資訊軟體人才培育推廣計畫

設計樣式 Part I

設計樣式 Part I

設計樣式簡介

薛念林 逢甲大學資工系

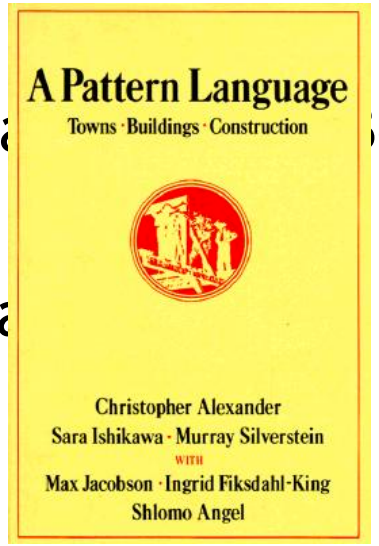
設計樣式 (Design Pattern)

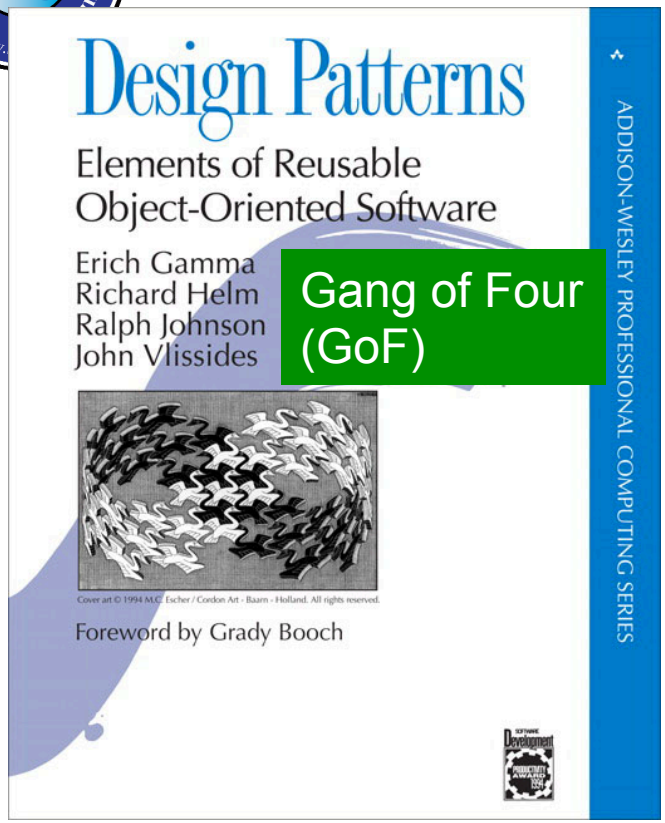
將有經驗的軟體工程師的設計經驗與技巧 抽鍊、整理、包裝成特定的框架, 用以解決系統設計上的問題，提昇系統的品質。

- 例如擴充性問題、維護性問題與重用性問題等

起源

- Architectural concept by Christopher Alexander (1977)
- Pattern La eck Kent and Cunningha





Pattern Languages of Programming Conference (PLoP), since 1994.

範圍 (scope)

用途

	類別	物件
生成	將部分物件的生成延遲到子類別決定。樣式：抽象工廠	將部分物件的生成委託給其他物件生成。樣式: 建築者、工廠方法、雛形、單例。
結構	使用繼承來組合介面或是實作。樣式：轉接器	使用物件組合實現新的功能及彈性，使得執行時可以更改物件間的組合。樣式：橋接器、複合、裝飾、外觀、輕量化、代理人
行為	使用繼承來分配類別間的行為。樣式：解譯、樣板方法	採用物件間的組合而非繼承，描述一群對等的物件如何協同合作以完成工作。樣式：責任鏈、命令、策略、訪問、覆迴、調停、紀念品、觀察者、狀態

23 個樣式 生成

- **抽象工廠 (Abstract Factory)** 。在不需要指定明確的類別下,提供一個介面以建立一群相關的物件。因此,當系統預建立新的一群物件時, 不需要改變既有的程式碼,只需擴充原來的類別即可。
- **建築者 (Builder)** 。將一個複雜物件的建構與其表達分開,藉此,相 同的建構程序可以用在不同的表達上,提供擴充上的彈性。
- **工廠方法 (Factory Method)**。定義一介面以生成物件,但將其生成延 遲給子類別來作決定。
- **雛形 (Prototype)** 。當直接生成物件的成本過高時,利用複製現有雛型實例的方式建立物件,而非採用生成的方法。
- **單例 (Singleton)**。確保一個類別只會生成單一的物件。

23 個樣式 結構

- **轉接器 (Adapter)**。在不修改既有介面的情況下將一介面轉成另一個 介面,藉以整合不同的物件。
- **橋接 (Bridge)**。將介面與實作分離,藉以提供介面與實作組合的多樣 性。
- **複合 (Composite)**。將物件組合成樹狀的結構並同時具備部分-全部的 包含關係。組合的結構讓客端的物件以相同的介面來看待個別物件與 複合物件,藉此簡化客端物件與服務端物件的的耦合力。

23 個樣式

- 裝飾品 (Decorator)。動態的增加物件的功能。相對於用繼承的方式來 擴充功能,裝飾品提供更彈性的方法來擴充物件的功能。
- 門戶 (Façade) 。為一個子系統內眾多的服務提供一個統一的介面,藉 以降低子系統間的耦合力。
- 輕量 (Flyweight)。使用分享的方式來協助有效的管理輕量級物件的資源。
- 代理人 (Proxy)。為某一物件提供一個中介控制的介面,以過濾對該 物件的取。

23 個樣式 行為

- **責任鏈 (Chain of Responsibility)**。避免將一個要求的提出者與接受者 直接連結以降低他們之間的耦合力。責任鏈允許多個物件處理一個相同的要求。要求會在責任鏈中傳遞直至真正可以處理該要求的物件。
- **命令 (Command)**。將物件的需求封裝為一個類別,藉此提供更彈性 的操作。例如將請求作排隊處理 (queue) 及提供請求回覆 (undo) 的功 能。
- **解析器 (Interpreter)** 。針對一個語言,提供該語言文法的表達法,以便於解析該語言內的結構與子句。
- **覆迴 (Iterator)** 。提供一個較安全的方式以循序性的取複合物件的內容 - 取者不會知曉複合物件的內部的細節。

23 個樣式

- **調停者 (Mediator)** 。將物件的互動封裝為一個物件,藉以降低這群務間之件的耦合力。
- **紀念品 (Memento)**。在不破壞封裝性的前提下,紀錄並外顯化物件的 狀態,以便該物件在之後可以回覆該狀態。
- **觀察者 (Observer)**。當一群物件間有一對多的相依關係時,當被依者 物件的資料改變時,會通知其他依靠者物件以作出回應。
- **狀態 (State)**。將物件的狀態自物件本身獨立出來,以提高物件行為變 化的彈性。

23 個樣式

- **策略 (Strategy)**。將演算法自其使用者中獨立出來,藉此提高該演算 法使用上的彈性。亦即,演算法的使用者可以在不修改自身程式的情 況下更換演算法。
- **樣板方法 (Template Method)**。定義一個方法演算法的結構為若干個步 驟的組合,但將每個步驟的真實演算法延遲到子類別定義,藉此提高 演算法變化的彈性。
- **拜訪者 (Visitor)** 。將方法自其會運作的物件中獨立出來,藉此,避免新增方法時對該物件結構作的改變。

結構

- 目的: 該設計樣式所預期達到之目標。
- 應用時機: 該設計樣式的適用時機、環境與限制。
- 結構: 該設計樣式之組成物件及他們之間的關係。
- 範例程式

Java, Python, C++, etc.

