有關本書作者消息更新 ....

※　高煥堂 Android 書 2009 年新版(繁體)上市

#1. <<Google Android 應用框架原理與程式設計 36 技>> 2009 第四版
#2. <<Google Android 應用軟體架構設計>> 2009 第二版
#3. <<Google Android 與物件導向技術>> 2009 第二版
#4. <<Google Android 設計招式之美>> 2009 初版

※　高煥堂於 2009/2/20-26
在上海開 Android 技術與產業策略講座

※　高煥堂於 2009/4/10-16
在北京 & 上海開 Android 技術與產業策略講座

連絡 E-mail： misoo.tw@gmail.com　高煥堂 老師 收
歡迎光臨: www.android1.net 及其「Android 大舞台」版

2 月 14-15(週六&周日)

新春 台北 Android 特訓班 高煥堂 主講

主題：**軟硬整合技術與行動應用開發**(共 11 小時)

時間：2 月 14　10:00am – 5:00pm (6hr) &

　　2 月 15　1:00pm – 6:00pm(5hr)

大綱：這是 Android 技術的精華課程，內容完整，包括三個層面：

1. 軟硬整合技術
   - ARM & Linux 核心 & VM 介紹
   - 交叉編譯 Toolchains 介紹及使用
   - JAVA 與 C++程式庫連結和安裝
   - C 程式庫開發&與 Driver 的連結
   - Android 的移植問題與經驗分享
2. 框架設計招式
   - 認識 Android 的 Binder 架構設計
   - 熟悉 Android 的 Security 安全架構
   - 演練 JNI(即 Java 與 C++整合架構)
   - 介紹 Android 其他各種設計招式
3. 應用軟體開發
   - 複習 Java 語言:如類別繼承等等
   - 模擬器及 Eclipse 開發環境安裝
   - 將可執行程式安裝到 G1 手機裡
   - AP 開發:從商業流程到通訊架構
   - 資料庫技術與 SQLite 應用實務
   - Location-based 行動軟體開發實務
   - 其他 Android 行動軟體開發技術

地點：台北市，捷運後山埤站旁(步行約 4 分鐘)

費用：NT$ **9,500** 元

報名：misoo.tw@gmail.com 高煥堂老師收

詢問：(02)2739-8367 找高煥堂老師

講師：高煥堂

　　學歷： 美國科羅拉多大學 資管研究所

淡江大學 管理科學研究所
中興大學 法商學院
專長： 32 年資深電腦軟、硬體系統開發經驗
出版： 4 本有關 Android 技術書籍...
#1. <<Google Android 應用框架原理與程式設計 36 技>>
#2. <<Google Android 應用軟體架構設計>>
#3. <<Google Android 與物件導向技術>>
#4. <<Google Android 設計招式之美>>

--- END ---

第 1 章

設計樣式與 Android 框架設計

1.1　設計與樣式(即招式)

圍棋有棋譜、烹飪有食譜、武功有招式、戰爭有兵法，..... 皆是專家和高手的經驗心得。雖然 Android 應用框架是一個氣象萬千的獨特設計(Unique Design)，例如它與 WinMobile、Synbim 等其他平台是不一樣的。但是，在其獨特氣質的幕後，卻是運用了許多專家們常用的「設計招(樣)式」(Design Pattern)。樣式運用得好，能化解衝突爲詳和，問題也迎刃而解，自然令人感到舒暢。好的設計師以流暢的方式將令人舒暢的樣式組合而成氣象萬千的獨特作品，其設計品自然令人感到快活。Android 就是這種令人心曠神怡的傑出作品。

樣式就是在某個領域(Domain)裡的專家，針對經常出現的問題(Problem)，其常用的解決之道(Solution)。因爲出自專家，所以樣式是高品質的。因爲它經常出現，所以有學習及推廣的價值。因爲從經驗萃取而得，所以具有高度實用價值。

傑出的設計品必需融入環境因素，並使環境與設計品構成和諧的整體。在設計過程中，常會面臨環境的各種需求和條件，來自不同方面的需求可能會互相衝突而呈現不和諧的現象。因而不斷運用樣式來化解衝突使其變爲均衡和諧，亦即不斷把環境因素注入樣式中而產生有效的方案來使衝突之力量不再互相激盪。有效的設計專家，會大量運用其慣用之樣式，比較少一切從頭創造新方案(Reinvent the Wheel)。

這並不意味著，使用樣式是缺乏創意。因爲樣式是抽象的，運用時必須視環境(Context)的特殊性而修正，然後才產生具體的方案。樣式會引導設計師的創意，使其融合別人的智慧，並充分考慮外在環境的獨特性。例如，食譜並未限制廚師的創新，反而常激發廚師的新創意。所以樣式告訴您理想的方案像什麼、有那些特性；同時也告訴您些規則，讓您依循之，而產生適合於環境的具體方案。當規則的掌握不靈活、或無法充分融入外在環境因素，可能無法得到具體有效的方案。因而新的設計師們經由樣式的引導，成長會加速，具有建設性。

樣式的組合就如同寫作文章，樣式是字彙或成語，而設計品就如同寫作出來的句子或文章。寫作文章時，人們依循文法規則及風格而將字彙成語流暢地組合成句子，再流暢地組合成文章。每位作家皆有各自的一套規則和風格來寫作文章。同理，每位設計家皆各有其慣用手法來將樣式流暢地組合成爲設計品。每位音樂家皆各有其慣用手法來將旋律樣式流暢地組合成爲樂章。每位詩人皆各有其

慣用手法來將詩歌樣式流暢地組合成爲詩篇。Android 就是這種令人心曠神怡的詩篇。

1.2 簡介設計樣式(Design Pattern)

剛才說過，樣式是抽象的典範，引導讓您套用它、修正它、加上外在環境因素，才會得到具體可行的方案(Solution)。它告訴您理想的方案像什麼、有那些特性；同時也告訴您些規則，讓您依循之，進而在您的腦海裏產生適合於環境的具體方案。只要靈活掌握規則、充分融入大環境因素，即能瞬間得到具體有效的方案。所以建築大師亞歷山大(Christopher Alexander) 做了如下之定義:

「樣式(Pattern)是某外在環境(Context) 下，對特定問題(Problem)
的慣用解決之道(Solution)。」

茲舉個例子，在平坦的校園裡(即 Context)，下述兩項需求是互相衝突的:

- 學生需要交通工具代步。
- 校園空氣必須保持清潔。

就產生問題了，該如何化解呢? 當您想到好的妙招時，可依據亞歷山大所建議的格式，撰寫如下：

校園交通樣式之一
Intent: 維護一個安靜又可交流智慧的學習環境。
Force 1: 學生需要交通工具代步。
Force 2: 校園空氣必須保持清潔。
Solution: 規定在校園中只能騎自行車，不能騎有污染的機車和汽車。
Consequences: 在廣大平坦的校區裡，獲得安靜又安全的讀書環境。

此解決之道是否有效，是取決於外在環境(即 Context) 。例如，在不平坦的

新竹清華大學校園中，上述解決之道就無效用了。這是因爲外在環境不一樣，因此必須修正既有樣式，而成爲新的樣式，如下：

校園交通樣式之二
Intent: 維護一個安靜又可交流智慧的學習環境。
Force 1: 學生需要交通工具代步。
Force 2: 校園空氣必須保持清潔。
Solution: 規定在校園中只能騎自行車或電動機踏車，不能騎有污染的機車或汽車。
Consequences: 如果配合方便的充電站，在廣大的校區裡，獲得安靜又安全的讀書環境。

樣式運用得好，能化解衝突爲詳和，問題也迎刃而解，自然令人感到舒暢。好的設計師以流暢的方式將令人舒暢的樣式組合而成設計品，其設計品自然令人感到快活。

1.3 從亞歷山大的建築樣式到 GoF(Gang of Four)設計樣式

由於本書是以 GoF 設計樣式爲基礎，所以在此說明 GoF 設計樣式的來源，及其發展歷程和背景。

1964-1971 年

1964 年，著名建築學家亞歷山大(Christopher Alexander)出版一本書：

Notes on the Synthesis of Form[參 1]

