
# 50分でわかるほんとのDI

Seasarファウンデーション理事（休職中）
株式会社スターロジック
羽生　章洋


# 自己紹介

- 羽生　章洋（はぶ　あきひろ）と申します。
  - 昭和43年生まれの今年38歳です。
  - 受託電算業務の伝票打ちから始まりました。
  - プログラマ～システムエンジニアを経てコンサルタントをやってきました。
  - 様々な業種・業務に関わってきました。
  - 今はスターロジックという会社の経営をやっています。
    - 今日はブース出してます！
  - Seasarファウンデーションの理事（但し休職中）でもあります。
    - 次のセッションと、次の次のセッションにも、理事として顔を出します。
- 今日のお話にご興味を持たれた方は、お気軽にご連絡ください ませ。
  - http://www.starlogic.jp/
  - habu@starlogic.jp
  - http://d.hatena.ne.jp/habuakihiro/（はぶにっき）


# Before DI

- 何故、DIが誕生したのか？
  ・・・J2EEへのアンチテーゼ
- DIの波を受けて生まれたのがJEE5

というわけで、まずはDI誕生までの流れを・・・


# J2EEって何？

- Java2 Enterprise Edition
- 規格の名称であり、実体はAPIの集合体
- Javaで基幹系を作ろう！　という野望
  - そのために、てんでバラバラだったメインフレームの機能を「とりあえず」全部Javaに持ってきた

- JTA 1.0
- Servlet 2.4
- JSP 2.0
- EJB 2.1
- JAXR 1.0 (JSR-93)
- JACC (JSR-115)
- JAXP 1.2
- JavaMail 1.3
- JAF 1.0
- JAX-RPC 1.1 (JSR-101)
- SAAJ 1.1
- Web Services (JSR-109)
- Management (JSR-77)
- Deployment (JSR-88)
- Connectors 1.5
- JMX 1.1
- JMS 1.1


# J2EEってどうして難しいの？

- Javaを勉強しただけでは使えないから。⇒どうして？
- J2EEのロールモデル
  （VBのビジネスモデルを真似しようとした）
  - J2EE製品プロバイダ（いわゆるAPサーバベンダ）
  - アプリケーションコンポーネントプロバイダ
    （Activeコンポーネントのベンダみたいなイメージ）
  - アプリケーションアセンブラ（いわゆるSIer、SE、PG）
  - デプロイヤ
  - システム管理者（運用時：いわゆる情報システム部門）
  - ツールプロバイダ（開発環境ツールベンダのイメージ）
- 元々が寄せ集め、かつ役割ごとのお約束が多い
- つまり、「役割に応じたコンテナの乗りこなし方」を別途マスタしないといけない」


# J2EEってどうして難しいの？

- 難しいので色んな例が出てきた
  - ブループリント（PetStore）
  - J2EEパターン
- 一番難しかったのがEJBの使い方
  - EJBの「JB」は名前だけ。実際には全然違う。
  - EJBの良し悪しではない。実際に使いこなせる人がなかなか増えないという「事実」
- ぶっちゃけ「頭が良くて勤勉な人」しか使えない（私は凹む）


# EJBって何？

- Enterprise JavaBeansの略で、J2EEのAPI群のひとつ
  - 本来のJavaBeansとは似ても似つかぬ別物
  - ちなみにJavaBeansはJavaのコンポーネント規格
- 分散アーキテクチャを前提
  - 大規模には分散でしょ？　⇒　そんなわけない！
- ビジネスロジックだけ書けば、システムサービスはコンテナが提供してくれる
  - コンテナにどっぷり依存してしまう
- 想定している企業規模が尋常ではない


# デベロッパの反乱

- オープンソースを中心に「もっとこうしようぜ！」
  - J2EEパターンも大変なので、フレームワークがいろいろと生まれた
  - フレームワークがどんどん進んできた
- 喫水線を超えたターニングポイント
  - **IoC（DI）という概念の誕生⇒Springの登場**
    - ロッド・ジョンソンさんの本（邦題：実践J2EEシステムデザイン）
  - **O/Rマッピングツールの進化⇒Hibernateの登場**
- 欲しいのはJ2EEコンテナではなく、コンテナが提供するサービス！
  具体的には、
  - ネームサービス
  - 配備サービス
  - トランザクションサービス
  - セキュリティサービス

