Chapter 1

如何寫一支程式

循環漸進、測試先行。

循環漸進：每次寫程式的『週期』(cycle)變得更短，甚至一次還寫不到一整個 class 的量，而是一次幾行、或一個 method。

測試先行：在寫程式之前，先寫可以自動及重複執行的單元測試程式。

這種方法有幾個好處：

- 程式碼是可測試的：因為先寫測試程式，所以，程式就是為了讓測試程式能過而寫就的。
- 測試程式都測過了：讓測試程式能過的程式還沒生出來時，我們會目睹測試不過，而當程式寫好，我們則親眼看到測試程式過關。
- 測試可以重複執行：因為測試都被『程式碼化』了。
- 測試形同程式文件：如果有人想知道物件如何運作、想動到程式碼，可以來看看這些測試程式，並把它們都跑過一遍。
- 設計將變得極簡化：我們的設計會“剛剛好足夠”到只讓測試程式能過。

我們會開發一個小型的書籍查詢系統（bibliographic system）來展示測試先行的寫程式方法。在這種方法中，我們將看到單元測試跟簡單設計如何相輔相成。整個寫程式的過程都是一次一小步，只加進足以讓測試程式過關的程式碼。整個過程如同鐘擺的節奏：測試一點、寫程式一點、測試一點、寫程式一點。

單元測試與 JUnit

單元測試程式是測試先行的關鍵所在。本書的測試程式將採用由 Kent Beck 與 Erich Gamma 所開發的 JUnit 單元測試框架（這是 Java 版的）。這個框架可以由 *www.junit.org* 取得。其它數種語言版本的測試框架則可以在 *http://www.xprogramming.com* 取得。

以下是一支單元測試程式的典型模式：

- 建立一些物件。
- 呼叫一些 method，改變這些物件的狀態。
- 假定改變的結果就是你斷定的那樣。

以 JUnit 來說，我們讓測試程式繼承一個自訂的樣版類別（stereotypical class）TestCase，以下的單元測試程式範例測的是 Vector 的數個 method：

```
import junit.framework.*;

public class TestVector extends TestCase {
  public TestVector(String name) {super(name);}

  protected void setUp() {
      // 執行每個測試案例之前要做的事
  }

  public void testAddElement() {
```

```
            // 建立一些物件
        Vector v = new Vector();
            // 呼叫一些 methods
        v.addElement("Some string");
        v.addElement("Another string");

            // 『斷定』測試結果
        assertEquals(2, v.size());
    }

    public void testSomethingElse() {
        // 跑別的測試
    }
}
```

會有一個 TestRunner class 來跑這個測試程式，它開始測試的方式是：先呼叫 `setUp()` method，然後再呼叫其它真正在做測試的 method；之後又是同一循環，亦即，再度先呼叫 `setUp()` method，然後再呼叫其它真正在做測試的 method（這些個別的測試 method，可以隨你用不同的次序呼叫）。如果有任何一個測試通不過你『斷定』（assert）的那樣，錯誤就會被記錄下來。

JUnit 還有其它功能，如果想知道更多的資訊，請參閱它的說明。

設計

假設我們的書籍資料結構是作者、書名、出版年份。我們的目的是寫一個可以依照指定內容做搜尋的系統，我們心裡設想的使用者介面如圖 1.1。

我們把這件事分成兩部分來做：非視覺化的物件模型（model）跟視覺化的使用者介面（user interface）。這一章展示的是物件模型的測試先行方式，如果你想對測試先行運用在使用者介面上有個快速的了解，請參見本章後面的附註（sidebar）。

一開始是個簡短的設計週期，我們最初的已知是，有一堆 Documents，Documents 存有本身的屬性（作者、書名、年份）。

我們設計的 Searcher 會知道如何搜尋到文件，運作的方式是：給定一個 Query，它會傳回一個 Result（符合搜尋條件的 Documents 集合），如圖 1.2 所示。

