OOP #4

Sam Xiao, Nov.3, 2017

Overview

在 homework 3 的 `MyBackupService` 還有 `FindFiles()` 尚未實做，本次 homework 將實做這個部份。

Outline

Conclusion

Recap

Homework 3 :

- `Interface` : 以 interface 取代 `繼承` ，彈性更高
- `Simple Factory Pattern` : 使用 `Factory` 取代 `new` ，將建立 object 的工作封裝在 `Factory`
- `Reflection` : 直接使用 string 建立 object，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

User Story

備份檔案來源可能增加

目前 `MyBackup` 雖然以處理本機的檔案為主，但未來不排除有其他備份檔案來源，如來自於 `FTP 的檔案` 、來自於 `AWS S3 的檔案` ... 等等，因此我們希望有足夠的彈性可以搜尋放在任何位置的檔案，而不只是本機上的檔案而已。

檔案數量眾多

在 homework 2，我們預期 `MyBackupService.DoBackup()` 如下所示：

MyBackupService.cs

```
public class MyBackupService
{
    public void DoBackup()
    {
        List<Candidate> candidates = this.FindFiles();

        foreach(Candidate candidate in candidates)
        {
            this.BoradcastToHandlers(candidate);
        }
    }
}
```

但實務上檔案數量可能會很龐大，若另外使用 `List` 儲存所有找到的 `檔案資訊`，將有以下缺點：

- 必須花很多時間建立 `List<;Candidate>;`
- 需要浪費大量記憶體儲存 `List<;Candidate>;`

Task

備份檔案來源可能增加

將 `搜尋檔案` 功能加以 `抽象化`，讓使用端以 `多型` 的方式 `搜尋檔案`，就算將來改變備份檔案來源，只要在 `Factory` 或 `設定檔` 加以修改即可，使用端不用做任何修改，將 `修改程度` 降到最低，符合 `開放封閉` 原則。

檔案數量眾多

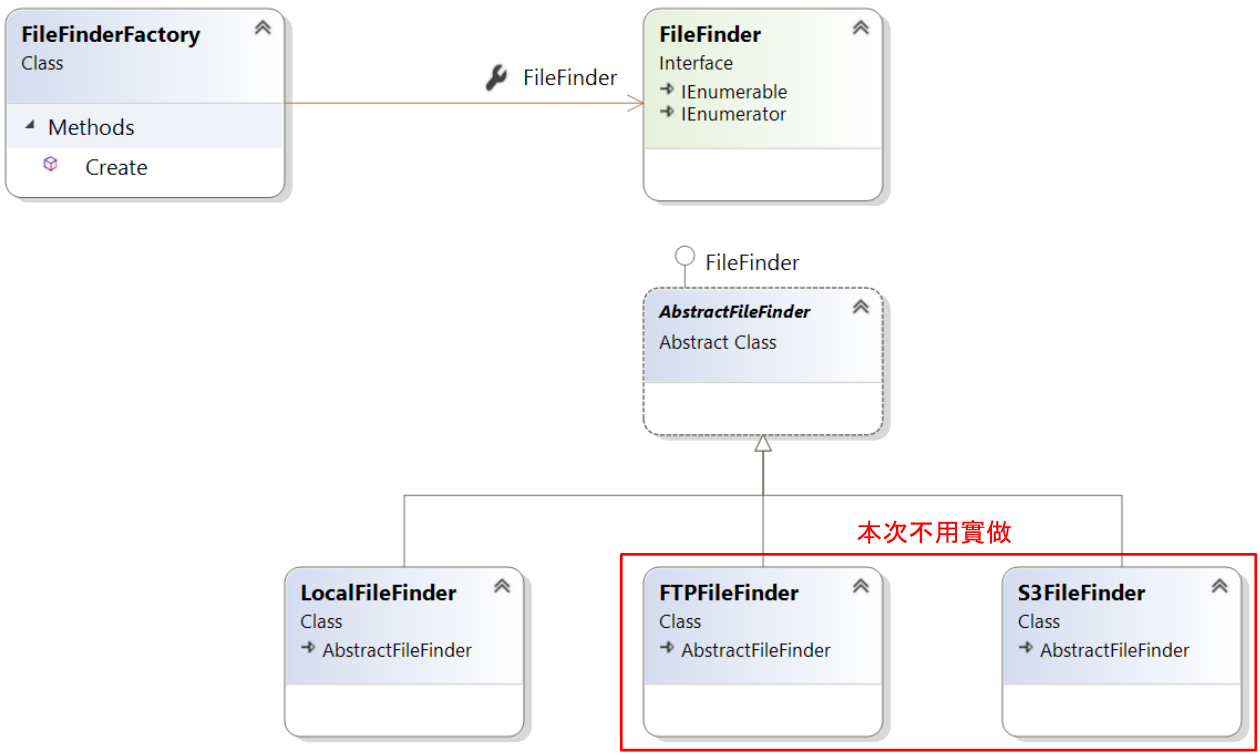
MyBackupService.cs

```
public class MyBackupService
{
     public void DoBackup()
     {
          FileFinder fileFinder =
FileFinderFactory.Create('file');

          foreach(Candidate candidate in fileFinder)
               this.BroadcastToHandlers(candidate);
     }
}
```

