第 4 章

# Factory Method 樣式

# 4.1 Factory Method 樣式美何在?

在上一章已經提到過，樣式本身並無美與醜，但是它常常會提升人們接受複雜的能力，因而讓人們不害怕複雜，這就是樣式之美的由來。例如，我門在上一章已經藉由 Template Method 樣式來簡化我們對 Binder 父、子類別複雜邏輯的認知，讓我們不再覺得 Binder 父、子類別是複雜的。Template Method 樣式之美提升了我們對付複雜事務的能力。就如同耶魯大學教授 David Gelernter 在其書：「力與美：電腦革命原動力」裡，他說到[參 1]：

> 「美對軟體很重要，大部分的技術人員並不願意討論美的重要，但美的重要是軟體史上一致的發展脈絡。」

他又說：

> 「就工程角度來看，美非常重要，因為軟體很複雜。複雜使得程式開發困難，也使得開發出來的軟體有難以使用之隱憂；而美就是對付複雜的最終防線。」

同樣地，本章將要介紹的 Factory Method 樣式之美也能發揮類似的威力。例如，Activity 的子類別都有 onCreate()函數如下：

```
public class ac01 extends Activity {
    private GraphicView gv = null;
    @Override public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
         gv = new GraphicView(this);
         setContentView(gv);
}}
```

起先，很多人搞不懂這 onCreate()函數的來源，只知道 Android 框架在啓動 ac01 時，會反向呼叫個函數而已。後來，知道這個 onCreate()函數由 Template Method 和 Factory Method 兩個樣式組合的結果，心中就突然清晰起來，並嘆曰：如此簡單。這就是設計樣式美的威力了。

***[參 1]*** *力與美：電腦革命原動力(Machine Beauty: Elegance and the Heart of Technology) 白屏方 譯(ISBN:957-621-470-X[312].*

# 4.2　介紹 Factory Method 樣式

## 4.2.1　誰來誕生應用類別之物件?

首先回顧上一章的 Template Method 樣式，如下圖：

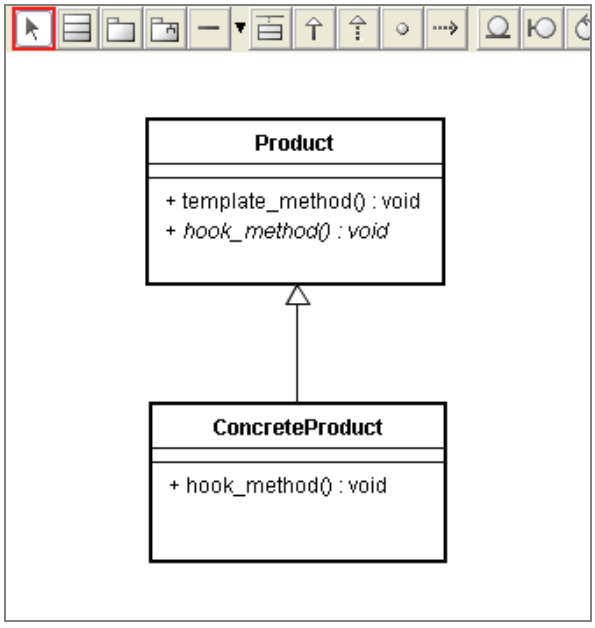



*圖 4-1　複習 Template Method 樣式*

基於此 Template Method 樣式，Product 抽象類別的 template_method()會反向呼叫 ConcreteProduct 應用類別的 hook_method()。然而，其複雜難解的問題是：

- 必須先誕生 ConcreteProduct 之物件後，才能返向呼叫到 ConcreteProduct 子類別的 hook_method()函數。
- 但是，在撰寫框架的抽象類別時，還不知有那些應用類別，又如何使用 new 指令去誕生應用類別之物件呢？