  ⇒ **JBossAOP：AOPで透過的なサービスが実現できることを示した**
- **合言葉は「POJOで行こう！」**

**POJO(Plain Old Java Object )**
**Plain Old＝「平凡な」を意味するイディオム**


# POJOな時代のJ2EE開発

- J2EE1.5改めJEE5によるEoD（開発を簡単に）時代の宣言
  - 具体的には、JavaServer Faces ,JAXB2.0 , JAX-RPC2.0 , JDBC4.0 , EJB3.0 , Scripting　など
- EJB3.0のコアスペックは「POJOで行こう！」
  - ようやくEJBの「JB」が名前と実態で一致する
  - 具体的には、Spring(DI,AOP)とHibernate(ORM)のフィードバック
- 心は「Javaプログラミングを知れば、J2EE開発ができるようになる」ようにしようということ


# JEE5の花形

- JEE5の目玉は何と言っても・・・
  - JSF（JavaServer Faces）
  - EJB3.0
- Seasarプロジェクトも対応しました！
  - JSFには、Teeda！
  - EJB3には、Kuina＋S2.4（SessionBean）
- そんな各プロダクトのキーコンセプトは・・・、


# Let's Seasar!　・・・なんだけど

- というわけで、
  さぁTeedaだ！　Kuinaだ！　S2.4だ！！

- ところが、ちょっと進化しすぎ(^^;
- DIって何？　って状態じゃ何が何やらさっぱり。
- しかも近頃のS2は設定ファイル(後述)も
  なくなっちゃった！


# S2Container、進化の歴史

- S2.0：DIコンテナとして登場
- S2.1～2.2：地道な改良
- S2.3：
  - CoCの影響
  - 規約ベースのコンポーネント自動登録を実現
  - 設定ファイルを不要に！
- S2.4：
  - Hot Deploy
  - SessionBeanへの対応


# 原点に戻って、もう一度DIをおさらい

- S2の便利な最新機能の恩恵に浴するために！

## DIとはどういうものか ということを理解しよう！


# 部品化の促進とDI

- 業務システム構築において、
  より開発生産性を高めたい！！
- 生産性向上のためには、部品化の促進をしたい！
- でも単なるクラスだけでは部品化に限界がある。
  ではどうするか・・・？
- 仕様と実装の分離を実現する
  - 仕様をインターフェイスとして定義する
    - ex: interface Logic または ILogic
  - インターフェイスに沿って実装を定義する
    - Java: public class LogicImpl inplements Logic
    - C#: public class LogicImpl : ILogic
    - PHP5: class LogicImpl inplements Logic

**・・・というわけで電卓のサンプルを元に考えてみる**


# 部品化の促進とDI

- まずCalculatorインターフェイスを定義する
- このインターフェイスは、multiplyというメソッドの“仕様”が定義されている（仕様書代わり）。

```
public interface Calculator {
  public int multiply(int source, int by);
}
```


# 部品化の促進とDI

- Culculatorインターフェイスは仕様だけなので、これを実装する。
- 実装は、CalculaMachineクラスとする

```
public class CalculaMachine implements Calculator {
  public int multiply(int source, int by) {
    int ret = 0;
    for (int i = 0; i < by; i++) {
      ret = ret + source;
    }
    return ret;
  }
}
```


# 部品化の促進とDI

- CalculaMachineクラスを使ってみる

```
public class Sample {

  public static void main(String[] args) {
    Calculator calc = new CalculaMachine();
    System.out.println(calc.multiply(10,100));
  }
}
```

*「Calculator」型の変数「calc」を宣言して、CalculaMachineクラスの実体を「calc」に代入している*

```
C:¥>java Sample
1000
```


# 部品化の促進とDI

- CalculaMachineクラスがイマイチなので、Culculatorインターフェイスを実装した別のクラスを作る。名前は、CalcMachineクラスとする

```
public class CalcMachine implements Calculator {
  public int multiply(int source, int by) {
    return source * by;
  }
}
```


