# 第二章 背景知識與相關研究

* 1. 例外處理壞味道
     1. Empty Catch Block

定義:

「Empty Catch Block」此壞味道意旨當程式發生例外並捕捉例外後並忽略此理外。

影響:

此壞味道的例外處理機制方式為不處理，此作法會隱藏潛在問題，會使開發者往後若遇到例外狀況發生，增加除錯的困難度。

範例:  
如下圖所式，第9行writeFile函式會創立一個新檔案接著把資料寫入該檔案中。根據Java 的官方 API文件說明，第12行的new FileWriter(“k:\\test.txt”) 有可能會發生IOException。若程式執行到第12行時發生IOException，此例外將會被第16行的Catch Block接住，但Catch Block 並未對其做處理，最終此壞味道將被忽略導致開發人員除錯困難度將會提升。



* + 1. Dummy Handler

定義:

「Dummy Handler」當程式發生例外並捕捉例外後，處理方式只印出或紀錄例外訊息，而沒有實質做處理，而如果在印出或紀錄壞味道的同時，也有將其例外丟出，則不算是壞味道。  
影響:

此壞味道的例外處理機制方式為印出或紀錄例外，其效果也幾乎等同忽略例外，在程式執行時，開發者和使用者很難觀察到這些訊息。

範例

如下圖所式，第12行 writeFile函式會創立一個新檔案接著把資料寫入該檔案中。根據Java 的官方 API文件說明，第12行的new FileWriter(“k:\\test.txt”) 有可能會發生IOException。若程式執行到第12行時發生IOException，此例外將會被第16行的Catch Block接住，但Catch Block 僅記錄錯誤訊息。



* + 1. Nested Try Statement

定義:

「Nested Try Statement」此壞味道的意旨在程式碼中存在著巢狀的Try Block。

影響:

對開發者來說，此壞味道複雜的結構將會影響程式碼的可讀性、測試性以及維護性。

範例：如下圖所示，在第17行的Finally Block 會進行資源釋放的工作，然而很多關閉資原的函數都會丟出例外，用來代表釋放資源失敗。因此在Finally Block中很容易發生巢狀 try statement 的情況，使得程式碼結構變得複雜及難以維護。



* + 1. Unprotected Main Program

定義:  
「Unprotected Main Program」此壞味道的意旨在程式碼中，主程式或執行緒沒有捕捉由下傳遞至自己身上的例外。

影響:  
當程式執行時發生例外，主程式或執行緒沒有捕捉由下傳遞至自己身上的例外，則主程式或執行緒會不預期的終止或產生錯誤。

範例  
如下圖所示



* + 1. Careless Cleanup

定義:

「Careless Cleanup」此壞味道的意旨在釋放資源前，發生例外導致資源無法被正常釋放。

影響

此壞味道因為在釋放資源之前發生例外導致資源無法正常被釋放或關閉，將導致資源耗盡並降低系統穩定度。

範例:

如下圖所示，此writeFile函式會創立一個新檔案接著把資料寫入該檔案中，並釋放資源。根據Java 的官方 API文件說明，此函示在第14行 FileWriter函示、第15行 write、第16行flush均會丟出IOException，這些例外若發生在第17行釋放資源之前將會導致此close沒有被執行，導致資源持續被占用無法釋放。



* + 1. Exception Thrown From Finally Block

定義:

「Exception Thrown From Finally Block」， 此壞味道的特徵是在Finally Block中發生例外且此例外也被往外丟。

影響:  
此壞味道發生例外時，在 Finally Block 發生的例外會覆蓋 Try Block 或 Catch Block 所發生的例外，覆蓋的例外處理訊息會誤導程式人員，難以得知該例外是由哪一個block所造成。

範例:

如下圖所示，此writeFile函式會創立一個新檔案接著把資料寫入該檔案中，並釋放資源。根據Java 的官方 API文件說明，此函示在第14行 FileWriter函示、第15行 write、第16行flush均會丟出IOException，此例外會被17行的Catch Block接住並往caller回傳此例外，但是再回傳之前會先進入19行的Finally Block，若此時21行的close函示也發生例外，此例外將會被22行的Catch Block接住並在23行將此例外往外丟，此時這個丟出去的例外將會把19行的例外覆蓋，導致開發者無從得知例外完整的例外狀況。



2.2 Robusta

Robusta 為一個以Java開發的開源專案，是一個靜態程式碼分析工具，主要的功能是用來協助開發人員偵測出其Java程式中潛藏的例外處理壞味道，來幫助開發者找出程式中的例外處理壞味道。只要在eclipse 中的marketplace 正確安裝Robusta，就能使其偵測目前已定義的六種例外處理壞味道，分別為Empty Catch Block、Dummy handler 、Nested Try statement、Unprotected Main Program、Careless cleanup，除此之外還可以藉由產生報表的方式，迅速的定位出例外處理壞味道的位置以及壞味道的數量，降低人工檢查大量程式碼的成本。

