

# Working Effectively with Legacy Code

## Chapter 25 解依賴技術 ( 精選 )

James Wang

2020 / 07 / 08

# I'm James

- 公司內擔任 Programmer 。
- 於 **AgileCommunity.tw** 與 **Domain Driven Design Taiwan** 擔任社群志工 。
- 偶爾出沒感興趣主題的社群 。
- 曾於 Agile Tour Taipei 和 Agile 新竹/高雄社群分享 。



**AgileCommunity.tw**  
( FB )



**DDD Taiwan**  
( FB )

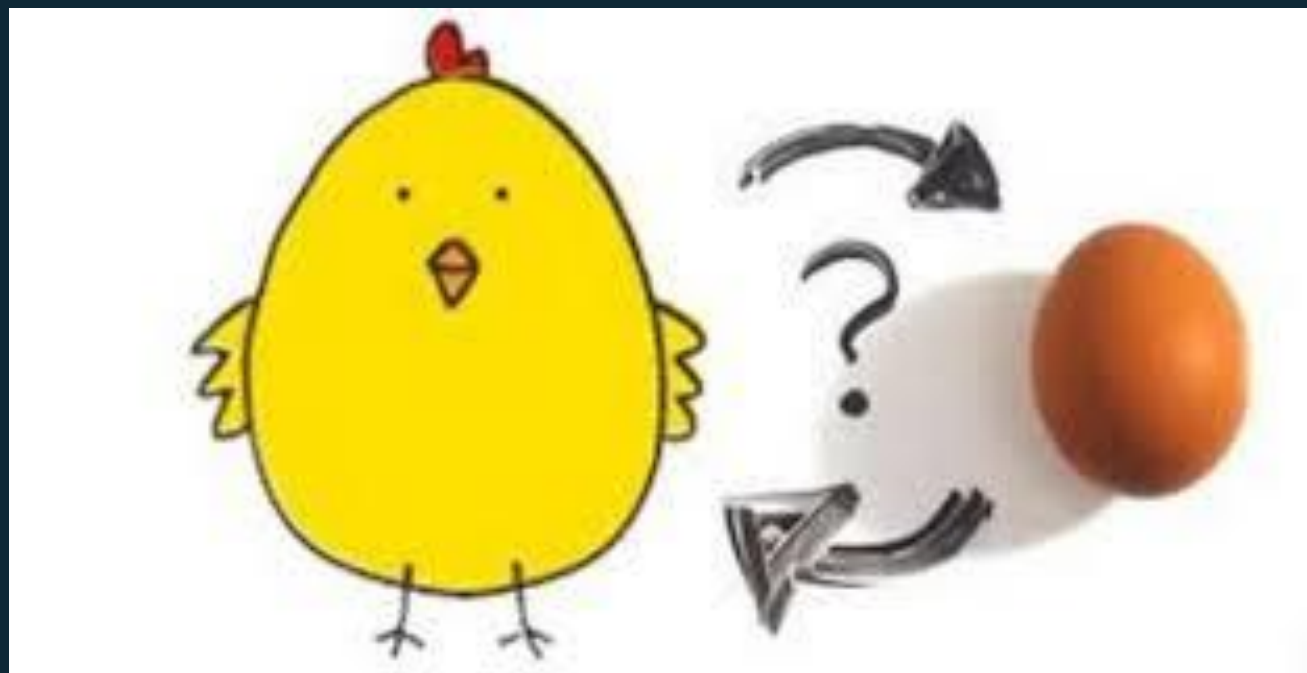
# DDD 社群 7 月活動



<https://tinyurl.com/y82u7xuj>

