國立臺灣大學電機資訊學院資訊工程學系

碩士論文

Department of Computer Science and Information Engineering
College of Electrical Engineering and Computer Science
National Taiwan University

Master Thesis

結合 NetBeans IDE 的設計模式特殊化

Design Pattern Specialization Integrated with NetBeans IDE

黃群凱

Huang, Chun-Kai

指導教授：陳俊良 博士

Advisor: Chen, Chuen-Liang Ph.D.

中華民國 98 年 7 月

July, 2009


國立臺灣大學
教育學術品德

致謝

誠摯的感謝 陳俊良教授對於學生在研究上的指導與啟發，老師自由開放的學風，對我們學生有莫大的影響，讓學生在求學過程中更具備獨立思考及自主學習的能力。在每次討論的過程中，老師總是能夠提供豐富且扎實的建議，激發了學生很多研究的想法並且解決許多難題。在論文接近完稿的最後階段，老師更是不厭其煩的為我審視論文內容，促使本論文得以順利完成。

感謝李秀惠教授、甘宗左教授，能夠在百忙之中抽空前來擔任學生的口試委員，不厭其煩的指正缺失，並且對本論文內容惠賜良多寶貴意見，使本論文更加完備。

感謝 R403 實驗室的學長政寰、宗翰在剛進入台大時給予幫助，讓我更快融入學習環境，感謝學弟沐恒，為實驗室增添豐富且歡樂的氣息，最後感謝同期的騏嘉，這兩年來共同歷經學習的心酸與歡樂，讓研究的生活能夠堅持下去。

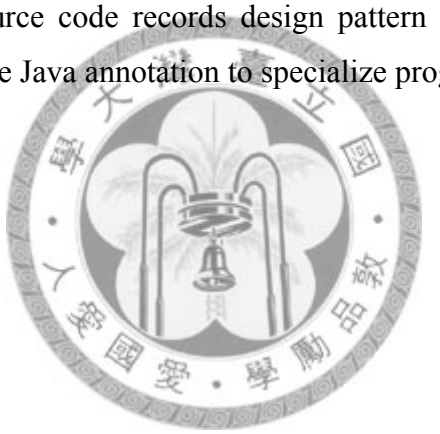
中文摘要

Design pattern 特殊化可以有效的從通用的(generalization)程式設計轉化成特殊要求(specialization)的程式設計。軟體設計上，通用的程式設計有助於程式的維護與修正，相對的，產品化時需要講求效率與其他特別要求、如嵌入式系統。為了權衡這兩者的對比，我們研究了 design pattern 特殊化的技術。

我們研究了如何在一個實現 design pattern 概念的 Java 程式上加入了 design pattern 的註記。為了達到這個目的，我們修改了 NetBeans IDE 的 UML 功能，藉由新的 UML 功能，來產生夾帶 design pattern 內容的 Java annotation。利用這樣的 design pattern 註記，我們可以針對 design pattern 設計的程式進行效率與特別要求的特殊化處理。

ABSTRACT

Design patterns raise the abstraction level at which people design and communicate design of object-oriented software. It offers many advantages in software development, but introduces overheads into programs. On deploy stage; we can use program specialization to eliminate these overheads. Furthermore, specialization can help program more suitable for embedded system. This paper describes the architecture and implementation of design pattern specialization. In our architecture, the user edits UML for software development and adds design pattern information to UML. The UML and design pattern information will generate java source code by NetBeans IDE. The generated java source code records design pattern information on java annotation. Then, we use Java annotation to specialize programs.



目錄


國立臺灣大學
教育愛國・學術卓越

圖目錄

表目錄

1. 簡介

1.1 研究動機

軟體的生命週期大體上可分為兩大階段：開發階段以及產品化階段。因圖 1-1 的白色箭頭代表了開發階段的流程。Java 的原始程式碼經過編譯器後被轉換成 class 檔；class 檔可以在 Java 虛擬機器[1][2]執行。假如執行結果不正確或是要添加新功能，則周而復始重複開發階段的流程。圖 1-1 的黑色箭頭表示當進入產品化階段，因為 Java 的特性，只要將 class 檔搬移到使用者的 Java 虛擬機器上執行即可。

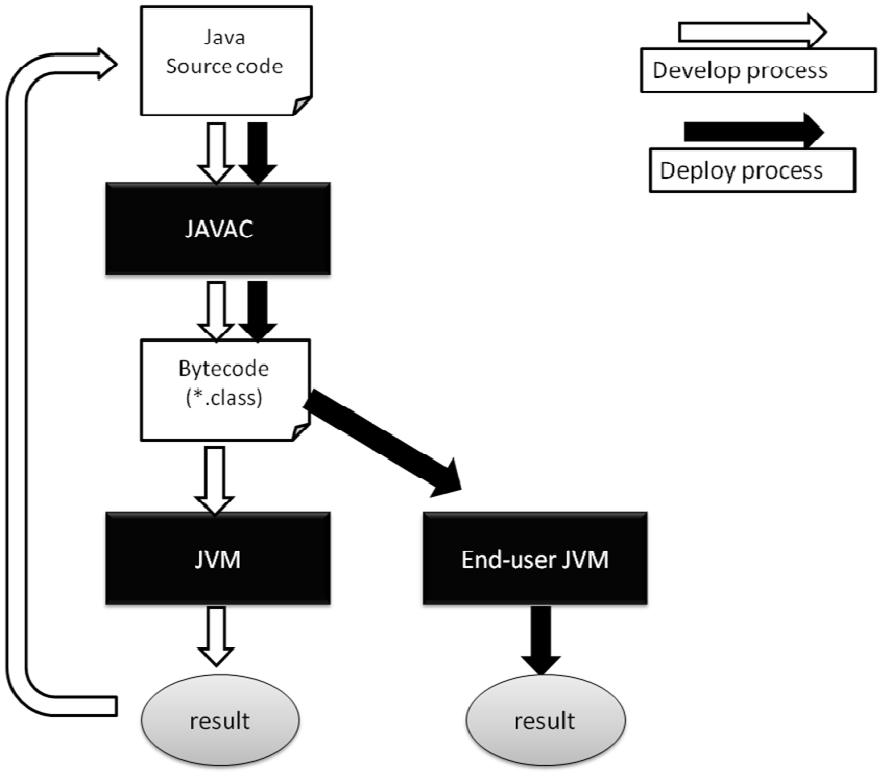


圖 1-1：現今軟體設計流程

design pattern 的概念自從 1995 年的 Gemma 等四人(泛稱 GOF)的書出版至今已經十多年[3]，現今有不少大型程式都基於 design pattern 的概念去設計與實做。程式引用 design pattern 有利於落實軟體工程的想法，可以顧及程式的維護與修改上的便利，適合多位程式設計人員共同開發。另一方面，引用 design pattern 的程式會犧牲程式的執行效率。為了達到雙贏，設計完成的原始程式碼在產品化的階段，可以將程式進行特殊化，藉由特殊化的處理使程式達到效率的提升或者達到嵌入式系統較嚴苛的環境限制。

圖 1-2 中，在產品化階段，我們可以先把開發階段用的 class 檔，考量使用者機器的特性，轉成特殊化的 class 檔，希望可以得到較佳的執行效果。

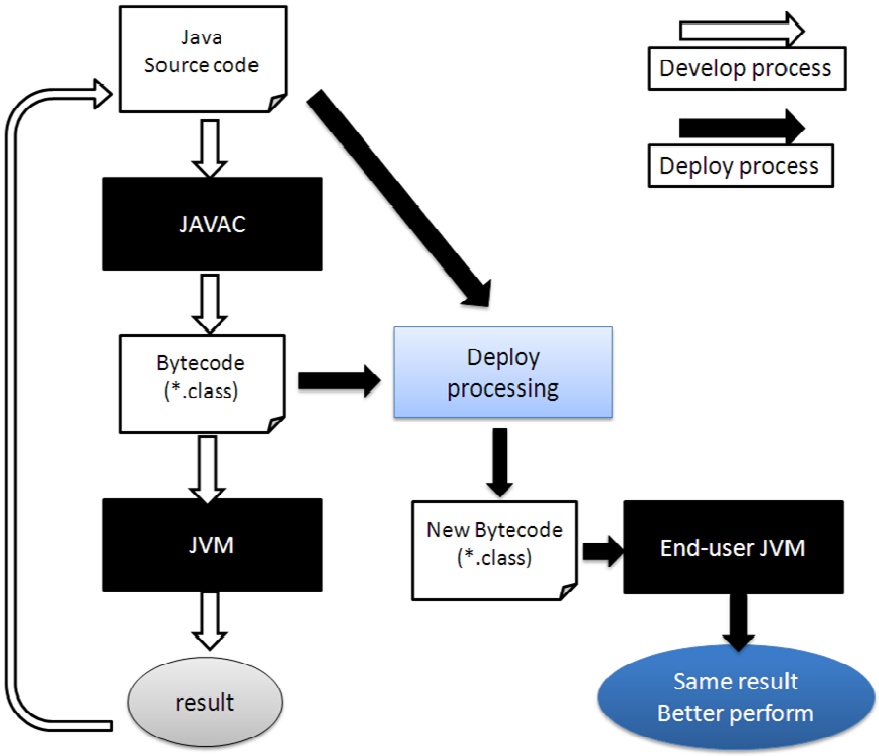


圖 1-2：構思的新流程

1.2 目標

我們的目標是要建立一套工具能夠將引用 design pattern 概念的程式進行特殊化處理，因此 design pattern 的資訊是不可缺少的。目前的編譯器都具有將程式的效率提升的選項與功能，但是 design pattern 是一個高階的程式設計概念，要從程式中分析出 design pattern 的概念是複雜且困難的工作，因此目前的編譯器幾乎均無法自動優化引用 design pattern 的程式。為了能夠得到 design pattern 的資訊，只好有賴於程式開發者提供相關的訊息，來判別程式運用何種 design pattern。但是假如在程式產品化階段才要求程式開發者觀察程式碼然後再輸入 design pattern 訊息，這項作業將太過繁雜，而且有判斷錯誤的可能性。

為了彌補上述的缺陷，我們設計了一套流程，來幫助程式開發者可以從設計階段就加入 design pattern 資訊，並且將該資訊保留到產品化階段。軟體開發初期經常運用 UML[4]來進行結構的設計。NetBeans([5])是一個目前廣泛使用的 IDE (Integrated Development Environment[6])。具有編輯 UML 的功能[7]且能夠引用 GOF design pattern 到 UML 中。在本論文中我們將修改 NetBeans，讓程式開發者運用 NetBeans 設計 UML 的同時，能夠加入 design pattern 的訊息。相較於在產品化階段再要求程式開發者加入 design pattern 的訊息，在編輯 UML 的同時加入 design pattern 訊息要顯得正確與簡易。

UML 設計完成後，程式開發者必須進行程式編輯的工作。在進入這個過程前，我們需要有將 UML 轉換成為程式碼的工具[8]，利用這樣的工具除了可以減輕程式開發者的負擔，也可以將 UML 設計中的 design pattern 資訊保留到程式碼當中。在本論文中我們選擇了 Java annotation[9]來儲存 design pattern 的資訊，Java annotation 並不會造成程式開發者撰寫程式上的影響，所以 Java annotation 可以留存到產品化階段，當作 design pattern 的資訊來運用。目前的 NetBeans IDE 已經能夠藉由 UML 來產生程式碼框架，我們修改了 NetBeans，加入了產生與編輯 Java annotation 的功

能，同時將 design pattern 的資訊事先存在 Java annotation 中。

下一個步驟，我們必須把具有 design pattern 資訊的程式碼進行特殊化處理。在本論文中我們引用若干對 Java 程式做特殊化的概念([11][12])，來對程式碼做修改。這項工作我們是把其嵌入了 Java 編譯器中，並在執行過程中以文字介面來顯示特殊化的步驟，幫助程式開發者了解特殊化的進度。

我們的流程結合了現有的 NetBeans IDE 所具有的功能以及軟體工程上的設計流程，希望程式開發者在 UML 設計階段便導入 design pattern 的概念，然後藉由我們修改後的 NetBeans IDE，程式開發者在設計階段的 design pattern 資訊能夠保留到產品化階段，進而針對 design pattern 程式進行特殊化處理。

圖 1-3 是我們上述的設計架構，淺色箭頭是原先的開發流程，差別在於 Java 程式碼當中帶有表達 design pattern 的訊息的 Java annotation。在產品化的階段，暗色箭頭流程會觸發特殊化的程序，產生執行效果較優異的 class 檔，該 class 檔會更適合使用者的平台來執行。

1.3 論文架構

本篇論文將在第 2 章介紹論文的背景以及所運用到的工具，其中包含了 NetBeans IDE 的架構介紹。第 3 章將討論如何修改 NetBeans IDE，並且將 design pattern 資訊加入到 NetBeans UML 當中。在 NetBeans UML 產生 Java code 的同時，如何將 UML 當中的 design pattern 的資訊轉成 Java annotation，並且加入到生成的程式碼中。第 4 章當中，我們假設程式中附有表達 design pattern 角色的 annotation。我們的特殊化作業將會設計一套命令語言，利用命令來修改程式碼。我們將可以利用命令的編輯來達到特殊化的作業，利用外部的命令編輯來提升特殊化作業的彈性，並提供使用者設計適用於自己程式的特殊化命令。第 5 章我們將利用前兩章的工具來展示開發過程 UML 的設計以及程式開發完成後，特殊化命令的編輯。

藉由這兩者的展示來表現 design pattern 特殊化隊程式碼的修改，同時測試特殊化前後 bytecode 的大小與執行效率。第 6 章是結論和未來工作的看法，我們認為 design pattern 特殊化有助於降低程式所需的硬體門檻，有利於嵌入式系統的開發。

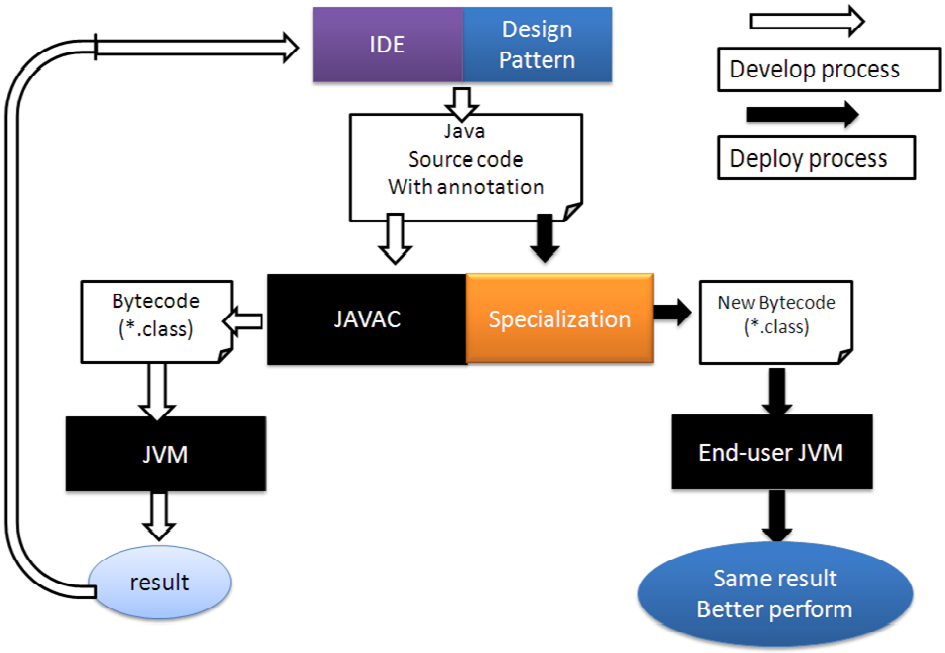


圖 1-3：論文實作流程圖

2. 背景知識

2.1 Java annotation

Java 系統在 1.5 版之後加入了 annotation 的功能[9]。Annotation 與本來就有的註解有何不同呢？我們整理在表 2-1 中。

Kind	範例	特性
Single line comment Or Block comment	**//comment Or /* comment */**	**1. 存在原始碼中 2. 只被設計人員辨識**
Documentation comment (Javadoc)	/** * @information */	**1. 存在原始碼中 2. 可被轉換成文件**
Annotation	**@annotation_name() public string toString(){ …. }**	**1. 存在原始碼和中間碼中 2. 可以被處理 3. 多種應用**

