

# オブジェクト指向 エクササイズのススめ

12-A-6

菅野洋史/大村伸吾

株式会社オージス総研  
オブジェクトの広場

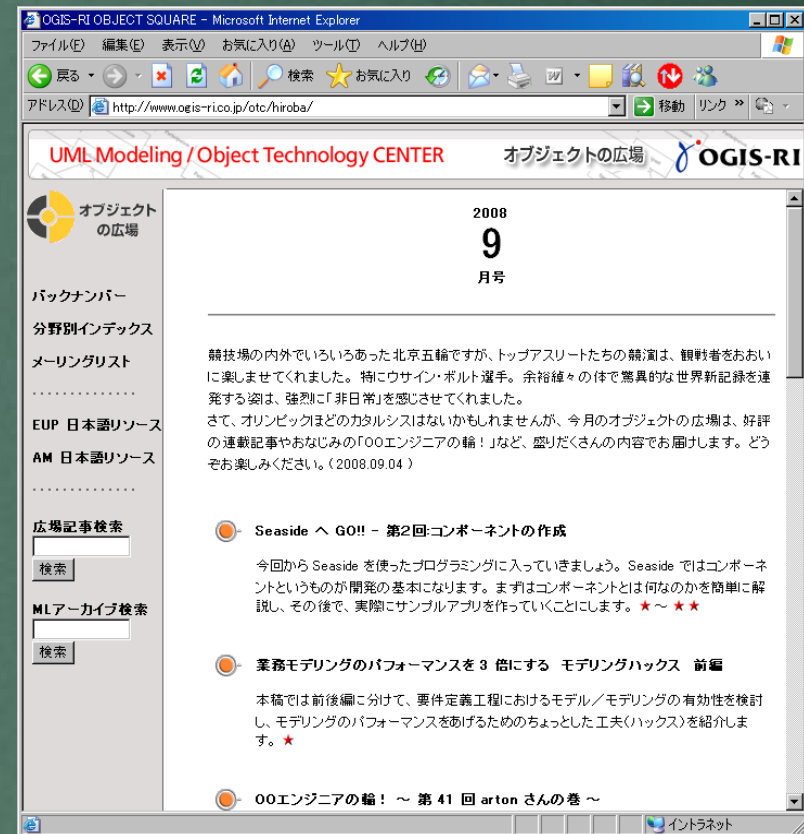
# 講演者のご紹介

- 株式会社オージス総研

## オブジェクトの広場編集部

- 月間のオンラインマガジン

- ThoughtWorksアンソロジーを  
翻訳しました!!



# ThoughtWorks アンソロジー

- ThoughtWorks社コンサルタントの

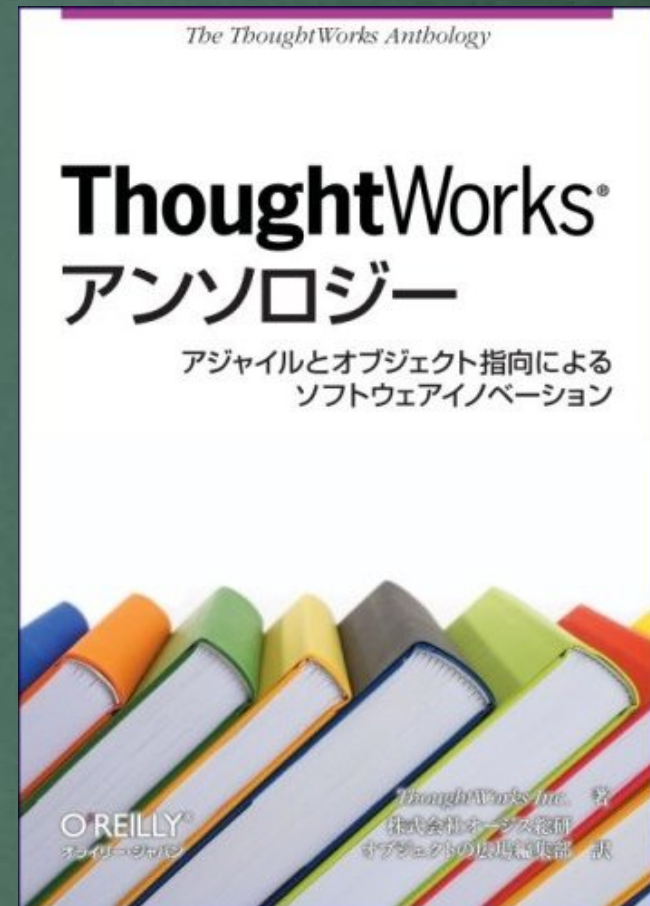
骨太なエッセイ集

- 様々な ジャンルを収録

DSL、プログラミング、設計、  
マネジメント、ビルド、デプロイ、テスト...

- オライリーさんブースで

絶賛販売中！



# はじめに


- このセッションではオブジェクト指向プログラミング(=設計)について語ります。
- エンジニアリングを考慮していない開発プロセスは失敗する  
<http://www.infoq.com/jp/news/2008/11/decline-of-agile>



ところで  
オブジェクト指向開発  
していますか？



# 本当にオブジェクト指向?

- オブジェクト指向言語使えばオブジェクト指向開発でしょうか?
  - 処理を全部、ロジッククラスに持たせてませんか?
  - 継承やインタフェース使えばオブジェクト指向だと言ってますか?
  - Struts (ry
- 

それは  
オブジェクト指向  
では無い





# オブジェクト指向でやるなら

責務を持ったオブジェクトが  
コラボレーションすることによって、  
複雑なシステムを構成すべき





# オブジェクト指向が出来ない理由

- 以前の慣習から抜け出すのが難しい
- 一部の開発者だけOOを分かっているればいいという風潮(特に設計やアーキテクチャ)
- 「難しい」
- 「マニアック」
- 「実践的じゃない」



必ずしも、  
オブジェクト指向は銀の弾丸じゃないが、  
武器は多い方が絶対にいい



# 教育と学習重要!

- 一部の開発者だけでなく、皆が知ればオブジェクト指向は武器になる
- 単なる耳学問では無く、体に叩き込む学習手段が欲しい



# そこで オブジェクト指向エクササイズ

オブジェクト指向プログラミングを強制的に身に  
着けるためにハードなコーディング規約を実際  
のプログラムに適用するエクササイズ



# 誰がやる？

- 開発の仕事には入って数年目の人
- ある程度、自分は出来ているという認識を持ってる人(の鼻っ柱を叩き折る)
- 最近、オブジェクト指向で開発していないなーというオブ厨の人



しばし、オブジェクト指向エクササイズの  
内容説明を・・・



## 9つのルール

- 1メソッドにつき1インデントまで
- プリミティブ型と文字列はラップする
- ファーストクラスコレクションを使用する
- 名前を省略しない
- else句は使わない
- すべてのエンティティ（要素）を小さく
- 1行につき1ドットまで
- Getter, Setter, プロパティを使用しない
- 1クラスにつきインスタンス変数は2つまで



- 1メソッドにつき1インデントまで
- プリミティブ型と文字列はラップする
- ファーストクラスコレクションを使用する
- 名前を省略しない
- else句は使わない
- すべてのエンティティ（要素）を小さく
- 1行につき1ドットまで
- Getter, Setter, プロパティを使用しない
- 1クラスにつきインスタンス変数は2つまで

- 1メソッドにつき1インデントまで
- プリミティブ型と文字列はラップする
- ファーストクラスコレクションを使用する
- 名前を省略しない
- else句は使わない
- すべてのエンティティ（要素）を小さく
- 1行につき1ドットまで
- Getter, Setter, プロパティを使用しない
- 1クラスにつきインスタンス変数は2つまで