他提出「形」(Form)的概念，認爲設計師可創造形化解環境中互相衝突的需求，使衝突變成爲和諧的景象。他也提出樣式(Pattern)觀念，樣式可引導設計師逐步創造出新奇的形，以便化解互相衝突之需求。1971 年，該書再版上市，此時正是

軟體結構化設計(Structured Design)方法的萌芽階段，該書對當時 Ed Yourdon 和 Larry Constantine 的結構化觀念的誕生具有決定性的影響力[參 2]。

1972-1985 年

在這期間，亞歷山大任教於加州柏克來大學，他和其同事共同研究樣式語言觀念，並出版了四本書：

1. **The Timeless Way of Building**[參 3]
   －－完整地介紹他的 Pattern 觀念，以及 Pattern Language 觀念。
2. **A Pattern Language**[參 4]
   －－實際列舉了 253 個建築方面的樣式。
3. **The Oregon Experiment**[參 5]
   －－ 敘述在奧勒蘭大學的實驗過程。
4. **The Production of Horses** [參 6]
   －－敘述在墨西哥的實驗情形。

1986-1989 年

這期間，隨著軟體業的 C++語言之誕生，物件導向(Object-Oriented，簡稱 OO)技術日益成熟，這時亞歷山大的樣式觀念再度影響軟體的設計方法。1987 年，Ward Cunningham 和 Kent Beck 兩人首先嚐試將 OO 觀念與樣式觀念結合起來。他們的研究著重於使用者介面(User Interface)方面，並在 OOPSLA/87 會議上發表其成果[參 7][參 8]。不過，他們的研究並未立即引起熱潮。

1990 年

在歐洲的 OOPSLA/90 會議上，由 Bruce Andreson 主持的"Architectural Handbook"研討會中，Erich Gamma 和 Richard Helm 等人開始談論有關樣式的話題。

1991 年

- 在 OOPSLA/91 會議上，由 Anderson 主持的研討會中，Erich Gamma 和伊利

諾大學教授 Ralph Johnson 等人一起討論樣式的相關問題。

- Erich Gamma 完成了他的博士論文──
  "Object-Oriented Software Development based on ET++: Design Patterns, Class Library, Tools " [參 9]
- Peter Coad 也進行有關樣式之研究。
- James Coplien 在貝爾實驗室裡，也著手研究有關企業組織方面的樣式(Organization Pattern)。
- 樣式觀念開始出現在通俗的電腦軟體雜誌上，例如：Tom Love 撰寫的"Timeless Design of Information Systems"[參 10]。這介紹如何將樣式觀念引入到軟體設計領域中。

1992 年

- 在 OOPSLA/92 會議上，Andreson 再主持研討會，樣式觀念已漸成爲熱門的話題。在研討會中，伊利諾大學教授 Ralph Johnson 發表其有關樣式與應用架構(Application Framework)之關係[參 11]。
- Peter Coad 在 ACM 期刊上發表了 OOA(Object-Oriebnted Analysis)方面的 7 個樣式[參 12]。
- 在通俗的電腦雜誌上也繼續出現許多有關於樣式的文章[參 13]。

1993 年

這年，樣式已躍居 OO 領域最熱門的話題。

- 在 1993 年 8 月份，Kent Beck 邀請 Grady Booch、Richard Helm、Ralph Johnson、Ward Cummingham、James Coplien、Ken Auer 及 Hal Hildebrand 一起聚會於美國中部科羅拉多(Colorado)州的落磯山(Rocky Mountain)麓，共同討論如何將亞歷山大的樣式觀念與 OO 技術結合起來。他們決定以 Erich Gamma 的「設計樣式」研究成果爲基礎，繼續努力研究下去。這個樣式研究組織就稱爲「山麓小組」(Hillside Group)。
- 在通俗的電腦軟體雜誌上，有關於樣式的文章日漸增加，例如：Grady Booch 談樣式與機制(Mechanism)觀念的相通之處[參 14]。R. Gabriel 在 JOOP 雜誌的"Critic-at-Large" 專欄中，自 1993 年 2 月份起[參 16]，連載他對

樣式觀念的看法，這專欄一直連續到 1994 年。

- Erich Gamma 在 ECOOP/93 會議上發表有關樣式與設計重用(Design Reuse)之關係。

1994 年

- 1994 年 8 月 4 日，由「山麓小組」發起，在 Illinois 的 Allerton Park 地方召開的第 1 屆世界性的 OO 樣式會議，名稱叫 Pattern Languages of Programs，簡稱爲 PLoP'94。這並非傳統的大型會議，而是由二十多位來自全世界各地的論文提供者互相討論，交流心得與意見。其論文則由 James Coplien 和 D. Schmidt 一起編輯成書，名稱爲 Pattern Languages of Program Design，並在 1995 年出版上市[參 17]。Ralph Johnson 也在 ROAD 雜誌上介紹這次會議的成果和評論[參 18]。James Coplien 也詳細介紹 PLoP'94 的討論情形[參 19]。
- J. Soukup 出版新書談 C++ 的樣式類別[參 20]。
- Grady Booch 的名著──Object-Oriented Analysis and Design:with applications 書上也引用許多樣式觀念。
- 6 月份, James Coplien 在紐約的 ObjectExpo 研討會上，談到有關軟體樣式方面的常見問題和答案[參 23]。
- 在通俗的電腦雜誌上，繼續出現有關於樣式的文章。例如，Kent Beck 說明了樣式是電腦人員之間的有效溝通工具[參 21]。Kent Beck 和 Ralph Johnson 說明樣式與架構(Architecture)間之關係[參 24]。 James Coplien 介紹組織與軟體發程序之樣式[參 25]。

1995 年

這年，樣式的應用層面逐漸增大，例如 GUI 、Networking 等應用。

- 9 月份 PLoP'95 仍在 Illinois 的 Allerton Park 地方舉行。共有 70 多人參加，論文題目比前一年更多樣化，包括 Web Pages 製作的樣式等等。D. Anthony 簡介 PLoP‘95 的討論情形。其論文將由 John Vlissides 等人負責編輯成書並發行上市。
- Erich Gamma、Richard Helm、Ralph Johnson 和 John Vlissides 共 4 人一起出版了驚動軟體界的經典名著：

Design Patterns:Elements of Reusable Object-Oriented Software

一書，成爲 1995 年最熱門的軟體設計書籍[參 26]。由於此書作者是 4 位聲名響亮的名家，大家就稱他們爲「設計樣式 4 人幫(Gang of Four)」，簡稱爲 GoF。在此書裡，列舉了軟體界的 23 個常用樣式，並給予深刻的闡述其思路和手藝。隨後至今，大家都稱這 23 個經典的設計樣式爲：GoF 設計樣式。

參考文獻

[1] Alexander, C., Notes on the Synthesis of Form. Harvard University Press, Cambridge, Massachusetts, 1971.

[2] Yourdon, E., "Object-Oriented Design", American Programmer, March 1990. PP.14-24.

[3] Alexander, C., The Timless Way of Building, Oxford University Press, New York, 1979.

[4] Alexander, C., A Pattern Language, Oxford University Press, New York, 1977.

[5] Alexander, C.,The Oregon Experiment, Oxford University Press, New York, 1978.

[6] Alexander, C., The Production of Houses, Oxford University Press, New York, 1985.

[7] Cunningham, W., “Panel on design methodology”, ACM SIGPLAN Notices, 1993 PP.94-95(Addendum to the Proceedings of OOPSLA'87).

[8] Beck, K., “Using a pattern language for programming”, Workshop on Specification and design, ACM SIGPLAN Notices PP.16, 1988 (Addendum to the Proceeding of OOPSLA'87).

[9] Gamma, E., Object-Oriented Software Development based on ET++:Design Patterns, Class Library, Tools(in German). PhD thesis, University of Zurich, 1991.

[10] Love, T., "Timeless Design of Information Systems", Object Magazine , PP.42-48, NOV-DEC 1991.

[11] Johnson, R., "Documenting Framework Using Patterns", In OOPSLA/92 Proceedings, PP.63-76, ACM Press, 1992.

[12] Coad, P., "Object-Oriented Patterns", Communications of ACM, PP.152-159, September 1992.

