OOP #2

Sam Xiao, Oct.18, 2017

Goal

在 homework 1 我們已經將 `真實事物` 使用 class 加以模擬，我們將以此 4 個 class 加以重構成更好的 OOP 架構。

Outline

Recap

- `抽象化`：由 `真實事務` 抽象成 `object`
- `抽象化`：再由 `object` 抽象成 `class`
- `Manager Object`：
  - 提供 object 的 container
  - 提供操作 object 的 method

User Story

使用端角度

我們在 homework 1 設計了 `ConfigManager` 與 `ScheduleManager` 兩個 class，因此使用端必須分別 `new` 兩個 class 之後，各自執行其 `ProcessConfigs()` 與 `ProcessSchedules()`：

```
ConfigManager configManager = new ConfigManager();
configManager.ProcessConfigs();
SheduleManager scheduleManager = new ScheduleManager():
sheduleManager.ProcessShedules();

(DoBackup()) ???
```

但對於使用端來說，這有幾個問題：

- 使用端必須了解太多內部 class 資訊

使用端必須知道 `ConfigManager` 與 `ScheduleManager` 兩個 class，還必須知道 `ProcessConfigs()` 與 `ProcessSchedules()` 兩個 method 才能順利讀取 JSON，增加了使用端的複雜度。

- 分成兩個 class 後，接下來動作該怎麼辦？

讀取完 JSON 後，接下來的功能是 `DoBackup()`，而 `DoBackup()` 該屬於 `ConfigManager`？還是屬於 `ScheduleManager` 呢？這會讓使用端感到疑惑。

開發端角度

- 兩個 class 出現重複的程式碼

`ConfigManager` 與 `ScheduleManager` 都需讀取 JSON，在`ProcessConfigs()` 與 `ProcessSchedules()` 內已經出現結構很類似的程式碼。

Task

使用端角度

回想一下使用其他 library 或 framework 的經驗，我們總希望：

> 使用的 class 越少越好
>
> 使用的 method 越少越好

因此使用端許願如下：

- 只使用 `MyBackupService` 單一 class 即可處理所有需求
- 只呼叫 `ProcessJSONConfig()` 單一 method 即可讀取 `config.json` 與 `schedule.json`
- 繼續呼叫 `MyBackupService` 的 `DoBackup()` 即可執行備份

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

`MyBackup` 這種 `單一入口` class 的寫法，有以下明顯的優點：

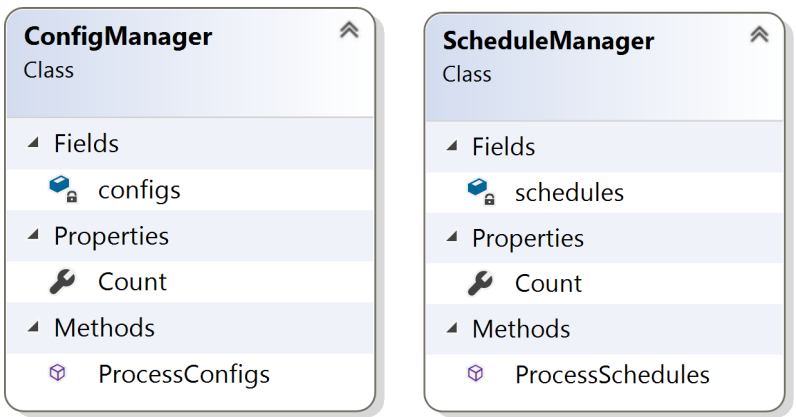
- 使用端程式碼更加簡潔
- 使用端不須了解內部 class 太多資訊，隱藏類別的複雜度
- 繼續使用同一個 class 的 `DoBackup()` ，也就是單一 class 即可完成工作

> 最小知識原則 LKP : Least Knowledge Principle
>
> 一個物件應該對其他物件有最少的了解

所以我們應該再新增 `MyBackupService` ，將 `ConfigManger` 與 `ScheduleManager` 做進一步的 `封裝` 。

> OOP 心法
>
> 以 `使用端` 程式碼最精簡、最容易理解、最不用修改為最高原則，而不是 `開發端` 。

開發端角度



`ConfigManager` 與 `ScheduleManager` 的結構相近，程式碼也類似， 因此適合使用 繼承 將 `ConfigManager` 與 `ScheduleManager` 抽象成 `JsonManager`，將重複的部分抽取出來，這樣相同的程式碼就只有一分在 `JsonManager`，而不會重複出現在 `ConfigManager` 與 `ScheduleManager` 中。

