

Design Patterns

設計範式

(14*3=42 hours)

- 學習高層次溝通與思考能力
- 學習前人經驗
- 檢討自身技術

侯捷
2006/02-06
元智大學資訊工程系





Bibliography

GOF : Gang of Four

pattern :

- *. 花樣,圖案,形態,樣式
- *. (服裝裁剪的)紙樣,(澆鑄用的)模具
- *. 模範,榜樣,典型

Design Patterns

物件導向
設計模式

(葉秉哲譯, 2001)



Bibliography

Refactoring

重構

改善既有程式的設計

(侯捷 / 熊節譯, 2003)



Bibliography

Refactoring to Patterns

重構
三步曲

— 七種式前進 —

Adapter, Builder, Command,
Composed Method,
Creation Method,
Decorator,
Factory Method,
Interpreter,
Iterator,
Null Object,
Observer,
Singleton,
State,
Strategy,
Template Method,
Visitor.

侯捷 / 陳裕城 譯, 2005



Bibliography

Agile Software Development

Abstract Server,
Active Object,
Adapter,
Bridge,
Command,
Composite,
Facade,
Factory,
Mediator,
Mono State,
Null Object,
Observer,
Proxy,
Singleton,
Stairway to Heaven,
State,
Strategy,
Template Method,
Visitor,

《敏捷軟體開發》
林昆穎 / 吳京子 譯, 2005



Bibliography

- Virtualizing Constructors and Non-member Functions,
- Limiting the Number of Objects of a Class,
- Requiring or Prohibiting Heap-Base Objects,
- Smart Pointers,
- Reference Counting,
- Proxy Classes,
- Making Functions Virtual with Respect to
More Than One Objects (Double Dispatching)

Effective C++

**More
Effective C++**

(侯捷 譯, 2000)



Bibliography

C++設計
新思維

侯捷 / 於春景 譯, 2003

- Generalized Functors,
- Singleton,
- Smart Pointers,
- Object Factories,
- Abstract Factory,
- Visitor,
- Multimethods,

**Modern
C++ Design**

Design Patterns

於Java語言上的實習應用

《Design Patterns
於Java語言上的實習應用》
by 結城 浩

<http://www.hyuki.com/dp/index.html>
http://www.hyuki.com/dp/dpsrc_2004-05-26.zip
採 The zlib/libpng License，不限制包括
商業應用在內的任何用途。



Bibliography

侯捷 / 王飛 / 羅偉 譯, 2003

**Small Memory
Software**

記憶體受限系統
之程式開發

內存受限系統之
軟件開發

1. Small Architecture

Memory Limit, Small Interfaces, Partial Failure, Captain Oates, Read-Only Memory, Hooks

2. Secondary Storage

Application Switching, Data Files, Resource Files, Packages, Paging

3. Compression

Table Compression, Difference Coding, Adaptive Compression

4. Small Data Structures

Packed Data, Sharing, Copy-on-Write, Embedded Pointers, Multiple Representations

5. Memory Allocation

Fixed Allocation, Variable Allocation, Memory Discard, Pooled Allocation, Compaction, Reference Counting, Garbage Collection



課程綱要 (42 hours)

見本課程網頁 <http://www.jjhou.com/course-dp-yz-2006.htm>



前言：誰適合聽這門課

要聽懂 Design Patterns，只需對 OOPL (C++ 或 Java...) 的 polymorphism (幾乎可以說就是 virtual functions 的應用) 有所認知並加上不算太少的 實作經驗（愈多愈好）即可。

但是要「領略」design patterns 就不是那麼容易。我常形容寫過 10 萬行 OOP codes 就可以對 design patterns 心領神會。10萬行也可以改為 3 萬行或 5 萬行。總之是這樣一個 order.

學生階段寫過上萬行 OOP codes 實屬鳳毛麟角，但這不成爲上這門課的門檻或阻礙。我儘可能抽取大型 libraries 的 source code 做爲講解與分析的內容，這會讓大家比較容易領受。只要 OOP 的基礎不差，聽懂應該是沒問題的。「心領神會」則看各人修爲。

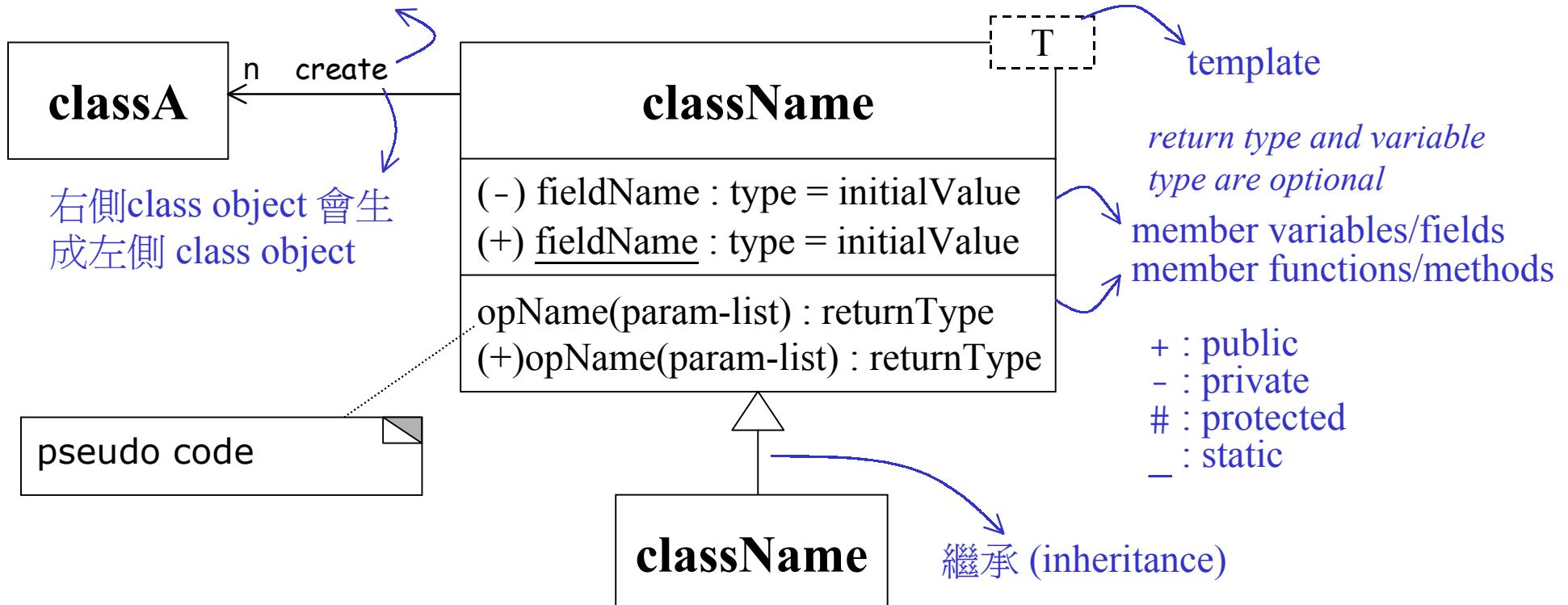


UML, class diagram

class diagram 表現不因時間而變化的部分

classes 之間的線條代表 **relationship**，亦即代表彼此可以傳遞訊息（互相呼叫）。最常以 **pointer to-** 或 **reference to-** 另一個 class 來實現。若帶箭頭表示 **navigable relationship** “ 被箭頭所指之 class 不認識其夥伴。

除了 **create** 之外也可以是 **use** 或 **notify...**



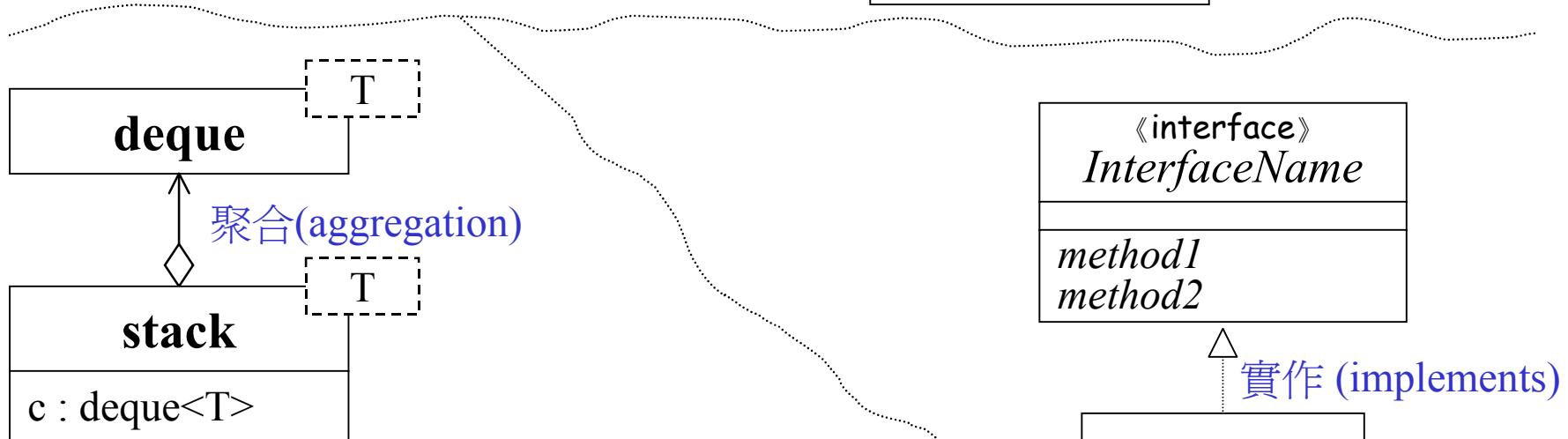


UML, class diagram

斜體字表示 abstract :

AbstractClass

《interface》
InterfaceName



aggregation 是一種特殊形式的 **relationship**，以白色菱形表示，帶有 "整體/成分" 意涵。緊臨白色菱形的是 "整體"，另一端是 "成分"。這種關係是隱喻的 (implicitly)。

"飛機場內有飛機" 是 aggregation
"汽車內有引擎" 是 composition

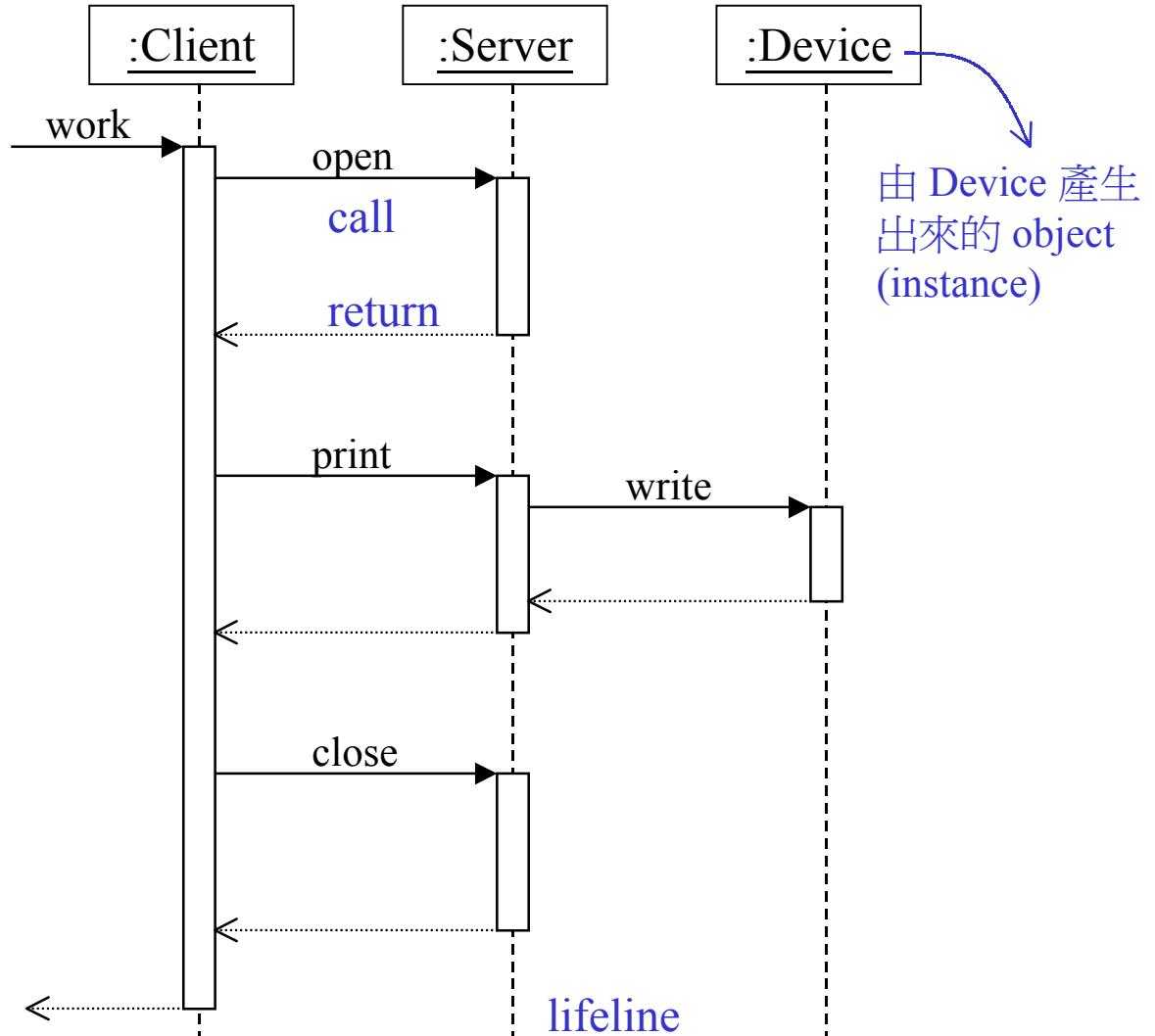
combination/containment/composition 是一種特殊形式的 **aggregation**，以黑色菱形表示，暗示 "整體" 要負責掌控其 "成分" 的壽命。“不含生成或刪除責任，而是（例如）"整體" 必須知道 "成分" 被刪除了。也就是說可直接刪除 "成分" 亦可將 "成分" 傳給另一個對其負有責任的實體。



UML, sequence diagram

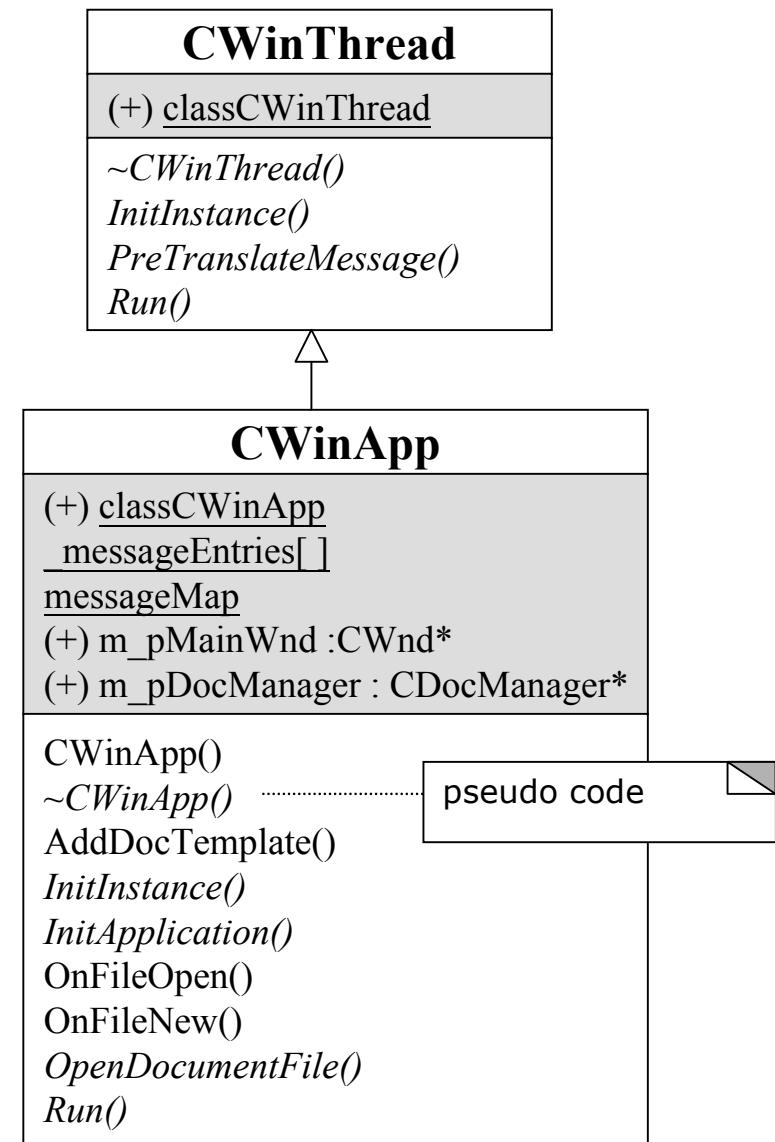
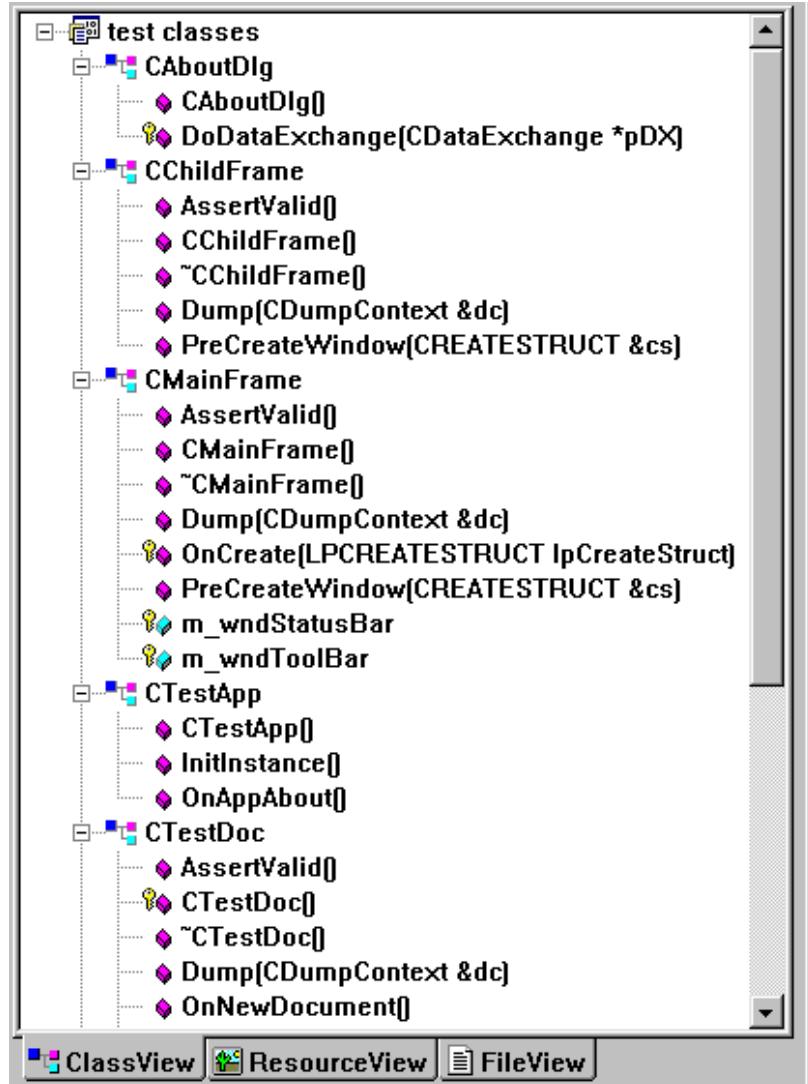
sequence diagram 表現隨時間而變化的部分

```
class Client {  
    Server s;  
    void work() {  
        s.open();  
        s.print("Hello");  
        s.close();  
    }  
    ...  
}  
  
class Server {  
    Device d;  
    void open() {  
        ...  
    }  
    void print(String s) {  
        d.write(s);  
        ...  
    }  
    void close() {  
        ...  
    }  
    ...  
}  
  
class Device {  
    void write(String s) {  
        ...  
    }  
}
```



UML classes diagram, Ex

有些 IDE 提供 UML-like diagram





Classic OO terms

object所宣告的operations都必須有name, parameters, return value，這三者就是所謂的operation **signature**（簽名式）。object所定義的一切signatures，就是該object的對外介面，也就是**interface**。

- **class, object, operations(methods), states(data, fields), message(request), dispatching**
- **signature, type, subtype, supertype, interface, implementation**
- **implementation inheritance (class inheritance)**
- **interface inheritance (subtyping)**

implementation inheritance 可共享/繼承 code（程式碼）和 representation（內部表述，亦即data或states）；**interface inheritance** 則描述object在什麼情況下可被其他object替代。這兩個觀念很容易混淆，因為許多語言並不明顯區別它們。C++ 的繼承動作同時涵蓋interface inheritance和implementation inheritance兩者。如果要在C++ 中只繼承interface，標準作法是以public方式繼承一個只擁有純虛擬函式的class；如果只是純粹想要繼承implementation，可以使用「private繼承」近似之。

OO程式由objects構成。object將**data**和可施行於該**data**之上的**operations**包裝在一起。客端發出**request**（或曰**message**），就會令收受者執行對應的**operation**。就語言層面而言，對著一個object喚起其某個**method**，也就是向該object發出**message**之意。

class定義的是object的內部狀態（internal states）和各操作/動作的實務細節（operations implementation）。**type**定義的只是**interface**，也就是「object所能反應的一組requests」。一個object可以有許多**types**，不同**classes**的**objects**可以有相同的**type**。

第一個 OO 原則：多用 interface inheritance，少用 implementation inheritance。
世上沒有完美的設計，必須視你的系統目標而定。

第二個 OO 原則：多用 object composition，少用 class inheritance。
但請注意，該是 inheritance 發揮之處，也無法強求於 composition。



Classic OO terms

class inheritance（類別繼承）和**object composition**（物件複合）是OO系統中最常見的兩項復用技術。前者是所謂的**white-box reuse**（白箱式復用），意味parent classes的內部對subclasses可見，無封裝性。後者是所謂的**black-box reuse**（黑箱式復用），被組裝物件（composed objects）需有定義明確（well-defined）的interfaces。

- **class (public) inheritance, *is-a***
- **object composition, *has-a***
- **delegation**
- **association (acquaintance or using)**
- **polymorphism**
- **Liskov Substitution Principle**

class inheritance在編譯期就定義妥當，因為它直接被語言支援。缺點是你無法在執行期改變繼承自parent classes的那些implementations。**object composition**可透過references to other objects，在執行期動態獲得複合事實。

當request被送至object身上，究竟哪個operation會被喚起，端視「什麼樣的request」以及「誰收到request」而定。執行期將一個 request和一個object operation結合起來，稱為**dynamic binding**（動態繫結）。dynamic binding意味，發出一個request並不就等於委託執行某一明確的implementation。dynamic binding允許你在執行期間以某些objects替代「擁有相同interface」的其他任何objects。這種替換性質稱為**polymorphism**（多型）。這是OO系統的一個關鍵概念。在C++中**polymorphism**（多型）一定得配合pointer或reference才能進行。

Liskov Substitution Principle 說：public base class object派得上用場的地方，一定可以採用其derived class object替代之

delegation（委派）是一種作法，可使composition像inheritance那麼具有復用威力。此法由兩個 objects 共同處理一個request：接收者（receiver）授權被委託人完成處理動作。這和「subclasses將requests推遲至parent classes去處理」頗為異曲同工。在繼承關係中，透過C++的'this'，被繼承的operation總是可以指涉（取用）receiver；如果我們希望在delegation中也達到相同效果，receiver必須將自己交給被委託者，期使被委託者得以指涉（取用）receiver自己。delegation的主要優點是可以非常輕易地在執行時期組合各種行為，並改變組合方式。例如Window可以變成圓的，只要將其內的Rectangle實體取代為Circle實體。“前提是Rectangle和Circle擁有相同的type。



Classic OO terms

parameterized types是OO系統中除了**class inheritance**和**object composition**之外的第三種行爲組合方法。三種方法的重要差異是：(1) object composition 允許你在執行期改變組合行爲，但它需要間接性，因此影響效率。(2) inheritance 允許你提供預設的operations implementation並讓subclass覆寫之。(3) parameterized types允許你改變class所能使用的types。哪一種方法比較好，視設計條件和實作條件而定。

- parameterized types (template, generic)**
- framework and application framework**

Framework是一組彼此合作的 classes，針對某類型軟件組成一個可復用的設計。Framework 將設計分割為 abstract classes 並定義它們的責任與合作，因而提供了結構性的導引（architectural guidance）。軟件開發者可藉由 subclassing and composing instances of framework classes 來定制 (customizes) framework，使其成為一個特定應用 (particular application)。

Application Framework 是特別用於 application architecture (而非其他主題如 memory, data structures, algorithms...) 的一種 framework。



Classes in C++ and in Java

Java code in JDK1.42

C++ code in MFC 4.2

```
class COBList : public CObject
{
protected:
    struct CNode
    {
        CNode* pNext;
        CNode* pPrev;
        CObject* data;
    };
public:
    COBList(int nBlockSize = 10);
    int GetCount() const;
    BOOL IsEmpty() const;
    CObject* RemoveHead();
    CObject* RemoveTail();
    void RemoveAll();
    ...
protected:
    CNode* m_pNodeHead;
    CNode* m_pNodeTail;
    int m_nCount;
    int m_nBlockSize;
    ...
public:
    ~COBList();
    void Serialize(CArchive&);
```

```
public class LinkedList extends AbstractSequentialList
    implements List, Cloneable, java.io.Serializable
{
    private transient Entry header = new Entry(null, null, null);
    private transient int size = 0;

    public LinkedList() {
        header.next = header.previous = header;
    }

    public Object getFirst() {
        if (size==0)
            throw new NoSuchElementException();
        return header.next.element;
    }

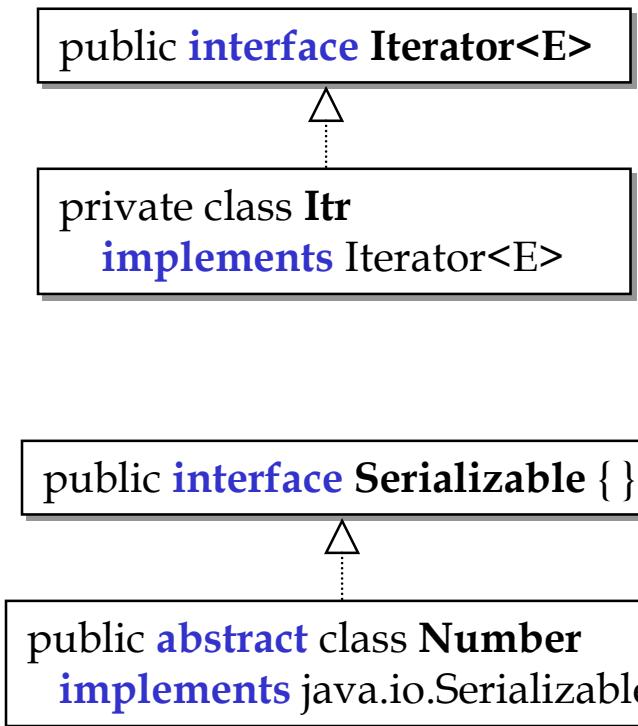
    public Object get(int index) {
        return entry(index).element;
    }

    private static final long serialVersionUID
        = 876323262645176354L;

    ...
}
```



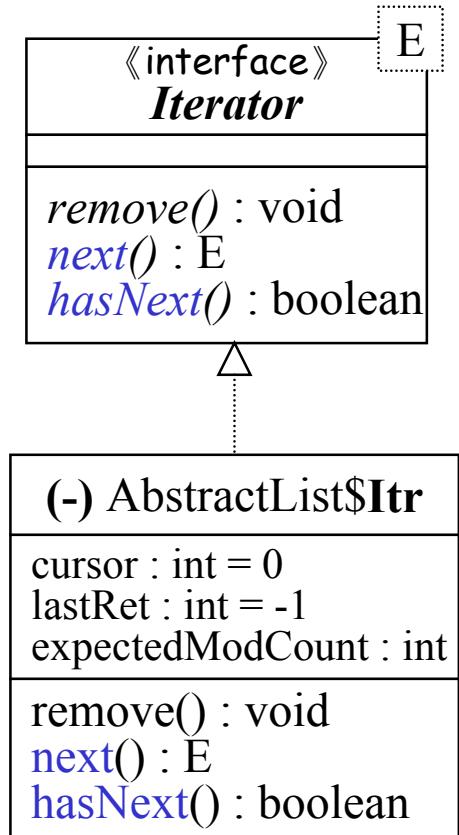
Interface and Implements



```
public interface Iterable<T> {
    Iterator<T> iterator();
}

public interface Collection<E>
extends Iterable<E> {
    int size();
    boolean isEmpty();
    Iterator<E> iterator();
    boolean add(E o);
    boolean remove(Object o);
    ...
}
```

Interface 可以
繼承 interface



interface 造出完全抽象的 class，不帶半點實作內容。

- Java **Interface** 可被多重繼承。
- C++ 以所有 member functions 都是 pure virtual 的 class 來表現 interface。



Abstract class

含有 abstract methods 者稱爲 *abstract class*。如果 class 含有一或多個 abstract methods（只有宣告而無定義者）就表示尚未完全定義，因此不允許產生實體，因此必爲 abstract class，便需以關鍵字 **abstract** 做爲飾詞，否則編譯器會報錯。如果繼承 abstract class 並希望爲新型別產生 object，那麼得爲其所有 abstract methods 都提供定義。如果沒有這麼做 derived class 便也成爲一個 abstract class，而且編譯器會強迫你以關鍵字 **abstract** 來修飾這個 derived class。也可以將不含任何 **abstract** methods 的 class 宣告爲 **abstract**。那麼它就不得被產生出任何實體。

- C++ 不提供關鍵字 **abstract**，以帶有 pure virtual function(s) 之 class來表現 abstract class.
- Java 提供關鍵字 **abstract**。

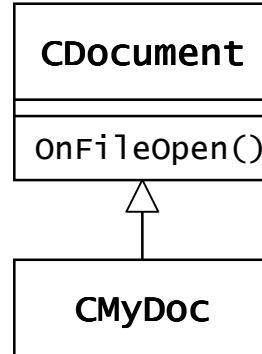
```
public abstract class AbstractList<E>
    extends AbstractCollection<E>
    implements List<E> {
    abstract public E get(int index);
    ...
}
```



Inheritance

C++

```
class CMyDoc : public CDocument  
{  
    ...  
};
```



Java

```
public class ArrayList<E> extends AbstractList<E>  
    implements List<E>, RandomAccess, Cloneable, java.io.Serializable
```



Reference in C++ and in Java

```
void func1(Cls* pobj) { pobj->xxx(); }
void func2(Cls obj) { obj.xxx(); }
void func3(Cls& obj) { obj.xxx(); }
```

被呼叫端寫法相同，很好。

Cls obj;

func1(&obj); —— 介面不同，困擾。

func2(obj);

func3(obj); > 呼叫端介面相同，很好。

reference 通常不用於變數的修飾，
而用於參數和回返型別的修飾。

使用 *by reference* 時，既享受
by pointer 的好處，又保持介面和
by value 時不變。

新建的objects乃配置於一塊被稱為heap的系統記憶體中。所有objects都經由object references進行存取。任何看起來持有object的變數，事實上內含的是一個「指向該object」的reference（此處可稱之為址參器）。這種變數的型別被稱為reference型別，對比於基本型別（其變數儲存的是該型別的值）。Object references如果不指向任何object，即為null。

大部分時候你可以模糊對待「實際的objects」和「references to objects」之間的區別。當你真正的意思是「傳遞一個object reference給某函式」時，你可以說成「傳遞一個object給某函式」。只有在兩者的區別造成影響時我們才謹慎對待之。大部分時候你可以互換使用object和object reference這兩個詞。

Java 有指標嗎？比較精確的說法是，Java有指標。是的，Java 之中除了基本型別，每個object的識別名稱都是指標。但它們的作用是受限的，不僅受到編譯器的保護，也受執行期系統的保護。換句話說 Java 有指標但是沒有指標運算。這正是 reference，可被想像為安全的指標。

那麼，Java 是 pass by value 抑或 pass by reference 呢？



Generics in C++ and in Java

Java code in JDK1.5.0

C++ code in C++ Standard Library (GCC 2.91)

```
template <class _FLT>
class complex
{
public:
    complex (_FLT r = 0, _FLT i = 0): re (r), im (i) { }
    complex& operator += (const complex&);
    complex& operator -= (const complex&);
    complex& operator *= (const complex&);
    complex& operator /= (const complex&);
    _FLT real () const { return re; }
    _FLT imag () const { return im; }
private:
    _FLT re, im;
};

class complex<float>;
class complex<double>;
class complex<long double>;
```

```
public interface Iterator<E> {
    boolean hasNext();
    E next();
    void remove();
}
```



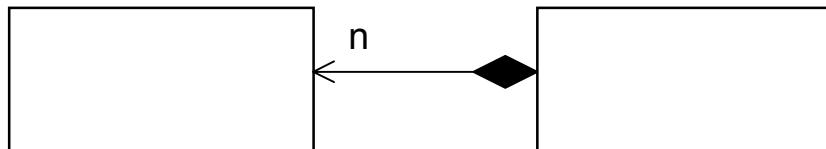
Composition

又名：containment, embedding, layering, has-a, combination

- 所謂複合，是指 class object 中有一些 class members。例如：

```
class CMyDoc : public CDocument
{
private:
    COBList myList;
};
```

- 我們說 object A擁有 (*own, have*) object B，或說 A 擔負/負責/承擔責任 (*is responsible for*) B，或說 B 是 A 的一部分 (*a part of*) 。A 和 B 的生命期完全相同。



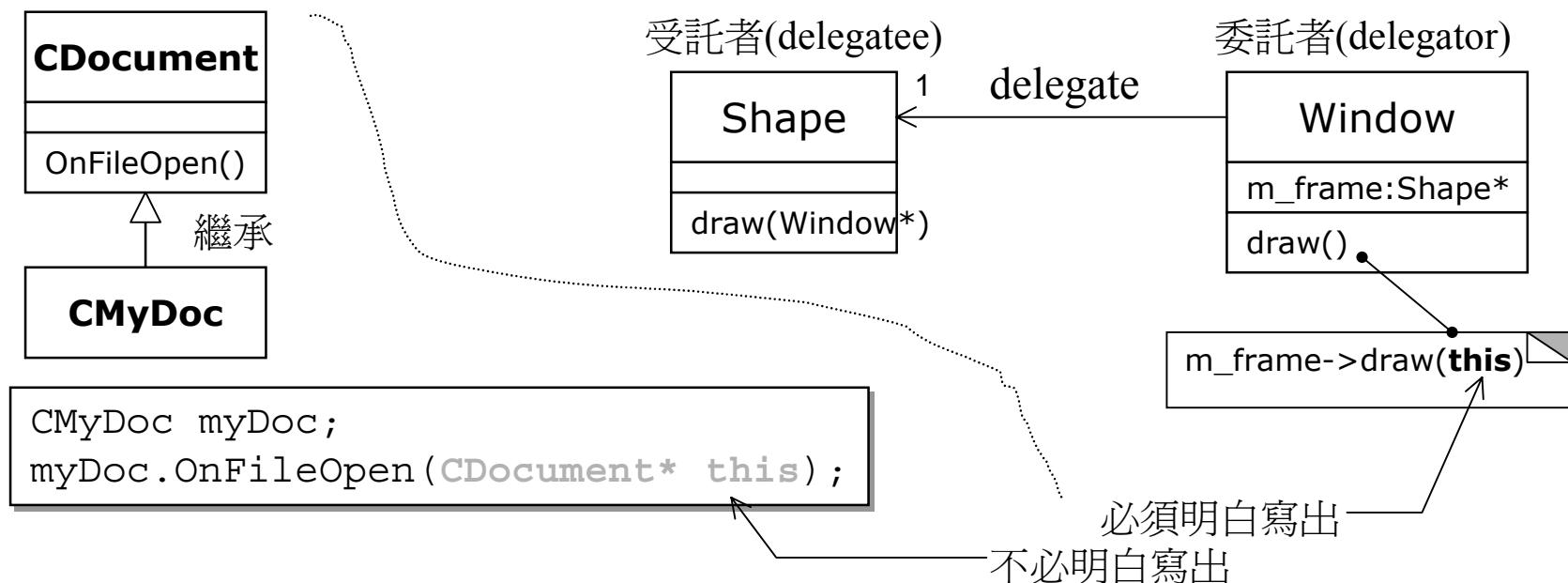
也有人稱 composition 為 aggregation。但細緻地說並不相同。

Java 雖然邏輯上有 composition，但實際上 (object model) 應該都屬於 delegation，因為 Java fields 都是 object reference，並沒有 stack object fields。



Delegation

在此技術下，兩個objects協同處理 message request。訊息接收者 (delegator) 授權給 delegatee (受託者) 全權處理訊息。這和「subclass 將訊息回應動作委託 base class 完成」的情況有一點兒類似。然而，在 Inheritance 情況下 base class 內的操作函式透過 'this' ptr 便能處理 subclass (訊息接收者) 本身。但在 delegation 情況下，delegator 必須將自己以參數型式傳給 delegatee，後者才有機會接觸 delegator 本尊。





SRP (Single-Responsibility Principle) 單一職責守則

ref 《Agile Software Development, Principles, Patterns, and Practices》

class 變更的原由應僅只一種

為什麼將兩個職責分離至個別的 classes 很重要？因為每一個職責都是一個改變的核心，當需求改變，這種改變會透過 class 間的職責變化而顯露出來。如果一個 class 擔負一個以上的職責，那麼它改變的理由就會超過一種。然而 class 變更的原因應僅只一種。

如果一個 class 有一個以上的職責，這些職責就會耦合在一起，於是改變一個職責就有可能損害或抑制此 class 滿足其他職責的能力。這一類耦合導致脆弱的設計，會在變更時以不可預期的方式出問題。

如果你能想到改變 class 的動機超過一個，那麼這個 class 就具有多於一個的職責。這有時很難辨別，我們都習慣以群組方式來考慮職責。我們會自然而然地結合職責，而實際軟件設計的精神就是找出並分離這些職責。但從另一方面說，如果應用程式的改變總會導致兩個職責同時改變，那就沒有必要將它們分開。更確切地說，把它們分開會引發「不必要的複雜度」。

如果只為毫無癥兆的事而採用 SRP 或其他守則，是不智之舉。



OCP (Open-Closed Principle) 開放封閉原則

軟件實物（classes、modules、functions...）對擴充應保持開放，對修改應維持封閉。

所有系統在它的生命週期中都會改變。如何才能在面對變動的情況下創造出穩定且可長久延續的設計呢？Betrand Meyer 在 1988 年創造了聞名當今的 *OCP*。

如果 *OCP* 應用得宜，未來的變更是以增添新程式碼達成，而不是修改運作正常的舊有程式碼。*OCP* 具有兩個主要特性：

- 對擴充開放**（Open for extension）。這表示模組的行為可擴充。也就是說我們可以改變模組所做的事。
- 對修改封閉**（Closed for modification）。擴充模組行為並不會對模組源碼或二進制碼造成改變。模組的二進制可執行版本不論是 linkable library 或 DLL 或 Java jar，都不會改動。

擴充模組行為的典型方法是修改模組源碼，不能修改的模組通常被認為具有固定的行為。怎麼可能不修改源碼而還能改變行為呢？**抽象化是關鍵所在**。

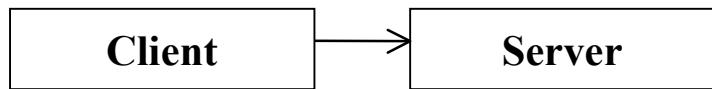
ref 《Agile Software Development, Principles, Patterns, and Practices》



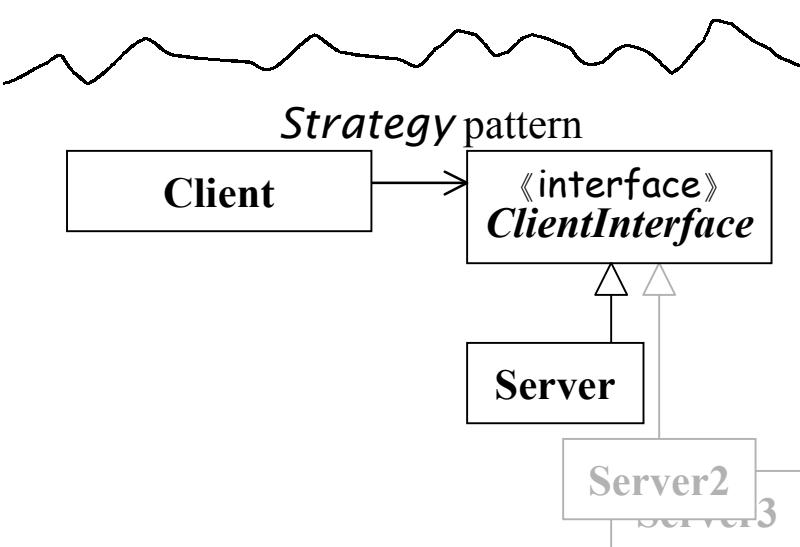
OCP (Open-Closed Principle) 開放封閉原則

在任何 OOPL 中都可以創造出「固定並且代表一組無限可能的行爲」的抽象概念。這些抽象概念就是 abstract base classes，而那組無限可能的行爲則以所有可能的 derived classes 來表現。

模組可以操作抽象概念。由於依存於固定之抽象概念，所以模組可對修改保持封閉，而又可透過創造抽象概念之 new derived classes 對模組的行爲加以擴充。



不符合OCP。Client 和 Server 都是 concrete classes。如果我們希望 Client object 使用不同的server object，那麼 Client class 內就必須改用不同的 server class 名稱。



符合OCP。ClientInterface 是個具有 abstract methods 的 abstract class。Client 使用這一抽象概念，然而 Client object 實際用到的將是 Server object。若欲 Client object 使用不同的 server object，可建立 ClientInterface 的其他 derived classes。於是 Client 可維持不變。

ref 《Agile Software Development, Principles, Patterns, and Practices》



OCP (Open-Closed Principle) 開放封閉原則

Template Method 和 *Strategy (Policy)* 是滿足 OCP 的最常見手法，它們都表達了一個清晰的分離 (separation) 概念，將通用功能性從該功能的實作細節中分離出來。

遵循 *OCP* 的代價是昂貴的，需要花費開發期的時間和人力來創造適當的抽象概念。這些抽象概念也增添了軟件設計的複雜度，而開發人員能夠承擔的抽象概念數量有其極限。顯然我們希望把 *OCP* 的應用限制在有可能發生的變更上。怎樣知道哪些變更有可能發生呢？我們做適當的研究調查、提出適切的問題、運用經驗與常識。最後就等變更發生！

在許多方面，*OCP* 是 OOD 的核心。遵循這個原則會帶來 OO 技術所宣稱的最大益處，亦即彈性、復用性和維護性。然而要符合此原則並非單靠使用 OOPL 就可達到，應用程式的每個部分都運用繁多的抽象化也不是個好主意。它極需仰賴開發人員只對「顯示出頻頻改變」特性之程式部份運用抽象化解法。抗拒「草率之抽象化」和抽象化本身一樣重要。

ref 《Agile Software Development, Principles, Patterns, and Practices》



LSP (Liskov Substitution Principle) 黎斯替派原則

Sub-types 必須可以替換它們的 base types

OCP 背後的主要機制是 abstraction 和 polymorphism。在靜態型別語言如 C++ 和 Java 中，支持上述兩者的關鍵機制之一就是 inheritance。Inheritance 的設計規則是什麼？最佳的 hierarchy 性質有哪些？又有哪些陷阱會導致我們產生不符合 *OCP* 的 hierarchy 呢？這些都是 *LSP* (1988) 所要解決的問題。

ref 《Agile Software Development, Principles, Patterns, and Practices》



LSP (Liskov Substitution Principle) 黎斯替化原則

如果對於每個 S type object s ，存在一個 B type object b ，使得在所有以 B 定義的 program P 中以 s 取代 b 時 P 的行爲維持不變，則 S 為 B 的 subtype。

Barbara Liskov first wrote this principle in 1988. She said, *What is wanted here is something like the following substitution property: If for each object o1 of type S there is an object o2 of type T such that for all programs P defined in terms of T, the behavior of P is unchanged when o1 is substituted for o2 then S is a subtype of T.*

假設 function f 帶有一個指向 base class B 的 pointer 或 reference 作為參數，同時假設有某個 B 's derived class D ，將 D 轉型成 B 後傳遞給 f 會導致 f 行爲不正常，那麼 D 就違反了 LSP。

也許 f 的作者會嘗試為 D 加入某種測試，使得當 D 傳遞給 f 時一旦通過測試 f 得以行爲正確。這種測試違反 OCP，因為現在 f 對於所有不同的 B derived 都不具有封閉性（也就是說 f function code 將因不同的 B derived 而修改），這樣的測試就是一種腐味（bad smell），是經驗不足或匆匆忙忙的開發人員違反 LSP 所製造出來的東西。

違反 LSP 常導致以嚴重違反 OCP 的方式使用 RTTI，亦即經常使用明顯的 if 述句或 if/else 述句鏈來決定 object type 以便選出與該 type 相稱的行爲。

ref 《Agile Software Development, Principles, Patterns, and Practices》

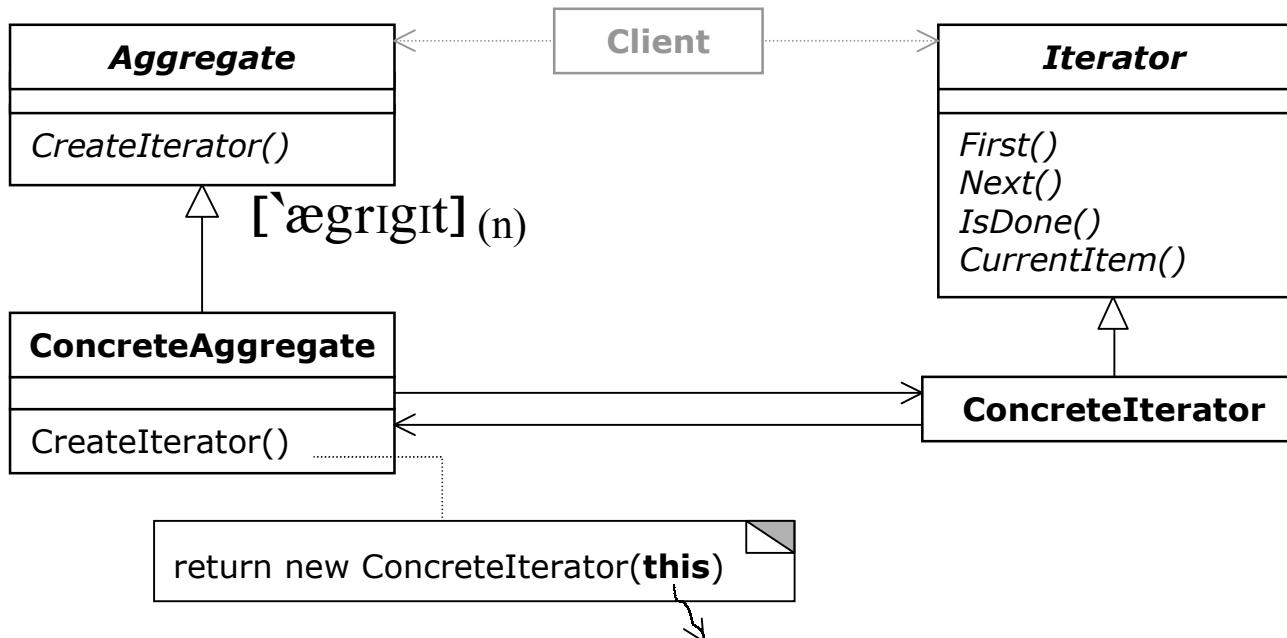


13. Iterator

[`ægrɪtə] (v)
[`ægrɪtə] (a)

Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

提供一種連續（相繼）巡訪「聚合物內各元素」的通用介面，並且不必曝露聚合物的底層表述（內部細節）。



Java Library 中由於 iterator classes 都被設計為 container class 的 inner class , 所以無需傳遞 'this'.

STL 中由於 iterator 直接指涉 container 內部結構所以也無需傳遞 'this'.



13. Iterator in Java

用法：

Generics Java 只允許在 collection 內放置 class object，不允許放置 primitive type data. 所以不得寫為：LinkedList<int> il=new...;

```
LinkedList<Integer> il = new LinkedList<Integer>();  
il.add(new Integer(0));  
il.add(new Integer(1));  
il.add(new Integer(5));  
il.add(new Integer(2));
```

```
Iterator ite = il.iterator();  
while(ite.hasNext()) {  
    System.out.println(ite.next());  
}
```

自從JDK5.0提供 box/unbox 之後，
可寫為：

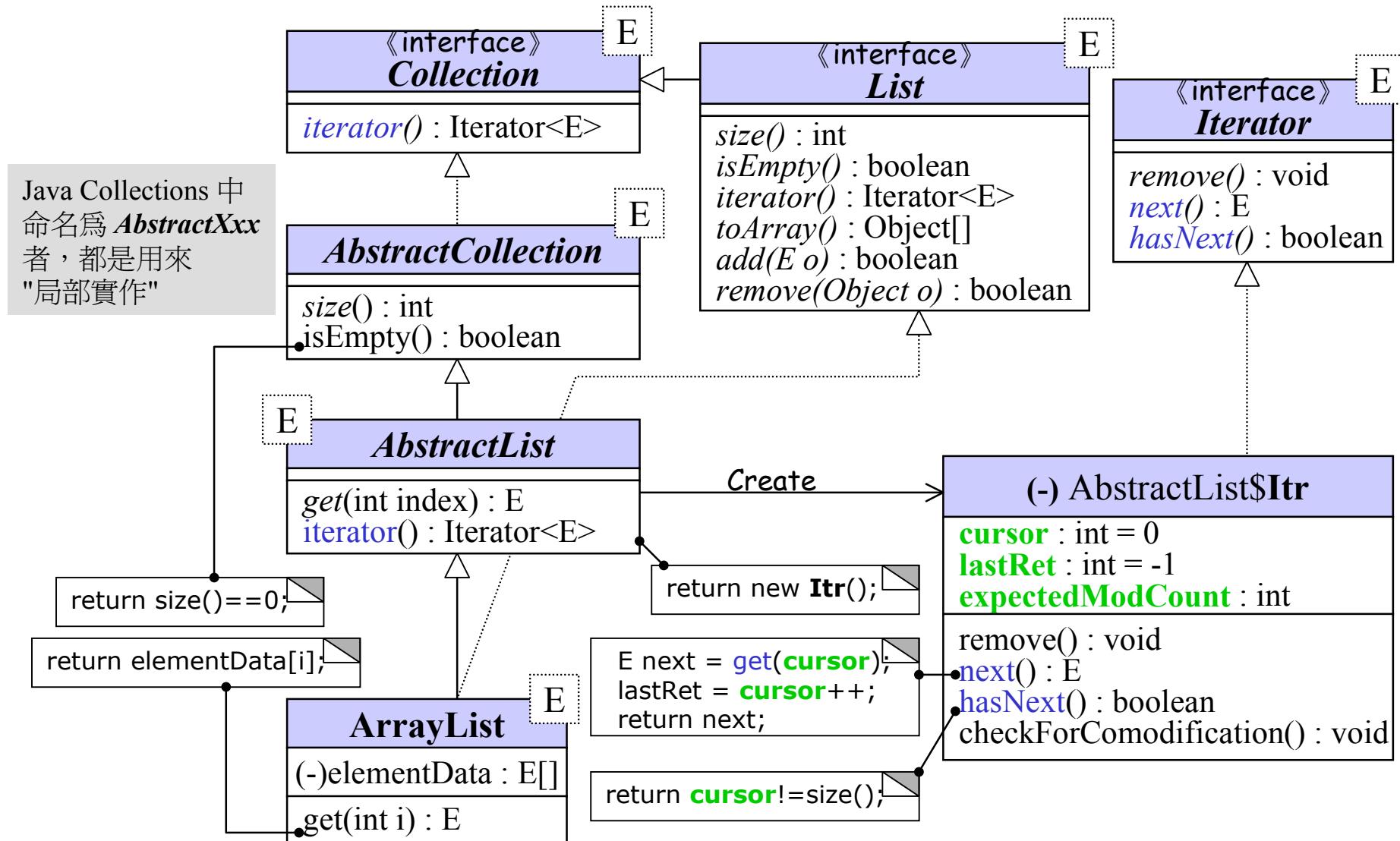
```
il.add(0);  
il.add(1);  
il.add(5);  
il.add(2);
```

每一種 Java collection 都具備 iterator()

每個 iterator 都具備這樣的 methods.