**【DDD TW 7月沙龍】用 DDD & BDD 開外掛：把共識變測試**

# 開始之前先回顧一下



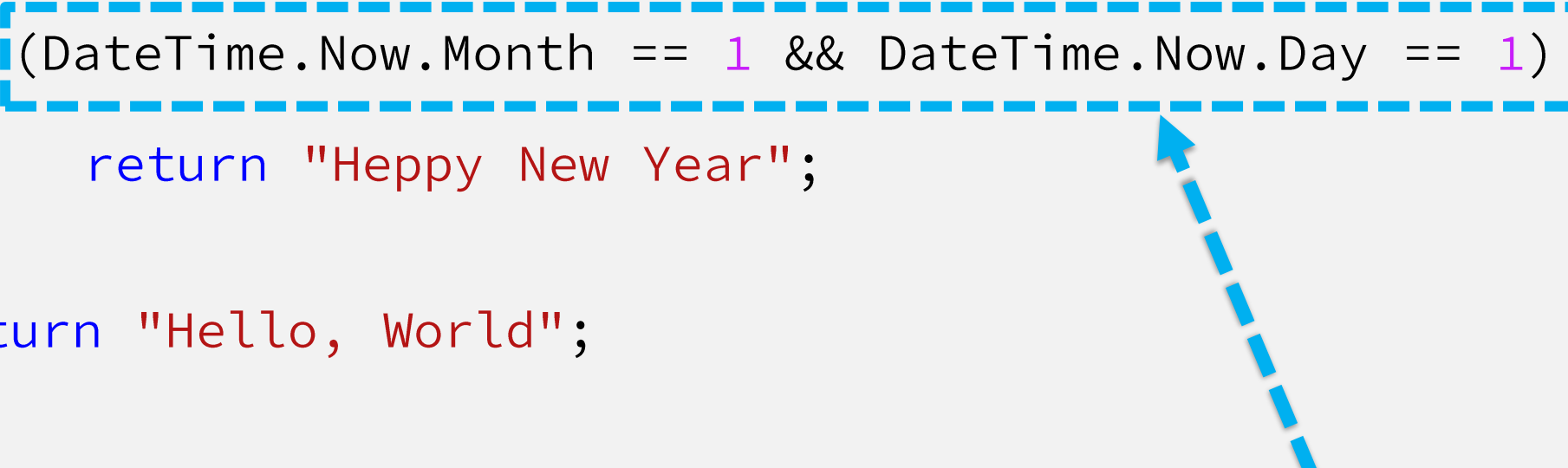
我們無法補上單元測試，那我們就補上整合測試

# 提取 ( Extract ) & 覆寫 ( Override )

# 提取誰？

```
public class Hello
{
    public string HelloWorld()
    {
        if (DateTime.Now.Month == 1 && DateTime.Now.Day == 1)
        {
            return "Heppy New Year";
        }

        return "Hello, World";
    }
}
```



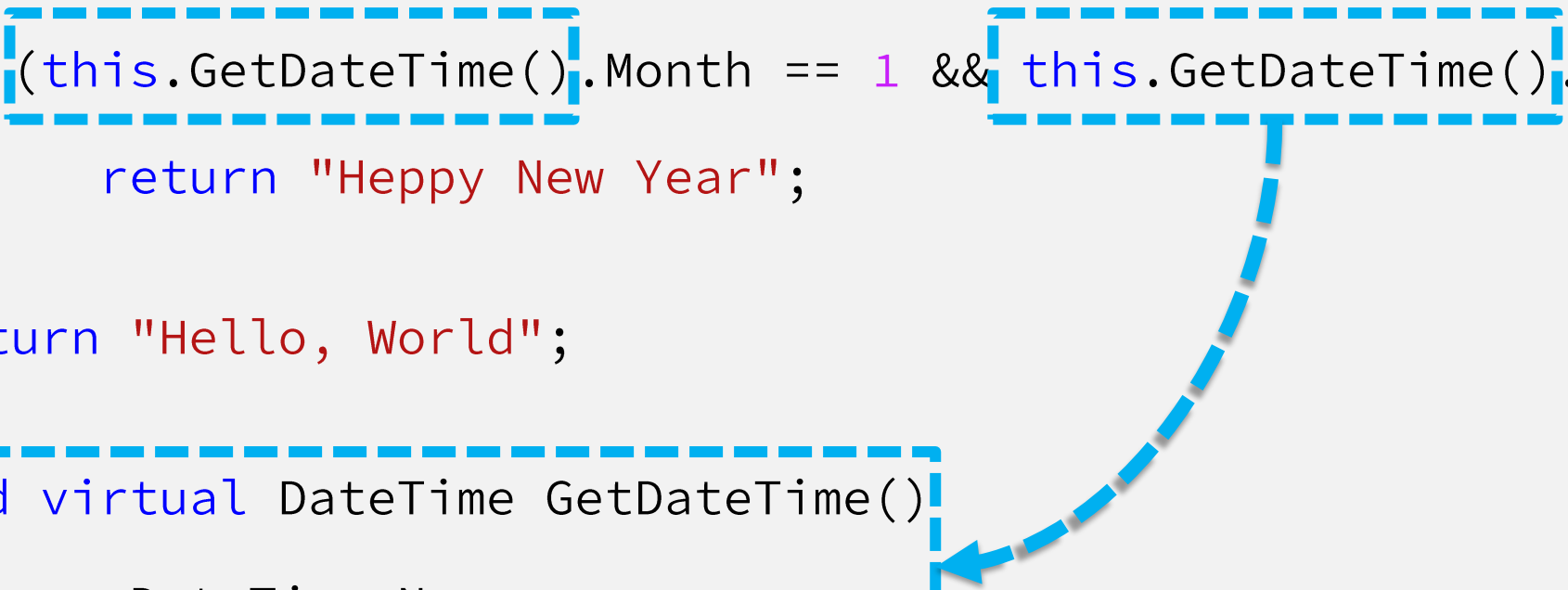
接縫 ( Seam )