表 2-1：三種註釋的比較

第一類的 single line comment 以及 block comment，程式設計人員編寫的註解只會存在於原始程式碼當中，目的在於對其他的程式設計人員解釋程式碼的意義。此類註解只會被人所辨識，編譯器不會對它有任何的動作。

第二類的 documentation comment 必須合乎特定的文法規則，由符號/**開始。這類的註釋可以利用外部的應用程式 javadoc 解讀註釋然後產生出說明文件，最常見的是 html 的格式。

第三類的 annotation 則是一個可被進一步的處理，annotation_name 必須事先被程式設計人員定義。然後程式開發者可以把此 annotation 當做 modifier，放在原來的 Java 程式中可以放置 modifier 的位置，比如 class、method 等宣告之前。

Annotation 的重點在於不只存在於原始程式碼，編譯時我們還可以把它保存在 class 檔中，甚至可以在執行時被使用到。APT[9]是一個提供程式開發者自行處理 Java annotation 的工具，程式開發者可以利用 APT 來設計對 annotation 處理的程序，這個程序將會在程式碼編譯前進行。

2.2 Java 編譯器

Java 編譯器的輸出是 bytecode，而不像其他編譯器產生組合語言碼或機器碼。bytecode 必須由 Java 虛擬機器所讀取，然後根據 bytecode 的指令執行。這樣的架構下最大的好處是程式具有可攜性，同樣的一份 bytecode，只要機器平台上有安裝 Java 虛擬機器，那 bytecode 就可以被執行。要完成上述的目標，Java 編譯器扮演了一個重要的角色，它各個階段如圖 2-1 所示，Scan 和 parse 階段將原始程式碼的輸入成為字元串流，並判斷是否有存在不合乎 Java program 的字彙，然後再判斷文法結構的正確，並且將原始檔案的資訊轉換成樹狀結構，此結構通稱為 IR Tree，後面的所有階段將會走訪 IR Tree。EnterTree 階段會簡單的分析 IR Tree 的結構，並針對 Class 和 Method 產生所需要的 symbol 資訊。Semantic check 階段將會在 IR Tree 上檢查語法的正確性，然後產生各個 Tree 所需要的資訊。然後再經過 Flow Analysis 和 desuger 兩個階段後，藉由 code generator 產生 bytecode(*.class)。

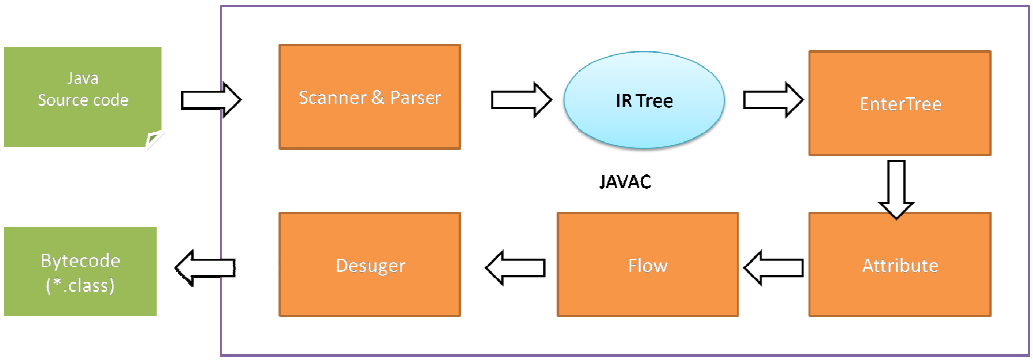


圖 2-1: Java Compiler 流程圖

2.3 NetBeans IDE

NetBeans[5]是由 Sun Microsystems 建立的開放原始程式碼的軟體開發工具，而且具有開放框架、可擴充功能的開發平台，可用於對 Java、C、C++等語言的開發。在 NetBeans 當中，應用軟體是用一系列的模組(modular software components)[10]所建構，UML 的功能便是其中一種模組。

NetBeans UML 提供圖形化的使用者介面(如圖 2-2)，使用者在編輯 UML 是以操作圖形的方式進行，有如繪圖一般的進行。使用者可對 UML 中的每個元件，如物件、關係等等，進行更深入的設定。

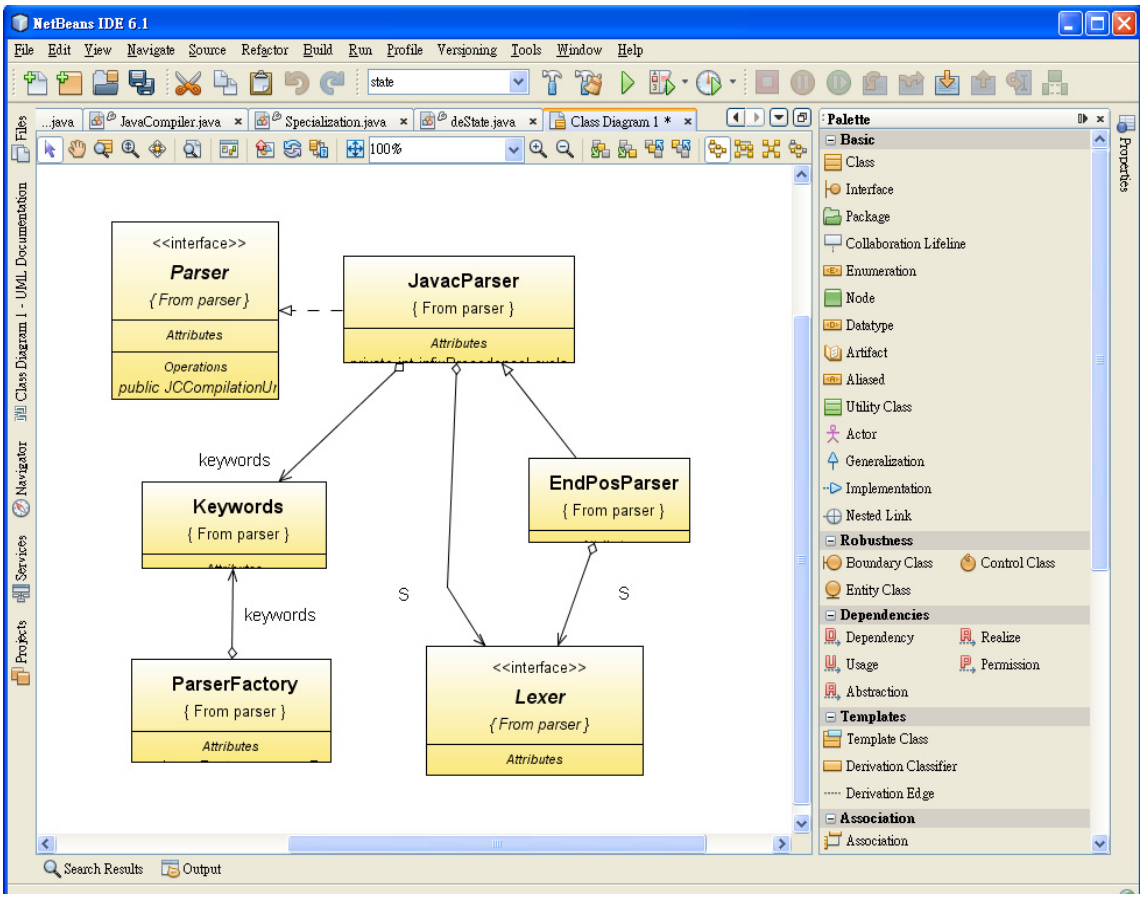


圖 2-2 : NetBeans UML 設計介面

NetBeans UML 還提供使用者直接引入一套 design pattern 的功能。在使用者進行 design pattern 的選擇與設定之後，便會在 class diagram 中產生一套 design pattern UML，讓使用者省去對照著 GOF 的設計，快速繪製出 design pattern 的 UML 圖形。

NetBeans UML 最後可以產生只有宣告和定義的框架程式碼，可將程式碼生成於 NetBeans 正在開啟程式專案，產生的程式碼會實現 UML 中編輯的關係，像是 class 之間的 inheritance、association、aggregation，class 對於 interface 的實作等等。

2.4 NetBeans 內部設計

2.4.1 UML 資料結構

NetBeans UML 雖然是用圖型的介面呈現，但是 NetBeans 的內部實作上，UML 的資料結構是以 XML 文件進行紀錄。UML 中的任何一個物件與關係，都將由一個 XML node 所記錄，包含當中的進階資訊。使用者所做的任何圖形編輯，都在 NetBeans 的模組功能下轉變為對 XML 文件的修改。紀錄 UML 的 XML 樹狀結構大致可以如下定義：

<UML:資料類型 資料資訊=??>

除了 NetBeans 自行定義的資料類型，資料類型大多是 UML 圖形中可以看到的元素。表 2-2 列出了我們所知的資料類型，其中包含了一般 UML 定義的元件[4]和 NetBeans 自身的專案資料。資料資訊(表 2-3)則是用來描述資料類型的資訊。

用於 UML 元件的資料類型	
Interface	紀錄 Interface 的資訊
Class	紀錄 Class 的資訊
Enumeration	紀錄 enumeration 的資訊
Operation	紀錄 Method 的資訊
Parameter	紀錄 parameter 的資訊
Attribute	紀錄 Field 的資訊
Package	紀錄 Package 的資訊
用於 UML 關係的資料類型	
Implementation	紀錄 Class impalements Interface 的資訊
Aggregation	紀錄 Aggregation 的資訊
Association	紀錄 Association 的資訊
Dependency	紀錄 Interface Dependency Class 的資訊
Comment	紀錄 Comment 的資訊
Generalization	紀錄 Class extends Class 的資訊
NetBeans 自定義的資料類型	
Project	UML 所屬的 project 資訊
Element	UML 中元件有包含的特性，則會用 Element 將之涵蓋
TaggedValue	紀錄 UML 編輯上的額外資訊

表 2-2 : UML node 資料類型

UML node 資料資訊	
xmi.id	表示元件在 UML 中的獨立 id
name	表示此類型的名稱
owner	表示包含此元件的元件 id
isAbstract	表示此元件是否 abstract
visibility	紀錄權限等級
資料資訊種類因太過繁複，因此只列舉部分，	

表 2-3 : UML node 資料資訊

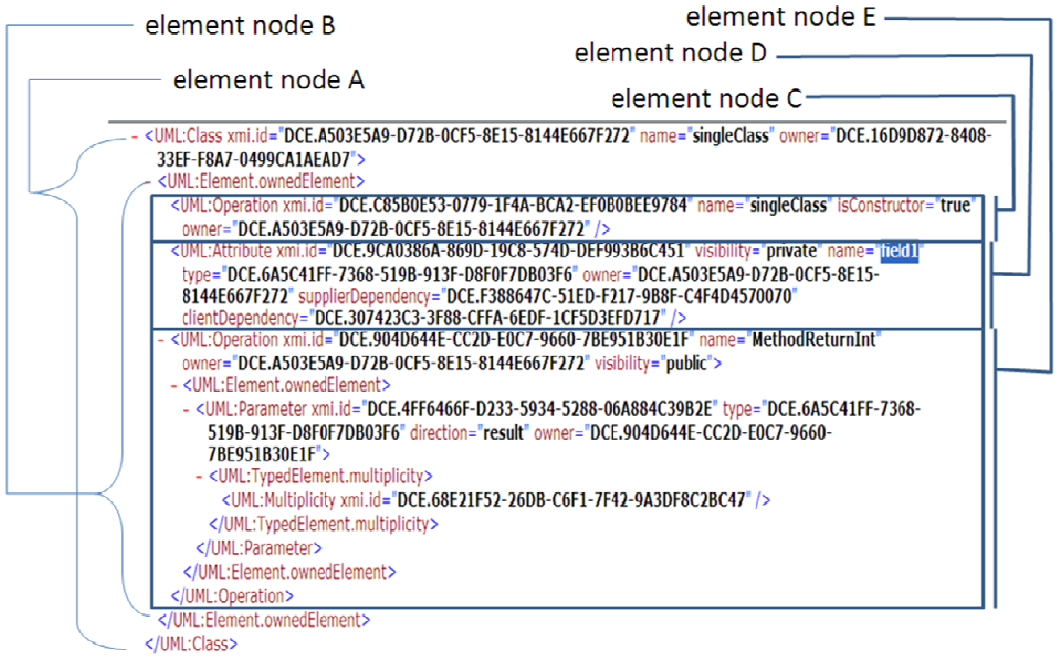


圖 2-3 : UML 內部資料實例

圖 2-3 所示的是一個 UML class diagram 的 XML 表示法，圖中最外層的 XML element A 的 XML attribute 記錄著該 class 的相關資訊，如名稱是 singleClass。element A 底下的 sub element (即 XML element B) <UML:Element.ownedElement>，這個 element B 的 sub element (圖 2-3 中的 element C、D、E)記錄著此 class 所具有的成員，如圖 2-3 所示，singleClass 具有兩個 operation 和一個 attribute，分別是 singleClass、

MethodReturnInt 和 field1。element C , D , E 如同 element A，它們的 attribute 記錄著本身的資訊。

NetBeans UML 模組在管理 UML 元件時有特定的格式，UML 元件(參考表 2-2)若包含其它元件資訊，便集中管理於 XML element <UML:Element.ownedElement>。若這個 element 不存在，便會建構它，在存入此 element。圖 2-3 的 element A 和 E 便是例子，它包含其它的元件資訊，因此這兩個 node 都有<UML:Element.owned-Element>此 element node 來儲存更深入的元件資訊。

NetBeans UML 模組還具有一個重要的功能，就是提供使用者直接引入一套 design pattern 的 UML diagram。這項功能在使用者輸入要引用的 pattern 名稱後，可直接建造出該 pattern 的 diagram，而省去使用者自行編輯 design pattern diagram 的步驟。事實上 design pattern 的 XML 是一個外部的檔案，NetBeans 再引入使用者所選取的 design pattern diagram，此引入的動作即是從檔案取一份拷貝。

2.4.2 生成原始程式碼

NetBeans UML 提供了將 UML diagram 轉變成程式碼的功能。使用者在設計完 UML 之後，可以選擇將 UML diagram 產生成框架程式碼，框架程式碼是一份只有架構的程式碼，程式中的邏輯和方法都尚未實做。接著 NetBeans IDE 會將程式碼產生在使用者所選擇的專案當中。

