OOP #3

Sam Xiao, Oct.25, 2017

Goal

在 homework 1 與 homework 2 我們已經將 `讀取設定檔` 部份解決了，`MyBackup` 主要是在 `加密`、`壓縮`、`備份` ... 等功能，本次 homework 主要在實做這些部份。

Outline

Recap

Homework 2 :

- `繼承`：將 `ConfigManager` 與 `ScheduleManager` 抽象化成 `JsonManager`
- `封裝`：將 `ConfigManager` 與 `ScheduleManager` 封裝在 `MyBackupService` 內
- `多型`：將 `ConfigManager` 與 `ScheduleManager` 的 `讀取 JSON` 部份以 `ProcessJsonConfig()` 多型使用
- `最小知識原則`：使用端應該對物件有最少的了解
- `開放封閉原則`：對於擴展是開放的，對於修改是封閉的
- `DRY 原則`：相同的程式碼只寫一次

User Story

目前 user 處理方式有 `壓縮`、`加密`、`儲存到其他目錄`，`儲存到資料庫` 4 種方式，未來不排除有新的方式會加入。

> `處理方式` 屬於 `易於變動` 部份。

> OOP 心法
>
> 若你對 domain 與需求夠熟，一開始就可以發現 `易於變動` 的部份，可以採用 `預先架構` 方式，將 `易於變動` 的部份加以 `抽象化`
>
> 若你對 domain 與需求沒那麼熟，一開始沒有發現 `易於變動` 的部份，可以採用事後 `重構` 方式，將 `易於變動` 的部份加以 `抽象化`
>
> OOP 就是 `依賴抽象，封裝變化`，所以將 `易於變動` 部份加以 `抽象化` 是一定要做的，只是要 `預先架構` 先做，還是等 `重構` 後做而已

```
switch (theDestination)
{

case ToCompress:
    DoCompressFile();
    break;
case ToEncode:
    DoEncodeFile();
    break;
case ToDatabase :
    SaveToDatabase();
    break;
case ToDirectory :
    SaveToSpecificDirectory();
    break;

}
```

- 每加一種新的 `處理方式` ，就要去加一個 `case`
- 新加入的程式碼會不會把原來的程式碼改壞？
- 這些 `處理方式` 並不是 `互斥` 的，而是可以 `排列組合` 。如 `加密` 後再 `壓縮` ，最後再 `儲存到資料庫`

> 假如只用 `switch` 與 `if ... else` 去寫程式，雖然可以達到需求，但程式碼中將充滿許多重複的地方，很容易造成 bug 也不容易維護。

期望有一致方法處理各種不同方式，不用每加一種方式就必須修改程式碼。

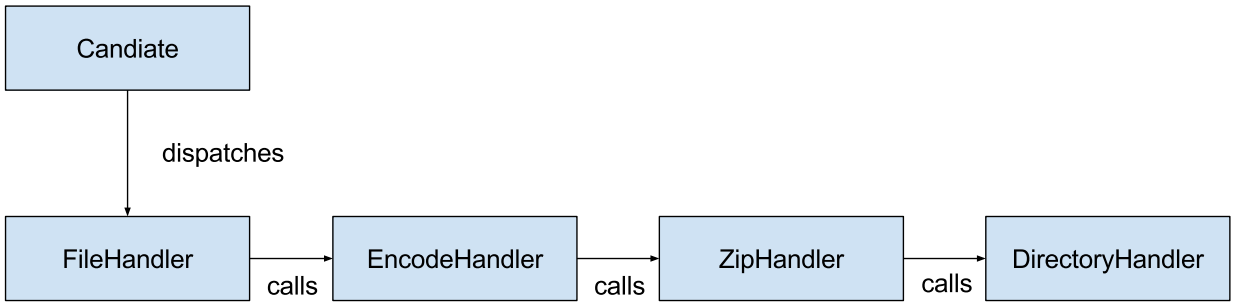
期望不同的方式能以 `排列組合` 的方式組合在一起。

Task

`壓縮` 、 `加密` 、 `儲存到其他目錄` ， `儲存到資料庫` 等 4 種方式，表面上看起來是不同的，但仔細想想，他們的行為其實是一致的：

> 接受 `上一個` 的 處理方式所處理過的資料，處理完後，再交給 `下一個` 的處理方式處理

以 `*.cs` 檔案格式為例，我們希望先 `編碼`，再 `壓縮` ，最後 `複製` 到其他目錄。



- `Candiate`：描述待處理檔案的資訊
- `FileHandler`：將 `檔案` 轉成 `byte[]`，或將 `byte[]` 轉成 `檔案`
- `EncodeHandler`：將 `檔案` 加以 `編碼`
- `ZipHandler`：將 `檔案` 加以 `壓縮`
- `DirectoryHandler`：將 `檔案` 複製到 `指定目錄`

我們可將各種 `處理方式` 抽象化成 handler，每個 handler 接受 `上一個` handler 處理的結果 `byte[]`，處理完後再將 `byte[]` 交給 `下一個` handler 處理。

如此每個 handler 可以獨立開發測試，卻又可以很巧妙的一起 `排列組合` 運作。

MyBackupService.cs

```
public class MyBackupService
{

public void DoBackup()
{
    List<Candiate> candidates = this.FindFiles();

    foreach(Candiate candidate in candidates)
    {
         this.BroadcastToHandlers(candidate);
    }
}

private void BroadcastToHanders(Candidate candidate)
{
    List<Handler> handlers = this.FindHandlers(candidate);

    foreach(Handler handler in handlers)
    {
        target = handler.Perform(candidate, target);
    }
}

```

第 3 行

```
public void DoBackup()
{

List<Candiate> candidates = this.FindFiles();

foreach(Candiate candidate in candidates)
{
    this.BroadcastToHandlers(candidate);
}

```

`DoBackup()` 根據 `FindFiles()` 得知總共有哪些 candidate ，再將每個 candidate 以 `廣播` 的方式，通知所有 handler 進行處理。

> `FindFiles()` 將於下一次 homework 討論

13 行

```
private void BroadcastToHanders(Candidate candidate)
{

List<Handler> handlers = this.FindHandlers(candiate);
byte[] target = null;

foreach(Handler handler in handlers)
{
    target = handler.Perform(candidate, target);
}

}
```

`BroadcastToHandlers()` 根據 `FindHandlers()` 與 `candidate` 得知總共有哪些 handler ，再一一執行，並將處理完的資料 `target` ，交給下一個 handler 處理。

將所有處理方式都抽象化成 handler 後，使用端不必再判斷檔案類型呼叫各種處理方式。

將來有新的處理方式，使用端程式碼也不用修改，使用 `多型` 處理即可。

> 稍後將討論 `FindHandlers()`

> 開放封閉原則 (OCP : Open Closed Principle)
>
> 對於擴展是開放的，對於修改是封閉的

> Q : 如何得知 `Candidate` 需要哪些 handler ?

config.json

```
{
"configs": [
    {
        "ext": "cs",
        "location": "c:\Projects",
        "subDirectory": true,
         "unit": "file",
         "remove": false,
         "handlers": ["zip", "encode"],
         "destination": "directory",
         "dir": "c:\MyArchieves",
         "connectionString": ""
    },
]
}
```

其中 `FileHandler` 為每種檔案格式都需要，接下來由 `handlers` key 決定套用哪些 handler，最後由 `destination` key 決定 `DirectoryHandler`。

> 在 homework 1 的 `config.json` 的 key 為 `handler`，且 value 為 `字串`，因為需求改變，key 改成 `handlers`，value 改成 `陣列` ，支援多個 handler 排列組合。

Architecture

Handler

MyBackupService.cs

```
private BroadcastToHanders(Candidate candidate)
{

List<Handler> handlers = this.FindHandlers(candiate);
byte[] target;

foreach(Handler handler in handlers)
{
    target = handler.Perform(candidate, target);
}

}
```

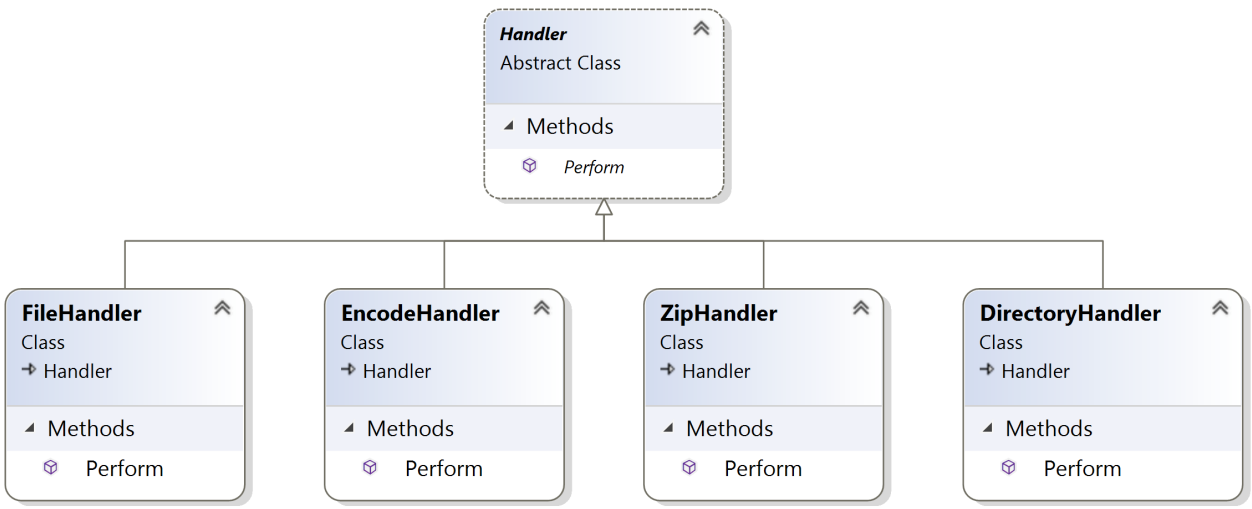
由於 處理方式 與 handler 都是預期 易於變動 的部份，因此將 handler 加以 抽象化 對用戶端是最好的方式：