13. Iterator in Java Lib.





13. Iterator in Java

Java Library 運用 (1) **Iterator** interface 提供 next() 和 hasNext()，再運用 (2) **Collection** interface 提供 iterator()。然後令所有 collections 都實作 **Collection** 且所有 iterators 都實作 **Iterator**。

因此，舉例，**ArrayList** 就必須實作 **Collection**'s iterator()，在其中生成其內部定義的一個 private **Itr** class object；**Itr** 必須實作 **Iterator**'s next() 和 **Iterator**'s hasNext()。**ArrayList** 內部定義自己的 iterator 是必要作法，因為只有它自己才知道該如何走訪自己的元素。

Q:前面顯示，**AbstractList** 會生成 **AbstractList\$Iter**，後者以 cursor++ 的形式遍歷整個容器。這似乎意味容器本身必須是連續空間 -- 這對 **AbstractList** 的 derived classes 如 **ArrayList** 或 **Vector** 的確成立，但對另一個 derived class **LinkedList** 呢？我們發現，它改走了另一條路；其 base **AbstractSequentialList** 提供的 iterator() 呼叫了自己專用的 listIterator(i)，可令某個 **ListIterator**（嚴格說是 **LinkedList\$ListIter**）指向 #i 元素。



13. Iterator in C++ STL

類似 Java iterator（也就代表 **Iterator** pattern）的概念在 C++ STL 也是一樣的，只是 STL 並沒有將所有 iterator 具備的共通屬性（例如 5 associated types, op++, op*, op->）提升為一個 interface，也沒有將所有 containers 具備的共通屬性（例如 createIterator()）提升為一個 interface，而是不厭其煩地在每個 iterator classes 和 container classes 內定義。



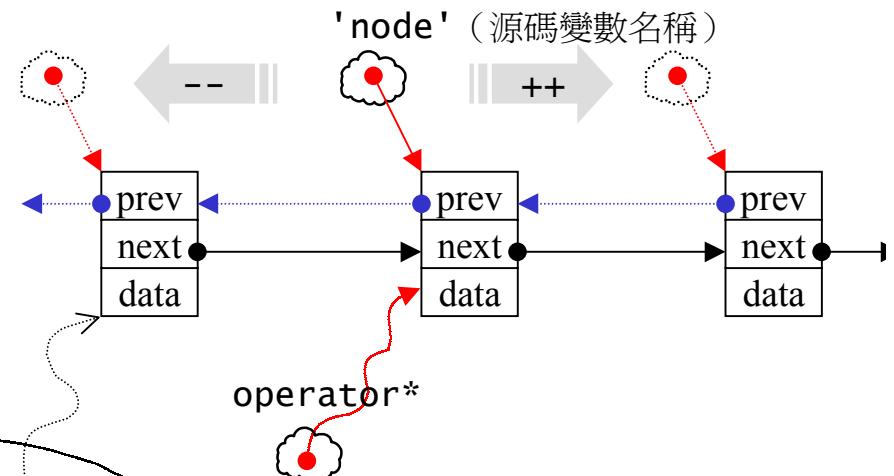
13. Iterator in C++ STL

```
template <class T>
struct __list_node {
    void* prev;
    void* next;
    T data;
};
```

```
template<class T, class Ref, class Ptr>
struct __list_iterator {
    typedef __list_iterator<T, T&, T*> iterator;
    typedef __list_node<T>* link_type;

    link_type node;

    bool operator==(const self& x) const;
    bool operator!=(const self& x) const;
    reference operator*() const;
    pointer operator->() const;
    self& operator++();
    self& operator+(int);
    self& operator--();
    self& operator-(int);
};
```



```
template <class T, class Alloc=alloc>
class list {
public:
    typedef __list_iterator<T,T&,T*> iterator;
};
```

Application:

```
int ia[5] = {0,1,2,3,4};
list<int> ilist(ia, ia+5);

list<int>::iterator ite=ilist.begin();
cout << *(++ite) << endl;
```



13. Iterator in MFC

MFC 徒有 iterator 的皮而無其骨。雖然從應用上看 MFC 的各個 collections 也提供了（幾乎）一致的介面（GetHeadPosition(), GetNext(), Remove()...），但其實是讓每一個collection 都實作出那些函式，並沒有 abstraction 在其中。

```
class CPtrList : public CObject
{
    POSITION GetHeadPosition() const { return (POSITION) m_pNodeHead; }
;
    void* GetNext(POSITION& rPosition) const; // return *Position++
    void RemoveAt(POSITION position)
};

class CMapWordToPtr
    : public CObject
{
    // iterating all (key, value) pairs
    POSITION GetStartPosition() const;
    void GetNextAssoc(POSITION& rNextPosition, WORD& rKey, void*& rValue) const
    BOOL RemoveKey(WORD key);
;
};
```

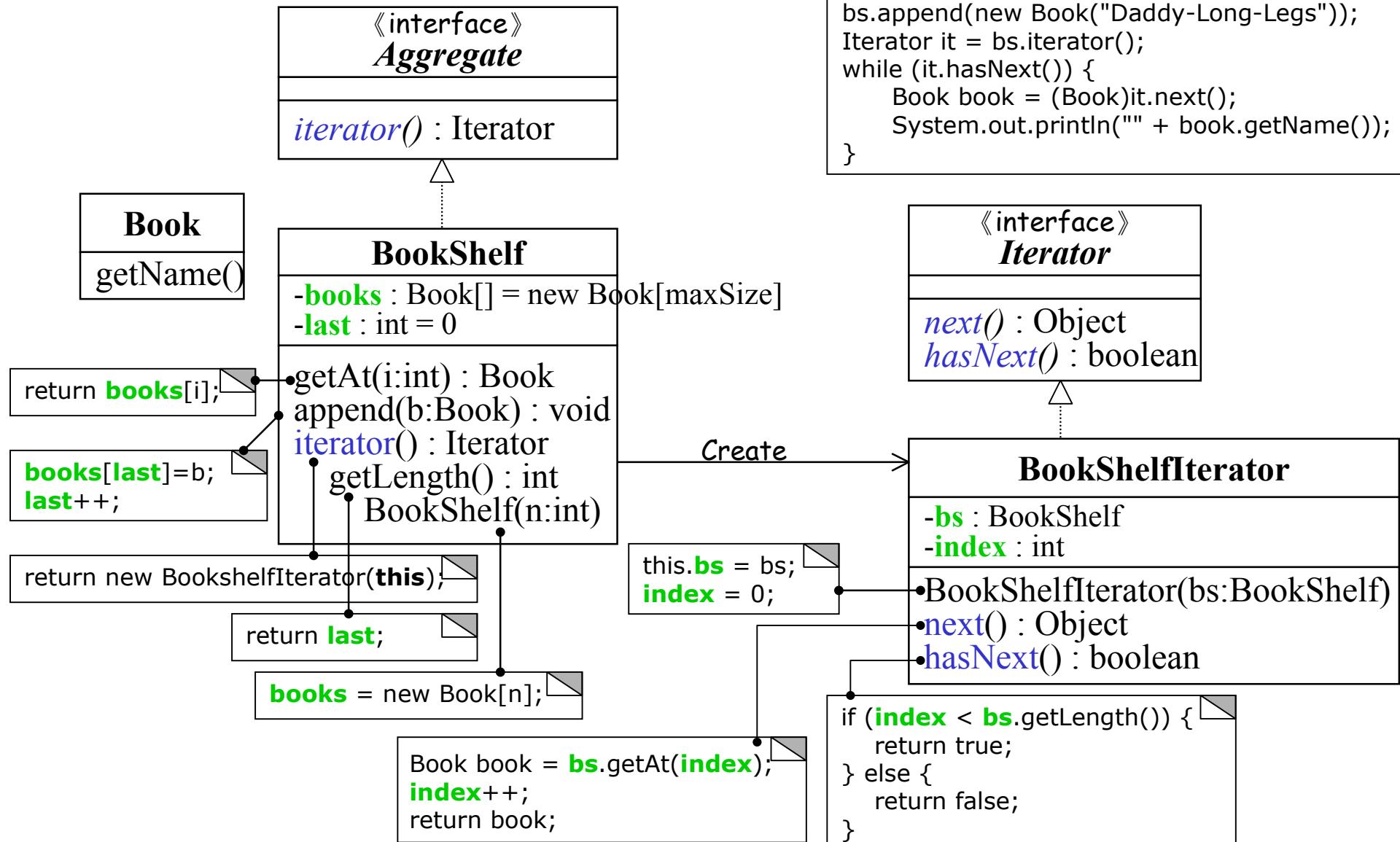
```
struct __POSITION { };
typedef __POSITION* POSITION;
```

```
POSITION pos = pDoc->myList.GetHeadPosition();
while (pos != NULL) {
    CShape* pS =
        (CShape*)pDoc->myList.GetNext(pos);
    pS->display();
}
```

Application



13. 習題

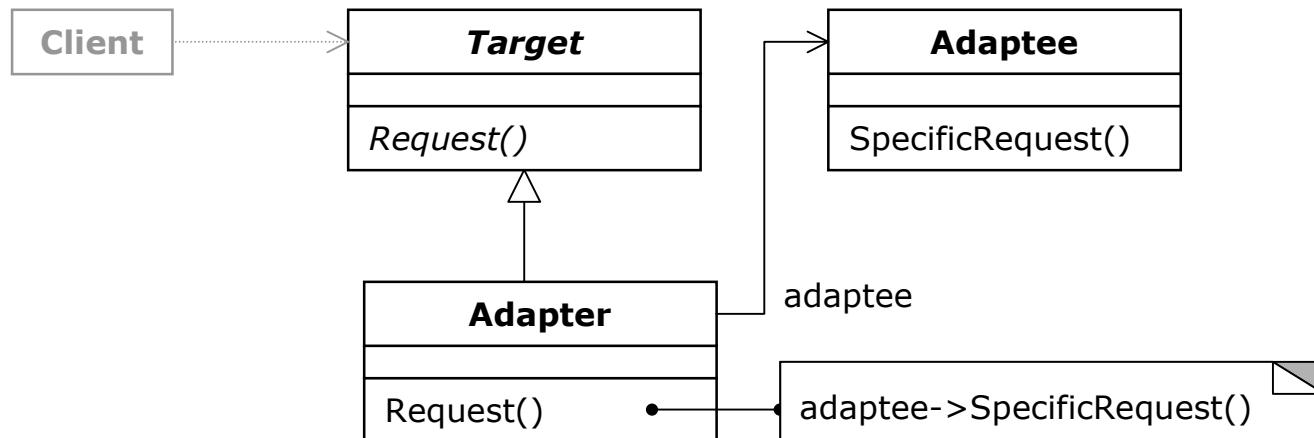




2. Adapter in GOF

Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

轉換 class 的介面使其為 client 所期望。Adapter 使得原本因「介面不相容」而無法合作的 classes 變得可以合作。



Adapter 是為了復用那些「已經做過許多測試、臭蟲不多」的 classes。也可用於軟件版本更迭之維護。**Adapter** 有兩種：

- **Class Adapter**（運用 inheritance 實現出來）
- **Object Adapter**（運用 delegation 實現出來）



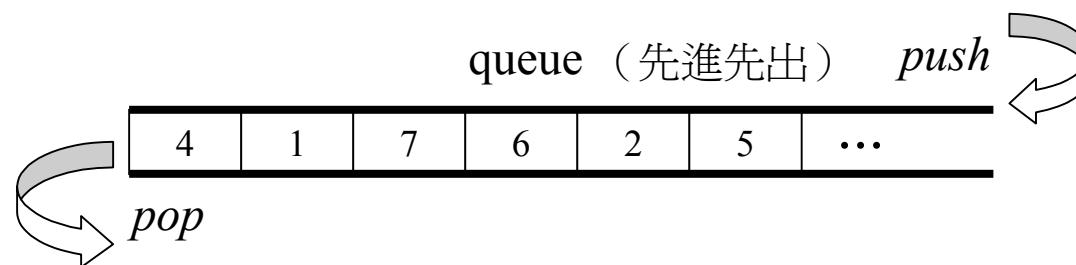
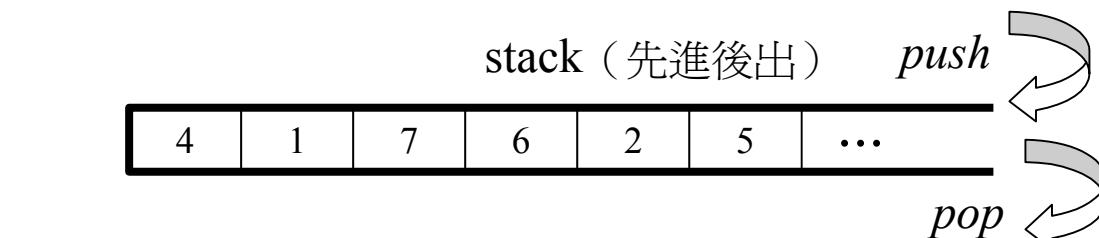
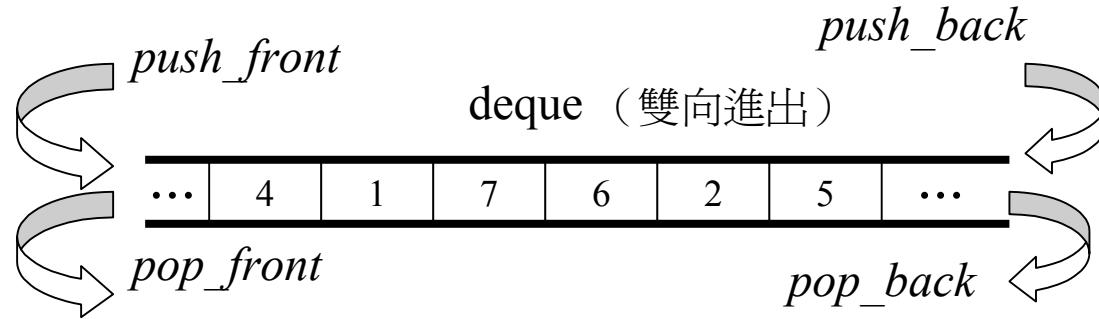
2. Adapters in STL

STL 提供了三種 Adapters :

- Container Adapters : stack, queue.
- Function Adapters : negater, binder, composer
- Iterator Adapters : io, reverser, inserter.



2. Adapter in STL : deque, queue, stack





2. Adapter in STL. Container Adapter : queue

這是屬於 Object Adapter

也可以是 list

```
template <class T, class Sequence = deque<T> >
class queue {
    ...
protected:
    Sequence c; // 底層容器
public:
    // 以下完全利用 Sequence c 的操作函式完成
    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }
    reference front() { return c.front(); }
    reference back() { return c.back(); }
    // deque 是兩端可進出，queue 是末端進前端出（先進先出）
    void push(const value_type& x) { c.push_back(x); }
    void pop() { c.pop_front(); }
};
```

composition



2. Adapter in STL. Container Adapter : stack

這是屬於 **Object Adapter**

也可以是 list

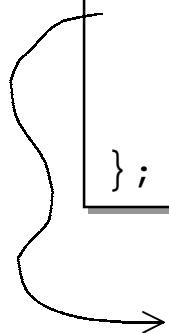
```
template <class T, class Sequence = deque<T> >
class stack {
...
protected:
    Sequence c;           // 底層容器 composition
public:
    // 以下完全利用 Sequence c 的操作
    bool empty() const { return c.empty(); }
    size_type size() const { return c.size(); }
    reference top() { return c.back(); }
    const_reference top() const { return c.back(); }
    // deque 是兩端可進出，stack 是末端進末端出（先進後出）
    void push(const value_type& x) { c.push_back(x); }
    void pop() { c.pop_back(); }
};
```



2. Adapter in STL. Functor adaptable

```
// STL規定，每一個 Adaptable Unary Function 都應該繼承此類別
template <class Arg, class Result>
struct unary_function {
    typedef Arg argument_type;
    typedef Result result_type;
};

// STL規定，每一個 Adaptable Binary Function 都應該繼承此類別
template <class Arg1, class Arg2, class Result>
struct binary_function {
    typedef Arg1 first_argument_type;
    typedef Arg2 second_argument_type;
    typedef Result result_type;
};
```



定義一個模子，提供三種 types，準備應付三個標準詢問。
任何打算回答那三個標準詢問的 classes 都可以在繼承
`binary_function` 後獲得回答能力。



2. Adapter in STL. adaptable functor

```
// 以下爲算術類 (Arithmetic) 仿函式
template <class T>
struct plus : public binary_function<T, T, T> {
    T operator()(const T& x, const T& y) const { return x + y; }
};

template <class T>
struct minus : public binary_function<T, T, T> {
    T operator()(const T& x, const T& y) const { return x - y; }
};

// 以下爲相對關係類 (Relational) 仿函式
template <class T>
struct less : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const { return x < y; }
};

// 以下爲邏輯運算類 (Logical) 仿函式
template <class T>
struct logical_and : public binary_function<T, T, bool> {
    bool operator()(const T& x, const T& y) const { return x && y; }
};
```

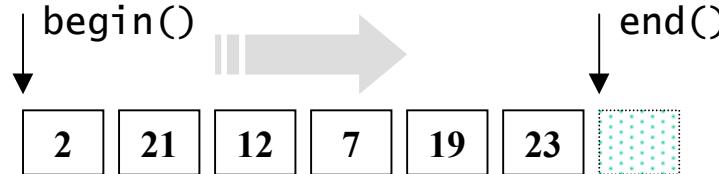
爲什麼要以 functor 取代 function？因爲 functor 做爲一個 object 可有無限想像空間，例如它可以攜帶 data，可提供如上/前頁的 typedef，形成 adaptable。

2. Adapter in STL. bind2nd() & binder2nd

```
// 以下配接器用來將某個 Adaptable Binary function 轉換為 Unary Function
template <class Operation>
class binder2nd
    : public unary_function<typename Operation::first_argument_type,
                           typename Operation::result_type> {
protected:
    Operation op;           // 內部成員
    typename Operation::second_argument_type value; // 內部成員
public:
    // constructor
    binder2nd(const Operation& x,
              const typename Operation::second_argument_type& y)
        : op(x), value(y) {} // 將運算式和第二引數記錄於內部成員
    typename Operation::result_type
    operator()(const typename Operation::first_argument_type& x) const {
        return op(x, value); // 實際呼叫運算式，並將 value 繫結為第二引數
    }
};

// 輔助函式，讓我們得以方便獲得 binder2nd<Op>；編譯器會自動幫我們推導 Op 的 type
template <class Operation, class T>
inline binder2nd<Operation> bind2nd(const Operation& op, const T& x) {
    typedef typename Operation::second_argument_type arg2_type;
    return binder2nd<Operation>(op, arg2_type(x)); // 一個 temp. object
} // 以上注意，先把x轉型為op的第二引數型別 (2nd argument type)
```

2. Adapter in STL. count_if() & bind2nd()



bind2nd(...) 會產生一個
binder2nd<Operation>(op,n); 物件。
此將傳給 count_if() 成為其 pred 引數。

```
count_if(iv.begin(), iv.end(), bind2nd(less<int>(), 12), i);
```

```
template <class InputIterator, class Predicate, class Size>
void count_if(InputIterator first, InputIterator last,
              Predicate pred, Size& n) {
    for ( ; first != last; ++first) // 整個走一遍
        if (pred(*first)) // 如果元素帶入pred的運算結果為 true
            ++n; // 計數器累加1
}
```

舊版count_if()

Q: 怎麼知道要用bind2nd()
而非bind1st()呢？

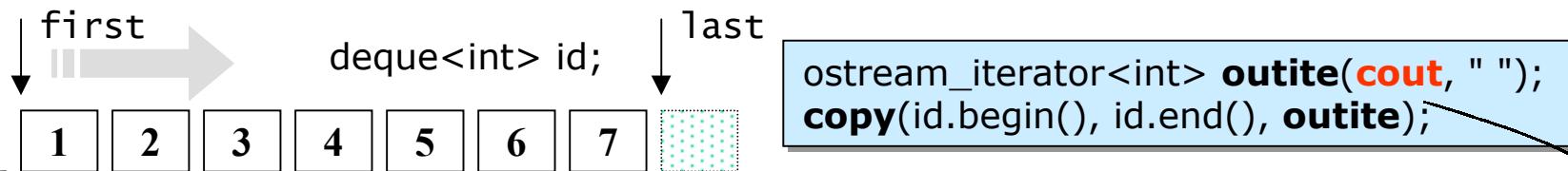
A: 看手冊或看源碼
本例既然傳入之元素將被執行 < 12 之比較，可見元素被做為 operator< 的左運算元，亦即第一參數，故應繫結第二參數，故使用 bind2nd().

```
template <class Operation>
class binder2nd : public unary_function<...> {
protected:
    Operation op; // 內部成員
    typename Operation::second_argument_type value;
public:
    binder2nd(const Operation& x,
              const typename Operation::second_argument_type& y)
        : op(x), value(y) {}
    typename Operation::result_type
    operator()(const typename Operation::first_argument_type& x) const {
        return op(x, value); // 將 value 繫結 (binding) 為第二引數
    }
};
```

我們當然就可以為所欲為了。
控制權轉到我們手上。

less<int>()和12將被
記錄在op和value中

2. Adapter in STL. ostream_iterator



設計adapter時
必須先觀察
adaptee的設計。
本例必須先知道
copy()中對
result做了什
麼動作，再去
設計 adapter

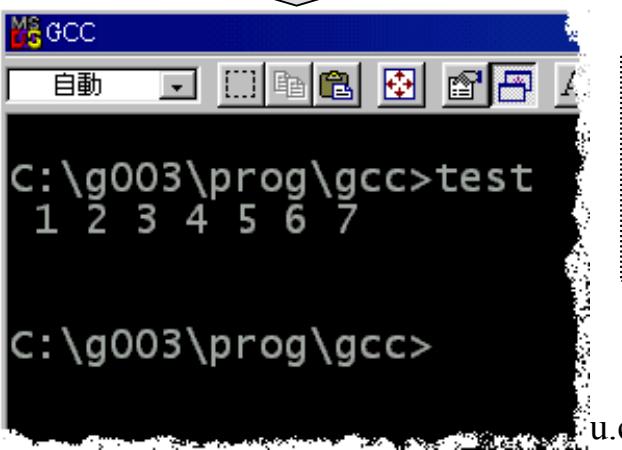
```
template <class RandomAccessIterator, class OutputIterator, class Distance>
inline OutputIterator
__copy_d(RandomAccessIterator first, RandomAccessIterator last,
         OutputIterator result, Distance*)
{
```

```
    for (Distance n = last - first; n > 0; --n, ++result, ++first)
        *result = *first;
    return result;
}
```

```
template <class T>
class ostream_iterator {
protected:
    ostream* stream;
public:
```

ctor將它塞入（儲存）

摘錄



```
// 對迭代器做賦值 (assign) 動作，就代表要輸出一筆資料
ostream_iterator<T>& operator=(const T& value) {
```

```
*stream << value; // 關鍵：輸出數值
return *this;
}
```

```
// 注意以下運算子對上方 __copy_d() 函式內的動作的影響
ostream_iterator<T>& operator*() { return *this; }
ostream_iterator<T>& operator++() { return *this; }
```

}; 由於 copy()對result做了op++, op*, op= 動作，所以這裡必須設計它們



2. Listener-Adapter in Java Lib.

Java Lib. 提供所謂的 listener。每一個 Swing component 都有兩個函式：
`addXXXLister()` 和 `removeXXXListener()`，其中 `XXX` 代表 event 種類。這樣一來 App. 就可以藉由植入 listener 而監聽（監看）component 發生什麼事。每個 listener 都是一個 object，其 class（通常寫為 UI business logic class 的 inner class，不但邏輯編組合理，而且可直接取用 base class fields）必須實現特定之 interface。下面是監視 component 一例（取材自 TIJ2, p.725）：

```
JTextField name = new JTextField(25);
...
class NameL implements ActionListener { //監聽器
    public void actionPerformed(ActionEvent e) { ... }
}
...
name.addActionListener(new NameL()); //產生
```

本來應該 implements WindowListener，改為 extends WindowAdapter

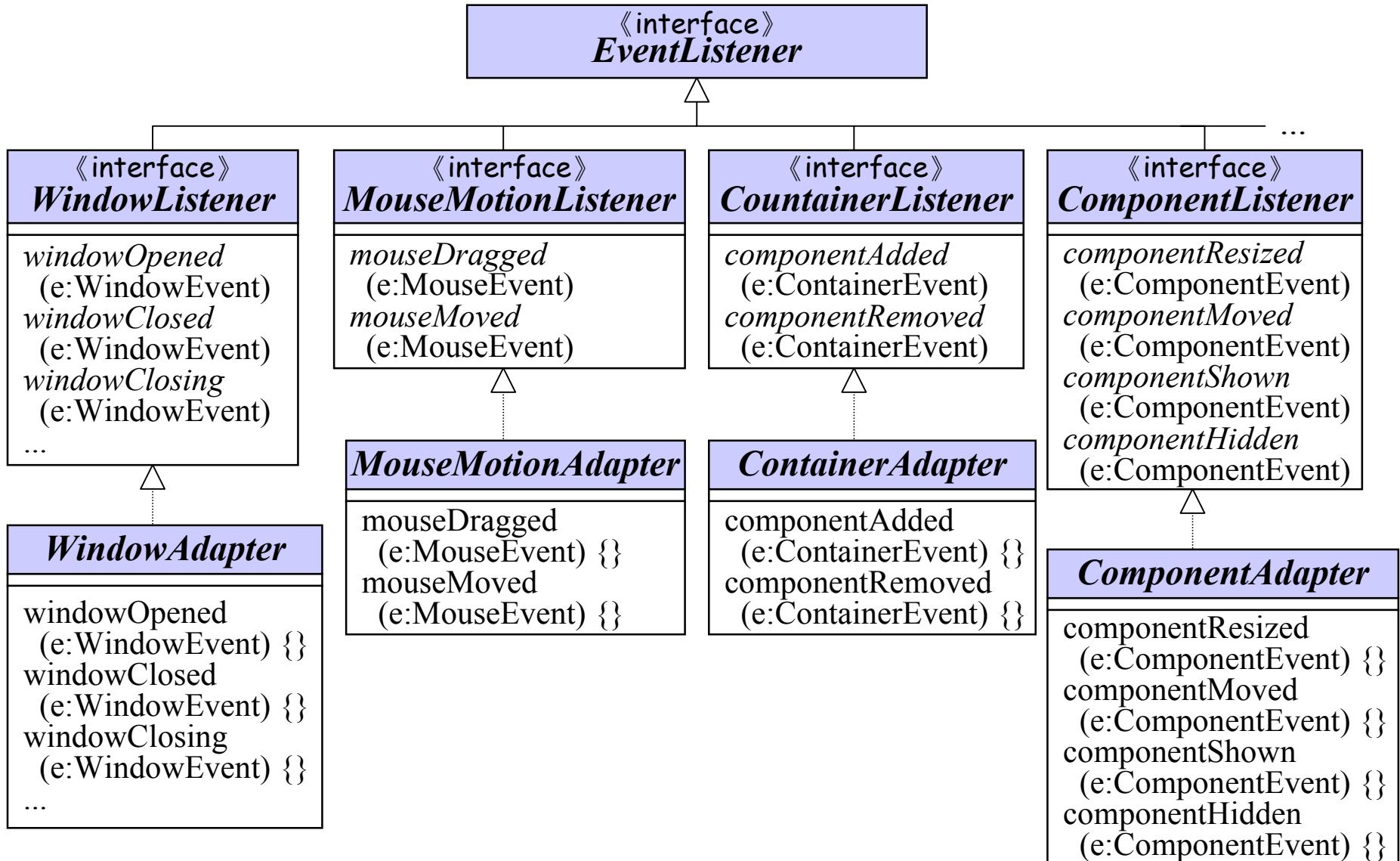
注意：萬一寫成 WindowClosing() 就麻煩了！編譯器不報錯。

```
class MyWindowListener
    extends WindowAdapter { //listener adapter
    public void windowClosing(WindowEvent e) { ... }
}
```

listener interface（例如 `WindowListener`）若有多個 methods，listener class 必須全部實作出來。這有時候形成困擾。因此某些 listener interface 搭配了所謂的 adapter，其中為它所搭配的 listener interface 的每一個 methods 實現出空函式。**Java Adapter classes 的中心思想是要簡化 listener class 的撰寫工程。**



2. Listener-Adapter in Java Lib.



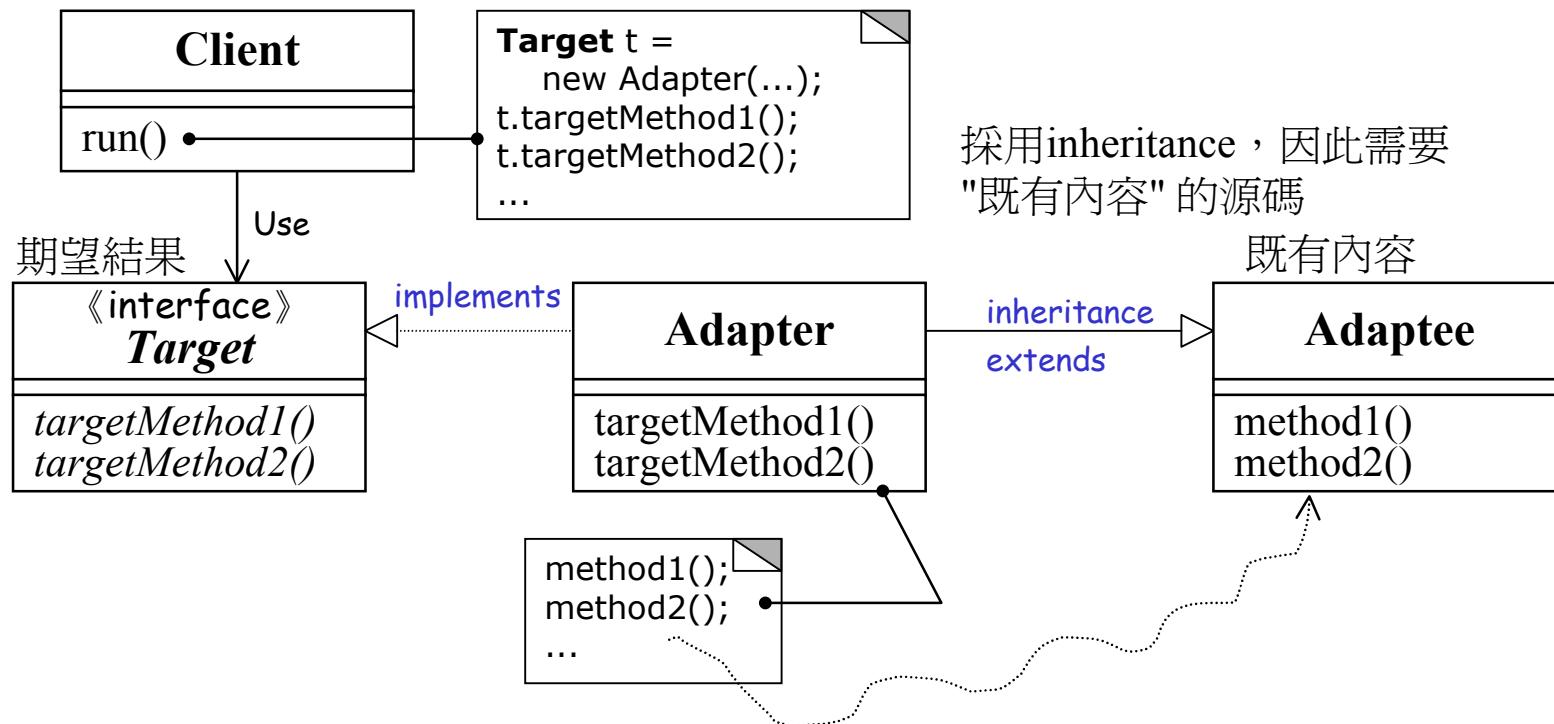


2. Class-Adapter in DP-in-Java

Adapter 有兩種：

1. **Class Adapter** (運用 inheritance)
2. **Object Adapter** (運用 delegation)

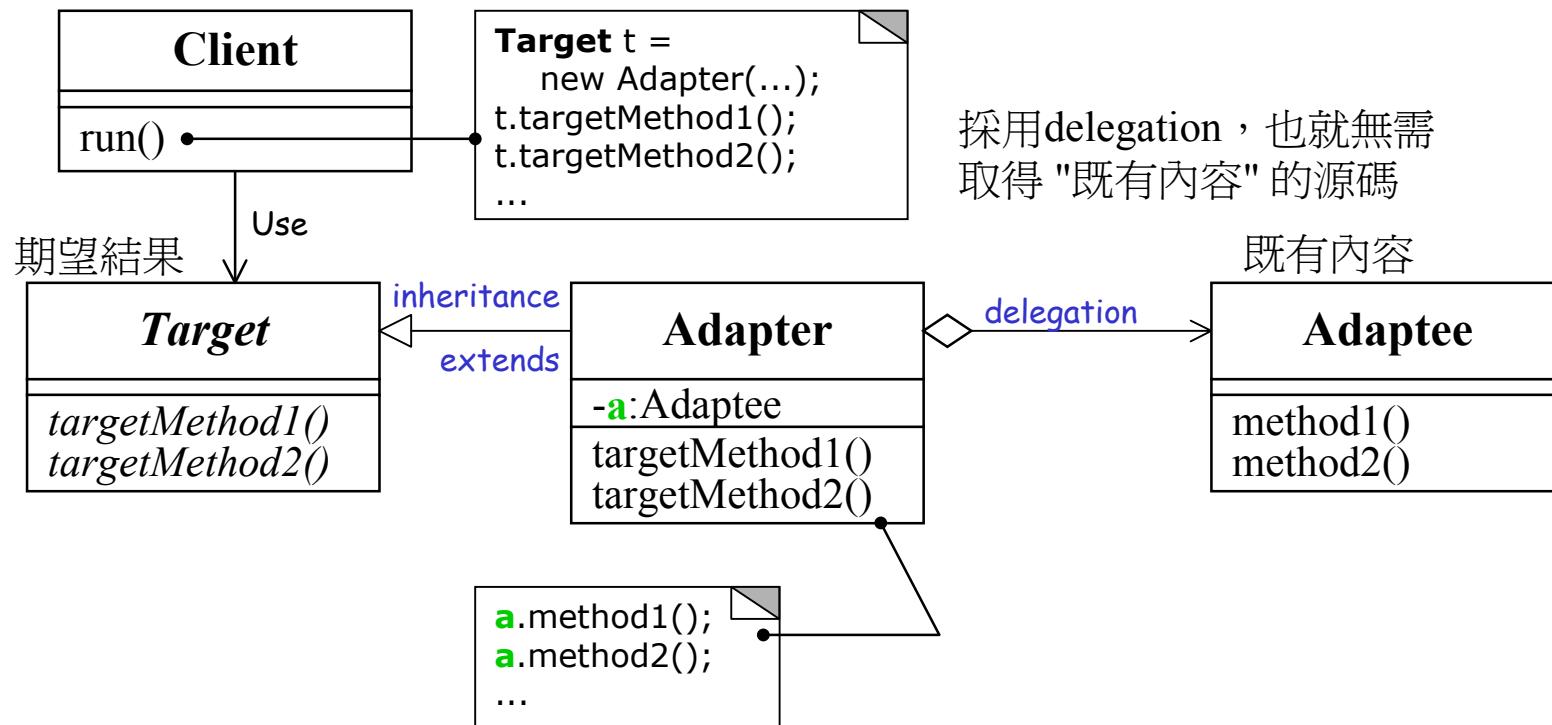
Class Adapter





2. Object-Adapter in DP-in-Java

Object Adapter





2. 習題

有一個 Java class 名為 `java.util.Properties`，用來管理像這樣的 key/value pairs. :

`year = 2006`

`month = 3`

`day = 16`

它有兩個 methods 如下：

`void load(InputStream in) throw IOException`

`void store(OutputStream out, String header)`

`throw IOException`

寫一個 Object Adapter 和一個 Class Adapter，使應用程式能夠運用以下的 `FileIO` interface
(和上述既有的 `Properties` class)，對檔案進行 key/value 的讀寫。

```
import java.io.*;
public class Main {
    public static void main(String[] args) {
        FileIO f = new FileProperties();
        try {
            f.readFromFile("file.txt");
            f.setValue("year", "2000");
            f.setValue("month", "11");
            f.setValue("day", "20");
            f.writeToFile("newfile.txt");
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

```
import java.io.*;
public interface FileIO {
    public void readFromFile(String filename) throws IOException;
    public void writeToFile(String filename) throws IOException;
    public void setValue(String key, String value);
    public String getValue(String key);
}
```



11. Flyweight

11. Flyweight (195)

Use sharing to support large numbers of **fine-grained** objects efficiently.

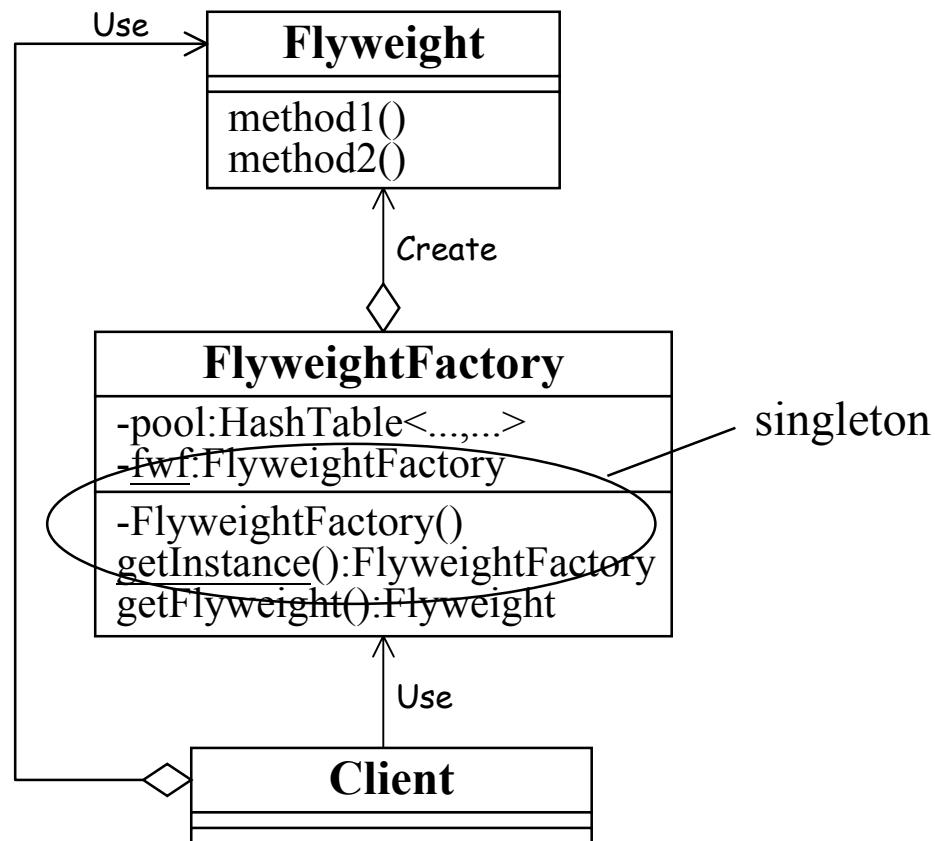
運用「共享技術」有效支援大量 fine-grained objects（細小的物件）。

- 可聯想到 Reference Counting
- 共用的部分一旦被修改，就會影響有關聯的全體（好或不好視情況而定）
- 共用的資料稱為 **intrinsic data**（固有的、內在的資料）
- 不共用的資料稱為 **extrinsic data**（非固有的、外在的資料）
- 被管理的 objects 不會被自動垃圾回收。這意思是即使程式有一長段時間都沒用到它們，或永遠再不會用到它們，它們還是佔用 memory，不會被回收。比起一般 Java 程式會自動做垃圾回收，這一點算是副作用。
- Flyweight 對於空間效率和時間效率都有幫助。



11. Flyweight

Flyweight 是「蠅量級」的意思，是拳擊賽中體重最輕的一級。Flyweight pattern 就是用來減輕 object 的「重量」，而通常「重量」是指「memory 使用量」。這個 pattern 的宗旨就是：儘量共用 object instances，不做無謂的 'new'。



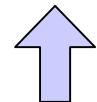


11. Flyweight

在獲得 bc 至 pool.put() 之間不能被其他 thread 插隊，所以 getBigChar() 必須宣告為 synchronized.

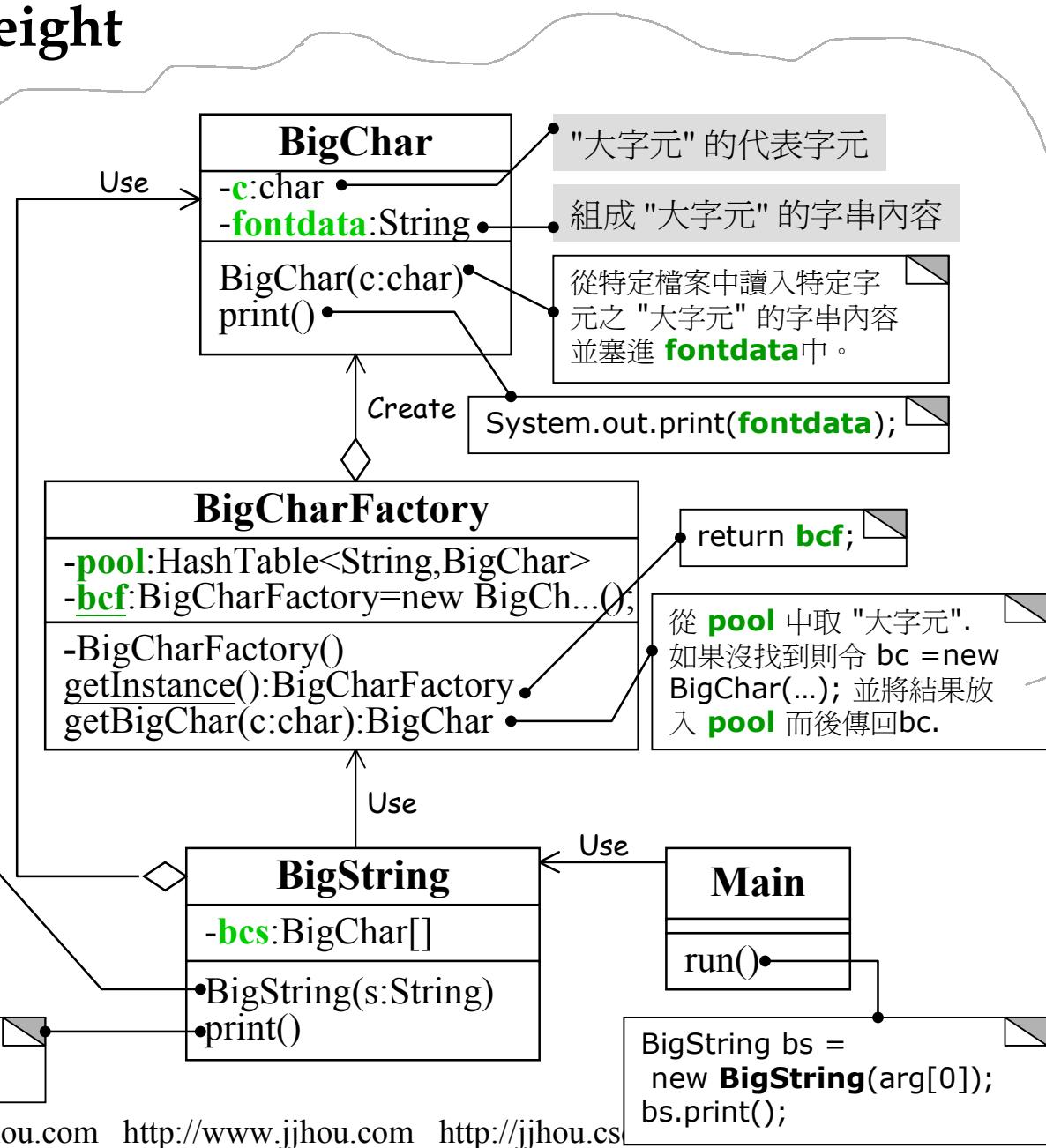
若不使用 Flyweight 則應該是：

```
bcs = new BigChar[...];
for(int i=0;...)
    bcs[i]=new BigChar(s.charAt(i));
```



```
bcs = new BigChar[...];
BigCharFactory f =
    BigCharFactory.getInstance();
for(int i=0;...)
    bcs[i]=f.getBigChar(s.charAt(i));
```

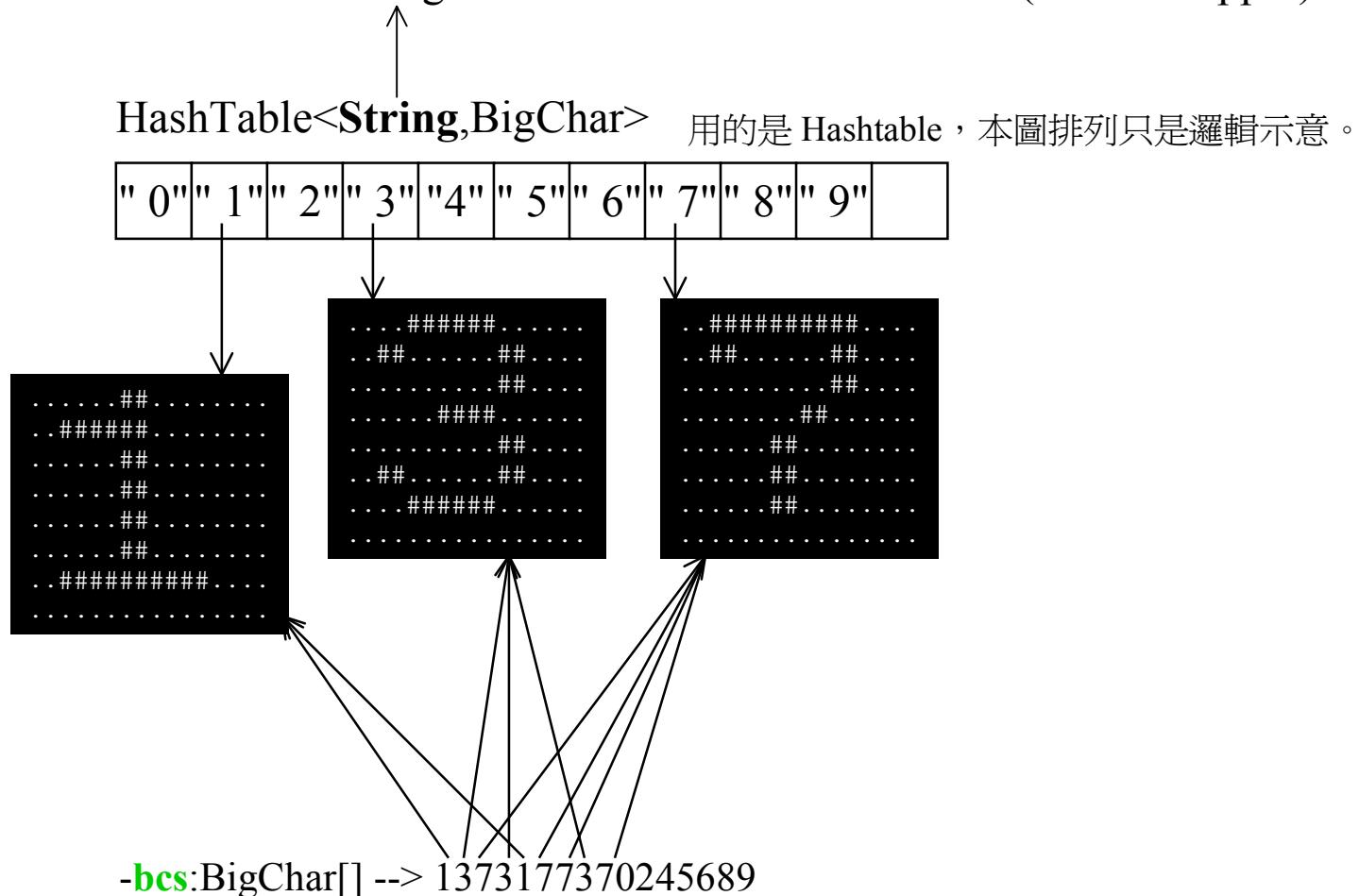
```
for(int i=0;...)
    bcs[i].print();
```





11. Flyweight

按道理應該是 char 就好，不過 Java Collection 不接受 primitive type。
這大概是作者使用 String 的原因。應該也可使用 Char (char's wrapper)





Reference Counting in MEC

ref MEC p183

Reference counting 這項技術，允許多個等值objects共享同一實值。此技術之發展有兩個動機，第一是為了簡化 heap objects 周邊的簿記工作。一旦某個object以 new 配置出來，記錄「object擁有者」是件重要的事，因為它（也只有它）有責任刪除該 object。但是程式執行過程中，object的擁有權可能移轉（例如當以pointer做為 function argument），所以記錄object擁有權並非是件輕鬆工作。**Reference counting** 可以消除「記錄object擁有權」的負荷，因為當object運用了 **Reference counting** 技術，它便擁有它自己。一旦不再有任何人使用它，它便自動摧毀自己。也因此，**Reference counting** 架構出垃圾收集機制（garbage collection）的一個簡單型式。

Reference counting 的第二個發展動機則只是為了實現一種常識。如果許多objects有相同的值，將那個值儲存多次是件愚蠢的事。最好是讓所有等值objects共享一份實值就好。這麼做不只節省memory，也使程式速度加快，因為不再需要建構和解構同值 objects的多餘副本。



Reference Counting in MEC. 累贅與浪費

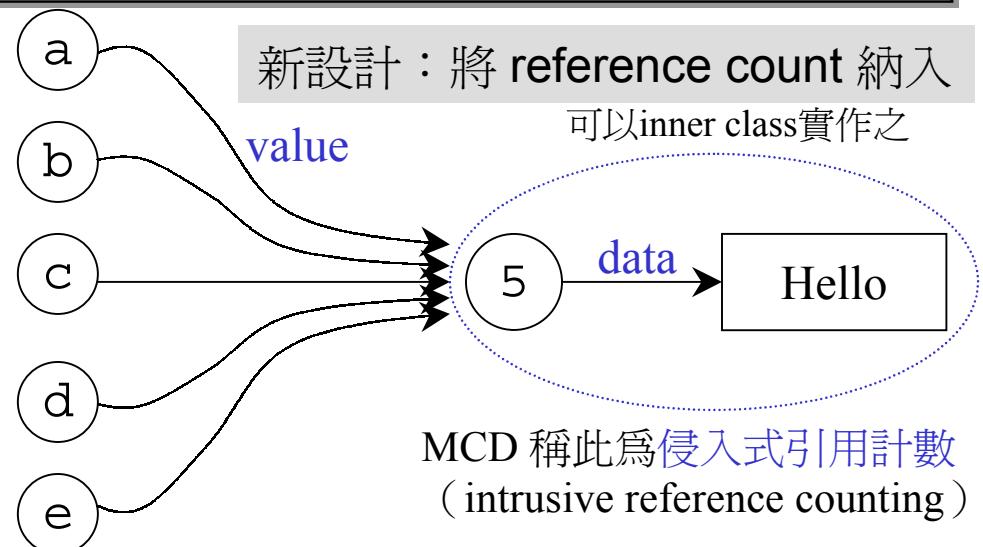
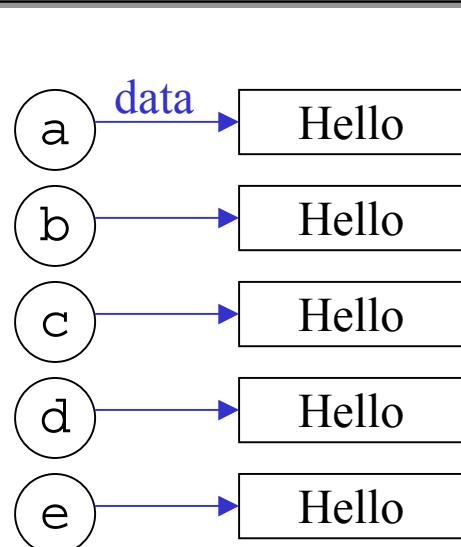
String 的一般實作：

