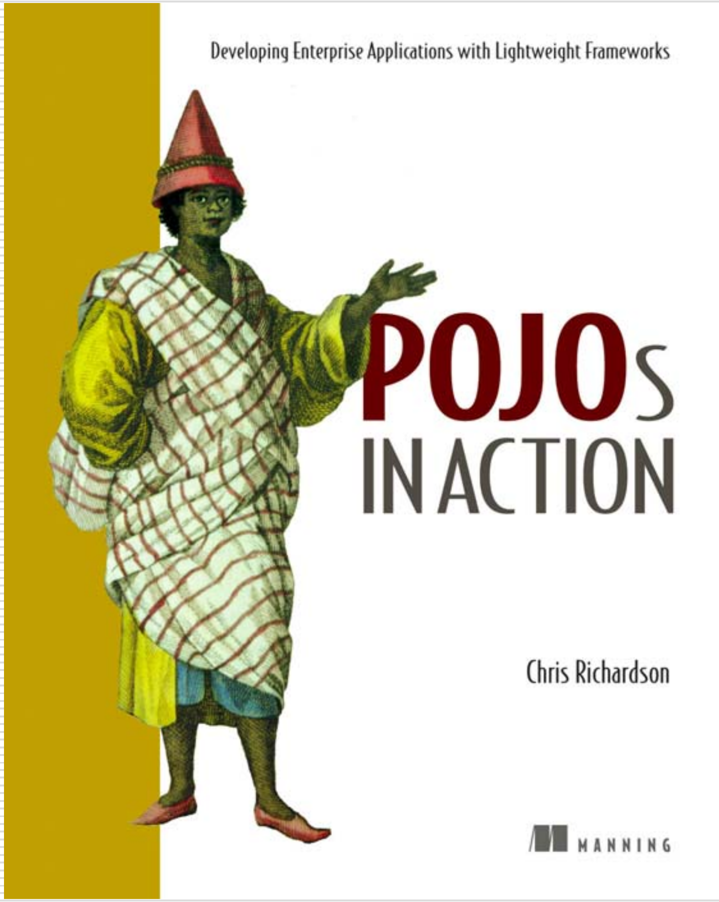
POJOs to the rescue

Easier and faster development with POJOs and lightweight frameworks

by

Chris Richardson
cer@acm.org
http://chris-richardson.blog-city.com

Who am I?



- Twenty years of software development experience
  - Building object-oriented software since 1986
  - Developing with Java since 1996
- Author of POJOs in Action
- Run a consulting company that helps organizations develop software more effectively

Overview

- EJBs really are (mostly) a bad idea
- POJOs and lightweight frameworks make development easier and faster

Agenda

- Overview of POJOs and lightweight frameworks
- The strengths and weaknesses of EJBs
- Developing applications with POJOs
- Example of a POJO design
- Where does EJB 3 fit in?
- Migrating to POJOs

POJO = Plain Old Java Object

- Java objects that don't implement any special interfaces
- Coined by Fowler to make it sound just as exciting as JavaBeans, Enterprise JavaBeans
- Simple idea with surprising benefits
- But POJOs are insufficient…

Lightweight frameworks

- Endow POJOs with enterprise features
- Object/relational mapping frameworks
  - Make POJOs persistence
  - JDO
  - Hibernate
- Spring framework
  - Popular open-source framework for simplifying J2EE development
  - Lightweight container for POJOs
  - Provides declarative transactions for POJOs
  - Makes it easier to use JDO and Hibernate

Agenda

- Overview of POJOs and lightweight frameworks
- The strengths and weaknesses of EJBs
- Developing applications with POJOs
- Example of a POJO design
- Where does EJB 3 fit in?
- Migrating to POJOs