# 部品化の促進とDI

- CalcMachineクラスを使ってみる

```
public class Sample {

  public static void main(String[] args) {
    Calculator calc = new CalcMachine();
    System.out.println(calc.multiply(10,100));
  }
}
```

*「Calculator」型の変数「calc」を宣言して、CalcMachineクラスの実体を「calc」に代入している*

実装だけを差し替えたことに注目!!

```
C:¥>java Sample
1000
```

# 部品化の促進とDI：ユニットテスト

- 実装クラスが間違っていたら困る！
  ⇒ 仕様を満たしていることの担保が必要 = ユニットテスト

■テストする実装クラス

```
public class CalcMachine implements Calculator {
  public int multiply(int source, int by) {
    return source * by;
  }
}
```

■ユニットテストのコード

```
public class CalcMachineTest extends TestCase {
    public void testMultiply() {
            CalcMachine calc = new CalcMachine();
            int ret = calc.multiply(10,100);
            assertEquals(1000, ret);
    }
}
```


# 部品化の促進とDI

- さらにやりたいことは・・・

いちいち書き換えなくても
差し替え可能にしたい！

```
public class Sample {
  public static void main(String[] args) {
    Calculator calc = [          ];
    System.out.println(calc.multiply(10,100));
  }
}
```

# DIContainerを使う

- コンポーネントの設定を外部に出す

```
public class Sample {
  public static void main(String[] args) {
    S2Container container =
        S2ContainerFactory.create(PATH);
    Calculator calc =
        (Calculator)container.getComponent("calc");
    System.out.println(calc.multiply(10,100));
  }
}
```

DIコンテナを起動する

コンテナからコンポーネントを取得する

登録されたクラスのメソッド名を指定して実行する

# DIContainerを使う

- 設定ファイル（.diconファイル）

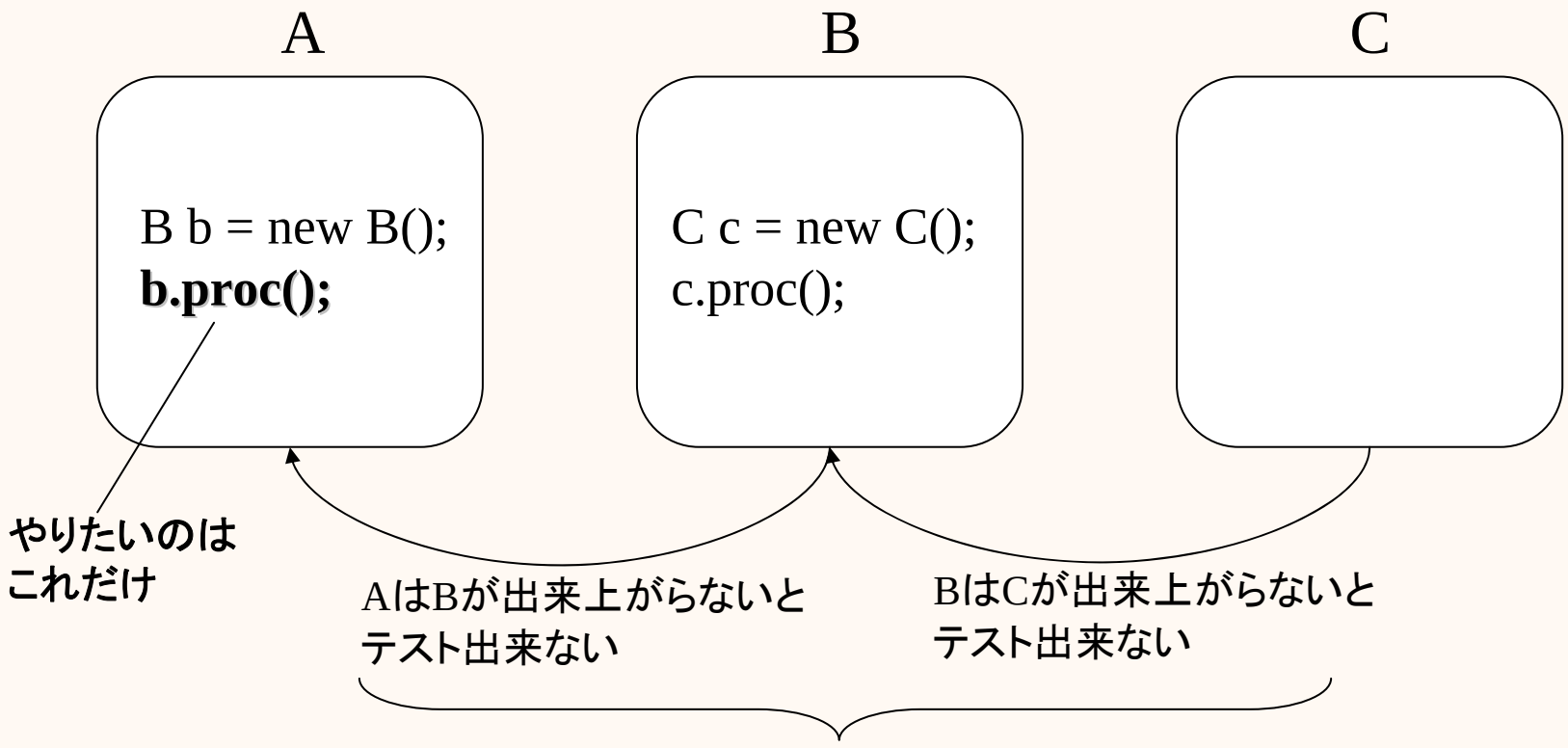
```
<?xml version="1.0" encoding="Shift_JIS"?>
<!DOCTYPE components PUBLIC "-//SEASAR//DTD S2Container//EN"
    "http://www.seasar.org/dtd/components.dtd">
<components>
    <component name= "calc" class= "CalcMachine">
    </component>
</components>
```

