AspectW：剖面導向之例外處理與重試機制
AspectW: an Aspect-Oriented Catch and Retry Mechanism

高沛功
國立政治大學資訊科學系
Email: 101971018@nccu.edu.tw

陳　恭
國立政治大學資訊科學系
Email:chenk@cs.nccu.edu.tw

摘　要

雖然絕大多數的現代程式語言都有嚐試與補獲（try-and-catch）的例外處理機制，提供開發人員撰寫模組性高的例外處理程式碼，既可以立即處理例外狀況，也可以將例外傳遞（propagate）到系統其他模組。但是很多例外情況是暫態的（transient），發生後，應用程式是有可能從例外狀況恢復（recovery）過來，而不必啟動例外處理的程序。例如：分散式系統在進行網路連線時，可能因為網路一時不穩定而失敗，但稍停幾秒後再進行連線就可以了。於是就有學者倡議擴充例外處理機制，增加補獲與重試（catch-and-retry）的功能。本研究採用剖面導向（aspect-oriented）的觀點，參考 AspectF 的方法來實作一個輕量級的補獲與重試模組，AspectW，讓開發人員可以“流利（fluent）介面”的方式輕鬆撰寫例外捕獲與重試的程式碼，達到了讓開發人員不必更動主功能邏輯程式碼就能簡單就能加入補獲與重試的功能。

關鍵詞—Exception, Exception handling, Try-and-Catch、AOP、Catch-and-Retry

一、緒論

1.1 前言

補獲與重試機制並非是個新穎的觀念，但在雲端運算(cloud computing)時代，網路傳輸量急劇增加，量變可能造成質變，因此更值得開發人員注意。尤其是網際網路社交應用系統，如：Facebook 、Twitter 等大規模分散式系統，每日傳遞的訊息已達十億次數等級。原來百萬次等級偶發的例外故障，到了每日十億次規模後那就不是偶發而是“常常”了。對於提供服務的廠商若不處理這“常常”發生的故障，對產品品質形象的質變是會有危害的，一旦當客戶們從網路等媒體開始流傳品質不良的傳言，是會影響產品行銷的。

1.2 研究動機

補獲與重試機制在單機系統比較單純，因此在討論系統架構時常是被忽略的。它在大規模分散式系統下才會顯得出重要性。然而，許多開發人員在討論軟體系統演算法或建構系統架構時，通常不會把補獲與重試放在心上。甚至在建構系統雛型階段時，也不見得會仔細考量它的設計，但問題就往往出現在後續的實作到上線的階段，所以它的迫切性就被忽略了，以致系統上線後問題才開始。對於接手維護的開發人員來說，長期維護的系統程式碼就容易成為頭痛的問題，常為了加入一項機制而必須牽動許多的程式碼。這又讓程式碼糾纏的狀態更加的嚴重，整體狀況變成了負向循環，也讓程式碼設計意圖的可讀性與理解度不斷的下降。

1.3 研究目的

要讓程式碼在長期維護過程中不會互相糾結纏繞的一個方法就是導入剖面導向程式設計（Aspect-oriented Programming, AOP）。本研究最主要的目的之一就是透過 AOP 的縫合技巧，把補獲與重試機制加入到系統裡。在這研究過程有幾項重要目標：

(1) 應用 AOP 的特性，使系統易於開發與維護。
(2) 提供補獲與重試能力並俱有這兩項特性：隔離(isolation)、復原(recovery)。

補獲與重試機制不只是一個指令而已，也不是終端使用者會提出的需求，它關涉到一個軟體產品的品質問題。過去面對網路通訊故障時，開發人員常用的應對方法整理如下：

(1) Retry，補獲到故障後重新執行出錯的指令，過程中也可能適當的變更參數以利重試。
(2) Restore，補獲到故障後立刻進行還原。
(3) Ignore，有時有些故障是可以忽略的，比如：按讚遺失。在重試過幾次，最終還是失敗，這時忽略故障若不影響大局那就忽略吧。
(4) Handle Failure，進階的操作。補獲的故障無法以常用的方法應對時，就必需依當時狀況訂定應對的操作。