[13] Bowles, A., "Developing Organic Systems", Object Magazine, PP.21-22, JAN-FEB 1992.

[14] Booch, G., "Patterns", Object Magazine, PP.24-28, July-August 1993.

[15] Bowles, A., "Lessons From Urban Planning:System zoning and building inspections", Object Magazine, PP.24-26, Nov-Dec 1993.

[16] Gabriel, R., "Habitability and Piecemeal Growth", JOOP, PP.9-14, Feb 1993.

[17] Coplien, J., and Schmidt, D. Pattern Languages of Program Design, Addition-Wesley, Reading, *MA 1995.

[18] Johnson, R., "A Report on PLOP'94", Report on Object Analysis and Design, NOV 1994.
[19] Coplien, J., "What I did on my summer vacation?", C++ Report, Nov-Dec 1994.
[20] Soukup, J., Taming C++:Pattern Classes and Persistence for Large Projects, Addison-Wesley, Reading, MA, 1994.
[21] Beck, K., "Patterns and Software Development", Dr. Dobb's Journal, PP.18-22, Feb 1994.
[22] Gabriel, R., "Pattern Languages", JOOP, PP.72-75, Jan 1994.
[23] Coplien, J., "Software Design Patterns:Common Questions and Answers", Object Expo Conference Proceddings, SIGS, 1994.
[24] Beck, K. and Johnson, R.,"Patterns Generate Architectures", Proceeding ECOOP'94.
[25] Coplien, J., "Pattern Languages for Organization & Process", Object Magazine, July-August 1994**.**
[**26] Gamma, E. et al.,** Design Patterns:Reusable elements of Object-Oriented Software, Addison-Wesley, Reading, MA, 1995.
[27] Coad, P., Object Models:Strategies, Patterns and Applications, Prentic-Hall Englewood Cliffs, NJ, 1995.
[28] Fred, K., "Client/Server Analysis and Design Patterns", Software Development, July 1995.
[29] Schmidt, D., "Virtual Interview with James Coplien", C++ Report, Sept. 1995.
[30] Coplien, J., "Software Development as Science, Art, and Engineering", C++ Report, July-August 1995.
[31] Schmidt, D., "The Timless Way of Developing Software", Software Development, May 1995.
[32] Koenig, A., "On Patterns and Antipatterns", JOOP, March-April 1995.
[33] Schmidt, D., and Stephenson, P. "Using Design Patterns to Evolve System Software from UNIX to Windows NT", C++ Report, March-April, 1995.
[34] Rumbaugh J., "What is a method?", JOOP, OCT 1995.
[35] Coplien, J., "Pattern Mining", C++ Report, OCT 1995.
[36] Vlissides, J., "Visiting Rights", C++ Report, Sept 1995.

1.4 舉例說明應用框架之意義

在本書的姐妹作品：<<Google Android 應用框架原理與程式設計 36 技>>一書裡，曾經介紹過應用框架(Application Framework)的原理、機制和特色了。在本節裡就不再重複說明了。但是爲了讓你對應用框架有更深刻的印象，在本節裡，將舉例說明應用框架之意義。首先請你先看圖 1-1：

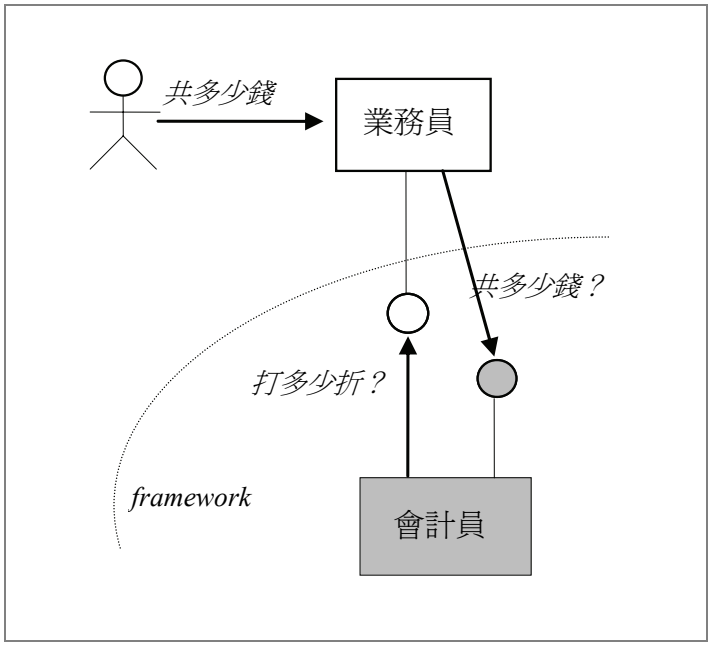

圖 1-1　框架的比喻

從上圖可以體會出框架的理念和人們日常生活經驗有許多相似之處。顧客買東西，問業務員共多少錢，業務員將銷售產品種類告訴會計員，請其計算總金額，含稅金等等。然而，到底打幾折。則是由業務員跟顧客討價還價而得的，所以會計員會問業務員。其中，會計員屬於框架，而業務員則屬於特殊應用(Application)。

同樣的思惟，就可以設計一個簡單的繪圖元件框架，親身經歷一下框架之設計。此框架可以簡單到只含有一個類別叫 Graphic，也定義了介面 IGraph，如下

圖 1-2 所示。

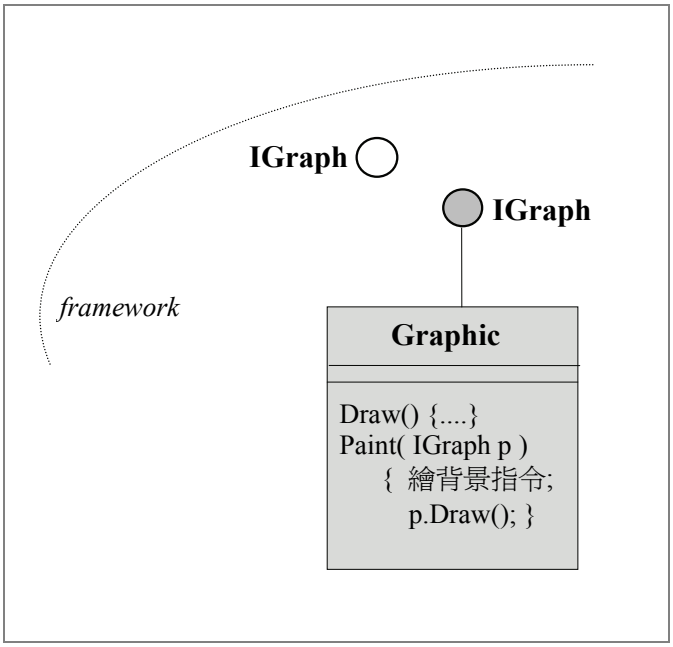

圖 1-2　簡單的繪圖軟體框架

此框架裡定義了一個介面，含有兩個函數，如下：

```
IGraph{
    Draw();
     Paint();
   }
```

有了此框架，您可以自己使用，也可以賣給別人(即您的客戶)。而且你並不需要提供 Graphic 類別定義的原始碼(Source Code)，只要將此框架裡的類別及介面定義編譯成爲二進位目的碼(Object Code)，再賣給您的客戶就行了。客戶買了您的應用框架之後，可以設計出各式各樣的特殊應用類別，然後透過 IGraph 介面進行雙向溝通，就將應用類別與框架結合起來了。如下圖 1-3 所示。