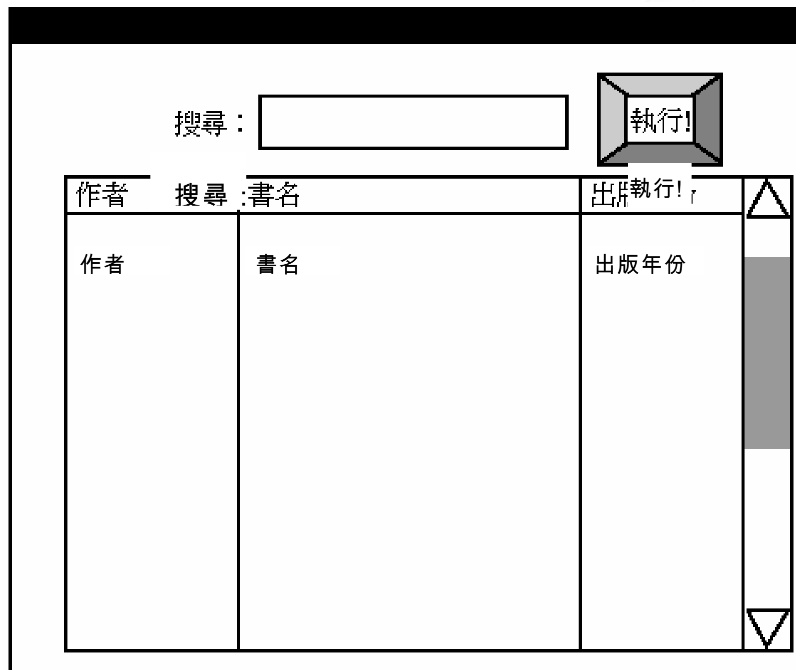

圖 1.1：搜尋系統的使用者介面

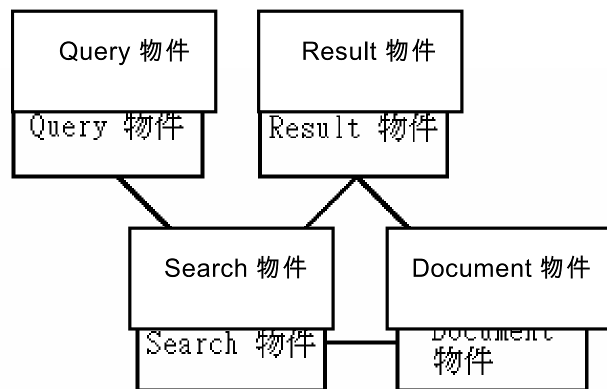

圖 1.2：四個主要物件

請注意，這樣的設計用幾張卡片就可以記錄（不需要動用到電腦）。如同在隱喻那章會討論到的，我們的設計用的是“單純的隱喻”（naive metaphor），亦即，物件均根據領域內涵而創造。

跟著這樣的設計走，我們可以“由下而上的”寫出這些物件的單元測試程式（以及這些物件本身）: Documents、Result、Query、Searcher 等（測試先行不一定需要“由下而上”，這裡會用這種方式，只因為它看起來還蠻直接了當的）。[1]

Document 物件

Document 要知道本身的作者、書名、年份。我們會先建立一個“資料袋”（data bag）class，但在這一切開始之前，還是先寫測試程式吧：

```
public void testDocument() {
    Document d = new Document("a", "t", "y");
    assertEquals("a", d.getAuthor());
    assertEquals("t", d.getTitle());
    assertEquals("y", d.getYear());
}
```

Document class 都還沒建呢！因此，這個測試程式在編譯時就不會過了，所以我們先把 Document class 建起來，method 的程式內容先空著。這時再把這個測試程式跑一遍，確定它不會過。這似乎有點可笑---我們不是希望測試過關嗎？對，我們希望，但先看著它不過，可以確定這個測試程式本身是正確的。而如果偶爾這個測試程式竟然過了，『那就好玩了！』

最後，我們把這個測試程式加上建構式，還有會讓測試過關的 method 們。

這整個迷你流程在圖 1.3 還會再強調一遍（程式重整留到下一章談）。整個流程確保我們同時看到測試過和不過的狀況，這可以讓你確定幾件事：

- 這個測試程式的確測了一些東西，
- 如果你動了程式碼，測試結果就真的會不同，
- 你的確已經為系統加進了有用的功能。

Kent Beck 和其它人會告訴你，不必操心那些簡單的屬性存取及設定的 methods（setter、getter）和建構式的測試程式，只需"測可能出錯的每個地方"（以及絕不能出錯的地方）即可。Kent 說："寫了太多的測試程式後，你就會知道適可而止了。" 我在這個例子中有用到屬性存取及設定 methods 的測試程式，但我自己實際的做法上是儘量少寫它們。這裡的用意只是想讓自己注意到，這類 method 大量存在表示該 class『體重過重』，把該 class 的一些『責任』分擔到其它 class 或許會更好。