# 提取接縫 ( Seam )

```
public class Hello
{
    public string HelloWorld()
    {
        if (this.GetDateTime().Month == 1 && this.GetDateTime().Day == 1)
        {
            return "Heppy New Year";
        }

        return "Hello, World";
    }

    protected virtual DateTime GetDateTime()
    {
        return DateTime.Now;
    }
}
```





# 於測試程式覆寫 ( Override )

```
public class HelloStub : Hello
{
    protected override DateTime GetDateTime()
    {
        return new DateTime(2021, 1, 1);
    }
}
```

# 提取並覆寫呼叫 ( Extract and Override Call ) ch25.6

於被測試對象 ( SUT ) 中：

- 將相依物件從 SUT 中擷取 ( Extract ) 出來，改成 protected virtual。
- SUT 使用擷取出來的方法。

於測試專案中：

- 建立新的 stub 類別，繼承 SUT 類別。
- 覆寫 ( override ) 方法，並實作覆寫內容。

於測試程式中：

- 呼叫新建立的 stub 類別。

# 提取與覆寫的其他版本

將 SUT 中相依物件提取 ( Extract ) 至 Getter 。

```
private DateTime? _mockNow;  
  
internal DateTime MockNow  
{  
    get  
    {  
        return _mockNow ?? DateTime.Now;  
    }  
    set { _mockNow = value; }  
}
```

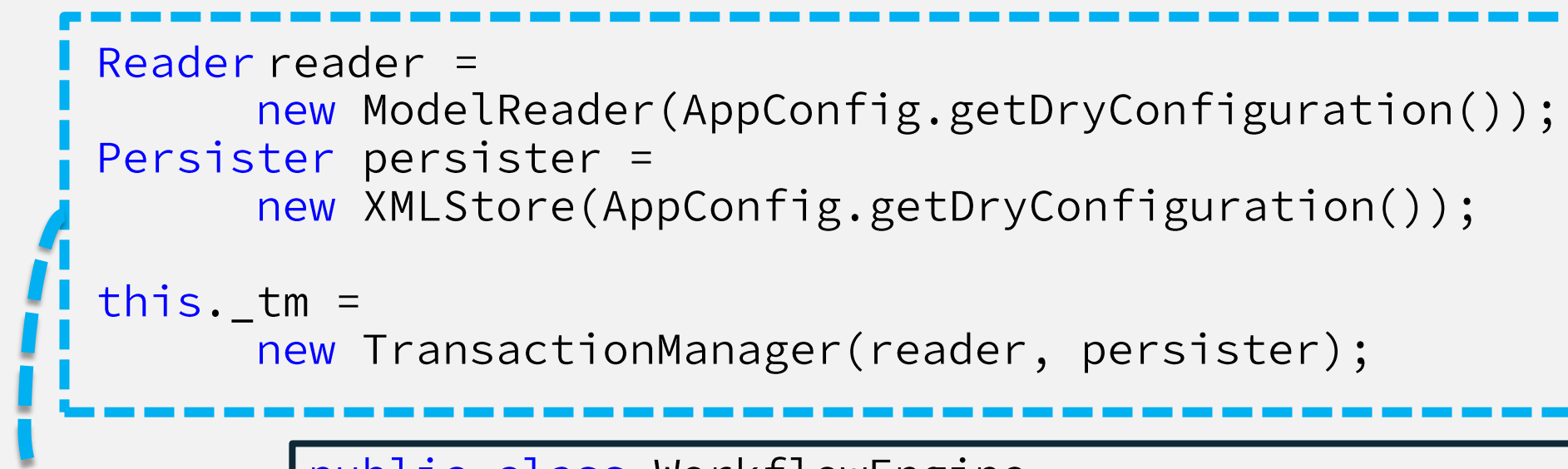
這個手法稱之為提取並覆寫獲取方法 ( Extract and Override Getter )