```
String& String::operator=(const String& rhs)
{
    if (this == &rhs) return *this; // 避免自我賦值(self assignment)

    delete [] data; // 刪除舊資料
    data = new char[strlen(rhs.data) + 1]; // 配置新空間
    strcpy(data, rhs.data); // 設定新值

    return *this; // 傳回自己以利 "鏈式賦值"
}
```

```
String a, b, c, d, e;
a=b=c=d=e= "Hello";
```





Reference Counting in MEC

```
class String {
public:
    ... // String members
private:
    struct StringValue {
        int refCount;
        char *data;
    };

    StringValue(const char *initValue);
    ~StringValue();
};

StringValue *value;
```

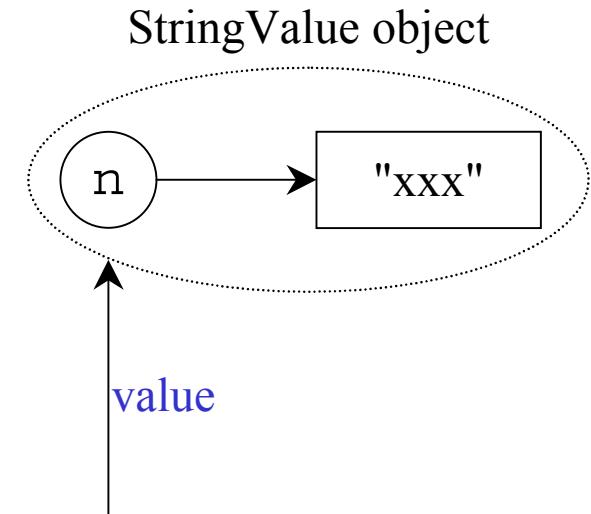
ctor → String::StringValue::StringValue(const char *initValue)
: refCount(1) {
 data = new char[strlen(initValue)+1];
 strcpy(data, initValue);
}
dtor → String::StringValue::~StringValue() {
 delete[] data;
}

The diagram shows a variable `n` pointing to a `StringValue` object. The `StringValue` object contains a `refCount` field and a `data` field. The `data` field points to a box containing the string `"xxx"`. A dotted oval labeled `value` encloses the `StringValue` object. Arrows indicate the relationships between `n`, `refCount`, `data`, and `value`.

Reference Counting in MEC

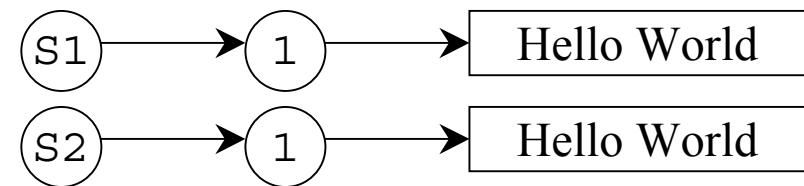
繼續上頁的 class String :

```
class String {  
public:  
    String(const char *initValue = "");  
    ... //copy ctor, copy assignment operator, dtor  
private:  
    ...  
    StringValue* value;  
};  
  
String::String(const char *initValue)  
    : value(new StringValue(initValue))  
{}
```



很好。惟這樣的實作（加上稍後的設計）無法避免以下情況：

```
String s1("Hello World");  
String s2("Hello World");
```



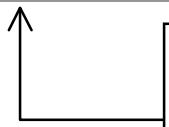
想法：在 `ctor` 中維護一個 `lookup table`，用來搜尋指定字串是否出現過。這或許可以解決上述問題。
但請考慮開發成本。



Reference Counting in MEC

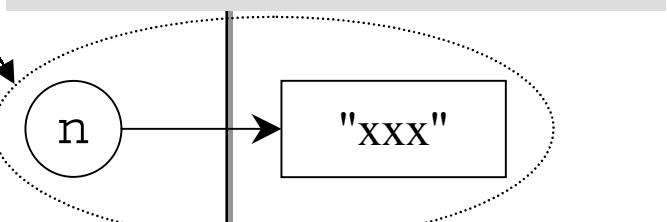
```
// 用途  
String a("hello");  
String b = a;  
// or String b(a);
```

爲達成目標，我們需要考量（介入）三種操作
(1) copy constructor
(2) destructor
(3) copy assignment operator



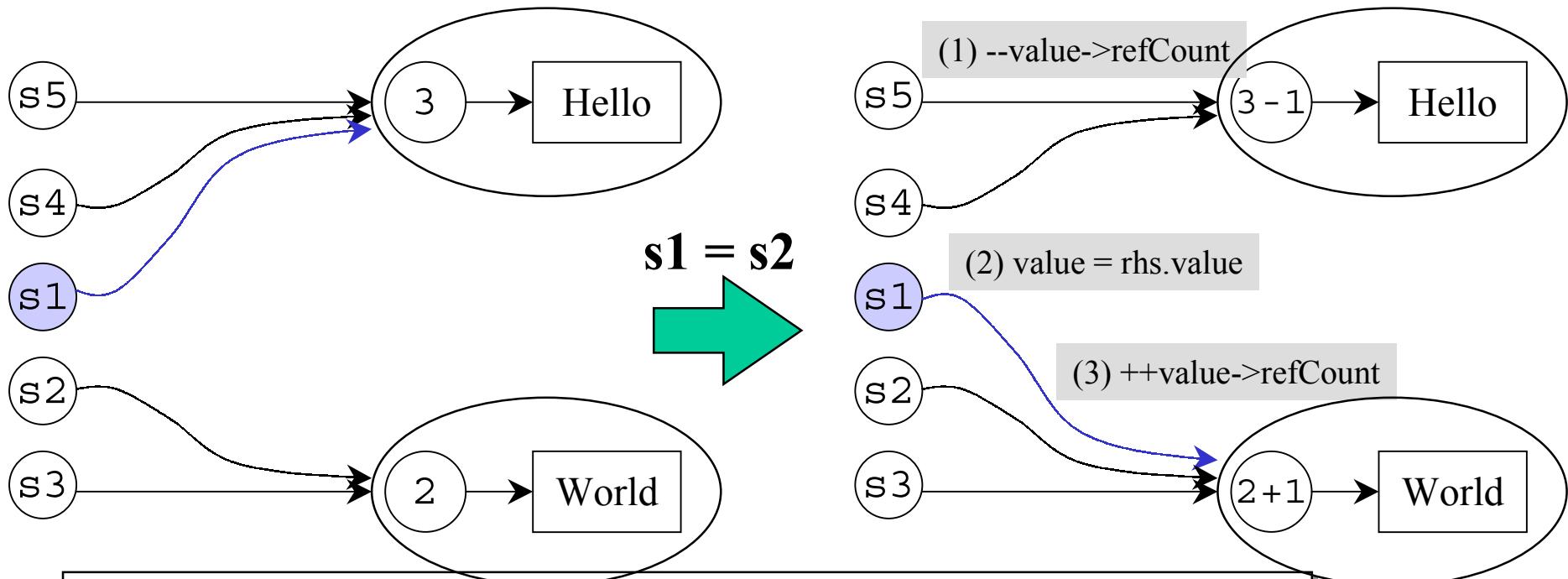
```
// (1) copy ctor  
String::String(const String& rhs)  
: value(rhs.value)  
{  
    ++value->refCount;  
}  
  
// (2) dtor  
String::~String()  
{  
    if (--value->refCount == 0) delete value;  
}
```

兩指標指向同一物件，形成alias。
但沒有關係，因爲已在控制之中。





Reference Counting in MEC



```
// (3) copy assignment operator
String& String::operator=(const String& rhs)
{
    if (value == rhs.value)           // 如果數值相同，什麼也不做。
        return *this;                // 這很類似對 &rhs 的慣常測試
    if (--value->refCount == 0)       // (1) 如果再沒其他人使用 lhs ,
        delete value;               // 就摧毀 *this 的數值。
    value = rhs.value;              // (2) 令 *this 共享 rhs 的數值。
    ++value->refCount;             // (3) 令 rhs 計數器加 1。
    return *this;
}
```



Reference Counting in MEC. copy-on-write

以上設計已能應付 copy 和 copy assignment。

當我們改動某個 String 值時，必須小心避免更動「共享同一 StringValue」的其他 String 物件。不幸，C++ 編譯器無法告訴我們 operator [] 被用於讀取或塗寫。所以我們必須悲觀地假設 non-const operator [] 的所有呼叫都用於塗寫，那就必須啓動 copy-on-write。

為安全實作出 non-const operator []，我們必須確保沒有其他任何「共享同一個 StringValue」的 String 物件因塗寫動作而改變。簡單地說，任何時候當我們傳回一個 reference，指向String 的 StringValue 物件內的一個字元時，我們必須確保該 StringValue 物件的參用次數確實實為 1

傳回 char 也可以。但會因此多做一個 copy 動作，而且無法形成 good error!

```
class String {  
public:  
    1 const char& operator[](int index) const; // 針對 const Strings  
    2     char& operator[](int index); // 下一頁，針對 non-const Strings  
    ...  
};
```

```
String s; // non-const  
... // 以下呼叫 non-const []  
cout << s[3]; // 這是一個讀取動作  
s[5] = 'x'; // 這是一個塗寫動作
```

Q: 有沒有可能 non-const String 嘴起 const op[]?
A: 當只存在 const op[] 時，non-const String 的確可嘴起它。但當 non-const op[] 並存時，non-const String 一定嘴起 non-const op[]

```
1 const char& String::operator[](int index) const  
{  
    return value->data[index];  
}
```

const object 只可能嘴起 const member function。
由於 const object 不可能被改動，所以此處
const member function 的動作是合宜而安全的。

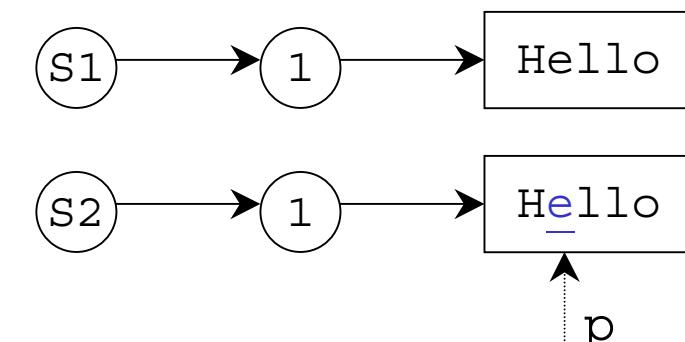
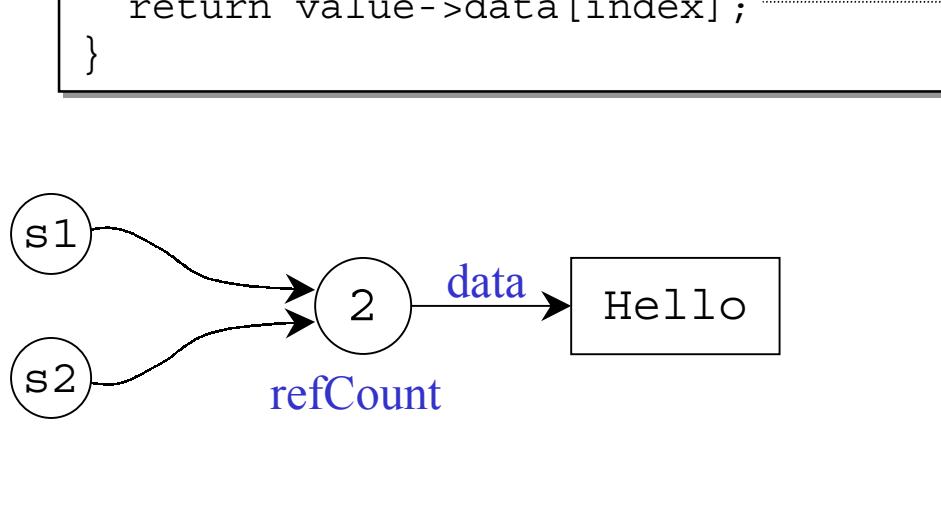
```
const String s("jjhou");  
cout << s[3]; // OK.  
s[5] = 'x'; // ERROR. 很好！  
// ERROR乃因 [] 傳回的是 const char&
```



Reference Counting in MEC. copy-on-write

2 `char& String::operator[](int index) // non-const 版，悲觀地假設將被用於塗寫`

```
{  
    // 如果本物件和其他 String 物件共享同一實值，  
    // 就分割（複製）出另一個副本供本物件自己使用；copy-on-write  
    if (value->refCount > 1) {  
        --value->refCount; // 將目前實值的參用次數減 1，因我們不再使用該值。  
        value = new StringValue(value->data); // 為自己做一份新副本，copy-on-write  
    }  
  
    // 傳回一個 reference，代表我們這個「絕對未被共享」的  
    // StringValue 物件內的一個字元。可被當做 L-value。  
    return value->data[index];  
}
```

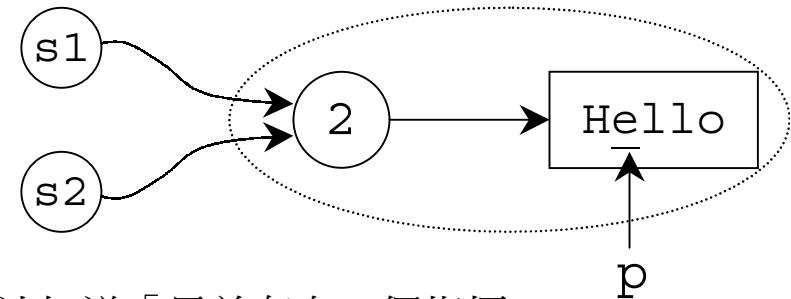




Reference Counting in MEC. shareable

問題的發生 (copy-on-write 之後 "copy" 被共享，而後這個被共享的 "copy" 才被 write) :

```
String s1 = "Hello";
char *p = &s1[1]; // 此時s1計數器為 1
String s2 = s1;    // 此後s1計數器為 2
*p = 'x'; // 同時修改了 s1 和 s2，不妙！
```



String copy constructor 無法偵測出這個問題，因為它無法知道「目前存在一個指標指向 s1 的 StringValue 物件」。解法是為每一個 StringValue 物件加上一個 flag 變數，用以指示可否被共享。一開始我們先豎立此 flag (表示物件可被共享)，一旦 non-const operator [] 作用於物件身上 (我們悲觀地假設它被用於塗寫)，就將該 flag 清除 (表示不可被共享)。

實作：

1. class StringValue 增加一個 bool shareable;
2. class StringValue 的 ctor initialization list 中將 shareable 設為 true.
3. class String 的 copy ctor 中判斷 shareable 真偽，決定相應的 copy 方式.
4. class String 的 non-const operator [] 中令 value->shareable 為 false.



Reference Counting in SGI string (G2.91)

ref. C:\cygnus\cygwin-b20\include\g++\std\bastring.h and bastring.cc

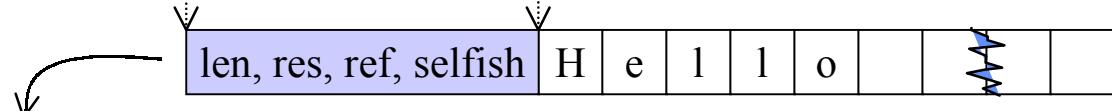
```
template <class charT,
          class traits = string_char_traits<charT>,
          class Allocator = alloc >
class basic_string
{
private:
    struct Rep { ... // inner class. (representation?) ...
        size_t len, res, ref; // ref 就是'reference counter'.
        bool selfish; // Shareable flag.
    };
public:
    static const size_type npos = static_cast<size_type>(-1);
private:
    void selfish() { unique(); rep()->selfish = true; }
private:
    static Rep nilRep;
    charT* dat;
};

basic_string<charT, traits, Allocator>::nilRep = { 0, 0, 1, false };

    // reference counter
    // Shareable flag.
    // inner class. (representation?)
```

typedef basic_string<char> string;

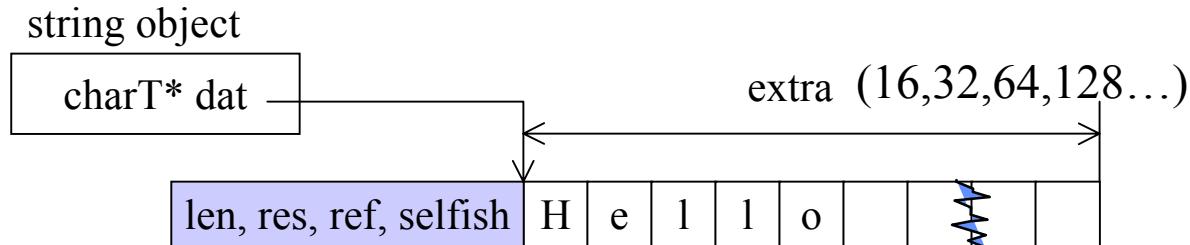
void unique() {
 if (rep()->ref > 1)
 alloc(length(), true);
}



注意：basic_string<T>::Rep::operator new() 被重载了 193



Reference Counting in SGI string (G2.91)



爲何不是`const charT&`（如 MEC 所述）？是否意味我可以修改`const string` 值？不，`return by value` 後萬一被修改並不會影響"本尊"。但無論如何，這個設計不若傳回 `const char&` 好。

copy-on-write 和 shareable flag 的處理。見上頁源碼

`charT&`

這是 `basic_string` 的第一個 template param。
在 `std::string` 中就是 `char`。

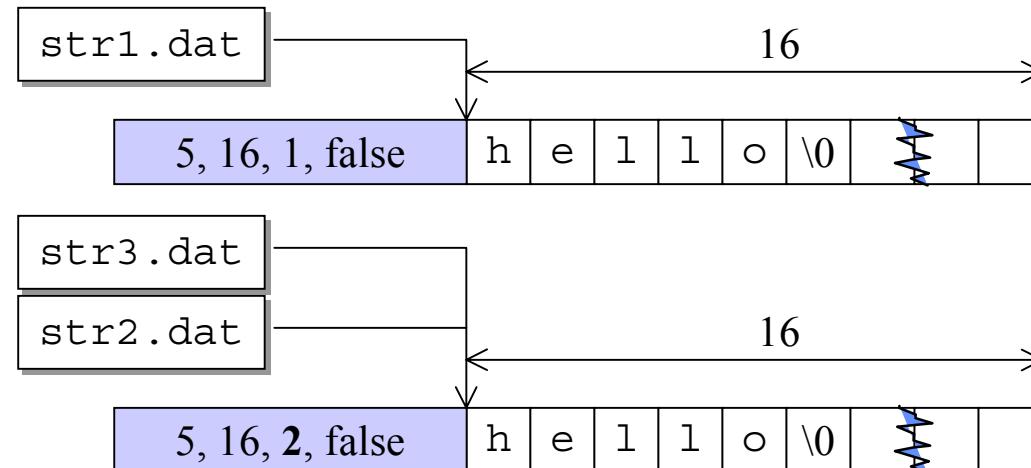
```
charT operator[] (size_type pos) const {  
    if (pos == length())  
        return eos();  
    return data() [pos];  
}  
reference operator[] (size_type pos)  
{ selfish(); return (*rep())[pos]; }
```



Reference Counting in SGI string

如果希望「偵測出 str1, str2內容相等進而採用 reference counting」，以目前所談技術辦不到。

```
string str1("hello");
string str2("hello");
string str3 = str2;
```

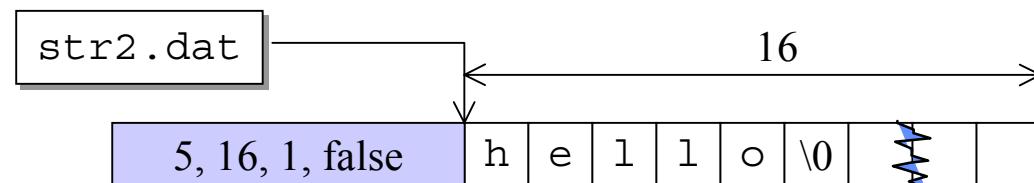
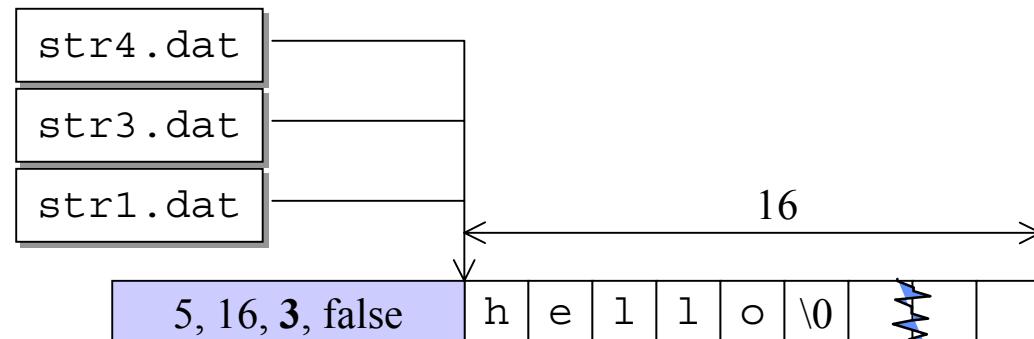




Reference Counting in SGI string

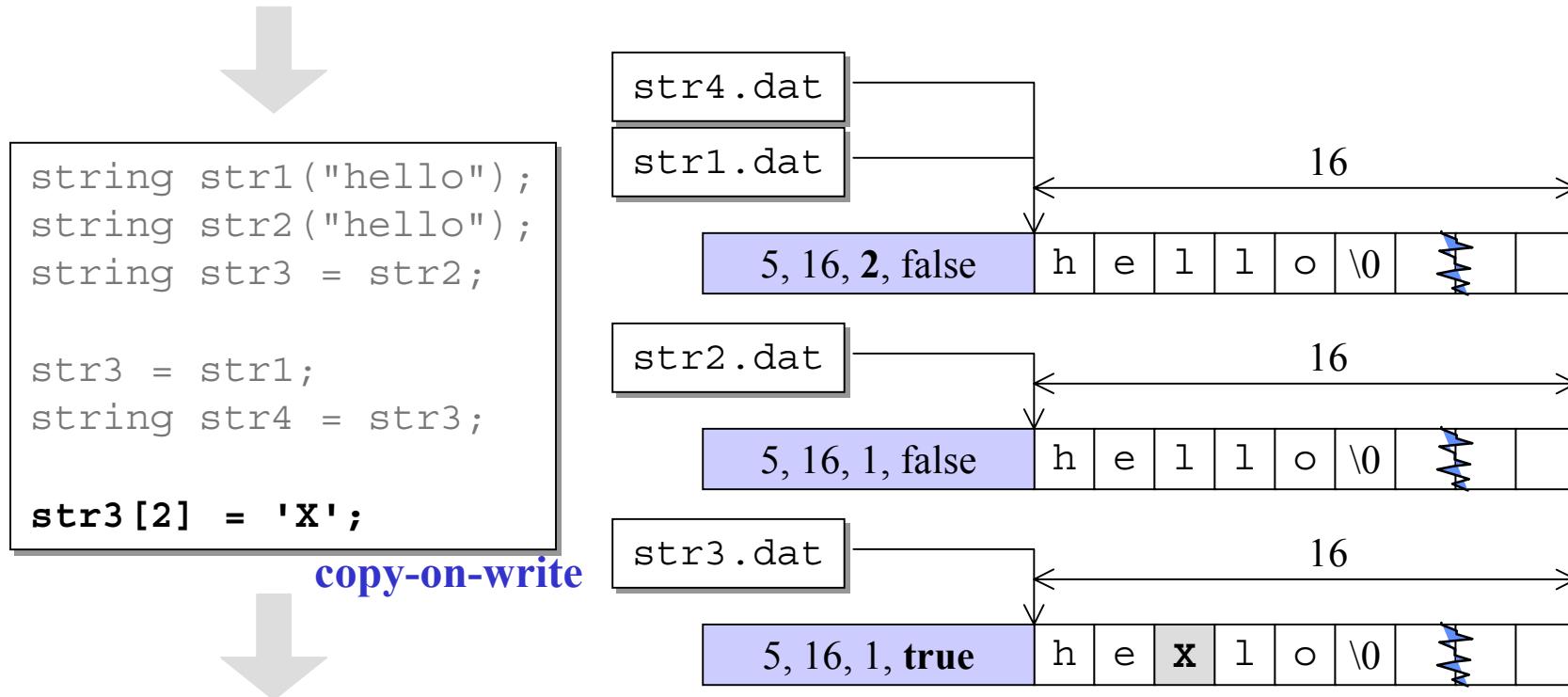
```
string str1("hello");
string str2("hello");
string str3 = str2;

str3 = str1;
string str4 = str3;
```





Reference Counting in SGI string





Reference Counting in SGI String

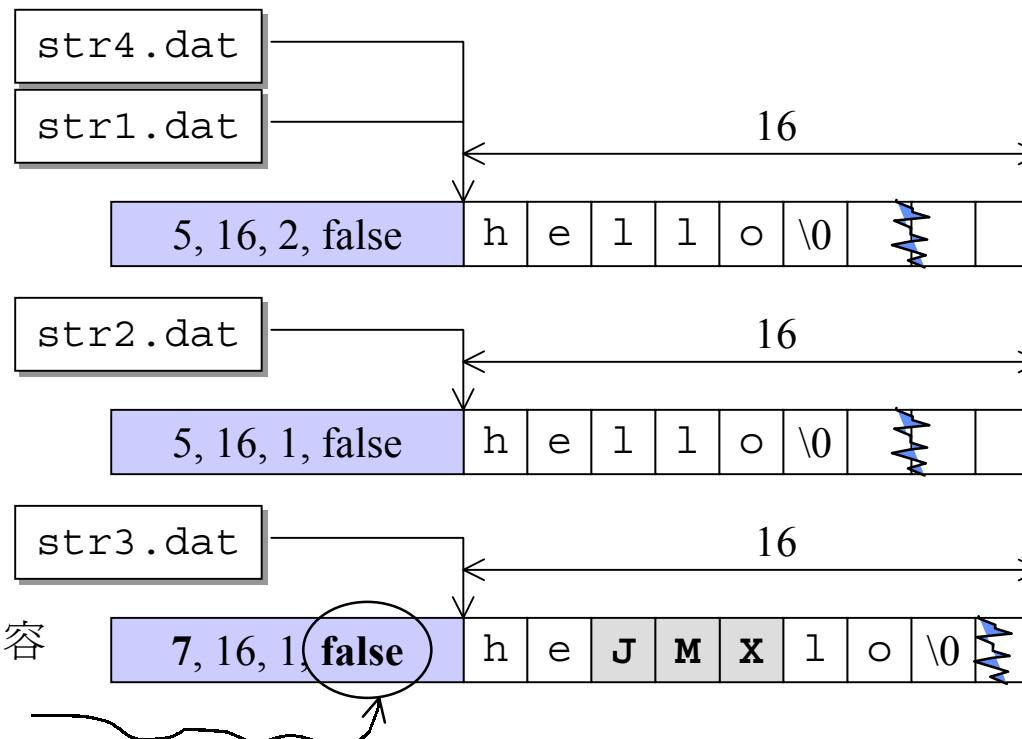


```
string str1("hello");
string str2("hello");
string str3 = str2;

str3 = str1;
string str4 = str3;

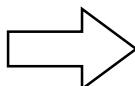
str3[2] = 'X';

str3.insert(2, "JM");
```



只要呼叫了某個可能改變string內容的函式，shareable flag 即被 reset
(先前的設定就都不認賬了)。

為什麼？



注意：insert()會檢查容量，判斷是否還夠容納將被安插進來的string。如果夠，就不做重新配置動作。但從源碼觀知，insert() 呼叫 replace() 呼叫 check_realloc()，後者竟不管3721地做了這動作：rep()->selfish = **false**；我認為應在需重新配置時（新配置得的string的 ref為1）才令selfish為false，否則不應改變其值。本例並未重新配置（由9string-dissect.cpp觀察得知）。



Reference Counting in SGI String

現在我來試驗可能出錯的一種情況



```
string str1("hello");
string str2("hello");
string str3 = str2;

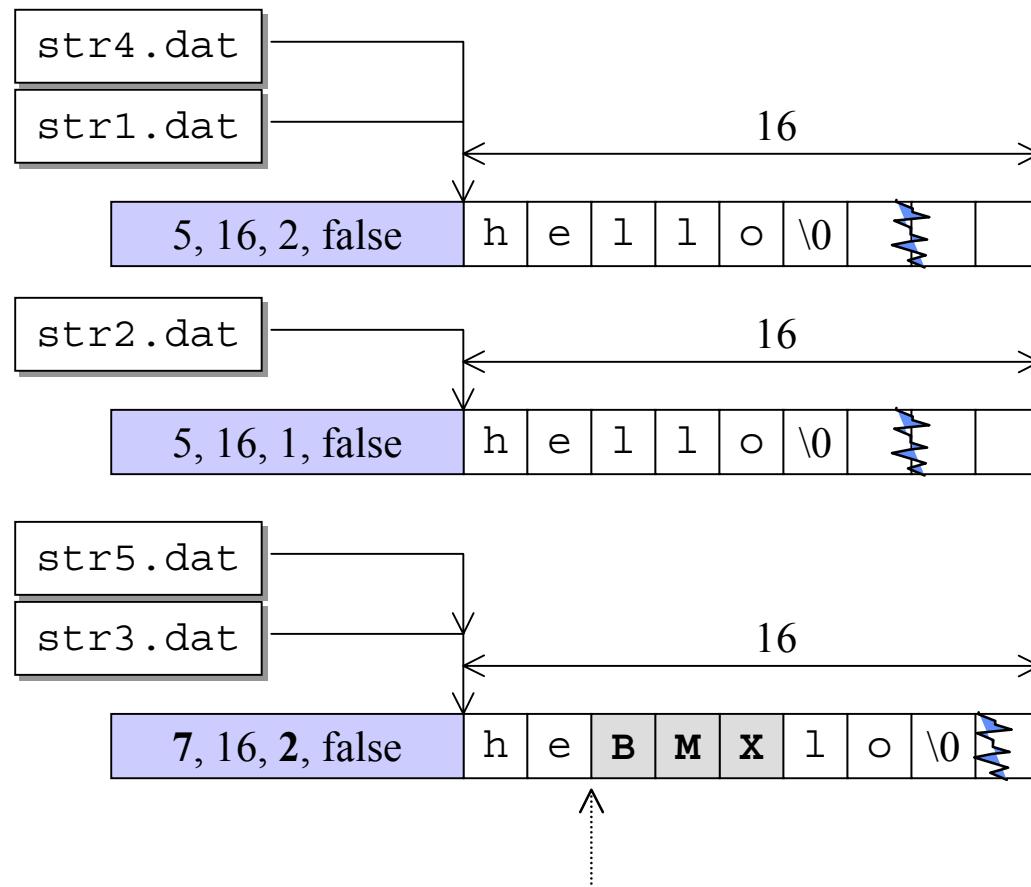
str3 = str1;
string str4 = str3;

str3[2] = 'X';
char* p = &str3[2];

str3.insert(2, "JM");

string str5(str3);
*p = 'B';
```

傳出 handle 後應為 "不可共享"，但 insert() 又將它改為 "可共享"，導致稍後真的被共享，於是出現控制外的 alias 現象。



這個動作「同時改變」了兩個string，不是該有的行爲。
問題的癥結在於 string str5(str3); 不該共享卻被共享了。

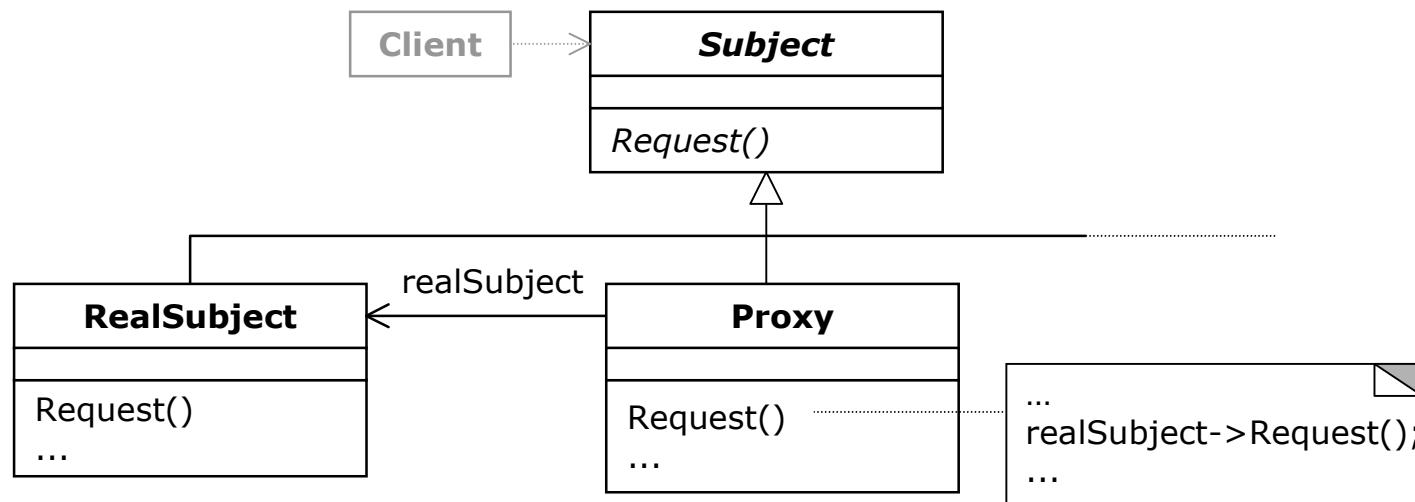


18. Proxy in GOF

Provide a surrogate or placeholder for another object to control access to it.
爲 object 提供一個代理人或佔位符號，用以控制對該 object 的存取

a Proxy references an underlying original object and forwards all the messages it receives to that object. (term. of Buschmann et al. 1996)

proxy 總是指涉（參考、引用）一個底部原件(original object)，並總是將收到的所有訊息轉發給該原件。

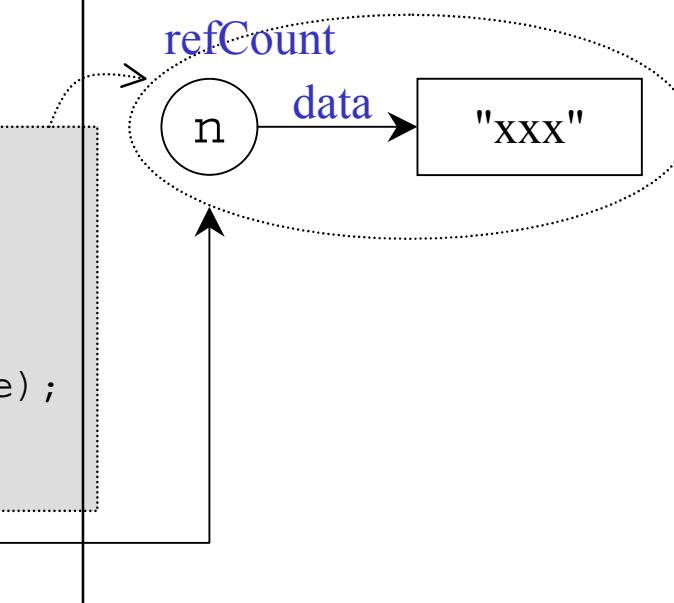




18. Proxy in MEC, distinguishing Reads from Writes via operator[]

string class with reference count :

```
class String {  
public:  
    ... // String members  
private:  
    struct StringValue {  
        int refCount;  
        char *data;  
        bool shareable;  
  
        StringValue(const char *initValue);  
        ~StringValue();  
    };  
    StringValue *value;  
};
```



我們希望區分 `operator[]` 的**左值運用和右值運用**，因為（特別是針對 **reference-counted** 資料結構）「讀取」動作可能比「塗寫」動作簡單快速得多。不幸的是 `operator[]` 內沒有任何辦法可以決定它被呼叫當時的情境。但雖然不可能知道 `operator[]` 是在左值或右值情境下被喚起，我們還是可以區分讀和寫“只要將我們所要的處理動作**延緩**，直至**知道** `operator[]` 的傳回結果將如何被使用為止。



18. Proxy in MEC, distinguishing Reads from Writes via operator[]

Proxy class 讓我們得以買到我們所需要的時間，因為我們可以修改operator []，令它傳回 string char 的一個 proxy，而不傳回 char 本身。然後我們可以等待，看看這個 proxy 如何被運用。如果它被讀，我們可以（有點過時地）將operator [] 的呼叫動作視為一個讀取動作。如果它被寫，我們必須將 operator [] 的呼叫動作視為一個塗寫動作。

對於 proxy，你只有三件事情可做：

- 產生它，本例也就是指定它代表哪一個字串字元。
- 以它做為賦值（assignment）的收受端，這種情況下你是對它所代表的 string char 做賦值動作。如果這麼使用，proxy 代表的將是喚起 operator [] 的那個 string 的左值運用。
- 以其他方式使用之。如果這麼使用，proxy 表現的是喚起 operator [] 的那個 string 的右值運用。

設計 proxy 時需注意：

1. 由於它是個代理人，代表另一個東西（本尊），因此 proxy class 中必須有足夠的 info. 使得取得本尊。所謂「足夠的 info.」以 String 為例是一個 String& 和一個 index，以 vector<bool> 為例是一個 int* 和一個 mask。
2. Proxy class 必須有個轉換函式，用來將自己（分身）轉型為本尊。
3. 何時傳回 proxy？以 String 為例是 op[]，以 vector<bool> 為例是 op[] 和 iterator's op*。
4. 通常還需在 proxy class 內定義 op=，兩個版本：(1) rhs 為 proxy (2) rhs 為 RealSubject type。



18. Proxy in MEC, distinguishing Reads from Writes via operator[]

```
class String { // reference-counted strings;
public:
    class CharProxy { // proxies for string chars
public:
    CharProxy(String& str, int index); // 建構
    1 CharProxy& operator=(const CharProxy& rhs); // 左值運用
    2 CharProxy& operator=(char c); // 左值運用
    operator char() const; // 右值運用。轉換函式
private:
    String& theString; // proxy 所附身（相應）之字串。
    int charIndex; // proxy 代表之（字串內的）字元。
};
```

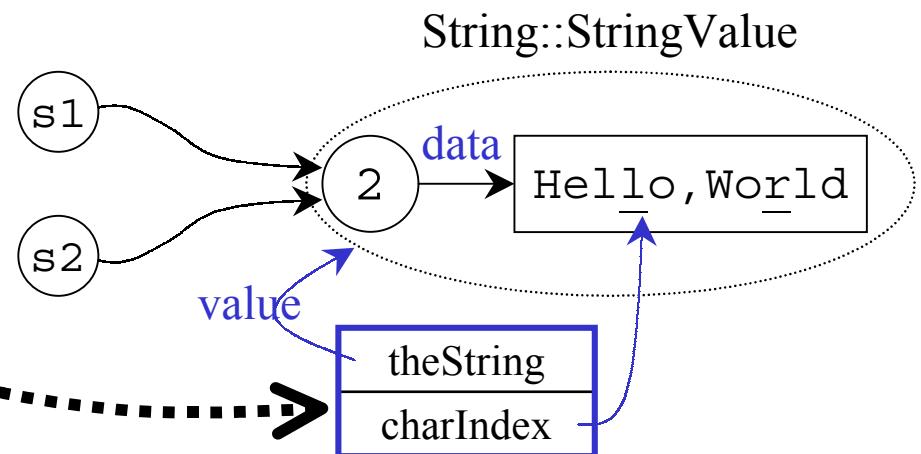
C++罕見情況：使用 reference variable

```
1 const CharProxy operator[](int index) const; // 針對 const Strings
2 CharProxy operator[](int index); // 針對 non-const Strings
...
friend class CharProxy;
private:
    struct StringValue { ... };
    StringValue* value;
};
```



18. Proxy in MEC, distinguishing Reads from Writes via operator[]

```
String s1("Hello,World");
String s2 = s1;
...
cout << s1[3]; // proxy被做為右值
s2[8] = 'x';
s2[3] = s1[8];
```



1 `const String::CharProxy String::operator[](int index) const`
{ return CharProxy(const_cast<String&>(*this), index); }

2 `String::CharProxy String::operator[](int index)`
{ return CharProxy(*this, index); }

const object 呼叫的是 const member function，所以這裡先將 *this (是個 const String) 轉為 non-const String。

proxy's ctor:

```
String::CharProxy::CharProxy(String& str, int index)
: theString(str), charIndex(index) {}
```

轉型函式

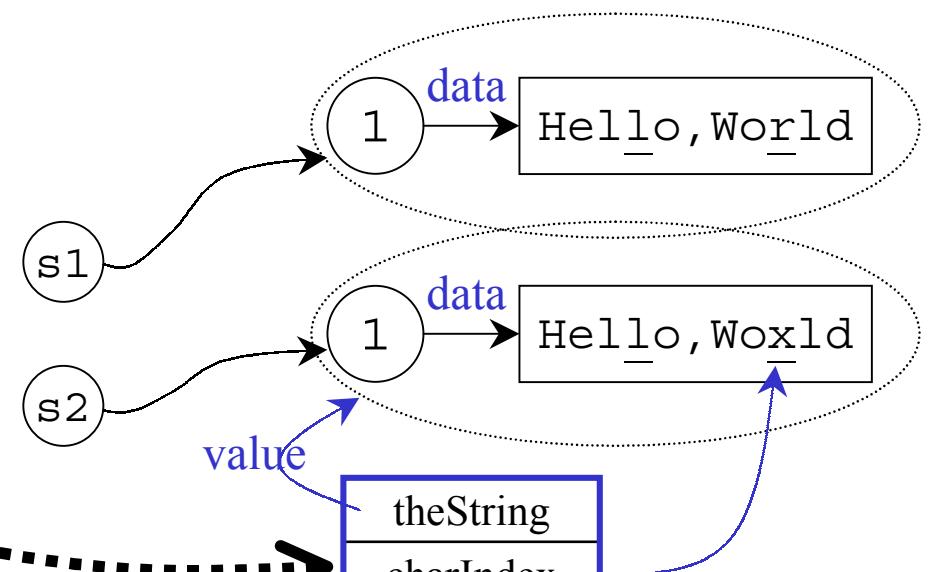
```
String::CharProxy::operator char() const
{ return theString.value->data[charIndex]; } jhou.csdn.net
```



18. Proxy in MEC, distinguishing Reads from Writes via operator[]

```
String s1("Hello,World");
String s2 = s1;
...
cout << s1[3];

s2[8] = 'x'; // proxy被做為左值
s2[3] = s1[8];
```



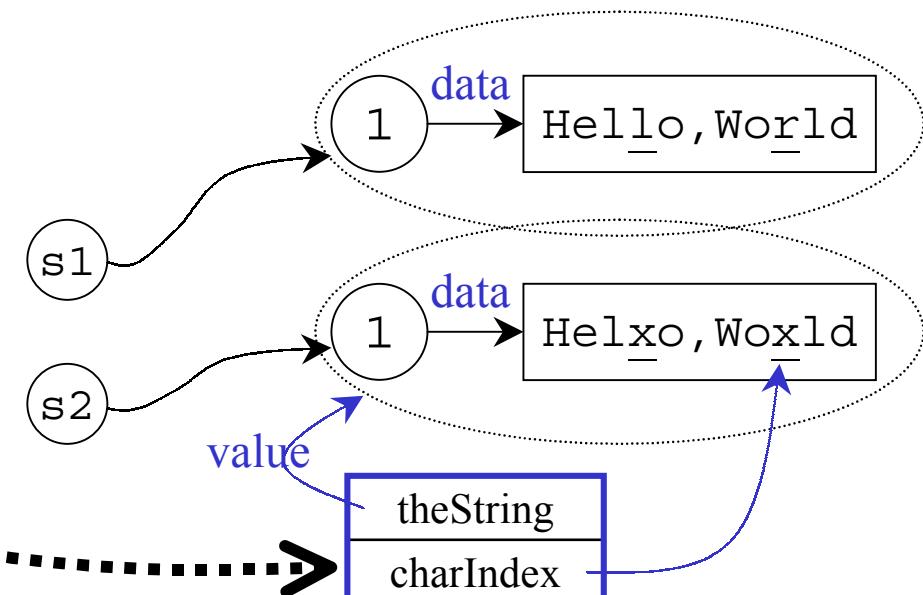
```
1 String::CharProxy& String::CharProxy::operator=(char c)
{
    if (theString.value->isShared()) { // copy-on-write
        theString.value = new StringValue(theString.value->data);
    }
    theString.value->data[charIndex] = c;
    return *this;
}
```



18. Proxy in MEC, distinguishing Reads from Writes via operator[]

```
String s1("Hello,World");
String s2 = s1;
...
cout << s1[3];

s2[8] = 'x';
s2[3] = s1[8]; // proxy被做為左值
```



2

```
String::CharProxy&
String::CharProxy::operator=(const CharProxy& rhs)
{
    if (theString.value->isShared()) { // copy-on-write
        theString.value = new StringValue(theString.value->data);
    }
    theString.value->data[charIndex] =
        rhs.theString.value->data[rhs.charIndex];
    return *this;
}
```



18. Proxy in MEC, distinguishing Reads from Writes via operator[]

我們希望 **proxy object** 能夠無間隙地取代它們所代表的**original object**，但是這樣的想法很難達成。因為除了賦值動作（assignment），物件亦有可能在其他情境下被當做左值使用，而那種情況下的 **proxies** 常常會有與 **original object**（本尊）不同的行爲。

一般而言，「對 **proxy** 取址」所獲得的指標型別和「對真實（本尊）物件取址」所獲得的指標型別不同。除非對 **proxy** 做了 **operator&()** 的重載

另一個問題是「透過 **proxies** 喚起 **original object** 之 member functions」。如果你直率地那麼做，會失敗。為了讓 **proxies** 模仿其所代表之物件的行爲，你必須將適用於「本尊」物件的每一個函式重載於 **proxies** 中。

Proxies 無法取代「本尊」物件的另一種情況是 ... (MEC, p226)

proxies 難以完全取代「本尊」物件的最後一個原因在於 ... (MEC, p226)

proxy classes 的缺點：如果扮演函式傳回值的角色，**proxy objects** 將是一種暫時物件，需要被產生和摧毀。而建構和解構並非毫無成本。此外**proxy classes** 的存在也增加了軟件系統的複雜度，因為額外的 **classes** 使產品更難設計、實作、瞭解、維護。

當 **class** 的身份從「與真實物件合作」移轉到「與 **proxies** 合作」，往往會造成 **class** 語意的改變，因為 **proxy objects** 所展現的行爲往往和本尊物件的行爲有些隱微差異。



18. Proxy in std::vector<bool>

利用編譯選項決定：(1) 如果編譯器支援偏特化，便做出如下東西：

```
#define __BVECTOR vector

#       include <stl_vector.h>
template<class Alloc>
class vector<bool, Alloc>
{
    typedef simple_alloc<unsigned int, Alloc> data_allocator;
    ...
}
```

實作內容相同

(2) 如果編譯器不支援偏特化，便做出如下東西：

```
#define __BVECTOR bit_vector
class bit_vector
{
    typedef simple_alloc<unsigned int, alloc> data_allocator;
    ...
}
```

根據先前數頁中的 "設計 proxy 時需注意" 的討論，這裡的設計焦點放在

1. proxy class 內需有足夠的 info. (本例為 int* 和 mask)
2. 轉型函式 (轉為本尊) : operator bool() const
3. vector<bool>'s op[] 和 vector<bool>::iterator's op* 兩處傳回 proxy。
4. proxy 本身必須重載 op=，其 rhs 有兩個版本：(1) proxy (2) bool (本尊型別)。



18. Proxy in std::vector<bool>

```
public:  
    typedef bool value_type;  
    typedef size_t size_type;  
    typedef ptrdiff_t difference_type;  
    → typedef __bit_reference reference; ← Proxy  
    typedef __bit_reference* pointer;  
    typedef __bit_iterator iterator;
```

```
protected:  
    iterator start;  
    iterator finish;  
    unsigned int* end_of_storage;  
    unsigned int* bit_alloc(size_type n) {  
        return data_allocator::allocate((n + __WORD_BIT - 1) / __WORD_BIT);  
    }  
    void deallocate() {  
        if (start.p)  
            data_allocator::deallocate(start.p, end_of_storage - start.p);  
    }
```

```
reference operator[](size_type n) {  
    return *(begin() + difference_type(n));  
}
```

檢討：string需要proxy，是為了判斷op[]被用於左值(write)或右值(read)，以便決定是否要copy-on-write。但vector<bool>為什麼也需要proxy呢？

如果n為1，就配置1個unsigned int
如果n為32，就配置1個unsigned int
如果n為33，就配置2個unsigned int
如果n為47，就配置2個unsigned int

Proxy

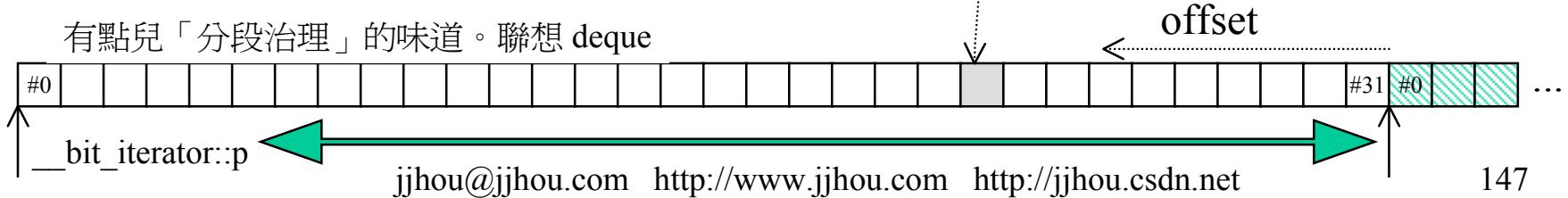


18. Proxy in std::vector<bool>::iterator

```
static const int __WORD_BIT = int(CHAR_BIT*sizeof(unsigned int));  
  
struct __bit_iterator : public random_access_iterator<bool, ptrdiff_t> {  
    typedef __bit_reference reference;  
    typedef __bit_reference* pointer;  
    typedef __bit_iterator iterator;  
  
    unsigned int* p;  
    unsigned int offset;  
  
    void bump_up() { //offset移一位。若因此臨界就跳一個unsigned int  
        if (offset++ == __WORD_BIT - 1) {  
            offset = 0;  
            ++p;  
        }  
    }  
    void bump_down() { //offset移一位。若因此臨界就跳一個unsigned int  
        if (offset-- == 0) {  
            offset = __WORD_BIT - 1;  
            --p;  
        }  
    }  
    ... (待續)  
}
```

例如 [p,9]，它所對應的 proxy 是 [p,00000200]

mask



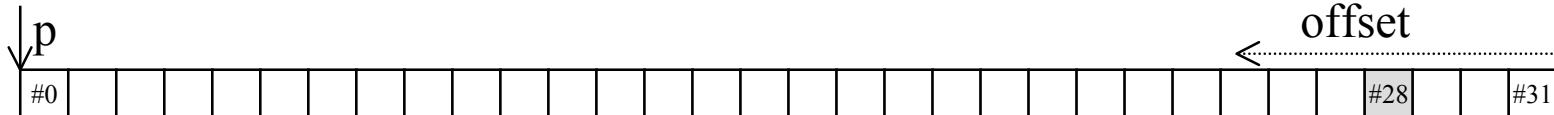


18. Proxy in std::vector<bool>::iterator

