

# 軟體品質模型與涵蓋度測試

軟體品質模型與涵蓋度測試

薛念林, 逢甲大學

Wednesday 20<sup>th</sup> July, 2016



# 目錄

<b>1 軟體品質</b>	<b>11</b>
1.1 軟體危機 . . . . .	12
1.2 軟體品質模型 . . . . .	17
1.3 ISO 9126 品質模型 . . . . .	23
1.4 練習 . . . . .	39

# 目錄

---

<b>2 涵蓋度測試</b>	<b>41</b>
2.1 規格、設計、測試 . . . . .	41
2.2 JUnit . . . . .	47
2.3 測試涵蓋度 . . . . .	55
2.4 Eclemma . . . . .	68
2.5 練習 . . . . .	72

## 版本聲明

本投影片僅能用於非營利之教學用途之上。內容有任何錯誤或建議，請利用軟體工程聯盟網站 (<http://www.sec.org.tw>)，聯絡課程內容編輯團隊。

軟體工程聯盟軟體測試課程編輯團隊

- 臺灣大學資訊工程學系李允中博士
- 臺北科技大學資訊工程系郭忠義博士
- 臺北科技大學資訊工程系劉建宏博士
- 逢甲大學資訊工程學系薛念林博士
- 海洋大學資訊工程學系馬尚彬博士
- 臺中教育大學資訊工程學系徐國勛博士
- 高雄師範大學軟體工程學系李文廷博士

## 軟體測試基礎訓練課程 (April, 2016)

1. 軟體工程簡介、軟體測試概念簡介、軟體測試技術(白箱)
2. 軟體測試技術(黑箱)、單元測試與 JUnit
3. TDD+JUnit、軟體測試工具(Selenium or Jmeter)

## 軟體測試進階訓練課程 (July, 2016)

1. 軟體品質模型與涵蓋度測試 (JUnit, Eclemma)
2. 建構管理 (Jenkins, Git, Maven)
3. Issue tracking (Redmine)
4. 壓力測試 (JMeter)
5. 網頁測試 (Selenium)

## Get the slide

<https://goo.gl/lb6ov9>

## 課前問卷

<http://goo.gl/forms/Ipxi3LDcpMnLJBtg2>

# 目錄

---

# Lecture 1

## 軟體品質

“

大家都知道物質不滅定律；我們更熟悉 Bug 不滅定律。

## 1.1 軟體危機

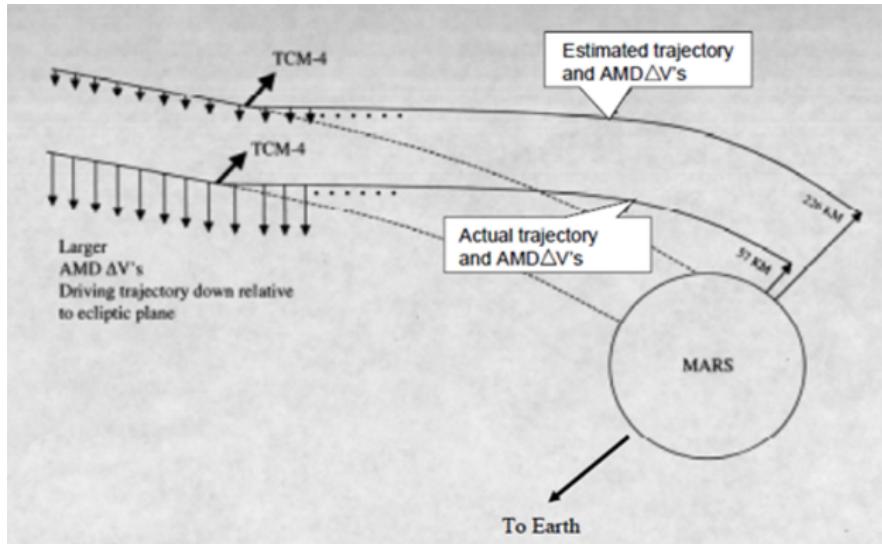
國際著名 Standish Group 研究機構 1995 年以美國境內 8000 個軟體專案為調查樣本，有 84% 的軟體計劃無法於既定的時間及經費當中完成，超過 30 % 的計畫執行到中途被取消，專案的預算平均超出 189%。

- 愛國者反導彈事件，毫秒的誤差
- NASA 火星氣候軌道探測器
- 華航名古屋空難

愛國者反導彈事件，毫秒的誤差 在 1991 年 2 月的第一次海灣戰爭中，一枚伊拉克發射的飛毛腿導彈準確擊中美國在沙地阿拉伯的達蘭基地 (Dhahran)，當場炸死 28 個美國士兵，炸傷 100 多人，造成美軍波斯灣戰爭中唯一一次傷亡超過百人的損失。

由於一個簡單的電腦 bug，使基地的愛國者反導彈系統失效，未能在空中攔截飛毛腿導彈。

當時，負責防衛該基地的愛國者反導彈系統已經連續工作了 100 個小時，每工作一個小時，系統內的時鐘會有一個微小的毫秒級延遲，這就是這個失效悲劇的根源。愛國者反導彈系統的時鐘暫存器設計為 24 位元，因而時間的精度也只限於 24 位元的精度。在長時間的工作後，這個微小的精度誤差被漸漸放大。在工作了 100 小時後，系統時間的延遲是三分之一秒。



NASA 火星氣候軌道探測器 1998 年，美國太空總署（NASA）發射了火星氣候軌道探測器（Mars Climate Orbiter），結果到了火星上空，便音訊全無。這個失敗導因於一個低級錯誤：兩個研究團隊使用的度量單位不同，一個用英制，一個用公制：原本應該從距地面 140 公里高度穿過火星大氣層，最後卻低於 60 公里，導致探測器經不起劇烈的大氣摩擦而焚燬。



華航名古屋空難 1994 年 4 月 26 日，華航編號為 B-1816 的空中巴士 A300-622R 型客機由桃園國際機場飛往名古屋的班機，在名古屋機場降落時不幸墜毀，造成 264 人死亡。

「副駕駛在操縱飛機降落時，不小心誤將飛機設定在「重飛」，而因駕駛員一直不知飛機設定在「重飛」的自動操作狀態下，駕駛員努力用手動操作，想要將機首壓低，而因電腦在「重飛」爬升自動操作狀態，電腦將機

尾的水平安定面設定到機首上升的狀態「糾正」駕駛員「錯」的壓低機首的手動操作，結果在電腦與駕駛員操作機首角度的爭鬥中，飛機向上衝的攻角過大而失去平衡，最後失速墜毀。

## 1.2 軟體品質模型

### 1.2.1 品質觀點

- 超自然觀點 Transcendental view。無法直接定義品質，但看（感受）到後就知道好壞。abstract but can be recognized if it is present.
- 使用者觀點 User view。符合使用者需求的程度。fitness for purpose or meeting user's needs.
- 製造觀點 Manufacturing view。符合流程的標準，例如通過 ISO 的認證。conformance to process standard.
- 產品觀點 Product view。產品本身的材質，例如樟木所做的桌子。inherent characteristics in the product itself.
- 價值觀點 Valued based view。顧客是否願意掏錢出來買。customers' willingness to pay for a software.