使用者選擇了產生程式碼的選項後，NetBeans 會呼叫 code generate 模組的程序。在這個程序當中會以 UML 中的 class，interface 和 enumeration 為一個檔案單位，產生相對應的 Java 檔案。產生程式碼的過程中，code generate 的程序會讀取 UML diagram 的內容，也就是 XML 文件，將一個 class(或者是 interface、enumeration)的資訊進行整合，然後產生一個 ClassInfo 的物件。ClassInfo 是一個用於描述 Java 類別的資料類別(metaclass)，ClassInfo 主要功能在於傳遞 NetBeans 的模組之間所需要的類別資訊。ClassInfo 的成員由的類別內容可以得知，此類別描述與紀錄 UML

當中一個 class 的完整資訊。

ClassInfo 物件除了用於紀錄類別資訊，UML 中的 operation、attribute 等其他相關元件都有相關的類別來紀錄它們的資訊，如 MethodInfo 類別和 MemberInfo 類別。而這些類別的物件都會被 ClassInfo 物件所參照。

ClassInfo 物件雖然記錄了 Java 檔案的大部分內容，但 Java 檔案的內容並不是由 ClassInfo 所產生。code generate 程序在完成了 ClassInfo 物件之後，會將物件傳給樣板引擎：FreeMarker([13])。FreeMarker 是一個會根據樣板產生文件的工具，而 FreeMarker 所引用的樣板文件(表 2-4)可從 NetBeans 的 templete tool 中編輯。FreeMarker 利用樣板文件和 ClassInfo 物件的資訊產生相對應的 Java 檔案。

```
<#--
  NormalClassDeclaration
-->
<#import "DeclLib.ftl" as lib
/>
<#macro NormalClassDeclaration classInfo nestingLevel
>
<@lib.TypeDeclarationComment classInfo nestingLevel
/>
<@lib.ident nestingLevel /><@lib.compress single_line=true >
<@lib.TypeDeclarationModifiers classInfo /> class ${classInfo.getShortClassName()}
<@lib.TypeDeclarationTypeParameters classInfo /> <@lib.NormalClassExtends classInfo />
<@lib.ClassImplements classInfo /></@lib.compress> {
<@lib.ClassBodyDeclaration classInfo nestingLevel+1 />
<@lib.ident nestingLevel />}
</#macro>
```

表 2-4 : class 樣板文件內容

3. Design Pattern 資訊儲存與轉換

從 NetBeans UML 設計開始，為了將 design pattern 的資訊利用 Java annotation 記錄下來，我們必須讓使用者從 UML 設計階段就要進行資訊的紀錄。在這個章節，我們將探討如何在 NetBeans UML 設計上加入 design pattern 訊息。我們對 XML 文件加入新的 XML elemen 和 XML attribute，用此儲存 design pattern 資訊。然後針對 NetBeans 編輯 UML 的功能進行修改，讓使用者可以對新的 element 和 attribute 進行編輯。最後，在產生 code 的階段，我們增加了 ClassInfo 的 annotation 欄位，並加入從 XML element 和 XML attribute 獲取資訊的程序。FreeMarker 參照的樣板也必須在適當的位置加入獲取 annotation 的方法。

3.1 Design Pattern 資訊

我們試圖在 UML 設計階段儲存 design pattern 資訊，在此我們將介紹這個資訊意義。圖 3-1 是 GOF 的 state pattern 設計，此圖用 UML 表達出設計概念。類別 Context 的方法 Request 使用 State 的變數呼叫方法 Handle，以及 State 使用 abstract 類別(或 interface)需要實作。這樣的設計方式不論是類別 Context、State 或方法 Request、Handle 都扮演特定的角色。

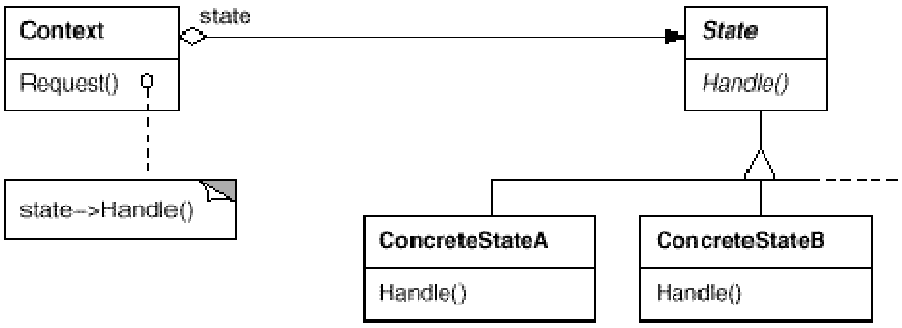


圖 3-1 : GOF state pattern diagram

程式開發者在設計 UML 的同時，可以引用 GOF design pattern 的設計概念，

在 UML 當中運用 state pattern 的概念。程式開發者雖然應用 design pattern 的概念，但是在類別與方法的命名上，還是取決於程式開發者設計上的想法，因此 UML 設計完成後我們很難從它的名稱和架構來判斷它所使用的 design pattern。圖 3-2 是命名過後的 state pattern，除了 interface 的名稱有 state 的文字以外，其它的 GOF 命名都不存在於設計中。

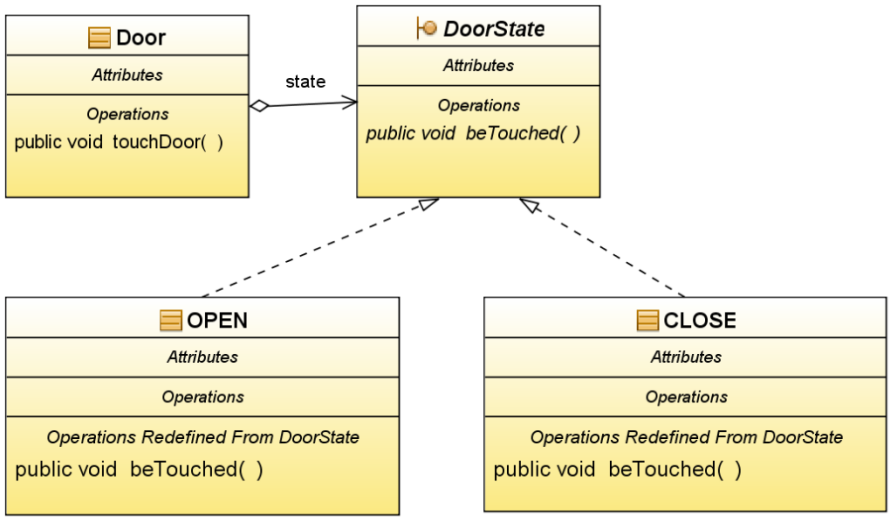


圖 3-2：經過命名後的 state pattern

我們所希望保存的 design pattern 資訊，便是引用 design pattern 時各個元件所代表 design pattern 的角色。以圖 3-2 的案例來說，類別 Door 是原先 state pattern 的類別 Context、interface DoorState 是原先 state pattern 的 abstrct 類別 State 而類別 OPEN 和 CLOSE 則是原先 state pattern 的類別 ConcreteState；同樣的方法(method)亦可類推。

我們將在後續介紹如何在 NetBeans UML 中儲存 design pattern 資訊。我們所要保存的資料是能夠讓程式到產品化階段仍能夠判別出 interface DoorState 是 state pattern 設計中的 abstrct 類別 State。

3.2 Design Pattern 資訊儲存

NetBeans UML 的內部資料結構是一個 XML 文件，根據 design pattern 的設計，

每個 interfaces、classes、methods 和 variables 都可能在 design pattern 中扮演一個角色。因此在 XML 文件當中每個描述 interfaces、classes、methods 和 variables 的 xml node 都要能夠追加描述 design pattern 資訊的欄位。我們選擇利用 NetBeans 現有的 XML element <UML:TaggedValue>，然後利用這個 node 來紀錄 design pattern 的資訊。按照 2.4.1 的介紹，element <UML:TaggedValue>是 element <UML:Element.ownedElement>的一個 sub element。

紀錄 design pattern 資訊的 XML element 如表 3-1 定義之，其中的黑體字是我們要填入的資訊。PatternType 可以被使用者編輯，但它的內容在後續規定，Pattern-Info 則任由使用者編輯，它的內容將會是 design pattern 的資訊。

```
-<UML:TaggedValue xmi.id="idformIDE" owner=" idformIDE " name="PatternType">
    <UML:TaggedValue.dataValue>PatternInfo</UML:TaggedValue.dataValue>
 </UML:TaggedValue>
```

表 3-1：紀錄 design pattern 資訊的 XML element

PatternType 所表達的 UML 元件	PatternType 的內容
Interface、Class or Enumeration	patternClass
Operation	patternOperation
Attribute	pattern Attribute

表 3-2：對應 UML 元件類型的 PatternType 內容

xmi.id 和 owner 將由 NetBeans 自行創建，並且內容由 NetBeans 負責管理，而 name 這個 attribute 用來紀錄 PatternType，PatternType 是此 XML element 所描述的元件類型，PatternType 的內容如表 3-2 的方式定義。PatternInfo 是 node 的內容，用來紀錄此元件的 design pattern 資訊。以圖 3-2 的類別 OPEN 為例子，它的 PatternType 的內容將會是 “patternClass”，而 PatternInfo 的內容則為 “ConcreteState”。PatternType 的內容取決於此 element 所描述的元件類型，由表 3-2

定義；而 PatternInfo 的內容將表達此元件在 design pattern 中扮演的角色，按照圖 3-1 與圖 3-2 的對應，PatternInfo 的內容為”ConcreteState”，因為類別 OPEN 代表的是 ConcreteState，它必須實做 abstrct 類別 State。表 3-3 將描述類別 OPEN 在 UML 中所帶有的 design pattern 資訊，這是利用<UML:TaggedValue>來記錄資訊。

```
-<UML:TaggedValue xmi.id="idformIDE" owner=" idformIDE " name=" patternClass ">
        <UML:TaggedValue.dataValue> ConcreteState </UML:TaggedValue.dataValue>
 </UML:TaggedValue>
```

表 3-3：類別 OPEN 的 design pattern 資訊

利用 <UML:TaggedValue> 紀錄 design pattern 資訊的主要原因是 <UML:TaggedValue>這個 node 是 NetBeans 本身提供給使用者編輯的欄位。使用 <UML:TaggedValue>紀錄 design pattern 資訊的好處是使用者可以利用 NetBeans 現有的介面直接進行編輯，而省去我們修改 NetBeans UI 的麻煩。另一個好處是因為 <UML:TaggedValue>可以有兩個以上，依照程式開發者設計的不同，同樣的 UML 元件可能同時扮演著 design pattern 中的兩個角色；另一個重要的原因是，UML 在生成 source code 時，我們將利用 java annotation 來記錄此資訊，同樣的兩個以上的 java annotation 可以同時描述於一個類別(或其他元件)。根據 UML 設計上的要求與 java annotation 的特性，<UML:TaggedValue>符合我們的需求，因此利用它來記錄 design pattern 資訊。

上述是利用 element <UML:TaggedValue>來記錄 design pattern 資訊。因為 <UML:TaggedValue>是一個可以編輯的欄位，為了避免使用者的錯誤編輯，我們另外設計了一個 XML attribute 來記錄 design pattern 資訊，而這個 node 將無法被使用者所編輯。

紀錄 annotation 的 XML attribute 如表 3-4 定義之，其中的黑體字是我們要填入的內容，但使用者並不能編輯。

<UML:(*UML 元件*) xmi.id=" *idformIDE* " name="*editByUser*" isAbstract=" *editByUser* " owner="*idformIDE*" **PatternType** =" **PatternInfo** ">

表 3-4： 紀錄 design pattern 資訊的 XML attribute

PatternType 與 PatternInfo 的內容與 element(表 3-1)的定義相同，差別在於 PatternType 將轉為 attribute 的名稱，PatternInfo 將會是 attribute 的內容。同樣的對照於圖 3-1 與圖 3-2，我們挑選 OPEN 類別的方法 beTouched 做為例子，它的 XML node 內容將會如表 3-5。PatternType 為“PatternOperation”用來代表這是一個 operation，接著 PatternInfo 為 “Handle”用來記錄它所扮演的角色。

<UML:Opration xmi.id=" *idformIDE* " name="beTouched" isAbstract="*no*" owner="*idformIDE*" PatternOperation =" Handle ">

表 3-5： 方法 beTouched 的 XML 內容與 design pattern 資訊

在往後的章節，不論是利用 XML element 或是 XML attribute，都將用來記錄 design pattern 的資訊，因此我們稱這些 node 為 pattern node。

3.3 將 Design Pattern 加入 NetBeans

在定義完成 Netbeans UML 如何紀錄 design pattern 資訊之後，我們必須將使用者引入 design pattern 的 UML 加入 pattern node 並且按照 GOF design pattern 的名稱去定義他。雖然引入 design pattern UML 是藉由外部的 XML 檔案。但我們不傾向修改外部檔案，原因是加入 pattern node 的行為實際上非常簡便，而且加入的資訊可從 XML 的檔案中獲取，因此沒有必要修改大量的外部檔案。

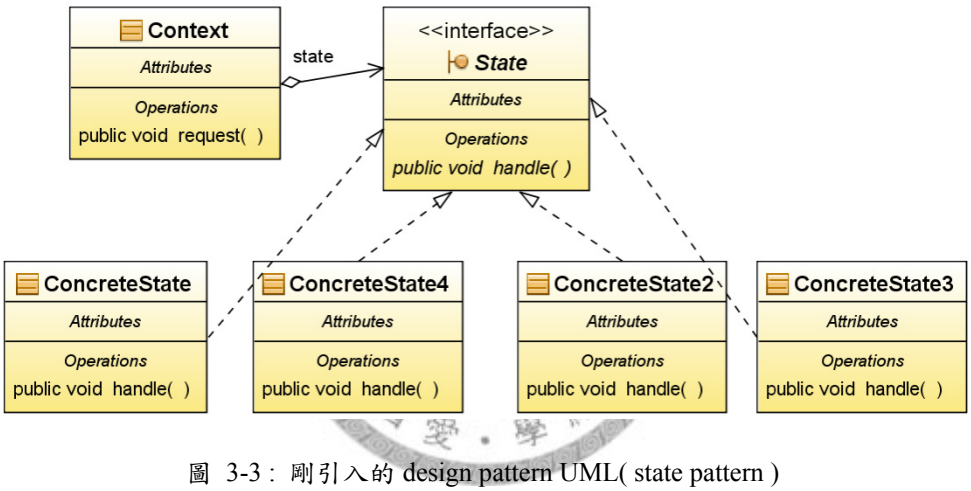


圖 3-3：剛引入的 design pattern UML(state pattern)

圖 3-3 為一個新引入的 design pattern UML，由圖可知它的 Classes、methods、variables 名稱都是取自於 GOF design pattern 的設計。而我們打算加入的 design pattern 資訊正是 GOF design pattern 最原始的名稱。因此我們輪巡所有元件，加上 pattern node，而 pattern node 的內容則取自該元件的名稱，因為這些名稱正是 design pattern 資訊。引用 design pattern 是使用者本身的動作，因此我們認定引入的 UML diagram 將會是 design pattern 的元件。我們將會利用 attribute 的方式來記錄 design pattern 資訊，避免使用者不慎的修改。