以上這四類功能是補獲到例外故障時常考量的應對行為：是否要重試？可以忽略不管嗎？故障使得應用程式狀態不正常嗎？要復原狀態嗎？常用的手法無效必須依個案加入特別的措施嗎？

1.4 研究成果

此研究主要的成果有：

(1) 重新思考 AOP，在觀點上更重視縫合(weaving)部份。
(2) 引入一種極輕量的 AOP 實作技術 AspectF。只需應用 lambda expression 的原理即可實作出來，不必採用複雜的 dynamic proxy、IL(Intermediate Language)等技術。
(3) 延用 AspectF[5]的核心概念，但重新設計出 AspectW 來提供捕獲與重試的功能。此名稱中的“W”代表著強調縫合(weaving)的能力。這是一個小型模組也可以用小工具來形容。
(4) 整理出補獲與重試機制的四個指令設計方向：retry、restore、ignore、handle-failure。

二、相關研究與技術背景

2.1 Catch-And-Retry

Catch-And-Retry[3,4]是一個常觀察到的真實現象。例如：瀏覽一個網址可能第一次失敗而在稍後幾秒再瀏覽相同網址卻成功了。透過同樣的SMTP 服務送出 email 第一次失敗，可是馬上再送出卻又成功了。特別是大規模分散式(large-scale distributed)的網路服務系統，必須對抗多重多樣性的例外故障，這些故障(fault)可以分成二類：

- 資料性錯誤(Data errors): 維持資料的新鮮度和一致性在分散式系統中的成本是昂貴的，甚至在某些狀況是不可能辦到的。
- 可用性錯誤(Availability errors): 也或許是因為網路分區(network partitions)，硬體故障(hardware failures)，性能跳針(performance stutters)，或軟體缺陷(software bug)的關係。

這些例外故障是無法根治的但有共同特性就是暫態的(transient),所以可以用 Catch-And-Retry 機制應對。它在功能面要能滿足幾種狀況：

- Basic Retry
  等待一小段額定時間後再重試，並指定重試上限次數以避免無限迴圈的危險。
- Parameterized Retry
  在重試前變更參數，有時不同的參數可滿足相同的目的，或許品質可能下降但可接受。
- Recovery From Multiple Exceptions
  可處理多重的例外，同一指令可能有多種可能的故障出現。
- Scheduling Control Over Retries
  對失敗的指令設定排程以重試。這在實務上是有條件的，該指令須俱有等冪(idempotent)特性否則隨意排程重試可能會破壞程序結構。

從規格來看要讓 Catch-And-Retry 機制運作，也就是重新執行 try 區塊時至少要保證二個要件：

(1) Restoring Application State
  應用程式狀態(variables, heap, stack, data structures, etc.)就是在那當下，在那第一次進入 try 區塊的當下。
(2) Isolation in try blocks
  該 try 區塊必須是隔離的(isolated)，這意味著如果執行外部的 I/O 操作，它必須能夠撤消它們或者它們必須是等冪的。

2.2 Think Again AOP

AOP(Aspect-oriented Programming[2], 剖面導向程式設計[1])的觀念是在 1997 年由 Xerox Palo Alto 實驗室所提出，此概念一提出立即引起程式語言與軟體工程方面學者與專家的重視與迴響。

學習 AOP 第一步就是先了解何謂 aspect，最早的定義是這樣寫的：

> *A component, if it can be cleanly encapsulated in a generalized procedure (i.e. objects, method, procedure, API).*
>
> *An aspect, if it can not be cleanly encapsulated in a generalized procedure.*

也是說 aspect 是指無法被清楚地判定可以封裝起來的程式碼。

為何要導入 AOP 呢？AOP 的研究背景是基於發現使用 OOP(object oriented programming)方法來開發軟體系統是不完美的，發現有些程式碼就是無法獨立不相依的封裝。在幾年開發與維護後最終程式碼總是會變成糾纏(tangled)狀態，即程式碼交互纏繞難以理解該段原始碼的邏輯與用意，不但維護難度隨時間增加，若要加入新的機制也是難上加難且還常有意想不到的例外狀況。這過程可用溫水煮青蛙迷思(boiling frog)[6]來形容。

