OOP #6

Sam Xiao, Nov.17, 2017

Overview

目前已經將該有的功能都完成了，唯一剩下 `資料庫備份` 還沒實作，這是本次 homework 的重點。

Outline

- OOP #6
  - Overview
  - Outline
  - Recap
  - User Story
    - 增加 Log 機制
    - Handler Interface 不適用
  - Task
    - 增加 Log 機制
    - Handler Interface 不適用
  - Architecture
    - DBHandler Family
    - DBAdapter
  - Implementation
    - DBHandler
    - AbstractDBHandler
    - DBBackupHandler
    - DBLogHandler
    - DBAdapter

AbstractTask
Summary
Conclusion

Recap

Homework 5 :

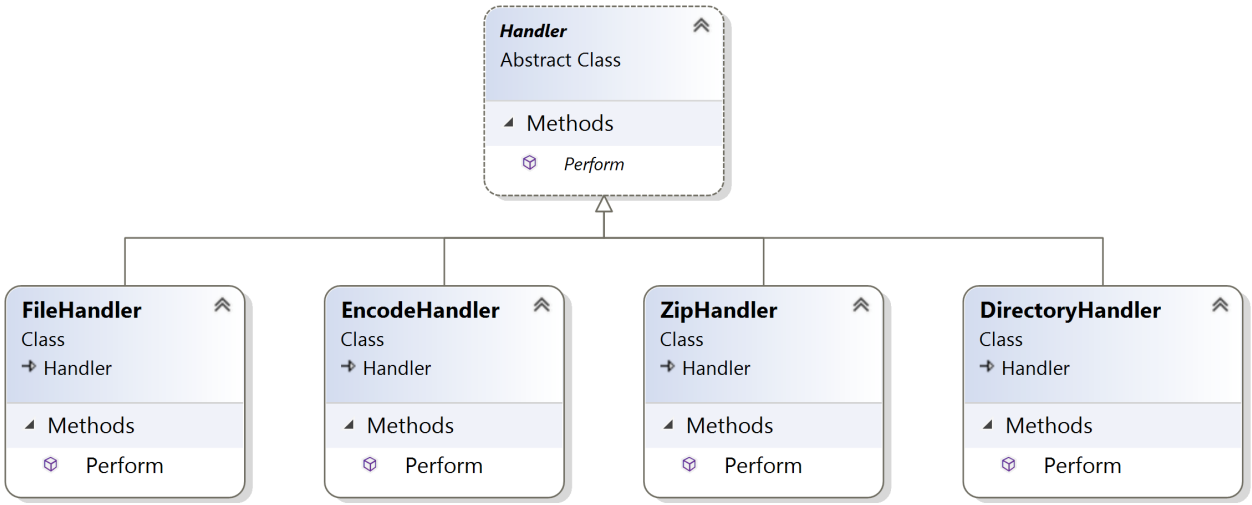
- `Value Object` / `Data Transfer Object` 正確的設計方式
- `Facade Pattern` : 為一堆 class 定義統一的入口，讓用戶端更容易使用
- `單一職責原則` : 應該只有一個 `原因` 使的 class / method 被修改
- `並不是所有地方都要用 Factory 取代 new`
  - Object 有 `抽換` 的需求
  - Object 有根據不同邏輯 `new` 的需求
  - 不想讓使用端直接 `new` object
- OOP 兩個實務上作法 :
  - 建立 `中介 class` 封裝 `複雜度`
  - 建立 `Interface` 增加 `擴充點`
- `完全不設計` 與 `過度設計`，都不是好的設計

User Story

增加 Log 機制

原本希望將備份資料存進資料庫即可，但 user 希望也能將 log 寫進資料庫，未來也不排除還有其他資料需要寫進資料庫，因此目前最少會有兩個 table 需要寫入：一個是 `MyBackup` table，令一個是 `MyLog` table。

Handler Interface 不適用



在 homework 3，我們曾經定義出 `Handler` interface，將來有任何新的功能，都會採用 `Handler` interface 繼續擴充，讓其他部份 `開放封閉`。也就是 `建立 Interface 增加擴充點` 的技巧。

一旦要加入 `DBHandler` 時，卻發現 `Handler` 只有 `Perform()` 實在不夠用，想對 `Handler` 增加 method。

但一旦對 `Handler` 增加 method，interface 就得被修改，這就會出現了幾個問題：

- OOP 就是 `依賴抽象，封裝變化`，也就是藉由依賴 `穩定的抽象` ，換來其他程式碼的 `開放封閉` ；若去修改 interface，就是親手 `破壞抽象`，則 `開放封閉` 必然崩潰，這就失去了 使用 OOP 的意義了
- `Handler` 新增的 method 必然是與 `DBHandler` 相關，也就是與資料庫相關的 method，這將造成其他 handler 對 interface 有 `空實作`，違反 `界面隔離原則`

> 界面隔離原則 (ISP : Interface Seperation Principle)
>
> 用戶端程式碼不應該依賴它用不到的界面
>
> 白話：應該相依最小界面，每個依賴都要花在刀口上

> 白話 : class 對於 interface 不該有空實作

`Handler` interface 不應該被修改，但原本的 interface 又不符合需求，改或不改面臨兩難。

> Q : 回想我們用別人 framework / package 最怕的是什麼？

A : framework / package 改版後的 `Breaking Change`

我們依賴了 framework / package 的 interface，結果對方 interface 改了之後，造成我所有的程式都要修改。

假如一個 framework / package 的 breaking change 嚴重的話，就不是一個好的 framework / library。

將心比心，我們也不應該任意修改 interface，這將會造成使用端的 `Breaking Change`

Task

增加 Log 機制

既然不排除有其他 table 需要寫入，根據 `依賴抽象，封裝變化` 與 `增加界面，幫助擴展` 原則，我們會使用 `建立 Interface 增加擴充點` 的技巧，讓程式碼符合 `開放封閉` 原則。

Handler Interface 不適用

既然 `Handler` interface 不應該被修改，但又不符合目前 `資料庫備份` 需求，只好另外新增一個 `Adapter` class，負責將 `Handler` interface 轉成 `DBHandler` interface。

