



WORKING EFFECTIVELY WITH LEGACY CODE

About me

- I'm River
- 正在學習 DDD
- 後端技術
- 及想變的更敏捷



主題

- Ch6 時間緊迫，但必須修改
- Ch7 漫長的修改時間

真的超急的，快幫我改



我現在就要 RIGHT NOW!!

來自老闆 / 客戶 的壓力

面對壓力之下的策略

對舊有程式碼以最少的方式變動

撰寫新需求並進行測試

還記得 legacy code 的定義？

遺留程式碼就是從其他人那裡得來的程式碼??

無法理解，難以修改的程式碼?? legacy code is simply code without tests

但修改只要 15 分鐘，但編寫測試卻要 2
個小時 ??

請小心，好好的寫測試才是王道

6.1 新生方法 Sprout Method

場景：某知名電商辦公室-行銷部

- 行銷 A：唉，今天訂單爆量耶，爽
- 行銷 B：真的耶，怎麼比昨天多了快一倍
- 行銷 A：怪了，今天又沒有行銷活動
- PM：[開始檢查訂單] - 完蛋了，怎麼這麼多重複訂單，快跟工程師講

場景：某知名電商辦公室-工程部

- PM: 挫塞，今天超多重複訂單
- PM: 這個應該很簡單，快動動你的小指頭。
- 工程師 S : ...



找到有問題的程式碼

```
public class TransactionGate
{
    public void postEntries(List entries) {
        for (Iterator it = entries.iterator(); it.hasNext();
            Entry entry = (Entry)it.next();
            entry.postDate();
        }
        transactionBundle.getListManager().add(entries);
    }
    ...
}
```

暴力解

```
public class TransactionGate
{
    public void postEntries(List entries) {
        List entriesToAdd = new LinkedList();
        for (Iterator it = entries.iterator(); it.hasNext();
        Entry entry = (Entry)it.next();
        if (!transactionBundle.getListManager().hasEntry(
            entry.postDate();
            entriesToAdd.add(entry);
        }
    }
    transactionBundle.getListManager().add(entriesToAdd);
}
...
}
```

改了那些？

重複項目的檢查

```
1 public class TransactionGate
2 {
3     public void postEntries(List entries) {
4+         List entriesToAdd = new LinkedList();
5         for (Iterator it = entries.iterator(); it.hasNext(); ) {
6             Entry entry = (Entry)it.next();
7+             if (!transactionBundle.getListManager().hasEntry(entry)) {
8                 entry.postDate();
9+                 entriesToAdd.add(entry);
10            }
11+        }
12+        transactionBundle.getListManager().add(entriesToAdd);
13    }
14    ...
15 }
```

比較好一點的解： 加了新生方法 uniqueEntries

```
public class TransactionGate
{
    ...
    List uniqueEntries(List entries) {
        List result = new ArrayList();
        for (Iterator it = entries.iterator(); it.hasNext();
            Entry entry = (Entry)it.next();
            if (!transactionBundle.getListManager().hasEntry(
                result.add(entry);
            }
        }
        return result;
    }
}
```

原有程式沒有進行太大的修改。

```
public void postEntries(List entries) {  
    //*****  
    List entriesToAdd = uniqueEntries(entries);  
    //*****  
  
    for (Iterator it = entriesToAdd.iterator(); it.hasNext(); ) {  
        Entry entry = (Entry)it.next();  
        entry.postDate();  
    }  
    transactionBundle.getListManager().add(entries);  
}  
...  
}
```

把更新時間及檢查重複的程式碼分離了

Steps:

1. 確定修改點
2. 在原方法中寫一個假的呼叫，並把他註解掉
3. 確定你需要原方法中的那些局部變數，並把他作為參數傳給新方法。
4. 確定是否需要回傳什麼值給原方法
5. 使用測試驅動的方式來開發
6. 將原方法中被註解掉的呼叫重新生效

如果還是不行？

- 依賴關係太惡劣的時後
- 採用 Pass Null (p.123 p.141) 的方式
- 再不行還可以用 static function 都在解決難
以在測試工具中建立 Instance 的問題