直接對 FileFinder 做 `foreach`，避免產生一個很大的 `List<;Candidate>;` 之後才做 `foreach`，也就是當每個檔案要對各 handler 做廣播時，才去產生 `檔案資訊` 的 `Candidate` 物件。

Architecture



備份檔案來源可能增加

由於將來不排除有其他備份檔案來源，也就是可預期的 `易於變動` 部份，因此與 homework 3 一樣先設計出 `FileFinder` interface 與 `AbstractFileFinder` class 將 `變動` 加以 `抽象化`，而 `LocalFileFinder` 為目前對本機的搜尋，未來可能有 `FTPFileFinder`、`S3FileFinder` … 等，最後由 `FileFinderFactory` 負責建立適合的 `FileFinder`。

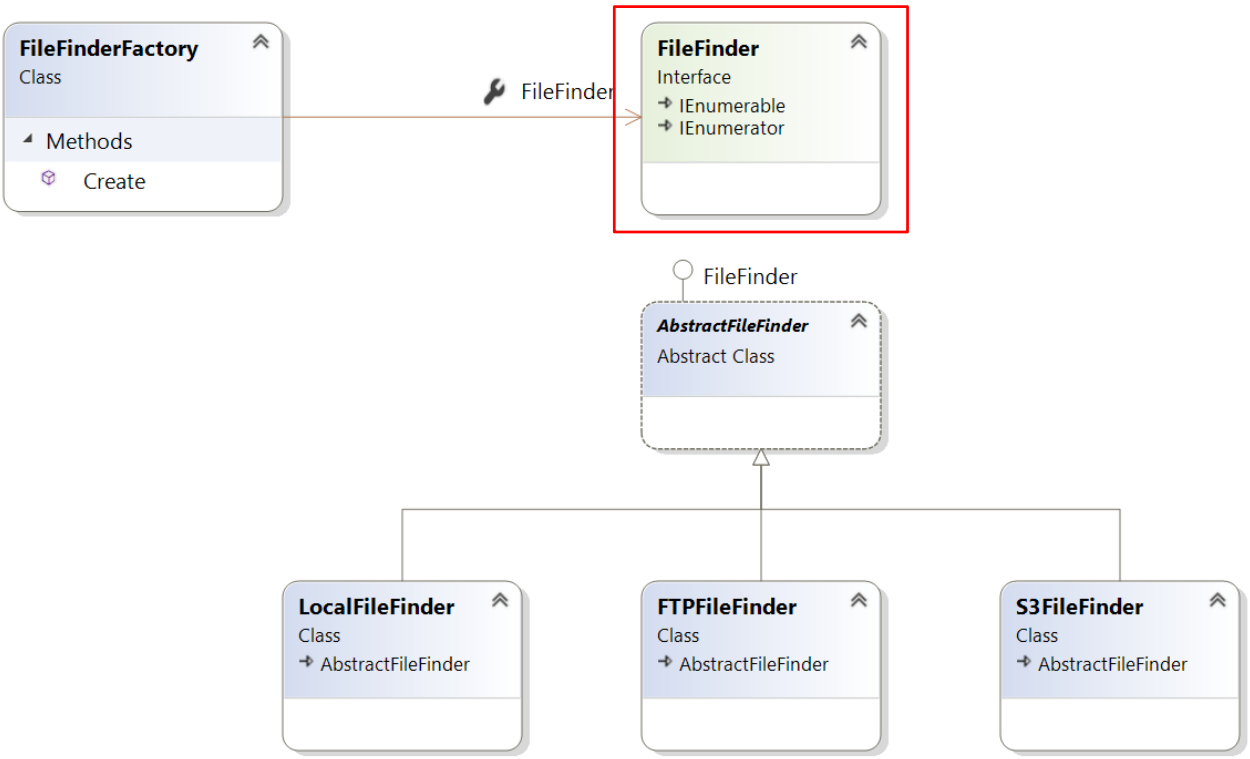
檔案數量眾多

將使 `FileFinder` 直接 implement `IEnumerable` 與 `IEnumerator` interface，讓使用端可以直接對 `FileFinder` 做 `foreach()`，節省另外建立 `List<;Candidate>;` 時間。

Implementation

FileFinder Interface

FileFinder.cs




```
1 public interface FileFinder: IEnumerable, IEnumerator
2 {
3 }
```

> 將所有 finder 抽象化成 `FileFinder` interface，使用端只依賴 (認識) 此 interface，而不會依賴實際 finder

`FileFinder` interface 繼承 .NET Framework 的 `IEnumerable` 與 `IEnumerator` interface。

> 因為將來要直接對 `FileFinder` 做 `foreach()`，所以必須 implement `IEnumerable` 與 `IEnumerator` interface。

