService {

  public AccountServiceDelegate() {
    this.service =  new
      AccountServiceImpl(
          new JdbcAccountDao(),
          new JdbcBankingTransactionDao(),
}
```

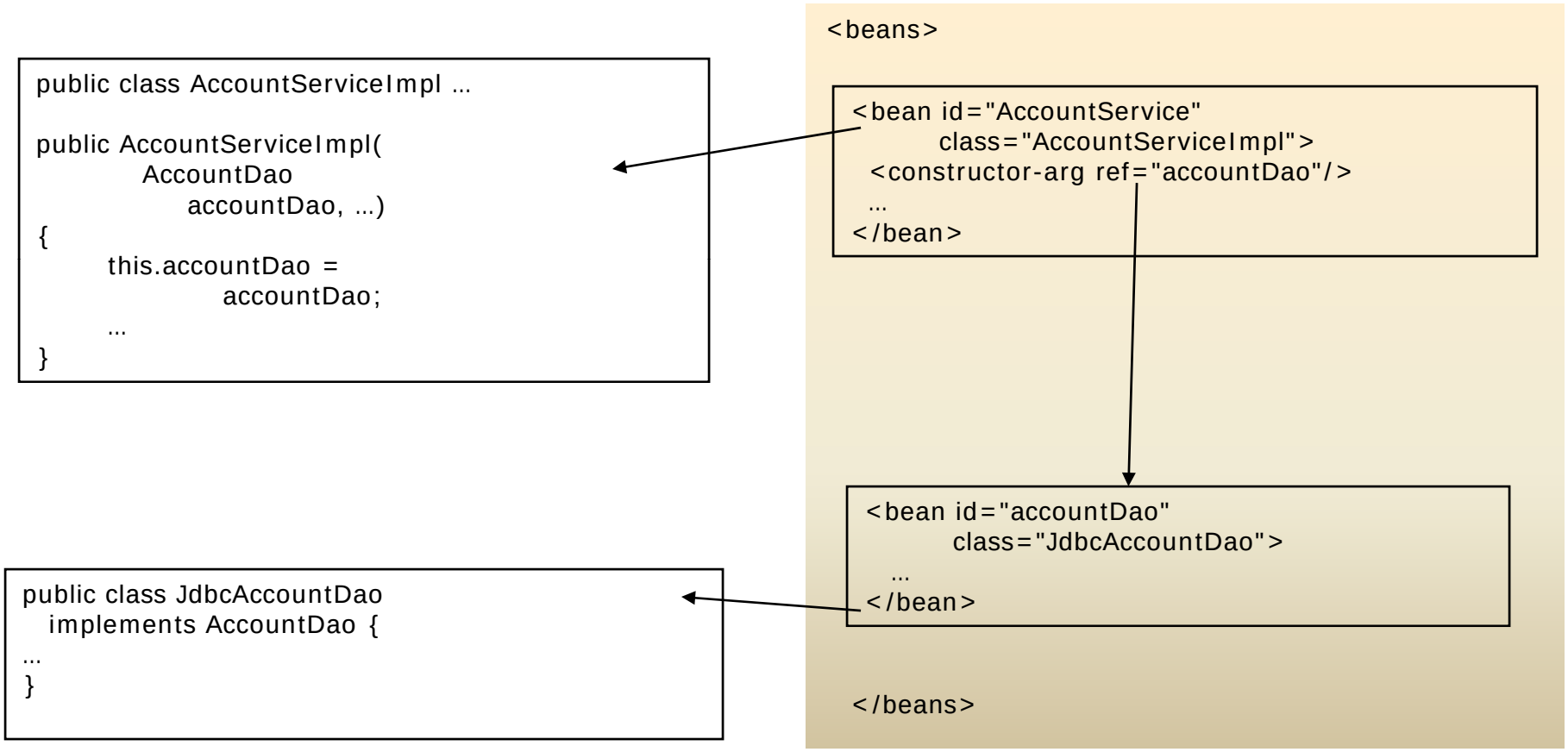
```
public class AccountServiceDelegate implements AccountService {
  public AccountServiceDelegate(AccountService service) {
    this.service =  service;
  }
}
```

```
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

Spring code example



Using Spring dependency injection

```
<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 it's client's use Spring)

```
public class TransactionManager {

public TransactionManager() {
}

public void begin() {…}
```

```
 <beans>

….
<bean id="transactionManager"
      factory-method="getInstance"
class="net.chrisrichardson.bankingExample.infras
tructure.TransactionManager"/>

<bean id="auditingManager"
   factory-method="getInstance"
class="net.chrisrichardson.bankingExample.infras
tructure.AuditingManager"/>


</beans>
```

Fast unit testing example

```
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();

  }
```

Create mock dependencies and inject them

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 AbstractDepdendency InjectionSpringConte xtTests for dependency injection into tests

```
<web>

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

 …
</web>
```

```
ApplicationCtx ctx =
   WebApplicationContextUtils.
         getWebApplicationContext(ServletContext)

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

```
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;
  }

…
}
```

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

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

private RestaurantRepository restaurantRepository;