請你先別煩惱，擅用 Factory Method 樣式就能迎刃而解了，不亦美哉! 現在，請先看一個傳統的解決途徑，如下：

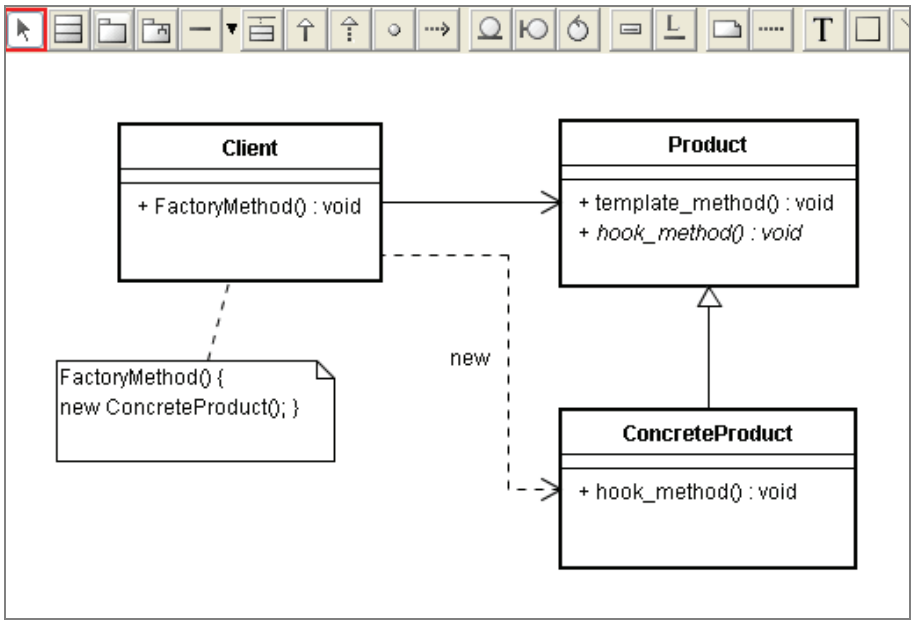
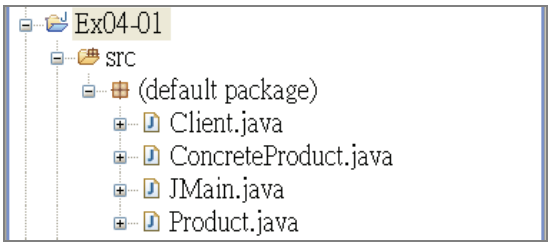



*圖 4-2　傳統的解決途徑*

這是由 Client 類別的 FactoryMethod()來負責誕生 ConcreteProduct 子類別的物件，這是可行的，如下之範例程式：

**<<撰寫程式>>**

Step-1. 建立一個 Java 應用程式專案：Ex04-01。




Step-2.　撰寫 Product 類別。

```
// Product.java
public abstract class Product {
        public void template_method(){
                System.out.println(hook_method());
```

```
        }
    protected abstract String hook_method();
}
```

Step-3.　撰寫 ConcreteProduct 類別。

```
// ConcreteProduct.java
public class ConcreteProduct extends Product{
    private String name;
    ConcreteProduct(String na)
            {   name = na;   }
    @Override protected String hook_method()
            {   return name;   }
}
```

Step-4.　撰寫 Client 類別。

```
// Client.java
public class Client {
        public void FactoryMethod() {
                Product obj = new ConcreteProduct("JEEP");
                obj.template_method();
 }}
```

Step-5.　撰寫 JMain 類別。

```
public class JMain {
        public static void main(String[] args) {
        Client ca = new Client();
        ca.FactoryMethod();
 }}
```

**<<說明>>**

由於 ConcreteProduct 是子類別，其包含著「會變」的部份，包括其名稱："ConcreteProduct"本身都是「會變」的。因而 Client 類別的 FactoryMethod()函數內之指令：

```
Product obj = new ConcreteProduct("JEEP");
```

也是「會變」(即會隨著應用類別的名稱之改變而變)的部份了。於是，基於「變與不變分離」原則，可以將 Client 類別裡的不變與會變部分加以分離開來，並納入父、子類別裡。