當我們實務上需要多個 interface 的 method 時，可以使用 一個 interface 繼承多個 interface，雖然 class 不能繼承多個 class，但 interface 卻可繼承多個 interface。

IEnumerable interface

```
public interface IEnumerable
{
    public IEnumerator GetEnumerator();
}
```

- `GetEnumerator()`：將自己以 `IEnumerator` 型別回傳

IEnumerator interface

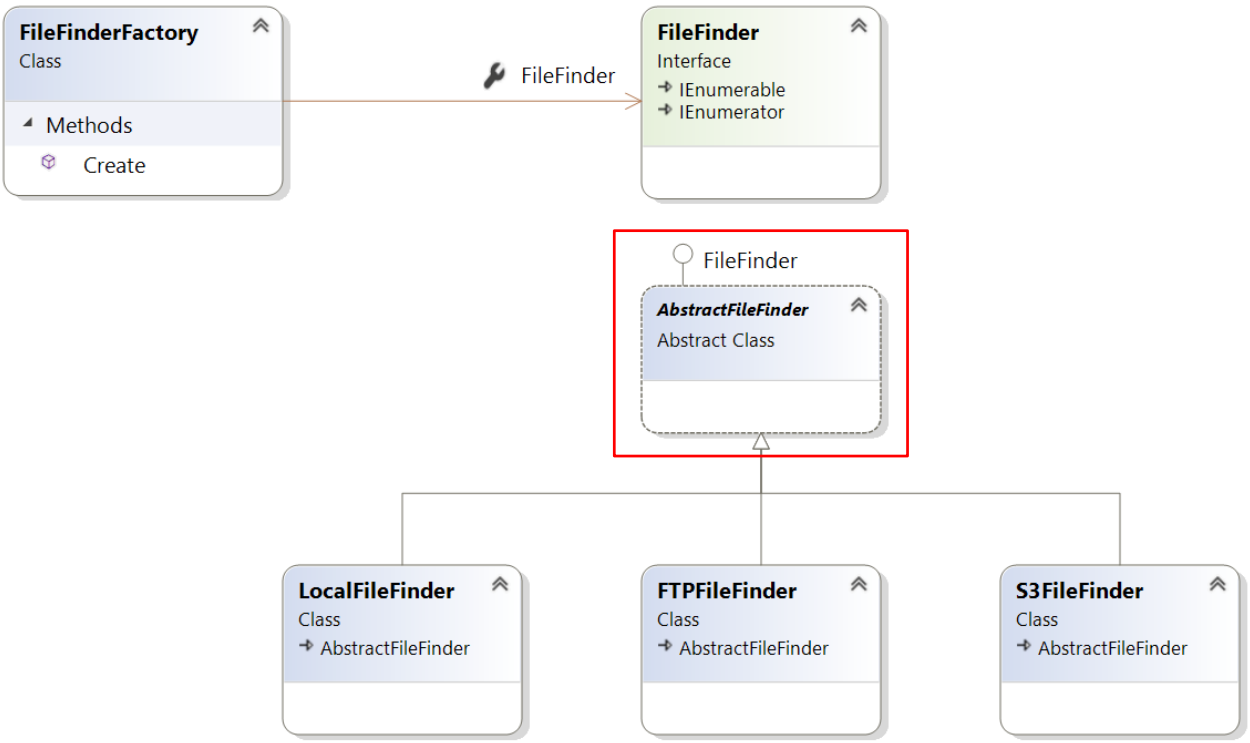
```
public interface IEnumerator
{
    public object Current { get; }
    public bool MoveNext();
    public void Reset();
}
```

- `Current`：取得目前 index 的資料
- `MoveNext()`：將 index 向前推進，若還沒超出範圍則回傳 `true`，否則傳回 `false`
- `Reset()`：設定初始 index

> OOP 心法
>
> 根據 `界面隔離原則`，一個 interface 通常都只有 1 到 3 個 method 而已，然後再由 class 去 implement 多個 interface，而不是宣告一個很多 method 的 interface 由 class 去 implement，如此使用端才可以根據自己的需求使用 `剛好` 與 `夠小` 的 interface，將 class 與 class 之間的耦合降到最低。

AbstractFileFinder Class

AbstractFinder.cs


FileFinderFactory
Class
Methods
Create
FileFinder
FileFinder
Interface
IEnumerable
IEnumerator
FileFinder
AbstractFileFinder
Abstract Class
LocalFileFinder
Class
AbstractFileFinder
FTPFileFinder
Class
AbstractFileFinder
S3FileFinder
Class
AbstractFileFinder

```
public abstract class AbstractFileFinder : FileFinder
{
    protected Config config;
    protected string[] files;
    protected int index = -1;

    protected AbstractFileFinder()
    {

    }

    protected AbstractFileFinder(Config config)
    {
        if (config != null)
            this.config = config;
    }

    // IEnumerator
    public object Current =>
this.CreateCandidate(this.files[this.index]);

    // IEnumerator
    public bool MoveNext()
    {
        this.index++;
        return (this.index < this.files.count);
    }

    // IEnumerable
    public IEnumerator GetEnumerator()
    {
        return (IEnumerator) this;
    }