Ch25.8

# 提取與覆寫的其他版本

```
public class WorkflowEngine
{
    public WorkflowEngine()
    {
        Reader reader =
            new ModelReader(AppConfig.getDryConfiguration());
        Persister persister =
            new XMLStore(AppConfig.getDryConfiguration());

        this._tm =
            new TransactionManager(reader, persister);
    }
}
```



提取工廠

```
public class WorkflowEngine
{
    public WorkflowEngine()
    {
        this._tm =
            makeTransactionManager();
    }
}
```

# 提取與覆寫的其他版本

將 SUT 中相依物件提取 ( Extract ) 至工廠。

```
public class WorkflowEngine
{
    public WorkflowEngine()
    {
        this._tm =
            makeTransactionManager();
    }
}
```

這個手法稱之為提取並覆寫工廠方法 ( Extract and Override Factory Method )

Ch25.7

# 總結 - 提取 ( Extract ) 與覆寫 ( Override )

上面的提取 ( Extract ) 與覆寫 ( Override ) 手法是物件導向程式中解決依賴的核心技術。

上面幾種方法從「子類別化並覆寫方法 ( Subclass and Override Method )」演化而來。( Ch 25.21 )

注意事項：

- 破壞封裝原則
- 手刻一堆只用於測試的類別
- 無法用於 `static` 和 `sealed` ( 在 C# 中代表禁止覆寫 ) 方法
- 測試涵蓋率會下降

# 總結 - 提取 ( Extract ) 與覆寫 ( Override )

## 使用時機：

看到被測試對象 ( SUT ) 中有很難解耦合的相依物件，又因為 Legacy Code 問題導致很難抽取介面和依賴注入，都可以擷取 ( Extract ) 相依部分，然後利用繼承與覆寫 ( Override ) 抽換，把不可控的相依物件改成可控的假物件。

