

摘要

論文名稱：利用 Robusta 消除例外處理壞味道

頁數：六十二頁

校所別：國立臺北科技大學 資訊工程 研究所

畢業時間：一百零六學年度 第二學期

學位：碩士

研究生：楊雅雯

指導教授：鄭有進教授、謝金雲教授

關鍵詞：例外處理、壞味道、強健度、重構

移除例外處理壞味道，可以提升軟體的強健度。然而，例外處理壞味道的移除，是相當具有挑戰性的工作，對於經驗不足的開發人員尤然。

針對各種不同的 JAVA 程式例外處理壞味道，本論文提供一系列以重構為基礎的處理壞味道移除參考範例，以導引開發者移除例外處理壞味道。我們並將相關作業程序實作在 Robusta 工具中，開發人員可透過 Robusta 工具自動化重構程式碼來消除壞味道，以提升軟體的強健度。

ABSTRACT

Title: Removing Exception Handling Bad Smells Using Robusta

Pages: 62

School: National Taipei University of Technology

Department: Information Engineering

Time: June, 2018

Degree: Master

Researcher: Ya-Wun Yang

Advisor: Chin-Yun Hsieh Ph.D., Yu Chin Cheng Ph.D.

Keywords: Exception Handling, Code Smells, Robustness, Refactoring

Exception handling bad smells may jeopardize the robustness of a program. While writing exception handling code without bad smells is challenging, the task of removing exception handling bad smells is even harder. This is particularly true for inexperienced developers.

Robusta is a static analysis tool capable of detecting exception handling bad smells in Java programs and exposing their impact to the program. As an extension of the previous work on Robusta, in this thesis we propose a series of refactoring-based smell removal methods corresponding to the exception handling bad smells detectable by the Robusta. The refactoring methods have been implemented in Robusta. Its usefulness has also been shown in an empirical study.

致謝

在軟體系統實驗室的這兩年，很感謝鄭有進老師、謝金雲老師提供了很好的學習環境，讓我有機會參與各種專案來增加自己的實作經驗，也能夠將課堂學到的知識應用在實作中。並且在專案或是論文遇到問題時，兩位老師也都提供我們許多寶貴的建議，讓我們能夠在迷茫中找到方向。

感謝李家政學長，當團隊開發遇到問題時，謝謝學長總是提供方向或是引導我們去找尋答案，讓功能能夠順利完成。

感謝實驗室的同學們，當我在專案或是課堂作業有問題的時候，總是協助我並且教導我。也在閒暇之餘一起出去吃飯、聊天、打桌遊，讓我有個充實的研究所生活。

感謝我的家人、親戚和朋友，謝謝家人支持我唸研究所，並提供生活費讓我能夠無憂無慮地完成這兩年的碩士學位；謝謝姑姑有回高雄的時候總是會送一些家裡的東西來給我；謝謝姨婆常常來看我，和煮好吃的料理給我吃；謝謝朋友偶爾陪我一起吃飯、聊天，雖然見面的時間不多，但是跟你們相處的時光很開心；謝謝哥哥總是在我生活遇到任何問題的時候來幫助我排除，也常常帶我出去玩。

最後，我想要謝謝隊友劉彥麟同學，在碩二的這一年你教了我很多，不論是軟工課的專案、Robusta、TCSE、或是找工作面試的一些問題，謝謝你一直帶著我，讓我在這兩年的研究所中學到了很多知識跟經驗。

目錄

摘 要	i
ABSTRACT	ii
致謝	iii
目錄	iv
表目錄	vii
圖目錄	viii
第一章 緒論	1
1.1 研究背景與動機	1
1.2 研究目標	1
1.3 論文組織架構	2
第二章 背景知識	3
2.1 Robusta	3
2.2 例外處理壞味道	3
2.2.1 Empty Catch Block	3
2.2.2 Dummy Handler	4
2.2.3 Unprotected Main Program	5
2.2.4 Nested Try Statement	6
2.2.5 Careless Cleanup	6
2.2.6 Exception Thrown From Finally Block	7
2.3 強健度等級	8
等級 0：未定義(Undefined)	8
等級 1：錯誤回報(Error reporting)	8

等級 2：狀態回復(State recovery)	9
等級 3：行為重試(Behavior recovery).....	10
2.4 Abstract Syntax Tree	11
第三章 研究方法	13
3.1 壞味道消除方法介紹	13
3.1.1 Dummy Handler & Empty Catch Block	13
3.1.2 Unprotected Main Program	17
3.1.3 Nested Try Statement	19
3.1.4 Careless Cleanup	22
3.1.5 Exception Thrown From Finally Block.....	23
3.2 過去與現在 Robusta 快速修復與重構差異	26
3.2.1 快速修復	26
3.2.2 重構.....	27
3.3 設計與實作	28
3.3.1 快速修復	28
3.3.2 重構.....	37
3.4 壞味道的偵測、曝露及消除流程	44
第四章 應用實例	45
4.1 Dummy Handler 應用實例	45
4.1.1 偵測 Dummy Handler	45
4.1.2 產生曝露 Dummy Handler 的測試案例	46
4.1.3 消除 Dummy Handler	47
4.2 Careless Cleanup 應用實例	49
4.2.1 偵測 Careless Cleanup	49
4.2.2 產生曝露 Careless Cleanup 的測試案例	49
4.2.3 消除 Careless Cleanup	50
4.3 Exception Thrown From Finally Block 應用實例.....	52
4.3.1 偵測 Exception Thrown From Finally Block.....	52

4.3.2 產生曝露 Exception Thrown From Finally Block 的測試案例.....	52
4.3.3 消除 Exception Thrown From Finally Block.....	53
4.4 Unprotected Main Program 應用實例	56
4.4.1 偵測 Unprotected Main Program	56
4.4.2 產生曝露 Unprotected Main Program 的測試案例	56
4.4.3 消除 Unprotected Main Program	57
第五章 結論與未來展望	59
5.1 結論	59
5.2 未來展望	59
參考文獻.....	61

表目錄

表 3-1、Robusta 快速修復功能差異.....	26
表 3-2、Robusta 重構功能差異.....	27

圖目錄

圖 2-1、Empty Catch Block 範例	4
圖 2-2、Dummy Handler 範例.....	5
圖 2-4、Nested Try Statement 範例.....	6
圖 2-5、Careless Cleanup 範例.....	7
圖 2-6、Exception Thrown From Finally Block 範例	7
圖 2-7、等級 0：未定義.....	8
圖 2-8、等級 1：錯誤回報.....	9
圖 2-9、等級 2：狀態回復.....	9
圖 2-10、等級 3：行為重試成功.....	10
圖 2-11、等級 3：行為重試失敗.....	10
圖 2-12、分析 Java 檔後的 AST view 樹狀結構	12
圖 3-1、Dummy Handler 範例.....	14
圖 3-2、Dummy Handler 壞味道快速修復結果.....	14
圖 3-3、Dummy Handler 重構結果.....	15
圖 3-4、Dummy Handler 重新修正後的快速修復結果.....	16
圖 3-5、選單提供的 Unchecked Exception.....	17
圖 3-6、Dummy Handler 重新修正後的重構結果.....	17
圖 3-7、Unprotected Main Program 壞味道快速修復結果.....	18
圖 3-8、錯誤物件的繼承架構.....	19
圖 3-9、Unprotected Main Program 改善後的快速修復結果.....	19
圖 3-10、選取 Eclipse 提供的 Refactor→Extract Method...功能	20
圖 3-11、在 Extract Method 視窗替函式命名	21
圖 3-12、Nested Try Statement 重構結果.....	21
圖 3-13、Robusta 提供的 Extract Method 功能	22

圖 3-14、Careless Cleanup 快速修復結果.....	23
圖 3-15、Exception Thrown From Finally Block 原先的重構結果	24
圖 3-16、Exception Thrown From Finally Block 改善後的重構結果	25
圖 3-17、Dummy Handler 和 Empty Catch Block QuickFix 功能的 Class Diagram.....	29
圖 3-18、Dummy Handler 和 Empty Catch Block QuickFix 功能的 Sequence Diagram.....	30
圖 3-19、Unprotected Main Program QuickFix 功能的 Class Diagram	32
圖 3-20、Unprotected Main Program QuickFix 功能的 Sequence Diagram.....	33
圖 3-21、Careless Cleanup QuickFix 功能的 Class Diagram	35
圖 3-22、Careless Cleanup QuickFix 功能的 Sequence Diagram.....	36
圖 3-23、Dummy Handler 和 Empty Catch Block Refactoring 功能的 Class Diagram.....	38
圖 3-24、Dummy Handler 和 Empty Catch Block Refactoring 功能的 Sequence Diagram	38
圖 3-25、Nested Try Statement Refactoring 功能的 Class Diagram	40
圖 3-26、Nested Try Statement Refactoring 功能的 Sequence Diagram.....	41
圖 3-27、Exception Thrown From Finally Block Refactoring 功能的 Class Diagram.....	42
圖 3-28、Exception Thrown From Finally Block Refactoring 功能的 Sequence Diagram	43
圖 3-29、壞味道偵測、曝露及消除流程圖	44
圖 4-1、JFreeChart 中 Dummy Handler 範例	46
圖 4-2、JFreeChart Dummy Handler 測試案例	46
圖 4-3、JFreeChart Dummy Handler 測試失敗	47
圖 4-4、Robusta 提供自動化消除 Dummy Handler.....	47
圖 4-5、JFreeChart 消除 Dummy Handler 的結果	48
圖 4-6、JFreeChart 中 readPieDatasetFromXML 函式正確處理例外後測試成功	48
圖 4-7、JFreeChart 中 Careless Cleanup 範例	49
圖 4-8、JFreeChart Careless Cleanup 測試案例	49
圖 4-9、JFreeChart Careless Cleanup 測試失敗	50
圖 4-10、Robusta 提供自動化消除 Careless Cleanup.....	50
圖 4-11、JFreeChart 消除 Careless Cleanup 的結果	51
圖 4-12、JFreeChart 中 encode 函式正確處理例外後測試成功	51

圖 4-13、JFreeChart 中 Exception Thrown From Finally Block 範例	52
圖 4-14、JFreeChart Exception Thrown From Finally Block 測試案例	53
圖 4-15、JFreeChart Exception Thrown From Finally Block 壞味道測試失敗	53
圖 4-16、Robusta 提供自動化消除 Exception Thrown From Finally Block	54
圖 4-17、JFreeChart 消除 Exception Thrown From Finally Block 的結果	54
圖 4-18、JFreeChart 中 saveChartAsPNG 函式正確處理例外後測試成功	55
圖 4-19、Tomighty 工具	56
圖 4-20、Tomighty 中 Unprotected Main Program 範例	56
圖 4-21、Tomighty Unprotected Main Program 測試案例	57
圖 4-22、Tomighty Unprotected Main Program 測試失敗	57
圖 4-23、Robusta 提供自動化消除 Unprotected Main Program	57
圖 4-24、Tomighty 消除 Unprotected Main Program 的結果	58
圖 4-25、Tomighty 中 main program 正確處理例外後測試成功	58
圖 5-1、兩層 Try Statement 的 Careless Cleanup 範例	59

第一章 緒論

本章節將先介紹本論文的研究背景與動機；接著描述本論文的預期目標；最後介紹本論文的組織架構。

1.1 研究背景與動機

例外處理為程式碼在執行過程中，遇到例外狀況時所做的處理方式。在 Java 程式碼中，常見的例外處理機制為 try-catch-finally。當 try、catch 或 finally 中含有壞味道時，容易因為不適當的例外處理而導致錯誤發生，降低系統的強健度。正確的例外處理設計及實作能夠讓軟體在遭遇例外時，也能正常執行功能。然而，對經驗不足的開發人員而言，容易因為相關知識不足而設計出含有例外處理壞味道[1]的程式碼。

撰寫出正確的例外處理程式碼是一件困難的事，因此為了協助開發人員能夠正確處理例外行為，根據陳建村等人的研究[2]，提出以強健度等級作為例外處理等級或能力的依據，透過重構程式碼[3]來消除例外處理壞味道，並將其功能實作於 Robusta[4]中且使其自動化。藉由 Robusta 來幫助開發人員發掘程式碼中例外處理壞味道，並且以自動化重構的方式來消除這些壞味道，進而提升軟體品質。

1.2 研究目標

歷屆多人的研究，Robusta 裡定義的壞味道不斷地被修正及更新，其偵測壞味道的功能也愈來愈準確。然而，部分例外處理壞味道被偵測出來後，其對應的重構方法尚未被實作，或是還有能夠改善的地方。因此，本論文將對 Robusta 重構的功能進行改善，並將部分壞味道尚未實作的重構功能補齊，讓 Robusta 藉由重構消除例外處理壞味道的功能更加齊全。

1.3 論文組織架構

本論文分為五個章節，第一章是緒論，描述本論文的背景與動機。第二章會介紹與本論文相關的背景知識。第三章為研究方法，會介紹消除例外處理壞味道的方法，並說明本論文是如何設計及實作於 Robusta 中。第四章則會運用本論文提供的壞味道消除方法在實際案例應用以提升程式的強健度。最後一章為本論文的結論與未來展望。

第二章 背景知識

2.1 Robusta

Robusta[4]是一個靜態分析 Java 程式碼的工具，它能夠偵測出程式碼中例外處理的壞味道並且產生報表，也能夠以自動化修改程式碼的方式消除這些壞味道。在 Eclipse[5] 的工具列中，點擊 Help→Eclipse Marketplace 後，搜尋 Robusta 安裝即可。安裝完畢後，對專案點擊右鍵選擇 Properties→Robusta Detecting Settings，能夠設定要偵測的壞味道類型。而在 Robusta 定義了六種例外處理壞味道[6]，分別為：

1. Empty Catch Block
2. Dummy Handler
3. Unprotected Main Program
4. Nested Try Statement
5. Careless Cleanup
6. Exception Thrown From Finally Block

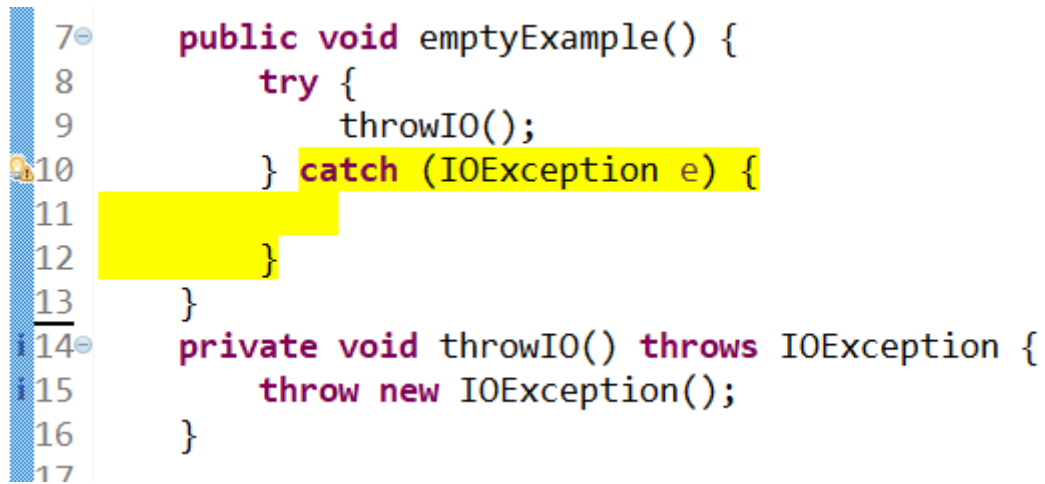
下一小節將會一一介紹。

2.2 例外處理壞味道

2.2.1 Empty Catch Block

Java 編譯器要求開發者對程式碼中會丟出 Checked Exception 的函式做處理，處理的方法有兩種：在介面宣告會丟出例外或用 try/catch 將函式包覆起來。為了不造成「介面演進」[1]的問題，普遍開發者會選擇用 try/catch 將函式包住。因此，當 try 裡的程式碼發生例外時，例外被 catch 捕捉後，如果 catch 裡不做任何事，會造成例外被掩蔽的現象，稱之為 Empty Catch Block 壞味道[6]。此種忽略例外的作法會造成開發者除錯不容

易，並且掩蔽例外發生的事實，讓程式碼帶著不正確的狀態繼續往下執行，而降低系統強健度。



```
7 public void emptyExample() {  
8     try {  
9         throwIO();  
10    } catch (IOException e) {  
11    }  
12 }  
13  
14 private void throwIO() throws IOException {  
15     throw new IOException();  
16 }  
17
```

圖 2-1、Empty Catch Block 範例

2.2.2 Dummy Handler

Dummy Handler 壞味道[6]定義和 Empty Catch Block 相似，差別在於當程式在 try 裡發生例外時，例外被 catch 捕捉後，catch 裡只有記錄例外訊息，沒有再做其他的處理。雖然例外有被記錄下來，但如果在圖形化、網頁化界面等等情境下，使用者或開發者不容易直接看到被印出的例外訊息，也容易產生例外已經被處理的假象，但實際上卻沒有對例外進行修復。可能產生的影響為當系統發生錯誤時，除了會造成開發人員不容易除錯，也會讓程式碼繼續在不正確的狀態往下執行。因此 Dummy Handler 會增加開發者或除錯者除錯的困難度，同時也會降低系統強健度。

```

7-   public void dummyExample() {
8       try {
9           throwIO();
10      } catch (IOException e) {
11          e.printStackTrace();
12      }
13  }
i14- private void throwIO() throws IOException {
i15     throw new IOException();
16 }
17

```

圖 2-2、Dummy Handler 範例

2.2.3 Unprotected Main Program

當例外被丟出後都沒有被 catch 捕捉時，這些例外都會被往上層回報。因此，這些例外最後會來到最上層的主程式中。如果主程式也不處理這些例外，將會使程式發生不預期的終止。因此，當主程式裡的程式碼沒有被 try/catch 包覆住，並在 catch 裡對例外做處理，稱之為 Unprotected Main Program 壞味道[6]。對使用者而言，程式不預期地終止是軟體品質不佳的表現。

```

7-   public static void main(String[] args) {
8       int[] array = new int[9];
9       for(int i = 0; i <= 9 ;i++) {
10          array[i] = i;
11      }
12  }
i13

```

圖 2-3、Unprotected Main Program 範例

2.2.4 Nested Try Statement

當 try、catch 或 finally 區塊中存在巢狀結構的 Try Statement，稱之為 Nested Try Statement 壞味道[6]。雖然 Nested Try Statement 不會對程式碼造成影響，也沒有程式邏輯的錯誤，但是對開發者來說，複雜的巢狀結構將不容易閱讀，並且不容易測試和維護。

```
7 public void nextedTryExample() {
8     try {
9         try {
10             method(-1);
11             FileInputStream file = new FileInputStream("example.txt");
12         } catch (FileNotFoundException e) {
13             System.out.println("FileNotFoundException happen!");
14             handler();
15         }
16         method(100);
17     } catch (Exception e) {
18         System.out.println("Exception happen!");
19         handler();
20     }
21 }
22 private void handler() {
23 }
24 private int method(int num) {
25     if(num < 0)
26         throw new RuntimeException();
27     return num;
28 }
29 }
```