2.3 AspectJ

AspectJ 是一種基於Java實做AOP(Aspect-Oriented Programming)的程式語言，

開發人員可以在不改動原有的程式碼下，額外增加原始碼的行為或改變狀態，  
再透過 before、after、around 決定要嵌入程式的時機。

將設計完的AspectJ程式與原始碼一起編譯並執行後，當程式執行到被鎖定的函式時，就會在特定的時機觸發AspectJ 嵌入程式碼。本研究將會以AspectJ搭配單元測試暴露例外處理壞味造成的影響。

# 第三章 研究方法

根據陳友倫***以Aspect揭露導因於例外處理的程式缺陷***的論文，該論文利用利用AspectJ的特性，在不更動原有程式碼下，在特定的程式碼中嵌入發生例外的程式碼，強制使其產生例外狀況，進而呈現壞味道對於軟體的影響。其研究結果發現，利用Robusta產生的AspectJ程式碼可以讓例外處理壞味道對於軟體的影響被暴露出來。

本論文將從陳友倫的研究為基礎，將其提出的方法應用於Robusta定義的壞味道中，讓所有的壞味道都能產生對應的AspectJ程式碼，藉由AspectJ在特定程式碼的位置嵌入例外藉此呈現出因例外處理壞味道所造成的影響。

3.1暴露壞味道分析

3.1.1不同壞味道暴露其例外的方式

下表為 Robusta 目前定義的壞味道以及對應的壞味道定義，此章節將針對定義的壞味道逐一設計其暴露例外的方式，藉此呈現出壞味道對系統的影響

|  |  |
| --- | --- |
| 壞味道種類 | 壞味道定義 |
| Dummy Handler | 當程式發生例外並捕捉例外後，處理方式只印出或紀錄例外訊息，而沒有實質做處理 |
| Empty Catch Block | 程式發生例外並捕捉例外後，沒有對此例外進行處理，亦即Catch Block 內為空的 |
| Exception Thrown From Finally Block | 此壞味道的特徵是在Finally Block中發生例外且此例外也被往外丟 |
| Careless cleanup | 此壞味道的意旨在釋放資源前，發生例外導致資源無法被正常釋放 |
| Unprotected Main Program | 此壞味道的意旨在程式碼中，主程式中的程式碼並沒有被Try Block 包覆起來入例外的類別。下一步我們將回到Try Block中，尋訪Try Block 中的 |

**Dummy Handler:**

「Dummy Handler」壞味道為當程式發生例外並捕捉例外後，處理方式只印出或紀錄例外訊息，而沒有實質做處理。接下來介紹如何在Dummy Handler的狀況下，讓此壞味道對於軟體的影響現行(圖7 第14~16行為Dummy Handle壞味道)。首先我們將鎖定該壞味道的Catch Block，並依據Catch Block 接取的例外做為要強制拋出例外的類別，以下圖7為例，此壞味道的Catch Block 接取的例外類別為 IOException，因此我們將用此例外類做為嵌入的例外類型，下一步將回到Try Block中尋訪所有的Method Invocation，並鎖定最先執行且與嵌入例外為相同類型的Method Invocation(fw.write())，將fw.write()視為即將丟出例外的點並嵌入IOException，藉由這種方式就可以將Dummy Handler這個壞味道所帶來的影響給揭露出來。



圖7 Dummy Handler 例外嵌入範例

**Empty Catch Block**

「Empty Catch Block」壞味道意旨當程式發生例外並捕捉例外後，沒有對此例外進行處理，亦即 catch black 內為空的。因而隱藏潛在問題。  
 接下來介紹如何在Empty Catch Block的狀況下，讓此壞味道對於軟體的影響現行(圖8 第14~15行為Empty Catch Block壞味道)。首先我們將鎖定該壞味道的Catch Block，並依據Catch Block 接取的例外做為要強制拋出例外的類別，以下圖8為例，此壞味道的Catch Block接取的例外類別為 IOException，因此我們將用此例外類別做為嵌入例外的類別。下一步將回到Try Block中尋訪所有的Method Invocation，並鎖定最先執行與嵌入例外為相同類型的Method Invocation(fw.write())，並將fw.write()視為即將丟出例外的點並嵌入IOException，藉由這種方式就可以將Empty Catch Block這個壞味道所帶來的影響給揭露出來。



圖8 Empty catch block例外嵌入範例

**Exception Thrown From Finally Block:**

「Exception thrown From Finally Block」壞味道意旨當在Finally Block中發生例外且此例外也被往外丟，因此在 Finally Block 發生的例外會覆蓋 Try Block 或 Catch Block 所發生的例外，覆蓋的例外處理訊息會誤導程式人員，難以得知該例外是由哪一個block所造成。