- 用戶端不必再判斷 handler 的型別
- 用戶端永遠呼叫相同的 method

也就是 OOP 的 多型。

上一次上課曾經提到：

> 多型 為 OOP 的結果，以 繼承 與 封裝 為手段，目的就是 依賴抽象，封裝變化



藉由 繼承，我們的確可以將所有的 handler 加以抽象化成 `Handler` class。

而且就目前的需求而言，只使用 `繼承` 完成 `抽象` 是夠用的。

> 早期的 OOP 強調 `繼承` ，以 `繼承` 達成 `程式碼共用` 與 `抽象`，但近代 OOP，則強調 interface，所謂的 `interface based design` 或 `interface based programming`，尤其 Java 正式把 interface 成為 keyword，而 Design Pattern 與 Refactoring 主要也是以 interface 實踐。

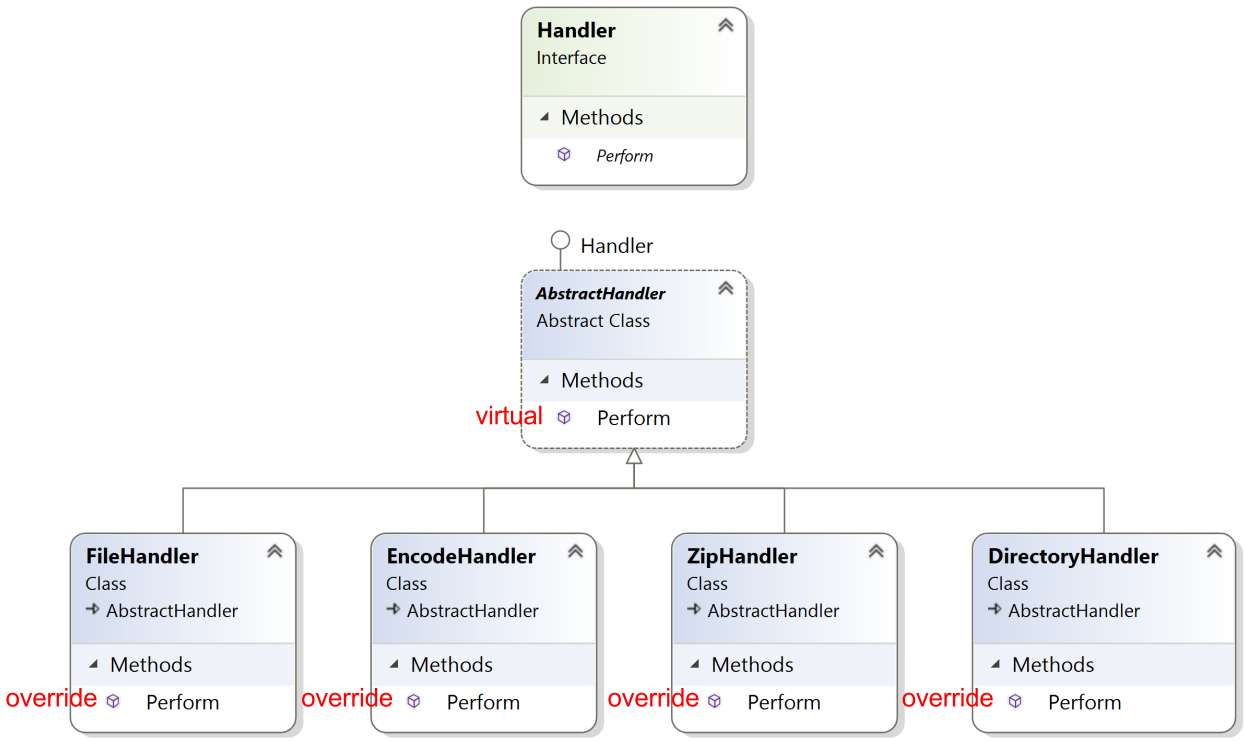
> Q : 繼承有什麼缺點嗎？為什麼要使用 interface 取代 `繼承` ？

- `繼承` 是 `侵入性` 與 `強耦合`
- `侵入性` : 子類別可能被強迫繼承不需要的特性
- `強耦合` : 當父類別被修改時，也會影響到子類別
- `單一繼承` : 除了 C++ 外，所有的近代 OOP 語言都只能 `單一繼承` ，若需求改變，可能造成父類別肥大 (違反 `界面隔離原則`)
- `可實現多個 interface` : 近代 OOP 都允許 implement 多個 interface，若需求改變，可以新增 interface，不會造成父類別肥大

> Q : Interface 有缺點嗎？

- 只能宣告 method 與 signature，不能有 `實做`

因此實物上常常使用 `interface + abstract class` 的手法，讓架構同時具有 `可實現多個 interface` 與 `能夠抽出相同程式碼` 的優點。



對用戶端的 `抽象` 為 `Handler` interface，`AbstractHandler` 目的只是提供 `FileHandler`、`EncodeHandler`、`ZipHandler` 與 `DirectoryHandler` 有 `抽出共用程式碼` 的地方。

此外，實務上會在 abstract class 將 method 設定為 `virtual`，將程式碼共用的部份寫在父類別的 method。

如 `初始化` 的程式碼通常會相同，則可寫在父類別的 `virtual` method，則子類別的 `override` method 只須以 `base.Perform()` 的方式呼叫父類別的 `Perform()` 即可，可避免程式碼重複。

由於用戶端看到的是 interface，而不是 abstract class，將來若有新的需求，可繼續新增 interface，使用端不會看到一個肥大的 abstract class。

> 界面隔離原則 (ISP : Interface Segregation Principle)
>
> 客戶端不應該倚賴它不需要的 interface
>
> Class 間的依賴應該建立再最小 interface 上

Factory

每個檔案格式的 `處理方式` 都不一樣，該如何根據根據不同檔案類型，建立不同的 `List<Handler>`，也就是 `FindHandlers()` 該如何設計？

第一個版本

MyBackupService.cs

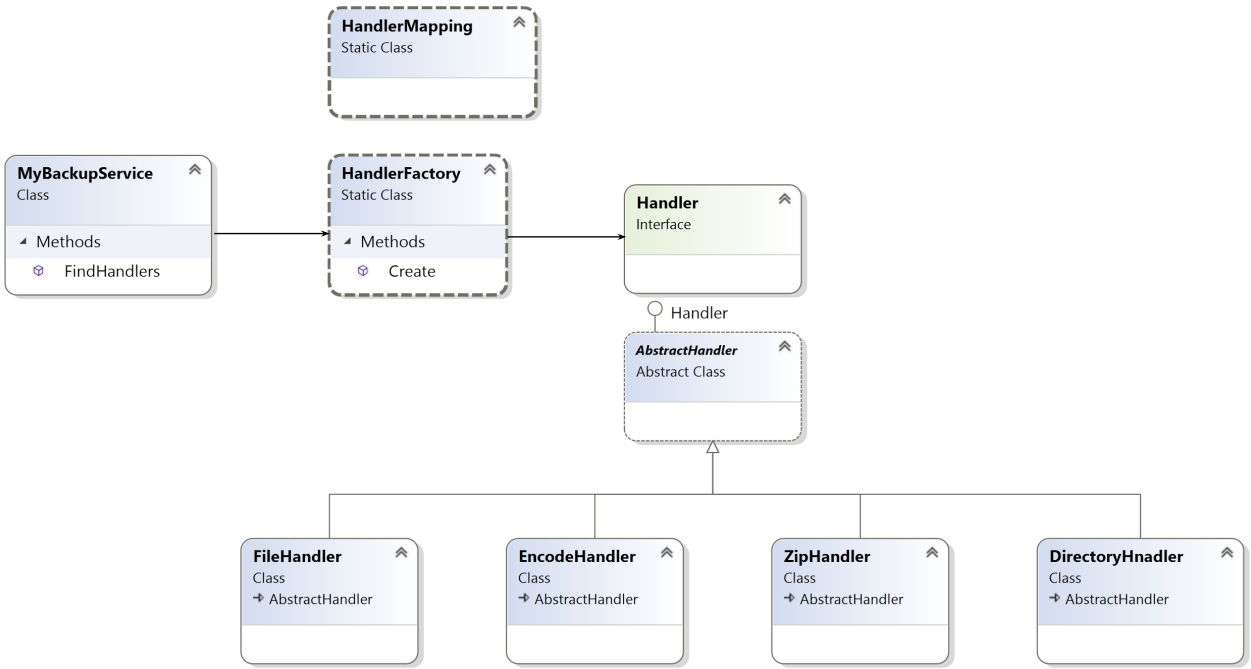
```
public class MyBackupService
{

private List<Handler> FindHandlers(Candidate candidate)
{
    List<Handler> handlers = new List<Handler>();
    handlers.Add(new FileHandler());

    ...

    foreach(string handler in config.Handlers)
    {
        if (handler == "encode") {
            handlers.Add(new EncodeHandler());
        }
        else if (handler == "zip") {
            handlers.Add(new ZipEncoder());
        }
    }

    if (config.Destination == "directory") {
        handlers.Add(new DirectoryHandler());
    }

    return handlers;
}
}
```

