POSD20180112 期末考

五大原則

SRP: The Single Responsibility Principle

A CLASS SHOULD HAVE ONLY ONE REASON TO CHANGE.

Why was it important to separate these two responsibilities into separate classes? Because each responsibility is an axis of change. When the requirements change, that change will be manifest through a change in responsibility amongst the classes. If a class assumes more than one responsibility, then there will be more than one reason for it to change.

為什麼將兩項功能分為兩個 class 是重要的?因為每項功能都是一個變化的軸心。當需求改變時，這種改變將透過 class 之間功能的改變而顯現出來。如果一個 class 承擔超過一個功能，那麼將有多一個的理由來改變。

If a class has more than one responsibility, then the responsibilities become coupled. Changes to one responsibility may impair or inhibit the class' ability to meet the others.

如果一個 class 有不只一個功能，那麼功能就會變得相互耦合。改變一個功能可能降低或阻礙 class 與其他 class 會面的能力。

For example, consider the design in Figure 8-1. The Rectangle class has two methods shown. One draws the rectangle on the screen, the other computes the area of the rectangle.

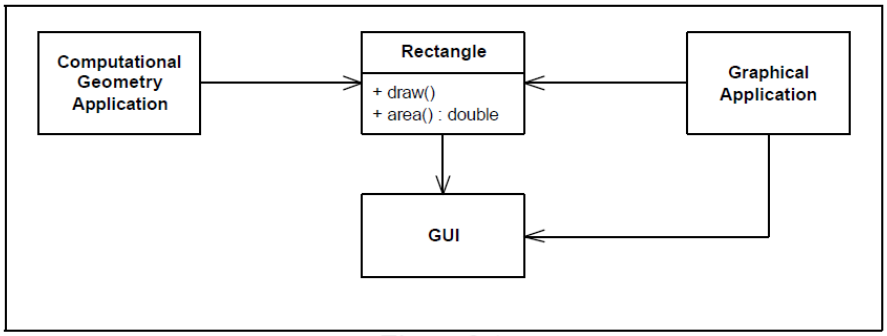


Figure 8-1
More than one responsibility

Two different applications use the Rectangle class. One application does computational geometry. It uses Rectangle to help it with the mathematics of geometric shapes. It never draws the rectangle on the screen. The other application is graphical in nature. It may also do some computational geometry, but it definitely draws the rectangle on the screen.

This design violates the SRP. The Rectangle class has two responsibilities. The first responsibility is to provide a mathematical model of the geometry of a rectangle. The second responsibility is to render the rectangle on a graphical user interface.

The violation of SRP causes several nasty problems. Firstly, we must include the GUI in the

computational geometry application. In .NET the GUI assembly would have to be built and deployed with the computational geometry application. Secondly, if a change to the GraphicalApplication causes the Rectangle to change for some reason, that change may force us to rebuild, retest, and redeploy the ComputationalGeometryApplication. If we forget to do this, that application may break in unpredictable ways.

A better design is to separate the two responsibilities into two completely different classes as shown in Figure 8-2. This design moves the computational portions of Rectangle into the GeometricRectangle class. Now changes made to the way rectangles are rendered cannot affect the ComputationalGeometryApplication.

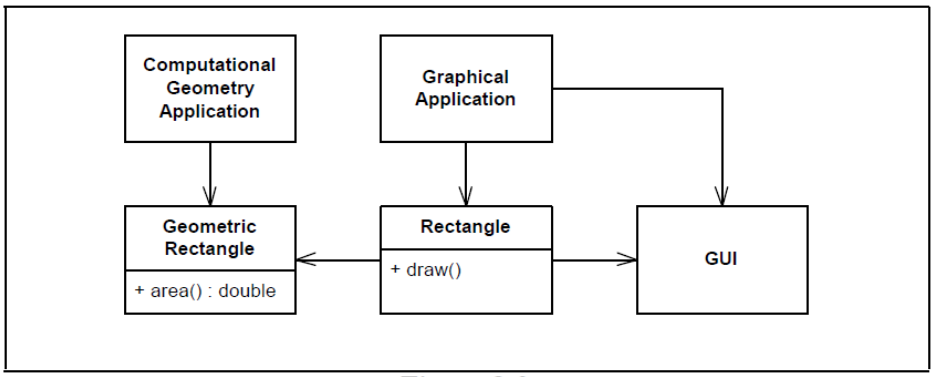


Figure 8-2
Separated Responsibilities

What is a Responsibility?

