Overview of POJO programming

A simpler, faster way to build long-lived applications

by

Chris Richardson
chris@chrisrichardson.net
http://www.chrisrichardson.net
About Chris…

- Grew up in England
- Live in Oakland
- 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
- Run a consulting company that helps organizations build better software faster
- Chair of the eBIG Java SIG in Oakland (www.ebig.org)
Overview

- POJOs + lightweight frameworks:
  - Simplify development
  - Accelerate development
  - Make applications immune to the volatility of enterprise Java technology

- Focus on the “backend” frameworks:
  - Business tier
  - Database access tier
Agenda

☐ The trouble with traditional enterprise Java frameworks
☐ Overview of POJOs
☐ Assembling POJO applications with dependency injection
☐ Persisting POJOs with Hibernate
☐ Making POJOs transactional with Spring
Classic EJB architecture example

<<session bean>>
TransferService

TransactionDetails transfer(fromAccountld, toAccountld, amount)

AccountDAO
AccountDTO
load Account(accountld)
saveAccount(AccountDTO)

TransactionDAO
createTransaction()

<<dto>>
Transaction Details

txnId
date

AccountDTO
accountld
balance
Problems with intertwined business logic and infrastructure

- Upgrading to new, better version of infrastructure framework is difficult/impossible:
  - Incompatible standards: EJB 1, EJB 2, EJB 3
  - Many persistence options: EJB CMP 1/2, Hibernate 1/2/3, JDO 1/2, EJB 3 persistence

- Makes development more difficult
  - Forced to think about business logic + infrastructure concerns simultaneously
  - Developers need to know both
...problems

☐ Makes testing more difficult
  ■ Must deploy code/tests in application server
  ■ Slows down the edit-compile-debug cycle

☐ EJB 2 prevented OO development

☐ EJB application servers are
  ■ Complex
  ■ Expensive (some)
EJB as a cult

- In 1999 I readily embraced EJBs and its development rituals:
  - writing DTOs and unused lifecycle methods
  - Waiting for EJBs to deploy

- According to http://en.wikipedia.org/wiki/Cult

  “a cult is a relatively small and cohesive group of people devoted to beliefs or practices that the surrounding culture or society considers to be far outside the mainstream”

- But there is a better way....
Agenda

- The trouble with traditional enterprise Java frameworks
- **Overview of POJOs**
- Assembling POJO applications with dependency injection
- Persisting POJOs with Hibernate
- Making POJOs transactional with Spring
POJO = Plain Old Java Object

- Java objects that don't implement any special interfaces or (perhaps) call infrastructure APIs
- Coined by Martin Fowler, Rebecca Parsons, and Josh MacKenzie to make them sound just as exciting as JavaBeans, Enterprise JavaBeans
- Simple idea with surprising benefits
POJO application design

POJO facade

Domain model

Database access

Account
debit(amount)
credit(amount)

<<interface>>
OverdraftPolicy

NoOverdraft
Policy
Limited
Overdraft

<<interface>>
Account
Repository
findAccount(id)

Hibernate
Account
Repository
findAccount(id)

HibernateBanking
Transaction
Repository
createTransaction(…)

Banking
Transaction

<<interface>>
Banking
Transaction
Repository
createTransaction(…)

Spring TransactionInterceptor

TransferFacade
BankingTransaction transfer(fromId, toId, amount)

TransferService
BankingTransaction transfer(fromId, toId, amount)
POJO code example

- Simple Java classes
- No lookup code – uses dependency injection instead
But POJOs are insufficient... ⇒ Lightweight frameworks

- Endow POJOs with enterprise features
- Object/relational mapping framework:
  - Persists POJOs
  - JDO, Hibernate, JPA, ...
- Spring framework:
  - Popular open-source framework
  - Declarative transaction management
  - Dependency injection
  - Remoting, security, ...
Key point: non-invasive frameworks

- Provide services without the application:
  - Implementing interfaces
  - Calling APIs
- Configured using metadata:
  - XML
  - Java 5 annotations
- POJOs + non-invasive frameworks ⇒ simple, faster development of applications that are immune to infrastructure changes
Deployment options

- Web container-only server
  - Tomcat or Jetty
  - Simple yet sufficient for many applications

- Full-blown server
  - WebLogic, JBoss, WebSphere
  - Richer set of features
  - Enhanced manageability and availability
  - JTA
  - JMS
  - ...

Benefits of using POJOs

- Separation of concerns
  - Business logic is decoupled from infrastructure
  - Switch frameworks or upgrade more easily
  - Not everybody has to be an infrastructure framework expert
- Simpler development
  - Think about one thing at a time
  - Business logic, persistence, transaction management....
- Faster development
  - Testing without an application server (or a database)
  - No deployment to slow you down
- More maintainable
  - Modular object-oriented code
  - Loosely coupled design
- Simpler, perhaps cheaper deployment
  - Deploy in a web-container only server
Drawbacks of POJOs...

