Chapter 2

什麼是程式重整

“重整是改善既有程式碼的設計。”

—*Martin Fowler*

Chapter 2 什麼是程式重整

*重整*是在不影響其外在行為的前提下，改善既有程式碼的設計，原因是為了讓程式碼儘可能保持簡單，可以為任何即將到來的改變做準備。關於重整更完整的討論，請參閱 Martin Fowler 的《*Refactoring*》一書（1999）。

在這章裡，我們會以實際的程式碼開始，然後對它進行幾次的重整動作。最後，程式碼的意圖會更清晰、設計會更完善、品質會更上一層。

重整動作需要這些東西：

- 原來的程式碼
- 單元測試程式（確定自己沒有不經意改動了程式的外在行為）
- 一個方法，確定原來的程式的確需要改進（程式碼有『壞味道』了嗎？）
- 一組我們知悉如何運用的重整技巧（重整技巧的型錄）
- 一套指導我們整件事如何進行的流程

原來的程式碼

下面是一段用來產生一份網頁的程式碼，它會把一個檔案中的內容讀到一個字串，然後把 `%CODE%` 和 `%ALTCODE%` 置換成我們輸入的一個 id。程式可以跑，但效能測試顯示它的執行速度實在太慢，主要是

因為程式用了太多的暫存字串，我們只要稍微注意一下，就可以知道這段程式碼的確需要好好清理一番。

```
import java.io.*;
import java.util.*;

/** 把 %CODE% 置換成我們輸入的一個 id,
**  把 %ALTCODE% 置換成我們輸入那個 id"中間加一槓（dashed)。
*/

public class CodeReplacer {
public final String TEMPLATE_DIR = "templatedir";
String sourceTemplate;
String code;
String altcode;

/**
* @param reqId java.lang.String
* @param oStream java.io.OutputStream
* @exception java.io.IOException The exception
description.
*/
public void substitute(String reqId, PrintWriter out)
                                  throws IOException
{
    // 讀入 template 檔
String templateDir = System.getProperty(TEMPLATE_DIR,
"");
StringBuffer sb = new StringBuffer("");
try {
    FileReader fr = new FileReader(templateDir +
                                     "template.html");
    BufferedReader br = new BufferedReader(fr);
```

```
    String line;
    while(((line=br.readLine())!="") && line!=null)
        sb = new StringBuffer(sb + line + "\n");
    br.close();
    fr.close();
} catch (Exception e) {
}
sourceTemplate = new String(sb);

try {
    String template = new String(sourceTemplate);
        // 置換 %CODE%
    int templateSplitBegin =
template.indexOf("%CODE%");
    int templateSplitEnd = templateSplitBegin + 6;
    String templatePartOne = new String(
            template.substring(0,
templateSplitBegin));

    String templatePartTwo = new String(
            template.substring(templateSplitEnd,
                               template.length()));
    code = new String(reqId);
    template = new String(
        templatePartOne+code+templatePartTwo);

        // 置換 %ALTCODE%
    templateSplitBegin =
template.indexOf("%ALTCODE%");
    templateSplitEnd = templateSplitBegin + 9;
    templatePartOne = new String(
        template.substring(0, templateSplitBegin));
    templatePartTwo = new String(
        template.substring(templateSplitEnd,
                           template.length()));
    altcode = code.substring(0,5) + "-" +
```

```
                    code.substring(5,8);

 out.print(templatePartOne+altcode+templatePartTwo);
 } catch (Exception e) {
     System.out.println("Error in substitute()");
 }
 out.flush();
 out.close();
 }
}
```

單元測試程式

要重整這段程式，首先要建立一些單元測試程式，驗證基本的功能運作無誤，如果你已經在用 XP 那種『循環漸進、測試先行』的程式設計方式，這些測試程式應該都已經齊備了，它們應該是那種程式設計方式過程中的副產品。