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

```
<aop:spring-configured />

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

Benefits of dependency injection

- Promotes loose coupling
- Simplifies code
- Makes testing easier

Agenda

- Coupling, tangling and scattering
- Using dependency injection
- **Untangling code with aspects**
- In search of real objects
- Cleaning up stinky procedural code

Crosscutting concerns

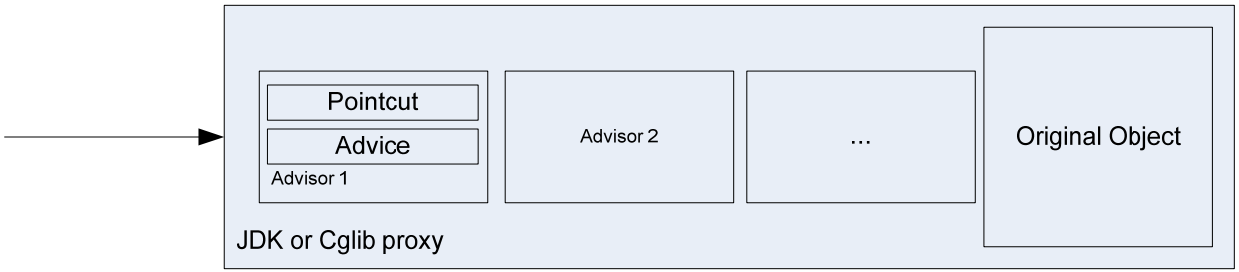
- Every service method:
  - Manages transactions
  - Logs entries and exits
  - Performs security checks
  - Does audit logging
- OO does not enable us to write this code in one place
- Scattered and tangled code

Aspect-Oriented Programming

- Aspect-Oriented Programming (AOP)
  - enables the modular implementation of crosscutting concerns
  - i.e. eliminates duplicate code
- Aspect
  - Module that implements a crosscutting concern
  - E.g. TransactionManagementAspect
  - Collection of pointcuts and advice(s)
- 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

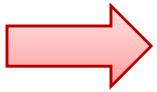
Spring AOP

- Spring AOP = simple, effective AOP implementation
- Lightweight container can wrap objects with proxies
- Proxy masquerades as original object and executes extra code (AOP advice):
  - Before invoking original method
  - After invoking original method
  - Instead of original method
- Spring uses proxies for:
  - transaction management
  - security
  - tracing
  - ...



Transaction Management Aspect

```
public 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();
    }
  }
```



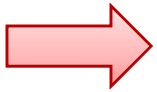
```
@Aspect
public class TransactionManagementAspect {

  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();
    }
  }

}
```

Spring configuration

```
<beans>

   <aop:aspectj-autoproxy />

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

</beans>
```

Logging Aspect

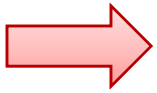
```
public class AccountServiceImpl ...

 public BankingTransaction transfer(
     String fromAccountId,
     String toAccountId, double amount) {
...
 logger.debug("Entering
                 AccountServiceImpl.transfer()");


...
  try {
      ...

 logger.debug("Leaving
                AccountServiceImpl.transfer()");

 } catch (RuntimeException e) {
     logger.debug(
        "Exception thrown in
               AccountServiceImpl.transfer()",
        e);
     throw e;
  }
```



```
@Aspect
public class LoggingAspect implements Ordered {

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

  @Around("serviceCall()")
  public Object doLogging(ProceedingJoinPoint jp) throws
Throwable {

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

    Signature signature = jp.getSignature();

    String methodName = signature.getDeclaringTypeName()
+ "." + signature.getName();

    logger.debug("entering: " + methodName);

    try {
      Object result = jp.proceed();

      logger.debug("Leaving: " + methodName);

      return result;
    } catch (Exception e) {

      logger.debug("Exception thrown in " + methodName, e);
      throw e;

    }
  }
```

Auditing Aspect

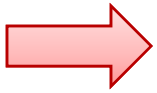
```
public class AccountServiceImpl …

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

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



```
@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(jp.getTarget().getClass(),
                          jp.getSignature().getName(),
                          jp.getArgs());
  }

}
```

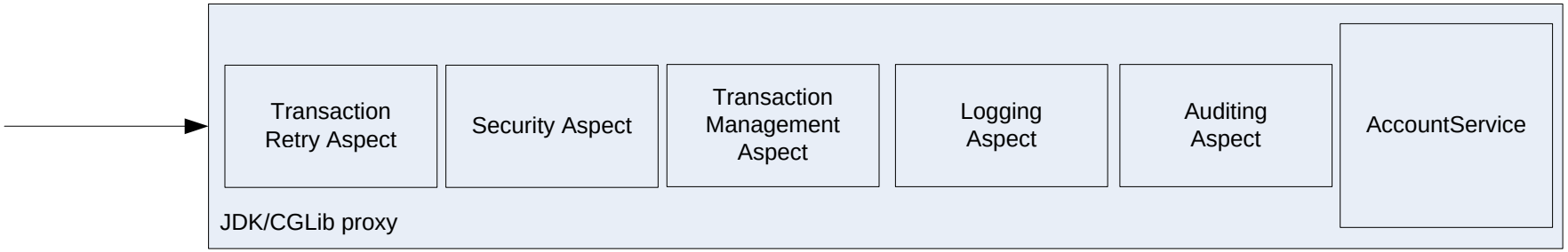
Security Aspect

```
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


Transaction Retry Aspect
Security Aspect
Transaction Management Aspect
Logging Aspect
Auditing Aspect
AccountService
JDK/CGLib proxy

Simpler AccountService

```
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)
    ....
  }

...
```

Fewer dependencies

Simpler code

It's a POJO!

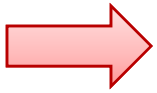
Simpler mock object test

