

# Working Effectively with Legacy Code

同地方多處修改的解依賴 | 修改時如何測試 | 函式庫依賴問題

Eason Kuo

2020/05/06 讀書會

# About Me

也可以叫我「永和陳奕迅」

Backend Engineer

Python / DDD / Node.js

Startup – Addweup, MenzTech



Eason Kuo

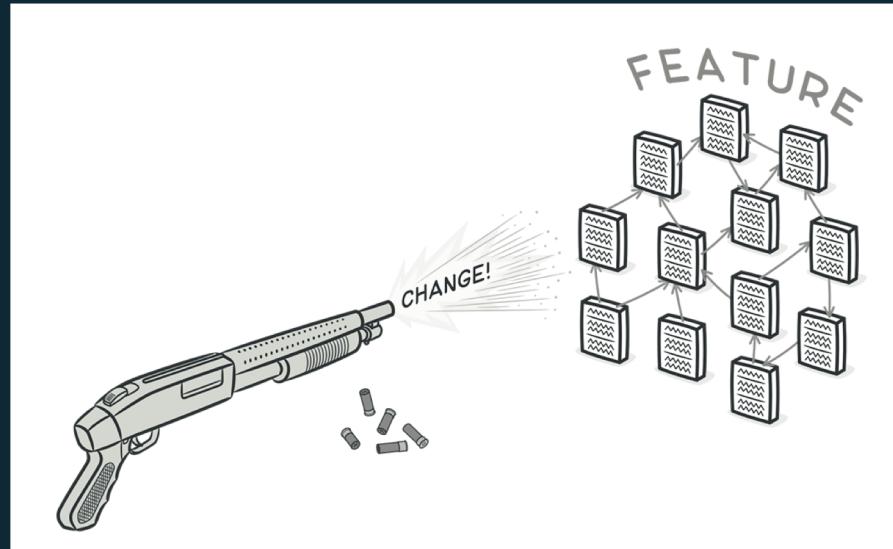
# Chapter 12

在同個地方做多處修改，是否需將所有相關類別解依賴

Ch 12 在同個地方做多處修改，是否需將所有相關類別解依賴？

# 起因

是否有過這樣的感覺？在一處進行多個修改，或因修改一處牽扯到其他處修改（特別是遺留程式碼中內）。

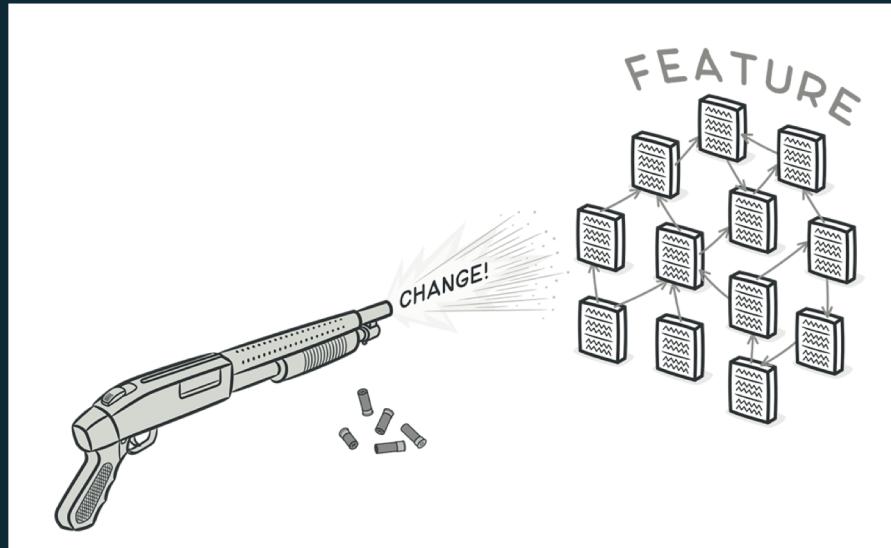


Ch 12 在同個地方做多處修改，是否需將所有相關類別解依賴？

# 起因

修改後為了做測試保護，卻因為修改處牽扯到多個類別，進而花費大量時間做解依賴。

但是否需要真的花費這些時間？  
有無更好的方法？



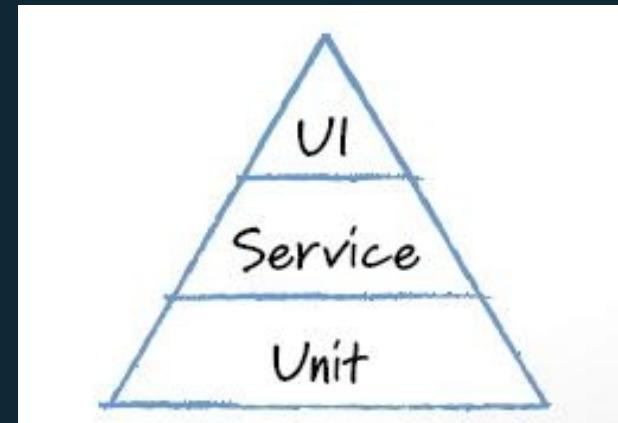
Ch 12 在同個地方做多處修改，是否需將所有相關類別解依賴？

## 心法

退一步海闊天空 > 退一層測試。

退後一層進而找到同夠替多處修改地方一次測試的位置。

E.g 從 API 測試可以一次測到許多功能修改。



Ch 12 在同個地方做多處修改，是否需將所有相關類別解依賴？

## 攔截點 (1) – 步驟

修改 (重構) 後，能夠探測到變化有影響的程式碼區塊，稱為攔截點。

切入步驟：

1. 先確定修改的點
2. 從修改點向外追蹤影響

透過尋找攔截點，來尋找安放寫測試的位置，驗證變化後的結果。

Ch 12 在同個地方做多處修改，是否需將所有相關類別解依賴？

## 攔截點 (2) – 舉例

Invoice 類別為運輸業的單據類別，會有 getValue 方法計算不運輸地同的費用

```
public class Invoice
{
    ...
    public Money getValue() {
        Money total = itemsSum();
        if (billingDate.after(Date.yearEnd(openingDate))) {
            if (originator.getState().equals("FL") ||
                originator.getState().equals("NY"))
                total.add(getLocalShipping());
            else
                total.add(getDefaultShipping());
        }
        else
            total.add(getSpanningShipping());
        total.add(getTax());
        return total;
    }
    ...
}
```