- ...none except that lightweight frameworks have their limitations

- Use EJBs if you need:
  - Distributed transactions initiated by a remote client
  - Some application server-specific features
  - ...

6/20/2006
Agenda

- The trouble with traditional enterprise Java frameworks
- Overview of POJOs
  - Assembling POJO applications with dependency injection
- Persisting POJOs with Hibernate
- Making POJOs transactional with Spring
Dependency injection

- Application components depend on:
  - One another
  - Infrastructure components

- Using JNDI or the new operator:
  - Introduces coupling
  - Complexity

- Solution:
  - Pass dependencies to a component
  - Setter injection
  - Constructor injection
Dependency injection example

```java
public class MoneyTransferServiceImpl
{
    public MoneyTransferServiceImpl(
        AccountRepository accountRepository, ...)
    {
        this.accountRepository = accountRepository;
        ...
    }
}
```

```java
public class HibernateAccountRepository implements AccountRepository {
    ...
}
```

- You can implement dependency injection by hand but ....
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

```java
public class MoneyTransferServiceImpl {
    ...
    public MoneyTransferServiceImpl(
        AccountRepository accountRepository, ...)
    {
        this.accountRepository = accountRepository;
        ...
    }
}

public class HibernateAccountRepository implements AccountRepository {
    ...
}
```

```xml
<bean name="MoneyTransferService"
      class="MoneyTransferServiceImpl">
    <constructor-arg ref="AccountRepository"/>
    ...
</bean>

<bean name="AccountRepository"
      class="HibernateAccountRepository">
    ...
</bean>
```
Spring 2 – dependency injection into entities

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

```java
@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

☐ Simplifies code
  ■ No calls to JNDI

☐ Decouples components from:
  ■ One another
  ■ Infrastructure

☐ Simplifies testing
  ■ Pass in a mock/stub during testing
Mock object code example

- Test the MoneyTransferServiceImpl without calling the real AccountRepository
Agenda

- The trouble with traditional enterprise Java frameworks
- Overview of POJOs
- Assembling POJO applications with dependency injection
- Persisting POJOs with Hibernate
- Making POJOs transactional with Spring
POJO persistence

- Use an object/relational framework:
  - Metadata maps the domain model to the database schema
  - ORM framework generates SQL statements
- Hibernate
  - Very popular open-source project
- JDO
  - Standard from Sun – JSR 12 and JSR 243
  - Multiple implementations: Kodo JDO, JPOX
- EJB 3/Java Persistence API (JPA)
Hibernate: code example

- Provides transparent persistence

- Pieces:
  - Account
  - HibernateBankingExample.hbm.xml
  - HibernateAccountPersistenceTests
  - HibernateAccountRepository
  - HibernateAccountRepositoryTests
  - Spring beans

- Only the repositories/DAOs call persistence framework APIs
ORM framework features 1

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

- **CRUD API**
  - E.g. Hibernate Session, JPA EntityManager

- **Query language**
  - Retrieve objects satisfying search criteria

- **Transaction management**
  - Manual transaction management
  - Rarely call directly - used by Spring

- **Detached objects**
  - Detach persistent objects from the DB
  - Eliminates use of DTOs
  - Supports edit-style use cases
ORM framework features 2

- Lazy loading
  - Provide the illusion that objects are in memory
  - But loading all objects would be inefficient
  - \( \Rightarrow \) load an object when it is first accessed

- Eager loading
  - Loading objects one at a time can be inefficient
  - \( \Rightarrow \) load multiple objects per-select statement

- Caching
  - Database often the performance bottleneck
  - \( \Rightarrow \) cache objects in memory whenever you can
  - Easy for readonly objects
  - Optimistic locking and cache invalidation for changing objects
O/R mapping framework 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 and when not to use an ORM framework

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

☐ Don’t use 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
Agenda

☐ The trouble with traditional enterprise Java frameworks
☐ Overview of POJOs
☐ Assembling POJO applications with dependency injection
☐ Persisting POJOs with Hibernate

➢ Making POJOs transactional with Spring
Making POJOs transactional

- EJB 2 container-managed transactions are great
- Spring provides declarative transactions for POJOs
- Similar to CM transactions but
  - Runs outside of an application server
  - More flexible exception handling
Spring AOP

- AOP enables the modular implementation of crosscutting concerns
- Spring AOP = simple, effective AOP implementation
- Lightweight container can wrap objects with proxies
- Proxy executes extra code:
  - Before original method
  - After original method
  - Instead of...
- Spring uses proxies for:
  - transaction management
  - security
  - tracing
  - ...

6/20/2006
Copyright (c) 2006 Chris Richardson. All rights reserved.
Spring TransactionInterceptor

1. call transfer()

2. begin transaction

3. begin transaction

4. call transfer()

5. transfer() returns

6. commit transaction

7. commit transaction

8. transfer() returns

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

Presentation Tier

Transaction Interceptor

Account Management Facade

Platform Transaction Manager
Spring code example

```xml
<bean
    name="AccountManagementFacade"
    class="AccountManagementFacadeImpl">
    ...
</bean>