> DRY 原則
>
> Don't Repeat Yourself
>
> 相同的程式碼只寫一次

Architecture

綜合 使用端 與 開發端 角度，我們有了以下結論：

- 使用端 希望新增 `MyBackup` 為單一入口降低複雜度
- 開發端 希望新增 `JsonManager` 將 `ConfigManager` 與 `ScheduleManager` 抽象化，並將重複的程式碼抽到 `JsonManager`

MyBackup

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

`MyBackupService` 接受使用端的呼叫來讀取 JSON 檔案，並把工作分配給已經設計好的 `ConfigManager` 與 `ScheduleManager` ，對使用端來說，好像是 `MyBackupService` 完成的。

第一個版本

MyBackupService.cs

```
1  class MyBackupService
2  {
3      private ConfigManager configManager;
4      private ScheduleManager scheduleManager;
5
6      public MyBackupService()
7      {
8          this.configManager = new ConfigManager;
9          this.scheduleManager = new ScheduleManager;
10     }
11
12     public void ProcessJsonConfigs()
13     {
14         this.configManager.processConfigs();
15         this.scheduleManager.ProcessSchedules();
16     }
17 }
```

第 3 行

```
1  private ConfigManager configManager;
2  private ScheduleManager scheduleManager;
```

將 `ConfigManager` 與 `ScheduleManager` 封裝到 private field。

第 6 行

```
1  public MyBackupService()
2  {
3      this.configManager = new ConfigManager;
4      this.scheduleManager = new ScheduleManager;
5  }
```

在 constructor 去 `new` `ConfigManager` 與 `ScheduleManager` 物件。

```
class MyBackupService
{
    public MyBackupService(ConfigManager configManager,
ScheduleManager scheduleManager) {}
}
```

若你使用的程式語言 / framework 有支援 DI (Dependency Injection)，可直接在 constructor 的參數列注入相依物件

> .NET : .NET Core 2 PHP : Laravel TypeSript : Angular

12 行

```
public void ProcessJsonConfigs()
{
    this.configManager.ProcessConfigs();
    this.scheduleManager.ProcessSchedules();
}
```

用戶端雖然呼叫 `ProcessJsonConfigs()`，但會轉發到 `ConfigManager.ProcessConfigs()` 與 `Schedule.ProcessSchedules()`。

這樣寫法的缺點：

`ProcessJsonConfigs()` 直接呼叫 `ConfigManager.ProcessConfigs()` 與 `ScheduleManager.ProcessShedules()`。

若將來新增 `PlateformManager`，則勢必還要修改 `ProcessJsonConfigs()`。

> 由於 `ConfigManager` 與 `ScheduleManager` 非常類似，有抽象化的本錢，因此我們再進一步將 `ConfigManager` 與 `ScheduleManager` 抽象化成 `JsonManager`。

> OOP 心法
>
> OOP 就是要 `擁抱變化`，要時時刻刻想著當需求變動時，怎樣的寫法才能讓修改 code 最少

第二個版本

MyBackupService.cs

```
1  class MyBackupService
2  {
3      private List<JsonManager> managers;
4
5      public MyBackupService()
6      {
7          this.managers.Add(new ConfigManager());
8          this.managers.Add(new ScheduleManager());
9      }
10
11     public void ProcessJsonConfigs()
12     {
13        for(int i = 0; i < this.managers.Count -1; i++)
14            this.managers[i].ProcessJsonConfig();
15     }
16 }
```

第 3 行

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

改用 List 存放所有 managers，其中 List 的泛型為 `JsonManager`，也就是將 `ConfigManager` 與 `ScheduleManager` 抽象化成 `JsonManager`。

JsonManager.cs

```
1  abstract class JsonManager
2  {
3      public abstract void ProcessJsonConfig();
4  }
```

因為 `ConfigManager` 與 `ScheduleManager` 已經抽象化成 `JsonManager`，因此原本的 `ProcssConfig()` 與 `ProcessShedule()` 需被進一步的抽象化成 `ProcessJsonConfig()`。

ConfigManager.cs

```
class ConfigManager: JsonManager
{
    public override void ProcessJsonConfig()
    {
        ...
    }
}
```