下面這個測試程式需要一個檔名為 template.html、內容為“xxx%CODE%yyy%ALTCODE%zzz”的檔案。

```
import java.io.*;
import junit.framework.*;

public class CodeReplacerTest extends TestCase {
   CodeReplacer replacer;

   public CodeReplacerTest(String
testName){super(testName);}
```

```
    protected void setUp() {replacer = new
CodeReplacer();}

    public void testTemplateLoadedProperly() {
        try {
            replacer.substitute("ignored",
                        new PrintWriter(new
StringWriter()));
        } catch (Exception ex) {
            fail("No exception expected, but saw:" + ex);
        }

        assertEquals("xxx%CODE%yyy%ALTCODE%zzz\n",
                    replacer.sourceTemplate);
    }

    public void testSubstitution() {
        StringWriter stringOut = new StringWriter();
        PrintWriter testOut = new PrintWriter
(stringOut);
        String trackingId = "01234567";
        try {
            replacer.substitute(trackingId, testOut);
        } catch (IOException ex) {
            fail ("testSubstitution exception - " + ex);
        }

        assertEquals("xxx01234567yyy01234-567zzz\n",
                    stringOut.toString());
    }
}
```

這個測試程式使用前一章介紹的 JUnit 單元測試框架。

程式碼有『壞味道』了嗎？

Martin Fowler 和 Kent Beck 用“ 程式碼的味道” 這個隱喻，來描述當你看到程式碼時的感覺[1]。程式碼的味道通常都偏向於一種“ 不好的訊號” ，而不是那種明顯錯誤的症狀。或許你聽過一個類似的概念叫“ 反模式” （anti-patterns, after Brown et al. 1988），或“ 事情不妙的直覺” （Spidey-sense, after Stan Lee’s Spider-man, 1996）

在程式碼中，你會看到哪些潛在的危險訊號呢？

- 太長的 class
- 太長的 method
- Switch 敘述（而不是用多型的技巧）
- “ 結構體” （Struct）class（除了屬性存取的 method 們之外，沒什麼其它的功能）
- 重複的程式碼
- 幾乎（但不*全然*是）重複的程式碼
- 太過倚賴原始型別（而不是創造一個內容更領域別的型別）
- 沒用的（或錯誤的）註解
- 還有其它許許多多『壞味道』...

有些壞味道是明顯而立即的，但有些非得等到你重整過程中才會發現。

回到上述那段程式碼，看看自己可以從中聞到哪些壞味道（別把自己侷限在這份壞味道清單裡！找出自己認為的壞味道）。

重整技巧的型錄

Martin Fowler 那本重整的書中，有一半的篇幅都是重整技巧的型錄所『貢獻』的，每一種技巧都是很簡單的轉換動作，Fowler 在那本書中解釋了改變的機制，也舉了一些例子（請參見圖 2.1）。我們這裡也要介紹一個例子，是那本書沒有的唷！請參見圖 2.2：

Extract Method 重整技巧

重整前

```
// 假設均為實體變數
// (instance variables)

void f() {
     ...
  // Compute score
     score = a * b + c;
     score -= discount;
}
```

重整後

```
void f() {
     ...
  computeScore();
}

void computeScore() {
     score = a * b + c;
     score -= discount;
}
```

圖 2.1：重整的一個例子

資料來源：Fowler, Martin. 1999. *Refactoring: Improving the Design of Existing Code*. Reading, MA: Addison-Wesley.

Replace String with StringBuffer 重整技巧	

重整前
 	
`String a, b, c;`
`  :`
`return a + b + c;`
	重整後
	
	`String a, b, c;`
	`  :`
	`StringBuffer sb = new`
	`  StringBuffer(a);`
	`sb.append(b);`
	`sb.append(c);`
	`return sb.toString();`

圖 2.2：重整的另一個例子

圖 2.2 的重整把簡單易讀的字串版（使用 String 型別）改成潛在而言更具效率的字串緩衝區（使用 StringBuffer 型別）版（或倒過來：許多的重整動作是把程式從複雜整頓到簡單）。有些編譯器可以自動幫我們做這件事，這樣一來，我們的程式碼就可以繼續保持高度的可讀性，效率的事，就交給編譯器。但下面舉的例子，我們還是自己來做這件事吧。