EJB unique strengths

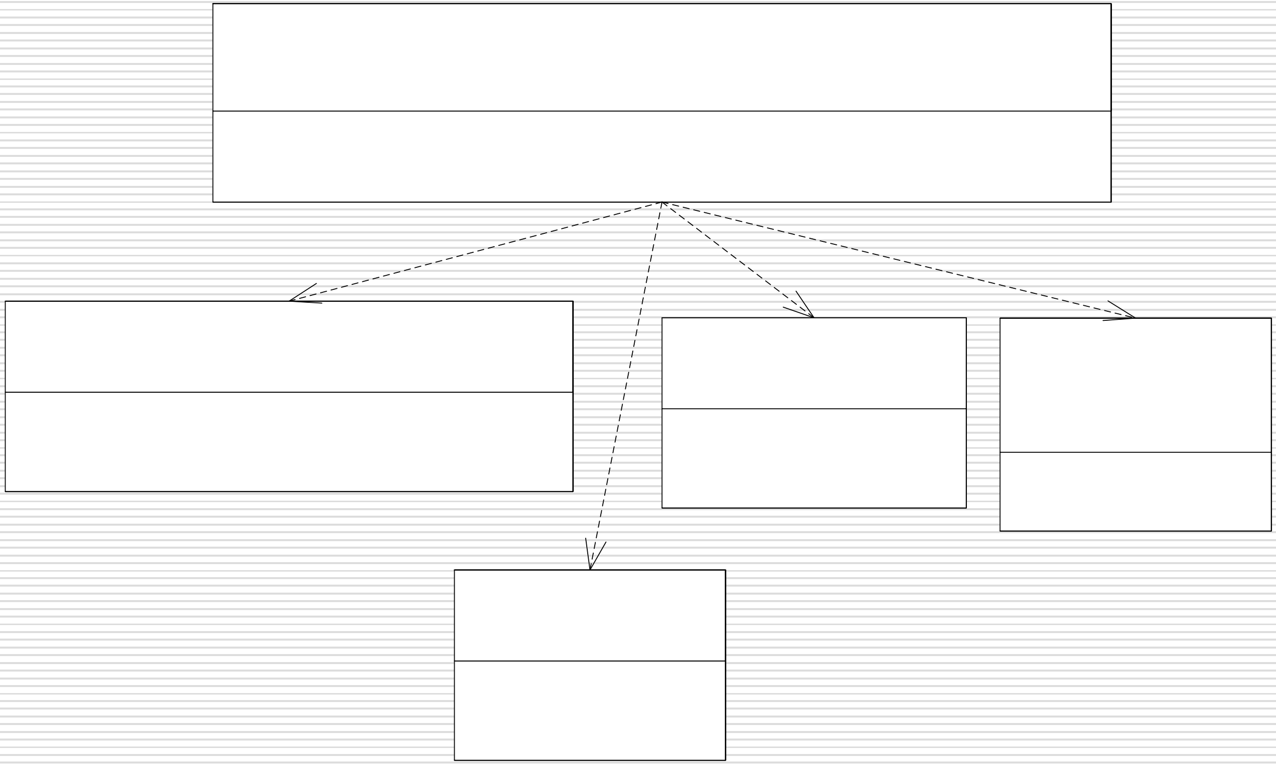
- Truly distributed applications where EJBs participate in transactions initiated by a remote client
- Messaging-oriented applications that can benefit from Message-driven beans
- Its a standard
- But there are better ways to have declarative transactions and security

EJBs weaknesses

- The deployment ritual
- Excessive complexity
- Writing code that does nothing
- Lack of support for OO development

⇒ Developing enterprise Java applications more difficult that it should be

Classic EJB architecture example



Look at the code

Problem #1: Lots of procedural code

- This a procedural design
  - Business logic is concentrated in the EJB
- Problems with procedural code
  - Doesn't handle complexity
    - Session beans contain large amounts of code
  - Difficult to extend

Why procedural?

- Its easy – just add more code to a session bean
- Encouraged by the J2EE literature, which emphasizes EJBs
  - EJB developers just love to talk about their beans
  - Session beans and Message-Driven beans play a central role but are procedural components
- Lack of support for persisting a domain model:
  - Entity beans are broken
  - Doing it with JDBC is too difficult

$\Rightarrow$ EJBs make procedural programming easy and object-oriented programming difficult

My break from object-oriented programming

- 1986-1999 – Object-oriented (CLOS/C++/Java)
- 1999-2002 – Procedural (EJB + JDBC)
- 2002-2004 - Simple object-oriented (EJB + EJB 2 CMP)
- 2004 - Object-oriented (Spring Hibernate/JDO)

Problem #2 – JDBC code

- EJB 2 entity beans
  - Bad reputation
  - Lots of limitations
- DAOs:
  - JDBC code
  - Handwritten SQL difficult to maintain
  - Not very portable
- EJB applications often contain a lot of it

Problem #3 - Code is coupled to the server environment

- EJBs are server-side components
- DAOs that use JNDI must run in the server

⇒ Long edit-compile-debug cycles:
  - Hot code replacement helps but it has its limitations
  - Once you make a non-trivial change you have to restart the server (2 minutes)

⇒ Testing is more difficult:
  - Remote interfaces
  - Local interfaces with Cactus

Problem #4 - complexity

- Lots of code that does nothing
  - ejbActivate()/Passivate() methods for stateless session beans
- XML deployment descriptors
  - Or XDoclet comments
- General development time complexity
  - Running XDoclet
  - Server configuration
  - IDE setup

⇒ All this extra stuff just to run some code

EJB as a cult

- 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"

- In 1999 I readily embraced EJBs and thought they were great

Once you escape the cult you realize…

- There is no reason why writing server-side code should be so different
- Development is slow
- Excessive complexity
- Lacking key features

$\Rightarrow$ Use EJBs only for
  - Distributed transactions
  - Message-driven beans

