OOP #5

Sam Xiao, Nov.9, 2017

Overview

在 homework 1 ~ homework 4 已經將 `MyBackup` 大部分的核心功能都做好了，逐漸到了收尾階段，本次 homework 要將所有的 class 整合在一起，並將 `排程備份` 功能完成。

Outline

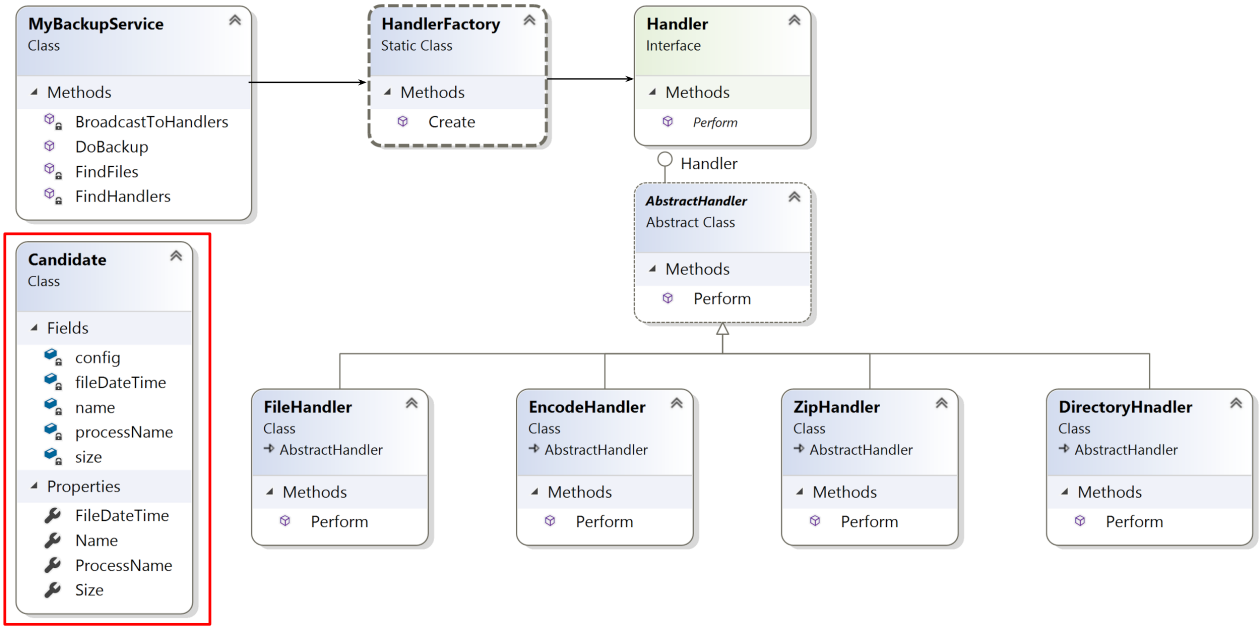
Recap

Homework 4 :

- 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`

User Story

Candidate



在 homework 3 雖然我們已經實做了 `Candidate` class，不過我們對此 class 著墨不多，只當成描述 `檔案資訊` 的 object，但也因為沒有特別處理，因此可能寫成這樣：

```
public class Candidate
{
    private Config config;
    private DateTime fileDateTime;
    private string name;
    private string processName;
    private int size;

    public DateTime FileDateTime
    {
        get => this.fileDataTime;
        set => this.fileDateTime = value;
    }

    public string Name
    {
        get => this.name;
        set => this.name = value;
    }

    public string ProcessName
    {
        get => this.processName;
        set => this.processName = value;
    }

    public int Size
    {
        get => this.size;
        set => this.size = value;
    }


    public Candidate()
```

```
    {

    }

    public Candidate(Config config, string name, DateTime fileDateTime, int size) {
        this.config = config;
        this.name = name;
        this.fileDateTime = fileDateTime;
        this.size = size;
    }
}
```

都為 `可讀可寫` 的 property。

LocalFileFinder.cs

```
public class LocalFinder
{
    protected Candidate CreateCandidate(string fileName)
    {
        FileInfo fileInfo;
        Candidate candidate;

        if (File.Exists(fileName))
        {
            fileInfo = new FileInfo(fileName);
            candidate = new Candidate(this.config, fileName, fileInfo.CreationTime, fileInfo.Length);
        }

        return candidate;
    }

}
```

但這樣寫可能有幾個問題：

- 由於 `Candidate` 只是普通的 `public class`，`LocalFinder` 可以 new `Candidate`，其他 class 也都可以 new `Candidate`
- 由於 property 是 `{ get; set; }` ，因此用戶端拿到 `Candidate` object 後，依然可以修改 `Candidate` object

理論上 `Candidate` object 代表的是 `檔案資訊`，不應該由任意 class 建立，也不應該能被用戶端修改，必須忠實的呈現 `檔案資訊` 。

因此需求端提出以下的需求：

- 提供一個一致的建立與初始化 `Candidate` 的方式
- 避免用戶端不正當的建立 `Candidate`

MyBackupService

```
MyBakcupService myBackupService = new MyBackupService();
myBackupService.ProcessJSONConfig();
myBackupService.DoBackup();
```

在 homework 2，我們已經提供 `MyBackupService` 單一入口，用戶端只要知道 `ProcessJSONConfig()` 與 `DoBackup()` 兩個 method 即可，但若進一步思考會發現：

- `ProcessJSONConfig()` 並不是 user 的需求，似乎不用提供 `ProcessJSONConfig()`
- `MyBackup` 原本的需求有兩種，一個是 `簡單備份`，令一個是 `排程備份`，目前的 `DoBackup()` 在功能上是 `簡單備份`，不過很難由 `DoBackup()` 的命名知道是 `立即備份` ，還是 `排程備份`

Task

Candidate

- 將 `Candidate` 內的資料改用 `readonly`，避免被任意修改，並僅提供 read only property
- `Candidate` 只能由 `CandidateFactory` 建立，改變 `Candidate` 與 `CandidateFactory` 的 namespace，並將 `Candidate` 由 `public class` 改成 `internal class`

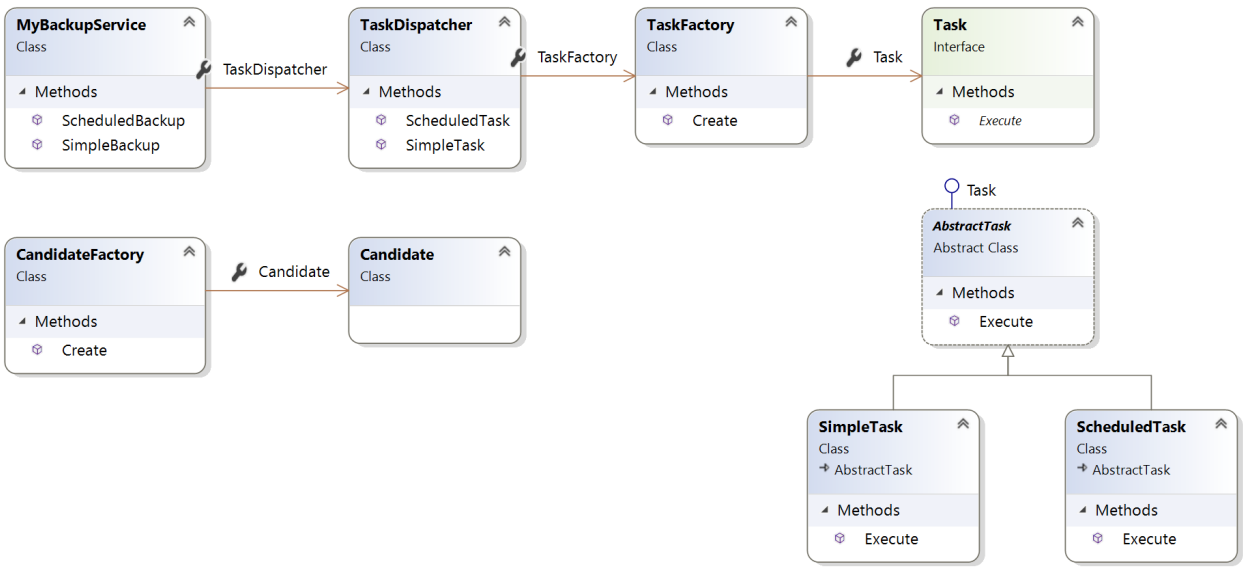
MyBackupService

```
MyBakcupService myBackupService = new MyBackupService();
myBackupService.SimpleBackup(); // 簡單備份
// myBackupService.ScheduledBackup(); // 排程備份
```