ここさえ書き換えればコードを変更せずに差し替え可能になる


# DIコンテナは何をする？

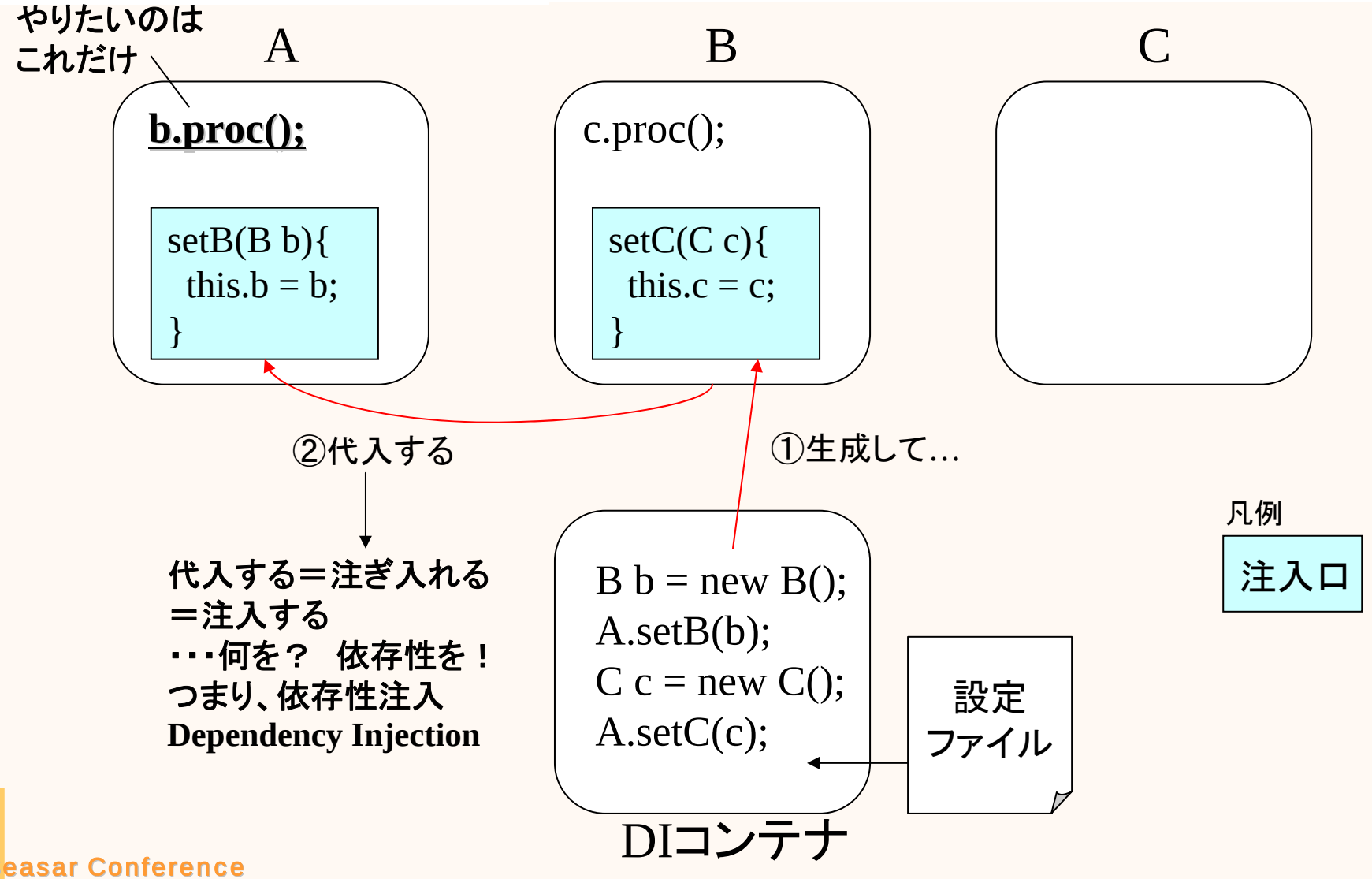
- Dependency Injection ＝ 依存性注入
  - 開発時における部品内の依存性を排除する。具体的には、
    - （1）newの排除（実装クラスに依存しない）
    - （2）変数へのインスタンス代入の排除
  - 実行時にコンテナが依存性を外部から注入する
    - （1）newの代わりにコンテナがインスタンスを生成
    - （2）部品内でインスタンスを指し示すためにコンテナが変数にインスタンスを代入
- それぞれのコンポーネントは、お互いのインターフェイスしか知らない。逆に言えば、**インターフェイス以外知らなくても構わない**
- DIの定義ファイルに書かれている内容に基づいて、DIコンテナが「勝手に」引き合わせてくれる
  - 実行時に、依存関係を動的に構築する
- **設定ファイルに従って、バラバラの部品を、仕様にあわせて実行時に組み合わせる仕組みがDIコンテナ**

# 従来の部品間の依存関係




部品化を進めるほど、開発時に待ちが生じて非効率となる。
・・・なので、ついついひとまとめにしたモジュールにしてしまう。
・・・・・すると、複雑になり品質確保が難しい。再利用性も損なわれる