設計樣式 Part I

Strategy

薛念林 逢甲大學資工系

動機

演算法（策略）是經常「變動」的

- 排序、資料加密、排版、最短路徑 …

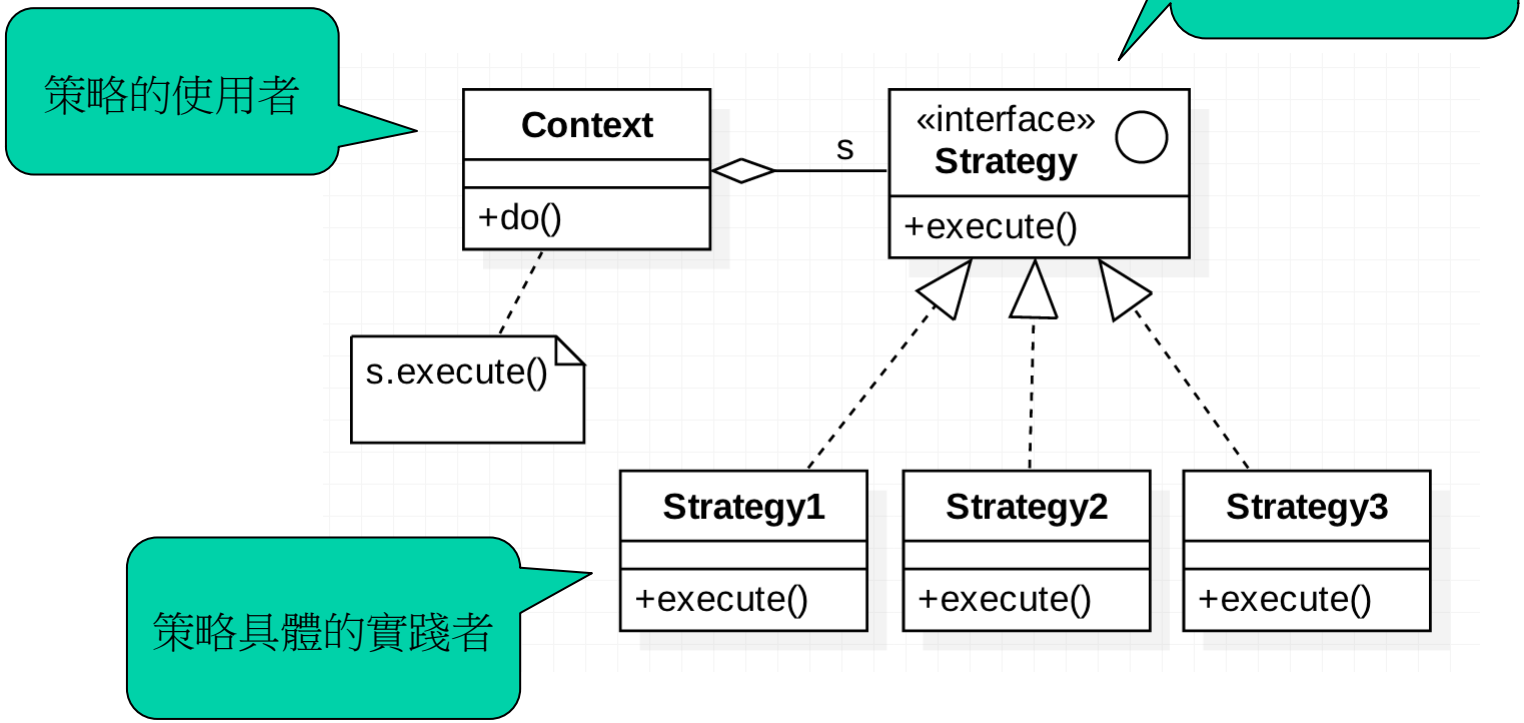
OCP 原則：不要因為採取不同演算法修改主程式

目的（Intent）

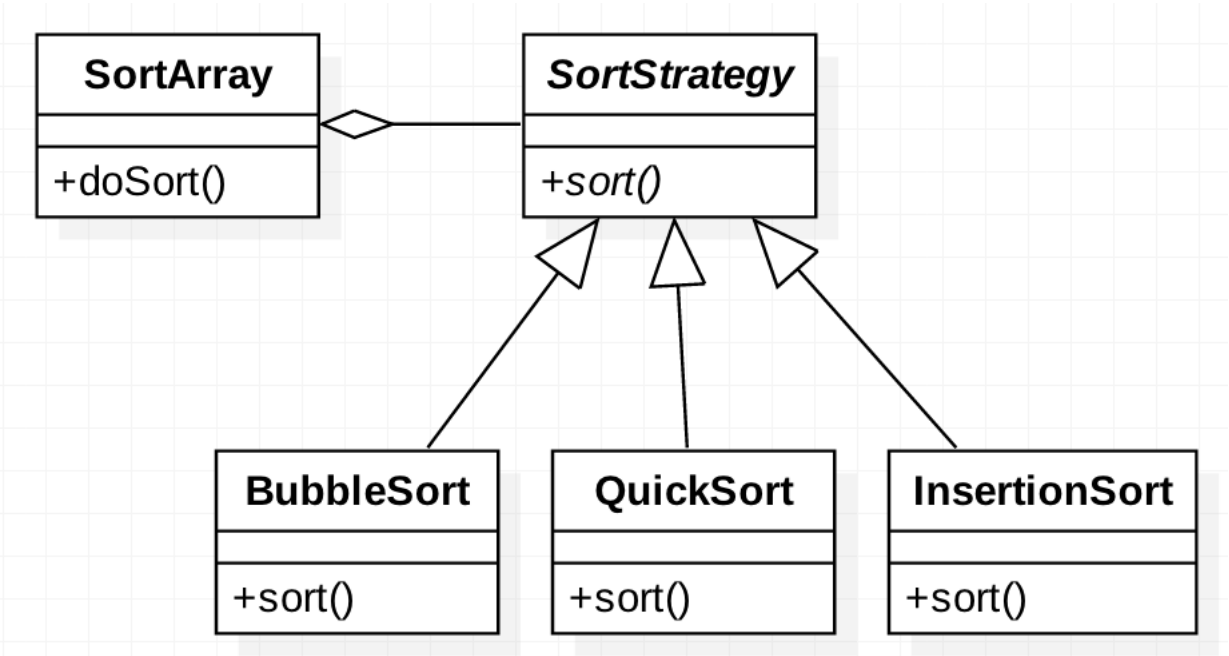
定義一群演算法，將每一個封裝成一個類別且使之可互換。使用 Strategy 讓演算法獨立於使用者。

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

結構



範例：Sort




```
1  interface SortStrategy {
2      public int[] sort(int [] d);
3  }
4
5  class SortArray {
6      int[] d;
7      private SortStrategy sortStrategy;
8      public SortArray(SortStrategy s) {
9          this.sortStrategy = s;
10     }
11     public int[] doSort() {
12         return this.sortStrategy.sort(d);
13     }
14 }
```

抽象策略

包含一個策略物件

設定（綁定）策略

委託 (delegation)

```
15
16  class QuickSort implements SortStrategy {
17      public int[] sort(int[] d) {
18          //do ....
19          return ...
20      }
21  }
22
23  class SelectionSort implements SortStrategy {
24      public int[] sort(int[] d) {
25          //do ....
26          return ...
27      }
28  }
29
30  //main program to test
```

實作

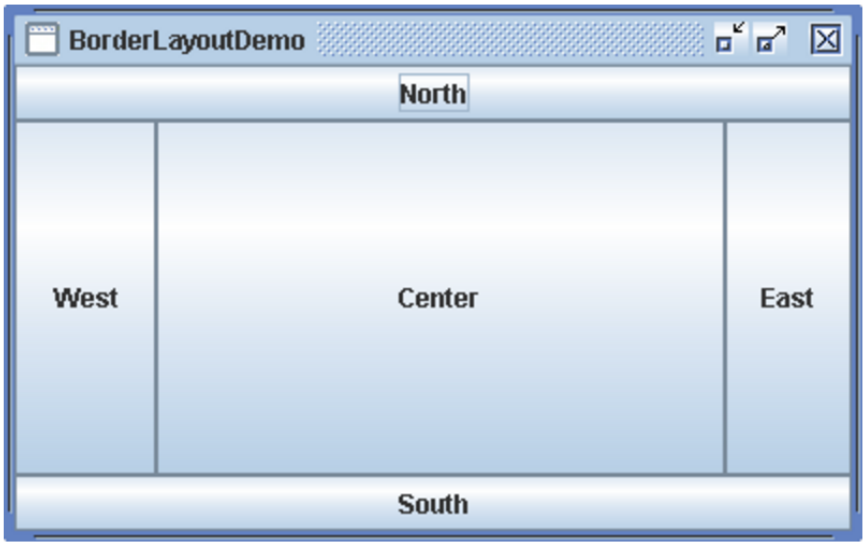
具體策略

具體策略

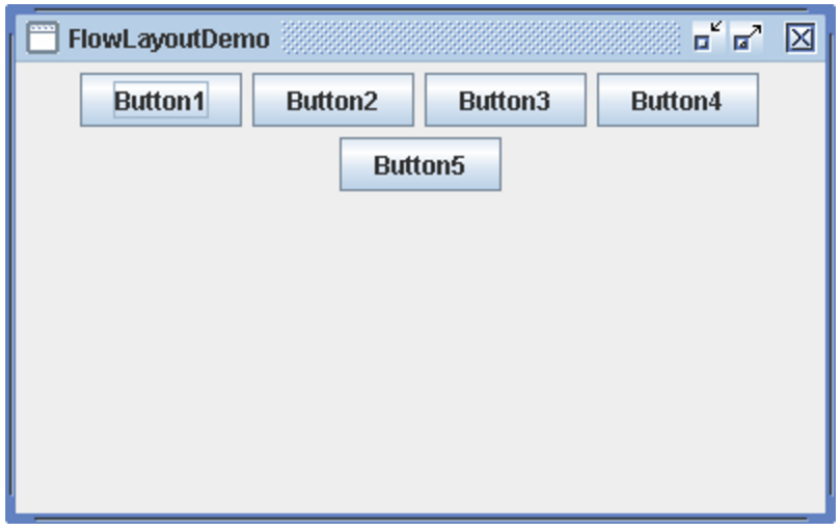

```
31  class StrategyExample {
32      public static void main(String[] args) {
33          SortArray context;
34          context = new SortArray(new QuickSort());
35          int[] resultA = context.doSort();
36
37          context = new SortArray(new SelectionSort());
38          int[] resultB = context.doSort();
39      }
40  }
```