UML 中的元件是否具有 pattern node 是決定使用者是否使用 design pattern 的

唯一依據。使用者為 UML 建構 pattern node 的方式有兩種，其一是引入 design pattern，我們的修改自動的為 UML 加入 pattern node，其二是使用者自行藉由 taggedValue 加入 pattern node。假設 UML 當中不存在任何的 pattern node，那我們將認定此 UML 不使用 design pattern 來進行設計，往後的編輯也不會詢問使用者是否要加入 pattern node。即使使用者自行仿造 GOF design pattern 繪製 UML 也是一樣，之後的編輯和產生框架程式碼將不會連帶產生表示 design pattern 資訊的 annotation， design pattern 的資訊亦不會保留。

3.4 NetBeans UML 編輯的修改

我們將 design pattern 的概念加入 design pattern UML 當中，使用者運用 NetBeans UML 來引用 design pattern 時，我們將為之加入 patern node，而後續使用者的編輯動作也與 design pattern 相關。NetBeans 本身的 UML 編輯設計功能十分完善，因此我們對所有編輯功能做了以下修改：針對具有 pattern node 的元件進行編輯，會進行是否加入 design pattern 資訊的詢問。詢問的內容是選擇 design pattern 資訊的名稱，使用者必須選擇適當的名稱，來表示使用者加入的元件將會扮演 design pattern 中的角色。

例如 design pattern 中的 state pattern，使用者可能會增加一個方法來操作 state 的操控，這個方法必須藉由使用者在 UML 當中新增，而新增的過程會有選項給予使用者選擇的 design pattern 角色。為了特殊化進行的正確性，使用者必須選擇正確的 design pattern 角色，來表達這個新增的方法在 design pattern 當中代表的涵義。

根據特殊化的要求越多，特殊化所需的資訊也會增加，加上使用者可能有獨特的程式設計模式，因此 GOF design pattern 基本的資訊可能不夠達到特殊化的要

求。UML 設計階段所要加入的 design pattern 資訊將不僅止於 GOF design pattern 所介紹的，因此額外的 design pattern 資訊是必要的。為了提升 design pattern 資訊的名稱擴充性，我們將 design pattern 資訊的內容利用外部檔案定義。以 state pattern 為例，外部檔案的定義的格式如表 3-6。

Name	**State**
Interface	State
Method	Handle
Field	Declaration
Class	Context
Method	request
Class	ConcreteState
Method	Handle

表 3-6：外部定義的 design pattern 資訊

表 3-6 是從外部檔案中擷取的 state pattern 部分，Name 欄位記錄著此 design pattern 的名稱，之後每個欄位根據元件類型的不同而具有後述 design pattern 資訊。如要在一個 state pattern 的 state interface 的 UML 圖形之中加入一個 method 的欄位，則會有 “handle” 這個 design pattern 角色可以選擇。同理打算在 ConcreteState 的類別當中加入 method 亦會有 “handle”這個 design pattern 角色。表格的內容是以樹狀結構解析，因此表格實質上可看作如表 3-7 的樹狀結構。

Name	State		
+	**Interface**	State	
	+	**Method**	handle
	+	**Field**	Declaration
+	**Class**	Context	
	+	**Method**	request
+	**Class**	ConcreteState	
	+	**Method**	handle

表 3-7：以樹狀結構表現的 annotation 選項

在 state interface 的 UML 圖形底下新增 method 可選擇 “handle”的角色，新增 field 可選擇 “Declaration”的角色，而在 ConcreteState Class 的 UML 圖形底下新增 field 則不能夠選擇 “Declaration”的角色。

Design pattern 資訊的表格利用.csv 的文件格式儲存(.csv 在同列但不同欄位間會用 “，”做區隔，不同列則用換行做區隔)。使用者可以進行修改來獲得相對應的 design pattern 角色的選擇。假設將表格修改成表 3-8，修改前的表格在 Concrete-State Class 的 UML 圖形底下新增 Method 則會產生圖 3-4 的詢問視窗，修改後則會產生圖 3-5 的詢問視窗。

圖 3-4：根據表 3-6 所產生的詢問

圖 3-5：根據表 3-8 所產生的詢問

Name	State
Interface	State
Method	Handle
Field	Declaration
Class	Context
Method	Request
Class	ConcreteState
Method	Handle
Method	New
Field	newF

表 3-8：重新編輯後的 annotation 選項

我們多了 NoComment 選項代表不賦予新增元件 design pattern 的角色。我們以選項的方式讓使用者加入 annotation 內容，使用者可直接編輯表格，來獲得需要的 annotation 內容。

3.5 生成含 Annotation 程式碼

藉由上述的修改，UML 在使用者的編輯下應能正確的包含 design pattern 資訊。為了產生包含 annotation 的框架程式碼，我們將再修改 Netbeans 的兩個部分，分別是 ClassInfo 產生的流程能夠擷取 pattern node 的 design pattern 資訊、以及 FreeMarker 所參考的樣板文件必須加入 annotation 的欄位。ClassInfo 的資訊將用於產生 java source code，因此他必須有記錄 design pattern 資訊的欄位和記錄 annotation 名稱的欄位。因為我們的設計是利用 annotation 來儲存與表示 design

pattern 資訊，兩者將會是同樣的內容，因此 ClassInfo 當中僅有記錄 annotation 的欄位，而此欄位在我們的設計下僅用於記錄 design pattern 資訊。

3.5.1 ClassInfo 的修改

ClassInfo 是在 generate code 模組的流程中產生的一個物件，NetBeans 會針對每個類別或 interface 的 UML 將會產生一個相對應的 ClassInfo 物件，ClassInfo 紀錄類別資訊的方式很簡易，僅利用 String 儲存。之前的 UML 編輯已經能夠描述夠多設計資訊，ClassInfo 主要的功能是做一個整合，讓 FreeMarker 能夠快速的取得名稱的訊息。

我們的目的在於產生 annotation 的內容，因此在 ClassInfo 建立一個紀錄 annotation 名稱的 String 欄位。在產生 ClassInfo 的過程，我們必須加入一個分析 XML pattern node 的流程，這個流程將會從 UML 的 pattern node 當中獲得 design pattern 資訊的內容。同樣的步驟也會套用在產生 MethodInfo 與 MemberInfo 的流程中。因為 FreeMarker 不能直接參照 field，必須加入回傳 annotation 內容的方法，以供 Freemarker 產生 source code 時，填入 annotation 名稱。

我們雖然可以藉由 annotation 套件的設計，來分辨 annotation 與它所描述的元件所屬的 design pattern。但為了處理上的迅速，我們在 ClassInfo 的 annotation 加上了 design pattern 名稱。實際的 annotation 內容將會如下：

- **@ “design pattern 名稱”_ “annotation 名稱”**

實際的案例如表 3-11 的程式碼所示，該介面(interface)是 state pattern 中的 state 角色，因此所顯示的 annotation 為 “@State_State”。而這項修改將不會套用於 MethodInfo 與 MemberInfo 的流程中，因為 method 和 field 可以靠類別或介面來分辨 pattern 的類型。

3.5.2 FreeMarker 樣板文件的修改

FreeMarker 的樣板文件可以直接使用 ClassInfo 物件所具有的方法。Java annotation 是標記在欲描述的 classes、interfaces、methods、variables 的前頭，因此我們追尋了樣板文件中產生 classes、interfaces、methods、variables 內文的部分，在這之前加上取得 annotation 名稱的方法(如表 3-6)。而想要在 FreeMarker 樣板文件中要 ClassInfo 的方法，必須用${method}的字串來引用。我們將呼叫的方法放在正確的位置，利用這樣的修改來使 FreeMarker 產生 Java source code 能夠包含 annotation。

```
<#--
  NormalClassDeclaration
-->
<#import "DeclLib.ftl" as lib
/>
<#macro NormalClassDeclaration classInfo nestingLevel
>
<@lib.TypeDeclarationComment classInfo nestingLevel
/>
${classInfo.getImportName()}
${classInfo.getPatternName()}
<@lib.ident nestingLevel /><@lib.compress single_line=true >
<@lib.TypeDeclarationModifiers classInfo /> class ${classInfo.getShortClassName()}
<@lib.TypeDeclarationTypeParameters classInfo /> <@lib.NormalClassExtends classInfo />
<@lib.ClassImplements classInfo /></@lib.compress> {
<@lib.ClassBodyDeclaration classInfo nestingLevel+1 />
<@lib.ident nestingLevel />}
</#macro>
```

表 3-9：修改後的 class 樣板內文

3.5.3 Annotation 套件和宣告

Java annotation 在文法的使用上，必須先經過宣告和引用，才能夠利用 annotation 來描述一個元件。因為宣告和引用方法在格式上固定，所以我們在 generate code 模組的流程中加入自動創建宣告和引用的功能。

在 generate code 模組的流程，我們可以藉由 ClassInfo 的資訊得知打算產生的 annotation 內容，因此我們利用建立檔案的方法產生 annotation 的宣告文件(參見表 3-10)，並且利用 package 來管理這些宣告文件，我們定義以下的名稱：

- ***NTU.Annotation.DesignPattern. "pattern 的名稱";***
- ***NTU.Annotation.DesignPattern. "pattern 的名稱".method;***
- ***NTU.Annotation.DesignPattern. "pattern 的名稱".field;***

這些套件來儲存相對應的 annotation 定義文件(圖 3-6 是實際範例)。我們必須利用這樣多層次的方式來儲存 annotation 定義文件，來解決 design pattern 定義的內容名稱重複的情形；利用套件名稱來獲得更詳細 design pattern 元件內容，並且利用自動產生檔案的方法來省去使用者負擔。

圖 3-6 : annotation 定義所存放的 package

```
package NTU.Annotation.DesignPattern.State;
import java.lang.annotation.*;
/* auto generate code */
@Retention(value = RetentionPolicy.SOURCE)
public @interface State_State {}
```

表 3-10 : annotation 的宣告文件

我們再產生的 Java 框架程式碼使用 annotation 描述元件，為了編譯的正確必

須引用 annotation 的宣告，這樣的宣告內容同樣的在 generate code 模組的流程自動產生。我們擷取一份 source code 當中使用到的 annotation 內容，將引用的程式語言加入到 ClassInfo 的欄位，因此同樣的利用 3.5.1 的方式修改 ClassInfo 的內容和 3.5.2 的方式修改 FreeMarker 樣板文件，便可在 Java 框架程式碼上加入引用 annotation 的宣告。

3.6 範例

這個小節我們展示修改過後的 NetBeans IDE 從 UML 設計到產生程式碼的流程。首先，我們從圖 3-7 所示的引用 design pattern 開始進行 UML 的設計，如同 3.3 所說的，不使用這個引用，則使用者必須自行加入 Pattern Node 並填入 design pattern 資訊，這樣之後的編輯才會有選項的介面。

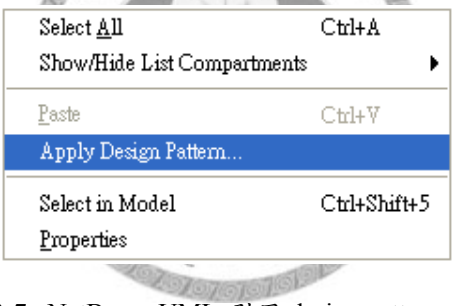


圖 3-7 : NetBeans UML 引用 design pattern 選項

再透過圖 3-8 和圖 3-9 的 design pattern wizard 選擇 design pattern 預設的 UML 圖之後便可產生包含 pattern node 的 design pattern UML，如圖 3-10 所表示。

再引入 design pattern UML 之後，使用者將可運行編輯 UML 的工作。當使用者在具有 pattern node 的類別(或 interface)上打算新建一個 operation，則會有詢問此 operation 是否要成為一個具有 pattern node 的 operation(圖 3-11)，選擇 yes 則會有圖 3-4 的進一步選項，選擇 no 則是 NetBeans 增加一個 operation 的程序。

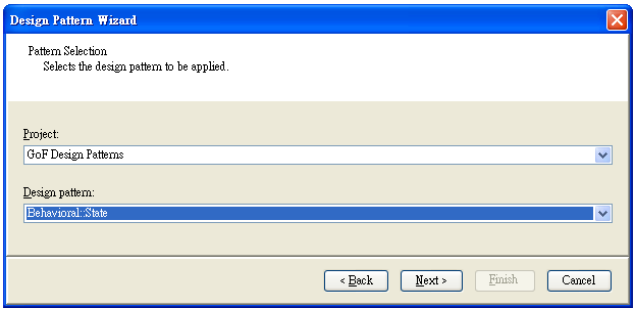


圖 3-8：引用 Design pattern 的 design pattern wizard，此處用於選擇 state pattern

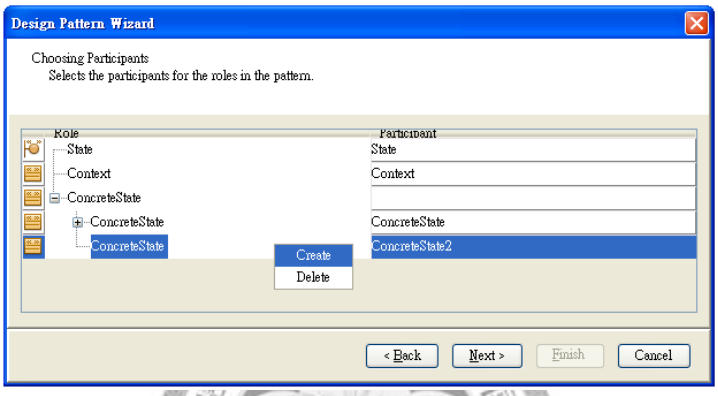


圖 3-9：此處可以調整初始 design pattern 的 UML 圖形所具有的 ConcreteState class 數量

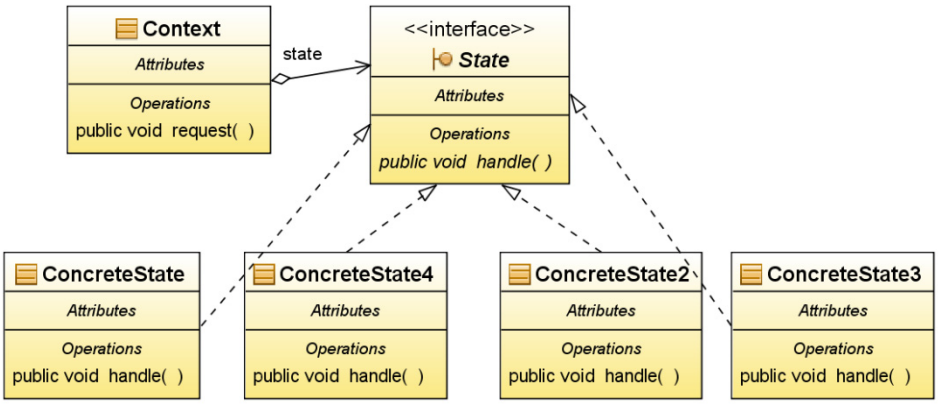


圖 3-10：初建置的 UML diagram (state pattern)