Ch 12 在同個地方做多處修改，是否需將所有相關類別解依賴？

## 攔截點 (2) – 變動

變動：立法機關新增了一項稅目，對紐約的運輸業造成影響，要把這項運輸費用算到客戶上

```
public class Invoice
{
    ...
    public Money getValue() {
        Money total = itemsSum();
        if (billingDate.after(Date.yearEnd(openingDate))) {
            if (originator.getState().equals("FL") ||
                originator.getState().equals("NY"))
                total.add(getLocalShipping());
            else
                total.add(getDefaultShipping());
        }
        else
            total.add(getSpanningShipping());
        total.add(getTax());
        return total;
    }
    ...
}
```

要對 **Invoice** 的 **getValue** 做些調整

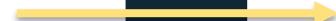
Ch 12 在同個地方做多處修改，是否需將所有相關類別解依賴？

## 攔截點 (2) – 修改處

修改：把負責運輸費用計算的代碼，提取到 ShippingPricer 類別

```
public class Invoice
{
    ...
    public Money getValue() {
        Money total = itemsSum();
        if (billingDate.after(Date.yearEnd(openingDate))) {
            if (originator.getState().equals("FL") ||
                originator.getState().equals("NY"))
                total.add(getLocalShipping());
            else
                total.add(getDefaultShipping());
        }
        else
            total.add(getSpanningShipping());
        total.add(getTax());
        return total;
    }
    ...
}
```

```
public class Invoice
{
    public Money getValue() {
        Money total = itemsSum();
        total.add(shippingPricer.getPrice());
        total.add(getTax());
        return total;
    }
}
```



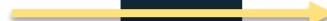
Ch 12 在同個地方做多處修改，是否需將所有相關類別解依賴？

## 攔截點 (2) – 修改處

修改：把負責運輸費用計算的代碼，提取到 ShippingPricer 類別

```
public class Invoice
{
    ...
    public Money getValue() {
        Money total = itemsSum();
        if (billingDate.after(Date.yearEnd(openingDate))) {
            if (originator.getState().equals("FL") ||
                originator.getState().equals("NY"))
                total.add(getLocalShipping());
            else
                total.add(getDefaultShipping());
        }
        else
            total.add(getSpanningShipping());
        total.add(getTax());
        return total;
    }
    ...
}
```

```
public class Invoice
{
    public Money getValue() {
        Money total = itemsSum();
        total.add(shippingPricer.getPrice());
        total.add(getTax());
        return total;
    }
}
```

