いまさら人には聞けない DI×AOP入門・2009

2009.9.12

小森 裕介(komo@littleforest.jp)

はじめまして！

- **名前**：小森 裕介
- **Blog**：http://d.hatena.ne.jp/y-komori/「こもりん日記」
- **所属**：ウルシステムズ株式会社（http://www.ulsystems.co.jp）
- **主な仕事**：
  - 各種ITコンサルティング
  - 各種SI支援

- **教育・各種執筆活動**：
  - 日経ソフトウエア「とことん作って覚える! Java入門」連載
  - 「なぜ、あなたはJavaでオブジェクト指向開発ができないのか」
- **Seasar2とのかかわり**
  - Urumaコミッタ、S2Containerコミッタ、S2JMSコミッタ

はじめに

DI×AOPが世に出て早数年・・・

SeasarやSpringをなんとなく使ってるけど、
「なにが良いの?」と聞かれてもうまく説明できない・・

自分の開発現場にも導入したいのだけど、
上司や同僚を納得させる自信がない・・・

そんなアナタに

DI×AOPの本質をお伝えします!

アジェンダ

1. 「依存性」の問題点
2. 「依存性」との戦いの歴史
3. POJOによる「継承関係・実装関係の依存」からの脱却
4. DIによる「オブジェクト利用の依存」からの脱却
5. DIによる「実装クラスへの依存」からの脱却
6. AOPはなぜ必要か
7. AOPの考え方
8. AOPの実現方法

DI×AOPの必要性

DI×AOPはなぜ必要か?

それは

「依存性からの脱却」

を促進するため

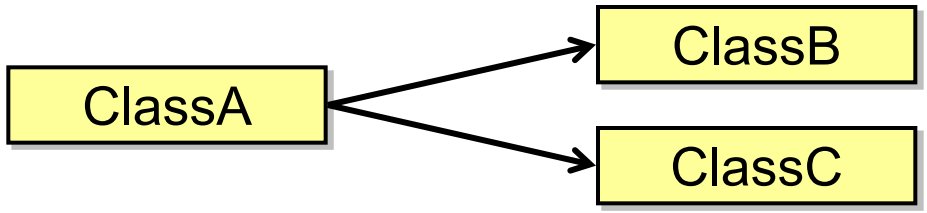
1.「依存性」の問題点
依存性とはなにか

■オブジェクト指向における「依存性」は3種類ある

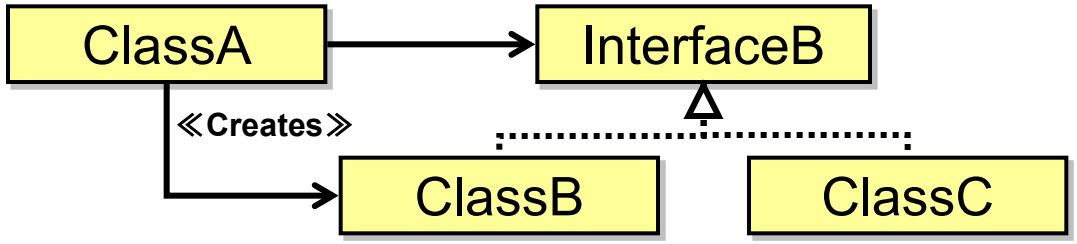
オブジェクト指向における「依存性」

継承関係・実装関係の依存

オブジェクト利用の依存



実装クラスへの依存

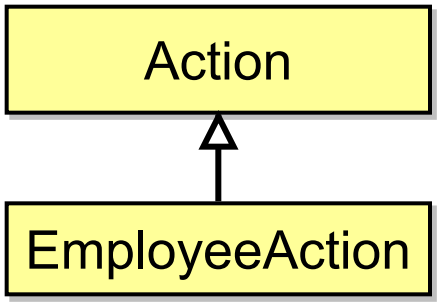


1.「依存性」の問題点
継承関係・実装関係の依存（1/2）

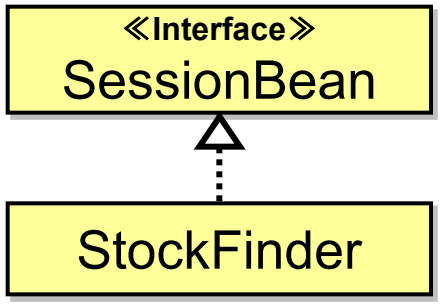
■継承関係・実装関係による依存

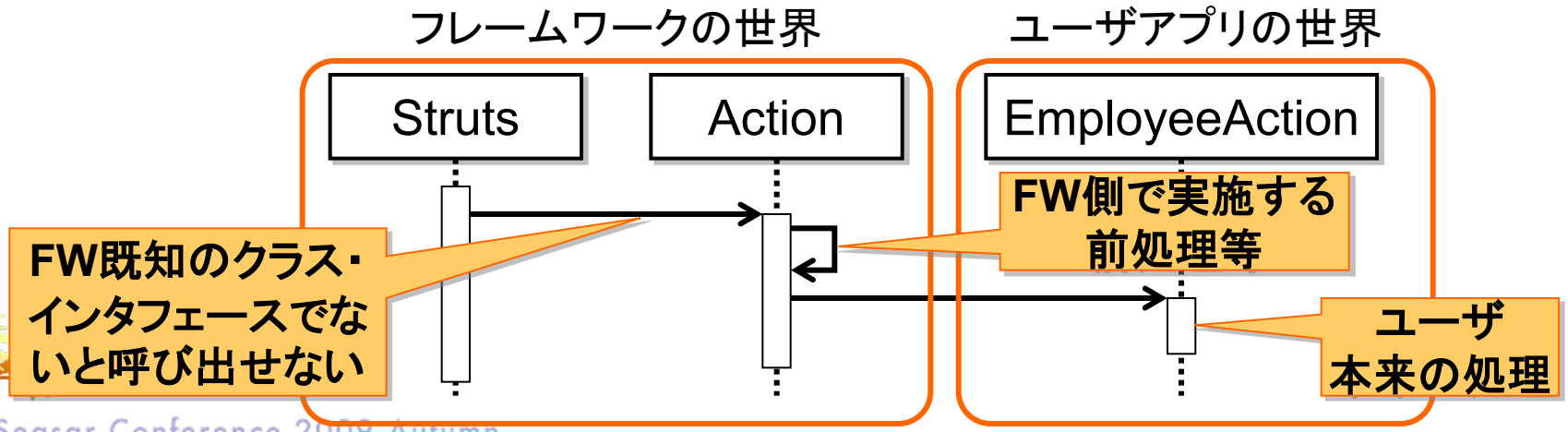
– フレームワークの提供するクラス・インタフェースを利用

【例1】Struts の Actionクラス



【例2】EJB2 の SessionBean





※ このシーケンス図は模式的なものであり、実際のStrutsの動作とは異なります

1.「依存性」の問題点
継承関係・実装関係の依存 (2/2)

■「継承関係・実装関係の依存」の問題点

– フレームワークにロックオンされる

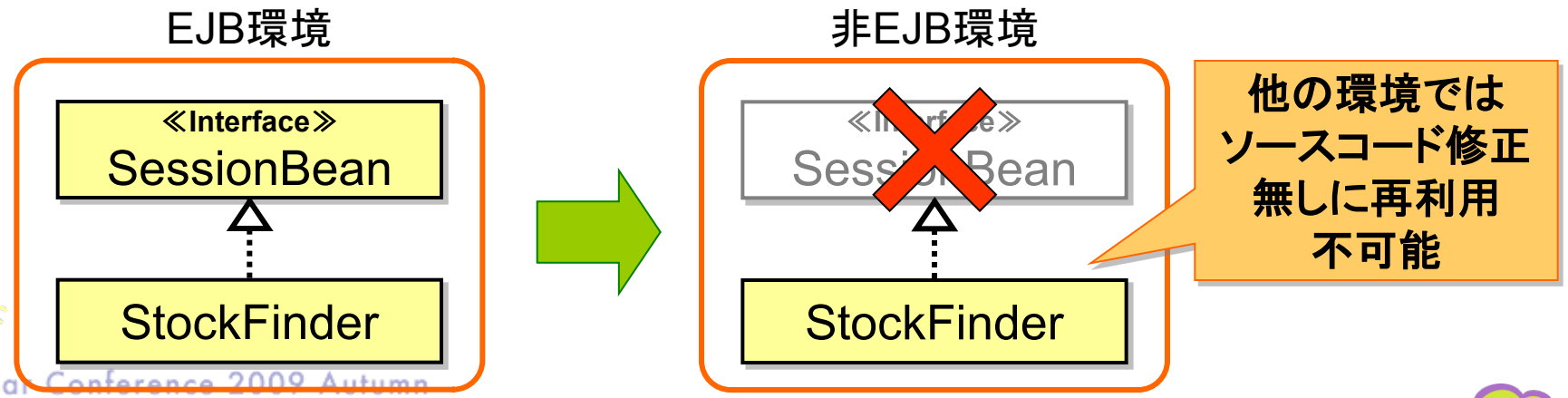
✓継承・実装関係があるため、ユーザが作成したコードを他のフレームワーク下でそのまま利用できない