## 4.2.2 GoF 的 Factory Method 樣式圖

針對上一小節的 Client 類別，進行「變與不變」分離，將會變的 new ConcreteProduct();指令部分抽離出來，寫入子類別裡，如下圖：

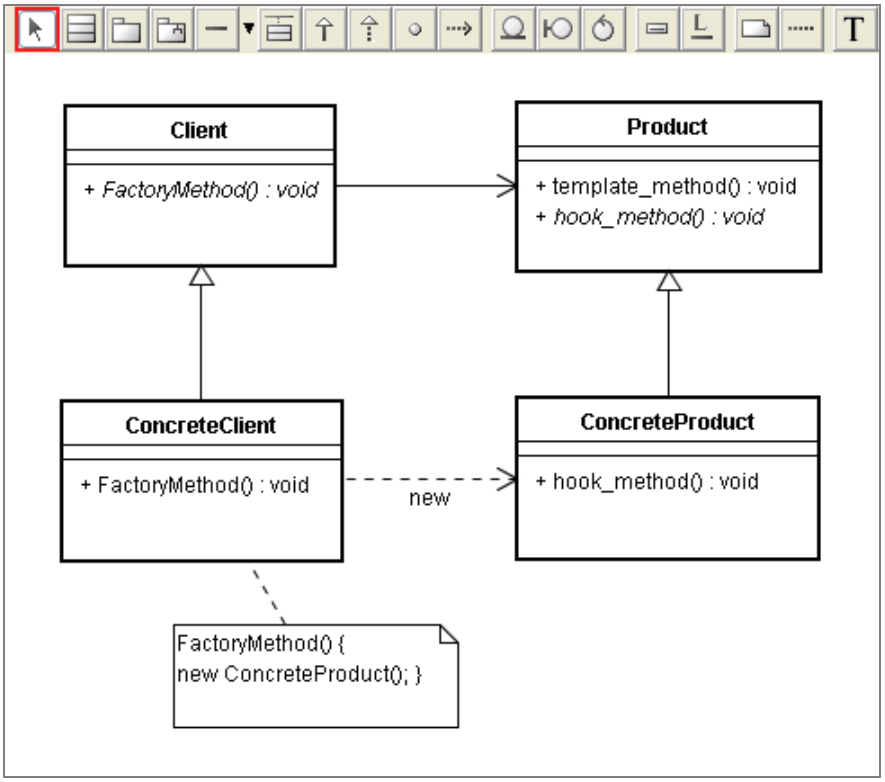


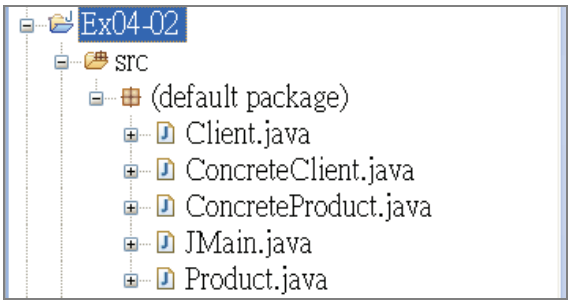

*圖 4-3　變與不變之分離*

這圖裡有兩的卡榫函數：FactoryMethod()和 hook_method()函數。其中，FactoryMethod()含有一個特殊任務：誕生 ConcreteProduct 應用類別之物件。因爲 Client 類別已經變成抽象類別了，可以納入框架裡。一旦納入框架裡，就必須把指令：new ConcreteProduct 寫在 ConcreteClient 類別裡。因爲 ConcreteClient 和 ConcreteProduct 兩者皆是應用類別，在撰寫 ConcreteClient 類別時，ConcreteProduct 應用類別是已知了，所以指令：new ConcreteProduct 寫在 ConcreteClient 類別裡是合理的。於是，上述的範例程式 Ex04-01 可改寫爲

Ex04-02，如下：

**<<撰寫程式>>**

Step-1. 建立一個 Java 應用程式專案：Ex04-02。




Step-2.　撰寫 Product 類別。

```
// Product.java
public abstract class Product {
        public void template_method()
          {     System.out.println(hook_method());     }
        protected abstract String hook_method();
}
```

Step-3.　撰寫 ConcreteProduct 類別。

```
// ConcreteProduct.java
public class ConcreteProduct extends Product{
       private String name;
       ConcreteProduct(String na)
              {   name = na;   }
       @Override protected String hook_method()
              {   return name;   }
}
```

Step-4.　撰寫 Client 類別。

```
// Client.java
public abstract class Client {
        protected Product obj = null;
        public void AnOperation() {
             FactoryMethod();
             obj.template_method();   }
```

```
    protected abstract void FactoryMethod();
}
```

Step-5. 撰寫 ConcreteClient 類別。

```
// ConcreteClient.java
public class ConcreteClient extends Client{
        protected void FactoryMethod() {
                obj = new ConcreteProduct("JEEP");
 }}
```

Step-6. 撰寫 JMain 類別。

```
// JMain.java
public class JMain {
        public static void main(String[] args) {
        Client sc = new ConcreteClient();
        sc.AnOperation();
 }}
```

**<<說明>>**

Client 類別裡的 AnOperation()函數是一個 template 函數，而其 FactoryMethod()則是一個卡榫函數。可以看出它也是一個 Template Method 樣式，只是比較特殊而已，它是要封藏有關於「誕生 ConcreteProduct 之物件」的會變部份而已。此時，Product 與 Client 兩個抽象父類別都只含有不變部分，可將之納入框架裡，成爲框架裡的類別，如下圖：