```
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);
...
}
```

- Fewer dependencies to mock
- We are just testing the business logic

Transaction Retry Aspect

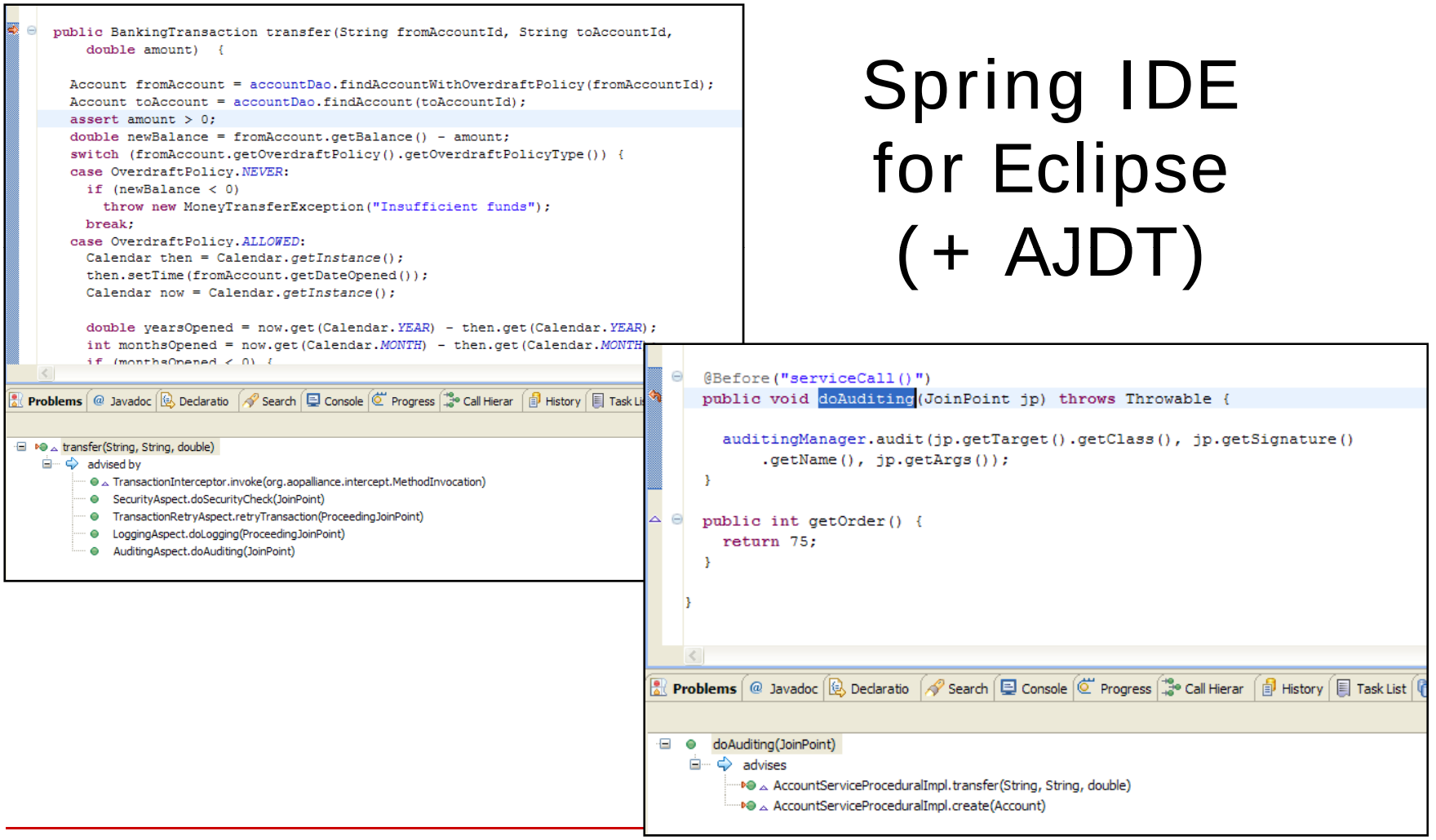
```
public class AccountServiceDelegate {

  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 remove this class!

Navigating through AOP code



Spring IDE
for Eclipse
(+ AJDT)

Demo

- Step through the code

Spring provided aspects

- Transaction Management
  - TransactionInterceptor
  - PlatformTransactionManager
- Spring Security a.k.a Acegi Security
  - MethodSecurityInterceptor

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

```
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;
  }


  @Watch
  private int x;

  private int y;
```

- Application needs to track changes to some fields
- Without AspectJ – write lots code
- With AspectJ
  - Define @Watch annotation
  - Write aspect that intercepts sets of @Watch fields

Change tracking aspect

```
public aspect ChangeTrackingAspect {

  private SecurityInfoProvider provider;

  public void setProvider(SecurityInfoProvider provider) {
    this.provider = provider;
  }

  pointcut fieldChange(Foo foo, Object newValue) :
    set(@Watch * Foo.*)  && args(newValue) && target(foo);

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