```
static const int __WORD_BIT = int(CHAR_BIT*sizeof(unsigned int));  
  
struct __bit_iterator : public random_access_iterator<bool, ptrdiff_t> {  
    ... (續上)  
    __bit_iterator() : p(0), offset(0) {}  
    __bit_iterator(unsigned int* x, unsigned int y) : p(x), offset(y) {}  
    reference operator*() const { return reference(p, 1U << offset); }  
    iterator& operator++() {  
        bump_up();  
        return *this;  
    }  
    iterator operator++(int) {  
        iterator tmp = *this;  
        bump_up();  
        return tmp;  
    }  
    iterator& operator--() {  
        bump_down();  
        return *this;  
    }  
    iterator operator--(int) {  
        iterator tmp = *this;  
        bump_down();  
        return tmp;  
    }  
    ...  
};
```



18. Proxy in std::vector<bool>::reference



```
struct __bit_reference { ← Proxy
    unsigned int* p;
    unsigned int mask; // 例如 00000008
    __bit_reference(unsigned int* x, unsigned int y) : p(x), mask(y) {}

public:
    __bit_reference() : p(0), mask(0) {}
    // 以下為轉換函式。由於此proxy 代表 bool，所以應該要有這樣一個轉換函式
❶ operator bool() const { return !(>(*p & mask)); } // 兩次not可保原值不變，而卻獲得bool
    // 以下針對此 proxy 設定 boolean 值(x)
❷ __bit_reference& operator=(bool x) {
    if (x)
        *p |= mask; // 只要以mask 操作之即可
    else
        *p &= ~mask; // 只要以mask 操作之即可
    return *this;
}
    // 以下令此 proxy 等同於另一個 proxy。呼叫以上兩個函式，即可達成
❸ __bit_reference& operator=(const __bit_reference& x) { return *this = bool(x); }
    void flip() { *p ^= mask; } // 只要以mask 操作之即可
};
```

如果 offset 為 3
則 mask 為 00000008

```
vector<bool> v;
```

- ❶ cout << v[5];
- ❷ v[5] = true;
- ❸ v[5] = v[3];

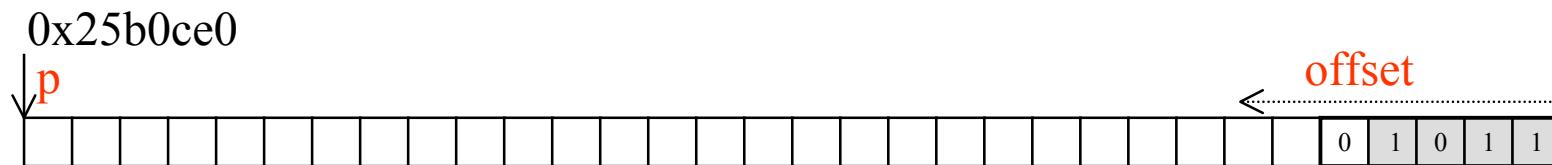


18. Proxy in std::vector<bool>::reference

ref\tass\prog\9vector-bool-dissect.cpp

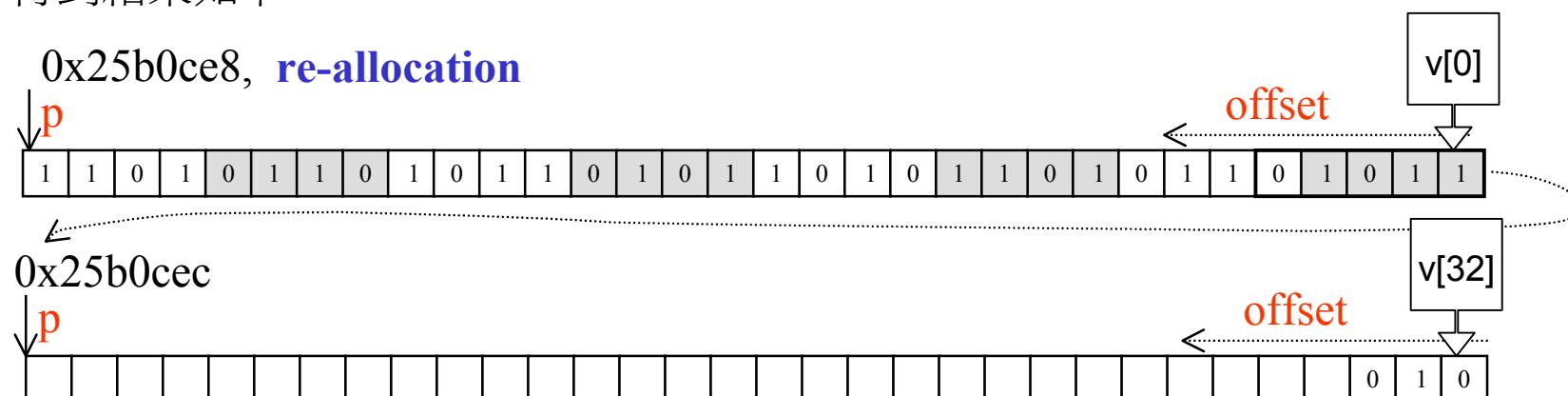
如果 push_back : 11010

得到結果如下：



然後再 push_back : 11010 11010 11010 11010 11010 11010

得到結果如下：



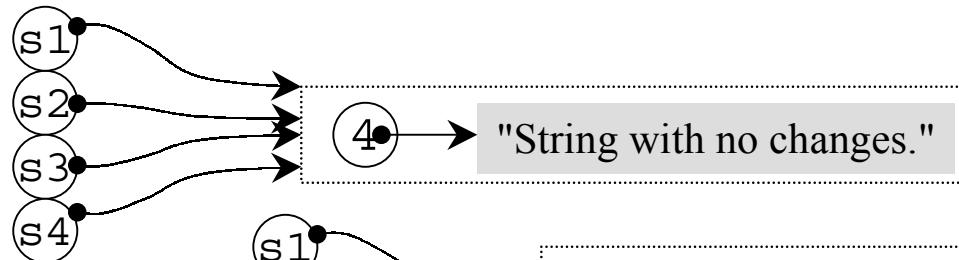
也就是說，各個 32-bit blocks 彼此並不顛倒放置，但每個 32-bit block 內卻是顛倒放置



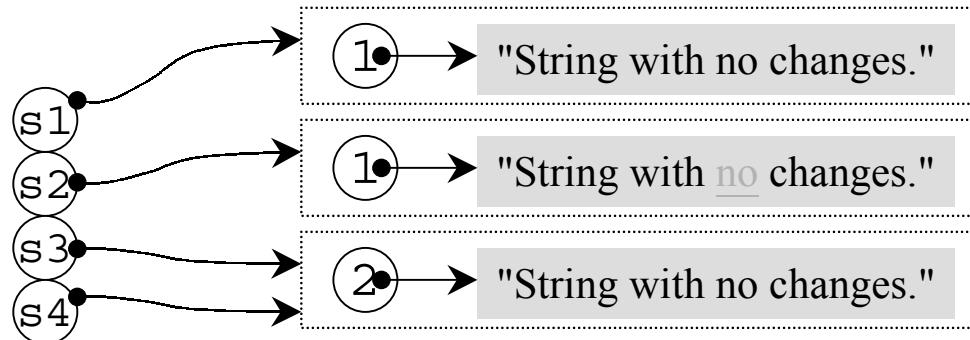
Reference Counting+Proxy 習題

```
String s1 = "String with no changes.";  
String s2(s1);  
String s3(s1);  
String s4;  
s4=s3;
```

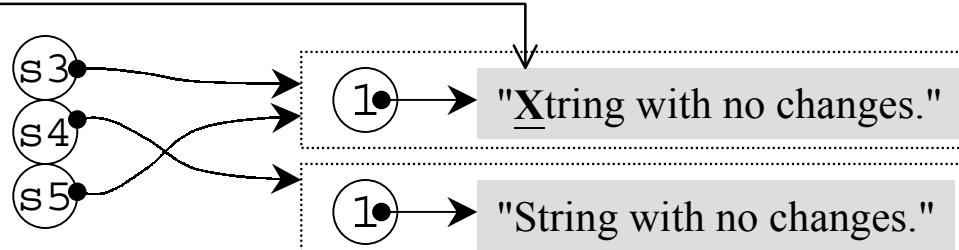
cout << s1[12] << s1[13] << endl;
//op[]引發 CopyOnWrite (冤枉了)



s2[12] = s2[13] = ' ';
//op[]引發 CopyOnWrite (不冤枉)



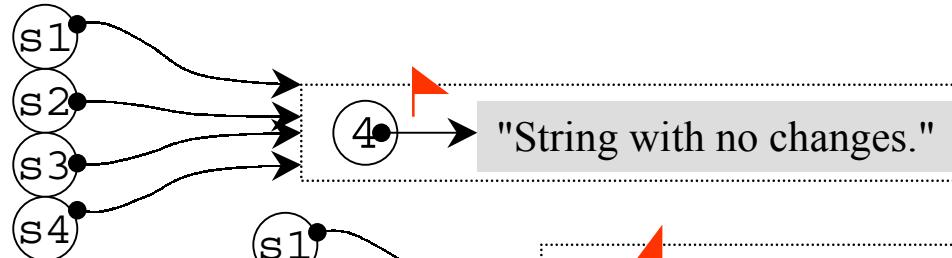
```
char* p = &(s3[0]);  
//op[]引發 CopyOnWrite  
  
String s5(s3);  
*p = 'X'; //s5會因此抓狂！
```



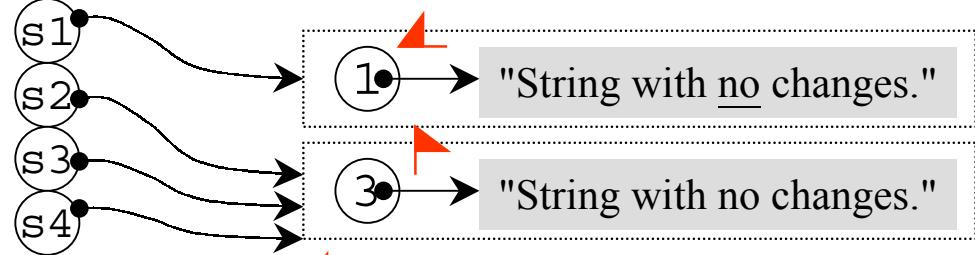


Reference Counting+Proxy 習題. 測試 v2, with shareable flag

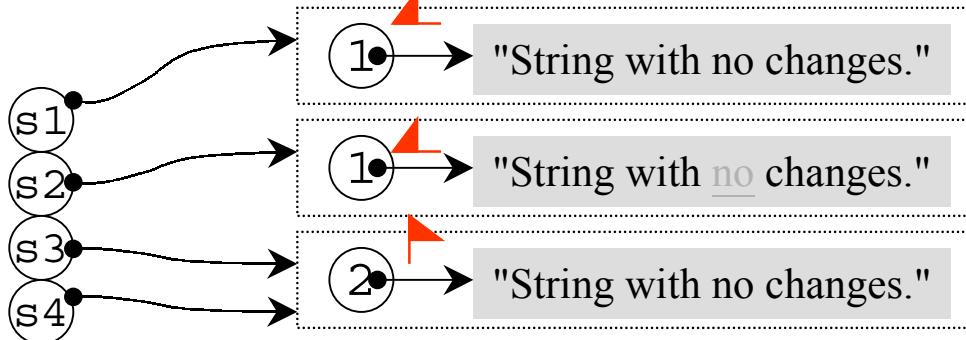
```
String s1 = "String with no changes.";  
String s2(s1);  
String s3(s1);  
String s4;  
s4=s3;
```



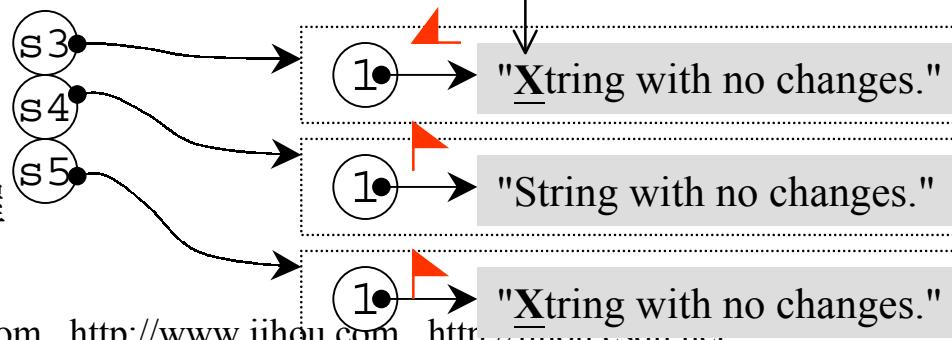
```
cout << s1[12] << s1[13] << endl;  
//op[]引發 CopyOnWrite (冤枉了)  
//flag off
```



```
s2[12] = s2[13] = ' ';  
//op[]引發 CopyOnWrite (不冤枉)  
//flag off (冤枉)
```



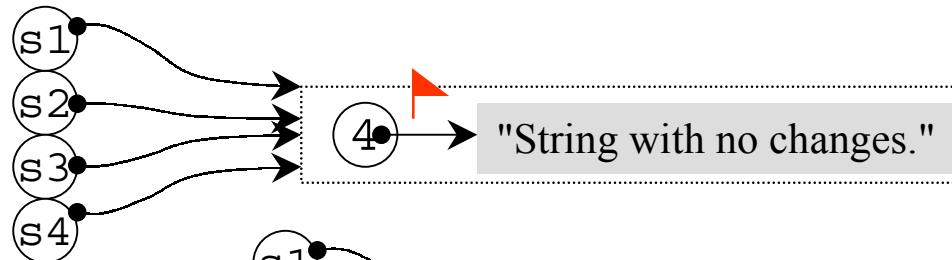
```
char* p = &(s3[0]);  
//op[]引發 CopyOnWrite  
//flag off  
String s5(s3); //s3不可共享喔  
*p = 'X'; //這次不再殃及s5
```



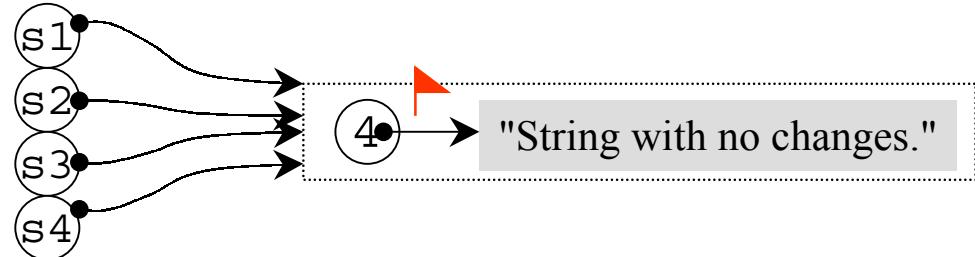


Reference Counting+Proxy 習題. 測試 v3, shareable flag+proxy

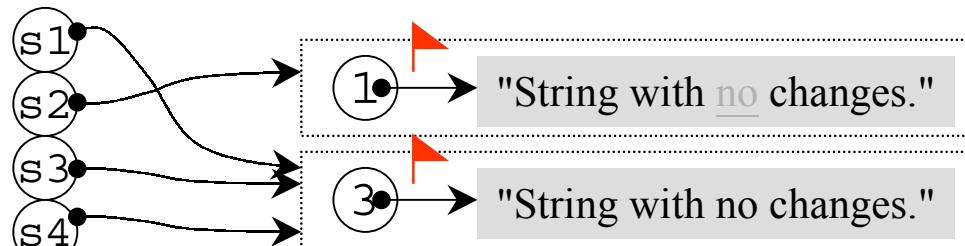
```
String s1 = "String with no changes.";
String s2(s1);
String s3(s1);
String s4;
s4=s3;
```



```
cout << s1[12] << s1[13] << endl;
//proxy's char() 不引發 COW
//省時省力
```

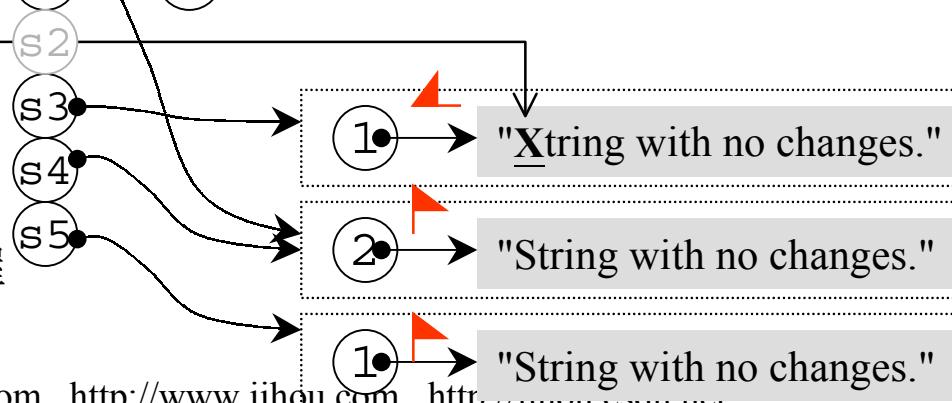


```
s2[12] = s2[13] = '';
//proxy's op= 引發 COW (不冤枉)
//flag on (新物當然可被共享)
//這裡引發 proxy's
// op=(char) 和 op=(proxy) 
```



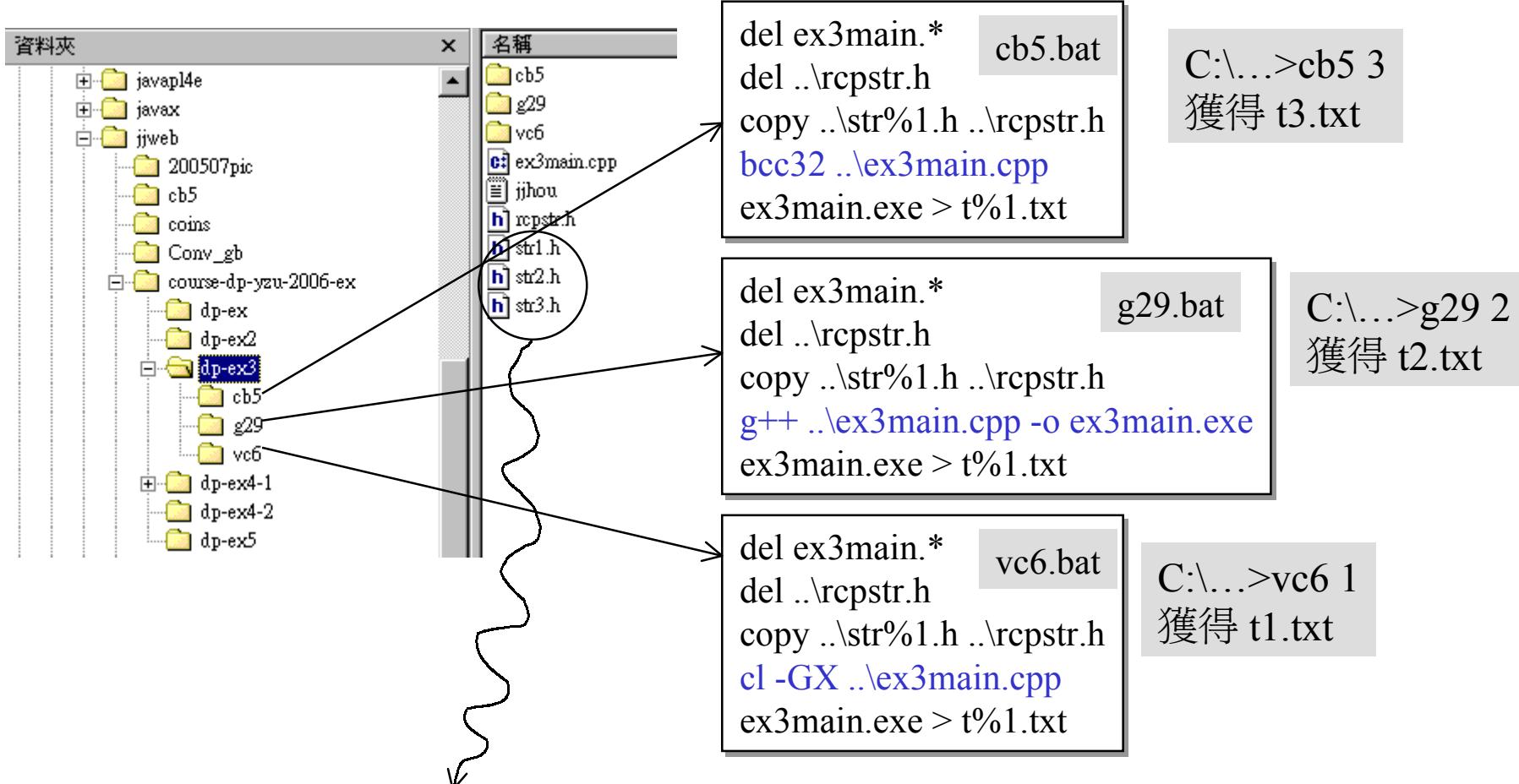
```
char* p = &(s3[0]);
//proxy's op& 引發 COW
//flag off (不冤枉)
String s5(s3); //s3不可共享喔
*p = 'X';

```





Reference Counting+Proxy 習題. 發展環境



str1.h : Reference Counting String without Shareable flag

str2.h : Reference Counting String with Shareable flag

str3.h : Reference Counting String with Shareable flag and CharProxy



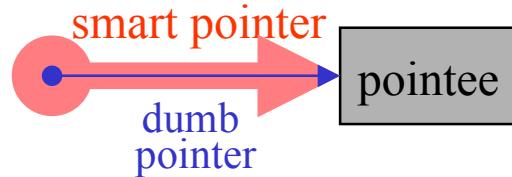
Smart Pointers, in MEC

所謂 smart pointers，是「看起來、用起來、感覺起來都像內建指標，但提供更多機能」的一種物件。當你以 smart pointers 取代 C++ 的內建指標（亦即所謂的 dumb pointers），你將獲得以下各種指標行爲的控制權：

- **建構和解構 (Construction and Destruction)**。你可以決定 smart pointer 被產生以及被摧毀時發生什麼事。通常我們會給 smart pointers 一個預設值 0，以避免「指標未獲初始化」的頭痛問題。某些 smart pointers 有責任刪除它們所指的物件“ 當指向該物件的最後一個 smart pointer 被摧毀時。這是消除資源遺失問題的一大進步。
- **複製和賦值 (Copying and Assignment)**。當一個 smart pointer 被複製或被賦值時，你可以控制發生什麼事。某些 smart pointer 會希望在此時刻自動為其所指之物進行複製或賦值動作，也就是執行所謂的深層複製 (deep copy)。另一些 smart pointer 則可能只希望指標本身被複製或賦值就好。還有一些則根本不允許複製和賦值。不論你希望什麼樣的行爲，smart pointers 都可以讓你如願。
- **提領 (Dereferencing)**。當 client 提領（取用）smart pointer 所指之物時，你有權決定發生什麼事情。例如你可以利用 smart pointers 協助實作出《More Effective C++》條款17所說的 lazy fetching 策略。



Smart Pointers, in MEC



常見的smart pointers :

- `str::auto_ptr<T>`
- `boost::shared_ptr<T>`
- `str::Container<T>::iterator`

```
template<class T>
class SmartPtr
{
public:
    SmartPtr(T* realPtr = 0);
    SmartPtr(const SmartPtr& rhs);
    ~SmartPtr();

    SmartPtr& operator=(const SmartPtr& rhs);

    T* operator->() const;
    T& operator*() const;

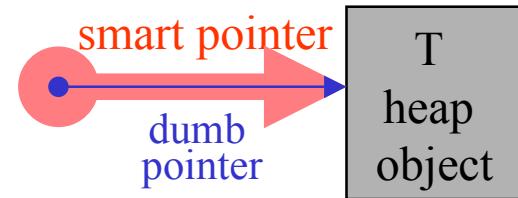
private:
    T *pointee; //dumb pointer
};
```

使用 **smart pointer** 和使用 **dumb pointer**，兩者之間沒有太大差別。這正是封裝（encapsulation）的有效證據。我們可以想像，**smart pointers** 的使用者能夠把它們視同 **dumb pointers**。但是，Daniel Edelson 下過一個註解：
smart pointers 雖然 *smart*，卻不是 *pointers*。你絕對無法成功設計出一個泛用型 **smart pointer**，可以完全無間隙地取代 **dumb pointer**。儘管如此，**smart pointers** 能夠為你完成一些原本極端困難達成的效果。

Smart Pointers, in MEC

常見的smart pointers :

- `str::auto_ptr<T>`
- `boost::shared_ptr<T>`
- `str::Container<T>::iterator`



```
template<class T>
class autoPtr
{
```

public:

```
    explicit autoPtr(T* p=0)
        : m_ptr(p) { }
```

ref. *More Effective C++*, p291
for a full implementation of auto_ptr

auto ptr 的關鍵在這兒

```
~autoPtr() { delete m_ptr; }
```

```
T& operator*() const { return *m_ptr; }
```

```
T* operator->() const { return m_ptr; }
```

```
private:
    T* m_ptr;
};
```

```
{
    auto_ptr<Complex> pc_auto(new Complex(3, 4));
    cout << *pc_auto << endl; // (3, 4i)
    pc_auto->real(3);
}
```

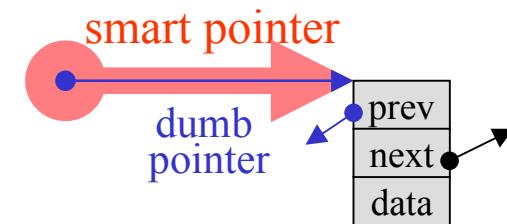
Q: 如果"雞婆"地於scope結束前自行delete ptr 會如何?
A: 重複 delete 兩次有可能出錯（視編譯器而定）



Smart Pointers, in MEC

常見的smart pointers :

- `std::auto_ptr<T>`
- `boost::shared_ptr<T>`
- `std::Container<T>::iterator`



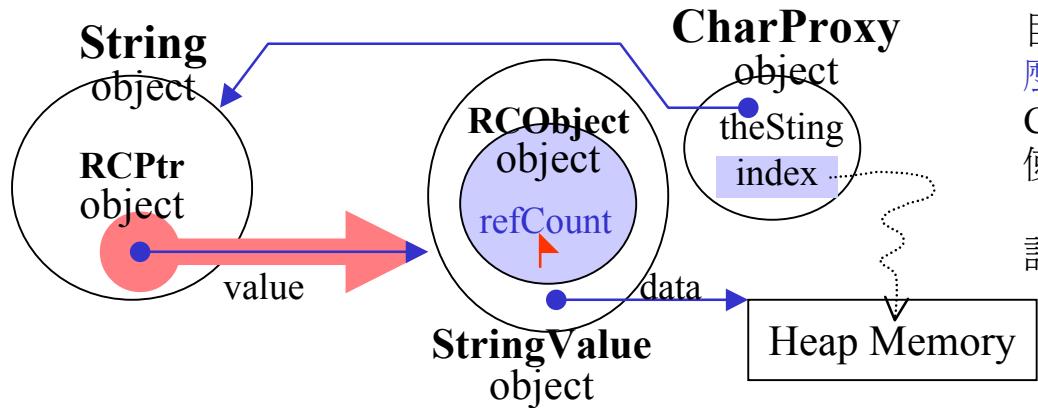
```
template<class T, class Ref, class Ptr>
struct __list_iterator { ←
    typedef __list_iterator<T, T&, T*> iterator;
    typedef __list_iterator<T, Ref, Ptr> self;
    typedef Ptr pointer;
    typedef Ref reference;
    typedef __list_node<T>* link_type;
    link_type node; // list迭代器內部
                     // 有個原生指標指向節點
    // constructor
    __list_iterator() {}
    reference operator*() const { return (*node).data; }
    pointer operator->() const { return &(operator*()); }
    self& operator++() { node = (link_type)((*node).next); return *this; }
    self& operator--() { node = (link_type)((*node).prev); return *this; }
    ...
};
```

```
template <class T, class Alloc = alloc>class list {
protected:
    typedef __list_node<T> list_node;
public:
    typedef list_node* link_type;
    typedef __list_iterator<T, T&, T*> iterator;
protected:
    link_type node; // 只要一個指標便可
                     // 表示整個環狀雙向串列
};
```

```
template <class T>
struct __list_node {
    typedef void* void_pointer;
    void_pointer prev;
    void_pointer next;
    T data;
};
```



RCObject, RCPtr in "MEC"

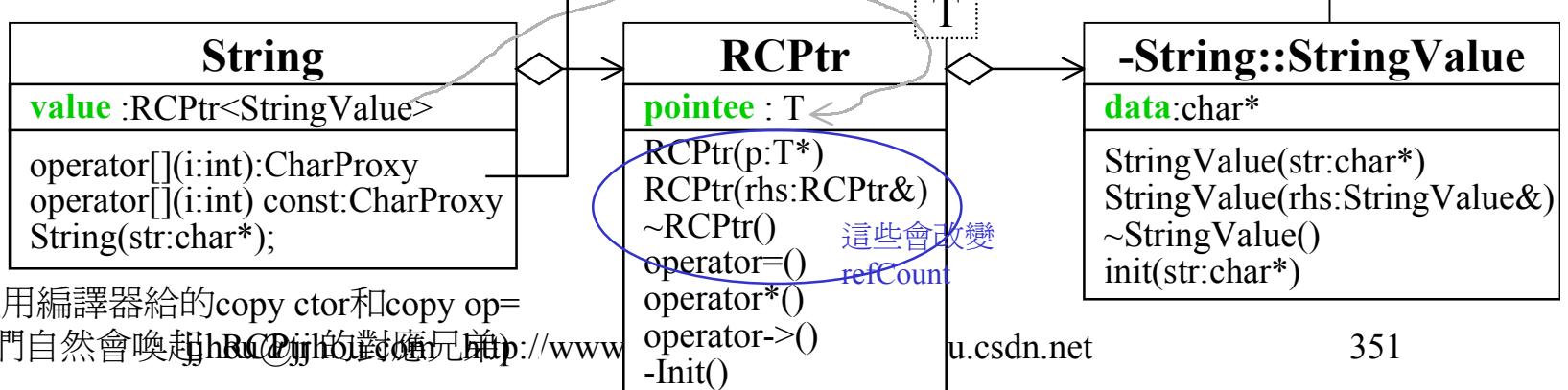


檢討Reference Counting String 發現，「可被共享物」如本例之 `StringValue`，其某些（專屬於 reference counting 技術上的）作為，應被 reuse，因此將它提昇為一個 base class **RCObject**。而當 `String::value` 指標發生移轉 (`op=`) 或建構/解構時總會（需要）修改 `refCount`，因此又令 `String::value` 成為一個 Smart Pointer **RCPtr**。如此一來 **RCObject** 和 **RCPtr** 都可 reuse。

目前的 `CharProxy` 內記錄的是 `String&`，為什麼不能直接記錄 `StringValue&`？因為 `CharProxy` 內的 COW 動作必須從 `String` 出發，使用其內的 `RCPtr`，才能獲得 smart ptr 帶來「自動處理 `refCount`」的好處。當然可以直接記錄 `RCPtr&`，那和記錄 `String&` 沒啥兩樣。

RCObject
<code>RefCount : int</code>
<code>shareable:bool</code>
<code>addRef()</code>
<code>removeRef()</code>
<code>markUnshareable()</code>
<code>isShared()</code>
<code>isShareable()</code>
<code>RCObject()</code>
<code>RCObject(rhs:RCObject&)</code>
<code>~RCObject()=0</code>
<code>op=(rhs:RCObject&)</code>

-String::CharProxy
<code>theString:String&</code>
<code>charindex:int</code>
<code>CharProxy(str:String&, i:int)</code>
<code>op=(c:char)</code> 這會 COW
<code>op=(rhs:CharProxy&)</code>
<code>op char():char</code>
<code>op&():char*</code> 這會改變





7. Composite

7. Composite (163)

Compose objects into tree structures to represent part-whole hierarchies.

Composite lets clients treat **individual objects** and **compositions of objects** uniformly.

將物件(s) 組成/構成 為樹狀結構，用以表示 "局部-全部" 階層體系。

Composite 可以讓 clients 以一致的方式對待「個別物件」和「合成物件」。

- **Composite**：對**內容**和**容器**一視同仁。或說對單複數一視同仁：把單數集中在一起成爲複數時，仍可視之爲一個單數。
- **Decorator**：對**內容**和**裝飾後的內容**一視同仁。



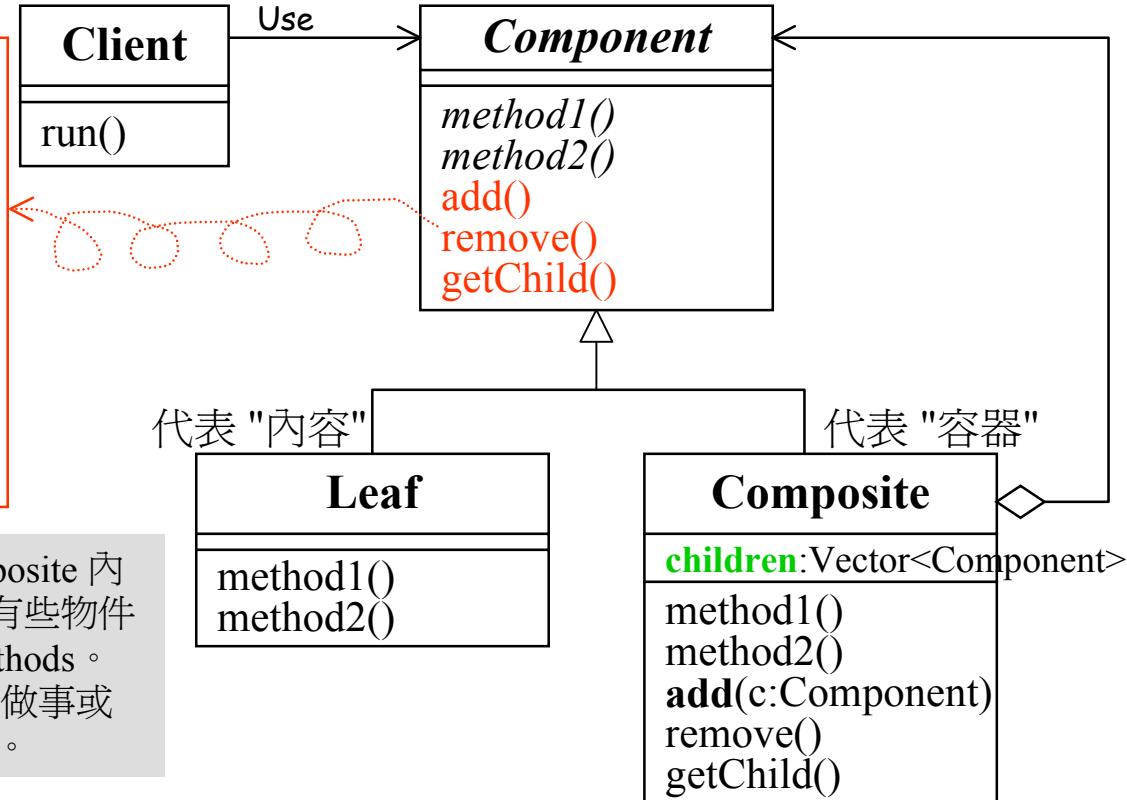
7. Composite

Component 負責兩個任務：(1)管理繼承體系（manages a hierarchy）(2)執行 Composite 相關操作（performs operations related to Composite），這就違反了 SRP。它以犧牲 SRP 來換取透通性（transparency）— 對Leaf 和Composite 一視同仁。

這些只在 **Composite** 才會用上的 methods，如果 Leaf 去用它們是不對的。因此有數種可能作法：

- 放在 **Component** 中並令其引發異常（如UnsupportedOperationException），再於 **Composite** 中覆寫之。
- 同上，但**Component** 內的是空實作。
- 在 **Component** 中只是宣告(abstract)。在 subclasses 內有需要則實作。
- 只放在 **Composite** 內。

爲了保有透通性（transparency），Composite 內所有物件都必須實作相同介面。這意味有些物件具備了一些（對它自己）沒有意義的methods。這種情況下，有時候可讓這些methods不做事或返回 null 或 false，更激烈則是丟出異常。



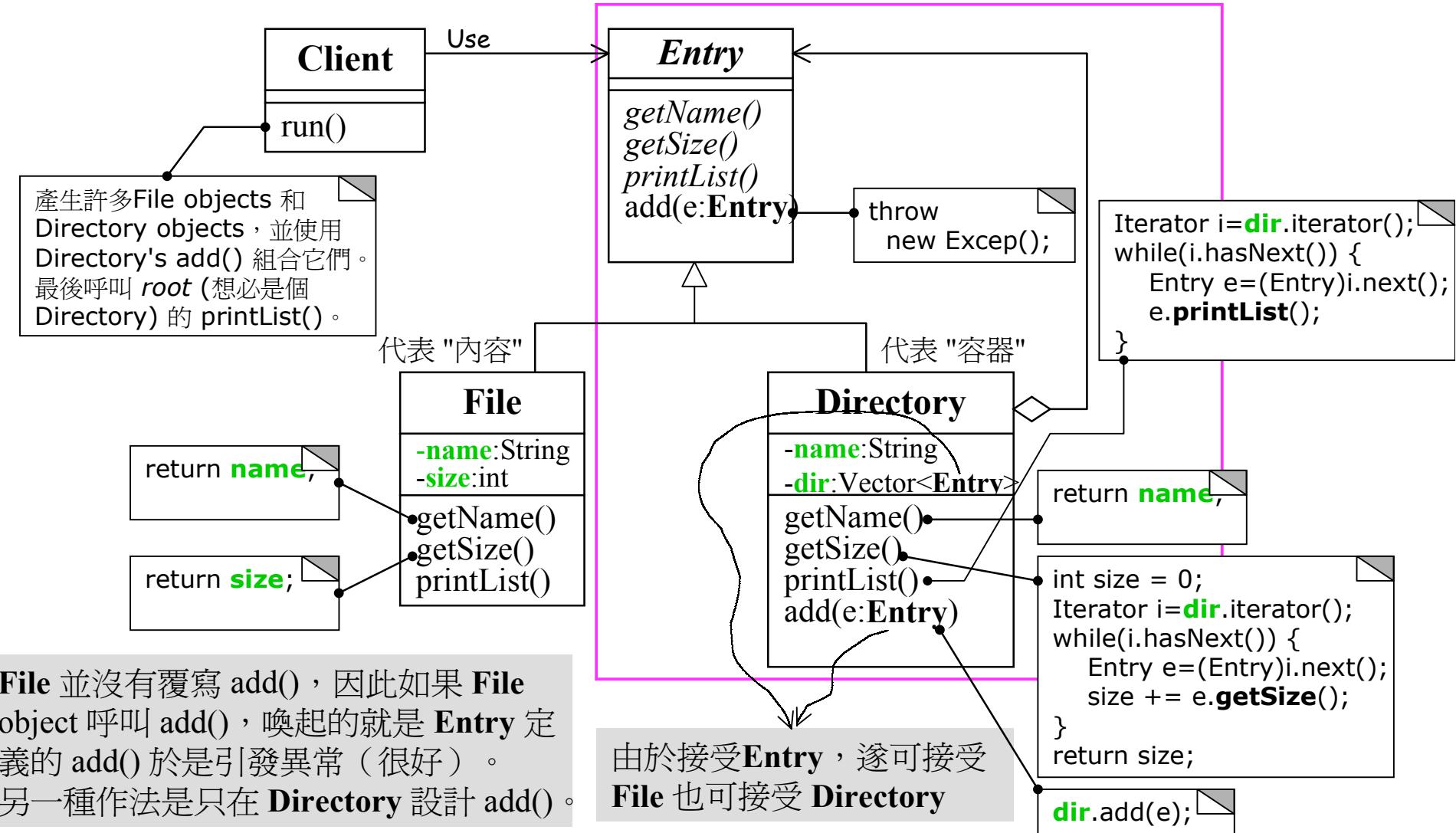
常見例子：

- File system : directory 內有許多 files。然而 files 和 directories 被一視同仁。
- Windowing system : 每一個視窗內還可以再開視窗。
- 另，本身是樹狀結構的 data structures，也就相當於 **Composite**。
- **Command** 中建立 macro command 時也會用到 **Composite**。



7. Composite in DP-in-Java

Composite (注意與 Decorator 的差異)

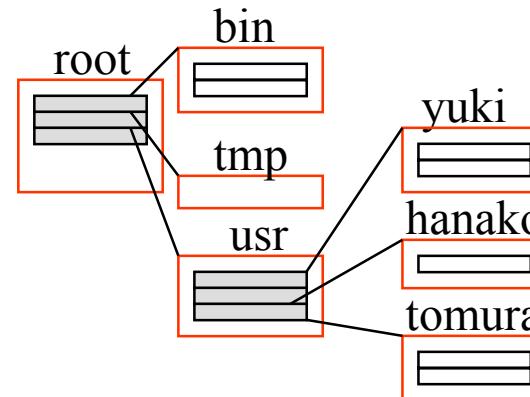




7. Composite in DP-in-Java

```
Making root entries...
/root (30000)
/root/bin (30000)
/root/bin/vi (10000)
/root/bin/latex (20000)
/root/tmp (0)
/root/usr (0)

Making user entries...
/root (31500)
/root/bin (30000)
/root/bin/vi (10000)
/root/bin/latex (20000)
/root/tmp (0)
/root/usr (1500)
/root/usr/yuki (300)
/root/usr/yuki/diary.html (100)
/root/usr/yuki/Composite.java (200)
/root/usr/hanako (300)
/root/usr/hanako/memo.tex (300)
/root/usr/tomura (900)
/root/usr/tomura/game.doc (400)
/root/usr/tomura/junk.mail (500)
```

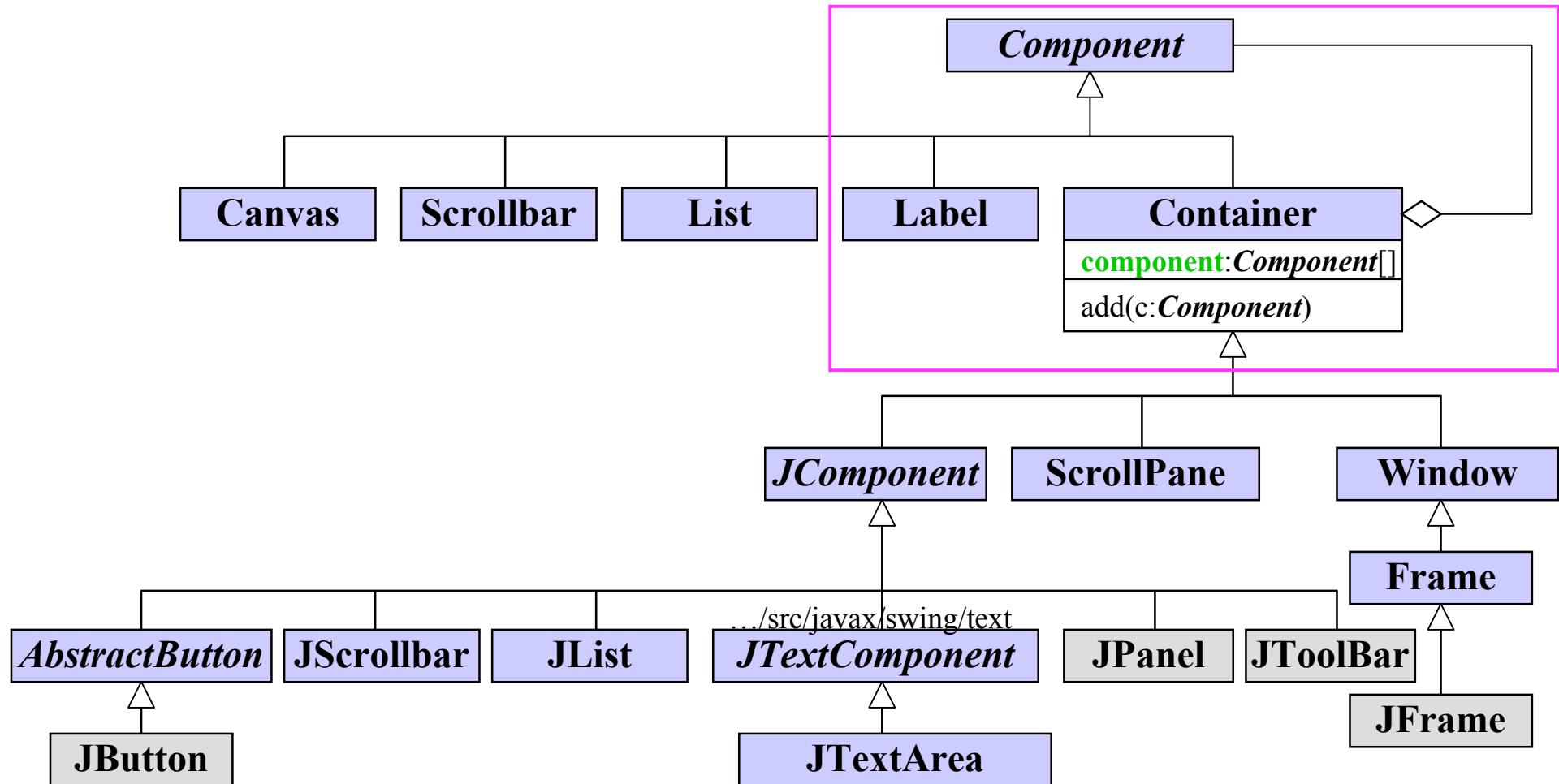




7. Composite in Java Lib.

in Java Library (.../src/java.awt, .../src/javax/swing)

Composite





8. Decorator

Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

將額外的「權與責」以動態方式附著於物件身上，使不必進行 subclassing 就能擴展功能。

Composite：對內容和容器一視同仁。Composite class 內含一個 "collection of Component" field。

Decorator：對內容和裝飾框（含被裝飾內容）一視同仁。Decorator class 內含一個 Component object（被飾物）field。**Decorator** 和 **Composite** 兩者在「處理遞迴架構」的部分很像，但目的不盡相同。

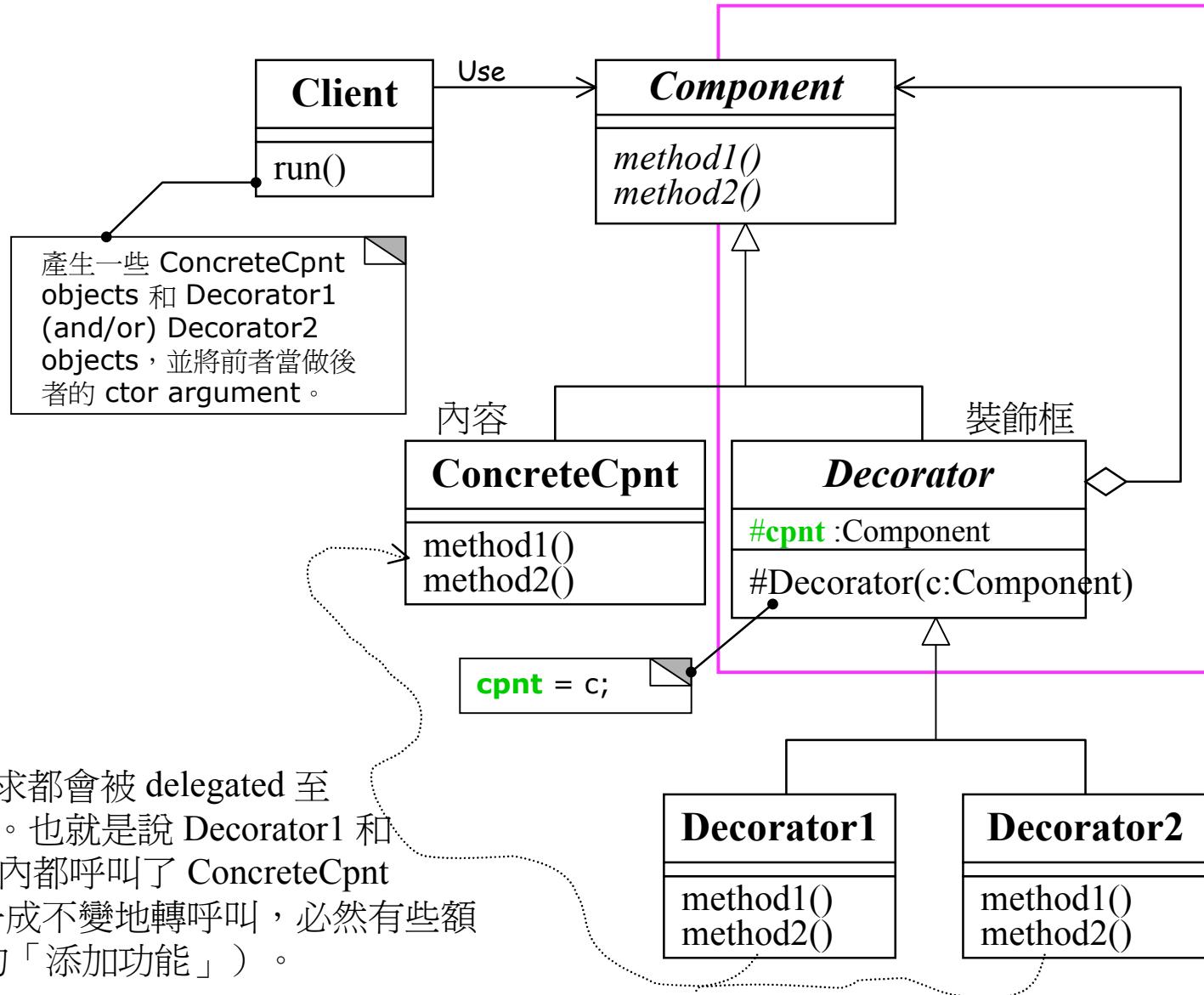
Decorator 使用多重外框的目的是為了添加功能。



8. Decorator in DP-in-Java

Decorator (注意與 Composite 的差異)

```
Hello, world.  
-----  
Hello, world.  
-----  
*-----*  
*Hello, world.*  
*-----*  
+-----+  
| // / / / / |  
| = = = = = |  
| * 您好。 * |  
| = = = = = |  
+-----+
```





8. Decorator in java.io

在Java I/O 中，我們很少使用單一 class 來產生 stream 物件，較多的是透過「疊合多層物件」的形式來獲得想要的功能。疊合多層物件（所謂 layered objects）藉以動態而透通地將工作職責賦予個別物件，稱為 **Decorator**。

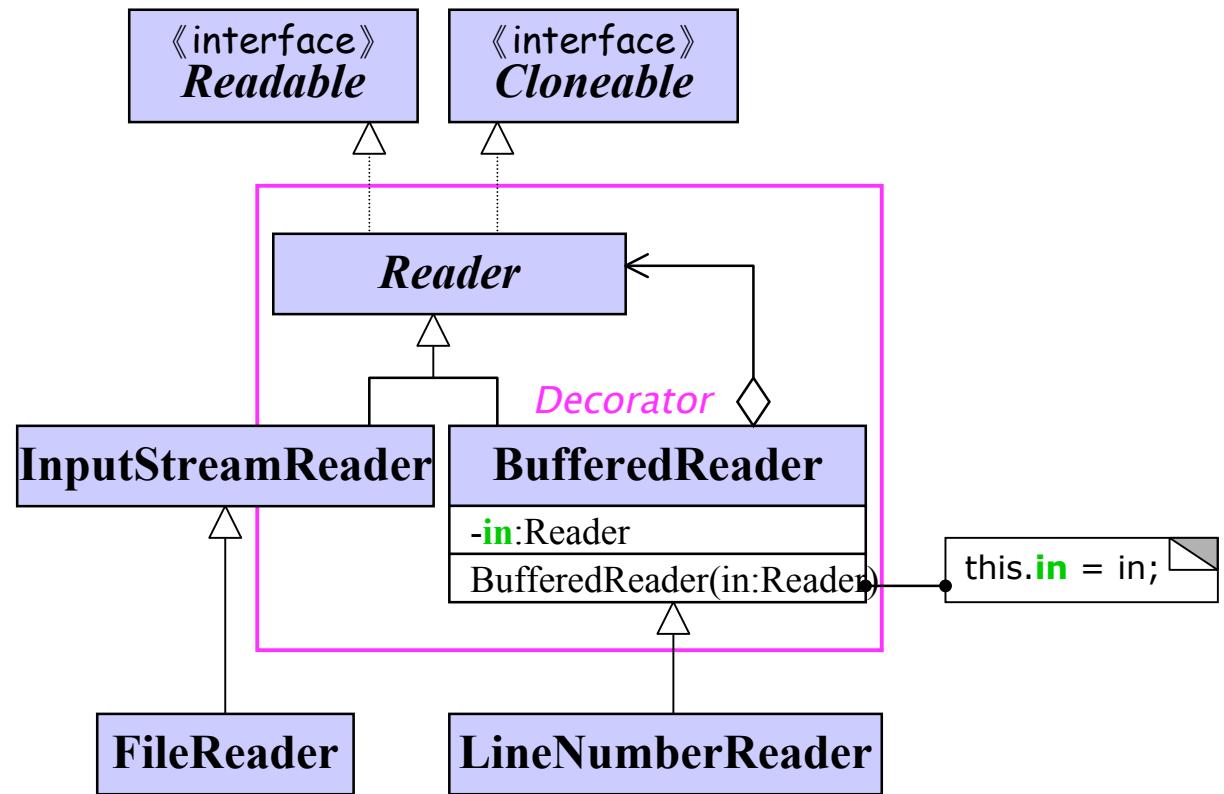
```
public class Worm implements Serializable {
    Worm w = new Worm(6, 'a');
    ObjectOutputStream out =
        new ObjectOutputStream(
            new FileOutputStream("worm.out"));
    out.writeObject("Worm storage");
    out.writeObject(w);
    out.close(); // Also flushes output
    ObjectInputStream in =
        new ObjectInputStream(
            new FileInputStream("worm.out"));
    String s = (String)in.readObject();
    Worm w2 = (Worm)in.readObject();
```