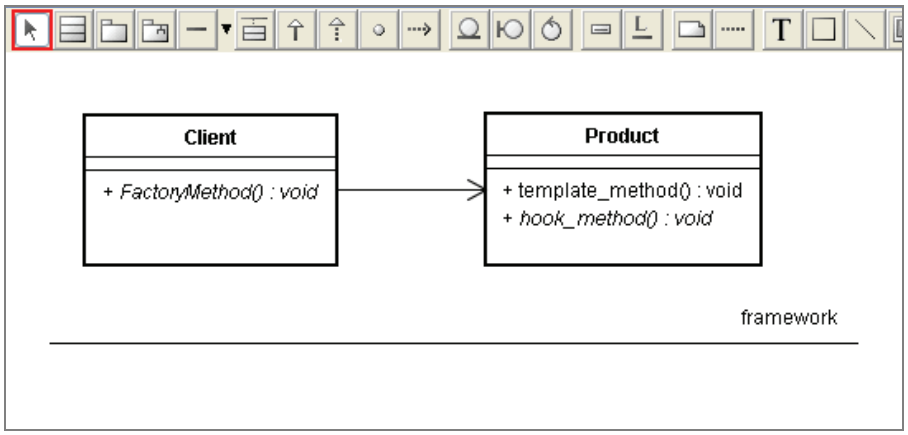



*圖 4-4 應用框架形成了*

上圖表達了框架設計者的意圖：

- 當應用程式執行時，Client 類別先反向呼叫其子類別的 FactoryMethod()函數，誕生一個 Product 的子類別的物件。
- 然後，呼叫 Product 類別的 template_method()函數，再反向呼叫其 hook_method()卡榫函數。

從上圖可以看到一個美好的框架了，也能體會出框架的設計之道了。有了框架之後，就能拿它來搭配各式各樣的應用類別。例如，配上一個新的 IntegerNumber 子類別：

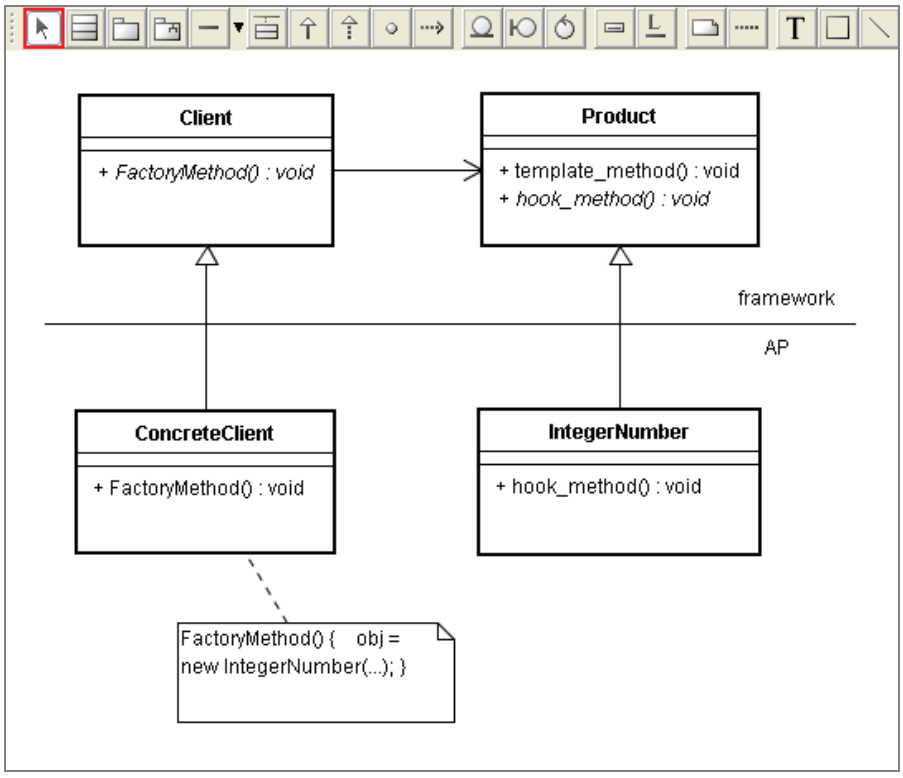


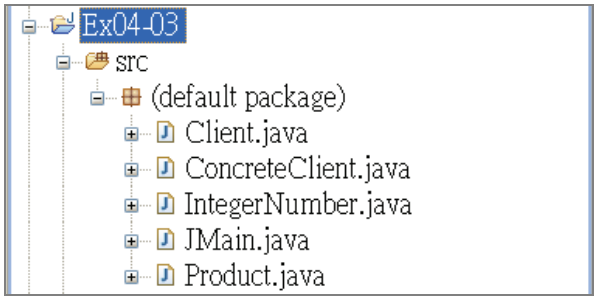

*圖 4-5　抽象類別搭配各種應用類別*

將圖 4-3 與圖 4-5 相比較，可以發現框架裡的 Client 和 Product 類別是穩定不

變的，不需改變就能搭配新的應用類別，這發揮了應用框架之本能：確保不變的抽象類別能隨時與多變的應用類別相搭配，然後組合出各種應用軟體。即使未來有需要更改上圖 4-5 裡的 IntegerNumber 類別名稱，也只需要更改 FactoryMethod() 函數內的 new IntegerNumber()指令而已。而不會發生『牽一髮動全身』的糟糕情形。茲寫個 Java 程式來實現上圖，如下：