We define a responsibility to be"a reason for change." If you can think of more than one motive for changing a class, then that class has more than one responsibility.

Listing 8-1
Modem.cs -- SRP Violation

```
public interface Modem
{
```

Listing 8-1 (Continued)
Modem.cs -- SRP Violation

```
  public void Dial(string pno);
  public void Hangup();
  public void Send(char c);
  public char Recv();
}
```

Should these two responsibilities be separated? That depends upon how the application is changing. If the application changes in ways that affect the signature of the connection functions, then the design will smell of Rigidity because the classes that call send and read will have to be recompiled and redeployed more often than we like. If, on the other hand, the application is not changing in ways that cause the two responsibilities to change at difference

times, then there is no need to separate them. Indeed, separating them would smell of Needless Complexity.

是否一定要將兩個功能分開需要看狀況。Dial 和 Hangup 是同一個功能 Connect，Send 和 Recv 是同一個功能 Communicate，如果對連接的功能需要修改時，Send 和 Recv 也不得不重新編譯和重新部屬，這不是我們想要的。另一方面如果沒有什麼改變導致兩個功能在不同時間發生變化，那麼就沒有必要將他們分開。事實上把它們分開會聞到"不必要的複雜性"。

Separating coupled responsibilities.

There are often reasons, having to do with the details of the hardware or OS, that force us to couple things that we'd rather not couple. However, by separating their interfaces we have decoupled the concepts as far as the rest of the application is concerned.

通常有一些原因，不得不處理硬體或作業系統的細節，迫使我們去耦合那些我們不想結合的東西。但是透過分開他們的介面，就應用程式的其餘部分而言我們已經將概念分離了。

Conclusion

The SRP is one of the simplest of the principles, and one of the hardest to get right. Conjoining responsibilities is something that we do naturally. Finding and separating those responsibilities from one another is much of what software design is really about.

The Open-Closed Principle

SOFTWARE ENTITIES (CLASSES, MODULES, FUNCTIONS, ETC.) SHOULD BE OPEN FOR EXTENSION, BUT CLOSED FOR MODIFICATION.

When a single change to a program results in a cascade of changes to dependent modules, that program exhibits the undesirable attributes that we have come to associate with "bad" design. The program becomes fragile, rigid, unpredictable and unreusable. The open-closed principle attacks this in a very straightforward way. It says that you should design modules that never change. When requirements change, you extend the behavior of such modules by adding new code, not by changing old code that already works.

當對程式進行一項改變會導致對依賴模組進行一連串的改變時，這個程式會顯示我們壞設計相關的不良屬性。這個程式變得脆弱、難以改變、不可預測和不可用。OCP 非常直接的攻擊這個問題。他說你應該設計永不改變的模組，當需求改變時，透過增加新的程式碼來擴充這些模組的行為，而不是透過改變已經可以執行的舊程式碼。

Description

Modules that conform to the open-closed principle have two primary attributes.

1. They are “Open for Extension”.
   This means that the behavior of the module can be extended. That we can make the module behave in new and different ways as the requirements of the application change, or to meet the needs of new applications.
   為了擴展而打開，模組的行為可以被擴展。我們可以讓模組因為需求或是應用程式的改變以及與其他新的應用程式的需求去有新的或不同的狀態。
2. They are “Closed for Modification”.
   The source code of such a module is inviolate. No one is allowed to make source code changes to it.
   為了修改而關閉，模組的程式碼是不可侵犯的。沒有人有允許修改程式碼。

It would seem that these two attributes are at odds with each other. The normal way to extend the behavior of a module is to make changes to that module. A module that cannot be changed is normally thought to have a fixed behavior. How can these two opposing attributes be resolved?