> 原來的 `Handler` interface 不被修改，另外開一個 `Adapter` interface，符合 `開放封閉原則`

> 原本的 `FileHandler`、`EncodeHandler` .... 不會有 空實作，符合 界面隔離原則

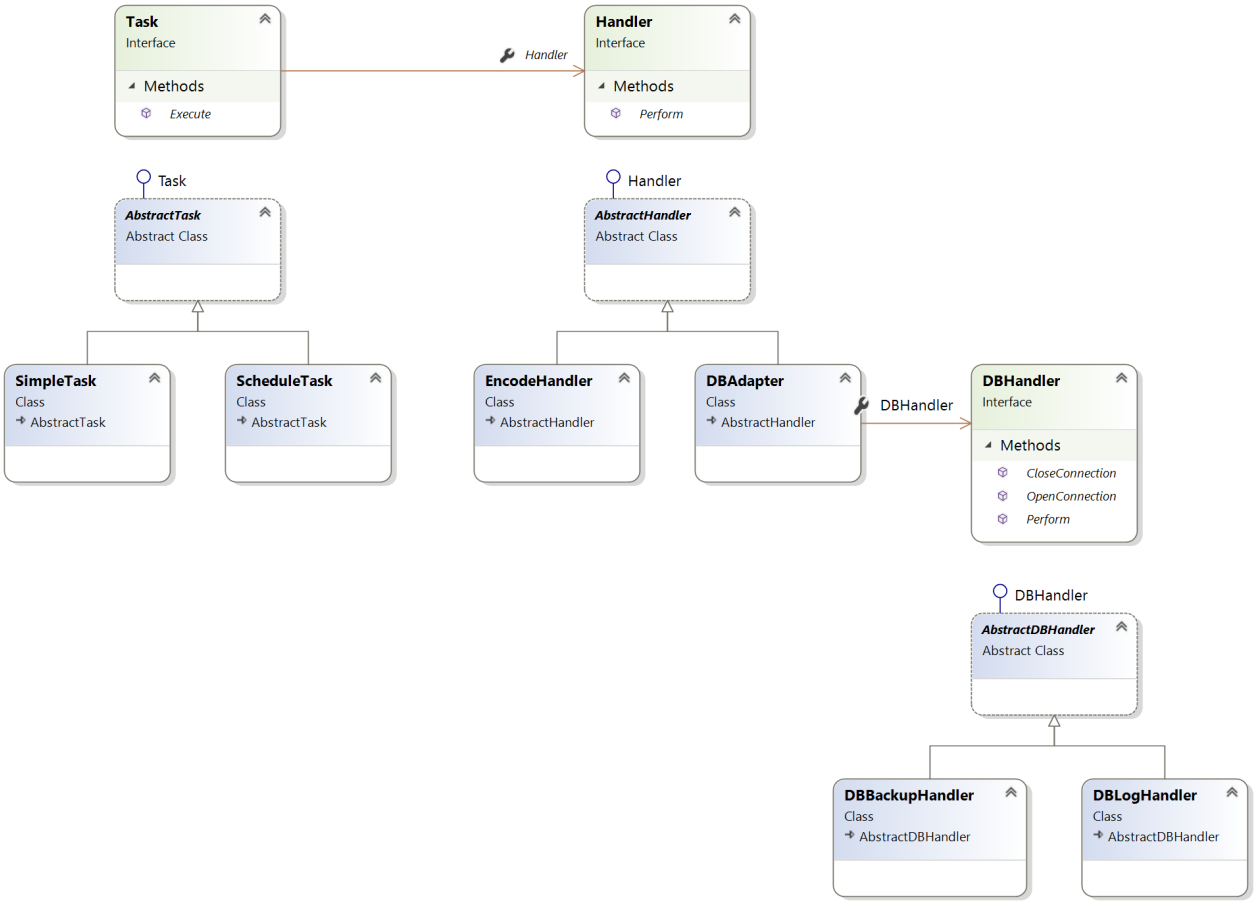
> Q : 為什麼寫 OOP 要考慮這麼多原則？

A : 理論上無論是 `SOLID` 原則也好，`Design Pattern` 也好，`Refactoring` 也好，只要程式寫的 夠久，為了讓程式碼好維護，最後每個人都會自然寫出這樣的程式碼。

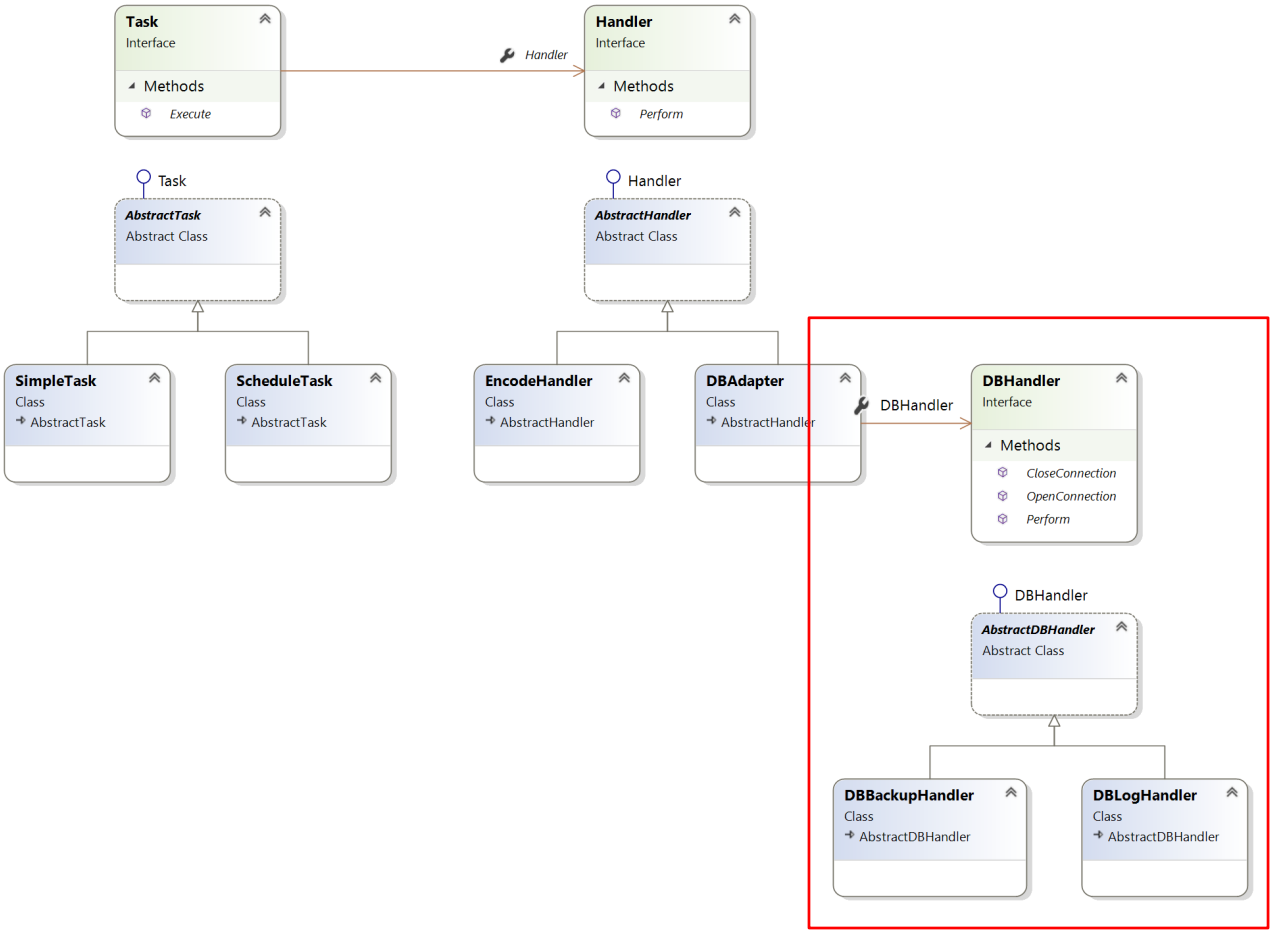
但 要寫多久 才會有這種領悟，則 因人而異，有人 20 幾歲就能悟道，也有人寫了一輩子程式也學不會 OOP，因此才有大師們將他們對 OOP 的 悟道經驗 歸納成一些 原則 與 模式，讓後輩能有更簡易的法門學習 OOP。