如果世界上所有的編譯器，都可以自動幫我們把程式碼重新組織，我們還需要這般的重整動作嗎？我相信還是需要的。圖 2.2 左邊的程式碼比較短，但它用了 Java 中“ +”這個運算子的多載功能，右邊則是使用了“ .”這一個普遍的運算子來處理 method 的叫用。如果我們會希望可以把資料型別替換掉的話(用 String 跟 StringBuffer 以外的型別), 最好從現在開始就用圖 2.2 右邊的那個版本。

整個指導流程

在一種可以讓你交替寫測試程式和改程式的環境中工作。做一次重整動作後，就接著跑一次單元測試，一直重複這樣的過程，直到你的程式碼清楚而簡單的表達出它想做的事，而且沒有重複。最初頻繁執行的測試程式可能會讓你覺得這一切很麻煩，但之後，這種做法肯定會加快你的速度（跑個測試只花幾秒鐘，卻能讓你安心，因為之前花幾分鐘改的程式是沒問題的）。

這個過程一直持續著，要到怎樣才算結束呢？當程式碼達到以下目標時：

1. 通過所有測試程式（表示這段程式碼可以運作無誤了）。
2. 表達出該程式碼需要表達出的東西。
3. 沒有重複。
4. 有著儘可能少的 class 跟 method。

上述目標是依重要性排下來的：例如，如果程式碼為了表達出它需要表達的東西而造成了重複，那重複就可以被允許。這些目標通常也被濃縮成一句話：“只做一次”（once and only once，第一個 once 指的是『運作無誤』，後面的 only once 指的是『不重複』）

在《*The Pragmatic Programmer*》這本書中，作者 Dave Thomas 和 Andy Hunt 談到了一個類似但適用性更廣的一個規則，叫做『DRY 原

則』(“ Don’t Repeat Yourself” ，本身切勿重複): “ 在一個系統之中，每一分的知識都必須有一個單一、清楚、可靠的表現方式。”

開始重整

當你打量著這章開頭的那一段程式時，有聞到什麼『壞味道』嗎？以下是我聞到的：

- 過長的 class
- 過長的 method
- 有些變數可以搬到 method 裡，變成局部變數
- method 的註解根本沒用
- template 可以只讀取一次而非每次都讀嗎？
- 程式碼被綁死在檔案系統了[2]
- 字串的比較用“ !=” 可能會有問題
- 用了 StringBuffer 型別，卻沒用到它的 append() method
- 在迴圈中重複建立了 StringBuffer 變數，浪費記憶體
- 關閉檔案的 `close()` method 沒有出現在 `catch` 或 `finally` 子句裡
- 例外處理程式段不一致，也不明確

- 用了太多的字串相加命令
- 處理“ %CODE%”跟“ %ALTCODE%”置換的兩小段程式碼幾乎重複
- 額外的 new String()命令太多
- 暫存變數太多

最壞的設計就屬 substitute() method，它太長了，所以我們利用 *Extract Method* 把它打散。本書使用的許多重整技巧，可以在這個網址取得：*http://www.refactoring.com*。

首先,我們把讀取 template 的動作拉出來,自成一個 readTemplate() method：

```
String readTemplate() {
 String templateDir = System.getProperty(TEMPLATE_DIR,
"");
 StringBuffer sb = new StringBuffer("");
 try {
     FileReader fr=new
FileReader(templateDir+"template.html");
     BufferedReader br = new BufferedReader(fr);
     String line;
     while(((line=br.readLine())!="")&&line!=null)
         sb = new StringBuffer(sb + line + "\n");
     br.close();
     fr.close();
 } catch (Exception e) {
 }
```

```
        sourceTemplate = new String(sb);
        return sourceTemplate;
    }
```

即使這個改變簡單到不可能出錯，還是*跑一下測試程式*吧（我第一次跑測試程式果然還是不過，因為我忘記在原來的程式中呼叫這段新的 method 了，所以，有必要強調先跑測試這種做法）！