```
<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"
/>
```

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

- Coupling, tangling and scattering
- Using dependency injection
- Untangling code with aspects
- **In search of real objects**
- Cleaning up stinky procedural code

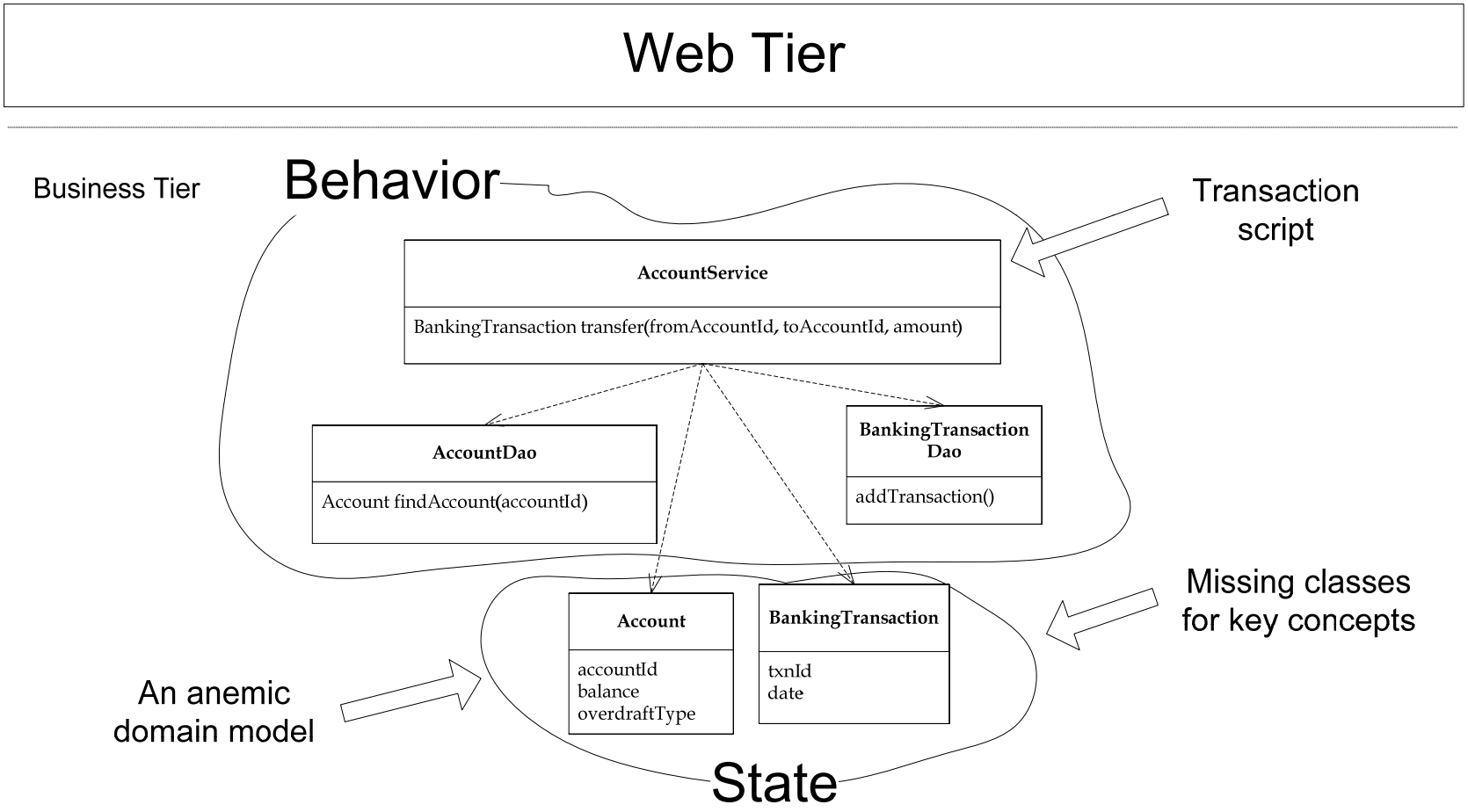
But where are the real objects?

- Using AOP has eliminated infrastructure concerns from the business logic

But

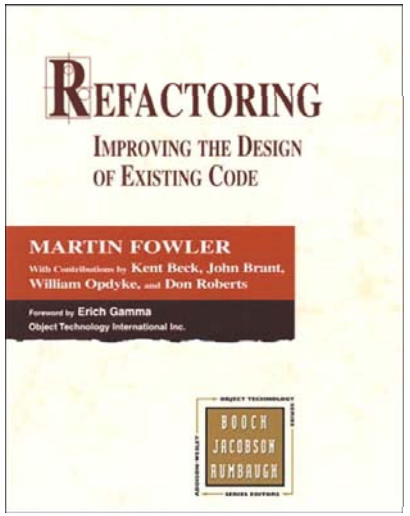
- Real Object = state + behavior
  - State – fields
  - Behavior – methods that act on the fields
- What we have here is a procedural design:
  - Business logic concentrated in fat services
  - Lack of modularity

A procedural design



Smelly procedural code

- Code smell = something about the code that does not seem right
- Impacts ease of development and testing
- Some are non-OOD
- Some are the consequences of non-OOD



Data class

- Classes that are just getters and setters
- No business logic - it's in the service
- Leads to:
  - Feature envy
- Fix by moving methods that act on data into class

```
public class Account {

 public static final int NEVER = 1;
 public static final int ALLOWED = 2;

private int id;
private double balance;
private int overdraftPolicy;
private String accountId;
private Date dateOpened;
private double requiredYearsOpen;
private double limit;