圖 2-4、Nested Try Statement 範例

2.2.5 Careless Cleanup

Careless Cleanup 壞味道[6]定義為當程式碼執行時，在執行到釋放資源的程式碼之前發生例外狀況，會無法執行到釋放資源的程式碼，而造成資源沒有正確被釋放。Careless Cleanup 會導致資源被消耗並且降低系統穩定度。如圖 2-5 所示，若程式碼執行到第 14 行發生例外時，則會進到第 16 行 catch 裡對例外進行處理，而造成程式沒有執行到第 15 行釋放資源的程式碼，導致資源沒有正確被釋放。


```

10 public void carelessCleanupExample() {
11     FileInputStream fileInputStream = null;
12     try {
13         fileInputStream = new FileInputStream("test.txt");
14         fileInputStream.read();
15         fileInputStream.close();
16     } catch (IOException e) {
17         handle();
18     }
19 }

```

圖 2-5、Careless Cleanup 範例

2.2.6 Exception Thrown From Finally Block

一段有 try/catch/finally 的程式碼，當程式在 try 或 catch 發生例外時，若對例外進行的處理方式為向上層呼叫者回報後，程式最後會執行 finally 裡的程式碼。如果當 finally 裡的程式碼也發生例外並向上層回報，會覆蓋掉原先 try 或 catch 裡所回報的例外，產生「例外蓋台」的現象，稱之為 Exception Thrown From Finally Block 壞味道[6]。這個壞味道會誤導開發者關注在 finally 發生的例外，而忽略了原先 try 或 catch 所發生的例外。如圖 2-6 所示，當 try 的第 12 行發生例外時，程式會來到第 13 行的 catch 裡將例外回報，接著執行 finally 裡第 16 行釋放資源的程式碼。但如果第 16 行發生例外時，會覆蓋掉原來在第 14 行回報的例外，而改回報第 16 行的例外。

```

8 public void exceptionThrownFormFinallyBlockExample() throws IOException {
9     FileInputStream fileInputStream = null;
10    try {
11        fileInputStream = new FileInputStream("test.txt");
12        fileInputStream.read();
13    } catch (IOException e) {
14        throw new IOException("Exception thrown in catch");
15    } finally {
16        fileInputStream.close();
17    }
18 }

```

圖 2-6、Exception Thrown From Finally Block 範例

2.3 強健度等級

在陳建村的研究[7]中，提出包含四個強健度等級的例外處理模型，作為規劃與判斷軟體元件例外處理等級或能力的依據。強健度的四個等級，分別為：

等級 0：未定義(Undefined)

當開發人員還沒對他的系統貼上強健度等級[7]的標籤時，則該系統的強健度等級為 0。在這個等級的系統發生例外時，處理的方法可能有印出例外訊息讓開發人員知道發生例外，也有可能是什么都不做直接忽略例外。因此當例外發生時，可能導致系統發生錯誤也可能使系統繼續正常執行，無法確實掌握。

如圖 2-7 所示，當 Component A 發生例外 E 時，由於 Component A 沒有定義強健度，造成 Component B 無法得知 Component A 是執行成功還是將例外掩蔽起來，而無法做後續的處理。

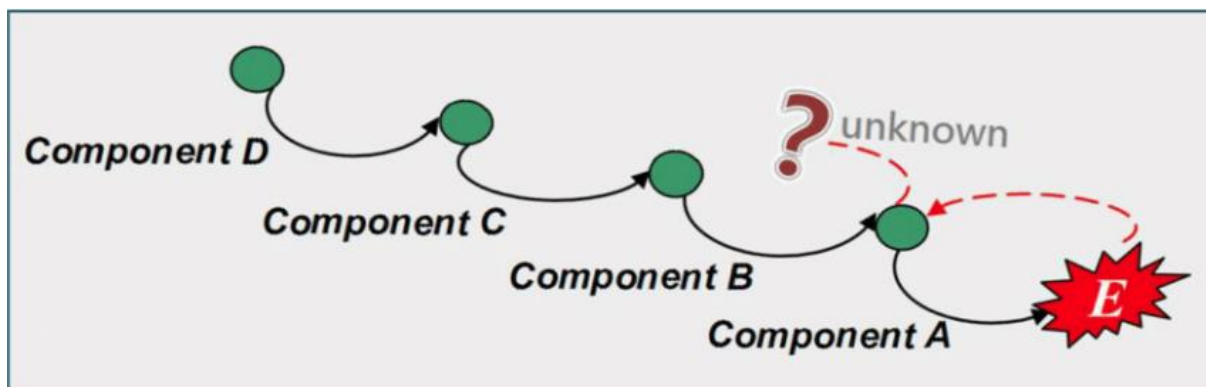


圖 2-7、等級 0：未定義

等級 1：錯誤回報(Error reporting)

錯誤回報[7]為當例外發生時，一定要讓呼叫者知道，讓呼叫者了解有例外發生，因此只要將例外往上一層丟出即可。

如圖 2-8 所示，當 Component A 發生例外時，如果不進行任何處理，會直接將例外

回報給上層 Component B，如果 Component B 也不處理，則繼續往上層回報，直到回報到最上層的 Component D。最後，由最上層 Component 處理例外或記錄例外訊息。

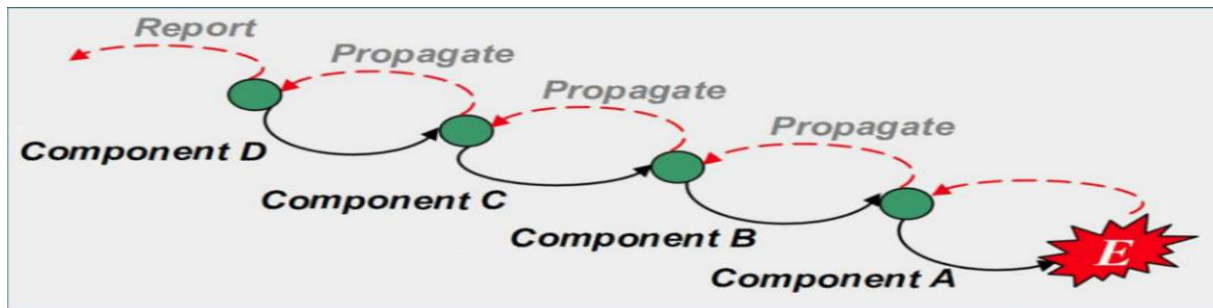


圖 2-8、等級 1：錯誤回報

等級 2：狀態回復(State recovery)

達到這個等級時，一定要先滿足等級 1，並且在錯誤發生後讓系統回復到原本正確的狀態。因此，例外發生後，系統還是能夠繼續正常執行[7]。

如圖 2-9 所示，Component A 發生例外後，將例外回報給上層 Component B；接著，Component B 再將例外回報給上層 Component C。由於 Component C 接到例外後造成狀態錯誤，因此先將狀態回復成讓程式能夠繼續正常提供服務後，再將例外回報給上層 Component D，由最上層 Component 處理例外或記錄例外訊息。

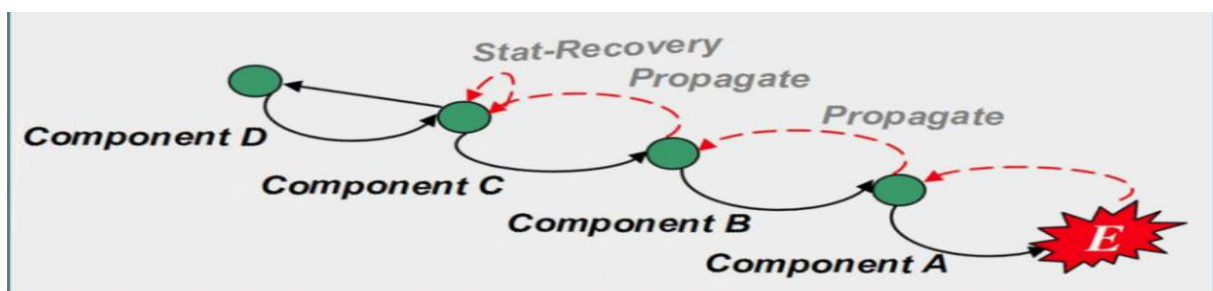


圖 2-9、等級 2：狀態回復

等級 3：行為重試(Behavior recovery)

系統強健度要到達等級 3，必須先滿足等級 2 的條件外，還要重試例外發生前的行為或是尋找其他方法來達成原本的任務。如果重試行為失敗或是尋找的其它方法也發生例外，則要將例外向上層回報[7]。

如圖 2-10 所示，Component A 發生例外後，會先將狀態回復之後，再重試行為，如果重試成功後，則程式繼續往下執行。如果重試失敗，如圖 2-11 所示，則會將例外向上層回報，由最上層 Component 處理例外或記錄例外訊息。

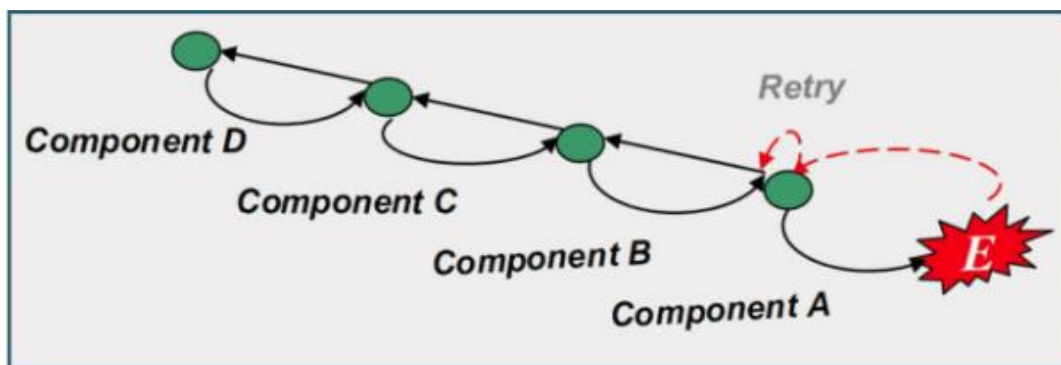


圖 2-10、等級 3：行為重試成功

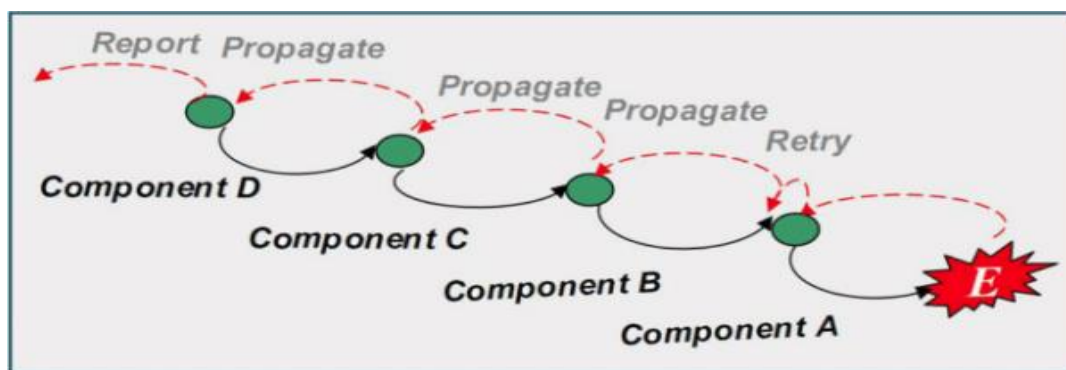


圖 2-11、等級 3：行為重試失敗

2.4 Abstract Syntax Tree

Abstract Syntax Tree 簡稱 AST[8]，是 Eclipse JDT[9]裡的一部份，用來表達 Java 檔案的結構。另外，Eclipse[5]提供 AST view[10]的 plugin，能夠將 Java 檔案轉換為樹狀結構，表達 Java 程式碼的語法架構。本論文消除壞味道的快速修復功能和重構功能，都是透過 AST 提供的 ASTParser[11]來分析 Java 程式碼的結構，藉由 AST 提供的走訪方式來走訪節點，並使用 ASTRewrite[12]來對目標節點進行修改，最後寫回 Java 檔中來變更程式碼內容。常見的 AST 節點有：

- Method Invocation：method 的節點在 AST 中稱為 Method Invocation。
- Try Statement：當程式碼遇到可能丟出例外的 Method Invocation 時，Java 編譯器會要求開發者將該 Method Invocation 用 try/catch 保護起來，或是在介面宣告該 Method Invocation 會丟出的例外型別。如果開發者用 try/catch 保護的話，此 try/catch 的節點在 AST 中稱為 Try Statement。Try Statement 包含了 try block、catch clause 和 finally block。
- Method Declaration：一段程式碼的區塊，包含程式碼界面和區塊中程式碼執行的內容，在 AST 中稱為 Method Declaration，而 Method Declaration index 為 Method Declaration 在該 Java 文件的排序。

如圖 2-12 所示，為 AST 在程式碼中的範圍和 AST view 樹狀結構的範例。

Method Declaration

```

public void example() throws IOException {
    method();
    try {
        hello();
    } catch (IOException e) {
        e.printStackTrace();
    }
    method();
    method();
}
```

- > PACKAGE
- > IMPORTS (4)
- ▼ TYPES (1)
 - ▼ TypeDeclaration [141+210]
 - > > type binding: ggg.eee
 - JAVADOC: null
 - > MODIFIERS (1)
 - INTERFACE: 'false'
 - > NAME
 - TYPE_PARAMETERS (0)
 - SUPERCLASS_TYPE: null
 - SUPER_INTERFACE_TYPES (0)
 - ▼ BODY_DECLARATIONS (2)
 - ▼ MethodDeclaration [164+108]
 - > > method binding: eee.example()
 - JAVADOC: null
 - > MODIFIERS (1)
 - CONSTRUCTOR: 'false'
 - TYPE_PARAMETERS (0)
 - > RETURN_TYPE2
 - > NAME
 - RECEIVER_TYPE: null
 - RECEIVER_QUALIFIER: null
 - PARAMETERS (0)
 - EXTRA_DIMENSIONS2 (0)
 - THROWN_EXCEPTION_TYPES (0)
 - ▼ BODY
 - ▼ Block [186+86]
 - ▼ STATEMENTS (1)
 - > TryStatement [191+77]

Method Invocation

Try Statement

圖 2-12、分析 Java 檔後的 AST view 樹狀結構

第三章 研究方法

本章節中，會先介紹要如何消除這些例外處理壞味道；然後，整理了 Robusta 過去與現在消除壞味道功能的差異；接著，將本論文提出的新增功能與改善的方法實作於 Robusta 中；最後，介紹本論文所提供的例外處理壞味道消除流程。

3.1 壞味道消除方法介紹

消除例外處理壞味道的方法為延續洪哲瑋的論文[3]。在 Dummy Handler、Empty Catch Block、Unprotected Main Program 和 Nested Try Statement 中，會先介紹洪哲瑋論文中所提供的方法是如何消除並實作於 Robusta 中。瞭解消除的方法後，介紹這些壞味道在 Robusta 中，哪些功能與當時提出的方法不一致或可以修正和改善的地方，並對其重新修正和改善；在 Careless Cleanup，本論文增加它的消除方法；在 Exception Thrown From Finally Block，會先介紹 Robusta 提供的重構功能，再介紹我們對此功能進行了什麼改善。最後，將本論文提出的功能實作於 Robusta 中，使其能夠自動化消除程式碼中例外處理的壞味道。

3.1.1 Dummy Handler & Empty Catch Block

根據洪哲瑋論文中提到的 Ignore Checked Exception，在楊智傑的論文[13]中被正式定義為 Empty Catch Block。而 Empty Catch Block 的消除方式與 Dummy Handler 一樣，於是本論文將 Dummy Handler 和 Empty Catch Block 放在本小節中一起敘述。如圖 3-1 所示，以下將以 Dummy Handler 為範例。

```

10 public void dummyExample() {
11     try {
12         throwIO();
13     } catch (FileNotFoundException e) {
14         e.printStackTrace();
15     } catch (IOException e) {
16         e.printStackTrace();
17     }
18 }
19 private void throwIO() throws IOException {
20     throw new IOException();
21 }

```

圖 3-1、Dummy Handler 範例

根據洪哲瑋的論文，這兩種壞味道的消除方法都是當 catch 捕捉到例外後，如果沒有要處理例外，應該將例外往上一層回報，提供的功能為快速修復功能及重構功能。而快速修復和重構兩種功能的差異點在於重構功能比較具有彈性，部分的參數能夠讓使用者自己決定，而快速修復的功能比較沒有彈性，使用者沒有辦法做更動。

- 快速修復功能：

如圖 3-2 第 15、17 行所示，當 catch 捕捉到例外後，透過 Robusta 的快速修復功能，自動於 catch 區塊中捕捉到的例外轉換為 RuntimeException，將例外回報給上一層。

```

10 @Robustness(value = { @RTTag(level = 1, exception = java.lang.RuntimeException.class) })
11 public void dummyExample() {
12     try {
13         throwIO();
14     } catch (FileNotFoundException e) {
15         throw new RuntimeException(e);
16     } catch (IOException e) {
17         throw new RuntimeException(e);
18     }
19 }
20 private void throwIO() throws IOException {
21     throw new IOException();
22 }

```

圖 3-2、Dummy Handler 壞味道快速修復結果

- 重構功能：

如圖 3-3 第 16、18 行所示，當使用者希望能夠丟出的例外型別為自己所定義的，而非上述所提到的 RuntimeException 時，洪哲瑋的論文說明能夠透過 Robusta 所提供的選

單，來選擇自己所希望拋出的例外型別來重構。

```
11 @Robustness(value = { @RTag(level = 1, exception = Example.UnhandleException.class) })
12 public void dummyExample() {
13     try {
14         throwIO();
15     } catch (FileNotFoundException e) {
16         throw new UnhandleException(e);
17     } catch (IOException e) {
18         throw new UnhandleException(e);
19     }
20 }
21 private void throwIO() throws IOException {
22     throw new IOException();
23 }
```

圖 3-3、Dummy Handler 重構結果

我們發現 Robusta 提供消除壞味道的功能與當時洪哲瑋論文所提出的不太一致，Robusta 提供的功能有三種，分別為：

1. 快速修復功能：丟出例外
2. 快速修復功能：丟出 RuntimeException
3. 重構功能：丟出 Unchecked Exception

在方法 3 的重構功能中，丟出的 Unchecked Exception 預設為 RuntimeException，我們認為方法 2 的快速修復功能與方法 3 的重構功能丟出的例外都屬於 Unchecked Exception，因此功能相似，所以將方法 2 拿掉，留下快速修復功能與重構功能，並根據現有功能來重新修正提供的功能。