```
// 1. Reading input by lines:
BufferedReader in =
    new BufferedReader(
        new FileReader("IODEmo.java"));
String s, s2 = new String();
while((s = in.readLine()) != null)
    s2 += s + "\n";
in.close();
```

```
// 1b. Reading standard input:
BufferedReader stdin =
    new BufferedReader(
        new InputStreamReader(System.in));
System.out.print("Enter a line:");
System.out.println(stdin.readLine());
```



8. Decorator in java.io

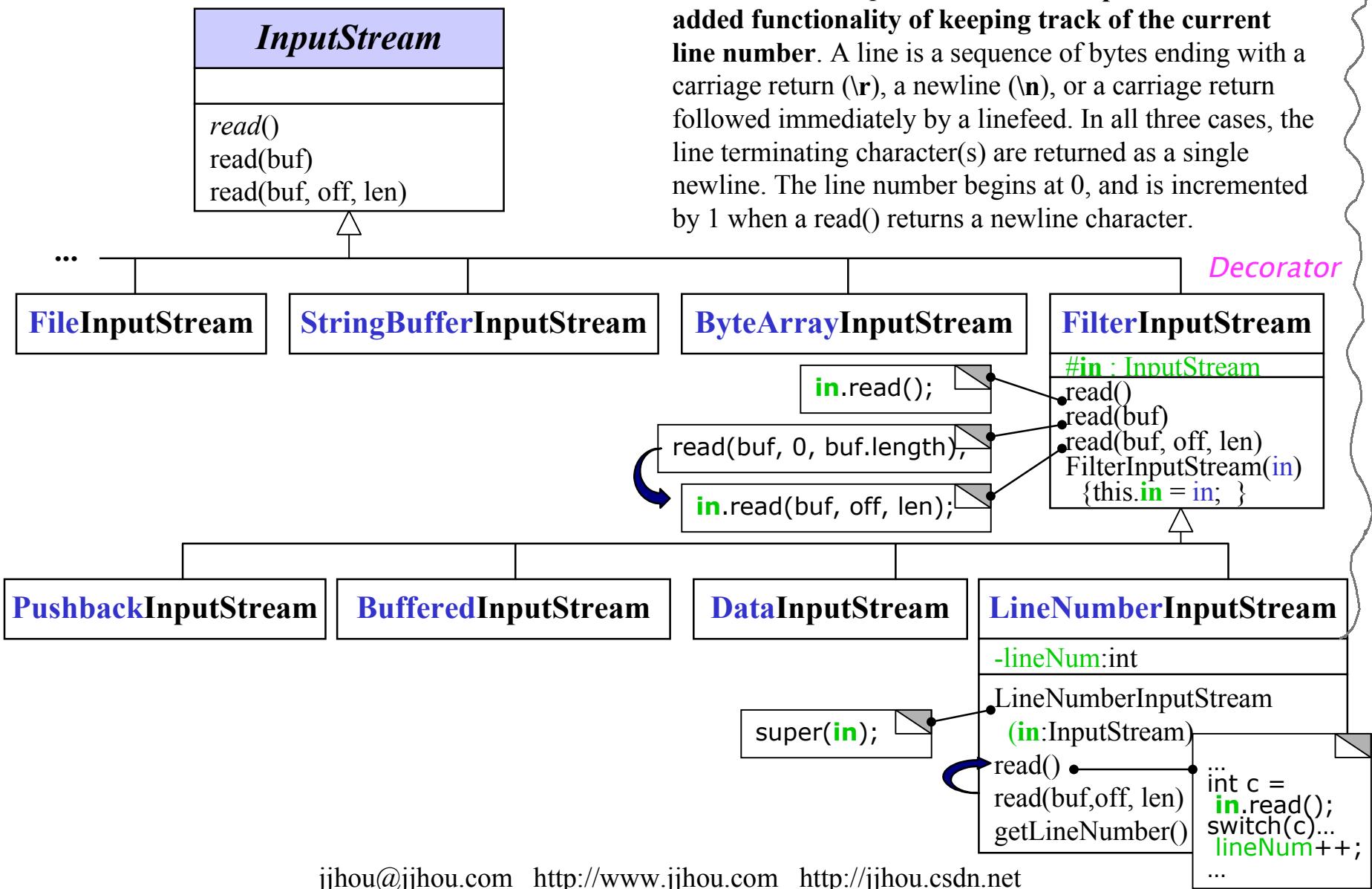


```
BufferedReader in =  
new BufferedReader(  
    new FileReader("IODEmo.java"));
```

you.com http://jjhou.csdn.net



8. Decorator in java.io





16. Observer

16. Observer (293)

Define a **one-to-many dependency** between objects so that when one object changes state, all its dependents are **notified and updated automatically**.

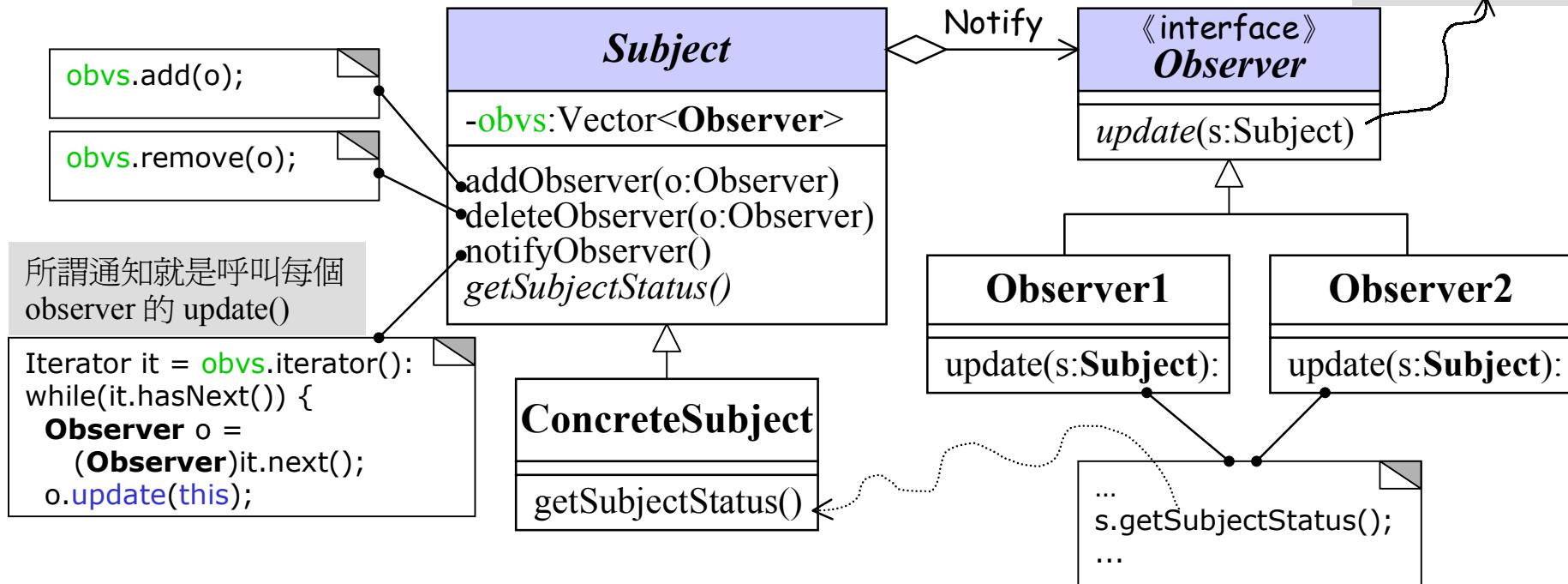
在 objects 之間定義 "一對多" 依存性，使得當 object 改變狀態時，它所依存的所有 objects 都會獲得通知並自動更新。

Observer 是被動地被通知，而不是主動觀察，所以這個 pattern 的另一個名稱 **publish-subscribe** 比較更合適些。



16. Observer in DP-in-Java

這個介面可以
視需求而調整



程式中以 `addObserver()` 加入觀察者到 **Subject** 身上。**Subject** 以 `notifyObserver()` 通知所有註冊在案的觀察者，而其實也就是呼叫 **Observer** 的 `update()` 並將 **Subject** 自己傳過去。於是 **Observer** 獲得 **Subject**，因此得以呼叫其 `getSubjectStatus()` 取得必要資訊。

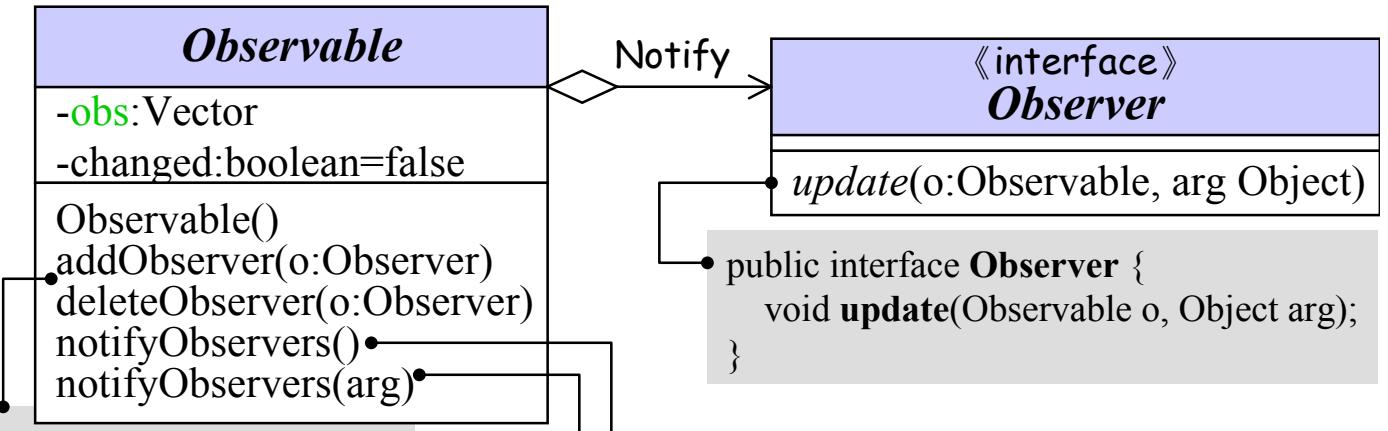
Observer 的註冊次序（也就是將來 `update()`s 的被呼叫次序）不應影響程式結果。“只要各個 **Observers** 保持足夠的獨立性應該可以做到這一點。

如果 **Subject** 狀態有變 --> 通知 **Observer** --> **Observer** 呼叫 **Subject's method** --> 造成 **Subject** 又發生狀態變化 --> 又通知 **Observer** ...形成遞迴就不宜。只要在 **Observer** 添加一個 flag 表示「目前是否有 **Subject** 的通知」即可解決。



16. Observer in Java Library

缺點：app. 的 ConcreteSubject 必須繼承 Java's **Observable**，而由於Java classes 都是單一繼承，因此ConcreteSubject 將無法再繼承其他 class。再者由於 Observable 不是個 interface，所以我們無法建立自己的 implement.



```

public class Observable {
    private boolean changed = false;
    private Vector obs;

    public Observable() {obs = new Vector(); }
    public synchronized void addObserver(Observer o) {
        if (!obs.contains(o)) obs.addElement(o);
    }
    public synchronized void deleteObserver(Observer o) {
        obs.removeElement(o);
    }
    ...
}
  
```

又是直接存取field，
又是透過accessor存取，
不一致。何不改為：
if(!hasChanged()) return;

//www.jj

```

① public void notifyObservers() {
    notifyObservers(null);
}

② public void notifyObservers(Object arg) {
    Object[] arrLocal;
    synchronized (this) {
        if (!changed) return;
        arrLocal = obs.toArray();
        clearChanged();
    }
    for (int i = arrLocal.length-1; i>=0; i--) {
        ((Observer)arrLocal[i]).update(this, arg);
    }
    // ...另有 setChanged(), clearChanged(),
    //      hasChanged(), countObserver()
}
  
```

(1) 法用 pull 法
(2) 法用 push 法

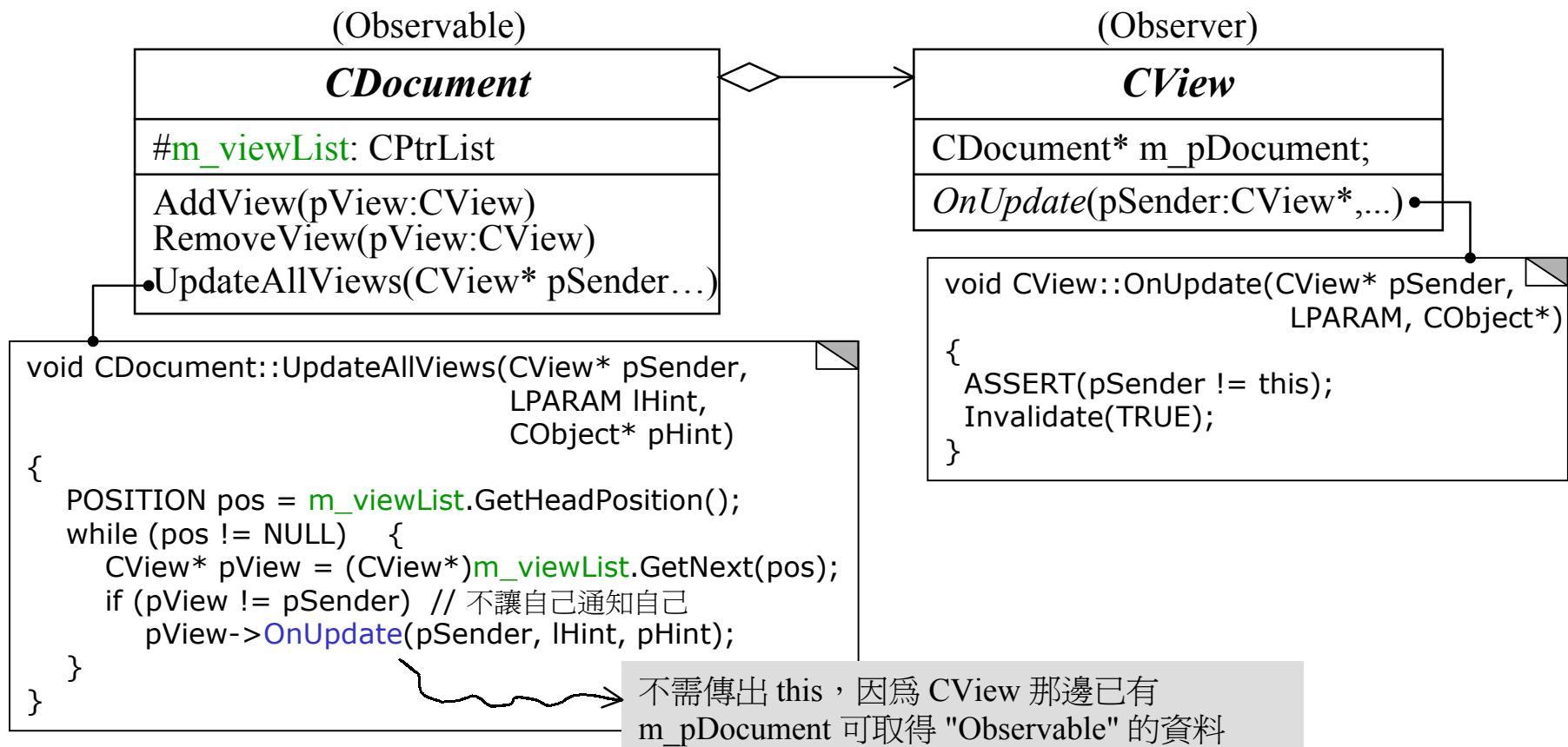


16. Observer

MVC中的 Model 和 View（例如 MFC 的 document/view）與「Subject 和 Observer」的關係相呼應。例如應用端這麼寫：

```
void CMyView::OnLButtonUp(...) {  
    ...  
    pDoc->UpdateAllViews(this);  
}
```

滑鼠左鍵被放開時決定更新所有 views





16. Observer vs. Listener

```
return (WindowListener)addInternal(a,b);
```

```
return new AWTEventMulticaster(a,b);
```

Window 可接受多種listener，其中以 `windowListener` 記錄 WindowListener(s)。每當要新增 WindowListener 就產生一個 AWTEventMulticaster (內有 a,b) 並轉型為 WindowListener 再記錄至 `windowListener` 中。AWTEventMulticaster 可被轉型為 EventListener 而後被轉型為各種 Listener，這正是其命名所由。整個 listener list 被記錄為此型式：(((a+b)+b)+b)+b...

AWTEventMulticaster

-`a,b:EventListener`

```
AWTEventMulticaster(a1,b1) {a=a1; b=b1;}
```

- add(a:WindowListener, b:WindowListener)
- addInternal(a:EventListener, b:EventListener)

: `EventListener` (注意竟可如此!!)

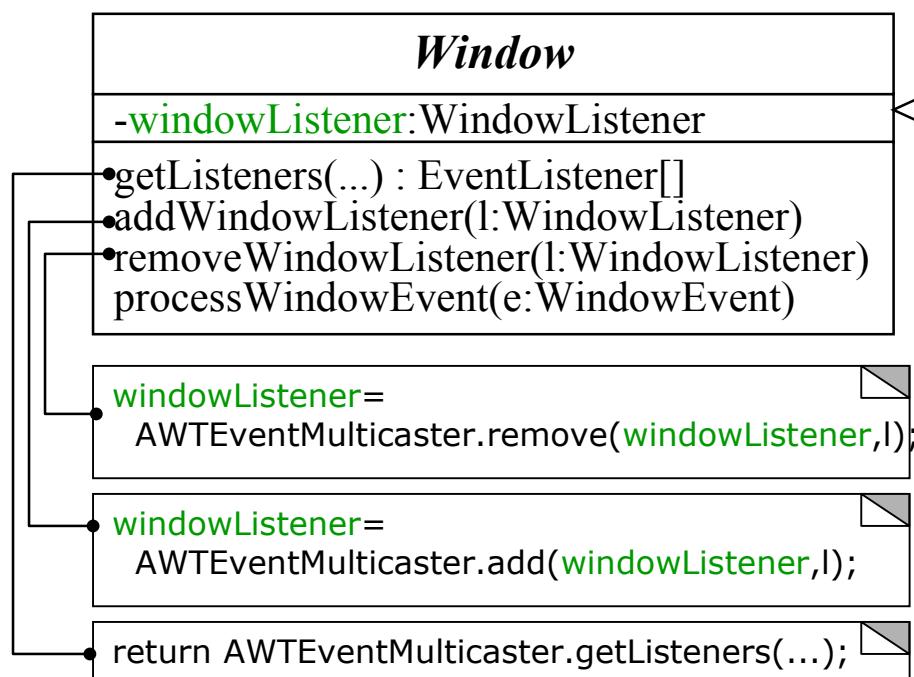
《interface》 *EventListener*

《interface》 *WindowListener*

```
windowOpened(e:WindowEvent)  
windowClosed(e:WindowEvent)  
windowClosing(e:WindowEvent)
```

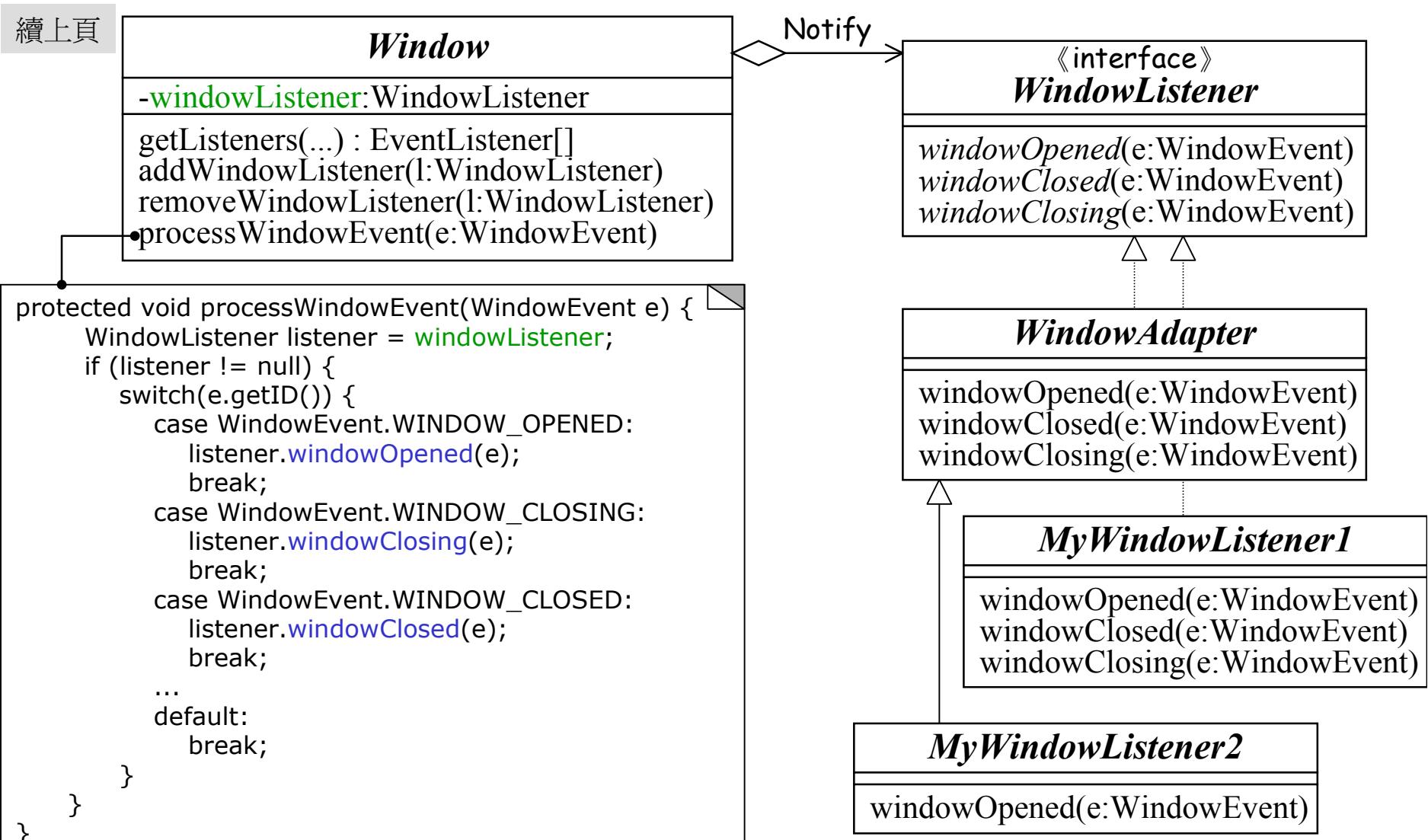
WindowAdapter

```
windowOpened(e:WindowEvent)  
windowClosed(e:WindowEvent)  
windowClosing(e:WindowEvent)
```





16. Observer vs. Listener





16. Observer vs. Listener

...src\java\awt\Window.java

```
public class Window extends Container implements Accessible {  
    public synchronized void addWindowListener(WindowListener l) {...};  
    public synchronized void removeWindowListener(WindowListener l) {...};  
}  
|    public void windowOpened(WindowEvent e);  
|    public void windowClosing(WindowEvent e);  
|    public void windowClosed(WindowEvent e);  
|    public void windowIconified(WindowEvent e);  
|    public void windowDeiconified(WindowEvent e);  
|    public void windowActivated(WindowEvent e);  
|    public void windowDeactivated(WindowEvent e);
```

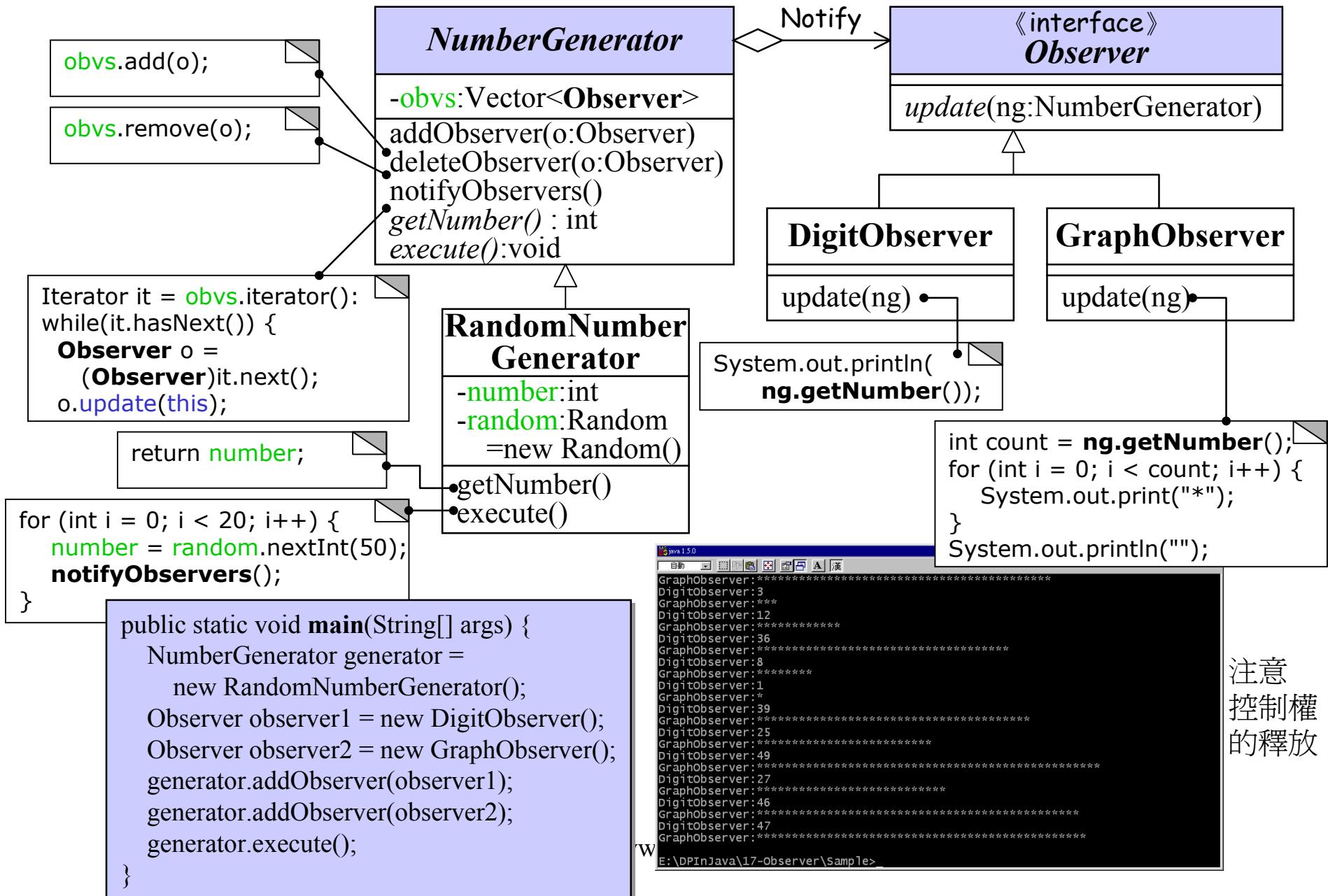
...src\java\awt\event\WindowAdapter.java

```
public abstract class WindowAdapter  
    implements WindowListener, WindowStateListener, WindowFocusListener {  
    public void windowOpened(WindowEvent e) {}  
    public void windowClosing(WindowEvent e) {}  
    public void windowClosed(WindowEvent e) {}  
    public void windowIconified(WindowEvent e) {}  
    public void windowDeiconified(WindowEvent e) {}  
    public void windowActivated(WindowEvent e) {}  
    public void windowDeactivated(WindowEvent e) {}  
    public void windowStateChanged(WindowEvent e) {}  
    public void windowGainedFocus(WindowEvent e) {}  
    public void windowLostFocus(WindowEvent e) {}  
}
```

...src\java\awt\event\WindowListener.java



16. Observer 習題

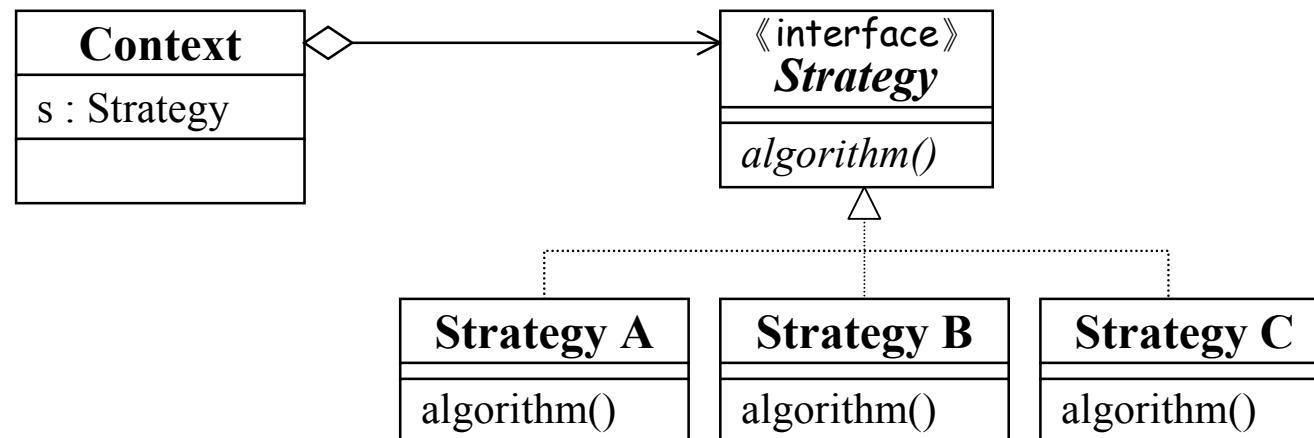




21. Strategy (Policy) in GOF

Define a family of algorithms, encapsulate each one, and make them interchangeable.
Strategy lets the algorithm vary independently from clients that use it.

定義一整個族系的演算法，將每一個封裝起來（成爲 class）並使得以被交換使用。
客戶端可抽換其中任何一個演算法而不受影響。





21. Strategy in DP-in-Java

一般 programming 時習慣把演算法的實作結合至 method 內。但 **Strategy** 故意把演算法與其他部分分開（jjhou：成爲各自獨立的 classes），再以 delegation 的方式來使用演算法。

這看起來使程式變得複雜，但其實不盡然。因爲 delegation 是一種「說分就分」的關係，所以很容易切換演算法，並可動態（runtime）切換。例如於執行期檢查 memory 多寡，多則使用某種演算法，少則使用另一種演算法；一切都在 runtime 進行。



21. Strategy in Agile-PPP

90年代初期—也就是OO發展的早期—我們都深深受繼承的觀念所吸引。有了繼承，我們就可以運用差異編程（program by difference）來設計程式！也就是說給予某個 class，它所完成的事大部分都是我們用得著的，那麼我們就可以建立一個 subclass，並只改變我們不想要的一小部分。只要繼承 class 就可以再利用程式碼！

到了 1995 年，繼承非常遭到濫用，而且代價昂貴。Gamma, Helm, Johnson, Vlissides 甚至強調「多使用 object composition 而少使用 class inheritance」。應該可以減少 inheritance 的使用，經常以 composition 或 delegation 來取代。

有兩個 patterns 可以歸納 inheritance 和 delegation 之間的差異：**Template Method** 和 **Strategy**。兩者都可以解決從detailed context（細節化脈絡/情境）中分離出 generic algorithm（一般化演算法）的問題，這種需求在軟件設計中極為常見。是的，我們有一個用途廣泛的 algorithm，為了符合**DIP**，我們想要確保這個 generic algorithm 不要倚賴 detailed implementation，而是希望 generic algorithm 和 detailed implementation 都倚賴於abstraction（抽象概念/抽象化）。

Template Method 讓一個 generic algorithm 可以操縱（manipulate）多個可能的 detailed implementations，而 **Strategy** 讓每一個 detailed implementation 都能夠被許多不同的 generic algorithms 操縱（manipulated）。

ref 《Agile Software Development, Principles, Patterns, and Practices》



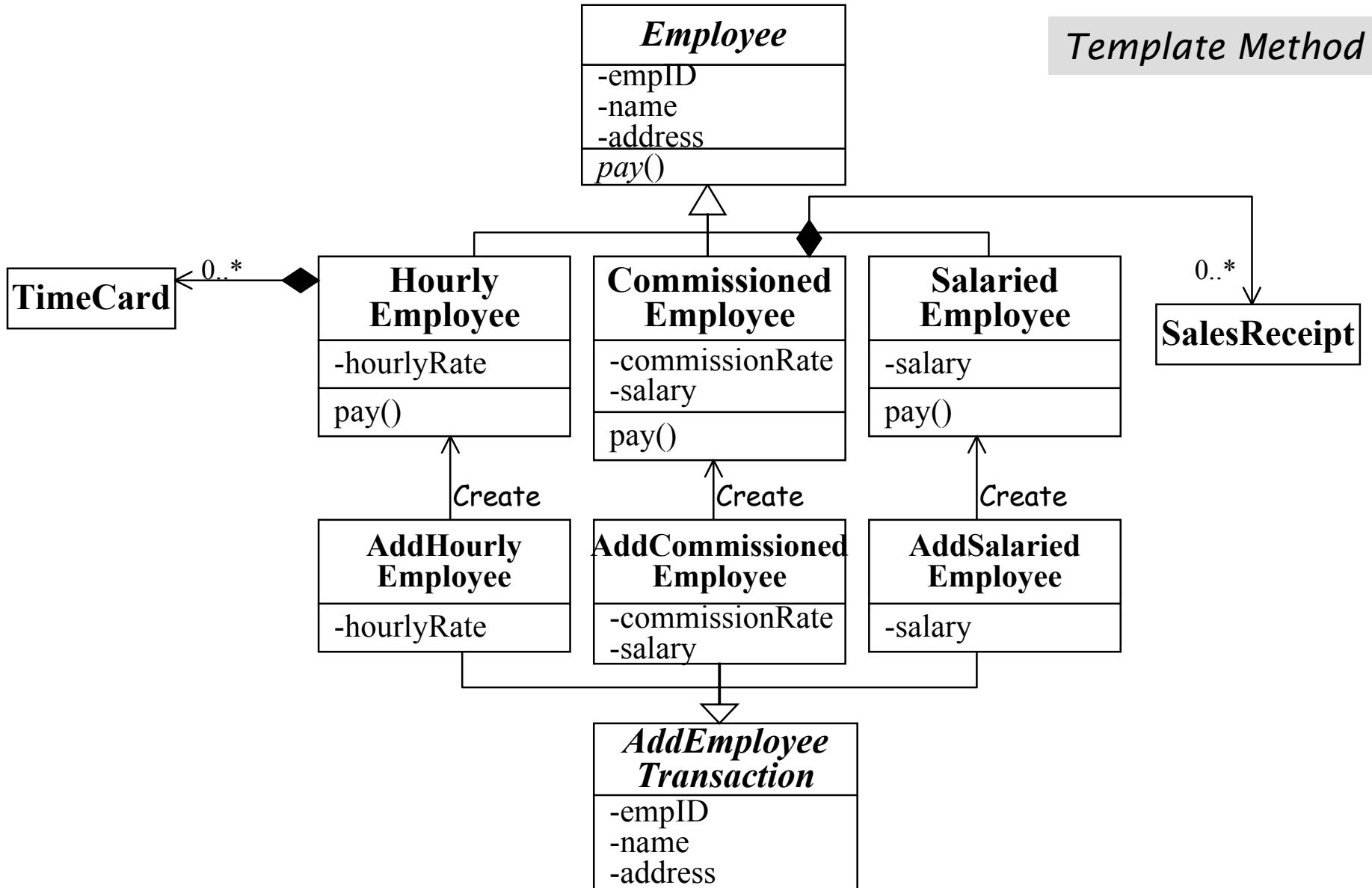
21. Strategy in Agile-PPP

在《Agile-PPP》p.199 的 Payroll 實例中，Employee 計薪方式（演算法）有三種（Salaried, Hourly, Commissioned），被封裝為各自獨立的 classes
（SalariedClassification、HourlyClassification、CommissionedClassification）。Employee 可任意抽換使用其中任何一個計薪方式（演算法）而不影響 Employee 自身結構（因為它用的是一個 PaymentClassification*，所謂的 aggregation pointer）。

Employee 不僅以 **Strategy** 解決薪資計算問題，也用以解決薪資領取方式（郵寄支票 or 親領支票 or 直接入戶）、薪資發放時間、薪資扣款（工會會費 or 特別服務費）等問題，因為任何員工都有可能在這些主題上換用演算法。

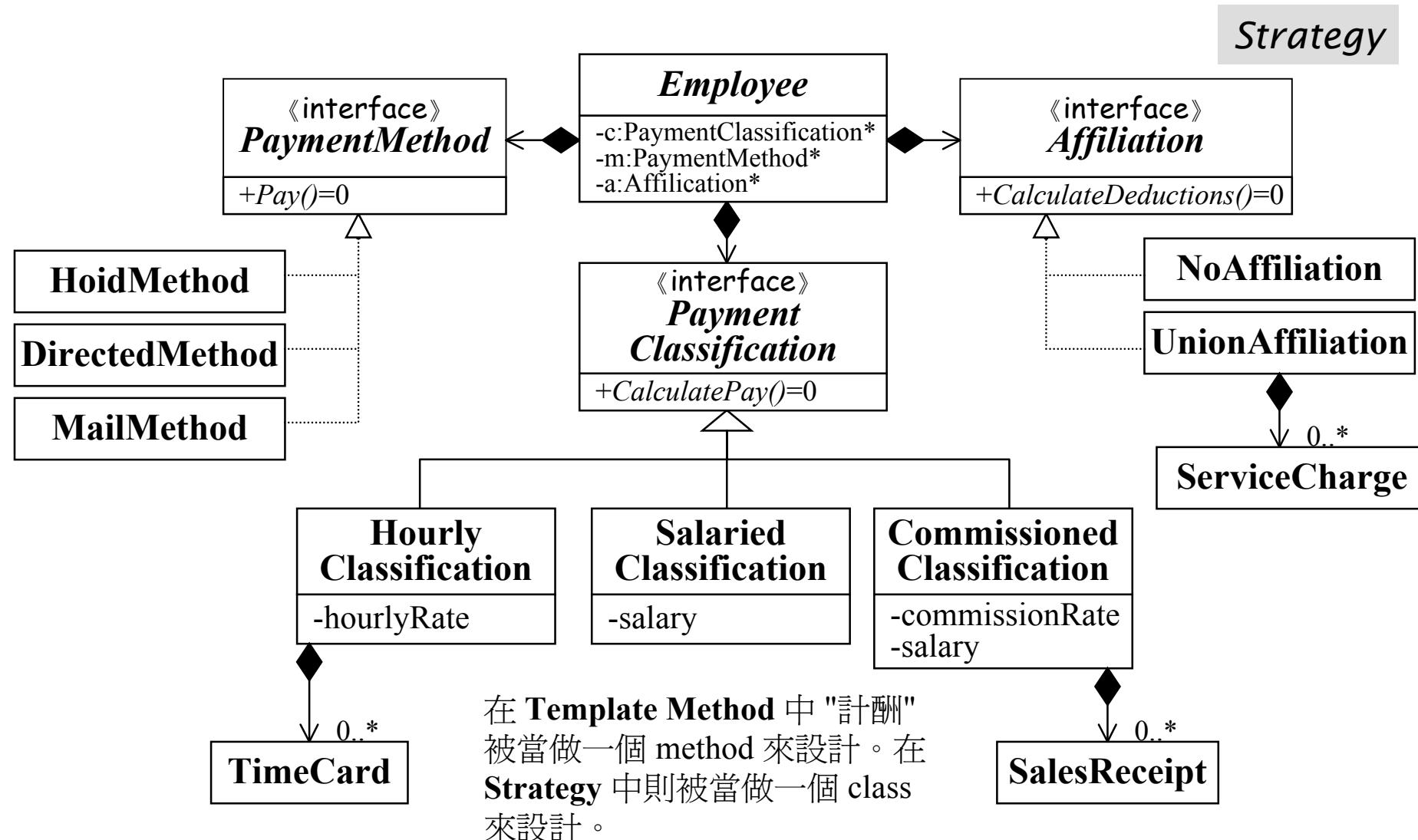


21. Strategy in Agile-PPP





21. Strategy in Agile-PPP

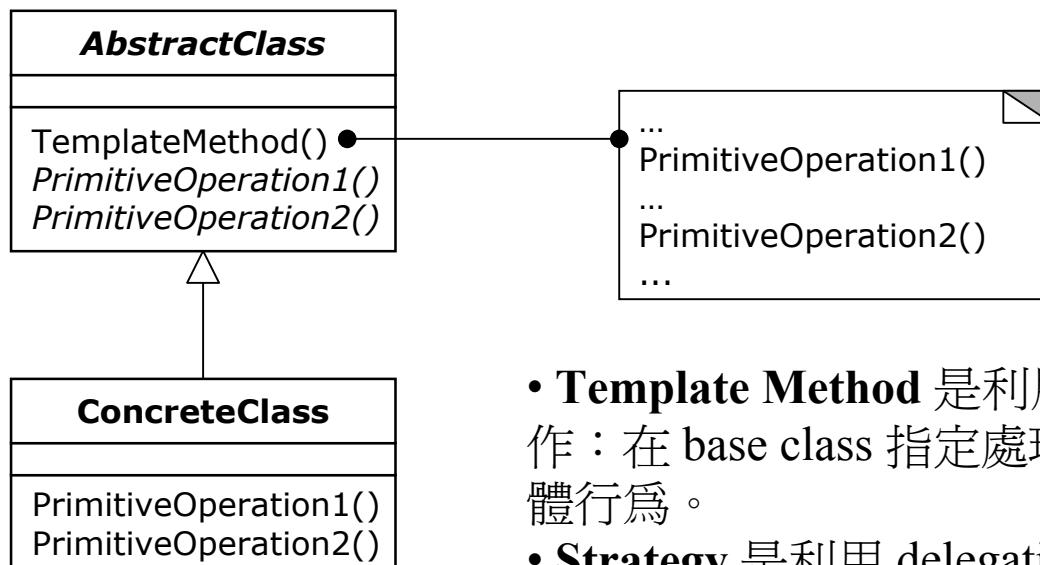




22. Template Method in GOF

Define the skeleton of an algorithm in an operation, **deferring** some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

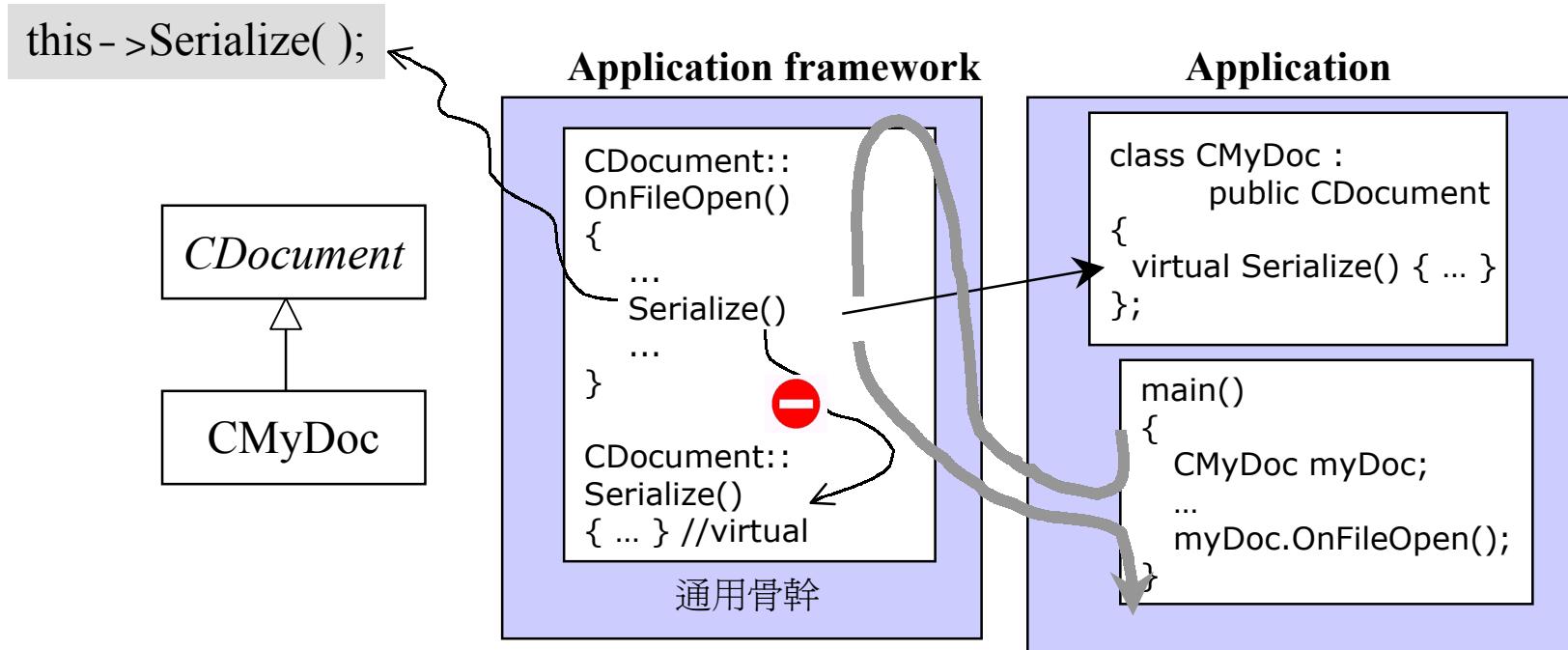
定義演算法骨幹，延緩其中某些步驟，使它們在subclasses 中才獲得真正定義。
Template Method 使 subclasses 得以重新定義演算法內的某些動作，而不需改變演算法的總體結構。



- **Template Method** 是利用 inheritance 來改變程式動作：在 base class 指定處理大綱，在 sub class 指定具體行為。
- **Strategy** 是利用 delegation 來改變程式動作：改變的不是演算法局部，而是切換整個演算法。



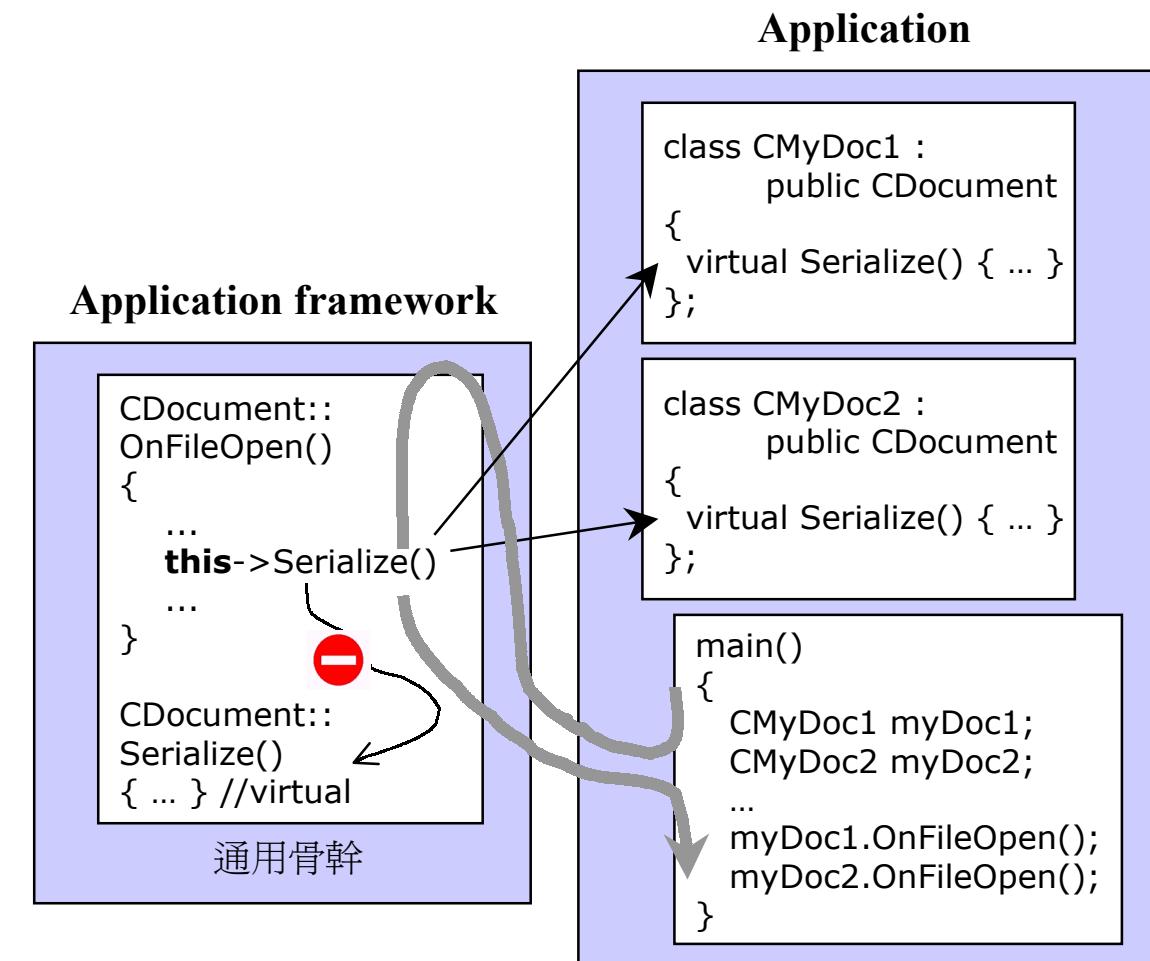
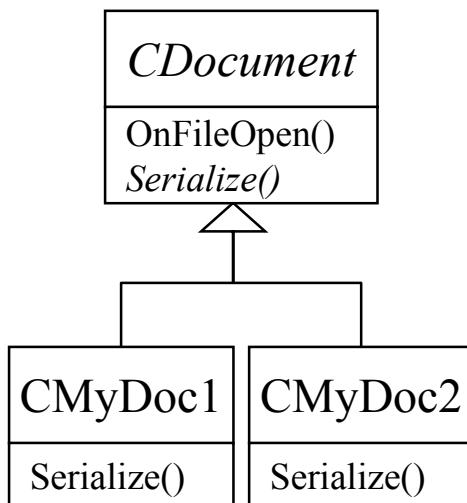
22. Template Method in MFC



《深入淺出MFC》p.84：*CDocument::OnFileOpen* 中所呼叫的 *Serialize* 是哪一個 class 的成員函式呢？如果它是一般（non-virtual）函式，毫無問題應該是 *CDocument::Serialize*。但因這是個虛擬函式，情況便有不同。既然 derived class 已經改寫了虛擬函式 *Serialize*，那麼理當喚起 derived class 之 *Serialize*函式。這種行爲模式非常頻繁地出現在 application framework 身上。



22. Template Method in MFC





22. Template Method simulation