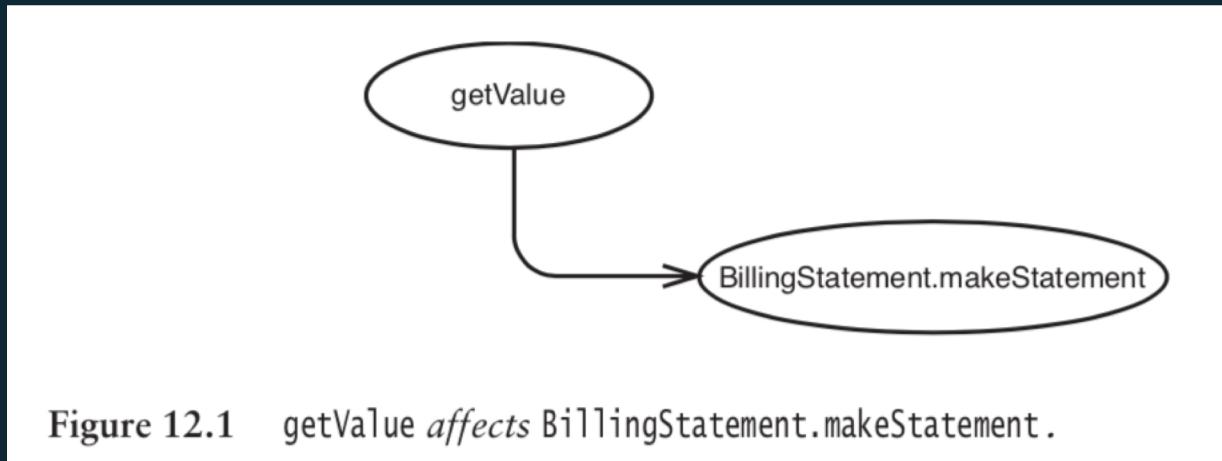


下一步：尋找攔截點

Ch 12 在同個地方做多處修改，是否需將所有相關類別解依賴？

## 攔截點 (3) – 你影響了誰

**getValue 影響了誰？** 外部有一個類別 **BillingStatement** 的 **makeStatement** 使用了 **getValue**



Ch 12 在同個地方做多處修改，是否需將所有相關類別解依賴？

## 攔截點 (4) – 誰會影響你

哪裡可能影響 `getValue`。`getValue` 中的程式碼有哪些會依賴 `Invoice`

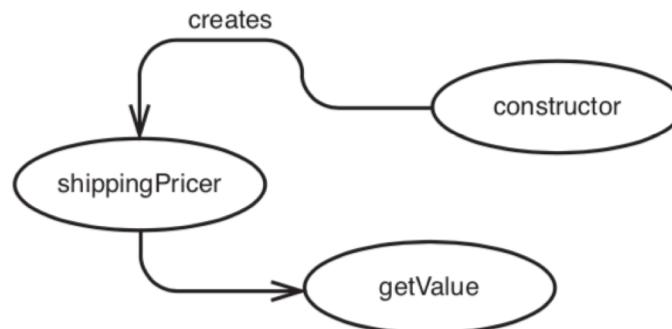
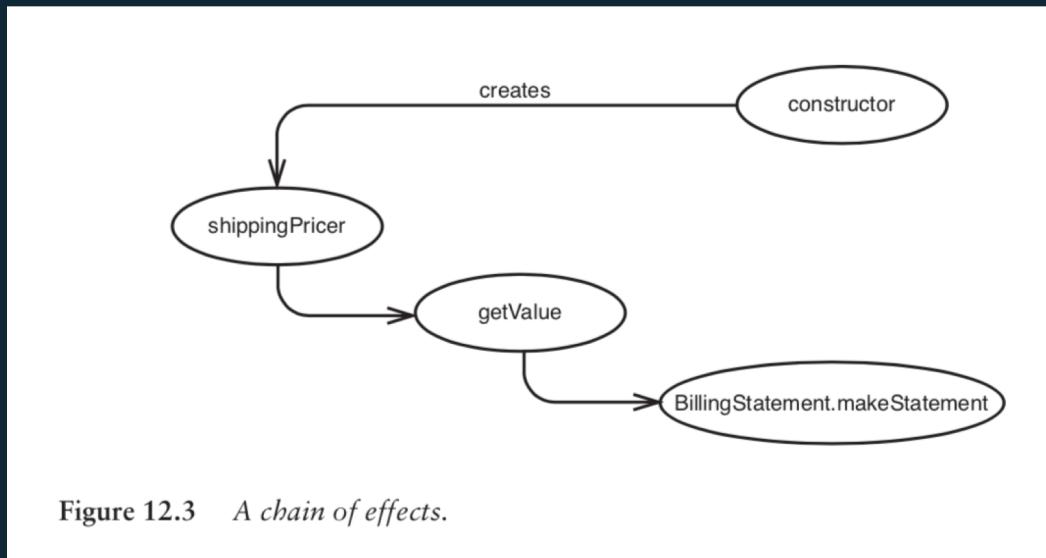


Figure 12.2 Effects on `getValue`.