- 重新修正後的快速修復功能：

不論 catch 所捕捉的例外為 Checked Exception 或 Unchecked Exception，都直接將所接到的例外丟出來向上層回報，並在介面宣告所丟出的例外，以最快速的方法來消除 Dummy Handler 和 Empty Catch Block。如圖 3-4 第 15、17 行所示，不論 catch 捕捉什麼例外，都直接丟出向上層回報，以達到快速修復來消除壞味道的目的。

```

10 @Robustness(value = { @RTag(level = 1, exception = java.io.FileNotFoundException.class), @RTag(level = 1, exception = java.io.IOException.class) })
11 public void dummyExample() throws FileNotFoundException, IOException {
12     try {
13         throwIO();
14     } catch (FileNotFoundException e) {
15         throw e;
16     } catch (IOException e) {
17         throw e;
18     }
19 }
20 private void throwIO() throws IOException {
21     throw new IOException();
22 }

```

圖 3-4、Dummy Handler 重新修正後的快速修復結果

- 重新修正後的重構功能：

我們發現Robusta在這兩種壞味道所提供的選單中，選單中只會顯示Unchecked Exception的類型，因此使用者能選擇自己定義的Unchecked Exception。並且在選擇Unchecked Exception丟出後，介面也不會宣告所回報的例外。因此Dummy Handler和Empty Catch Block提供的重構功能為丟出Unchecked Exception來進行回報。如圖3-5所示，選單會提供所有的Unchecked Exception，並根據使用者的輸入來從選單中來對Unchecked Exception進行篩選，圖中的CustomRobustaException為使用者自己定義的Unchecked Exception。如圖3-6所示，透過回報Unchecked Exception來消除壞味道後不會在介面進行宣告。

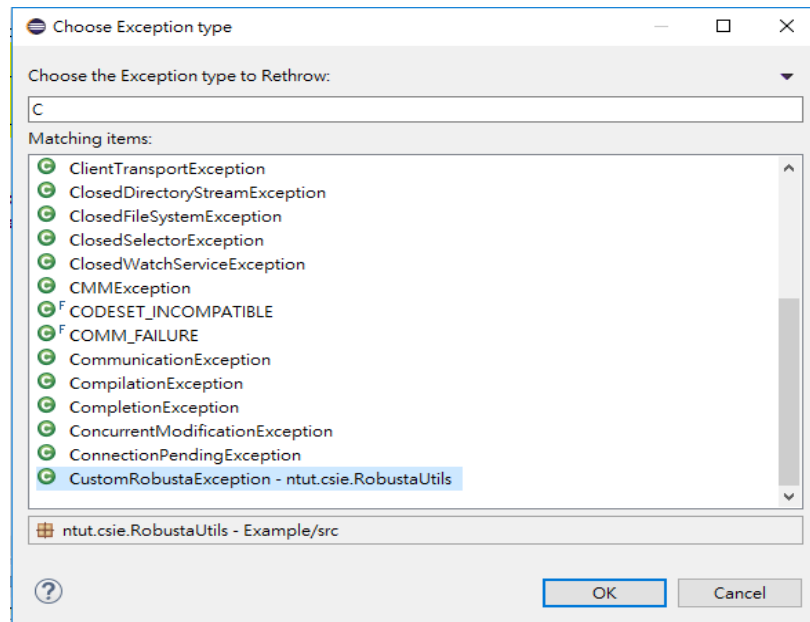


圖 3-5、選單提供的 Unchecked Exception

```

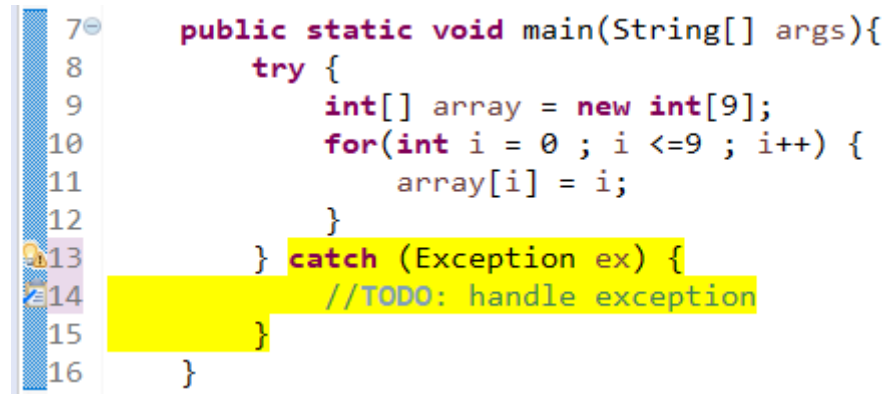
11 @Robustness(value = { @RTag(level = 1, exception = ntut.csie.RobustaUtils.CustomRobustaException.class) })
12 public void dummyExample() {
13     try {
14         throwIO();
15     } catch (FileNotFoundException e) {
16         throw new CustomRobustaException(e);
17     } catch (IOException e) {
18         throw new CustomRobustaException(e);
19     }
20 }
21 private void throwIO() throws IOException {
22     throw new IOException();
23 }
24

```

圖 3-6、Dummy Handler 重新修正後的重構結果

3.1.2 Unprotected Main Program

根據洪哲瑋的論文[3]，消除 Unprotected Main Program 的方法是利用快速修復的功能用 try/catch 將其 main program 保護起來。如圖 3-7 所示，快速修復功能為在 main program 中用 try/catch 將 main program 保護起來，並且在捕捉 Exception[14]型別的例外，最後在 catch 區塊中標示//TODO: handle exception 來提醒使用者要處理這個例外。



```

7 public static void main(String[] args){
8     try {
9         int[] array = new int[9];
10        for(int i = 0 ; i <=9 ; i++) {
11            array[i] = i;
12        }
13    } catch (Exception ex) {
14        //TODO: handle exception
15    }
16 }

```

圖 3-7、Unprotected Main Program 壞味道快速修復結果

根據洪哲偉的論文，我們發現此壞味道的快速修復功能會產生兩種問題。

第一種問題為：雖然透過快速修復功能將 Unprotected Main Program 消除了，但會產生 Empty Catch Block。消除 Unprotected Main Program 的方法為用 try/catch 將 main program 保護住，如果還不清楚 catch 要如何處理例外的話，能夠先將 catch 區塊中例外處理以顯示錯誤或記錄到日誌檔的方式來處理，最後結束程式執行。因此我們將 catch 區塊中的//TODO: handle exception 改為將例外訊息記錄到日誌檔中，讓 main program 的 catch 區塊有記錄例外訊息。

第二種問題為：根據 Unprotected Main Program 的定義，當主程式發生例外後，軟體會發生不預期的終止。如圖 3-8 為錯誤物件的繼承架構圖，雖然 catch 有去捕捉 Exception 型別的例外，但如果系統發生 Error[15]時，main program 還是會因為沒有捕捉 Error[15]而造成系統異常終止。因此我們將 catch 捕捉的 Exception[14]改為 Throwable[16]，如圖 3-9 為 Unprotected Main Program 改善後的快速修復結果。

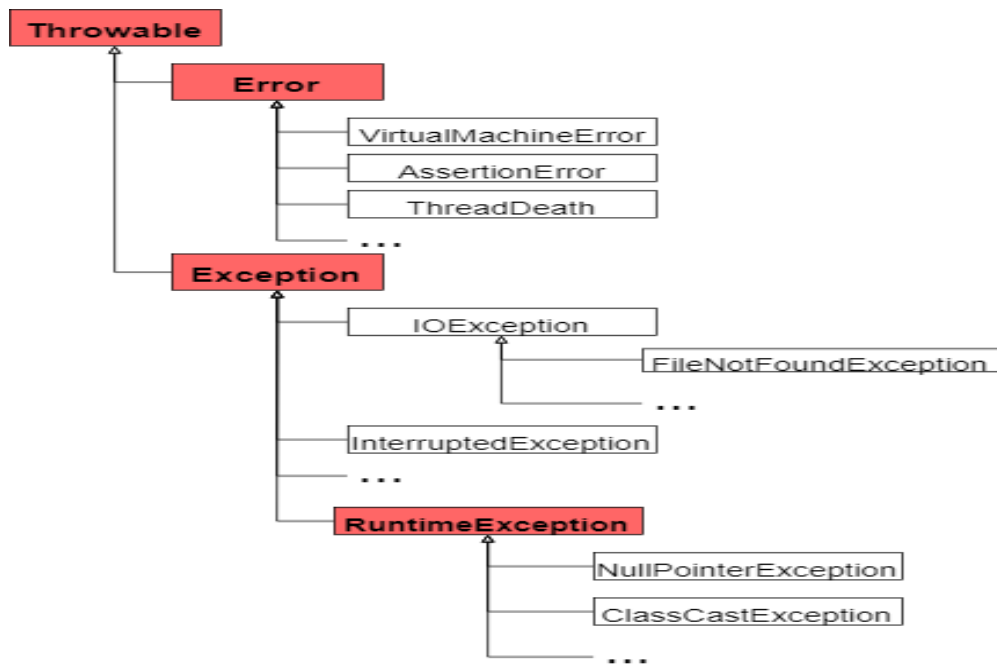


圖 3-8、錯誤物件的繼承架構

```

9     private static Logger logger = Logger.getLogger(UnprotectedMainProgram.class);
10
11     public static void main(String[] args) {
12         try {
13             int[] array = new int[9];
14             for (int i = 0; i <= 9; i++) {
15                 array[i] = i;
16             }
17         } catch (Throwable ex) {
18             PropertyConfigurator.configure("log4j.properties");
19             logger.info(ex);
20         }
21     }
  
```

圖 3-9、Unprotected Main Program 改善後的快速修復結果

3.1.3 Nested Try Statement

Nested Try Statement 為 Try Statement[8]裡的 try、catch 或 finally 區塊中又有 Try Statement，多層的巢狀結構容易讓開發人員或維護人員不容易閱讀程式碼。消除 Nested Try Statement 壞味道的方法為將巢狀結構的 Try Statement 抽成一個函式，降低程式碼的複雜度。根據洪哲偉的論文[3]，因為 Eclipse 已經提供重構功能，所以不在 Robusta 中

提供重構功能。因此如果要消除 Nested Try Statement，先將要抽成函式的 Try Statement 圈選起來，點擊滑鼠右鍵選取 Refactor→Extract Method...按鈕；接著在 Extract Method 的視窗中替選取的 Try Statement 命名要抽成的函式名稱，點擊 OK 後 Nested Try Statement 就消除了。如圖 3-10 所示，選取第 9~15 行的 Try Statement 結構[8]，選取 Extract Method；接著如圖 3-11 所示，在視窗中的 Method name 填寫要命名的函式名稱；最後如圖 3-12 所示，Nested Try Statement 就被消除了。

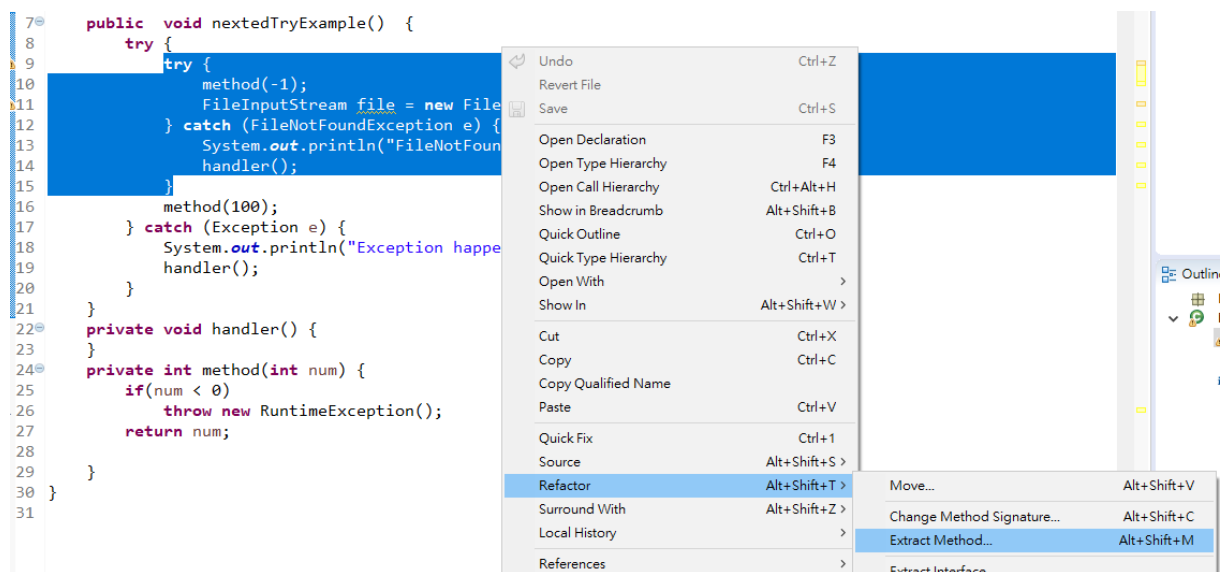


圖 3-10、選取 Eclipse 提供的 Refactor→Extract Method...功能

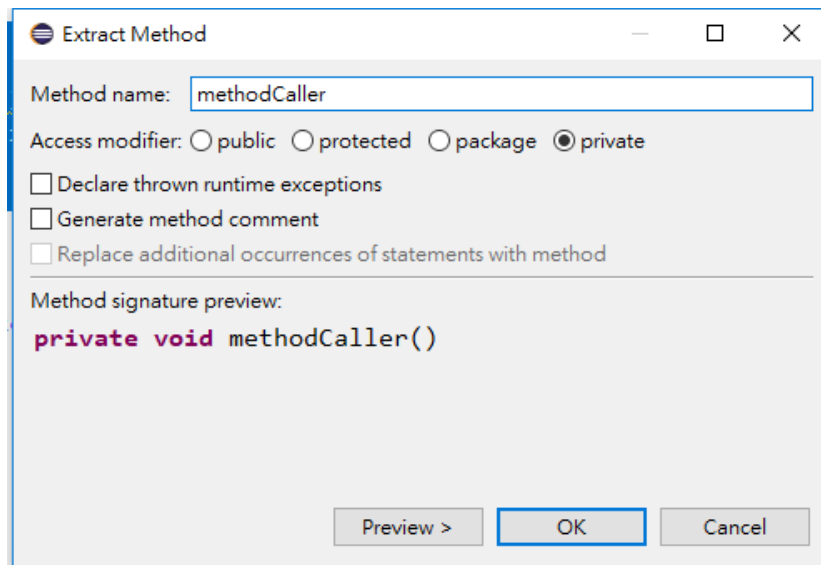


圖 3-11、在 Extract Method 視窗替函式命名

```
7 public void nextedTryExample() {
8     try {
9         methodCaller();
10        method(100);
11    } catch (Exception e) {
12        System.out.println("Exception happen!");
13        handler();
14    }
15 }
16 private void methodCaller() {
17     try {
18         method(-1);
19         FileInputStream file = new FileInputStream("example.txt");
20     } catch (FileNotFoundException e) {
21         System.out.println("FileNotFoundException happen!");
22         handler();
23     }
24 }
```

圖 3-12、Nested Try Statement 重構結果

我們發現在 Robusta 中有提供 Nested Try Statement 的自動化重構功能，雖然重構後的結果是一樣的，但 Robusta 會與 Eclipse 的 Extract Method 功能做結合，使 Nested Try Statement 的重構功能自動化，因此使用者不用再圈選出要獨立抽成函式的 Try Statement。如圖 3-13 所示，點擊 Robusta 的「Refactor==>Extract Method」後，Robusta 會自動連結

Eclipse 的 Extract Method 功能，而顯示如圖 3-11 的 Extract Method 視窗，填寫函式名稱後，如圖 3-12 所示，Nested Try Statement 就被消除了。

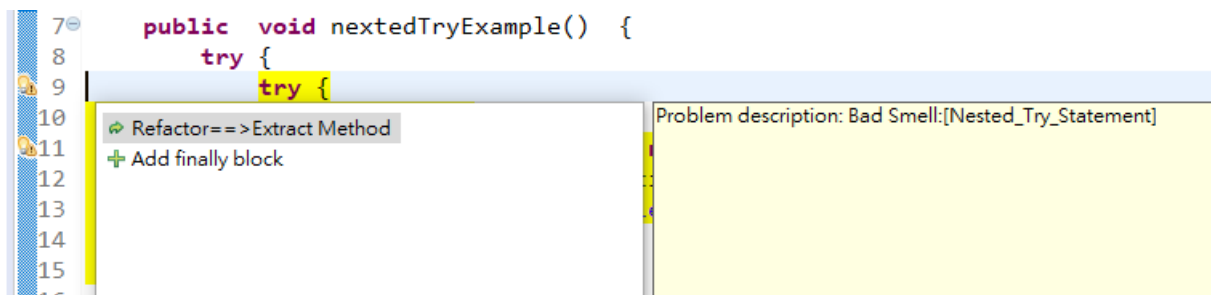


圖 3-13、Robusta 提供的 Extract Method 功能

3.1.4 Careless Cleanup

Careless Cleanup 為當程式碼執行到釋放資源之前發生例外，會造成釋放資源的程式碼不會被執行。因此要消除 Careless Cleanup，首先在有壞味道的 Try Statement 增加 finally 區塊；接著，為了避免程式碼在執行釋放資源時發生意外，讓例外在 finally 區塊被丟出，而造成 Exception Thrown From Finally，因此在 finally block 裡加入 try/catch 保護程式碼，並在 catch block 中將捕捉資源釋放所發生的例外訊息記錄到日誌檔中；最後，在 try block 裡將釋放資源的程式碼放入，並在執行釋放資源之前，檢查物件是否為空。如圖 3-14 所示，為 Careless Cleanup 快速修復後的結果，雖然會產生 Nested Try Statement，但再使用 Robusta 提供的重構功能來消除即可。


```

11     private static Logger logger = Logger.getLogger(CarelessCleanup.class);
12
13     public void carelessCleanupExample() throws IOException{
14         FileInputStream fileInputStream = null;
15         try {
16             fileInputStream = new FileInputStream("test.txt");
17             fileInputStream.read();
18         } catch (IOException e) {
19
20             handle();
21         } finally {
22             try {
23                 if(fileInputStream != null)
24                     fileInputStream.close();
25             } catch (IOException e) {
26                 /*
27                  * Although it's a Dummy Handler bad smell,
28                  * Robusta recommends that you keep this bad smell
29                  * instead of choosing Quick Fix or Refactor that we provide.
30                  */
31                 PropertyConfigurator.configure("log4j.properties");
32                 logger.info(e);
33             }
34         }
35     }