看起來這兩個屬性是互相矛盾的。擴展模組行為的正常方法是對模組進行修改。一個不能修改的模組通常被認為有固定的行為。如何解決這兩個對立的屬性呢？

Abstraction is the Key.

The abstractions are abstract base classes, and the unbounded group of possible behaviors is represented by all the possible derivative classes. It is possible for a module to manipulate an abstraction. Such a module can be closed for modification since it depends upon an abstraction that is fixed. Yet the behavior of that module can be extended by creating new derivatives of the abstraction.

無限可能的行為是由所有可能衍生出來的類別所表示。模組可以操縱一個抽象。這樣的模組可以被關閉已進行修改，因為他取決於得抽象是固定的。模組的行為可以藉由創建新衍生的抽象來擴增。

Since programs that conform to the open-closed principle are changed by adding new code, rather than by changing existing code, they do not experience the cascade of changes exhibited by non-conforming programs.

由於符合 OCP 的程式是透過添加新的程式碼而不是透過改變現有的程式碼來改變，所以他們不會經歷尚未符合 OCP 程式碼所引起的一連串影響。

Strategic Closure

In general, no matter how “closed” a module is, there will always be some kind of change

against which it is not closed. Since closure cannot be complete, it must be strategic. That is, the designer must choose the kinds of changes against which to close his design. This takes a certain amount of prescience derived from experience. The experienced designer knows the users and the industry well enough to judge the probability of different kinds of changes. He then makes sure that the open-closed principle is invoked for the most probable changes.

一般來說，不管一個模組如何封閉，總是會有某種變化不是封閉的。由於封閉不能完，所以必須有策略性。設計師必須選擇哪種變化來關閉他的設計，來確保最有可能的改變符合 OCP。

以 Shape 為例子，可以建立一個 typeOrderType，藉由比較物件位子就能判斷該物件為哪種圖形。有了這種資料結構後當有需要增加新的圖形時只要去 typeOrderType 新增就好。

Extending Closure Even Further.

We have managed to close the Shape hierarchy, and the DrawAllShapes function against ordering that is dependent upon the type of the shape. However, the Shape derivatives are not closed against ordering policies that have nothing to do with shape types. It seems likely that we will want to order the drawing of shapes according to some higher level structure.

我們設法關閉了 Shape 層次結構，讓 DrawAllShapes 針對形狀的排序。然而，形狀的衍生並不是針對與形狀類型無關的排序策略而關閉的。看起來很可能會根據更高層次的結構來排列形狀的繪製。

一開始就把所有設計做好？這是很難的因為一開始所有東西都是我們的臆測。

Make all Member Variables Private.

This is one of the most commonly held of all the conventions of OOD. Member variables of classes should be known only to the methods of the class that defines them. Member variables should never be known to any other class, including derived classes. Thus they should be declared private, rather than public or protected.

這是 OOD 的慣例中最常見的一種。Class 的變數成員只能有定義他們的 class 使用。變數成員不應該被其他 class 所使用，包括衍生的 class。因此他們應該被宣告為 private 而不是 public 或 protected。

When the member variables of a class change, every function that depends upon those variables must be changed. Thus, no function that depends upon a variable can be closed with respect to that variable.

當變數成員的class改變時，所有依賴變數的function都必須改變。因此就這個變數而言，沒有依賴這個變數的function可以關閉。

如果只是”讀”這個參數的話，即使加了const還是要能夠保證他的type不會改變。如果很清楚使用的人不會去改變他的type，那宣告成public是可以的。

No Global Variables -- Ever.

No module that depends upon a global variable can be closed against any other module that might write to that variable. Any module that uses the variable in a way that the other modules don't expect, will break those other modules. It is too risky to have many modules be subject to the whim of one badly behaved one.