– クラス単体でテストがしにくい

→ 再利用性の低下

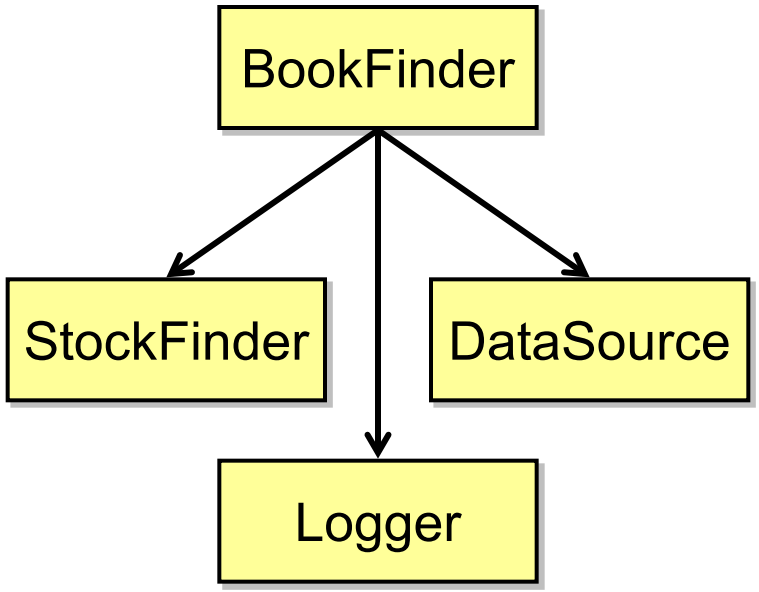
✓フレームワークがインスタンス生成をしていると単体テスト時にインスタンス作成できず、テストが困難

→ 品質の低下



1.「依存性」の問題点 オブジェクト利用の依存（1/2）

■あるオブジェクトが他のオブジェクト利用する場合




```
public class BookFinder {
    private StockFinder stockFinder;
    private DataSource dataSource;
    private Logger logger;

    public BookFinder(StockFinder stockFinder) {
        this.stockFinder = stockFinder;
        Context ctx = new InitialContext();
        DataSource ds =
            (DataSource)ctx.lookup("java:comp/env/jdbc/MySQL");
        this.logger = LoggerFactory.getLogger(this.getClass());
    }
    public List<Book> findBook(Condition cond) {
        ..............
    }
}
```

コンストラクタ経由の参照渡し

JNDIからのルックアップ

Factoryからの取得

1.「依存性」の問題点 オブジェクト利用の依存 (2/2)

■「オブジェクト利用の依存」の問題点

- 学習量が多い
  - ✓インスタンス取得のための様々な作法を知らなくてはならない

→ 生産性の低下

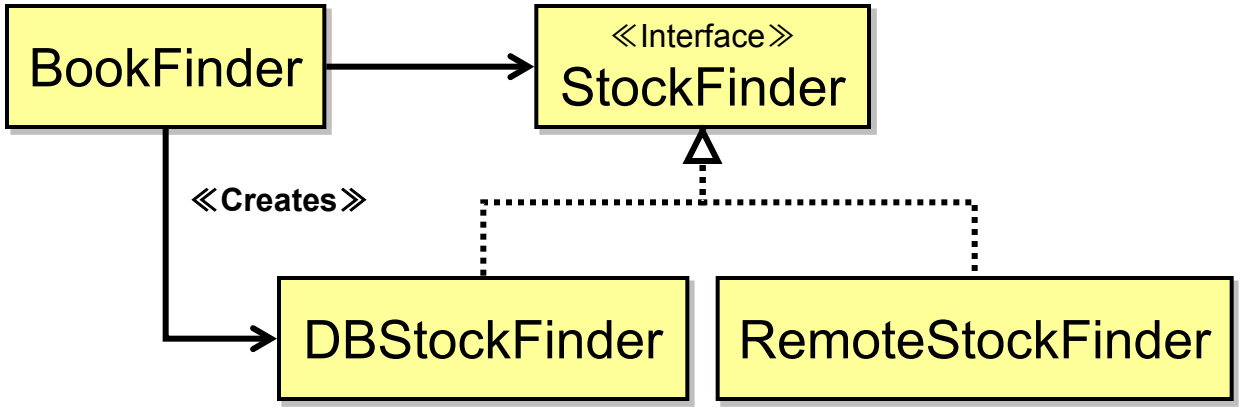
- コード量が多くなる
  - ✓「オブジェクト組み立て」のための本質的ではない大量のコード記述が必要

→ 保守性の低下

```
private Logger logger;
public BookFinder(StockFinder stockFinder) {
    this.stockFinder = stockFinder;
    Context ctx = new InitialContext();
    DataSource ds =
        (DataSource)ctx.lookup("java:comp/env/jdbc/MySQL");
    this.logger = LoggerFactory.getLogger(this.getClass());
}
```

1.「依存性」の問題点 実装クラスへの依存（1/2）

■あるオブジェクト内部で他のオブジェクトを new している場合




```
public class BookFinder {
    private StockFinder stockFinder;
    public BookFinder() {
        this.stockFinder = new DBStockFinder();
    }
}
```

利用側で実装クラスを直接 new している

```
public class BookFinder {
    private StockFinder stockFinder;
    public BookFinder() {
        this.stockFinder = new RemoteStockFinder();
    }
}
```

実装クラスを変更するためには、利用側のコードを書き換えなければならない

1.「依存性」の問題点 実装クラスへの依存 (2/2)

■「実装クラスへの依存」の問題点

- 「仕様と実装の分離」を徹底できない
  - ✓ せっかくインタフェースで仕様を分離しているのに、インスタンス生成のために実装クラスへ依存している

→ 保守性の低下

- モックオブジェクトへの差し替えが難しい
  - ✓ テスト用のモックオブジェクトを使用するにはテスト対象コードの変更が必要

→ 品質の低下

```
public class BookFinder {
    private StockFinder stockFinder;

    public BookFinder() {
        this.stockFinder = new DBStockFinder();
    }
}
```

利用側で実装クラスを直接 new している

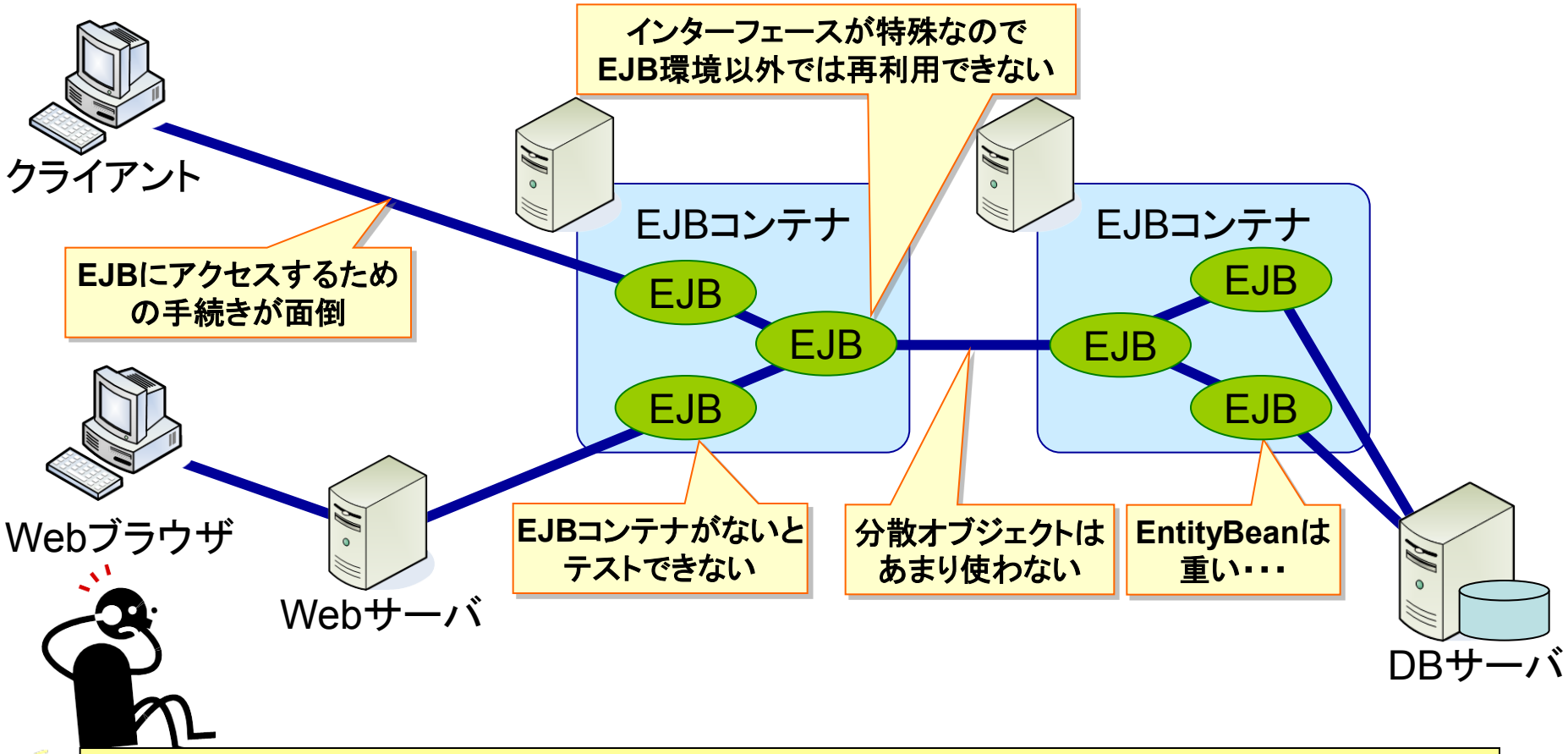
```
public class BookFinder {
    private StockFinder stockFinder;

    public BookFinder() {
        this.stockFinder = new RemoteStockFinder();
    }
}
```