```

圖 3-14、Careless Cleanup 快速修復結果

3.1.5 Exception Thrown From Finally Block

當 finally block 裡的程式碼丟出例外時，會覆蓋掉原先 try 或 catch 區塊中所丟出的例外，而誤導開發人員或維護人員關注在 finally block[8]丟出的例外，而忽略 try 或 catch 區塊中所丟出的例外，稱為 Exception Thrown From Finally Block。

要消除這個壞味道的方法，就是不要讓 finally block 丟出例外。因此，要消除 Exception Thrown From Finally Block，首先將 finally block 會丟出例外的程式碼用 try/catch 保護住；接著，當釋放資源失敗而在 catch 捕捉到例外後，記錄到日誌檔中；最後，將 finally 裡的這個 Try Statement 獨立抽成一個函式即可。如圖 3-15 所示，為 Robusta 對 Exception Thrown From Finally Block 的重構結果。消除 Exception Thrown From Finally Block 後，雖然會產生 Dummy Handler，但在 Exception Thrown From Finally Block 跟 Dummy Handler 這兩種壞味道對程式碼影響的權衡之下，我們認為留下 Dummy Handler

對程式碼的影響相對來說是比較小的，因此才會在 Exception Thrown From Finally Block 消除後而殘留下 Dummy Handler。

```
8 public void exceptionThrownFormFinallyBlockExample() throws IOException {
9     FileInputStream fileInputStream = null;
10    try {
11        fileInputStream = new FileInputStream("test.txt");
12        fileInputStream.read();
13    } catch (IOException e) {
14        throw new IOException("Exception thrown in catch");
15    } finally {
16        Close(fileInputStream);
17    }
18 }
19
20 private void Close(FileInputStream fileInputStream) {
21     try {
22         fileInputStream.close();
23     } catch (IOException e) {
24         e.printStackTrace();
25     }
26 }
```

圖 3-15、Exception Thrown From Finally Block 原先的重構結果

雖然透過 Robusta 消除了 Exception Thrown From Finally Block，但衍生的 Dummy Handler 仍然會在 Robusta 提供的壞味道分析報表中顯示。為了避免使用者對殘留下來的 Dummy Handler 抱有疑慮，如圖 3-16 所示，我們將殘留的 Dummy Handler 增加一些註解，向使用者解釋為何我們在這裡留下 Dummy Handler。此外，我們也將印出例外訊息的部分修改為將例外訊息記錄到日誌檔中。

```

11 private Logger logger = Logger.getLogger(ExceptionThrownFormFinallyBlock.class.getName());
12
13 public void exceptionThrownFormFinallyBlockExample() throws IOException {
14     FileInputStream fileInputStream = null;
15     try {
16         fileInputStream = new FileInputStream("test.txt");
17         fileInputStream.read();
18     } catch (IOException e) {
19         throw new IOException("Exception thrown in catch");
20     } finally {
21         Close(fileInputStream);
22     }
23 }
24
25 private void Close(FileInputStream fileInputStream) {
26     try {
27         fileInputStream.close();
28     } catch (IOException e) {
29         /* Although it's a Dummy Handler bad smell,
30          * Robusta recommends that you keep this bad smell
31          * instead of choosing Quick Fix or Refactor that we provide.
32          */
33         PropertyConfigurator.configure("log4j.properties");
34         logger.info(e);
35     }
36 }

```

圖 3-16、Exception Thrown From Finally Block 改善後的重構結果

3.2 過去與現在 Robusta 快速修復與重構差異

本小節整理 Robusta 提供的快速修復和重構功能，將 Robusta 原本存在的快速修復與重構功能，和本論文新增或改善的快速修復與重構功能做比較。

3.2.1 快速修復

表 3-1、Robusta 快速修復功能差異

壞味道種類	原本的 Robusta	本論文改善後提供的方法
Dummy Handler & Empty Catch Block	catch 捕捉到的例外後丟出 RuntimeException 來進行回報。	將 catch 捕捉到的例外型別直接丟出來進行回報。
Unprotected Main Program	產生 try/catch 保護主程式，且 catch 捕捉 Exception 類別後不做任何事。	產生 try/catch 保護主程式，catch 捕捉 Throwable 類別並將例外寫入日誌中。
Careless Cleanup	無	增加快速修復功能。

3.2.2 重構

表 3-2、Robusta 重構功能差異

壞味道種類	原本的 Robusta	本論文改善後提供的方法
Dummy Handler & Empty Catch Block	使用者能夠選擇自己定義的例外類別來將例外丟出。	使用者能夠選擇 Unchecked 例外類別，包含使用者定義的 Unchecked 例外，來將例外丟出。
Nested Try Statement	與 Eclipse 的 Extract Method 做連結，自動化重構程式碼來消除壞味道。	無
Exception Thrown From Finally Block	將釋放資源的函式用 try/catch 包住，並且 Extract Method，讓使用者自己定義獨立出來的函式名稱，自動化重構程式碼來消除壞味道。	改善 Robusta 自動化重構功能，將獨立出來的函式修正為 catch 捕捉例外後，將例外寫入日誌檔中，並增加註解，解釋為什麼這裡留下 Dummy Handler。

3.3 設計與實作

在 Robusta 偵測完例外處理壞味道後，會標示警告訊息提醒使用者要消除壞味道，並針對不同的壞味道來提供快速修復或重構的功能。

3.3.1 快速修復

在 Dummy Handler、Empty Catch Block、Unprotected Main Program 和 Careless Cleanup 中，有提供快速修復(QuickFix)的功能來消除壞味道。當使用者點擊警告訊息要消除壞味道時，RLQuickFixer 會根據使用者所選擇的壞味道種類來提供對應的快速修復方法。

3.3.1.1 Dummy Handler & Empty Catch Block

當使用者點擊 Dummy Handler 和 Empty Catch Block 的 Quick Fix 功能時，RLQuickFixer 類別會觸發 DummyQuickFix 類別，執行快速修復的功能來消除壞味道。

1. 透過標記的 IMarker[17]來取得壞味道在該 Java 文件的 MethodDeclaration index[8]。
2. 設定 AST[8]相關資訊取得 CompilationUnit。
3. 藉由取得的 MethodDeclaration index 和 CompilationUnit 取得壞味道所在的 MethodDeclaration。
4. 取得該 MethodDeclaration 中指定壞味道所在的 catch 資訊。
5. 將 MethodDeclaration 的介面宣告指定 catch 所接住的例外型別。
6. 檢查 catch 是否有印出例外訊息的方法，若有則移除。
7. 將 catch 捕捉到的例外向上層回報出去。

完成上述步驟即可完成 Dummy Handler 和 Empty Catch Block 自動化快速修復功能。

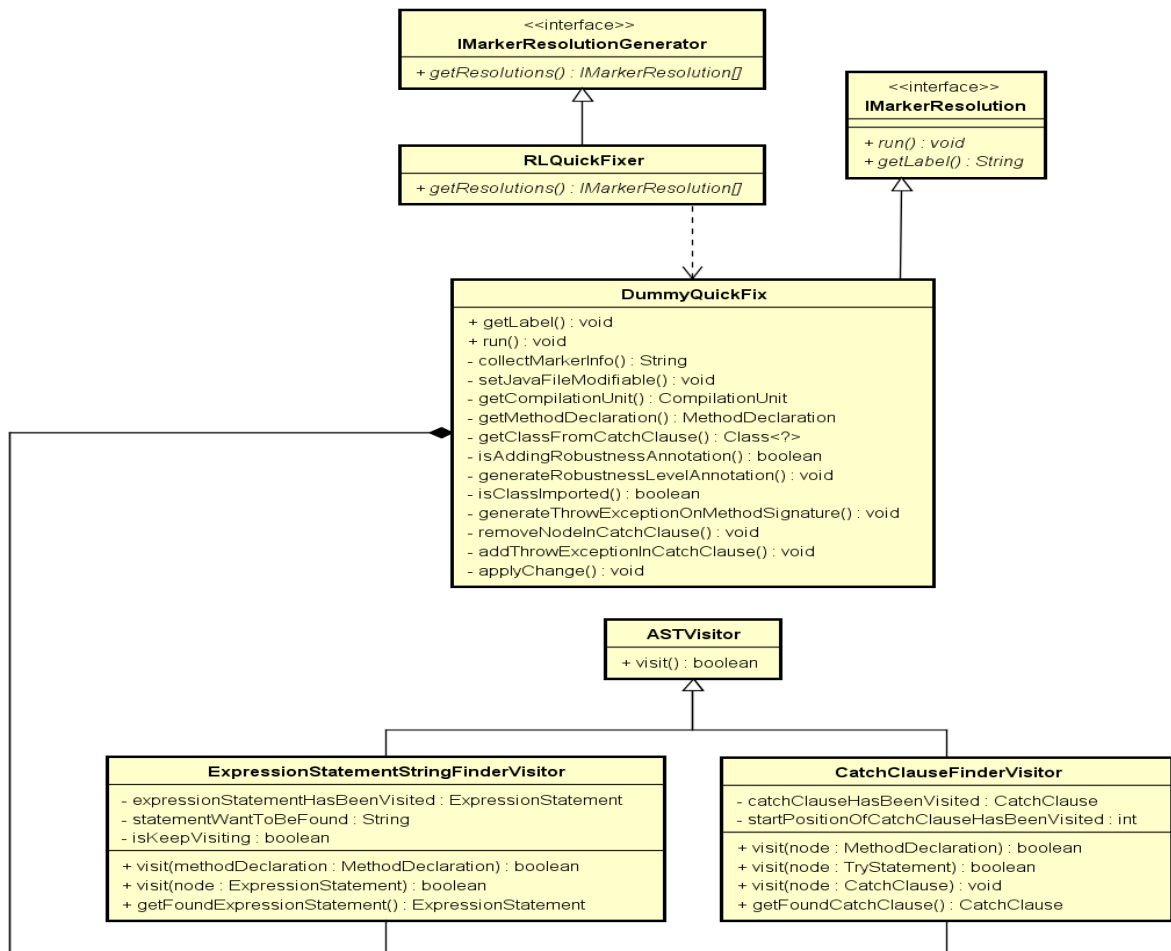


圖 3-17、Dummy Handler 和 Empty Catch Block QuickFix 功能的 Class Diagram

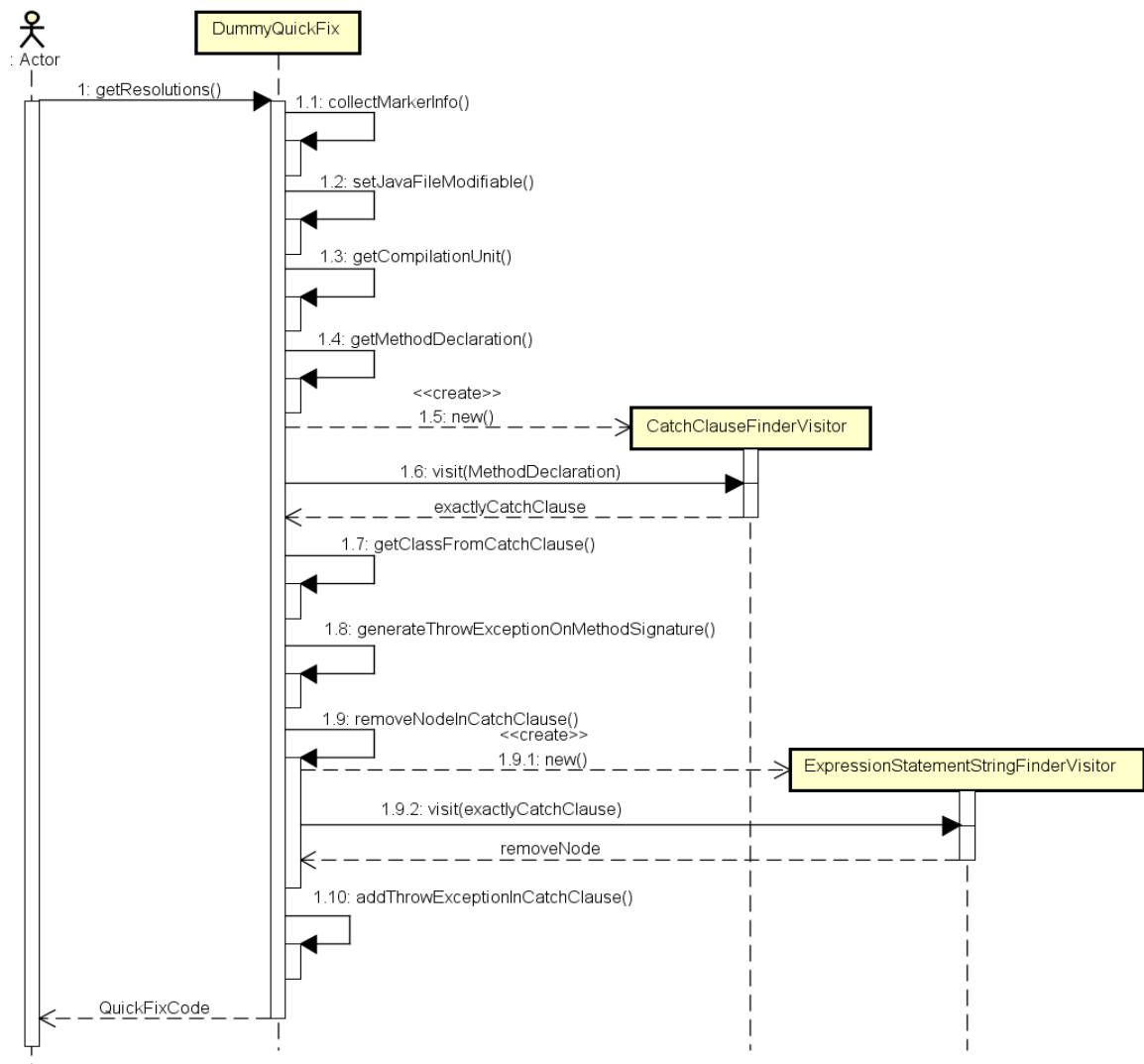


圖 3-18、Dummy Handler 和 Empty Catch Block QuickFix 功能的 Sequence Diagram

3.3.1.2 Unprotected Main Program

當使用者點擊 Unprotected Main Program 的 Quick Fix 功能時，RLQuickFixer 類別會觸發 UMQuickFix 類別，執行快速修復的功能來消除壞味道。

1. 透過標記的 IMarker 來取得壞味道在該 Java 文件的 MethodDeclaration index 。
2. 設定 AST 相關資訊取得 CompilationUnit 。
3. 藉由取得的 MethodDeclaration index 和 CompilationUnit 取得壞味道所在的 MethodDeclaration，此 MethodDeclaration 即為主程式。
4. 產生 try/catch 將主程式保護住，在這一步驟中，首先會先蒐集主程式內所有的 Try Statement，而將主程式分為兩種情境：主程式內有 Try Statement 和主程式內沒有 Try Statement。
 - 主程式內沒有 Try Statement：
 - (1) 產生 try/catch，並在 catch 捕捉 Throwable 類別，並將捕捉到的例外記錄日誌檔中。
 - (2) 將主程式內的程式碼移入 try 區塊裡。
 - 主程式內有 Try Statement：
 - (1) 檢查 catch 是否有捕捉 Throwable 類別或是 Exception 類別，如果沒有，則增加捕捉 Throwable 類別的 catch 區塊，並將例外捕捉後記錄日誌檔中。
 - (2) 將不在 try/catch 裡的程式碼移入 try/catch 中。