```

```
34      // IEnumerator
35      public void Reset()
36      {
37          this.index = -1;
38      }
39
40      protected abstract Candidate CreateCandidate(string
    fileName);
41  }
```

> 所有 finder 共用程式碼處

如 `IEnumerable` 與 `IEnumerator` 要 implement 的 method，在各 FileFinder 都適用，可以統一實做在 `AbstractFileFinder`。

第 4 行

```
1  protected string[] files;
```

儲存所找到的檔案名稱，因為 .NET 的 `Directory.GetFiles()` 回傳的就是 `string[]`，因此沒有特別在使用 `List<;string>;`。

第 5

```
1  protected int index = -1;
```

因為 `MoveNext()` 一開始就 `index++`，所 `index` 起始值要從 `-1` 開始。

12 行

```
1  protected AbstractFinder(Config config)
2  {
3      if (config != null)
4          this.config = config;

5  }
```

由 constructor 傳入 `Config` 物件。

18 行

```
1 // IEnumerator
2 public object Current =>
  this.CreateCandidate(this.files[this.index]);
```

為 property，取得目前 index 的資料。

原本應該輕鬆傳回 `this.files[this.index]`，不過 `files[]` 放的只是檔案名稱 string，而非 `Candidate`，故須呼叫 `this.CreateCandidate()` 回傳 `Candidate` 物件。

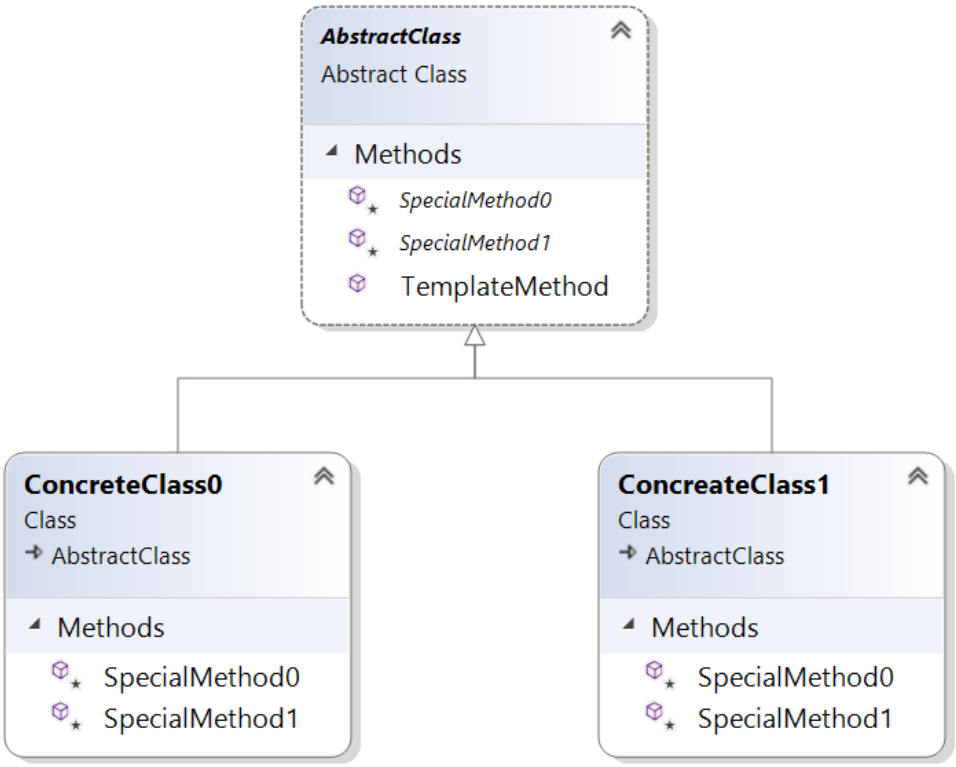
這是關鍵，每次 `foreach()` 才去 new `Candidate`，避免回傳一個龐大的 `List<;Candidate>;`。

40 行

```
1 protected abstract Candidate CreateCandidate(string
  fileName);
```

`CreateCandidate()` 為 `abstract` method，因為子類別建立`Candidate` 物件的方式各有不同，目的是要讓子類別自行實做自己建立 `Candidate` 的方法，因為 `LocalFileFinder`、`FTPFileFinder` 、`S3FileFinder` 都會有不同的建立 `Candidate` 方式。

> Template Method Pattern
>
> Public method 放在父類別，與父類別不同的實做宣告成 protected abstract method，由子類別自行實做