這個版本雖然可行，但有一些缺點：

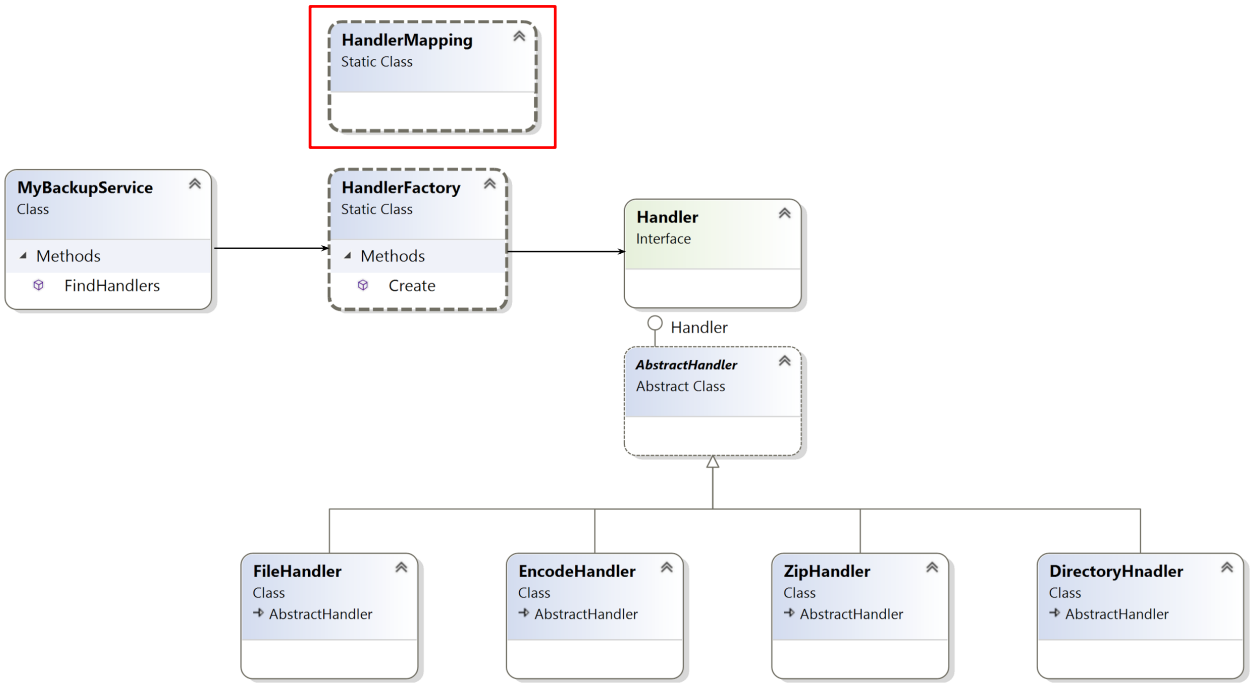
- `encode`、`zip` 與 `directory` 直接 hardcode
- 每加一種新的 `處理方式` ，就要去加一個 `if else`
- 新加入的程式碼會不會把原來的程式碼改壞？
- 這類根據 `字串` 建立不同 `物件` 的 `if else` 判斷，可能會重複散落在程式碼各處，出現重複的程式碼

> 我們需要一個專職建立 handler 的 Factory，才能避免以上問題

第二個版本



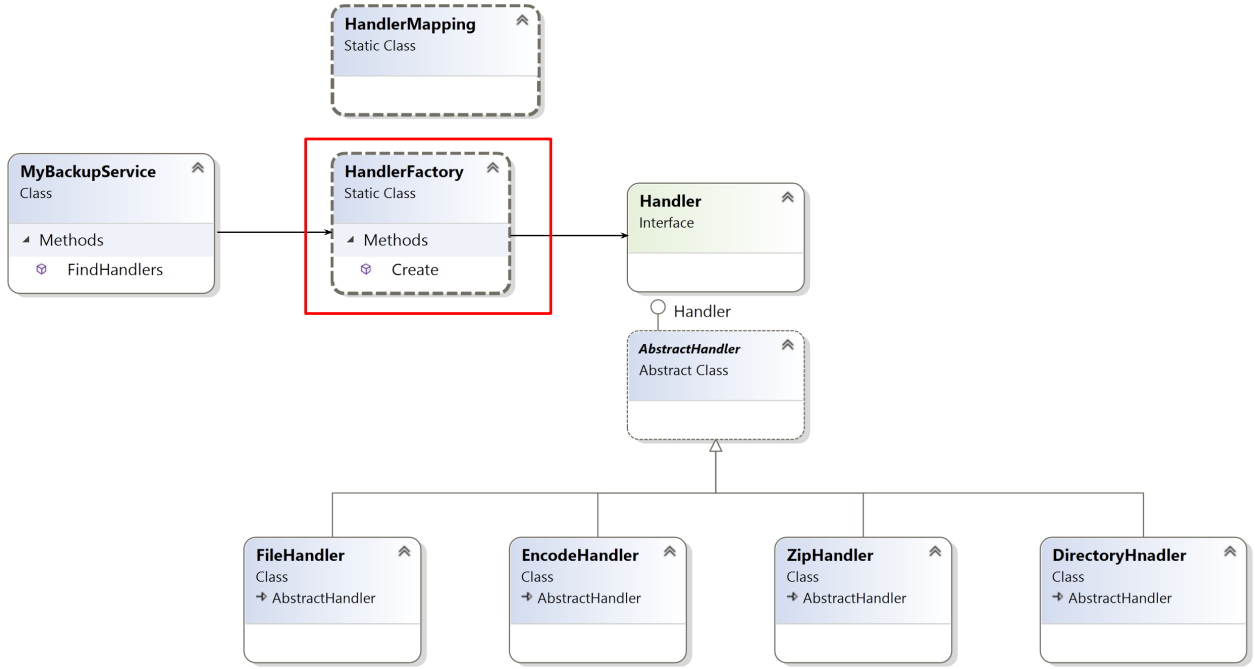
HandlerMapping.cs




```
public class HandlerMapping
{

  public static string FILE_HANDLER = "file";
  public static string ENCODE_HANDLER = "encode";
  public static string ZIP_HANDLER = "zip";
  public static string DIRECTORY_HANDLER = "directory";
}
```

為了避免 `file`、`encode`、`zip` 與 `directory` 等 hardcore 在 `MyBackupService` 中，我們另外新增 `HandlerMapping`，使用 static field 紀錄這些 key。

HandlerFactory.cs

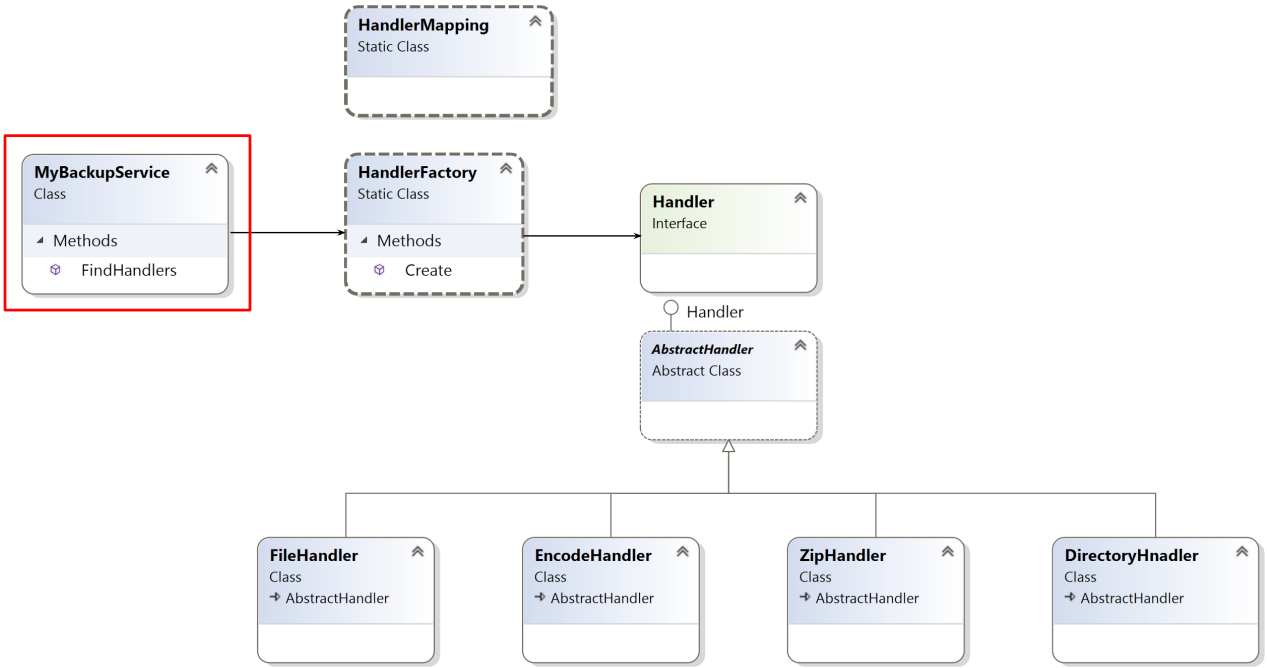



```
public class HandlerFactory
{

public static Handler create(string handler)
{
    if (handler == HandlerMapping.FILE_HANDLER) {
        return new FileHandler();
    }
    else if (handler == HandlerMapping.ENCODE_HANDLER) {
        return new EncodeHandler();
    }
    else if (handler == HandlerMapping.ZIP_HANDLER) {
        return new ZipHandler();
    }
    else if (handler == HandlerMapping.DIRECTORY_HANDLER) {
        return new DirectoryHandler();
    }
}
}
```

