Javaオープンソースによる J2EE開発

伊藤忠テクノサイエンス(株)
アドバンスト・テクノロジー技術部
浜松 耕太

はじめに

- 本セミナーでは
  - Javaオープンソースの J2EEランタイムに注目します
  - 妥当なモデルとして軽量コンテナの適用を考えます
  - 軽量コンテナの代表として Springを扱います
  - Springを用いて以下の基礎技術の理解に注力します
    - Dependency Injection
    - AOPトランザクション
  - Java、J2EE、設計の基礎知識を必要とします

Agenda

- Javaオープンソースの背景
- POJOベース開発の潮流
- テストしやすい設計とは
- Dependency Injection
- AOPトランザクション

J2EEコンテナの広がり

- J2EE Application Serverの例
  - JBoss (※)
  - ObjectWeb JOnAS (※)
  - Apache Geronimo
  - Caucho Resin

(※)　J2EE1.4 CTS (Compatibility Test Suit) Passed

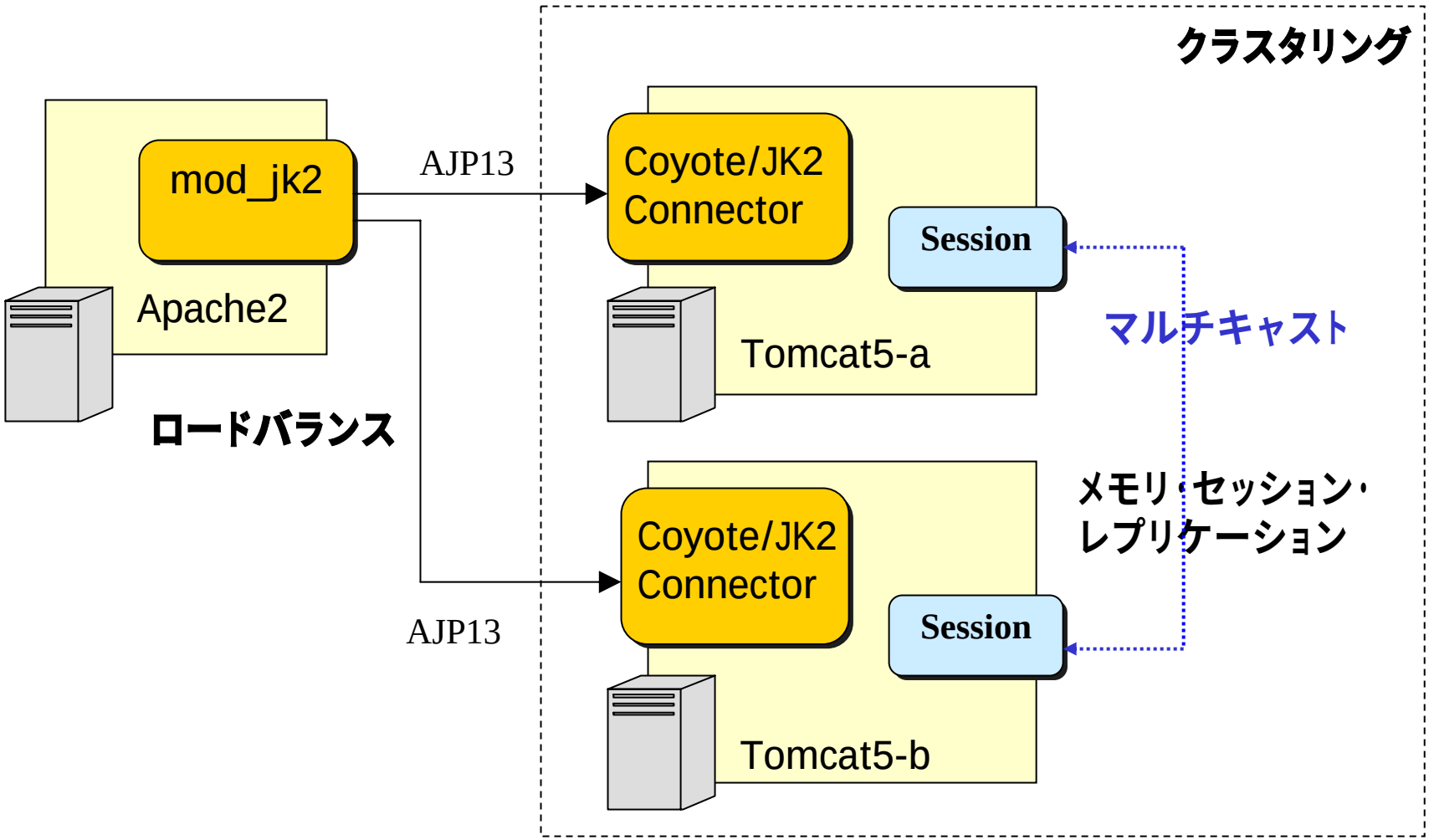
J2EEとは何だったのか

- **Applicationサーバのための仕様**
  - Application**サーバ間のポータビリティの確保**
  - Java Enterprise API**のスナップショットと利用規定**
  - **配備作業の導入**
- **Javaプラットフォームのサーバ適用の宣言**
  - **ブランド戦略**

ブロックと糊

- J2EEパターンの必要性
- ビルディング・ブロックの発展
  - 特定の仕切り内でイテレーティブに開発されるコンポーネントやフレームが発展
  - 開発者は要件にあわせて選択できる反面、何を選択してよいかが難しくなる
- 統合技術（ブロックの糊）の必要性
  - もっと、管理・変更がやさしくならないの？

Tomcatの成長（フェイルオーバー）



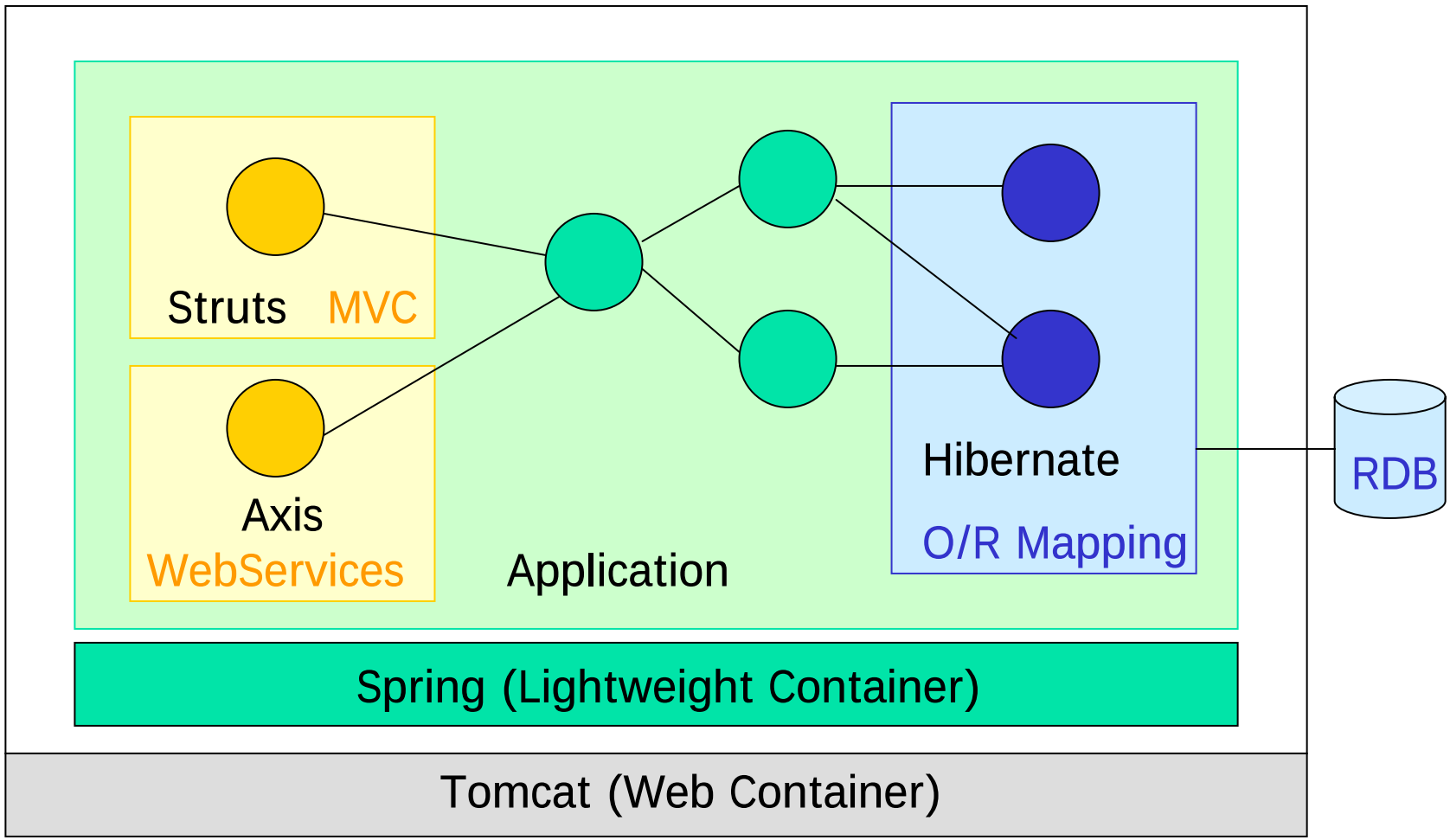
性能向上の期待感

- HWの性能改善とコストダウンに便乗
  - スループット・コンピューティング
  - CMT (チップ・マルチスレッディング)
    - 1つのプロセッサが複数のスレッドを同時に実行
    - Sun：Niagara, Rock
- オープンソースのメリット
  - CPU単位の商用ライセンスの問題を心配する必要がない