XP 的『寫測試程式/寫程式』週期

- 寫一個測試程式。
- 編譯該測試程式。應該不會過，因為你還沒寫測試程式要測的那個程式。
- 寫剛好夠測的程式，然後編譯它（如果必要的話，先進行程式重整）。

- 執行測試程式，看著它不過。
- 寫『剛好讓測試程式能過』的程式量。
- 執行測試程式，看著它過。
- 重整程式，讓它的架構變得更清晰，並去除掉程式碼重複的部分。
- 重複整個過程。

圖 1.3：XP 的『寫測試程式/寫程式』週期

你可能會有一些問題要問：

*這個週期要花多久？*1 至 5 分鐘，最多也許會多到 10 分鐘。

*如果超過 5 分鐘呢？*那就把測試程式縮小一些。

*5 分鐘，真的嗎？*對，是真的。

*在測試程式跟被測程式之間這樣跳來跳去，成本不會很高嗎？*不一定。例如，在 IBM 的 VisualAge for Java 整合環境中，你寫好這個測試程式然後存檔，它會告訴你編譯有錯，所以你就把原來那些空的 method 補上內容、然後存檔、執行 JUnit（這個程式一直是常駐的），此舉會重新把 class 載入並執行測試程式，結果不過。然後你接著寫要真正要跑的程式、存檔、再測一遍，結果過了。如果你無論如何是打

算寫單元測試程式的，你增加的負擔只不過是寫一個測不過的method，然後看著它不過而已。

即使你是以命令列的方式做，狀況也沒糟到哪裡，因為你還是可以讓整合環境的視窗一直處於開啟的狀態。

*在日後的系統維護階段時，這些測試程式不會降慢你寫程式的速度嗎？如果狀況有變，不是連程式跟測試程式都要一起改嗎？*不會的，事實上正好相反，在系統維護階段，測試程式還可以讓你寫程式的速度變快呢！因為它們讓你有信心去改變。如果做了什麼不對的事，你知道測試程式會警告你。的確，如果物件的介面變了，你必須連同測試程式一起變，但這也不是那麼難。況且，你可能會發現這種測試先行的寫程式方式，會傾向於讓你在一開始就打造出一個較為不易改變的介面。

Result 物件

Result 要知道兩件事：它所包含的 Document 總數跟那些 Document 的內容。我們先來測試一個總數為零的空 Result。

```
public void testEmptyResult() {
  Result r = new Result();
  assertEquals(0, r.getCount());
}
```

建個 Result class，先寫一個空的 getCount() method，在加上 return 0 這一行之前，看著它不過，這時請注意我們的程式設計方式：先以最簡單的解決方案來做。

下面的程式範例，測的是包含 2 個 Document 的 Result：

```
public void testResultWithTwoDocuments() {
  Document d1 = new Document("a1", "t1", "y1");
  Document d2 = new Document("a2", "t2", "y2");
  Result r = new Result(new Document[]{d1, d2});
  assertEquals(2, r.getCount());
  assertSame(d1, r.getItem(0));
  assertSame(d2, r.getItem(1));
}
```

加上 getItem() method(此時先讓它傳回 null 值), 看著測試不過(我之後就不再提這點了，但請你繼續照這樣的方式做下去，這的動作只花幾秒鐘，但卻可以讓你稍稍再次確定你的測試程式本身是正確的)。一個簡單的 Result class 實作大概會是下面這樣子：

```
public class Result {
  Document[] collection = new Document[0];

  public Result() {}

  public Result(Document[] collection) {
      this.collection = collection;
  }

  public int getCount() {return collection.length;}
```

```
  public Document getItem(int i) {return
collection[i];}
}
```