## 1.2.2 軟體品質

一個系統，元件或流程滿足所指定的需求的程度。The degree to which a system, component, or process meets specified requirements. (Crosby, 1979)

一個系統，元件或流程滿足顧客或使用者的需求或期望的程度。The degree to which a system, components, or process meets customer or user needs or expects (Juran, 1998)



### Software Quality

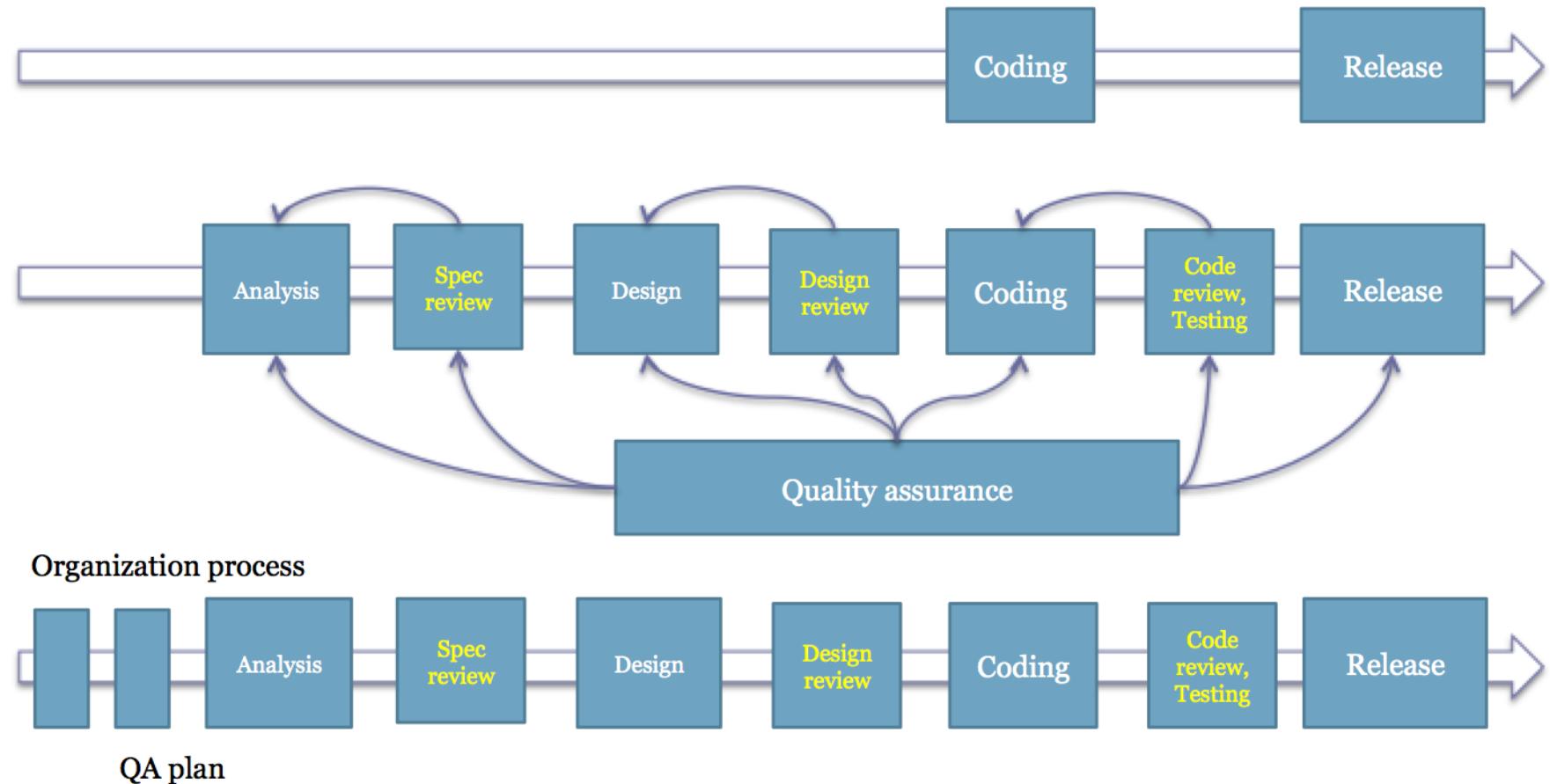
符合明訂的功能與效能需求，明訂的開發標準，及非明定的專業軟體特性的程度。

*Conformance to explicitly stated functional and performance requirements, explicitly documented development standards, and implicit characteristics that are expected of all professionally developed software (Pressman)*



### 製造業的品質控管

1. 自動化。製造過程中使用機器 - 自動化，因為機器不容易犯錯且可不斷重複單調無聊工作。
2. 簡單化。需要人工的部份，讓每個生產線員工只做一件簡單到不容易出錯的工作。



軟體品質確保之活動



不同物品的品質特性各有不同

## 1.3 ISO 9126 品質模型

### 1.3.1 功能性 Functionality

- 功能正確性 **Accurateness** 功能的運作是否正確，例如計算帳戶的餘款是否正確。
- 規格合適性 **Suitability** 功能特性是否符合軟體的規格或該系統的特性。例如 Evernote 在 2015 加入了一個 chat 的功能，一個筆記軟體是否需要一個談天功能？
- 相互運作性 **Interoperability** 和其他元件或系統的互動是否正確。這個特性與系統的「可整合性」有很大的關係。目前許多的系統都提供 API (Application Interface) 讓該系統可以容易的和其他的系統或模組整合，其 interoperability 就比較好。這個特性也有助於可維護性，如果是在維護階段，因為需求的變更需要修改時，我們透過 API 來做局

部的修改，而不是改變內部的程式碼，其變動性就會比較低。

- 規範符合性 **Compliance** 是否符合特定業界標準與規範或法律。例如我們產生的格式是否符合 JSON 格式、是否符合 SCORM 標準等。
- 安全性 **Security** 是否能夠阻擋非法的存取或控制。