新增 `HandlerFactory`，根據傳入 key 與 `HandlerMapping` 比對，決定要 `new` 那一個 handler。

> OOP 一般不建議使用 `static`，因為會喪失原本 OOP 的 `抽象` 與 `多型`，但 `Factory` 是例外，因為 `Factory` 不需要 `抽象` 與 `多型`，使用端可直接相依於 `Factory` 無仿。
>
> 只要 `Factory` 能傳回 `抽象` 與 `多型` 的物件即可，`Factory` 本身不需要 `抽象` 與 `多型`。

MyBackupService.cs



```
public class MyBackupService
{

private List<Handler> FindHandlers(Candidate candidate)
{
    List<Handler> handlers = new List<Handler>();
    handlers.Add(HandlerFactory.create('file');

    ...

    foreach(string handler in config.Handlers)
    {
        handlers.Add(HandlerFactory.create(handler));
    }

    handlers.Add(HandlerFactory.create(config.Destination));


    return handlers;
}
}
```

原本需要 `if else` 判斷與 `new` 的地方，全部改用 `HandlerFactory.create()`。

這樣有幾個好處：

- 對於 `MyBackupServce` 而言，不必知道 (相依) `FileHandler`、`EncodeHandler`、`ZipHandler` 與 `DirectoryHandler`，只須知道 `HandlerFactory.create()` 即可，符合 `最小知識原則` 要求
- 若將來新增其他 handler，`MyBackupService` 也不用修改，符合 `開放封閉原則` 要求
- 將 `if else` 集中在 `HandlerFactory` 內，避免 `if else` 散落各地將來難以維護

目前 `MyBackupService` 已經 `開放封閉`，但 `HandlerMapping` 與 `HandlerFactory` 還沒。

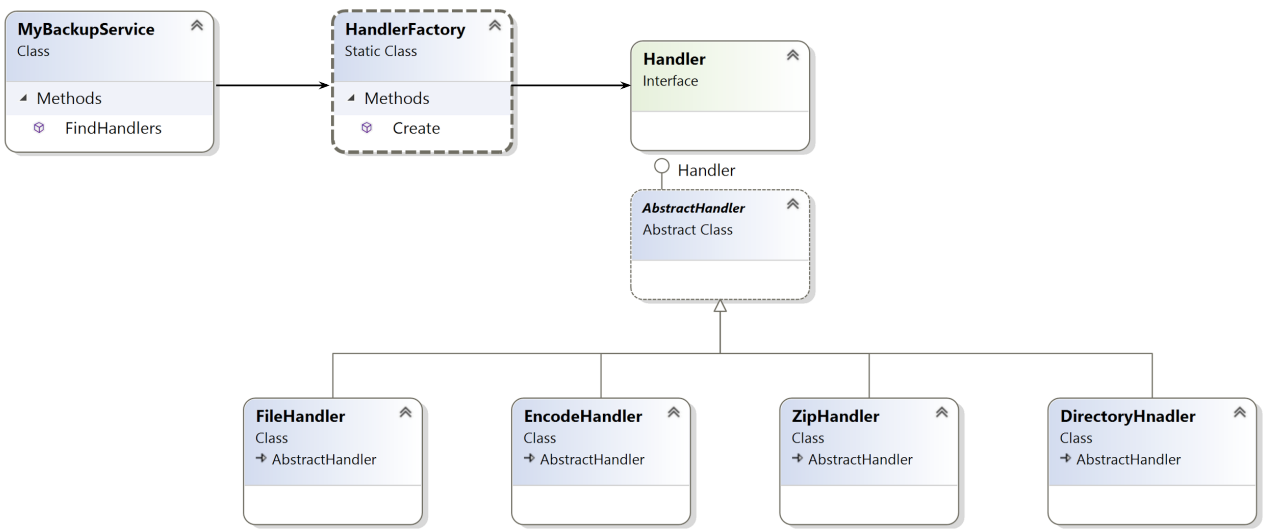
> Simple Factory 模式
>
> 使用 `Factory.Create()` 取代 `new`，將建立物件的工作封裝在 `Factory`

> OOP 心法
>
> 實物上使用 `if else` 難以避免，而 `if else` 通常也是違反 `開放封閉原則` 的地方，若無法完全避免使用 `if else`，最少要將 `if else` 的使用量降到最低，最好只限制在 `Factory` 能使用 `if else`，也就是只剩下 `Factory` 能違反 `開放封閉原則`，其他地方都應該遵守。

> Q : 原本的 `if else` 是在 `MyBacupService`，現在搬到 `HandlerFactory` 而已，`MyBackupService` 雖然 `開放封閉` 了，但換來 `HandlerMapping` 與 `HandlerFactory` 沒有 `開放封閉`，這樣真的比較好嗎？

目前的確 `HandlerMapping` 與 `HandlerFactory` 沒有 `開放封閉`，所以我們繼續 `重構` 下去。

第三個版本



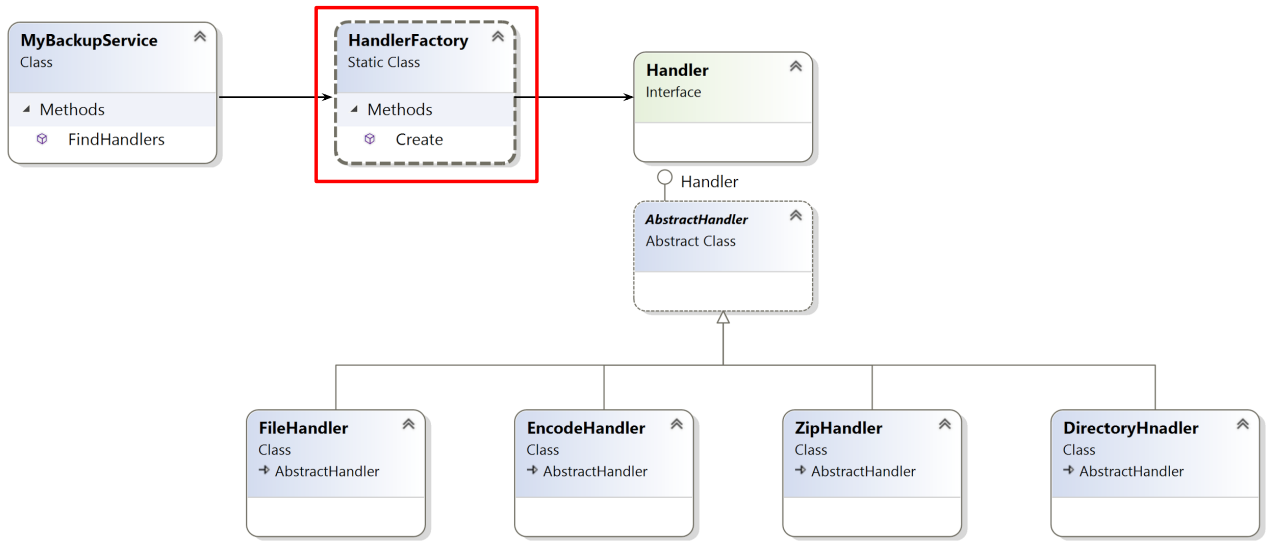
handler_mapping.json

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

將 `HandlerMapping.cs` 改成 `handler_mapping.json`，將來若要新增 handler，修改 `handler_mapping.json` 即可。

> 設定檔本來就是要拿來 `設定`，因此沒有 `開放封閉` 的問題

HandlerFactory.cs



```
public class HandlerFactory
{
    private static Dictionary<string, string> handlerDictionary;

    static HandlerFactory()
    {
        string jsonString = File.ReadAllText("handler_mapping.json");
        HandlerFactory.handlerDictionary = JsonConvert.DeserializeObject<Dictionary<string, string>>(jsonString);
    }

    public static Handler Create(string key)
    {
        return (Handler)Activator.CreateInstance("MyBackupService", HandlerFactory.handlerDictionary[key]).Unwrap();
        }
    }
}
```

第 3 行

```
private static Dictionary<string, string> handlerDictionary;
```

將 handler 的 mapping，由 `handler_mapping.json` 改用 Dictionary 存放。

因為 `create()` 為 static，因此所有資料也要搭配 static。

```
static HandlerFactory()
{
 string jsonString = File.ReadAllText("handler_mapping.json");
 HandlerFactory.handlerDictionary =
JsonConvert.DeserializeObject<Dictionary<string, string>>
(jsonString);
}
```

因為 `create()` 為 static，所以 constructor 也必須 static。

從 `handler_mapping.json` 讀進來成 string，再由 `JsonConvert.DeserializeObject()` 轉成 Dictionary。

11 行

```
public static Handler Create(string key)
{
  return (Handler)Activator.CreateInstance("MyBackupService",
HandlerFactory.handlerDictionary[key]).Unwrap();
}
```