AbstractClass.cs

```
public AbstractClass
{
    public void TemplateMethod()
    {
        ...                    // (相同程式碼)
        this.SpecialMethod0(); // (不同程式碼)
        ...                    // (相同程式碼)
        this.SpecialMethod1(); // (不同程式碼)
        ...                    // (相同程式碼)
    }

    protected abstract void SpecialMethod0();
    protected abstract void SpecialMethod1();
}
```

使用端看到的是 `AbstractClass` 與 `TemplateMethod()`。

ConcreteClass0.cs

```
public class ConcreteClass0 : AbstractClass
{
    protected override void SpecialMethod0()
    {
        ...
    }

    protected override void SpecialMethod1()
    {
        ...
    }
}
```

將各 class 的差異躲在子類別的 `SpecialMethod0()` 與 `SpecialMethod1()`。

> 預先設計
>
> 由父類別定義 method 的架構，由子類別去實踐差異部份

傳統 `繼承`，我們都是將 `相同` 的 method `往上抽`，但是 Template Method Pattern 卻是反過來，將 `不同` 的 method `往下抽`。

> 重構
>
> 將相同的程式碼放在父類別，將不同的程式碼放在子類別

重構前

Class0.cs

```
public Class0
{
   public void Func()
   {
       xxxx;
       oooo;
       pppp;
   }
}
```

Class1.cs

```
public Class1
{
    public void Func()
    {
        xxxx;
        qqqq;
        pppp;
    }
}
```

`Class0.Func0` 與 `Class1.Func0`，其中 `xxxx` 與 `pppp` 程式碼相同，只有 `oooo` 與 `qqqq` 不同。

第一次重構

Class0.cs

```
public Class0
{
    public void Func()
    {
        this.xxxx();
        oooo;
        this.pppp();
    }

    private void xxxx()
    {
        xxxx;
    }

    private void pppp()
    {
        pppp;
    }
}
```

`Class0` 先抽出 `xxxx()` 與 `pppp()` 兩個 private method。

Class1.cs

```
public Class1
{
     public void Func()
     {
          this.xxxx();
          qqqq;
          this.pppp();
     }

     private void xxxx()
     {
          xxxx;
     }

     private void pppp()
     {
          pppp;
     }
}
```

`Class1` 也抽出 `xxxx()` 與 `pppp()` 兩個 private method。

我們發現 `Class0` 與 `Class1` 都有 `xxxx()` 與 `pppp()` method。

第二次重構

AbstractClass.cs

```
public class AbstractClass
{
    protected void xxxx()
    {
        xxxx;
    }

    protected void pppp()
    {
        pppp;
    }
}
```

Class0.cs

```
public class Class0 : AbstractClass
{
    public void Func()
    {
        this.xxxx();
        oooo;
        this.pppp();
    }
}
```

Class1.cs

```
public class Class1 : AbstractClass
{
    public void Func()
    {
        this.xxxx();
        qqqq;
        this.pppp();
    }
}
```

將 `xxxx()` 與 `pppp()` 抽到 `AbstractClass` 後，最少 `xxxx()` 與 `pppp()` 沒有重複，符合 `DRY` 原則。

但 `Class0.Func()` 與 `Class1.Func()` 的架構非常類似。

第三次重構

AbstractClass.cs

```
public class AbstractClass
{
    public void Func()
    {
        xxxx;
        this.oqoq();
        pppp;
    }

    protected abstract void oqoq();
}
```

將 `Func()` 全部搬到 `AbstractClass`，將不同的部份向下抽成 abstract `oqoq()`。

Class0.cs

```
public class Class0
{
    protected override void oqoq()
    {
        oooo;
    }
}
```

Class1.cs

```
public class Class1
{
    protected override void oqoq()
    {
        qqqq;
    }
}
```

如此 `Class0.Func()` 與 `Class1.Func()` 相同的架構被抽到 父類別，`Class0` 與 `Class1` 只實做不同的 `oooo` 與 `qqqq` 即可。

> Template Method Pattern 是比較好的 繼承 使用方式，傳統繼承是將相同 method 放到 父類別，而 template 是將不同的 method 放在 子類別，寫法會更精簡。
>
> 而且子類別不會有 `public`，不會破壞原本的抽象。

21 行

```
// IEnumerator
public bool MoveNext()
{
     this.index++;
     return (this.index < this.files.count);
}
```

將 index 向前推進，若還沒超出範圍則回傳 `true`，否則傳回 `false`。

28 行

```
// IEnumerable
public IEnumerator GetEnumerator()
{
     return (IEnumerator) this;
}
```

將自己以 `IEnumerator` 型別回傳。

34 行

```
// IEnumerator
public void Reset()
{
     this.index = -1;
}
```

設定初始 index。

因為 `MoveNext()` 一開始就 `index++`，故將 `index` 起始狀態設定為 `-1`。

> Q : 為什麼要使用 `foreach()` 就要實做 `IEnumerable` 與 `IEnumerator` interface ?

```
int[] data = {1, 3, 5, 7, 9};
foreach(int value in data)
{
    Console.WriteLine(value)
}
```