我們有時間做多餘的功能，卻沒有時間把必要的功能做對。



ISO 9126 軟體品質模型

### 1.3.2 可靠性 Reliability

- 成熟度 **Maturity** 失效的次數越少成熟度越高。可靠度的評量通常可用 MTTF (mean time to failure) 來計算，就是平均多久失效一次。
- 容錯度 **Fault tolerance** 當環境或其他元件出錯時，能夠持續保持一定的運行的能力，能容忍錯誤的能力。
- 回復性 **Recoverability** 當環境或其他元件出錯時，能夠回復到正常運行的能力。例如有些系統要一天後才能回復，有些系統指需要停機一小時。有一些系統一個星期才備份一次，所以系統回復時可能喪失一個星期的資料，就是回復性差。

### 1.3.3 可用性 Usability

ATM 自動提款機上出現的提款金額：\$1000.00, \$2000.00 \$3000.00。

- 易了解性 **Understandability** 系統的功能或概念是否容易了解？是否符合使用者所認知的心理模型？設計上如果可以有一些擬真的設計可以提昇意了解性。例如電子書的設計再翻頁時做出 3D 的翻頁效果，就可以幫助亦了解性。
- 易學習性 **Learnability** 需要花多少力氣來學習？最好的設計是不需要說明書，使用者一看就知道如何使用。但許多新架構新功能是不太可能不透過學習的，但必須要容易學習，只教一次就會使用。例如 iPhone 在 kill App 時需要長按該 App, 等待刪除符號出現後在刪除即可。第一次需要人教，但只要一次之後就不會忘記，就是一種好的設計。
- 易操作性 **Operability** 在指定環境下操作是否容易？

可用性如何檢驗？我們可以透過學習的時間數或是產能來做量化的計算。例如你設計一個需要資管人員輸入的收費單系統，如果有經驗的資管人員一小時只能打 10 個停車單，表示系統的設計有問題。如果設計上有很多快速鍵，移動也很方便，一個資管人員一小時可以打上 30-40 個停車單。

### 1.3.4 效能 Efficiency

主要著眼在系統提供功能時系統資源被使用的狀況：磁碟空間，記憶體空間，網路用量等。

- 時間效能 **Time behavior** 回應時間的長短。或是單位時間能夠處理的量。
- 資源效能 **Resource behavior** 消耗資源的多寡。例如 memory, cpu, disk and network usage。一般而言，時間效能越好資源效能越差，反之亦然。例如一個系統需要上傳照片，但解析度的要求並不高，現代多半用收機拍照後上傳，檔案可能上到 5M，當量大的時候就會造成很大的負擔。如果我們能夠在系統上傳之時做壓縮或是轉換其解析度，這系統吃的資源就不會那麼重了。



時間效能和資源效能常常會相互衝突，設計需要取捨 – 這也是

為什麼品質沒有絕對，需要與使用者或設計師商討。

### 1.3.5 可維護性 Maintainability

- 可分析性 **Analyzability** 是否容易找出錯誤的原因。系統運行的時候如果能夠留下 log，則有助於其可分析性。
- 可變動性 **Changeability** 要花多少力氣來改變系統？例如新增一個功能需要花幾個人月（man-power, man-month）。
- 穩定性 **Stability** 對系統變動的敏感度，系統變動時對其他部分所造成的負面衝擊。例如一個人事的系統模組增加的了一個人員型態，居然導致財務結算的系統錯誤，其系統的敏感度太高，可能是當初設計的時候模組沒有切好，彼此的相依性太高所致。
- 可測試性 **Testability** 當系統變動時需要花多少 effort 來做確認測試？系統的可測試性高不高？測試環境是否容易建立？虛擬模組容不容易建置？是否有相關的測試資料？

“

傻瓜都能寫出電腦能理解的程式。優秀的工程師寫出的是人類能讀懂的程式。

### 1.3.6 可移植性 Portability

可移植性考慮到系統能夠適應到新的環境的能力。如果一個系統功能正確，可靠性高，執行很有效率，也很容易維護，但卻只能在某一特定機型、特定作業系統、特定組織環境下運作，當我們想把它移植到新的環境時就會出錯，那麼他也不是一個好系統。

- 適應性 **Adaptability** 對於新的作業系統，作業環境或新規格的適應性。
- 易安裝性 **Installability** 花多少力氣安裝系統？
- 相容性 **Conformance** 和功能性裡面的 compliance 類似，但這裡強調的是相容性，例如與某資料庫的相容性。
- 易置換性 **Replaceability** 容易抽換某元件的能力。我們在 Blackboard 系統上開發了許多與學習相關的功能（例如點名），但這些系統是用 Blackboard 的 Building Block 框架來做的，是一個特有的框架。當我

們移植到 Moodle 的系統後這些功能就不能用了。許多後期開發的模組是一般的 web 程式（建構在 .Net 上），它就很容易的與新的 Moodle 整合。

Eclipse 這個開發工具平台採取的架構就是可以擴充的 plug in 架構，所以許多第三方的開發者可以自己開發許多套件來整合。EclEmma，一種測試包含度的外掛，就是其中一個例子。



雄太曾經為某研究機構開發針對專案人員的出勤刷卡系統，透過刷卡來確定其上下班是否正常，整個流程包含多個系統：

- 行政人員在 A 系統（以 client-server 架構開發）所做的中輸入參與人員的資料，包含她所參與的計畫；
- 專案主持人登入 B 系統（web 架構）做一個人員確認；
- 行政人於把專案人員資料匯入個刷卡機的主機系統
- 專案參與人員每日做出勤刷卡的動作
- 每月結算人員的出勤狀況，如果符合要求，就會把資料彙整到撥款系統。

大虎公司覺得這一套系統比他們現行的卡式打卡鐘系統好多了，希望能夠以低價的方式移植，是否可行？需要考慮什麼？

### 1.3.7 ISO 9126 的度量

“除了上帝，我只相信數字。*In God we trust. All others must bring data.* (W. Edwards Deming)

ISO 9126 除了定義有哪些品質項目以外，它還定義了這些品質項目的量化檢驗方法（metric）。有三個：

- 內部度量 internal metric。不需透過程式的執行，直接檢驗程式碼本身。例如模組的耦合力、註解的撰寫狀況等都屬於。
- 外部度量 external metric。需要透過程式的執行才能度量。例如消耗多少記憶體，平均的回應時間等。
- 使用品質度量 quality in use metric。使用上的度量，通常與 usability 相關。

範例：針對某大學的課程管理與學習系統，我們試著描述一下其品質：

1. Functionality: (1) 完整的學習管理系統; (2) 和校務的資料（學生選課，老師授課的資料）可以整合; (3) 可以和 LDAP 認證系統整合。
2. Reliability: (1) 舊的學習系統平均每年的 downtime 是 3 days; 新系統是 5hours (2) 應用 VM 提供備援機制。(3) 透過 SAN ( Storage Area Network ) 架構，提昇備份與備援機制。
3. Usability: (1) 移除不需要的訊息宣傳，畫面更乾淨俐落 (2) 建立「使用說明」的課程來提供操作說明與回答問題。
4. Efficiency: (1) 透過分散式系統架構來提升效能。
5. Maintainability- (1) Open source 提昇系統的可除錯性 (2) 避免修改 Moodle PHP 的程式碼，降低修改的敏感度，提昇系統的穩定度 (3) 所有的程式行為都有詳細的 log 。