> Q : 如果有人想成為更棒的 PHP 工程師，你會怎麼建議？
>
> A : 學習出色的 Design Pattern。這不只適用在 PHP，你可以在任何程式語言使用這些pattern。尤其是 SOLID 原則。把這 6 個徹底學好，它會把你帶到新的境界，我每次寫code 幾乎都在想這 6 個原則。
>
> PHP Laravel Framework Creator : Taylor Otwell

Architecture



DBHandler Family

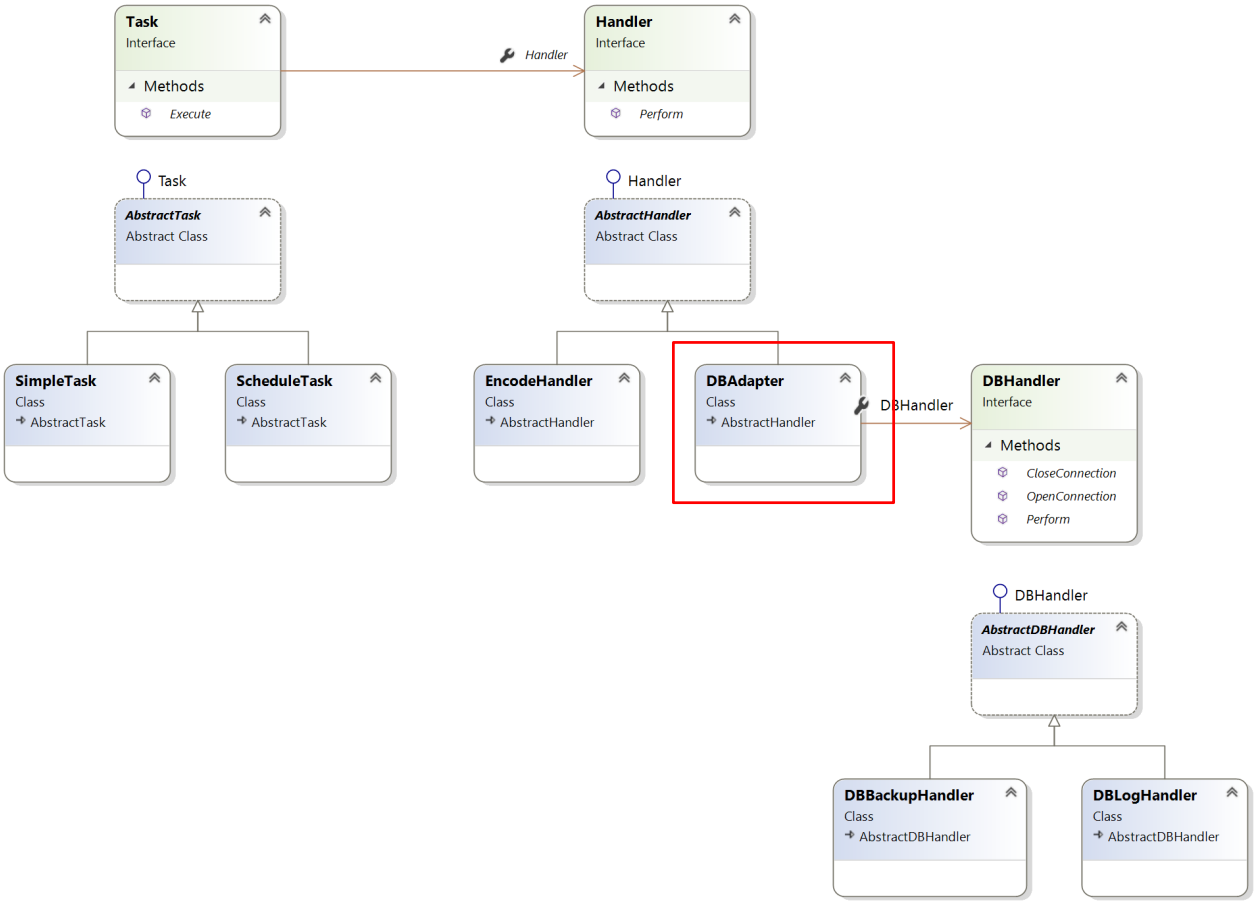


原本 `DBHandler` 應該直接實作 `Handler` interface 的 class，如同 homework 3 的 `EncodeHandler` 一樣，但因為：

- 新增了 `寫入` log 需求，因為 `寫入 table 不同`，因此實際上會有 `DBBackupHandler` 與 `DBLogHandler` 兩個不同 class
- 由於不排除有新的寫入資料庫需求，根據 `建立 Interface 增加擴充點` 的技巧，我們將 `DBBackupHandler` 與 `DBLogHandler` 抽象化成 `DBHandler` interface

> 由於各程式語言與 framework 都有自己的資料庫存取架構，因此 `DBHandler` interface 的 method 不一定要與範例一樣，但唯一可確定的是：原本 `Handler` 只有一個 `Perform()` 一定不夠用，因問資料庫有自己額外的需求

DBAdapter



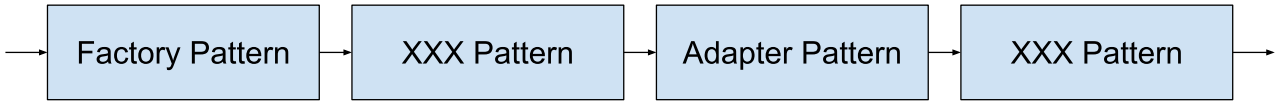
因為 `DBHandler` 的 interface 與 `Handler` interface 不同，在 `開放封閉` 的原則下，我們不希望去改變 `Handler` interface， 導致原本的 `多型` 機制被破壞，因此引入了 `DBAdapter` ，負責將 `Handler` interface 轉成 `DBHandler` interface。

如此 `DBHandler` interface 下的 class 可以繼續擴展，而原本在 `Handler` 的 `抽象` 與 `多型` 體制下的程式碼也不用修改，符合 `開放封閉` 原則的要求。