`ConfigManager` 繼承於 `JsonManager`，因此原本的 `ProcessConfig()` 需重構成 `ProcessJsonConfig()`。

ScheduleManager.cs

```
class ScheduleManager: JsonManager
{
    public override void ProcessJsonConfig()
    {
        ...
    }
}
```

`ScheduleManager` 繼承於 `JsonManager`，因此原本的 `ProcessSchedule()` 需重構成 `ProcessJsonConfig()`。

PlateformManager.cs

```
class PlateformManager: JsonManager
{
    public override void ProcessJsonConfig()
    {
        ...
    }
}
```

若將來新增 `PlateformManager`，只需同樣繼承 `JsonManager`，並將讀取 XML 部分也寫在 `ProcessJsonConfig()` 內即可，`MyBackupService` 的 `ProcessJsonConfigs()` 不用做任何修改。

MyBackupService.cs

```
class MyBackupService
{
    public MyBackup()
    {
        this.managers.Add(new ConfigManager());
        this.managers.Add(new ScheduleManager());
        this.managers.Add(new PlateformManager());
    }
}
```

當然 `MyBackupService` 的 constructor 一樣要 `this.managers.Add(new PlateformManager());`，這個無法避免，但最少 `MyBackupService` 的修改已經縮小到 constructor 而已。

```
class MyBackupService
{
    public void ProcessJsonConfigs()
    {
       for(int i = 0; i < this.managers.Count -1; i++)
           this.managers[i].ProcessJsonConfig();
    }
}
```

因為 `ProcessConfigs()` 與 `ProcessSchedules()` 都已經被抽象化成相同的 `ProcessJsonConfig()`，因此只要用 for 迴圈執行一次 `ProcessJsonConfig()` 即可。

就算將來新增了 `PlateformManager`，因為也是繼承 `JsonManager`，所以一樣也是 `ProcessJsonConfig()`，因此 `ProcessJsonConfigs()` 永遠都不用修改。

> 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

JsonManager

重構前

ConfigManager.cs

```
class ConfigManager
{
    private const string PATH = @"../../Configs/config.json";
    private List<Config> configs = new List<Config>();

    public void ProcessConfigs()
    {
        object configObject = this.GetConfigObject();

        foreach (var item in configObject["configs"])
        {
            this.configs.Add(new Config(
                    (string)item["ext"],
                    (string)item["location"],
                    (bool)item["subDirectory"],
                    (string)item["unit"],
                    (bool)item["remove"],
                    (string)item["handler"],
                    (string)item["destination"],
                    (string)item["dir"],
                    (string)item["connectionString"]
                );
            );
        }
     }

    private object GetConfigObjet()
    {
        string configValue = File.ReadAllText(this.PATH);
        return configObj =
JsonConvert.DeserializeObject(configValue);
    }
}
```

ScheduleManager.cs

```
class ScheduleManager
{
    private const string PATH =
@"../../Configs/schedule.json";
    private List<Schedule> schedules = new List<Schedule>();

    public void ProcessSchedules()
    {
        object scheduleObject = this.GetScheduleObject();

        foreach (var item in scheduleObject["schedules"])
        {
            this.schedules.Add(new Schedule(
                (string)item["ext"],
                (string)item["time"],
                (string)item["interval"]
            ));
        }
    }

    private object GetScheduleObject()
    {
        string configValue = File.ReadAllText(PATH);
        return JsonConvert.DeserializeObject(configValue);
    }
}
```

`ConfigManager.GetConfigObject()` 與 `ScheduleManager.GetScheduleObject()`，除了 method 名稱不一樣外，程式碼完全一樣。

這類 `重複` 的程式碼，就適合透過 `重構` 搬到 `JsonManager`。

重構後

JsonManager.cs

```
abstract class JsonManager
{
    protected object GetJsonObject()
    {
        string configValue = File.ReadAllText(PATH);
        return JsonConvert.DeserializeObject(configValue);
    }

    public abstract void ProcessJsonConfig();
}
```

將 `ConfigManager.GetConfig()` 與 `ScheduleManager.GetSchedule()` 抽取到 `JsonManager`，並改名為 `GetJsonObject()`。

> `JsonManager` 之所以要使用 `abstract class`，而不使用 `interface`，因為要將 `ConfigManager` 與 `ScheduleManager` 重複的程式碼搬到 `JsonManager`。