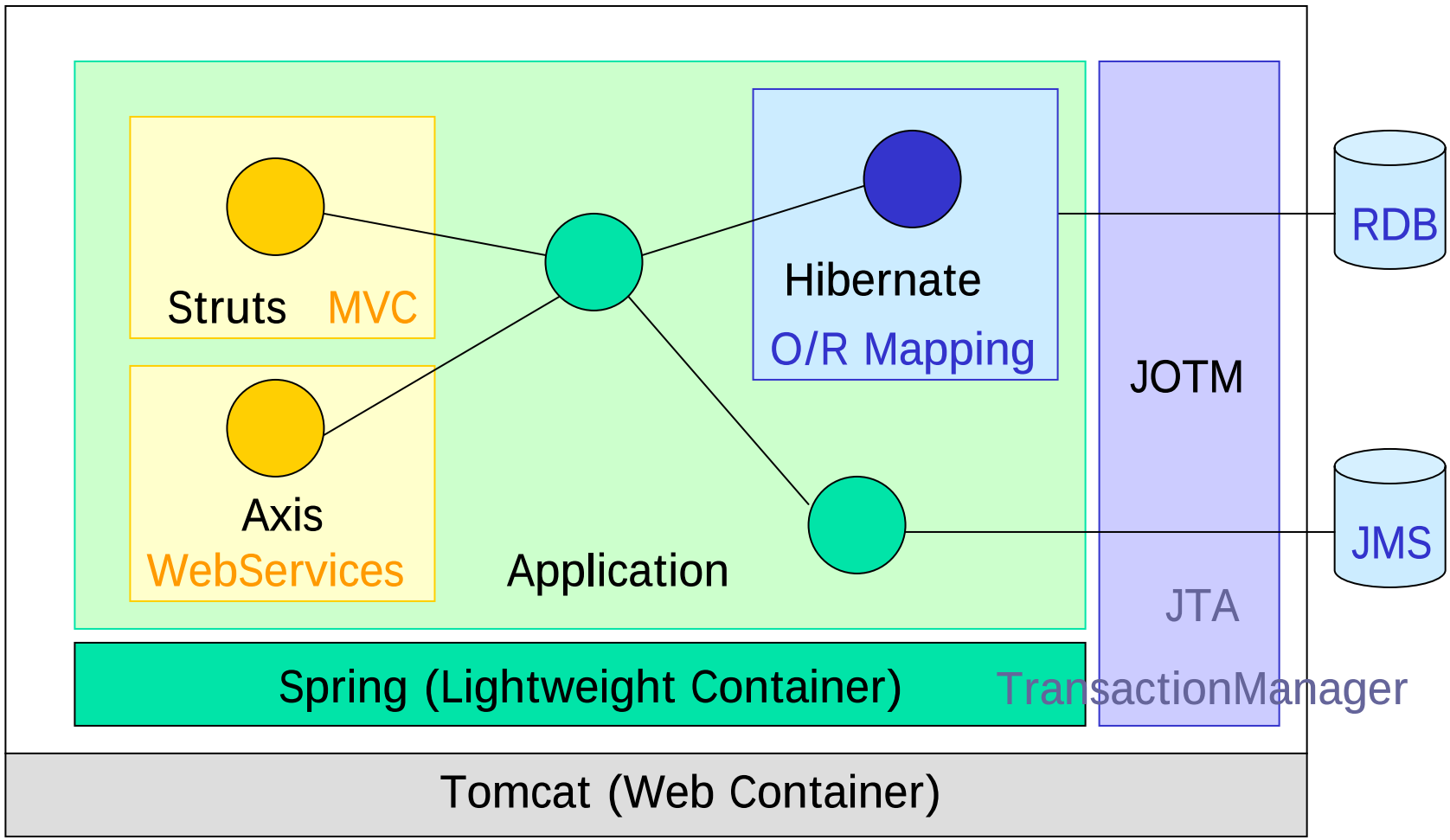
軽量コンテナの台頭

- Spring
  - 代表的な軽量コンテナ
  - Rod Johnson氏が中心となって開発と啓蒙が進められている
- Springの主な機能
  - DIコンテナ with AOP
  - トランザクション管理のサポート
  - プレゼンテーション層のサポート
  - DAOプログラミングのコード量を減らすライブラリ

ひとつのモデル



グローバル・トランザクション



Agenda

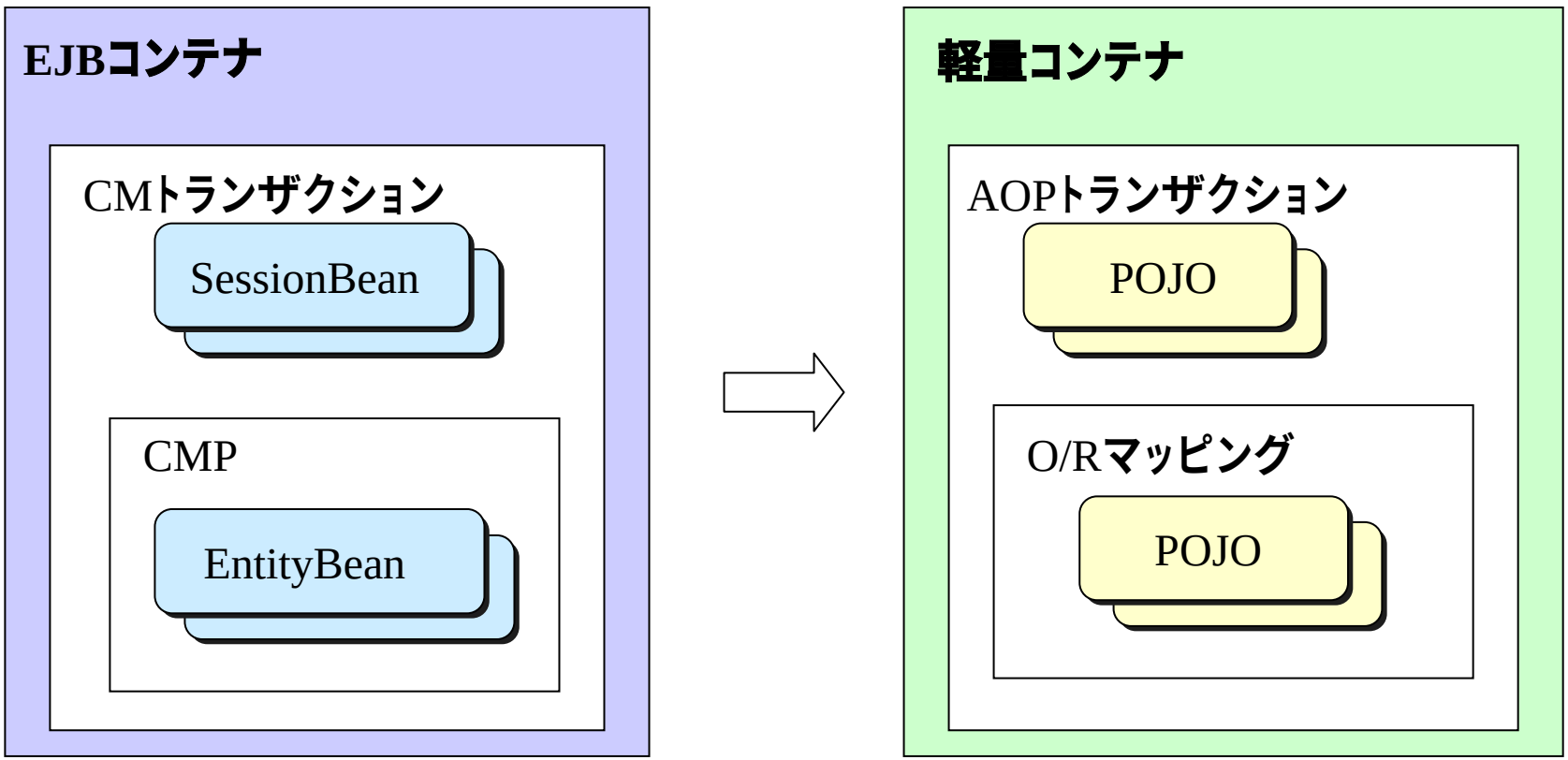
- Javaオープンソースの背景
- **POJOベース開発の潮流**
- テストしやすい設計とは
- Dependency Injection
- AOPトランザクション

POJOとは

- Plain Old Java Object
  - 普通のJavaオブジェクト
  - Not EJB!!
- POJOベース開発の目差すところ
  - POJOでEJBが担うべき妥当な機能を実現すること
  - ビジネスロジックとシステムサービスを分離すること