実装クラスを変更するためには、利用側のコードを書き換えなければならない

2.「依存性」との戦いの歴史
「依存性」の権化・EJB2.0

■高度な機能を提供するため、とても複雑になった

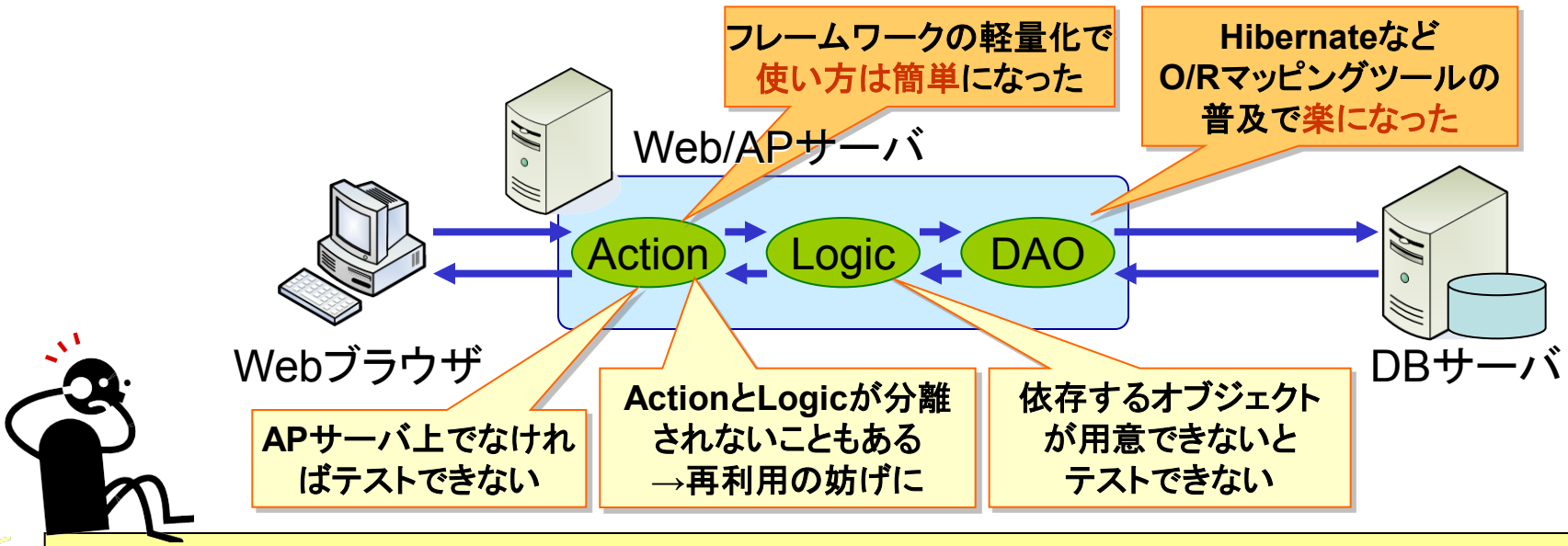


もっと簡単にコンポーネント技術を利用する方法はないの？

2.「依存性」との戦いの歴史
なぜこの状況が放置されたのか？

■Javaのエンタープライズ利用は
Webアプリケーションが主流となった

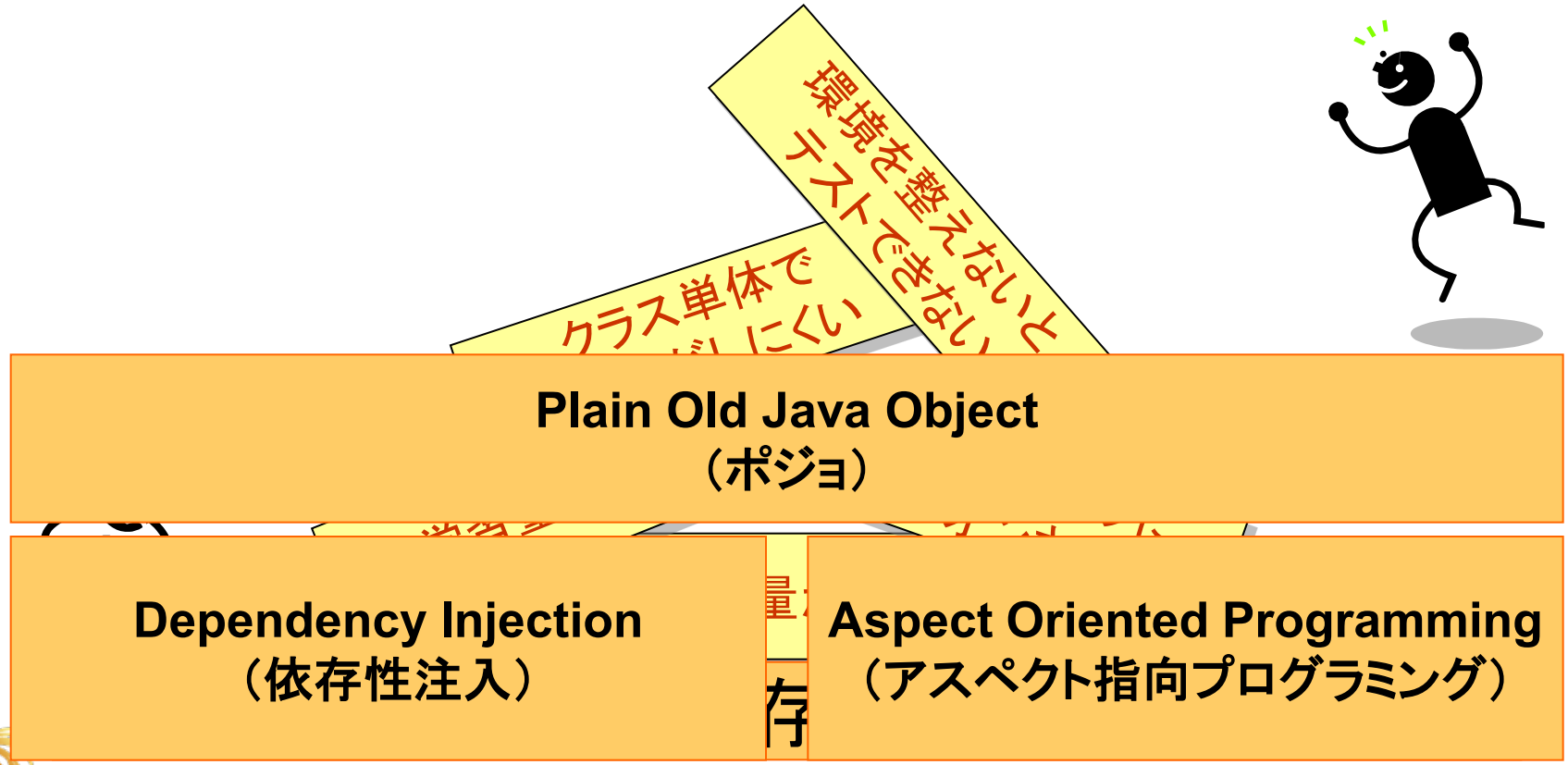
→ Struts をはじめとする「Webアプリケーションフレームワーク」が台頭。
中・小規模のシステム開発では、
EJBの提供する機能がなくてもあまり困らなかった



本質的な問題は未解決。大規模システムでは影響が顕著に。

POJO with DI×AOP による依存性からの脱却

■「依存性」が生み出す様々な問題を
POJO with DI×AOP で解決しよう!!