Agenda

- Overview of POJOs and lightweight frameworks
- The strengths and weaknesses of EJBs
- Developing applications with POJOs
- Example of a POJO design
- Where does EJB 3 fit in?
- Migrating to POJOs

Developing with POJOs

- How I escaped
- The characteristics of a POJO design
- Benefits of a POJO design

Entity Beans $\Rightarrow$ POJOs + Hibernate

- Classic J2EE architecture
  - Session beans for declarative transactions/security
  - Entity beans persisted a simple domain model
  - DAOs for queries that couldn't use Entity beans
  - Ran on WebLogic
- But
  - Jumped through hoops to persist a domain model
  - Long edit-compile-debug cycles
- The final straw was when we needed to support WAS and WLS
  - Non-standard CMP
- Motivated us to migrate to Hibernate
  - Provided portability
  - Simplified development of the persistence layer
  - Enabled us to develop a very elaborate domain model

Session beans ⇒ POJOs + Spring

- Spent three days at TSSJS 2004 being indoctrinated:
  - Spring
  - Dependency injection
  - AOP
- Use POJO facades instead of session beans
  - Spring provides declarative transaction management
- Development went so much faster
  - Test code outside of the server
  - Test using Jetty, which starts up in a couple of seconds
- Spring+Hibernate totally transformed the development experience
- This was a real Tivo moment

EJB design vs. POJO design

Design decision	EJB design	POJO design
Organization	Procedural-style business logic	Object-oriented design
Implementation	EJB-based	POJOs
Database access	JDBC/SQL or Entity beans	Persistence framework
Returning data to the presentation tier	DTOs	Business objects
Transaction management	EJB container-managed transactions	Spring framework
Application assembly	Explicit JNDI lookups	Dependency injection