EJBからPOJOへ

- 軽量コンテナで実現するPOJOへの回帰

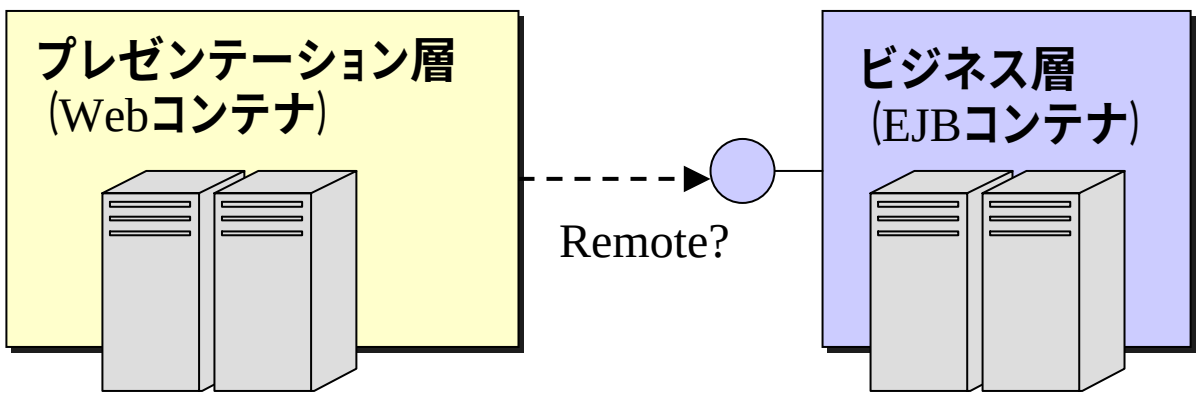


POJO回帰の理由（1）

- EJBの複雑さに対するアンチテーゼ
  - 開発しやすい？
  - コントロール（管理・運用）しやすい？
  - そもそもEJBはWebアプリケーション開発のために設計されたものではない

POJO回帰の理由（2）

- EJBの分散層に対する疑問の目
  - ローカル･インターフェイスの導入(EJB2.0)
  - ローカルなら高価なEJBコンテナのクラスタリング機能は必要ない
  - ビジネス･コンポーネントの再利用性の低さも要因



POJO回帰の理由（3）

- テスト駆動型開発(TDD)の普及
  - TDDに於いてユニットテストのしやすい設計がなにより好まれる
  - 品質向上
  - 開発効率の向上
- POJOはユニットテストが易しくなる
  - EJBはユニットテストが難しい

Agenda

- Javaオープンソースの背景
- POJOベース開発の潮流
- テストしやすい設計とは
- Dependency Injection
- AOPトランザクション

ユニットテストの戦略

- ユニットテストがしやすい設計とは
  - 依存している環境や関連するクラスから分離してテストが実行できる設計
- どのような方法で分離し、どのような方法で統合するのが効果的なのか？
  - 分離は基本的にJ2EEパターンの層構造に従う
  - 問題は統合

設計の考察 (1)

- インターフェイスによる実装の分離

```
package app;
public class Foo {
   private BarDao barDao;
   ....
}

package dao;
public interface BarDao {
   public Bar getBar(String id);
}
```

設計の考察 (2-1)

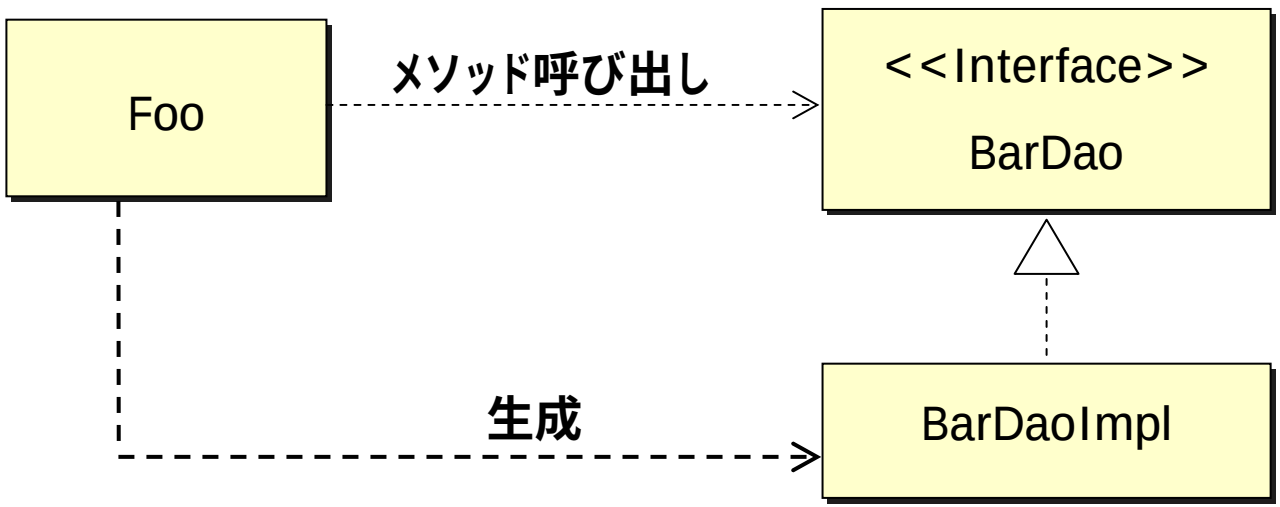
- 問題は、インターフェイスの実装クラスをどのようにつなげるか

```
package app;
public class Foo {
    private BarDao barDao;
    public Foo() {
        barDao = new BarDaoImpl(...);
    }
}
```

実装クラスを交換するとき、ソースコードの変更が必要になる

設計の考察 (2-2)

- 依存関係図



設計の考察 (3-1)

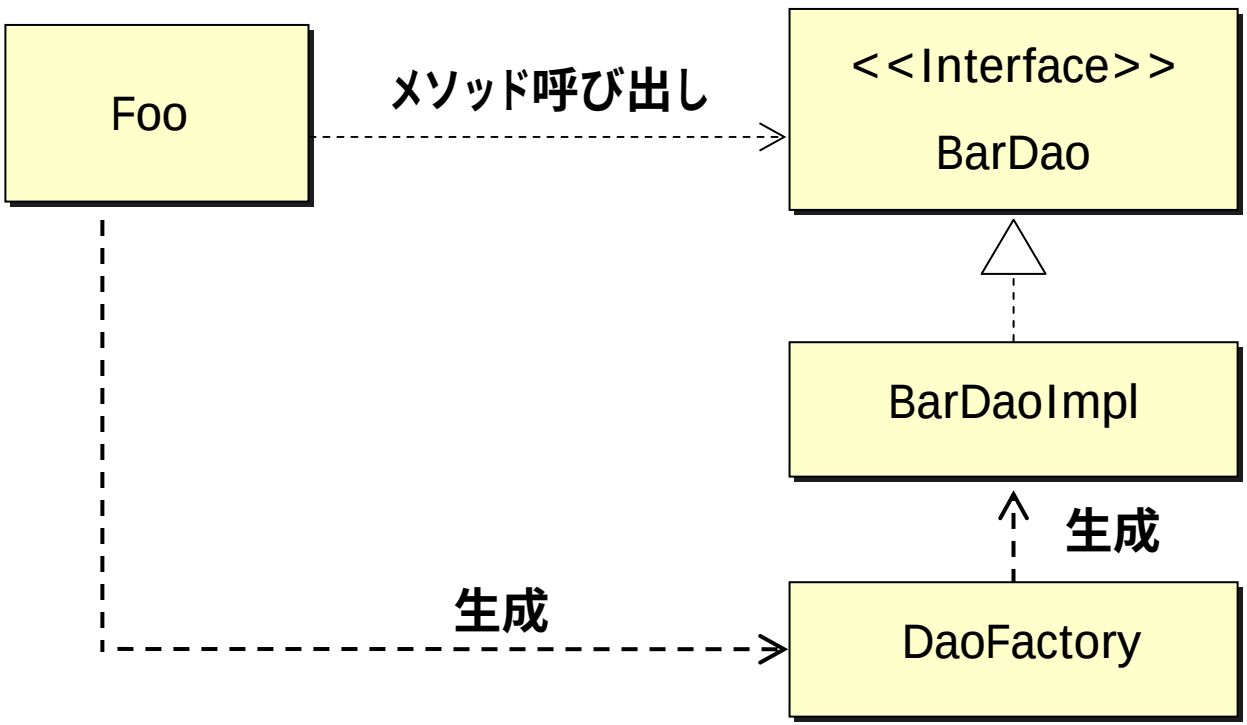
- **Factoryクラス (サービスロケータ) の利用**

```
package app;
    public class Foo {
    private BarDao barDao;
    public Foo() {
      DaoFactory factory = new DaoFactory();
      barDao = (BarDao) factory.getDao(...);
    }
   .....
}
```

設定・管理はFactoryクラスの実装依存。コードも読みにくい

設計の考察 (3-2)

- 依存関係図



設計の考察 (4-1)