Account() {}


public Account(String accountId, double balance, int overdraftPolicy,
                Date dateOpened, double requiredYearsOpen, double
limit)
 {….. }

public int getId()  {return id;}

public String getAccountId() {return accountId;}

public void setBalance(double balance) { this.balance = balance; }

public double getBalance() { return balance; }

public int getOverdraftPolicy() { return overdraftPolicy; }

public Date getDateOpened() {   return dateOpened;  }

public double getRequiredYearsOpen() {     return requiredYearsOpen; }

public double getLimit() {return limit;  }
}
```

Feature Envy

- Methods that are far too interested in data belonging to other classes
- Results in:
  - Poor encapsulation
  - Long methods
- Fix by moving methods to the class that has the data

```
public class MoneyTransferServiceProceduralImpl implements MoneyTransferService {

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)
       throw new MoneyTransferException("In sufficient funds");
     break;
   case Account.ALLOWED:
     Calendar then = Calendar.getInstance();
     then.setTime(fromAccount.getDateOpened());
     Calendar now = Calendar.getInstance();

     double yearsOpened = now.get(Calendar.YEAR) - then.get(Calendar.YEAR);
     int monthsOpened = now.get(Calendar.MONTH) - then.get(Calendar.MONTH);
     if (monthsOpened < 0) {
       yearsOpened--;
       monthsOpened += 12;
     }
     yearsOpened = yearsOpened + (monthsOpened / 12.0);
     if (yearsOpened < fromAccount.getRequiredYearsOpen()
         || newBalance < fromAccount.getLimit())
       throw new MoneyTransferException("Limit exceeded");
     break;
   default:
     throw new MoneyTransferException("Unknown overdraft type: "
         + fromAccount.getOverdraftPolicy());

   }
   fromAccount.setBalance(newBalance);
   toAccount.setBalance(toAccount.getBalance() + amount);
   TransferTransaction txn = new TransferTransaction(fromAccount, toAccount,
       amount, new Date());
   bankingTransactionDAO.addTransaction(txn);
   return txn;
 }
```

Long method

- Methods should be short
- But business logic is concentrated in the services ⇒ long methods
- Long methods are difficult to:
  - Read and understand
  - Maintain
  - Test
- Fix:
  - Splitting into smaller methods

```
public class MoneyTransferServiceProceduralImpl implements MoneyTransferService {

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)
       throw new MoneyTransferException("In sufficient funds");
     break;
   case Account.ALLOWED:
     Calendar then = Calendar.getInstance();
     then.setTime(fromAccount.getDateOpened());
     Calendar now = Calendar.getInstance();

     double yearsOpened = now.get(Calendar.YEAR) - then.get(Calendar.YEAR);
     int monthsOpened = now.get(Calendar.MONTH) - then.get(Calendar.MONTH);
     if (monthsOpened < 0) {
       yearsOpened--;
       monthsOpened += 12;
     }
     yearsOpened = yearsOpened + (monthsOpened / 12.0);
     if (yearsOpened < fromAccount.getRequiredYearsOpen()
         || newBalance < fromAccount.getLimit())
       throw new MoneyTransferException("Limit exceeded");
     break;
   default:
     throw new MoneyTransferException("Unknown overdraft type: "
         + fromAccount.getOverdraftPolicy());

   }
   fromAccount.setBalance(newBalance);
   toAccount.setBalance(toAccount.getBalance() + amount);
   TransferTransaction txn = new TransferTransaction(fromAccount, toAccount,
      amount, new Date());
   bankingTransactionDAO.addTransaction(txn);
   return txn;
 }
```

Switch Statements

- Use of type codes and switch statements instead of polymorphism
- Key concepts are not represented by classes
- Consequences:
  - Longer methods
  - Poor maintainability caused by code duplication
  - Increased code complexity
- Fix by introducing class hierarchy and moving each part of switch statement into a overriding method

```
public class Account {

 public static final int NEVER = 1;
 public static final int ALLOWED = 2;
...
```

```
public class MoneyTransferServiceProceduralImpl
implements MoneyTransferService {

public BankingTransaction transfer(String
fromAccountId, String toAccountId,
     double amount) throws MoneyTransferException {
...
    switch (fromAccount.getOverdraftPolicy()) {
    case Account.NEVER:
      ...
     break;
    case Account.ALLOWED:
      ...
    default:
...
    }
  ...
}
```

Primitive Obsession

- Code uses built-in types instead of application classes
- Consequences:
  - Reduces understandability
  - Long methods
  - Code duplication
  - Added complexity
- Fix by moving data and code into new class

```
public class Account {
  private Date dateOpened;
}
```

```
public class Account {
  private Date dateOpened;
}

public class MoneyTransferServiceProceduralImpl implements
MoneyTransferService {

public BankingTransaction transfer(String fromAccountId, String
toAccountId,
     double amount) throws MoneyTransferException {
   Account fromAccount = accountDAO.findAccount(fromAccountId);
   Account toAccount = accountDAO.findAccount(toAccountId);
   ...
    Calendar then = Calendar.getInstance();
    then.setTime(fromAccount.getDateOpened());
    Calendar now = Calendar.getInstance();

    double yearsOpened = now.get(Calendar.YEAR) -
                              then.get(Calendar.YEAR);
    int monthsOpened = now.get(Calendar.MONTH) –
                            then.get(Calendar.MONTH);
    if (monthsOpened < 0) {
     yearsOpened--;
     monthsOpened += 12;
    }
    yearsOpened = yearsOpened + (monthsOpened / 12.0);
    if (yearsOpened < fromAccount.getRequiredYearsOpen()
       || newBalance < fromAccount.getLimit())
...
}
```

Data clumps

- Multiple fields or method parameters that belong together
- Consequences:
  - Long methods
  - Duplication
- Fix by:
  - Moving fields into their own class
  - Eliminate resulting Feature Envy