ConfigManager.cs

```
class ConfigManager: JsonManager
{
    private const string PATH = @"../../Configs/config.json";
    private List<Config> configs = new List<Config>();

    public void ProcessJsonConfig()
    {
        object configObject = this.GetJsonObject();

        foreach (var item in configObject["configs"])
        {
            this.configs.Add(new Config(
                    (string)item["ext"],
                    (string)item["location"],
                    (bool)item["subDirectory"],
                    (string)item["unit"],
                    (bool)item["remove"],
                    (string)item["handler"],
                    (string)item["destination"],
                    (string)item["dir"],
                    (string)item["connectionString"]
                );
            );
        }
    }
}
```

第 8 行

```
object configObj = this.GetJsonObject();
```

改成呼叫繼承的 `GetJsonObject()`。

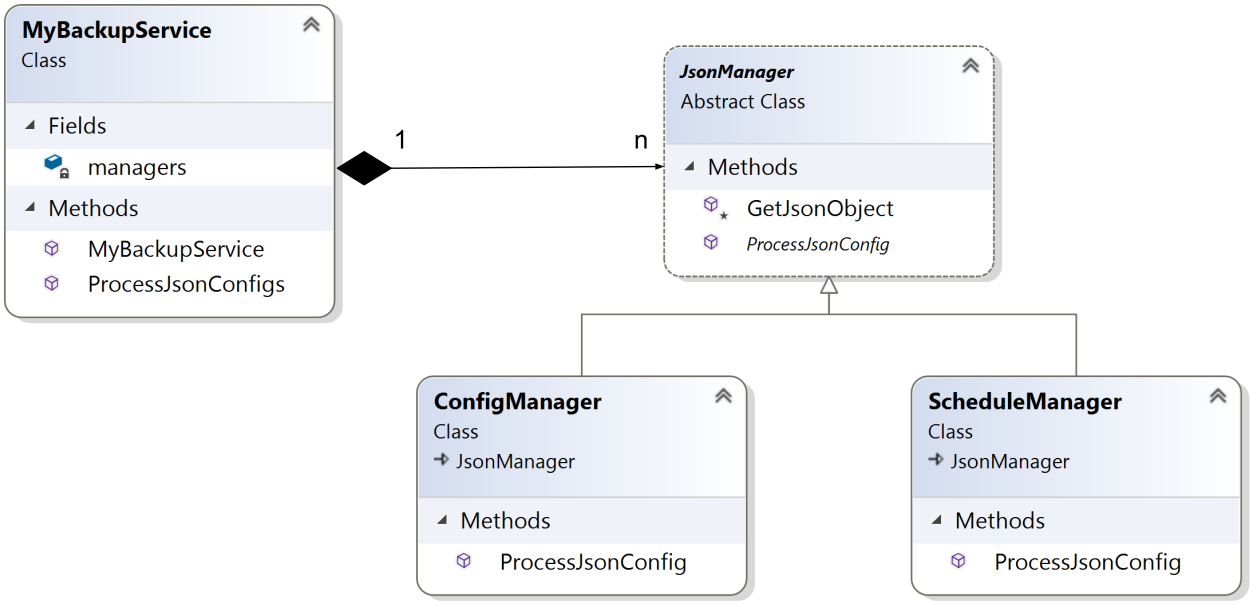
ScheduleManager.cs

```
1  class ScheduleManager: JSONManager
2  {
3      private const string PATH =
   @"../../Configs/schedule.json";
4      private List<Schedule> schedules = new List<Schedule>();
5
6      public void ProcessJsonConfig()
7      {
8          object scheduleObject = this.GetJsonObject();
9
10         foreach (var item in scheduleObject["schedules"])
11         {
12             this.schedules.Add(new Schedule(
13                 (string)item["ext"],
14                 (string)item["time"],
15                 (string)item["interval"]
16             ));
17         }
18     }
19 }
```

第 8 行

```
1  object scheduleObject = this.GetJsonObject();
```

改成呼叫繼承的 `GetJsonObject()`。



Class 的種類

Class 可分為 4 大類：

- Domain Class
- Hidden Class
- Architecture Class
- Helper Class

Domain Class

Domain 中最容易被發現的 class，代表 domain 的特定事物。

> 這些 class 需要進一步定義他們之間的關係，如垂直的 `繼承` 關係，水平的 `使用` 關係。

如由 `config.json` 找到 `Config` class，`schedule.json` 找到 `Schedule` class，屬於 domain class。