> Adapter Pattern
>
> 將一個 class 的界面變成使用端所期待的另外一種界面，從而使原本因界面不匹配而無法一起合作的 class 都能在一起工作
>
> 白話 : 將 interface 做轉換，讓不同 interface 的家族也能一起合作

> OOP 心法

> `增加界面，幫助擴展` 讓我們將來的需求能以此 interface 為擴展點加以展開，因此實務的 OOP 常常這樣使用 interface :



- `Factory Pattern` : 負責選擇此 interface 下 `適合` 的 class
- `XXX Pattern` : 根據需求選擇適合的 pattern
- `Adapter Pattern` : 若 interface 不符合需求，轉成 `適合` 的 interface

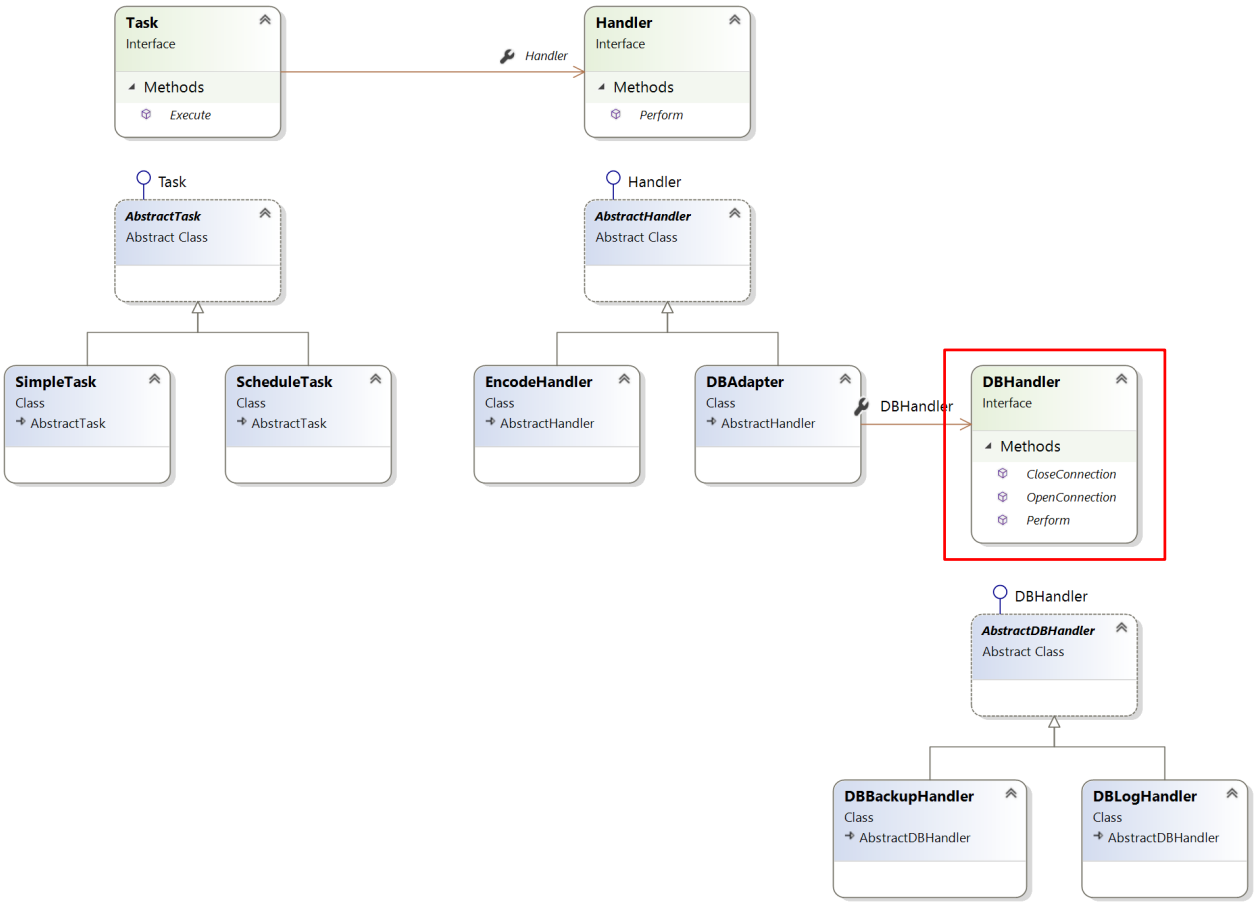
`Factory` 與 `Adapter` 是 OOP 最常用的 pattern，以 `Factory` 當先發，以 `Adapter` 當終結，或者再繼續串其他 `Pattern` 。

舉個生活上的例子，新的 notebook 幾乎都不提供 VGA 界面，只提供 HDMI 或 mini Display Port 界面，但大部分的投影機都還只支援 VGA，我們不可能對 notebook 加以修改，因為一修改就破壞保固了，因此我們會去買 `VGA Adapter` 轉成 VGA，而不是去修改 notebook，因為硬體使用 `Adapter Pattern` 與 `開放封閉原則` 。

軟體設計也該如此，既然 interface 已經決定，就不該 `隨便修改` interface，而是應該使用 `軟體 Adapter`，將界面加以轉換，這樣才能在原本程式不被修改的情況下，擴充其他新功能，而不是去修改原本的程式碼。