```
public class Account {

 public static final int NEVER = 1;
 public static final int ALLOWED = 2;

private int id;
private double balance;
private String accountId;
private Date dateOpened;

private int overdraftPolicy;
private double requiredYearsOpen;
private double limit;

Account() {}

}
```

A seductive programming style

- Implementing new functionality is easy
  - Add a new transaction script
  - Add code to a new transaction script
- No need to do any real design, e.g.
  - Create new classes
  - Determine responsibilities

Unable to handle complexity

- Works well for simple business logic
  - E.g. the example wasn't that bad
- But with complex business logic:
  - Large transaction scripts: 100s/1000s LOC
  - Difficult/impossible to understand, test, and maintain
- What's worse: business logic has a habit of growing
  - New requirements ⇒ Add a few more lines to the transaction script
  - Many new requirements ⇒ big mess

DEMO

Code Walkthrough

Agenda

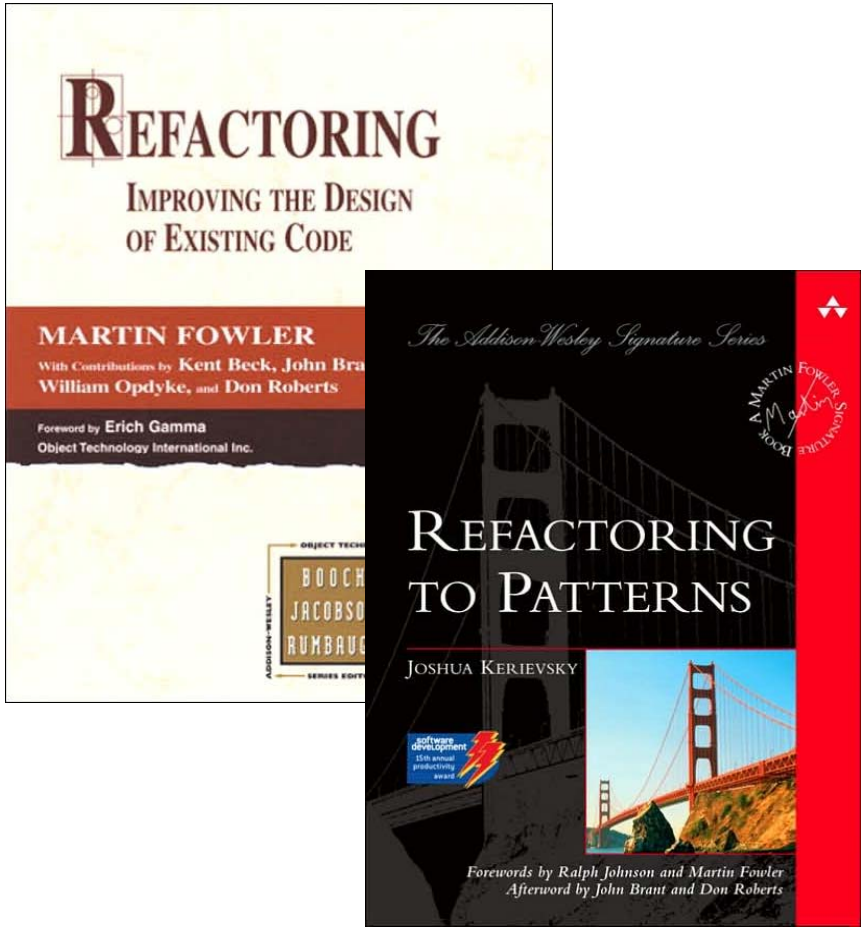
- Coupling, tangling and scattering
- Using dependency injection
- Untangling code with aspects
- In search of real objects
- **Cleaning up stinky procedural code**

Transforming procedural code

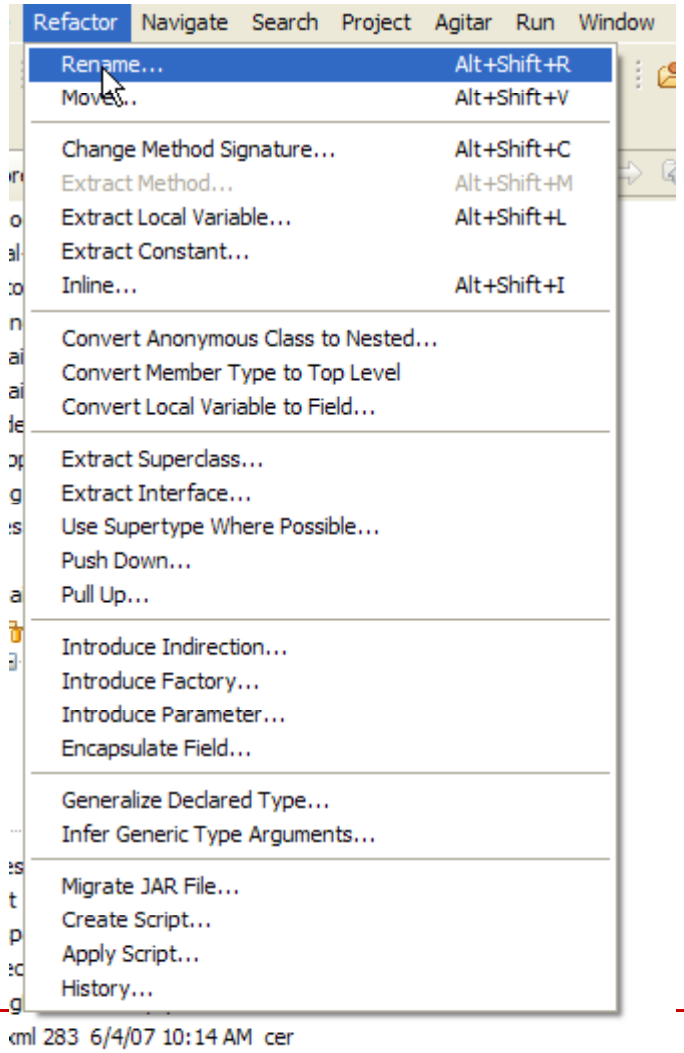
- Inside every procedural design is a domain model just trying to get out
- Incrementally transform a procedural design into an OO design
  - Small, localized changes
  - Something to do tomorrow morning!

Refactoring to an OO design

- Transform a procedural design to an OO design by applying refactorings
- Refactoring:
  - Restructure the code
  - Without changing behavior
- Essential cleanups for decaying code



Basic refactorings



- Extract Method
  - Eliminates long methods
- Move Method