- 1メソッドにつき1インデントまで
- プリミティブ型と文字列はラップする
- **ファーストクラスコレクションを使用する**
- 名前を省略しない
- else句は使わない
- すべてのエンティティ（要素）を小さく
- 1行につき1ドットまで
- Getter, Setter, プロパティを使用しない
- 1クラスにつきインスタンス変数は2つまで

- 1メソッドにつき1インデントまで
- プリミティブ型と文字列はラップする
- ファーストクラスコレクションを使用する
- 名前を省略しない
- else句は使わない
- すべてのエンティティ（要素）を小さく
- 1行につき1ドットまで
- Getter, Setter, プロパティを使用しない
- 1クラスにつきインスタンス変数は2つまで

# 省略したくなるのはこんな時

Before

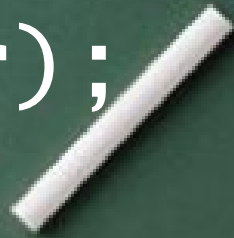
```
OrderService os = ...;  
os.shipOrderByOrderFromShopToCustomer(o.s,c);  
  
class OrderService{  
    void shipOrderByOrderFromShopToCustomer  
        (Order orderID,  
         String shopID,  
         String customerID)  
    {...}  
}
```



# 責務の配置を考え直せる

After

```
Shop shop = ...;  
Customer customer = ...;  
Order order = ...;  
order.ship(ship, customer);
```






- 1メソッドにつき1インデントまで
- プリミティブ型と文字列はラップする
- ファーストクラスコレクションを使用する
- 名前を省略しない
- **else句は使わない**
- すべてのエンティティ（要素）を小さく
- 1行につき1ドットまで
- Getter, Setter, プロパティを使用しない
- 1クラスにつきインスタンス変数は2つまで



# else句は使わない

Before


```
if (age < 20) {  
    doNotDrink();  
} else {  
    drink();  
}
```



# else句は使わない

After

```
if (age < 20) {  
    doNotDrink();  
    return;  
}  
drink();
```



- 1メソッドにつき1インデントまで
- プリミティブ型と文字列はラップする
- ファーストクラスコレクションを使用する
- 名前を省略しない
- else句は使わない
- **すべてのエンティティ (要素)  
を小さく**
- 1行につき1ドットまで
- Getter, Setter, プロパティを使用しない
- 1クラスにつきインスタンス変数は2つまで

- 1メソッドにつき1インデントまで
- プリミティブ型と文字列はラップする
- ファーストクラスコレクションを使用する
- 名前を省略しない
- else句は使わない
- すべてのエンティティ（要素）を小さく
- **1行につき1ドットまで**
- Getter, Setter, プロパティを使用しない
- 1クラスにつきインスタンス変数は2つまで

# 1行に付き1ドットまで

## Before

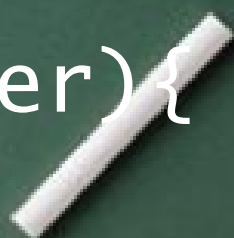
```
StringBuilder builder = ...;  
builder.append(omura.getName());
```

# 1行に付き1ドットまで

## After

```
StringBuilder builder = ...;  
omura.appendName(builder);
```

```
// in class Person...  
void appendName(  
    StringBuilder builder){  
    buf.append(name);  
}
```



- 1メソッドにつき1インデントまで
- プリミティブ型と文字列はラップする
- ファーストクラスコレクションを使用する
- 名前を省略しない
- else句は使わない
- すべてのエンティティ（要素）を小さく
- 1行につき1ドットまで
- Getter, Setter, プロパティを  
使用しない
- 1クラスにつきインスタンス変数は2つまで




- 1メソッドにつき1インデントまで
- プリミティブ型と文字列はラップする
- ファーストクラスコレクションを使用する
- 名前を省略しない
- else句は使わない
- すべてのエンティティ（要素）を小さく
- 1行につき1ドットまで
- Getter, Setter, プロパティを使用しない
- 1クラスにつきインスタンス変数は2つまで

1 クラスにつきインスタンス変数は2つまで

Before

```
class Person {  
    String firstName;  
    String lastName;  
    int age;  
}
```



1 クラスにつきインスタンス変数は2つまで

After

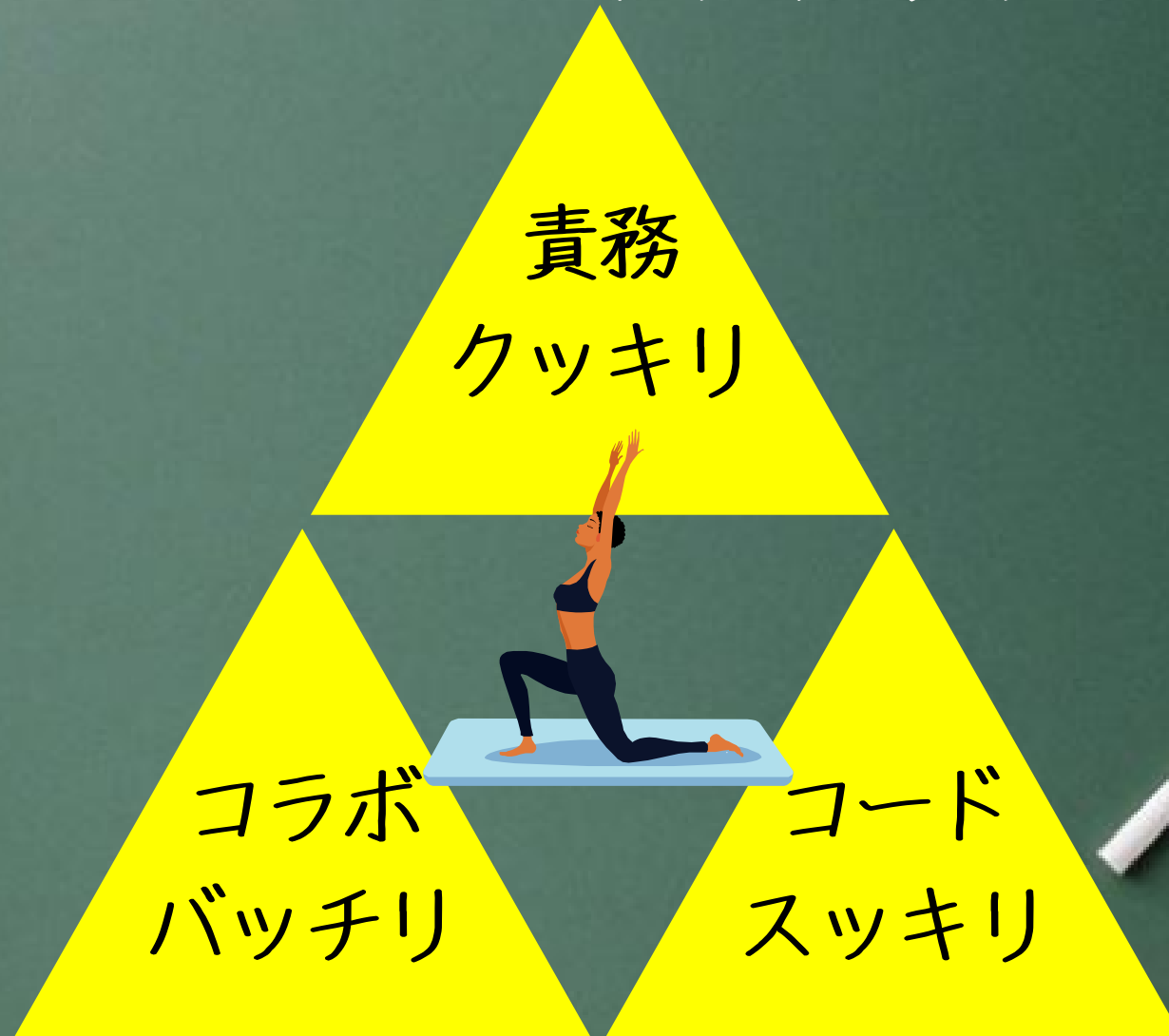
```
class Person {  
    Name name;  
    Age age;  
}
```

```
class Age {    int age;    }  
class Name {  
    String firstName;  
    String lastName;  
}
```