解決方案則是提出了 cross-cutting concern 的概念。程式(program)分成元件(component)與剖面(aspect)兩個部份，再透過縫合(weave)技術組合起來。可以用下面式子描述：

Weave(*Component program, Aspect program*)
→ Program

從邏輯推理看 AOP

從數理邏輯推論來看 AOP。一份完整的業務流程程式碼可分成主功能邏輯與非主功能機制，在之間還存在一些鷹架碼做融合，其中鷹架碼是相依可以不記，用數學式字寫下：

業務流程 = 主功能 + 非主功能 ------ (公式 1)

AOP 的組成元素有三個，描述如下：

AOP∈{*主功能, 剖面功能，縫合區*} ------ (公式 2)

然而，非主功能部份大多是可重複使用的，最典型的就是 log，就可以轉化成元件或剖面功能來使用。寫成數學式子如下：

非主功能 → 剖面功能 ------- (公式 3)

要注意一點，非主功能並不等於剖面功能，而是轉化成剖面功能。剖面功能通常不能單獨存在，需要補入一些參數才能運作，就算是 log 也要指定紀錄內容。當然有些資訊可以由系統環境取得，但畢竟不是由自己本身產出。再重寫如下：

非主功能 = 縫合區 + 剖面功能 ------ (公式 4)

把公式 1 與公式 4 聯結起來就變成了：

業務流程 = 主功能 + 非主功能
= 主功能 + 縫合區 + 剖面功能
→ 主功能 + 縫合區 ----------(公式 5)

業務流程程式碼中的非主功能變成縫合區與剖面功能的組合。而剖面功能就像共用元件一樣可放到後端再引用即可。換句話說，開發人員只須撰寫主功能邏輯與縫合區程式碼即可。

2.3 AspectF

在研究了數種 AOP 的實作方法我們認為 AspectF[5]是比較適合我們的應用目標。其它的方法多以 AspectJ 為學習對象，其功能也隨改版更強大。而 AspectF 功能較精簡但好用，在使用語法導

入 Fluent Interface[7]設計，核心原理是 lambda expression，沒有 dynamic proxy、IL 等等複雜的動態技術。這個方法超乎想像的簡單卻能讓程式碼乾淨易懂且容易維護。

直接以使用案例來比較。現在比較受歡迎的 AOP 使用方式是貼上標籤，在 Java 稱為 annotation；在 C#稱為 attribute，如圖 1 所示，此例是定義一個函式表示新增一筆客戶資料並追縱與紀錄花費時間。貼籤標的方法簡單好用不過也在函數定義時期就被綁定了。

圖 2 是 AspectF 的使用範例，其剖面功能與主邏輯是一體的。此例不是定義一個函式，而是一串處理流程。在流程中可依需要橫切加入剖面功能。此例表示在流程中先新增一筆客戶資料過程中又做了一些處理然後更新客戶資料，在同時追縱與紀錄花費時間。這在一般的 AOP 方案是無法做到這一點的，此點特性可以稱做流利的(fluent)。

與一般 AOP 解決方案比較在特徵上的差異有:

- 不在類別的成員函數外面定義剖面而是在內部直接定義。
- 以函數實現剖面取代以類別實現的方式。

再從特性來看有以下優勢：

(1) 使得業務流程的剖面更清晰。
(2) 可以不用過度考慮性能的損失，因為它只是一個輕量級的類別。
(3) 剖面功能可以傳遞參數。有些 AOP 方案是不允許這麼做的。
(4) 甚至不能稱為框架因為它只是一個類別而已。
(5) 可以在流程程式碼的任何位置設置剖面，也能用於迴圈內與巢狀結構使用彈性極大。

我們做了一份比較表格比較 AspectJ 與 AspectF，請參考表 1。

三、系統設計與架構

3.1 設計理念與考量

在建置大規模分散式的網際網路應用服務，開發人員必需花費不少精力處理資料性故障與可用性故障，目的是讓系統不因一些意外而停止運作或讓使用者感覺上的認為“又當機了”。這些的故障大部份都是暫態的所以能用 Catch-And-Retry 機制來應對，而使用 AOP 實現是普遍認為是好的解法。