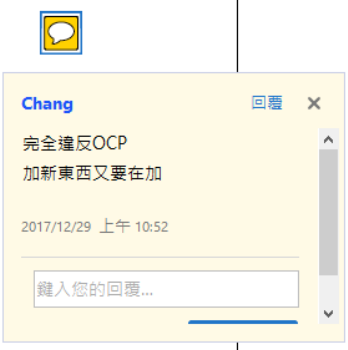
任何依賴於全域變數的模組是無法關閉的，因為其他模組可能會要使用該變數。任何以其它模組不期望的方式使用該變數的模組會破壞其它模組。有太多的模組被一個不好的行為左右，這樣太過冒險。

RTTI is Dangerous.

Run time type identification (RTTI) is intrinsically dangerous and should be avoided.

RTTI 是很危險應該被避免的。RTTI 主要的兩個 function 是 dynamic cast 和 typeid(*static cast 不是 RTTI，在 compiler time 轉型過去如果失敗就會爆掉，而 dynamic cast 則是失敗回傳 0 或例外)，但是真的都不能使用 dynamic cast 嗎？應該要依照狀況來使用。

```
void DrawAllShapes(Set<Shape*>& ss)
{
    for (Iterator<Shape*>i(ss); i; i++)
    {
        Circle* c = dynamic_cast<Circle*>(*i)
        Square* s = dynamic_cast<Square*>(*i)
        if (c)
            DrawCircle(c);
        else if (s)
            DrawSquare(s);
    }
}
```




```
void DrawSquaresOnly(Set<Shape*>& ss)
```

RTTI that does not violate the open-closed Principle.

```
{
    for (Iterator<Shape*>i(ss); i; i++)
    {
        Square* s = dynamic_cast<Square*>(*i);
        if (s)
            s->Draw();
    }
}
```

Conclusion

In many ways this principle is at the heart of object oriented design. Conformance to this principle is what yeilds the greatest benefits claimed for object oriented technology; i.e. reusability and maintainability. Yet conformance to this principle is not achieved simply by using an object oriented programming language. Rather, it requires a dedication on the part of the designer to apply abstraction to those parts of the program that the designer feels are going to be subject to change.

這個原則在許多方面是OO設計的核心。遵照這個原則對OO是最有利於OO技術，如可重用性和可維護性。然而，僅僅透過使用OO的程式語言並不能簡單地符合這個原則。相反地，他需要設計者的奉獻精神，將抽象應用到設計者認為需要改變的程序部分。

Liskov Substitution Principle

FUNCTIONS THAT USE POINTERS OR REFERENCES TO BASE CLASSES MUST BE ABLE TO USE OBJECTS OF DERIVED CLASSES WITHOUT KNOWING IT.

The importance of this principle becomes obvious when you consider the consequences of violating it. If there is a function which does not conform to the LSP, then that function uses a pointer or reference to a base class, but must know about all the derivatives of that base class. Such a function violates the Open-Closed principle because it must be modified whenever a new derivative of the base class is created.

如果有一個不符合 LSP 的 function，那麼該 function 使用一個 pointer 或 reference 指向一個 class，必須知道該 class 所有的衍生。這樣的函數違反了 OCP 原則，因為無論何時新增 class 的新衍生物，都必須對其進行修改。

A Simple Example of a Violation of LSP

One of the most glaring violations of this principle is the use of C++ Run-Time Type Information (RTTI) to select a function based upon the type of an object.

Square and Rectangle, a More Subtle Violation.

A square is a rectangle, and so the Square class should be derived from the Rectangle class. However this kind of thinking can lead to some subtle, yet significant, problems. Generally these problem are not foreseen until we actually try to code the application.