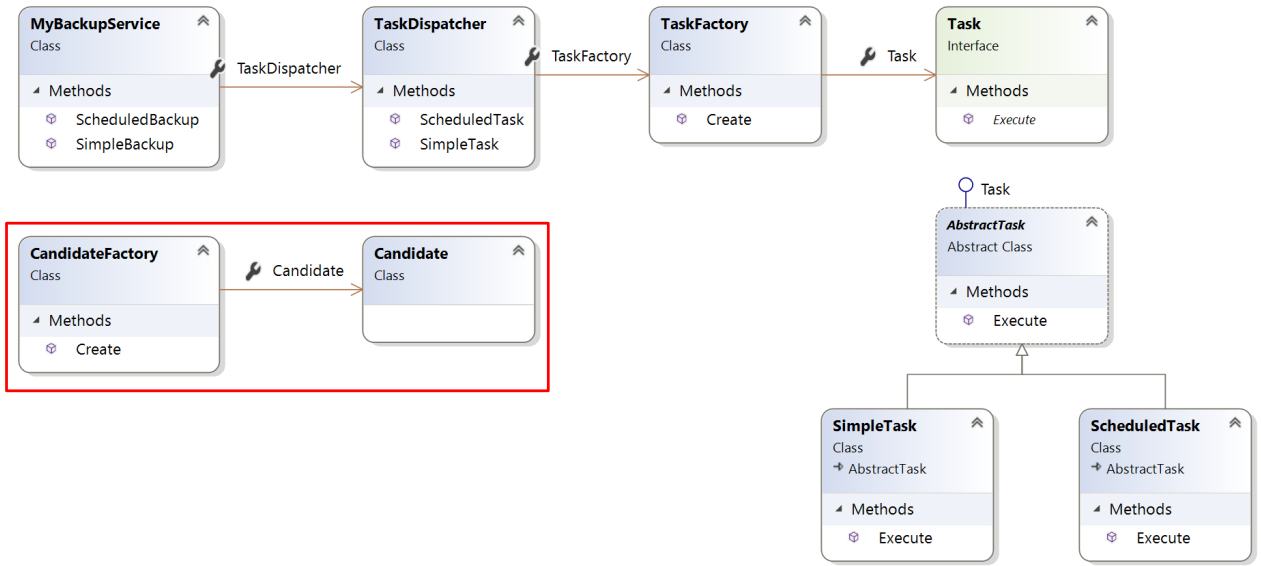
- 將 `ProcessJSONConfig()` 由 `public` 移除，畢竟這不是 user 所要的功能

- 根據 user 的原始需求：`簡單備份` 與 `排程備份`，只提供 `SimpleBackup()` 與 `ScheduledBackup()` 兩個 `public` method

Architecture

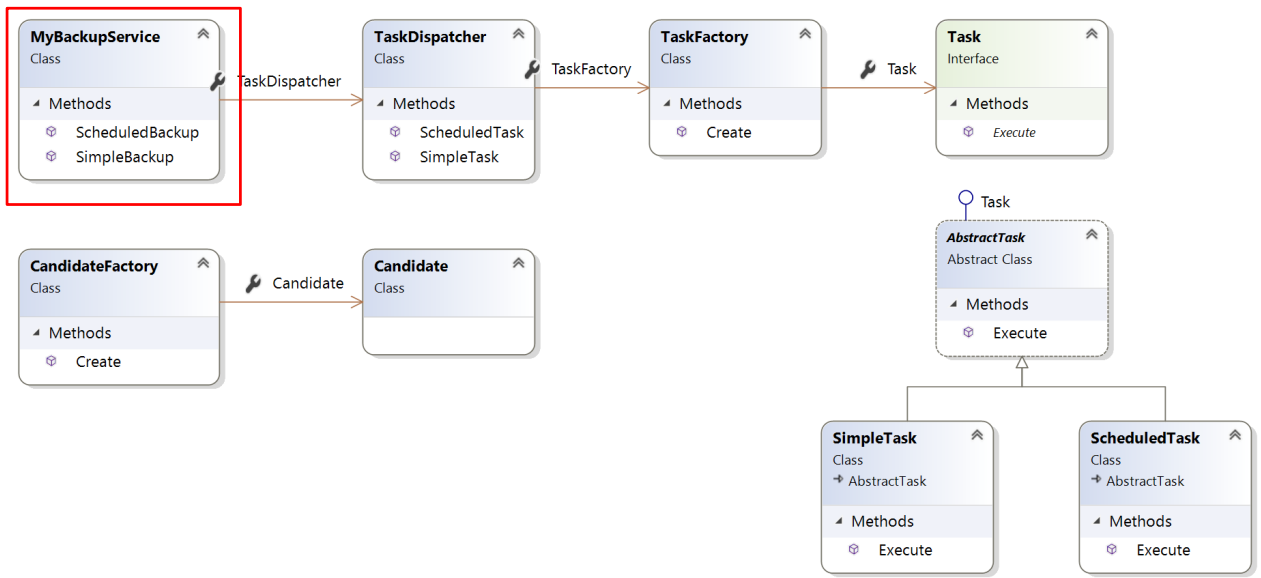


Candidate



`Candidate` 比須透過 `CandidteFactory` 建立。

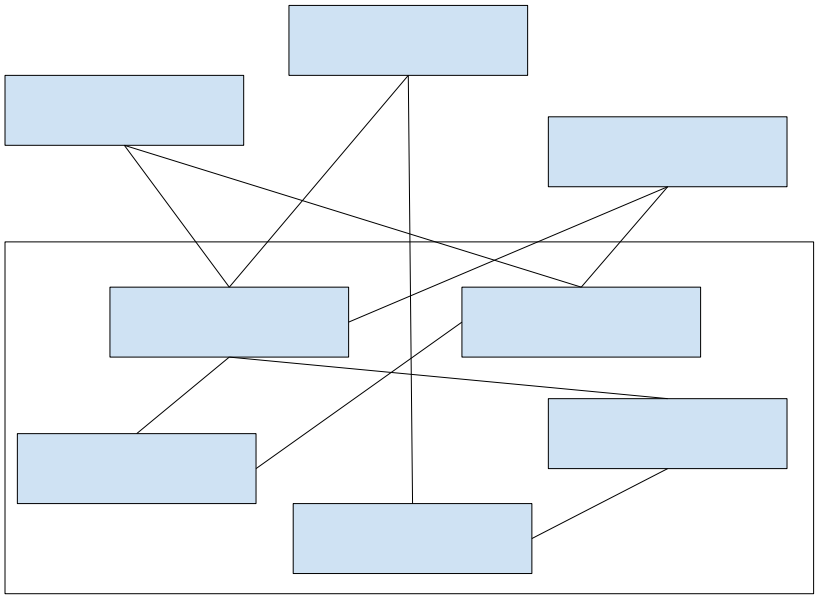
MyBackupService



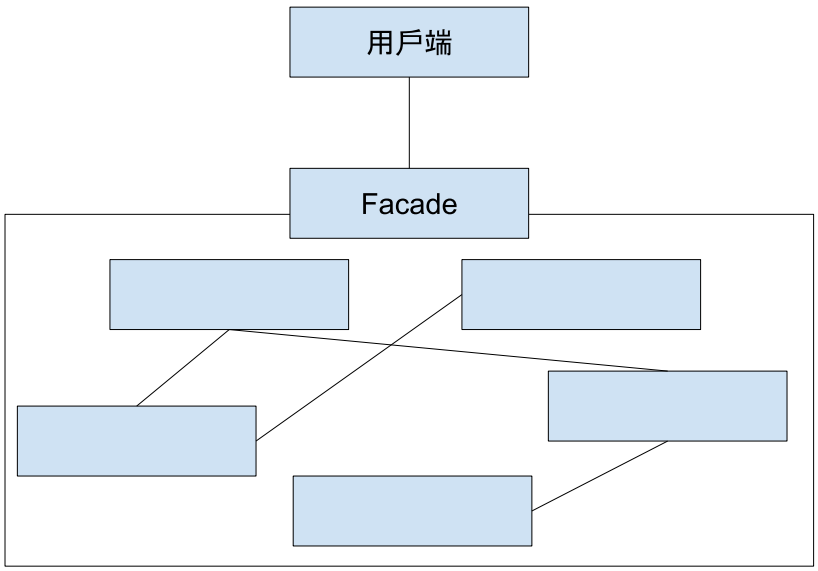
目前我們已經建立了很多 class，如 homework 2 的 `JsonManager` 家族、homework 3 的 `Handler` class 家族，homework 4 的 `FileFinder` class 家族，當然我們可以將這些 class 全丟給用戶端管理，但這會使的用戶端邏輯非常複雜且難以維護。