Ch 12 在同個地方做多處修改，是否需將所有相關類別解依賴？

## 攔截點 (5) – 善用影響結構圖

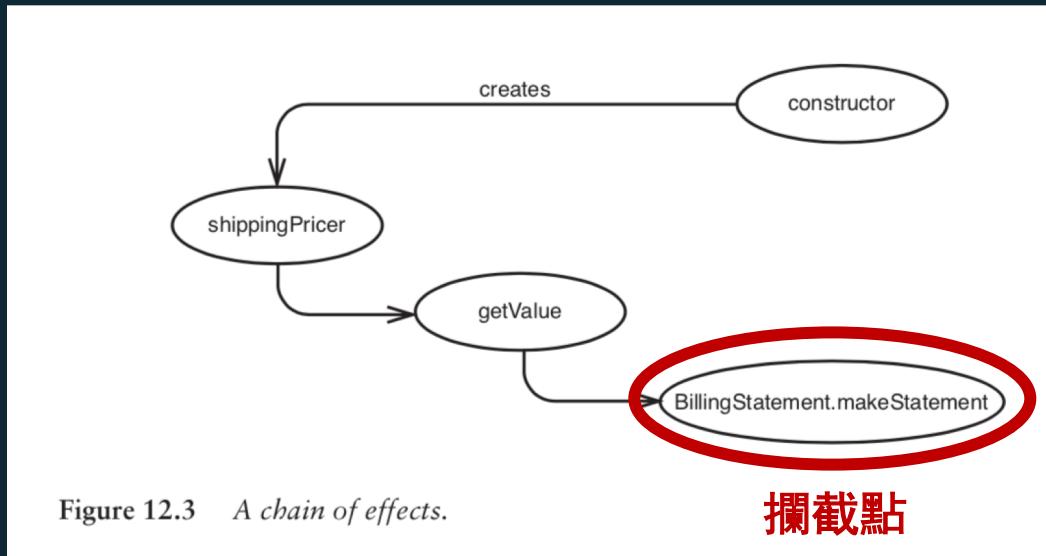
在修改點中，找出你影響了誰；找出誰會影響你，並劃出影響結構圖。



Ch 12 在同個地方做多處修改，是否需將所有相關類別解依賴？

## 攔截點 (5) – 善用影響結構圖

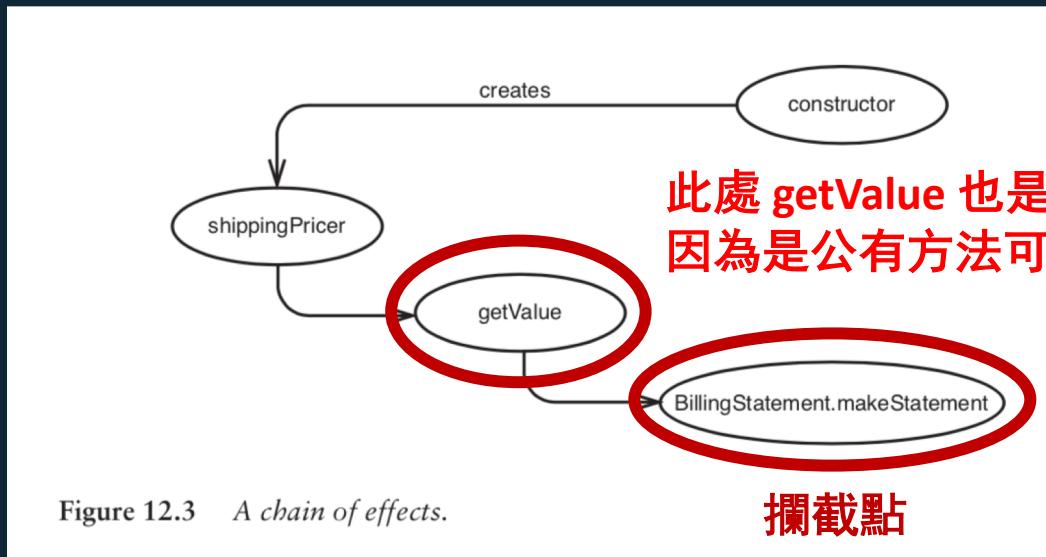
在修改點中，找出你影響了誰，找出誰會影響你，並劃出影響結構圖。



Ch 12 在同個地方做多處修改，是否需將所有相關類別解依賴？

## 攔截點 (5) – 善用影響結構圖

若修改點本身可以被探測到，也會是一個攔截點。



Ch 12 在同個地方做多處修改，是否需將所有相關類別解依賴？

## 攔截點 (6) – 回顧

修改 (重構) 後，能夠探測到變化有影響的程式碼區塊，稱為攔截點。

切入步驟：

1. 先確定修改的點
2. 從修改點向外追蹤影響 (使用影響結構圖)
3. 標記可以被探測 (測試) 並觀察到的位置為一個攔截點

探測到變化有影響，也可以說我們能測試到有變化。

Ch 12 在同個地方做多處修改，是否需將所有相關類別解依賴？

## 攔截點 (7) – 建議

攔截點離修改點越近越好。 離越遠複雜度會越高。

會越難探測出變化後影響的結果。

例如：下圍棋。



Ch 12 在同個地方做多處修改，是否需將所有相關類別解依賴？

## 攔截點 (8) – 越近越好？

多數情況，最容易找到，最近的攔截點，就是修改點附近的公有方法，或修改點本身。

雖然攔截點離修改點越近越好，但也並非最近的就都是最佳的攔截點

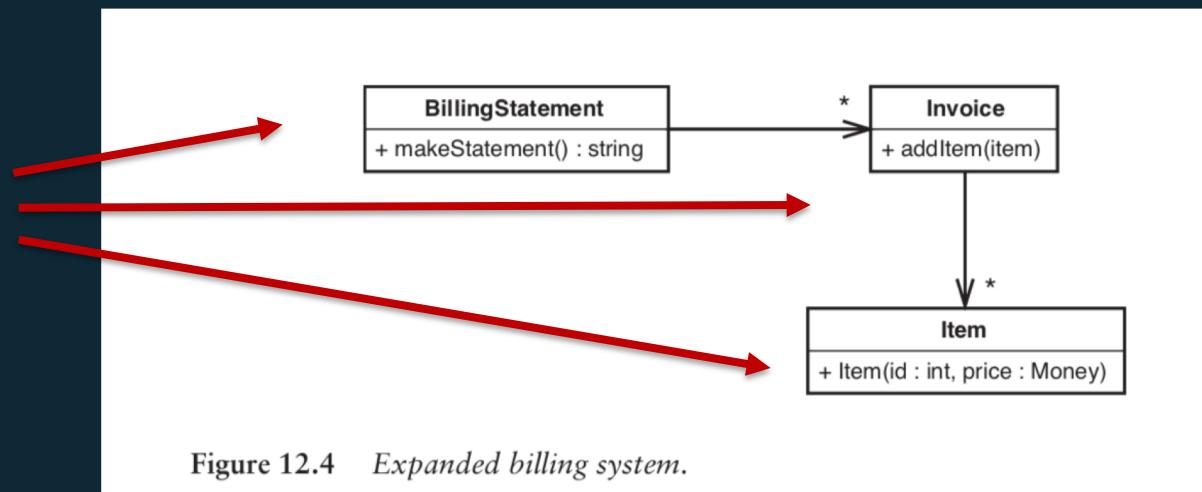
看下一個例子：高階攔截點

Ch 12 在同個地方做多處修改，是否需將所有相關類別解依賴？

# 高階層攔截點 (1) – 情境

新變動：除了原先的 **Invoice** 計算修改，還需修改 **Item** 類別，給成員變數來記錄運輸方式，此外 **BillingStatement** 也需要修改，給每個托運人設定一個獨立的細目分類。