Hidden Class

雖然屬於 domain 的 class，但在一開始不容易被發現。

> 需要 senior 的經驗，或者逐漸 `重構` 才會被發現。

如 `ConfigManager` 與 `ScheduleManager` 一開始不容易被發現，屬於 hidden class。

Architecture Class

由 developer 決定使用的 architecture 或 design pattern 所衍生的 class。

> 也稱為 artificial class。

如 `JsonManager` 是為了將 `ConfigManager` 與 `ScheduleManager` 抽象化而產生的 class，而 `MyBackup` 為了簡化用戶端使用，這些屬於 architecture class。

Helper Class

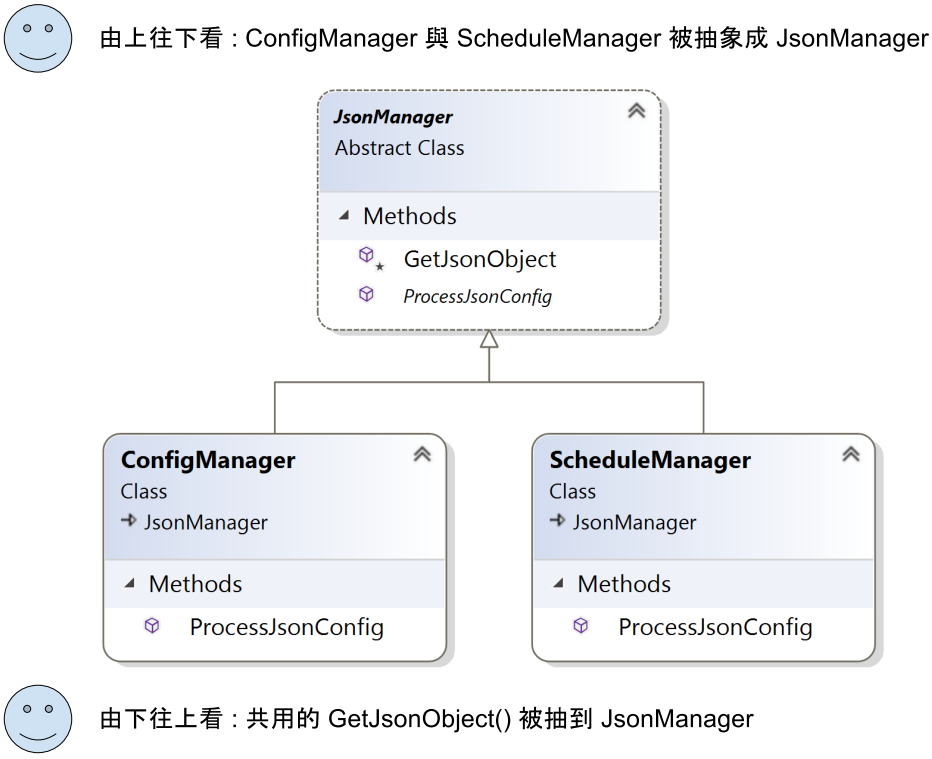
用來幫助其他 class 完成工作的 class。

> 如某些功能會大量被其他 class 重複使用，會在 `重構` 階段整理出 helper class。

繼承、封裝與多型

繼承

> 由上往下看 (由外往內看): 結構相似的 class 被抽象化成另外一個 class，將來可以 `多型` 方式使用。
>
> 由下往上看 (由內往外看): 共用的 method 被提到 parent class 共用，將來可重複使用該 method