正方形也是一種矩形，所以正方形應該是矩形這個 class 所衍生的。然而這種想法會

導致一些微妙但是重要的問題。一般來說這些問題在我們寫應用程式之前是無法預料的。

```
class Rectangle
{
  public:
    void   SetWidth(double w)  {itsWidth=w;}
    void   SetHeight(double h) {itsHeight=w;}
    double GetHeight() const   {return itsHeight;}
    double GetWidth() const    {return itsWidth;}
  private:
    double itsWidth;
    double itsHeight;
};
```

例如正方形長和寬一樣，不需要兩個變數來存放長和寬，但是繼承矩形的情況下就會有兩個變數，很明顯地這是很浪費的。

However, let's assume that we are not very concerned with memory efficiency. Are there other problems? Indeed! Square will inherit the SetWidth and SetHeight functions. These functions are utterly inappropriate for a Square, since the width and height of a square are identical.". This should be a significant clue that there is a problem with the design.

假設我們不再記憶體效能問題，仍然有別的問題。正方形也會繼承SetWidth 和SetHeight functions，這些function完全不適合正方形，因為正方形的長和寬是一樣的。這是設計中存在問題的重要線索。

```
void f(Rectangle& r)
{
  r.SetWidth(32); // calls Rectangle::SetWidth
}
```

If we pass a reference to a Square object into this function, the Square object will be corrupted because the height won't be changed. This is a clear violation of LSP. The f function does not work for derivatives of its arguments. The reason for the failure is that SetWidth and SetHeight were not declared virtual in Rectangle.

如果我們傳遞正方形物件的位置到這個function，正方形物件會被破壞因為長度不會改變。這是明顯違反LSP。F function並不是用於其衍生的參數。真正的原因是SetWidth以及SetHeight沒有被宣告為virtual。

The Real Problem

Moreover, you can pass a Square into a function that accepts a pointer r reference to a Rectangle, and the Square will still act like a square and will remain consistent. Thus, we might conclude that the model is now self consistent, and correct. However, this conclusion would be amiss. A model that is self consistent is not necessarily consistent with all its users! Consider function g below.

```
void g(Rectangle& r)
{
  r.SetWidth(5);
  r.SetHeight(4);
  assert(r.GetWidth() * r.GetHeight()) == 20);
}
```

This function invokes the SetWidth and SetHeight members of what it believes to be a Rectangle. The function works just fine for a Rectangle, but declares an assertion error if passed a Square. So here is the real problem: Was the programmer who wrote that function justified in assuming that changing the width of a Rectangle leaves its height unchanged?

Clearly, the programmer of g made this very reasonable assumption. Passing a Square to functions whose programmers made this assumption will result in problems. Therefore, there exist functions that take pointers or references to Rectangle objects, but cannot operate properly upon Square objects. These functions expose a violation of the LSP. The addition of the Square derivative of Rectangle has broken these function; and so the Open-Closed principle has been violated.

Conclusion

The Open-Closed principle is at the heart of many of the claims made for OOD. It is when this principle is in effect that applications are more maintainable, reusable and robust. The Liskov Substitution Principle is an important feature of all programs that conform to the Open-Closed principle. It is only when derived types are completely substitutable for their base types that functions which use those base types can be reused with impunity, and the derived types can be changed with impunity.

OCP原則是關於OOD要求的核心。正是因為這個原則使應用程式更好維護、重複使用以極強大。LSP是一個重要特徵代表所有程式符合OCP。只有當衍生類型完全可以替代期原本類型時，那些使用原本類型的函數才能被重用(reuse)而不受懲罰，並且衍生類型可以隨意改變。

Interface Segregation Principle

> ***CLIENTS SHOULD NOT BE FORCED TO DEPEND UPON INTERFACES THAT THEY DO NOT USE.***

When clients are forced to depend upon interfaces that they don’t use, then those clients are subject to changes to those interfaces. This results in an inadvertent coupling between all the clients. Said another way, when a client depends upon a class that contains interfaces that the client does not use, but that other clients do use, then that client will be affected by the changes that those other clients force upon the class. We would like to avoid such couplings where possible, and so we want to separate the interfaces where possible.