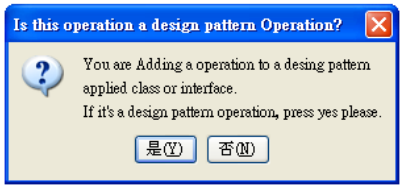


圖 3-11：詢問是否建構具有 pattern node 的 operation

在經過編輯之後，我們假設編輯過後的 UML 如圖 3-12 所顯示， NetBeans UML 當中 tagged value 的內容將會緊接著 operation 和 attribute 的表示之後已{tagged value}的方式顯是在 UML 圖形中(圖 3-13)。因為我們將 pattern node 編輯在 tagged value 當中，因此{tagged value}所顯示的便是 design pattern 資訊(同時也是 annotation 的名稱)。

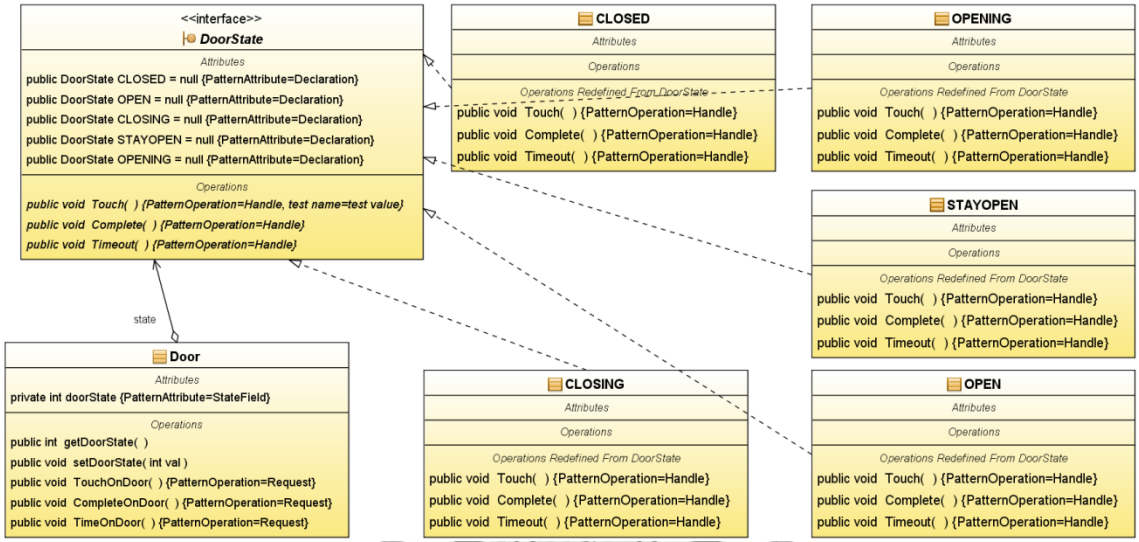


圖 3-12：編輯完成的 UML (state pattern)

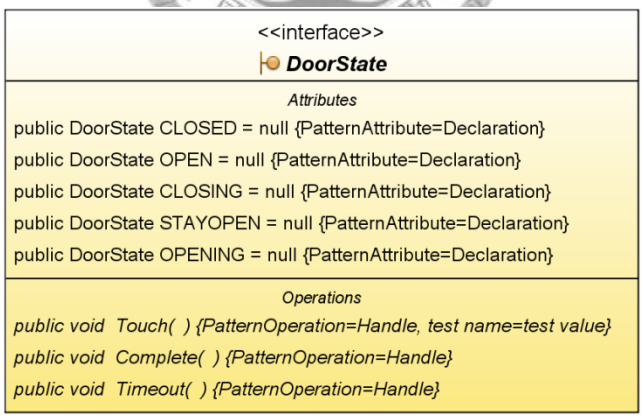


圖 3-13：圖 3-12 的 DoorState

之後我們進行 generate code 模組的程序，照著 NetBeans 產生 source code 的流程，選擇要輸出的專案之後，程式碼便會產生，表 3-11 的程式碼便是根據圖 3-13 中 DoorState 這個 interface 元件所產生。從表 3-11 的程式碼可以看到，每個加註

pattern node 的元件在程式碼中都有 annotation 的描述，annotation 的名稱正是 design pattern 資訊，而這也是我們所要的結果，表 3-11 的程式碼開頭 1~4 行是引用 annotation 的程式語言，這段 statement 也是在 generate code 的過程產生，同時存放 annotation 定義的 package 資料夾以及 annotation 宣告文件(表 3-10)也會在 generate 的過程產生。

```
import NTU.Annotation.DesignPattern.State.Field.Declaration;
import NTU.Annotation.DesignPattern.State.Method.Handle;
import NTU.Annotation.DesignPattern.State.State_State;
@State_State
public interface DoorState {
	@Declaration
	public static final DoorState CLOSED =null;
	@Declaration
	public static final DoorState OPEN = null;
	@Declaration
	public static final DoorState CLOSING = null;
	@Declaration
	public static final DoorState STAYOPEN = null;
	@Declaration
	public static final DoorState OPENING = null;

	@Handle
	public void Touch (Door door);
	@Handle
	public void Complete (Door door);
	@Handle
	public void Timeout (Door door);
}
```

表 3-11：從 UML 所產生的 Java code(對應圖 3-13 的元件)

4. Design Pattern 特殊化

Design patter 有著提高程式維護性，但降低執行效率的特色。而在產品化階段的程式碼，已經不再需要考慮程式維護的需求。因此 Design pattern 特殊化(specialization)目的是將程式中的 design pattern 架構消除，藉此達到效率的提升。特殊化的行為是高階語言的修改，要達到特殊化我們必須了解程式的架構，進而修改程式內容。

在上一章，我們探討了如何在 UML 設計過程加入 annotation 的紀錄，並且將 design pattern 的訊息以 annotation 的形式記錄下來。在這一章，我們將假設程式開發者在符合 design pattern 的概念完成程式碼的編輯，並且保留程式中的 annotation。我們利用這些 annotation，來找到程式中各元件所扮演的 design pattern 角色。雖然我們可以參考 design pattern 的設計，研究出特殊化的方法，但是這個方法不一定適用於任何引用 design pattern 的程式。因此我們設計修改程式碼的命令，讓程式開發者根據自己的程式架構來編輯命令，再利用命令來修改程式碼。

我們將在後續討論利用命令設計來達到 design pattern 特殊化的優點，並且介紹命令的格式與使用。在介紹命令使用的段落，我們參考[11]和[12]的想法來進行程式特殊化。

4.1 特殊化程式架構

我們的目的是對於 design pattern 進行特殊化，這是一項對於高階語言的程式優化作業。我們將讀入原始程式碼，然後輸出特殊化後的程式碼，輸出的程式碼依舊是以高階語言呈現，因此我們將輸出的程式碼整合到 NetBeans 的專案當中，讓程式開發者可以在同一個介面下，瀏覽特殊化後的程式碼。特殊化的作業我們將

它加入到 Javac 的 pass 當中(如圖 4-1)，原因是 Javac 的 IR Tree 可以幫助特殊化的作業，並有助於產生排版後的程式碼，利於程式開發者瀏覽。

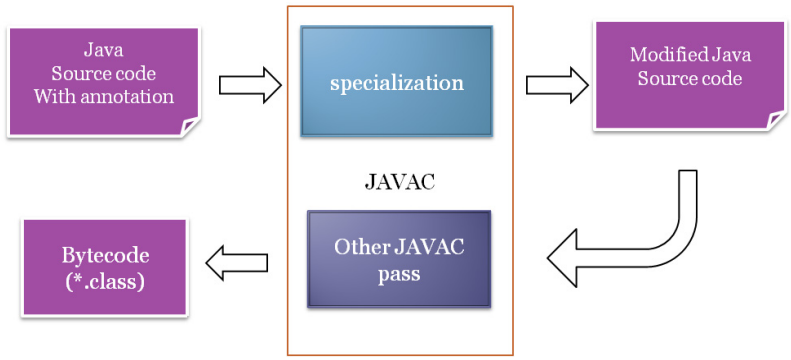


圖 4-1：建構於 Javac 內部的特殊化程序

我們不利用 Annotation Processing Tool (APT[14])來進行特殊化的作業，原因是因為 APT 的執行是以各個 annotation 獨立進行，而且 APT 執行的過程，我們無法觸碰到沒有 annotation 的程式，除非我們大費周章的去讀入所有程式碼。特殊化的進行是所有被 annotation 注記的程式碼交互進行，因此需要一個全面碰觸到原始程式碼的環境，所以我們選定在 Javac 的內部。

4.2 特殊化策略

對於一個引用 Design pattern 設計的程式，我們可以個別的對不同的 pattern 進行特殊化，進而達到全面特殊化的效果。因此我們原先的特殊化策略是利用 annotation 來分辨個別 pattern 的元件，對單一的 pattern 進行特殊化。這樣的策略會使得一個 pattern 需要一套特殊化方法以及程式，若特殊化的方法不適用於使用者的設計，那將必須修改方法或者設計另一套方法。雖然一個 pattern 一套方法的策略顯得較容易達成，但在於方法的擴充和修改上太過貧乏，因此我們決定利用命令設計的策略，讓使用者學習命令的使用與編輯，來創造適合自身程式設計的方法。

利用命令來設計特殊化的方法，除了能在方法上有更靈活的運用，同時命令本身是一個可分享的文件，若不同使用者之間引用的 design pattern 類似，某一個使用者設計的命令，有可能適用於其他使用者。因此在命令設計的策略下，我們可以設計較通用的命令，藉由命令文件的分享，讓入門的使用者不用理解命令的編輯與操作，便能夠達到 design pattern 特殊化的效果。

4.3 命令格式與介紹

4.3.1 敘述元件

首先我們先定義命令中使用的敘述元件，敘述元件是利用 annotation 來表示出指定的元件。根據之前的論述，我們使用 annotation 來表達程式碼中 design pattern 的角色，因此在命令當中，我們用 annotation 作為指定角色的敘述。使用 annotation 來表示元件有如表 4-1 的格式。

敘述句	內容
***@ClassAnnotation*(參數)**	指定被*@ClassAnnotation* 修飾的類別
***@ClassAnnotation*(參數)** ***@MethodorOrFieldAnotation*(參數)**	指定被*@MethodorOrFieldAnotation* 修飾的 method 或 field (必須在指定的類別內容中)

表 4-1：指定元件的敘述

因為同一個 annotation 可以修飾於多個元件，因此無法單靠 annotation 就指定出想要的元件，因此我們追加定義了參數，利用參數來更深入的指定同樣的 annotation 當中所要的元件。參數的定義如表 4-2，為了配合實作上的設計，我們

規定參數結尾必須加入符號「:」，來表示這是一個參數。

敘述元件參數	效果
ALL:	引入所有此 annotation 的元件
ORIGINAL:	與修改的目的元件相同的元件
NAME:	由程式開發者自行列舉所要的元件名稱
無參數	此 annotation 唯一存在，故不需指定

表 4-2：用於指定元件的變數

底下我們舉一個範例來說明，表 4-3 的程式碼是一個 interface DoorState，這個 interface 在特殊化的階段是眾多檔案中的一個，在命令中要指定此 interface 則在命令的敘述元件中使用 “@State_State()”。“@State_State()”將會指定 DoorState 這整個 interface 的程式碼內容，對於類別也是相同的操作。若要指定 interface 與類別當中的 field 或方法，則必須先指定 interface 或類別再指定之。表 4-4 是運用敘述元件來表達 DoorState(表 4-3) 的所有元件範例。

```
1  import NTU.Annotation.DesignPattern.State.Field.Declaration;
2  import NTU.Annotation.DesignPattern.State.Method.Handle;
3  import NTU.Annotation.DesignPattern.State.State_State;
4  @State_State
5  public interface DoorState {
6      @Declaration
7      public static final DoorState CLOSED = new CLOSED();
8      @Declaration
9      public static final DoorState OPENED = new OPENED();
10     @Handle
11     public void Touch (Door door);
12 }
```

表 4-3：範例程式碼 DoorState.Java

元件敘述	指定內容

@State_State()	@State_State
	public interface DoorState {
	@Declaration
	public static DoorState CLOSED = new CLOSED();
	@Declaration
	public static DoorState OPENED = new OPENED();
	@Handle
	public void Touch (Door door);
	}

@State_State()@Handle()	public void Touch (Door door);
@State_State()@Handle(ALL:)	
@State_State()- @Handle(NAME:Touch)	
@State_State()@Declaration()	ERROR：有兩個以上符合敘述，無法確認何者為所要求的元件。
@State_State()@Declaration(ALL:)	@Declaration public static DoorState CLOSED = new CLOSED();
	@Declaration public static DoorState OPENED = new OPENED();
@State_State()- @Declaration(NAME: CLOSED)	@Declaration public static DoorState CLOSED = new CLOSED();
@State_State()- @Declaration(NAME: OPENED)	@Declaration public static DoorState OPENED = new OPENED();
敘述中的“-”表示連續，如 (@State_State()- @Handle(NAME: OPENED) 等同於 @State_State()@Handle(NAME: OPENED)	

表 4-4：表 4-3 “DoorState”能指定的敘述元件

在後續章節我們介紹現有的命令，命令的效果是對於程式碼進行修改，被修改過後的程式碼並不會在特殊化的流程中檢查程式的正確性，但會在流程過後的 Javac 進行檢查，程式開發者可藉由特殊化流程所產生的特殊化後程式碼進行命令的校正。在介紹命令的過程所運用的敘述元件和元件所參考的程式碼如下：表 4-5、表 4-6、表 4-7 和表 4-8。

```
1  @State_Context
2  public class Door {
3      @StateField
4      private DoorState mDoorState = DoorState.CLOSED;
5
6      @Request
7      public void TouchOnDoor () {
8          mDoorState.Touch(this);
9      }
10     @NoComment
11     public void setDoorState (DoorState val) {
12         this.mDoorState = val;
13     }
14 }
```

表 4-5：範例程式碼 Door.Java

```
1  @State_State
2  public interface DoorState {
3      @Declaration
4      public static final DoorState CLOSED = new CLOSED();
5      @Declaration
6      public static final DoorState OPENED = new OPENED();
7
8      @Handle
9      public void Touch (Door door);
10     @Handle
11     public void Complete (Door door);
12 }
```

表 4-6：範例程式碼 DoorState.Java

```
1  @State_ConcreteState
2  public class CLOSED implements DoorState {
3      @Handle
4      public void Touch (Door door) {
5          door.setDoorState(DoorState .OPENED);
6      }
7  }
```

表 4-7：範例程式碼 CLOSED.Java

```
1  @State_ConcreteState
2  public class OPENED implements DoorState {
3      @Handle
4      public void Touch (Door door) {
5          door.setDoorState(DoorState .CLOSED);
6      }
7  }
```

表 4-8：範例程式碼 OPENED.Java

4.3.2 命令：Copy

這個段落我們介紹現有的命令，分別是“Copy”、“Delete”和“Modify-Convert-To”。我們列出它們的格式以及它們的使用範例和效果，使用範例中所運用的敘述元件和元件所參考的程式碼於表 4-5、表 4-6、表 4-7 和表 4-8。

命令 Copy 是一個用於複製元件的命令，程式開發者可以運用它來將程式中的元件進行複製。命令 Copy 的格式如表 4-9，此命令會將來源的元件複製到目的的元件。敘述元件的介紹可參考 4.3.1 節和表 4-4 的介紹。表 4-10 是命令 Copy 的範例，敘述元件“@State_State()@Declaration(ALL:)”所指出的元件是表 4-6 的第 3～6 行。這兩個元件將會複製到“@State_Context()”也就是表 4-5 的類別內，“@State_Context()”的類別 Door 的內容將會修改成表 4-11，網底的內容是命令 Copy 所修改的部分。

```
Copy  來源敘述元件  目的敘述元件 ;
```