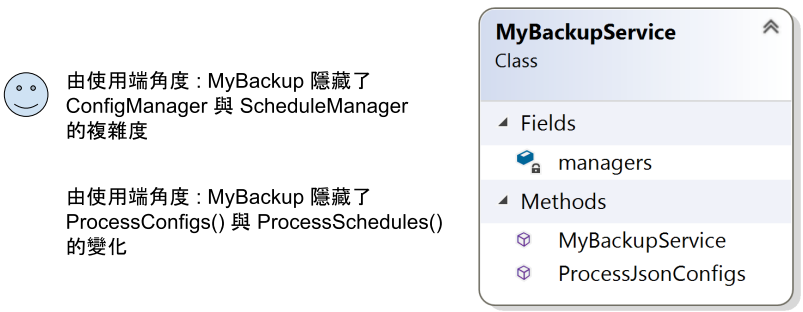


> 典型的誤用繼承
>
> 因為要 class 的某個功能，就去繼承該 class

你應該是要去 new 該 class (或 DI 注入)，而不是去繼承該 class。

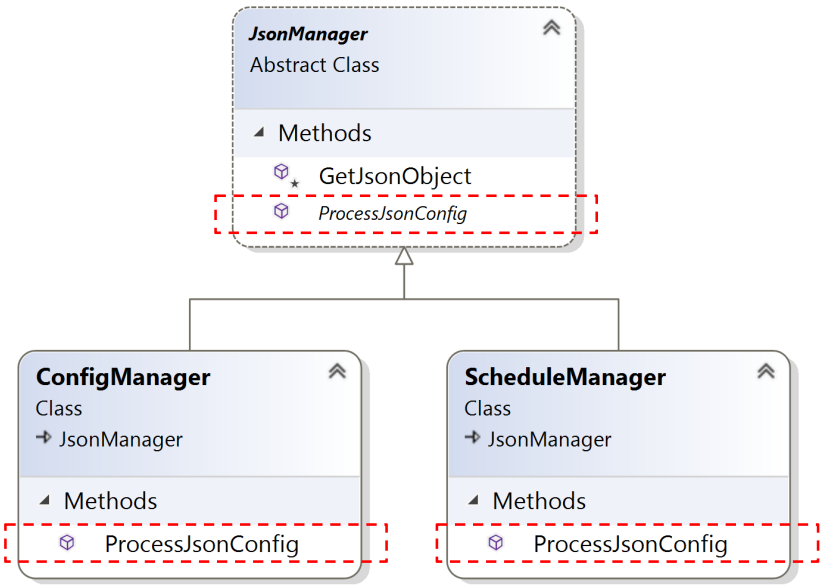
封裝

> 隱藏外界所不必要知道的資訊
>
> 隱藏行為的變化



多型

> 以抽象化的 method 使用其他物件的 method



由原本的 `ConfigManager.ProcessConfigs()` 與 `ScheduleManager.ProcessShedules()` 抽象化成 `JsonManager.ProcessJsonConfig()`。

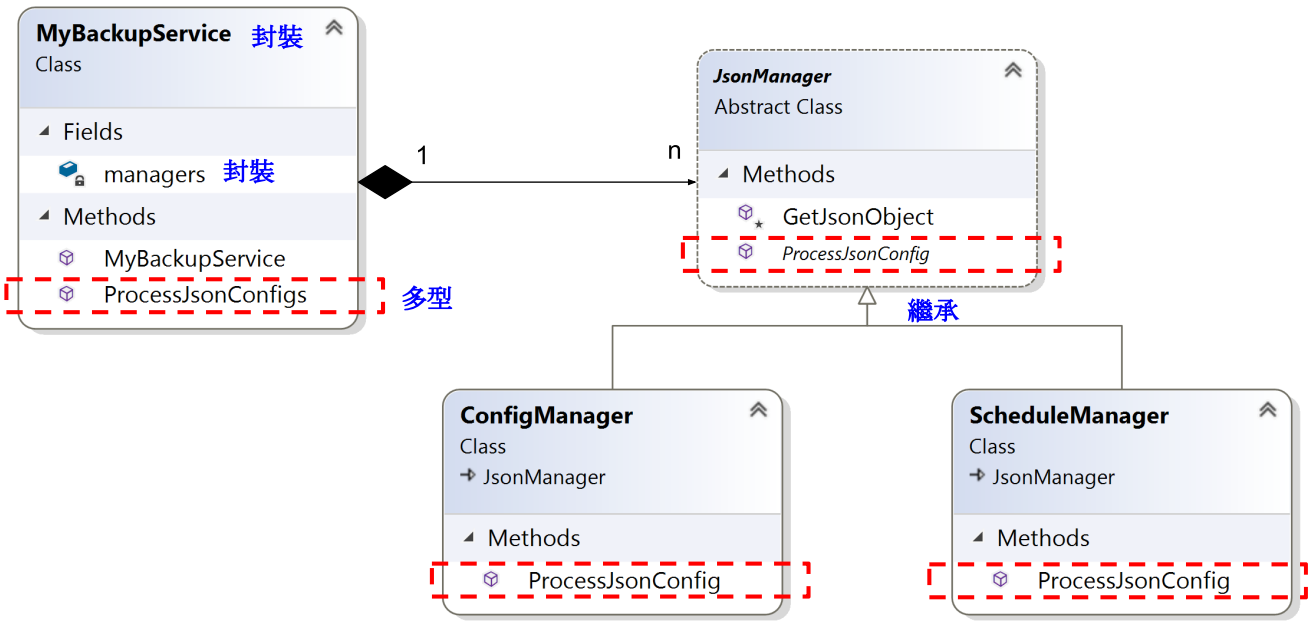
MyBackupService.cs

```
class MyBackupService
{
    public void ProcessJsonConfigs()
    {
       for(int i = 0; i < this.managers.Count -1; i++)
           this.managers[i].ProcessJsonConfig();
    }
}
```