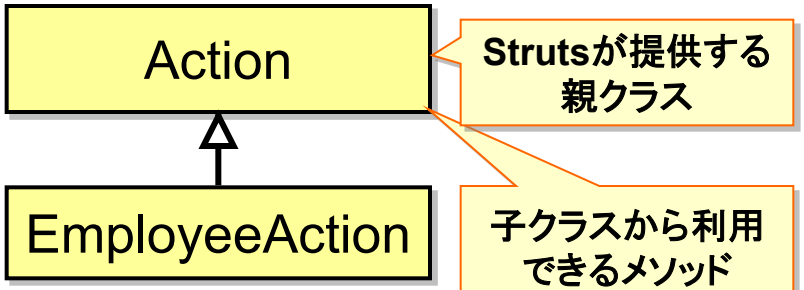
3. POJOによる「継承関係・実装関係の依存」からの脱却 Before POJO

■POJO（Plain Old Java Object）という考え方

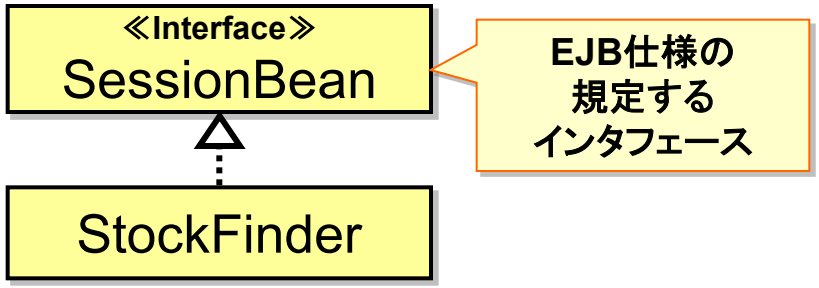
– FW特有のI/F・親クラスを持たない、「昔ながらのただのJavaオブジェクト」

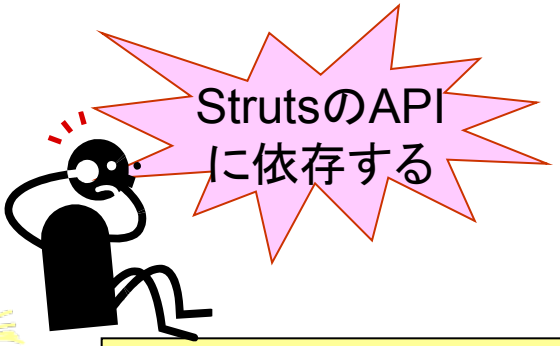
Before POJO

【例1】Struts の Actionクラス



【例2】EJB2 の SessionBean





EJBとして利用するために特定メソッドの実装が強制される

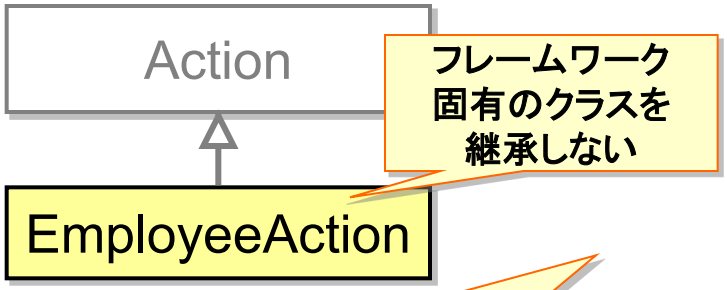
自由な再利用やJunitによる単体テストの妨げになった

3. POJOによる「継承関係・実装関係の依存」からの脱却 After POJO

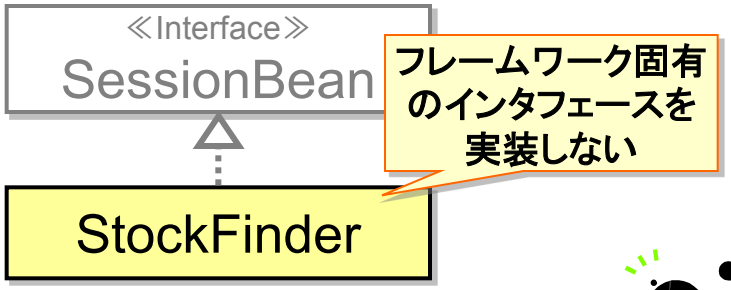
■POJO（Plain Old Java Object）という考え方
- FW特有のI/F・親クラスを持たない、「昔ながらのただのJavaオブジェクト」

After POJO

【例1】Struts の Actionクラス



【例2】EJB2 の SessionBean



簡単にかける！

どこでも使える！

フレームワーク利用者の記述するコードはPOJOにする

■フレームワークの影響を受けず再利用が可能 → 再利用性の向上

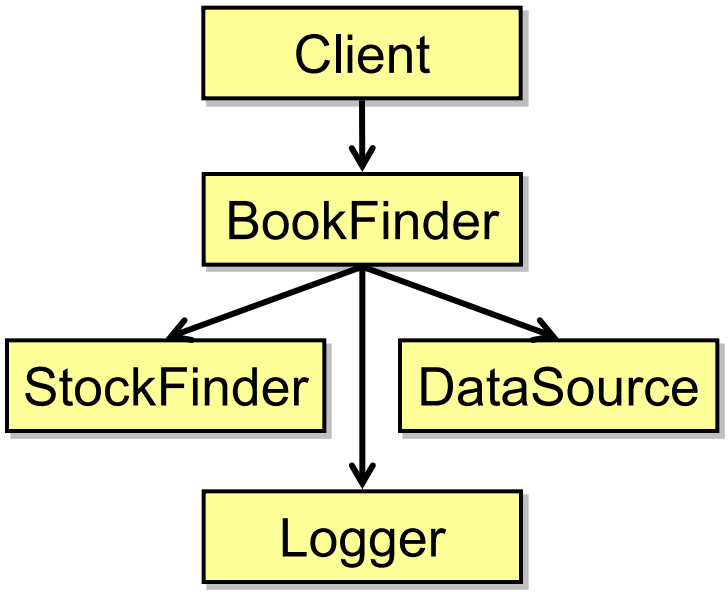
■クラス単体でのテストがしやすくなる → 品質の向上

4. DIによる「オブジェクト利用の依存」からの脱却 Before DI

■依存性注入（Dependency Injection）という考え方

– 必要なオブジェクトはDIコンテナが生成して注入する

Before DI




```
public class BookFinder {
    private StockFinder stockFinder;
    private DataSource dataSource;
    private Logger logger;

    public BookFinder(StockFinder stockFinder) {
        this.stockFinder = stockFinder;

        Context ctx = new InitialContext();
        DataSource ds =
            (DataSource)ctx.lookup("java:comp/env/jdbc/MySQL");

        this.logger = LoggerFactory.getLogger(this.getClass());
    }

    public List<Book> findBook(Condition cond) {
        logger.log(“書籍検索を開始します”);
        List<Book> books = dataSource.select();
        logger.log(“書籍検索を終了しました”);
        return books;
    }
}
```