Implementation

DBHandler



DBHander.cs

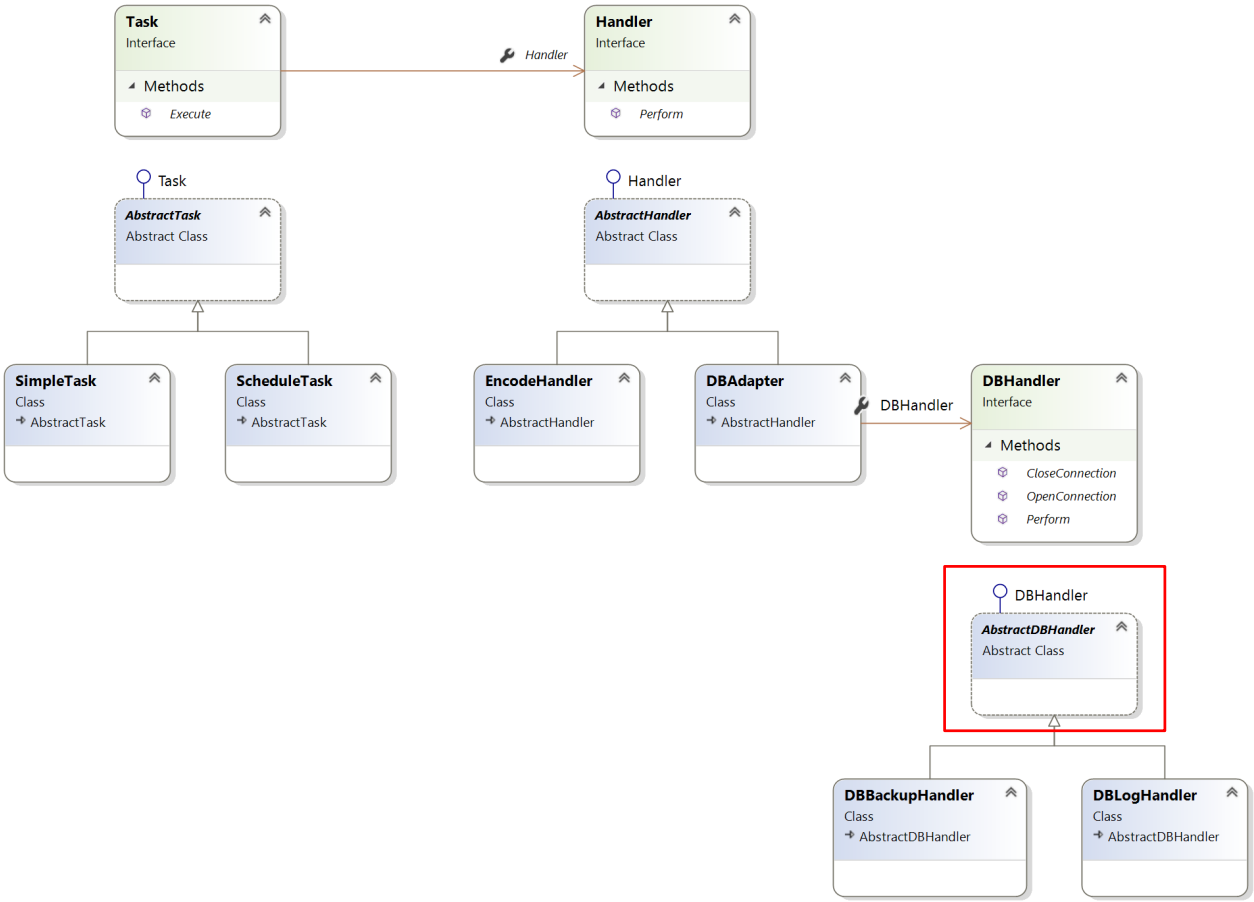
```
public interface DBHandler
{
    void OpenConnection();
    byte[] Perform(Candidate candidate, byte[] target);
    void CloseConnection();
}
```

> 所有與資料庫相關 handler 的抽象化 interface，使用端只依賴此 interface，而不會依賴實際 handler

- `Perform()`：與原本 `Handler` interface 的 `Perform()` 完全一樣
- `OpenConnection()`：負責建立 database connection
- `CloseConnection()`：負責關閉 database connection

> 再次強調 `DBHandler` interface 可以不用與範例一樣，可以自己根據 framework / ORM 定義適合的 method

AbstractDBHandler



AbstractDBHandler.cs

```
1  public abstract class AbstractDBHandler : DBHandler
2  {
3      public virtual byte[] Perform(Candidate candidate,
   byte[] target)
4      {
5          ...
6      }
7
8      public virtual void OpenConnection()
9      {
10         ...
11     }
12
13     public virtual void CloseConnection()
14     {
15         ...
16     }
17 }
```

> 所有 `DBHandler` 共用程式碼處

第 3 行

```
1  public virtual byte[] Perform(Candidate candidate, byte[]
   target)
2  {
3      ...
4  }
```

將 `Perform()` 開成 `virtual`，一些每個 `DBHandler` 都會執行的程式碼可寫在父類別的 `Perform()`。

第 8 行

```
1  public virtual void OpenConnection()
2  {
3      ...
4  }
```

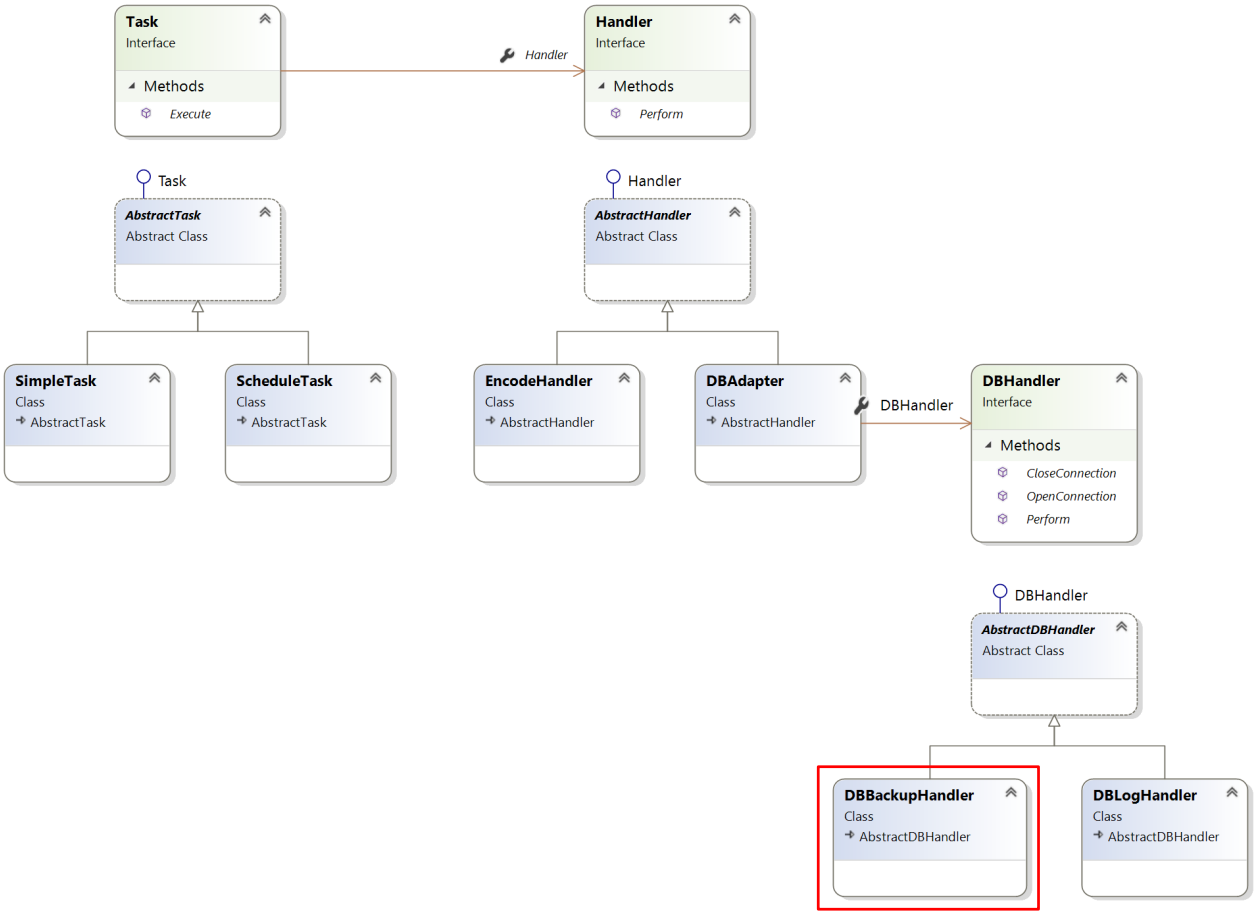
每個 `DBHandler` 都需要開啟 database connection，可將共用程式碼寫在 `AbstractDBHandler` 的 `OpenConnection()`。