由 `Activator.CreateInstance()` 直接透過 string 建立 object，其中第 1 個參數字串為 assembly name，第二個參數要 new 的 class name，也是 string。

但傳進來的 key 並不是我們要 new 的 class name，必須先透過 `handlerDictionary` 根據 key 找到其 class name，才能交給 `Activetor.CreateInstance()` 去建立 object。

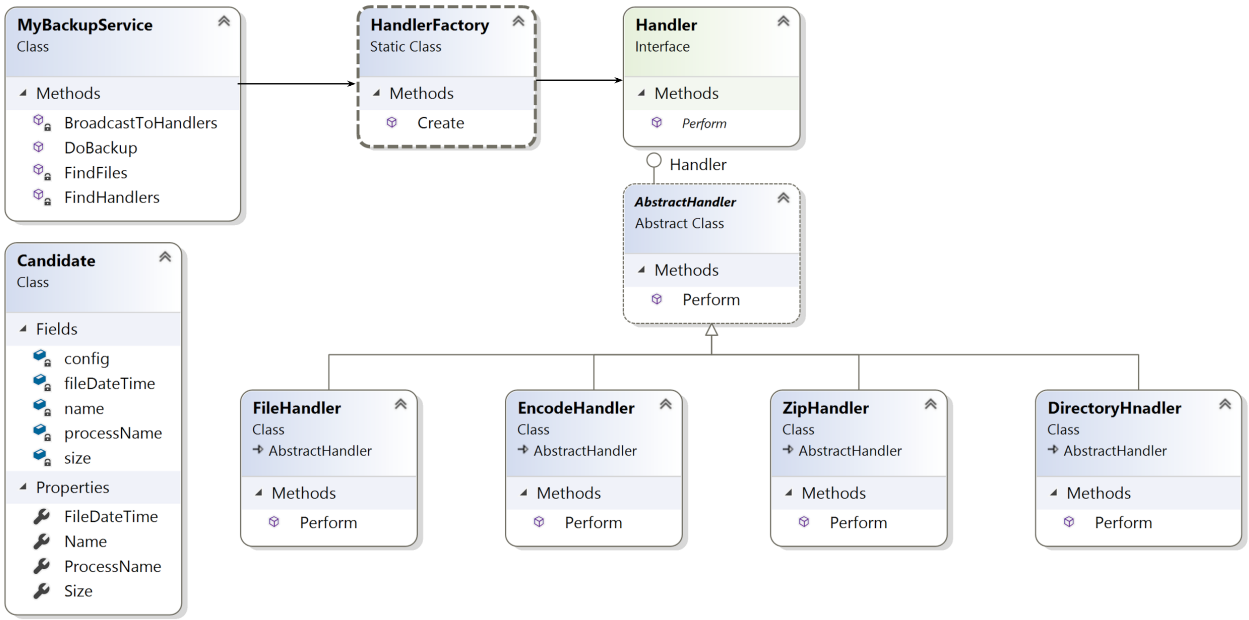
但 `CreateInstance()` 回傳的型別為 `ObjectHandle`，必須透過 `Unwrap()` 之後，才是 `object` 型別，最後再由 `object` 轉為 `Handler` 型別。

將來若有新的 handler，只要修改 `handler_mapping.json` 即可，因為 `HandlerFactory` 完全沒有任何 `if else`，因此也不用修改，達到 `開放封閉` 的要求。

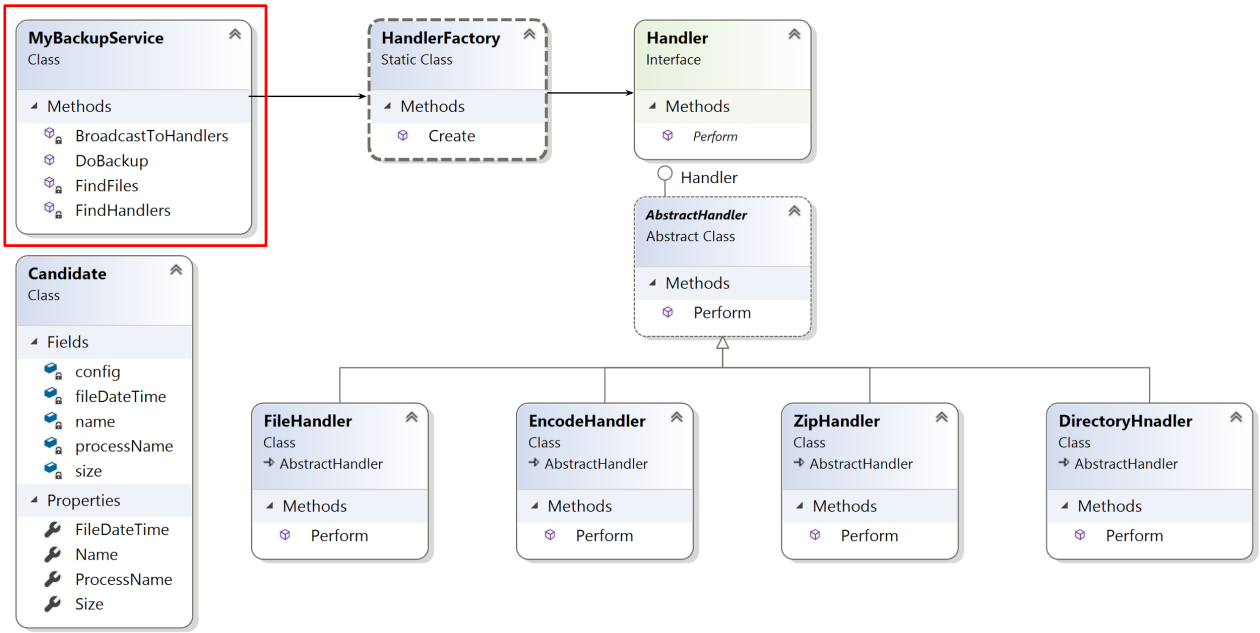
> Reflection
>
> 讓強型別的 C# 也能如弱型別的 PHP 一樣，由 string 就可以建立 object，因為實務上，我們在設定檔內的設定都是 string，能在 run time 用 string 建立 object 是最靈活的

> Reflection 讓我們完全可以使用 string 來操作物件，非常靈活，但相對的，也就喪失了 intellisense 與 compiler 檢查，算是讓強型別語言也能兼具弱型別的彈性，但用在 `Factory` 則非常合適

Implementation



MyBackupService.cs


MyBackupService
Class
Methods
BroadcastToHandlers
DoBackup
FindFiles
FindHandlers
HandlerFactory
Static Class
Methods
Create
Handler
Interface
Methods
Perform
Handler
AbstractHandler
Abstract Class
Methods
Perform
Candidate
Class
Fields
config
fileDateTime
name
processName
size
Properties
FileDateTime
Name
ProcessName
Size
FileHandler
Class
AbstractHandler
Methods
Perform
EncodeHandler
Class
AbstractHandler
Methods
Perform
ZipHandler
Class
AbstractHandler
Methods
Perform
DirectoryHnadler
Class
AbstractHandler
Methods
Perform

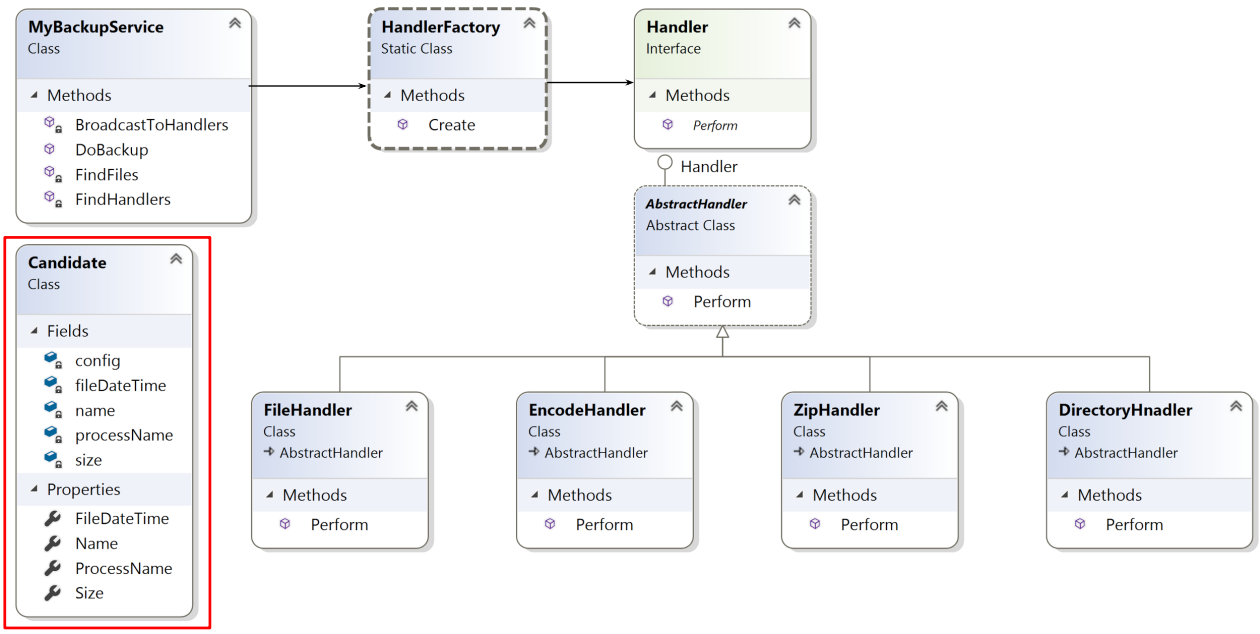
```
public class MyBackupService
{

public void DoBackup()
{
    List<Candiate> candidates = this.FindFiles();

    foreach(Candiate candidate in candidates)
    {
         this.BroadcastToHandlers(candidate);
    }
}

private void BroadcastToHanders(Candidate candidate)
{
    List<Handler> handlers = this.FindHandlers(candidate);

    foreach(Handler handler in handlers)
    {
        target = handler.Perform(candidate, target);
    }
}

private List<Candidate> FindFiles()
{
    // Homework 4
}

private List<Handler> FindHandlers(Candidate candidate)
{
    List<Handler> handlers = new List<Handler>();
    handlers.Add(HandlerFactory.create('file');

    ...


    foreach(string handler in config.Handlers)
```

```
36      {
37          handlers.Add(HandlerFactory.create(handler));
38      }
39
40      handlers.Add(HandlerFactory.create(config.Destination));
41
42      return handlers;
43  }
44
```