接下來介紹如何在Exception Thrown From Finally Block的狀況下，讓此壞味道對於軟體的影響現行(圖9第27行為Exception Thrown From Finally Block壞味道)。為了要呈現這個壞味道帶來的影響，我們需要製造兩個例外，這樣才可以重現出這個壞味道帶來的影響，首先我們將鎖定該壞味道的Finally Block，找尋在Finally Block第一個會丟出例外的Method Invocation(fw.close())，如下圖9所示，並將fw.close()視為即將丟出例外的點並對其嵌入IOException，如此一來我們就可以重現從Finally Block 丟出例外的狀況了，接下來我們需要找尋Try Block第一個會丟出例外的Method Invocation(fw.write())，下圖9所示，將fw.write()視為即將丟出例外的點並對其嵌入例外，以上步驟都完成後我們就可以嵌入兩個例外，並運用此方式將Exception Thrown From Finally Block這個壞味道所帶來的影響給暴露出來。



圖9 Exception Thrown From Finally Block例外嵌入範例

**Careless cleanup:**

「Careless cleanup」此壞味道的意旨在釋放資源前，發生例外導致資源無法被正常釋放。接下來我們接介紹如何在Careless cleanup的狀況下，讓此壞味道對於軟體的影響現行(第xx行為 Careless cleanup壞味道)。為了要呈現這個壞味道帶來的影響，我們需要再釋放資源的Method Invocation前遷入例外，在釋放資源之前若發生例外，則在此Try Block內的釋放 Method Invocation 則不會被執行，讓資源無法正常被釋放，這樣才可以重現出這個壞味道帶來的影響，首先我們將鎖定該壞味道的 Try Block，並找尋此Try Block 內的釋放資源Method Invocation，以此點做為起點，往使用資源的物件宣告點尋找，找出離釋放資源Method Invocation最遠且有可能發生例外的Method Invocation視為即將丟出例外的點並對其嵌入例外，並運用此方式將 Careless cleanup這個壞味道所帶來的影響給暴露出來，最後我們要蒐集 xx method name、method name、exception type、class name、method類型，以便生成AspectJ程式。

**Unprotected Main Program:**

「Unprotected Main Program」此壞味道的意旨在程式碼中，主程式或執行緒沒有捕捉由下傳遞至身上的例外，因此當未被捕捉的例外往上傳遞，最終傳到主程式或執行緒而導致程式不預期的終止執行。接下來介紹如何在Unprotected Main Program的狀況下，讓此壞味道對於軟體的影響現行(圖11第9~11行為Unprotected Main Program壞味道)。首先將以main function中第一個Method Invocation(UIManger.setLookAndFeel())視為即將丟出例外的點並對其嵌入例外，為了呈現這個壞味道帶來的影響，我們採用的作法為不管Method Invocation是否會發生例外，直接指定嵌入的例外為RunTimeException，並運用此方式將 Unprotected Main Program這個壞味道所帶來的影響給暴露出來。



圖11 Unprotected Main Program例外嵌入範例

3.1.2 一般壞味道呈現方式

根據陳友倫***以Aspect揭露導因於例外處理的程式缺陷*** 的研究，在尚未使用AspectJ來暴露例外之前，若想要在執行階段讓例外發生，如下圖12，為了暴露出壞味道帶來的影響，必須強制寫入會丟出例外的函式，但強制寫入會丟出例外的函式後，接續的程式碼編譯器則會跳出unreachable code 提醒使用者該行程式碼因為被強制丟出例外的關係，所以將不會被執行且無法編譯。

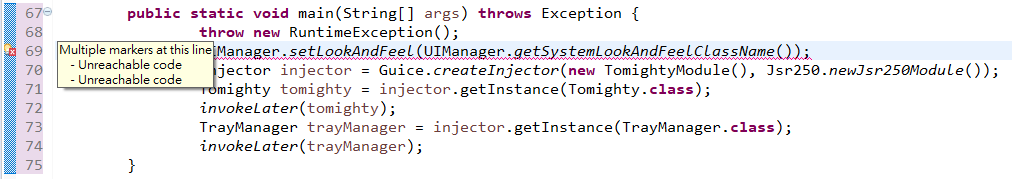


圖12

如圖12所示，程式以經無法正常被執行，若要讓程式可以正常被執行，以圖13為例，需要將強制寫入例外的程式碼下方的程式用註解的方式註解掉。

圖13

為了讓程式碼可以回到尚未揭露例外之前的結構，如圖14，需要把強制的例外程式移除並把註解的部分拿掉，如此一來才可以回到原有的功能才能正常被執行。

圖14

**3.2利用AspectJ呈現壞味道的方法**

到目前為止，我們知道程式碼中若有壞味道存在時，通常就是程式有bug存在的表徵。這些潛伏於例外處理中的不正確程式碼，經常不容易在測試的時候被發現，因為例外處理行為往往正是被理解的最少且測試程度最為不足的部分。