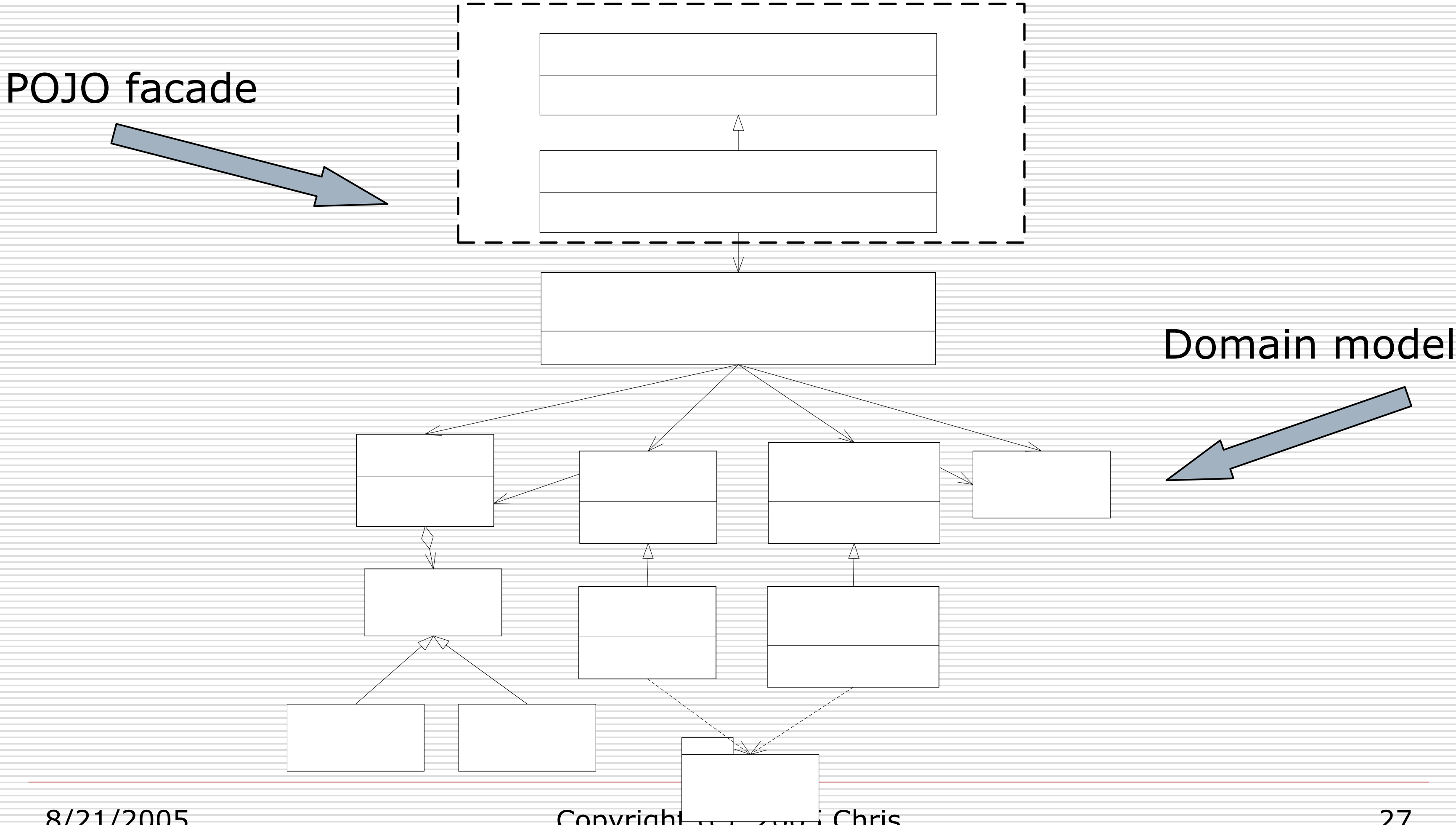
Benefits of using POJOs

- Simpler development
  - Test without an application server
  - Business logic and persistence are separate
- Faster development
  - Test without deploying
  - Easier testing
- More maintainable
  - Modular object-oriented code
  - No handwritten SQL
  - Loosely coupled design
- Decouple technologies from core business logic

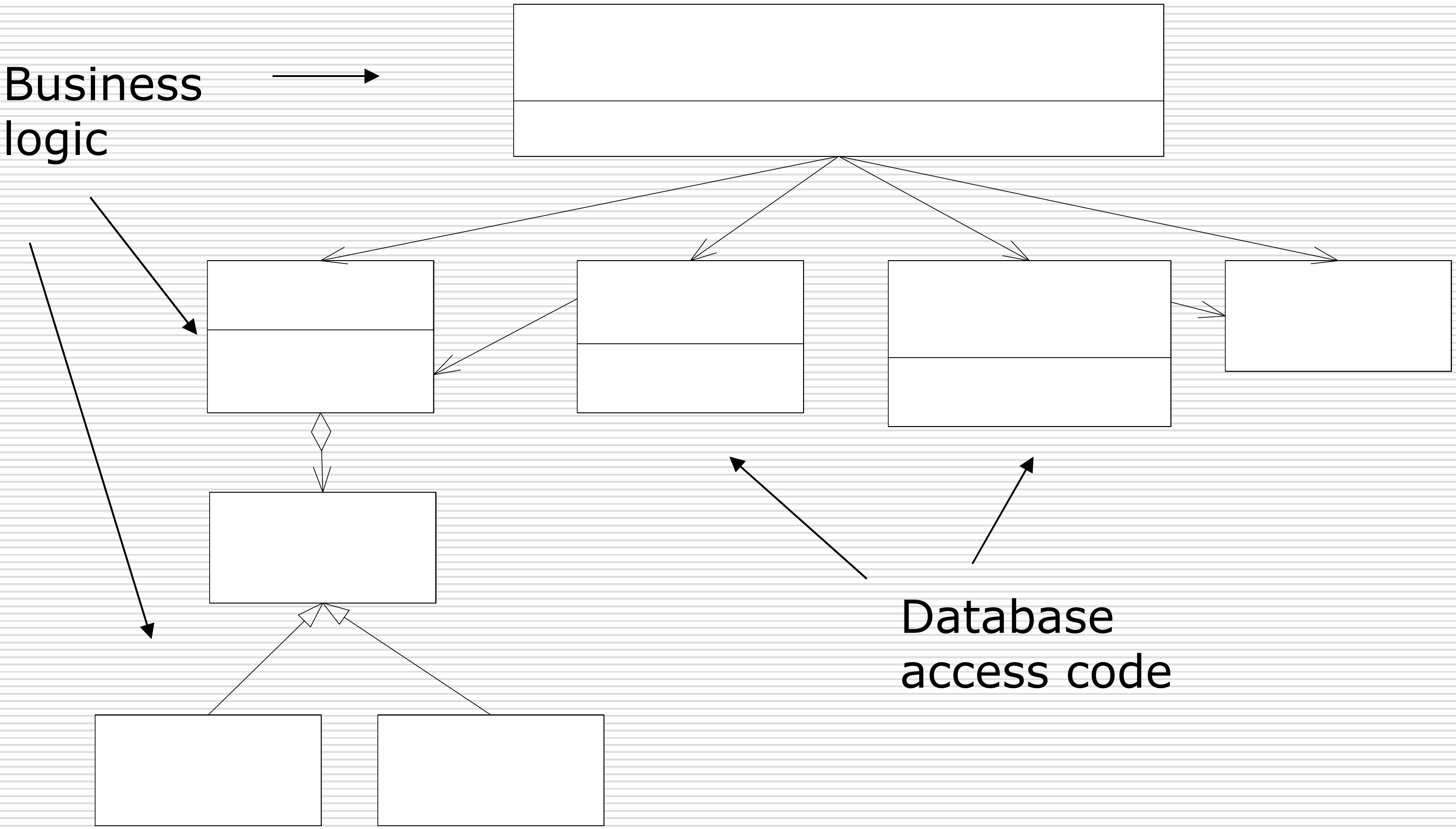
Agenda

- Overview of POJOs and lightweight frameworks
- The strengths and weaknesses of EJBs
- Developing applications with POJOs
- Example of a POJO design
- Where does EJB 3 fit in?
- Migrating to POJOs