  - Move a method to a different class (field or parameter)
  - Moves method to where the data is
- Push Down
  - Move a method into subclasses
  - Optionally leave an abstract method behind
  - Part of eliminating conditional logic
- ...

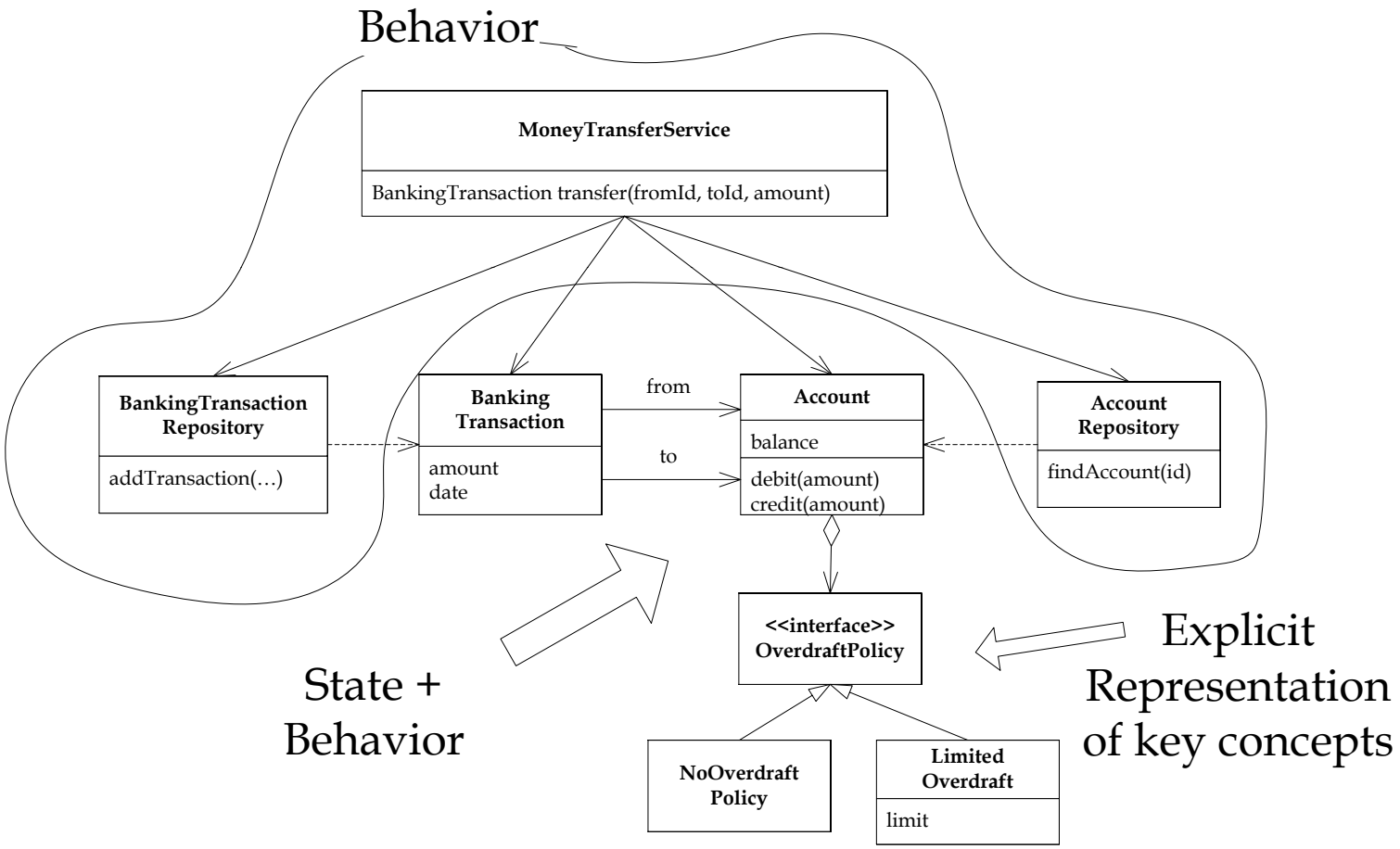
Compound refactorings

- A sequence of simpler refactorings
- Compose method
  - Apply Extract Method repeatedly
  - Use to replace long method with more readable shorter methods
- Replace Type Code With Strategy
  - Define GOF Strategy class for each type code
- Replace Conditional With Polymorphism
  - Turn into part of a switch statement into an overriding method in a subclass
- Replace Data Value with Object
  - Move field into it's own class
  - Eliminates Primitive Obsession

DEMO

- Refactoring procedural code

An example domain model



Benefits of the Domain Model Pattern

- Improved maintainability
  - The design reflects reality
  - Key domain classes are represented by classes
  - The design is more modular
- Improved testability
  - Small classes that can be tested in isolation
- Improved reusability
  - Classes can be used in other applications
- Building a domain model
  - Creates shared understanding
  - Develops an ubiquitous language

Quantifiably simpler code

Procedural – few, longer, more complex methods

Metric	Total	Mean	Std. D...	Maximum
Total Lines of Code	284			
Method Lines of Code (avg/max per method)	130	3.94	6.4	34
net.chrisrichardson.bankingExample.domain	116	4.14	6.8	34
net.chrisrichardson.bankingExample.domain.hibernate	14	2.8	3.12	9
McCabe Cyclomatic Complexity (avg/max per method)		1.33	1.15	7
net.chrisrichardson.bankingExample.domain		1.39	1.23	7
MoneyTransferServiceProceduralImpl.java		4	3	7
ExampleOfTransactionScriptSprawl.java		1.71	1.03	4
Account.java		1	0	1