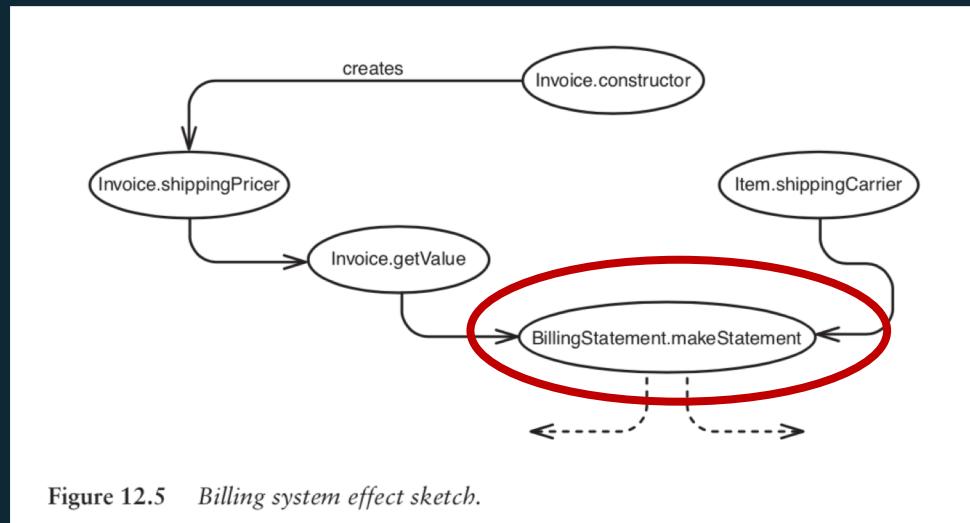
例：一次修改  
多處相關的類別



Ch 12 在同個地方做多處修改，是否需將所有相關類別解依賴？

## 高階層攔截點 (2) – 好處

更高效率的方式，找出最具代表性可以探測到多個修改處的區塊，作為「高階攔截點」



Ch 12 在同個地方做多處修改，是否需將所有相關類別解依賴？

## 高階層攔截點 (2) – 好處

好處：需要進行的解依賴減少，可以從更高層一次夾住所有修改區塊的測試。

```
void testSimpleStatement() {  
    Invoice invoice = new Invoice();  
    invoice.addItem(new Item(0,new Money(10)));  
    BillingStatement statement = new BillingStatement();  
    statement.addInvoice(invoice);  
    assertEquals("", statement.makeStatement());  
}
```

此時從 **BillingStatement** 層級測試較佳

Ch 12 在同個地方做多處修改，是否需將所有相關類別解依賴？

## 高階層攔截點 (3) – 關鍵

若高階攔截點符合：「單一地點探測所有的影響」，又被稱作匯點

匯點：影響結構圖中的交通要道，此處測試能夠覆蓋大量的修改，更能減少工作量提高效率。

注意：匯點是根據修改點來決定。

Ch 12 在同個地方做多處修改，是否需將所有相關類別解依賴？

## 匯點 (1) – 舉例

再修改：在 Item 中提供一個方法，以便能夠獲取 (getSupplier) 及設置貨物 (setSupplier) 的供應商。

**InventoryControl** 與 **BillingStatement**  
兩個類別也會使用這個供應商名稱  
(Item getSupplier)

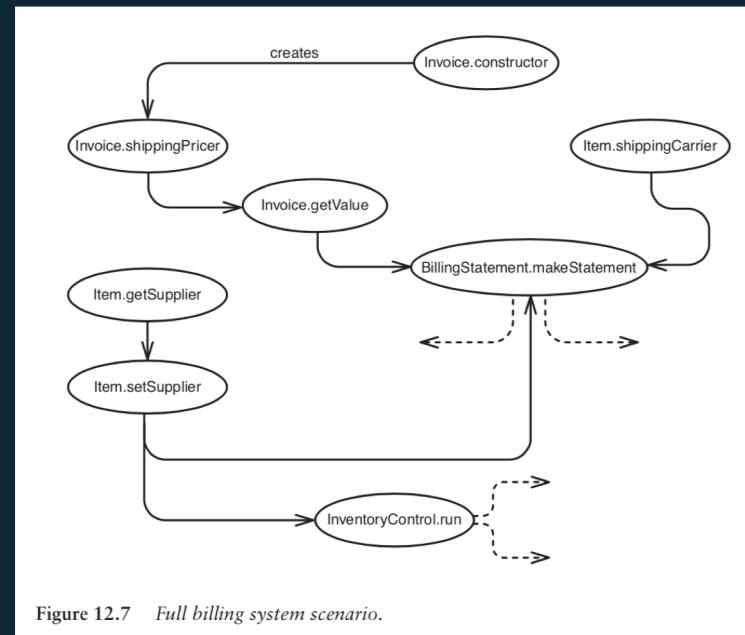


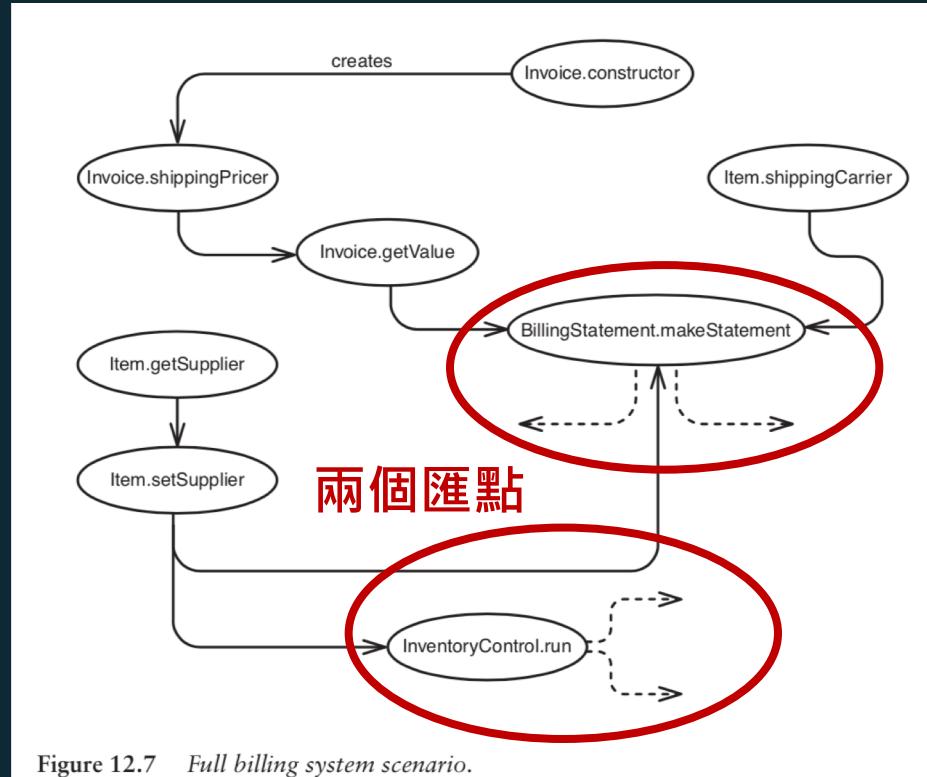
Figure 12.7 Full billing system scenario.