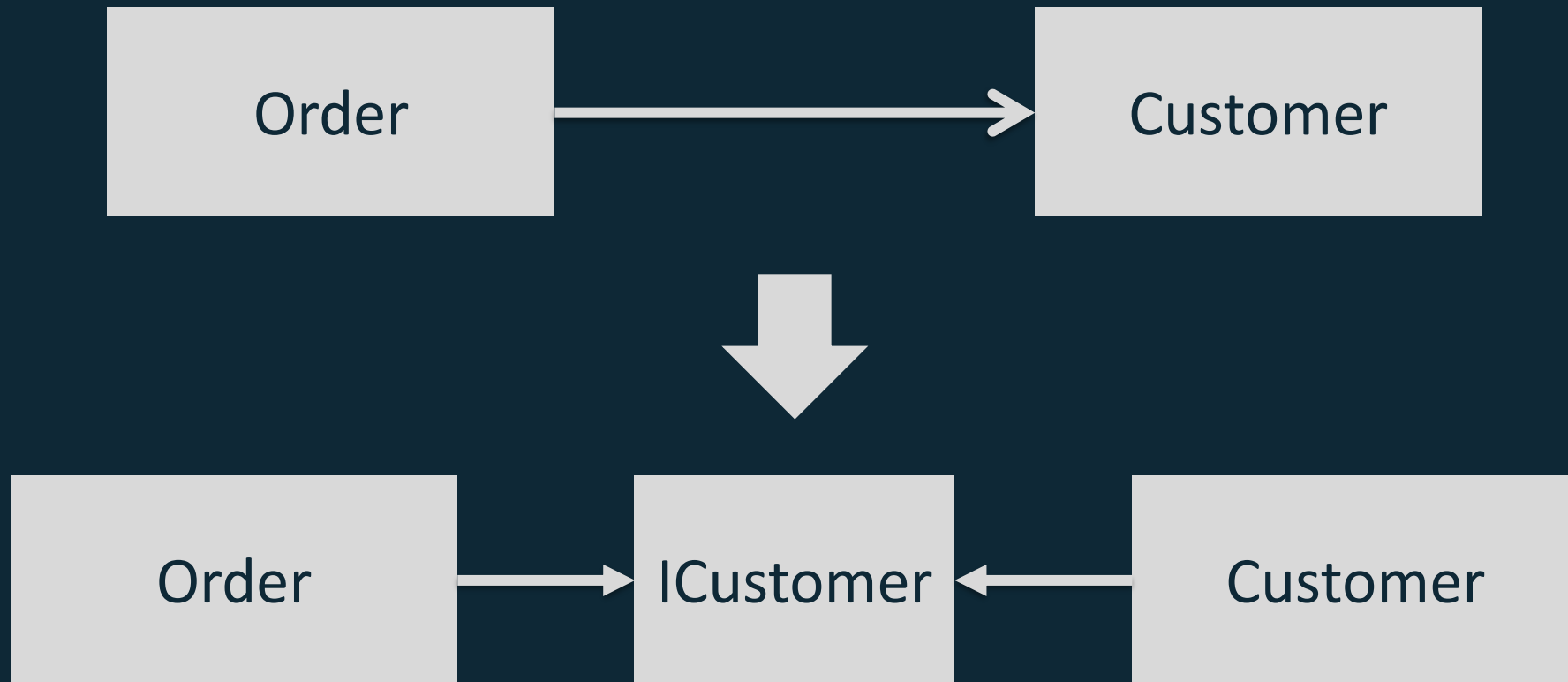
解決了依賴問題就能寫測試了。建議，這招只是讓你快速解耦合並套上 Unit Test，當有測試保護後，建議還是去建立 Interface 和使用 DI 技巧。

更多提取 ( Extract )



# 介面提取 ( Extract Interface )

採用正統的依賴抽象與依賴注入方式解耦合。



\* 記得要給介面好的命名

# 實作提取 ( Implemeter )

- Step 1：將 A 類別複製到另外一個檔案並給個好命名 B
  - 作者說 B 的命名通常是添加「Production」前綴。
- Step 2：A 只保留 public 方法。
- Step 3：A 改成抽象類別。
- Step 4：B 實作 A。

實作提取對解依賴沒有改善，只能發現原本使用 A 的都錯誤了，藉此逐一觀察使用端如何使用 A 的，然後逐一調整。

# 使用參數解決依賴問題

# 參數化建構子 ( Parameterize Constructor )

```
public class MailChecker
{
    public MailChecker(int checkPeriodSeconds)
    {
        this.receiver = new MailReceiver();
        this.checkPeriodSeconds = checkPeriodSeconds;
    }
}
```

①

依賴物件

使用參數傳遞解依賴

```
public class MailChecker
{
    public MailChecker(MailReceiver receiver, int checkPeriodSeconds)
    {
        this.receiver = receiver;
        this.checkPeriodSeconds = checkPeriodSeconds;
    }
}
```

②

# 參數化建構子 ( Parameterize Constructor )

優點：透過控制反轉快速解決依賴問題。

缺點：

- 強迫使用端傳遞額外參數給建構子。
- 將 new MailReceiver 依賴送給使用端。

```
public class MailChecker
{
    public MailChecker(MailReceiver receiver, int checkPeriodSeconds)
    {
        this.receiver = receiver;
        this.checkPeriodSeconds = checkPeriodSeconds;
    }
}
```

# 參數化建構子 ( Parameterize Constructor )

```
public class MailChecker
{
    public MailChecker(MailReceiver receiver, int checkPeriodSeconds)
    {
        this.receiver = receiver;
        this.checkPeriodSeconds = checkPeriodSeconds;
    }
}
```



```
public class MailChecker
{
    public MailChecker(int checkPeriodSeconds)
    {
        new MailChecker(new MailReceiver(), checkPeriodSeconds);
    }

    public MailChecker(MailReceiver receiver, int checkPeriodSeconds)
    {
        this.receiver = receiver;
        this.checkPeriodSeconds = checkPeriodSeconds;
    }
}
```