> 回想在我們的 MVC 架構中，若 controller 需要大量的 `new / 注入` 其他service，將導致大量 service 管理邏輯都寫在 controller 中，造成 controller 異常的肥大而難以維護。

使用 Facade 前



使用 Facade 後



> Facade Pattern

> 為一堆 class 定義統一的入口，讓用戶端更容易使用
>
> 降低用戶端對內部 class 的依賴，只依賴 facade 即可

事實上 `MyBackupService` 的設計就是 `Facade Pattern` 的實踐，讓用戶端 (controller) 不用面對大量的 `JsonManager`、`Handler` 與 `FileFinder` 等 class，只要簡單面對 `MyBackupService` 即可，用戶端邏輯會非常簡單且好維護。

> OOP 心法
>
> OOP 設計要以用戶端好用好維護為前提，而不是開發端好寫為前提

> Q : `MyBackupService` 該如何設計呢？

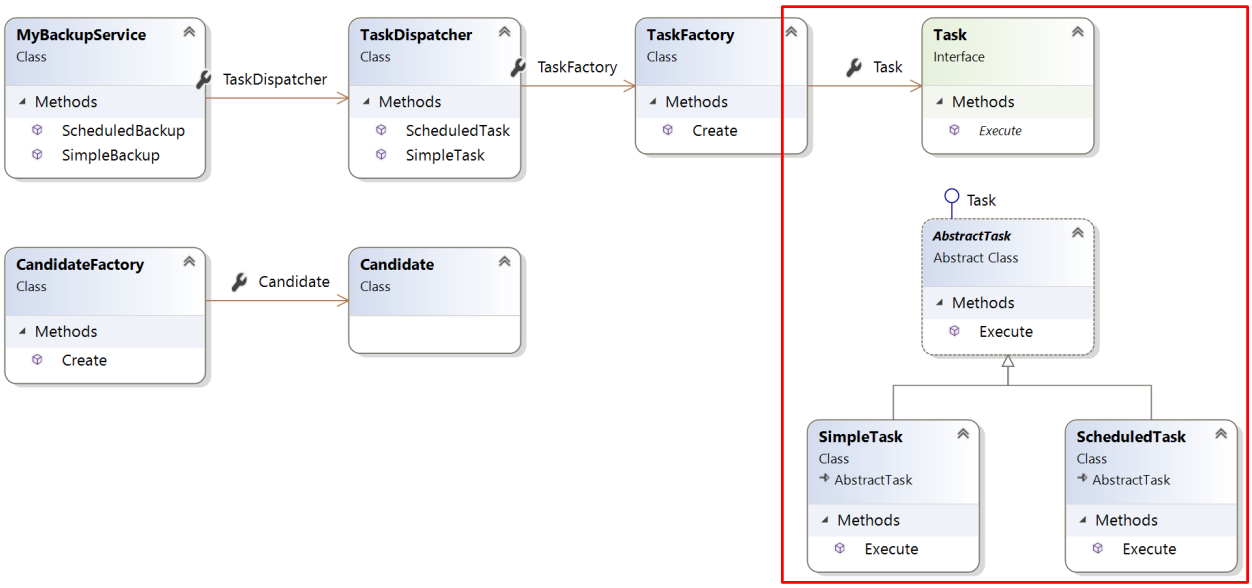
`MyBackupService` 作為一個 `Facade`，目的就是當一堆 class 的門面，讓用戶端可以用最簡單的方式完成工作，對用戶端來說，`MyBackup` 主要就兩件事情 :

- `簡單備份` : `SimpleBackup()`
- `排程備份` : `ScheduledBackup()`

因此只要提供兩個 method 即可，這對使用端是最方便的。

> Facade Pattern 就是 `最小知識原則` 的實踐

Task Family



若只是將原本使用端複雜的程式碼搬到 `Facade` 內，那問題依舊沒有解決，只是將原本難以維護的使用端，變成難以維護的 `MyBackupService` 而已。

> 單一職責原則 (SRP : Single Responsibililty Principle)
>
> 一個 class / method 應該只有一個 `職責`
>
> class / method 應該要專業分工，而非萬能 class / method 包含所有功能

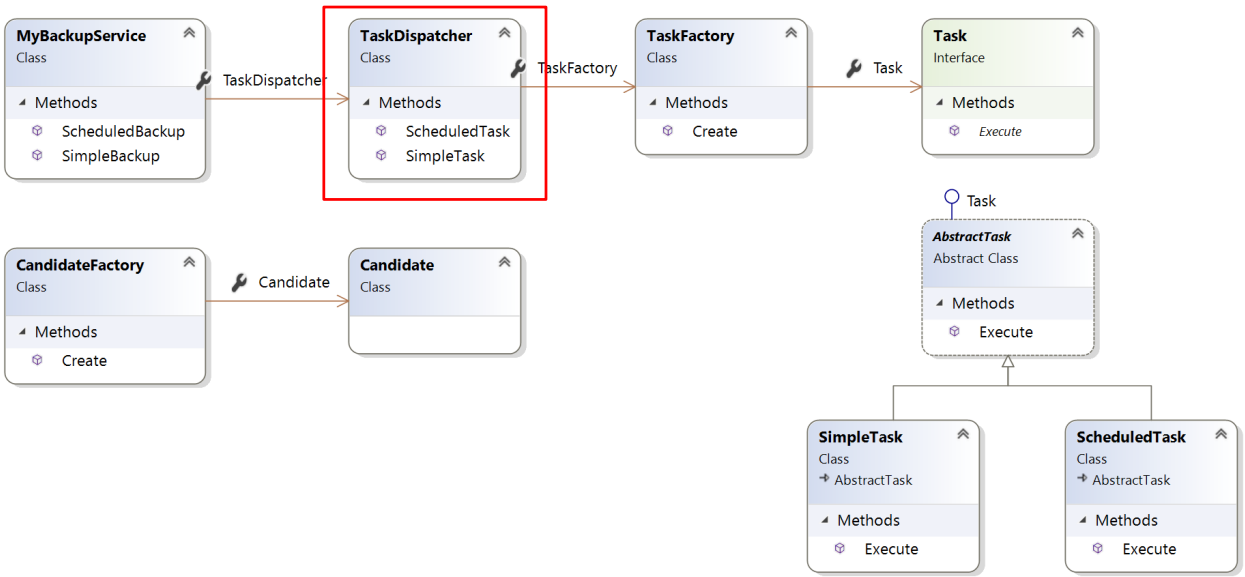
`MyBackupService` 身為 `Facade`，本身的職責就是將 `工作` 指派給專業的 `class` 去做，而非親自去實做，否則 `Facade` 依然會非常複雜而難以維護。

目前我們底層實做部份已經有一堆 `Manger`、`Handler` 與 `Finder` class，但使用端要的是 `簡單備份` 與 `排程備份` 兩個 `工作`，且這兩個 `工作` 又會有不同使用 `Manager`、`Handler` 與 `Finder` 方式，基於 `單一職責` 與 `專業分工` 原則：

- `簡單備份` 工作：由 `SimpleTask` class 負責
- `排程備份` 工作：由 `ScheduledTask` class 負責
- `抽象介面`：對外統一將 `工作` 抽象為 `Task` interface