Ch 12 在同個地方做多處修改，是否需將所有相關類別解依賴？

## 匯點 (3) – 多個匯點

某個修改可能會同時存在  
多個匯點也處。

不一定每個匯點處都要寫測試。



Ch 12 在同個地方做多處修改，是否需將所有相關類別解依賴？

## 匯點 (4) – 建議

當匯點不易尋找，或影響的範圍複雜度過大，思考能否一次只針對一兩組修改尋找匯點。

若找不到匯點，就以最近的修改點為最佳的攔截點原則來編寫測試。

Ch 12 在同個地方做多處修改，是否需將所有相關類別解依賴？

## 匯點 (4) – 建議

當匯點不易尋找，或影響的範圍複雜度過大，思考能否一次只針對一兩組修改尋找匯點。

若找不到匯點，就以最近的修改點為最佳的攔截點原則來編寫測試。

能找到匯點 > 就在匯點處寫測試。

找不到匯點 > 在最近的修改點處寫測試

Ch 12 在同個地方做多處修改，是否需將所有相關類別解依賴？

## 匯點的陷阱

勿拿匯點寫單元測試，否則會逐漸變成「迷你型整合測試」。

單元測試：小型的功能或模組的獨立測試。

匯點寫測試：像圈地遊戲，是一種可以隨著重構的範圍越大，也會影響範圍的測試。

隨著重構與測試所掌握的範圍越大後，便可以逐漸捨棄原先編寫匯點的測試，而改用單元測試來支援自己開發。

# Chapter 13

## 修改時應該怎樣寫測試？

Ch13修改時應該怎樣寫測試？

## 起因 - 在遺留程式中編寫測試的困難

大多數遺留程式中，可能沒有任何測試包覆，修改後也難驗證，導致修改時不小心改變原有行為。

最好的方式是把想要修改的區塊先用測試製作「安全網」，並在修改過程中解決 Bug

Ch13修改時應該怎樣寫測試？

# 認識系統、認識你要修改的地方為第一步

大多數遺留程式中，可能沒有任何測試包覆，修改後也難驗證，導致修改時不小心改變原有行為。

But

我連系統可以做什麼都不清楚，該如何開始撰寫測試？

https://www.youtube.com/watch?v=...

Ch13修改時應該怎樣寫測試？

## 特徵測試 (1) – 用途

所以對於遺留系統，更重要的不是系統「應該能」做什麼，而是這個系統「目前能」做什麼。

除了收集領域知識外，在了解代碼中的意涵上，我們可以透過「特徵測試」，來減少代碼的不確定性。

特徵測試：一種用於行為保持的測試。

Ch13修改時應該怎樣寫測試？

## 特徵測試 (2) – 步驟

撰寫特徵測試的步驟：

1. 在測試工具中選擇一段你要測試的代碼
2. 編寫一個你知道測試會失敗的斷言 (Assert)
3. 從斷言的失敗中得知程式碼的部分行為
4. 修改測試，使被測代碼的行為能被測試預期。
5. 重複上述步驟。

Ch13修改時應該怎樣寫測試？

## 特徵測試 (3) – 範例

特徵測試刻畫了程式碼的實際行為，而不是猜想「他應該會具有這個行為」，或者「我想他會這樣運作吧」。

1. 驗證呼叫 `.generate()` 預期的行為
2. 得知預期為空字串，失敗

```
void testGenerator() {  
    PageGenerator generator = new PageGenerator();  
    assertEquals("fred", generator.generate());  
}
```

```
.F  
Time: 0.01  
There was 1 failure:  
1) testGenerator(PageGeneratorTest)  
junit.framework.ComparisonFailure: expected:<fred> but was:<>  
    at PageGeneratorTest.testGenerator  
        (PageGeneratorTest.java:9)  
    at sun.reflect.NativeMethodAccessorImpl.invoke0  
        (Native Method)  
    at sun.reflect.NativeMethodAccessorImpl.invoke  
        (NativeMethodAccessorImpl.java:39)  
    at sun.reflect.DelegatingMethodAccessorImpl.invoke  
        (DelegatingMethodAccessorImpl.java:25)  
  
FAILURES!!!  
Tests run: 1, Failures: 1, Errors: 0
```