POJO design



Banking domain model



Benefits of a domain model

- Easier to understand and maintain
  - More modular
  - Some classes mirror the real world
- Easier to test
  - Because of the modularity
- Easier to extend
  - e.g. Strategy and Template method design patterns

Drawbacks of a domain model

- Need OO design skills
- Requires an object/relational mapping framework
- Not suitable for some applications
  - Bulk updates
  - Functions that are best performed by the database

Implement using POJOs

- Use the features of the Java language
  - Inheritance
  - Recursive calls
  - Fine-grained objects
- Things that EJB prevented you from using

Walk through the domain model code

- Look at classes
- Run some tests

Benefits of POJOs

- Easier development
  - Less restrictions
  - None of the complexity of EJBs
  - Develop and test without worrying about the database
- Faster development
  - No deployment
- Improved portability
  - Not tied to a particular framework

Use an object/relational mapping framework

- Map the domain model to the database schema
- Hibernate
  - Very popular open-source project
- JDO
  - Standard from Sun – JSR 12 and JSR 243
  - Multiple implementations
  - Commercial:
    - Kodo JDO
  - Open-source
    - Versant
    - JPOX

ORM framework features

- Declarative mapping
- CRUD API
- Query language
- Transaction management
- Lazy and eager loading
- Caching
- Detached objects

ORM benefits

- Improved productivity
  - High-level object-oriented API
  - 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

ORM Drawbacks

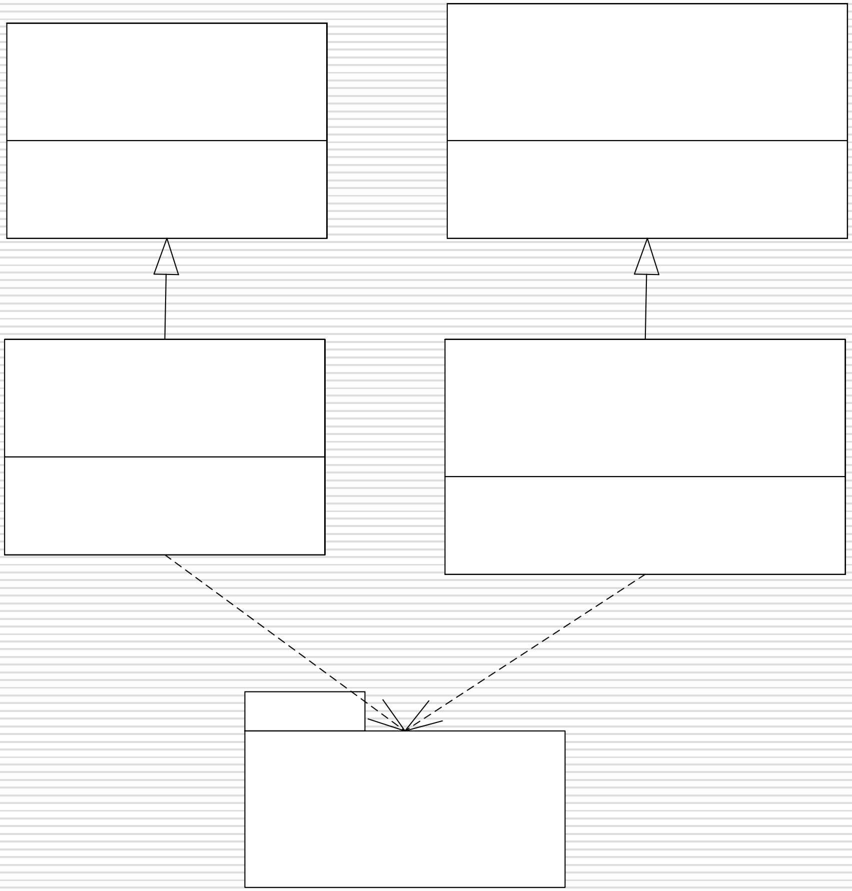
- Less control over the SQL
  - But sometimes you need to use database specific features
- Object/relational mapping limitations
  - Weird object models
  - Weird database schemas

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
  - Lots of updates
  - Relational style

Encapsulating calls with repositories

- Insulates the rest of the application from the ORM framework
- Enables the domain model to be tested without the database
- Makes switching persistence mechanisms relatively easy