```
01 #include <iostream>
02 using namespace std;
03
04 // 這裡扮演application framework 的角色
05 class CDocument
06 {
07 public:
08     void OnFileOpen() // 此即所謂Template Method
09     {
10         // 這是一個演算法，骨幹已完成。每個cout輸出代表一個實際應有動作
11         cout << "dialog..." << endl;
12         cout << "check file status..." << endl;
13         cout << "open file..." << endl;
14         ? Serialize(); // 喚起哪一個函式？
15         cout << "close file..." << endl;
16         cout << "update all views..." << endl;
17     }
18
19     virtual void Serialize() { }; // 沒有它行不行？
20 };
... 接下頁 ...
```

分析：如果這個class未宣告**Serialize()**，則編譯器面對呼叫動作時，並不將之編碼為 **this->Serialize()**。因此會喚起全域函式::Serialize()，但其實無此函式，所以編譯失敗。又，如果這個成員函式未被定義為**virtual**，則編譯時將被編死（靜態繫結）。



22. Template Method simulation

執行結果：

```
... 接上頁 ...
21 // 以下扮演 application 的角色
22 class CMyDoc : public CDocument
23 {
24 public:
25     virtual void Serialize()
26     {
27         // 只有應用程式員自己才知道如何讀取自己的文件檔
28         cout << "CMyDoc::Serialize()" << endl;
29     }
30 };
31 int main()
32 {
33     CMyDoc myDoc; // 假設主視窗功能表的 [File/Open] 被按下後執行至此。
34     myDoc.OnFileOpen();
35 }
```

```
dialog...
check file status...
open file...
CMyDoc::Serialize()
close file...
update all views...
```

Object Factory/Factory Method 是一種典型的 **Template Method**，只不過是應用在 object 創建工作上而已（其 return type 有比較嚴格的限制，需為各 objects 之 base class）。



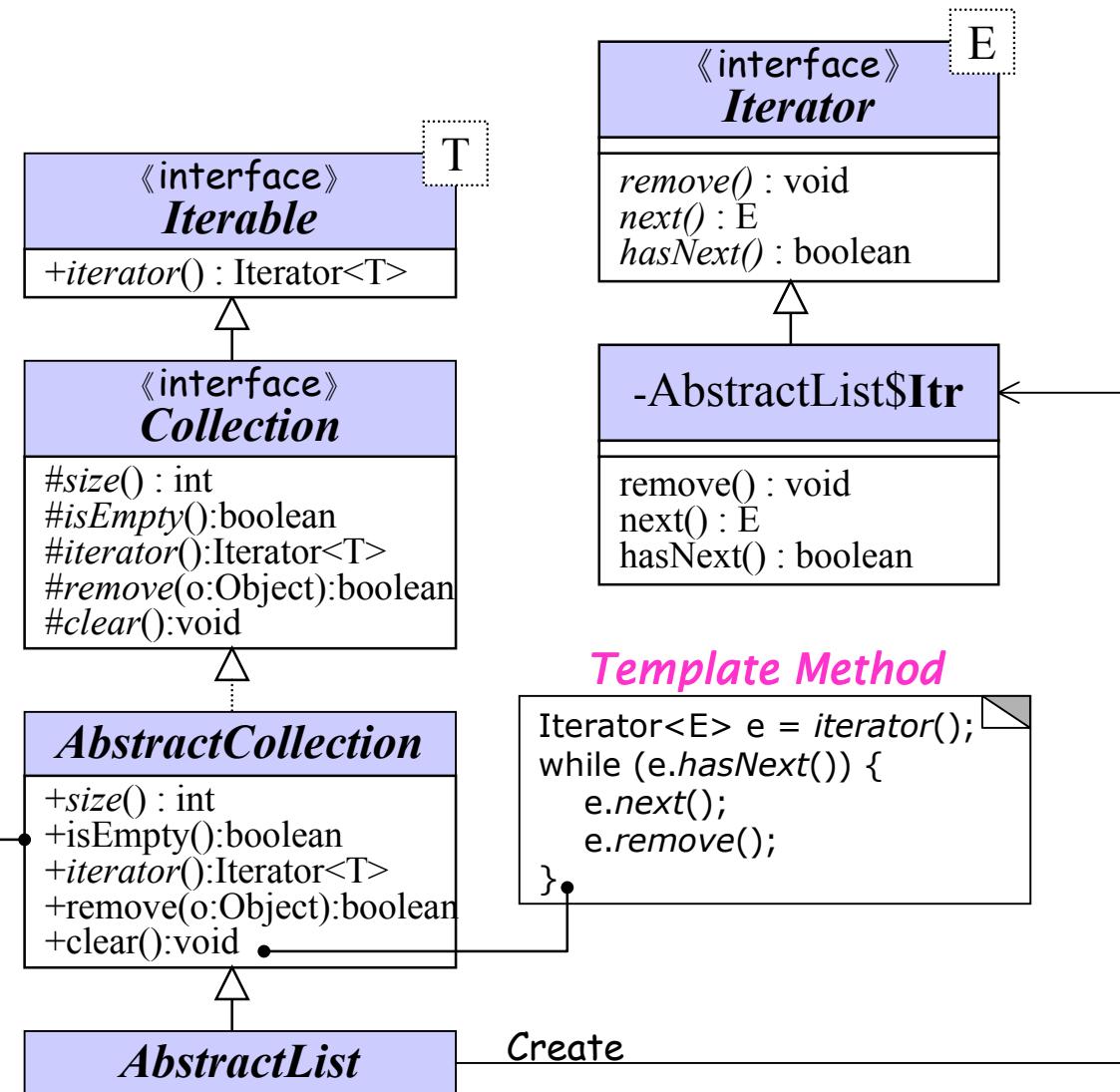
22. Template Method in Java

Template Method 是 virtual function 自然而然的應用，是建構 OO class libraries 的最基本技法。

Template Method 很常見。對 framework 來說這個 pattern 棒透了。由 framework 控制該做的事情（演算法），而由你（framework user, subclass designer）指定演算法中的（某些）步驟細節。

Template Method

```
return size() == 0;
```





22. Template Method via NVI idiom

```
class GameCharacter {  
public:  
    virtual int healthValue() const; //傳回人物的健康指數；  
    ... //derived classes可重新定義它。  
};
```

一個有趣的流派主張，virtual函式應該幾乎總是private。這個流派的擁護者建議，較好的設計是保留healthValue()為public成員函式，但讓它成為non-virtual，並令它呼叫一個 private virtual函式（例如doHealthValue()）進行實際工作。令客戶「透過 public non-virtual成員函式間接呼叫private virtual函式」稱為 *non-virtual interface* (NVI) 手法，是 **Template Method** 的一個獨特表現形式。Scott Meyers 在《Effective C++》3/e 中把這個non-virtual函式稱為virtual函式的外覆器（wrapper）。

```
class GameCharacter {  
public:  
    int healthValue() const  
    {  
        ...  
        int retVal = doHealthValue();  
        ...  
        return retVal;  
    }  
private:  
    virtual int doHealthValue() const  
    {  
        ...  
    }  
};
```

//derived classes不重新定義它，
//做一些事前工作。
//做真正的工作。
//做一些事後工作。

//derived classes可重新定義它。
//預設演算法，計算健康指數。



22. Template Method via NVI idiom

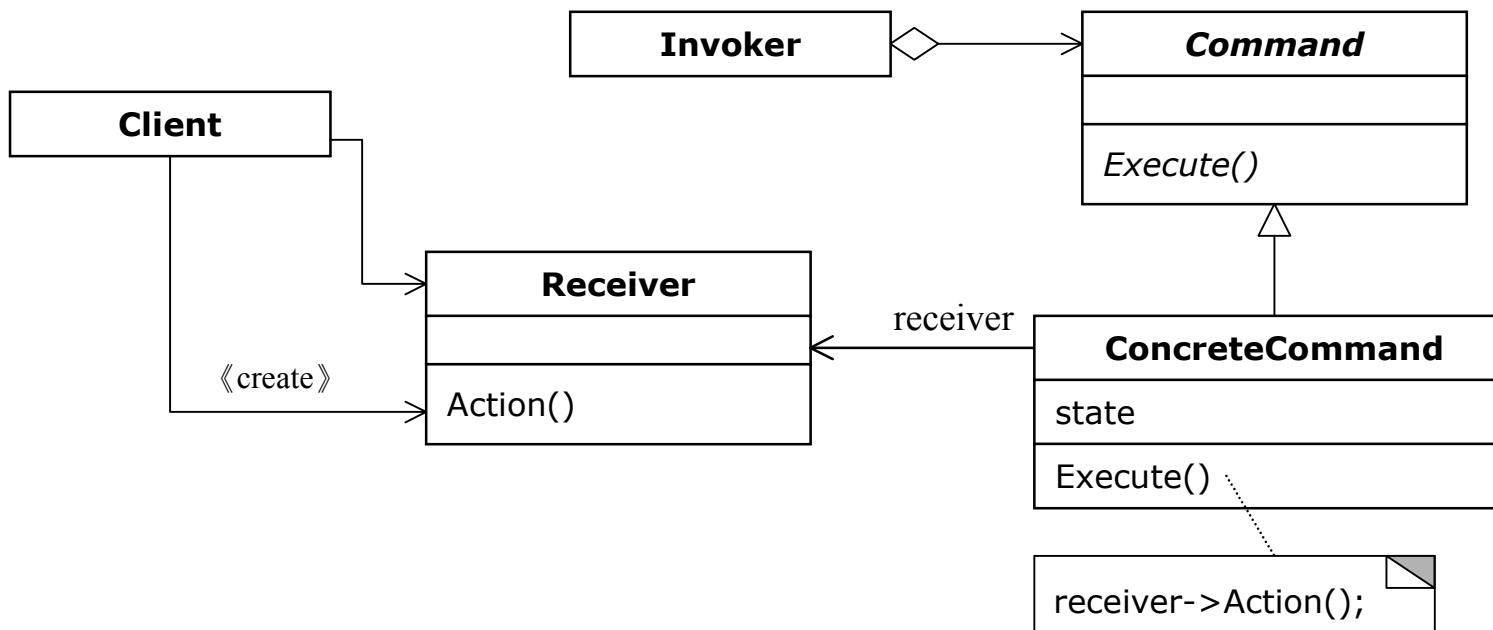
NVI手法的一個優點隱身在程式碼註釋「做一些事前工作」和「做一些事後工作」之中。那些註釋用來告訴你當時的程式碼保證在「virtual函式進行真正工作之前和之後」被呼叫。這意味外覆器（wrapper）確保得以在一個virtual函式被呼叫之前設定好適當場景，並在呼叫結束之後清理場景。「事前工作」可以包括鎖定互斥器（locking a mutex）、製造運轉日誌記錄項（log entry）、驗證class約束條件、驗證函式先決條件等等。「事後工作」可以包括互斥器解除鎖定（unlocking a mutex）、驗證函式的事後條件、再次驗證class約束條件等等。如果你讓客戶直接呼叫virtual函式，就沒有任何好辦法可以做這些事。



6. Command in GOF

Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support **undoable** operations.

將 request 封裝為 object，讓你得以不同的 requests 將 client 參數化，或是將 requests 放進隊列中或誌記起來；並支援 **undo** 操作。





6. Command in STL

```
template <class InputIterator, class Function>
Function for_each(InputIterator first,
                  InputIterator last,
                  Function f) {
    for ( ; first != last; ++first)
        f(*first);
    return f;
}
```

「收集」某個處理動作所需環境（environment）的時刻，和「執行」該處理動作的時刻並不相同。在兩個時刻之間，程式將該處理請求當作一個物件來保存和傳遞。如果沒有這種時序（timing）上的需要，就不會有 **Command** 範式的存在。從這個角度看，**Command**物件的存在歸因於時序問題：由於你需要延後處理，所以你得有個物件將請求保存至那個時刻。

在 STL 中，許多操作（可執行體，executable entity）的目的是為了和 STL Algorithm 配合。它的執行情境（such as 函式參數）必須在執行的那一刻才能確定（通常都是在 STL algorithm 巡訪區間內各元素時以元素做為參數），因此需要 Command 。



6. Command Binder in STL

假設有個 Functor 取兩個整數作為參數，你想將其中一個整數繫結（綁定）為某固定值，只讓另一個可變化。所謂 Binder 者會產出一個「只取單一整數」的 Functor
“ “ 因為另一個整數參數是固定的，因而也是可知的。

Binder 是一項威力強大的功能。你不但可以保存「可呼叫體（callable entities）」，還可以保存它們的部份（或全部）引數。這大大提高了 Functor 的表達能力，因為它可以讓你包裝函式和引數，無需添加做為「黏膠」之用的程式碼。

Functor 所保存的只是「計算」，並沒有保存和「計算」相關的任何環境信息。
Binder 可以讓 Functor 將部份環境連同計算一起保存下來，並逐步降低呼叫時刻所需的環境信息。

STL 的 **function adapter** 就是 command binder 的一種實現。



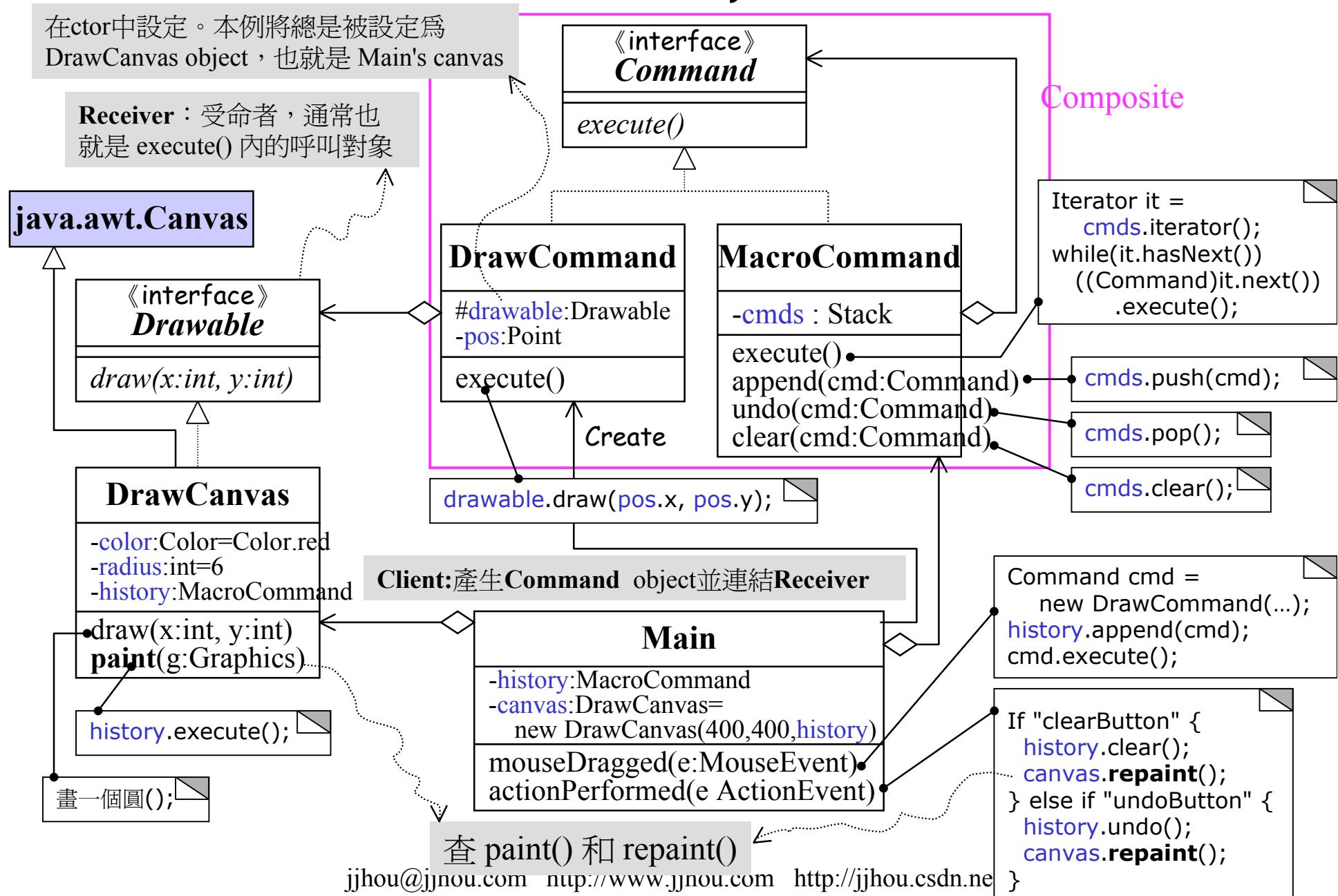
6. Command in DP-in-Java



"命令" 該含哪些信息，因情況而異。本例只有畫點（滑鼠點）位置，沒有大小、色彩、形狀...等信息。如果加上 time stamp，甚至可記錄（並於將來重現）滑鼠移動快慢情況。

如果不使用 Command，一樣可以記錄以上所有資訊。試考慮 MFC Scribble example，在 Stroke 中設計各種 fields 用以記錄資訊即可。並把所有 data 也放進 Stack 中（那麼似乎也可以做到 undo?）

6. Command in DP-in-Java





6. Command in DP-in-Java

Canvas's repaint() :

This method **causes a call to this component's update()** as soon as possible.

(這個 method 繼承自 `java.awt.Component`)

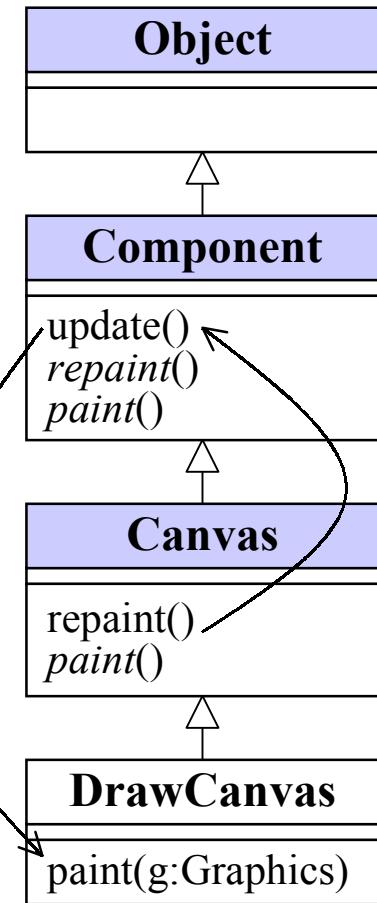
Component's update() :

Updates this component. The AWT calls the update() in response to a call to repaint(), update() or paint(). You can assume that the background is not cleared.

The update() method of Component does the following:

- Clears this component by filling it with the background color.
- Sets the color of the graphics context to be the foreground color of this component.
- Calls this component's **paint()** method to completely redraw this component.

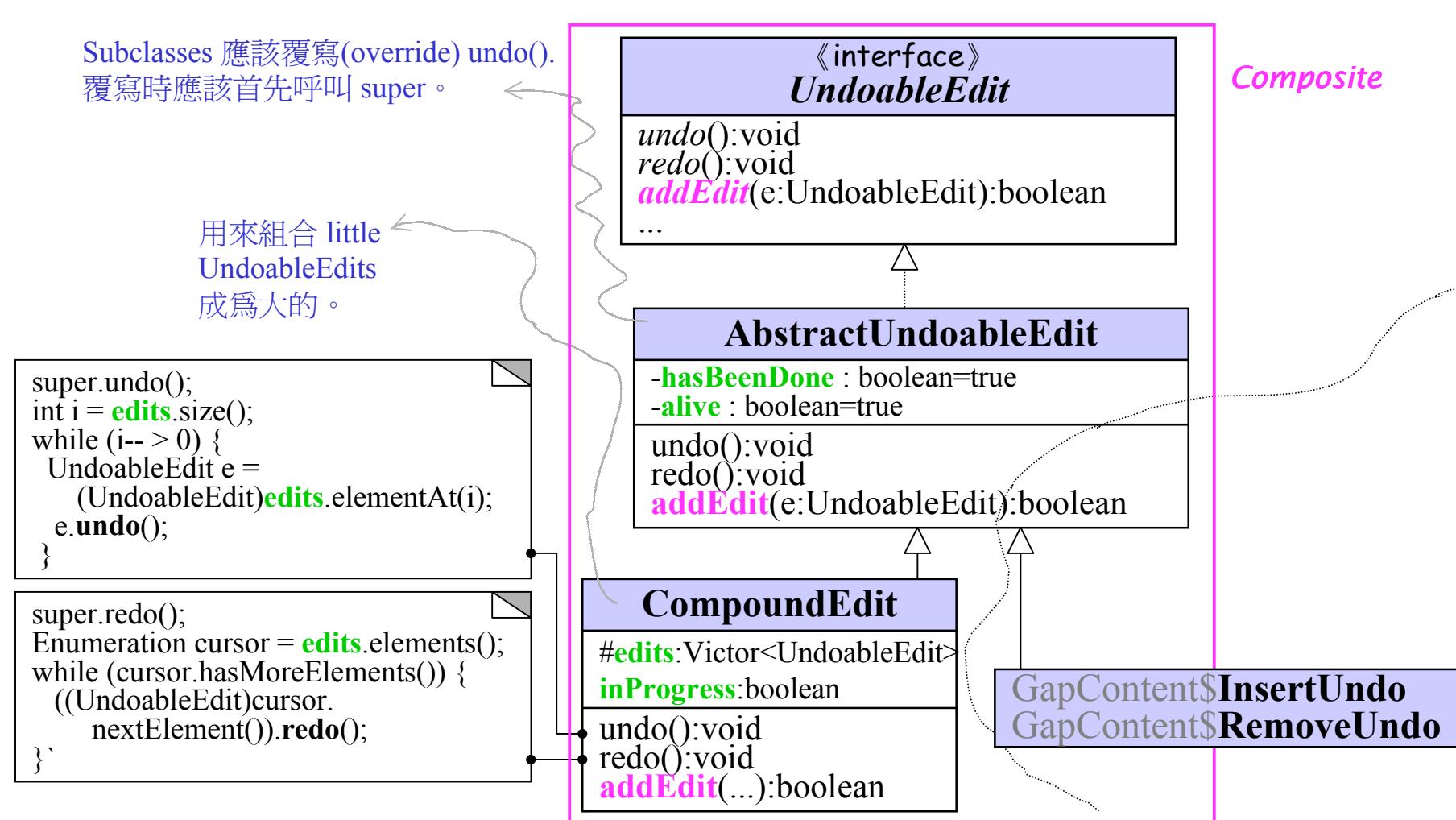
Component's paint() : Paints this component. This method **is called when** the contents of the component should be painted in response to the component first being shown or damage needing repair. **The clip rectangle in the Graphics parameter** will be set to the area which needs to be painted. For performance reasons, Components with zero width or height aren't considered to need painting when they are first shown, and also aren't considered to need repair.



Undoable (for TextArea)

in Java Library (.../src/javax/swing/undo)

<http://www.java2s.com/ExampleCode/Swing-JFC/Undoredotextarea.htm>



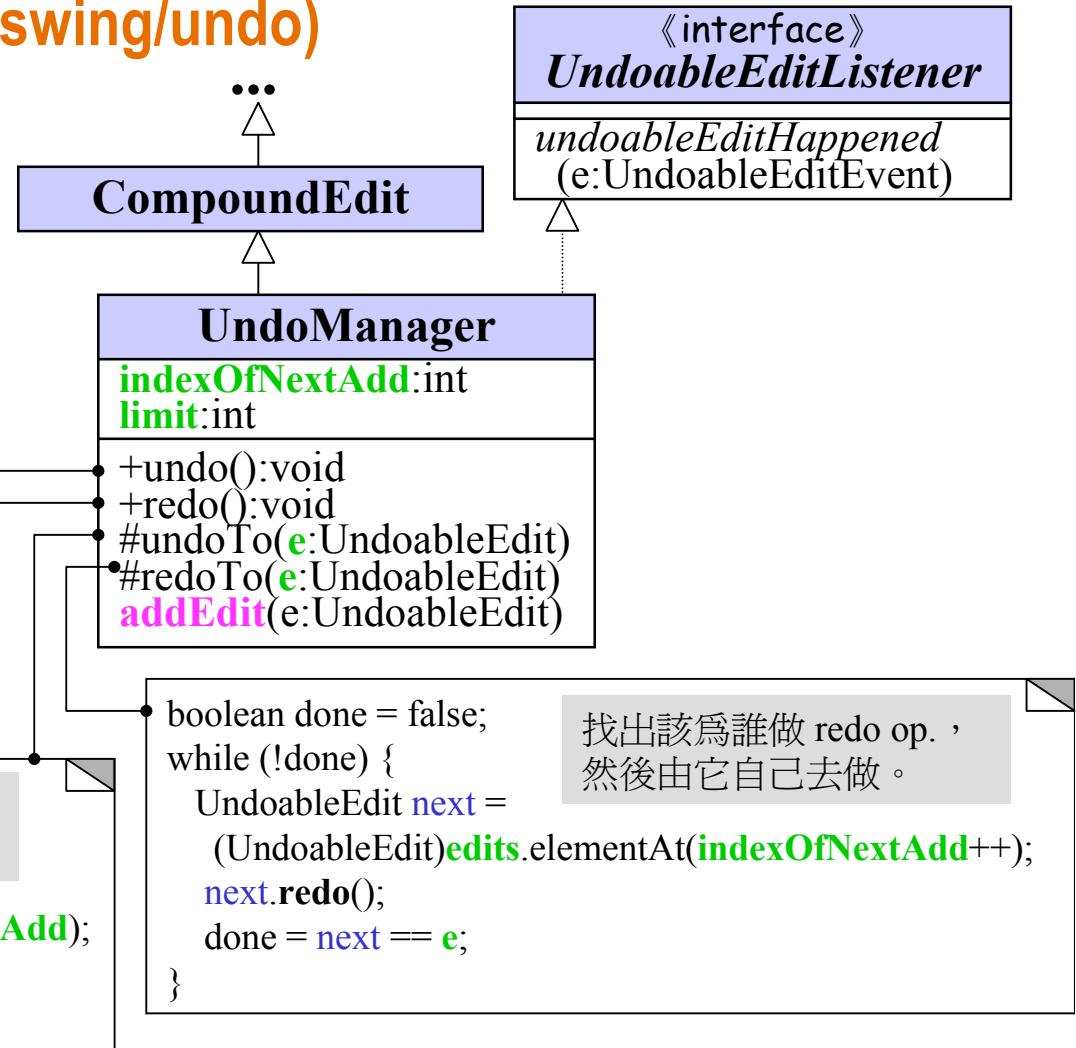
Undoable (for TextArea)

in Java Library (.../src/javax/swing/undo)

```
if (inProgress) {
    UndoableEdit edit = editToBeUndone();
    if (edit == null) throw ...;
    undoTo(edit);
} else {
    super.undo();
}
```

```
if (inProgress) {
    UndoableEdit edit = editToBeRedone();
    if (edit == null) throw ...;
    redoTo(edit);
} else {
    super.redo();
}
```

```
boolean done = false;
while (!done) {
    UndoableEdit next =
        (UndoableEdit) edits.elementAt(--indexOfNextAdd);
    next.undo();
    done = next == e;
}
```



Undoable (for TextArea) 運行剖析, 1



javax.swing.undo.UndoManager@6c585a hasEdit
inProgress: true edits:

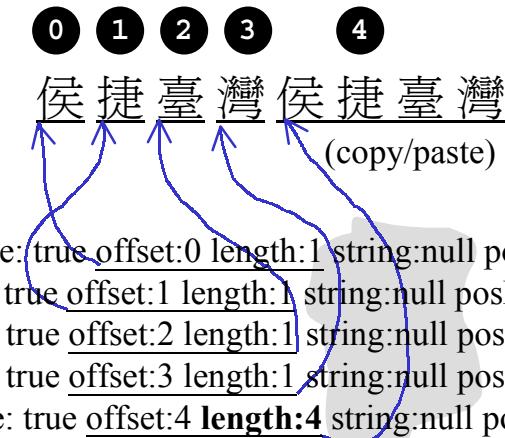
- ① [[javax.swing.text.GapContent\$InsertUndo@11ca803 hasBeenDone: true alive: true offset:0 length:1 string:null posRefs:null],
 - ② [[javax.swing.text.GapContent\$InsertUndo@5a67c9 hasBeenDone: true alive: true offset:1 length:1 string:null posRefs:null],
 - ③ [[javax.swing.text.GapContent\$InsertUndo@766a24 hasBeenDone: true alive: true offset:2 length:1 string:null posRefs:null],
 - ④ [[javax.swing.text.GapContent\$InsertUndo@32784a hasBeenDone: true alive: true offset:3 length:1 string:null posRefs:null],
 - ⑤ [[javax.swing.text.GapContent\$InsertUndo@15c07d8 hasBeenDone: true alive: true offset:4 length:4 string:null posRefs:null]]]
- limit: 100 indexOfNextAdd: 5

按一次 [Undo] :

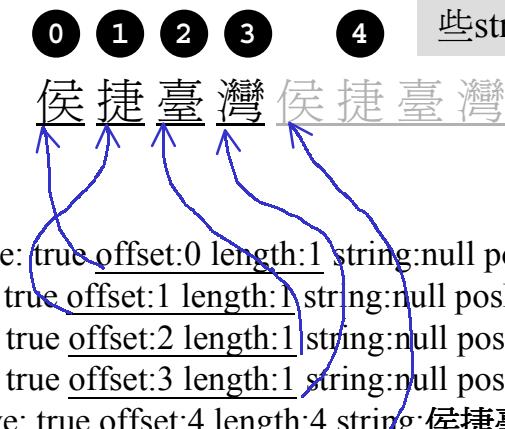


javax.swing.undo.UndoManager@6c585a hasEdit
inProgress: true edits:

- ① [[javax.swing.text.GapContent\$InsertUndo@11ca803 hasBeenDone: true alive: true offset:0 length:1 string:null posRefs:null],
 - ② [[javax.swing.text.GapContent\$InsertUndo@5a67c9 hasBeenDone: true alive: true offset:1 length:1 string:null posRefs:null],
 - ③ [[javax.swing.text.GapContent\$InsertUndo@766a24 hasBeenDone: true alive: true offset:2 length:1 string:null posRefs:null],
 - ④ [[javax.swing.text.GapContent\$InsertUndo@32784a hasBeenDone: true alive: true offset:3 length:1 string:null posRefs:null],
 - ⑤ [[javax.swing.text.GapContent\$InsertUndo@15c07d8 hasBeenDone: false alive: true offset:4 length:4 string:侯捷臺灣
posRefs:[javax.swing.text.GapContent\$UndoPosRef@13c7378]]]]
- limit: 100 indexOfNextAdd: 4



只有在 undo 後這些 string 才有內容

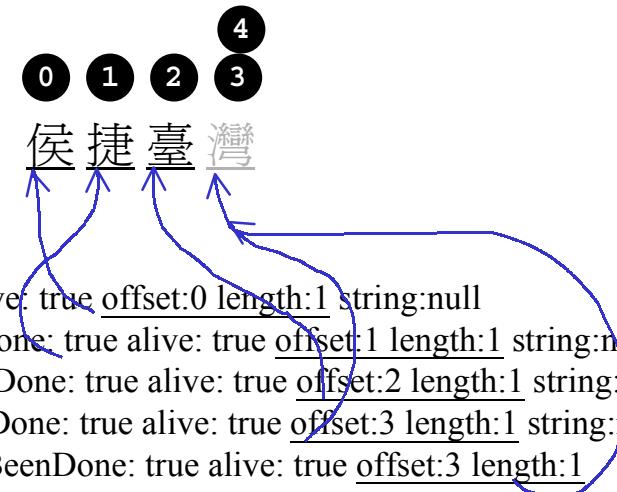


Undoable (for TextArea) 運行剖析,2

續上頁，按一次 [BackSpace] :

javax.swing.undo.UndoManager@53fb57 has
inProgress: true edits:

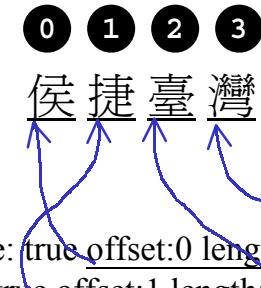
- ① [[javax.swing.text.GapContent\$InsertUndo@19a32e0 hasBeenDone: true alive: true offset:0 length:1 string:null posRefs:null], [javax.swing.text.GapContent\$InsertUndo@8238f4 hasBeenDone: true alive: true offset:1 length:1 string:null posRefs:null], [javax.swing.text.GapContent\$InsertUndo@1b5340c hasBeenDone: true alive: true offset:2 length:1 string:null posRefs:null], [javax.swing.text.GapContent\$InsertUndo@16c163f hasBeenDone: true alive: true offset:3 length:1 string:null posRefs:null], [javax.swing.text.GapContent\$RemoveUndo@15e0873 hasBeenDone: true alive: true offset:3 length:1 string:灣 posRefs:[javax.swing.text.GapContent]]]
- ② limit: 100 indexOfNextAdd: 5



按一次 [Undo] :

javax.swing.undo.UndoManager@53fb57 ha
inProgress: true edits:

- ① [[javax.swing.text.GapContent\$InsertUndo@19a32e0 hasBeenDone: true alive: true offset:0 length:1 string:null posRefs:null], [javax.swing.text.GapContent\$InsertUndo@8238f4 hasBeenDone: true alive: true offset:1 length:1 string:null posRefs:null], [javax.swing.text.GapContent\$InsertUndo@1b5340c hasBeenDone: true alive: true offset:2 length:1 string:null posRefs:null], [javax.swing.text.GapContent\$InsertUndo@16c163f hasBeenDone: true alive: true offset:3 length:1 string:null posRefs:null], [javax.swing.text.GapContent\$RemoveUndo@15e0873 hasBeenDone: false alive: true offset:3 length:1 string:null posRefs:null]]
- ② limit: 100 indexOfNextAdd: 4



Undoable (for TextArea) 運行剖析,3

續上頁，按一次 [Undo] :

```

javax.swing.undo.UndoManager@453807 hasBe
inProgress: true edits:
① [[javax.swing.text.GapContent$InsertUndo@618d26 hasBeenDone: true alive: true offset:0 length:1 string:null
② posRefs:null], [javax.swing.text.GapContent$InsertUndo@79e304 hasBeenDone: true alive: true offset:1 length:1 string:null
③ posRefs:null], [javax.swing.text.GapContent$InsertUndo@3fa5ac hasBeenDone: true alive: true offset:2 length:1 string:null
④ posRefs:null], [javax.swing.text.GapContent$InsertUndo@95cfbe hasBeenDone: false alive: true offset:3 length:1 string:灣
posRefs:[javax.swing.text.GapContent$UndoPosRef@15c07d8]],
⑤ [javax.swing.text.GapContent$RemoveUndo@1878144 hasBeenDone: false alive: true offset:3 length:1 string:null
posRefs:null]]
limit: 100 indexOfNextAdd: 3

```



輸入 '北' :

```

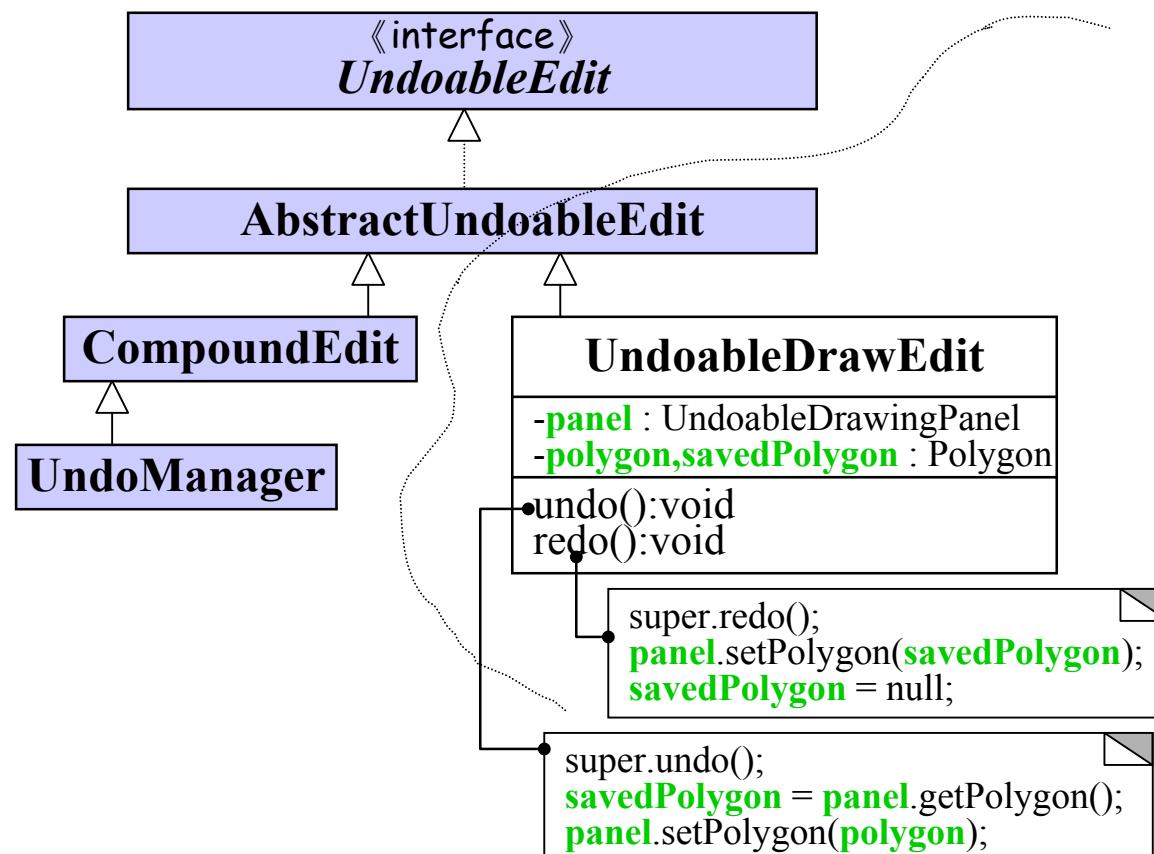
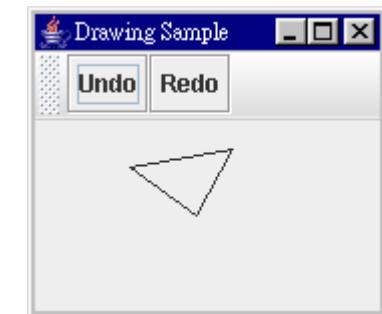
javax.swing.undo.UndoManager@453807 hasBe
inProgress: true edits:
① [[javax.swing.text.GapContent$InsertUndo@618d26 hasBeenDone: true alive: true offset:0 length:1 string:null posRefs:null],
② [javax.swing.text.GapContent$InsertUndo@79e304 hasBeenDone: true alive: true offset:1 length:1 string:null posRefs:null],
③ [javax.swing.text.GapContent$InsertUndo@3fa5ac hasBeenDone: true alive: true offset:2 length:1 string:null posRefs:null],
④ [javax.swing.text.GapContent$InsertUndo@13c7378 hasBeenDone: true alive: true offset:3 length:1 string:null posRefs:null]]
limit: 100 indexOfNextAdd: 4

```

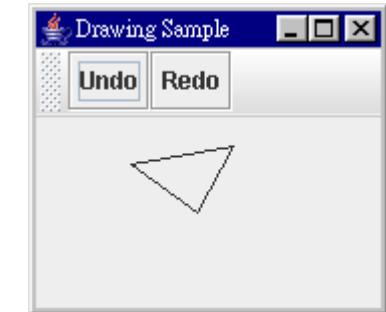


Undoable (for Drawing)

<http://www.java2s.com/ExampleCode/Swing-JFC/UndoDrawing.htm>



Undoable (for TextArea) 運行剖析, 1



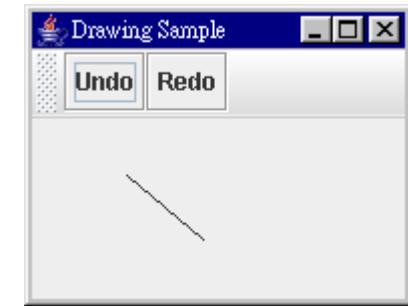
```
javax.swing.undo.UndoManager@119dc16 hasBeenDone: true alive: true
inProgress: true edits:
[UndoableDrawEdit@c05d3b hasBeenDone: true alive: true
panel:UndoableDrawingPanel[,0,34,292x89,layout=java.awtFlowLayout,alignmentX=0.0,alignmentY=0.0,border=,flags=9,maximumSize=,minimumSize=,preferredSize=]
polygon:java.awt.Polygon@18f1d7e npoints:0 xpoints[4]. ypoints[4]. bounds:null
savedPolygon:null,
UndoableDrawEdit@64883c hasBeenDone: true alive: true
panel:UndoableDrawingPanel[,0,34,292x89,layout=java.awtFlowLayout,alignmentX=0.0,alignmentY=0.0,border=,flags=9,maximumSize=,minimumSize=,preferredSize=]
polygon:java.awt.Polygon@2c1e6b npoints:1 xpoints[1]. ypoints[1]. bounds:null
savedPolygon:null,
UndoableDrawEdit@153f67e hasBeenDone: true alive: true
panel:UndoableDrawingPanel[,0,34,292x89,layout=java.awtFlowLayout,alignmentX=0.0,alignmentY=0.0,border=,flags=9,maximumSize=,minimumSize=,preferredSize=]
polygon:java.awt.Polygon@15bdc50 npoints:2 xpoints[2]. ypoints[2]. bounds:null
savedPolygon:null]
limit: 100 indexOfNextAdd: 3
```

Undoable (for TextArea) 運行剖析,2

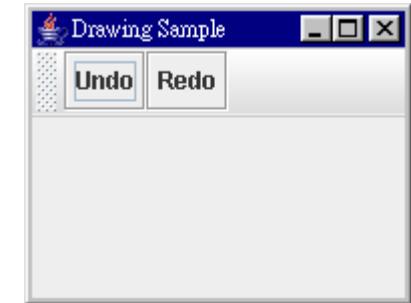
續上頁 [Undo]一次

```

javax.swing.undo.UndoManager@119dc16 hasBeenDone: true alive: true
inProgress: true edits:
[UndoableDrawEdit@c05d3b hasBeenDone: true alive: true
panel:UndoableDrawingPanel[,0,34,292x89,layout=java.awtFlowLayout,alignmentX=0.0,alignmentY=0.0,border=,flags=9,maximumSize=,minimumSize=,preferredSize=]
polygon:java.awt.Polygon@18f1d7e npoints:0 xpoints[4]. ypoints[4]. bounds:null
savedPolygon:null,
UndoableDrawEdit@64883c hasBeenDone: true alive: true
panel:UndoableDrawingPanel[,0,34,292x89,layout=java.awtFlowLayout,alignmentX=0.0,alignmentY=0.0,border=,flags=9,maximumSize=,minimumSize=,preferredSize=]
polygon:java.awt.Polygon@2c1e6b npoints:1 xpoints[1]. ypoints[1]. bounds:null
savedPolygon:null,
UndoableDrawEdit@153f67e hasBeenDone: false alive: true
panel:UndoableDrawingPanel[,0,34,292x89,layout=java.awtFlowLayout,alignmentX=0.0,alignmentY=0.0,border=,flags=9,maximumSize=,minimumSize=,preferredSize=]
polygon:java.awt.Polygon@15bdc50 npoints:2 xpoints[2]. ypoints[2]. bounds:null
savedPolygon:java.awt.Polygon@170bea5 npoints:3 xpoints[3]. ypoints[3]. bounds:null]
limit: 100 indexOfNextAdd: 2
```



Undoable (for TextArea) 運行剖析,3



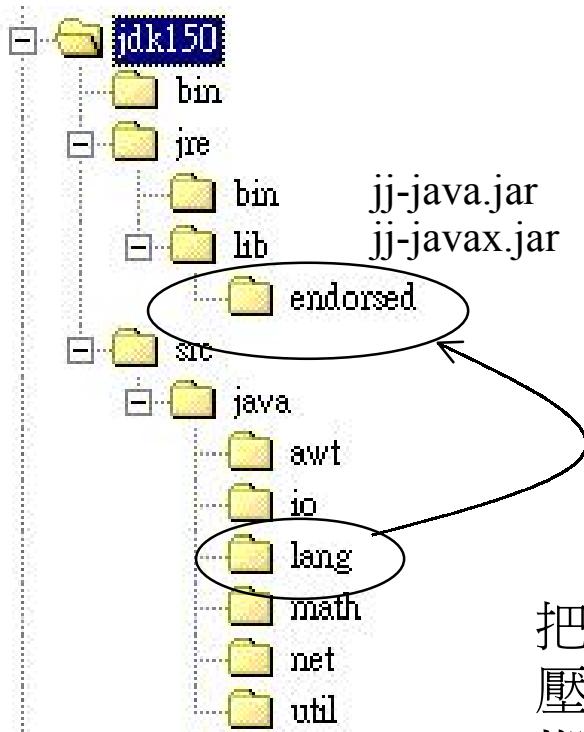
續上頁再 [Undo]一次

```

javax.swing.undo.UndoManager@119dc16 hasBeenDone: true alive: true
inProgress: true edits:
[UndoableDrawEdit@c05d3b hasBeenDone: true alive: true
panel:UndoableDrawingPanel[,0,34,292x89,layout=java.awtFlowLayout,alignmentX=0.0,alignmentY=0.0,border=,flag
s=9,maximumSize=,minimumSize=,preferredSize=]
polygon:java.awt.Polygon@18f1d7e npoints:0 xpoints[4]. ypoints[4]. bounds:null
savedPolygon:null,
undoableDrawEdit@64883c hasBeenDone: false alive: true
panel:UndoableDrawingPanel[,0,34,292x89,layout=java.awtFlowLayout,alignmentX=0.0,alignmentY=0.0,border=,flag
s=9,maximumSize=,minimumSize=,preferredSize=]
polygon:java.awt.Polygon@2c1e6b npoints:1 xpoints[1]. ypoints[1]. bounds:null
savedPolygon:java.awt.Polygon@9cbd4b npoints:2 xpoints[2]. ypoints[2]. bounds:null,
undoableDrawEdit@153f67e hasBeenDone: false alive: true
panel:UndoableDrawingPanel[,0,34,292x89,layout=java.awtFlowLayout,alignmentX=0.0,alignmentY=0.0,border=,flag
s=9,maximumSize=,minimumSize=,preferredSize=]
polygon:java.awt.Polygon@15bdc50 npoints:2 xpoints[2]. ypoints[2]. bounds:null
savedPolygon:java.awt.Polygon@170bea5 npoints:3 xpoints[3]. ypoints[3]. bounds:null]
limit: 100 indexOfNextAdd: 1

```

Java 源碼修改經驗