Advantages

- 新舊程式碼被清楚的隔離
- 新舊程式碼中有清楚的介面

Disadvantages

- 放棄了原方法及所屬類別，暫時不打算寫測試
且改善他們
- 後人會不理解，為何一個小小的功能要另外寫
一個新生方法來實現

6.2 新生類別

- 依賴關係過於複雜
- 無法在合理的時間內將類別在測試控制工具 (Test harness) 被實例化。
- 在原有的程式中加入全新的職責

下面有一段產生報表的程式
現在要幫這個報表加入表頭，該如何進行？

```
class QuarterlyReportGenerator {
  ...
  generate(beginDate: string, endDate: string) {
    const reportRule6 = new ReportRule6();

    const results: Array<Result> = database.queryResults(beginDate, endDate);
    let pageText: string = "<html><head><title>"

      "Quarterly Report"
      "</title></head><body><table>";

    if (results.length != 0) {
      for (const it of results){
        pageText += "<tr>";
        pageText += "<td>" + it.department + "</td>";
        pageText += "<td>" + it.manager + "</td>";


  
```

用 Sproud Class 試試

Demo Code

steps

1. 確定修改點
2. 在原方法中建立一個類別的物件，並呼叫它的方法，並把他註解掉
3. 確定你需要原方法中的那些局部變數，並把他作為參數傳給新方法。
4. 確定是否需要回傳什麼值給原方法
5. 使用測試驅動的方式來開發
6. 將原方法中被註解掉的呼叫重新生效

來自同事們的挑戰

- 這個類別也太好笑了吧，這麼小？
- 設計上也沒有好處？
- 又引入全新概念，變的更亂。



Advantages

- 進行侵入性比較強的修改時更有自信。
- 利用抽象化的 interface, 將會有機會把大部份的工作移到有被測試過的新生類別中
- 一個系統的特異點，刺激思考

Disadvantages

- 概念複雜化
- 破壞 programmer 對 code base 中的認知

6.3 外覆方法

- 在沒有接縫的情況下
- Sproud Method 的另一種選擇

在員工計算薪水的程式中加入日誌記錄

```
public class Employee {  
    ...  
    public void pay() {  
        Money amount = new Money();  
        for (Iterator it = timecards.iterator(); it.hasNext();) {  
            Timecard card = (Timecard)it.next();  
            if (payPeriod.contains(date)) {  
                amount.add(card.getHours() * payRate);  
            }  
        }  
        payDispatcher.pay(this, date, amount);  
    }  
}
```

Wrap Method - 一之型

- 建立一個與原方法同名的新方法，並在新方法中呼叫更名後的原方法
- 外部的呼叫無需進行修改便可添加新行為

demo

steps:

1. 確定要修改的方法(method)並將舊方法改名
2. 建立新方法，要注意新方法的簽章(Signatures)
要與舊方法的相同
3. 在新方法中呼叫更名後的舊方法
4. 把要添加的新特性編寫一個方法，並在第 2 步
中建立的方法中進行呼叫(要以 TDD 的方式進行
開發)

Wrap Method - 二之型

- 新增一個無人呼叫的新方法
- 修改舊程式呼叫的對象

demo

steps:

1. 確定要修改的方法(method)
2. 把要添加的新特性編寫一個方法(要以TDD 的方式進行開發)
3. 建立另一個新方法來呼叫新舊兩個方法

Kent Beck 說

Now users have the option of paying in either way. It was described by Kent Beck in *Smalltalk Patterns: Best Practices* (Pearson Education, 1996).

Wrap Method is a great way to introduce seams while adding new features. There are only a couple of downsides. The first is that the new feature that you add can't be intertwined with the logic of the old feature. It has to be something that you do either before or after the old feature. Wait, did I say that is bad? Actually, it isn't. Do it when you can. The second (and more real) downside is that you have to make up a new name for the old code that you had in the method. In this case, I named the code in the pay() method dispatchPayment(). That is a bit of a stretch, and, frankly, I don't like the way the code ended up in this example. The dispatchPayment() method is really doing more than dispatching; it calculates pay also. If I had tests in place, chances are, I'd extract the first part of dispatchPayment() into its own method named calculatePay() and make the pay() method read like this:

dispatchPayment ??