大量のオブジェクト組み立てコード

ビジネスロジック本来の処理

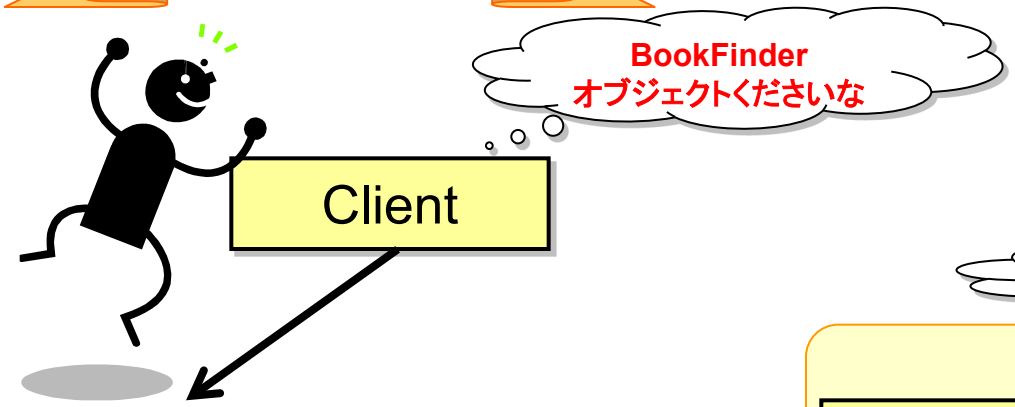
オブジェクトの組み立てが大変だった

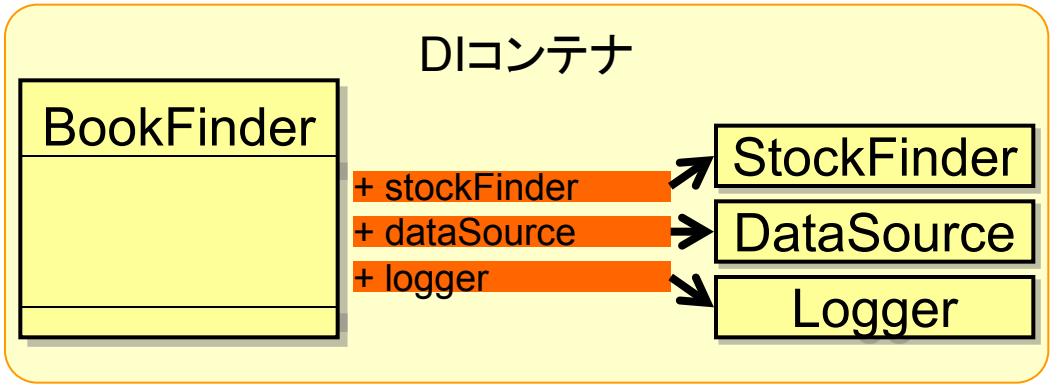
4. DIによる「オブジェクト利用の依存」からの脱却 After DI

■依存性注入（Dependency Injection）という考え方

– 必要なオブジェクトはDIコンテナが生成して注入する

After DI



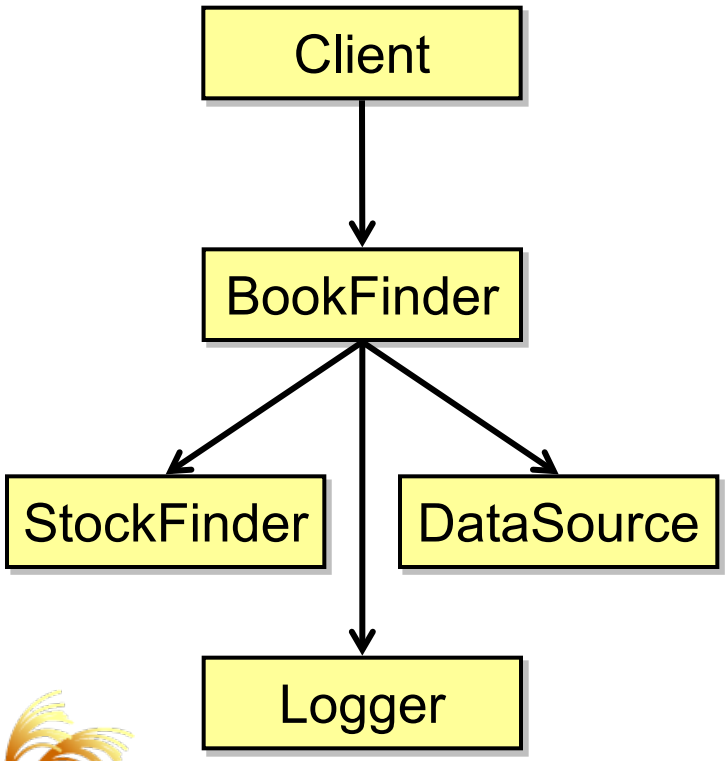


4. DIによる「オブジェクト利用の依存」からの脱却 After DI

■依存性注入（Dependency Injection）という考え方

– 必要なオブジェクトはDIコンテナが生成して注入する

After DI



依存するオブジェクトがどこで生成されるかは気にしなくてよい！

```
public class BookFinder {
    public StockFinder stockFinder;
    public DataSource dataSource;
    public Logger logger;

    public List<Book> findBook(Condition cond) {
        logger.log("書籍検索を開始します");
        List<Book> books = dataSource.select();
        logger.log("書籍検索を終了しました");
        return books;
    }
}
```

ビジネスロジック本来の処理

■学習量が減る → 生産性の向上

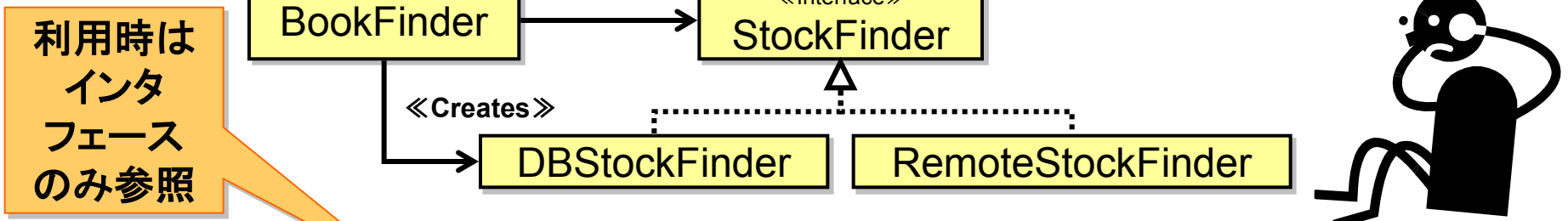
■コード量が減る → 保守性の向上

5. DIによる「実装クラスへの依存」からの脱却 Before DI

■DIによる実装クラスの外部定義

– インタフェースに対する実装クラス外部定義化・生成はDIコンテナが行う

Before DI




```
public class BookFinder {
    private StockFinder stockFinder;
    public BookFinder() {
        this.stockFinder = new DBStockFinder();
    }
}
```

利用側で実装クラスを直接 new している

```
public class BookFinder {
    private StockFinder stockFinder;
    public BookFinder() {
        this.stockFinder = new RemoteStockFinder();
    }
}
```

実装クラスを変更するためには、利用側のコードを書き換えなければならない

「仕様と実装の分離」が徹底できなかった

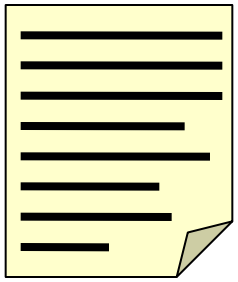
5. DIによる「実装クラスへの依存」からの脱却 After DI

■DIによる実装クラスの外部定義

– インタフェースに対する実装クラス外部定義化・生成はDIコンテナが行う

After DI

```
public class BookFinder {
    public StockFinder stockFinder;
}
```