完成上述步驟即可完成 Unprotected Main Program 自動化快速修復功能。

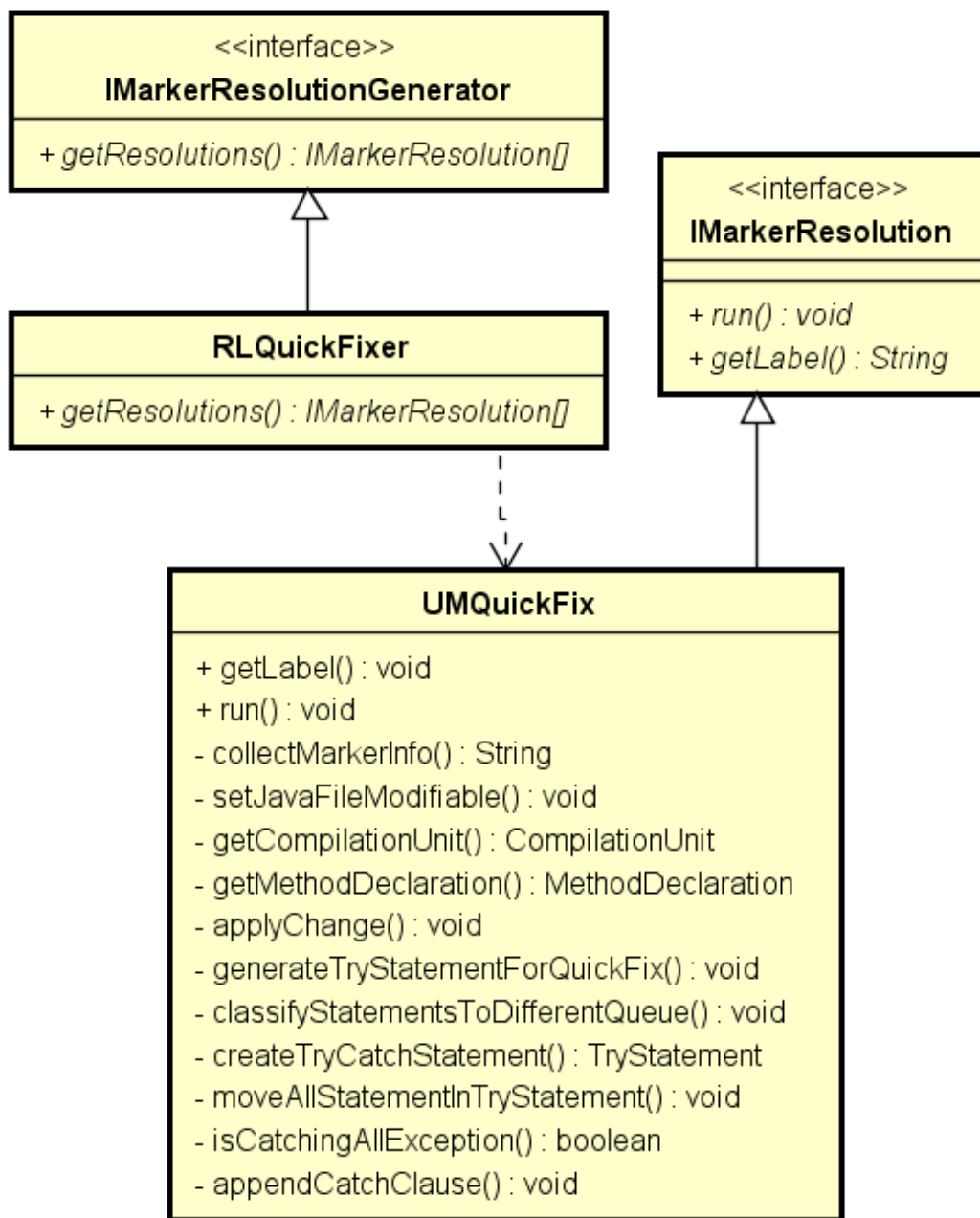


圖 3-19、Unprotected Main Program QuickFix 功能的 Class Diagram

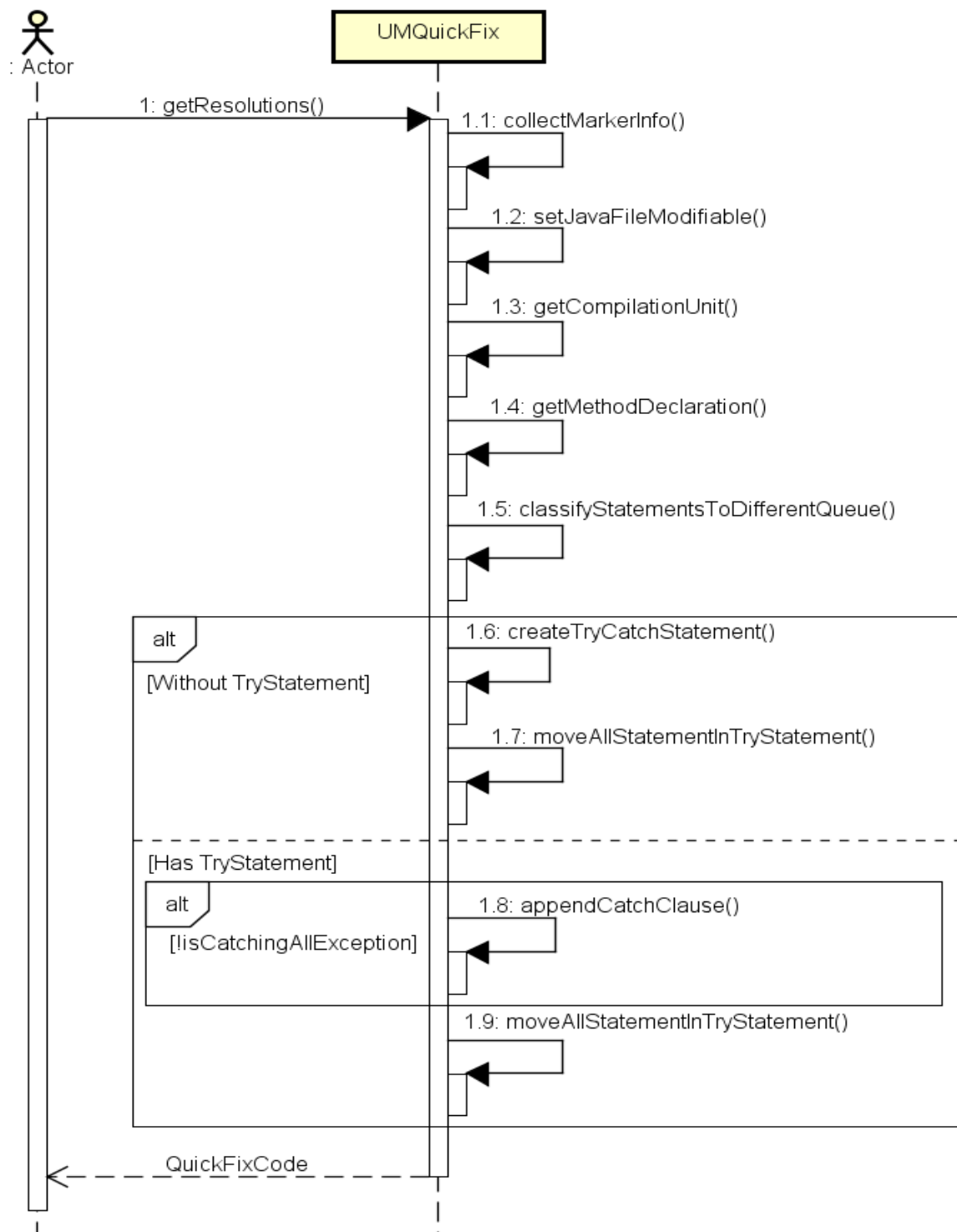


圖 3-20、Unprotected Main Program QuickFix 功能的 Sequence Diagram

3.3.1.3 Careless Cleanup

當使用者點擊 Careless Cleanup 的 Quick Fix 功能時，RLQuickFixer 類別會觸發 CCQuickFix 類別，執行快速修復的功能來消除壞味道。

1. 透過標記的 IMarker 來取得壞味道在該 Java 文件的 MethodDeclaration index。
2. 設定 AST[8]相關資訊取得 CompilationUnit。
3. 藉由取得的 MethodDeclaration index 和 CompilationUnit 取得壞味道所在的 MethodDeclaration。
4. 分析 IMarker，取得壞味道的行數。
5. 藉由壞味道的行數和 MethodDeclaration 取得釋放資源的函式。
6. 判斷 MethodDeclaration 內是否 Try Statement，如果有 Try Statement，則
 - (1) 取得釋放資源函式會丟出的例外型別。
 - (2) 在 MethodDeclaration 的介面宣告釋放資源函式會丟出的例外。
7. 產生 Try Statement，並將釋放資源的函式放入 finally block 中。
8. 將原來釋放資源的函式移除。
9. 將 MethodDeclaration 內剩下的函式移入 Try Statement。

完成上述步驟即可完成 Careless Cleanup 自動化快速修復功能。

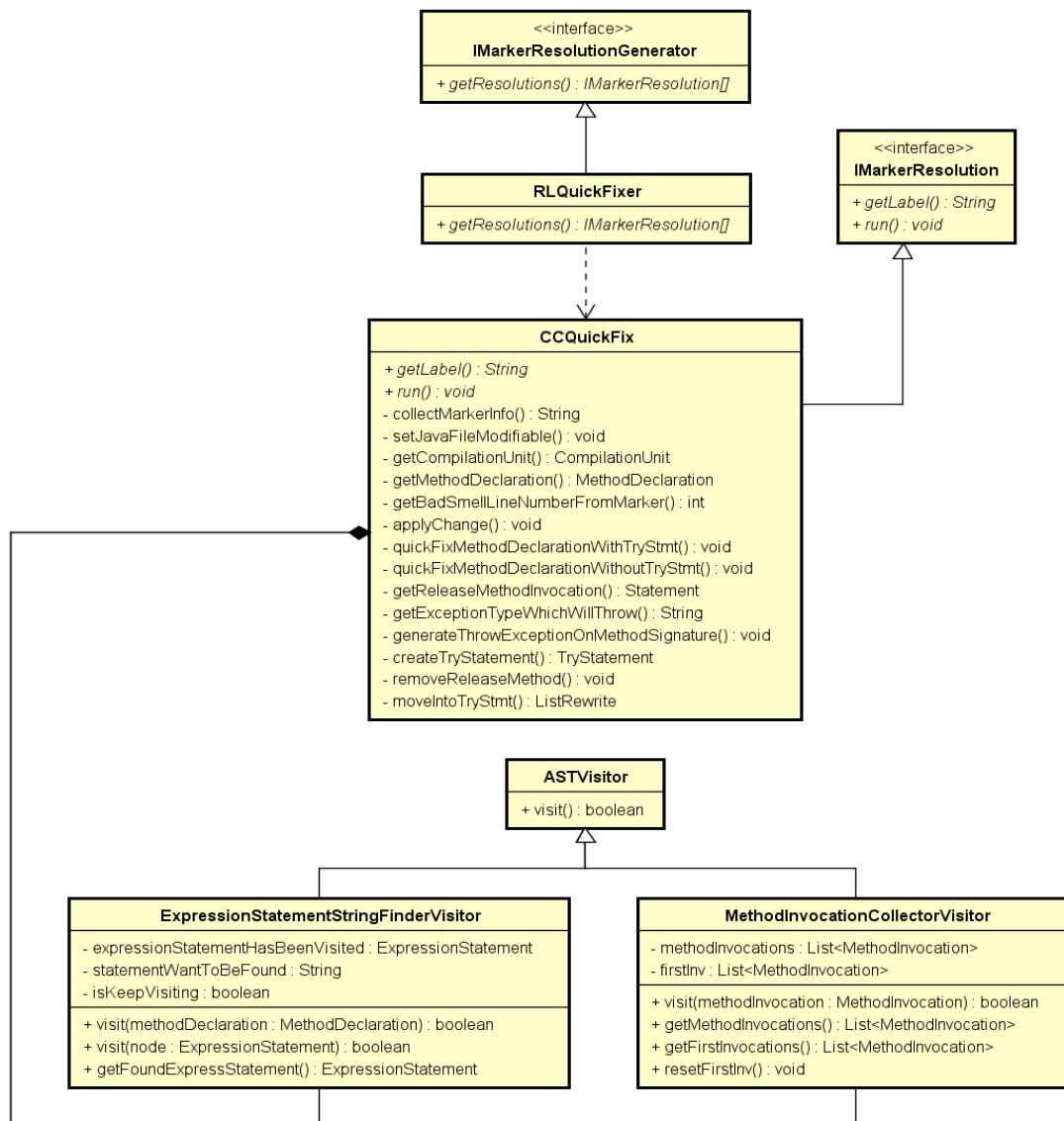


圖 3-21、Careless Cleanup QuickFix 功能的 Class Diagram

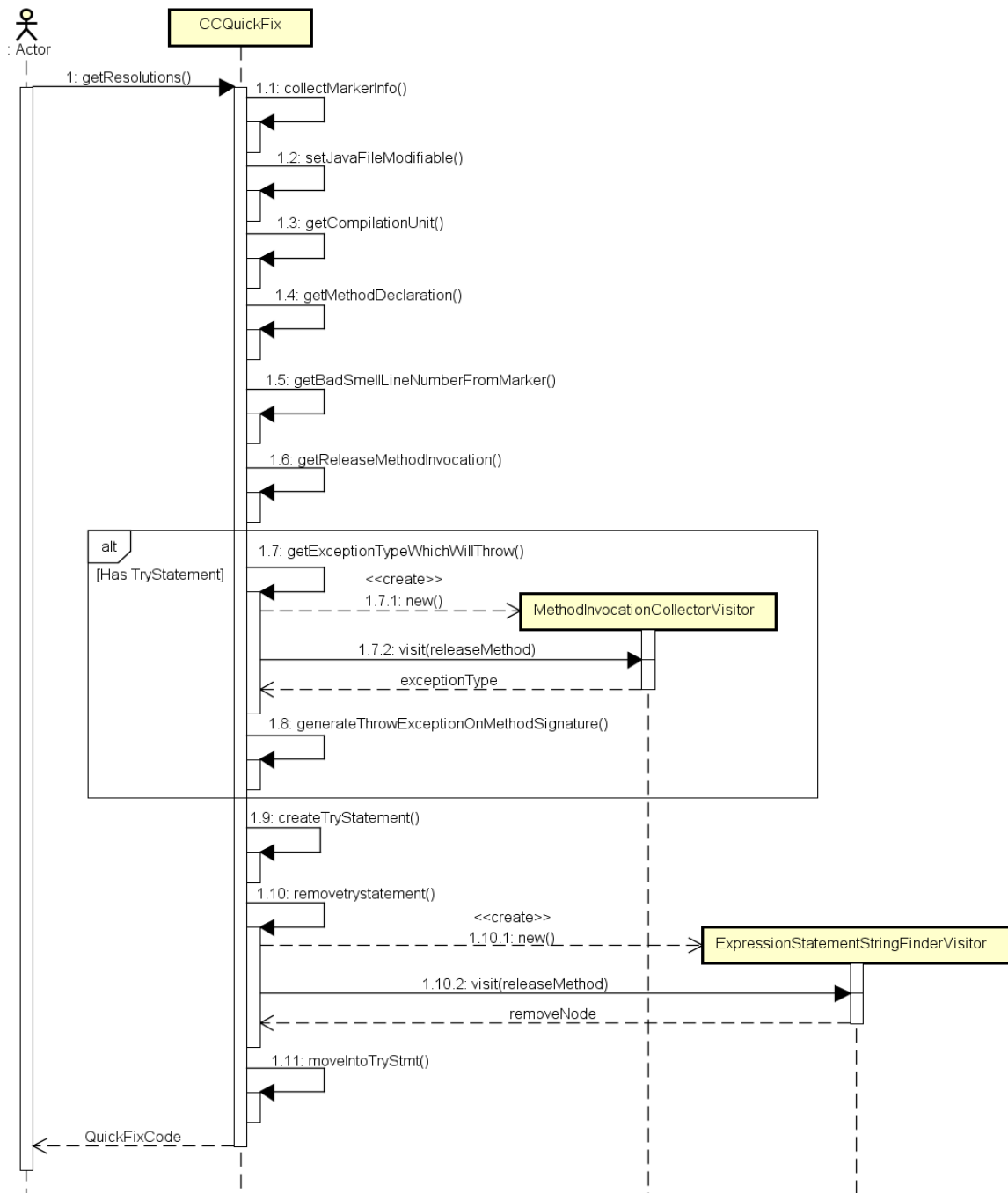


圖 3-22、Careless Cleanup QuickFix 功能的 Sequence Diagram

3.3.2 重構

在 Dummy Handler、Empty Catch Block、Nested Try Statement 和 Exception Thrown From Finally Block 中提供了壞味道消除的重構的功能。當使用者點擊警告訊息要消除壞味道時，RLQuickFixer 會根據使用者所選擇的壞味道種類來提供對應的重構方法。

3.3.2.1 Dummy Handler & Empty Catch Block

當使用者在 Dummy Handler 和 Empty Catch Block 選擇警告訊息提供的重構方法時，RLQuickFixer 會執行 RethrowUncheckedExAction 的類別，啟動重構頁面讓使用者設定相關參數和要回報的例外型別等。RethrowExWizard 繼承 Eclipse 的 RefactoringWizard 類別，因此能夠提供預覽畫面協助使用者設定重構的相關資訊。在使用者設定完成後，RethrowExInputPage 會將使用者設定的資訊傳到 RethrowExRefactoring 類別，對 Dummy Handler 和 Empty Catch Block 進行重構。

1. 新增 RethrowExWizard 類別，並產生 RethrowExInputPage 類別，提供 Unchecked Exception 選單畫面讓使用者選擇。
 2. 使用者選擇 Unchecked Exception。
 3. 將使用者設定的重構資訊傳入 RethrowExRefactoring 類別進行重構程式碼。
- 完成上述步驟即可完成 Dummy Handler 和 Empty Catch Block 自動化重構功能。

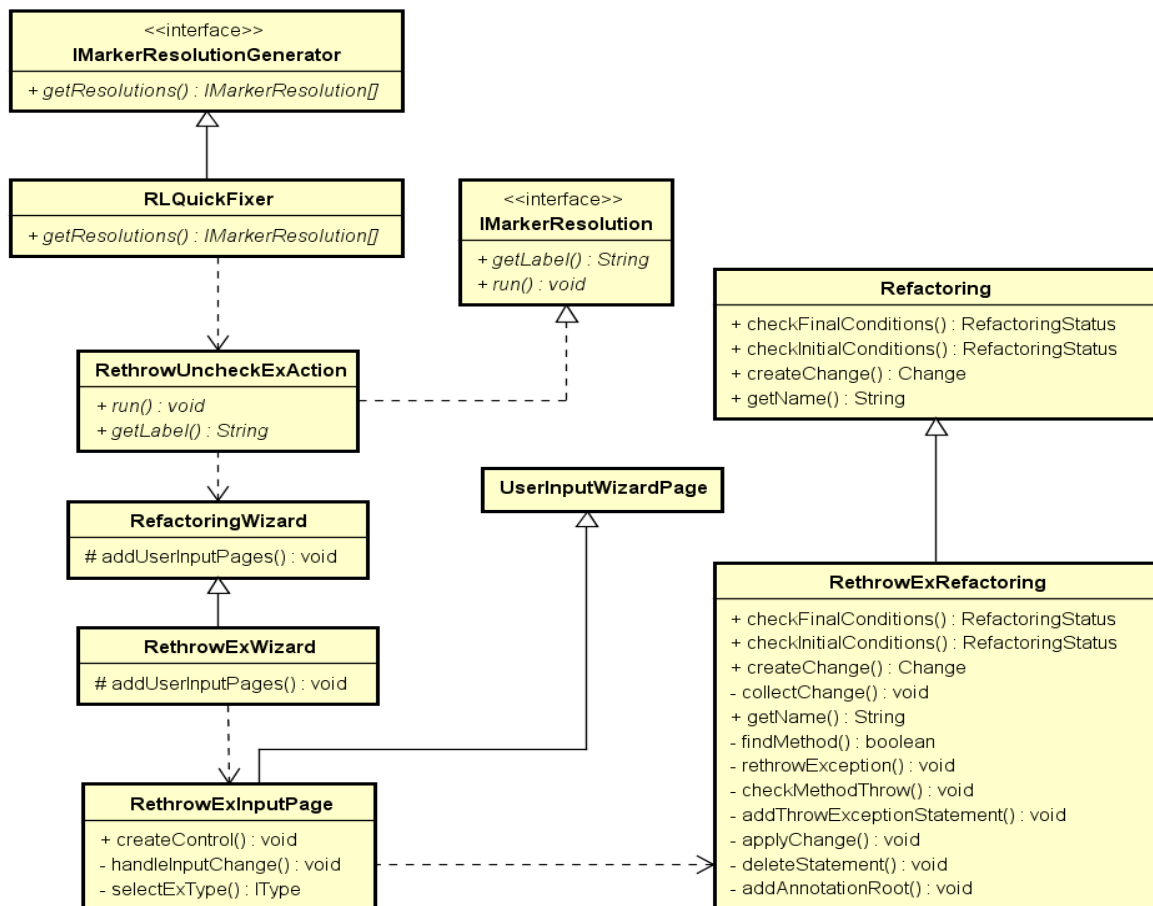


圖 3-23、Dummy Handler 和 Empty Catch Block Refactoring 功能的 Class Diagram

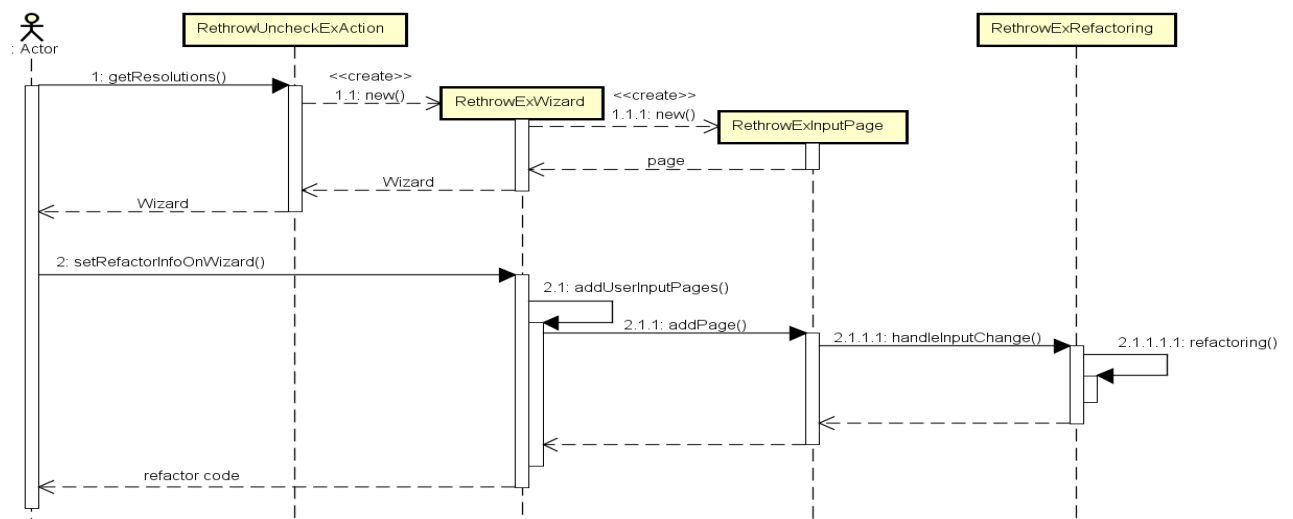


圖 3-24、Dummy Handler 和 Empty Catch Block Refactoring 功能的 Sequence Diagram

3.3.2.2 Nested Try Statement

當使用者在 Nested Try Statement 選擇警告訊息提供的重構方法時，RLQuickFixer 會執行 NTMarkerResolution 的類別，與 Eclipse 結合來啟動重構頁面讓使用者設定相關參數和要回報的例外型別等。連結 Eclipse 的 ExtractMethodWizard 類別，它繼承了 RefactoringWizard 類別，提供預覽畫面協助使用者設定重構的相關資訊。在使用者設定完成後，連結 Eclipse 的 ExtractMethodInputPage，將使用者設定的資訊傳到 Eclipse 的 ExtractMethodRefactoring 類別，對 Nested Try Statement 進行重構。