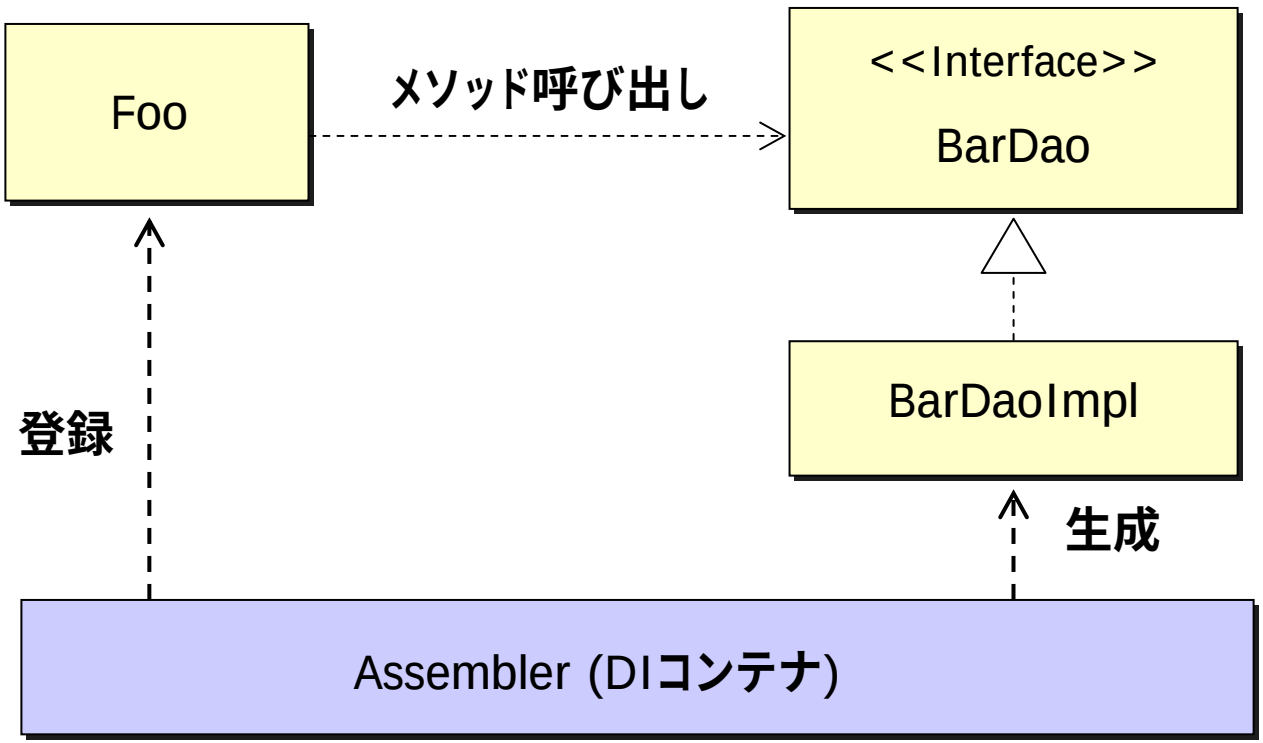
- IoC(Inversion of Control)パターン

```
package app;
public class Foo {
    private BarDao barDao;
    public void setBarDao(BarDao barDao) {
        this.barDao = barDao;
    }
    ....
}
```

別の層のクラスのインスタンスは、内部で生成しない。外部から設定してもらう

設計の考察 (4-2)

- 依存関係図



IoCパターンの恩恵

- ユニットテストが易しくなる
  - モックオブジェクトを用いることで環境やリソースに依存するオブジェクトから分離してユニットテストができる

```
public class FooTest  extends junit.framework.TestCase {
    public void testDoSomething() {
        Foo foo = new Foo();
        foo.setBarDao(new BarDaoMock());
        foo.doSomething();
    }
}
```

モックオブジェクト

- モックオブジェクトとは
  - テスト対象のコードがやり取りするオブジェクトの代理として用意するオブジェクト

- expectation検証とは
  - テスト対象コードが、モックオブジェクトのコードを期待する通りに呼び出しているかどうかを確認するテスト

モックオブジェクト･ジェネレータ

- **モックオブジェクト･ジェネレータとは**
  - **インターフェイスから、実行時にモックオブジェクトを透過的に生成するツール**（DynamicProxy**を利用**）
  - **モックオブジェクトのメソッドに期待される返り値や例外発生をコントロールすることが可能**
  - expectation**検証をサポート**
- **ツール**
  - EasyMock　(extention1.1)
  - DynaMock

EasyMockの例

- **EasyMockControlのコード例**

インターフェイス

```
MockControl control = MockControl.createControl(BarDao.class);
BarDao mock = (BarDao) control.getMock();
Foo foo = new Foo();
foo.setBarDao(mock);

control.expectAndReturn(mock.getBar(...), new Bar(...));
control.replay();

foo.doSomething(...);
control.verify();
```

このメソッドでMockが返す値を設定

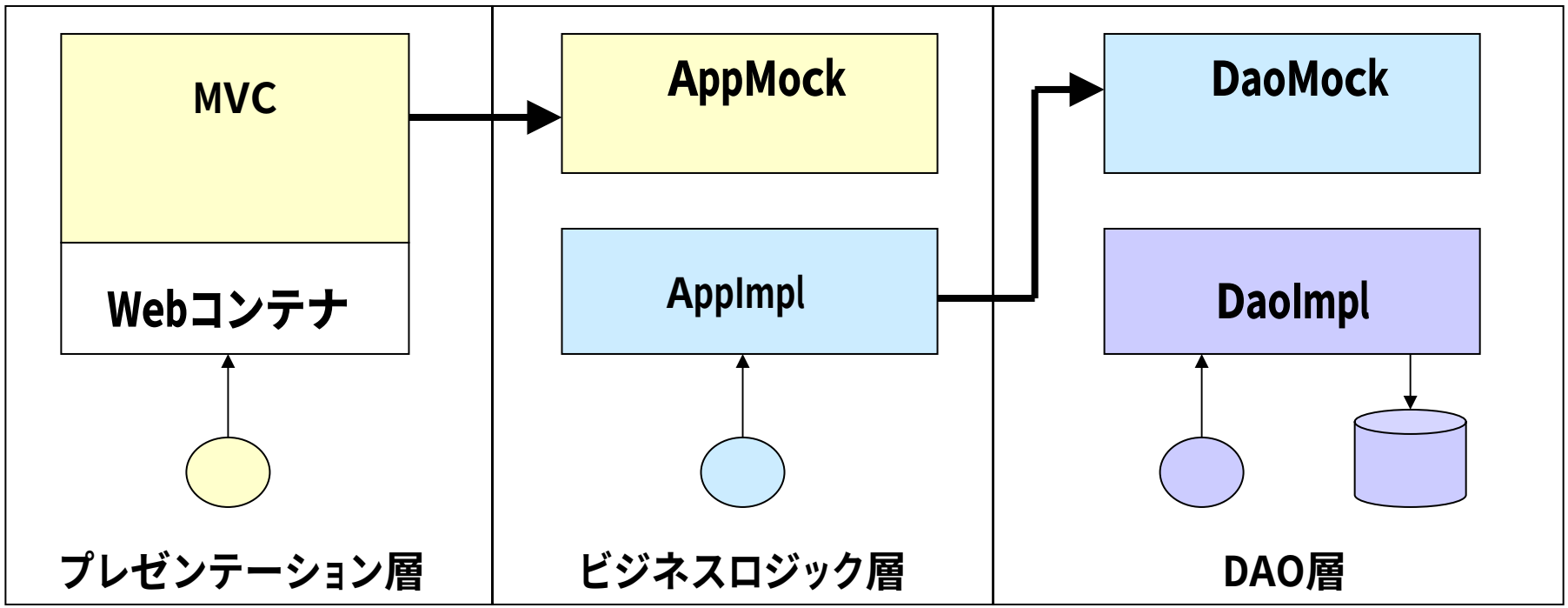
このメソッド内で「mockのgetBarがよばれることを期待する」ことを登録

IoCとテストの関係

- IoCパターンの適用
  - テスト対象のクラスに含まれる依存関係を分離してモックオブジェクトを利用することで、ユニットテスト及び並行開発がしやすくなる
- DIコンテナの適用
  - 設計上分離された依存関係をコンフィグレーション・レベルで容易に構築することが可能になる