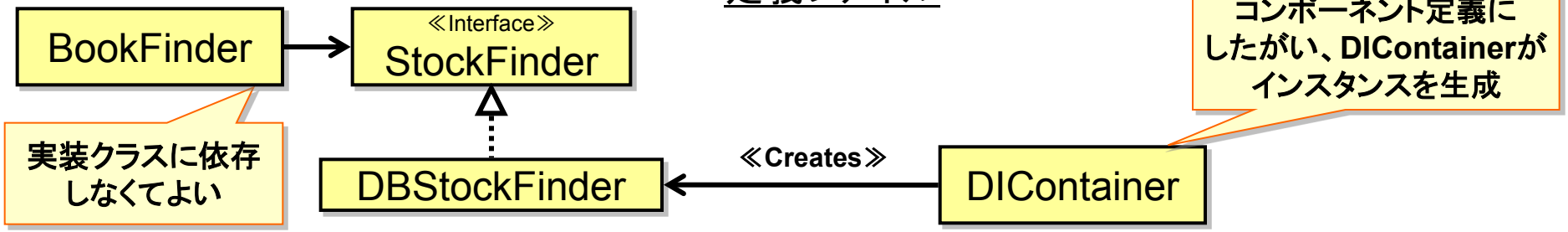


コンポーネント定義ファイル

Seasar2ではdiconファイル

```
<component class="DBStockFinder" />
```

実装クラスを変更する場合、コンポーネント定義を修正する



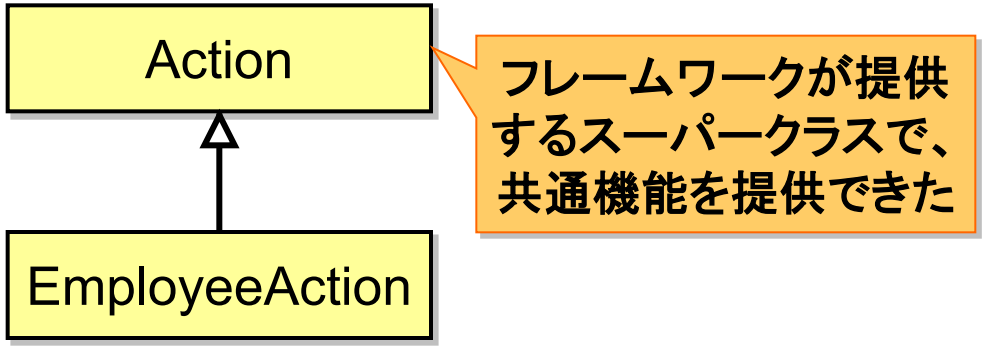
■「仕様と実装」が完全に分離できる → 保守性の向上

■テスト用のモックオブジェクトが導入しやすい → 品質の向上

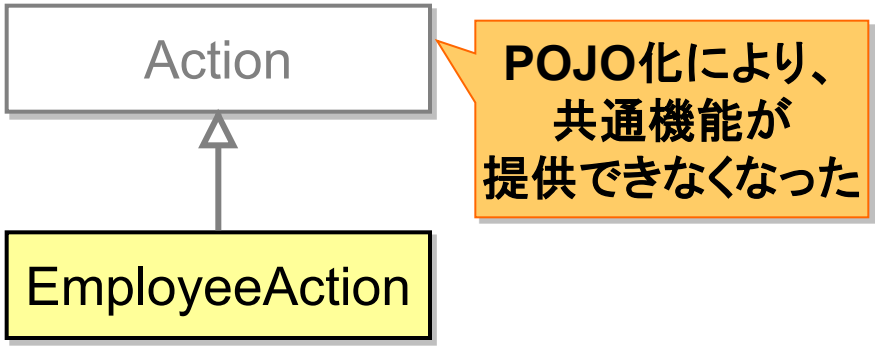
6. AOPはなぜ必要か
POJO化によるクラス拡張の欠落

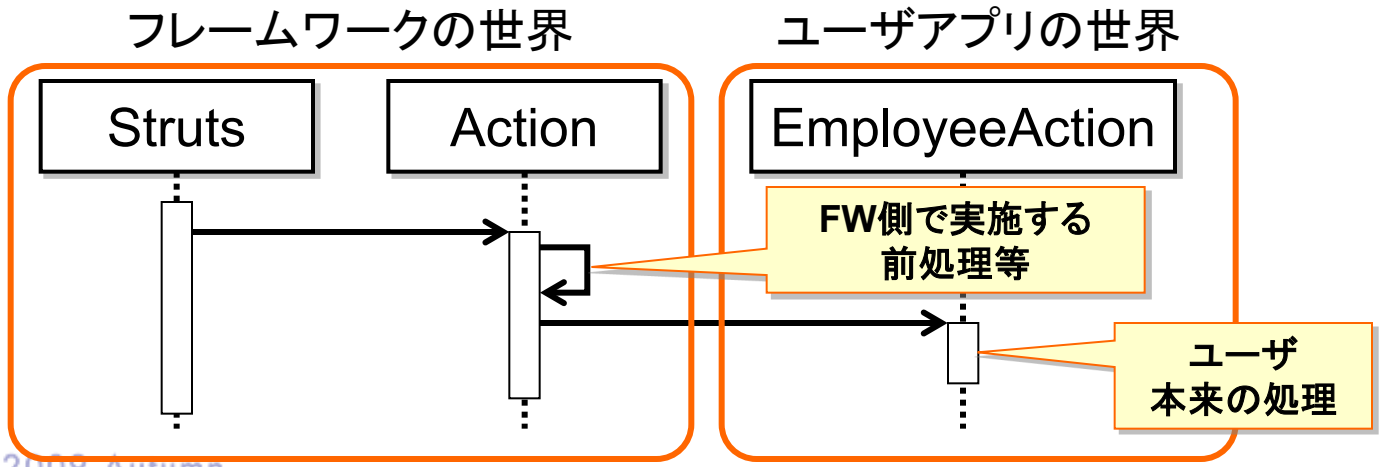
■フレームワーク利用者のコードをPOJO化すると、フレームワーク側からの機能追加ができなくなる