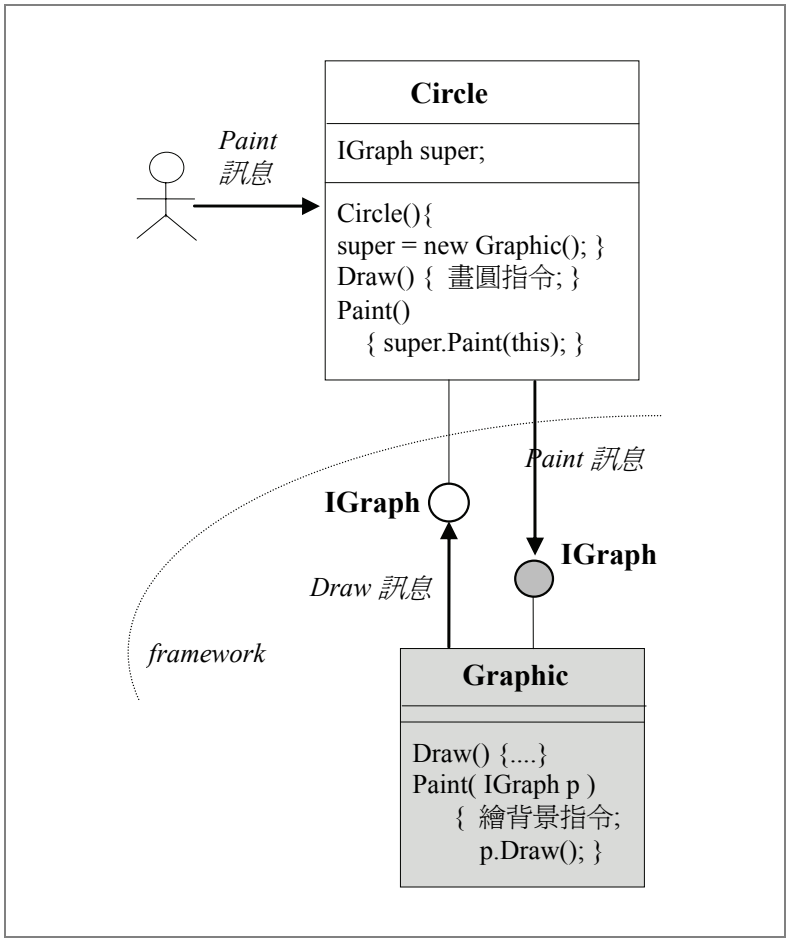

圖 1-3　框架裡的 Graphic 類別協助 Circle 應用類別

在執行期間，此框架內的 Graphic 類別會負責繪出圖形背景。而由應用類別負責繪出前景圖形，如圓形等。當您的客戶設計好 Circle 類別，就可以寫個主程式如下：

```
void main(){
    Circle c();
    c.Paint()
 }
```

執行時，此主程式會誕生一個 Circle 之物件，同時也要求框架誕生一個 Graphic 之物件來協助 Circle 之物件。就像剛才圖 1-1 裡的會計員，用來協助業務員去服務顧客。其原理是相通的。例如，主程式呼叫 Circle 的 Paint()函數時，此 Paint()呼叫 Graphic 的 Paint()函數，當 Graphic 的 Paint()函數繪完圖形背景之後，會反過來呼叫 Circle 的 Draw()函數，由 Circle 的 Draw()函數繪出前景的圓形。

由於框架所含的是可以重複使用的部份，所以它可以搭配特殊的應用類別而畫出其它多樣性的圖形，如下圖 1-4 就搭配 Triangle 應用類別而繪出三角形了。

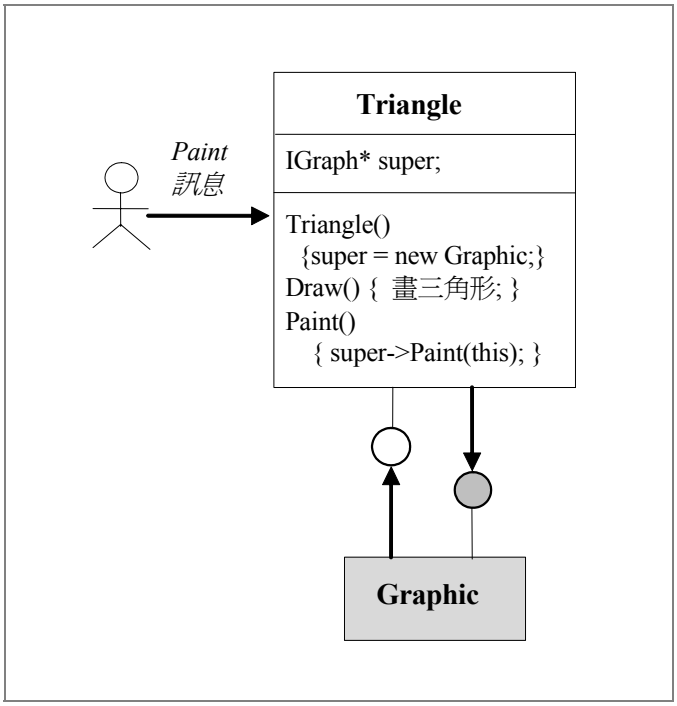

圖 1-4　框架可搭配各式各樣的應用類別

基於上述的框架特性，就能繼續擴充框架的內涵，納入更多的可重複使用之類別。於是，框架就逐漸由一個 Graphic 類別而擴大成爲較複雜的繪圖框架了。如下圖 1-5 所示。

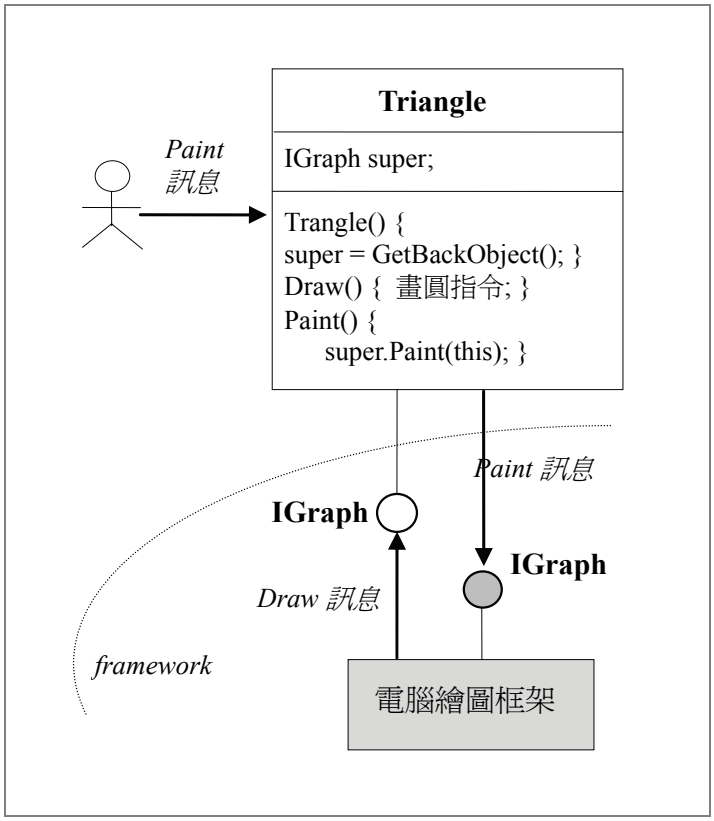

圖 1-5　黑箱框架的彈性：可有更多的服務

使用框架時，Triangle 應用類別設計者的腦筋裡不須要知道框架裡有 Graphic 類別，只要透過 GetBackObject()函數就可以取得框架裡的物件來協助其繪圖。這通稱爲黑箱(Black-Box)型框架(即您的客戶看不出來框架裡有個 Graphic 類別)，有一天您將 Graphic 類別名稱更改爲 GraphBackClass 類別時，您的所有客戶的應用類別皆不會受到影響。在 Android 應用框架裡，也很善用這樣的設計手藝。

與黑箱框架相對的，就白箱(White-Box)框架了。白箱框架採取程式語言(如 C++、Java 等)的繼承(Inheritance)機制，當程式進行編譯時，編譯程式(Compiler)建立框架內類別與應用類別之間的溝通介面。

白箱框架因爲有編譯程式的協助，所以比較容易設計，而且從藉由類別繼承機制，能有效減輕應用類別的程式複雜度，可加速應用類別的開發。例如，黑箱

框架不藉助於程式語言的繼承功能，所以黑箱框架設計者在缺乏編譯程式在編譯階段的協助，只得自己訂定雙向溝通的介面，還得在執行期間檢查多樣性元件是否支持框架所定義的介面，如果多樣性元件並未定(或忘了)實作該介面的話，框架就無法進行雙向溝通。這類在執行期間可能發生的錯誤，在白箱框架裡並不會發生，因爲編譯程式會檢查這類的錯誤情況。如果將上圖 1-3 改爲繼承機制，其結構就如下圖：