## 9つのルール

- 1メソッドにつき1インデントまで
- プリミティブ型と文字列はラップする
- ファーストクラスコレクションを使用する
- 名前を省略しない
- else句は使わない
- すべてのエンティティ（要素）を小さく
- 1行につき1ドットまで
- Getter, Setter, プロパティを使用しない
- 1クラスにつきインスタンス変数は2つまで

オブジェクト指向エクササイズで  
Let's シェイプアップ！



# オブジェクト指向エクササイズで スツキリ！ クツキリ！ バツチリ！

プリミティブ型はラップする

インスタンス変数  
は2つまで

1行につき1ドットまで

Getter, Setter  
を使用しない

エンティティ（要素）を小さく

責務  
クツキリ

ファーストクラス  
コレクションを使用

else句は使わない

1メソッド  
1インデントまで

名前を省略しない

コラボ  
バツチリ

コード  
スツキリ

実際にためしてみる






# 「お題」

一見00っぽい感じだけど、エクササイズの観点で見るとダメダメな、あるツールをエクササイズのルールに従うように書き直す。



# Amazon中古価格調査ツール

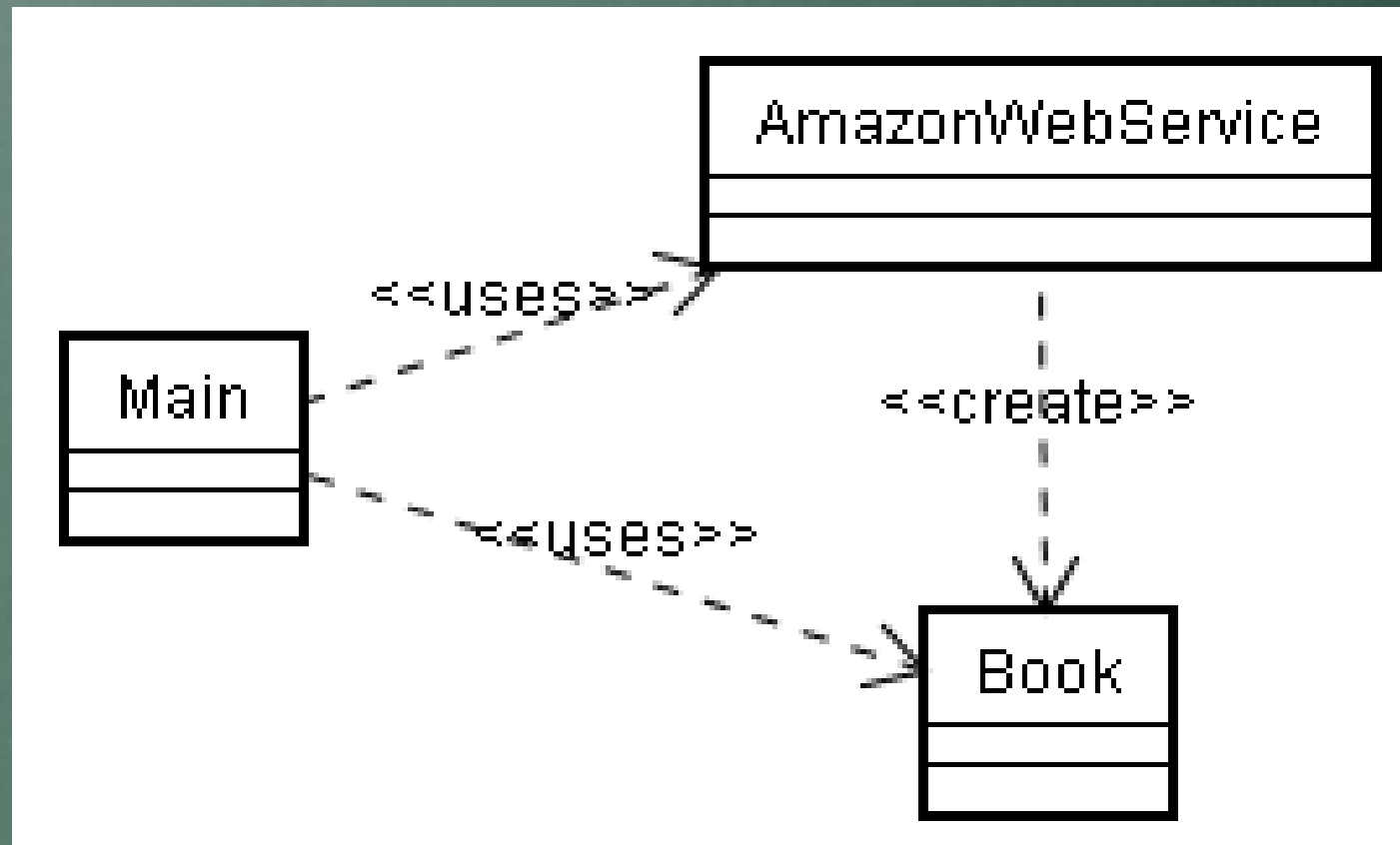
以下の仕様を持つ単純なコマンドラインツールを  