測試過關，所以這個 class 算是完成了。

請注意我們用以表現一堆 Document 的簡單方式 : 陣列。這是 XP “ 可以運作的最簡單設計” 原則 (有時也被縮寫成 DTSTTCPW) 的一個例子，因為目前的系統並不需要任何比陣列更炫的設計，就算真有那麼一天需要，我們也會知道，那時再來傷腦筋。

*你真的就是這樣做嗎？還是只是為了舉例而簡化這一切？*對，我真的就是這樣做，即使在寫實際要上線跑的程式時，我都很少寫了幾行的程式碼還不寫測試程式的（往往寫不超過 5 行就會寫個測試程式了)。

Query 物件

Query 物件可以直接用一個查詢字串來做，這個字串的值就是我們要搜尋的內容，如下：

```
public void testSimpleQuery() {
  Query q = new Query("test");
  assertEquals("test", q.getValue());
}
```

建個 Query class，給它一個建構式，好讓它記住自己的這個查詢字串，再加上一個 `getValue()` method，用來傳回查詢字串的值。

Searcher 物件

Searcher 是最引人入勝的 class，先講比較容易的部分：如果 Document 是一個空集合的話，我們應該什麼也搜尋不到。

```
public void testEmptyCollection() {
  Searcher searcher = new Searcher();
  Result r = searcher.find(new Query("any"));
  assertEquals(0, r.getCount());
}
```

這個測試程式編譯不會過，所以我們先把 Searcher 物件的 method 名稱寫出來，內容先空著沒關係。

```
public class Searcher {
  public Searcher() {}
  Result find(Query q) {return null;}
}
```

測試程式編譯過了，但執行結果不對（因為這時 `find()` method 傳回的還是個 null 值呢），我們改一下程式就可以修正這個錯誤了：`public Result find(Query q) {return new Result();}`

當我們真正嘗試做搜尋時，狀況會變得比較具備挑戰性，因為我們接著會面對一個問題：Searcher 從哪裡搜尋 Document 呢？所以我們一開始要傳個陣列到 Searcher 的建構式讓它接，但在寫這些之前，還是先寫它的測試程式吧！

```
public void testOneElementCollection() {
  Document d = new Document("a", "a word here", "y");
  Searcher searcher = new Searcher(new Document[]{d});
  Query q1 = new Query("word");
  Result r1 = searcher.find(q1);
  assertEquals(1, r1.getCount());
  Query q2 = new Query("notThere");
  Result r2 = searcher.find(q2);
  assertEquals(0, r2.getCount());
}
```

這個測試程式說明了我們不但要測試*找得到*的，連*找不到*的也要測。

要實作出這個功能，我們得提供一個新的建構式，好讓這個測試程式先能編譯過關（雖然執行起來結果還是不對），之後就是真的要寫執行起來也會過關的程式碼了。

首先，Searcher 物件必須要在別人多次呼叫其 `find()` method 之際，仍能保留它的 Document 內容，所以這裡會加一個成員變數來記錄那些結果，也可以由建構式中接一個引數來初始化這個成員變數，如下：

```
Document[] collection = new Document[0];

public Searcher(Document[] docs) {
```

```
    this.collection = docs;
}
```

現在，find() method 的最精簡版本可以先做了，它會依次掃過自己的每一份 Document, 然後把符合 Query 條件的 Document 加到 Result 裡，如下：

```
public Result find(Query q) {
    Result result = new Result();
    for (int i = 0; i < collection.count; i++) {
        if (collection[i].matches(q)) {
            result.add(collection[i]);
        }
    }
    return result;
}
```