在考量上功能面部份要符合常用的流程模式：

Step1: 補獲失敗訊息。
Step2: 等待一小段額定週期。
Step3: 還原應用程式的不正常狀態。
Step4: 再重新執行。

非功能部份考量需求有：

(1) 流程程式碼不要因此而散開，以避免在流程上的設計意圖難以識別。
(2) 加入 Catch-And-Retry 機制的指令要儘可能簡潔，鷹架碼越少越好。

3.2 Catch-And-Retry 指令設計

在指令的設計上，依據論文探討[3,4]，在語法

```
[Log]
[TimeExecution]
public void InsertCustomer(string firstName, string lastName, int age)
{
   CustomerData data = new CustomerData();
   data.Insert(firstName, lastName, age, attributes);
}
```

圖 1　AOP 的一般範例

```
// Declare Instance
CustomerData data;

// Insert Customer Data
AspectF.Define
   .Log(Logger, "Inserting customer the easy way")
   .HowLong(Logger, "Starting insert", "Inserted in {1} sec.")
   .Do(()=>
   {
      data = new CustomerData();
      data.Insert(firstName, lastName, age);
   });

..... // Do something here

// Update Customer Data
AspectF.Define
   .MustBeNonNull(data)
   .Log(Logger, "Update customer profile")
   .HowLong(Logger,"Starting update","Updated in {1} sec.")
   .Do(()=>
   {
      data.Update(firstName, lastName, age);
   });
```

圖 2 AspectF 的一般範例

表 1　AspectJ 與 AspectF 比較表

	AspectJ	AspectF
主要特性	物件導向	程序導向
縫合功能比較	縫合功能強大。支援萬用字元，主功能與縫合區有多對多的縫合關係。	主功能與縫合區只能一對一的縫合。
使用比較	1) aspect class 2) annotation	weaving
擴充比較	advice class realize annotation	aspect function
長期維護	因為分散，長期需求變動後可讀性反而降低，並導致越來越難維護。	因為集中，長期可讀性可維持，維護難度持平。
主要技術	Spring dynamic-proxy byte-code weaving	lambda expression
學習曲線	相對較高	相對低很多

結構上有五個執行段落：

Try - Catch - Retry - Fail - Finally

到了試用階段發現要十個執行段落才好用：

Enter - Try - Catch - Restore - Wait - Recovery - Retry - Fail - Ignore - Leave

這十個段落一一說明如下：

- Enter，進入被包裹的程式碼區塊時觸發。
- Try，即被包裹的程式碼區塊。也就是主功能邏輯程式碼區塊。在 AspectW 由 Do 函式負責。
- Catch，補獲例外故障。
- Restore，當補獲故障後立即還原本地端狀態
- Wait，等待一小段額定的時間
- Recovery，復原操作，與 Restore 不同的是需自訂復原操作程式碼。
- Retry，重新執行失敗的指令。
- Fail，最終還是失敗時在此決定要如何處理。
- Ignore，忽視例外故障。
- Leave，離開被包裹的程式碼區塊時觸發。與關鍵字 finally 用意相同。

這十段再依使用頻率與情境再次調整，分成五個主要指令型，這些指令因應狀況有多載變形，以下說明這五個主要指令型：

(1) OnEnterLeave(Action enterAction,
Action leaveAction)

段落：Enter-Leave

說明：以 event 的形式來對待，在進入或離開被包裹的程式碼區塊時觸發。

(2) Retry<TException>(int retryDuration,
int retryCount)
Retry<TException>(int retryDuration,
int retryCount,
Action<TException> recoveryAction,
Action<TException> failAction)

段落：Try-Catch-Wait-[Recovery]-Retry

說明：這是最主要的重試指令因應狀況有多種變形。這裡是一個基本版與另一個進階版。

(3) Restore(params RestoreWhenFail[] restoreArgs)

段落：Restore

說明：還原本機端應用程式的狀態。其中類別 RestoreWhenFail 是輔助類別協助狀態還原。

(4) HandleFail(Func<Exception,
HandleFailMethod> failHandler)

段落：Fail

說明：設計來做進階操作，也就是當使用 Retry 指令無法滿足時才派上場。其中回傳型別 HandleFailMethod 指定處理方式，有 ThrowOut 向外傳遞或 Retry 重試或 Ignore 忽略。