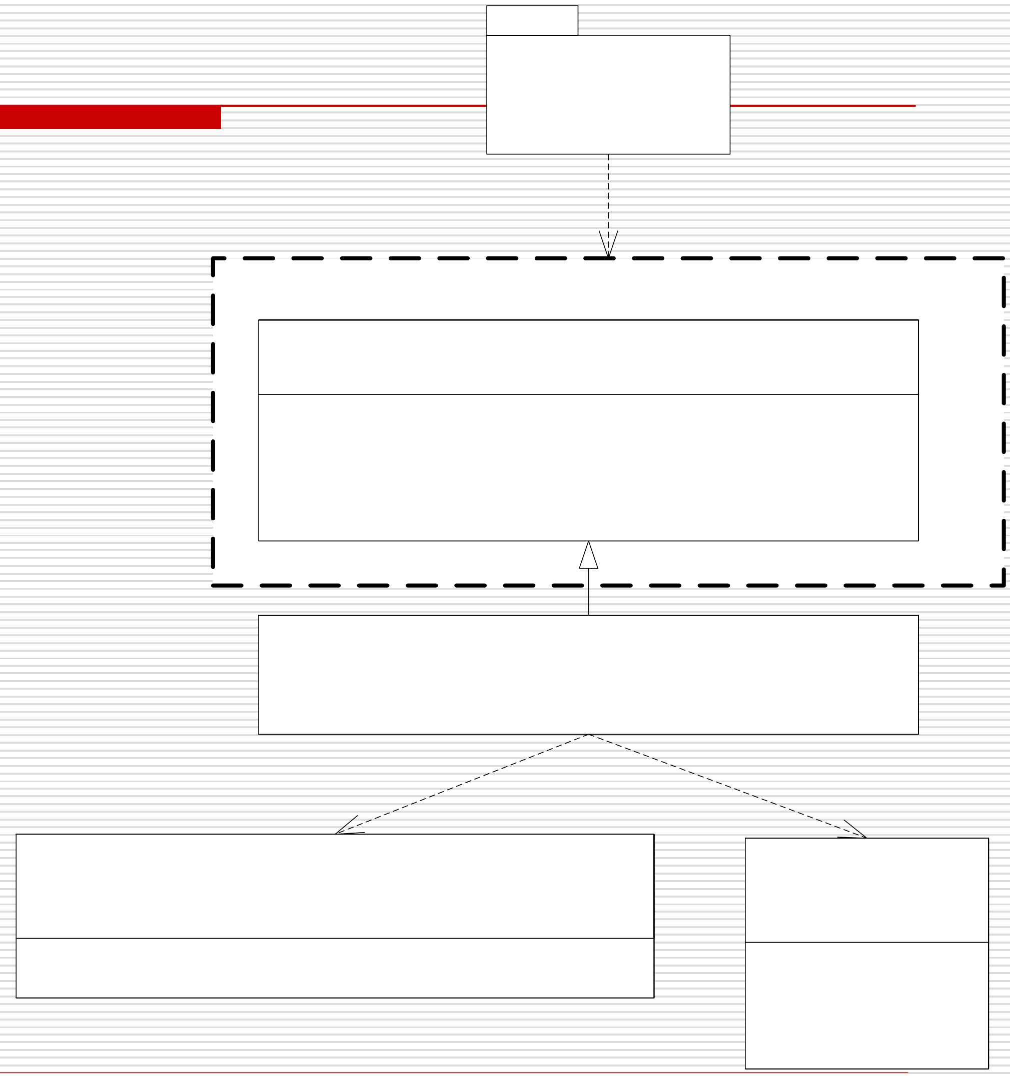


Look at the example code

- Object/relational mapping
- Repositories
- Run some tests

POJO facade

- Similar to an (EJB) Session Facade
- Handles requests from the presentation tier
- Gathers data that the presentation tier requires
- Delegates to the domain model
- Using Spring for Transaction management
- Returns detached objects instead of DTOs
- Using dependency injection instead of JNDI lookups



Look at the example facade code

- AccountManagementFacade
- AccountManagementFacadeImpl
- Run some tests

Managing transactions with Spring

- Declarative transactions is one of main motivations for using EJBs
  - Simplifies the code
  - Less error-prone
- POJOs need an equivalent mechanism

  $\Rightarrow$ Spring framework

What is the Spring framework?