應用一: Java Layout

Strategy



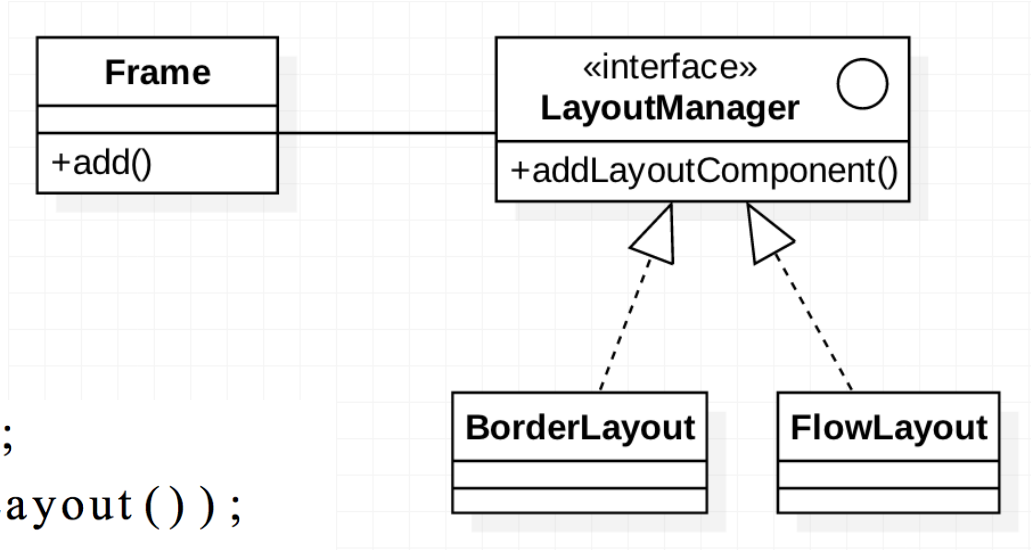
Borderlayout



Flowlayout

Layout

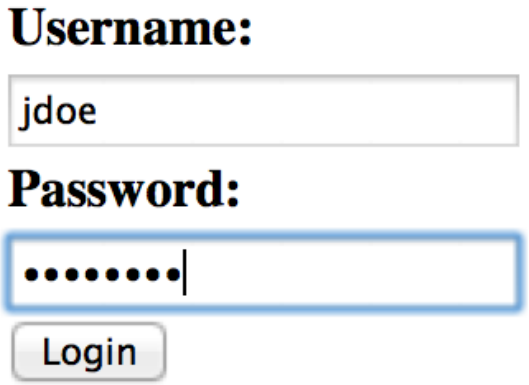
Strategy

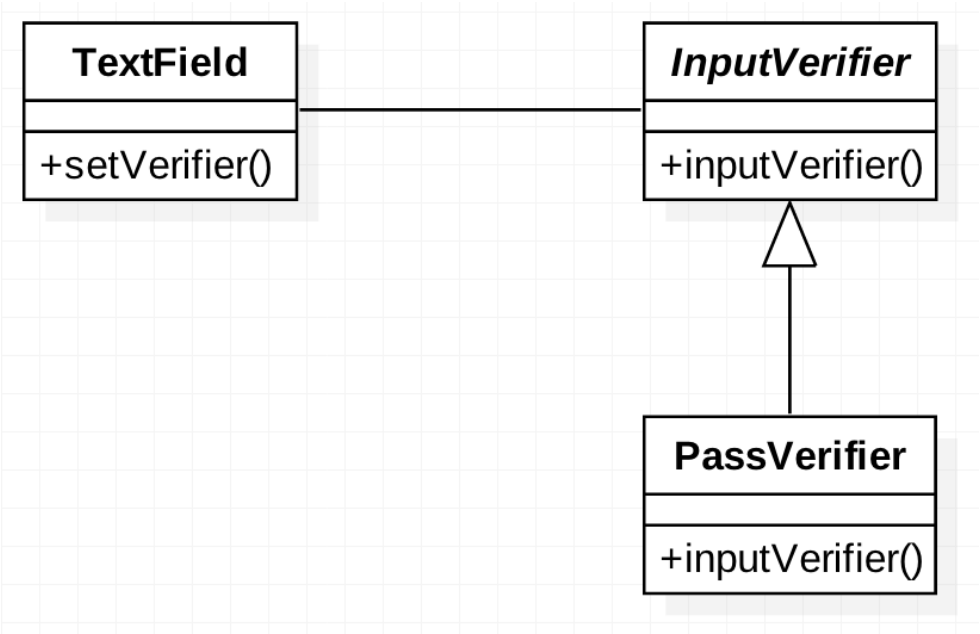



```
1    Frame f = new Frame();
2    f.setLayout(new FlowLayout());
3    f.add(new Button("Press"));
4
5    f.setLayout(new BorderLayout());
6    ...
```

應用二: InputVerifier

Strategy





```
1  public class VerifierTest extends JFrame {
2      public VerifierTest() {
3          JTextField tf1 = new JTextField ("Type␣\"pass\"
             ␣here");
4          getContentPane().add (tf1, BorderLayout.NORTH);
           tf1.setInputVerifier(new PassVerifier());

18     class PassVerifier extends InputVerifier {
19         public boolean verify(JComponent input) {
               JTextField tf = (JTextField) input;
               return "pass".equals(tf.getText());
22         }
23     }
```

設定檢查策略

具體策略

設計樣式 Part I

Factory Method

薛念林 逢甲大學資工系

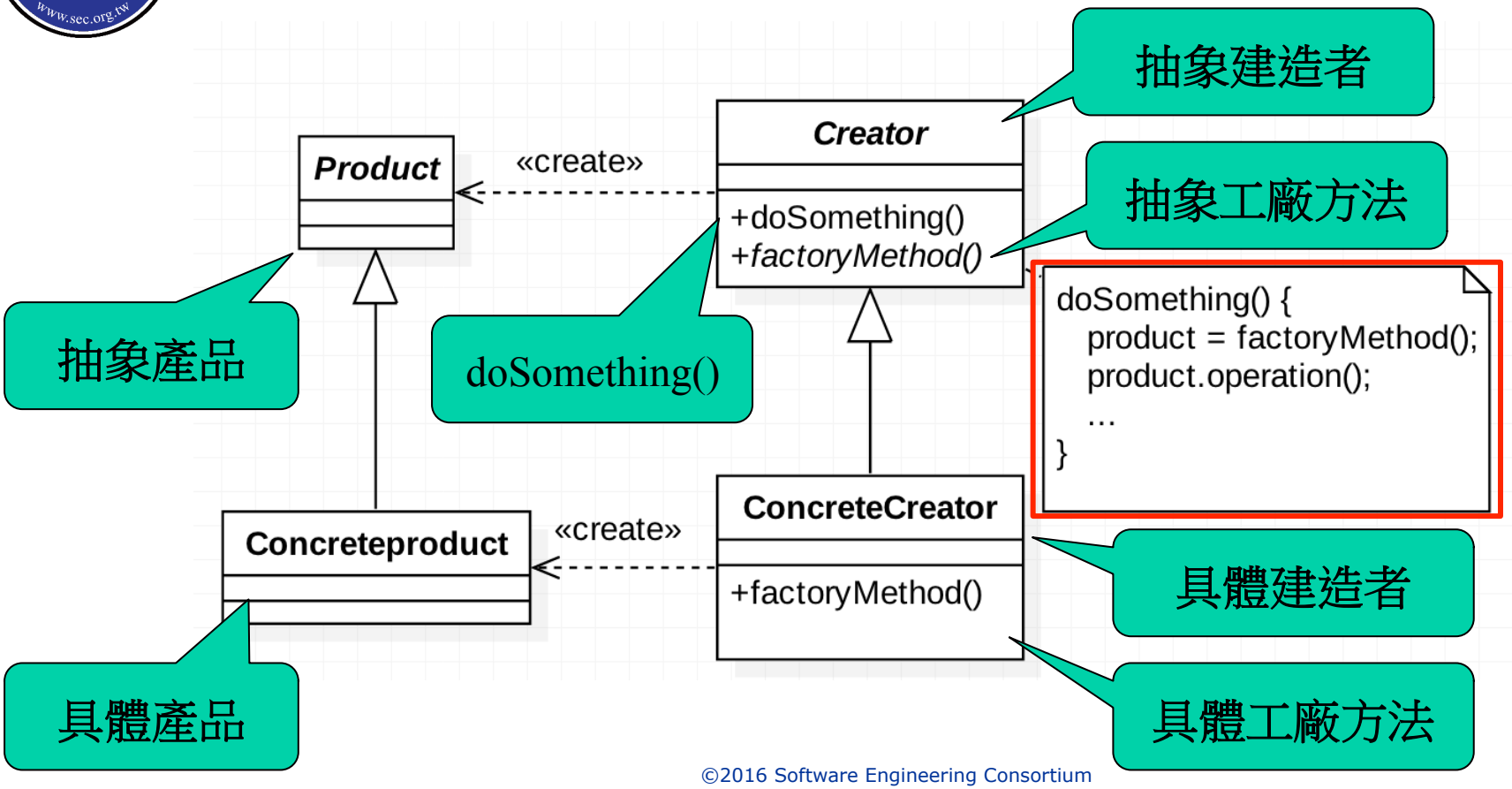
資訊軟體人才培育推廣計畫

Factory Method

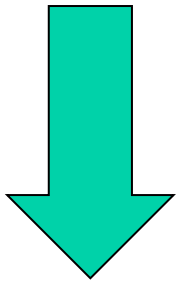
定義一介面以生成物件, 但將其生成**延遲給子類別**來作決定。

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

結構



product = new Product()



product = makeProduct()