(5) Ignore()

段落：Ignore

說明：如同其名設計來忽略例外故障

3.3 AspectW 語法與應用

用 EBNF 形式寫下 AspectW 的使用語法。首先用“AspectW.Define”進行宣示，再來先寫下 Do 敘述包裹好目標程式碼區塊，然後再一一加入需要的剖面功能也就是非主功能機制即可。

```
<AspectW-Syntax> ::= “AspectW.Define”
                     {“.”<Aspect-Function>}
                     “.Do(()=>{”
                          <Logic-Function-Block>
                     “});”
<Aspect-Function> ::= Action
<Logic-Function-Block> ::= {<Statement>}
```

指令應用的目標採用論文探討的情境為主，以語例形式來說明。

(1) 指令基本順序

在開始前還是要從基本功開始。剖面功能本身是沒有順序的，但實際執行時還是有順序關係，順序由外向內包裹機制；而例外由內向外引發。首先了解 Catch-And-Retry 機制完整十段落執行順序的指令下法。一開始應該會不習慣，多試幾次就會開始習慣並愛上了。

```
BarClass bar = new BarClass();
AspectW.Define
  .OnEnterLeave(FooEnterAction, FooLeaveAction)
  .Ignore<FooException>()
  .HandleFail(FooFailHandler)
  .Retry(3000, 2, (ex) => FooRecoveryAction())
  .Restore(new RestoreWhenFail(bar, (v)=>bar=(BarClass)v))
  .Do(() =>
  {
    bar.DoSomeChange();
  });
```

(2) 基本形式 Basic Retry

```
BarClass bar = new BarClass();
AspectW.Define
  .Retry(3000, 2)
  .Restore(new RestoreWhenFail(bar, (v)=>bar=(BarClass)v))
  .Do(() =>
  {
    bar.DoSomeChange();
  });
```

這語法表示：當執行主功能邏輯出現故障時立刻回復狀態，然後等待 3 秒再重新執行，重試的上限次數 2 次。

(3) 多重故障 Multiple Exception

```
BarClass bar1 = new BarClass();
BarClass bar2 = new BarClass();
AspectW.Define
  .Retry<InvalidDataException>(3000, 2)
  .Retry<ApplicationException>(3000, 2)
  .Restore(new RestoreWhenFail(bar1, (v) => bar1 = (BarClass)v)
           ,new RestoreWhenFail(bar2, (v) => bar2 = (BarClass)v))
  .Do(() =>
  {
    bar1.DoSomeChange();
    bar2.DoSomeChange();
  });
```

把 Retry 指令串接多次即可，關鍵點在透過泛型參數識別各類例外故障。

(4) 參數化重試 Parameterized Retry

```
BarClass bar4 = new BarClass();
string protocol = "HTTPS";
AspectW.Define
   .Ignore()
   .RetryParam(3000, () => protocol = "HTTP", () => protocol = "SMS")
   .Do(() =>
   {
      FooSendMessage(bar4, protocol);
   });
```

有時候換個參數就通了。比如一開始用較嚴謹的 HTTPS 協定傳送訊息但失敗，也許改用 HTTP 協定就通了，若還不行再改用備用方案 SMS，最終還是失敗的話可以考慮忽略故障。

(5) 發射導彈 Firing a Missile

```
BarClass bar3 = new BarClass();
RestoreWhenFail res3 = new RestoreWhenFail(bar3, (v) => bar3 = (BarClass)v);
AspectW.Define
   .Retry(3000, 3, (ex) => {
       FooFireCancelMissile(bar3); // firing a Patriot.
       res3.Restore(); }) // restore state
   .Do(() =>
   {
      bar3.DoSomeChange();
      FooFireUpdateMissile(bar3); // firing a missile.
   });
```