リファクタリング

- 引数でASINを与えて起動
  - AmazonマーケットプレイスをWebAPIで呼び出し、最も安い値段を検索する
  - 結果出力
- 

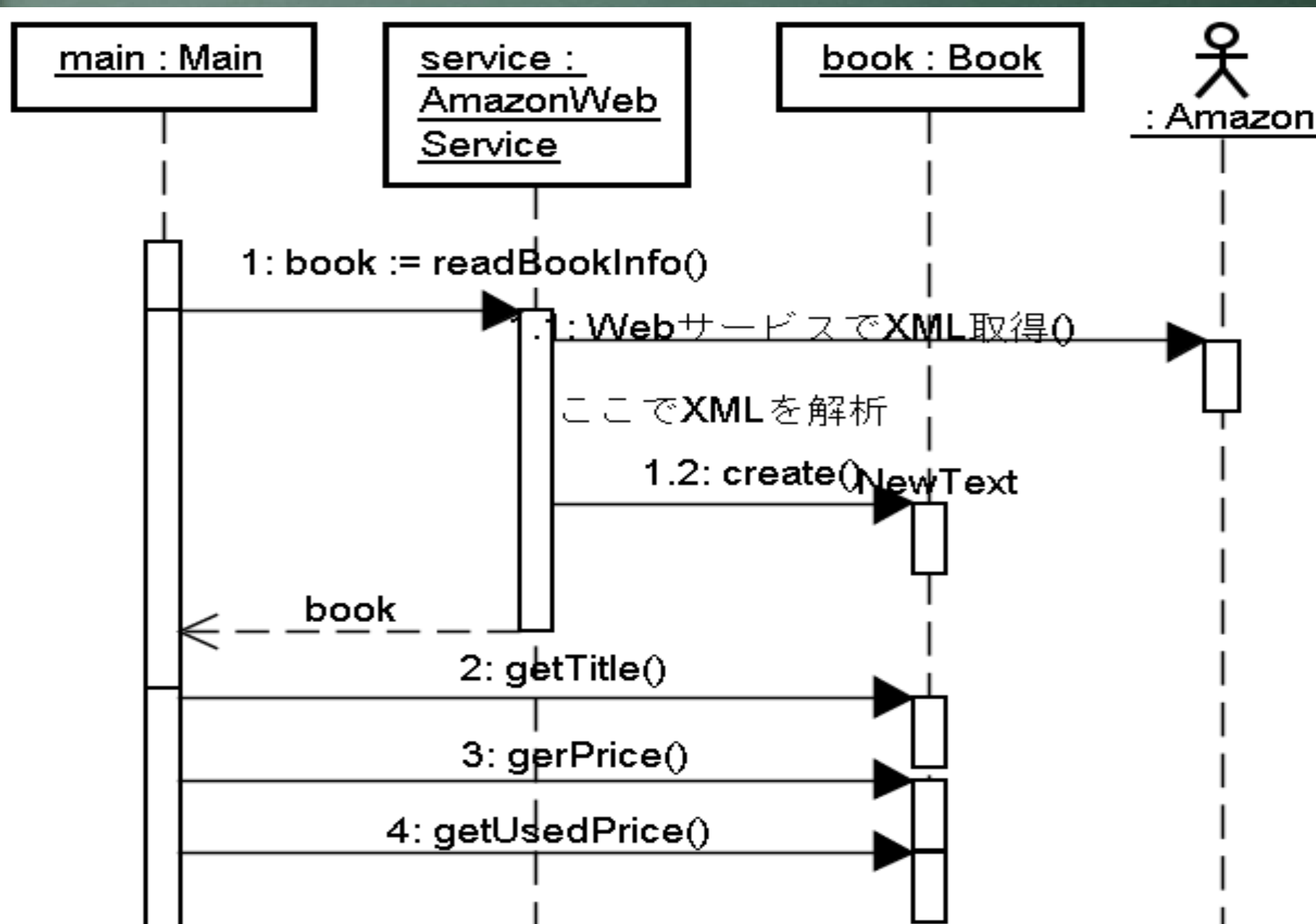
デモ



# 当初の構造



# シーケンス図



こんなもんでしょうか・・・

エクササイズの視点で見るとまだまだ余地がある  
まだ人間の限界じゃない!



# ルールの適用

今回は、9つ全部は時間が足りないので、二つの視点でルール群に注目する

- 責務を明確にする
- オブジェクト同士がきちんとコラボして動作するようなモデルにする





# 責務をクッキリさせる

- すべてのエンティティを小さくすること
- すべてのプリミティブ型と文字列型をラップすること
- 一つのクラスにつきインスタンスは二つまでにすること?



# Bookクラス

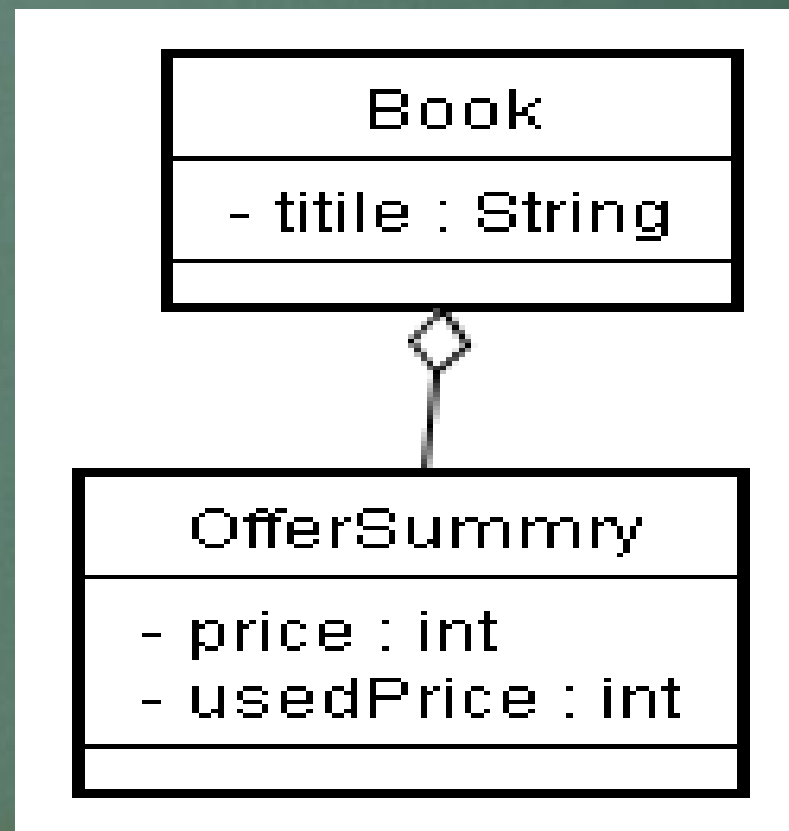
「属性3つ」 「priceがint」 「title」がString

Book
<ul style="list-style-type: none"><li>- title : String</li><li>- price : int</li><li>- usedPrice : int</li></ul>