1. 新增 ExtractMethodWizard 類別，並產生 ExtractMethodInputPage 類別，提供 Extract Method 畫面讓使用者設定。
2. 使用者設定 Extract Method 資訊。
3. 將使用者設定的重構資訊傳入 ExtractMethodRefactoring 類別進行重構程式碼。

完成上述步驟即可完成 Nested Try Statement 自動化重構功能。

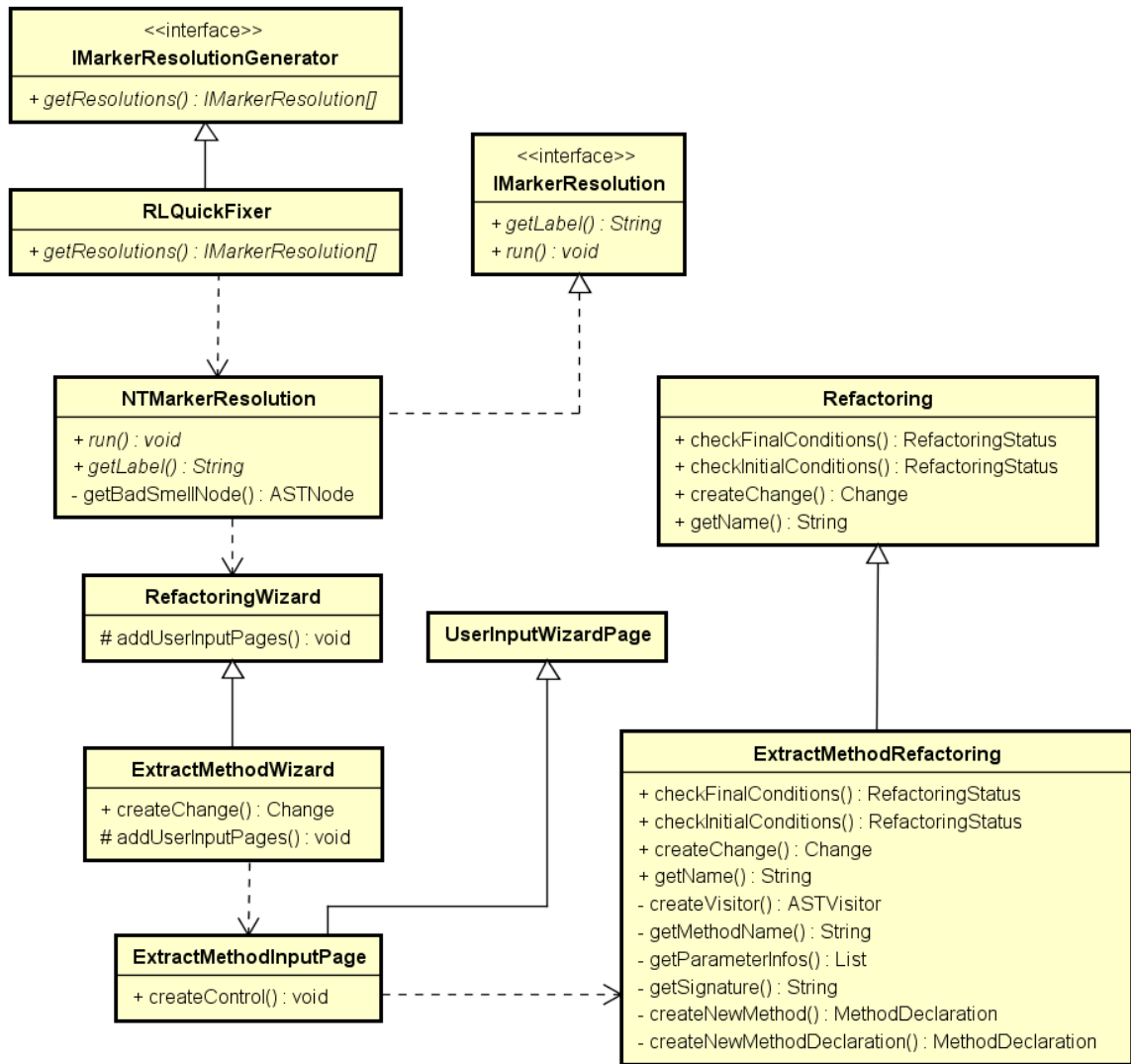


圖 3-25、Nested Try Statement Refactoring 功能的 Class Diagram

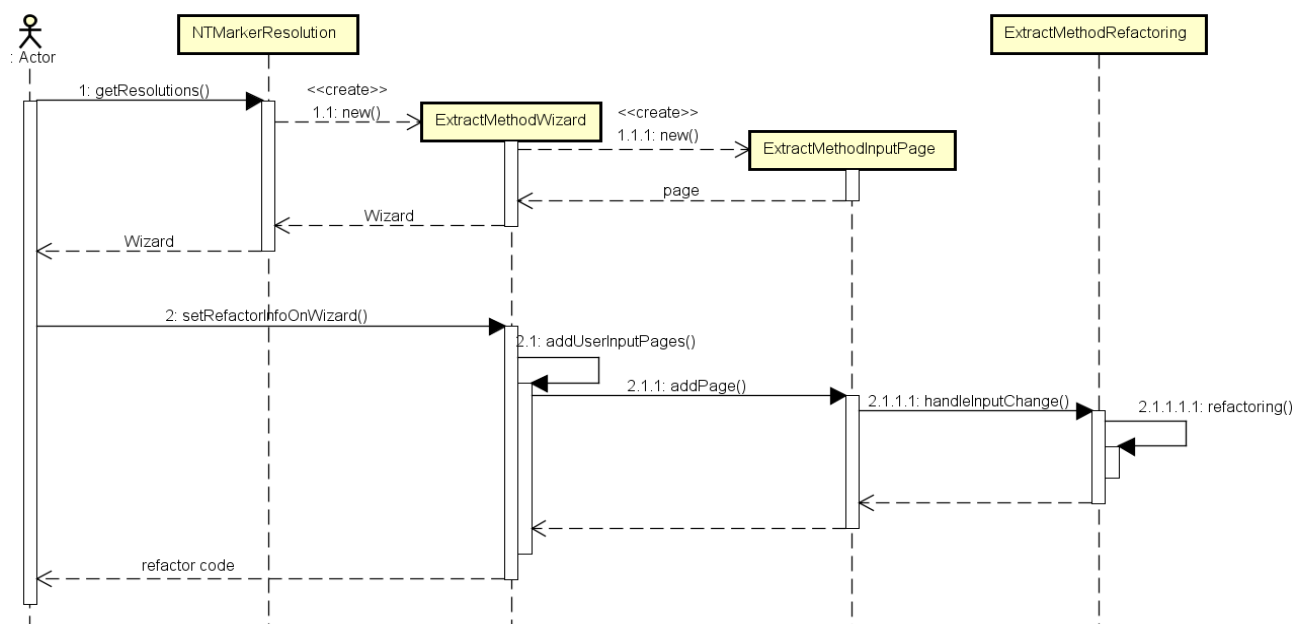


圖 3-26、Nested Try Statement Refactoring 功能的 Sequence Diagram

3.3.2.3 Exception Thrown From Finally Block

當使用者在 Exception Thrown From Finally Block 選擇警告訊息提供的重構方法時，RLQuickFixer 會執行 TEFBExtractMethodMarkerResolution 的類別，啟動重構頁面讓使用者設定相關參數和要回報的例外型別等。CodeSmellRefactoringWizard 繼承 Eclipse 的 RefactoringWizard 類別，因此能夠提供預覽畫面協助使用者設定重構的相關資訊。在使用者設定完成後，ExtractMethodInputPage 會將使用者設定的資訊傳到 TEFBExtractMethodRefactoring 類別，對 Exception Thrown From Finally Block 進行重構。

1. 新增 CodeSmellRefactoringWizard 類別，並產生 ExtractMethodInputPage 類別，提供 Extract Method 畫面讓使用者設定。
 2. 使用者設定 Extract Method 資訊。
 3. 將使用者設定的重構資訊傳入 TEFBExtractMethodRefactoring 類別進行重構程式碼。
- 完成上述步驟即可完成 Exception Thrown From Finally Block 自動化重構功能。

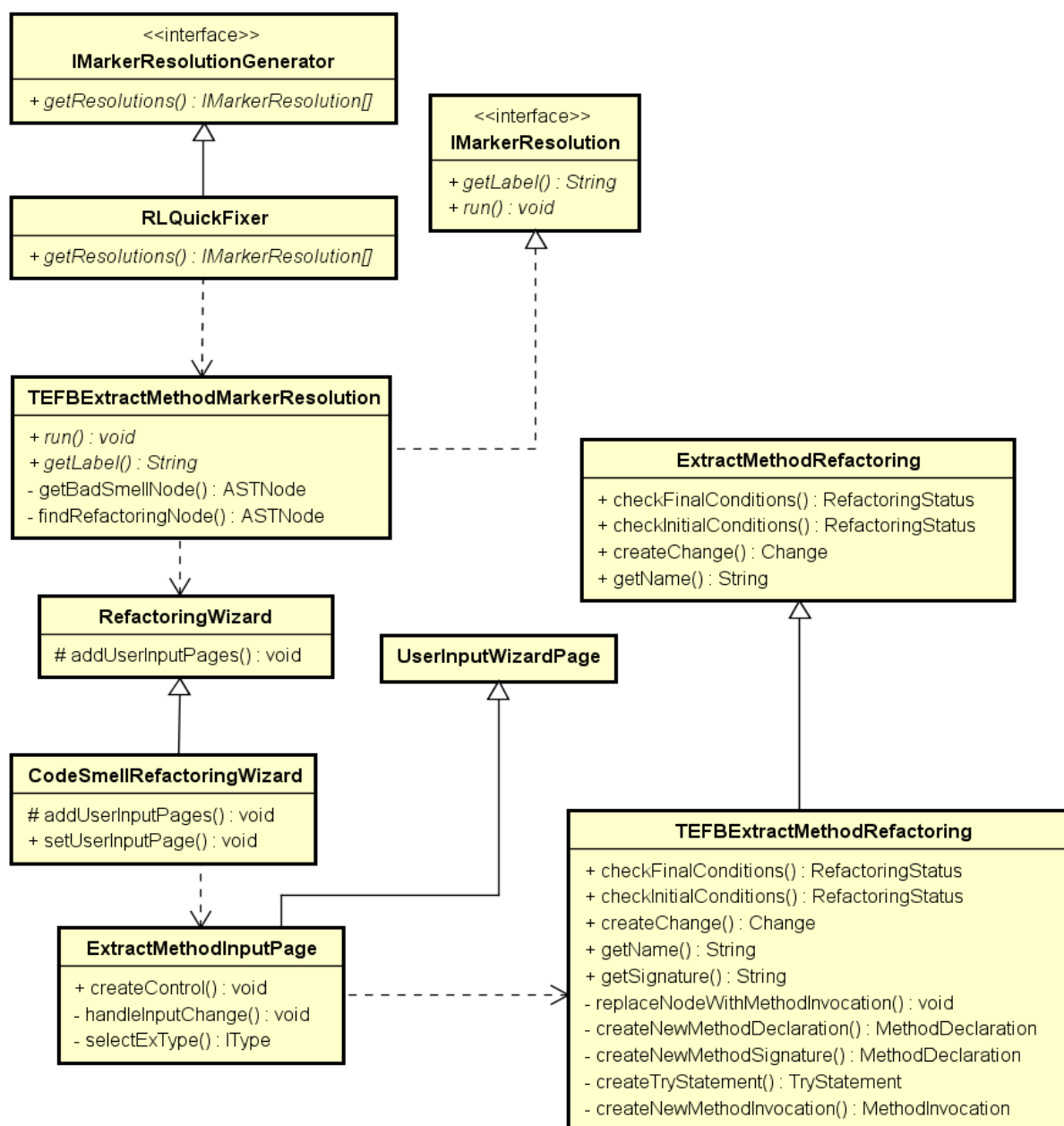


圖 3-27、Exception Thrown From Finally Block Refactoring 功能的 Class Diagram

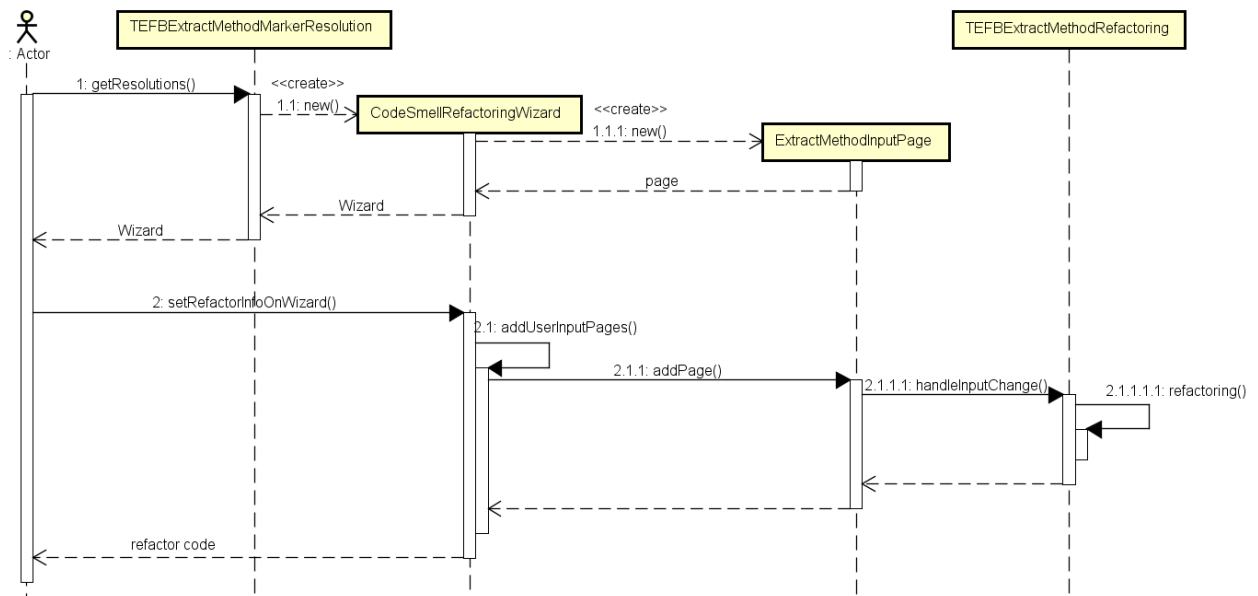


圖 3-28 、Exception Thrown From Finally Block Refactoring 功能的 Sequence Diagram

3.4 壞味道的偵測、曝露及消除流程

要消除例外處理壞味道，我們提供了一套偵測、曝露及消除壞味道的流程[18]，總共有五個步驟。第一步：對專案點擊右鍵→Robusta→Detect smell，來偵測專案有哪些例外處理壞味道，產生例外處理壞味道的報表，並透過報表來到某一段有例外處理壞味道的程式碼，程式碼旁邊會有 Robusta 對壞味道的標記；第二步：產生曝露程式碼壞味道的測試案例，根據劉彥麟論文[19]提供的測試案例並藉由它來曝露出壞味道所帶來的影響；第三步：執行測試案例，如果程式碼沒有正確的處理例外，則測試會失敗；第四步：藉由本論文介紹的壞味道消除方法，消除程式碼的壞味道並正確的處理例外；最後：再執行一次剛剛失敗的測試案例，測試通過代表例外已經被正確的處理，強健度因此提升。如圖 3-29 所示，為我們提供的壞味道曝露及消除流程圖。