為了讓例外處理壞味道的影響可以呈現，根據陳友倫***以Aspect揭露導因於例外處理的程式缺陷***的論文研究，該論文以AspectJ來強制嵌入例外，進而揭露出壞味道對於軟體的影響。本論文基於陳友倫的研究，將原有只針對Dummy Handler壞味道的aspect程式碼實作，增加其他四種壞味道aspect實作及設計，分別為: Empty Catch Block、Unprotected Main Program、Careless cleanup及Exception Thrown From Finally Block。

3.2.1AspectJ使用說明

AspectJ是一種實作Aspect Oriented Programming(AOP)技術的程式語言，允許在既有的Java程式內動態插入新的程式結構，如此我們便能夠在不改變原始碼的情況下，把程式的正常行為與例外行為加以分隔，

3.2.2AspectJ觸發時機

根據AspectJ的特性，若AspectJ程式碼符合語法規則且指定將嵌入程式碼的對象、時機、作用範圍，就可以達到嵌入程式碼的動作。當指定的Method Invocation被執行時，AspectJ程式碼就會被啟動並達到嵌入程式碼的動作，本論文的啟用時機以陳友倫***以Aspect揭露導因於例外處理的程式缺陷***的論文實作做改善及調整。

陳友倫***以Aspect揭露導因於例外處理的程式缺陷***的論文實作部分，觸發嵌入例外的時機為指定的Method Invocation被執行時，AspectJ程式碼就會被啟動並達到嵌入程式碼的動作，但這樣的啟動時機點會造成不管任何情況下都會嵌入程式碼並丟出例外，會導致正常的程式碼無法啟動，所以需要在AspectJ的程式碼中而外加入一個switch的開關來手動控制例外是否要被丟出。  
 本論文根據陳友倫***以Aspect揭露導因於例外處理的程式缺陷***的論文實作，將其觸發時機限定於執行測試時自動啟動，就不需要手動去調整該壞味道是否否要觸發AspectJ來嵌入例外。

3.2.3 AspectJ 依據不同壞味道的材料收集

當Robusta完成掃描後，我們可以得知例外處理壞味道的範圍(如圖中螢光筆畫線的位置)，Robusta會將壞味道的資訊紀錄在iMarker中(圖中的燈泡位置) ，當我們按下Expose bad smell 選項後Robusta會針對選定的壞味道案例蒐集資料，而蒐集資料的流程，下一個小節將針對不同的壞味道進行介紹和分析。

**Dummy Handler & Empty Catch Block**

1.取得壞味道的範圍

根據Robusta的功能，在掃描完壞味道後，Robusta會將壞味道的程式碼畫上螢光筆來告知使用者壞味道的範圍，此外Robusta會將壞味道的相關資訊記錄在該壞味道的iMarker中(圖中的燈泡)，iMarker中記錄了該壞味道的Method Invocation和 Method Declaration 如此一來，就可以蒐集到產生AspectJ的相關素材了。

2.取得被標記的Catch Block 所捕捉的例外類別

根據Dummy Handler壞味道 Robusta 會在 Catch Block的地方標記此處為Dummy Handler壞味道，藉由取得標記的iMarker資訊，可以得知該Catch Block捕捉的例外類型。

3.取得會發生與Catch Block中相同例外的 Method Invocation

取得Catch Block捕捉的例外類別，接下來尋找Try Block中，一樣也會丟出與該壞味道相同例外類型的Method Invocation，用來當作嵌入對象。

4.分析 Try Block 找出先被執行且與該Catch Block相同例外類型的 Method Invocation

根據上一個了流程，我們若在Try Block中找到許多與Catch Block中會接住相同例外的Method Invocation，只要針對第一個會丟出與catch block例外一致的Method Invocation 當做嵌入對象即可。取得該Method Invocation得時做方法為利用Visitor 將 Try Block中的Method Invocation都走訪一遍，Visitor在尋訪每個ASTNode 時，能夠取得其程式結構，我們可以利用裡面結構的資訊，來判定是否會丟出同類型的例外。

5.解析Method Invocation 相關資訊

根據上一個步驟，取得第一個會丟出與catch相同例外且會被執行的Method Invocation，接下來我們將進一步解析，取得該Method Invocation的函式名稱、丟出例外類型。