6. Portability: (1) 所有校務功能都獨立到 Apps 的獨立模組，透過 SSO 做整合，可取代性高。但也因此介面會比較醜 (2) Moodle 可安裝至 Window, Unix 系列的作業系統 (3) 各 AP 採用無硬碟系統，系統容易安裝。

## 1.4 練習

<http://goo.gl/forms/k9Wcz9L6EesaE5Hs1>

1. ● 以下問題或作法分別與哪個 ISO 9126 特性（及次因子）相關？
  - (a) 需求規格變了，軟體就需要大幅的修改，造成很大的人力損失。
  - (b) 經過了教育訓練，使用者還是不太會操作該軟體。
  - (c) 所開發的系統可以透過 LDAP<sup>1</sup>和別人的帳號認證系統整合。
  - (d) 所開發的系統提供 API 方便其他系統呼叫。
  - (e) 開發系統時做適當的 logging，記錄適當的系統資訊。
  - (f) 每個畫面提供 help 的按鈕，提供操作的介紹。
  - (g) 預防錯誤資料所成的系統崩潰。
  - (h) 系統每天晚上都重新開機才會正常。

---

<sup>1</sup>一種目錄服務標準

2. 想想看，你使用的軟體，哪些是你認為「品質好」的軟體？請以 ISO 9126 的品質模型 來描述之。
3. ISO 9126 的品質度量有三種：內部、外部、使用。以下分別是屬於何種？
  - (a) 模組耦合率 0.38
  - (b) 平均回應時間 2.38 秒
  - (c) 顧客滿意度 4.28
  - (d) 程式碼註解率 12%