Advantages

- 與新生方法比起來外覆方法並不會增加舊方法的體積
- 很明顯的讓新功能獨立於既有功能

Disadvantages

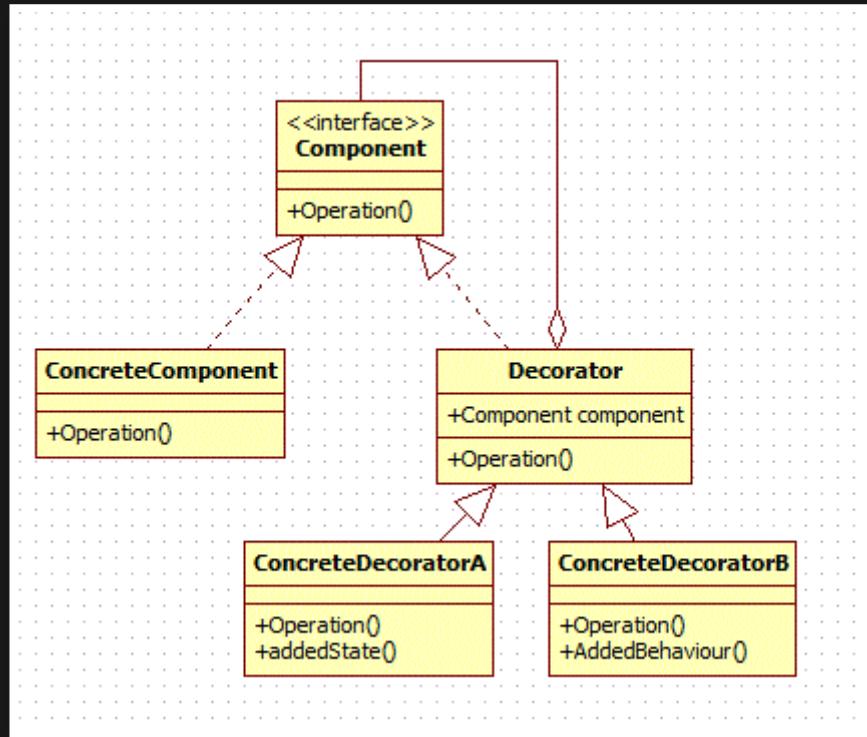
- 有可能會有糟糕的命名

外覆類別 (Wrap Class)