Before POJO



After POJO

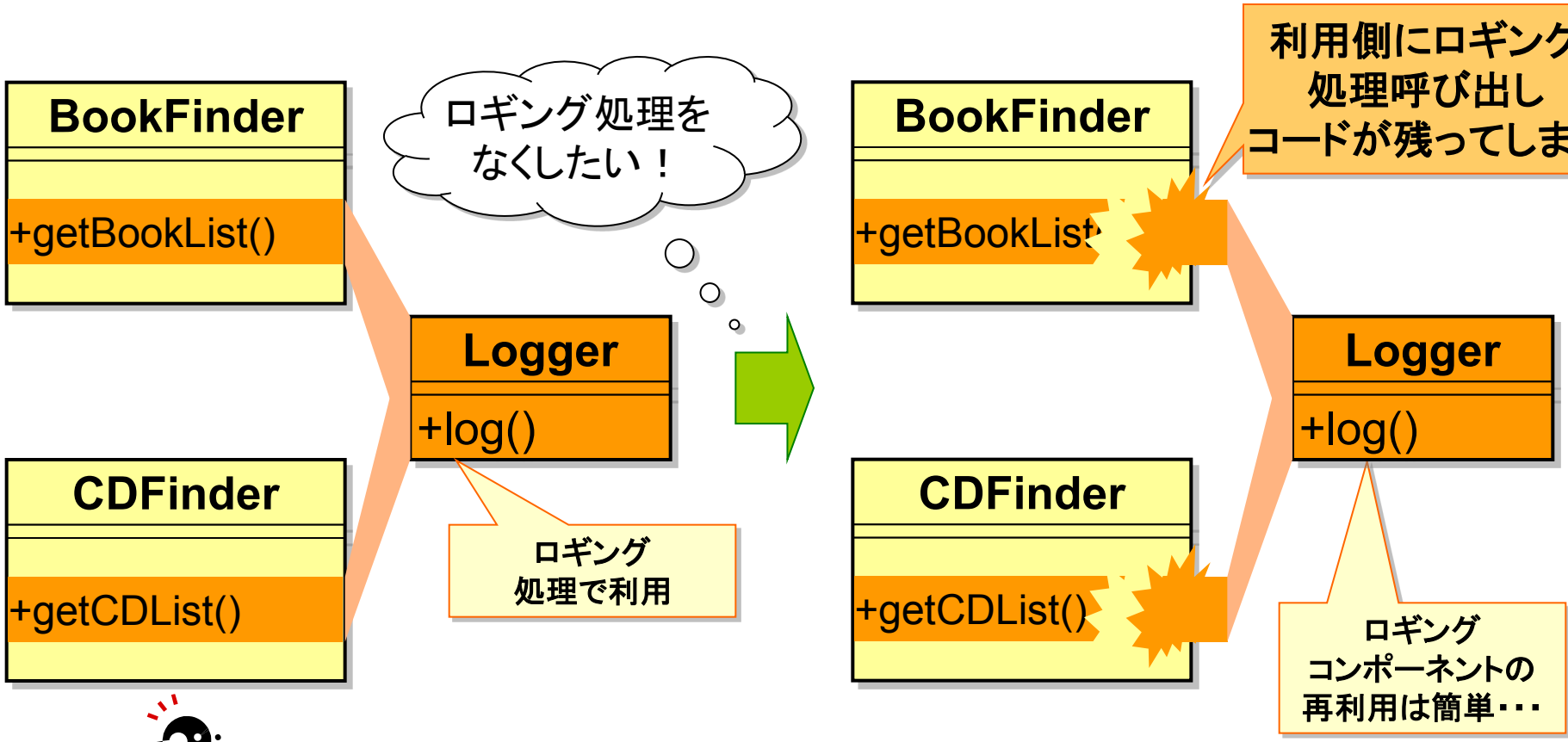




※ このシーケンス図は模式的なものであり、実際のStrutsの動作とは異なります

6. AOPはなぜ必要か
複数クラスから呼び出される共通機能への依存

■例:ロギング処理で発生する問題



「共通コンポーネント」を利用すると、
各所に呼び出しコードが混ざってしまう

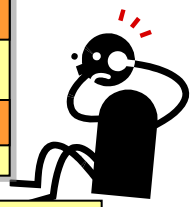
6. AOPはなぜ必要か
非機能要件はモジュール単独分離が難しい

■非機能要件に関する処理は、全機能に影響するため、モジュールとして分離しにくい

➡ ロギング処理も非機能要件の一つ

非機能要件
システムの本来の機能とは関係ないが、
信頼性や保守性、使いやすさを向上させるための要件

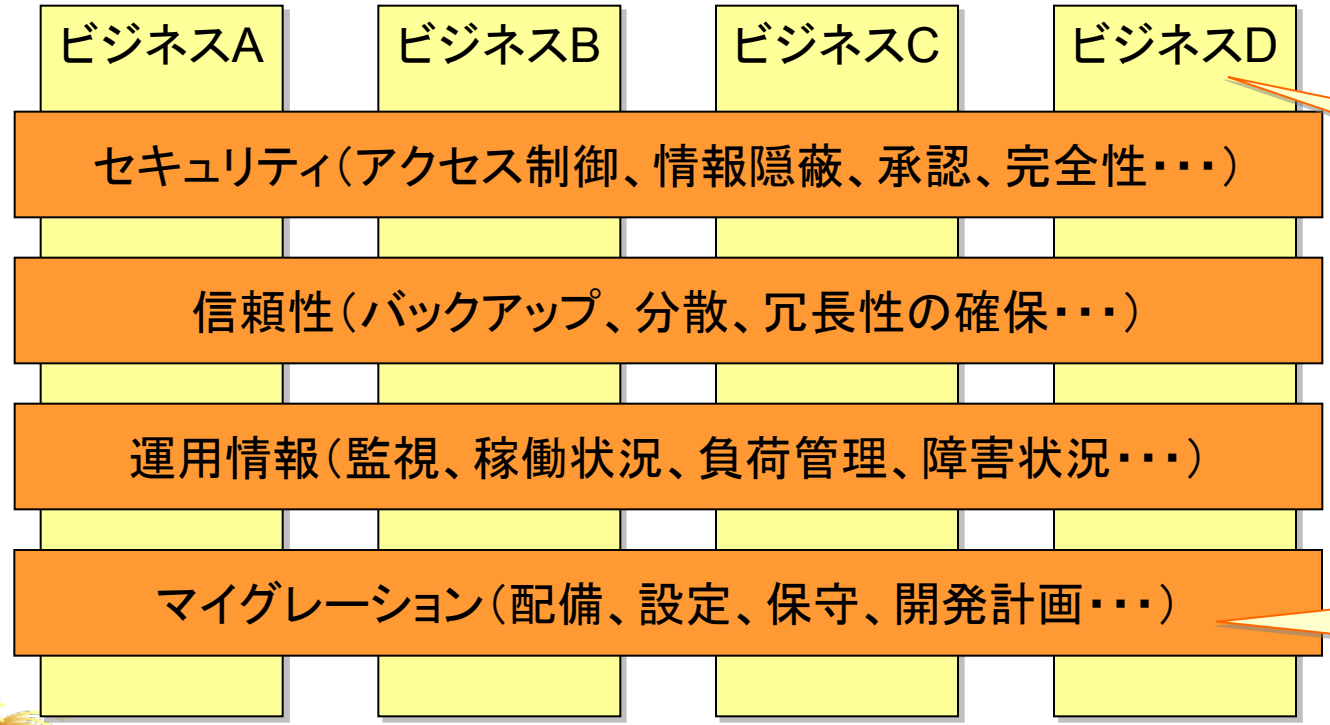
機能要件A	機能要件B	機能要件C	機能要件D
非機能要件Ⅰ	非機能要件Ⅰ	非機能要件Ⅰ	非機能要件Ⅰ
非機能要件Ⅱ	非機能要件Ⅱ	非機能要件Ⅱ	非機能要件Ⅱ
非機能要件Ⅲ	非機能要件Ⅲ	非機能要件Ⅲ	非機能要件Ⅲ
非機能要件Ⅳ	非機能要件Ⅳ	非機能要件Ⅳ	非機能要件Ⅳ



各機能に共通する処理をモジュール化する方法はないの？

7. AOPの考え方
アスペクト指向は関心事の分離から

■システムを2種類の要件に分け、横断的関心事を分離するのがアスペクト指向の考え方



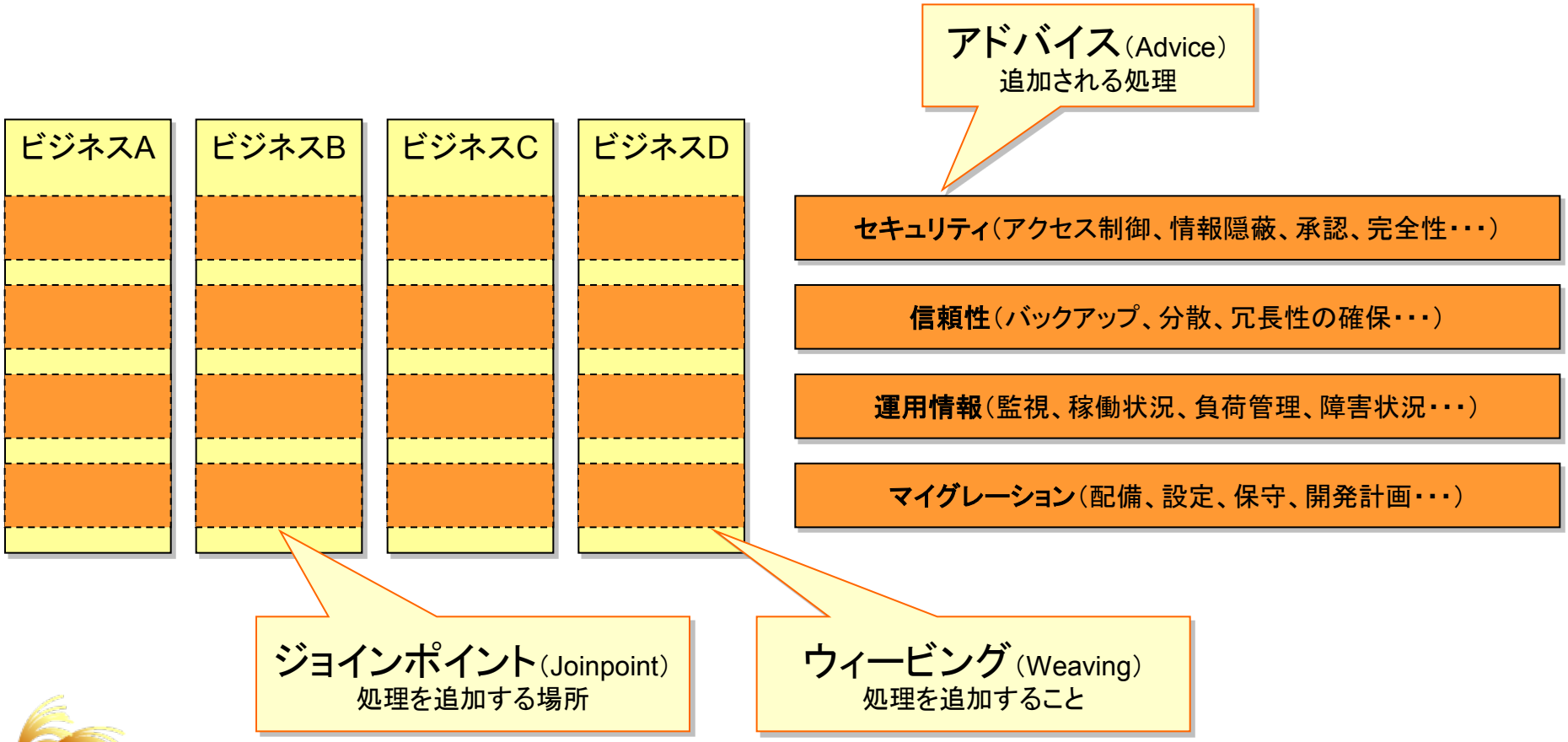
中心的関心事
（Core concern）

横断的関心事
（Cross cutting concern）

【出典】『Seasar2で学ぶDIとAOP』（arton著、技術評論社）p20

7. AOPの考え方
ジョインポイントとアドバイス

■アスペクト指向では、機能の中にジョインポイントを定義し、アドバイスをウィービングする

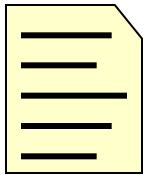


8. AOPの実現方法
アスペクトコンパイラによるAOPの実現

■AOPの実現には専用のコンパイラが必要だった

Before DI × AOP

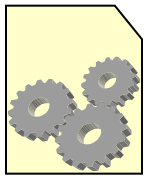
Javaソースコード



アスペクト(Aspect)