# Lecture 2

## 涵蓋度測試

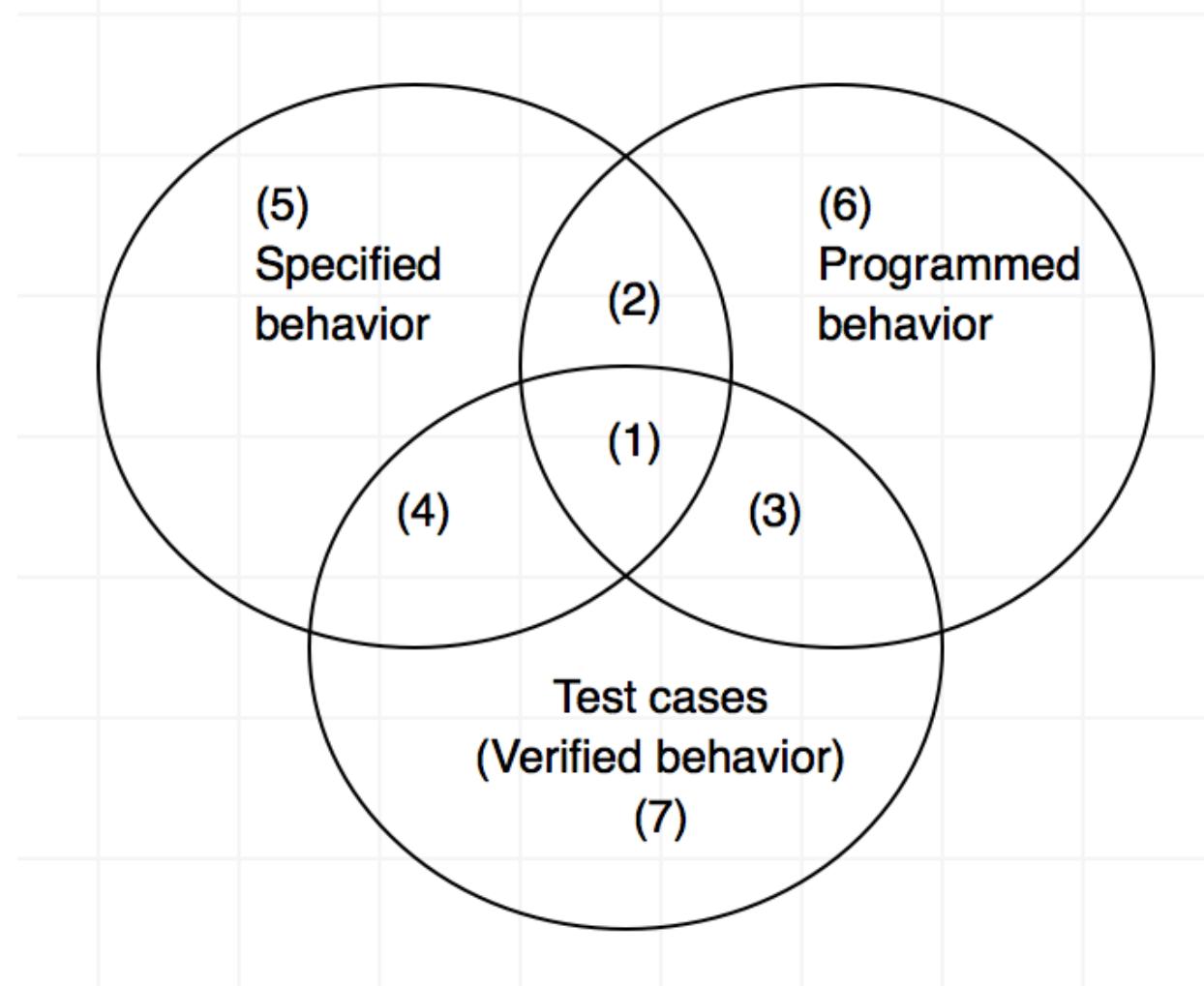
### 2.1 規格、設計、測試

“

你想的，我寫的，他測的，常常搭不在一塊。

測試案例的完整性，和需求規格，程式的行為的關聯性。

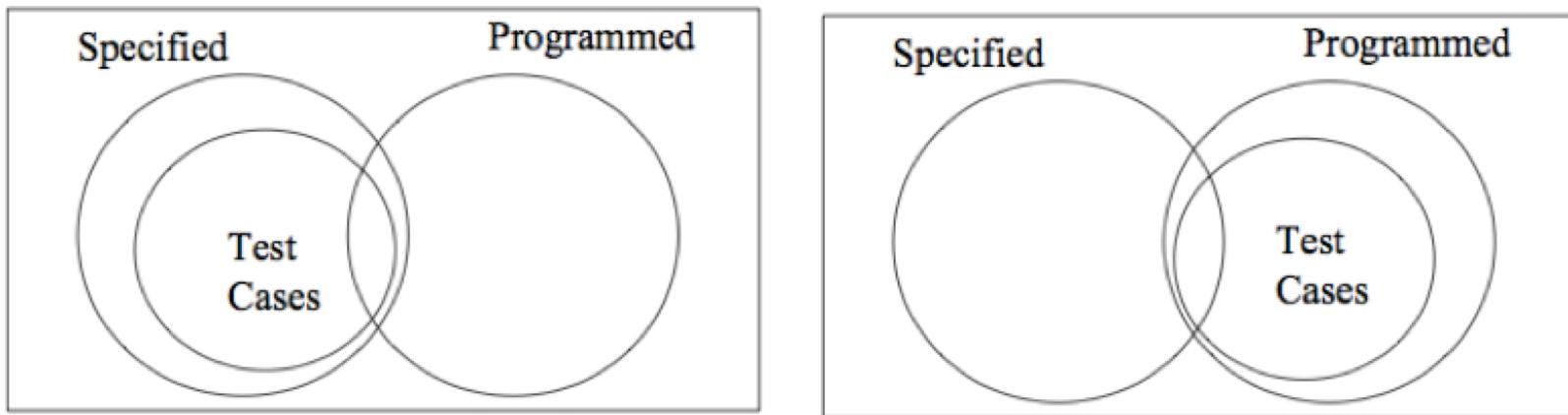
- 規劃的行為 (specified behavior)：規格書裡面所規劃的需求或期待系統所表現出的行為。
- 程序化的行為 (programmed behavior)：最後真的被寫出來的系統，所表現出來的行為。理想上當然規劃的行為應該和程序化的行為應該是相等的，但實際上兩者常常會有所不同，甚至差距很大，
- 驗證的行為 (verified behavior)：系統在測試時所表現出的行為。你的測試可能無效的，沒有測到規格書的內容或是程式的行為。當然，我們期待驗證的行為能和前面兩個能夠重疊，這樣最好。



測試案例與規格，程式行為的關係

## LECTURE 2. 涵蓋度測試

---



功能測試與結構測試

### 2.1.1 測試設計與測試資料

#### 測試設計

- 1 分母 = 0
- 2 分母 != 0
- 3 整除
- 4 不整除
- 5 進位
- 6 不進位

#### 測試資料

- 1 分母 = 0 , (5,0) => 錯誤
- 2 分母 != 0
- 3 整除 , (4,2) => 2
- 4 不整除
- 5 進位 , (5.1, 3) => 2
- 6 不進位 , (4,3) => 1.3



設計測試案例時，預期結果是很重要的，可能是一個數值、字串、資料的更新、畫面的呈現、現象。不論是什麼，都該具體而清楚的描述。

練習：設計「判斷是何種三角形」的程式的「測試案例」與「測試資料」。輸入是三邊長，型態為 double, 輸出是字串。例如輸入 1,1,1 輸出為正三角形。

## 2.2 JUnit

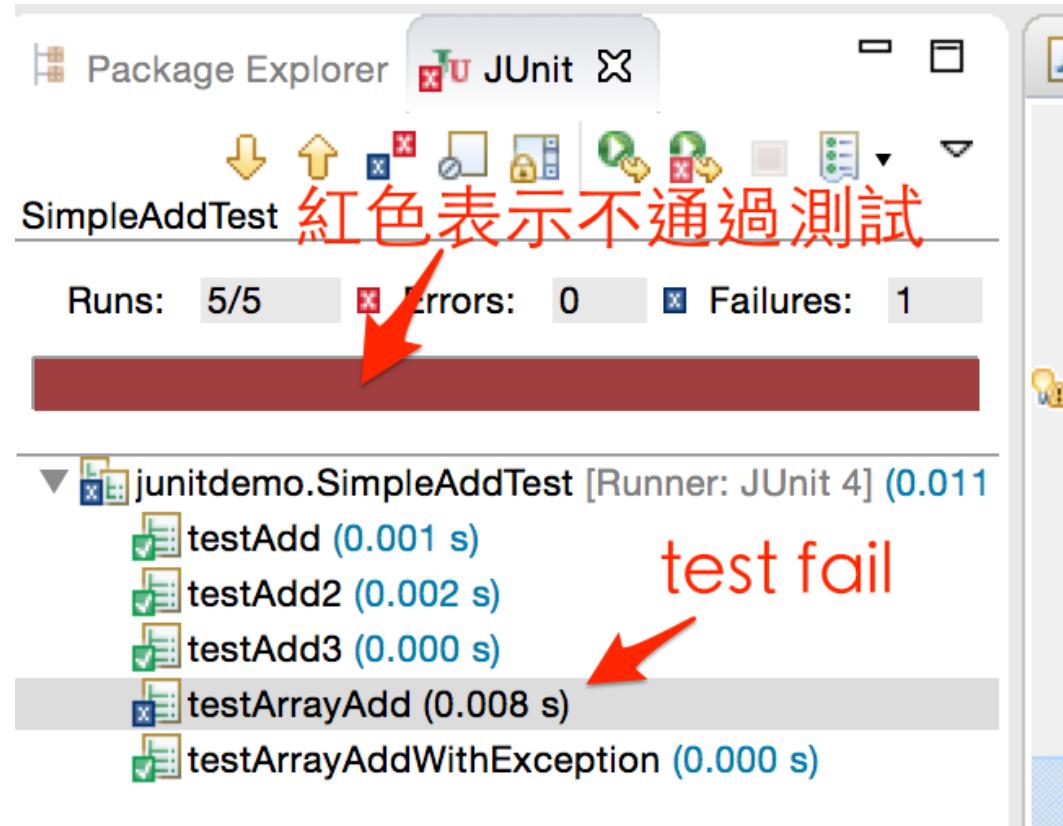
### 測試先行 Test First Development

以前你是：設計 => 寫程式 => 測試。測試先行：設計 => 寫測試碼 => 寫程式 => 測試

1. 設計程式框架
2. 設計測試案例：利用 JUnit 設計測試案例。
3. 執行測試碼：執行 JUnit 測試碼，會跑出很多的「錯誤」。
4. 撰寫程式
5. 執行測試碼：執行 JUnit 測試碼，錯誤應該會逐漸減少。

記得要 **import org.junit.Assert.\***。在 Eclipse 時使用時很方便，在你要測試的類別中按右鍵點選 New » JUnit class 就可以新增 junit class。也可以