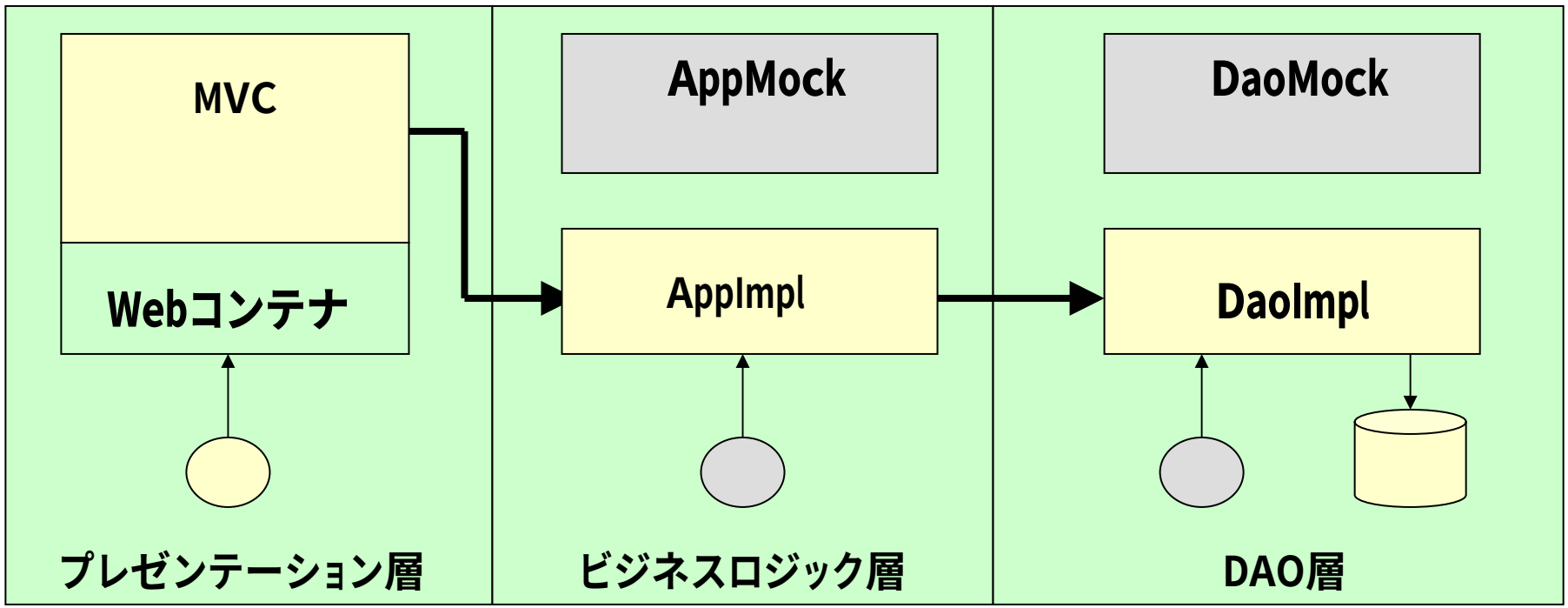
層構造で分離したTDDの例（1）

- モックオブジェクト・アプローチ



層構造で分離したTDDの例（2）

- DIコンテナによる統合

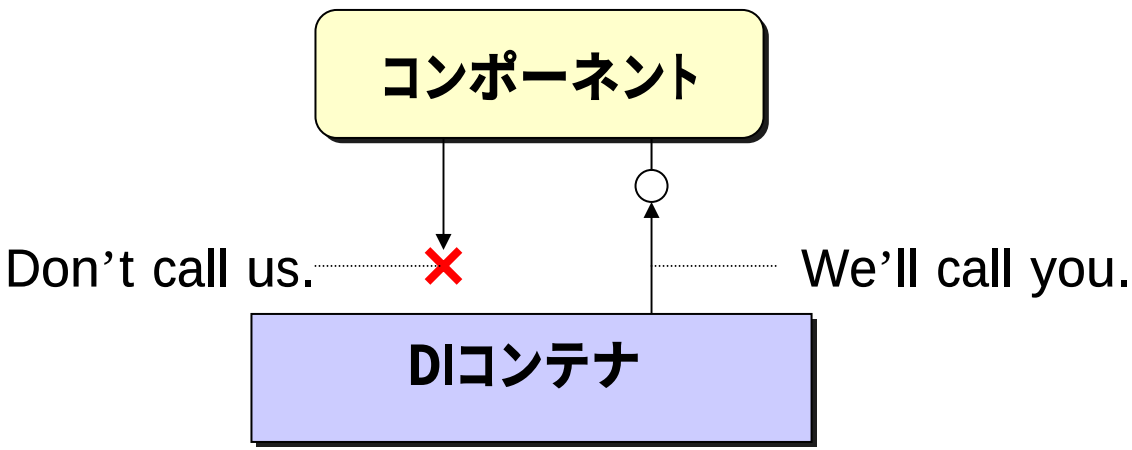


Agenda

- Javaオープンソースの背景
- POJOベース開発の潮流
- テストしやすい設計とは
- Dependency Injection
- AOPトランザクション

広義のIoC

- **設定と利用の分離**
  - **ハリウッドの原則**：Don’t call us. We’ll call you.
  - **コンポーネント間の依存関係をコンテナに任せる**



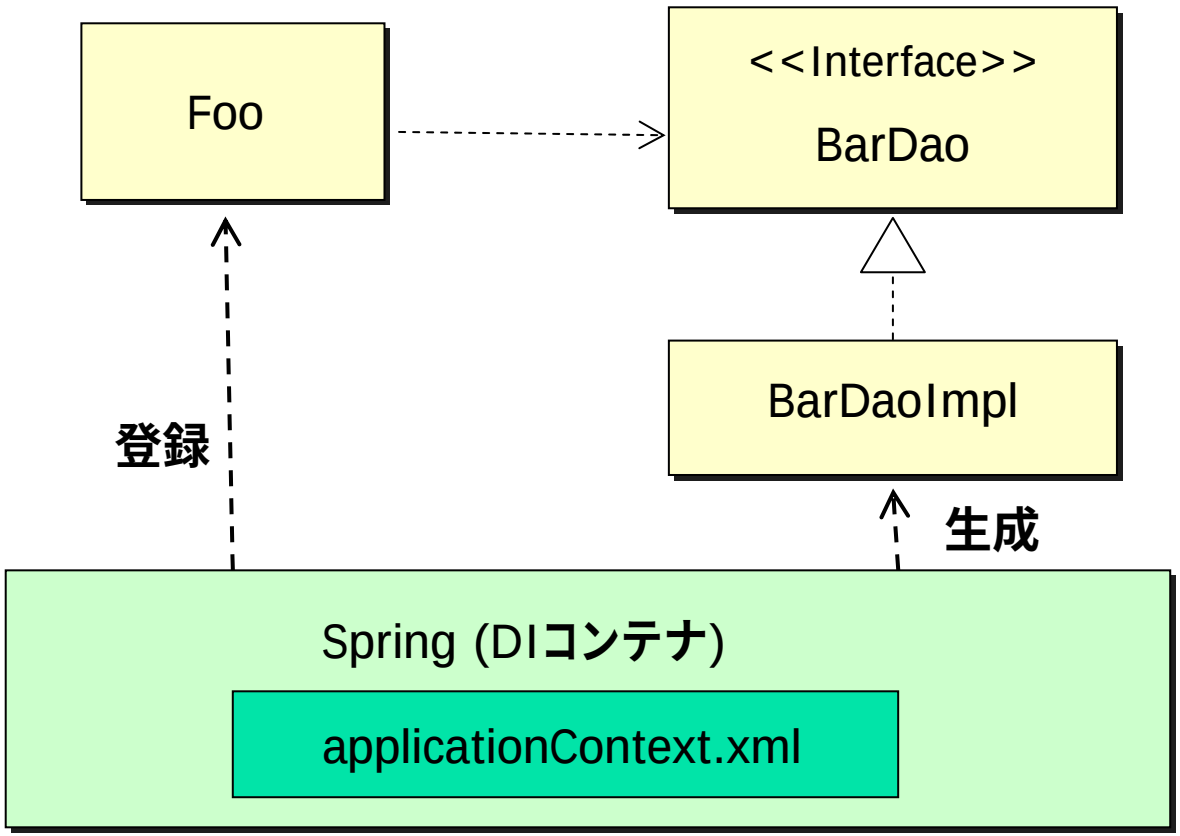
Dependency Injection

- **Dependency Injection(DI)とは**
  - **狭義のIoC**
  - **コンポーネントからコンテナ依存のAPIを排除**
  - Setter Injection
  - Constructor Injection (Immutable**クラスへの対応**)

- **DIコンテナとは**
  - DI**を実現するコンテナ**
  - **クラスの依存関係をコンフィグレーションから構築する**
  - **大雑把に云うと**POJO**のプラグイン･フレームワーク**

SpringのDIの設定法は？

- XMLで記述します



SpringのDI設定例 (1-1)

- **Setter Injection**
  - **オブジェクトの依存関係**

```
<beans>
    <bean id="foo" class="app.Foo" singleton="false">
        <property name="barDao">
            <ref bean="bar"/>
        </property>
    </bean>
    <bean id="bar" class="dao.BarDaoImpl" singleton="false">
            .......
    </bean>
</beans>
```

シングルトンも設定可能

SpringのDI設定例 (1-2)

- **Setter Injection**
  - **初期値やデフォルト値の設定も可能**

```
<beans>
    <bean id="hoge" class="app.Hoge">
        <property name="stringProp">
            <value>Hello</value>
        </property>
        <property name="intProp">
            <value>1</value>
        </property>
    </bean>
</beans>
```

PropertiesやCollectionによる設定も可能

```
<props>
  <prop key="key">value</prop>
    ....
</props>
```

SpringのDI設定例 (2-1)

- **Constructor Injection**
  - **パラメータつきのコンストラクタ**

```
public Foo(BarDao barDao) {…}

<beans>
    <bean id=“foo” class=“app.Foo” singleton=“false”>
      <constructor-arg><ref bean=“bar”/></constructor-arg>
    </bean>
    <bean id=“bar” class=“dao.BarDaoImpl” singleton=“false”>
        ‘‘‘‘
</beans>
```

SpringのDI設定例 (2-2)

- **Constractor Injection**
  - Static Factory メソッドも可能

```
public class Hoge {
    public static Hoge getInstance(String hoge) {...}
}

<beans>
    <bean id=“hoge” class=“app.Hoge” singleton=“true”
      factory-method="getInstance">
      <constructor-arg><value>Hello</value></constructor-arg>
    </bean>
</beans>
```

SpringのDI設定例 (2-3)

- **Constractor Injection**
  - Factoryクラスにも可能

```
public class HogeFactory {
    public Hoge createHoge(String hoge) {…}
}

<beans>
    <bean id=“hoge” class=“app.Hoge” singleton=“false”
      factory-bean=“app.HogeFactory"
      factory-method=“createHoge">
      <constructor-arg><value>Hello</value></constructor-arg>
    </bean>
</beans>
```

ApplicationContext

- **プログラムはDIコンテナからオブジェクトを取得するのか？**

```
ApplicationContext context = new
    FileSystemXmlApplicationContext("applicationContext.xml");

Foo foo = (Foo) context.getBean(“foo”);
BarDao barDao = foo.getBarDao();
```

ちょとした疑問

- 「先のメカニズムってJNDIと同じじゃないの？」
  - Tomcatのリソース設定

```
<Resource name="UserTransaction" auth="Container“
                    type="javax.transaction.UserTransaction"/>
<ResourceParams name="UserTransaction">
      <parameter>・・・・</parameter>
 </ResourceParams>
```