有些指令就像是發射導彈一樣是無法返悔的，這時候只能靠“愛國者飛彈”，再送出另一個補救指令請求取消、廢除或無效化剛發出的指令。

四、系統實作與展示

在設計與開發完成後理想上的展示應該在真正的目標系統上。而問題正在此處，AspectW 的 Catch-And-Retry 指令設計的應用目標是大規模分散式網際網路應用系統，這是展示與測試時都難以實現的，又若完全憑空想像也不對。考量後，論文探討時有提過一些真實的案例模型，那就以這些模型為應用目標。而比較對象是有採用 AspectW 與完全沒有採用 AOP 但邏輯上完全相同的程式碼。

4.1 案例比較 Parameterized Retry

論文探討時舉了幾個 AOP 應用案例，挑選了一個比較代表性的，不會太極端也太簡單。比較的重點放在程式碼糾纏的程度與可讀性。比較的方法是白箱測試，直接比較程式碼。

此例是我們開發出來測試用的程式碼片段，所以會有一些雜支細節。如圖 3 是有使用 AspectW 的例子。可以看出語法與設計目標相差不遠，程式碼相當精簡。在沒有下註解的狀況下依然可以清楚判斷各項設計意圖。一下就能在 Do 函式內找出主功能邏輯；而非主功能機制也就是剖面功能有三個，其中 WaitCursor 管制 UI 防呆用； Trace 做紀錄，在 CRM 應用可以留下使用歷程；RetryParam 參數化重試，若失敗就變更 protocol 參數再重試。

同樣一模一樣的邏輯不用 AOP 再寫一次程式碼如圖 4 所示。若沒有加註解的話實在難以看出各項設計意圖。若開發人員換了可能還會理解錯誤，尤其是 Retry 機制的程式碼相當分散在未來的維護只會更難。就我們而言，光是少寫很多程式碼就能完成相同邏輯這一點，就夠吸引人的了。

```
string protocol = "HTTPS";
AspectW.Define
    .WaitCursor(this, btnParamRetry)
    .RetryParam(3000,()=> protocol="HTTP",()=> protocol="SMS")
    .Trace(() => this.MyTrace("BEGIN : Parameterized Retry"),
           () => this.MyTrace("END\r\n"),
           (ex) => this.MyTrace("CATCH : {0}", ex.Message))
    .Do(() =>
    {
        _svc.SendMessage("John", "see you again.", protocol);
    });
```

圖 3 Parameterized Retry with AspectW

4.2 二個 Facebook 案例套用

接下來再引入二個論文探討 CatchAndRetry[3] 中提到的 Facebook 案例，雖只是模擬相同的需求邏輯，不過也更接近真實的應用狀況來使用 AspectW。這兩個案例是：

(1) Facebook Status Updates：

取得用戶他所有朋友們最新的狀態並更新到自己臉書的首頁。

用圖 5 來表示這個情境。用戶 Alan 的朋友們不定時的加入新的訊息到 Facebook，而 Alan 想消化他所有好友們的新訊息，在取得訊息期間若有好友的訊息取得失敗的話，就忽略掉吧，可以下一輪再取且還有很多其他好友的新訊息要消化。

圖 6 是這段測試用程代碼片段，其中有一些執行環境與資源模擬。在這段程式碼中可以看到 AspectW 是放在 foreach 迴圈內，這在一般的 AOP 方案是無法做到的。其中 Ignore 剖面用來忽略好友資訊取得故障的狀況，讓系統不因此而停擺。

```
try
{
    // OnEnter
    this.Cursor = Cursors.WaitCursor; //@ UI control, mouse
    btnParamRetry.Enabled = false;
    // parameters for doing retry
    string[] retryParamAry = new string[]{"HTTPS","HTTP","SMS"};
    int retryParamIndex = 0;
    string protocol = retryParamAry[retryParamIndex++];
    for (; ; ) //@ for retry mechanism.
    {
        try
        {
            // before
            this.MyTrace("BEGIN : Parameterized Retry");
            // Biz-Logic
            _svc.SendMessage("John", "see you again.", protocol);
            // after
            this.MyTrace("END\r\n");
            break; // success & leave
        }
        catch (Exception ex)
        {
            // exception tracing
            this.MyTrace("CATCH : {0}", ex.Message);
            //@ for retry mechanism.
            if (retryParamIndex < retryParamAry.Length) {
                Thread.Sleep(3000);
                protocol = retryParamAry[retryParamIndex++];
                continue; // retry
            }
            throw; // fail & leave
        }
    }
}
finally
{
    // OnLeave
    this.Cursor = Cursors.Default; //@ UI control, mouse
    btnParamRetry.Enabled = true;
}
```