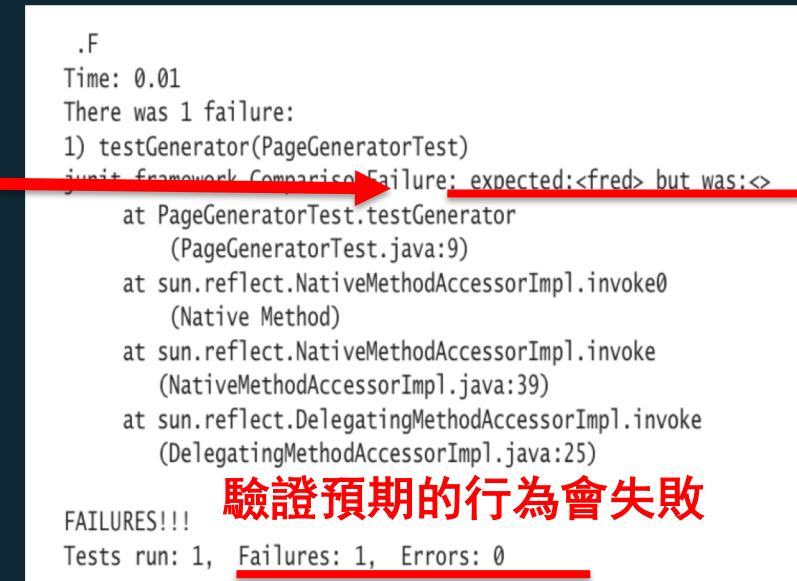
Ch13修改時應該怎樣寫測試？

## 特徵測試 (3) – 範例

特徵測試刻畫了程式碼的實際行為，而不是猜想「他應該會具有這個行為」，或者「我想他會這樣運作吧」。

1. 驗證呼叫 `.generate()` 預期的行為
2. 得知預期為空字串，失敗

```
void testGenerator() {  
    PageGenerator generator = new PageGenerator();  
    assertEquals("fred", generator.generate());  
}
```



.F  
Time: 0.01  
There was 1 failure:  
1) testGenerator(PageGeneratorTest)  
junit.framework.ComparisonFailure: expected:<fred> but was:<>  
at PageGeneratorTest.testGenerator  
(PageGeneratorTest.java:9)  
at sun.reflect.NativeMethodAccessorImpl.invoke0  
(Native Method)  
at sun.reflect.NativeMethodAccessorImpl.invoke  
(NativeMethodAccessorImpl.java:39)  
at sun.reflect.DelegatingMethodAccessorImpl.invoke  
(DelegatingMethodAccessorImpl.java:25)

**驗證預期的行為會失敗**

FAILURES!!!  
Tests run: 1, Failures: 1, Errors: 0

Ch13修改時應該怎樣寫測試？

## 特徵測試 (4) – 關於 Bug

特徵測試並非一般的測試，不是用來尋找程式的 Bug。

而是描述系統中各部分實際行為，同時有一個機制以便未來修改時可以尋找與目前系統行為不一致的 Bug 行為。

若發現某些結果與預期不一致，那麼可能需要去標記並釐清他，因為有可能是遺留系統中的 Bug，要弄清若修正可能會對系統帶來什麼影響。

Ch13修改時應該怎樣寫測試？

## 特徵測試 (4) – 何時該停止

使用編寫特徵測試時，可以一邊閱讀被測的程式碼區塊，一邊理解他的意圖，若不清楚就編寫特徵測試來「詢問」它

寫到自己足夠理解遺留系統中的程式碼。

Ch13修改時應該怎樣寫測試？

## 特徵測試 (4) – 何時該停止

使用編寫特徵測試時，可以一邊閱讀被測的程式碼區塊，一邊理解他的意圖，若不清楚就編寫特徵測試來「詢問」它

寫到自己足夠理解遺留系統中的程式碼。

以我為例：我可以跟其他人講解這段代碼或系統的每一處時，那我就會停止。

Ch13修改時應該怎樣寫測試？

# 目標測試 (1)

編寫特徵測試了解意圖後，接著需要確認編寫的測試是否有真的覆蓋到程式碼。

目標測試：是一種針對修改處為目標的測試。

Ch13修改時應該怎樣寫測試？

## 目標測試 (2) – 情境

FuelShare 是出租燃料罐的類別，有一個是計算出燃料罐中燃料總價值的方法。



```
public class FuelShare
{
    private long cost = 0;
    private double corpBase = 12.0;
    private ZonedHawthorneLease lease;

    ...
    public void addReading(int gallons, Date readingDate){
        if (lease.isMonthly()) {
            if (gallons < Lease.CORP_MIN)
                cost += corpBase;
            else
                cost += 1.2 * priceForGallons(gallons);
        }
        ...
        lease.postReading(readingDate, gallons);
    }
    ...
}
```

Ch13修改時應該怎樣寫測試？

## 目標測試 (3) – 修改

修改：把其中的判斷式提取到另一個 `ZoneHawthorneLease` 類別中的新方法

```
public class FuelShare
{
    private long cost = 0;
    private double corpBase = 12.0;
    private ZonedHawthorneLease lease;
    ...
    public void addReading(int gallons, Date readingDate){
        if (lease.isMonthly()) {
            if (gallons < Lease.CORP_MIN)
                cost += corpBase;
            else
                cost += 1.2 * priceForGallons(gallons);
        }
        ...
        lease.postReading(readingDate, gallons);
    }
    ...
}
```



Ch13修改時應該怎樣寫測試？

## 目標測試 (2) – 範例

判斷式被抽出來外，原先的 `priceForGallons(gallons)` 被修改為 `totalPrice`  
> 需要驗證 `else` 的邏輯

```
public class FuelShare
{
    public void addReading(int gallons, Date readingDate){
        cost += lease.computeValue(gallons,
            priceForGallons(gallons));
        ...
        lease.postReading(readingDate, gallons);
    }
    ...
}
```

```
public class ZonedHawthorneLease extends Lease
{
    public long computeValue(int gallons, long totalPrice) {
        long cost = 0;
        if (lease.isMonthly()) {
            if (gallons < Lease.CORP_MIN)
                cost += corpBase;
            else
                cost += 1.2 * totalPrice;
        }
        return cost;
    }
    ...
}
```