經過編譯後會變成

```
int[] data = {1, 3, 5, 7, 9};
IEnumerator iterator = data.GetEnumerator();

while(iterator.MoveNext())
{
    Console.WriteLine(iterator.Current);
}
```

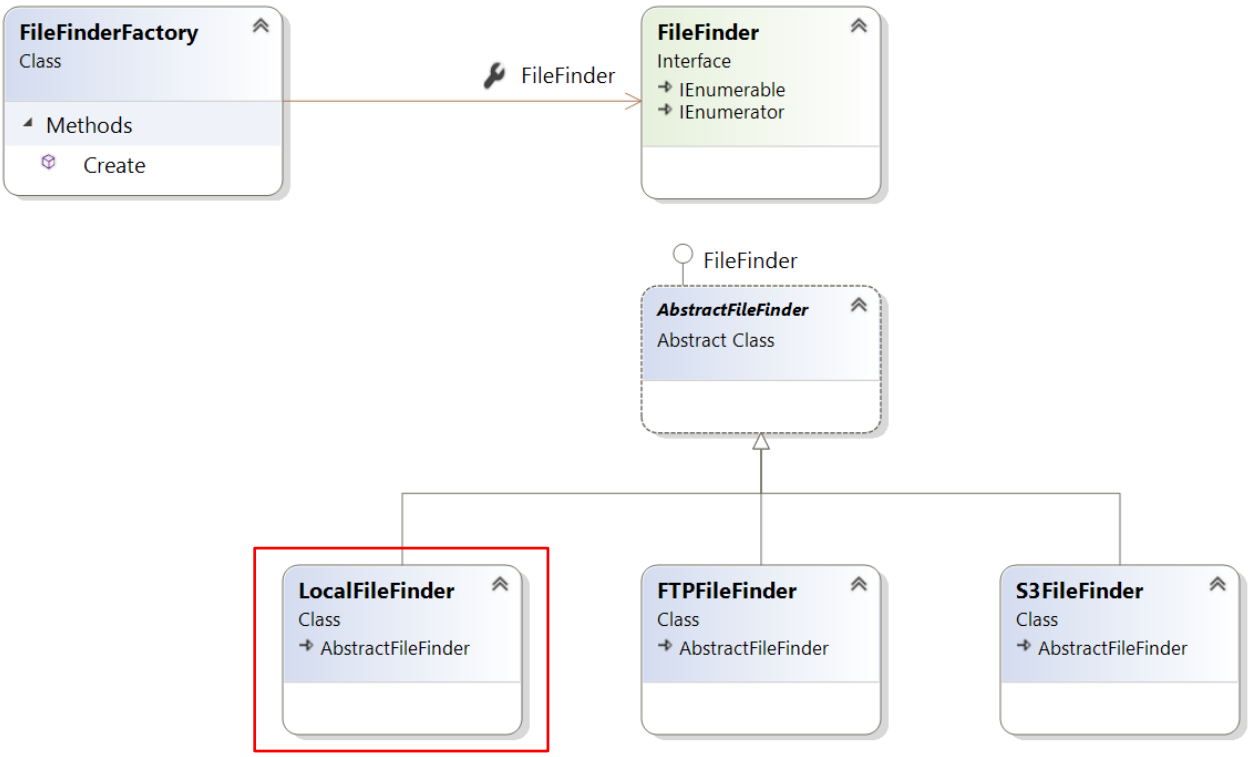
這也是為什麼我們必須實做 `GetEnumerator()`、`MoveNext()` 與 `Current` ... 等 method 與 property。

> Iterator Pattern
>
> 讓用戶端可以使用 `foreach` 操作物件內部資料

由於 .NET 已經定義了 `IEnumerable` 與 `IEnumerator` interface，直接配合即可。

LocalFileFinder Class

LocalFileFinder.cs


FileFinderFactory
Class
Methods
Create
FileFinder
FileFinder
Interface
IEnumerable
IEnumerator
FileFinder
AbstractFileFinder
Abstract Class
LocalFileFinder
Class
AbstractFileFinder
FTPFileFinder
Class
AbstractFileFinder
S3FileFinder
Class
AbstractFileFinder

```
1  public class LocalFileFinder: AbstractFileFinder
2  {
3      public LocalFileFinder()
4      {
5
6      }
7
8      public LocalFileFinder(Config config): base (config)
9      {
10         if (config.SubDirectory)
11             this.files = this.GetSubDirectoryFiles(config);
12         else
13             this.files =
   Directory.GetFiles(config.Location, "*." + config.Ext);
14     }
15
16     private string[] GetSubDirectoryFiles(Config config)
17     {
18         ...
19     }
20
21     protected override Candidate CreateCandidate(string
   fileName)
22     {
23         ...
24     }
25 }
```

第 8 行

```
1  public LocalFileFinder(Config config): base(config)
2  {
3      if (config.SubDirectory)
4          this.files = this.GetSubDirectoryFiles(config);
5      else
6          this.files = Directory.GetFiles(config.Location,
   "*." + config.Ext);
7  }
```