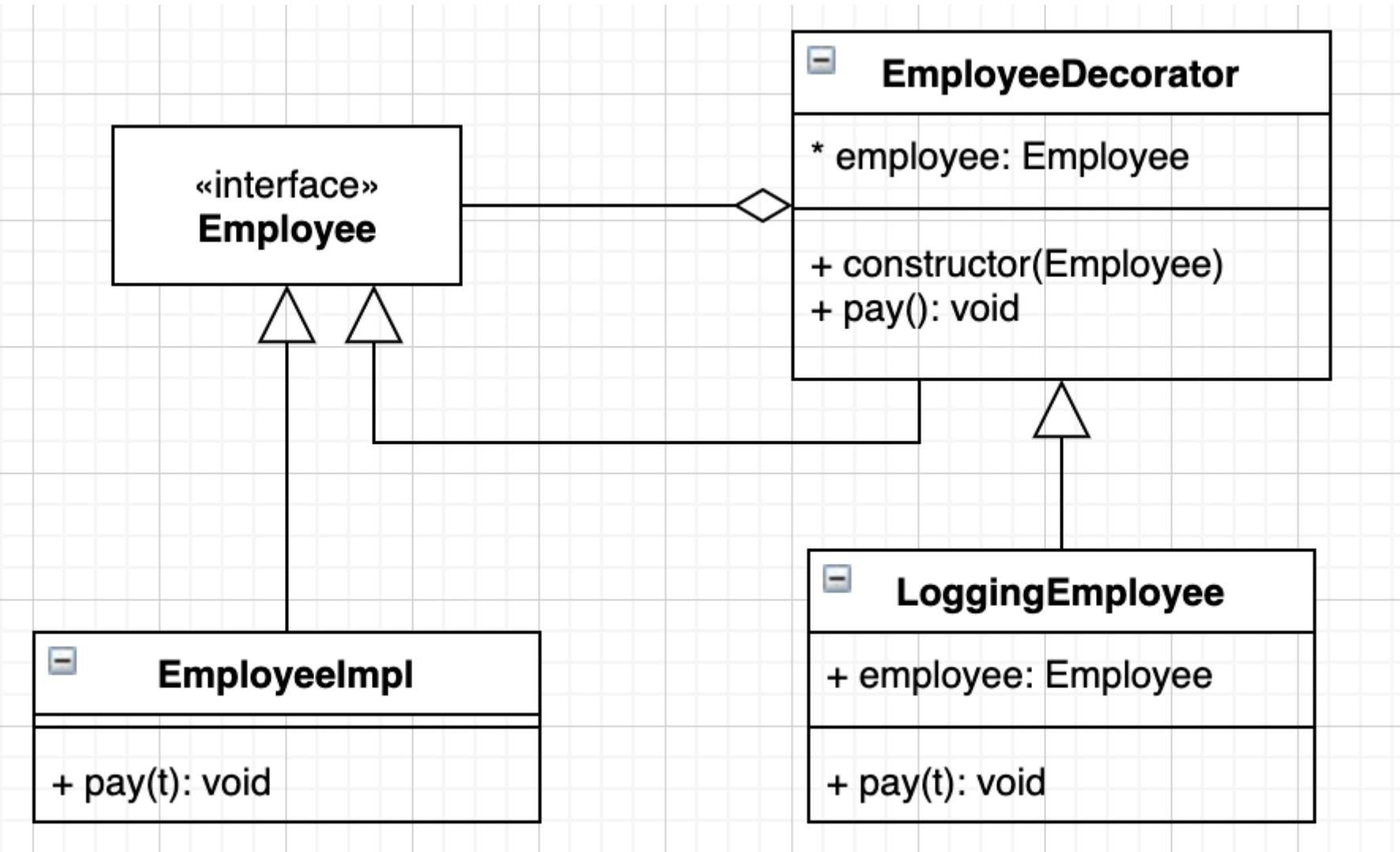
- 概念上跟 Wrap method 幾乎一模一樣
- 原類別已經太大了
- 要在測試中實例化原類別的成本太高

```
public class Employee {  
    ...  
    public void pay() {  
        Money amount = new Money();  
        for (Iterator it = timecards.iterator(); it.hasNext(); ) {  
            Timecard card = (Timecard)it.next();  
            if (payPeriod.contains(date)) {  
                amount.add(card.getHours() * payRate);  
            }  
        }  
        payDispatcher.pay(this, date, amount);  
    }  
}
```

裝飾模式 decorator pattern



from www.codeproject.com



demo code

呼叫的型式會變成

```
Employee employee  
= new BuddyEmployee(  
    new LoggingEmployee(  
        new EmployeeImpl()  
    ) );
```

比較不 Decorator 的方式

```
1 class LoggingPayDispatcher
2 {
3     private Employee e;
4     public LoggingPayDispatcher(Employee e) {
5         this.e = e;
6     }
7     public void pay() {
8         employee.pay();
9         logPayment();
10    }
11    private void logPayment() {
12        ...
13    }
14    ...
15 }
```

呼叫的型式會變成

```
const employee = new Employee();
const dispatcher = new LoggingPayDispatcher(employee);
dispatcher.pay();
```

steps

1. 確定修改點
2. 建立新類別，Extract Implementation(p.372) or Extract Interface(p.362)
3. 在新的外覆類別中編寫新的方法去實作想要新增的功能，另外再寫一個方法，去執行剛剛的新方法，並呼叫舊的。
4. 在系統中需要新行為的地方進行修改

方法

使用時機

Sprout
Method

- 如果現行的方法已經可以清楚的表達它的功能時

Sprout
Class

- 如果現行的方法已經可以清楚的表達它的功能
- 原類別在建立時有大量的依賴，無法在測試中實例化。

方法

使用時機

Wrap
Method

- 添加新特性的重要性與原方法不相上下時
- 所要新增的功能適合放在原類別中

Wrap
Class

- 添加的行為是完全獨立的，不想要讓不相關的行為污染到現有類別
- 原類別已經太大了

阻力

這麼小的功能，卻要用一個新的 class / method 來實作？

雖小，但好切入 一天一點讓自己變更好

- 勿以惡小而為之，勿以善小而不為
- 對於現況的不滿足
- 想要把舊程式碼變的更好!!