13 行

```
1  public virtual void CloseConnection()
2  {
3      ...
4  }
```

每個 `DBHandler` 都需關閉 database connection，可將共用程式碼寫在 `AbstractDBHandler` 的 `CloseConnection()`。

DBBackupHandler



DBBackupHandler.cs

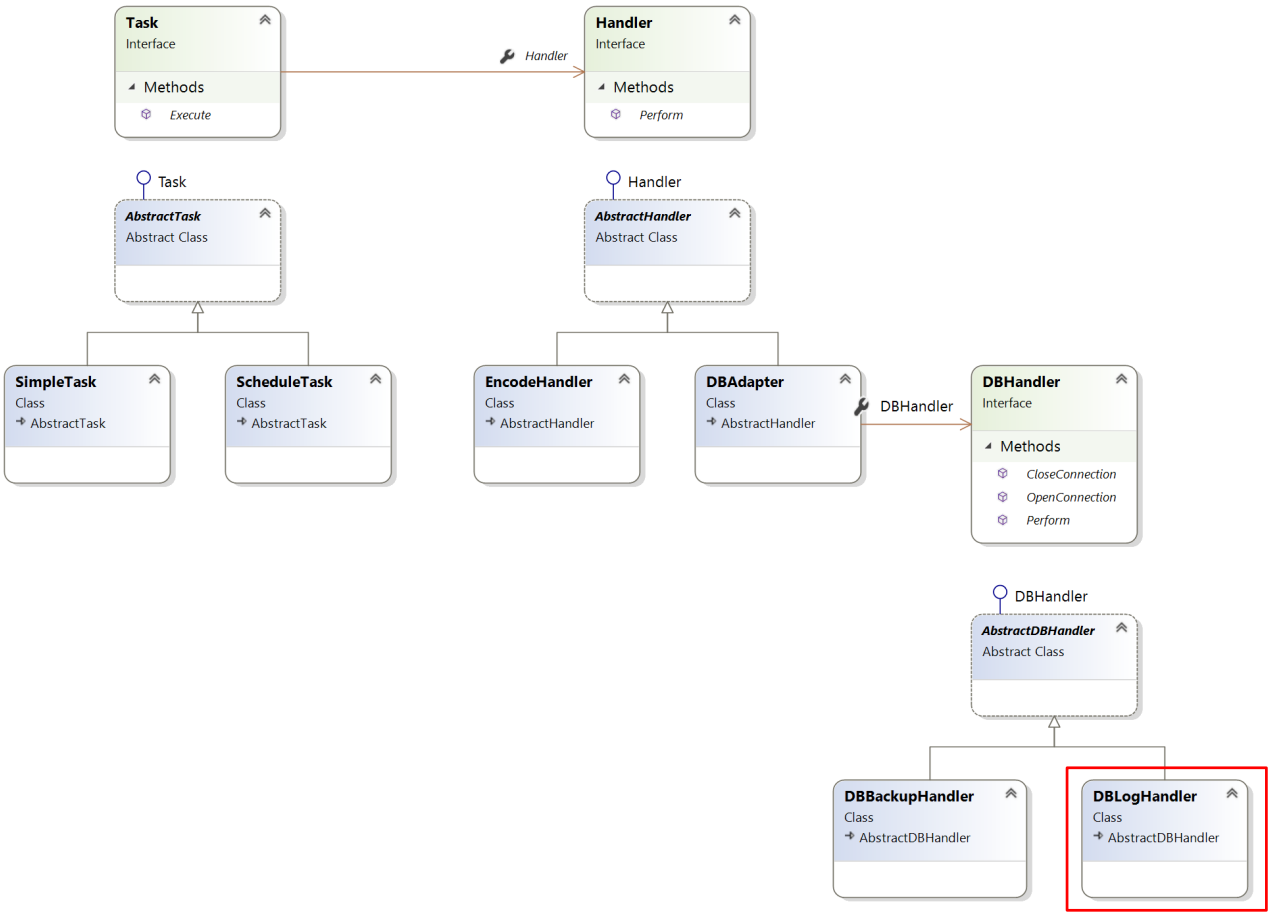
```
public class DBBackupHandler : AbstractDBHandler
{
    public byte[] Perform(Candidate candidate, byte[] target)
    {
        ...

        return target;
    }
}
```