圖 4 Parameterized Retry without AOP mechanism

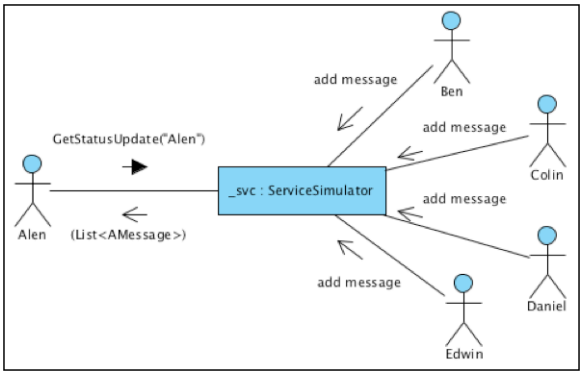

圖 5　Facebook Status Updates - scenario

```
public List<AMessage> GetStatusUpdate(string name)
{
  List<AMessage> result = new List<AMessage>();
  Person[] f = friends.Get(name);
  foreach (Person p in f)
  {
    AspectW.Define
      .Ignore<IOException>()
      .TraceException<IOException>((ex)=>
_logger.WriteLine("IGNORE : " + ex.Message))
      .Do(()=>
        result.Add(statusUpdates.Get(p))
      );
  }
  return result;
}
```

圖 6　Facebook Status Updates with AspectW

(2) Organizing a Facebook Event：

情境如下：Alan 辦了一場派對想邀請所有好友來同歡但因實際住的距離只能過濾出跟自己同一城市 Taipei 的好友來參加。用圖 7 來表達此情境。

把關鍵的程式碼截取於後，如圖 8。這段程式碼已應用了 AspectW 仍讓人有糾纏的感覺。若連這類簡化過測試用的案例都能讓人有程式碼糾纏的感覺，那們未導入 AOP 的程式碼纏繞的複雜程度肯定遠大於此。

這個例子也是前面數個範例的綜合應用，Basic Retry 、Parameterized Retry 、Recovery From Multiple Exception 同時使用。原始碼中的 WhenTrue 剖面有 filter 效果。一般的流程程式碼若加入多項非主功能機制後，想找出主功能邏輯通常變成了開發人員的一個挑戰，使用 AspectW 後只要找出 Do 函式就行了，非主功能機制也在縫合區一條條的列好。這一案例也說明 AspectW 與流程是一體的，在函式呼叫方與被呼叫方都可以使用，迴圈與巢狀結構也有支援。

這是個複合性也是較接近真實狀況的案例。雖是以 Facebook 為標的，其實在各類的社交應用都有類似的操作行為。它們的程式碼大概都有一個特性，就是非主功能機制比主功能邏輯還多，會有這現象主要是對品質的要求。