> OOP 心法
>
> 實務上實現 SRP，就是每個 class / method 都 只做一件事情，然後將其他事情交給其他 專業 的 class / method 做，而不是想用一個 class / method 完成整件事情。
>
> class 多並不難維護，只要 class 的職責分配得合理，搭配適當的 namespace 管理，就會很好維護，比較可怕的是一個 method 一兩千行，一個 class 三四萬行，這才是最難維護的。

TaskDispatcher



當我們將 工作 抽象化成 `Task` 後，的確可以直接在 `MyBackupService` 去建立 `SimpleTask` 與 `ScheduledTask`，由於這兩個 class 都已經抽象化成 `Task` interface，也有相同的 `Execute()`，因此可以使用 多型 的方式操作 `Task`。

但這樣有個問題：

- `Facade` 主要職責是整合一堆底層 class，而分配 `Task` 已經超出原本 `Facade` 該有的職責

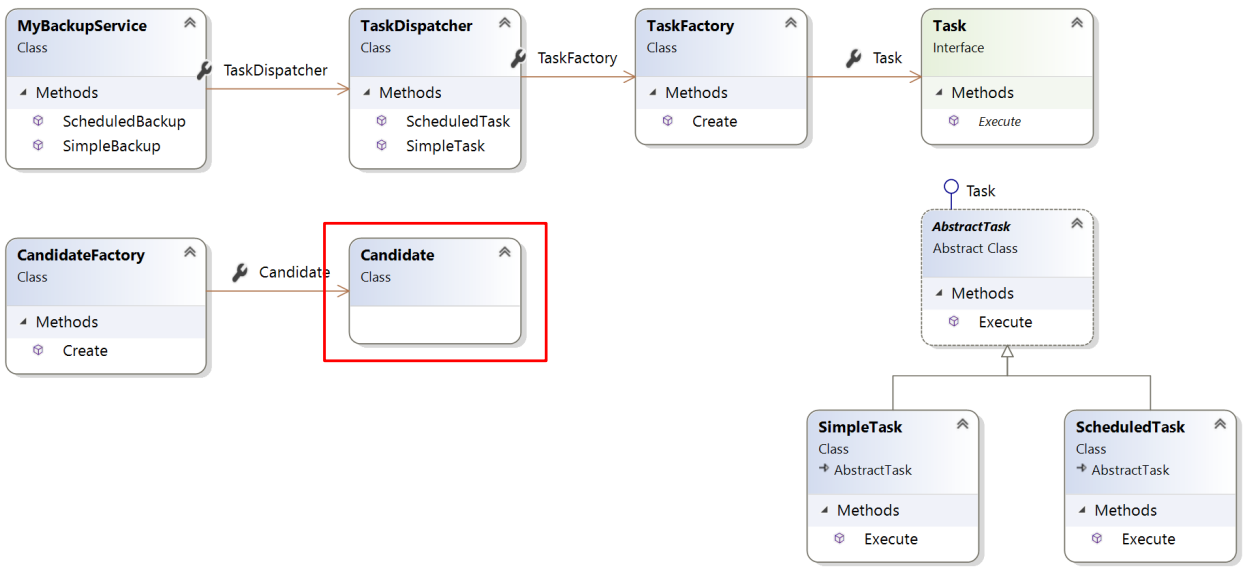
基於 單一職責 與 專業分工 原則：

- 分配 `Task`：由 `TaskDispatcher` class 負責

> OOP 心法
>
> SOLID 原則內，最難理解的是 依賴反轉原則，最難實現的是 單一職責原則。
>
> 主要是每個人對於 職責 的認知不同，而 職責 可大可小，因此 class / method 也可大可小，這就要靠經驗與團隊透過 code review

Implementation

Candidate



Candidate.cs

```
public class Candidate
{
     private readonly Config config;
     private readonly DateTime fileDateTime;
     private readonly string name;
     private readonly string processName;
     private readonly int size;

     public DateTime FileDateTime
     {
          get => this.fileDataTime;
     }

     public string Name
     {
          get => this.name;
     }

     public string ProcessName
     {
          get => this.processName;
     }

     public int Size
     {
          get => this.size;
     }

     public Candidate()
     {

     }


```

```
34      public Candidate(Config config, string name, DateTime
    fileDateTime, int size) {
35          this.config = config;
36          this.name = name;
37          this.fileDateTime = fileDateTime;
38          this.size = size;
39      }
40  }
```

第 3 行

```
1  private readonly Config config;
2  private readonly DateTime fileDateTime;
3  private readonly string name;
4  private readonly string processName;
5  private readonly int size;
```

將 field 宣告成 `readonly` ，只能由 consturctor 修改 field，其他 method 都不能修改。

14 行

```
1  public string Name
2  {
3      get => this.name;
4  }
```

只留下 property get，拿掉 property set，因此用戶端只能讀取 property，不能寫入 property。

> 簡單的說，`readonly` 是為了防止 class 內其他的 method 不能修改 field，只能由 constructor 修改，而只有 property get 是防止用戶端不能修改 property。
>
> readonly + property get 才算最嚴謹的寫法

CandidateFactory.cs

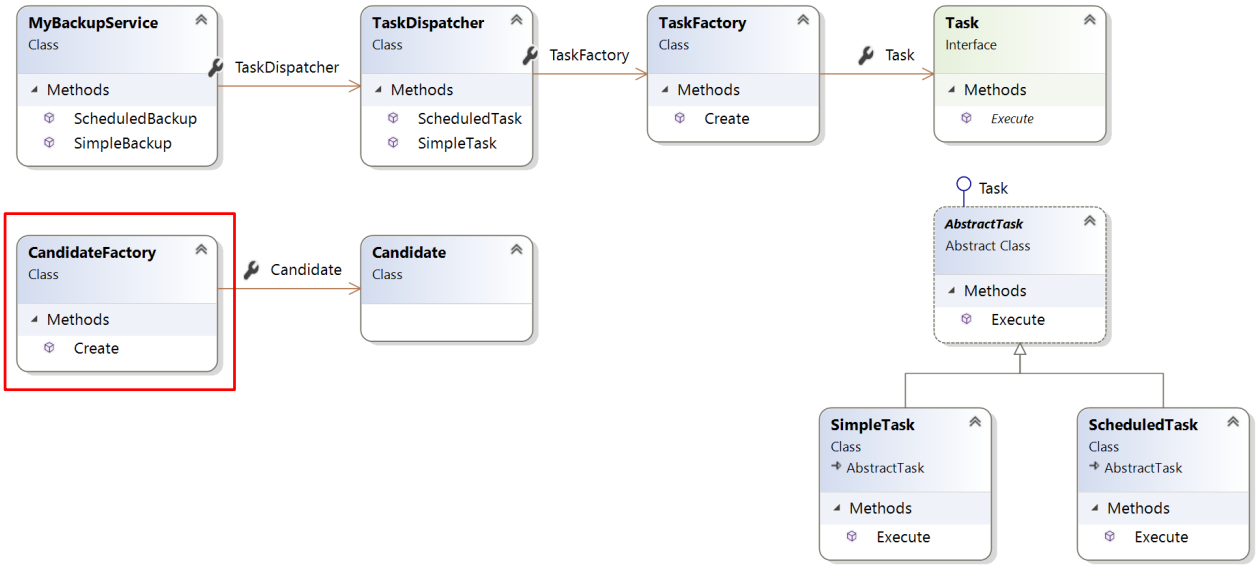
```
1 public class CandidateFactory
2 {
3     public static Candidate Create(Config config, string
  name, DateTime fileDateTime, long size)
4     {
5         return new Candidate(config, name, fileDateTime,
  size);
6     }
7 }
```

需求端要求 `提供一個一致的建立與初始化 Candidate 的方式`，因此使用 `Simple Factory Pattern` ，將原本到處 `new` 改成統一個 `CandidateFactory.Create()`。

> `Simple Factory Pattern` 幾乎每次上課都會出現，就表示這是實務上天天都要使用的方式，無論是 `依賴抽象，封裝變化` 的角度，或是 `增加界面，幫助擴展` 的看法，最後都會面臨在 `抽象` 前提下，到底要 new 哪個 class 達成 `多型` 的基本問題，`Simple Factory Pattern` 取代 `new`，提供了一致的建立與初始化 object 的方式，封裝了建立物件與初始化的邏輯，也避免 new 的邏輯到處散布，造成日後難以維護。