> 實際將 `byte[]` 存進 `MyBackup` table

private method 部份請自行設計。

DBLogHandler



DBLogHandler.cs

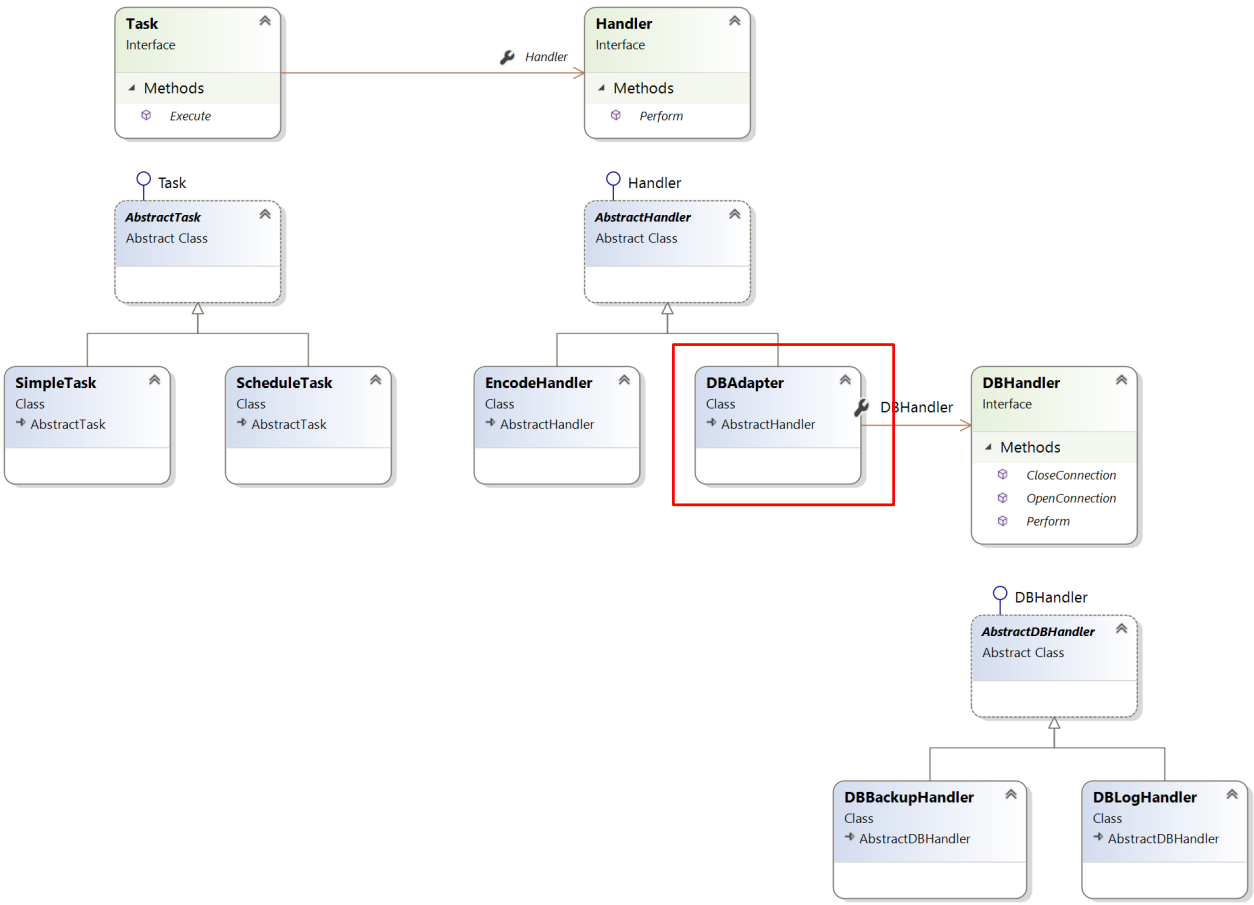
```
public class DBLogHandler : AbstractDBHandler
{
    public byte[] Perform(Candidate candidate, byte[] target)
    {
        ...

        return target;
    }
}
```

> 實際將 log 存進 `MyLog` table

private method 部份請自行設計。

DBAdapter



DBAdapter.cs

```
1  public class DBAdapter : AbstractHandler
2  {
3      DBHandler backupHandler = new DBBackupHandler();
4      DBHandler logHandler = new DBLogHandler();
5
6      public override byte[] Perform(Candidate candidate,
   byte[] target)
7      {
8          base.Perform(candidate, target);
9          this.SaveBackupToDB();
10         this.SaveLogToDB();
11
12         return target;
13     }
14
15     private void SaveBackupToDB()
16     {
17         this.backupHandler.OpenConnection();
18         this.backupHandler.Perform();
19         this.backupHandler.CloseConnection();
20     }
21
22     private void SaveLogToDB()
23     {
24         this.logHandler.OpenConnection();
25         this.logHandler.Perform();
26         this.logHandler.CloseConnection():
27     }
28 }
```

> 將 `Handler` interface 轉成 `DBHandler` interface

第 3 行

```
DBHandler backupHandler = new DBBackupHandler();
DBHandler logHandler = new DBLogHandler();
```

建立 `DBBackupHandler` 與 `DBLogHandler` ？

> Q：這裡該使用 `new` 直接耦合 `DBBackupHandler` 與 `DBLogHandler` 嗎？

A：雖然我們有抽 `DBHandler` interface，但目前是一起使用 `DBBackupHandler` 與 `DBLogHandler`，也就是並不是 `DBBackupHandler` 與 `DBLogHandler` 二擇一，因此不用使用 `Factory`，直接 `new` 即可。

第 6 行

```
public override byte[] Perform(Candidate candidate, byte[] target)
{
    base.Perform(candidate, target);
    this.SaveBackupToDB();
    this.SaveLogToDB();

    return target;
}
```

呼叫 `SaveBackupToDB()` 與 `SaveLogToDB()` 分別執行 `備份進資料庫` 與 `寫 log 進資料庫`。

15 行

```
1  private void SaveBackupToDB()
2  {
3       this.backupHandler.OpenConnection();
4       this.backupHandler.Perform();
5      this.backupHandler.CloseConnection();
6  }
```

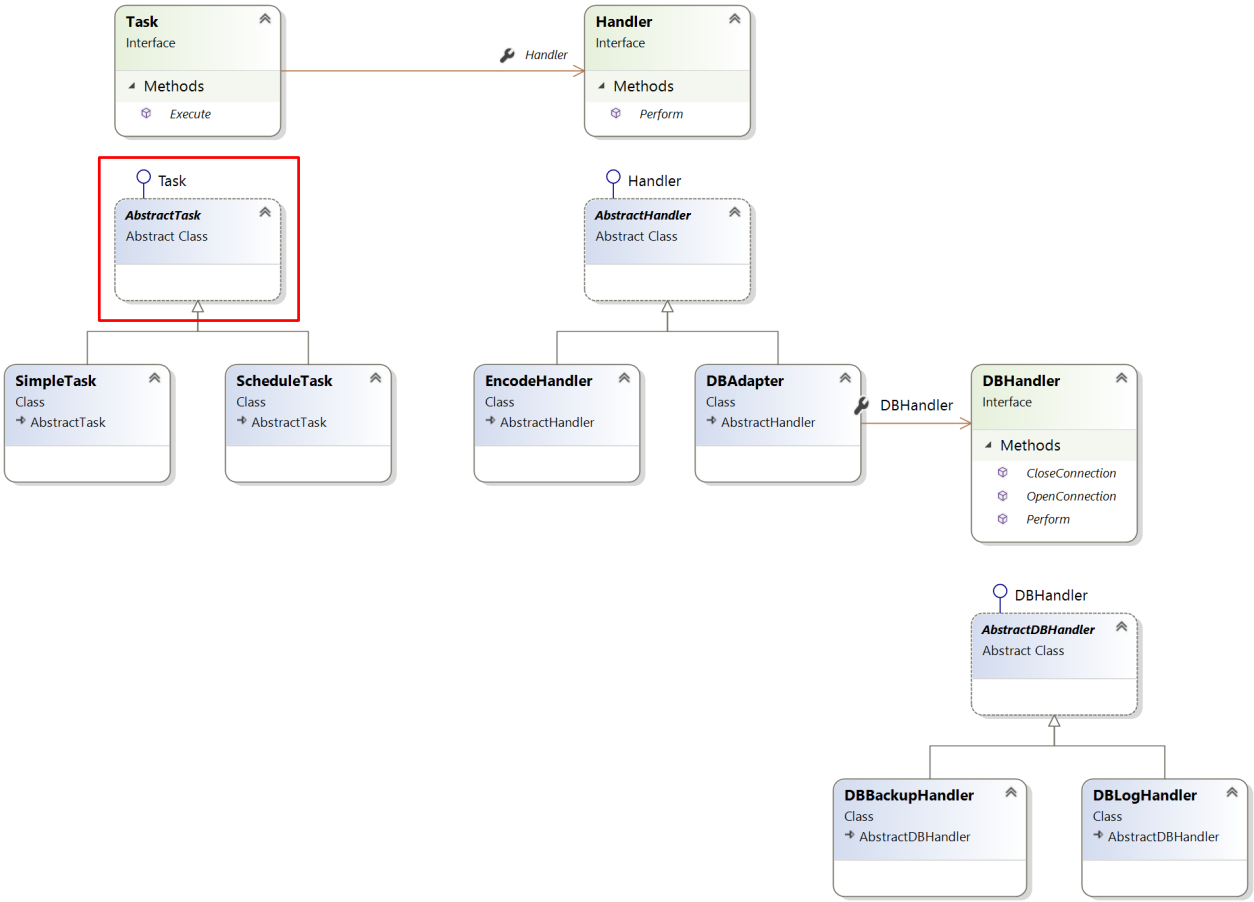
先開啟 database connection，在開始備份，最後關閉 database connection。