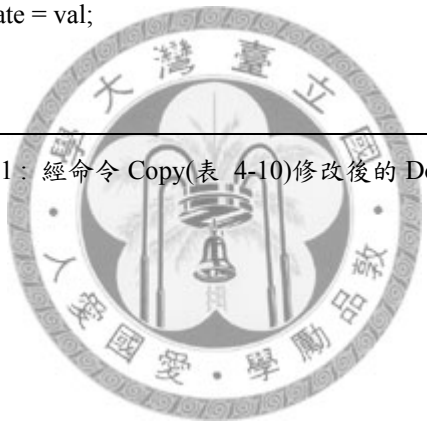
表 4-9：命令“Copy”格式

```
Copy  @State_State()@Declaration(ALL:)  @State_Context() ;
```

表 4-10：命令“Copy”範例

```
1   @State_Context
2   public class Door {
3       @StateField
4       private DoorState mDoorState = DoorState.CLOSED;
5       @Declaration
6       public static final DoorState CLOSED = new CLOSED();
7       @Declaration
8       public static final DoorState OPENED = new OPENED();
9       @Request
10      public void TouchOnDoor () {
11          mDoorState.Touch(this);
12      }
13      @NoComment
14      public void setDoorState (DoorState val) {
15          this.mDoorState = val;
16      }
17  }
```

表 4-11：經命令 Copy(表 4-10)修改後的 Doorjava



4.3.3 命令：Delete

命令 Delete 是一個用於刪除元件的命令，程式開發者可以運用它來將程式中的元件進行刪除。命令 Delete 的格式如表 4-12，此命令會將指定的元件進行刪除的動作。表 4-13 是命令 Delete 的範例，敘述元件 “@State_Context()@Request()” 所指出的是表 4-5 的第 6 ~ 9 行，這個元件將會從類別中刪除，“@State_Context()” 的元件─類別 Door 的內容將會修改成表 4-14，網底的部分是 Delete 所移除的內容。

Delete　敘述元件;

表 4-12：命令“Delete”格式

Delete　@State_Context()@Request() ;

表 4-13：命令“Delete”範例

1	@State_Context
2	public class Door {
3	@StateField
4	private DoorState mDoorState = DoorState.CLOSED;
5	//method TouchOnDoor is delete.
6	@NoComment
7	public void setDoorState (DoorState val) {
8	this.mDoorState = val;
9	}
10	}

表 4-14：經命令 Delete(表 4-13)修改後的 Door.java

4.3.4 命令： Modify-Convert-To

命令 Modify-Convert-To 是一個較為複雜的命令，命令的大概格式如表 4-15，這個命令用來改變 statement 的內容，程式開發者必須先用 Modify 指定要修改的 statement 位於的元件，接著用 Convert 指定要修改的 statement 格式，再利用 To 設計 statement 要轉換而成的內容。Modify 使用方法與 Delete 類似，用於指定出欲修改的元件位置。要特別注意的是，在特殊化的過程中，我們只處理 method body 內的 statement，因此 Modify 指定的元件必須為一個 method，否則將會沒有效用。Convert 指定的元件必須為一個 method 的呼叫，而且此 method 要能夠利用元件敘述來表達。在 Java 中呼叫 method 的格式如表 4-16，其中的變數和方法我們都可以利用敘述元件的方式來表現。舉例而言，表 4-5(@State_Context) 的第 8 行(@StateField) 會呼叫到表 4-6(@State_State) 的第 9 行(@Handle) ，我們就可以表達成表 4-17 的第一列第二欄的內容。

Modify　敘述元件
{
Convert　方法敘述元件
To　“　修改的內容　”(To 可使用的命令參數) ;
}

表 4-15 : 命令 “Modify-Convert-To”格式

變數名稱. 方法名稱() ;
this . 方法名稱() ;

表 4-16 : Java 中的方法格式。

呼叫方法的 statement	表達該 statement 的方法敘述元件
mDoorState.Touch(this) ; **/* 表 4-5 的第 8 行 */**	@State_Context()@StateField() . @State_State()@Handle()
this.TouchOnDoor () ;	@State_Context()@Request()

表 4-17：方法呼叫轉換成敘述元件

雖然從方法的角度表達敘述元件並不困難，但在命令中必須用敘述元件來表示方法，之前定義的參數並沒有辦法讓使用者完全的掌控方法的呼叫。以表 4-18 為案例，@State_State 是一個 interface，它的方法並沒有實體的內容。在多型的想法之下，類別 OPENED 和類別 CLOSED 的方法 Touch 都可以被 interface DoorState 的變數使用，因此類別 OPENED 和類別 CLOSED 都符合“@State_Context() @StateField.@State_State()”這個敘述（此敘述並不能當元件使用）。因此@Handle 的方法將會從類別 OPENED 和類別 CLOSED 當中尋找。這也表示出命令設計人員無法利用敘述元件表達出@StateField 實體的元件。表 4-18 的網底將是這個方法敘述元件含糊的原因，因此我們需要額外的參數來解決。

方法敘述元件	對應的方法
@State_Context()@StateField().- **@State_State()@Handle(ALL:)**	Interface DoorState 的方法 Touch
	Interface DoorState 的方法 Complete
	類別 OPENED 和類別 CLOSED 的方法 Touch
	類別 OPENED 或類別 CLOSED 的方法 Touch

表 4-18：方法敘述元件對應的方法不明確案例

我們在方法呼叫的敘述元件上追加了表 4-19 兩個參數的使用，利用類型和物件來區分出命令設計人員實際所需要的方法。藉由這兩個參數，表 4-18 的敘述元件將會有所區別，表 4-20 將是藉由參數更為細分的方法敘述元件。

敘述元件參數	效果
TYPEOF:	使用 Method Invocation 時，指定該變數的類型。
INSTANCEOF:	使用 Method Invocation 時，指定該變數的物件。

表 4-19：用於方法呼叫時指定類型或物件的參數

方法敘述元件	對應的方法
@State_Context()@StateField().-@State_State(TYPEOF:)@Handle(ALL:)	Interface DoorState 的方法 Touch
	Interface DoorState 的方法 Complete
	類別 OPENED 和類別 CLOSED 的方法 Touch
@State_Context()@StateField().-@State_State(INSTANCEOF:)@Handle(ALL:)	Interface DoorState 的方法 Touch
	Interface DoorState 的方法 Complete
	類別 OPENED 或類別 CLOSED 的方法 Touch (由@StateField()物件的類型決定)

表 4-20：方法敘述元件對應的方法

在定義完成 Convert 所使用的方法敘述元件後，Modify-Convert-To 所選取的方法已經可以定義，而被選定的 statement 將會被 To 的內容所取代，To 的內容將決定命令設計人員的特殊化實做，它必須能夠有更豐富的敘述，因此我們將深入的介紹 To 所能轉換出的程式內容。

在 To 的使用上，我們利用 Modify-Convert-To 的範例命令來說明，表 4-21 是一個 Modify-Convert-To 的範例命令。根據先前的介紹，Modify 將會指定到類別 Door 中的方法 TouchOnDoor，而在這個方法當中，“mDoorState.Touch(this);”這一個 statement 將會符合 Convert 所要求的方法格式，因此該 statement 將會被轉換成 To 所編輯的內容，To 的內容中可以使用敘述元件。為了與實際的程式碼做為分隔，敘述元件必須用“<敘述元件>”的方式指定。經過這個命令的執行後，類別 Door 將會編輯成表 4-22 的內容，網底的部分是經過命令轉換的結果。

```
Modify @State_Context()@Request(ALL:)
{
Convert @State_Context()@StateField().@State_State(TYPEOF:)@Handle(ALL:)    To
        "System.out.println("This statement generate from Modify-Convert-To command.");
          DoorState newState = new <@State_ConcreteState(NAME:OPENED)>;
        "
}
```

表 4-21 : Modify-Convert-To 命令範例 A

```
1   @State_Context
2   public class Door {
3       @StateField
4       private DoorState mDoorState = DoorState.CLOSED;
5       @Request
6       public void TouchOnDoor () {
7           {
8               System.out.println("This statement generate from Modify-Convert-To command.")
9               DoorState newState = new OPENED();
10          }
11      }
12      @NoComment
13      public void setDoorState (DoorState val) {
14          this.mDoorState = val;
15      }
16  }
```

表 4-22 : Modify-Convert-To 命令 A(表 4-21)修改後的 Door.java

在上一個命令範例中，To 運用了實際的 java 程式碼與設計元件內容，命令設計人員可以選定方法敘述元件轉換成自行編輯的程式碼內容。表 4-23 是另一個 Modify-Convert-To 的範例命令，這個範例與之前差異在 To 的內容。這次 To 的內容運用了兩個之前未提到的元素，其中一個是 To 命令本身的命令參數，該參數的說明和效果可見表 4-24。

```
Modify @State_Context()@Request(ALL:)
{
Convert @State_Context()@StateField().@State_State(TYPEOF:)@Handle(ALL:)    To
        "if(<@State_Context(ORIGINAL:)@StateField(ORIGINAL:)> instanceof    <@State_ConcreteState(IFLIST:ALL:)>)
                DoorState newState = new <@State_ConcreteState(CORRESPONDING:)> ;
        "(IFLIST:)
}
```

表 4-23 : Modify-Convert-To 命令範例 B

To 的命令參數	效果說明
IFLIST:	將 if statement 的內容根據符合敘述元件的數量做延伸。
IFLIST 的效果實例	
運用內容	**轉換內容**
if(<@State_ConcreteState(IFLIST:NAME:CLOSED)>) **System.out.println("user edit statement");**	if(<@State_ConcreteState(NAME:CLOSED)>) System.out.println("user edit statement")
if(<@State_ConcreteState(IFLIST:NAME:OPENED)>) **System.out.println("user edit statement");**	if(<@State_ConcreteState(NAME:OPENED)>) System.out.println("user edit statement");
if(<@State_ConcreteState(IFLIST:NAME:OPENED,CLOSED)>) **System.out.println("user edit statement");**	if(<@State_ConcreteState(NAME:OPENED)>) System.out.println("user edit statement"); else if(<@State_ConcreteState(NAME:CLOSED)>) System.out.println("user edit statement")
if(<@State_ConcreteState(IFLIST:ALL:)>) **System.out.println("user edit statement");**	if(<@State_ConcreteState(NAME: CLOSED)>) System.out.println("user edit statement"); else if(<@State_ConcreteState(NAME: OPENED)>) System.out.println("user edit statement")

表 4-24 : To 所使用的命令參數

由表 4-24 的轉變，命令設計人員可以依照符合敘述元件的數量，造出相對應的 if statement；令一個元素是敘述元件參數的補充，雖然可以造就不同條件的 if statement，但 if 的內容若不能隨著條件有所變化，那這項編輯並沒有意義。另外 To 是 Convert 的內容的轉換，To 的內容與 Convert 所指定的方法通常有密切相關，因此我們新增了 To 內容的敘述元件參數(表 4-25)。由這兩個參數的使用，表 4-23 的命令中的<@State_Context(ORIGINAL:)>，將會在 Convert 當中的@State_Context()

內容決定後，就會以用它的內容。表 4-23 的命令中的<@State_ConcreteState-(CORRESPONDING:)>將會跟 if 條件中的<@State_ConcreteState()>內容相同，if 條件中的<@State_ConcreteState()>會根據符合的元件數量創造出不同的內容，因此<@State_ConcreteState(CORRESPONDING:)>的內容將會跟隨著他做變化。經過表 4-23 的 Modify-Convert-To 的轉換，類別 Door 的內容將會轉變成表 4-26，網底的部分是 To 內容所轉換而成。

敘述元件參數	效果
ORIGINAL:	取得與 Convert 敘述元件相同的元件
CORRESPONDING:	取得 IFLIST 所運用的敘述元件

表 4-25： 用於編輯 To 內容的參數

```
1   @State_Context
2   public class Door {
3       @StateField
4       private DoorState mDoorState = DoorState.CLOSED;
5       @Request
6       public void TouchOnDoor () {
7           if(  mDoorState  instanceof  CLOSED) {
8               DoorState newState = new CLOSED();
9           }else if( mDoorState  instanceof  OPENED ) {
10              DoorState newState = new OPENED ();
11          }
12      }
13      @NoComment
14      public void setDoorState (DoorState val) {
15          this.mDoorState = val;
16      }
17  }
```

表 4-26 : Modify-Convert-To 命令 B(表 4-23)修改後的 Door.java

表 4-27 是最後一個 Modify-Convert-To 的範例命令，這個命令在 To 的內容中，可運用命令來撰寫敘述元件，表 4-28 是可以運用的命令。表 4-27 的 “Modify-

Convert-To”的範例命令目的是，將 Convert 所指定的方法轉變成指定的方法的身體(method body)內容。經過表 4-27 的命令處理後，表 4-29，網底的部分是 To 內容所轉換而成，命令 Inline 能夠將方法的參數轉變為區域變數使用。

```
Modify @State_Context()@Request(ALL:)
{
Convert @State_Context()@StateField().@State_State(TYPEOF:)@Handle(ALL:)    To
      "<Inline @State_ConcreteState(NAME:CLOSED)@Handle(ORIGINAL:)> "
}
```

表 4-27 : Modify-Convert-To 命令範例 C

命令名稱與格式	效果
Inline 方法敘述元件	指定的一個方法元件，將方法的身體(method body)內容回傳。

表 4-28 : 運用命令的敘述元件

```
1   @State_Context
2   public class Door {
3       @StateField
4       private DoorState mDoorState = DoorState.CLOSED;
5       @Request
6       public void TouchOnDoor () {
7           Door    inline1door = this;
8           {
9               inline1door.setDoorState(DoorState .OPENED);
10          }
11      }
12      @NoComment
13      public void setDoorState (DoorState val) {
14          this.mDoorState = val;
15      }
16  }
```

表 4-29 : Modify-Convert-To 命令 C(表 4-27)修改後的 Door.java

4.4 特殊化流程

在這個段落我們將介紹命令在特殊化流程當中實際的處理，以及命令如何對程式碼進行修改。特殊化的流程將如圖 4-2 所表示，總共有四個程序(圖中黑色方塊)，分別是：analysis code、command parser、query generate 和 query commit。