> 使用端的統一入口

- 實做 `DoBackup()`
- 新增 private 的 `BroadcastToHanders()` 、 `FindFiles()` 與 `FindHandlers()`
- `FindFiles()` 是下次 homework 重點
- 本次 homework 要實做 `BroadcastToHandlers()` 與 `FindHandlers()`

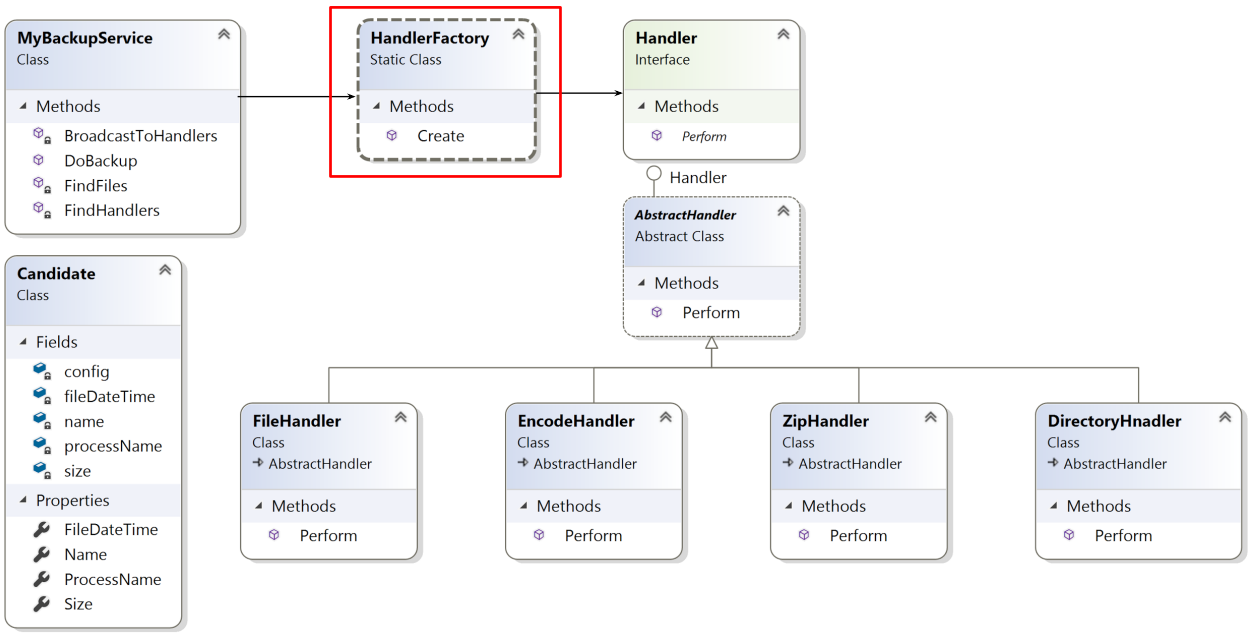
Candidate.cs



> 描述待處理檔案的資訊

- `Config`：所根據的 `Config` 物件，由 constructor 傳入
- `FileDateTime`：檔案的日期與時間
- `Name`：檔案名稱
- `ProcessName`：處理檔案的 process，以後會用到
- `Size`：檔案 size

HandlerFactory.cs

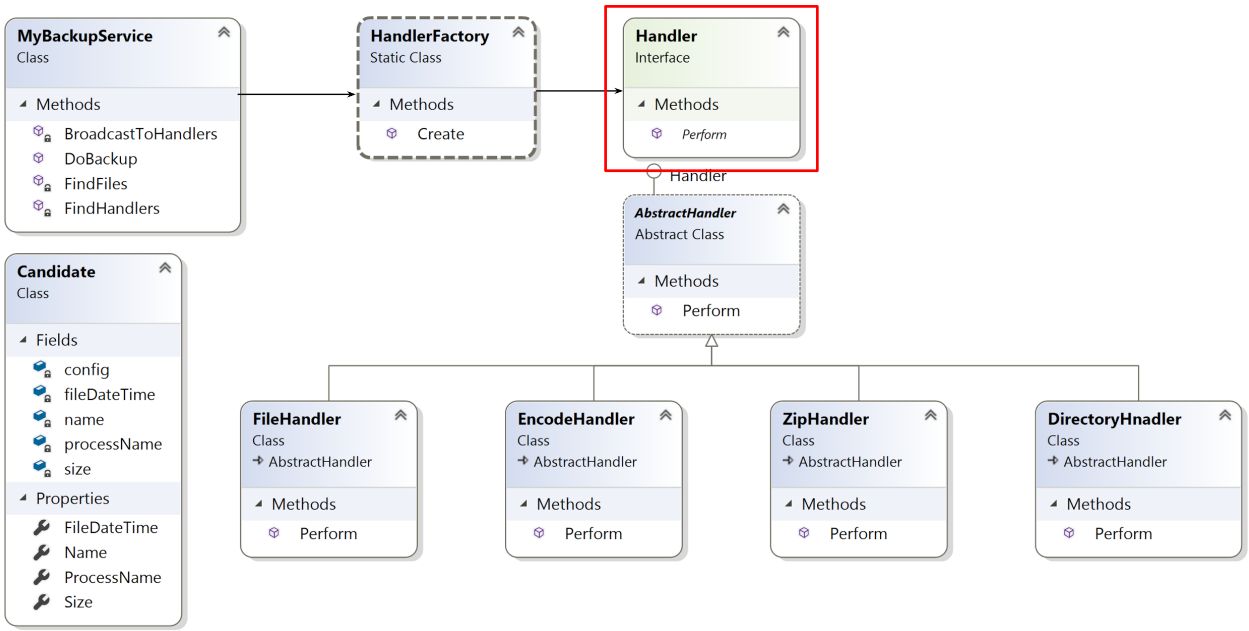


```
public class HandlerFactory
{
    private static Dictionary<string, string> handlerDictionary;

    static HandlerFactory()
    {
        string jsonString = File.ReadAllText("handler_mapping.json");
        HandlerFactory.handlerDictionary = JsonConvert.DeserializeObject<Dictionary<string, string>>(jsonString);
    }

    public static Handler Create(string key)
    {
        return (Handler)Activator.CreateInstance("MyBackupService", HandlerFactory.handlerDictionary[key]).Unwrap();
    }
}
```

> 負責 new handler

- 使用 reflection 方式建立 object，使 factory 完全開放封閉

Handler.cs

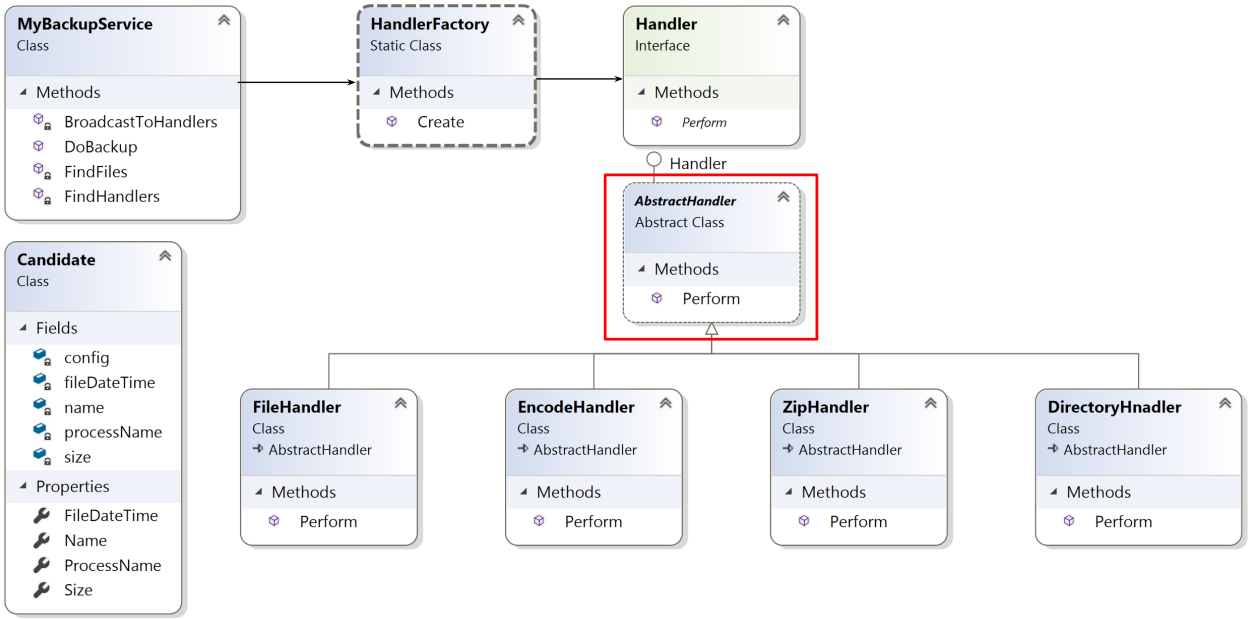



```
public interface Handler
{

  byte[] Perform(Candidate candidate, byte[] target);
}
```