<bean
    id="transactionProxyCreator"
    class="...BeanNameAutoProxyCreator">
    <property name="beanNames">
        <list>
            <idref
                bean="AccountManagementFacade"/>
        </list>
    </property>
    <property name="interceptorNames">
        <list>
            <idref
                bean="BankingTransactionInterceptor"/>
        </list>
    </property>
</bean>

<bean
    id="myTransactionManager"
    class="HibernateTransactionManager">
    ...
</bean>

<bean id="BankingTransactionInterceptor"
    class="TransactionInterceptor">
    <property name="transactionManager"
        ref="myTransactionManager"/>
</bean>

<bean id="myTransactionManager"
    class="HibernateTransactionManager">
    ...
</bean>
```
Spring 2 – simplified XML

```xml
<bean
    name="AccountManagementFacade"
    class="AccountManagementFacadeImpl">
    ...
</bean>

<bean id="transactionManager"
    class="HibernateTransactionManager">
    ...
</bean>

<aop:config>
    <aop:advisor
        pointcut="execution(* *..*Facade.*(..))"
        advice-ref="txAdvice"/>
</aop:config>

<tx:advice id="txAdvice">
    <tx:attributes>
        <tx:method name="*"/>
    </tx:attributes>
</tx:advice>
```
Spring remoting

- Remoting
  - Spring HTTP
  - Hessian/Burlap
  - RMI
  - ...
- Server uses a
  - `<Xyz>Exporter bean`
  - Service to expose
  - Interface to expose
- Client uses a
  - `<Xyz>ProxyFactoryBean`
  - URL to remote service

```xml
<bean name="/accountManagement"
  class="org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter">
  <property name="service"
    ref="TransferFacade"/>
  <property name="serviceInterface"
    value="net.chrisrichardson...TransferFacade"/>
</bean>

<bean id="httpInvokerProxy"
  class="org.springframework.remoting.httpinvoker.HttpInvokerProxyFactoryBean">
  <property name="serviceUrl"
    value="http://somehost:8080/accountManagement"/>
  <property name="serviceInterface"
    value="net.chrisrichardson...TransferFacade"/>
</bean>
```
Spring Security

- Acegi Security
  - Open source project
  - Extension to Spring
- MethodSecurityInterceptor
- Verifies that caller is authorized
  - Invoke method
  - Access instances

```xml
<bean id="transferSecurity" class="org.acegisecurity.intercept.method.aopalliance.MethodSecurityInterceptor">
  ...  
  <property name="objectDefinitionSource">
    <value>
      net.chrisrichardson...
      TransferFacade.*=
      ROLE_CUSTOMER, ROLE_CSR
    </value>
  </property>
</bean>
```
Deploying a Spring application

- Often packaged as a WAR
- Web.xml lists bean definition files
- ServletContextListener creates Spring bean factory
- Web tier is either:
  - Injected with Spring beans
  - Calls getBean()

```xml
<web-app>
  <context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/beans1.xml
    /beans2.xml</param-value>
  </context-param>
  <listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
  </listener>
  ...
</web-app>
```
Summary

- POJOs + Non-invasive frameworks = Non-invasive frameworks

  - Simplify development
  - Accelerate development
  - Improve maintainability
  - Increase immunity to rapidly evolving infrastructure frameworks
For more information

☐ Buy my book 😊

☐ Send email: chris@chrisrichardson.net

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

☐ Please hand in your session evaluations
Extra slides
Thoughts about EJB 3 and POJOs

😊 Better than EJB2
😊 Supports POJOs
😊 Reasonable ORM
😊 Entity beans = JPA
😊 Annotations are concise
😊 Has dependency injection
😊 It’s a standard

😊 Less powerful than Spring, e.g. DI relies on JNDI
😊 Less powerful than Hibernate, e.g. List<String>
😊 Session beans/MDBs must be deployed
😊 Complexity of EJB lurking within
😊 Annotations couple your code to EJB3
🤞 EJB’s poor track record as a standard
Using Spring with EJBs

- Simplify EJB client code with Spring
  - Spring encapsulates JNDI lookup
  - Client gets EJB reference from Spring
  - Better: Client is injected with EJB reference

- Move business logic into Spring beans
  - Session EJBs delegate to Spring beans
  - Use Spring dependency injection
  - Simpler code, easier testing

- Simplify DAOs with Spring JDBC
  - Eliminates error-prone boilerplate code
Migrating to POJOs – part 1

- 2 year old application:
  - Session EJBs
  - Entity Bean-based domain model
  - Some JDBC DAOs
  - Beginning development of version 2

- Replaced entity beans with Hibernate:
  - WAS vs. WLS portability
  - Test business logic without persistence
  - Test persistence without a server
  - A much richer domain model
Migrating to POJOs – part 2

- Used Spring beans for V2 code
- Incrementally replaced V1 session beans with Spring beans when:
  - Enhancing it
  - V2 code needed to call V1 code
- End result:
  - Richer domain model
  - Faster development
  - V2 code was deployable as a web app.