- It's a framework that makes it easier to develop J2EE application
- Lots of features
  - Lightweight container
  - ORM utility classes such as HibernateTemplate
  - …
  - MVC-based web framework
- And, declarative transaction management:
  - Write a small amount of XML
  - Supports Java 5 annotations also

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 and initialize the objects:

```
<bean name="AccountManagementFacade"
            class="AccountManagementFacadeImpl">
    <constructor-arg ref="AccountRepository"/>
    <constructor-arg ref="MoneyTransferService"/>
</bean>
```

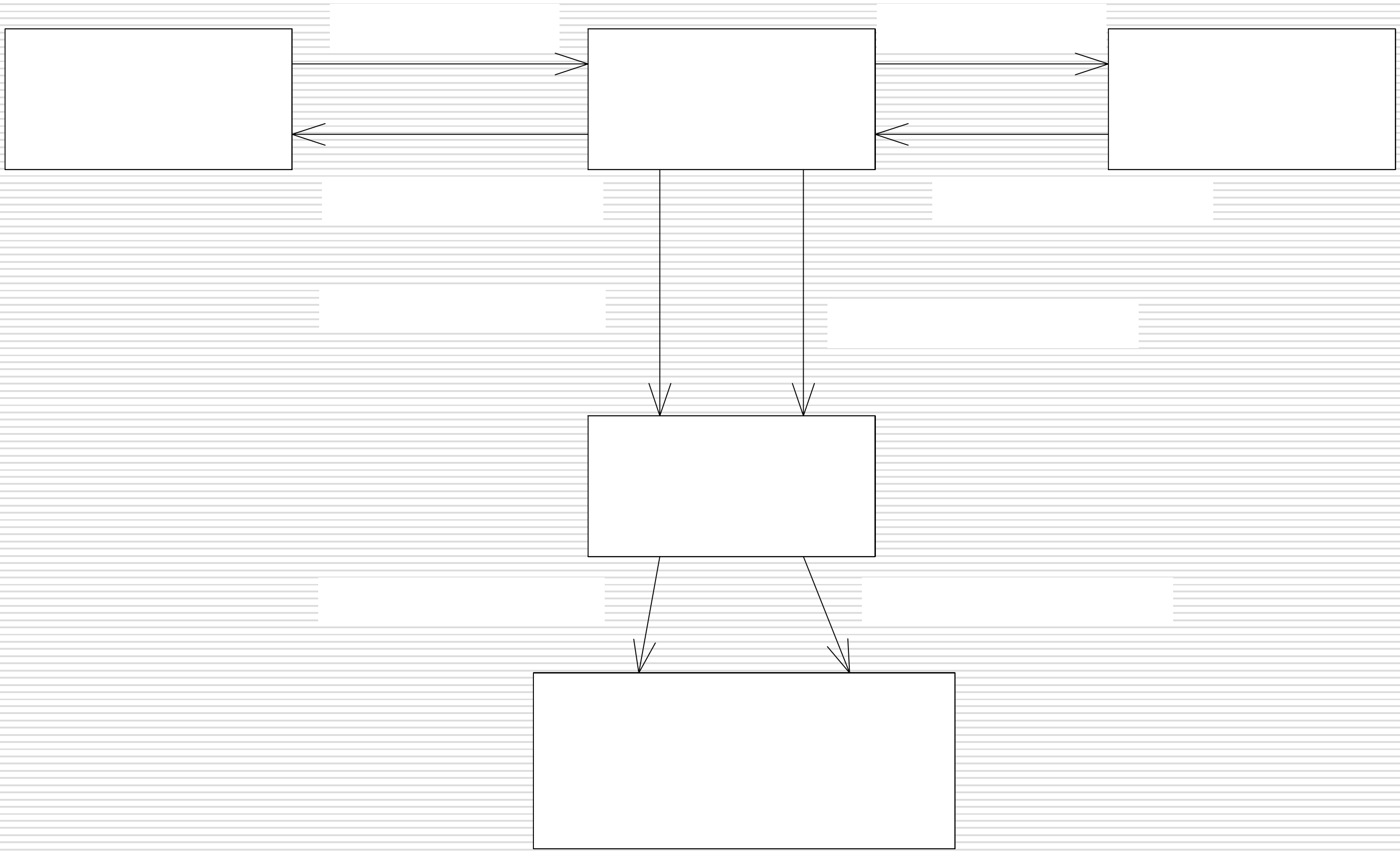
- Application calls:

```
beanFactory.getBean("AccountManagementFacade",
 AccountManagementFacade.class)
```

Spring AOP proxies

- Spring's lightweight container can do more than simply instantiate objects
- It can wrap an object with a proxy a.k.a interceptor
- Proxy masquerades as the original object
- Proxy executes arbitrary code before and after method call

Spring TransactionInterceptor



1. call

Spring PlatformTransactionManager

- Used by the TransactionInterceptor
- Encapsulates the transaction management APIs
- Multiple implementations:
  - JtaTransactionManager
    - JTA/UserTransaction
  - DataSourceTransactionManager
    - Connection.commit()/rollback()
  - HibernateTransactionManager
    - Session.getTransaction()/ Transaction.commit()/rollback()
  - JdoTransactionManager
    - Transaction.begin()/commit()/rollback()

Look at bean definitions source code

- TransferFacade
- TransactionInterceptor
- TransactionManager
- BeanNameAutoProxyCreator