如果不是在建構子，在方法（ Method ）內也能使用將依賴提取至參數的技巧。

稱之為「參數化方法（ Parameterize Method ）」。

Ch25.15

再來討論一下全域變數



# 關於全域變數

將全域變數依職責與使用情境（譬如有些變數常一起出現）封裝到類別中。

記得其他有使用到全域變數的地方都要改。

以上手法稱之為【封裝全域參照（Encapsulate Global References）】。 **ch25.4**

然後就能透過「參數化方法」或「參數化建構子」前面講過方法解決依賴問題。

# 關於全域變數 – 靜態方法也算是全域變數一種嗎？

將靜態方法放到一個 Getter Method 中，然後覆寫他。

感覺和前面說到的「提取並覆寫呼叫 ( Extract and Override Call ) 」有 87% 像。

然後作者給他了一個專門的手法名稱，叫做「以獲取方法替換全域參照 ( Replace Global Reference with Getter ) 」。

**Ch25.20**

其他與「提取並覆寫呼叫（ Extract and Override Call ）」有 87% 像技術

# 特性提升 ( Pull up Feature ) 和 <sup>Ch25.17</sup> 依賴下推 ( Push Down Dependency ) <sup>Ch25.18</sup>

被測試對象 ( SUT ) 使用到自己類別中其他屬性或方法。要測試會很麻煩 ( 不是不行 ) 。

可以將被測試對象 ( SUT ) 與其相依屬性或方法「提升到抽象類別 ( A ) 」或將相依屬性或方法「下推到子類別 ( C ) 」，然後寫個假類別繼承 A / C 即可。

還記得前面的『時間』那一題嗎？那是抽到方法 ( method ) 然後寫個假類別覆寫。這邊也是雷同，假類別繼承 A / C 後就能覆寫想覆寫的東西了。

# 以下介紹簡單好用的一些技巧

# 暴露靜態方法 ( Expose Static Method )

## Context :

沒有測試，然而要調整某個 Method，能否快速補上測試後重構。

## Problem :

要補上測試的方法其類別難以實例化 ( new )。

## Force :

觀察發現本次要補上測試的 Method 沒有使用類別內的屬性或其他方法。

# 暴露靜態方法 ( Expose Static Method )

Example :

```
public class RSCWorkflow
{
    // 這個 class 很難實體化，寫出來投影片不夠放，所以刪除了。
    // 你們只要知道很難很難很難就對了。

    // 下面是我們這次對象
    public void validate(Packet packet)
    {
        if (packet.getOriginator().equals("MIA")
            || packet.getLength() > MAX_LENGTH
            || !packet.isValidChecksum())
        {
            throw new InvalidFlowException();
        }
        .....
    }
}
```

# 暴露靜態方法 ( Expose Static Method )

Solution :

```
public class RSCWorkflow
{
    // 這個 class 很難實體化，寫出來投影片不夠放，所以刪除了。
    // 你們只要知道很難很難很難就對了。

    // 下面是我們這次對象
    public static void validate(Packet packet)
    {
        if (packet.getOriginator().equals("MIA")
            || packet.getLength() > MAX_LENGTH
            || !packet.isValidChecksum())
        {
            throw new InvalidFlowException();
        }
        .....
    }
}
```



# 暴露靜態方法 ( Expose Static Method )

## Resulting Context :

- 使用端可能需要改成 static 。
- 使用端依賴此 static method ( 範例中的 static validate ) 。意思是未來使用端很難加上測試 。

# 到目前我們學了 13 個解依賴手法了

大家還記得多少呢？

- 提取並覆寫呼叫
- 提取並覆寫工廠方法
- 提取並覆寫獲取方法
- 實作提取
- 介面提取
- 子類別化並覆寫方法
- 參數化建構子
- 參數化方法
- 封裝全域參照
- 以獲取方法替換全域參照
- 特性提升
- 依賴下推
- 暴露靜態方法

# 記得要多練習才能熟能生巧

所以我們進入練習環節吧！

# 練習題



<https://tinyurl.com/yc8x9hao>