  - JNDIプログラム

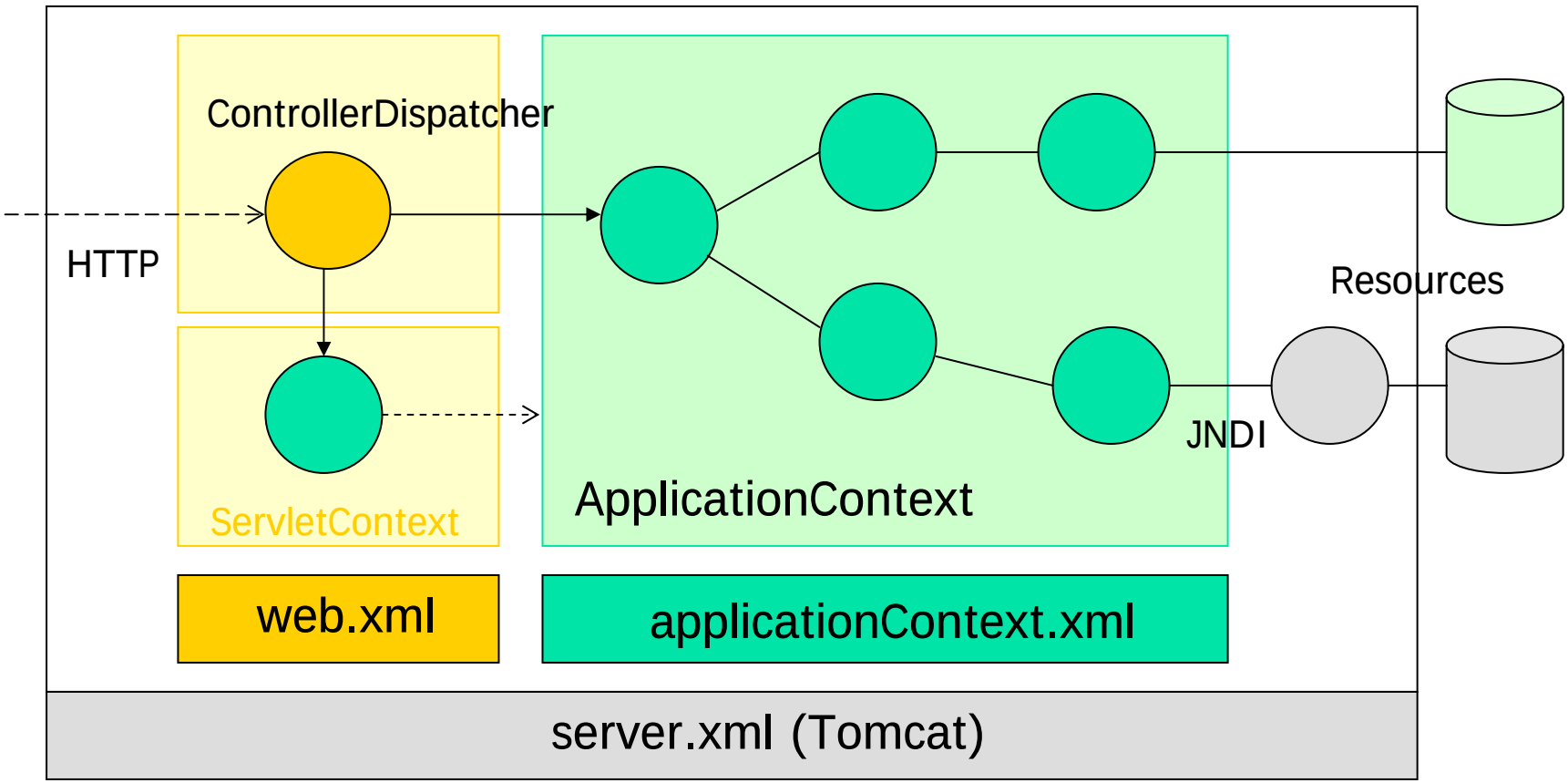
```
Context ctx = new InitialContext();
UserTransaction tx =
      (UserTransaction)ctx.lookup("java:comp/UserTransaction");
```

J2EE/JNDIとの違い

J2EE/JNDI	Spring/ApplicationContext
サーバが起動していないとプログラムは動かない	コンテナの起動と無関係。**ライブラリと同じように使える**
ユニットテストが難しい インコンテナ･アプローチ	**ユニットテストが簡単** モックオブジェクト･アプローチ
JNDI経由で取得できるのはリソースのみ Dependency Injectionが不可能	リソースは当然ながらPOJOもOK **Dependency Injectionが可能**
JNDIのコードがプログラムのいたるところに散らばる	ApplicationContextを取得するクラスは限定される
設定ファイルが散らばる	設定ファイルがまとまるので、変更･管理が楽になる
標準APIである	標準APIではない

Webアプリケーションの構造

- ApplicationContextの参照関係



Agenda

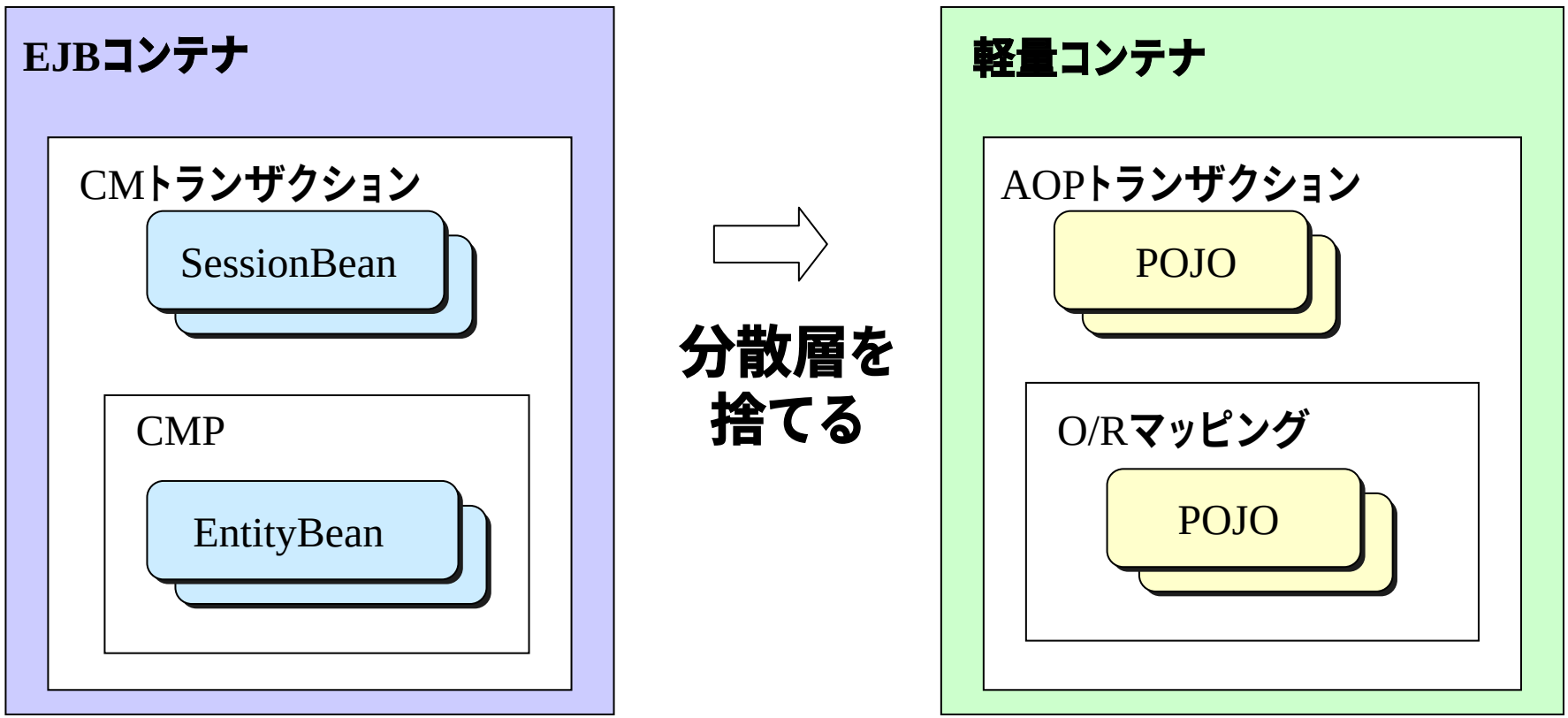
- Javaオープンソースの背景
- POJOベース開発の潮流
- テストしやすい設計とは
- Dependency Injection
- AOPトランザクション

AOP

- AOPとは
  - 複数のクラス内に散らばる共通機能を、再利用を目的としてモジュール(Aspect)化するプログラミング
  - 共通のルールブックを作成し、対象とする各ロジックに対してルールを適用するようなもの
  - Aspectを作成し、joinpointに対してadviceをweaveする