Spring AOP

- AOP = Declarative mechanism for changing the behavior of an application
- Spring AOP is less powerful than other AOP solutions such AspectJ
- Much easier to use
- Doesn't require its own compiler
- Comes with a library of aspects for building enterprise Java applications
  - Managing transactions
  - Managing Hibernate and JDO
- You can also write your own

Using a POJO facade

- ✓ Encapsulate the business logic with a POJO facade
- ✓ Using Spring for declarative transactions
- ➢ Return detached objects instead of DTOs
- ☐ Use dependency injection to access resources and components

Replace DTOs with detached objects

- Developing DTOs is one of the more tedious aspects of EJB development
- Use detached objects instead
  - Instead of copying from domain object into a DTO
  - Return the domain object
- Hibernate
  - Objects automatically detached
  - Just load them
- JDO
  - Explicitly call to JDO API to detach them
- Tricky part:
  - Ensuring that enough of the object graph has been detached

Configuring applications with dependency injection

- Avoid
  - JNDI lookups
  - Explicit instantiation
- Instead, pass dependencies
  - Constructor arguments
  - Calling setters
- Benefits
  - Loosely coupled applications
  - Easier testing

Look at the example facade code

- AccountManagementFacade
- AccountManagementFacadeImpl
- Run some tests

Deployment options

- Deploy as a web application
- Jetty/Tomcat
- JBoss/WAS/WLS is only required if
  - JMS
  - JTA
  - Some app. server specific feature
  - …

Benefits of a POJO facade

- Faster and easier development
- Potentially eliminates need for EJB container

Drawbacks of a POJO facade

- Compared to EJB
  - No support for transactions initiated by a remote client
  - No equivalent to MDBs
  - Non-standard security, e.g. ACEGI security
  - Client must be able to get facade from container
- Detaching objects is potentially fragile
- Lack of encapsulation of domain model

POJO design - summary

- Yes, you still must write some XML but its simpler
- Less code
  - No DTOs
  - No JNDI lookup code
  - No low-level database code
- Easier to test
  - Outside of container
  - Loosely coupled code
- Able to use object-oriented design

Agenda

- Overview of POJOs and lightweight frameworks
- The strengths and weaknesses of EJBs
- Developing applications with POJOs
- Example of a POJO design
- Where does EJB 3 fit in?
- Migrating to POJOs

What about EJB 3 - good news

- Much better than EJB 2
- EJB 3 beans are POJOs
- Simplified configuration (using annotations)
- Improved persistence API
- EJB 3 entity beans support J2EE and J2SE
- Standardized O/R mapping
- Entity beans can be detached

EJB 3 - bad news

- O/R mapping weaker than Hibernate/JDO
- Session beans and message-driven beans are server-side components
- Limited form of dependency injection
- EJB 3 is an ease-of-use veneer on top of an application server

EJB 3 - conclusion

- In its current state you will most likely require vendor-specific extensions
- Carefully consider whether it is worth using
- Be skeptical
- No need to rejoin the cult

Agenda

- Overview of POJOs and lightweight frameworks
- The strengths and weaknesses of EJBs
- Developing applications with POJOs
- Example of a POJO design
- Where does EJB 3 fit in?
- Migrating to POJOs

Migrating an existing application

- Write new code using Spring
- Migrate existing code incrementally:
  - You cannot rewrite tens of session beans and DAOs overnight
- But sometimes new code must call old code
- Moreover, you will want to migrate old code:
  - You will hate working on it

Stateless Session bean ⇒ POJO façade

- Component interface ⇒ POJO interface
- Bean class ⇒ POJO that implements interface
- EJB CMT ⇒ Spring-managed transaction
- EJB security ⇒ ACEGI security
- JNDI lookup of EJB's home ⇒ BeanFactory.getBean()

Entity beans $\Rightarrow$ Hibernate

- Encapsulate code that calls home interface within repository
- Entity bean class $\Rightarrow$ concrete POJO
- Abstract accessors $\Rightarrow$ concrete accessors+fields
- ejbCreate() $\Rightarrow$ constructor
- Add code to manage bidirectional relationships
- Finders $\Rightarrow$ named queries + repository method
- Define O/R mapping
- …

Handling connections

- Handling database connections when DAO/Repository used by old and new code
- Original DAO code:
  - DataSource.getConnection()
  - Connection.close()
  - Connection(s) associated with JTA transaction
- To ensure one JDBC connection per transaction use
  - DataSourceUtils.getConnection()
  - DataSourceUtils.releaseConnection()

Conclusion

- Use EJBs for:
  - Distributed transactions from remote clients
  - Message-driven beans
- For everything else use:
  - POJOs, Spring
- And, when you can:
  - Use an object-oriented design
  - Object/relational mapping framework
- Adoption:
  - Write new code using POJOs/Spring/…
  - Incrementally migrate existing code

For more information

- Email: cer@acm.org
- Blog:
  - http://chris-richardson.blog-city.com
- My book (Oct05):
  - POJOs in Action