這裡也請注意一下，我們並不會一次就馬上把程式大卸八塊，而是一次一小步一小步的進行，發展出一種穩定的步調：改一些程式、跑個測試、改一些程式、跑個測試...。在沒有確定一切都 OK 前，我們絕不會改太多，這樣一來，如果有什麼錯誤，一定也是發生在我們剛剛才做的動作上。

讓我們只從一個地方取得 template 的名字（利用 *Introduce Explaining Variable* 重整技巧），把 `templateDir` 置換成：

```
String templateName =
System.getProperty(TEMPLATE_DIR,"") + "template.html";
```

（跑一次測試）然後把 fr 變數拿掉（利用 *Inline Temp* 重整技巧）：

```
BufferedReader br = null;
...
br = new BufferedReader(newFileReader(templateName));
```

（`fr.close()` method 也可以拿掉了）（跑一次測試）

現在可以來修正我們之前注意到的某個 bug 了：如果程式出錯的話，檔案不會被關閉。

```
try {
    ...
} catch (Exception ex) {
} finally {
    if (br != null) try {br.close();}
        catch (IOException ioe_ignored) {}
}
```

（跑一次測試）

接下來看另一個潛在的問題：字串的“ !=”運算子。我們很確定（靠著到處看、到處問）template reader 不打算在讀到空白行或類似內容時跳出迴圈，所以那個 while 條件沒有意義。

```
String line = br.readLine();
while (line != null) {
    sb = new StringBuffer(sb + line + "\n");
    line = br.readLine();
}
```

（跑一次測試）

Martin Fowler（1999）指出，把修正 bug 跟重整的動作分開來做，會比較安全一點。他的做法是，（1）先寫個新的測試程式說明 bug 在哪

裡，(２) 然後重整程式碼，(３) 再然後才是回過頭去修正那個 bug。不過我們這個小小例子省略了前兩個步驟。

再看看 sb 變數的設值動作，它重複做了：當我們可以把讀進的內容加進一個已經有的 StringBuffer 變數時，實在沒有理由每次都建立一個新的。

再者，我們也可以用 StringBuffer 的 `append()` method 來代替字串相加的動作 (*Replace String with StringBuffer* 重整技巧)

```
sb.append(line);
sb.append('\n');
```

(跑一次測試)

不用 `sourceTemplate = new String(sb)` 這樣的方式，我們把建立字串的工作放到 StringBuffer 裡：`sourceTemplate = sb.toString()`；(跑一次測試)。

這段程式設值給 sourceTemplate，並且同時傳回這個值。我們把設值的任務移到呼叫程式身上，此處只保留 `return sb.toString()`，並把傳回值的型改成 String (*Reapportion Work between Caller and Callee* 重整技巧)(跑一次測試)。

至於例外處理，我們把程式改成丟出一個 IOException 的例外，然後把空的 catch 子句消掉，由呼叫程式去處理所有的例外。這會讓程式的行為有所改變（而且...，嗯，還沒測試唷！），亦即，如果此時發生錯誤的話，連部分的 template 都不會被傳回呼叫程式。我們要多看多問，確定是不是真的可以這樣做[3]（跑一次測試）。

現在，我們的程式變成下面這樣了：

```
String readTemplate() throws IOException {
   String templateName =
System.getProperty(TEMPLATE_DIR, "")
                           + "template.html";
   StringBuffer sb = new StringBuffer("");
   BufferedReader br = null;
   try {
       br = new BufferedReader(new
FileReader(templateName));
       String line = br.readLine();
       while (line != null) {
           sb.append(line);
           sb.append('\n');
           line = br.readLine();
       }
   } finally {
       if (br != null) try {br.close();}
                   catch (IOException ioe_ignored) {}
   }
   return sb.toString();
}
```