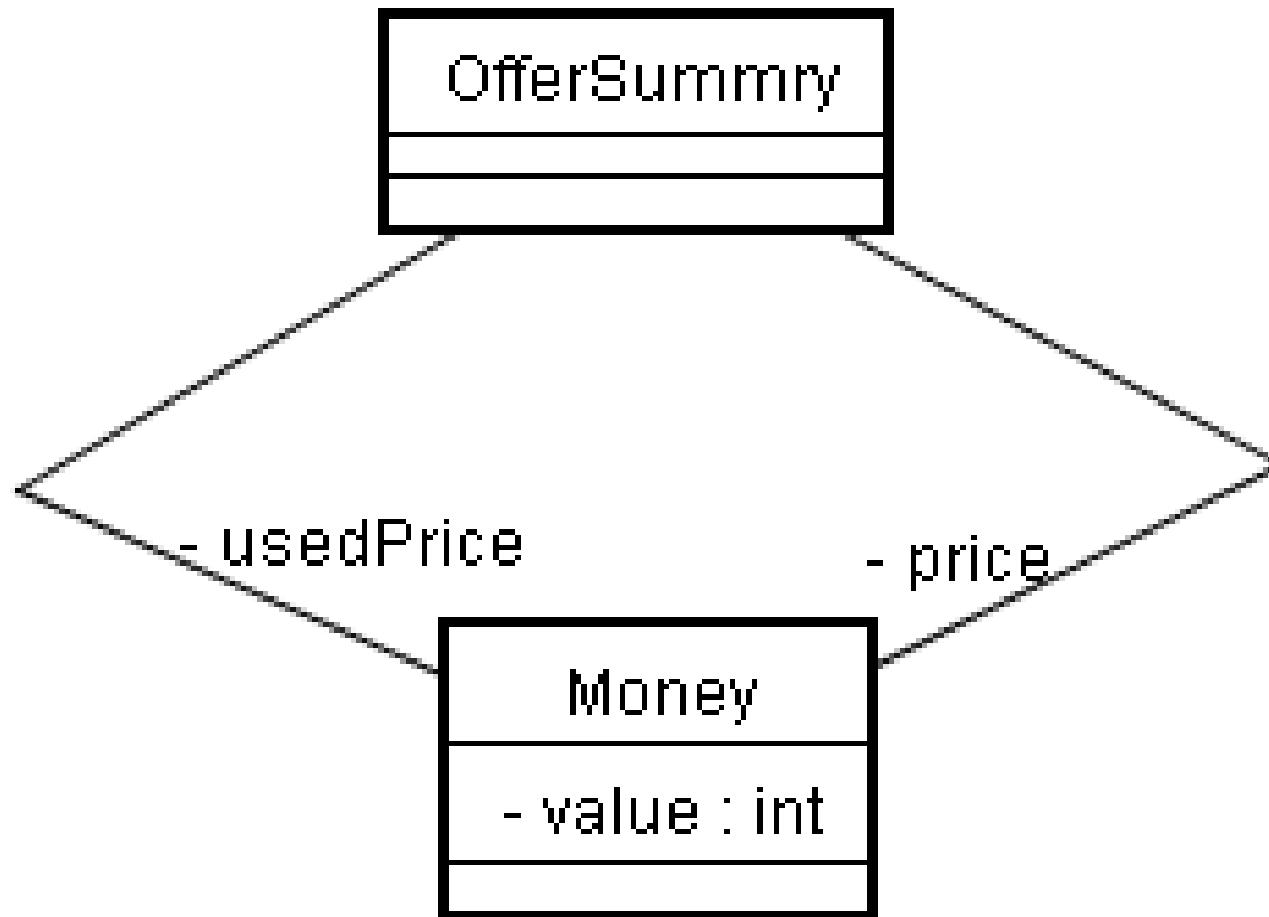
# 販売情報の抽出

- 「一つのクラスにつきインスタンスは二つまでにすること」



# Moneyによるラップ

「すべてのプリミティブ型と文字列型をラップ」



# コードで書くと

```
public class OfferSummary {  
    Money price;  
    Money usedPrice;
```

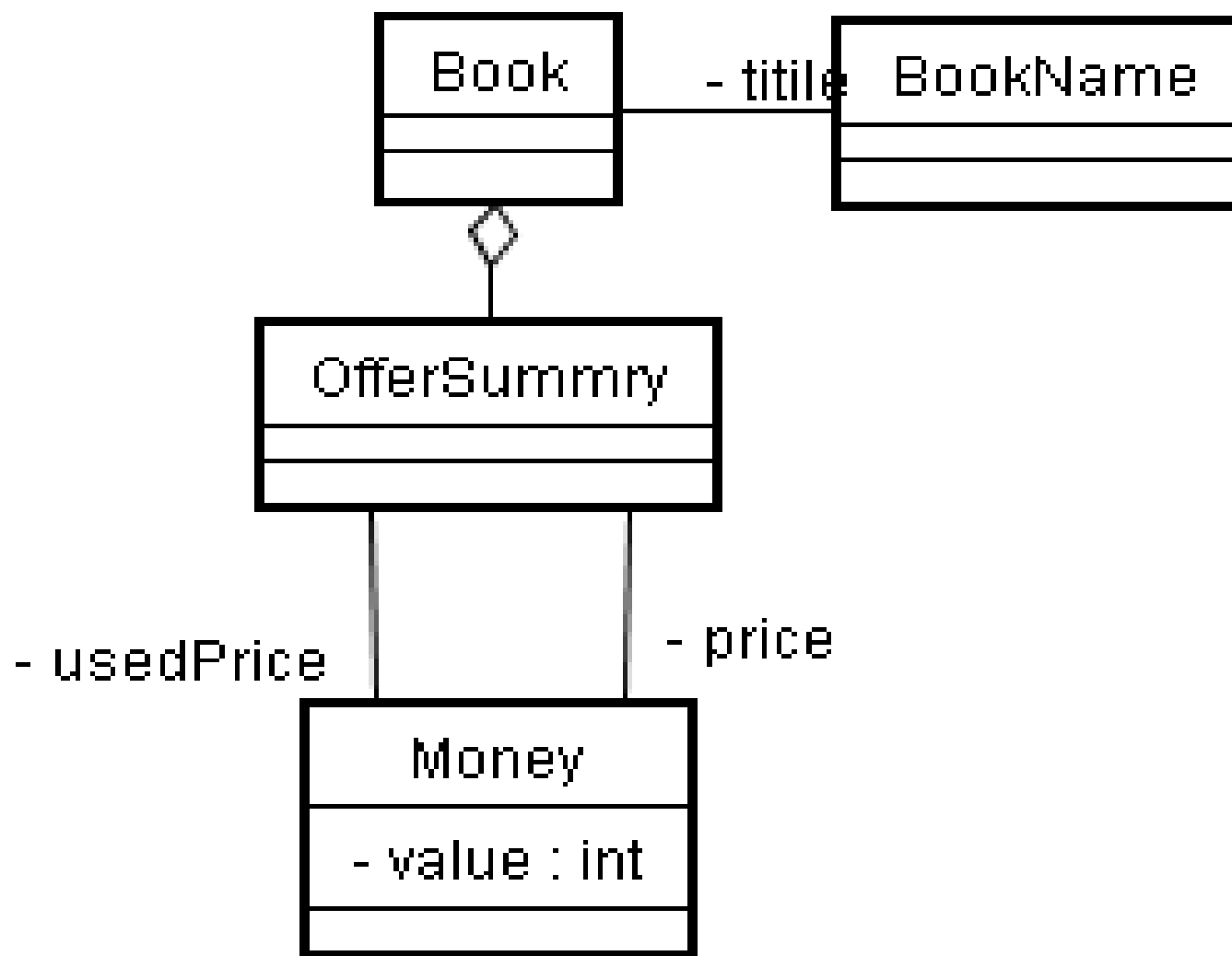
```
public class Monry {  
    int value:  
}
```



Bookクラスにあった「String title」  
にも同様の処理を施して・・・



# 最終的なクラス図





# 最初のBookをコードで書くと

```
public class Book {  
    String title;  
    int price;  
    int usedPrice;
```



# 最後のBookをコードで書くと

```
public class Book {  
    BookName title;  
    OfferSummary offerSummary;  
public class BookName {  
    String title;  
public class OfferSummry {  
    Money price;  
    Morny usedPrice;  
public class Money {  
    int price;
```



# 結果

- 「Book」が持っていた過剰な責務を適切に配分することができた。
  - MoneyのことはMoneyに
- 最初の時点では思いつかなかったクラスを抽出できた
  - 価格情報を扱うOfferSummary



# コラボレーションに関するルール

オブジェクトをより能動的にして、コラボレーションによってシステムの機能を実現する

## 関係するルール

- getter/setterを利用しない



# 理由

getter/setterを定義したクラスの中身は  
スカスカになるから

- 責務がgetter/setterを呼び出す側に偏る
- 本来は、Tell, Don't Ask! 「たずねるな! 命じよ!」



しかし!