**<<撰寫程式>>**

Step-1. 建立一個 Java 應用程式專案：Ex04-03。




Step-2. 複製 AF 的類別。

```
// Product.java
public abstract class Product {
        public void template_method()
              {    System.out.println(hook_method());    }
        protected abstract String hook_method();
}
```

```
// Client.java
public abstract class Client {
        protected Product obj = null;
        public void AnOperation() {
              FactoryMethod();
              obj.template_method();    }
    protected abstract void FactoryMethod();
}
```

Step-3. 撰寫 IntegerNumber 類別。

```
// IntegerNumber.java
public class IntegerNumber extends Product{
```

```
private int value;
IntegerNumber(int v)
      {   value = v;      }
@Override protected String hook_method() {
       return String.valueOf(value);
}}
```

Step-4. 改寫 ConcreteClient 類別。

```
// ConcreteClient.java
public class ConcreteClient extends Client{
       protected void FactoryMethod() {
              obj = new IntegerNumber(1250);
   }}
```

Step-5.　撰寫 JMain 類別。

```
// JMain.java
public class JMain {
       public static void main(String[] args) {
       Client sc = new ConcreteClient();
       sc.AnOperation();
   }}
```

**<<說明>>**

如此就輕易地「組裝」出一個新應用軟體了。剛才說明過，在上述的 Client 類別裡含有一個特殊的 Template Method 樣式，如下：

```
public abstract class Client {
protected Product obj = null;
       public void AnOperation() {
             FactoryMethod();
             obj.template_method();   }
    protected abstract void FactoryMethod();
}
```

其中的 AnOperation()函數是 template 函數，而其 FactoryMethod()則是卡榫函數。在 GoF 的<<Design Patterns>>一書裡，將這種特殊的 Template Method 樣式稱爲「Factory Method 樣式」，如下圖所示：

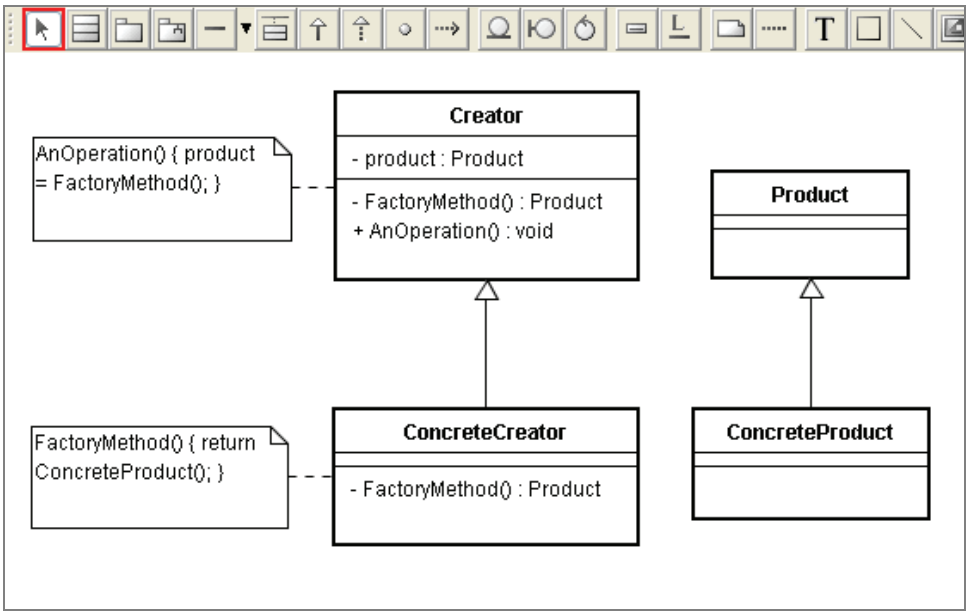



*圖 4-6　GoF 的 Factory Method 樣式圖*

在上述 Ex04-03 範例程式裡，你會發現 JMain 類別扮演真正的 Client 角色了，它會誕生 ConcreteCreator(或範例裡的 ConcreteClient)子類別之物件，再呼叫 Creator(或範例裡的 Client)裡的 AnOperation()函數。然後，間接呼叫到 Product 父、子類別的函數。表面上看來，多了 Creator 父、子類別，似乎多此一舉。但是設計樣式的奧妙與美感就蘊藏於其中。如果心中沒懷著設計樣式，就去看上圖或看上述的範例程式，就會因不理解而覺得很複雜。