當客戶被迫依賴於他們不使用的介面時，那些客戶將會受到這些介面的改變。這導致所有客戶之間的偶然耦合。換句話說，當一個客戶依賴於一個包含客戶不使用的介面的class，而另一個客戶使用時，這個客戶將會受到其他客戶對這個class修改的影響。我們希望盡可能避免這種耦合，所以我們希望在盡可能的情況下區分介面。

Conclusion

In this article we have discussed the disadvantages of "fat interfaces"; i.e. interfaces that re not specific to a single client. Fat interfaces lead to inadvertent couplings between clients that ought otherwise to be isolated. By making use of the ADAPTER pattern, either through delegation (object form) or multiple inheritance (class form), fat interfaces can be segregated into abstract base classes that break the unwanted coupling between clients.

在這篇文章中，我們討論了胖介面的缺點。胖介面導致客戶之間偶然被隔離偶然連接。透過使用adapter，無論是透過委託還是多重繼承，胖介面都可以被分離到抽象的基本class中，進而打破客戶間不必要的耦合。

Dependency Inversion Principle

> ***A. HIGH LEVEL MODULES SHOULD NOT DEPEND UPON LOW LEVEL MODULES. BOTH SHOULD DEPEND UPON ABSTRACTIONS.***
>
> ***B. ABSTRACTIONS SHOULD NOT DEPEND UPON DETAILS. DETAILS SHOULD DEPEND UPON ABSTRACTIONS.***

The Definition of a "Bad Design"

1. It is hard to change because every change affects too many other parts of the system. (Rigidity)
   很難改變，因為每一個變化都會影響系統的其他部分。(剛性、僵化)
2. When you make a change, unexpected parts of the system break. (Fragility)
   進行更改時，系統中斷在意外的部分。(脆弱)
3. It is hard to reuse in another application because it cannot be disentangled from the current application. (Immobility)
   很難從另一個應用程式再使用，因為他不能從當前的應用程式中分離出來。(不動)

Such rigidity is due to the fact that a single change to heavily interdependent software begins a cascade of changes in dependent modules. When the extent of that cascade of change cannot be predicted by the designers or maintainers, the impact of the change cannot be

estimated. This makes the cost of the change impossible to predict.

這種僵化是由於對相互依賴的軟體發生單一改變時，產生了隸屬模組的一連串改變。當設計師或維護人員無法預測這種變化時，變化的影響就會無法估計，這使用變更的成本無法預測。

Often the new problems are in areas that have no conceptual relationship with the area that was changed. Such fragility greatly decreases the credibility of the design and maintenance organization. Users and managers are unable to predict the quality of their product. Simple changes to one part of the application lead to failures in other parts that appear to be completely unrelated. Fixing those problems leads to even more problems, and the maintenance process begins to resemble a dog chasing its tail.

新的問題常常出現在與改變的部分沒有概念化關係的地方。這種脆弱性大大降低設計和維護機構的可信度。使用者和管理人員無法預測其產品的質量。對應用程式一個部分進行簡單的更改會導致看起來完全不相關的其他部份發生故障。解決這些問題導致更多的問題，維護過程開始像一隻追逐尾巴的狗。