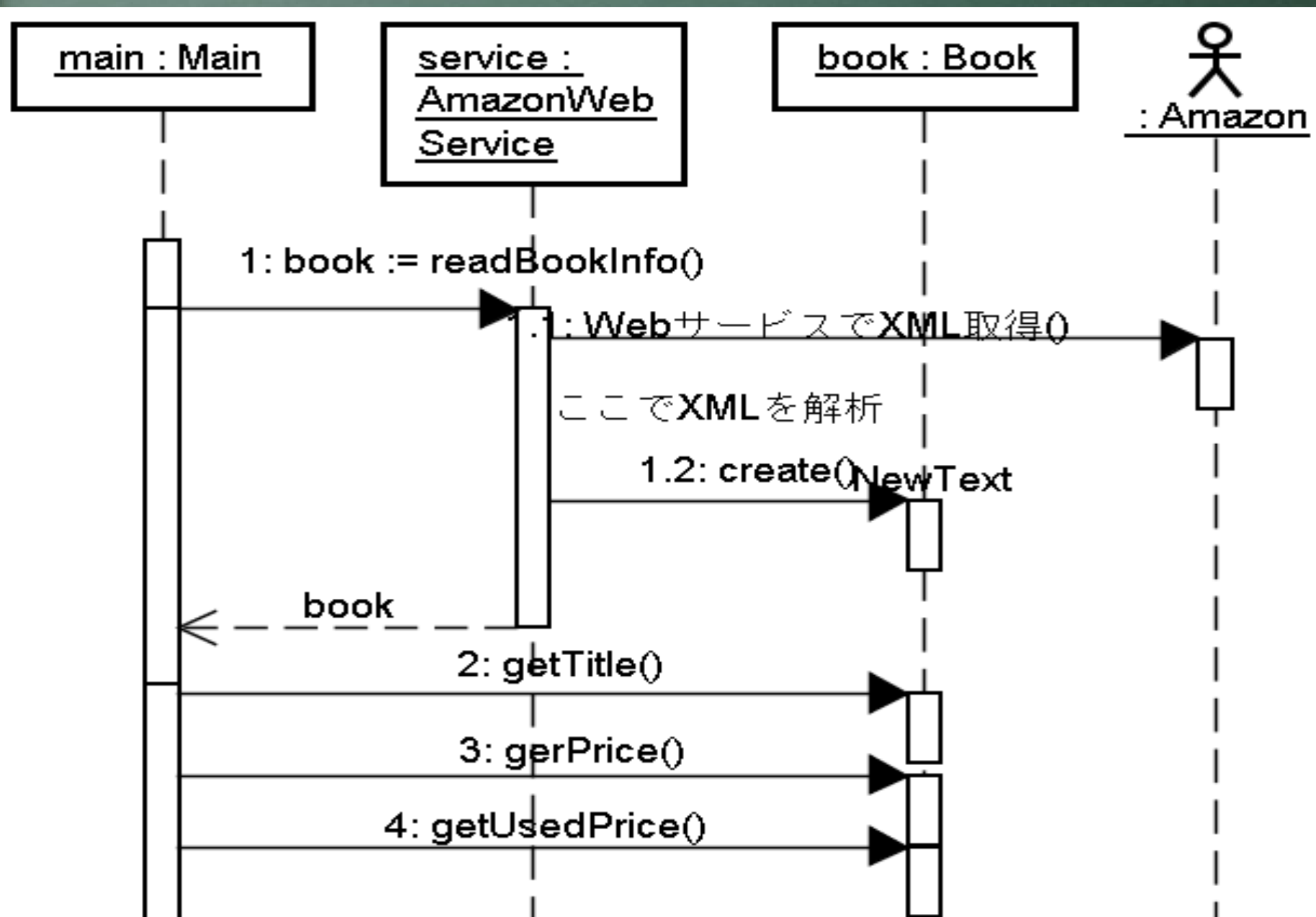
例えば画面表示をする場合、

Bookから情報取得しないとどうしようも無い

どうすればいい?



# もう一回シーケンス図





# 考えられる戦略

- 妥協する(エクササイズ的にはNG)
- DTO的戦略
- ダブルディスパッチ戦略
- etc . .



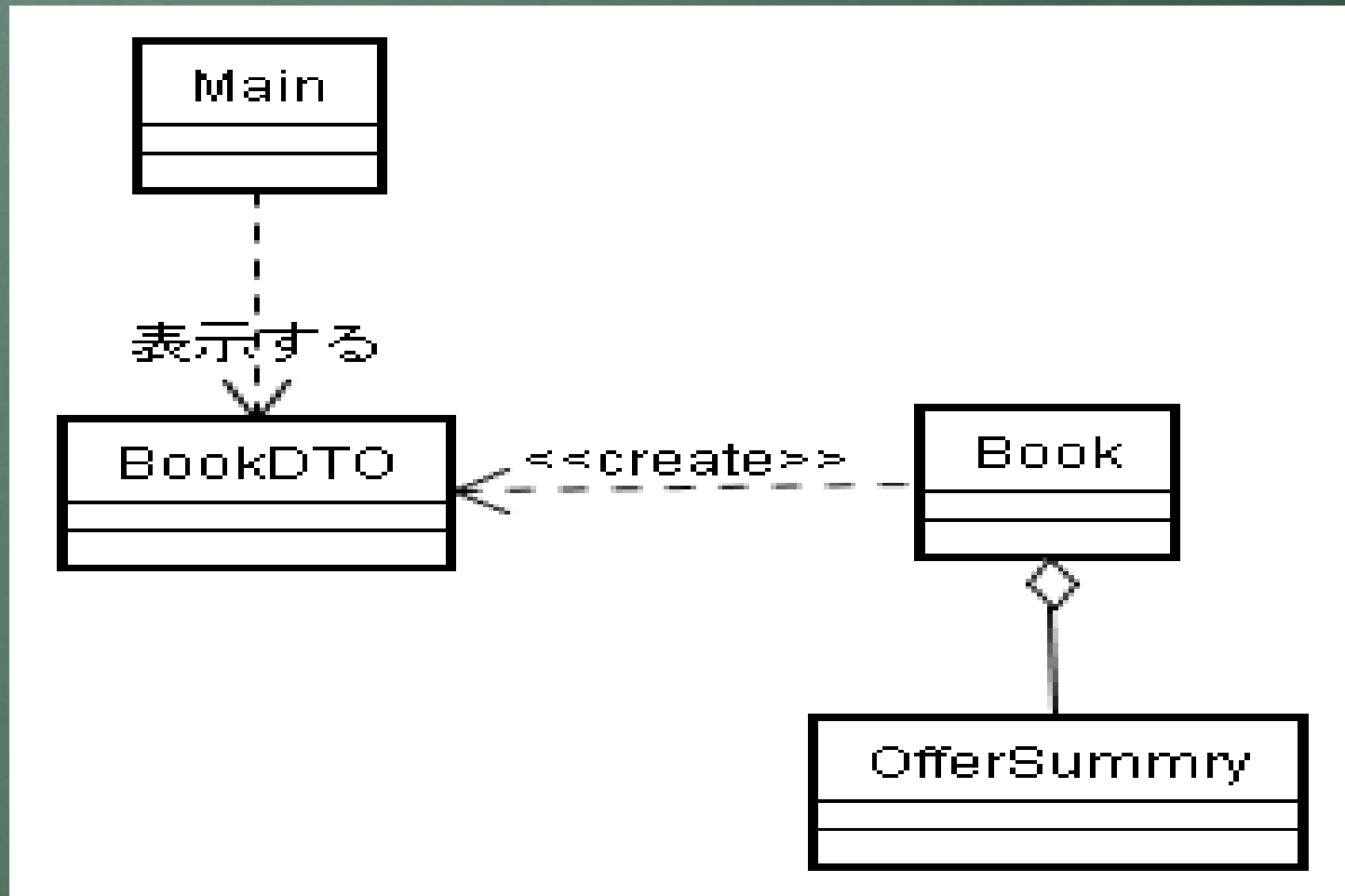
# DTO戦略

画面表示に関係する情報を  
一つの塊(DTO)にパックして  
表示側に返却する




# BookDTO

- 画面表示用のオブジェクトを、Bookクラスが作って返却する



# コード例

```
class Book {  
    public BookDTO represent() {  
        return  
            new BookDTO(  
                title, price(), usedPrice());  
    }  
}
```



```
class BookDTO {  
    String title;  
    String price;  
    String usedPrice;  
    //略
```

```
    public String toString() {  
        //いい感じに文字列編集して返却
```

# Mainで表示する

```
AmazonWebService service =  
    new AmazonWebService();  
Book book = service.readBookInfo(args[0]);  
BookDTO dto = book.represent();  
System.out.println(dto);
```