這段程式中另一個我不喜歡的地方，是它把 template 的來源跟讀取方法都定死了，這使它完全耦合在檔案系統上，這在未來可能會是個問題（如果我們要 template 從其它地方來的話）。其實眼下就有問題了：就是我們的測試程式必須動用到外部檔案。

由於我還不確定正確的方式應該是怎樣，這個問題就先擱著吧。

置換“ %CODE%”

重整到目前為止，程式還是太長了，而且在置換字串方面的程式碼，也還是有著幾乎重複的現象。所以，下一步我們會利用 *Extract Method*，重整置換“ %CODE%”這部分的程式碼。

```
String substituteForCode(String template, String reqId)
{
    int templateSplitBegin =
template.indexOf("%CODE%");
    int templateSplitEnd = templateSplitBegin + 6;
    String templatePartOne = new String(
        template.substring(0, templateSplitBegin));
    String templatePartTwo = new String(
        template.substring(templateSplitEnd,
template.length()));
    code = new String(reqId);
    template = new
String(templatePartOne+code+templatePartTwo);
    return template;
}
```

們 substitute() method 裡還剩下幾個變數，我們接著來調整一下它的宣告方式（跑一次測試）。

我第一件注意到的事情，是“ %CODE%”這個字串跟“ 6”這個值（“ %CODE%”的字串長度），我們把這個字串拉出來，之後就可以直接拿來用，而不必每次都用“ %CODE%”了（*Replace Magic Number with Calculation* 重整技巧）。

```
String pattern = "%CODE%";
int templateSplitBegin = template.indexOf(pattern);
int templateSplitEnd =
templateSplitBegin+pattern.length();
```

（跑一次測試）

我們也建了太多新的 String 變數：所有 new String()建構式都重複到了，因為這些建構式的引數本身就已經是 String 了，所以我們把程式改成這樣（*Remove Redundant Constructor Calls* 重整技巧）:

```
String templatePartOne =
       template.substring(0,templateSplitBegin);
String templatePartTwo =
       template.substring(templateSplitEnd,
template.length());
code = reqId;
return templatePartOne + code + templatePartTwo;
```

（跑一次測試）

最後,我們也會把剩下的字串相加動作重整一番(在 return 命令那兒),但先來關心一下“ %ALTCODE%”的部分。

置換“ %ALTCODE%”

對於原程式中置換“ %ALTCODE%”的那段,我們也用相同的 *Extract Method* 重整技巧跟簡單化的動作。弄完之後,現在來看看我們做到哪兒了。

```
void substituteForAltcode(String template, String code,
                          PrintWriter out) {
String pattern = "%ALTCODE%";
int templateSplitBegin = template.indexOf(pattern);
int templateSplitEnd =
templateSplitBegin+pattern.length();
String templatePartOne = template.substring(
            0, templateSplitBegin);
String templatePartTwo = template.substring(
            templateSplitEnd, template.length());
altcode = code.substring(0,5) + "-" +
code.substring(5,8);
out.print(templatePartOne + altcode + templatePartTwo);
}
```

這段程式碼跟上面一段的 substituteForCode()還真是像呢,所以很明顯,我們應該可以把這兩段很像的程式改得一樣。不過它們之間還是有三處不同:要找的字串不同、置換成不同的值、最後寫到的地方也不同。

我們用 *Parameterize Method* 把它們改得一樣：亦即，把要找的字串跟要置換的字串當作引數傳進去。

```
void substituteForAltCode(String template, String
pattern,
   String replacement, PrintWriter out) {
   int templateSplitBegin =
template.indexOf(pattern);

   int templateSplitEnd =
templateSplitBegin+pattern.length();
   String templatePartOne = template.substring(
               0, templateSplitBegin);

   String templatePartTwo = template.substring(
               templateSplitEnd, template.length());
   out.print(templatePartOne + replacement +
templatePartTwo);
}
   ...
   altcode = reqId.substring(0,5) + "-" +
reqId.substring(5,8);
       substituteForAltCode(template, "%ALTCODE%",
altcode, out);
```

(跑一次測試)