動機

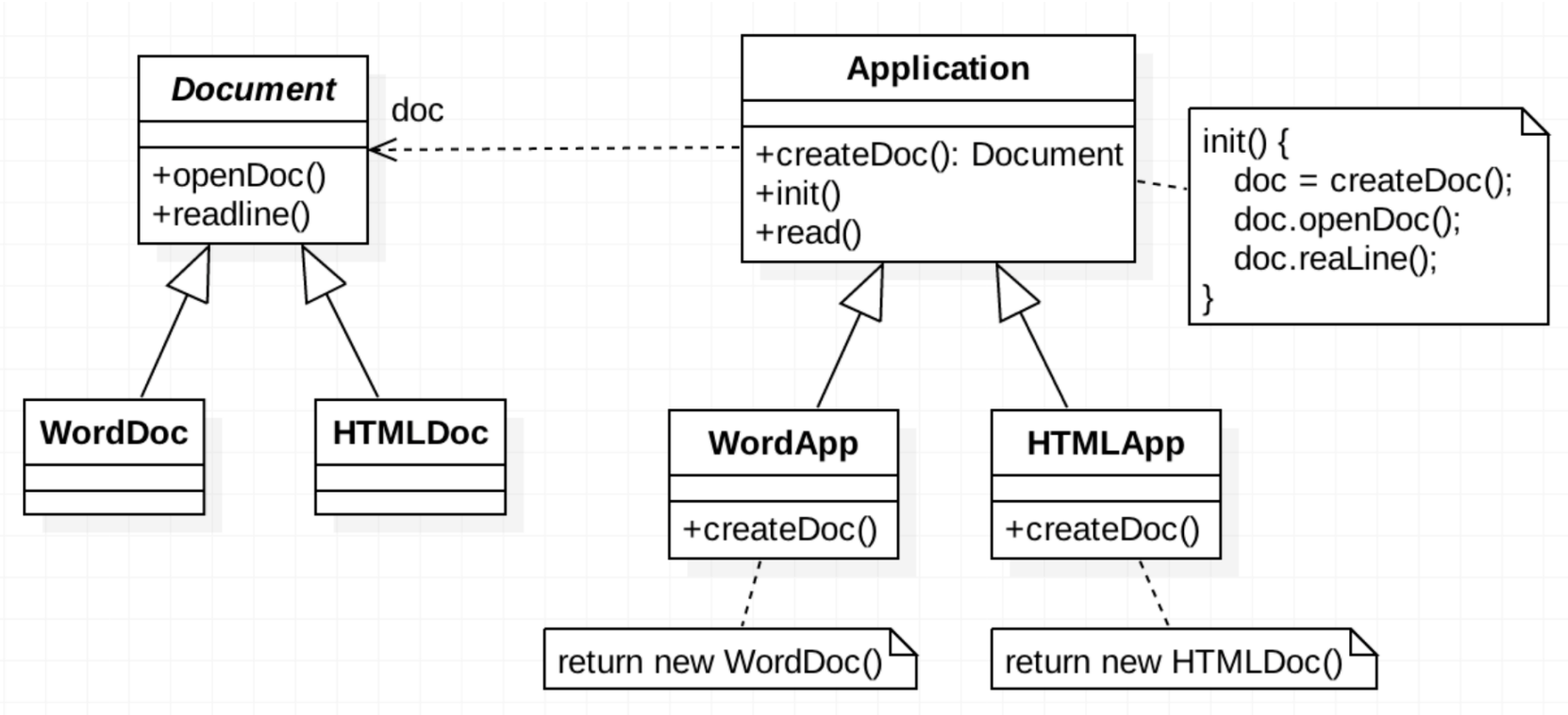
假設某一個 Application 物件可以產生 Document 物件以供其使用完成 Application 物件的工作。如果直接在 Application 某方法內生成文件物件

```
1    class Application {
2        void operation1() {
3            doc = new Document();
4        }
5    }
```

日後 Application 物件想建立不同型態的文件物件 (例如 HTML 文件、 Word 文件) 時,則必須修改 operation1() 方法如下

doc = new HTMLDocument();

使用 Factory method




```
3   abstract class Creator {
4       Product p;
5
6       public void doSomething() {
7           p = factoryMethod();
8           // ...
9       }
10
11      public abstract Product factoryMethod();
12  }
```

抽象建造者

產生物件

工廠方法是抽象的

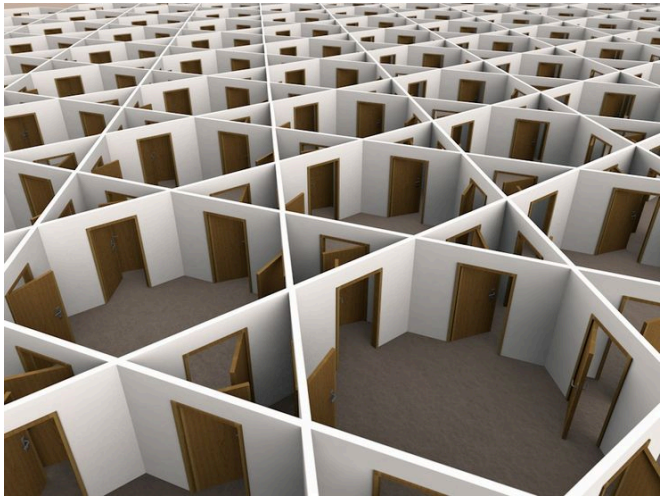

```
14  class ConcreteCreator extends Creator {
15      public Product factoryMethod() {
16          return new ConcreteProduct();
17      }
18  }
19
20  abstract class Product {
21      abstract void operation();
22  }
23
24  class ConcreteProduct extends Product {
25      void operation() {
26          // ...
27      }
28  }
```