嗯，看起來不錯，但還有 2 個問題：Document 還沒有 matches() method、而且 Result 也還沒有 add() method。

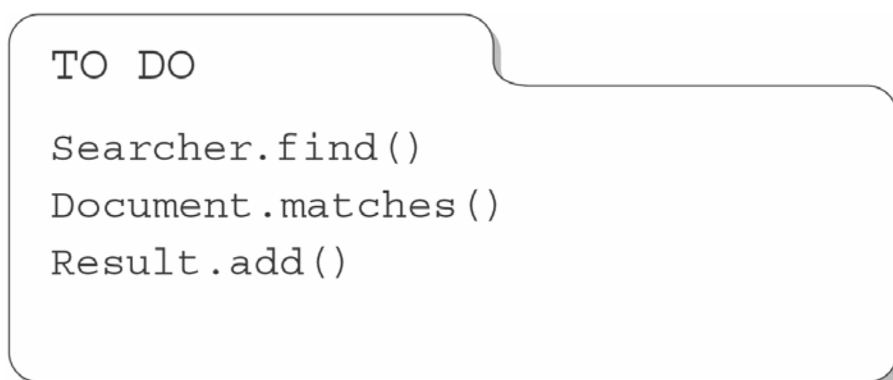

圖 1.4：待辦事項清單（To-Do List）的長相

目前為止，我們已經做了一些東西出來，而我常常發現，把一些待辦事項（To-Do）記錄在一張卡片，對做事情很有幫助，我可以把一串想做的事都『網羅』在卡片上，如圖 1.4 那樣。

我們再加上一個測試：Document 裡的每個元素（field）都可以被符合條件的 Query 搜尋到，反之則搜尋不到。

```
public void testDocumentMatchingQuery() {
  Document d = new Document("1a", "t2t", "y3");
  assert(d.matches(new Query("1")));
  assert(d.matches(new Query("2")));
  assert(d.matches(new Query("3")));
  assert(!d.matches(new Query("4")));
}
```

我們最終必須處理 3 種狀況的查詢：空白值的查詢、部分字串的查詢、還有是否要區分大小寫等。目前我們會假設『空白值』跟『部分字串』都算符合，我們的 Searcher 也會區分大小寫，但以後我們可能會改變這想法。

好了，現在的資訊已經足以讓我們實作出 `matches()` method 了。

```
public boolean matches(Query q) {
  String query = q.getValue();
  return
      author.indexOf(query) != -1
  || title.indexOf(query) != -1
  || year.indexOf(query) != -1;
}
```

這個 method 可以讓 `testDocumentMatchingQuery()` 那個測試程式過，但 `testOneElementCollection()` 這個 method 卻會有問題，因為 Result 物件還沒有 `add()` method 呢！所以，先幫這個即將要寫的 method 寫個測試程式吧：

```
public void testAddingToResult() {
  Document d1 = new Document("a1", "t1", "y1");
  Document d2 = new Document("a2", "t2", "y2");

  Result r = new Result();
  r.add(d1);
  r.add(d2);

  assertEquals(2, r.getCount());
  assertSame(d1, r.getItem(0));
  assertSame(d2, r.getItem(1));
}
```

這個測試程式不會過，因為我們還要把 `Result.add()` method 寫出來才行，但在寫之前，有必要把之前的程式碼重整一番：之前的 Result 物件是用陣列來存 Document 的，但對於一個容量需要改變的結構而言，陣列並非最佳的選擇，於是我們改用 Vector。

```
Vector collection = new Vector();

public Result(Document[] docs) {
  for (int i = 0; i < docs.length; i++) {
      this.collection.addElement(docs[i]);
  }
}
public int getCount() {return collection.size();}
```

```
public Document getItem(int i) {
  return (Document)collection.elementAt(i);
}
```

改用 Vector 之際，確定舊有的單元測試程式 testEmptyResult() 跟 testResultWithTwoDocuments()仍然可以過，接下來，我們可以為 Result 加上 add()這個新的 method 了：

```
public void add(Document d) {
  collection.addElement(d);
}
```

我們來看一下 Result(Document [])這個建構式，它當初是用來讓 testResultWithTwoDocuments()這個測試過關而做的設計，因為要建立一個包含文件的 Result，這是唯一的方式。但稍後我們引入了 Result.add()這個設計，這是 Searcher 物件需要的。因此陣列式的建構式就不再需要了。此時，我們把思維轉到測試的角度，修正這個設計，把：

```
Result r = new Result(new Document[]{d1,d2});
```

改成：

```
Result r = new Result();
r.add(d1);
r.add(d2);
```

我們確定了一下，所有測試仍舊可以過，所以，現在把陣列式的建構式拿掉，不會有什麼問題了。再把所有測試跑過一遍，確定沒有其它東西有呼叫到這個建構式。從這裡我們也看到了，`testAddingToResult()` 現在基本上已經跟 `testResultWithTwoDocuments()` 做相同的事了，所以稍後我們也會一併將之拿掉。像這種在重整時，測試程式或多或少會變動的情形還算常見，class 的介面會變、甚至連測試程式本身要做的事也會變。