```

rem "rejava.bat" usage :
rem (1) enter the directory which you want to rewrite java source.
rem (2) backup the original java source "x.java" to "x.java_ori".
rem (3) modify java source.
rem (4) rejava path filename (no extension name)
rem     ex: rejava javax\swing\text GapContent

del c:\jdk150\jre\lib\endorsed\jj-javax.jar
del c:\jdk150\jre\lib\endorsed\jj-java.jar
javac %2.java
move *.class e:\%1
e:
cd e:\%
jar cvfM c:\jdk150\jre\lib\endorsed\jj-javax.jar javax
jar cvfM c:\jdk150\jre\lib\endorsed\jj-java.jar  java
dir e:\%1
dir c:\jdk150\jre\lib\endorsed
c:

```

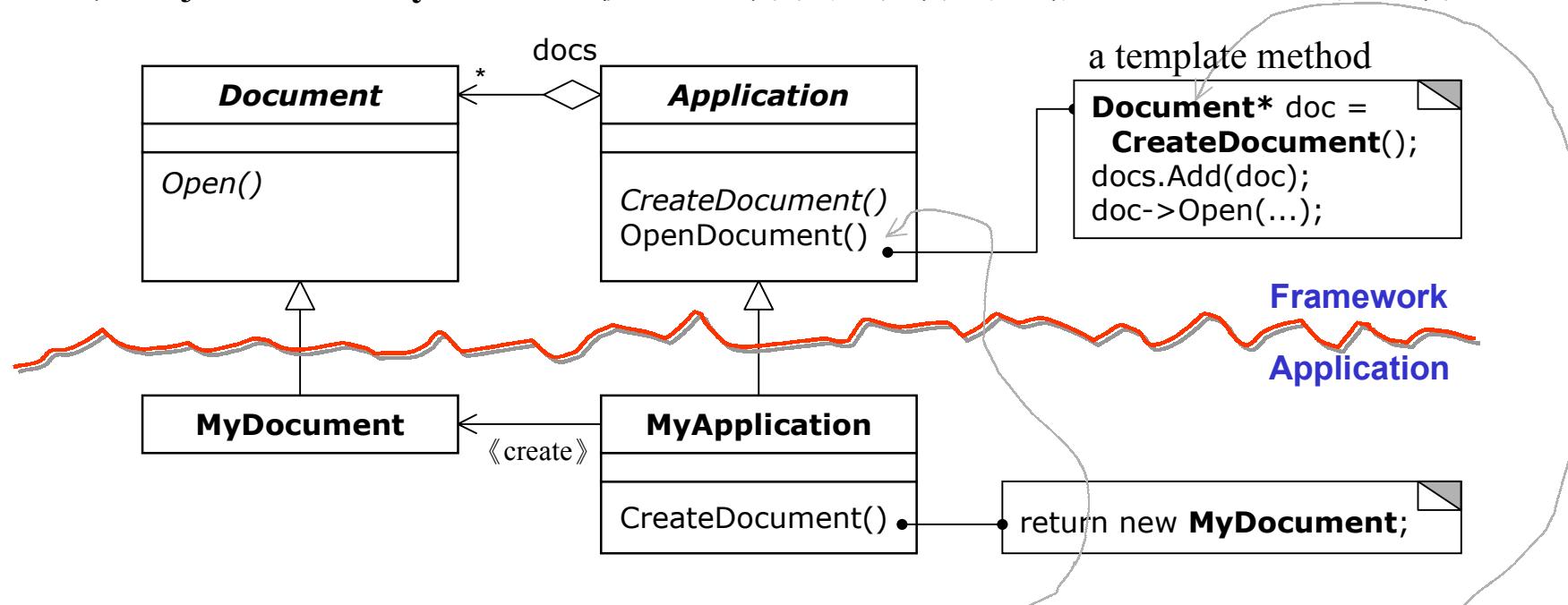
把修改後的.java 編譯為.class，
 壓縮為 xxx.jar (需帶路徑例如 **java\lang**) 並
 搬移到**endorsed**，
 即可被class loader優先讀取。



10. Factory Method in GOF

Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class **defer** instantiation to subclasses.

定義一個用來 creating object 的介面，但讓 subclasses 決定最終要具現出哪一種 object。Factory Method 使 class 得將具現行爲延緩至 subclasses 再進行。

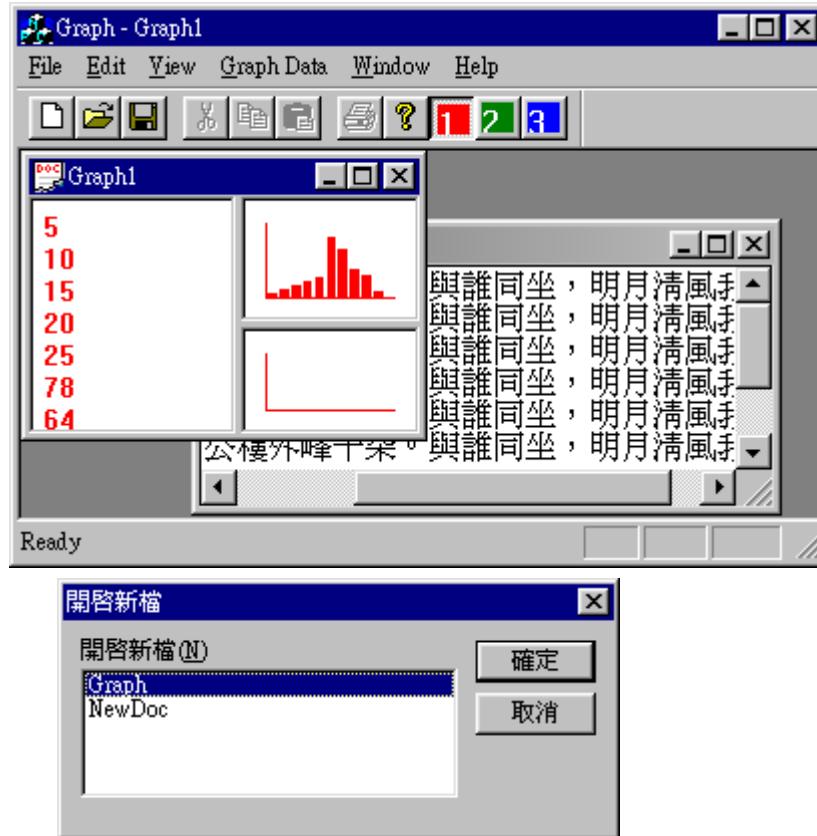


Framework 無法預先知道該產生哪一種document object，所以利用**Factory Method**將決定權延緩至subclass。Subclass 位於 Application 這邊，必能夠知道應該產生何種document。由於 framework 無法知道確切的 document 種類，所以宣告其 type 時必須夠模糊、夠泛化“採用 document base class 是最保險的作法！”

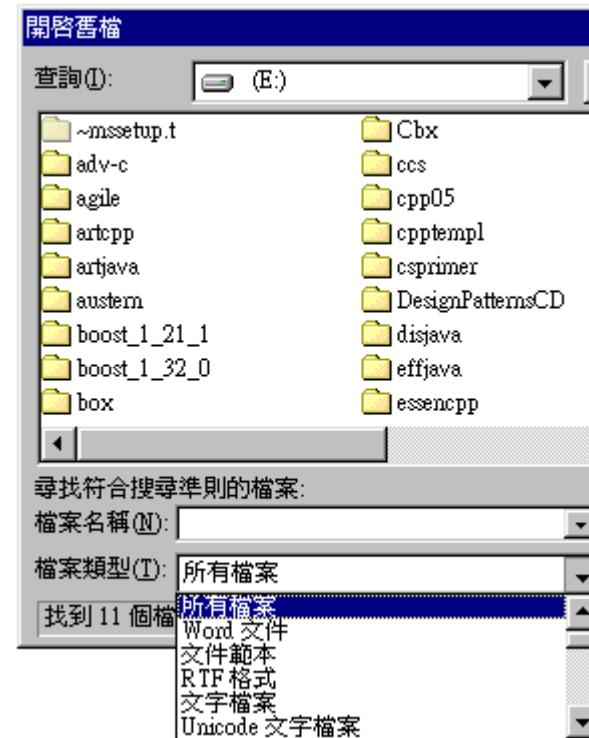


10. Factory Method, usage

《深入淺出MFC》chap13 範例



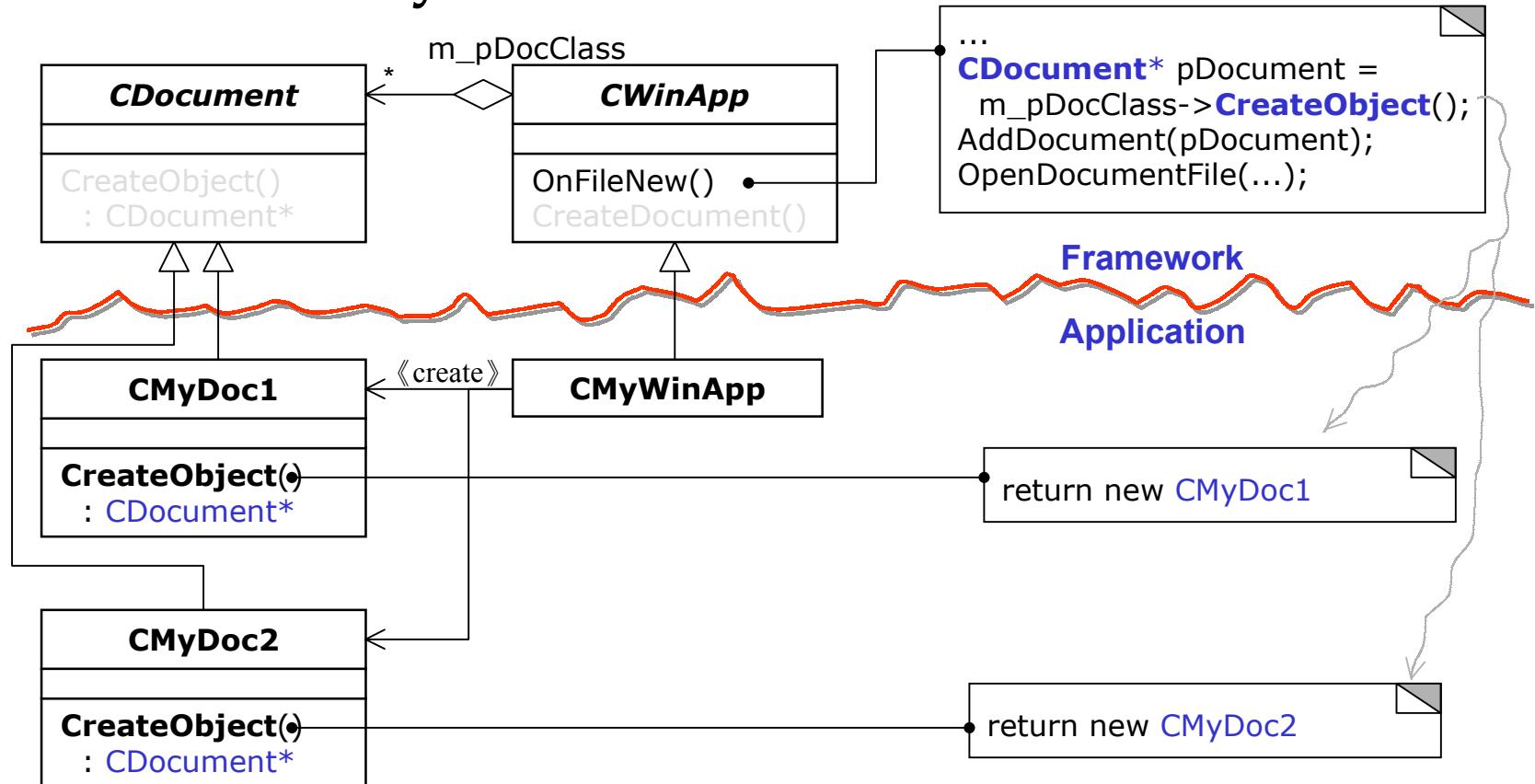
Microsoft Word



Template Method 是在 base class 建立處理大綱，在 subclass 內完成具體內容。如果將 **Template Method** 應用在 object creation 方面，就成為所謂的 **Factory Method**。也就是說以 **Template Method** 架構出「用以產生 object 的工廠」。



10. Factory Method in MFC



注意：MFC 中的 `CreateObject()` 係由 **DECLARE_DYNCREATE** 和 **IMPLEMENT_DYNCREATE** macro 擴展而得（而 **DECLARE_SERIAL** 和 **IMPLEMENT_SERIAL** macro 又涵蓋了前者）。

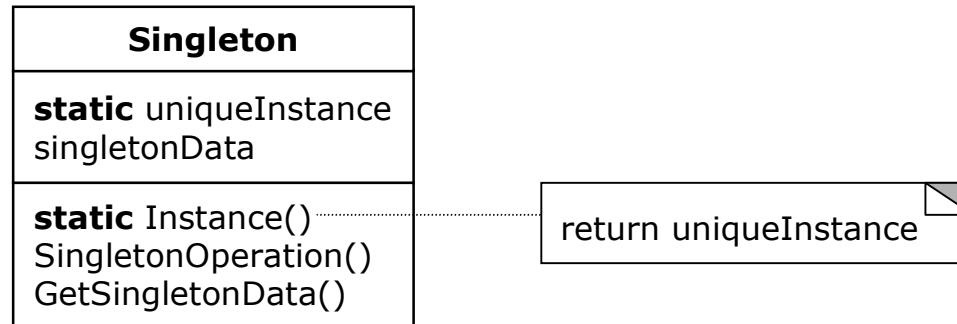
注意：**CDocument** 並沒有 `virtual CreateObject()`，其 subclasses 則被規定應帶有上述 macros 以求展開獲得 `CreateObject()`。**CWinApp** 中也沒有 `virtual CreateDocument()`。這便是為什麼 **CWinApp::OnFileNew()** 呼叫 `m_pDocClass->CreateObject()` 而非如 GoF 呼叫 `CreateDocument()` 的緣故。



19. Singleton in GOF

Ensure a class only has one instance, and provide a global point of access to it.

確保某 class 只能生成唯一一個實體，並為它提供單一的全域存取窗口。



下面這些 patterns 通常都只有一個 instance，可使用 Singleton.

- Abstract Factory
- Builder
- Façade
- Prototype
- State (←侯捷補充)
- Flyweight's factory (←侯捷補充)



19. Singleton in MEC. function static (禁絕法)

每當即將產生一個物件，我們確知一件事情：會有一個 constructor 被喚起。

「阻止某個 class 產出物件」的最簡單方法就是將其 constructors 宣告為 private：

```
class Printer {  
public:  
    static Printer& getInstance();  
    ...  
private:  
    Printer(); <  
    Printer(const Printer& rhs);  
    ...  
};  
Printer& Printer::getInstance() {  
    static Printer p;  
    return p;  
}
```

單一全域存取窗口

clients 取用印表機：

Printer::getInstance().xxx();

此物件在函式第一次被呼叫時才產生。如果該函式從未被呼叫，這個物件也就絕不會誕生。我們確切知道一個 function static 的初始化時機：在該函式第一次被呼叫時，並且在該 static 被定義處。至於一個 class static 則不一定在什麼時候初始化。



19. Singleton in MEC. object counting (計數法)

另一個作法是計算目前存在的物件個數，並於外界申請太多物件時，於 constructor 內丟出一個 exception。

```
class Printer {  
public:  
    class TooManyObjects{}; // 當外界申請太多物件時，丟出這種 exception  
    Printer();  
    ~Printer();  
    ...  
private:  
    static size_t numObjects;  
    Printer(const Printer& rhs);  
    // 有著「印表機個數永遠為 1」的限制，所以絕不允許複製行為，所以 private  
};  
size_t Printer::numObjects = 0;  
  
Printer::Printer() {  
    if (numObjects >= 1) throw TooManyObjects();  
    proceed with normal construction here;  
    ++numObjects;  
}  
Printer::~Printer() {  
    perform normal destruction here;  
    --numObjects;  
}
```

此法直截而易懂，也很容易被一般化，使物件的最大數量可以設定為 1 以外的值。



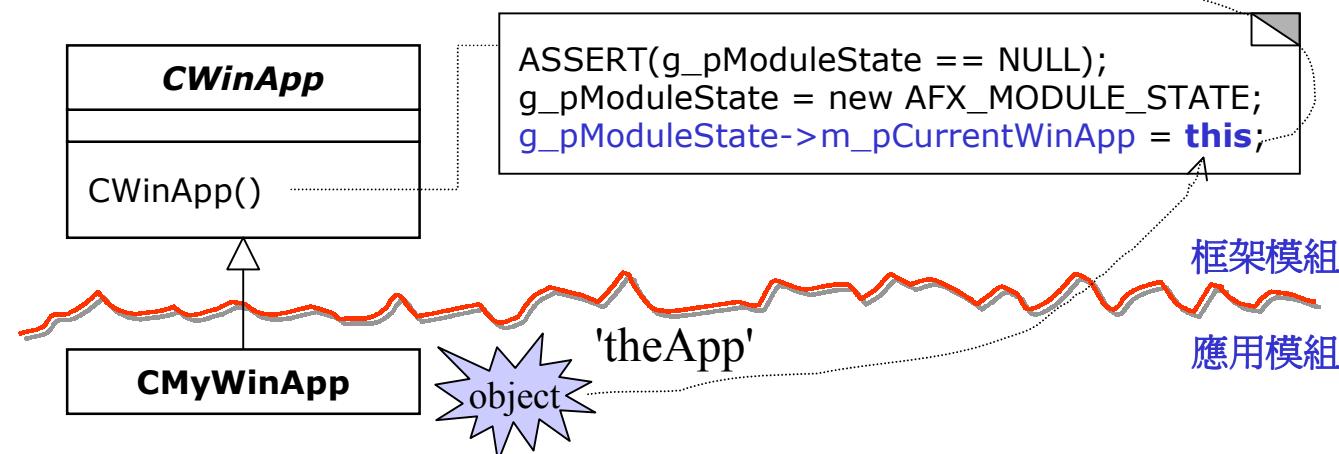
19. Singleton in MFC

```
AFX_MODULE_STATE* AfxGetModuleState() { return g_pModuleState; }  
#define afxCurrentWinApp AfxGetModuleState()->m_pCurrentWinApp
```

```
AFX_MODULE_STATE* g_pModuleState = NULL;  
  
CWinApp::CWinApp()  
{  
    ASSERT(g_pModuleState == NULL); ←  
    g_pModuleState = new AFX_MODULE_STATE;  
    g_pModuleState->m_pCurrentWinApp = this;  
}
```

一開始是NULL，爾後只要會產生 CWinApp-derived object，就設其值而不再是NULL。下次再準備產生 CWinApp-derived object 時檢查發現不再是NULL，編譯器就會報錯。

```
CWinApp* AfxGetApp()  
{ return g_pModuleState->m_pCurrentWinApp; }
```





1. Abstract Factory

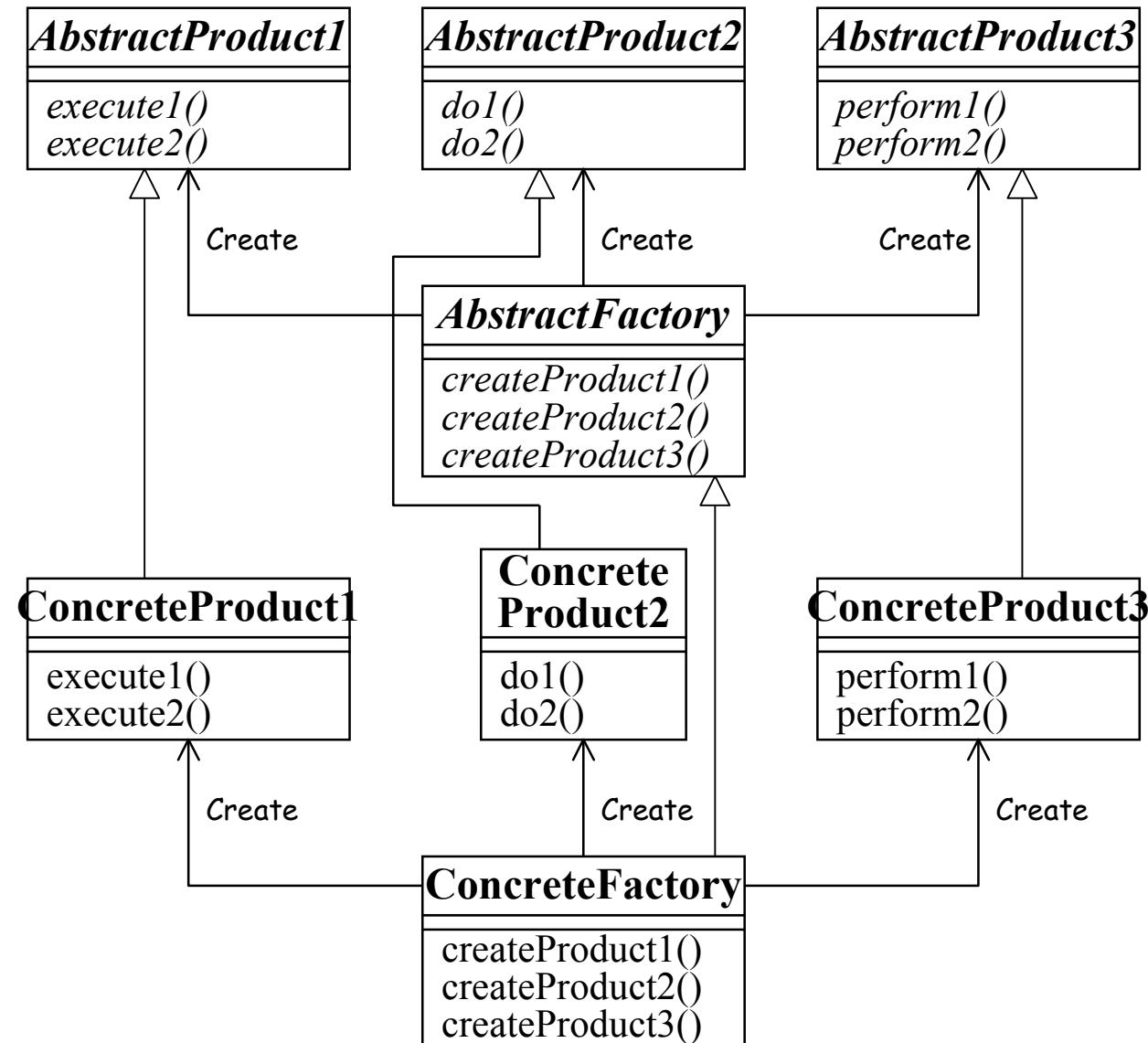
1. Abstract Factory, Kid (87)

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

提供單一介面，不需指明 concrete classes（而是使用 abstract class -- 也就是抽象工廠）就可以產生一整族系相關或相依的 objects（i.e. 產品）。



1. Abstract Factory





1. Abstract Factory in DP-in-Java

"工廠"是生產各種零組件並將其組合成完整產品的地點，所作所為再具體不過了。抽象工廠是違反常理的說法。但其實在 Abstract Factory 中不但有抽象工廠，也有抽象零件和抽象產品。

抽象工廠就是把抽象零件組合為抽象產品的地方。關鍵在於「以介面完成」而非「以具體實作來完成」。

舉例：有些 items 要做成一個 Web page。呈現方式有兩種，(1) list 或 (2) table，其所需要的元素是同一套（caption, title, url...），但 list 或 table 所使用的 HTML tag 不同。因此，可以寫出兩個具象工廠：ListFactory 和 TableFactory。

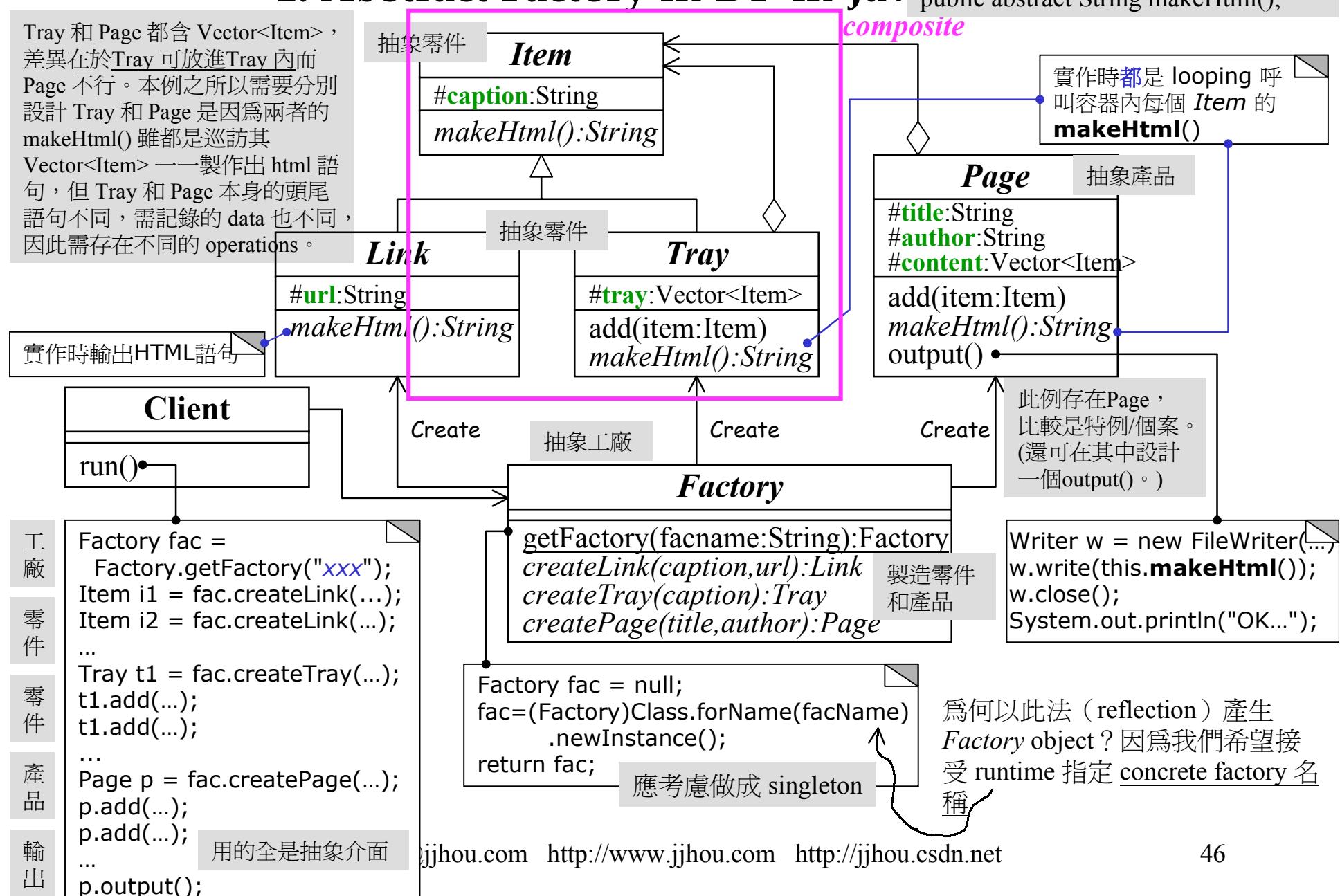


1. Abstract Factory in DP-in-Java

若更嚴謹可讓 Item 和 Page 都 implements Htmlable，此介面內只有 public abstract String makeHtml();

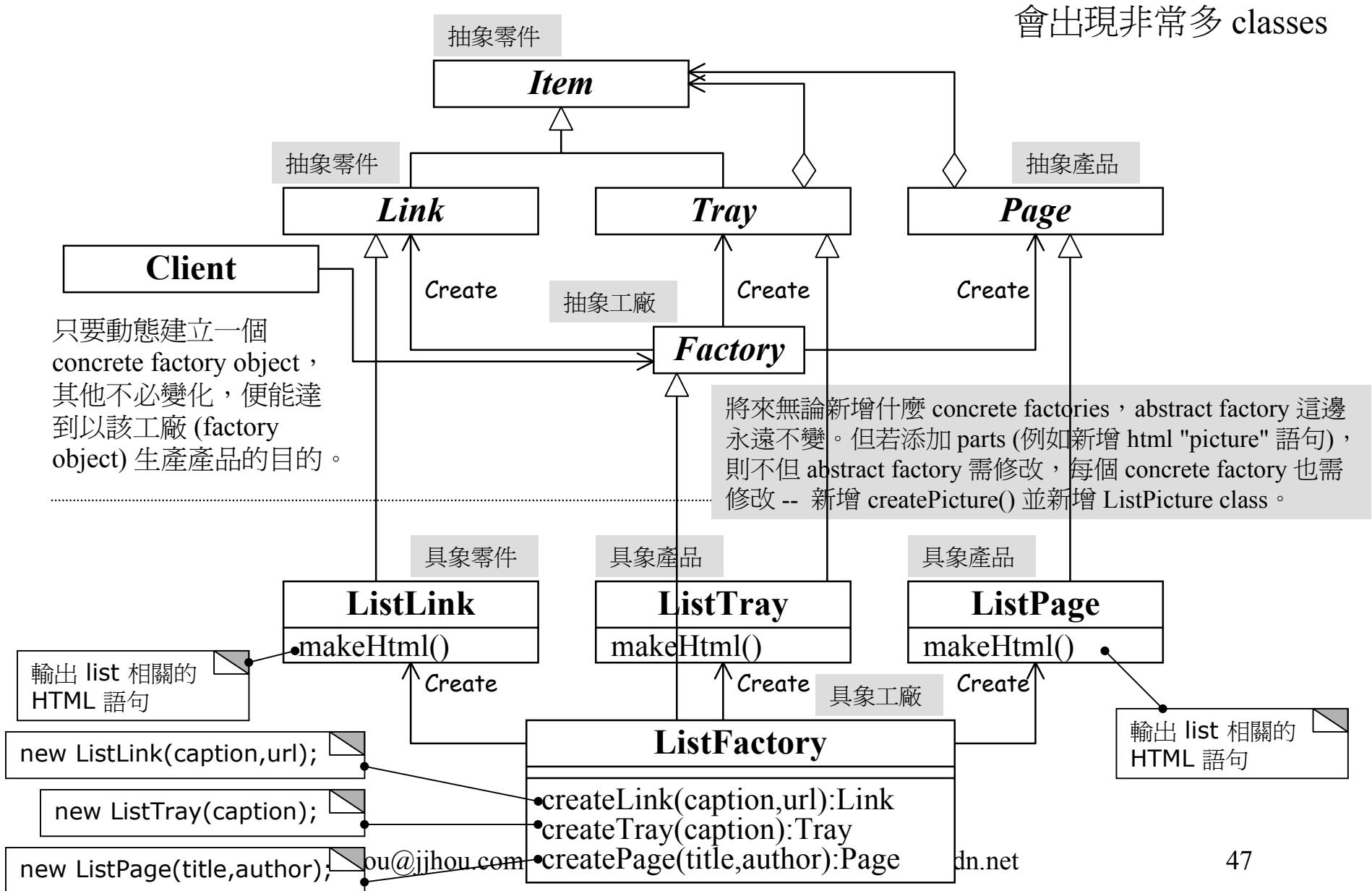
Tray 和 Page 都含 `Vector<Item>`，
差異在於 Tray 可放進 Tray 內而
Page 不行。本例之所以需要分別
設計 Tray 和 Page 是因為兩者的
`makeHtml()` 雖都是巡訪其
`Vector<Item>` 一一製作出 html 語
句，但 Tray 和 Page 本身的頭尾
語句不同，需記錄的 `data` 也不同
因此需存在不同的 `operations`。

實作時輸出HTML語句



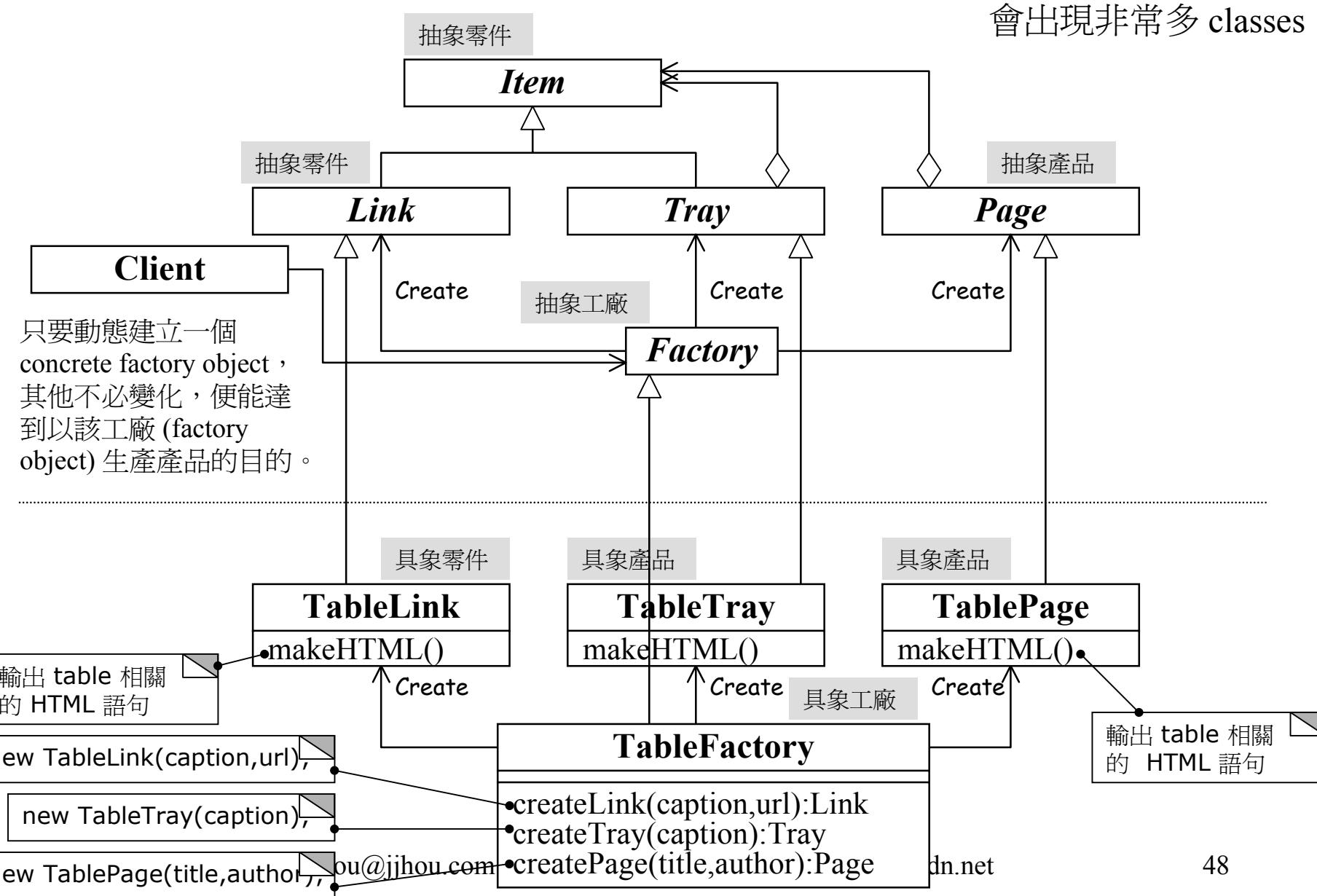


1. Abstract Factory in DP-in-Java





1. Abstract Factory in DP-in-Java



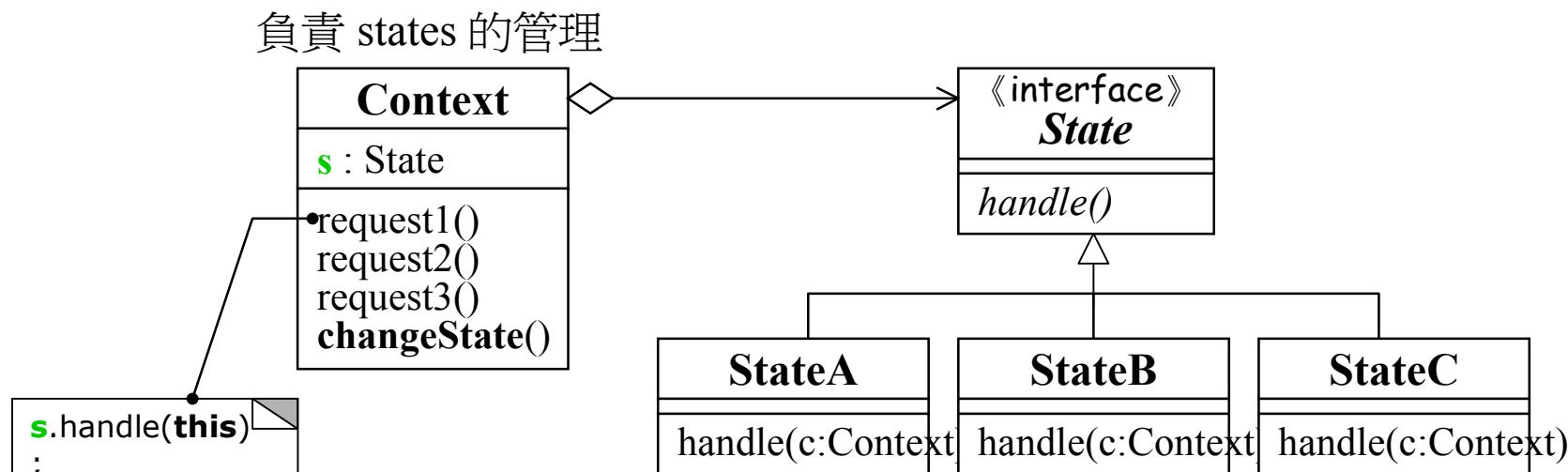


20. State

20. State (305)

Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

允許 object 在其內部狀態有變化時改變其行爲，使得好像其 class 也隨之調整了（因為行爲改變了）。



Context object 一旦呼叫 `changeState()` 就是要改用 `StateA` 或 `StateB` 或 `StateC`，而後 Context object 的各個動作仍然維持呼叫 `s.handle()`，但行爲已經變了。

此圖十分類似 **Strategy**



20. State in DP-in-Java

在 OOP 中，programmer 都知道以 class 來表達程式。但 **程式中的哪些東西該以 class 來表現** 就見仁見智。Class 所對應的東西有可能是真實生活中的實物，但也有可能 "虛無縹渺"。

State pattern 就是以 class 來表現 "狀態" (state) 。

Command pattern 則是以 class 來表現 "命令" (動作, command) 。

以 class 表現 state 之後，只要 切換 class 就能表現出 "狀態變化" 的現實。

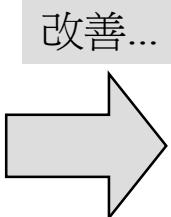
當 **程式的進行取決於某物狀態** 時，可使用 **State**。例如在一個金庫系統中，有金庫 + 電話 + 警鈴。這三種東西的功能和操作結果都依 "白天"、"夜晚" 兩種狀態而不同，這便適合使用 **State** 來解決問題。



20. State in DP-in-Java

例如在一個金庫系統中，有金庫+電話+警鈴。這三種東西的功能和操作結果都依 "白天"、"夜晚" 兩種狀態而不同，這便可使用 **State** 來解決問題。

```
doUse() { //使用金庫
    if(day) ...
    else if(night) ...
}
doAlarm() { //警鈴
    if(day) ...
    else if(night) ...
}
doPhone() { //電話
    if(day) ...
    else if(night) ...
}
```



作法：各種 state 出現在 method 中以 if, else 來調查。狀態種類愈多，條件判斷式就愈多，而且每一個 do-methods 都要寫相同的條件判斷式

作法：各種 states 以 class 表現，method 內不再有狀態調查。

```
class Day {
    doUse() {
        ...
    }
    doAlarm() {
        ...
    }
    doPhone() {
        ...
    }
}
class Night {
    doUse() {
        ...
    }
    doAlarm() {
        ...
    }
    doPhone() {
        ...
    }
}
```

concrete state class 常常只生成一個 object。這時可使用 **Singleton**。



20. State in DP-in-Java



```
buttonUse.addActionListener(this);
buttonAlarm.addActionListener(this);
buttonPhone.addActionListener(this);
buttonExit.addActionListener(this);
```

加上監聽器。
若有人按下按鈕則跳到這裡

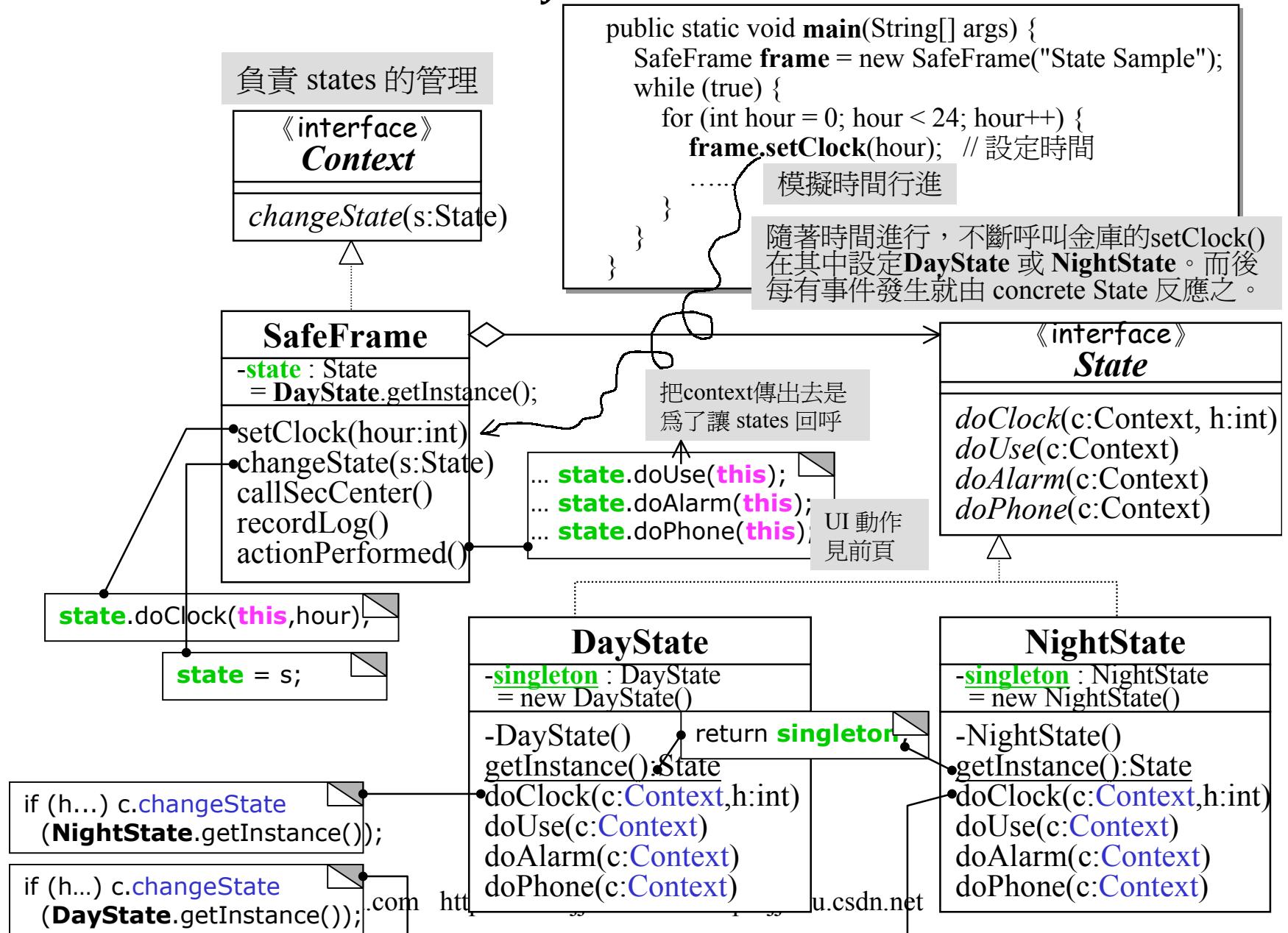
in class SafeFrame (金庫)

Listener/Observer pattern

```
public void actionPerformed(ActionEvent e) {
    System.out.println("'" + e);
    if (e.getSource() == buttonUse) {          // 使用金庫
        state.doUse(this);
    } else if (e.getSource() == buttonAlarm) {   // 警鈴
        state.doAlarm(this);
    } else if (e.getSource() == buttonPhone) {   // 一般通話
        state.doPhone(this);
    } else if (e.getSource() == buttonExit) {     // 結束
        System.exit(0);
    } else {
        System.out.println("?");
    }
}
```



20. State in DP-in-Java





4. Builder

4. Builder (97)

Separate the **construction** of a **complex object** from its **representation** so that the same construction process can create different representations.

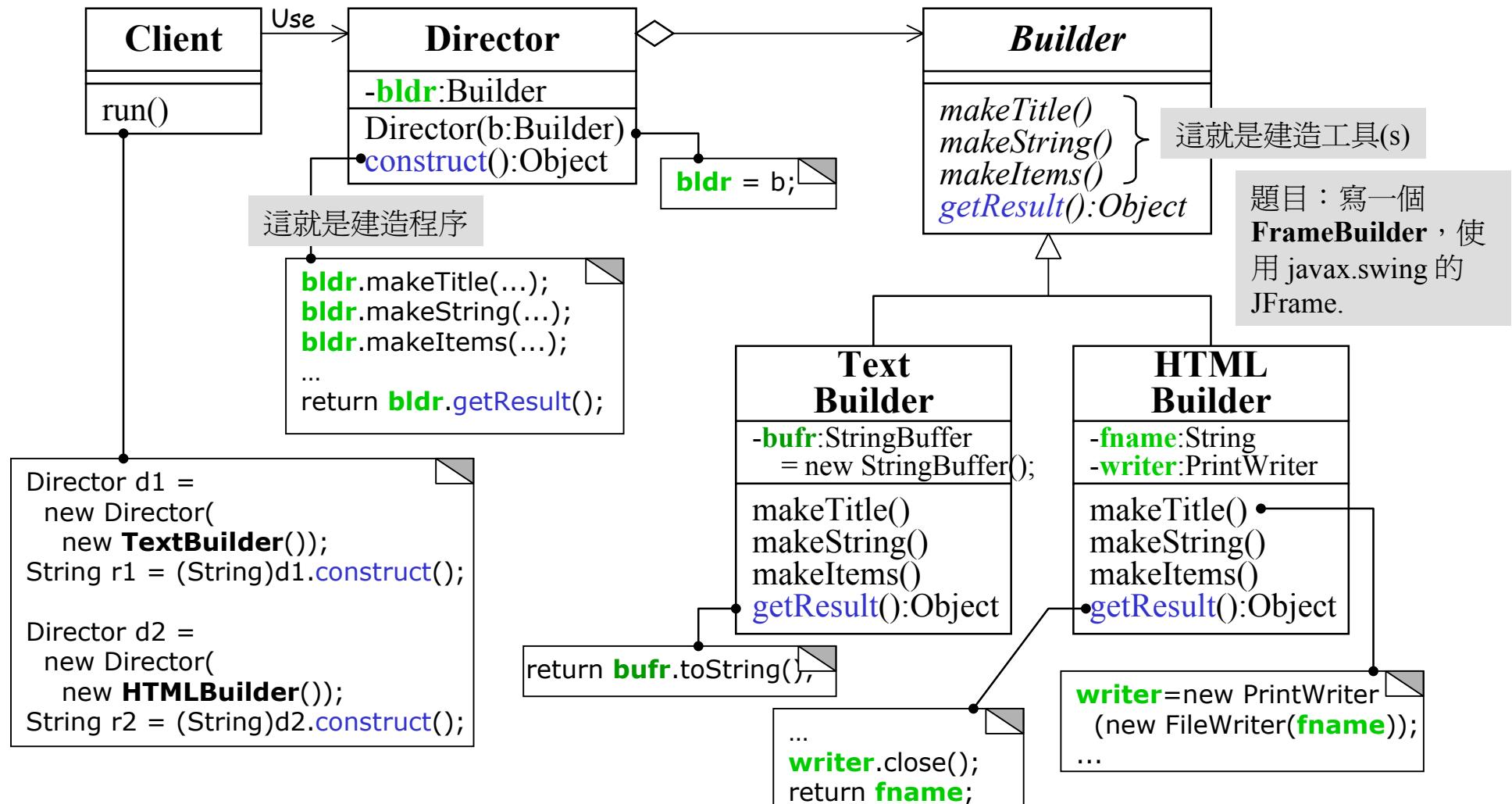
從複雜物件的 representation/表述格式中分離（提取）出 construction/建造程序，使相同的 construction process/建造程序可以產生不同的 representation/表述格式。

例子 in GoF：我們希望 RTF reader 可以將 RTF 轉換為其他文件格式（例如 Ascii, HTML...）。但文件格式和轉換動作似乎無窮止盡，似乎該想個一勞永逸的辦法，使得即使日後有新文件格式問世，我們也不必動手修改 RTF Reader。



4. Builder in DP-in-Java

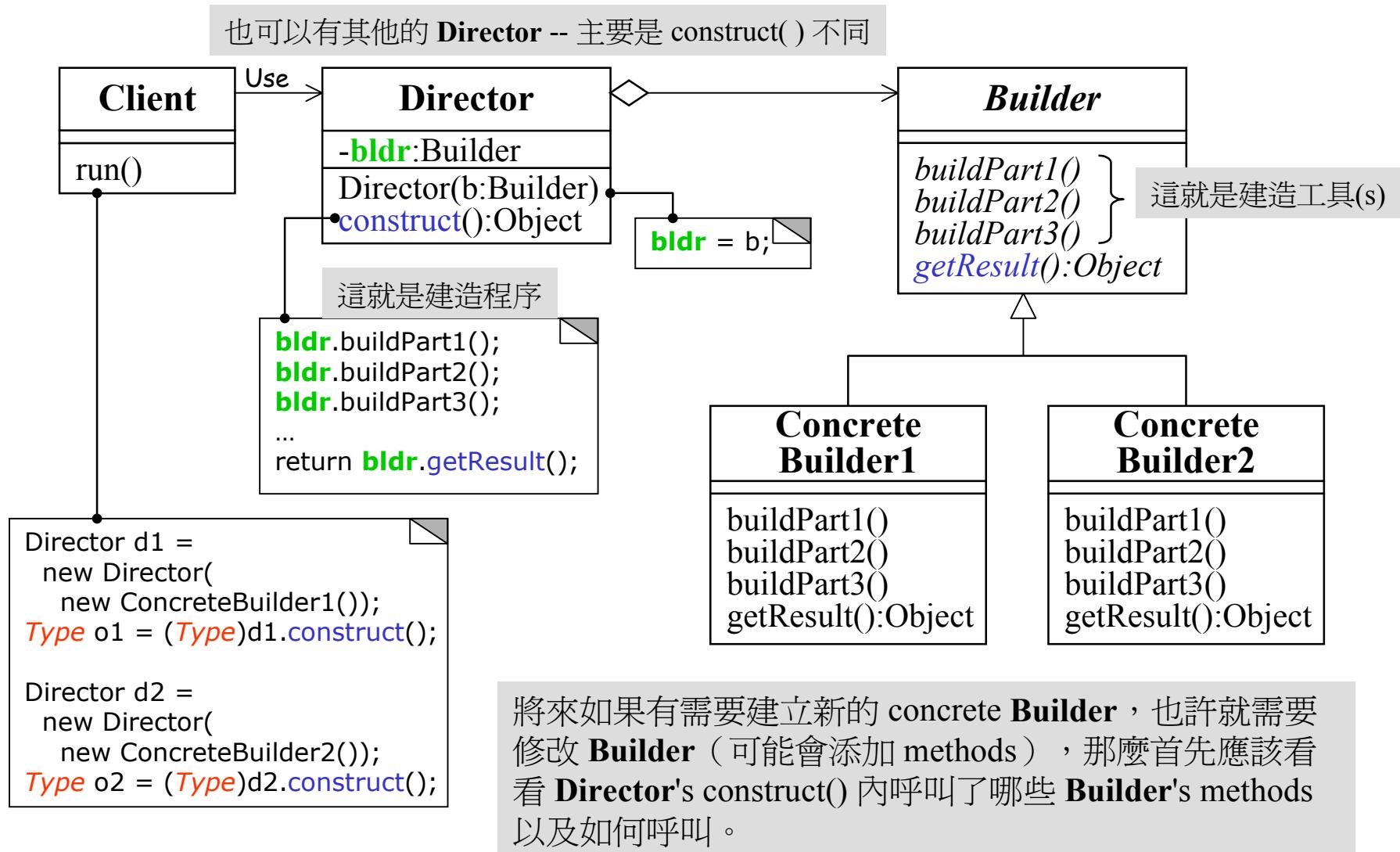
本例假想製造文件是一件複雜的工作，這份文件的構造是：含有一個標題，一些字串，一些條目。



Director 只會（只可以/只能夠）利用 **Builder's methods** 來產生東西。這樣 **Director** 就可以不必管實際負責生產的是哪一個 **Concrete Builder**（那將在 runtime 指定）。



4. Builder





9. Facade

9. Facade (185)

Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

爲 subsystem 內的一整組 interfaces 提供一個統一介面。Facade 定義出一個高階介面使得 subsystem 比較容易被使用。

GoF 舉例的「複雜子系統」包括：

- IDE 中的編譯器子系統
- OS 中的虛擬記憶體子系統
- ET++ 中的瀏覽器子系統



9. Façade in DP-in-Java

程式愈做愈大，classes 之間的關係愈來愈錯綜複雜。導致使用時必須花精力確實了解 classes 之間的關係，以正確的次序使用（呼叫）之。

既然如此，就設一個專門窗口。如此一來就不需個別控制classes，而是將需求丟給該專門窗口即可。Façade 能整合錯綜複雜的來龍去脈而提供較高級介面 (API)。

太多的 classes 和 methods，只會令 programmer 猶豫不知該使用哪一個，還得注意呼叫次序不能搞錯。

Programmers 可能會下意識地閃避建立 **Façade**。(1) 或許是 senior 已經把整個系統都勾勒在腦海裡，因此不成問題。(2) 或許有心炫耀自己的能力。

Façade 是單行道，**Mediator** 是雙向通車。



9. Façade in DP-in-Java

```
Properties prop=new Properties();
prop.load(new FileInputStream("maildata.txt"));
```

maildata.txt :

```
hyuki@hyuki.com=Hiroshi Yuki
hanako@hyuki.com=Hanako Sato
tomura@hyuki.com=Tomura
mamoru@hyuki.com=Mamoru Takahashi
```

makeWelcomePage()



Welcome to Hiroshi Yuki's page

歡迎來到Hiroshi Yuki的網頁。

等你來信喔！

[Hiroshi Yuki](#)



Welcome to Hanako Sato's page!

歡迎來到Hanako Sato的網頁。

等你來信喔！

[Hanako Sato](#)



Welcome to Tomura's page!

歡迎來到Tomura的網頁。

等你來信喔！

[Tomura](#)



Welcome to Mamoru Takahashi's page!

歡迎來到Mamoru Takahashi的網頁。

等你來信喔！

[Mamoru Takahashi](#)

makeLinkPage()



Link page

[Hanako Sato](#)

[Mamoru Takahashi](#)

[Hiroshi Yuki](#)

[Tomura](#)

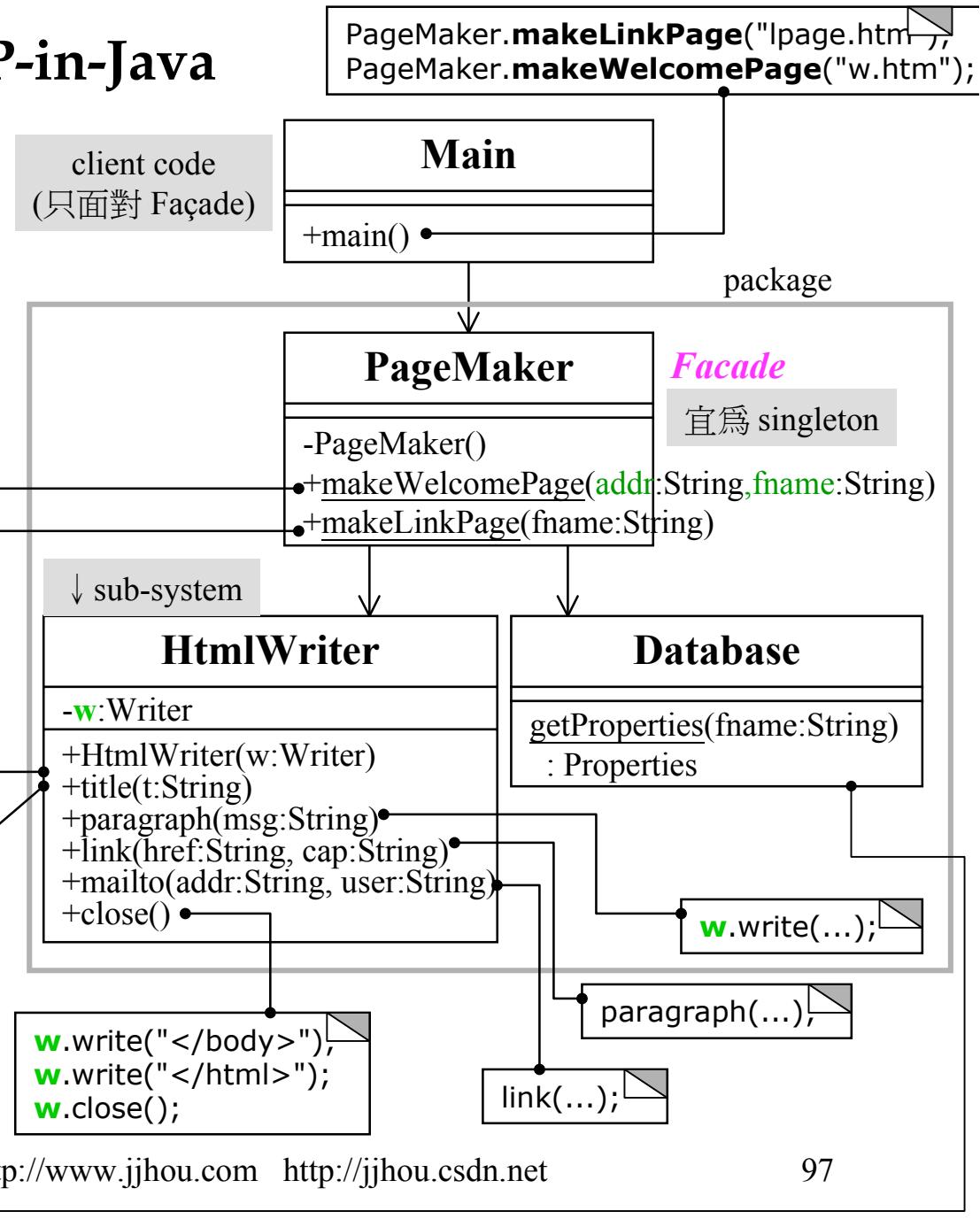
9. Façade in DP-in-Java