> 所有 handler 的抽象化 interface，使用端只依賴此 interface，而不會依賴實際 handler

`Handler` interface 只有 `Perform()` 1 個 method，其中 `target` 型別為 `byte[]`，為前一個 handler 的 `Perform()` 回傳值，回傳值亦為 `byte[]`，將傳入下一個 handler 的 `Perform()`。

AbstractHandler.cs

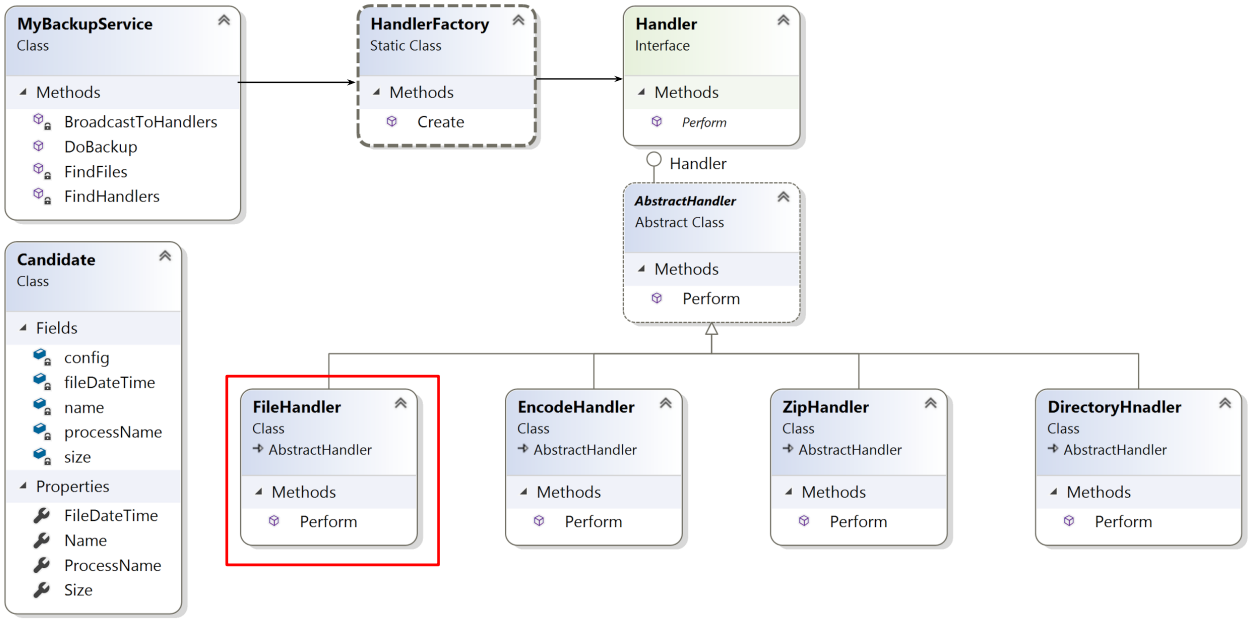



```cs
public abstract class AbstractHandler : Handler
{
    public virtual void Perform()
    {
        ...
    }
}
```

> 所有 handler 共用程式碼處

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

FileHandler.cs


MyBackupService
Class
Methods
BroadcastToHandlers
DoBackup
FindFiles
FindHandlers
HandlerFactory
Static Class
Methods
Create
Handler
Interface
Methods
Perform
Handler
AbstractHandler
Abstract Class
Methods
Perform
Candidate
Class
Fields
config
fileDateTime
name
processName
size
Properties
FileDateTime
Name
ProcessName
Size
FileHandler
Class
AbstractHandler
Methods
Perform
EncodeHandler
Class
AbstractHandler
Methods
Perform
ZipHandler
Class
AbstractHandler
Methods
Perform
DirectoryHnadler
Class
AbstractHandler
Methods
Perform

```
public class FileHandler : AbstractHandler
{
  public override byte[] Perform(Candidate cadidate, byte[]
target)
  {
     byte[] result = target;

     base.Perform(candidate, target);

     if (target == null) {
         result = this.ConvertFileToByteArray(candiate);
     }
     else {
       this.ConvertByteArrayToFile(candidate, target);
     }

     return result;
  }

  private byte[] ConvertFileToByteArray(Candidate candidate)
  {
      ...
  }

  private void ConvertByteArrayToFile(Candidate candidate,
byte[] target)
  {
      ...
  }
}
```

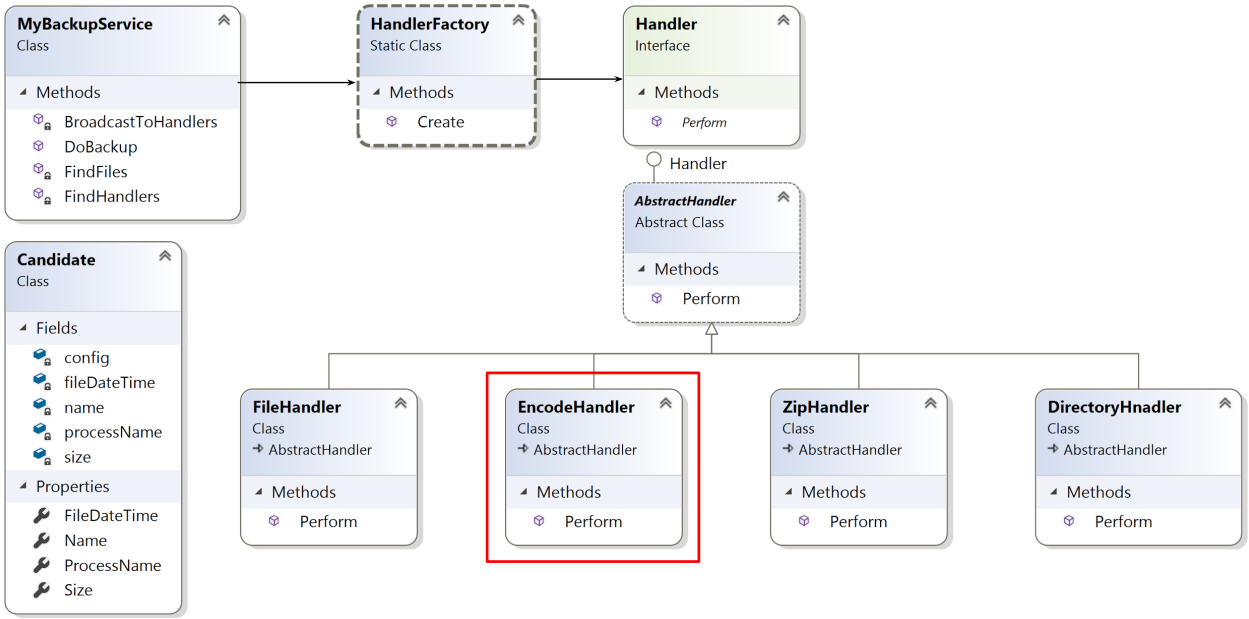
> 將檔案轉成 byte[]，或將 byte[] 轉成檔案

當第二個參數為 null 時，會將檔案轉成 byte[]；當傳入 byte[] 時，會將 byte[] 轉成檔案。

- `ConvertFileToByteArray()` : 將檔案轉成 byte[]
- `ConvertByteArrayToFile()` : 將 byte[] 轉成 file

> 此自行實做 `ConvertFileToByteArray()` 與 `ConvertByteArrayToFile()`

EncodeHandler.cs



```
public class EncodeHandler : AbstractHandler
{
 public override byte[] Perform(Candidate cadidate, byte[] target)
 {
    byte[] result = target;

    base.Perform(candidate, target);

    if (target != null) {
        result = this.EncodeData(candidate, target);
    }

    return result;
 }

 private byte[] EncodeData(Candidate candidate, byte[] target)
 {
     ...
 }
}
```