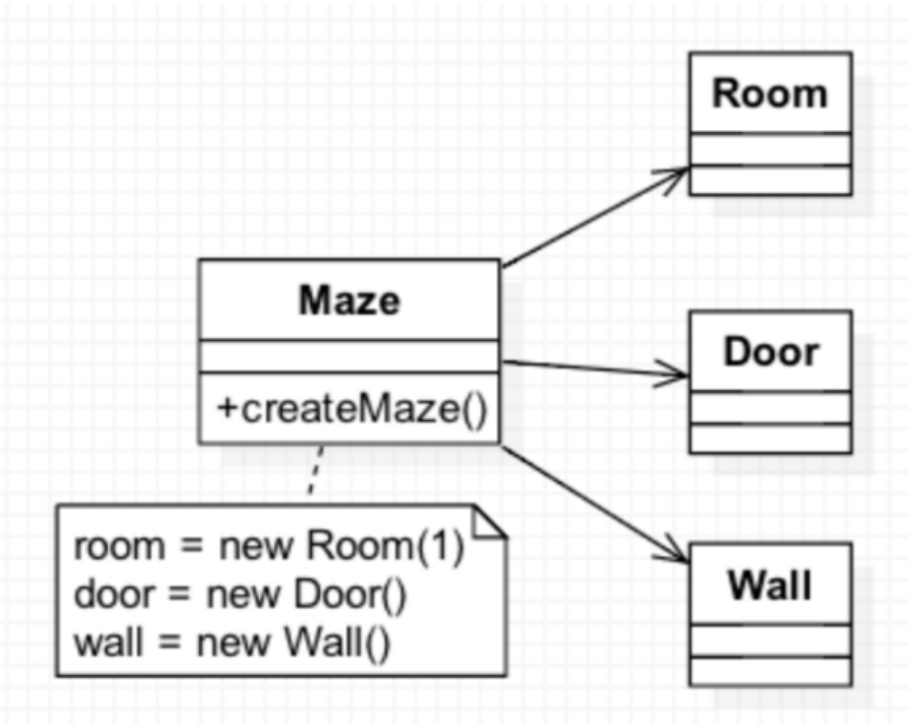
指定具體要產生的物件

抽象的，指定產品物件必要的功能

具體的產品物件

應用一



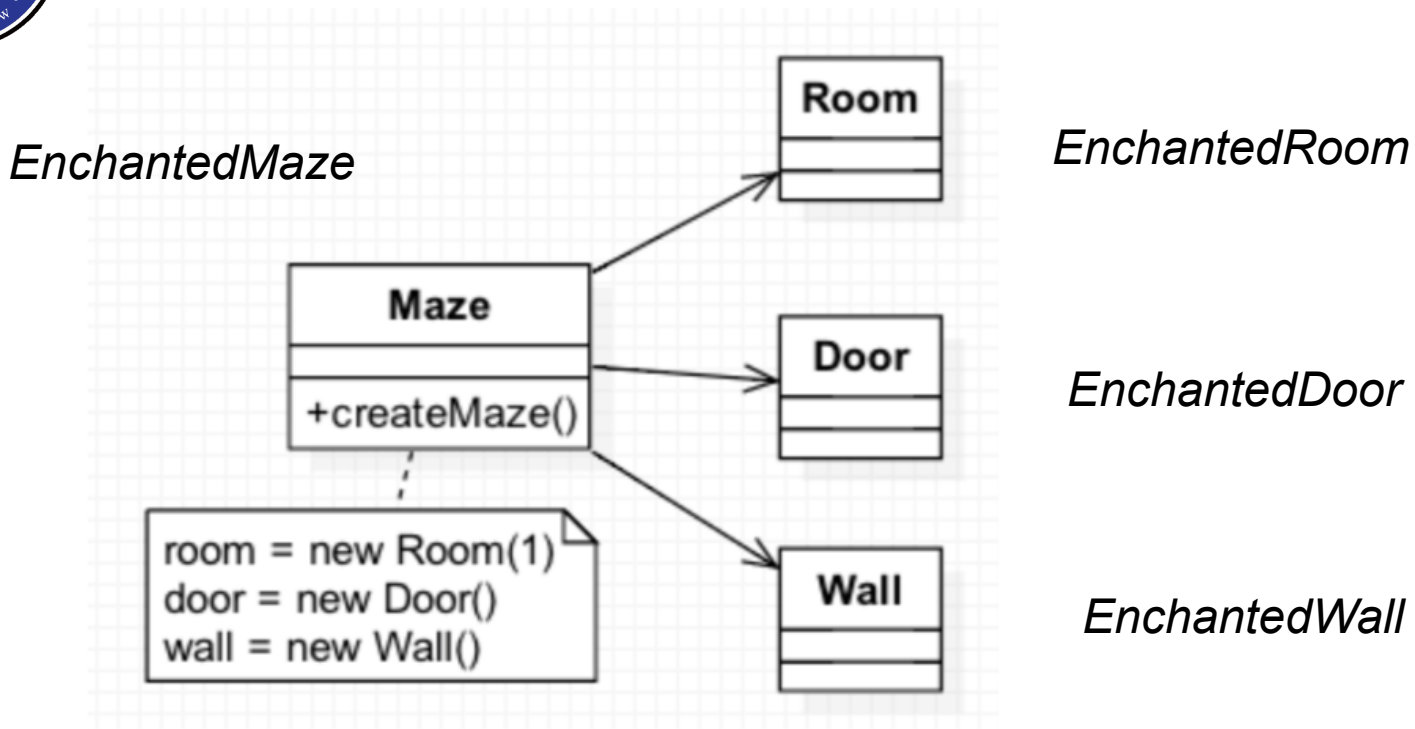


應用一

Solution 1: 未使用設計樣式

```
1  public class MazeGame {
2      // 建立一個迷宮
3      public Maze createMaze() {
4          //產生建立迷宮遊戲的所需零件物件，包括 Maze、Door、Room 等
5          Maze maze = new Maze();
6          Room r1 = new Room(1);
7          Room r2 = new Room(2);
8          Door door = new Door(r1, r2);
```


```
9           //建立各子物件之間的關聯
10          maze.addRoom(r1);
11          maze.addRoom(r2);r1.setSide(MazeGame.North, new
              Wall());
12          r1.setSide(MazeGame.East, door);
13          r1.setSide(MazeGame.South, new Wall());
14          r1.setSide(MazeGame.West, new Wall());
15          r2.setSide(MazeGame.North, new Wall());
16          r2.setSide(MazeGame.East, new Wall());
17          r2.setSide(MazeGame.South, new Wall());
18          r2.setSide(MazeGame.West, door);
19          return maze;
20      }
21  }
```



採用 Factory Method

Solution 2: 使用設計樣式

```
1 public class MazeGame {
2     // 建立一個迷宮
3     public Maze createMaze() {
4         //產生建立迷宮遊戲的所需零件物件，包括 Maze、Door、Room 等
5         Maze maze = makeMaze(); // 採用工廠方法
6         Room r1 = makeRoom(1); // 採用工廠方法
7         Room r2 = makeRoom(2); // 採用工廠方法
8         Door door = makeDoor(r1, r2); //採用工廠方法
9         //建立各子物件之間的關聯
```


```
21      public Maze makeMaze() {
22          //將生產 Maze 物件的工作抽象成一個方法
23          return new Maze();
24      }
25      public Room makeRoom(int n) {
26          //將生產 Room 物件的工作抽象成一個方法
27          return new Room(n);
28      }
```

EnchantedMazeGame

```
1   public class EnchantedMazeGame extends MazeGame {
2       public Room makeRoom(int n) {
3           return new EnchantedRoom(n);
4       }
5       public Wall makeWall() {
6           return new EnchantedWall();
7       }
8       public Door makeDoor(Room r1, Room r2)  {
9           return new EnchantedDoor(r1, r2);
10      }
```

設計樣式 Part I

Observer

薛念林 逢甲大學資工系

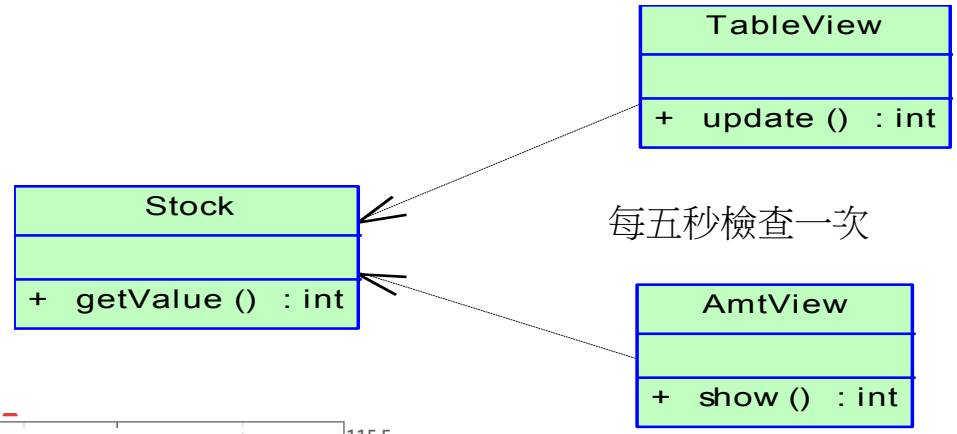
資訊軟體人才培育推廣計畫

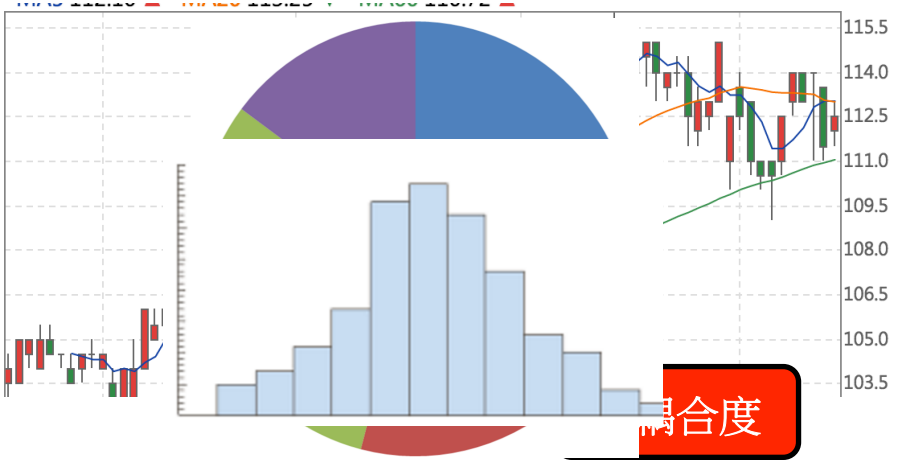
Observer 觀察者

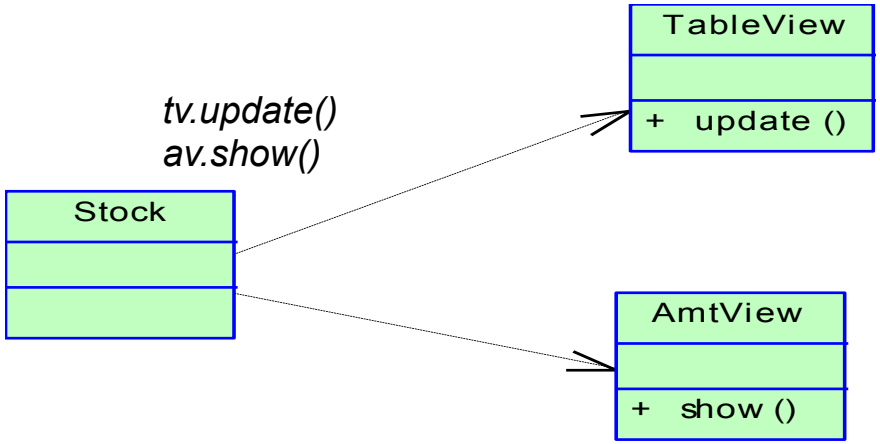
目的: 定義一個「一對多」的相依關係, 使得當一個主體物件狀態改變時, 所有相依的觀察者物件會被自動通知到並作適當的修改。