最後，我們的測試程式測 Document、Result、Query、Searcher 等物件都沒問題了。

XP 在使用者介面的『寫測試程式/寫程式』週期

對於圖形使用者介面（GUI）而言，並沒有很多的 XP 團體在用循環漸進、測試先行的程式設計方式，但知道怎麼用這種方式，卻也還蠻有用的。

我在 Ron Jeffries 等人所寫的《*Extreme Programming Installed*》（2000 年出版）這本書中負責“ A Java Perspective” 那一章，這個附註是那章的摘要。

- 先寫測試程式再寫程式的方式可以運用在 GUI 的很多方面，但不能涵蓋到所有層面。

- 即使螢幕的物件配置（layout）有所變動，GUI 的測試程式一樣可以罩得住。
- 在 Java 裡的一些元件，像是文字輸入欄位（field）以及按鈕（button）等，照樣可以用程式化的方式如 `getText()`、`setText()`、`doClick()`等加以模擬。
- 在 Java 裡，相對位置也可以利用 `getLocationOnScreen()` 這種函式來測試。
- GUI 可以善用非視覺化模型（即使其 methods 內容仍暫為空白），在測試上取得更為細密的控制能力。

但即使有這些技巧，你還是沒有辦法測到 GUI 的每一個層面（特別是那種在對話框和多個螢幕之間互動的狀況更難測到）。在 GUI 的程式設計上，所有關於商業邏輯的部分，最好還是放在中間層的 model 裡。

載入 Document 物件

Searcher 物件從何得知它要搜尋的 Document 物件呢？目前我們是在它的建構式傳一個 Document 的陣列，但現在我們希望 Searcher 能夠自己載入 Document。

還是先寫測試程式，這次我們先傳一個 Reader 物件給它，然後等著看它丟出一個例外。我們也先假定它有個 `getCount()` method，這個 method 是用來給測試是不是真的載入了某些東西。我們把測試程式跟

被測的程式放在同一個包裹 (package) 中 , 這種做法的一個優點是 , 你可以讓測試程式利用非公開 method 長驅直入被測程式的內部 , 一窺物件的內部狀態。

(但大多數的時候我們並不需要如此做 : 通常經由公開介面就足以把一個物件『測透透』了。)

```
public void testLoadingSearcher() {
  try {
                    // \t=欄位分隔符號, \n=記錄分隔符號
      String docs = "a1\tt1\ty1\na2\tt2\ty2";
      StringReader reader = new StringReader(docs);
      Searcher searcher = new Searcher();
      searcher.load(reader);
      assertEquals(2, searcher.getCount());
  } catch (IOException e) {
      fail ("Loading exception: " + e);
  }
}
```

注意一下 , 這裡的 Searcher 用的還是陣列的方式 (當時最直接了當的選擇), 我們會對它做跟 Result 一樣的事 : 把陣列『重整』成 Vector。

```
package search;
import java.util.*;

public class Searcher {
  Vector collection = new Vector();
  public Searcher() {}

  public Searcher(Document[] docs) {
```

```
        for (int i = 0; i < docs.length; i++) {
            collection.addElement(docs[i]);
        }
    }

    public Query makeQuery(String s) {
        return new Query(s);
    }

    public Result find(Query q) {
        Result result = new Result();
        for (int i = 0; i < collection.size(); i++) {
            Document doc =
        (Document)collection.elementAt(i);
            if (doc.matches(q)) {
                result.add(doc);
            }
        }
        return result;
    }
}
```

確定既有的測試程式都過了之後,我們就可以開始寫 load 這個 method 了，如下：

```
// Searcher:
public void load(Reader reader) throws IOException {
    BufferedReader in = new BufferedReader(reader);
    try {
        String line = in.readLine();
        while (line != null) {
            collection.addElement(new Document(line));
            line = in.readLine();
        }
```

```
    } finally {
        try {in.close();} catch (Exception ignored) {}
    }
}

int getCount() {
    return collection.size();
}

// Document:
public Document(String line) {
    StringTokenizer tokens = new StringTokenizer(line,
"\t");
    author = tokens.nextToken();
    title = tokens.nextToken();
    year = tokens.nextToken();
}
```