```
Properties mp=Database.getProperties(...);
String uname =mp.getProperty(addr);
HtmlWriter w = new HtmlWriter(
                new FileWriter(fname));
w.title("Welcome to ...");
w.paragraph(...);
w.paragraph(...);
w.mailto(addr,uname);
w.close();
```

```
HtmlWriter w = new HtmlWriter(  
                           new FileWriter(fname));  
w.title("Link Page");  
... =Database.getProperties("maildata");  
while(...) {... w.mailto(...); }  
w.close();
```

```
w.write("<html>");  
w.write("<head>");  
w.write("<title>" + ...);  
w.write("</head>");  
w.write("<body>\n" + ...);  
w.write("<h1>" + ...);
```

```
Properties prop =  
    new Properties();  
prop.load(new  
    FileInputStream(fname));  
return prop;
```



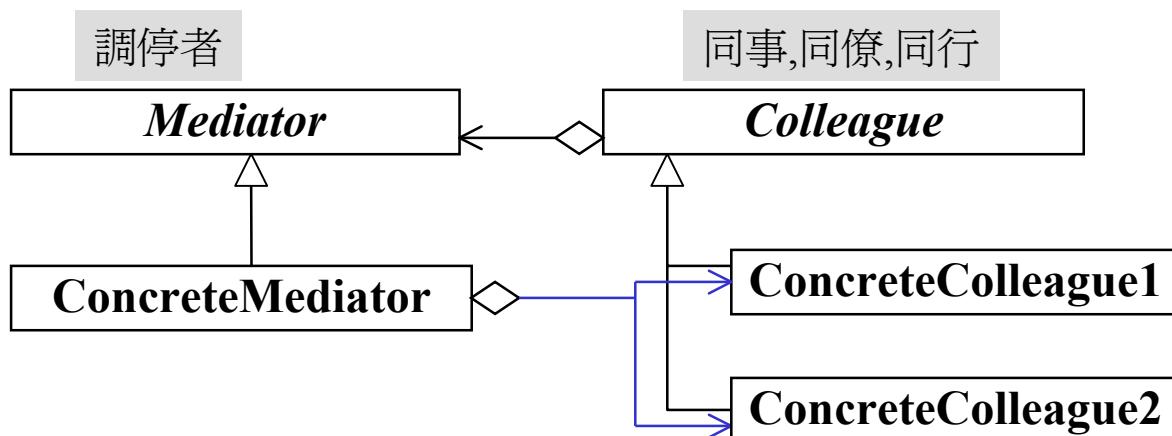


14. Mediator

14. Mediator (273)

Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

定義一個 object，將「某一群 objects 的互動方式」封裝起來。**Mediator**（調停者）推動鬆耦合，作法是令 objects 不明顯指涉（引用）彼此，於是彼此之互動影響得以獨立（並集中）於某處。





14. Mediator in DP-in-Java

程式需求：

- 挑選 Guest 或 Login

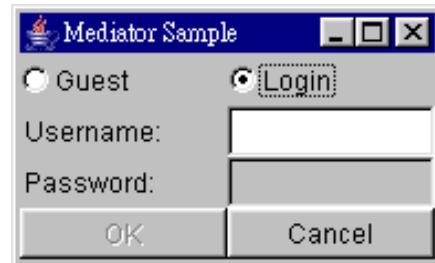
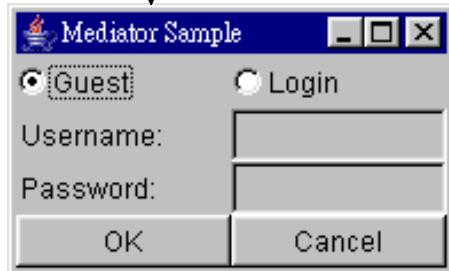
- 如果挑選 Login，則 Username 和 Cancel 可用，Password 和 OK 禁用

- 一旦 Username 有任何輸入，則 Password 可用

- 一旦 Password 有任何輸入，則 OK 可用

- 如果挑選 Guest，則 Username 和 Password 禁用，OK 和 Cancel 可用

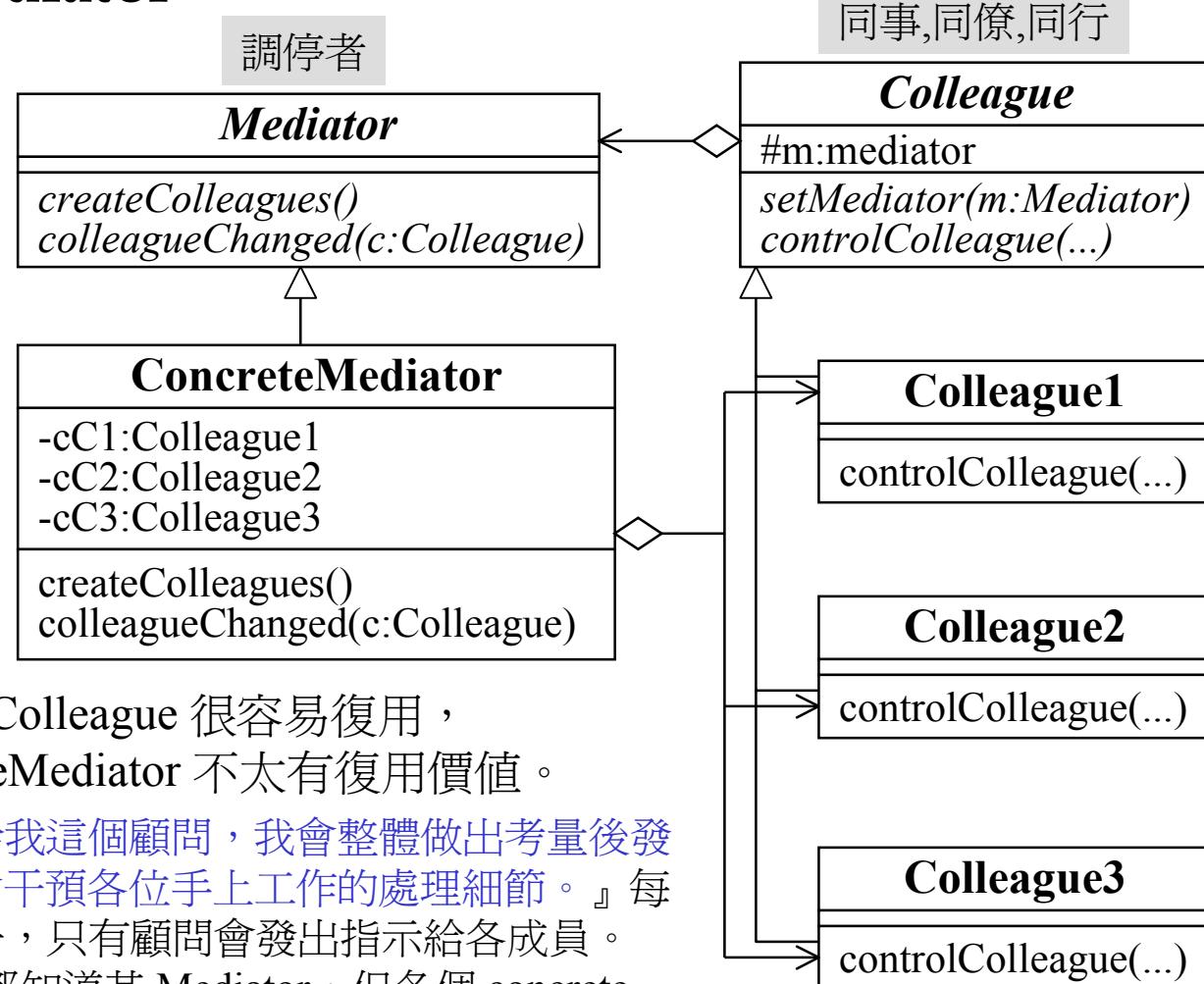
- 初始設定為 Guest



以上細節該寫在哪兒好？像這樣需要協調多個objects，就是
Mediator 發揮的時候。這些細節就寫在 Mediator 內。



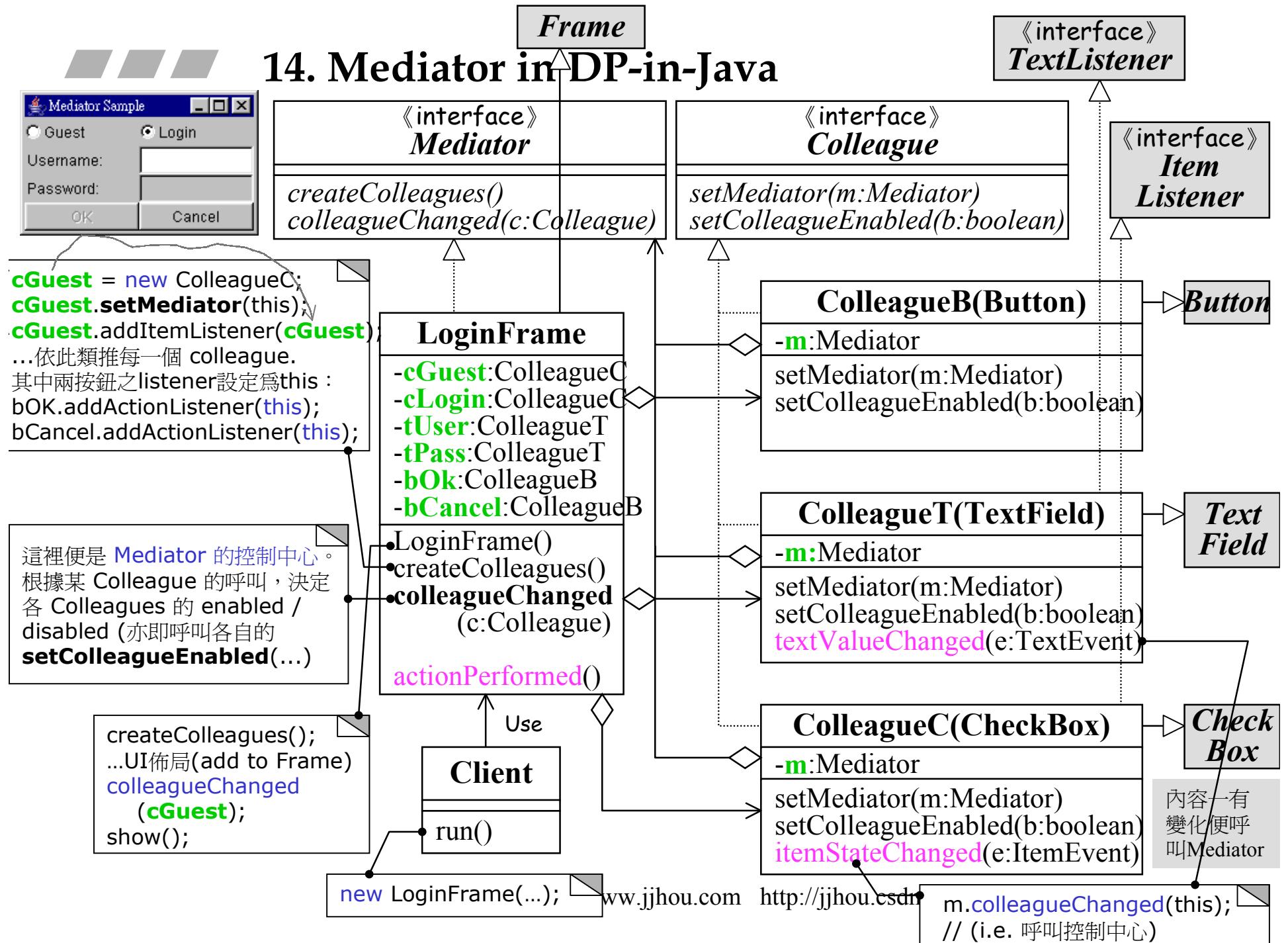
14. Mediator



ConcreteColleague 很容易復用，
ConcreteMediator 不太有復用價值。

『請各位把所有狀況回報給我這個顧問，我會整體做出考量後發給各位適當的指示。我不會干預各位手上工作的處理細節。』每位成員都只對顧問提出報告，只有顧問會發出指示給各成員。每一個 concrete colleagues 都知道其 Mediator，但各個 concrete colleague 互相不知道對方。

每一個 Windows dialog function 都可視為這層意義上的 Mediator。





14. Mediator in DP-in-Java

程式需求：

- 挑選 Guest 或 Login
 - 如果挑選 Login，則 Username 和 Cancel 可用，Password 和 OK 禁用
 - 一旦 Username 有任何輸入，則 Password 可用
 - 一旦 Password 有任何輸入，則 OK 可用
 - 如果挑選 Guest，則 Username 和 Password 禁用，OK 和 Cancel 可用
- 初始設定為 Guest

```
public void colleagueChanged(Colleague c) {  
    if (c == checkGuest || c == checkLogin) {  
        if (checkGuest.getState()) {           // Guest模式  
            textUser.setColleagueEnabled(false);  
            textPass.setColleagueEnabled(false);  
            buttonOk.setColleagueEnabled(true);  
        } else {                         // Login模式  
            textUser.setColleagueEnabled(true);  
            userpassChanged();  
        }  
    } else if (c == textUser || c == textPass) {  
        userpassChanged();  
    } else {  
        System.out.println("colleagueChanged:unknown colleague = " + c);  
    }  
}
```



3. Bridge, Handle/Body

3. Bridge (151)

Decouple an abstraction from its implementation so that the two can vary independently.

將 abstraction 和 implementation 分離（解耦），使兩者可以獨立變化。

所謂 **Abstraction**，這裡指的是「功能型 class hierarchy」。

例如 subclass 要為 base class 添加新功能。

Type

 └ TypeBetter
 └ TypeBest

所謂 **Implementation**，這裡指的是「實作型 class hierarchy」。

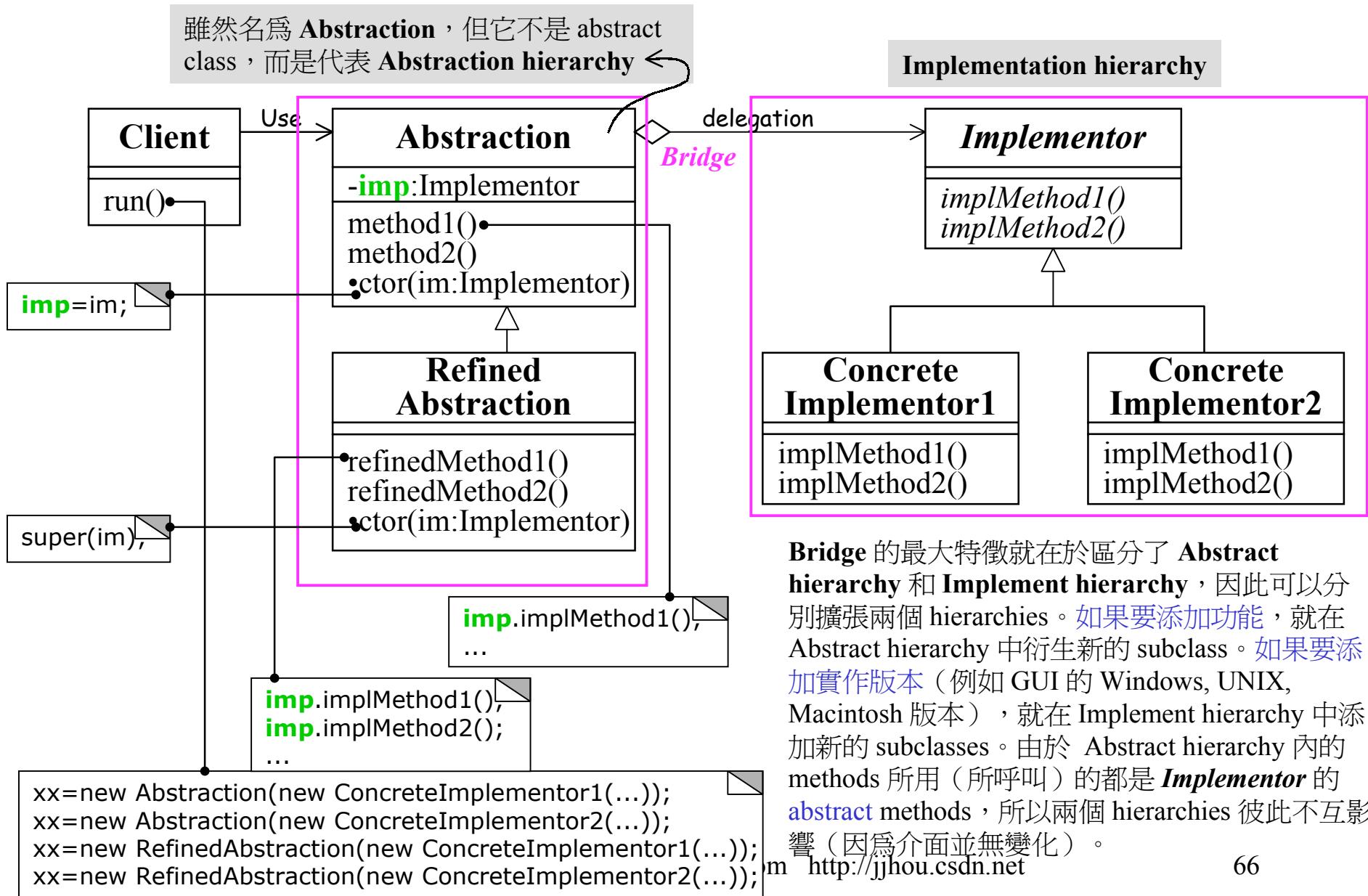
例如 subclass 要為 base class's *abstract method* 實作出函式定義。

Type

 ConcreteType1
 └ ConcreteType2

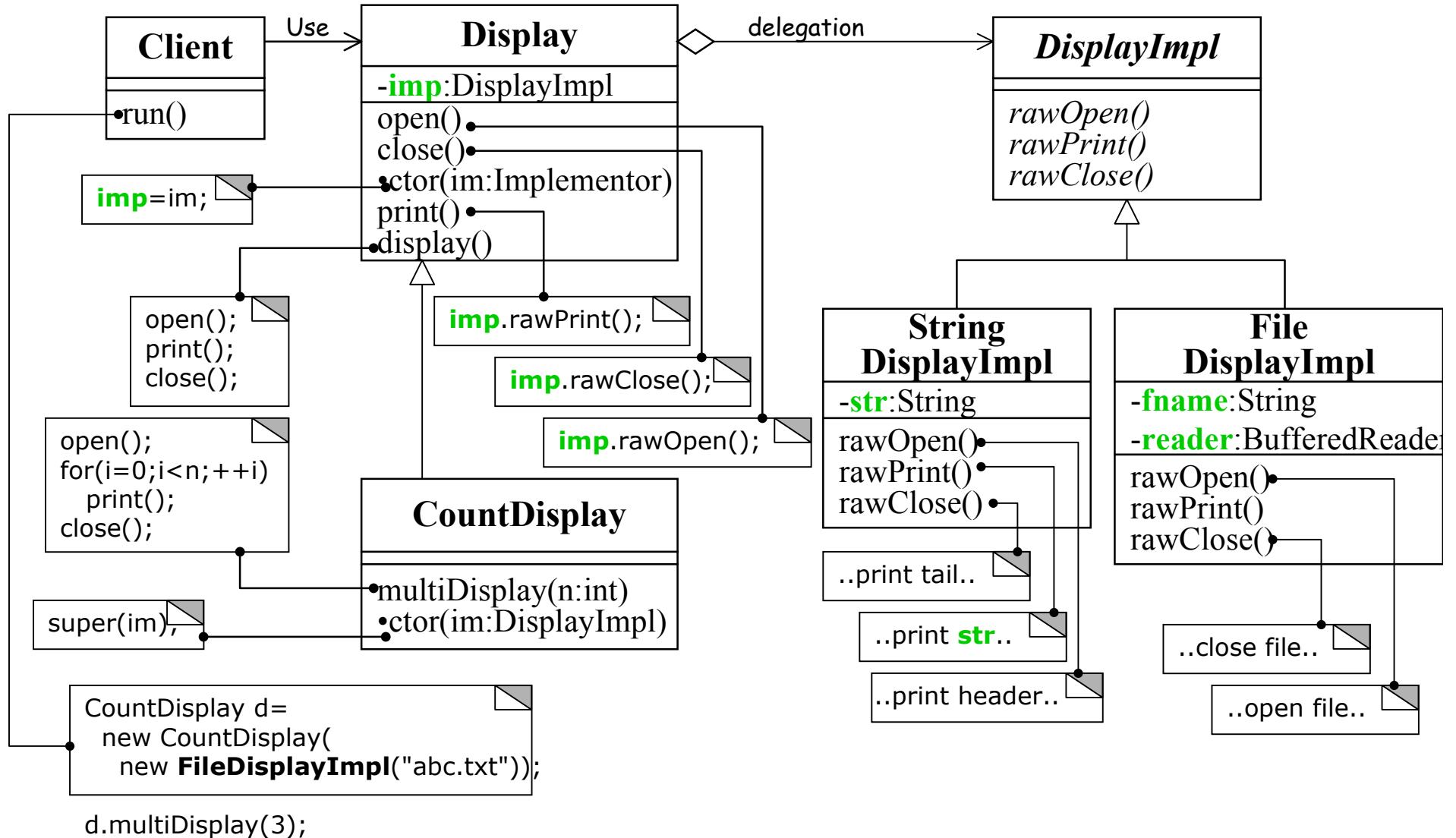


3. Bridge



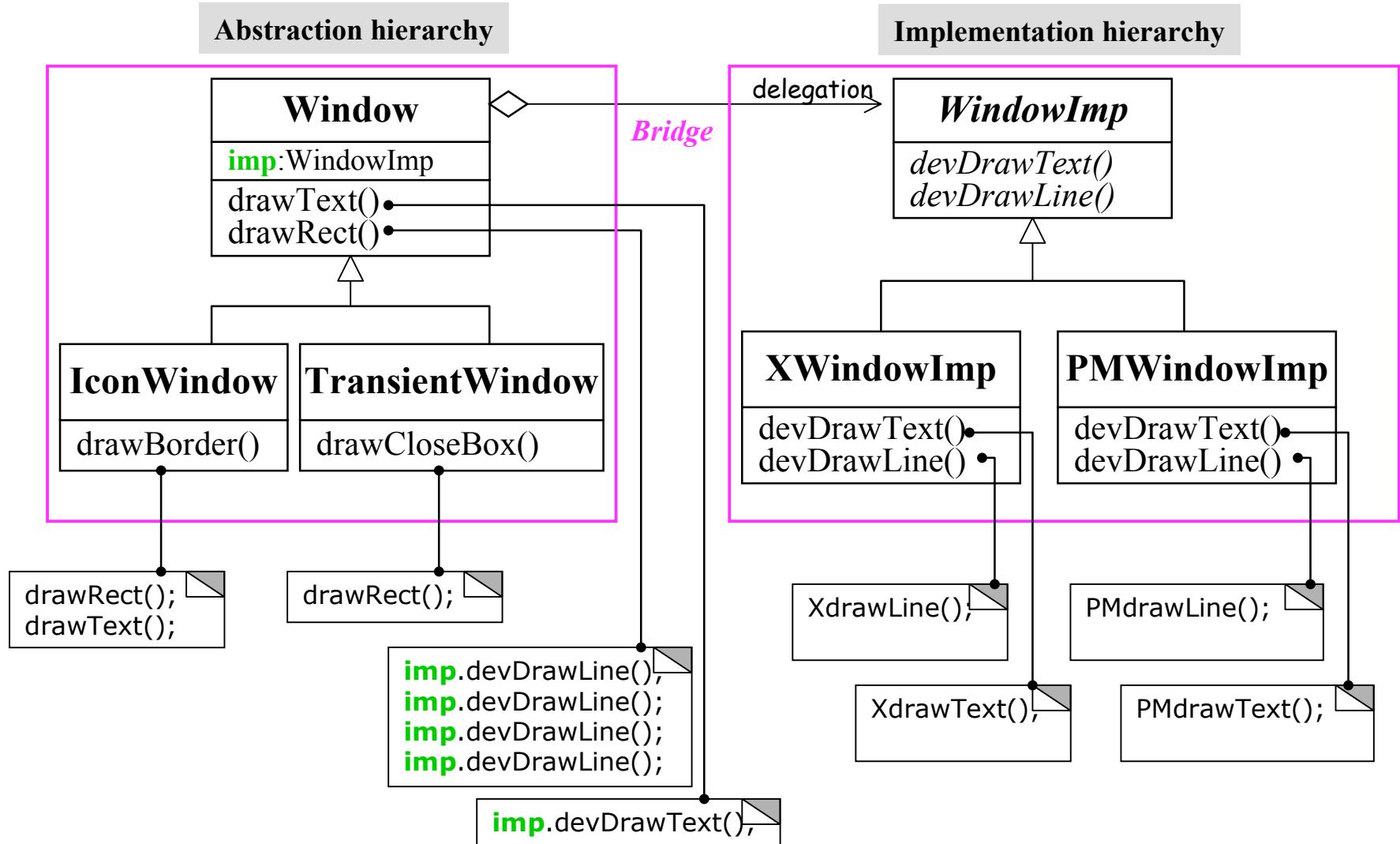


3. Bridge in DP-in-Java





3. Bridge in GoF





15. Memento in GoF

15. Memento (283)

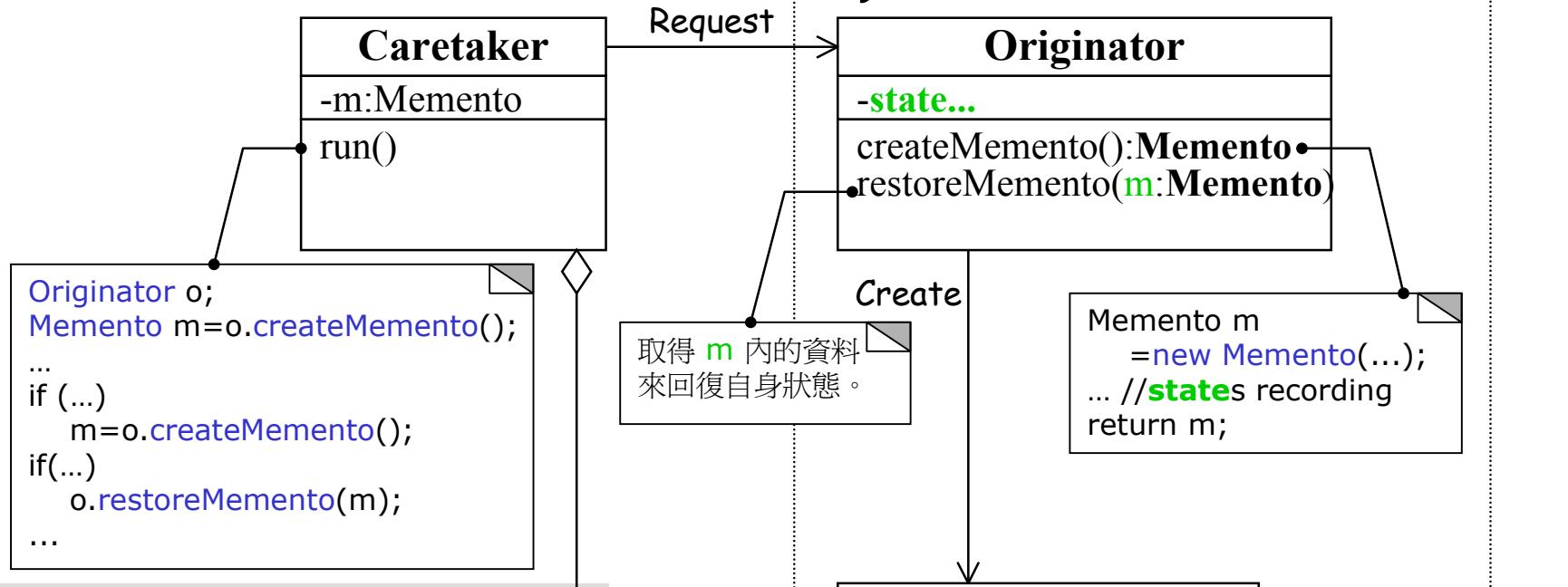
Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

在未違反封裝原則的前提下，捕捉 object 的內部狀態（資料）並賦予外形（使具體化 / 存在於外部），使該 object 日後得以恢復該狀態。



15. Memento in DP-in-Java

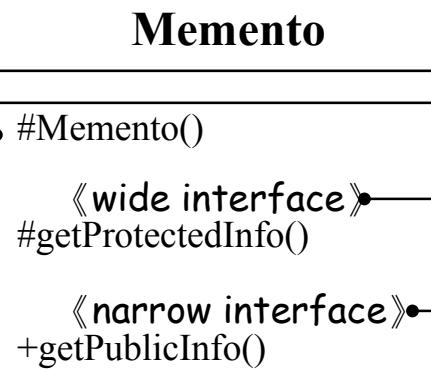
package/component



Caretaker 只能使用 **Memento** 提供的兩種介面中的 narrow interface，不能存取 **Memento** 的內部信息。它只會把別人（**Originator**）產生的 **Memento** 照單全收儲存起來，當作一個黑箱（block box;未知區域）

舉例：**Originator** 是個下注遊戲（亂數產生器、下注輸贏規則）而 **Caretaker** 負責玩的過程（何種情況下贏太多要記錄狀態，何種情況下輸太多要恢復先前狀態）。

#表示只有同一個 package 內的 class 才能生成 Memento object



會洩漏 Memento 的內部狀態，所以只有 Originator 可使用之。

不會洩漏 Memento 的內部狀態，是給 Caretaker 用的。

Caretaker 負責決定在哪個時間點做 snapshot（瞬間快照），何時復原，以及如何保留 **Memento**。
Originator 負責產生 **Memento** 以及利用 **Memento** 恢復自己的狀態。



17. Prototype

17. Prototype (117)

Specify the kinds of objects to create using a **prototypical instance**, and create new objects by copying this prototype.

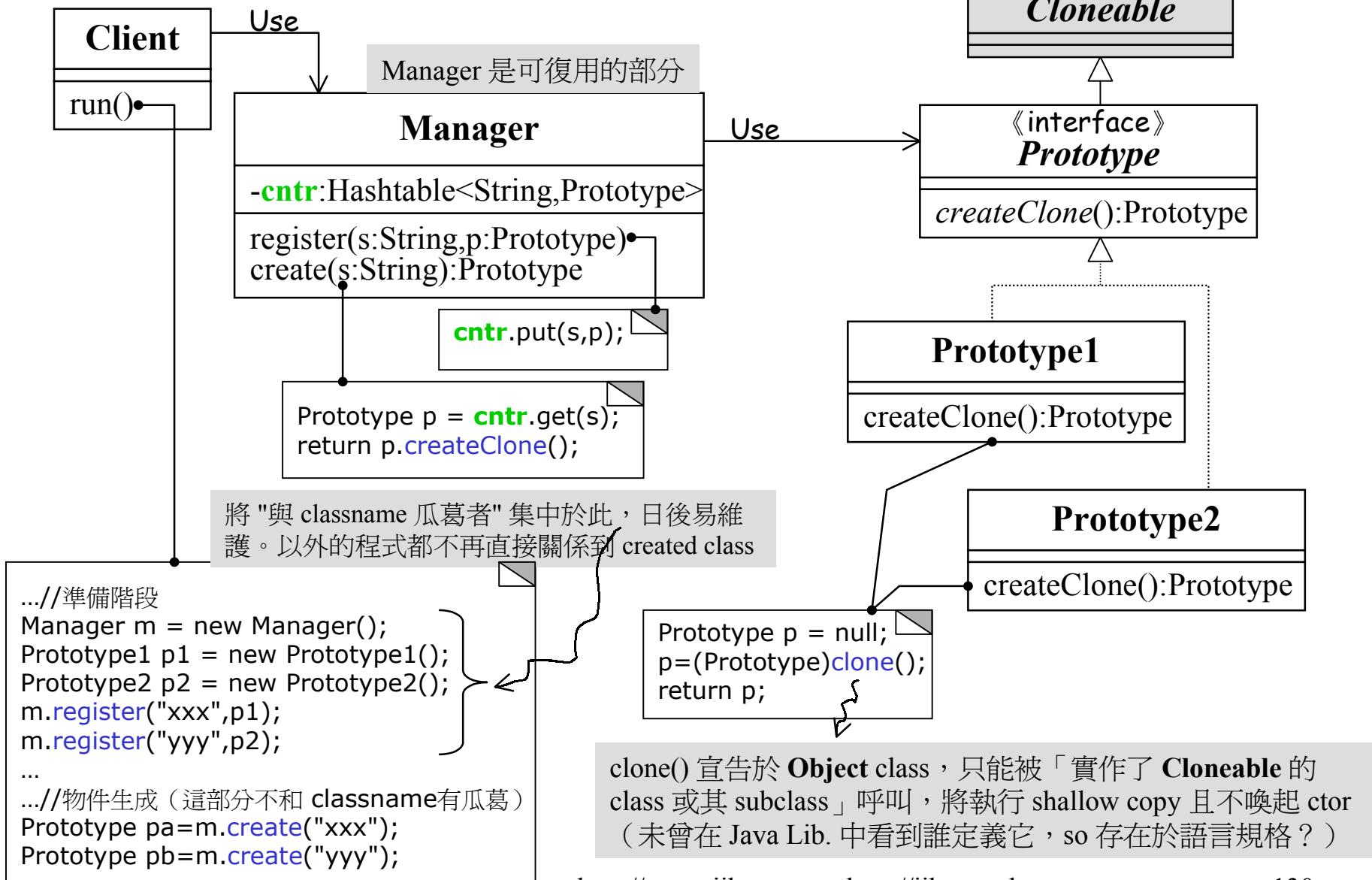
具體指定/登錄/註冊「可藉由**原型實體**生成」的 object 種類。日後便以「複製原型實體」的方式生成這種 objects（而不使用 new）。

•當不容易以 class 來產生 object 時“ 例如 object 是以人工操作方式繪製而成的一個圖形，這種情況下若要另建立一個相同的 object instance，使用 Prototype 會比較容易得多。

•如果想把「製造 objects 的框架」和「所產生的 objects」分開（低耦合），也適合使用 **Prototype**。也就是說不再 new *className*; 而改為
`m.create("className"); // m 是 Manager object，接受 string 而非 className`
難道程式中出現 class 名稱是一種束縛嗎？關鍵在於當寫下 class 名稱，這份 source code 就無法與該 class 分離供再利用了。在 **Prototype** 中 Manager 是可再利用的部分。



17. Prototype





5. Chain of Responsibility

5. Chain of Responsibility (223)

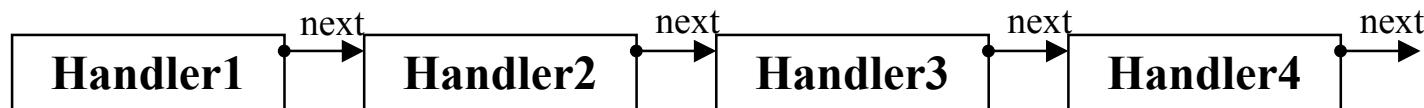
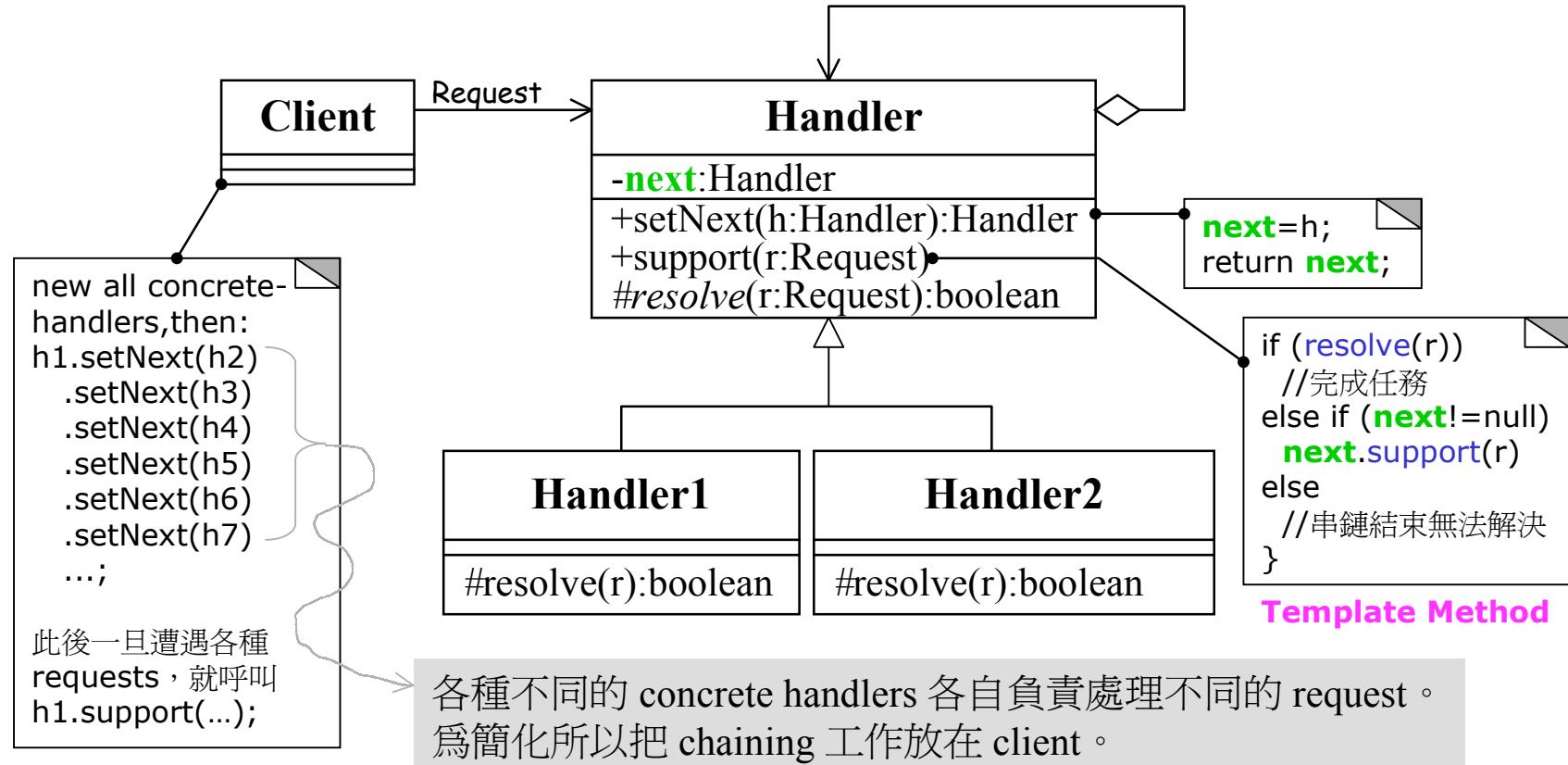
Avoid coupling the **sender** of a request to its **receiver** by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

給予一個以上的 object 處理 request 的機會，以避免 request sender 和 request receiver 之間耦合。此法將 receiving objects 串鏈起來並沿著這條鏈傳遞 request，直到其中某個 object 處理了它。



5. Chain of Responsibility

我們希望外界呼叫 support() 而非 resolve()，所以令前者為 + 後者為 #，那麼只要不是 subclass 或同一個 component 內的 classes，都不能呼叫 # method.





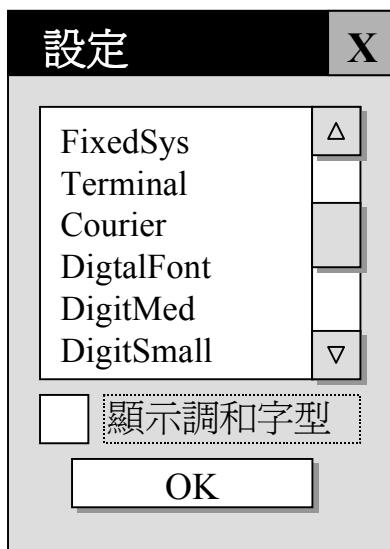
5. Chain of Responsibility

Chain of Responsibility 的重點在於讓「request 提出者」（Client）和「request 處理者」（ConcreteHandler）兩者之間的結合關係不要那麼緊密。

如果不是採用這個 pattern，就得採用中央集權方式，令某人掌握「哪個 request 該由誰來處理」的原則。

處理速度會比較慢，但這是 trade-off。

有些 window system 允許 user 在 window 上新增 GUI 組件（按鈕或文字輸入欄），這時 Chain of Responsibility 就能發揮功效。



當 active window 是 listbox 時，↑ ↓ 鍵可用來選擇字型。

當 active window 是 checkbox 時，↓ 鍵無作用而 ↑ 鍵會令 listbox 成為 active。

作法：↑ ↓ 鍵會以 event 型式送入 active window。Listbox 會自行處理這些 events，所以不再傳給 **next**。Checkbox 無法處理這些 events，所以只能傳給 **next** 也就是 dialog box，於是後者在收到 ↑ 鍵時 切換 active window。



23. Visitor

23. Visitor (331)

Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

用以表現一個「執行於某物件結構內之各元素身上」的操作。Visitor 讓你得以定義一個新操作而不需要改變其所操作之元素的所屬 classes。

Iterator 和 Visitor 都是在 Data Structure 身上進行處理。Iterator 用於「想逐一取得 Data Structure 的元素」時，Visitor 用於「想逐一對 Data Structure 的元素進行某種處理」時。Visitor 需要 Iterator 協助。



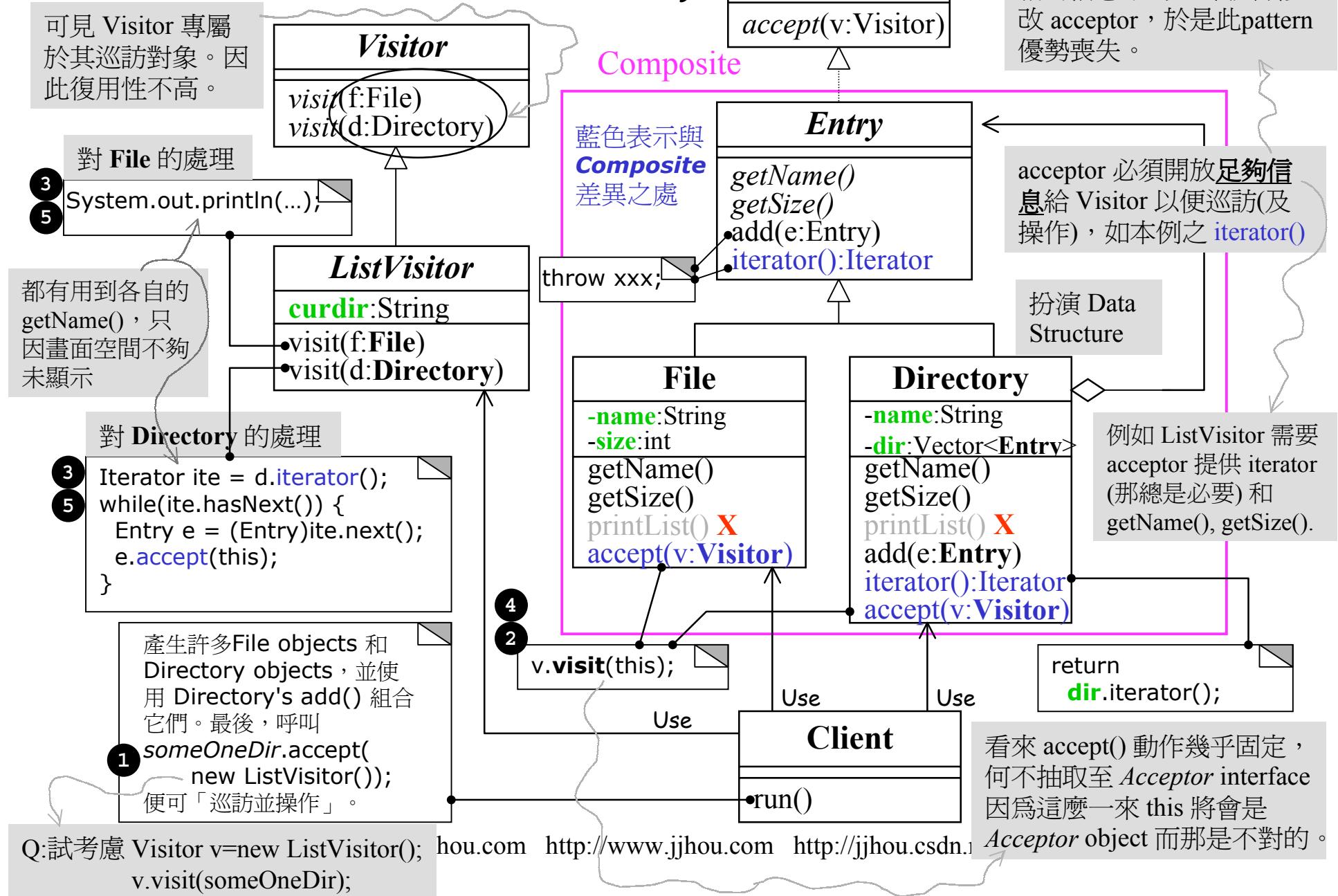
23. Visitor

Data Structure 內儲存了許多元素。假設現在要對所有元素進行某個 operation，那麼這個 operation 該寫在何處？按說應寫在 data structure class 內。於是每當設計新的 operation 就得修改 data structure class。**Visitor** 令 data structure 和 operations 兩者分開，在 data structure class 之外另寫一個可在 structure 內穿梭來去的所謂 **Visitor class**，讓 **Visitor** 負責 operations。爾後如果要添加新的 operations，只需建立新的 visitor subclass 即可。

拿 **Composite** 的例子為例。該例以 **Composite** 組織出 "Directories+Files" 之後呼叫 Directory's printList() 列印所有成員（亦即 directories 和 files 的名稱和大小）。彼時（未使用 **Visitor**）就是把 printList() 分別寫在 **File class** 和 **Directory class** 內。如果使用 **Visitor**（如下頁圖），則是把 printList() 功能寫成一個 **ListVisitor**。這麼一來，將來若要再加一個 operation，就再加一個 **concrete Visitor** subclass 即可，資料結構（Directory 和 File）那一部分完全不改動。

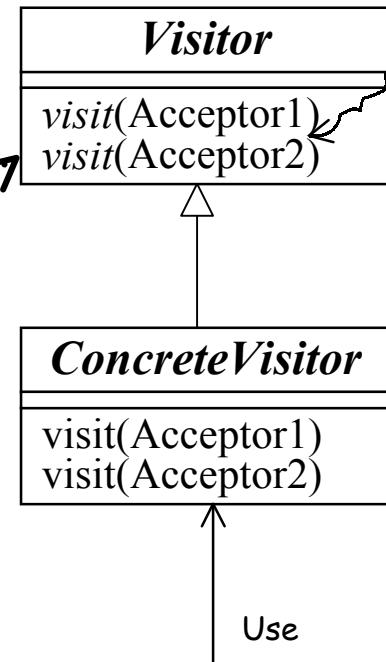
23. Visitor in DP-in-Jav

可見 Visitor 專屬於其巡訪對象。因此復用性不高。

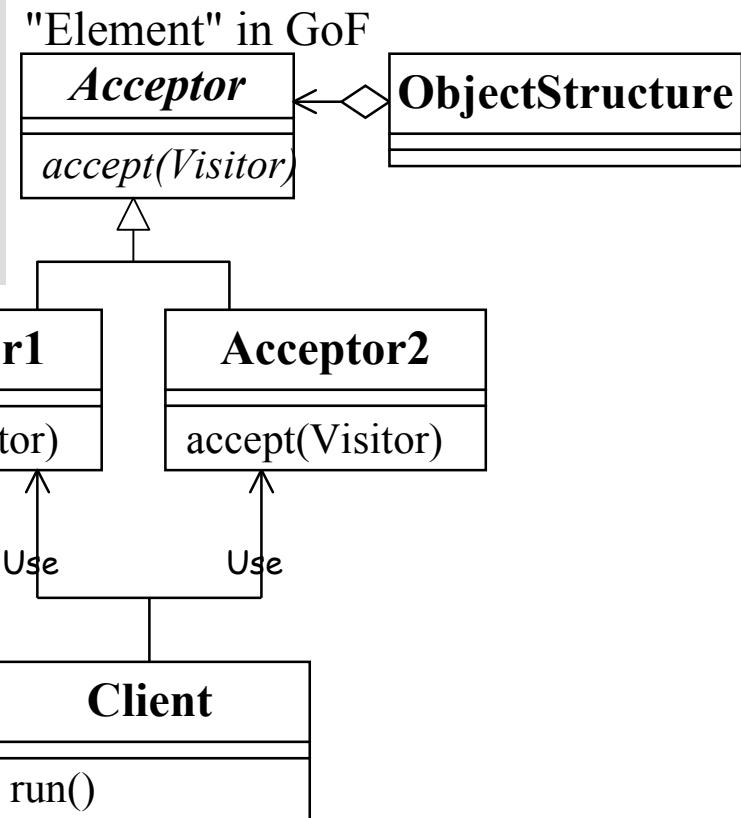




23. Visitor in DP-in-Java



相依於 concrete class. 可見 Visitor 專屬於其巡訪對象；復用性不高。就算設計為 `visit(Acceptor)`; 也是必須在內部區分是哪種concrete acceptor，功並沒有少做。你倒也不可能設計使 `visit(Acceptor)` 做的動作通用於各種 concrete acceptor，那似乎是不可能的。



Visitor pattern 的目的是把 operation 從 Data Structure 中分離出來。畢竟「維護一個結構」和建立一個「以此結構為基礎的 operation」是兩回事。除非有特殊理由，否則設計 classes 時都應該容許日後擴充，但應該禁止被修改。這也就是 Open-Closed Principle。因此欲新增一個 operation，只需增加一個 **ConcreteVisitor** 並做為 client 端呼叫 `accept()` 時的參數即可。但若欲添加一個 concrete **Acceptor** 就不是那麼容易，需要改動 **Visitor** 為它添加一個 `visit(Acceptor3)`。