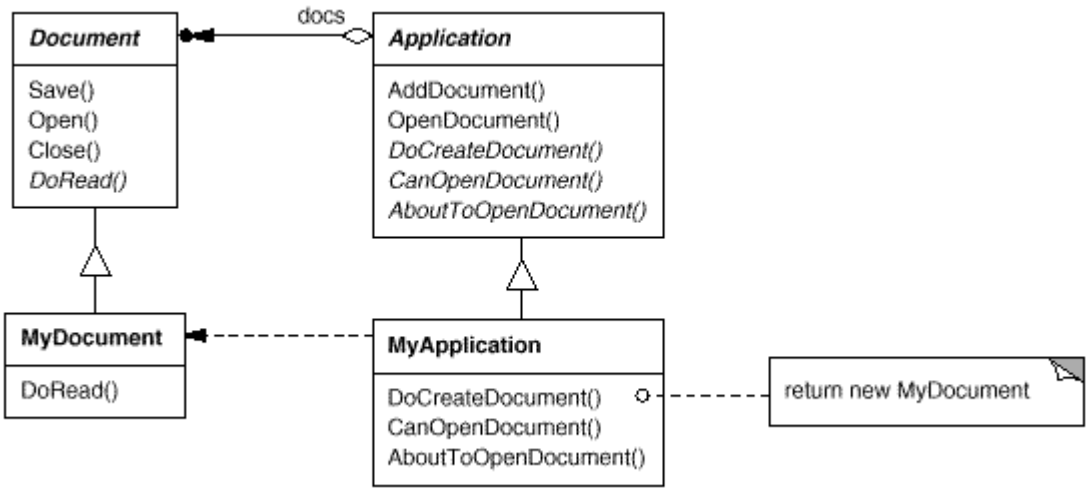
A design is immobile when the desirable parts of the design are highly dependent upon other details that are not desired. Designers tasked with investigating the design to see if it can be reused in a different application may be impressed with how well the design would do in the new application. However, if the design is highly interdependent, then those designers will also be daunted by the amount of work necessary to separate the desirable portion of the design from the other portions of the design that are undesirable. In most cases, such designs are not reused because the cost of the separation is deemed to be higher than the cost of redevelopment of the design.

當設計所需部分高度依賴於不需要的其他細節時，設計是不可移動的。設計師負責調查設計，看看他是否可以在不同的應用程序中重複使用，設計將在新的應用程式中執行。然而，如果設計高度相互依賴，那麼設計者也會因為將設計的理想部分與不希望設計的其他部分分開所需的工作量而受到損害。多數情況下，這樣的設計不會被重複使用，因為分離的成本被認為高於重建的成本。

Others

A good example of this principle(ISP) is the TEMPLATE METHOD pattern from the GOF1 book. In this pattern, a high level algorithm is encoded in an abstract base class and makes use of pure virtual functions to implement its details. Derived classes implement those detailed virtual functions. Thus, the class containing the details depend upon the class containing the abstraction.

Template Method



Intent

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.

在操作中定義演算法的框架，將一些步驟等到子類別在處理。Template Method 可以使子類別可以在不改變演算法結構的情況下重新定義演算法的某些步驟。

Motivation

Consider an application framework that provides Application and Document classes. The Application class is responsible for opening existing documents stored in an external format, such as a file. A Document object represents the information in a document once it's read from the file.

Applications built with the framework can subclass Application and Document to suit specific needs. For example, a drawing application defines DrawApplication and DrawDocument subclasses; a spreadsheet application defines SpreadsheetApplication and SpreadsheetDocument subclasses.

Applicability

The Template Method pattern should be used

- to implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.
- when common behavior among subclasses should be factored and localized in a common class to avoid code duplication. This is a good example of "refactoring to generalize" as described by Opdyke and Johnson [OJ93]. You first identify the differences in the existing code and then separate the differences into new operations. Finally, you replace the differing code with a template method that calls one of these new operations.
- to control subclasses extensions. You can define a template method that calls "hook" operations (see Consequences) at specific points, thereby permitting extensions only at those points.

Consequences

Template methods are a fundamental technique for code reuse. They are particularly important in class libraries, because they are the means for factoring out common behavior in library classes.

Template methods lead to an inverted control structure that's sometimes referred to as "the Hollywood principle," that is, "Don't call us, we'll call you" [Swe85]. This refers to how a parent class calls the operations of a subclass and not the other way around.