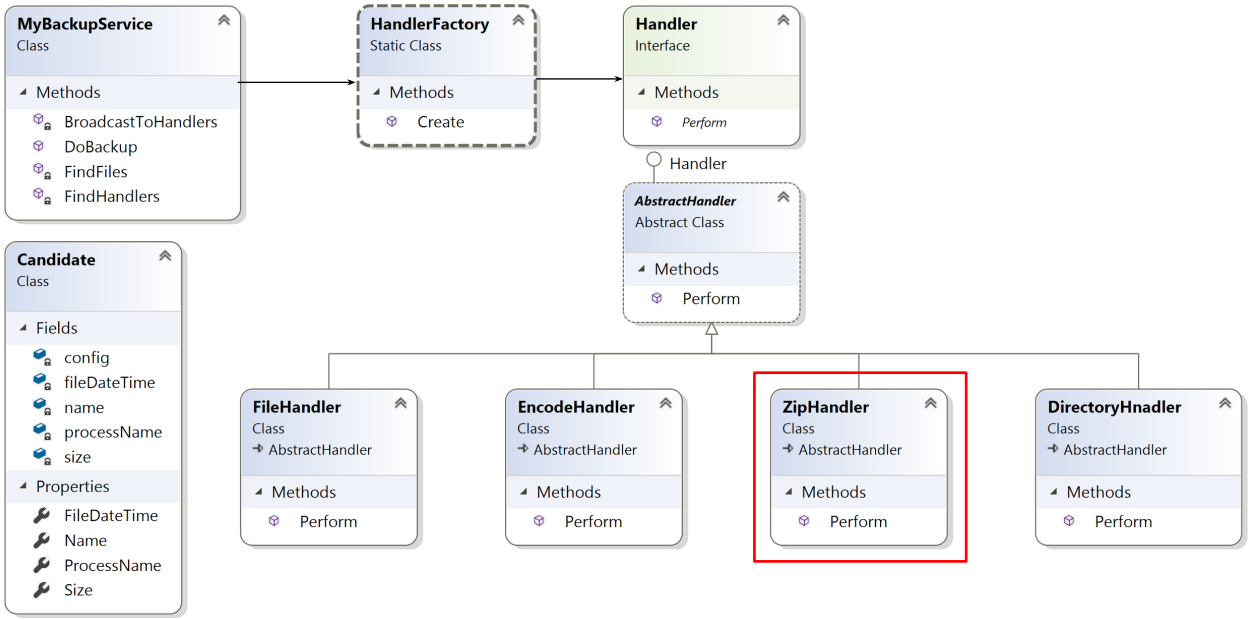
> 將 byte[] 加以編碼，並傳回 byte[]

當第二個參數為 null 時，不做任何編碼；當傳入 byte[] 時，編碼後回傳 byte[]。

- `EncodeData()`：將 byte[] 編碼後回傳 byte[]

> 請自行實做 `EncodeData()`

ZipHandler.cs




```
public class ZipHandler : AbstractHandler
{

  public override byte[] Perform(Candidate cadidate, byte[]
target)
  {
     byte[] result = target;

     base.Perform(candidate, target);

     if (target != null) {
         result = this.ZipData(candidate, target);
     }

     return result;
  }

  private byte[] ZipData(Candidate candidate, byte[] target)
  {
      ...
  }

}
```

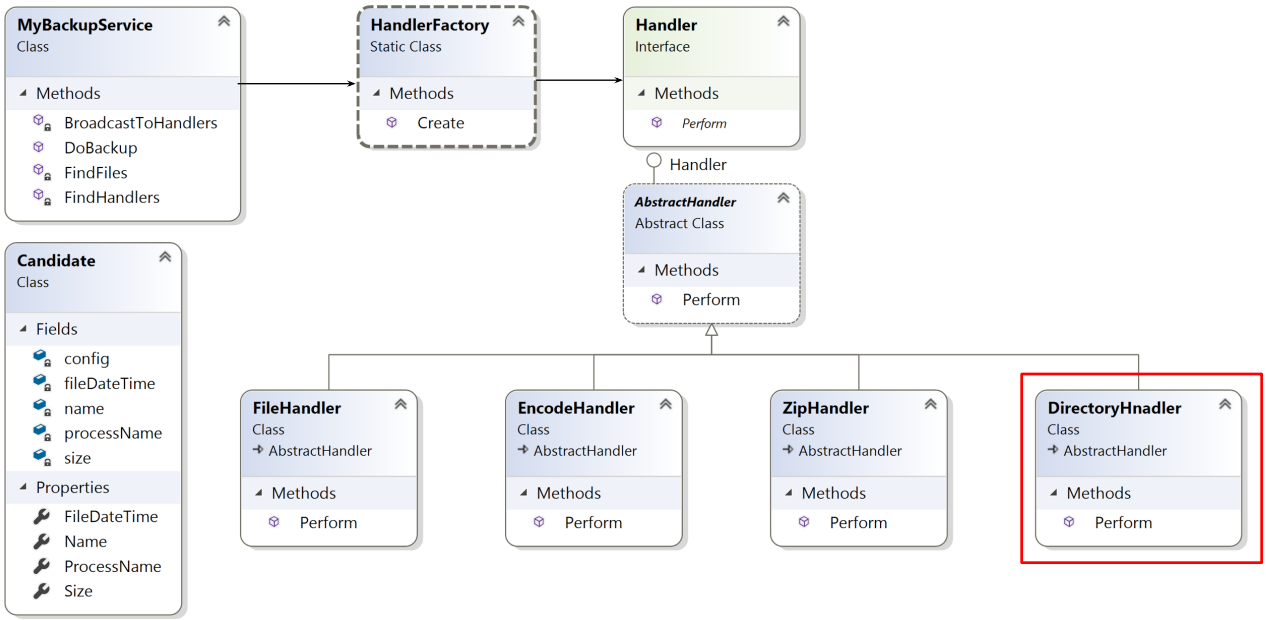
> 將 byte[] 加以壓縮，並傳回 byte[]

當第二個參數為 null 時，不做任何壓縮；當傳入 byte[] 時，壓縮後回傳 byte[]。

- **`ZipData()`：將 byte[] 壓縮後回傳 byte[]**

> 請自行實做 `ZipData()`

DirectoryHandler.cs



```
public class DirectoryHandler : AbstractHandler
{
 public override byte[] Perform(Candidate cadidate, byte[]
target)
 {
    byte[] result = target;

    base.Perform(candidate, target);

    if (target != null) {
        result = this.CopyToDirectory(candidate, target);
    }

    return result;
 }

 private byte[] CopyToDirectory(Candidate candidate, byte[]
target)
 {
     ...
 }
}
```

> 將 byte[] 還原成檔案，並複製到其他目錄

當第二個參數為 null 時，不做任何複製；當傳入 byte[] 時，複製到其他目錄。

- `CopyToDirectory()` : 將 byte[] 還原成檔案，並複製到其他目錄

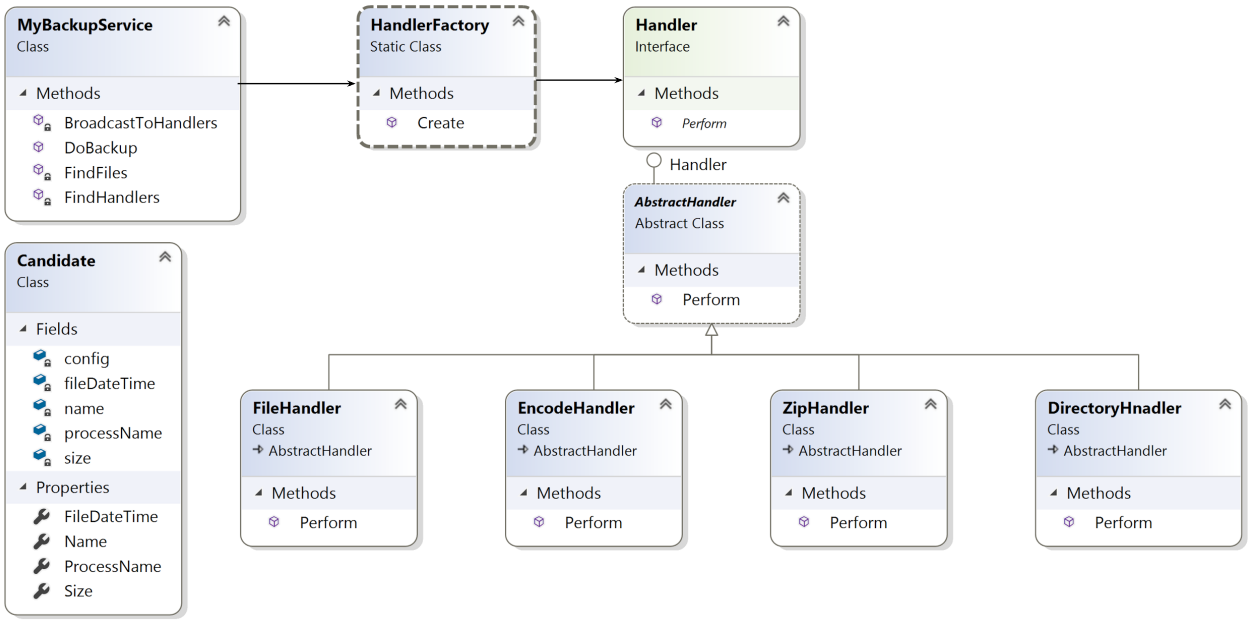
> 請自行實做 `CopyToDirectory()`

Summary

- `Interface` : 以 interface 取代 `繼承` ，彈性更高

- `Simple Factory 模式`：使用 `Factory` 取代 `new`，將建立物件的工作封裝在 `Factory`
- `Reflection`：讓 `Simple Factory 模式` 也能 `開放封閉`
- `界面隔離原則`：Class 間的依賴應該建立在最小 interface 上

Conclusion



- 程式語言不限，請依照 class diagram
  - `config.json` 的 `hadler` 由 `字串` 改成 `陣列`，表示支援同時支援多個 handler
  - `MyBackupService` 新增 `DoBackup()`、`BroadcastToHandlers()`、`FindHandlers()` method
  - 新增 `Candidate`、`HandleFactory`、`Handler`、`AbstractHandler`、`FileHandler`、`EncodeHandler`、`ZipHandler` 與 `DirectoryHandler` class