當真的要面對 Lagecy Code 的時後

Ch9, I Can't Get This Class into a Test Harness

Ch20, This Class Is Too Big and I Don't Want It to
Get Any Bigger, are good places to start.

Ch 7. 漫長的修改

造成修改需要大量時間的原因

- 理解程式碼
- 時滯(Lag Time)

理解程式碼

因為要理解舊程式碼而需要花上大量的時間

Ch16, I Don't Understand the Code Well Enough to
Change It

Ch17, My Application Has No Structure, to get
some ideas about how to proceed.

時滯

時滯的影響 - 大腦的特性

愈短的等待時間，能夠有愈快的反應

愈小的 lag time 有愈好的效率

造成時滯(lag time)的原因

每次的 compile 都要花上大量的時間

單元測試過於肥大

如何減少時滯？

- 減小 package 的大小
- 換用效能較高的設備

解依賴

The Dependency Inversion Principle

依賴反轉原則

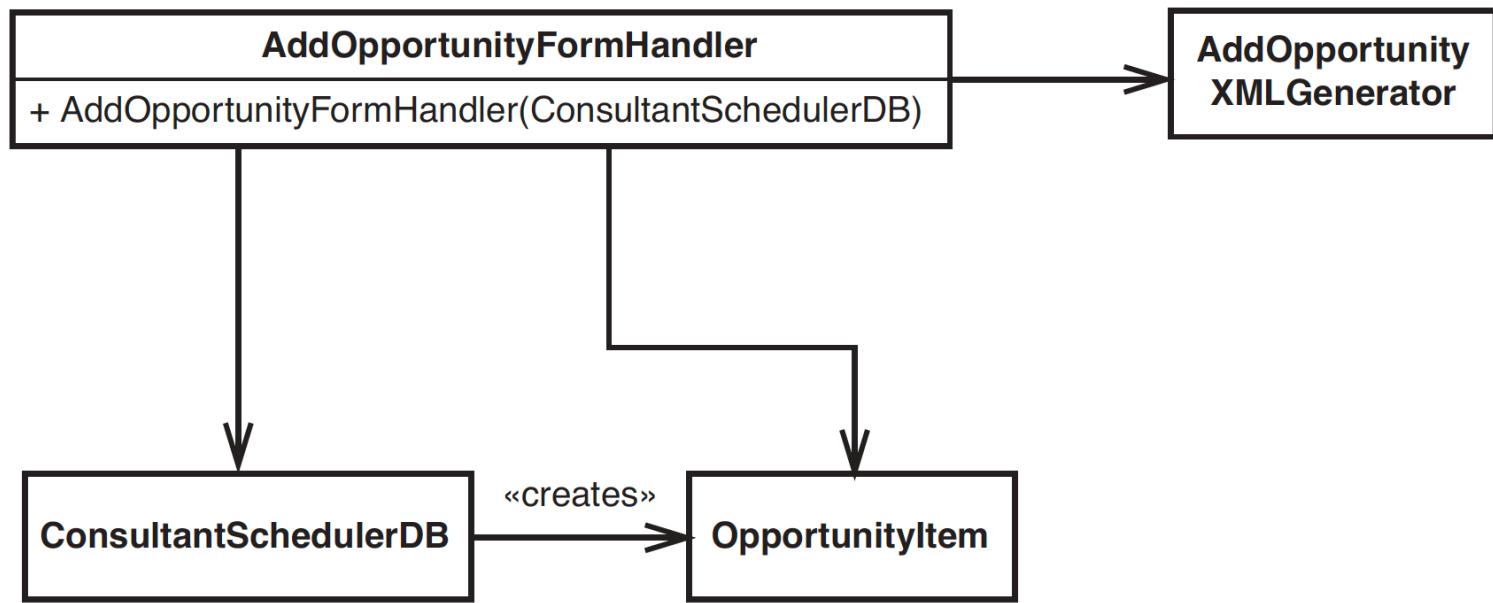


Figure 7.1 *Opportunity handling classes.*

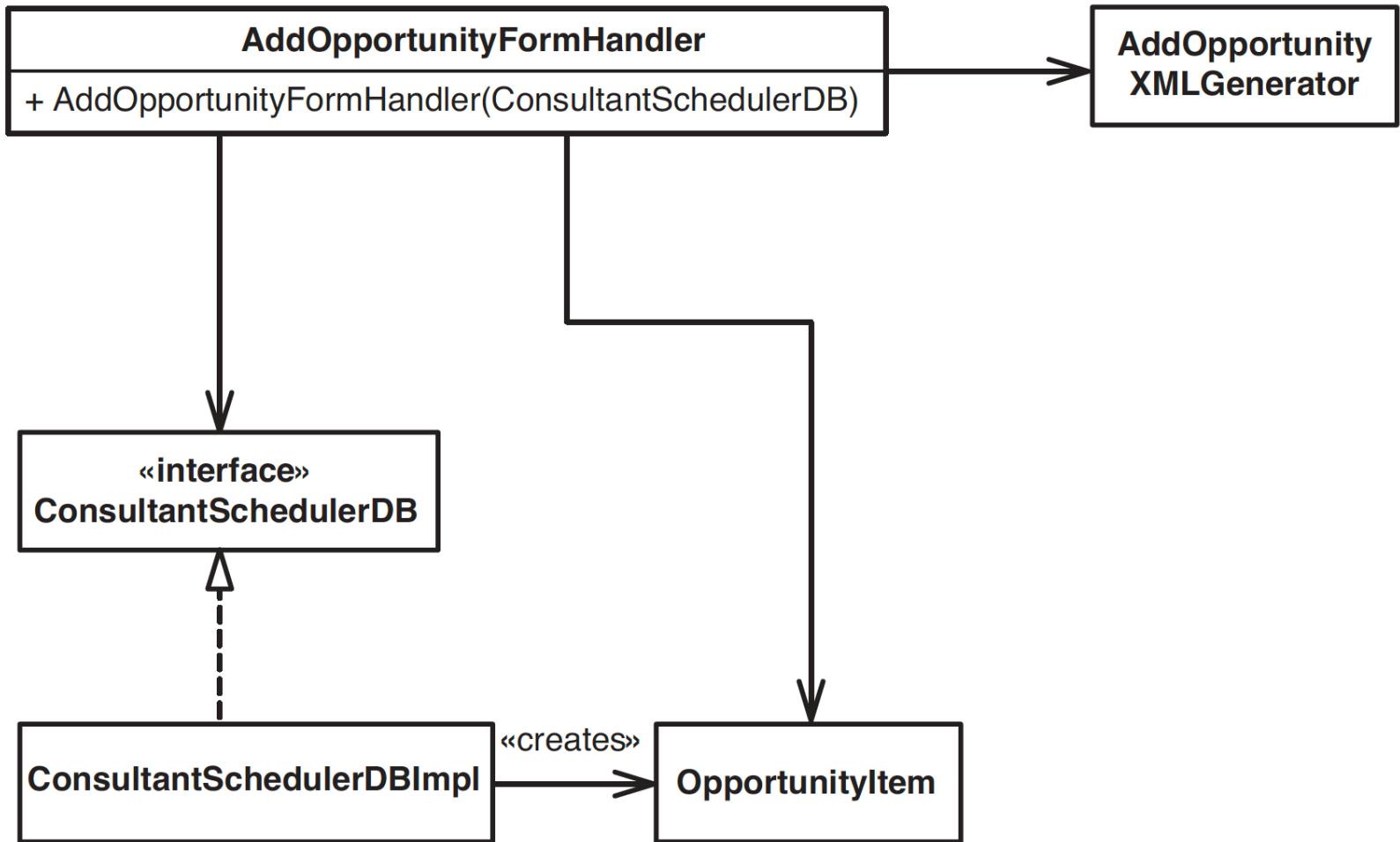


Figure 7.2 Extracting an implementer on `ConsultantSchedulerDB`.