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

動機







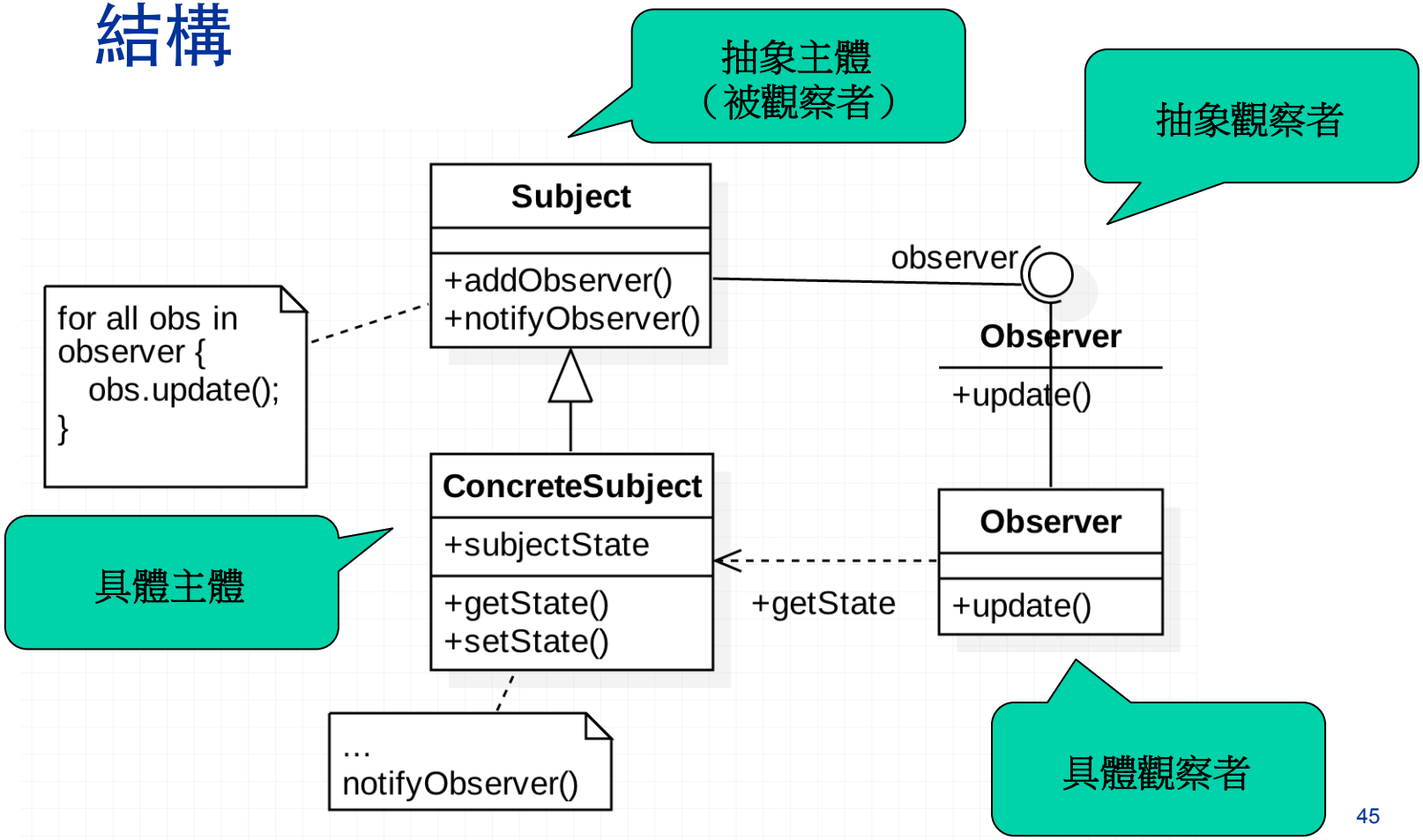
主體物件要通知觀察者物件

耦合度要低

- 當「多」的一端（觀察者）有增加減少時，「一」（主體）的一端不需修改

- 觀察者不需知道主體的細節

結構



Sample Code

```
1  package observer;
2
3  import java.util.Observable;
4
5  public class ObserverTemplate {
6
7      public static void main(String[] args) {
8          Subject s = new Subject();
9
10         View1 v1 = new View1();
11         View2 v2 = new View2();
12         s.addObserver(v1);
13         s.addObserver(v2);
```

Java API

主體物件

觀察者物件

註冊，建立關係


```
14        }
15
16    }
17
18    class Subject extends java.util.Observable {
19        int data;
20
21        public Subject() {
22            data = 0;
23        }
24
25        public void setData(int newValue) {
26            data = newValue;
27            this.setChanged();
28            this.notifyObservers();
29        }
```

具體主體

抽象主體

主體內的資料

不用指定被通知的物件

通知觀察者


```
30
31  }
32
33  class View1 implements java.util.Observer {
34      public void update(Observable arg0, Object arg1) {
35          // update the view
36      }
37  }
38
39  class View2 implements java.util.Observer {
40      public void update(Observable o, Object arg) {
41          // update the view
42      }
43  }
```

觀察者

抽象觀察者

主體物件

變更後的值

觀察者

抽象觀察者


```
public void setPrice(float price) {
    this.price = price;
    setChanged();
    notifyObservers(new Float(price));
}
```

Subject

Observer

```
public void update(Observable obj, Object arg) {
    if (arg instanceof Float) {
        price = ((Float)arg).floatValue();
        System.out.println("水果價格變成" + price);
    }
}
```

Observable 是怎麼設計的？

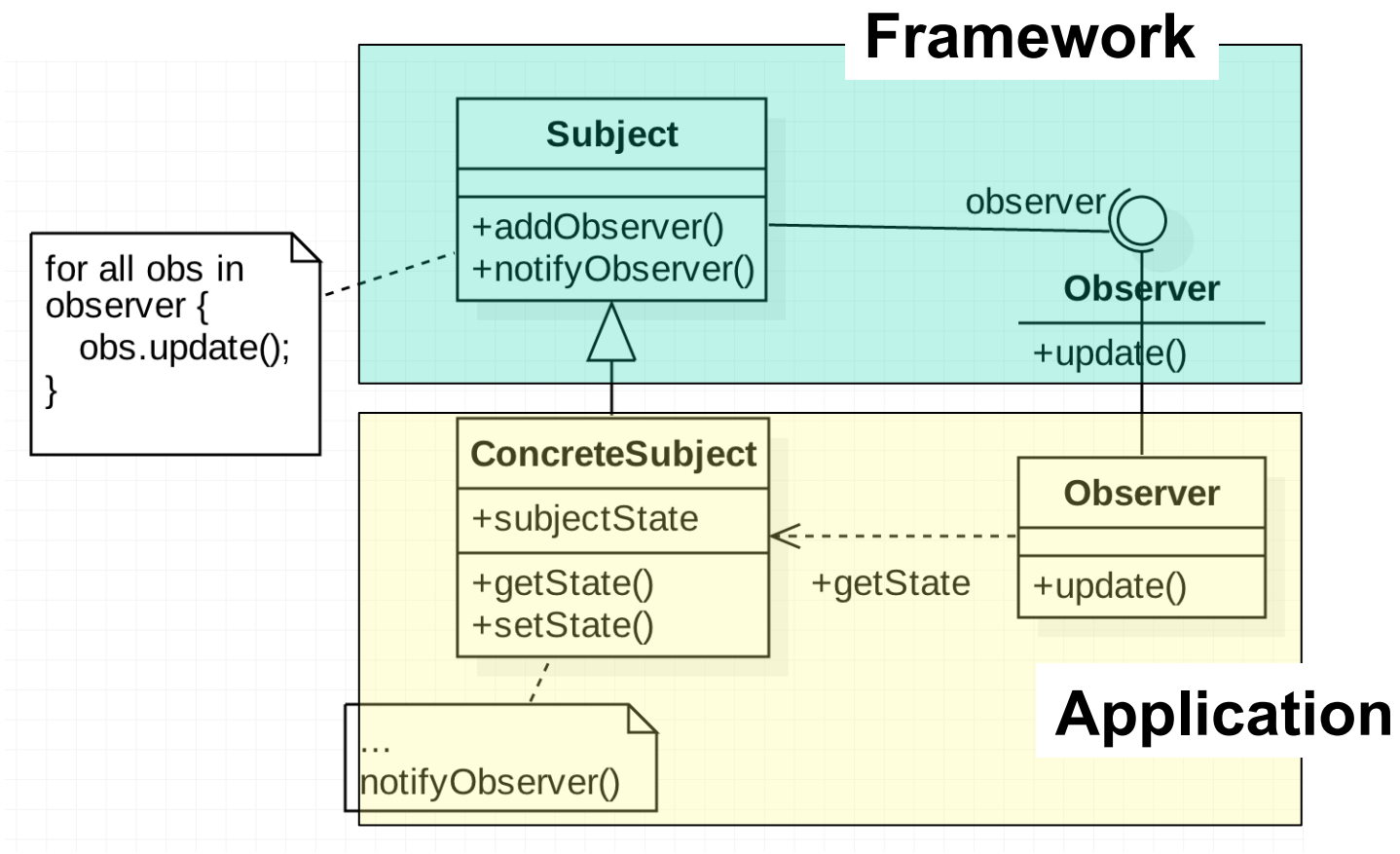

```
1   public class Observable {
2       private boolean changed = false;
3       private Vector obs;
4
5       public Observable() {
6           obs = new Vector();
7       }
8
9       public synchronized void addObserver(Observer o) {
10          if (o == null)
11              throw new NullPointerException();
12                  if (!obs.contains(o)) {
13                      obs.addElement(o);
14              }
15      }
16  }
```