# 欠点

DTO自体が振る舞いをもっていない、  
エクササイズの目的と離れている

「getter/setterとどう違うの？」





# ダブルディスパッチ戦略

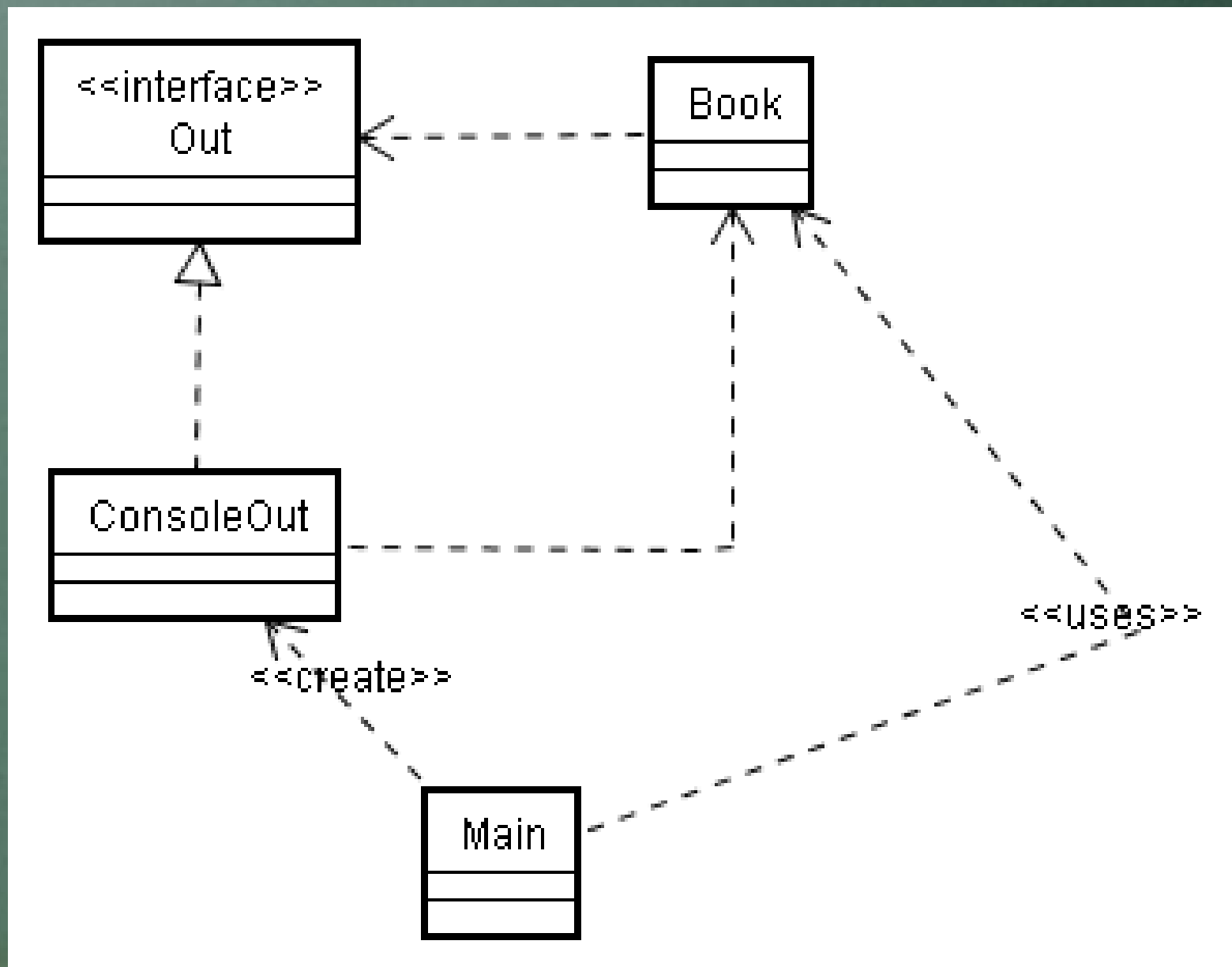
単にオブジェクトをメソッドで  
呼び出すだけではなく、  
相手から呼び返してもらう



- 画面が表示用オブジェクトを作成
- Bookに引き渡す
- それを受け取ったBook側で、表示オブジェクトに出力メッセージを送る(呼び返す)
- 表示用オブジェクトが画面に出力する

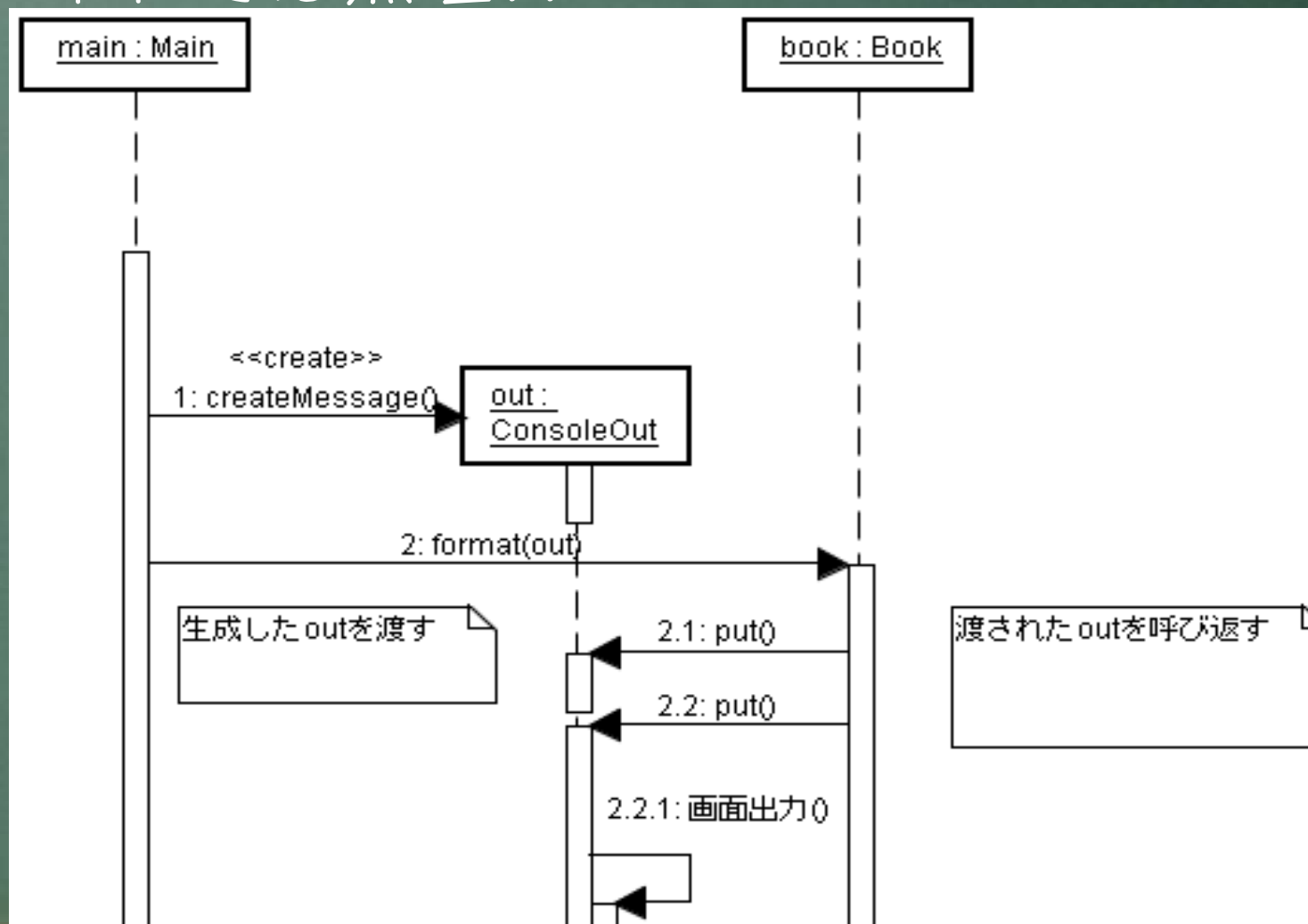
このようにBook側が能動的に動く

# クラス図



# シーケンス図

- スライドでは無理!!