雖然已經獨立出 `CandidateFactory`，但使用端要使用 `new`，你也是擋不了，所以我們必須做更進一步的動作。

CandidateFactory



CandidateFactory.cs

```
namespace MyBackupCandidate
{
    public class CandidateFactory
    {
        public static Candidate Create(Config config, string name, DateTime fileDateTime, long size)
        {
            return new Candidate(config, name, fileDateTime, size);
        }
    }
}
```

將 `CandidateFactory` 搬到新的 `MyBackupCandidate` namespace，有別於原本的 `MyBackup` namespace。

Candidate.cs

```
namespace MyBackupCandidate
{
    public class Candidate
    {
        private readonly Config config;
        private readonly DateTime fileDateTime;
        private readonly string name;
        private readonly string processName;
        private readonly int size;

        public DateTime FileDateTime
        {
            get => this.fileDataTime;
        }

        public string Name
        {
            get => this.name;
        }

        public string ProcessName
        {
            get => this.processName;
        }

        public int Size
        {
            get => this.size;
        }

        internal Candidate()
        {

```

```
34          }
35
36          internal Candidate(Config config, string name,
    DateTime fileDateTime, int size) {
37              this.config = config;
38              this.name = name;
39              this.fileDateTime = fileDateTime;
40              this.size = size;
41          }
42      }
43  }
```

第 1 行

```
1  namespace MyBackupCandidate
2  {
3      public class Candidate
4      {
5      }
6  }
```

將 `Candidate` 改放在 `MyBackupCandidate` namespace 內，也就是目前 `Candidate` 與 `CandidateFactory` 都放在相同的 namespace 內。

31 行

```
internal Candidate()
{

}

internal Candidate(Config config, string name, DateTime
fileDateTime, int size) {
    this.config = config;
    this.name = name;
    this.fileDateTime = fileDateTime;
    this.size = size;
}
```

將 contructor 改成 `internal`，也就是只允許 `相同 namespace` 的 class 才可以 new，因此 `Candidate` 只剩下 `CandidateFactory` 可以 `new`，其他的 class 都不能 `new`。

LocalFileFinder.cs

```
1  public class LocalFinder
2  {
3      protected Candidate CreateCandidate(string fileName)
4      {
5          FileInfo fileInfo;
6          Candidate candidate;
7
8          if (File.Exists(fileName))
9          {
10             fileInfo = new FileInfo(fileName);
11             candidate =
   CandidateFactory.Create(this.config, fileName,
   fileInfo.CreationTime, fileInfo.Length);
12         }
13
14         return candidate;
15     }
16 }
```

11 行

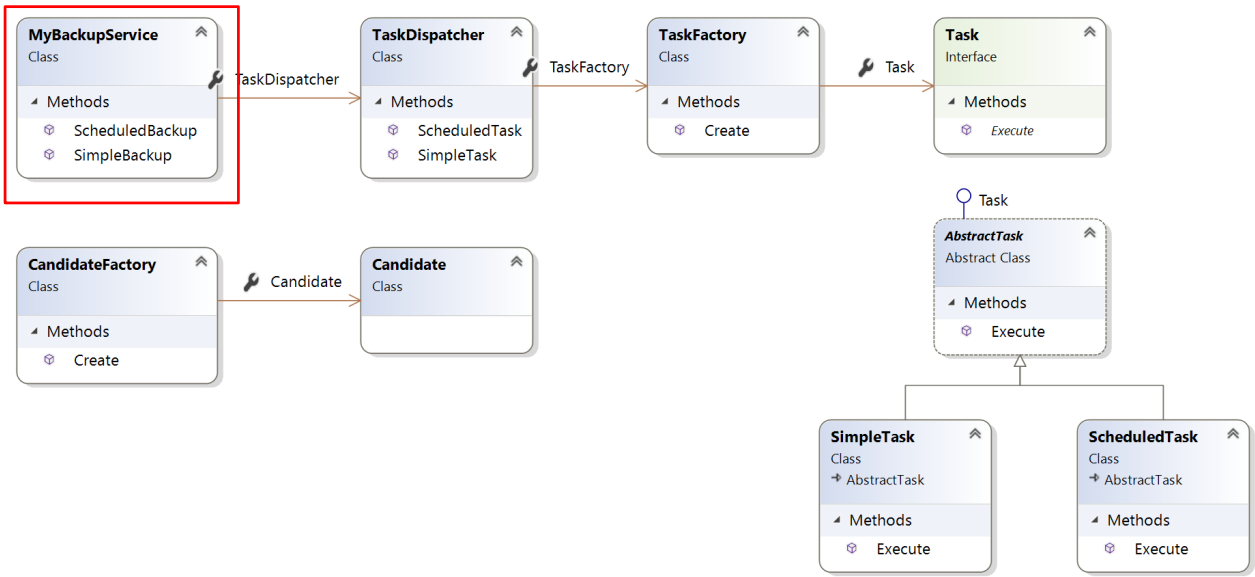
```
1  candidate = CandidateFactory.Create(this.config, fileName,
   fileInfo.CreationTime, fileInfo.Length);
```

由原本的 `new Candidate()` 改成 `CandidateFactory.Create()`。

這就解決了用戶端的 `避免用戶端不正當的建立 Candidate` 的需求，必須統一用 factory 建立 object，若還是使用 `new`，則會編譯錯誤。

> `Candidate` 算是一種 vaule object / data trasfer object，也就是目的只是呈現資料狀態，一開始建立後，就不應該被修改，且必須限制使用端必須透過 factory 建立，則以上算是最完整的作法。

MyBackupService



MyBackupService.cs

```
public class MyBackupService
{
    private List<JsonManager> managers;
    private TaskDispatcher taskDispatcher;

    public MyBackupService()
    {
        this.managers.Add(new ConfigManager());
        this.managers.Add(new ScheduleManager());
        this.taskDispatcher = new TaskDispatcher();

        this.Init();
    }

    private void Init()
    {
        this.ProcessJsonConfigs()
    }

    public void SimpleBackup()
    {
        this.taskDispatcher.SimpleTask(managers);
    }

    public void ScheduledBackup(
    {
        this.taskDispatcher.ScheduledTask(managers);
    }

    private void ProcessJsonConfigs()
    {
        for(int i = 0; i < this.managers.Count -1; i++)

            this.managers[i].ProcessJsonConfig();
```

```
34      }
35  }
```

> 使用端的統一入口

將原本的 `MyBackupService` 重構成只剩下 `SimpleBackup()` 與 `ScheduledBackup()` 兩個 method 而已。

由 `MyBackupService` 分派給 `TaskDispatcher()` 相對應的 method 處理。

第 3 行

```
1  private List<JsonManager> managers;
```

Homework 2 我們已經建立了 `managers`，負責放所有 manager，包含 `ConfigManager` 與 `ScheduleManager`。

第 4 行

```
1  private TaskDispatcher taskDispatcher;
```

本次新建立的 `TaskDispatcher`，負責根據需求分配不同的 task。

第 5 行

```
1  public MyBackupService()
2  {
3      this.managers.Add(new ConfigManager());
4      this.managers.Add(new ScheduleManager());
5      this.taskDispatcher = new TaskDispatcher();
6
7      this.Init();
8  }
```

直接使用 `new` 建立所需要的相依物件。

> Q : 這裡直接用 `new` 不怕 `MyBackupService` 與 `JsonManager` 與 `TaskDispatcher` 強耦合嗎？

A : 在 homework 2 曾經提到 OOP 有 4 種 class

- Domain class
- Hidden class
- Architecture class
- Helper class

主要我們是不希望用戶端與 domain class 耦合 (相依)，因此會墊一些 `中介` class，如 `Factory` 、`Manager` ... 之類的 architecture class，也就是利用耦合 architecture class，而換來解耦合 domain class。

因為 architecture class 是我們自己加上去的，我們可以控制，但 domain class 卻會伴隨著 domain 需求而變動，我們無法控制，因此我們希望能耦合一個 `不變` 的 class，而不是去耦合一個會 `變動` 的 class。

`JsonManager` 與 `TaskDispatcher` 皆屬於 architecture class，因此可以使用 `new` 直接耦合，換來與 `ConfigManager` / `ScheduleManager` 與 `SimpleTask` / `ScheduledTask` 的解耦合。