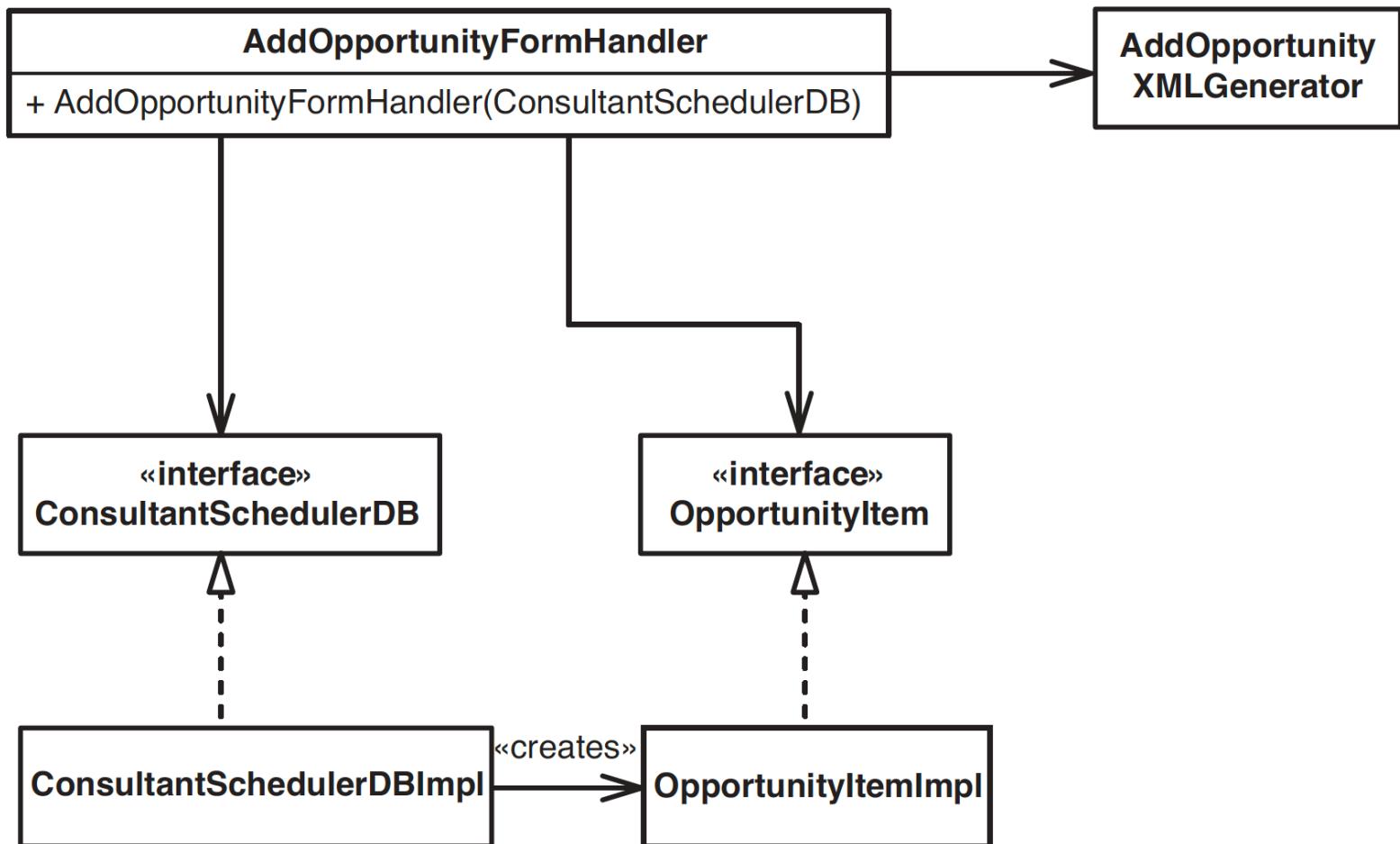


Figure 7.3 *Extracting an implementer on OpportunityItem.*

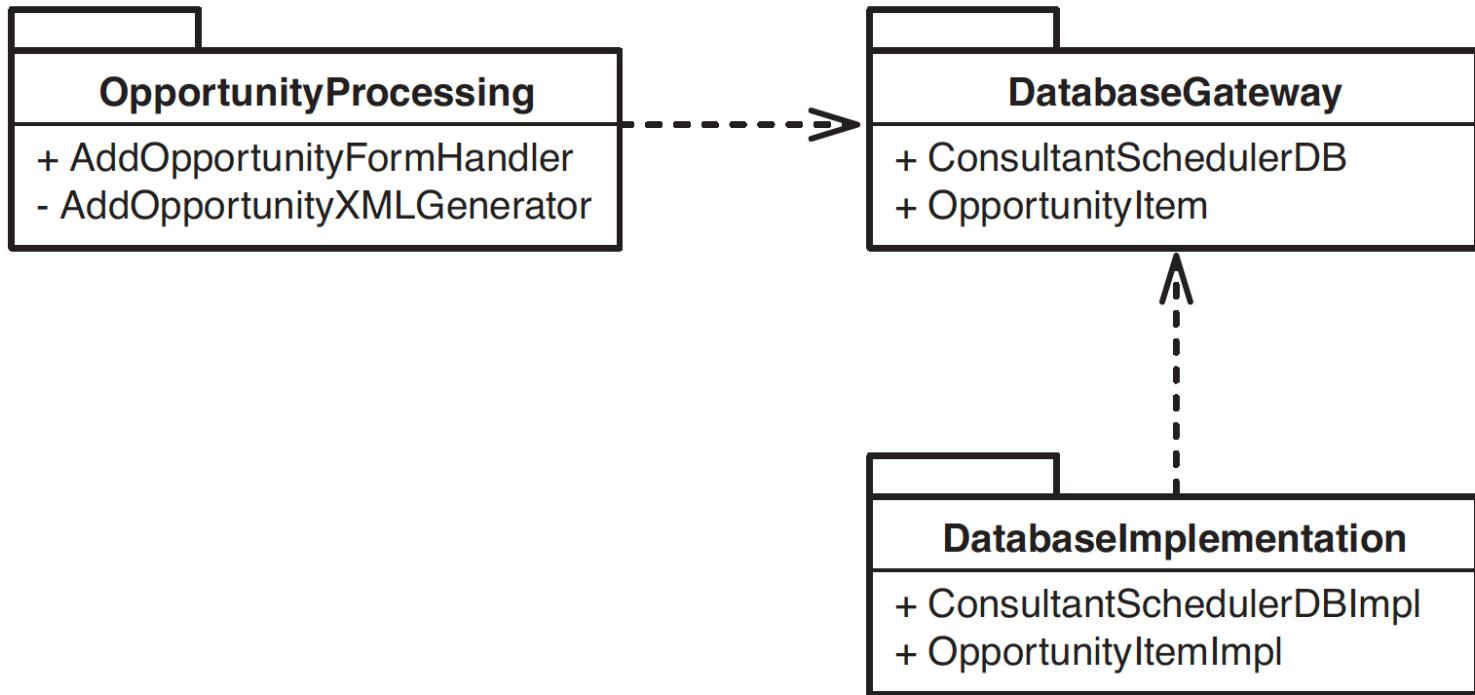


Figure 7.4 Refactored package structure.

Pro and Con

減少了 Package 的大小
但增加了概念的複雜度

練習

<https://github.com/FongX777/trip-service-kata>



讀書會 Telegram Group

<https://t.me/joinchat/L7y14xOmJCgK3Qs3ccykTQ>



<https://github.com/FongX777/trip-service-kata>

請利用剛剛講的四種方式

1. 請用 ch6 介紹的四種方式，在每次的查詢上加上 log 的功能
2. 請用 ch6 介紹的四種方式，在每次查詢反回 Trip 時，用熱門程度進行排序
(假設 Trip 有個欄位叫作 Popularity)
3. 請用 ch7 介紹的解依賴做法來將 DAO 進行依賴反轉

Resource

DDD Taiwan

[https://www.facebook.com/groups/dddтайван/?
ref=br_rs](https://www.facebook.com/groups/dddтайван/?ref=br_rs)

讀書會 Telegram Group

<https://t.me/joinchat/L7y14xOmJCgK3Qs3ccykTQ>

Study Group Github Repo

[https://github.com/ddd-tw/2020-legacycodes-
studygroup](https://github.com/ddd-tw/2020-legacycodes-
studygroup)

The End