用 vector 儲存所有的觀察者

將 Observer 加到 vector 中


```
1  public void notifyObservers(Object arg) {
2      Object[] arrLocal;
3      synchronized (this) {
4          if (!changed)
5              return;
6          arrLocal = obs.toArray();
7          clearChanged();
8      }
9
10     for (int i = arrLocal.length-1; i>=0; i--)
11         ((Observer)arrLocal[i]).update(this, arg);
12 }
```

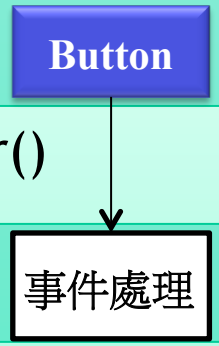
通知所有的觀察者做修改



應用：ActionListener

JAVA 的 event model 應用了 Observer 的架構

Java API	Pattern role
AbstractButton	Subject
fireActionListener()	notifyObserver()
ActionListener	Observer
actionPerformed	update



應用：ActionListener

```
1  public class TestEventModel extends JFrame{
2        private JButton b1;
3        public TestEventModel() {
4           b1 =  new JButton("Button");
5           getContentPane().add(b1);

7           //相當於 addObserver()
8           b1.addActionListener(new Listener1());
9           b1.addActionListener(new Listener2());
10          ...
```

b1 是主體

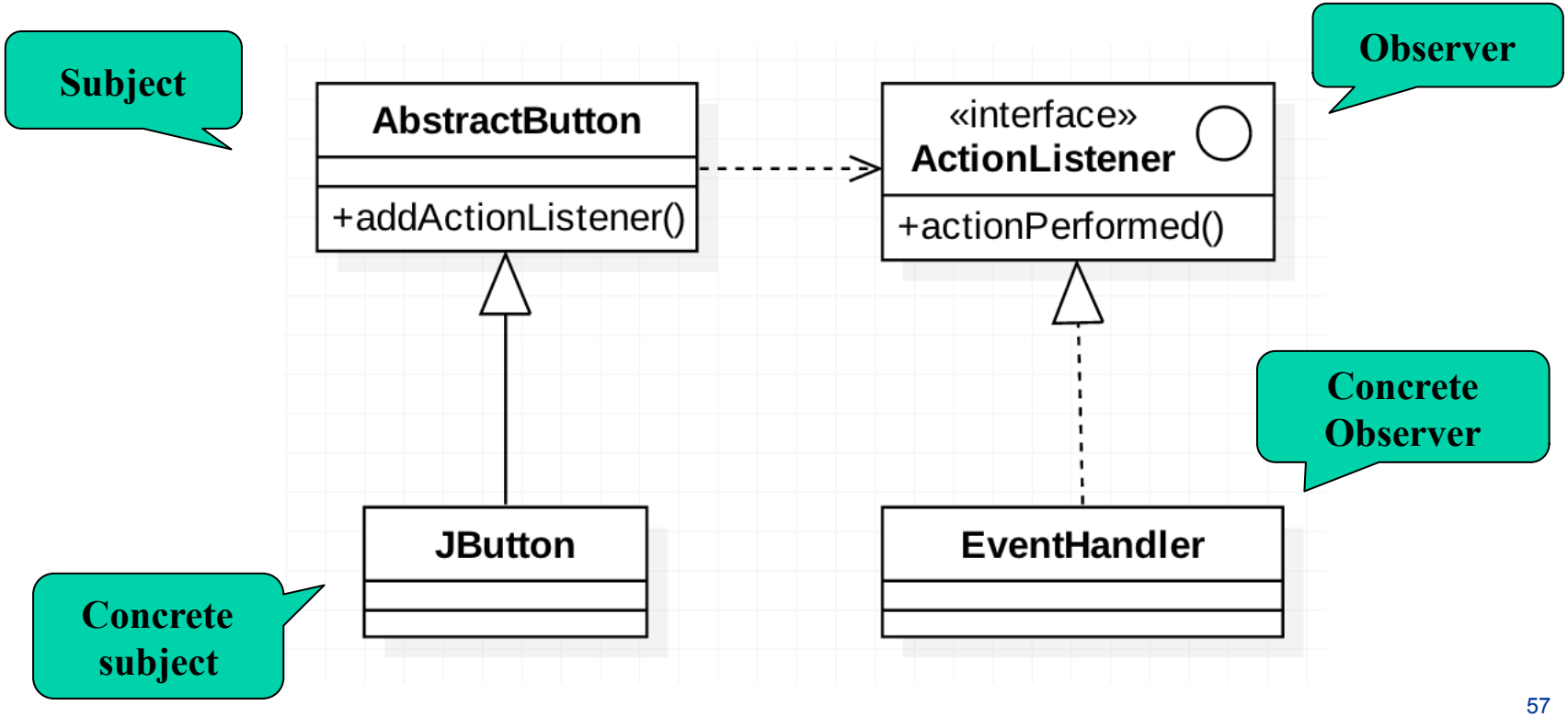
加上觀察者

應用：ActionListener

Observer

```
14  //ActionListener 相當於 Observer, actionPerformed 相當於 update()
15  class Listener1 implements ActionListener {
16      public void actionPerformed(ActionEvent e) {
17          System.out.println("Event happen");
18      }
19  }
```

update()



設計樣式 Part I

Adapter

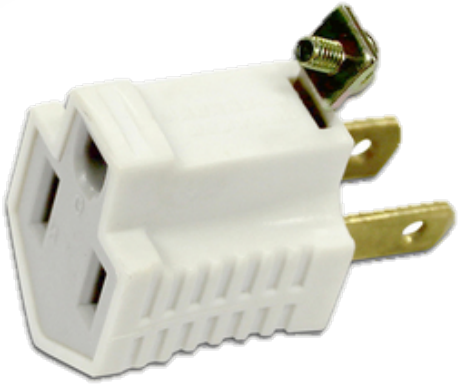
薛念林 逢甲大學資工系

Adapter

目的: 轉換類別的介面成為另一個介面所預期的樣式。Adapter 能夠讓不相容的介面，用轉接的方式合作, 並且相容。

*Convert the **interface** of a class into another interface clients expect. An adapter lets classes work together that could not otherwise because of incompatible interfaces.*