> OOP 心法
>
> 並不是什麼地方都一定要用 `Factory` 取代 `new`
>
> 會使用 `Factory` 的場合
>
> - Object 有 `抽換` 的需求 (需求容易變動)
> - Object 有根據不同邏輯 `new` 的需求 (避免邏輯到處散布)
> - 不想讓使用端直接 `new` object (value object / data transfer object)
>
> 會使用 `new` 場合

> - 目前穩定，沒有改變或抽換需求
> - 允許直接耦合的 class (如直接耦合 architecture class 或 helper class)
>
> 當有使用 `Factory` 需求出現時，再 `重構` 成 `Factory`。

15 行

```
1  private void Init()
2  {
3      this.ProcessJsonConfigs()
4  }
```

`MyBackupService` 的初始化程式碼寫在此。

> OOP 心法
>
> Constructor 與 `Init()` 的差別：
>
> 很多初學者會將初始化的程式碼直接寫在 constructor，而造成 constructor 的肥大，根據 `單一職責原則`：一個 class / method 應該只有一個 `職責`，constructor 的 `職責` 是建立相依物件，而 `Init()` 才是對相依物件做 `初始化` 的動作，所以應該將 constructor 與 `Init()` 的 `職責` 分開，而不是通通寫在 constructor 內

32 行

```
1  private void ProcessJsonConfigs()
2  {
3      for(int i = 0; i < this.managers.Count -1; i++)
4          this.managers[i].ProcessJsonConfig();
5  }
```

Homework 2 曾經寫過，以 `多型` 的方式將各種 `config.json` 與 `schedule.json` 讀進來轉成物件。

20 行

```
1  public void SimpleBackup()
2  {
3      this.taskDispatch.SimpleTask(managers);
4  }
```

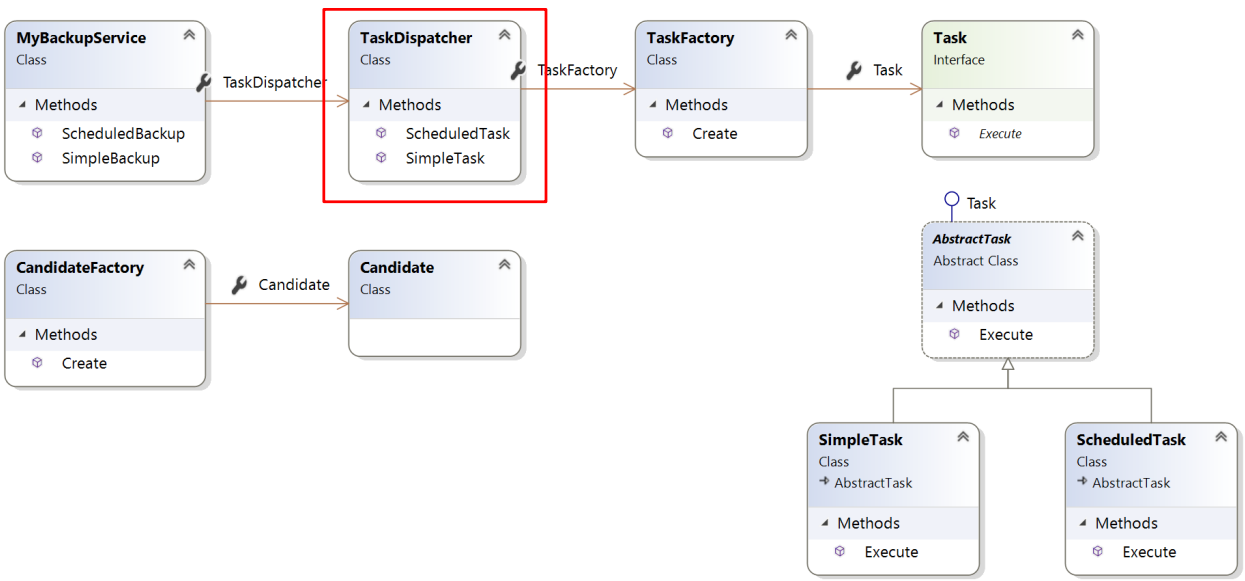
處理 `簡單備份`，將 `managers` 傳入 `TaskDispatcher.SimpleTask()`。

26 行

```
1  public void ScheduledBackup(
2  {
3      this.taskDispatcher.ScheduledTask(managers);
4  }
```

處理 `排程備份`，將 `managers` 傳入 `TaskDispatcher.ScheduledTask()`。

TaskDispatcher



TaskDispatcher.cs

```
1  public class TaskDispatcher
2  {
3      private Task task;
4
5      public void SimpleTask(List<JsonManager> managers)
6      {
7          ...
8          this.task = TaskFactory.Create("simple");
9          this.task.Execute(configs, null);
10     }
11
12     public void ScheduledTask(List<JsonManager> managers)
13     {
14         ...
15         this.task = TaskFactory.Create("scheduled");
16         this.task.Execute(configs, schedules);
17     }
18 }
```

> 根據需求分配不同的 task

第 5 行

```
1 public void SimpleTask(List<JsonManager> managers)
2 {
3     ...
4     this.task = TaskFactory.Create("simple");
5     this.task.Execute(configs, null);
6 }
```

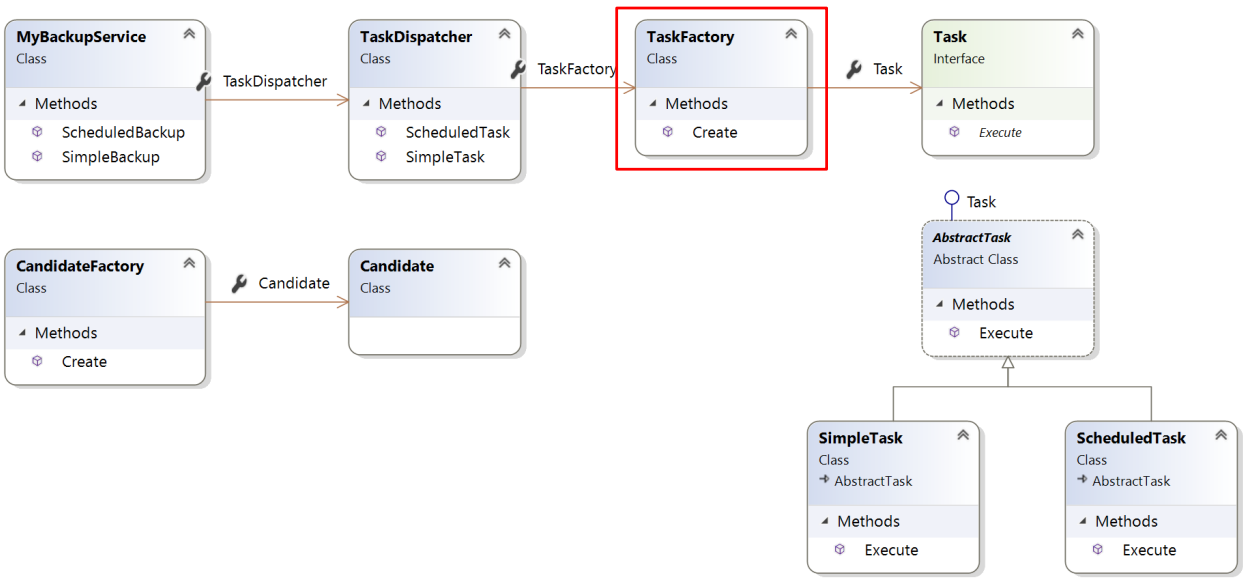
由 `TaskFactory` 建立 `SimpleTask`，再以 `多型` 執行 `Execute()`。

12 行

```
public void ScheduledTask(List<JsonManager> managers)
{
    ...
    this.task = TaskFactory.Create("scheduled");
    this.task.Execute(configs, schedules);
}
```