將 `Config` 透過 `FileFinder` 的 constructor 傳入。

若 `SubDirectory` 為 `true`，則由 `GetSubDirectoryFiles()` 連同子目錄一起處理。

若 `SubDirectory` 為 `false`，則由 .NET Framework 的 `Directory.GetFiles()` 直接抓目前目錄下所有的 `.cs` 檔案。

32 行

```
1  private string[] GetSubDirectoryFiles(Config config)
2  {
3      ...
4  }
```

回傳目前目錄與子目錄下的所有檔案，請同學自行實做。

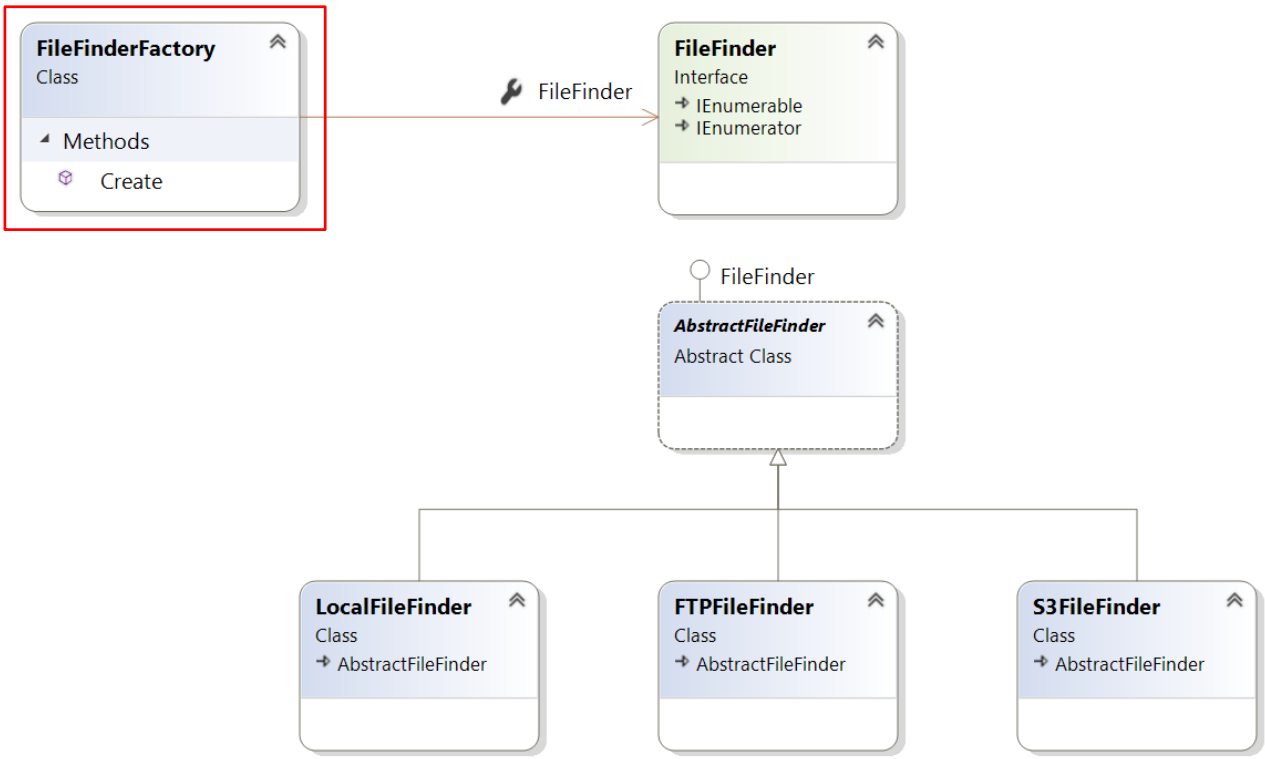
37 行

```
1  private Candidate CreateCandidate(string fileName)
2  {
3      ...
4  }
```

由檔案名稱建立 `Candidate` 物件，請同學自行實做。

FileFinderFactory Class

FileFinderFactory.cs




```
1  public class FileFinderFactory
2  {
3      public static FileFinder Create(string finder, Config
   config)
4      {
5          if (finder == "file")
6              return new LocalFileFinder(config);
7          else
8              return null;
9      }
10 }
```

> 負責 new finder

由於目前只有一個 `LocalFileFinder`，所以 `FileFinderFactory.Create()` 簡單的使用 `if else` 即可，最少將 `if else` 封裝在 `Factory` 內。

若將來有 `FTPFileFinder` 與 `S3FileFinder`，可考慮使用 homework 3 的 `reflection + 設定檔` 的方式，將 `FileFinderFactory` `開放封閉` 起來。