動機

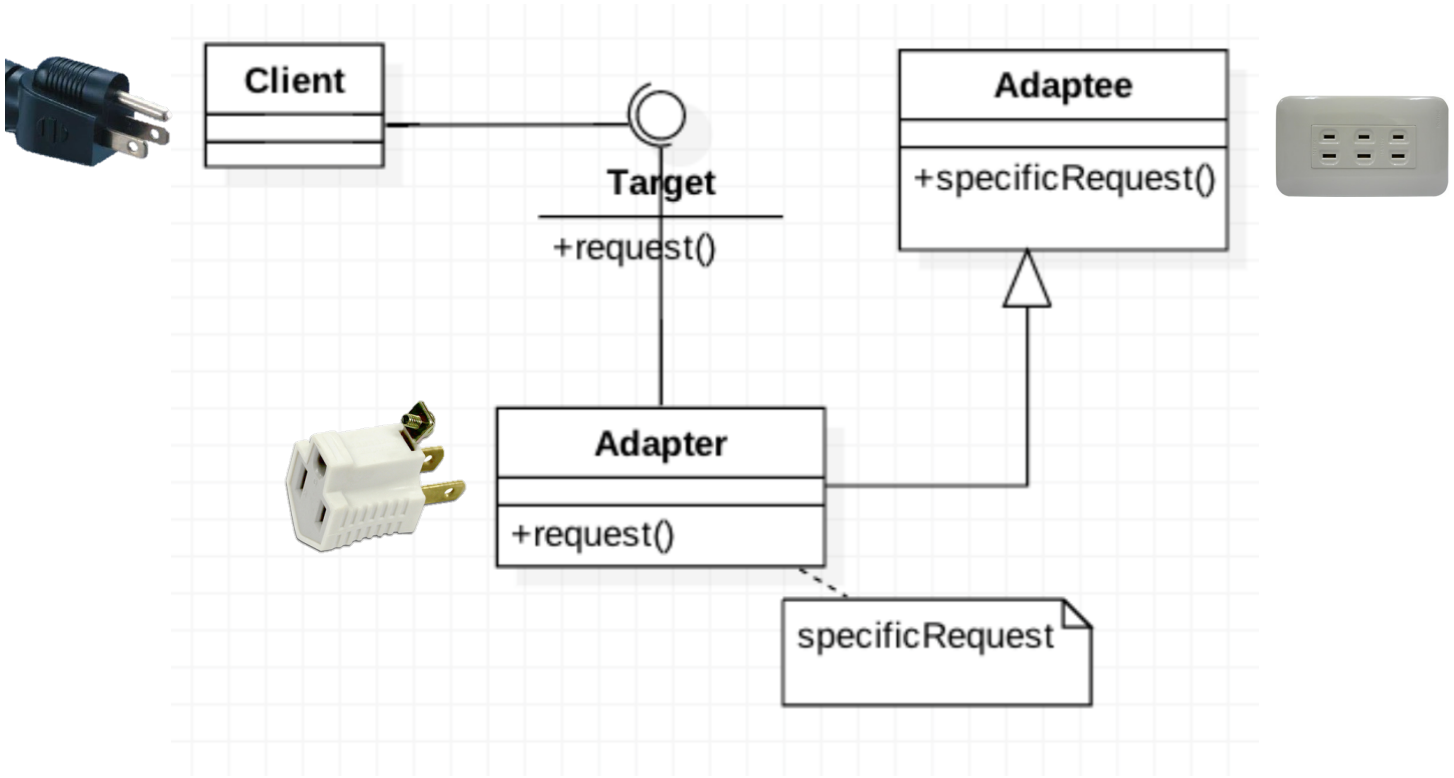


結構

Adapter 所以分為 2 種

- Class Adapter: 用繼承 (inheritance) 的技巧
- Object Adapter: 使用委託 (delegation) 的技巧

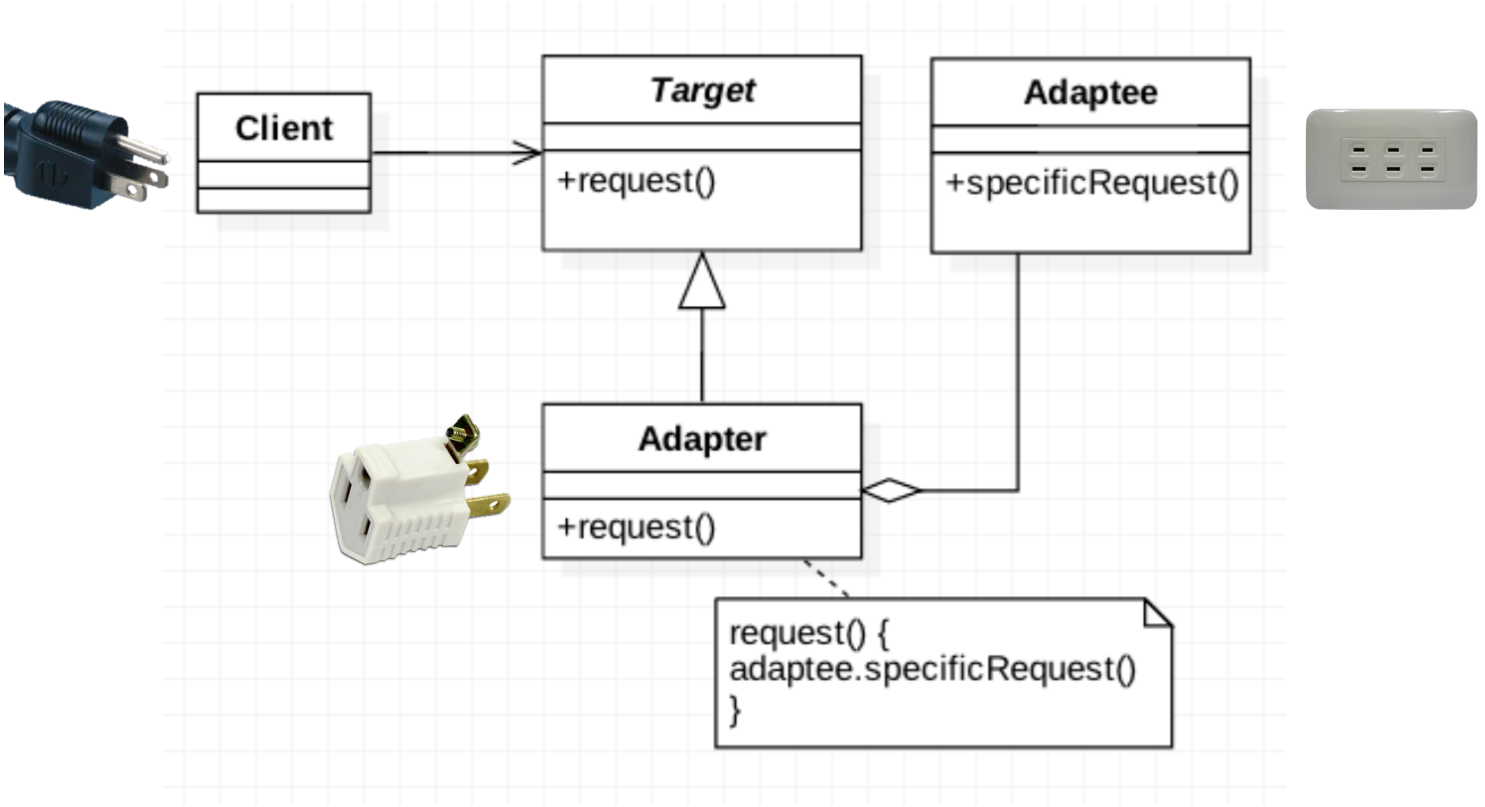
Class adaptor




```
2 // Adaptee 的 specificRequest() 對應到 Target 的 request()
3 interface Target {
4     public void request(Object arg);
5 }
6
7 class Adaptee {
8     public void SpecificRequest(Object arg) {
9         //...
10    }
11 }
12
```


```
13 class Adapter extends Adaptee implements Target {
14     public void request(Object arg) {
15         this.SpecificRequest(arg);
16     }
17
18 }
19
20 class Client {
21     // t 可以是一個 Target, 或是一個 Adapter (實作了 Target)
22     public void makeRequest(Target t, Object o) {
23       t.request(o);
24     }
25 }
```

Object adaptor




```
10 class Adapter extends Target {
11     Adaptee adaptee;
12
13     public Adapter(Adaptee a) {
14         this.adaptee = a;
15     }
16
17     public void request(Object arg) {
18         adaptee.specificRequest(arg);
19     }
20 }
```

應用：copy

VectorUtility.copy(Vector): Vector

- Vector 內的元素必須實作 *Copyable* 且 *isCopyable* 的介面

```
1 Vector v = new Vector( );
2 v.add(new Book("b1"));
3 v.add(new Book("b2"));
4 VectorUtility vu = new VectorUtility( );
5 Vector v2 = vu.copy(v);
```

Book 有實作 *Copyable* 介面


```
1  class VectorUtility {
2      public static Vector copy(Vector vin) {
3          Vector vout = new Vector( );
4          Enumeration e = new vin.elements( );
5          while (e.hasMoreElements( )) {
6              Copyable c = (Copyable)e.nextElement( );
7              if (c.iscopyable) {
8                  vout.addElement(c);
9              }
10         }
11     }
```

需符合 *isCopyable* 介面

Student 有實作 isValid 的介面，其含意與 isCopyable 相同，*VectorUtility.copy* 卻不能應用

- 修改 Student ?
- 修改 VectorUtility?
- ✓ 設計一個 StudentAdaptor


```
1   public class StudentAdapter implements Copyable {
2       private Student s;
3       public StudentAdapter(Student s) {
4           this.s = s;
5       }
6       public boolean isCopyable( ) {
7           return s.isValid( );
8       }
9   }
```


```
1    Vector v = new Vector( );
2    v.add(new StudentAdapter(new Student("s1")));
3    v.add(new StudentAdapter(new Student("s2")));
4    VectorUtility vu = new VectorUtility( );
5    Vector v2 = vu.copy(v);
```