6.解析Method Declaration 相關資訊

接下來要把持有Method Declaration的類別名稱、Method Declaration名稱等相關資訊記錄下來。

7.組裝AspectJ 程式碼

完成以上步驟之後，將蒐集好組成AspectJ的素材根據Dummy handler的特性下去拼裝，如同下圖程式碼所示。

**2.Exception Thrown From Finally Block**

1.取得壞味道的範圍

根據Robusta的功能，在掃描完壞味道後，Robusta會將壞味道的程式碼畫上螢光筆來告知使用者壞味道的範圍，此外Robusta會將壞味道的相關資訊紀錄在該壞味道的iMarker中(圖中的燈泡)，iMarker中記錄了該壞味道的Method Invocation和 Method Declaration 如此一來，就可以蒐集到產生AspectJ的相關素材了。

2.取得被標記的Finally Block內可能會丟出的例外

根據 Exception thrown from Finally Block壞味道 Robusta 會在 Finally Block的地方標記此處為 Exception thrown from Finally Block壞味道，因此我們就可以知道若要暴露這個壞味道帶來的影響，我們需要在AspectJ嵌入哪種例外，讓其例外發生。

3.分析Try Block 找出先被執行且會丟出例外的Method Invocation

Try Block第一個會丟出例外的Method Invocation，將此鎖定的Method Invocation視為嵌入對象即可。取得該Method Invocation得時做方法為利用Visitor 將 Try Block中的Method Invocation都走訪一遍，Visitor在尋訪每個ASTNode時，能狗取得該Node的程式結構，我們可以利用裡面結構的資訊，來得到該Node是否會丟出例外以及他的例外類型。

5.解析Method Invocation 相關資訊

根據上面的步驟，我們已經取得兩個 Method Invocation分別為Finally Block中第一個會丟出例外的函式、Try Block中第一個可能會丟出例外的Method，在此步驟我們進一步解析，取得這兩個Method Invocation的函式名稱、丟出例外類型。

6.解析Method Declaration 相關資訊

接下來我們要把Method Declaration的類別名稱、Method Declaration名稱等相關資訊。

7.組裝AspectJ 程式碼

完成以上步驟之後，將蒐集好組成AspectJ的素材根據Exception Thrown From Finally的特性下去拼裝，如同下圖程式碼所示。

**3.Unprotected Main Program**

1.取得壞味道的範圍

根據Robusta的功能，在掃描完壞味道後，Robusta會將壞味道的程式碼畫上螢光筆來告知使用者壞味道的範圍，此外Robusta會將壞味道的相關資訊在該壞味道的iMarker中(圖中的燈泡)，iMarker中記錄了該壞味道的Method Invocation和 Method Declaration 如此一來，就可以蒐集到產生AspectJ的相關素材了。

2.找出會丟出例外的Method Invocation

首先分析main 裡面所有的Method Invocation，情況分成兩個，分別是完全沒有被try程式包覆的程式碼及部分程式碼被try包住的部分，若程式碼均沒有被程式碼包住的話，我們將用Visitor 將Main Program中的Method Invocation都走訪並把所有會丟出例外的結構都記錄下來，Visitor 在尋訪每個ASTNode 時，能夠取得該Node的程式結構，並全部以RunTimeException做嵌入的動作。若是部分有被try包覆住的程式碼，我們一樣也是用Visitor的方式去走訪每個ASTNode，但這次我們會針對Try Block之外最先被執行到的Method Invocation做為嵌入的對象，並全部以RunTimeException做嵌入的動作。

5.解析Method Invocation 相關資訊

根據上一個步驟，我們已經取得沒有被Try Block 所包覆住的Method Invocation，接下來我們將進一步解吸，取得該Method Invocation的函式名稱。

6.解析Method Declaration 相關資訊

接下來我們要把Method Declaration的類別名稱、Method Declaration名稱等相關資訊。

7.組裝AspectJ 程式碼

完成以上步驟之後，我們就蒐集好組程AspectJ的素材，將其根據Unprotected Main Program的特性下去拼裝，如同下圖程式碼所示。

3.3驗證AspectJ程式碼功能

當我們把按照以上的收及步驟把AspectJ的素材收集完成後，接下來要來驗證我們的AspectJ是否可以運作並幫我們把例外做嵌入的動作，有兩個步驟可以幫我們達到，首先我們要前往eclipse market place 安裝 AspectJ Plugin即可.這一個plugin可以提供AspectJ的運行環境，如下圖所示，當安裝完畢之後，如果AspectJ的程式碼正確，第x行的around關鍵字的行數旁邊會出現AspectJ的標籤，除了AspectJ檔案之外，原始碼中會依據組合出來的AspectJ的規則嵌入程式碼，該段被AspectJ嵌入的程式碼也會有AspectJ的標記，這兩個標記能夠證明產生出來的AspectJ與我們的原始碼相呼應且規則相通，至於AspectJ是否符合我們預期的幫我們暴露出壞味道帶來的影響及丟出例外，接下來我們將利用單元測試的方式來幫我們重現因為例外處理壞味道所帶來的影響。