現在可以來處理過多的字串相加動作了，我們以一串的 `print()` 命令來取代它 (*Replace String Addition with Output* 重整技巧)，並且把 `out.flush()` 也給拉出去，跟這些 `print()` 命令放一起。

```
   out.print(templatePartOne);
```

```
        out.print(replacement);
        out.print(templatePartTwo);
        out.flush();
```

（跑一次測試）

現在只剩下寫入的目的地不同了，“ %CODE%”傳回一個 String，而“ %ALTCODE% ” 則是寫到一個 PrintWriter，這可以用 Java.io.StringWriter class 把它們改得一個樣。我們建立一個 StringWriter 的變數當作 PrintWriter 的引數，讓 substituteForCode() 接這個 PrintWriter 當作它的資料流（stream）（*Unify String and I/O* 重整技巧）（跑一次測試）。

我怎麼知道要找像 StringWriter class 這樣的東西呢？首先，我可以在跟同事的對話之中發現這個 class，總是有人會知道一些你不知道的、藏在偏僻角落的一些功能。

（在 XP 裡，你總是會有個搭檔跟你一起開始，如果你的搭檔也不知道，團隊中其它人就在同一個房間裡，你可以問他們）其次，我經歷過很多系統，所以我知道這些系統中有很多可以讓你把字串當作 I/O，或把 I/O 當作字串來處理。再者，當我學習 Java 時，可以從對 Java API『讀透透』的過程中學到它。最後，這個 class *就在*文件中，真的需要用時，隨時都找得到。

現在這兩段程式碼都一樣了，所以我們把 substituteForAltcode() 消掉，直接用 substituteForCode() 即可（*Merge Identical Routines* 重整技巧）（跑一次測試）。

回到readTemplate()

在一開始時就把 template 讀進來，而非每次呼叫 substitute() 時都再去讀一次，我們確定這種做法可接受（問客戶而確定的）。因此，我們讓建構式丟出一個 IOException 的例外，然後把原程式改成呼叫 readTemplate()。

```
sourceTemplate = readTemplate(
    System.getProperty(TEMPLATE_DIR, ""));
```

（跑一次測試）

改成這樣，有助於我們少建立一些字串。

最後，我們可以來對付之前那根眼中釘了：直接從檔案中讀進一個 template。目前 readTemplate() 生成檔名的方式是接一個目錄名稱加上一個檔名，我們會改成傳一個 Reader 給它，讓 Reader 接手做這件事。第一步，先把 FileReader 的建立動作拉到 CodeReplacer 的建構式中（*Reapportion Work between Caller and Callee* 重整技巧）。

```
public CodeReplacer() throws IOException {
    String templateName =
```

```
System.getProperty(TEMPLATE_DIR, "") +
"template.html";
    sourceTemplate = readTemplate(
                          new
FileReader(templateName));
}

public String readTemplate(Reader reader)
                    throws IOException {
    ...
}
```

(跑一次測試)

還有一種做法是，在 CodeReplacer 的另一個建構式中接收一個 Reader，由呼叫程式負責建立這個 Reader(可能用 `getProperty()` 的方式來做)。

```
public CodeReplacer(Reader reader) throws IOException
{
    sourceTemplate = readTemplate(reader);
}
```

(跑一次測試)

轉回到測試者的思維，我修正了我的測試程式，現在，`testTemplateLoadedProperly()`的動作可以簡單化了，因為測試 template 不需要再做置換的動作了，只需載入它即可。另外，也不必再準備一個 `template.html` 的外部檔案，用 StringReader 來測試就可以了，這有助於我們把測試程式跟作業環境脫鉤。

```
final static String templateContents =
                        "xxx%CODE%yyy%ALTCODE%zzz\n";
        ...
   replacer = new CodeReplacer(new StringReader(
                                templateContents));
        ...
public void testTemplateLoadedProperly() {
   assertEquals(templateContents,
replacer.sourceTemplate);
}
```