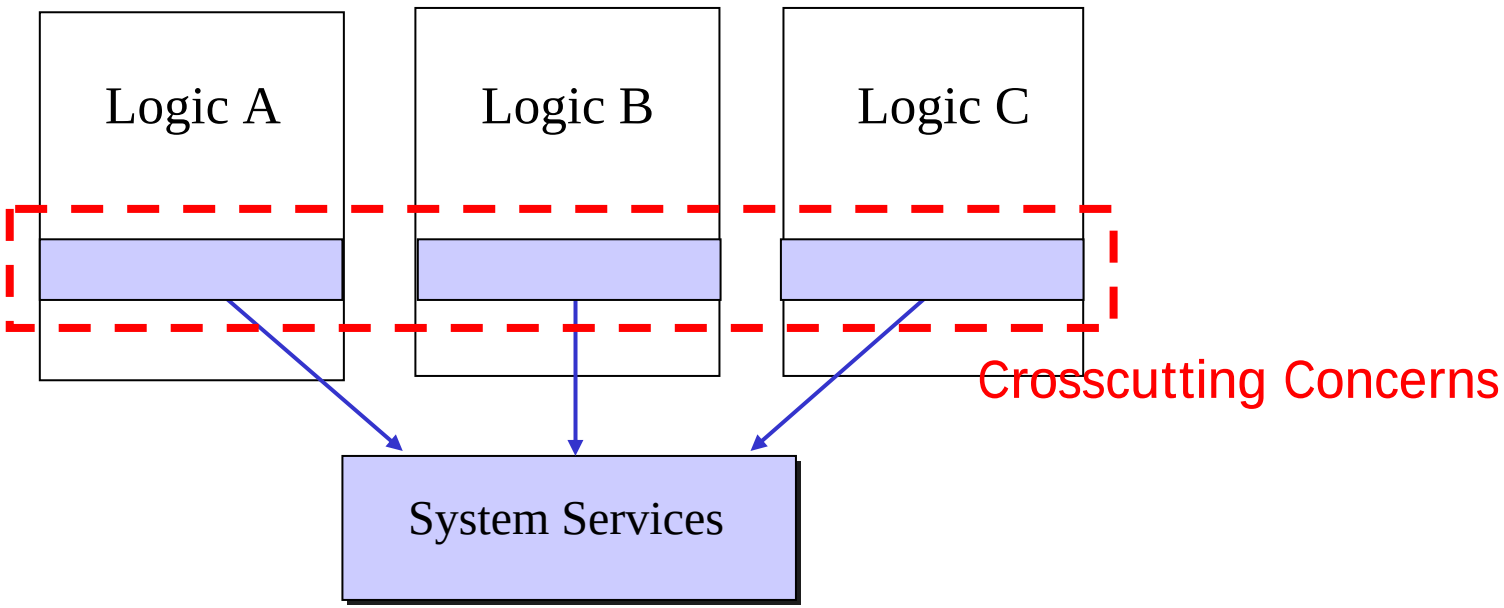
EJBからPOJOへ (再)

- 軽量コンテナで実現するPOJOへの回帰



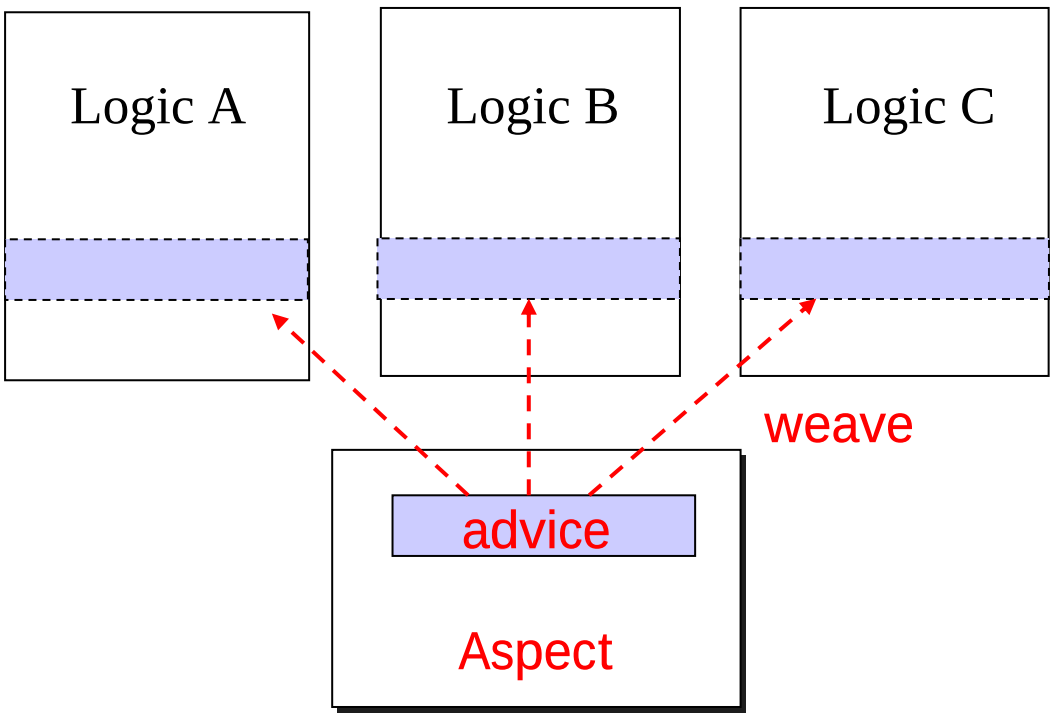
システムサービス依存の分離

- Crosscutting ConcernsのAspect化



POJOへのシステムサービス提供

- システムサービス機能の注入(weave)

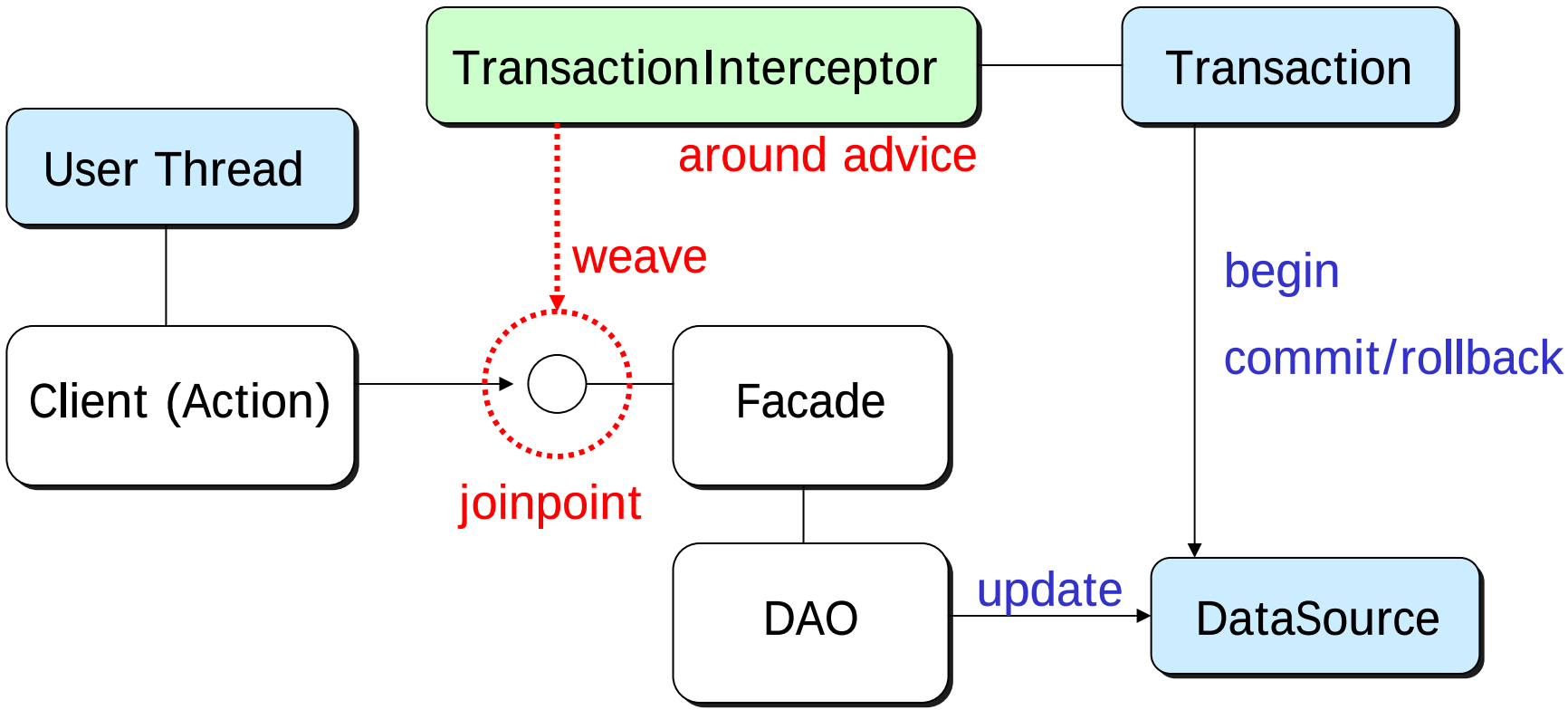


Springのトランザクション

- AOPトランザクション
- DIコンテナ+AOPトランザクションのメリット
  - 静的なコンパイルやクラスローダの仕掛けを必要としない（DynamicProxyを利用）
  - トランザクション境界を一元管理