反之，如果心中懷著設計樣式，再去看上圖或看上述範例程式，就會覺得它是簡單有序的。此外，還可以領悟設計樣式背後的美的考量：誕生物件的過程經常是很複雜的，其複雜部分可先寫入 Creator 抽象類別裡，就能大幅簡化 ConcreteCreator 和 JMain 類別的複雜性了。例如，在 Android 平台裡，經常需要誕生遠距(Remote)的跨進程(Process)的物件，誕生之後還得進行複雜的介面轉換工作等。也是因爲 Android 框架能封裝大量的複雜邏輯，而達到大幅簡化應用類別的目標，才會吸引許多人去使用它。Android 框架創造了簡而美，但也需要心中有設計樣式之美者才能體會出 Android 框架之美了。

### 4.2.3　Factory Method 樣式的延伸

在上圖 4-6 的 Creator 與 ConcreteCreator 父、子類別裡，含有一個 Factory Method 樣式。基於這個 Factory Method 樣式，我們還可以舉一反三，發揮一下 Java 的介面(Interface)機制。如果上圖裡的 Creator 抽象類別裡面並沒有 AnOperation()函數，而只有 FactoryMethod()抽象函數時，Creator 類別就成爲純粹抽象類別(Pure Abstract Class)了，這種類別純粹扮演介面角色，就相當於 Java 語言的 interface 機制，如下圖：

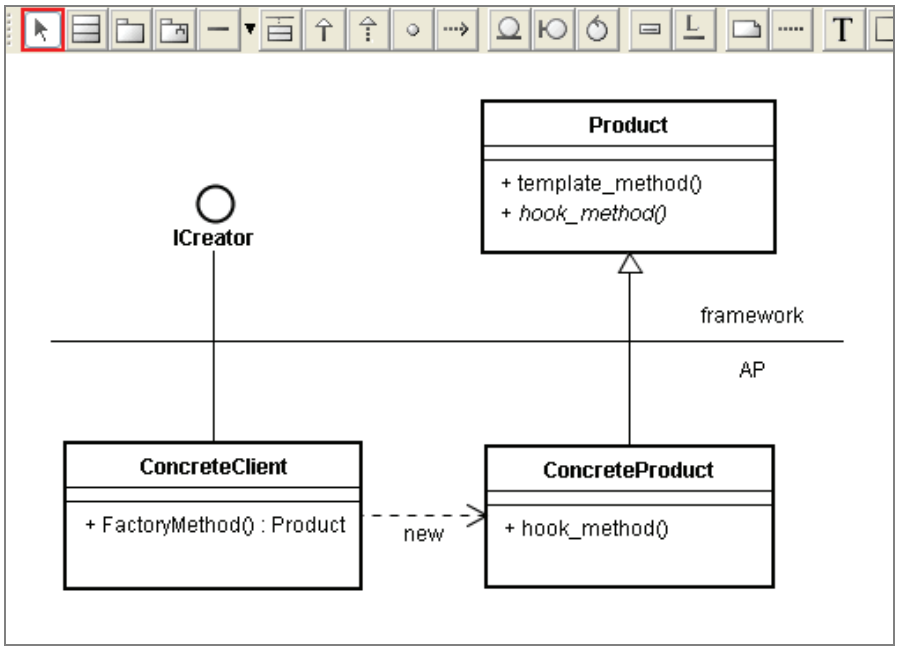
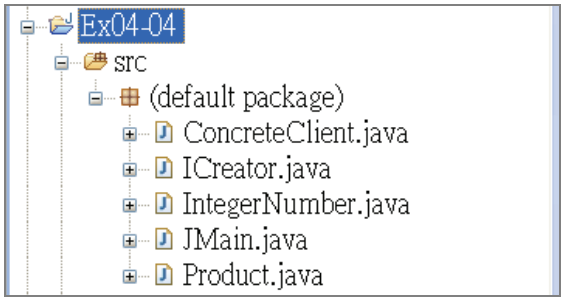



*圖 4-7　善用 interface 機制*

雖然父類別 Creator 已經變爲 ICreator 介面了，其 FactoryMethod 函數還是負責誕生 ConcreteProduct 物件，Factory Method 樣式還是存在那裡，只是形式有些變化而已。茲寫個 Java 程式來實現上圖，如下：

**<<撰寫程式>>**

Step-1. 建立一個 Java 應用程式專案：Ex04-04。




Step-2. 撰寫 AF 的類別和介面。

```
// Product.java
public abstract class Product {
        public void template_method()
            {   System.out.println(hook_method());    }
        protected abstract String hook_method();
}
```

```
// ICreator.java
public interface ICreator {
        Product FactoryMethod();
}
```