（跑一次測試）

最後，在原程式 `substitute()` 裡的 `close()` 似乎應該移出去了，因為這個物件的呼叫程式會開啟 stream，所以我們也希望它一併負責關閉的動作。我們會修改 `substitute()` 跟呼叫它的程式（*Reapportion Work between Caller and Callee* 重整技巧）。

（再把測試最後跑一次）

最後的版本

下面是 CodeReplacer.java 的全新版本：

```
import java.io.*;
import java.util.*;
/** 把 %CODE% 置換成 requested id，把 %ALTCODE% 置換成 "
中間加一槓（dashed）的 id。
```

```
*/

public class CodeReplacer {
    String sourceTemplate;

    public CodeReplacer(Reader reader) throws IOException
{
        sourceTemplate = readTemplate(reader);
    }

    String readTemplate(Reader reader) throws IOException
{
        BufferedReader br = new BufferedReader(reader);
        StringBuffer sb = new StringBuffer();
        try {
        String line = br.readLine();
        while (line!=null) {
            sb.append(line);
            sb.append("\n");
            line = br.readLine();
        }
        } finally {
            try {if (br != null) br.close();}
            catch (IOException ioe_ignored) {}
        }
        return sb.toString();
    }

    void substituteCode (String template, String pattern,
                         String replacement, Writer out)
                            throws IOException {
    int templateSplitBegin = template.indexOf(pattern);
    int templateSplitEnd =
            templateSplitBegin + pattern.length();
    out.write(template.substring(0, templateSplitBegin));
    out.write(replacement);
```

```
    out.write(template.substring(
                templateSplitEnd, template.length()));
    out.flush();
    }

    public void substitute(String reqId, PrintWriter out)
                    throws IOException {
        StringWriter templateOut = new StringWriter();
        substituteCode(sourceTemplate, "%CODE%",
                        reqId, templateOut);

        String altId = reqId.substring(0,5) + "-" +
                        reqId.substring(5,8);
        substituteCode(templateOut.toString(),
"%ALTCODE%",
                        altId, out);
    }
}
```

這裡則是 CodeReplacerTest.java 的程式碼：

```
import java.io.*;
import junit.framework.*;

public class CodeReplacerTest extends TestCase {
    final static String templateContents =
                    "xxx%CODE%yyy%ALTCODE%zzz\n";

    CodeReplacer replacer;

    public CodeReplacerTest(String testName)
    {super(testName);}

    protected void setUp() {
```

```
        try {
            replacer = new CodeReplacer(new StringReader(

    templateContents));
        } catch (Exception ex) {
            fail("CodeReplacer couldn't load");
        }
    }

    public void testTemplateLoadedProperly() {
        assertEquals(templateContents,
                    replacer.sourceTemplate);
    }

    public void testSubstitution() {
        StringWriter stringOut = new StringWriter();
        PrintWriter testOut = new PrintWriter (stringOut);
        String trackingId = "01234567";

        try {
            replacer.substitute(trackingId, testOut);
            testOut.close();
        } catch (IOException ex) {
            fail ("testSubstitution exception - " + ex);
        }
        assertEquals("xxx01234567yyy01234-567zzz\n",
                    stringOut.toString());
    }
}
```

分析

現在，我們已經有了一個較完美的版本了。一路下來，我們修正了一些 bug，也把一些程式語意不明確的地方變得更明確，很明顯的，程式的可讀性變高了，因為我們把重複的程式碼都給『擠』出去了；我們也把多處的字串相加動作改成耗費資源較省的方式；把 template 跟檔案系統隔離開來，改用一個 Reader 來載入它。

還有一些重整的動作可做：變數名稱可以取得更好（例如像是 `sb` 和 `br` 這種名稱就可以改進）；另外，那個帶有“ -”（dash）的置換結果字串，它的形成方式也不明顯。