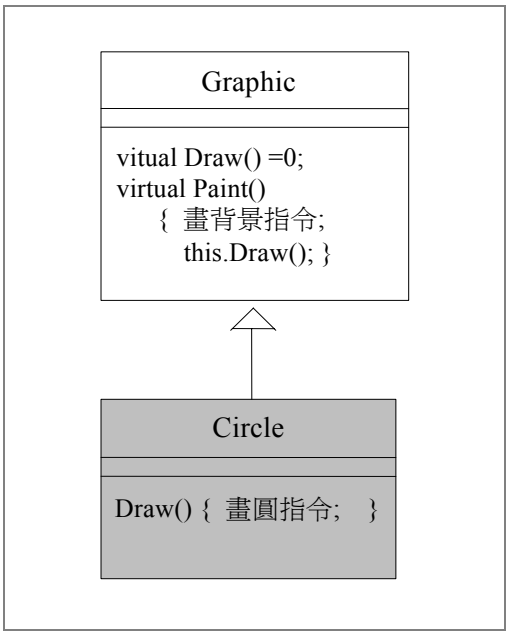

圖 1-6　採取白箱框架之設計

俗語說，天下沒有白吃的午餐。上述的白箱優點的背後也帶來一些不便之處(即缺點)。其中最主要的缺點是：應用類別與框架類別的相依性(Dependency)較高，獨立性與彈性隨之降低。就如 Fayad 在其書中所說[Fayad99a, p.17]：

> "Subclass are tightly coupled to their superclasses, so this way of using a framework requires more knowledge about the abstract classes than the first way. ... On the other hand, changes to the abstract classes can break existing concrete classes, and this way will not work when the main purpose of the framework is to build open systems."
>
> （因爲子類別繼承父類別，兩者之間具有緊密的相連性，所以使用白箱框時必

須對父類別有較多的瞭解才行。而使用黑箱框架時，就不須太費心去瞭解了。此外，當您更改父類別時，很可能會傷害到其已有的子類別。這種框架並不適合用來建構開放式的系統。）

黑箱框架就比較沒有上述這項缺點，但是黑箱框架必須自行定義雙向溝通的介面，卻增加了應用類別開發者的工作量。在 Android 應用框架裡，則採取灰箱(Gray-Box)框架設計，也就是混合了黑箱與白箱設計的優點，截長補短成爲精良的應用框架。至於，如何截長補短，並且能結合得天無縫呢? 這就有賴於本書的焦點：擅用設計樣式了。

參考資料

[高 97] 高煥堂(1997) *應用架構入門實例*, 物澤出版
[Fayad99a] Fayad, M.E., Schmidt, D.C. and Johnson, R.E. *Building Application Frameworks*, Wiley, 1999.
[Fayad99b] Fayad, M.E., Schmidt, D.C. and Johnson, R.E. *Domain-Specific Application Frameworks*, Wiley, 1999.
[Fayad99c] Fayad, M.E., Schmidt, D.C. and Johnson, R.E. *Implementating Application Frameworks*, Wiley, 1999.

1.5　舉例說明框架設計基本思維：

「變與不變之分離」

應用框架設計是基於其應用領域的知識(Domain Knowledge)。然而知識有許多種，其變化的來源和時間經常是不一致的。例如，一家餐廳，其基本菜色、材料大多能天天相同，其知識並不會隨著客人的不同而改變，我們稱它爲不變的。但是有些酸、甜程度等知識，就隨著客人而異了，而且在時間上必須等到客人來到時才知道，我們稱它爲會變的。所以，有關於客人些酸、甜程度等知識，與基本菜色、材料等知識的變化上是不一致的，其獲得的時間點也是不一致的。我們必須將兩者分離開來，並將其不變部份納入框架裡，則應用框架就於焉而成了。

在進行「變與不變之分離」時，必須秉持「知之爲知之，不知爲不知」的原則，明確敘述那些是已知的知識、那些是未知的知識、那些是善變的知識。設計師就依據這些敘述而決定那些部份應該留空白(就如杯子內挖空才能裝飲料)，那些類別應該分離，並定義介面，讓它們未來能隨時組合起來。

茲舉個例子來說吧！童話故事裡有位川頓國王，他有 7 個女兒，最小的女兒叫小美人魚。大家發現到，川頓國王嫁女兒的速度，愈來愈快。大家猜測其中的原因，主要是：嫁女兒的經驗、文件、物品等都能重複使用(Reuse)，能有效縮短「從訂婚到結婚的時間」，所以愈嫁愈快。除了經驗讓其駕輕就熟之外，有許多用品(例如籃子等)都不必再買了，喜帖也能在確定嫁給誰之前先印好，新郎名稱、日期、喜宴地點等部份預留空白即可了。待訂婚之後，知道結婚對象、日期、喜宴地點時，才填入喜帖即可了。

從上述可知，有些需求知識在結婚之前就已知(Available)了，於是設計師就依據這些知識而著手工作了。例如，已知需求有：

- 需要喜帖，喜帖的格式及某些內容。
  ----- 於是，設計師就依據這些知識而著手設計喜帖、印製喜帖。
- 需要新娘捧花，花的種類顏色。
  ----- 於是，設計師就著手設計(或委外設計)捧花、預訂捧花。
- 需要照片的相框，相框材質及大小。
  ----- 於是，設計師就先購買相框。

另一方面，也有些需求知識在結婚之前是未知(Unavailable)的。此時，設計師必

須預留空白、訂定介面、或只設計而不落實(Implement)等。例如，未知需求有：

- 新郎名字是未知的。
  ----- 於是，設計師就在其所設計的喜帖上預留空白。這就像老子設計畚箕時，因爲還未知農夫會拿去裝什麼東西，所以老子就將畚箕中間挖空(預留虛的空間)。
- 結婚日期未定，那一天需要新娘捧花是未知的。
  ----- 於是，設計師不能著手買花。也就是能先進行設計，但還不能實際執行(Implement)！
- 還沒有婚禮照片。
  ----- 於是，設計師購買的相框元件必須訂定有適當的介面，待婚禮之後，照片元件就能順利放入相框裡。

無論在時光隧道的任何一個定點，都有已知的需求、有未知的需求、也有過時的需求。設計師的素養是：在需求複雜多變、攸關知識不齊備的環境之下，依賴優越的洞悉力、判斷力，做出美好的設計決策。據說川頓國王的王室家族的結婚喜帖，有其特定的格式。小美人魚結婚了，婚後生了一個女兒，名叫：美樂婷。依據王室的喜帖格式，我們能「預知」她們的喜帖封面，如下圖：