由 `TaskFactory` 建立 `ScheduledTask`，再以 `多型` 執行 `Execute()`。

TaskFactory



TaskFactory.cs

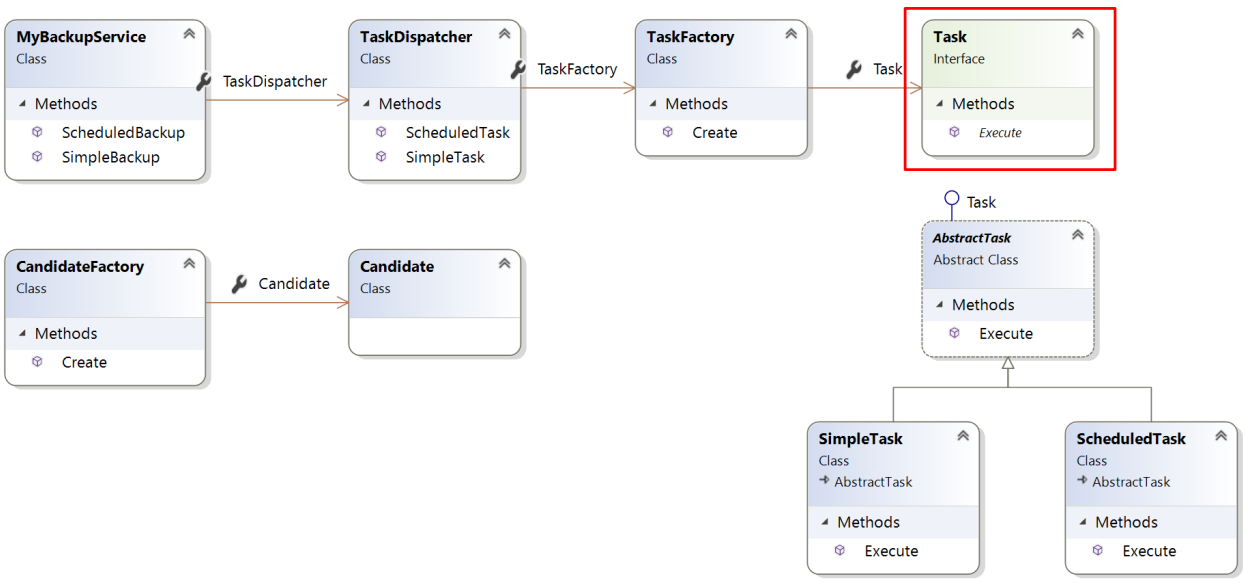
```
public static class TaskFactory
{
    public static Task Create(string task)
    {
        if (task == "simple")
            return new SimpleTask();
        else if (task == "scheduled")
            return new ScheduledTask();
        else
            return null;
    }
}
```

> 負責 new Task

由於目前只有兩個 `Task`，所以 `TaskFactory.Create()` 簡單使用 `if else` 即可，最少將 `if else` 封裝在 `Factory` 內。

若將來有更多的 `Task`，可考慮使用 homework 3 的 `reflection + 設定檔` 方式，將 `TaskFactory` `開放封閉` 起來。

Task



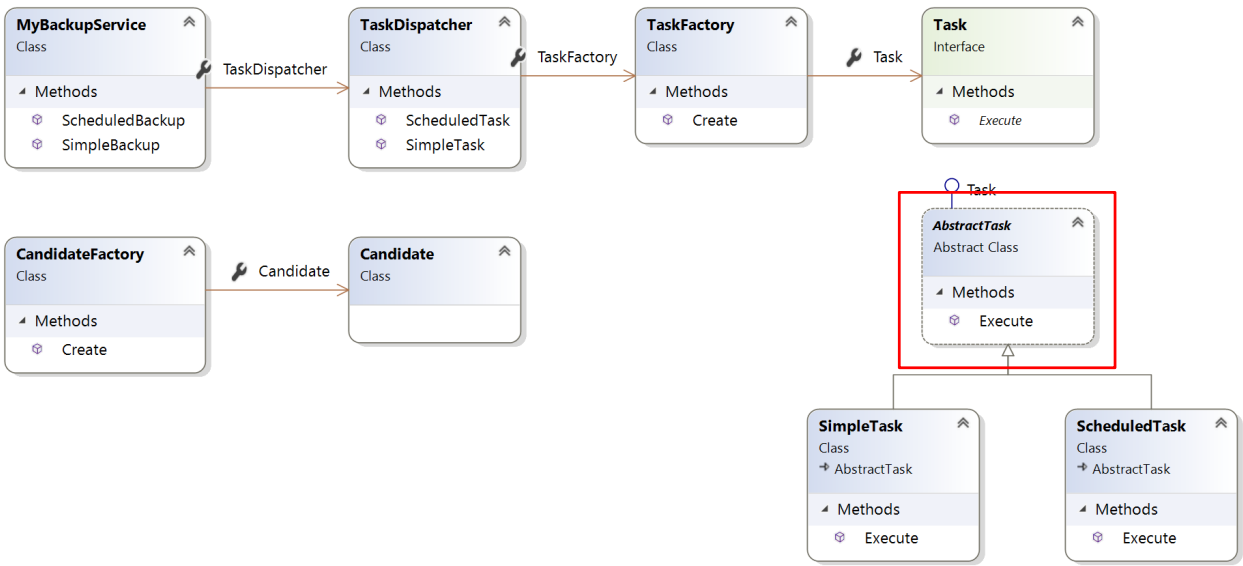
Task.cs

```
public interface Task
{
    void Execute(Config config, Schedule shedule);
}
```

> 將所有 task 抽象化成 `Task` interface，使用端只依賴 (認識、耦合) 此 interface，而不會依賴實際 task

只定義一個 `Execute()`，傳入 `Config` 與 `Schedule` 兩個物件。

AbstractTask



AbstractTask.cs

```
public abstract class AbstractTask : Task
{
    protected FileFinder fileFinder;

    public virtual void Execute(Config config, Schedule schedule)
    {
        this.fileFinder = FileFinderFactory.Create("file", config);
    }

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

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

    protected 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));

29          return handlers;
30      }
31  }
```

> 所有 Task 共用程式碼處

第 3 行

```
1  protected FileFinder fileFinder;
2
3  public virtual void Execute(Config config, Schedule
   schedule)
4  {
5      this.fileFinder = FileFinderFactory.Create("file",
   config);
6  }
```

無論是 `SimpleTask`，或是 `ScheduledTask` ，一開始一定要將要備份的檔案全部找出來，首先必須由 `FileFinderFactory` 建立正確的 `FileFinder`。

因為是共用的部份，所以寫在 `AbstractTask` 的 `Create()`，並宣告成 `virtual`，讓所繼承的 method 可以加以 `override`。

第 10 行

```
1  protected void BroadcastToHanders(Candidate candidate)
```

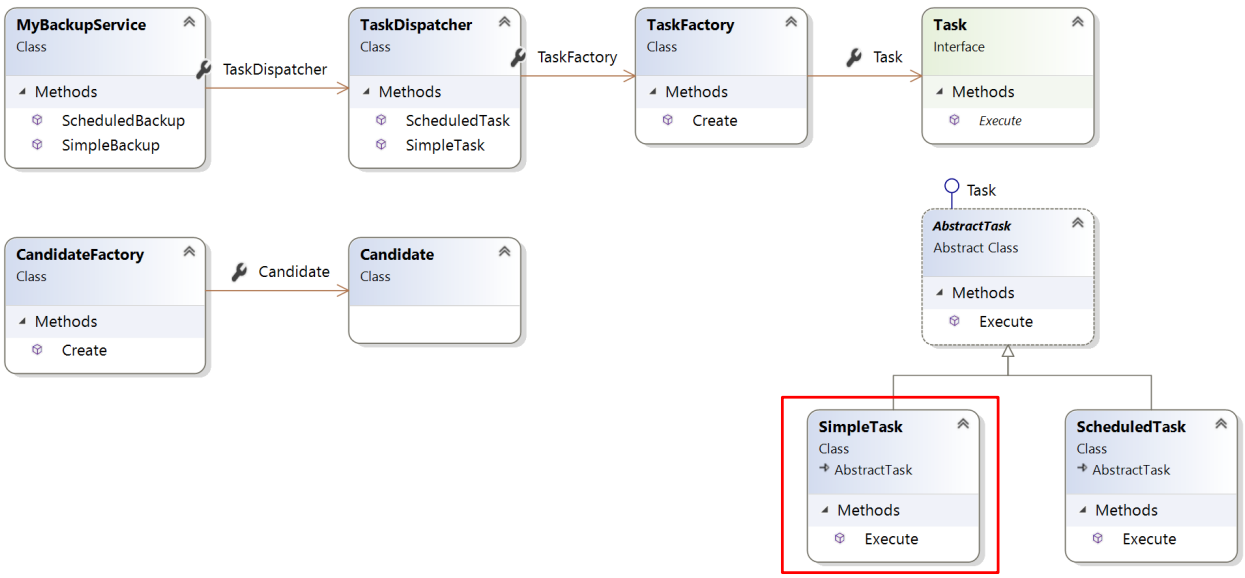
Homework 3 曾經在 `MyBackupService` 實做 `BroadcastToHandlers()`，將其重構到 `AbstractTask` 內。