新版的程式表現出三樣基本的東西：

1. template 的表現方式（目前的做法是用字串）
2. 置換的進行方式
3. 把什麼置換成什麼（“ `%CODE%`”及“ `%ALTCODE%`”）

前兩樣是 template 運作的部分細節，第三樣是“ 商業邏輯”（business logic），說明如何處理這個 template。一個獨立運作的 Template class 可以滿足前兩點，在本章一開始的那個爛程式中，這點還並不明顯。這也顯示了一點：亦即，我們對程式『壞味道』的感覺，是會隨時間而改變的：像是我現在就可以辯說，太過依賴 String 型別，會讓我們在一種太低的層次上架構我們的系統。

把 template 拉出來變成一個獨立的 class, 可以讓我們更針對效能做改進，因為這樣一來，我們就可以任意的改變 template 的表現方式，目前的版本在找出需要置換的字串時，走了兩次的 template，我們或許可以對 template 做些前置處理 (preprocess), 以便找出可能需要置換之處，這可以藉由改變置換過程的介面達成：把『什麼要置換成什麼』先放在一個清單內，如此，只需走一次 template，置換動作就可以全部完成。

總結

所以，從這一切的重整動作中，我們學到了什麼？

- 我們可以循環漸進的對程式碼做非常微小的改變, 讓每一個最新的版本都比前一版好一些, 而不必冒上把程式“ 全部殺掉重新來過” 的潛在風險。
- 單元測試程式可以長期扮演一種保護傘的角色, 我們絕不會脫離它們所提供的保險，即使只是幾分鐘。
- 某些改善的結果使得更進一步的改善變得可能, 例如在本章開頭的程式中, Template class 獨立出來這種需求, 在最早的版本中根本就看不出來。
- 雖然某些重整的動作可能會傷到效能, 但它們往往也為更大幅度的效能改善開啟了可能性。把 `readTemplate()` 獨立出來的結果, 使我們注意到這個 method 在每次置換動作時都被呼叫了一

次，而其實它是可以全部只被呼叫一次的，未來的 Template 版本或許還可以一次就把置換動作通通做掉。跟 *Extract Method* 當初所導致的“額外”程序呼叫所造成的效能流失比起來，這些被開啟的可能性，在效能改進所扮演的角色上，才真正算是『大尾』的呢。

- 在寥寥幾步的重整動作中（大約 20 步），我們已經大幅度的改善了原有的程式碼。

讓重整（包含測試）成為你程式設計正規做法中的一部份吧！讓自己對程式的壞味道變得敏感，並且去學習可以對付這些壞味道的重整技巧。這些動作終究會在你的系統所擁有的簡單性與彈性之中獲得報償。

資源

- Beck, Kent. 2000. Extreme Programming Explained. Boston: Addison-Wesley.
- Bentley, Jon L. 1982. Writing Efficient Programs. Englewood Cliffs, NJ: Prentice-Hall.
- Bentley, Jon L. 1988. More Programming Pearls: Confessions of a Coder.Reading, MA: Addison-Wesley.
- Bentley, Jon L. 2000. Programming Pearls, Second Edition. Boston: Addison-Wesley.

- Brown, William H., Raphael C. Malveau, Hays W. McCormick III, and Thomas J. Mowbray. 1988. AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis. New York: John Wiley and Sons.
- Fowler, Martin, et al. 1999. Refactoring: Improving the Design of Existing Code. Reading, MA: Addison-Wesley.

  重整技巧及程式壞味道的一份型錄。
- Fowler, Martin. 2001. Online catalog. 網址：http://www.refactoring.com.
- Hunt, Andrew, and David Thomas. 2000. The Pragmatic Programmer:From Journeyman to Master. Boston: Addison-Wesley.
- JUnit. 網址：http://www.junit.org.
- Lee, Stan. ed. 1996. The Ultimate Spider-Man. New York: Boulevard

  （Berkeley Publishing Group 的一個子公司）

譯註

1. 這句的原文是“ what you sense” ，在英文裡，sense 也包含味覺，the five senses 就是視覺、聽覺、嗅覺、觸覺、味覺。

2. 程式被寫成只能由檔案中讀取 template。

3. 問客戶，看看他們可不可以接受這種狀況。