# DIによる部品間の依存関係




# 再び、S2Container進化の歴史

- S2.0：DIコンテナとして登場
- S2.1～2.2：地道な改良
- S2.3：
  - CoCの影響
  - 規約ベースのコンポーネント自動登録を実現
  - 設定ファイルを不要に！
- S2.4：
  - Hot Deploy
  - SessionBeanへの対応

**S2をバンバン使ってると・・・**
**「設定ファイルうざいよね！」**
**「だったらなくしちゃえ！！」**

設定ファイルに、コンポーネントの記述をしなくても、規約に従ってコンテナがクラスを探してくれるようになった！

DI初心者には、いきなりだと何がなにやら・・・(^^;


# DIはinterface重視の手法

- 感じをつかむまでは、設定ファイルを書いてみたほうがいい
  - でも多分1時間もやれば、S2.3以降のありがたみを実感できるようになるはず
- 「Interfaceを先に決める」という設計・開発スタイルへの移行が鍵となる


# DIの最も重要なポイント

- インターフェイスと実装を分離し
  実装クラス同士が
  依存しあうことをなくす
- 得られるメリット
  - メンテナンス性の向上
  - 品質の向上
  - 分業による開発期間の短縮
  - 再利用性の向上(う～ん^^;)


# Thank you!

# ありがとう
# ございました