物件導向

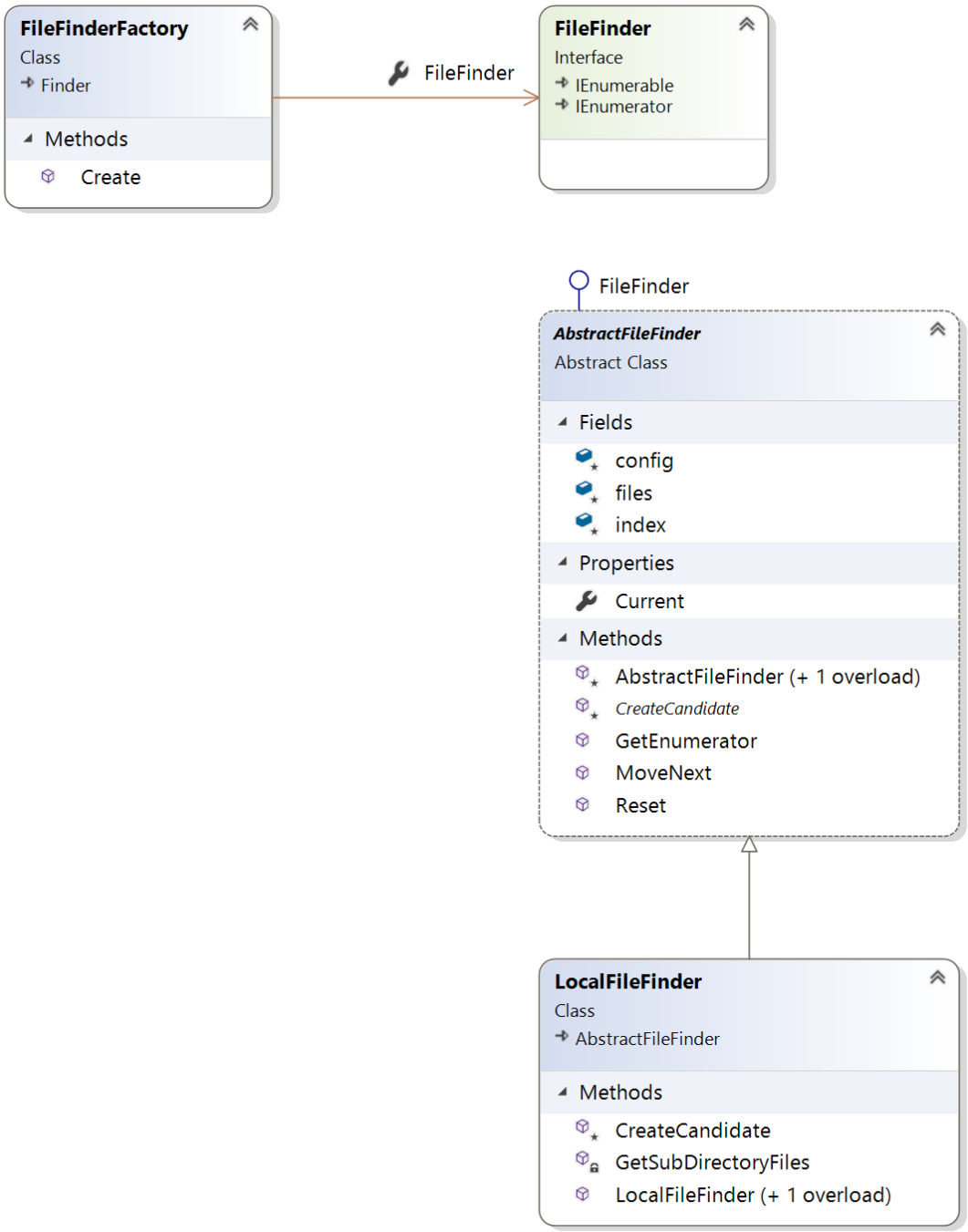
> 模擬世界，加以處理
>
> 依賴抽象，封裝變化
>
> 增加界面，幫助擴展

使用 `interface` 取代 `if else`：有新的需求，應該先定義出 `界面`，再根據 `界面` 去實做功能，而不是用 `if else` 判斷去執行新功能。

Summary

- Interface 可繼承多個 interface，相當於將多個 interface 統整成一個 interface
- `Template Method Pattern`：Public method 放在父類別，與父類別不同的實做宣告成 protected abstract method，由子類別自行實做
- `Iterator Pattern`：讓用戶端可以使用 `foreach` 操作物件內部資料，並藉由實做 `IEnumerable` 與 `IEnumerator` interface 實現 `Iterator Pattern`
- 物件導向第 3 定義：`增加界面，幫助擴展`

Conclusion



- 程式語言不限，請依照 class diagram
  - 新增 `FileFinder`、`AbstractFileFinder`、`LocalFileFinder` 與 `FinderFactory` 4 個 class
  - 實做 `LocalFileFinder` class 的 `GetSubDirectoryFiles()` 與 `CreateCandidate()`
  - 實做 `IEnumerable` 與 `IEnumerator` interface，使 `foreach` 能直接使用 `FileFinder` 獲得 `Candidate` 物件