Searcher 那個陣列式的建構式現在不再需要了，我們把它拿掉，順便去改一下測試程式：

```
public void testOneElementCollection() {
    Searcher searcher = new Searcher();
    try {
        StringReader reader = new StringReader(
                        "a\ta word here\ty");
        searcher.load(reader);
    } catch (Exception ex) {
        fail ("Couldn't load Searcher: " + ex);
    }

    Query q1 = searcher.makeQuery("word");
    Result r1 = searcher.find(q1);
    assertEquals(1, r1.getCount());
```

```
  Query q2 = searcher.makeQuery("notThere");
  Result r2 = searcher.find(q2);
  assertEquals(0, r2.getCount());
}
```

回顧

現在把焦點轉到設計，從兩個角度看我們所發展出來的 method 們：一個是 Search 客戶端（client），另一個是 Searcher 物件[3]（如表 1.1），誰用到了哪些公開 method[4]？

再來看看 Document 和 Query 這兩個物件，我想知道它們是不是夠稱之為物件（因為它們做的不過是資料儲存體（data bag）做的事罷了）。但它們看起來都還不錯，頗有『準領域物件』（near-domain class）的架勢，所以我們暫時先抑制住衝動，不去動這塊。另外，在 Result 跟 Searcher 物件的責任分擔方面，感覺也還蠻平衡的。

表 1.1：我們所發展出的公開 methods

Search client	**Searcher class**
`Document.getAuthor()`	`New Document()`
`Document.getTitle()`	`Document.matches()`
`Document.getYear()`	`Query.getValue()`
`New Query()`	`New Result()`
`Result.getCount()`	`Result.add()`
`Result.getItem()`	
`Searcher.find()`	

問答

*我們把陣列改成 Vector 這種資料結構的動作就做了兩次，這樣的開發流程，難道不會產生一堆僵局嗎？這算是流程中的一種瑕疵嗎？*不，它不算是一種瑕疵，我們引入陣列那種設計的當時，它的確是夠用的，只是當我們有其它需要時，它就被改變了。只要不是『有去無回』[2]，我們並不介意用簡單的方法。我們又不是無所不知，所以自然會有改變心意的一天；重要的是在於我們要確定自己不致被差勁或過度的設計（overly complex design）卡得進退不得。

*我們應該要具備多少的測試程式呢？*如果單元測試程式的程式量是被測程式的 1 到 3 倍，不要覺得意外。

總結

我們已經以 XP 的方式開發出了書籍查詢系統的模型，這種開發方式強調設計簡單化、以及測試程式跟被測程式之間交互輪替的撰寫過程。單元測試程式對我們的設計、實作、重整均有極大的幫助，我們這個系統，最終可以選擇要掛上一個簡單的命令列操作介面，或是圖形操作介面。

資源

- Beck, Kent. 2000. Extreme Programming Explained. Boston: Addison-Wesley.
- Fowler, Martin, Kent Beck, John Brant, William Opdyke, and Don Roberts. 1999. Refactoring: Improving the Design of Existing Code. Reading, MA: Addison-Wesley.
- JUnit. 網址：http://www.junit.org.
- Wake, William C. 2000. "A Java Perspective." In Extreme Programming Installed, by Ron Jeffries, Ann Anderson, and Chet Hendrick-son. Boston: Addison-Wesley.
  這章的『續集』。

譯註

1. 以下四個『物件』, 原文寫的是 class , 類別之意。在程式碼實際的撰寫上 , 我們在建立這些『物件』之時 , 當然是先建立它的『類別』, 但我們卻常說 : “ 你的系統有哪些物件？” 這裡的物件指的既是類別也是物件 , 一者為靜態 , 一者為動態 , 對 OO 稍有了解的讀者當能體會。我們不也說『物件導向』而不說『類別導向』嗎？ ☺

2. 指缺乏彈性的方法。

3. Searcher 物件可以視之為 search 伺服（server）端。

4. 表 1.1 是為了看看我們所發展出的 method 都被誰叫用了？這有助於檢視物件間責任分配的平衡。