3.4利用測試案例來達到錯誤重現

本論文主要是利用AspectJ的技術在程式內的指定位置插入特定型態的例外，以激發程式的例外處理行為，並利用測試的方式來重現，下我們會針對不同的壞味道來進行設計測試案例。

我們的測試案例的設計會依照以下的步驟來完成，組成單元測試的元素在上一個小節中，我們已經在產生AspectJ的時候都蒐集完成了，接下來我們要依據收集到的素材來組成我們的單元測試。

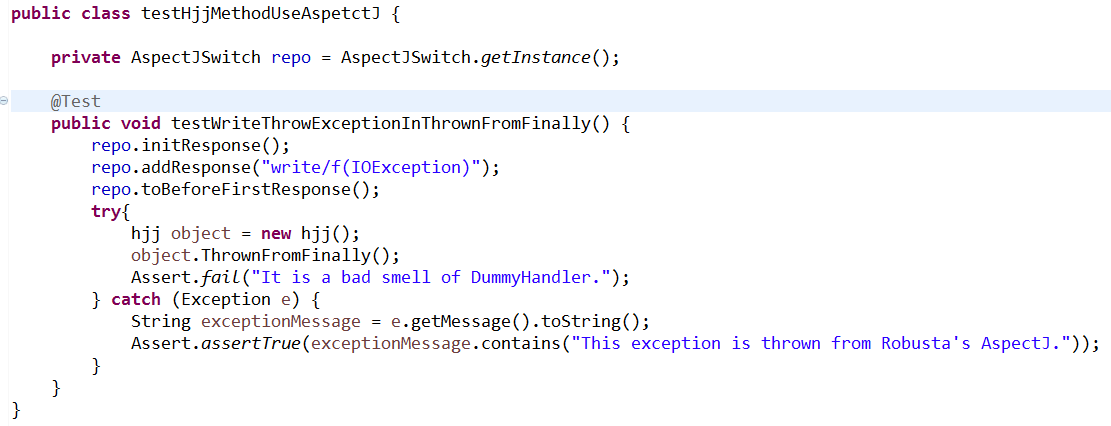
**Dummy Handler & Empty Catch Block**

在Dummy Handler和Empty Catch Block中

1. 取得壞味道中，第一個會丟出例外的Method Invocation name、丟出例外的類型
2. 取得該Method Invocation 所在的 class 名稱，
3. 依據該Method Invocation的 method declaration的類別，判斷其類別是public、private、protected 來把這個物件給創建出來
4. 呼叫該物件指定的method declaration並讓其例外重現

錯誤重現並驗證

根據下圖的程式碼，若Dummy Handler、Empty Catch Block的壞味道沒有被消除，則表示該程式在遭遇例外狀況的時候，採取的方式單存只有log、display，則此測試案例便會來到Assert Fail的地方，此方法可以還原出該測試案例的情境，並將錯誤還原出來，若將程式碼的壞味道消除後，則表示該程式在遭遇例外狀況的時候，根據我們的建議方式，原始碼程式有將他的例外拋出來，所以該壞味道就被消除了，回到此單元測試可以看到若我們接收到我們利用AspectJ帶入例外的資訊，則表示該段程式碼有對dummy handler做處理，因此這個測試案例就通過了。



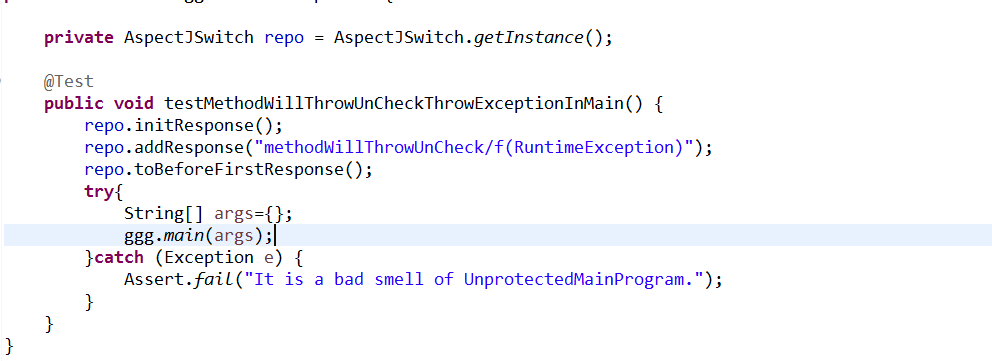
**Unprotected Main Program**

在Unprotected Main Program中

1. 取得壞味道中，第一個會丟出例外的Method Invocation name、丟出例外的類型
2. 取得該Method Invocation 所在的 class 名稱
3. 呼叫該物件指定的Method Declaration並讓其例外重現