## LECTURE 2. 涵蓋度測試



直接新增 junit class 時選擇所要測試的類別及方法。更多的 **JUnit API**。

**@Test** 在測試案例（測試程式碼）前面加上 **@Test** 的標記，這樣 JUnit 就會認得這是一個測試程式碼。例如下例中的 `addition()` 加上 **@Test** 後表示該方法為 `addition` 的測試碼。這和 jUnit 3 的作法有所不同，jUnit 3 必須命名為 `additionTest()`。

```
1  @Test  
2  public void addition() {  
3      assertEquals(12, simpleMath.add(7, 5));  
4  }  
5  @Test  
6  public void subtraction() {  
7      assertEquals(9, simpleMath.subtract(12, 3));  
8  }
```

[Get the code- SimpleAdd] [Get the code- SimpleAddTest]

**@Before and @After** 用來分別表示方法的的設定（setup）與回復（tear down）。我們在測試進行之前，可能需要進行環境的準備與設定，@Before 就是用來指定該方法的。同樣的，當測試完成後若有需要回復成原始狀態或特定狀態也可以寫在 @After 中。

```
1  @Before  
2  public void runBeforeEveryTest() {  
3      // 在測試執行前將物件生成出來  
4      simpleMath = new SimpleMath();  
5  }  
6  @After  
7  public void runAfterEveryTest() {  
8      // 測試執行後把物件清掉  
9      simpleMath = null;  
10 }
```

[Get the code- JUnitDemo01]

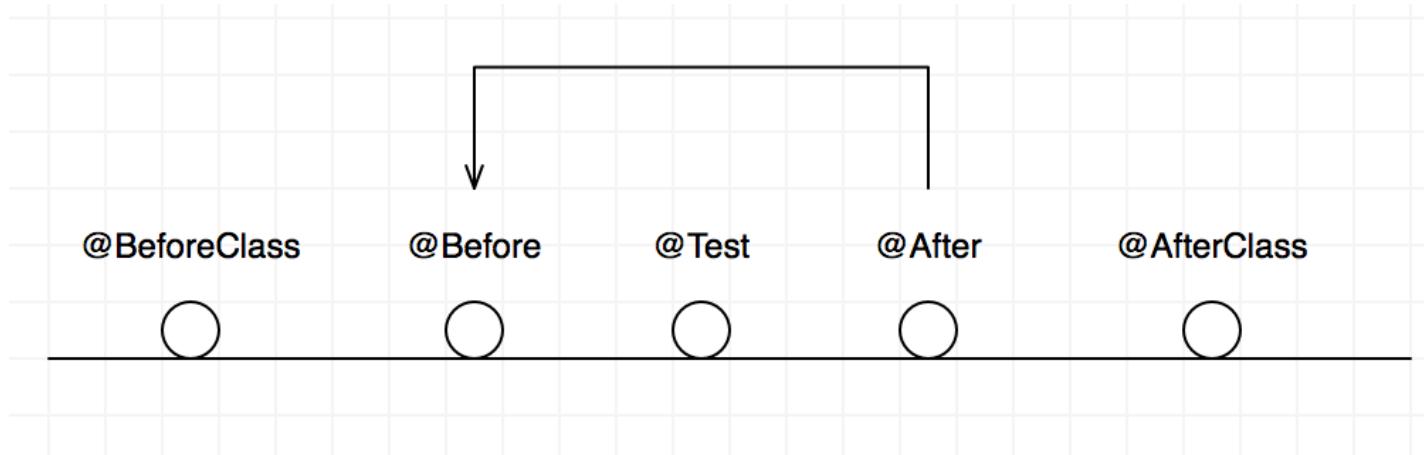


圖 2.1: jUnit 中 Before 與 After 的用法示意圖

**Exception Handling** 程式在某些狀況會拋出例外並停止程式的執行。我們希望測試例外是否會如預期的拋出，就可以在 @Test 之後加上 expected = xx.class 的參數，其中 xx 為拋出的例外類別名稱。

```
1 @Test(expected = ArithmeticException.class)  
2 public void divisionWithException() {  
3     // divide by zero, 會拋出例外  
4     simpleMath.divide(1, 0);
```

5 }

**@Ignore** 可以加上 @Ignore 讓該測試碼暫時失效。後面參數用來說明忽略此測試案例的原因。

```
1 @Ignore("Not_Ready_to_Run")
2 @Test
3 public void multiplication() {
4     assertEquals(15, simpleMath.multiply(3, 5));
5 }
```

**Timeout** 有時候程式的錯誤造成無窮迴圈或跑太久，如果測試碼一直等這樣的錯誤的程式會降低開發效率。我們加上一個 timeout 的參數，後面的時間的單位是毫秒（千分之一秒）

```
1 @Test(timeout = 1000)
2 public void infinity() {
```

```
3     while ( true )  
4         ;  
5 }
```

**Array assertEquals** 後面的兩個參數可以是一個陣列物件，用來確定他們的內容是不是都是相同的。

```
1 @Test  
2 public void listEquality() {  
3     List expected = new ArrayList();  
4     expected.add(5);  
5     List actual = new ArrayList();  
6     actual.add(5);  
7     assertEquals(expected, actual);  
8 }
```

### JUnit 常用功能

static void	assertEquals(double expected, double actual) : actual 的值應該與預期的 expected 值一樣。
static void	assertEquals(double expected, double actual, double delta) : 同上，但允許有 delta 的誤差值。
static void	assertTrue(boolean condition) : condition 的條件為真。
static void	assertArrayEquals(int[] expected, int[] actual) : 兩陣列的值相同。
static void	assertNotNull(Object obj) : obj 不應為 null
static void	assertNull(Object obj) : obj 應為 null
static void	fail() : 表示測試案例不同過，通常用來未測試。

## 2.3 測試涵蓋度

```
1 input A,B,X  
2 if (A>1) and (B=0) then  
3   Y=A  
4 if (A=2) or (X>1) then  
5   Y=X  
6 print Y
```

### 2.3.1 敘述涵蓋度

設計測試案例，使每一條指令敘述至少執行一次。輸入為  $(A,B,X) = (2,0,3)$  就可覆蓋所有可執行指令，結果會得到 3。即便所有的敘述都被執行了，此方法並不是很「嚴謹」的方法：如果我們犯了以下的錯誤：

- 行號 2 的 AND 改成 OR，或是
- 行號 3  $Y=A$  改成其他的敘述
- 行號 4 的  $X>1$  改成  $X>0$ ，或是

測試案例  $(2,0,3)$  並不能找出你犯的錯誤，這表示這組測試案例太弱了。事實上，敘述覆蓋準則是這幾個方法中最弱的一個，白箱測試至少要做到此測試。

### 2.3.2 分支涵蓋度

分支涵蓋度（branch coverage）又稱為決策涵蓋度（decision coverage）。其目標是設計測試案例使程式每個判斷取真分支和取假分支至少執行一次。上述的例子有兩個 branch:

- $(A>1) \text{ AND } (B=0)$
- $(A=2) \text{ OR } (X>1)$

測試涵蓋表

	$(A>1) \text{ AND } (B=0)$		$(A=2) \text{ OR } (X>1)$	
$(A,B,X)$	True	False	True	False
(3,0,3)	V		V	
(3,1,1)		V		V

分支涵蓋度所找出來的測試案例會比敘述涵蓋度的多，也因此比較能找出程式的錯誤。但，很顯然的，也不可找出所有的錯誤。例如若我們將  $(A>1)$  寫成  $(A>2)$ ，涵蓋度一樣 100%，執行結果也一樣，我們就無法知道程式寫錯了。

### 2.3.3 條件涵蓋度

使程式中每個判斷的每個條件至少執行一次。前述程式具有下列四條件  $A > 1$ 、 $B = 0$ 、 $A = 2$ 、 $X > 1$ 。欲使這四個條件都能產生真與假的值，測試案例須包含以下八種案例：

- (1)  $A > 1$ , (2)  $A \leq 1$
- (3)  $B = 0$ , (4)  $B \neq 0$
- (5)  $A = 2$ , (6)  $A \neq 2$
- (7)  $X > 1$ , (8)  $X \leq 1$

測試資料  $(2,0,3)$  滿足 (1)、(3)、(5)、(7)。 $(1,1,1)$  滿足 (2)、(4)、(6)、(8)。因此，測試案例  $(2,0,3), (1,1,1)$  便可達成這個目標。

滿足條件涵蓋度的測試資料，一定滿足分支涵蓋度嗎？看起來很像是這麼一回事，因為前者把條件從分支中分離出來做詳細的檢查了阿！但並非如此：

## LECTURE 2. 涵蓋度測試

---

	p	q	p & q
t1	True	False	False
t2	False	True	False

上表中的測試案例 t1, t2 雖然可以是條件 p, q, 都有 true, false, 但 p&q 這個分支的值卻都是 false 的。

### 2.3.4 分支與條件涵蓋度

如上節所言，條件覆蓋嚴密性通常比決策覆蓋高，但非絕對。例如(1,0,3)和(2,1,1)這組數據，雖然使上述四個條件均產生真與假的值，但並未使運算式 $(A>1) \text{ AND } (B=0)$ 和 $(A=2) \text{ OR } (X>1)$ 具有真與假的值。因此，『分支與條件涵蓋度』設計足夠的測試案例，使判斷中每個條件的所有可能值至少執行一次，同時每個判斷的所有可能判斷結果至少執行一次。上例中，(2,0,3)和(1,1,1)可達成此目標。



順著程式的邏輯來做測試無法檢驗功能，只能檢驗涵蓋度。

針對以下程式(1)設計一測試案例達到百分百敘述涵蓋度(SC100);(2)設計一測試案例達到百分百條件涵蓋度(CC100)，但非百分百分支涵蓋度；(3)設計一測試案例達到百分百分支涵蓋度(BC100)，但非百分百條件涵蓋度(!CC100)。請畫出測試涵蓋表來檢驗。

```
1  read(X, Y)
2  if ((X>10) && (Y==1))
3      X=1;
4  else if ((X-Y) < 2)
5      X=2;
6  if (Y >10)
7      X=3;
8  print X
```

要完成敘述涵蓋(SC100)是容易的，只要能夠讓 03, 05, 07 的敘述進入即可，因此我們設計(X=11, Y=1) 及 (X=12, Y=11) 兩筆資料即可達到此涵

蓋。

要完成 CC100 却不 BC100，表示在複合條件上需要做處理，也就是 02 行的條件判斷。假設所有的條件分別是  $c_1, c_2, c_3, c_4$ 。我們讓這五個條件的 True False 都經歷過，但  $b_1 (c_1 \& \& c_2)$  只有經歷 True, False 其中一個，唯一可行的是  $b_1$  只經過 False。因此測試案例如下：

X	Y	$c_1$	$c_2$	$b_1$	$c_3$	$c_4$
9	1	F	T	F	F	F
12	11	T	F	F	T	T

要完成 BC 却不 CC，表示  $b_1$  必須經歷 T/F，但  $c_1$  或  $c_2$  其中之一只經歷 T/F 其中一個。因此測試案例如下。其中  $c_1$  只經歷 T，而  $b_1$  經歷 T/F。

## LECTURE 2. 涵蓋度測試

---

X	Y	c1	c2	b1	c3	c4
13	1	T	T	T	F	F
12	11	T	F	F	T	T

### 2.3.5 多重條件組合涵蓋度

多重條件組合涵蓋（multiple-condition combination coverage）的方法是從決策/條件涵蓋方法延伸，目標是使測試資料涵蓋每個布林運算式中各種條件組合。例如前述程式，第一個布林運算式有下列 4 種條件組合：

- (1) A>1 , B=0
- (2) A>1 , B<>0
- (3) A<=1 , B=0
- (4) A<=1 , B<>0

第二個布林運算式也有下列 4 種條件組合：

- (5) A=2 , X>1
- (6) A=2 , X<=1
- (7) A<>2 , X>1
- (8) A<>2 , X<=1

測試資料  $(2,0,4)$  滿足 1、5。 $(2,1,1)$  滿足 2、6。 $(1,0,2)$  滿足 3、7。 $(1,1,1)$  滿足 4、8。因此這四個測試資料所組合成的測試資料群便可涵蓋上述 8 種條件組合。