Step-3. 撰寫 IntegerNumber 類別。

```
// IntegerNumber.java
public class IntegerNumber extends Product{
private int value;
IntegerNumber(int v){
        value = v;
    }
@Override protected String hook_method() {
        return String.valueOf(value);
 }}
```

Step-4. 改寫 ConcreteClient 類別。

```
// ConcreteClient.java
public class ConcreteClient implements ICreator{
        public Product FactoryMethod() {
```

```
            return new IntegerNumber(1250);
 }}
```

Step-5.　撰寫 JMain 類別。

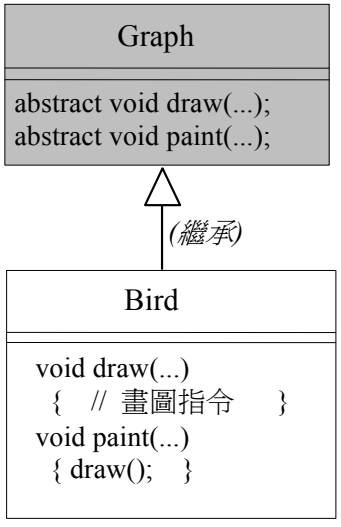
```
// JMain.java
public class JMain {
        public static void main(String[] args) {
        ICreator sc = new ConcreteClient();
        Product obj = sc.FactoryMethod();
        obj.template_method();
}}
```

**<<說明>>**

茲複習一下 Java 語言的介面機制，Java 提供兩個機制來表示介面：

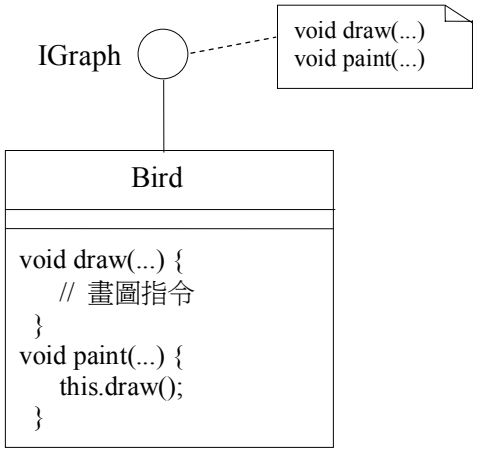
1. 以純粹抽象類別(Pure Abstract Class)表達之。
2. 以 Interface 機制表達之。

所謂純粹抽象類別，就是該類別裡的每一個函數都是 abstract 函數。這種函數就是空的函數，只有定義而無實作指令。例如：




這 Graph 就是一個純粹抽象類別，它只含有函數原型(Prototype)定義而已，並沒有函數的實作內容(Implementation)，其就扮演介面的角色了。此時可將上圖

改爲更親切的介面圖示，如下圖所示：




由於上述 IGraph 介面是純粹抽象類別的化身，而純粹抽象類別內含「不變」的部分，屬於框架的內涵，所以框架通常也定義了各式各樣的介面。例如 Android 框架裡就包含了 OnItemClickListener、SurfaceHolder.Callback 等各式各樣的介面。

# 4.3　Android 框架與 Factory Method 樣式

心懷樣式，心自美；
心中有美，萬物皆自美。

## 4.3.1　Factory Method 樣式範例之一

在 Android 框架裡，處處可見 Factory Method 樣式。例如，Activity、Service 等抽象類別裡都定義了 onCreate()函數，它就是一個典型的 FactoryMethod 函數。就像大家熟悉的簡單範例程式：

**<<撰寫程式>>**

Step-1. 建立一個 Android 應用程式專案：Ex04-05。


Step-2.　撰寫 GraphicView 類別。

```
// GraphicView.java
package com.misoo.pkxx;
import android.content.Context;
import android.graphics.Canvas;
import android.graphics.Color;
import android.graphics.Paint;
import android.view.View;

public class GraphicView extends View{
        private Paint paint= new Paint();

        GraphicView(Context ctx) {      super(ctx);        }
        @Override protected void onDraw(Canvas canvas) {
            int line_x = 10;      int line_y = 50;
            canvas.drawColor(Color.WHITE);
            paint.setColor(Color.GRAY);        paint.setStrokeWidth(3);
            canvas.drawLine(line_x, line_y, line_x+120, line_y, paint);
```

```
            paint.setColor(Color.BLACK);
            paint.setStrokeWidth(2);
            canvas.drawText("這是GraphicView繪圖區", line_x, line_y + 50, paint);
            int pos = 70;       paint.setColor(Color.RED);
            canvas.drawRect(pos-5, line_y - 5, pos+5, line_y + 5, paint);
            paint.setColor(Color.YELLOW);
            canvas.drawRect(pos-3, line_y - 3, pos+3, line_y + 3, paint);
    }}
```