SpringのAOPトランザクション

- **AOPトランザクションの構造**



コードリーディング

- **TransactionInterceptor**

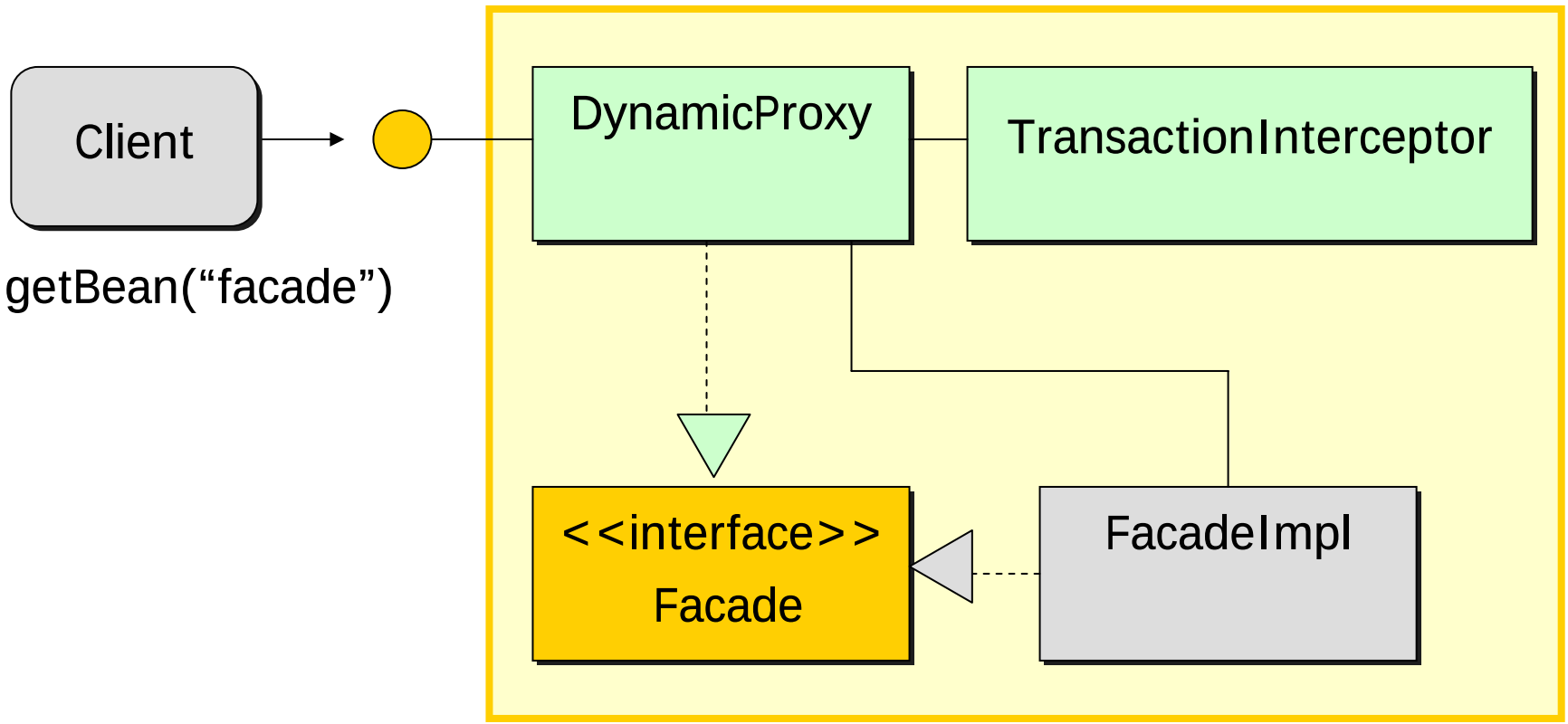
```
public Object invoke(MethodInvocation invocation) throws Throwable {
        Class targetClass = (invocation.getThis() != null) ? invocation.getThis().getClass() : null;
        TransactionInfo txInfo = createTransactionIfNecessary(invocation.getMethod(), targetClass);
        Object retVal = null;
        try {
                retVal = invocation.proceed();
        }  catch (Throwable ex) {
                doCloseTransactionAfterThrowing(txInfo, ex);
                throw ex;
        }
        finally {
                doFinally(txInfo);
        }
        doCommitTransactionAfterReturning(txInfo);
        return retVal;
}
```

AOPトランザクションの設定例

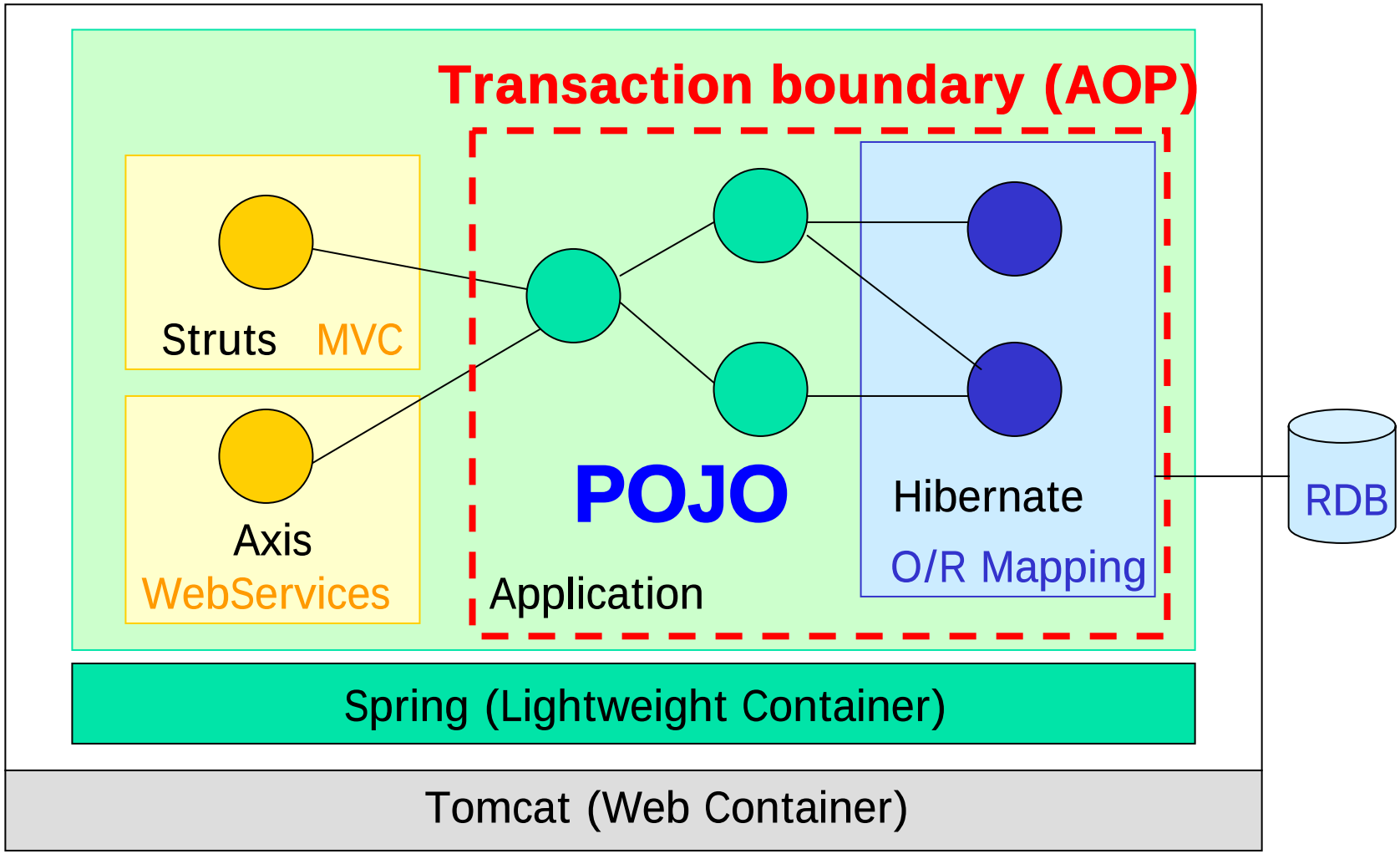
```
<bean id="facade"
        class="org.springframework.transaction.interceptor.
                                        TransactionProxyFactoryBean">
  <property name="transactionManager">
     <ref  bean="transactionManager"/>
  </property>
  <property name="target"><ref bean="FacadeImpl"></property>
  <property name="transactionAttributes">
     <props>
        <prop key="insert*">PROPAGATION_REQUIRED</prop>
        <prop key="update*">PROPAGATION_REQUIRED</prop>
        <prop key="*">PROPAGATION_REQUIRED,readOnly</prop>
     </props>
  </property>
</bean>
```

DynamicProxy

- 参照するのは別の顔



先のモデルをもう一度



Springを使うと何がよいか？

- POJOベースで開発できる
- テストがやさしくなる
  - サーバを起動しなくてもテストができる
  - 品質向上と開発効率の改善が期待できる
- 変更・管理がやさしくなる
  - 依存性管理、トランザクション管理、リソース管理の一元化ができる
- オープンソースである
  - コードリーディングによるスキルアップも期待できる

まとめ

- Javaオープンソースによる J2EE開発は、システム要件の適性にあわせてビルディングブロックを選択･統合する技術ともいえます。

- Webコンテナの信頼性の向上と期待効果の高い軽量コンテナの出現によって、Javaオープンソースによる POJOベースの J2EEシステム開発が実現可能な時代になりつつあります。

手がかりを探す場所

- **http://www.springframework.org/**
  - Spring Reference Documentation
  - Developing a Spring Framework MVC application step-by-step
- **参考書**
  - **軽快なJava ― Better, Faster, Lighter Java**

    Bruce A.Tate, Justin Gehtland (原著)，岩谷 宏 訳

    O'Reilly Japan
  - **JUnitイン・アクション**

    Vincent Massol, Ted Husted (原著)，クイープ 訳

    ソフトバンクパブリッシング

Thank you!