18 行

```
protected List<Handler> FindHandlers(Candidate candidate)
```

Homework 3 曾經在 `MyBackupService` 實做 `FindHandlers()`，將其重構到 `AbstractTask` 內。

SimpleTask



SimpleTask.cs

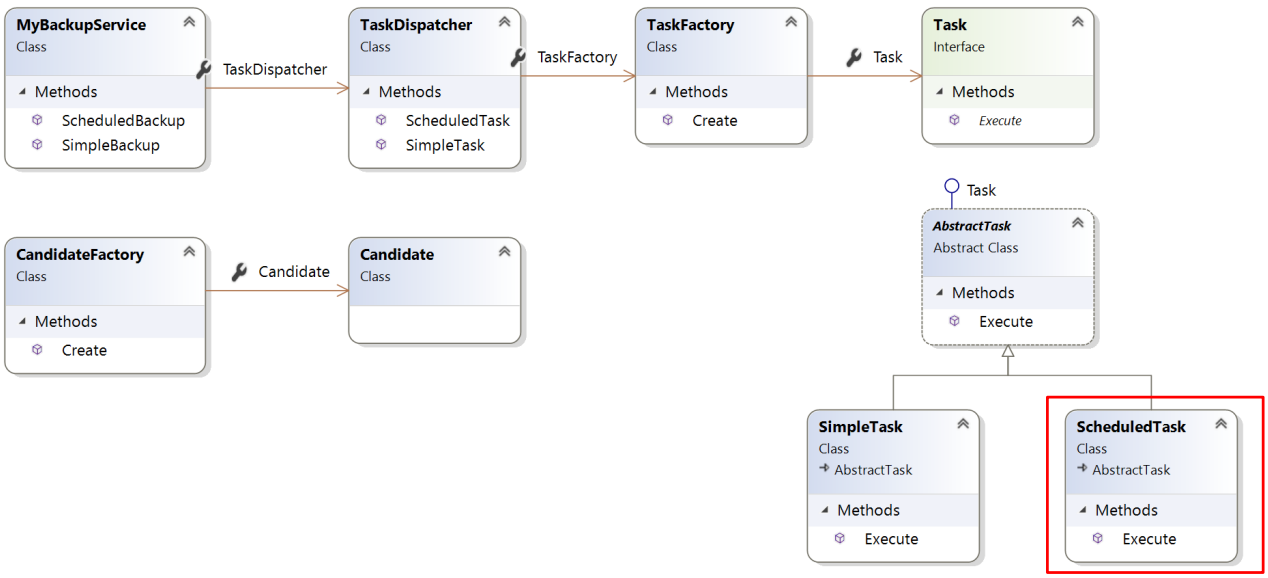
```
public class SimpleleTask : AbstractTask
{
    public override void Execute(Config config, Schedule schedule)
    {
        base.Execute(config, schedule);

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

使用 `base.Execute()` 找到全部要備份的檔案。

在對所有的檔案以 broadcast 方式執行所有 handler 。

ScheduledTask



ScheduledTask.cs

```
1  public class ScheduledTask : AbstractTask
2  {
3      public override void Execute(Config config, Schedule
   schedule)
4      {
5          base.Execute(config, schedule);
6
7          ...
8
9          foreach(Candidate candidate in this.fileFinder)
10             this.BroadcastToHandlers(candidate);
11     }
12 }
```

一樣使用 `base.Execute()` 找到全部要備份的檔案。

最後在對所有的檔案以 broadcast 方式執行所有 handler 。

由於要以排程方式備份，這裡請同學自行加以實做。

> Q : 我們從 homework 1 到 homework 5 我們建立了很多 class，感覺是將簡單的問題複雜化，我們真的需要建立這麼多 class / interface 嗎？

A : OOP 最主要就兩個實務上手法

- 建立 `中介 class` 封裝 `複雜度` : 如 `Facade`、`Factory`、`Manager` ... (封裝)
- 建立 `Interface` 增加 `擴充點` : 將來 `擴充` 就有此 interface 開始，其他部份 `開放封閉` (繼承、多型)

當然代價就是會增加 class / interface 與程式碼複雜度，所以 OOP 應該要用在刀口上，一開始若覺的目前某部份 `不可能變動`，`不可能會擴展`，可以不使用 OOP，不使用 Design Pattern，不開一堆 class 與 interface，直接耦合即可；一旦需求改變，發現有 `變動` 與 `擴展` 的需求時，就要趕緊 `重構` 改用 OOP，而不是繼續 `if else` 下去，讓架構持續惡化。

> OOP 心法
>
> 沒有完美的架構，只有最適合的架構
>
> `完全不設計` 與 `過度設計`，都不是好的設計

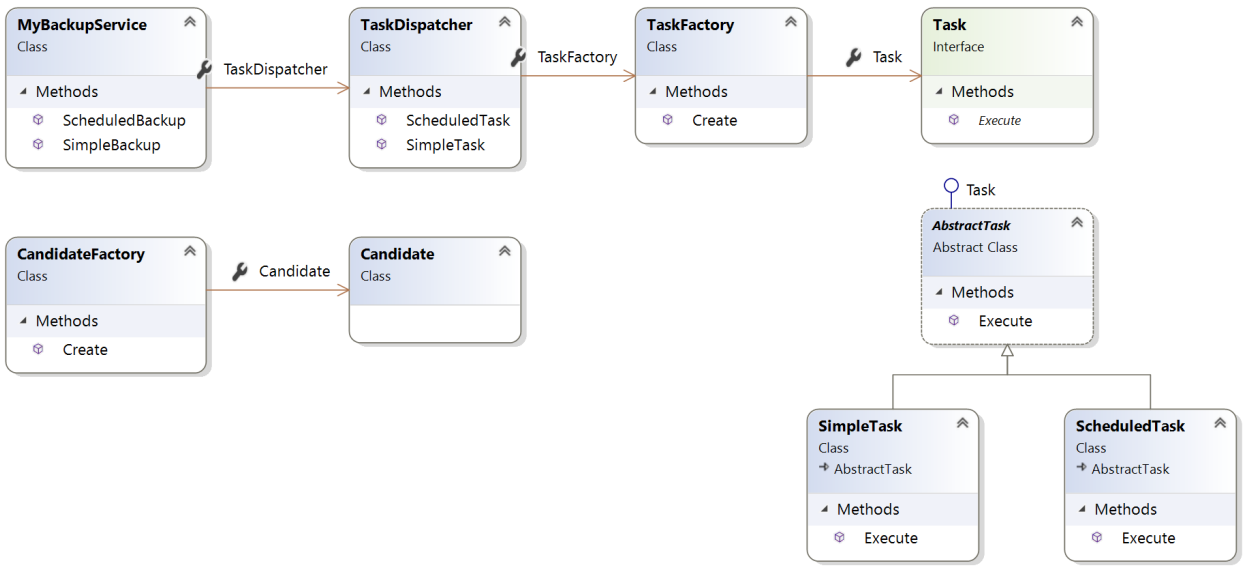
Summary

- `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` : 一個 class / method 應該只有一個 `職責`
- `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

Conclusion



- 程式語言不限，請依照 class diagram
    - 將原本 `Candidate` 使用 `CandidateFactory` 加以重構
    - 新增 `TaskDispatcher`、`TaskFactory` 與 `Task` 家族
    - 將 `MyBackupService` 大部分程式碼重構到 `AbstractTask`
    - 實做出 `ScheduleTask`，並依照 `schedule.json` 時間進行 `排程備份`
    - 請用 PlantUML 將 class diagram 畫出來