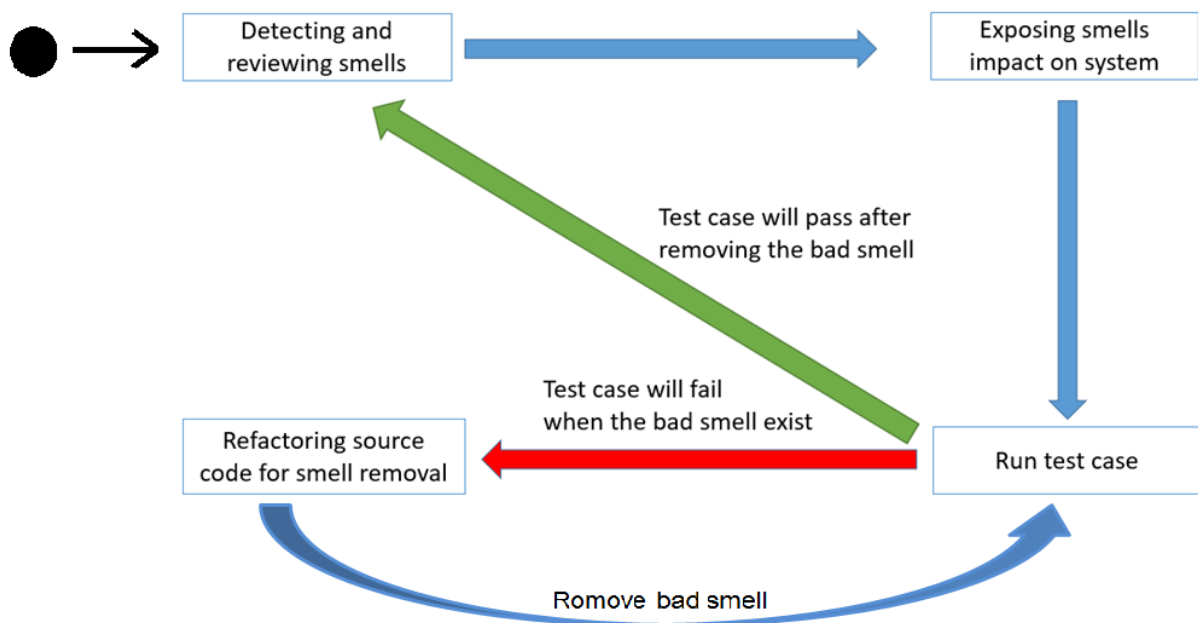


圖 3-29、壞味道偵測、曝露及消除流程圖

第四章 應用實例

本章節會以不同的開源專案：JFreeChart 和 Tomighty 為範例，將第三章介紹的壞味道消除方法實作在 Robusta 中並使其自動化，再透過壞味道消除流程[18]來對案例進行分析與應用。

JFreeChart 由 Java 語言所撰寫，是 Java 平台上的免費的圖表庫，能夠支援多種圖表，並應用於客戶端(例如：JavaFX 或 Swing)或伺服器端(匯出各種圖表格式，例如：PNG、SVG)。本論文將用 JFreeChart 來分析 Dummy Handler 及 Exception Thrown From Finally Block 兩種壞味道。

Tomighty 是一個採用「番茄鐘工作法」的桌面的時間管理工具，由 Java 語言所撰寫，來幫助使用者管理時間，提高工作效率。本論文將用 Tomighty 來分析 Unprotected Main Program 壞味道。

4.1 Dummy Handler 應用實例

4.1.1 偵測 Dummy Handler

如圖 4-1 所示，Robusta 偵測 JFreeChart 後，發現專案中 readPieDatasetFromXML 的函式第 99、102 行含有 Dummy Handler。這是一段讀 XML 檔中 DataSets 的程式碼，catch block[8]會捕捉兩種例外，為 SAXException 和 ParserConfigurationException，我們將以 SAXException 為例來進行分析與介紹。當第 93 行的 newSAXParser()函式發生 SAXException 時，會到第 98 行被 catch 捕捉住並印出例外訊息，因此這是一個 Dummy Handler，最後程式會回傳 null。

```

87 public static PieDataset readPieDatasetFromXML(InputStream in)
88     throws IOException, SAXException {
89     PieDataset result = null;
90     SAXParserFactory factory = SAXParserFactory.newInstance();
91     try {
92         SAXParser parser = factory.newSAXParser();
93         PieDatasetHandler handler = new PieDatasetHandler();
94         parser.parse(in, handler);
95         result = handler.getDataset();
96     }
97     catch (SAXException e) {
98         System.out.println(e.getMessage());
99     }
100    catch (ParserConfigurationException e2) {
101        System.out.println(e2.getMessage());
102    }
103    return result;
104 }

```

圖 4-1、JFreeChart 中 Dummy Handler 範例

4.1.2 產生曝露 Dummy Handler 的測試案例

我們認為當程式碼在圖 4-1 的第 93 行發生例外時，程式碼已經在不正確的狀態，被 catch 捕捉後卻沒有對狀態進行回復或任何處理，只有印出例外訊息後讓程式碼繼續帶著不正確的狀態執行下去，如圖 4-2 所示，藉由劉彥麟論文[19]產生曝露壞味道影響的測試案例並執行。如圖 4-3 所示，測試案例因為程式碼沒有正確處理例外而造成測試失敗。

```

14 @Test
15 public void testDueToNewSAXParserCatchBlockShouldThrowExceptionInReadPieDatasetFromXML() {
16     aspectJSwitch.initResponse();
17     aspectJSwitch.addResponse("newSAXParser/f(SAXException)");
18     aspectJSwitch.toFirstResponse();
19     try{
20         DatasetReader.readPieDatasetFromXML(null);
21         Assert.fail("Exception is not thrown from the catch block.");
22     } catch (Exception e) {
23         String exceptionMessage = e.getMessage().toString();
24         Assert.assertTrue(exceptionMessage.contains("This exception is thrown from DummyHandler's unit test by using AspectJ."));
25     }
26 }
27 }

```

圖 4-2、JFreeChart Dummy Handler 測試案例

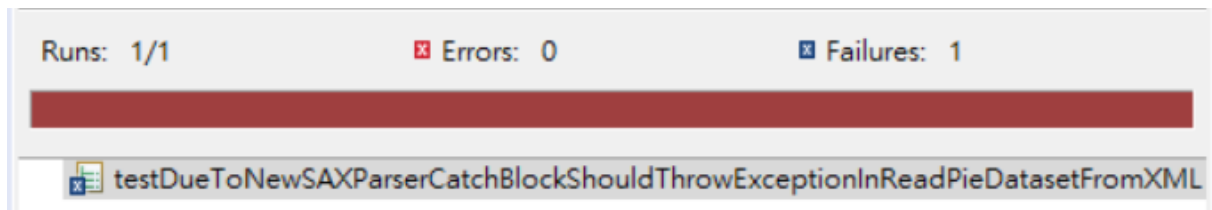


圖 4-3、JFreeChart Dummy Handler 測試失敗

4.1.3 消除 Dummy Handler

如圖 4-4 所示，接著透過本論文在 Robusta 提供的自動化快速修復或重構功能來消除壞味道，本論文以重構功能為例。如圖 4-5 所示，第 100 行為當 catch 捕捉到例外後以 RuntimeException 將例外向上層回報。如圖 4-6 所示，原先失敗的測試案例因為例外被正確處理而成功。

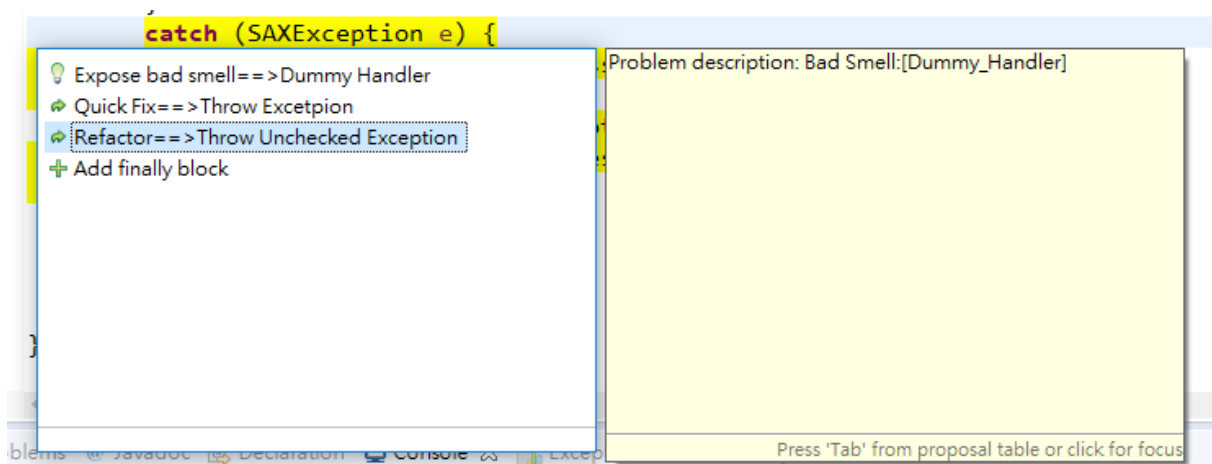


圖 4-4、Robusta 提供自動化消除 Dummy Handler

```

87= @Robustness(value = { @RTag(level = 1, exception = java.lang.RuntimeException.class) })
88 public static PieDataset readPieDatasetFromXML(InputStream in)
89     throws IOException, SAXException {
90     PieDataset result = null;
91     SAXParserFactory factory = SAXParserFactory.newInstance();
92     try {
93         SAXParser parser = factory.newSAXParser();
94         PieDatasetHandler handler = new PieDatasetHandler();
95         parser.parse(in, handler);
96         result = handler.getDataset();
97     }
98     catch (SAXException e) {
99         throw new RuntimeException(e);
100     }
101     catch (ParserConfigurationException e2) {
102         System.out.println(e2.getMessage());
103     }
104     return result;
105 }

```

圖 4-5、JFreeChart 消除 Dummy Handler 的結果

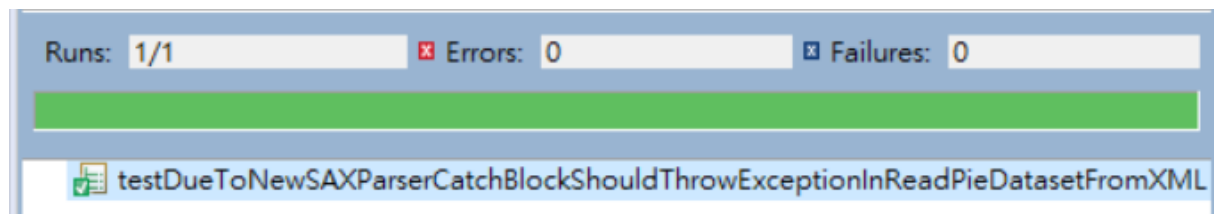


圖 4-6、JFreeChart 中 readPieDatasetFromXML 函式正確處理例外後測試成功

4.2 Careless Cleanup 應用實例

4.2.1 偵測 Careless Cleanup

如圖 4-7 所示，Robusta 偵測 JFreeChart 後，發現專案中 encode 的函式第 180 行含有 Careless Cleanup 壞味道。這是一段將圖片以 JPEG 格式來進行編碼並輸出的程式碼。當第 177、178 行的 write() 函式或 flush 函式發生 IOException 時，例外會被向上層回報，導致程式碼不會執行到第 180 行釋放資源的程式碼，因此這是一個 Careless Cleanup。

```
169= @Override
170 public void encode(BufferedImage bufferedImage, OutputStream outputStream) throws IOException {
171     Iterator iterator = ImageIO.getImageWritersByFormatName("jpeg");
172     ImageWriter writer = (ImageWriter) iterator.next();
173     ImageWriteParam p = writer.getDefaultWriteParam();
174     ImageOutputStream ios = ImageIO.createImageOutputStream(outputStream);
175     Args.notNullNotPermitted(bufferedImage, "bufferedImage");
176     Args.notNullNotPermitted(outputStream, "outputStream");
177     p.setCompressionMode(ImageWriteParam.MODE_EXPLICIT);
178     p.setCompressionQuality(this.quality);
179     writer.setOutput(ios);
180     writer.write(null, new IIOMemorySource(bufferedImage, null, null), p);
181     ios.flush();
182     writer.dispose();
183     ios.close();
184 }
```

圖 4-7、JFreeChart 中 Careless Cleanup 範例

4.2.2 產生曝露 Careless Cleanup 的測試案例

如圖 4-8 所示，藉由劉彥麟論文[19]產生曝露壞味道影響的測試案例並執行。如圖 4-9 所示，測試案例因為程式碼沒有正確處理例外而造成測試失敗。

```
15= @Test
16 public void testReleaseMethodShouldExecuteInEncode() {
17     aspectJSwitch.initResponse();
18     aspectJSwitch.addResponse("write/f(IOException)");
19     aspectJSwitch.addResponse("close/AOPCheckResources");
20     aspectJSwitch.toFirstResponse();
21     try{
22         SunJPEGEncoderAdapter object = new SunJPEGEncoderAdapter();
23         object.encode(null, null);
24         Assert.assertTrue(aspectJSwitch.isResourceCleanup());
25     }catch (Exception e) {
26         Assert.assertTrue(aspectJSwitch.isResourceCleanup());
27     }
28 }
```

圖 4-8、JFreeChart Careless Cleanup 測試案例

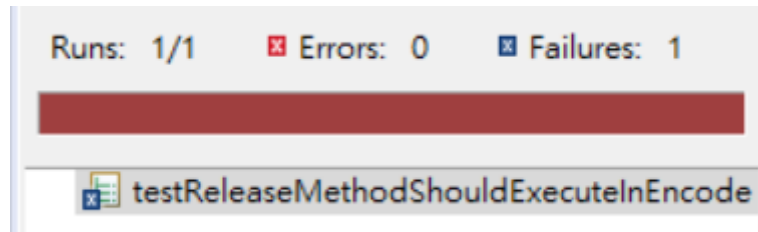


圖 4-9、JFreeChart Careless Cleanup 測試失敗

4.2.3 消除 Careless Cleanup

如圖 4-10 所示，接著透過本論文在 Robusta 提供的自動化快速修復功能來消除壞味道。如圖 4-11 所示，產生 try/finally 來保護程式碼，並且在 finally 裡用 try/catch 將釋放資源的例外保護住，避免產生 Exception Thrown From Finally Block。如圖 4-12 所示，原先失敗的測試案例因為例外被正確處理而成功。

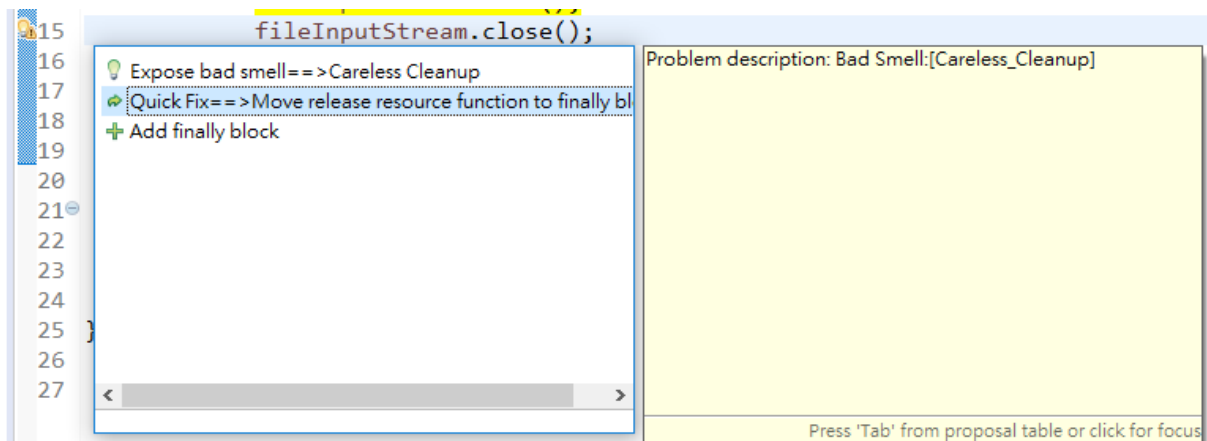


圖 4-10、Robusta 提供自動化消除 Careless Cleanup

```

174= @Override
175 public void encode(BufferedImage bufferedImage, OutputStream outputStream) throws IOException {
176     Iterator iterator = ImageIO.getImageWritersByFormatName("jpeg");
177     ImageWriter writer = (ImageWriter) iterator.next();
178     ImageWriteParam p = writer.getDefaultWriteParam();
179     ImageOutputStream ios = ImageIO.createImageOutputStream(outputStream);
180     try {
181         Args.nullNotPermitted(bufferedImage, "bufferedImage");
182         Args.nullNotPermitted(outputStream, "outputStream");
183         p.setCompressionMode(ImageWriteParam.MODE_EXPLICIT);
184         p.setCompressionQuality(this.quality);
185         writer.setOutput(ios);
186         writer.write(null, new IIOMImage(bufferedImage, null, null), p);
187         ios.flush();
188         writer.dispose();
189     } finally {
190         try {
191             ios.close();
192         } catch (IOException e) {
193             /*
194              * Although it's a Dummy Handler bad smell,
195              * Robusta recommends that you keep this bad smell
196              * instead of choosing Quick Fix or Refactor that we provide.
197              */
198             PropertyConfigurator.configure("log4j.properties");
199             Logger.info(e);
200         }
201     }
202 }

```

圖 4-11、JFreeChart 消除 Careless Cleanup 的結果

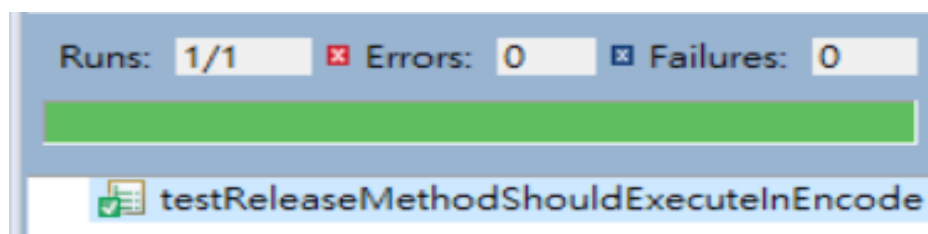


圖 4-12、JFreeChart 中 encode 函式正確處理例外後測試成功

4.3 Exception Thrown From Finally Block 應用實例

4.3.1 偵測 Exception Thrown From Finally Block

如圖 4-13 所示，Robusta 偵測 JFreeChart 後，發現專案中 saveChartAsPNG 的函式含有 Exception Thrown From Finally Block。這段程式碼是將圖片存成 PNG 的格式。當 308 行發生例外時，會將例外向上層回報，接著執行 finally block 裡第 311 行釋放資源的程式碼。如果第 311 行發生例外，將會覆蓋第 308 行原本要回報的例外，改向上層回報第 311 行的例外，因此這是一個 Exception Thrown From Finally Block。

```
304 public static void saveChartAsPNG(File file, JFreeChart chart,
305     int width, int height, ChartRenderingInfo info)
306     throws IOException {
307     Args.nullNotPermitted(file, "file");
308     OutputStream out = new BufferedOutputStream(new FileOutputStream(file));
309     try {
310         ChartUtils.writeChartAsPNG(out, chart, width, height, info);
311     }
312     finally {
313         out.close();
314     }
315 }
```