Ch13修改時應該怎樣寫測試？

# 目標測試 – 確認有無測試覆蓋

修改後的代碼如下後，觀察修改部分的測試是否有覆蓋到。

```
public void testValueForGallonsMoreThanCorpMin() {  
    StandardLease lease = new StandardLease(Lease.MONTHLY);  
    FuelShare share = new FuelShare(lease);  
  
    share.addReading(FuelShare.CORP_MIN +1, new Date());  
    assertEquals(12, share.getCost());  
}
```

Ch13修改時應該怎樣寫測試？

## 目標測試 – 確認有無測試覆蓋

除關注的測試分支外，要確認是否還存有其他通過的條件，若不確定的話，可以使用感測變數 (Ch22) 或 Debug 工具來檢查。

要確認是否不會因為提供的輸入值，反而造成本該失敗的測試成功了，  
e.g : 語言的隱性型別處理

Ch13修改時應該怎樣寫測試？

## 目標測試 (3) – 回顧

	特徵測試	目標測試
用途	認識程式碼實際行為的一種測試。	針對修改處為目標的測試。
使用時機	尚未修改前使用	修改後使用

Ch13修改時應該怎樣寫測試？

## 目標測試 (3) – 回顧

1. 為準備修改的代碼先寫特徵測試，直到自己認為已經理解這塊代碼。
2. 開始進行修改或重構
3. 針對修改與重構的每一處編寫目標測試，並做驗證（含型別轉換）

# Chapter 14

# 棘手的函式庫依賴問題

# 過分依賴函式庫可能帶來的困境

避免過份依賴函式庫，特別是付費的函式庫。

困境 (含個人經驗心得)：

1. 函式庫廠商費用與後續服務費用拉高，影響公司收益
2. 需配合供應商的函式庫調整
3. 供應商服務出問題時，會連帶受到影響
4. 諮詢或提出需求客製化調整，需要看供應商臉色與配合度。

# 減少直接使用

減少在程式中直接使用呼叫函式庫：善用接縫、介面。

一個好的函式庫，能夠定義明確類別與介面，並能使函式庫使用者可以在測試環境中執行，做到在測試中「偽造」。

若函式庫無法提供可偽造的測試環境，則只能夠過外覆類別 (Ch 6.4 Wrap Class) 來處理 (並寫信抱怨)。

# Exercise 練習試試看

Exercise 練習試試看

# Supermarket Receipt Refactoring Kata

來源: <https://github.com/kokokuo/SupermarketReceipt-Refactoring-Kata>

場景描述:

歡迎來到 DDD 社群超級市場。

此市場有一個目錄其中包含不同類型的產品  
( 大米 , 蘋果 , 牛奶 , 牙刷等 ) 。

每個產品都有一個價格 , 並且購物車的總價是商品所有價格的總和。

您會收到一張收據 , 其中詳細說明了您所購買的商品 , 總價以及所應用的任何折扣。



Exercise 練習試試看

# Supermarket Receipt Refactoring Kata

來源：<https://github.com/kokokuo/SupermarketReceipt-Refactoring-Kata>

場景描述 (Con't):

另外 DDD 超級市場現在特價！

- 買兩把牙刷，免費送一把。( 普通牙刷的價格是 €0.99)
- 蘋果可享受 20% 的折扣。 ( 正常價格為每公斤 € 1.99 )
- 大米可享受 10% 的折扣。 ( 每袋正常價格 €2.49 )
- 五支牙膏的價格為 € 7.49 。 ( 原價一隻 €1.79 )
- 兩盒櫻桃番茄的價格為 € 0.99 。 ( 正常價格為每盒 € 0.69 )



Exercise 練習試試看

# Supermarket Receipt Refactoring Kata

來源：<https://github.com/kokokuo/SupermarketReceipt-Refactoring-Kata>

練習：

1. 對 ShoppingCart 的 HandleOffers 方法嘗試做特徵測試，寫出目前該方法的三至四特徵種測試案例，此處可以透過市場的特價情境來測試 HandleOffers。
2. 對 HandleOffers 方法做 Extract Method 重構，並根據你的重構的結果，練習用影響結構圖找出自己認為最佳的攔截點位置。
3. 對重構後的新方法，嘗試結合找到的最佳攔截點，來實作修改處的目標測試。
4. (Optional) 新功能 - Discounted bundles，撰寫並找攔截點，做目標測試。

Exercise 練習試試看

# Supermarket Receipt Refactoring Kata

來源：<https://github.com/kokokuo/SupermarketReceipt-Refactoring-Kata>

語言：

- C++
- C#
- Java
- Kotlin
- Typscript
- Python
- Ruby



Exercise 練習試試看

## 總結

修改處如何解依賴：找攔截點，若攔截點是一個匯點，則在匯點寫測試  
否則找離修改處最近的攔截點即可。

匯點：可以涵蓋所有修改處的攔截點。檢測變化的為交通要道

遺留系統如何開始寫測試：認識系統的領域知識，寫特徵測試。

特徵測試：寫會失敗的測試 > 認識你系統的程式碼行為

目標測試：修改(重構)後寫的測試，測試你修改的地方有無預期行為。

Exercise 練習試試看

# Resource

DDD TW Facebook 社團: <https://www.facebook.com/groups/dddtaiwan/>

DDD TW Telegram: <https://t.me/joinchat/Gb2jERB7BGSXIOgGkMKhxw>

Legacy Code Github: <https://github.com/ddd-tw/2020-legacycode-studygroup>

Legacy Code Telegram: <https://t.me/joinchat/Ky6eQROmJCjFpNzw80ZZwA>

# End