Template methods call the following kinds of operations:

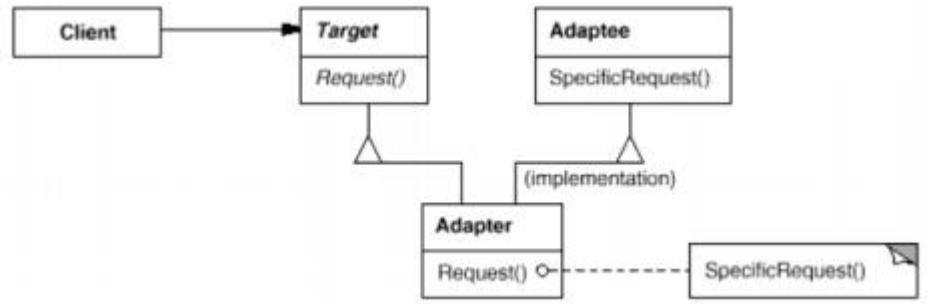
- concrete operations (either on the ConcreteClass or on client classes);
- concrete AbstractClass operations (i.e., operations that are generally useful to subclasses);
- primitive operations (i.e., abstract operations);
- factory methods (see Factory Method (107)); and
- **hook operations**, which provide default behavior that subclasses can extend if necessary. A hook operation often does nothing by default.

It's important for template methods to specify which operations are hooks (*may* be overridden) and which are abstract operations (*must* be overridden). To reuse an abstract class effectively, subclass writers must understand which operations are designed for overriding.

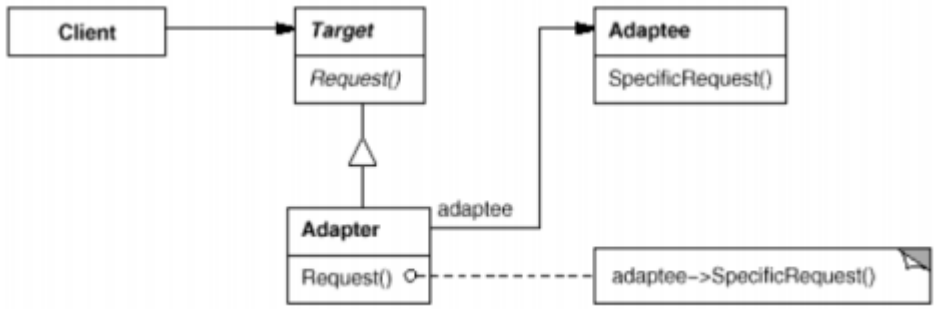
- 只定義演算法流程,某些步驟留給子類別做,以便在不改變演算法整體架構。
- TemplateMethod 為 non-virtual function、PrimitiveOP1、PrimitiveOP2 為 virtual function。
- Non-Virtual 該不該不 Override?不該!若被 Override 則無法確保流程一致,Virtual function 被 Override 是為了 Polymorphism。

Adapter

A class adapter uses multiple inheritance to adapt one interface to another:



An object adapter relies on object composition:



- 將類別的介面轉換成外界所預期的另一種介面，讓原先介面不相容問題而無法使用的類別可以合作使用。
- 使用時機：想利用現有的類別，但它的介面與你所需不符。
- Class Adapter
  - 優點：
    - 容易 override Adaptee 原本的行為。因為 Class Adapter 繼承了 Adaptee，所以可以輕易的 override Adaptee。
    - 代碼較精簡。
  - 缺點：
    - 只能轉換單一 Adaptee。
    - 無法動態改變欲轉換的 Adaptee。因為使用繼承技術，在 compile-time 已經決定了要繼承的 Adaptee，所以無法動態改變 Adaptee。
- Object Adapter
  - 優點：
    - 可轉換多個 Adaptee。因為使用了組合技術，配合 polymorphism(多型/多態)，所以能轉換 class 和其 derived class。
    - 可動態改變欲轉換的 Adaptee。因為使用了組合技術，可以在 run-time 的改變欲轉換的 Adaptee。
  - 缺點：
    - 難 override Adaptee 原本的行為。
- Class Adapter 和 Object Adapter 優缺點剛好互補，可依實際需求決定之，大體上而

言，Object Adapter 優於 Class Adapter，因為彈性較大，且可面對將來未知的 class，也應證了那句『Favor object composition over class inheritance』的 Design Pattern 真言。

會問繼承的 adapter

Factory

Iterator

Visitor

尋找適當的物件

補充