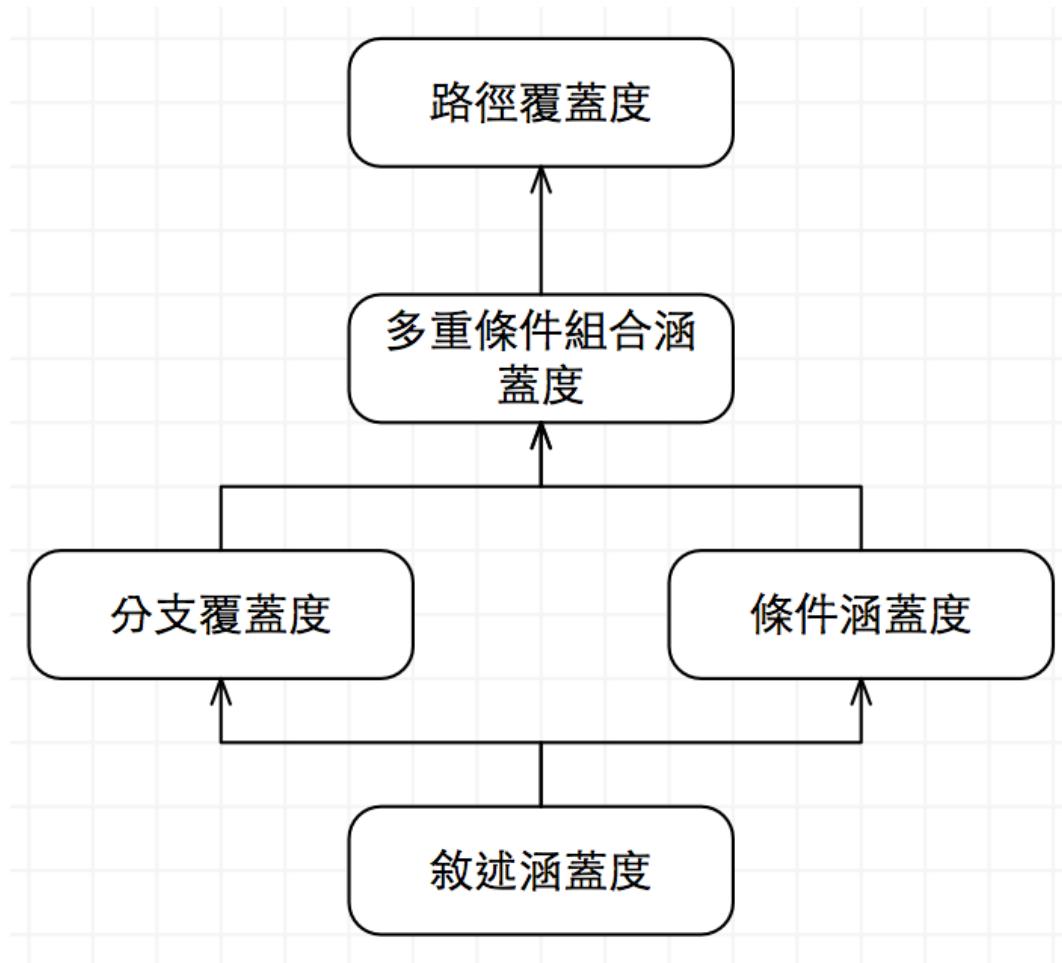


圖 2.2: 各種涵蓋度的關係

## 2.4 Eclemma

EclEmma 是一個在 Eclipse 上的 Java 程式碼涵蓋工具外掛，與 JUnit 配合它可以檢驗還有哪些敘述還沒有被執行到、測試到。

### 安裝及使用步驟

1. 在 Eclipse > Help > Install New Software > Add > <http://update.eclemma.org/>;
2. 安裝完後重新啟動 Eclipse，會看到 tool bar 上有一個多了一個 button;
3. 撰寫程式及其相關的 JUnit 測試碼;
4. 點擊 EclEmma，會執行 JUnit 並呈現涵蓋狀況。

圖 2.4 為 EclEmma 跑出來的樣子：綠色表示該敘述有被完全執行、黃色表示部分、紅色表示尚未執行。Eclemma 採取的是條件涵蓋度 conditional coverage，如果條件涵蓋度不為百分百，你會發現該行是標記為黃色。例如執行到 `if (x>0)` 這個指令，如果你的測試碼只有 `x=10`, 則該行就會標記為

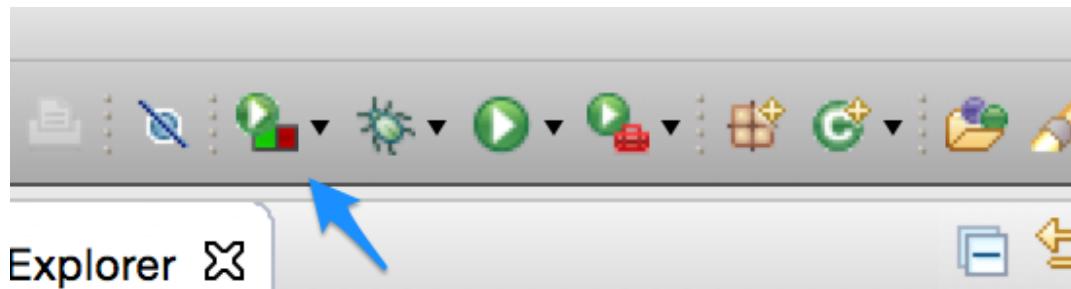


圖 2.3: Eclemma 圖示

黃色，即便你的敘述涵蓋度已經是百分百。如果再加上  $x=-10$  的測試碼，則條件涵蓋度為百分百就回呈現綠色。使用時，你可以將游標移到程式碼左方的菱形上，它會出現你所檢核過的狀況。

**動手實驗：**針對以下的程式（`xAdd()` 絶對值相加），撰寫 JUnit 測試碼。變化你的測試碼來觀察涵蓋度的變化<sup>1</sup>。

```
1 public int xAdd(int x, int y) {  
2     if (x<0 && y<0)
```

## LECTURE 2. 涵蓋度測試

The screenshot shows a Java code editor with the following code:

```
public String checkTriangle() {  
    String result = "";  
  
    if ((a + b < c) || (b + c < a) || (c + a < b))  
        result = "not a triangle";  
    else if (a == b && b == c && a == c) {  
        result = "regular triangle";  
    } else if ((a * a + b * b == c * c) || (b * b + c * c == a * a) || (c *  
        result = "right-angled triangle";  
    } else result = "triangle";  
  
    return result;  
}
```

The code is annotated with colored highlights and icons on the left side of the editor. There are three yellow diamonds above the first if statement, one yellow diamond above the second if statement, and one red diamond above the third if statement. The text 'not a triangle' is highlighted in pink, 'regular triangle' in light green, and 'right-angled triangle' in light blue.

圖 2.4: Eclemma 程式碼涵蓋工具

```
3      return -1*(x+y);  
4      if ( x<0 & y<0)  
5          return -1*(x+y);  
6      if ( x< 0)  
7          return -1*x+y;
```

## LECTURE 2. 涵蓋度測試

---

```
8     if (y< 0)
9         return -1*x+y;
10    return x+y;
11 }
```

### 2.5 練習

1. 寫一個程式判斷三角形，並寫一點 JUnit 測試碼，執行測試時，改以 EclEmma 來測試此程式。EclEmma 會標示你沒有測試到的部份，這時候在加上 JUnit 測試碼，逐步提高測試率。[\[參考程式碼\]](#)
2. 針對以下程式，設計測試案例，以達到最佳的分支包含度。

```
1 if (height > 2 || height <= 1)  
2     return -1;  
3 if (weight > 100 || weight <= 40)  
4     return -2;  
5 double BMI = height/(weight*weight);  
6 if (BMI > 120) return -3;  
7 if (BMI < 15) return -4;  
8 return BMI;
```

## 課後問卷

<http://goo.gl/forms/rAroFPJaElCgboFs2>