# コード例:Book側

```
public void format(Out<String> out) {  
    out.put("タイトル", title);  
    out.put("価格", price());  
    out.put("中古価格", usedPrice());  
}
```

outというオブジェクトが

formatメソッドに飛んできたら、


出力したいオブジェクトをputする。

どこに出力されるかは知らないが



# コード例:画面表示用オブジェクト

```
interface Out<VALUE> {  
    void put(String attr , VALUE obj);  
}  
  
class ConsoleOut implements Out<String> {  
    public void put(  
        String attr, String value) {  
        System.out.printf(  
            "%s %s %n" , attr, value);  
    }  
}
```



# コード例:画面表示用オブジェクト

- BookクラスはOutというインターフェースのputメソッド呼び出すと、何らかの形で自分の情報を外部に出力できる事だけを知ってる
- ConsoleOutクラスはOutインターフェースを実装し、標準出力に情報を出す





# コード例:Main

```
AmazonWebService aws = new AmazonWebService();  
Book book = aws.readBookInfo(args[0]);  
Out out = new ConsoleOut();  
book.format(out);
```

BookをAmazonWebServiceから取得して、  
そのBookに出力を依頼している



# 効果

- 画面側からモデル側の情報を一方的に引き出すという関係が消えた
- 責務が配分され、オブジェクトのコラボレーションにより機能が実現されている



# その他の設計案

- Bookクラス自体に表示能力を持たせる
  - Squeak派
- 単純にBookクラスのtoString()を実装する
  - Java王道派
- getterという名前じゃないんだけど、同じ機能のメソッドを用意する
  - ひたすらずるい

# 実は、、正解は無い

その場の「制約」をどうバランスさせるか  
判断することが重要

- このような設計時の判断の集積はアーキテクチャになり、チームのコモンセンスになる
- エクササイズによって  
議論が引き出される効果がある



さらにエクササイズを進める



# AmazonWebService

AmazonWebServiceには責務が3つある

- Amazon本家へのアクセス
- XMLの解析
- Bookオブジェクトの生成



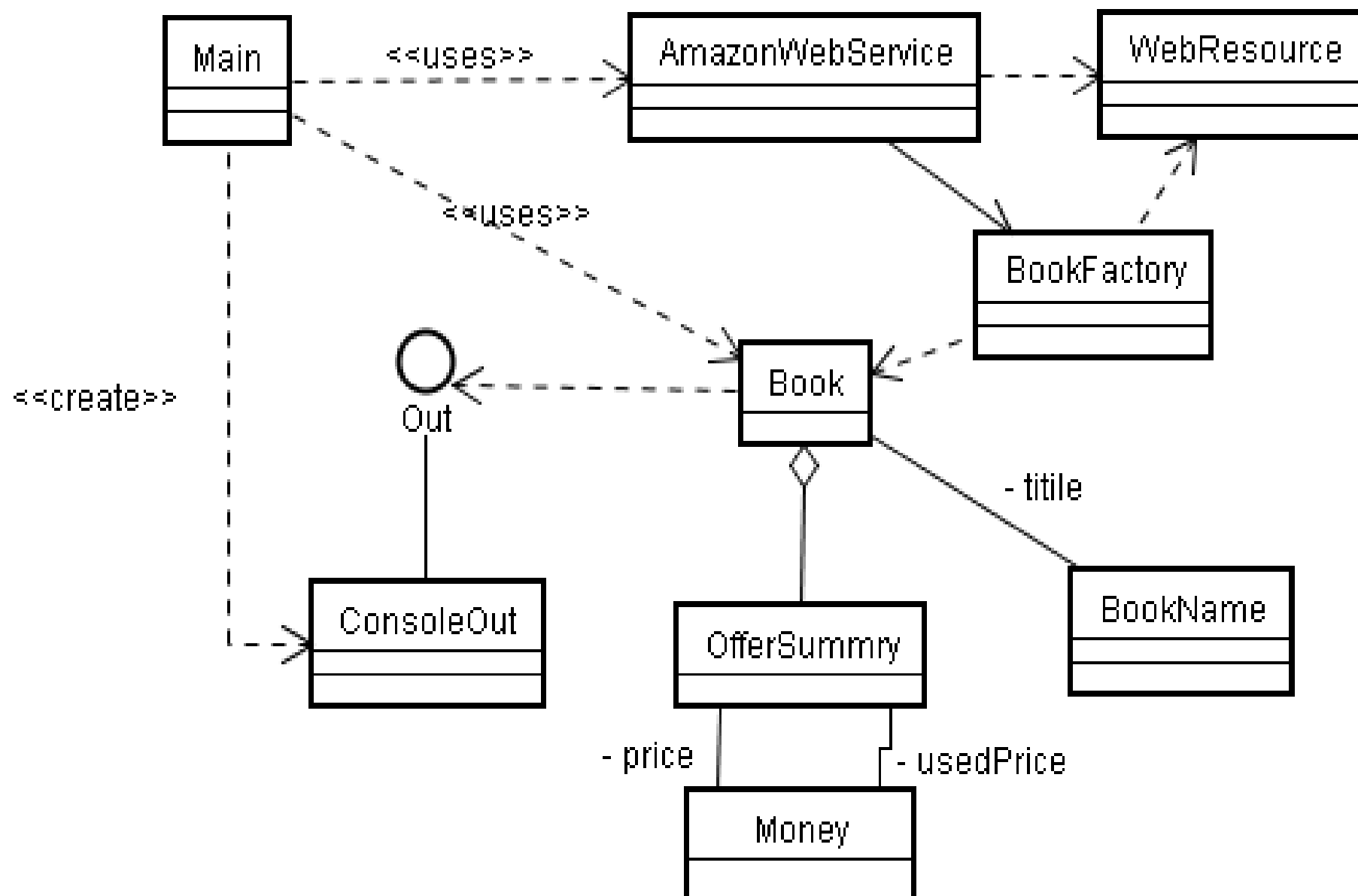
# 「エンティティを小さく!」

- AmazonWebService
- BookFactoryの分離
- WebResourceクラスの抽出





# 最終的なクラス図



# 過剰設計じゃない？

- エクササイズは設計手法そのものではない
- 著者はこれで実プロジェクトを行っているが・・・



# 演習のまとめ

責務が分割されて、オブジェクト間のコラボレーションにより機能が実現されるようになった。

学習効果としては、

- 惰性で行っていた設計の見直しが出来た
- 制約が設計に影響することを実感できた
- いろいろなテクニックを引き出した

最後に



# エクササイズで壁を超えよう

- とにかくきつい!
- あらゆるテクニック、知識を総動員しろ!
- 議論を巻き起こせ!



なお、効果には個人差があります



# スペシャルサンクス

- Jef Bay氏  
(原著者/ThoughtWorks社テクノロジープリンシパル)
- 宮川 直樹さん(オライリー・ジャパン)
- 村上 未来 氏(翻訳)
- オブジェクトの広場編集委員(佐藤さん, 田中くん, 辻くん, 山内くん, 山野さん)
- 書評をくださった皆様