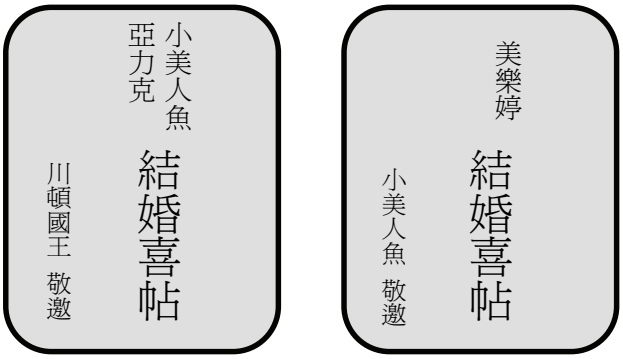

圖 1-7　王室家族的喜帖格式

從小美人魚結婚到她的女兒結婚，經過時間、空間、人物的轉換，設計師發

現到其中有些「會變」的部份，也有些是「不變」的部份。接下來，設計師就做些創造性動作：依據經驗和洞察力，將變與不變的部份分離開來，成爲數個元素(或組件)，如下圖：

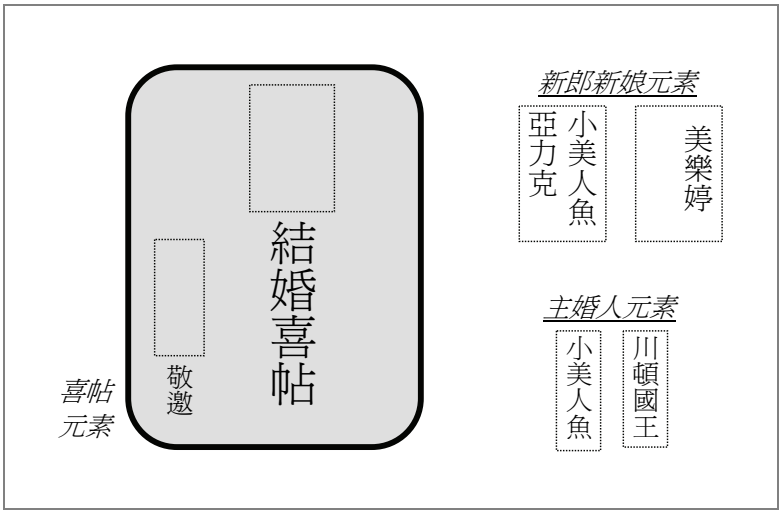

圖 1-8　分離出喜帖的小元素

其實，這種手藝也蠻簡單的，在數千年前的老子已經使用過了，他曾說：畚箕中間必須「挖空」才有用！ 雖然簡單，卻是千年不朽的設計手藝。而「變與不變之分離」就是這個簡單的設計思維罷了。

挖出來的會變部份成爲元素，但有趣的是：挖出之後留下的空間才是重點，它能容納原來被挖出的元素，也能容納未來的其它元素。依據老子的思維，這個挖空的喜帖元件看似「無用」，其實是「爲用大矣」！例如能跟 Kitty 貓元素相互結合，就成爲現實可用的喜帖了，如下圖：

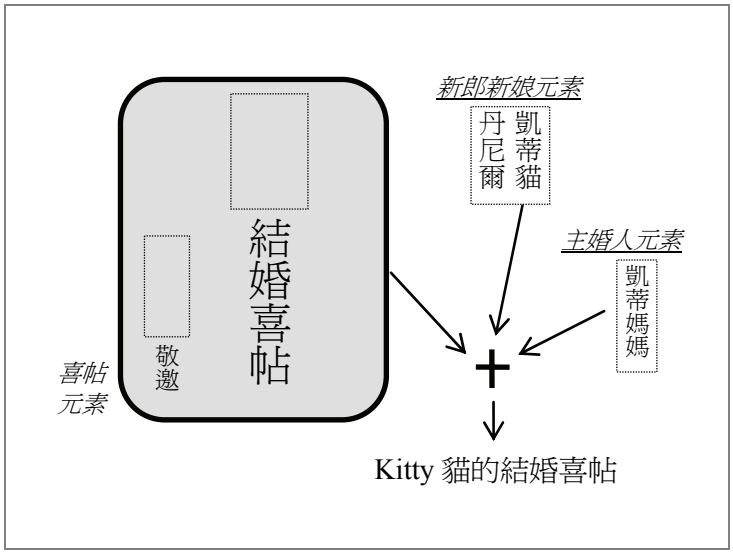

圖 1-9　不變元素與會變元素之搭配

以上是藉由童話故事來闡述框架設計的基本手藝。現在，來看看它在軟體設計上的運用。例如有兩個 Java 類別，各代表大學裡的「大學生」與「研究生」，如下圖：

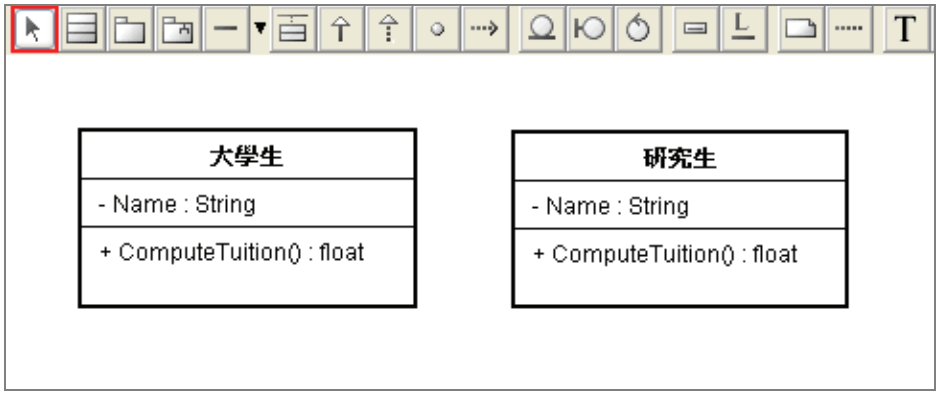

圖 1-10　兩個簡單的類別

茲以 Java 語言表達如下：

```
// 大學生.java
package _objects;
public class 大學生{
    private String Name;
    public float ComputeTuition(int credit) {
        if (credit > 6)
            credit = 6;
        return (credit -1) * 500 + 5000;
 }}

// 研究生.java
package _objects;
public class 研究生{
    private String Name;
    public float ComputeTuition(int credit) {
        if (credit > 6)
            credit = 6;
        return credit * 700 + 5000;
    }}
```

其中的 ComputeTuition()函數可計算出大學生或研究生的學費。現在，就運用本章前面各節所介紹的「抽象」步驟來打造介面。茲比較上述的兩個類別，看出其不一樣之處：

即「大學生」類別裡的指令 ----- (credit-1)*500
與「研究生」類別裡的指令 ----- credit *700

其餘部份則是一樣的。這導致兩個 ComputeTuition()不能直接擺入抽象類別中。現在設計一個 Overridable 函數：GetValue()抽象函數來封藏之，吸收其相異點，就能讓 ComputeTuition()飛上枝頭變鳳凰了，如下圖：

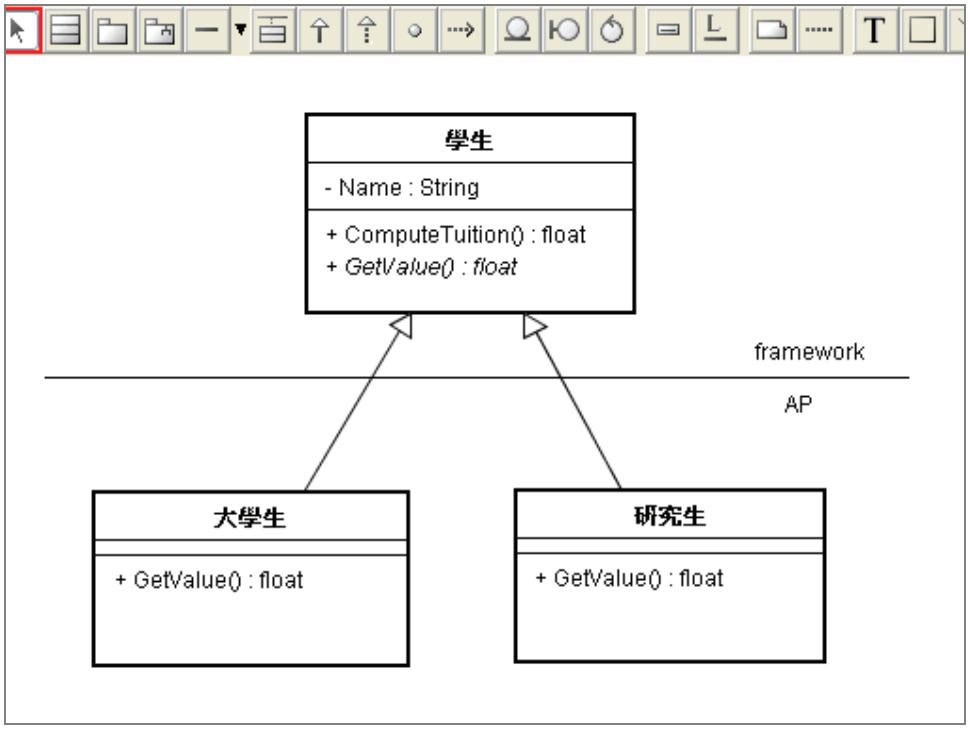

圖 1-11　變與不變之分離(白箱)