圖 4-13、JFreeChart 中 Exception Thrown From Finally Block 範例

4.3.2 產生曝露 Exception Thrown From Finally Block 的測試案例

如圖 4-14 所示，藉由劉彥麟論文[19]產生曝露壞味道影響的測試案例並執行。如圖 4-15 所示，測試案例因為程式碼沒有正確處理例外而造成測試失敗。

```

15 @Test
16 public void testDueToCloseFinallyShouldNotThrowAnyExceptionInSaveChartAsPNG() {
17     aspectJSwitch.initResponse();
18     aspectJSwitch.addResponse("writeChartAsPNG/f(CustomRobustaException)");
19     aspectJSwitch.addResponse("close/f(IOException)");
20     aspectJSwitch.toFirstResponse();
21     try{
22         ChartUtils.saveChartAsPNG(null, null, 0, 0, null);
23     } catch (CustomRobustaException e) {
24         e.printStackTrace();
25         Assert.assertEquals("This Exception is thrown from try/catch block, so the bad smell is removed.", e.getMessage());
26     } catch (Exception e) {
27         e.printStackTrace();
28         Assert.fail("Exception is thrown from finally block.");
29     }
30 }

```

圖 4-14、JFreeChart Exception Thrown From Finally Block 測試案例

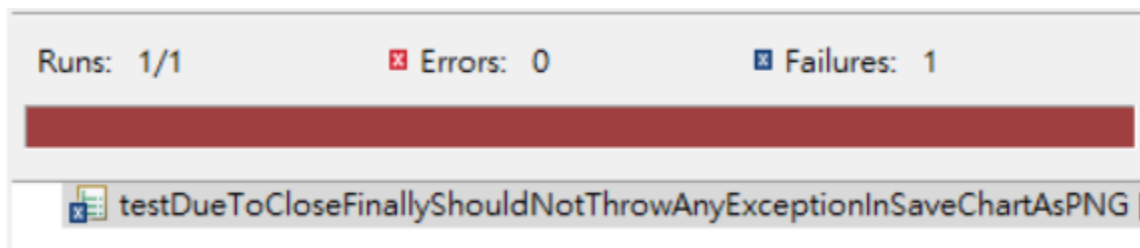


圖 4-15、JFreeChart Exception Thrown From Finally Block 壞味道測試失敗

4.3.3 消除 Exception Thrown From Finally Block

如圖 4-16 所示，接著透過本論文在 Robusta 提供的自動化重構功能來消除壞味道。如圖 4-17 所示，會將原本 finally block 釋放資源的程式碼抽成一個函式，並用 try/catch 保護住，讓 finally block 不再丟出例外。最後，再執行一次測試案例，如圖 4-18 所示，原先失敗的測試案例因為例外被正確處理而成功。

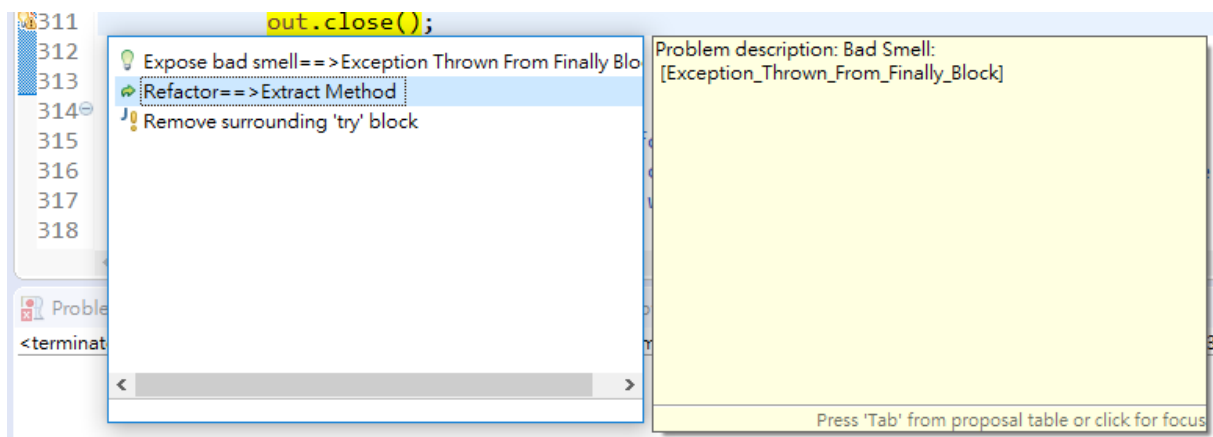


圖 4-16、Robusta 提供自動化消除 Exception Thrown From Finally Block

```

306 public static void saveChartAsPNG(File file, JFreeChart chart,
307     int width, int height, ChartRenderingInfo info)
308     throws IOException {
309     Args.nullNotPermitted(file, "file");
310     OutputStream out = new BufferedOutputStream(new FileOutputStream(file));
311     try {
312         ChartUtils.writeChartAsPNG(out, chart, width, height, info);
313     }
314     finally {
315         Close(out);
316     }
317 }
318
319 private static void Close(OutputStream out) {
320     try {
321         out.close();
322     } catch (IOException e) {
323         /* Although it's a Dummy Handler bad smell,
324          * Robusta recommends that you keep this bad smell
325          * instead of choosing Quick Fix or Refactor that we provide.
326          */
327         PropertyConfigurator.configure("log4j.properties");
328         Logger.info(e.getMessage());
329     }
330 }

```

圖 4-17、JFreeChart 消除 Exception Thrown From Finally Block 的結果

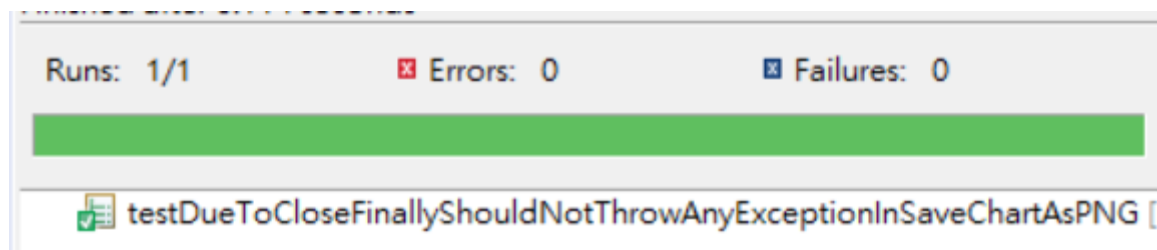


圖 4-18、JFreeChart 中 saveChartAsPNG 函式正確處理例外後測試成功

4.4 Unprotected Main Program 應用實例

如圖 4-19 所示，為 Tomighty 工具正常啟動時的畫面。



圖 4-19、Tomighty 工具

4.4.1 偵測 Unprotected Main Program

如圖 4-20 所示，為 Tomighty 專案的 main program 所在位置，由於 main program 沒有被 try/catch 保護，因此透過 Robusta 的偵測顯示為 Unprotected Main Program。當 main program 發生例外時，程式會發生不預期的終止，而被使用者認為是軟體品質不佳的表現。

```
67 public static void main(String[] args) throws Exception {  
68     UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());  
69     Injector injector = Guice.createInjector(new TomightyModule(), Js250Module());  
70     Tomighty tomighty = injector.getInstance(Tomighty.class);  
71     invokeLater(tomighty);  
72     TrayManager trayManager = injector.getInstance(TrayManager.class);  
73     invokeLater(trayManager);  
74 }
```

圖 4-20、Tomighty 中 Unprotected Main Program 範例

4.4.2 產生曝露 Unprotected Main Program 的測試案例

如圖 4-21 所示，藉由劉彥麟同學的論文[19]產生曝露壞味道影響的測試案例並執行。

如圖 4-22 所示，測試案例會因為程式碼沒有正確處理例外而造成測試失敗。

```

12  @Test
13  public void testDueToSetLookAndFeelMainShouldNotThrowAnyException() {
14      aspectJSwitch.initResponse();
15      aspectJSwitch.addResponse("setLookAndFeel/f(RuntimeException)");
16      aspectJSwitch.toFirstResponse();
17      try{
18          String[] args={};
19          Tomighty.main(args);
20      }catch (Throwable e) {
21          Assert.fail(e.getMessage());
22      }
23  }

```

圖 4-21、Tomighty Unprotected Main Program 測試案例

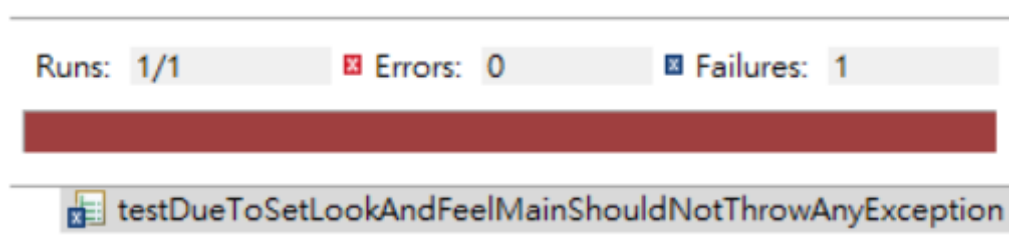


圖 4-22、Tomighty Unprotected Main Program 測試失敗

4.4.3 消除 Unprotected Main Program

如圖 4-23 所示，接著透過本論文在 Robusta 提供的自動化快速修復功能來消除壞味道。如圖 4-24 所示，將 main program 用 try/catch 保護起來，且 catch 捕捉 Throwable 類別，最後將例外訊息寫入日誌檔中。如圖 4-25 所示，原先失敗的測試案例因為例外被正確處理而成功。

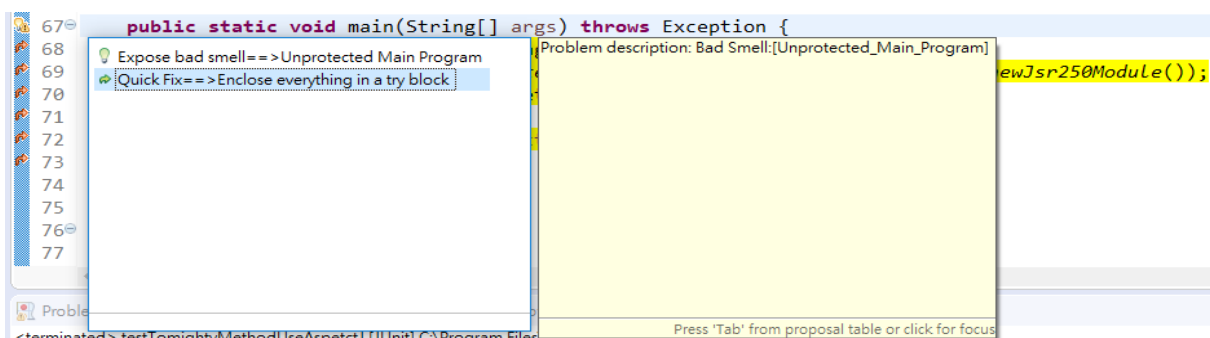


圖 4-23、Robusta 提供自動化消除 Unprotected Main Program

```

69     private static Logger logger = Logger.getLogger(Tomighty.class);
70
71     public static void main(String[] args) throws Exception {
72         try {
73             UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
74             Injector injector = Guice.createInjector(new TomightyModule(), Jsr250.newJsr250Module());
75             Tomighty tomighty = injector.getInstance(Tomighty.class);
76             invokeLater(tomighty);
77             TrayManager trayManager = injector.getInstance(TrayManager.class);
78             invokeLater(trayManager);
79         } catch (Throwable ex) {
80             PropertyConfigurator.configure("log4j.properties");
81             logger.info(ex);
82         }
83     }

```

圖 4-24、Tomighty 消除 Unprotected Main Program 的結果

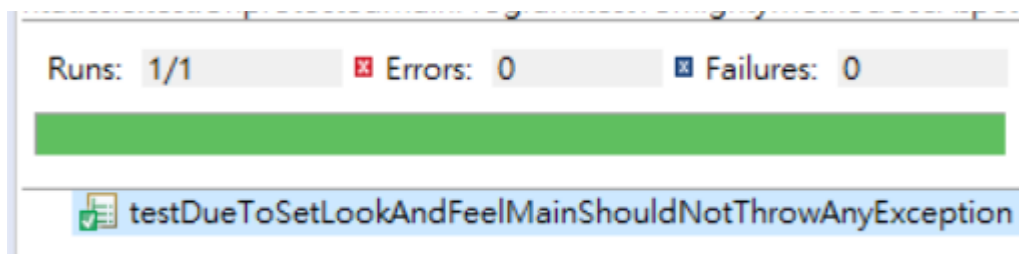


圖 4-25、Tomighty 中 main program 正確處理例外後測試成功

第五章 結論與未來展望

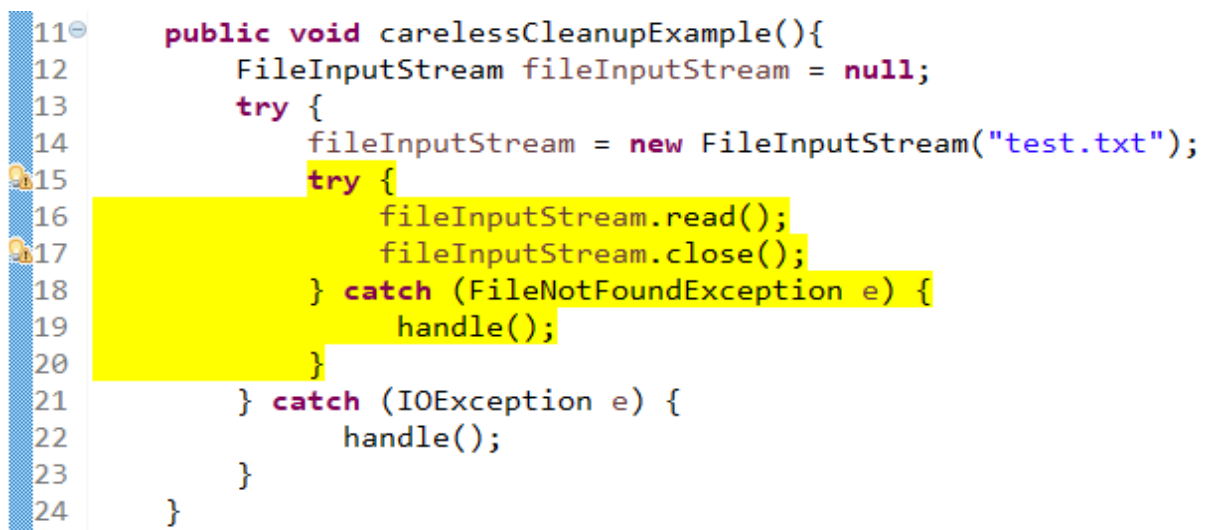
5.1 結論

本論文將 Robusta 原有的壞味道消除方法重新修正及改善，並且將 Careless Cleanup 的快速修復功能實作於 Robusta 中；最後，將 Robusta 應用於開源專案，且成功的消除程式碼中例外處理的壞味道，進而提升專案的系統強健度。

5.2 未來展望

根據壞味道消除的功能，目前有以下幾點功能能夠改善。

第一點：在 Careless Cleanup 的快速修復功能中，目前只提供對一層 Try Statement 來進行快速修復，如果釋放資源的程式碼在超過一層的巢狀結構，如圖 5-1 所示，則無法消除 Careless Cleanup。因此如果能夠讓 Careless Cleanup 能夠在多層巢狀結構下快速修復的話，消除壞味道的功能將會更完善。



```
11 public void carelessCleanupExample(){
12     FileInputStream fileInputStream = null;
13     try {
14         fileInputStream = new FileInputStream("test.txt");
15         try {
16             fileInputStream.read();
17             fileInputStream.close();
18         } catch (FileNotFoundException e) {
19             handle();
20         }
21     } catch (IOException e) {
22         handle();
23     }
24 }
```

圖 5-1、兩層 Try Statement 的 Careless Cleanup 範例

第二點：目前 Careless Cleanup 在快速修復後，會衍生出 Nested Try Statement，如果 Careless Cleanup 能夠提供重構功能，讓使用者能夠填寫要獨立出來的函式名稱，並將衍生的 Nested Try Statement 自動獨立成一個函式，對使用者將會方便很多。

參考文獻

- [1] Chien-Tsun Chen. 例外處理設計的逆襲. 悅知文化，2014
- [2] 陳建村，爪哇例外處理:模型、重構、與樣式，博士論文，國立臺北科技大學機電科技研究所博士班，台北，2008
- [3] 洪哲瑋，例外處理程式壞味道的自動化偵測與重構，碩士論文，國立臺北科技大學資訊工程系碩士班，台北，2009
- [4] Robusta at Eclipse Marketplace, <https://marketplace.eclipse.org/content/robusta-eclipse-plugin> [Accessed 01 June 2018]
- [5] Eclipse, <http://www.eclipse.org/> [Accessed 01 June 2018]
- [6] 陳友倫、以 Aspect 揭露導因於例外處理的程式缺陷，碩士論文，國立臺北科技大學資訊工程系碩士班，台北，2016
- [7] Chien-Tsun Chen, Yu Chin Cheng, Chin-Yun Hsieh and I-Lang Wu, “Exception Handling Refactorings: Directed by Goals and Driven by Bug Fixing,” Journal of Systems and Software, vol.82, no.2, pp.333-345, Feb. 2009.
- [8] ASTNode document,
<https://help.eclipse.org/neon/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2FASTVisitor.html> [Accessed 01 June 2018]
- [9] Eclipse JDT, <https://www.programcreek.com/2011/01/best-java-development-tooling-jdt-and-astparser-tutorials/> [Accessed 01 June 2018]
- [10] AST View at Eclipse Marketplace, <https://marketplace.eclipse.org/content/ast-view> [Accessed 01 June 2018]
- [11] ASTParser document,
<https://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2FASTParser.html> [Accessed 01 June 2018]

2018]

[12] ASTRewrite document,

<https://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.jdt.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fjdt%2Fcore%2Fdom%2Frewrite%2FASTRewrite.html>

[Accessed 01 June 2018]

[13] 楊智傑，Robusta- 一個對於 Java 例外處理壞味道的偵測工具，碩士論文，國立臺北科技大學資訊工程系碩士班，台北，2013.

[14] Exception document, <https://docs.oracle.com/javase/7/docs/api/java/lang/Exception.html>

[Accessed 01 June 2018]

[15] Error document, <https://docs.oracle.com/javase/7/docs/api/java/lang/Error.html> [Accessed 01 June 2018]

[16] Throwable document, <https://docs.oracle.com/javase/7/docs/api/java/lang/Throwable.html>

[Accessed 01 June 2018]

[17] Eclipse IMarker document,

<http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Freference%2Fapi%2Forg%2Feclipse%2Fcore%2Fresources%2FIMarker.html> [Accessed 01 June

2018]

[18] 廖振傑，透過偵測及移除例外處理壞味道提升軟體強健度:以 ezScrum 為例，碩士論文，國立臺北科技大學資訊工程系碩士班，台北，2016

[19] 劉彥麟 利用 AspectJ 搭配測試案例曝露例外處理壞味道的影響，碩士論文，國立臺北科技大學資訊工程系碩士班，台北，2018