アスペクト
コンパイラ

Javaクラスファイル



アドバイスとポイントカット
を記述したもの

コンパイルと同時に
ウィービングを行って
クラスファイルを出力する

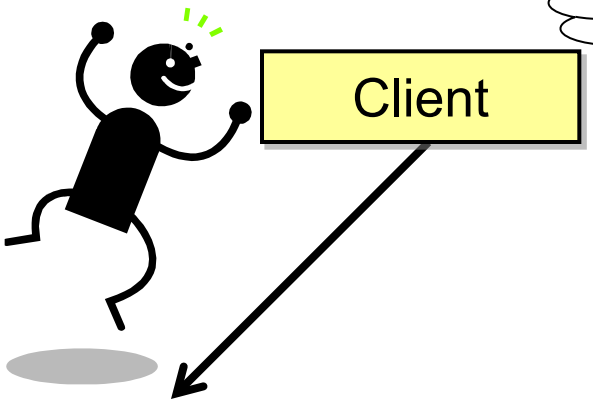
ポイントカット・・・ジョインポイントとアドバイスの結びつきを定義したもの

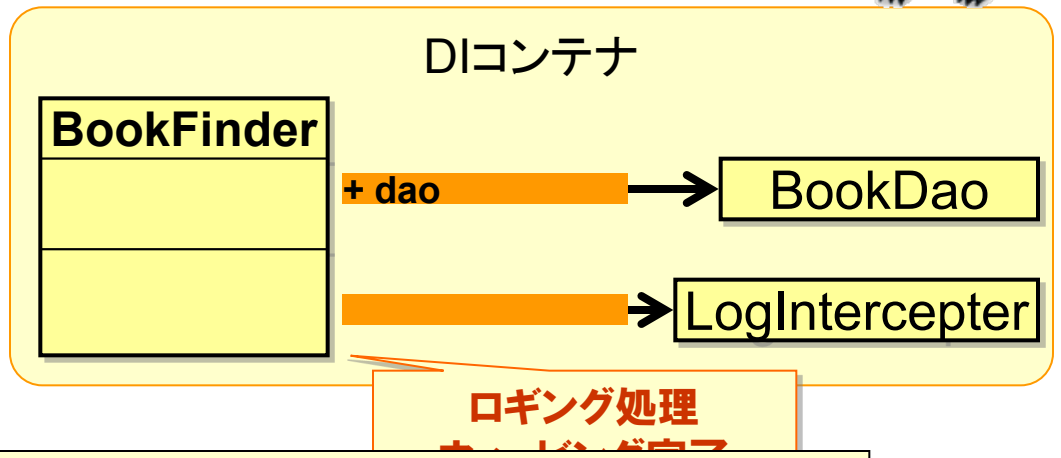
こいつぁ、面倒だ・・・

8. AOPの実現方法
DIコンテナと連携したAOPの実現

■DIコンテナで生成時にウィービングを実施

After DI×AOP





DI×AOPで自動ウィービングが可能に！

DI×AOPの効果

■POJOの良さを生かしながら、フレームワークによるユーザコードの機能拡張を実現

- ロギング処理
  - ✓ S2Container(TraceInterceptor)
- 例外処理
  - ✓ S2Container(ThrowsInterceptor)
- セッションオブジェクト管理
  - ✓ S2Container(Remove/InvalidateSessionInterceptor)
- toString()メソッドの自動実装
  - ✓ S2Container(ToStringInterceptor)
- トランザクション処理
  - ✓ S2Tx
- リモートオブジェクト化
  - ✓ S2Remoting
- Webアプリケーションでのログイン状態チェック
  - ✓ 独自実装で可能

DIコンテナに対する5つのギモン

■依存関係はどうやって判断するの？

- Seasar2なら、インターフェースに基づいて自動判断します

More Info! http://s2container.seasar.org/2.4/ja/DIContainer.html#AutoBindingMode

■結局設定ファイルをたくさん書くのでは？

- Seasar2なら、AutoRegisterでカンタンに登録！

More Info! http://s2container.seasar.org/2.4/ja/DIContainer.html#ComponentAutoRegister

■設定ファイルのデバッグが大変？

- Kijimuna（キジムナ）で記述の誤りをチェックできます！

More Info! http://kijimuna.seasar.org/

■結局はリフレクションでしょ、遅くないの？

- 独自のキャッシュ機構で速度低下はほとんどありません

More Info! http://s2container.seasar.org/ja/benchmark/20060412_seasar_vs_spring.ppt

■Setterを書くのが面倒くさい！

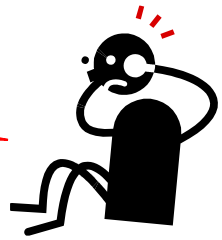
- publicフィールドインジェクション で setter いらず！

More Info! http://s2container.seasar.org/2.4/ja/DIContainer.html#FieldInjection

どうやって使っていったらよいの？

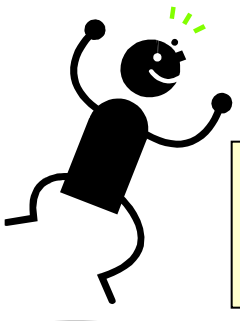
■DIコンテナのメリットはなんとなくわかった・・・

でも、
開発現場で
どう役立てればいいの？



DIコンテナの機能を
フルに使い切って設計するのは、
けっこう**ムズカシイ**！

そこで・・・



まずは、S2Containerを100%生かして作られた
周辺プロダクトを使ってください！

まずは、SAStruts、Teeda、S2JDBC、S2Dao
がオススメです！

本日のまとめ

- 「依存性」の持つ問題点を理解した
  - 継承関係・実装関係の依存
  - オブジェクト利用の依存
  - 実装クラスへの依存
- POJO with DI×AOPによる「依存性からの脱却」方法と、そこから得られるメリットを理解した
  - 生産性の向上
  - 保守性の向上
  - 拡張性の向上
  - 品質の向上
  - 再利用性の向上

お勧め書籍
Seasar2・AOPについて、もっと知りたい方へ（1/2）

■『Seasar入門　～はじめてのDI&AOP～』
- 監修：ひが やすを　著：須賀 幸次 他
- 価格：3,570円
- 出版社：ソフトバンククリエイティブ
- ISBN：4797331968

DI×AOPの
キホンをきっちり
学習

■ Seasar2で学ぶDIとAOP アスペクト指向によるJava開発
- 著：arton
- 価格：3,360円
- 出版社: 技術評論社
- ISBN: 4774128554

DI×AOPの考え方と
Webアプリ開発について理解
（S2JSF+S2Dao）

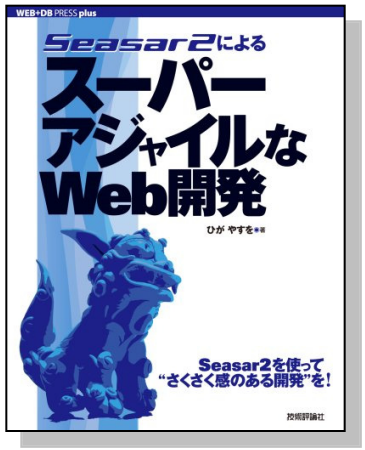
■『アスペクト指向入門』-Java・オブジェクト指向からAspectJプログラミングへ
- 著：千葉 滋
- 価格：￥2,480＋税
- 出版社：技術評論社
- ISBN：4-7741-2581-4

アスペクト指向の考え方を
しっかり身につけたい方へ

お勧め書籍
Seasar2・AOPについて、もっと知りたい方へ（2/2）

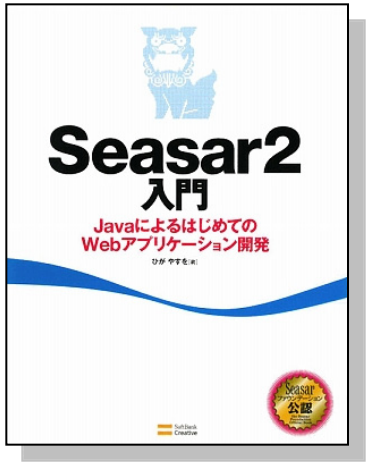
- Seasar2によるスーパーアジャイルなWeb開発
  - 著：ひが やすを
  - 価格：2,499円
  - 出版社：技術評論社
  - ISBN：4774134368

Teeda+S2Daoによる
Webアプリケーション開発
について解説



- Seasar2入門 Javaによるはじめ てのWebアプリケーション開発
  - 著：ひが やすを
  - 価格：2,730円
  - 出版社：ソフトバンククリエイティブ
  - ISBN：4797345241

SAStruts+S2JDBCによる
Webアプリケーション開発
について解説



ご静聴
ありがとうございました