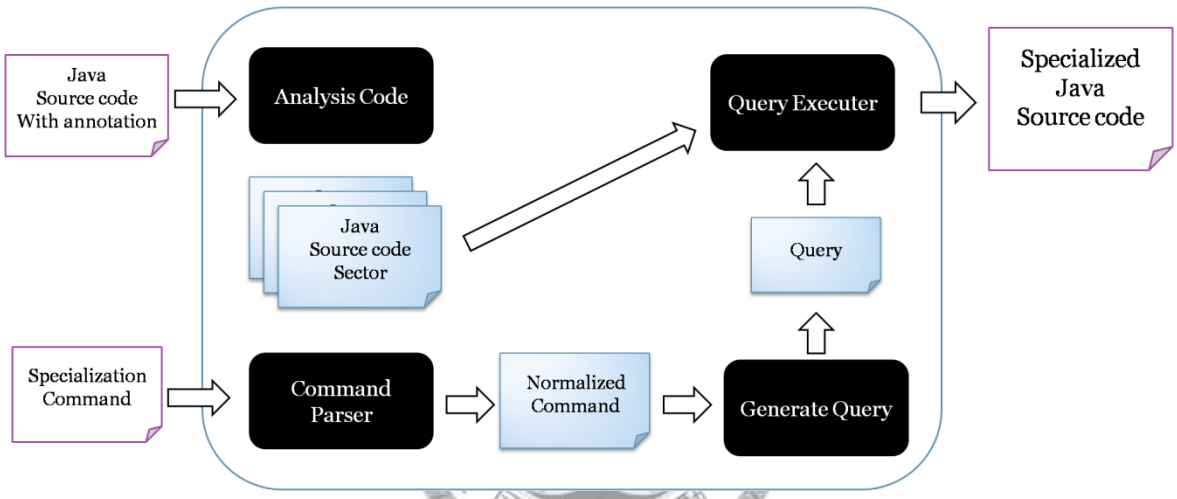


圖 4-2：特殊化流程圖

4.4.1 Analysis Code

在這個程序，我們將對原始程式碼進行分割的動作，以 file、class、field、method 為分割的依據，並且建立標籤(名稱、annotation 等等)來描述此區段的內容，之後程式碼的修改作業將會在區段內進行。特殊化完成後程式碼將會由各個區段內容組合後產生，此階段的目的在於先建立原始程式碼的索引，以至於快速的取得敘述元件，並且以區段做為原始程式碼的暫存，使原始程式碼的內容能保留。特殊

化的流程圖中，analysis code 會產生 Java source code section 的內部資料，該資料的索引如圖 4-3，內容為各區段的程式碼。

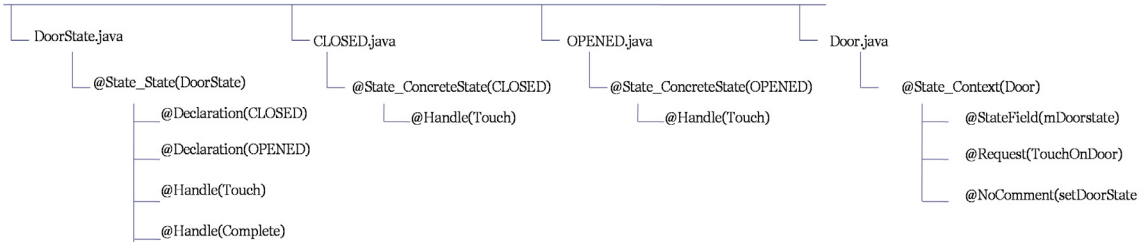


圖 4-3 : annotaion 與名稱建構的索引區段

4.4.2 Command Parser

在這個程序我們將對命令進行分類，並且產生相對於命令的物件來執行該命令的後續工作。命令完後分類後，我們將對命令進行正規化的作業，正規化是一項解析命令的動作，

在進行命令正規化的過程，遇到敘述元件會根據 annotation 和參數，到索引區段中找到符合的元件名稱。非 NAME:的參數會轉變成 NAME:參數加上符合的元件名稱；而 NAME 參數會確認元件是否存在後，將元件名稱當作參數填入，此步驟如表 4-30，再填入的過程中，若符合的元件有 n 個，那將會分裂成 n 項，此步驟如表 4-31。在分裂的過程中，其他的命令內容會一同保留，如表 4-32 所示。因此在 parser 的過程，命令會根據符合敘述元件的數量而增加。

敘述元件	轉換後
@annotation(參數)	@annotation(NAME:符合的元件*)
@annotation(NAME:元件 A , **元件 B , …)**	@annotation(元件 A) @annotation(元件 B) …

表 4-30 : 正規化過程敘述元件的轉換

@State_State(DoorState)@Declaration(ALL:)	@State_State(DoorState)@Declaration(CLOSED)
	@State_State(DoorState)@Declaration(OPENED)

表 4-31： 敘述元件的參數轉換與分裂

原先的命令
Copy @State_State(DoorState)@Declaration(ALL:) @State_Context() ;
分裂後的命令
Copy @State_State(DoorState)@Declaration(CLOSED) @State_Context() ;
Copy @State_State(DoorState)@Declaration(OPENED) @State_Context() ;

表 4-32 :敘述元件分裂的命令變化

在正規化的過程，敘述元件的參數會轉變為實際的元件名稱，同一命令會因為參數指定的元件數量而增加，正規化的程序將會到所有敘述元件的參數都經過轉變為止。正規化完成後，命令中的敘述元件，其參數應為實際的元件。

4.4.3 Generate Query

當所有命令轉變為正規化的形式，此時我們將解析命令的內容，找出命令欲修改的區塊與內容，進而生成 query。query 是一個執行字串的複製、刪除、插入的單位，它的內容是在指定的區塊進行複製、刪除與插入。命令 Copy 和 Delete 可以直接增加和刪除區段來達到命令的效果。Modify-Convert-To 的 query 的內容可以如表 4-33 的形式定義，選定程式區段，修改位置和長度以及要覆蓋的內容。

依照此方式建構 query，我們可以把所有修改的行為集合在一起，再一次交與下一個程序進行 query 的執行。

modifySector_modifyPosition_modifyLength_modifyContent

表 4-33 :Modify-Convert-To 的 query 字串表示法

4.4.4 Query Executer

最後這個階段將是逐步的進行 query。因為 query 是實際更動到原始程式碼的單位行為，我們將會產生使用者介面讓命令設計者觀察到正規化命令與 query 實際修改的內容與位置，以讓程式開發者在運用命令上能夠有更好的掌握，並能夠將執行的 query 進行復原，或放棄 query 的動作。

Query 再進行的過程會有更改到相同區段的問題，我們的設計上是先執行命令 Copy 和命令 Delete 的 query，假如命令設計者有需要複製修改後的方法，那他必須先複製再進行修改。再 Modify-Convert-To 的 query 作用在同一個程式區段，那將會造成 query 的資訊過期的現象，此時 query 上的修改位置將是錯誤的，因此在每個 Modify-Convert-To 的 query 作用後，必須在該區段建立長度變化的資訊，之後的 query 要先經過資訊的檢驗才能確保修改的位置是原先想要的位置。

5. 範例與測試

在第三章，我們介紹了如何利用 UML 設計與產生 java source code 來保留 design pattern 資訊，在第四章，我們利用 design pattern 資訊來判斷各個元件的 design pattern 角色，並利用 annotation 敘述元件設計出特殊化的命令。在這一章我們將從 UML 設計的圖形和撰寫後的程式碼與特殊化後的程式碼，來展示對 State、Strategy 和 TemplateMethod 三個 pattern 進行特殊化作業，並且比較前後的速度增進與 bytecode 檔案大小縮減。我們所針對的 Pattern 在 GOF 的分類中是屬於 Behavior Pattern，參考的程式碼引用於[15] 和[16]，在 Builder Pattern 方面的 specialization 方法可以參考[17]。

5.1 特殊化結果

- **State Pattern**

State pattern 的程式範例引用於[16]的套件 “oozinoz.carousel2”。我們對於它的特殊化策略是將 State 的宣告(@State_State@Declaration)複製到@State_Context 的內容中，並將@Context 類別當中的@Request 方法進行修改。將呼叫@Handle 方法修改成 if 的敘述句，判斷呼叫的變數屬於哪個 State 的物件，而 if 的內容則是被呼叫的方法的內容。在這個修改下，我們可以刪除@State_State 底下的@Declaration，還有每個@ConcreteState 的@Handle 內容。

該 pattern 的 UML 設計可以參考圖 5-1，我們將撰寫 UML 產生的程式碼，讓他有 state pattern 的內容。撰寫後的程式碼可見表 5-1，我們將運用表 5-2 的內容作為特殊化的命令。再經過特殊化程序處理後，可以得到一個特殊化後的程式碼(表 5-3)

這樣的特殊化修改我們省去兩次方法的呼叫，換來的是 if 判斷是的負擔，在

State 的數量不多的情況下則在執行速度上有些微的提升，而類別@State_Context 的檔案雖然會擴大，但能刪去所有類別@ConcreteState 的@Handle 內容，來獲得空間上的改善。

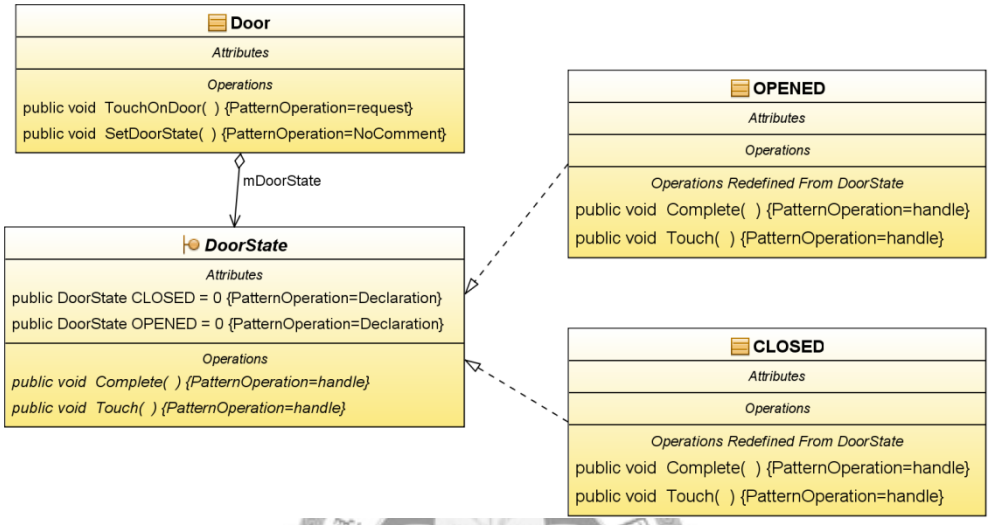


圖 5-1： 設計完成的 state pattern UML diagram

Context	
1	@State_Context
2	public class Door {
3	@StateField
4	private DoorState mDoorState = DoorState.CLOSED;
5	
6	@Request
7	public void TouchOnDoor () {
8	mDoorState.Touch(this);
9	}
10	@Useless
11	public void setDoorState (DoorState val) {
12	this.mDoorState = val;
13	}
14	}

State	
1	@State_State
2	public interface DoorState {
3	@Declaration
4	public static final DoorState CLOSED = new CLOSED();
5	
6	@Declaration
7	public static final DoorState OPENED = new OPENED();
8	
9	@Handle
10	public void Touch (Door door);
11	@Handle
12	public void Complete (Door door);
13	}
1	@State_ConcreteState
2	public class CLOSED implements DoorState {
3	@Handle
4	public void Touch (Door door) {
5	door.setDoorState(OPENED);
6	}
7	}
1	@State_ConcreteState
2	public class OPENED implements DoorState {
3	@Handle
4	public void Touch (Door door) {
5	door.setDoorState(CLOSED);
6	}
7	}

表 5-1 : state pattern 的程式碼

```
Copy @State_State()@Declaration(ALL:) @State_Context();
Modify @State_Context()@Request(ALL:)
{
Convert @State_Context()@StateField().@State_State(TYPEOF:)@Handle(ALL:) To
"if( <@State_Context(ORIGINAL:)@StateField(ORIGINAL:)> instanceof  <@State_ConcreteState(IFLIST:ALL:)>) {
    <Inline @State_Context(ORIGINAL:)@StateField(ORIGINAL:).@State_ConcreteState(CORRESPONDING:)@Handle(ORIGINAL:)>
}"(IFLIST:)
};
```

表 5-2： 特殊化 state pattern 使用的命令

```
1   @State_Context
2   public class Door {
3       @StateField
4       private DoorState mDoorState = DoorState.CLOSED;
5       @Declaration
6       public static final DoorState CLOSED = new CLOSED();
7       @Declaration
8       public static final DoorState OPENED = new OPENED();
9       @Request
10      public void TouchOnDoor () {
11          if(  mDoorState  instanceof  CLOSED) {
12              this.setDoorState(OPENED);
13          }else if( mDoorState  instanceof  OPENED ) {
14              this.setDoorState(CLOSED);
15          }
16      }
17      @Useless
18      public void setDoorState (DoorState val) {
19          this.mDoorState = val;
20      }
21  }
```

表 5-3 :特殊化 state pattern 後的 Door.java (@State_Context)

■ Strategy Pattern

Strategy Pattern 的結構與 State Pattern 類似，因此特殊化的方法可以如同 State 的操作，將@contextInterface 呼叫 Strategy 利用 if 進行修改。在這次的特殊化範例我們不使用特殊化 state 的方法，我們將特殊化的重心往更上層修正，也就是呼叫@StrategyMethod 的部分，在 headfirst 的 strategy 範例中，加入了使用 simulate strategy 的部分，因為 strategy 方法的呼叫由一個 reference 控制，我們改成直接呼叫@StrategyMethod。

Strategy pattern 的 UML 設計可以參考圖 5-2，撰寫後的程式碼可見表 5-5，我們運用表 5-4 的內容作為特殊化 strategy pattern 的命令。再經過特殊化程序處理後，可以得到一個特殊化後的程式碼(表 5-6 和表 5-7)

從這個修改省去了一次的方法呼叫，我們引入@Strategy_User 的內容到@Strategy_Context 當中，因為不會再使用到@Strategy_User 的部份內容，所以可以刪去其中的定義。在 class 的檔案大小上我們省下的一段方法的宣告。

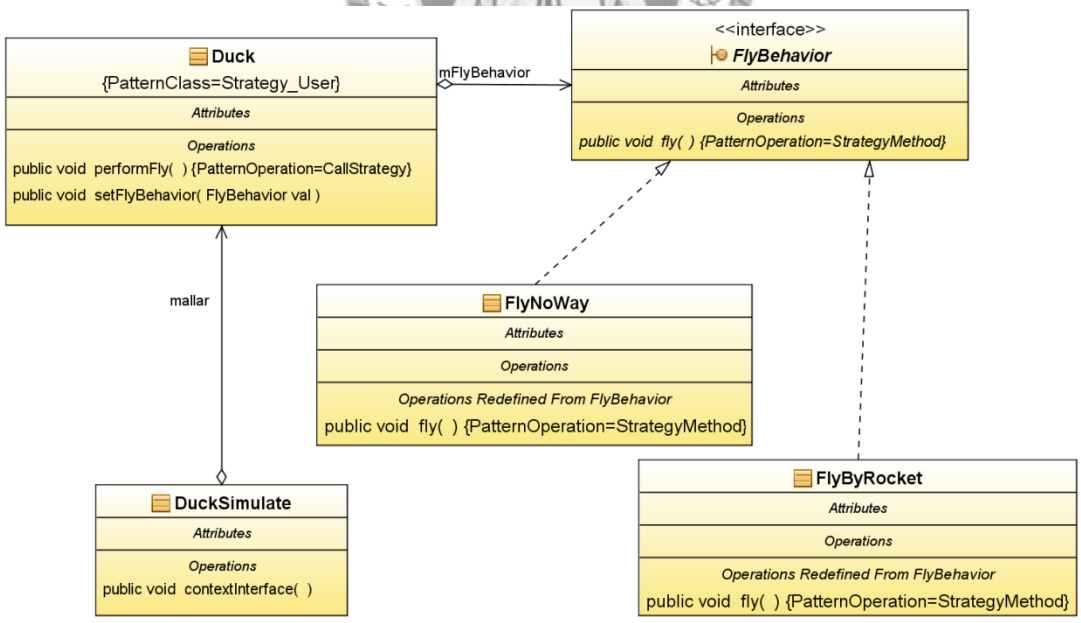


圖 5-2：設計完成的 strategy pattern UML diagram

```
Modify @Strategy_Context()@contextInterface(ALL:)
{
    Convert @Strategy_Context()@Strategy_User_Reference(ALL:).@Strategy_User(TYPEOF:)@CallStrategy () To
        "<@Strategy_Context@Strategy_User_Reference(ORIGINAL:)>.
        <@Strategy_User(NAME:Duck)@Strategy_Reference()>.<@Strategy_Strategy()@StrategyMethod()>
        "
} ;
Modify @Strategy_Context()@contextInterface(ALL:)
{
    Convert @Strategy_Context()@Strategy_User_Reference(ALL:).@Strategy_User(TYPEOF:) @Set_Strategy () To
        "<@Strategy_Context@Strategy_User_Reference(ORIGINAL:)>.
        <@Strategy_User(NAME:Duck)@Strategy_Reference()> = new FlyByRocket();
        "
} ;
Delete  @Strategy_User()@Set_Strategy() ;
Delete  @Strategy_User()@CallStrategy() ;
```