22 行

```
1  private void SaveLogToDB()
2  {
3       this.logHandler.OpenConnection();
4       this.logHandler.Perform();
5       this.logHandler.CloseConnection():
6  }
```

先開啟 database connection，在開始寫入 log，最後關閉 database connection。

AbstractTask



AbstractTask.cs

```
1  public abstract class AbstractTask : Task
2  {
3      protected void BroadcastToHandlers(Candidate candidate)
4      {
5          List<Handler> handlers =
   this.FindHandlers(candidate);
6
7          foreach(Handler handler in handlers)
8              target = handler.Perform(candidate, target);
9      }
10 }
```

由於 `Handler` interface 沒有被修改，因此原本設計的 抽象 與 多型 依然存在，使用端完全不用修改，符合 開放封閉 原則，

只要 interface 不被修改，原來客戶端的程式碼就不用修改，這才是使用 OOP 的目的。

handler_mapping.json

```
{
  "file": "FileHandler",
  "encode": "EncodeHandler",
  "zip": "ZipHandler",
  "directory": "DirectoryHandler",
  "db" : "DBAdapter"
}
```

唯一要修改的是 `handler_mapping.json` 設定檔，新增 `";db"; :";DBAdapter";`，當使用 `db` 為 key 時，對應 `DBAdapter`。

> OOP 心法
>
> `依賴抽象，封裝變化`，因此 `抽象` 不能被破壞，只要 `抽象` 也被修改，OOP 就崩潰了

> Q : 如何評判 OOP 寫的好不好？

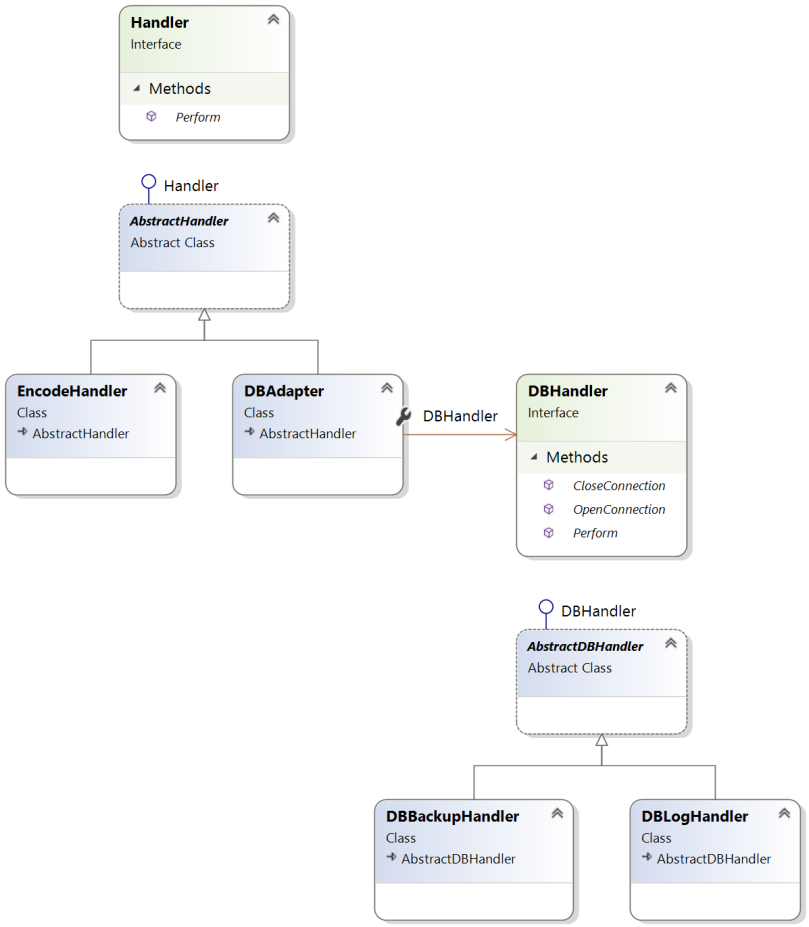
1. 若你的程式碼需要用到 `switch`
   - 還沒抽 `interface` 加以 `抽象化`
   - 唯一例外是 `Factory` 可以用 `switch`
   - `Refactoring 手法` : Replace conditional with Polymorphism
2. 若你的程式碼需要用到 `轉型`
   - 表示你 `多型` 沒用好
   - `抽象化` 已經被破壞
   - 或者該改用 `泛型`
3. 若你的程式碼需要用 `typeof()` 去判斷 `型別`

    - 尚未抽 interface 加以 `抽象化`
    - 表示你 `多型` 沒用好
    - `抽象化` 已經被破壞
    - `Refactoring 手法` : Replace type code with State / Strategy

Summary

- 我們不應該任意修改 interface，造成使用端的 breaking change
- 應該使用 `Adapter Pattern`，將不同的 interface 加以轉換，而不是任意修改 interface 破壞 `開放封閉`，這樣就失去使用 OOP 的意義
- OOP 常常是 `Factory Pattern` + `XXX Pattern` + `Adapter Patter` + `XXX Pattern` 的組合，以 `Factory` 為開始，以 `Adapter` 為結尾或繼續接其他 pattern
- 寫程式時，隨時檢查自己的 OOP 品質
    - 避免使用 `switch`
    - 避免使用 `轉型`
    - 避免使用 `typeof()`

Conclusion



- 程式語言不限，請依照 class diagram
  - `DBHandler` 的 interface 定義不見的要與 class diagram 相同，可自己根據 framework / ORM 使用上的方便，決定 `DBHandler` 的 interface
  - 新增 `DBHandler` interface、`AbstractDBHandler`、`DBBackupHandler` 與 `DBLogHandler` class
  - 新增 `DBAdapter` 將 `DBHandler` interface 轉成 `DBAdapter` interface