Step-3. 撰寫 ac01 類別。

```
// ac01.java
package com.misoo.pkxx;
import android.app.Activity;
import android.os.Bundle;

public class ac01 extends Activity {
        private GraphicView gv = null;
        @Override public void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                gv = new GraphicView(this);
                setContentView(gv);
    }}
```

**<<說明>>**

在 Activity 抽象類別的設計上，就使用了 Factory Method 樣式。上述 ac01 子類別的 onCreate()就是一個 FactoryMethod 函數。其中的指令：

```
public void onCreate(Bundle savedInstanceState) {
    ……..
    gv = new GraphicView(this);
    ……..
}
```

就負責誕生 GraphicView 類別之物件。由於 GraphicView 是應用類別，其名稱是會變的，所以指令：

```
gv = new GraphicView(this);
```

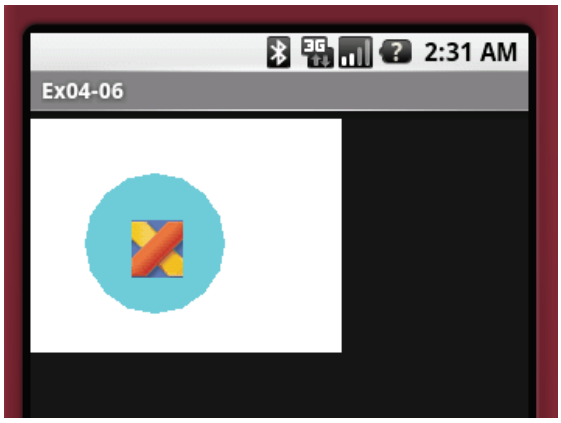
也是屬於會變的部份。依據 Factory Method 樣式，必須將這個指令寫在 ac01 子類別的 onCreate()函數裡面。

## 4.3.2　Factory Method 樣式範例之二

在 Android 的 SurfaceView 的子類別裡，通常會實作 SurfaceHolder.Callback 介面，此介面定義了一個 surfaceCreated()函數，它也負有誕生一些物件的責任。請看 Android 的範例程式 Ex04-06，如下：

**<<操作情境>>**

此程式執行時，呈現畫面如下：




**<<撰寫程式>>**

Step-1. 建立一個 Android 應用程式專案：Ex04-06。


Step-2.　撰寫 MySurfaceView 類別。

```
// MySurfaceView.java
package com.misoo.pkzz;
import android.graphics.Bitmap;
import android.graphics.BitmapFactory;
import android.graphics.Canvas;
import android.graphics.Color;
```

```
import android.graphics.Paint;
import android.view.SurfaceHolder;
import android.view.SurfaceView;

class MySurfaceView extends SurfaceView implements SurfaceHolder.Callback {
    MySurfaceView(ac01 context) {
            super(context);
            getHolder().addCallback(this);
        }
    public void surfaceCreated(SurfaceHolder holder) {
            Canvas cavans = holder.lockCanvas();
            cavans.drawARGB(255, 255, 255, 255);
            Bitmap bmp = BitmapFactory.decodeResource(getResources(), R.drawable.x_xxx);
            Paint paint = new Paint();
            paint.setColor(Color.CYAN);
            cavans.drawCircle(80, 80, 45, paint);
            cavans.drawBitmap(bmp, 65, 65, paint);
            holder.unlockCanvasAndPost(cavans);
      }
    public void surfaceDestroyed(SurfaceHolder holder) { }
    public void surfaceChanged(SurfaceHolder holder, int format, int w, int h) { }
}
```

Step-3. 撰寫 ac01 類別。

```
// ac01.java
package com.misoo.pkzz;
import android.app.Activity;
import android.os.Bundle;
import android.widget.LinearLayout;

public class ac01 extends Activity {
       private MySurfaceView sv;

      @Override public void onCreate(Bundle icicle) {
            super.onCreate(icicle);
            setContentView(R.layout.main);
            sv = new MySurfaceView(this);
            LinearLayout layout = new LinearLayout(this);
            layout.setOrientation(LinearLayout.VERTICAL);
            LinearLayout.LayoutParams param =
                    new LinearLayout.LayoutParams(200, 150);
            param.topMargin = 5;
            layout.addView(sv, param);
```

```
        setContentView(layout);
}}
```

**<<說明>>**

在 SurfaceHolder.Callback 介面的設計上，就使用了 Factory Method 樣式。上述 MySurefaceView 子類別的 surfaceCreated()就是一個 FactoryMethod 函數。其內含的指令是屬於會變的部份。依據 Factory Method 樣式，必須將這個指令寫在 MySurefaceView 子類別的 surfaceCreated()函數裡面。◆

(SPACE)