表 5-4：特殊化 strategy pattern 使用的命令

	Strategy
1	@Strategy_Strategy
2	public interface FlyBehavior {
3	@StrategyMethod
4	public void fly();
5	}
1	@Strategy_ConcreteStrategy
2	public class FlyByRocket implements FlyBehavior {
3	@StrategyMethod
4	public void fly() {
5	System.out.println("I'm flying with a rocket"); }
6	}
1	@Strategy_ConcreteStrategy
2	public class FlyNoWay implements FlyBehavior {
3	@StrategyMethod
4	public void fly() {System.out.println("I can't fly");}
5	}

Strategy_User	
1	@Strategy_User
2	public class Duck {
3	
4	@Strategy_Reference
5	FlyBehavior flyBehavior;
6	public Duck() {}
7	@Set_Strategy
8	public void setFlyBehavior(FlyBehavior fb) {
9	flyBehavior = fb;
10	}
11	@CallStrategy
12	public void performFly() {
13	flyBehavior.fly();
14	}
15	}
Context	
1	@Strategy_Context
2	public class DuckSimulate {
3	@Strategy_User_Reference
4	public Duck mallard = new DuckModel();
5	@contextInterface
6	public void contextInterface2 () {
7	mallard.performFly();
8	mallard.setFlyBehavior(new FlyByRocket());
9	mallard.performFly();
10	}
11	}

表 5-5 : Strategy pattern 的程式碼

```
1   @Strategy_Context
2   public class DuckSimulate {
3       @Strategy_User_Reference
4       public Duck mallard = new DuckModel();
5       @contextInterface
6       public void contextInterface () {
7               mallard.flyBehavior.fly();
8               mallard.flyBehavior = new FlyByRocket();
9               mallard.flyBehavior.fly();
10              }
11  }
```

表 5-6：特殊化後的 DuckSimulate.java (@Strategy_Context)

```
1   @Strategy_User
2   public class Duck {
3
4       @Strategy_Reference
5       FlyBehavior flyBehavior;
6       public Duck() {}
7   }
```

表 5-7：特殊化後的 Duck.java (@Strategy_User)

■ TemplateMethod Pattern

在 TemplateMethod 的設計上，Context 呼叫@AbstractClass 的@templateMethod 應該是由一連串固定的方法呼叫構成，而藉由@ConcreteClass 來編輯細節的不同，因此我們將@templateMethod 當中呼叫方法用之前 if 的敘述句引入，而沒有被 override 的方法將可以唯一存在，這樣的修改同樣會有上述的優點與改善。

TemplateMethod pattern 的 UML 設計可以參考圖 5-3，撰寫後的程式碼可見表 5-9，我們運用表 5-9 的內容作為特殊化 TemplateMethod pattern 的命令。再經過特殊化程序處理後，可以得到一個特殊化後的程式碼(表 5-10)

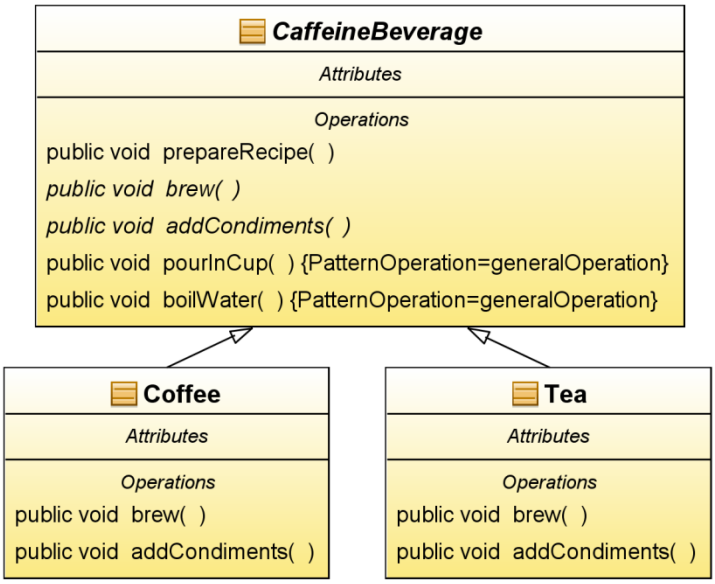


圖 5-3： 設計完成的 TemplateMethod pattern UML diagram

```
Modify @TemplateMethod_AbstractClass()@templateMethod()
{
        Convert @TemplateMethod_AbstractClass()@generalOperation(ALL:) To
        "<inline @TemplateMethod_AbstractClass(ORIGINAL:)@generalOperation(ORIGINAL:)>"
}
Modify @TemplateMethod_AbstractClass()@templateMethod()
{
        Convert @TemplateMethod_AbstractClass()@primitiveOperation (ALL:) To
        "if( this instanceof   <@TemplateMethod_ConcreteClass(IFLIST:ALL:)>) {\n
                <inline @TemplateMethod_ConcreteClass(CORRESPONDING:)@primitiveOperation1(ORIGINAL:)>
        }"(IFLIST:)
}
Delete @TemplateMethod_AbstractClass()@primitiveOperation(ALL:)
```

表 5-8： 特殊化 TemplateMethod pattern 使用的命令

TemplateMethod	
1	@TemplateMethod_AbstractClass
2	public abstract class CaffeineBeverage {
3	@templateMethod
4	public final void prepareRecipe() {
5	boilWater();
6	brew();
7	pourInCup();
8	addCondiments();
9	}
10	@primitiveOperation
11	public abstract void brew ();
12	@primitiveOperation
13	public abstract void addCondiments ();
14	
15	public void pourInCup() {
16	System.out.println("Pouring into cup");
17	}
18	public void boilWater() {
19	System.out.println("Boiling water");
20	}
21	}
1	@TemplateMethod_ConcreteClass
2	public class Tea extends CaffeineBeverage {
3	@primitiveOperation
4	public void brew () {
5	System.out.println("Steeping the tea");
6	}
7	@primitiveOperation
8	public void addCondiments () {
9	System.out.println("Adding Lemon");
10	}
11	}
1	@TemplateMethod_ConcreteClass
2	public class Coffee extends CaffeineBeverage {
3	@primitiveOperation
4	public void brew () {

```
5           System.out.println("Dripping Coffee through filter");
6       }
7       @primitiveOperation
8       public void addCondiments () {
9           System.out.println("Adding Sugar and Milk");
10      }
11  }
```

表 5-9 : TemplateMethod Pattern 的程式碼

```
1   @TemplateMethod_AbstractClass
2   public abstract class CaffeineBeverage {
3       @templateMethod
4       public final void prepareRecipe() {
5           System.out.println("Boiling water");
6
7           if (this instanceof Coffee)
8               System.out.println("Dripping Coffee through filter");
9           else if (this instanceof Tea)
10              System.out.println("Steeping the tea");
11
12          System.out.println("Pouring into cup");
13
14          if (this instanceof Coffee) {
15              System.out.println("Adding Sugar and Milk");
16          } else if (this instanceof Tea) {
17              System.out.println("Adding Lemon");
18          }
19      }
20      public void pourInCup() {
21          System.out.println("Pouring into cup");
22      }
23      public void boilWater() {
24          System.out.println("Boiling water");
25      }
26  }
```

表 5-10 : 特殊化後的 CaffeineBeverage.java (@TemplateMethod_abstract)

5.2 結果測試

在這個階段我們將運用上述的特殊化的範例，與原始的設計進行比較，我們將對程式執行的時間與 bytecode 的大小進行評比，程式執行的效能我們用 5000 次的呼叫所花費的時間來比較，至於 bytecode 檔案由 Java1.5 版進行編譯，以此為基準來比較檔案的大小。

design pattern		original	specialized	improve
State	Time	2429.679	2080.524	1.1678
	code size	5,78KB	4.47KB	-1.31 KB
Strategy	Time	2261.640	1714.781	1.3189
	code size	3.42KB	3.28KB	-0.14 KB
Template Method	Time	7383645.509	7019975.755	1.051
	code size	3.32KB	1.31KB	-2.01 KB

表 5-11：特殊化前後的差異(Time 單位為 micro second)

每個 design pattern 在經過特殊化之後，在效能和檔案大小有些微的優化，效能提升的原因來自減少多餘的方法呼叫，靠著不呼叫方法，或直接呼叫最重要的方法，而在檔案大小方面，則是將方法的執行轉移到其他方法去，呼叫地點的程式量會因此增加，被呼叫的方法若不再被需要，那它減少的不只是程式的量還包含方法的宣告，若呼叫地點太多，那增加的程式量將會倍增，而使得檔案大小不減反增，因此這個行為必須挑選在被呼叫次數少的方法以及只有 delegation 的方法上。

6. 結論與未來工作

6.1 結論

根據我們的設計，Annotation 可以運用來幫助類別和介面表達 design pattern 的角色，當程式開發進入產品化階段，我們將在不違反結果的情況下進行特殊化，來幫助程式有更好的表現。這樣的設計最主要的目的，是希望能夠彌補編譯器在編譯的過程，能夠獲得更多的資訊。而這些資訊通常不能藉由機器來辨識，在仰賴程式開發者的同時，我們希望能降低辨識作業的困難度。因此我們在設計 UML 的階段，要求程式開發者在 UML 中加入他們所需要的訊息。程式開發者在 UML 設計階段不只能夠加入 design pattern 資訊，他所加入的訊息都能夠在產品化階段被找到。

在特殊化的階段，我們設計的方法是針對 annotation 名稱做比對，因此特殊化方法不僅對 design pattern 的設計有效果，還包括了程式開發者自行的設計。產品化階段的程式碼，已經不需要再考慮程式維護的要求，我們可以靠著 annotation 所記錄的資訊來找到適合特殊化程式碼的位置。不論是靠著命令進行特殊化或程式開發者自行編輯特殊化，都能夠有效的提升程式執行的速度和降低程式的大小。

Design pattern 在提高程式維護性的同時，也造成了額外的負擔，我們所期望的特殊化方法是能夠消除這些負擔，讓 CPU 時間能夠花費在主要的執行上。最後 annotation 所表達的 design pattern 角色，也會在這個階段一同被移除，而不會造成程式的累贅。

在我們的命令設計下，許多的 specialization 和 partial evaluation 策略都可以經由命令實現。而 design pattern 與 annotation 的運用也可幫助開發時期的程式的可讀性，同時能夠幫助程式開發者了解撰寫的程式意義為合。我們與現今廣泛使用的

NetBeans IDE 的結合，目的在於能讓程式開發者更快速的上手並且減輕程式開發者負擔。

6.2 未來工作

Design pattern 的資訊加入到 UML 的設計雖然容易實現，但是 specializa- tion 和 partial evaluation 方法並不容易利用命令來實現，這問題點在於命令的數量和過於複雜。論文的流程上 UML 的設計可以很容易的用圖形達到編輯，選項的功能可以降低設計人員的門檻，自行編輯 design pattern 能讓設計人員更靈活的運用，這些功能在操作上非常簡易。但特殊化的作業卻沒有辦法達到如此，命令設計本身不容易了解，程式開發者必須了解特殊化方法的意義和如何運用命令實現。而且在程式規模小的情況下，命令的設計作業反而比程式開發者自行修改程式碼更複雜。因此命令的設計上還有待於修改與加強，除了能夠提供更多的命令，還要能夠讓開發人員容易編輯。

實際上 design pattern 的優化對於嵌入式的系統有一定的貢獻。現今軟體的開發會隨著個人電腦硬體的進步而提高硬體需求。而嵌入式系統的硬體進步並沒有個人電腦的迅速，因此特殊化的要求也將更加需要。但是目前我們所設計的特殊話方法架構於 design pattern 的設計，如何將特殊化的方法拓展到 design pattern 以外的層面，這是一個可以思考的議題。

參考文獻

[1] Tim Lindholm and Frank Yellin, *The Java Virtual Machine Specification, Second Edition*, Addison-Wesley Professional, 1999.

[2] Sun Microsystems, Java Virtual Machine, JDK™ 6 Documentation, Virtual Machine, http://java.sun.com/javase/6/docs/technotes/guides/vm/index.html.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.

[4] OMG, Unified Modeling Language Specification, Version 2.2, Technical Report, Object Management Group, 2009

[5] Netbean Community, Netbeans IDE, http://www.netbeans.org

[6] Dana Nourie, "Getting Started with an Integrated Development Environment", http://java.sun.com/developer/technicalArticles/tools/intro.html.

[7] Netbean Community, NetBeans Unified Modeling Language (UML) Plugin, http://www.netbeans.org/features/uml/.

[8] Frank Budinsky, Marilyn Finnie, Patsy Yu and John Vlissides , “Automatic Code Generation from Design Patterns,” *IBM Systems Journal,* Vol. 35(2). Pages : 151 – 171,1996.

[9] Sun Microsystems, Java Annotation, JDK™ 5 Documentation, Java Programming Language, http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html.

[10] Netbean Community, NetBeans Platform and Module Development, http://www.netbeans.org/kb/trails/platform.html.

[11] Ulrik P. Schultz, Julia L. Lawall and Charles Consel, “Specialization Patterns,” Research report 1242, IRISA, Rennes, France, 1999.

[12] Ulrik P. Schultz, Julia L. Lawall and Charles Consel, “Automatic Program Specialization for Java,”. *ACM Transactions on Programming Languages and Systems (TOPLAS)* Vol. 25, No. 4 ,452-499, 2003.

[13] FreeMarker , http://freemarker.org/ .

[14] Sun Microsystems, Annotation Processing Tool, JDK™ 6 Documentation, Tools , http://java.sun.com/javase/6/docs/technotes/guides/apt/index.html.

[15] Elisabeth Freeman, Eric Freeman, Bert Bates and Kathy Sierra, Head First design patterns, O' Reilly & Associates, Inc., 2004.

[16] Steven John Metsker, William C. Wake, Design Pattern in Java™, Addison-Wesley, 2006.

[17] Igor Dysko “Specialization of Object-Oriented Programs Written in Java Language,” *Slovak University of Technology* , 2005.