於是， 我們就可以將「學生」抽象類別納入應用框架裡了。茲以 Java 表達如下：

```
// 學生.java
package _student;
import _interface.*;
public class 學生 {
        private String Name;
       public float ComputeTuition(int credit) {
             if (credit > 6)
                    credit = 6;
             return tc.GetValue(credit) + 5000;
       }
       protected abstract float GetValue(int credit);
}

public class 大學生 extends 學生 {
```

```
        public float GetValue(int credit) {
                return (credit -1) * 500;          }
}

public class  研究生 extends  學生  {
        public float GetValue(int credit)
                {    return credit * 700;       }
}
```

以上是採取類別繼承的機制，也就是上一節所介紹的白箱框架設計。接著，換個角度，如果採取黑箱框架設計，又如何呢?此時，可將抽象函數 GetValue()獨立出來，單獨擺入一個抽象類別裡，就能爲純粹抽象類別了，也就能以介面表示出來，如下圖：

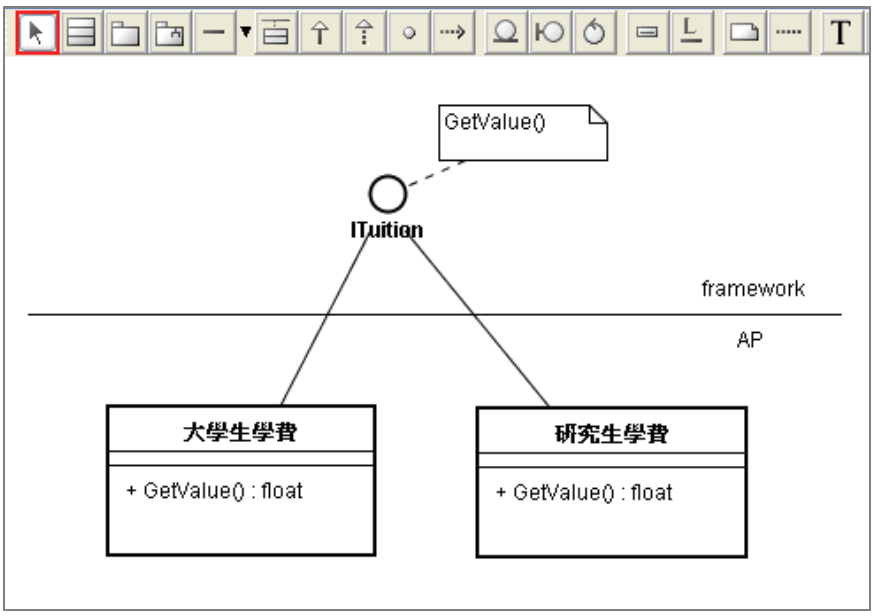

圖 1-12　變與不變之分離(黑箱)

於是，我們就可以將 ITuition 介面定義納入應用框架裡了。茲以 Java 表達如下：

```
// ITuition.java
package _interface;
public interface ITuition {    //學費介面
        public float GetValue(int credit);
}

// 大學生學費.java
package _tuition_plugin;
import _interface.*;
public class 大學生學費 implements ITuition {
      public float GetValue(int credit) { return (credit -1) * 500;    }
   }

// 研究生學費.java
public class 研究生學費 implements ITuition {
      public float GetValue(int credit)    {    return credit * 700;    }
   }
```

在框架裡已經有了 ITuition 介面了，就能改寫「學生」類別如下：

```
// 學生.java
package _student;
import _interface.*;
public class 學生 {
       private String Name;
       private ITuition tc;
       public void Setter(ITuition tuiObj)
              {    tc = tuiObj;  }
        public float ComputeTuition(int credit) {
            if (credit > 6)
                    credit = 6;
            return tc.GetValue(credit) + 5000;
   }}
```

這個「學生」類別裡的指令只用到 ITuition 介面，而沒有用到「大學生」或「研究生」應用類別。它只含有不變的部份，所以能將之納入框架裡，如下圖：

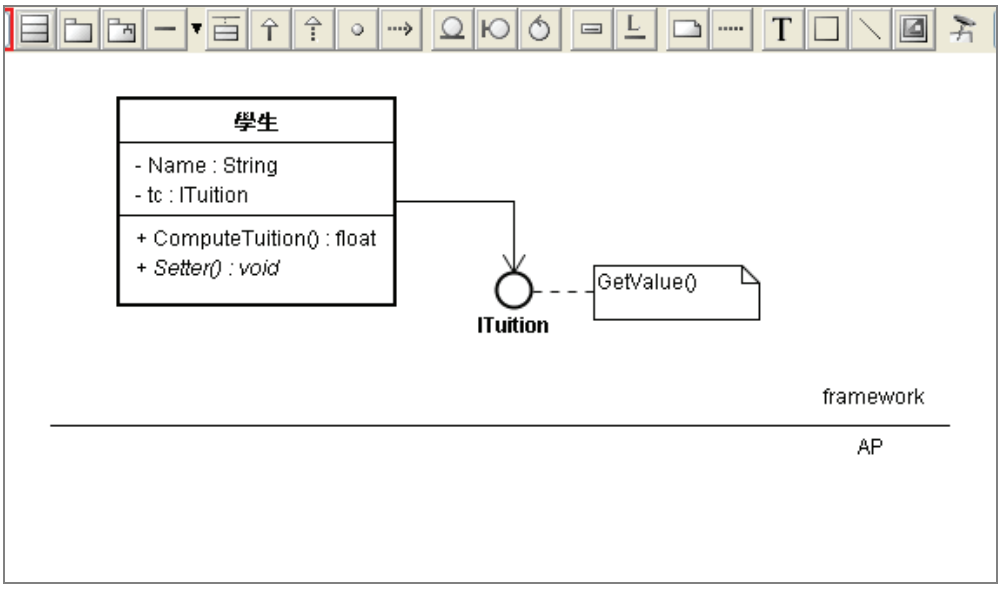

圖 1-13　簡單的黑箱框架之例

接著，基於上述的黑箱框架，就能將框架裡的「學生」類別與「大學生學費」應用類別結合起來了。此外也能將「學生」與「研究生學費」類別結合起來。於是經由多樣化的結合，而產生了各式各樣的軟體系統了。茲寫個 JMain 類別來促成上述的結合，如下之 JMain 類別：

```
// JMain.java
import _student.*;
import _tuition_plugin.*;
public class JMain {
        public static void main(String[] args) {
                float t1, t2;
                學生  Lily = new  學生();
                大學生學費  under_tui = new  大學生學費();
                Lily.Setter(under_tui);
                t1 = Lily.ComputeTuition(5);
                學生  Peter = new  學生();
                研究生學費  grad_tui = new  研究生學費();
                Peter.Setter(grad_tui);
                t2 = Peter.ComputeTuition(7);
```

```
        System.out.println( "Lily: " + String.valueOf(t1) + ", Peter: "
                                    + String.valueOf(t2));
    }}
```