五、結論與建議

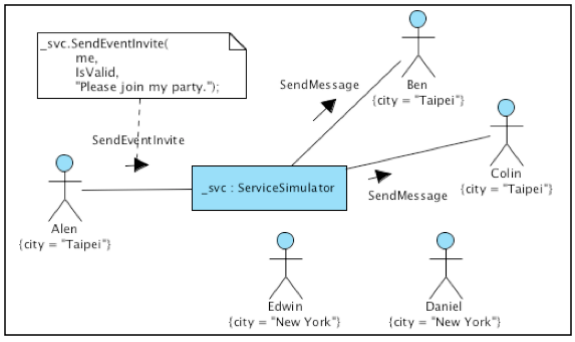

圖 7　Organizing a Facebook Event – scenario

```
AspectW.Define
  .WaitCursor(this, btnComposite)
  .Trace(() => this.MyTrace("BEGIN:SendEventInvite"),
         () => this.MyTrace("END\r\n"))
  .Do(() =>
  {
    Person me = _svc.GetPerson("Alan");
    _svc.SendEventInvite(me, IsValid,
                         "Please join my party.");
  });
//-----------------------------------------------
public void SendEventInvite(Person me,
                    Func<string, bool> isValid,
                    string message)
{
  foreach(Person friend in friends.Get(me))
  {
    AspectW.Define
      .OnLeave(() => _logger.WriteLine("Done"))
      .Retry<DataException>(3000, 5,
        (ex)=> RefreshAddress(friend), // recovery
        (ex)=> _logger.WriteLine("5 retries failed"))
      .TraceException((ex) =>
_logger.WriteLine("CATCH@L1: {0}", ex.Message))
      .Do(()=>
      {
        string address = friend.GetAddress();
        string protocol = "HTTPS";
        AspectW.Define
          .WhenTrue(isValid(address))
          .RetryParam<ApplicationException>(3000,
                ()=> protocol="HTTP" )
          .TraceException((ex) =>
_logger.WriteLine("CATCH@L2: {0}", ex.Message))
          .Do(()=>
            SendMessage(friend, message, protocol)
          );
      });
    }
}
```

圖 8　Organizing a Facebook Event with AspectW

5.1 結論與心得

Catch-And-Retry 與 AOP 搭配應用的研究並非新穎的研究項目，在這個研究裡應用了來自 AspectF 中一個重新看待 AOP 的觀點，實現方向上更重視縫合(weaving)的觀點。這一點差異也造成特性上的極大差異。這差異請參考表 1。

在多次應用 AspectW 組織程式碼的過程發覺一件事，組織程式碼的思考邏輯變成了二維式。在

這之前寫程式就是由上往下寫，這是一個方向一個維度的組織程式碼過程。比較於應用 AspectW 就不一樣了，寫程式時由上往下只須考慮主功能邏輯即可，而非主功能機制則是額外再橫切進去縫合，過程可用由左向右形容。縫合過程感覺像是以另一維度來組織程式碼的組成邏輯。

5.2 未來發展

我們提出幾個目標，以便在未來的研究工作中繼續加強功能。

一、引入 STM(Software Transactional Memory)若未來技術成熟的話。支援 Catch-And-Retry 機制的幾個規格項目之一是應用程式交易機制，這是另一項獨立的研究主題以 STM 為主。在設計 AspectW 時發現並不成熟故未採用。

二、主功能與非主功能狀態交換。這一點在一般的 AOP 方案是沒有的，但在多次試用 AspectW 縫合剖面時發現似乎是需要的。

三、程式語言直接支援縫合的關鍵字與語法這樣語法可以更精簡。

參考文獻

[1] 陳恭,「剖面導向程式設計(AOP/AOSD)簡介」website:http://www.cs.nccu.edu.tw/~chenk/AOP-intro.pdf

[2] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, Aspect-oriented programming, in ECOOP '97 Object-Oriented Programming 11th European Conference," Finland (M. Aksit and S. Matsuoka, eds.), vol. 1241, pp. 220-242, New York, NY: Springer-Verlag, 1997. website:http://www.cs.ubc.ca/~gregor/papers/kiczales-ECOOP1997-AOP.pdf

[3] Emre Kıcıman, Benjamin Livshits, Madanlal Musuvathi (October 2009). CatchAndRetry: Extending Exceptions to Handle Distributed System Failures and Recovery, Microsoft Research. website:http://research.microsoft.com/en-us/um/people/livshits/papers/pdf/plos09.pdf

[4] Bruno Cabral, Paulo Marques (n.d.). Implementing Retry - Featuring AOP. 4th Latin-American Symposium on Dependable Computing (LADC'09), September 2009 website:http://pmarques.dei.uc.pt/papers/bcabral_pmarques_ladc09.pdf

[5] Omar Al Zabir (11 Jun 2011). AspectF Fluent Way to Add Aspects for Cleaner Maintainable Code. website:http://www.codeproject.com/Articles/42474/AspectF-Fluent-Way-to-Add-Aspects-for-Cleaner-Main

[6] Matthew D. Groves, AOP in .NET chapter 1,2 - Practical Aspect-Oriented Programming, Copyright 2013 Manning Publications. website: http://www.manning.com/groves/

[7] Fluent interface, wikipedia website:http://en.wikipedia.org/wiki/Fluent_interface