錯誤重現並驗證

根據下圖的程式碼，若Unprotected main program的壞味道沒有被消除，則表示該程式在遭遇RunTimeException的時候，則該程式會不預期的終止，則此測試案例便會因為例外的狀況來到該測試Catch Block的地方，因此就會出現Assert Fail的狀況，我們也可以藉此來顯示出壞味道所帶來的影響，若此壞味道被消除後，儘管在Main Program中出現例外狀況，但是因為Main Program被Try Catch包起來，所以Main Program的程式並不會因為這樣就被中只，至少需要Log 或Display來讓做紀錄，根據我們設計的測試案例來說該程式有對其壞味道做處理，因此這個測試案例就通過了。



**Exception Thrown From Finally Block**

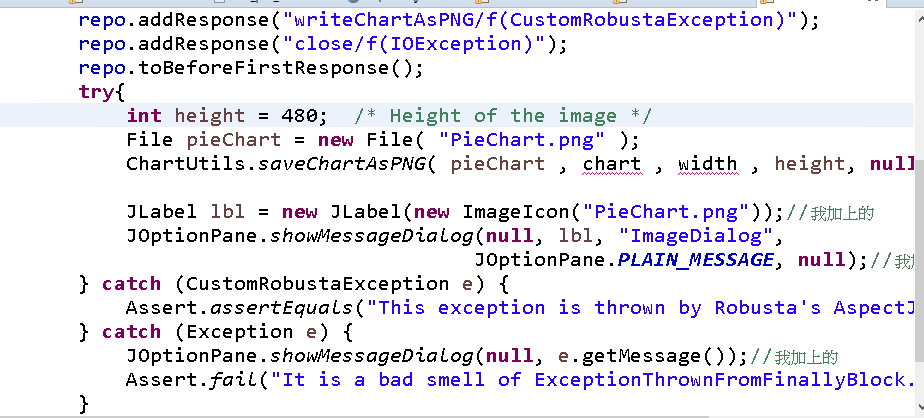
在Exception thrown from Finally Block中

1. 取得壞味道中，Try Block第一個會丟出例外的Method Invocation name、嵌入我們產生的例外類型名稱
2. Finally Block 中第一個會丟出例外的Method Invocation name、丟出例外的類型
3. 取得該Method Invocation 所在的 class 名稱，
4. 依據該Method Invocation的 method declaration的類別，判斷其類別是public、private、protected 來把這個物件給創建出來
5. 呼叫該物件指定的method declaration並讓其例外重現

錯誤重現並驗證

根據下圖的程式碼，我們可以發現若是在進入Finally Block前發生了例外，且Finally Block本身也會發生例外的話，則會造成Finally Block的例外會掩蓋Finally Block前所發生的例外，所以根據這樣的情境，如果今天壞味道存在source code的話，根據我們在Try Block中嵌入例外，並且在執行到Finally Block時，會再丟出一次例外，我們並不希望在Finally Block中被丟出來，所以當丟出例外的時候會來到最下面的區塊中的Catch Block，因此就會出現Assert Fail的狀況，此方法可以還原出該測試案例的情境，並將錯誤還原出來，若將程式碼的壞味道消除後，我們建議使用者不要在Finally Block中丟出例外，所以根據我們在Try Block 和Finally Block中的例外，這個案例因為Try Block中丟出了例外，所以會來到我們客制的Catch Block，因此這個測試案例就通過了。





**Careless Cleanup**

在Careless Cleanup中

1. 取得壞味道中，離release method最遠的Method Invocation且會丟出例外的Method Invocation name、丟出例外的類型
2. 取得release method的Method Invocation name、嵌入我們產生的例外類型名稱
3. 取得該Method Invocation 所在的 class 名稱，依據該Method Invocation的 method declaration的類別，判斷其類別是public、private、protected 來把這個物件給創建出來
4. 呼叫該物件指定的method declaration並讓其例外重現

錯誤重現並驗證

根據下圖的程式碼，我們可以發現若這樣的程式碼在close之前發生了例外，則會導致close不會被執行，而讓資源不會被正常釋放，所以根據這樣的情境我們設計了測試案例，我們的AspectJ會在離close method最遠且會發生例外的Method Invocation 嵌上其對應的例外，以及在close method 中嵌上我們自定義的例外，在這樣的況下，若壞味道沒有被消除，則會來到我們自定義的Exception的Catch Block，因此就會出現Assert Fail的狀況，此方法可以還原出該測試案例的情境，並將錯誤還原出來，若將程式碼的壞味道消除後，則最終我們會得到由Finally Block中release method所丟出來的例外，並被自定義例外的Catch Block給接起來，因此這個測試案例就通過了。