茲以圖形表示如下：

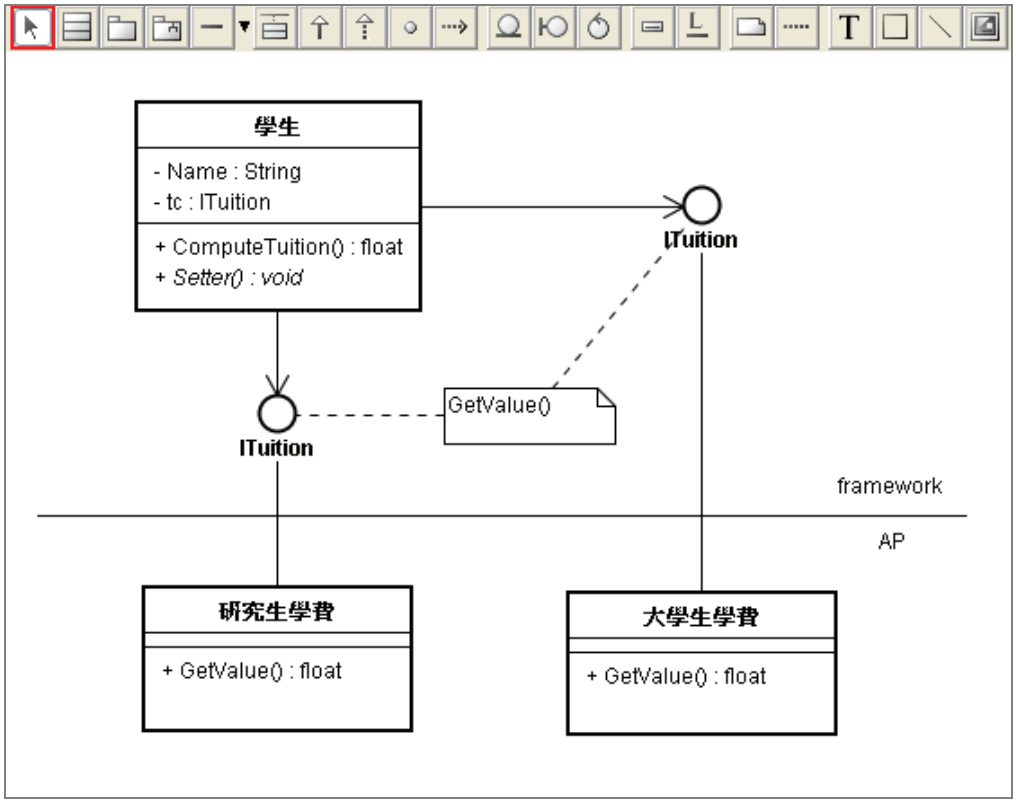

圖 1-14　框架之應用：易於創造出多樣化之組合

根據聖經故事，上帝從亞當分離出肋骨而創造出夏娃，讓亞當與夏娃的後代得以多樣化結婚(合)，且讓人類子孫繁榮至今。同樣地，Android 框架裡的類別與應用類別的多樣化結婚(合)，也將讓多樣化的軟體繼茁壯、無限繁榮。至於，到底有那些美妙的結合方式呢? 那就是本書的焦點：「設計樣式」了。

1.6 欣賞 Android 框架裡的 13 個樣式

1.6.1 框架與樣式之微妙關係

框架(Framework)與樣式(Pattern)都是抽象的事物，所以很容易讓人弄不清楚兩者之微妙區別。其實兩者是很容易區別的。

框架是一種特殊領域(Domain)裡的共同架構。例如 Android 就是手機應用軟體的共同架構，它只限於「手機」領域而已(Domain-Specific)，會因不同領域而變，只是它在該領域裡並不會隨著應用程式而改變。

樣式是針對特定問題(Problem)的共同(的抽象)解決方案(Solution)。無論是框架或是其他架構設計，只要遇到類似的問題，都可以使用該共同的抽象(Abstract)解決方案，配合當時的環境元素(Context)，就能得出美好的具體(Concrete)解決方案，就能解決實際的問題了。

就 GoF 的設計樣式來說，手機領域的 Android 框架正是設計樣式的美好應用環境(Context)。舉凡在 Android 框架的設計過程中，當遭遇類似的設計問題時，就能善加發揮 GoF 設計樣式的威力了。換言之，框架與樣式是基於兩種互補的抽象角度，所以兩者能互補，且密切結合，不但能化解設計難題，還能提供高雅的設計結構，創造精緻的軟體系統架構。

1.6.2 簡介本書的 13 個設計樣式

GoF 的 *Design Patterns* 書裡共列舉了 23 個設計樣式。由於本書的焦點在於 Android 應用框架設計(無論你是設計者，還是鑑賞者)，所以只選擇其中比較常用於框架的 13 個設計樣式，做爲本書探討的主題。茲逐一簡介如下：

#1. Template Method 樣式

框架設計的基礎功夫有二：

- **變與不變分離**(Separate code that changes from the code that doesn't.)。不變部分歸於抽象類別，會變部份歸於(具象)應用類別。然後將抽象類別納入框架

中。

- **反向控制**(IoC: Inversion of Control)。分離變與不變之後，抽象類別「反向呼叫」應用類別的函數。框架的抽象類別通常開發在先，然後才搭配應用類別，組成應用軟體系統。此時，抽象類別掌握軟體執行的控制權，所以稱爲反向控制。Template Method 樣式是實現框架反向呼叫的基本招式。

#2. Factory Method 樣式

框架裡的抽象類別，不能誕生物件。應用類別是具體類別，才能拿來誕生物件。但是抽象類別可以透過反向呼叫來誕生應用類別之物件。Factory Method 樣式就實現這種特殊的反向呼叫。

#3. Observer 樣式

上述的 Template Method 樣式主要是依賴類別繼承來實現反向控制，那是白箱框架的實現機制。至於黑箱框架設計上，因爲不採用繼承機制，所以 Template Method 樣式並不合適。此時 Observer 樣式就成爲主要的實現招式了。

#4. Abstract Factory 樣式

上述的 Factory Method 樣式是透過繼承機制要求子類別去誕生應用類別之物件，那是白箱框架的實現機制。至於黑箱框架設計上，因爲不使用繼承機制，所以 Factory Method 樣式並不合適，而 Abstract Method 樣式就成爲主要的實現招式了。

#5. Adapter 樣式

Adapter 物件在框架裡扮演轉接器之角色，其主要用途是改變介面。介面相當於純粹抽象類別(Pure Abstract Class)，適合定義於框架裡。Adapter 樣式讓框架裡的抽象類別能誕生 Adapter 應用類別之物件，來實現定義於框架之新介面。

#6. Composite 樣式

Composite 樣式讓我們能把不變的遞迴(Recursive)性聚合(Aggregate)關係定義於框架的抽象類別之間；以協助應用類別建立複雜的樹狀結構。

#7. Strategy 樣式

框架裡的抽象類別，常會視不同應用場合而選擇(或搭配)不同的策略。各個策略皆定義於應用策略類別裡，Strategy 樣式讓框架裡的抽象類別能隨時抽換不同的應用策略類別。

#8. State 樣式

應用軟體在其生命週期中，會歷經各種不同的狀態(State)。各個狀態皆定義於狀態類別裡，State 樣式讓框架裡的抽象類別能隨時抽換不同的狀態類別。

#9. Proxy 樣式

上述 Adapter 樣式的用意在於改變介面，讓 Client 覺得很容易使用。Proxy 樣式很像 Adapter，但其扮演分身，而且提供相同介面。框架常藉由 Proxy 樣式來封裝 Proxy 與 Server 之間的複雜 IPC 遠距通訊。

#10. Bridge 樣式

框架裡的抽象類別可衍生成爲一個類別體系。此體系之類別的實作(Implementation)部分獨立出來，委託給另一個類別體系。Bridge 樣式提供機制以扮演這兩個類別體系間之溝通橋樑。

#11. Iterator 樣式

框架常需要提供一致的介面，讓 Client 瀏覽資料結構裡的一群資料。Iterator 樣式提供了絕佳機制來滿足這項需要。

#12. Mediator 樣式

USB 隨身碟可以插在硬體的主機板(Motherborad)上。Mediator 樣式讓框架成

爲軟體主機板，讓應用類別更容易搭配到框架上。

#13. Façade 樣式

四合院建築物提供一個門面，也含有舒暢的院子和房屋，給予人們成長的空間。框架常藉由 Façade 樣式提供一致的介面，讓其幕後子系統(含有許多類別)擁有高度的變動及成長空間。

1.6.3　結語

俗語說，外行人看熱鬧，內行人看門道。例如，觀看高手下象棋時，常常覺得深奧莫測。如果有位內行人替你解釋，告訴你高手所用的招式(例如「將軍抽車」)，你會恍然大悟曰：妙招妙招! 同樣地，觀看一座宏偉大教堂時，只能看看表象而已。如果有位內行人替你解釋，告訴你教堂建築師所用的招式(例如「巴洛克之窗」)，你就不再是外行人看熱鬧，而是內行人看門道了。Android 框架就如同一座宏偉大教堂，本書就爲你講解 Android 框架的設計招式，你就不再是外行人看熱鬧，而是內行人看門道了。

於是，閱讀本書後，你將能從設計招式而深刻認識 Android 的高雅氣質，又能從 Android 精緻設計中領悟設計招式之美。然後合卷而嘆曰：此曲只應天上有，人間難得幾回聞!◆

(SPACE)

(SPACE)