因此使用端只需抽象地使用 `ProcessJsonConfig()` 即可，而不用明確地使用 `ProcessConfigs()` 與 `ProcessSchedules()`。

物件導向

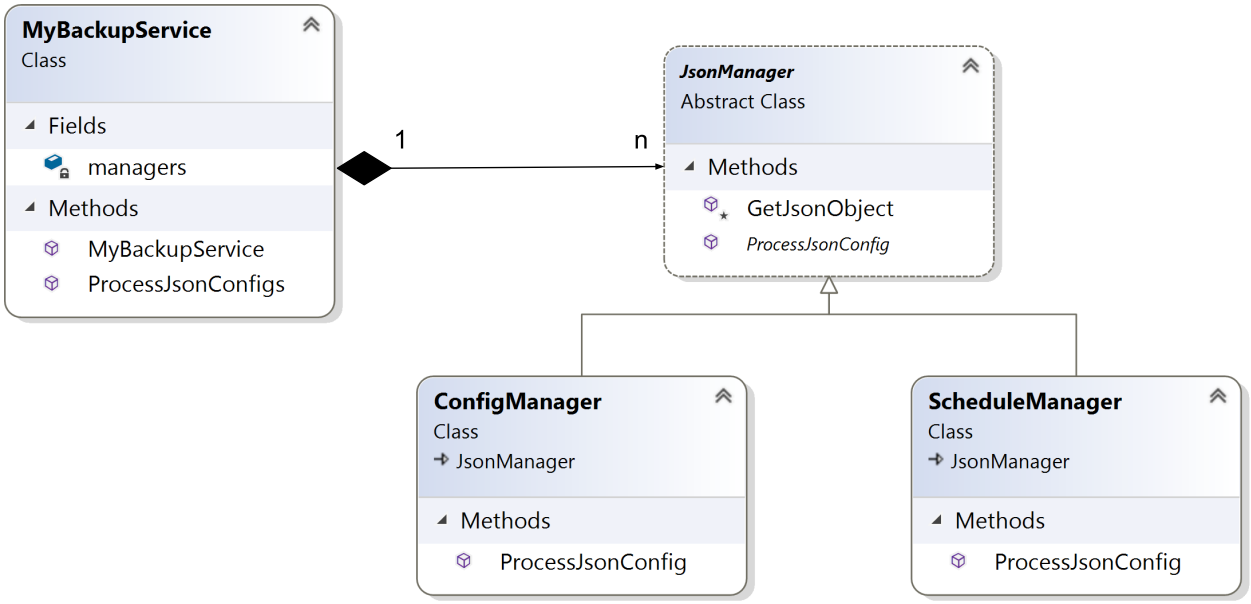
> 模擬世界，加以處理
>
> 依賴抽象，封裝變化



`多型` 為 OOP 的結果，`繼承` 與 `封裝` 的手段，目的就是 `依賴抽象，封裝變化`。

- `多型`：無論需求怎麼變化，從使用端抽象看起來都是一樣的
- `繼承`：將變化加以抽象化
- `封裝`：將變化藏在 class 內部，因此使用端不用跟著需求修改

Conclusion



- 程式語言不限，請依照 class diagram
  - 新增 `MyBackupService` 與 `JsonManager`
  - 重構 `ConfigManager` 與 `ScheduleManager`
  - 將 `ProcessJsonConfig()` 內重複的程式碼重構到 `JsonManager`