**第四章 案例實做與分析**

本章節將會以開源社群裡的開源專案為例，使用第三章所介紹的暴露壞味道所帶來影響的方法來進行分析及實做

Unprotected Main Program 案例分析與實作

**偵測並分析Protected Main Program**

如圖所示，Tomighty.java中的程式碼截圖。此段程式碼為程式的main program，根據Robusta的分析，此main program 並沒有被 try catch 包覆起來，如果主程式或執行緒沒有捕捉由下拋出至身上的例外，則主程式或執行緒會不預期地終止執行，所以這是一種Unprotected Main的壞味道。



呈現Unprotected Main Program 對系統的影響

當Tomighty 正常啟動時，會開啟主程式並正常運作該程式，如下圖所示



為了觀察例外發生時壞味道造成的影響，我們藉由Robusta產生了一個aspectJ程式來幫助我們注入例外，強制暴露出壞味道帶來的影響，如下圖，在第六行的地方定義丟出例外的時機點、method的類型、要嵌入的class、嵌入例外的method，第10行會幫我們丟出runtimeexception來幫我們暴露出這個壞味道造成的影響



為了讓例外在測試的狀態如期發生，我們還需要搭配我們的測試才可以丟出例外，第15行為要注入的methodInvocation以及要丟出例外的類型。當程式執行到圖中的58行時，aspect程式注入了runtimeexception，此例外發生後使得Tomight.main主程式因為沒有對例外進行處理，所以程式便異常終止了，如此一來我們就可以把Unprotected Main Program這個壞味道帶來的影響給暴露出來，此外我們的設計案例會因為壞味道存在所以導致測試fail，讓使用者知道壞味道仍存在





消除Unprotected Main Program

根據楊雅雯論文提供的修復功能移除壞味道後，再次對此測試重新驗證一次，因為壞味道被消除了，所以儘管程式被注入例外，但因為外層被try statement包住，捕捉所有例外，並視需要記錄或在畫面上顯示清楚錯誤訊息，此外我們的設計案例會因為壞味道被消除所以測試案例通過，讓使用者知道壞味道已經消除了。



Exception Thrown From Finally Block案例分析與實作

**偵測並分析Exception Thrown From Finally Block**

如圖所示，ChartUtils.java中的程式碼截圖。此段程式碼為幫助使用者把chart轉存為PNG的檔案格式，根據Robusta的分析，此saveChartAsPNG 函式存在著Thrown Exception From Finally的壞味道，根據文獻探討，很多關閉資源的函數都會丟出例外，用來代表釋放資源失敗，根據下圖的程式碼範例，若Try block 或是 catch block發生例外，而finally block也發生例外的話，finally block的例外則會掩蓋住前try block 或是 catch block發生例外，所以這是一種Thrown Exception From Finally的壞味道。



呈現Exception Thrown From Finally Block 造成的影響

為了觀察例外發生時對壞味道造成的影響，我們藉由Robusta產生了一個AspectJ程式來幫助我們注入例外，強制暴露出壞味道帶來的影響，如下圖，在第10、17行的地方定義丟出例外的時機點、method的類型、要嵌入的class、嵌入例外的method，第14行會幫我們丟出IOException，

第21行會幫我們丟出自定義CustomRobustaExxception來幫我們暴露出這個壞味道造成的影響



為了讓例外在測試的狀態如期發生，我們還需要搭配我們的測試才可以丟出例外，第22行為要注入的methodInvocation以及自定義的例外，此外在

23行注入IOException的例外。如此一來，當程式執行到圖中的308行時，aspect程式注入了CustomRobustaException自定義的例外，使其產生例外，此實程式碼因為有finally block的關係，接續執行311行的釋放資源，同一時間aspect注入了IOException，根據此發法我們讓try block、finally block都使期發生例外，如此一來finally產生的例外便會覆蓋try block裡面所發生的例外，便可以將Exception Thrown From Finally Block這個壞味道帶來的影響給暴露出來，此外我們的設計案例會因為壞味道存在所以導致測試fail，讓使用者知道壞味道仍存在





消除Exception Thrown From Finally Block  
根據楊雅雯論文提供的修復功能移除壞味道後，再次對此測試重新驗證一次，因為壞味道被消除了，所以儘管finally block被注入例外，但因為此層被try statement包覆住，捕捉例外不讓它在finally中將例外丟出去，並視需要記錄或在畫面上顯示清楚錯誤訊息，用此做法我們就可以消除此壞味道，此外我們的設計案例會因為壞味道被消除所以測試案例通過，讓使用者知道壞味道已經消除了。