TransferTransaction.java		1	0	1
MoneyTransferException.java		1	0	1

Object-oriented – more, simpler, shorter methods

Metric	Total	Mean	Std. D...	Maximum
Total Lines of Code	239			
Method Lines of Code (avg/max per method)	66	1.83	2.22	10
net.chrisrichardson.bankingExample.domain	57	1.9	2.37	10
net.chrisrichardson.bankingExample.domain.hibernate	9	1.5	1.12	4
McCabe Cyclomatic Complexity (avg/max per method)		1.14	0.54	4
net.chrisrichardson.bankingExample.domain		1.17	0.58	4
LimitedOverdraft.java		1.75	1.3	4
NoOverdraftAllowed.java		1.5	0.5	2
CalendarDate.java		1.25	0.43	2
TransferTransaction.java		1	0	1
MoneyTransferException.java		1	0	1
Account.java		1	0	1
MoneyTransferServiceImpl.java		1	0	1

Drawbacks of the Domain Model pattern

- Requires object-oriented design skills
- Requires domain model to be transparently "mappable" to the data
  - E.g. nice database schema
  - Ugly schemas and data stored in other applications is a challenge

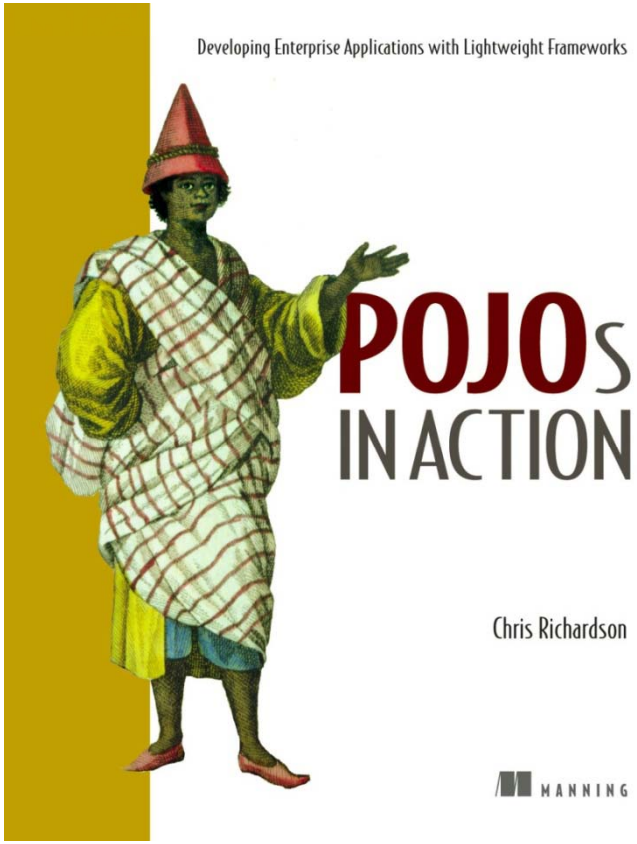
When to use it

- The business logic is reasonably complex or you anticipate that it will be
- You have the skills to design one
- You can use an ORM framework

Summary

- Dependency injection
  - Promotes loose coupling
  - Simplifies code
  - Makes code easier to test
- Aspect-oriented programming
  - Modularizes crosscutting concerns
  - Simplifies business logic
  - Decouples it from the infrastructure
- Object-oriented design
  - Organizes the business logic as classes with state AND behavior
  - Improves maintainability and testability
  - Incrementally apply by refactoring

For more information



- Buy my book ☺
  - Go to manning.com
- Send email:

chris@chrisrichardson.net

- Visit my website:

http://www.chrisrichardson.net

- Talk to me about consulting and training