Homework 1

Sam Xiao, Oct.10, 2017

Goal

- 透過 8 次連貫的作業 (8 次 sprint)，實際以 OOP 的方式從無到有設計一個專案
- OOP 不再只是理論，透過模仿學習，實際感受 OOP 的開發方式
- OOP 不限語言，你可以使用自己擅長的語言實現

Outline

Homework 1
    Goal
    Outline
    User Story
    Use Case
    Traditional Method
    Task
    Specification
        config.json
        schedule.json
    Architecture
        從真實事物找出 Object
        從 Object 抽象化成 Class
        設計 Class 行為
            Config Class
            ConfigManager Class
                Fields
                Properties
                Methods

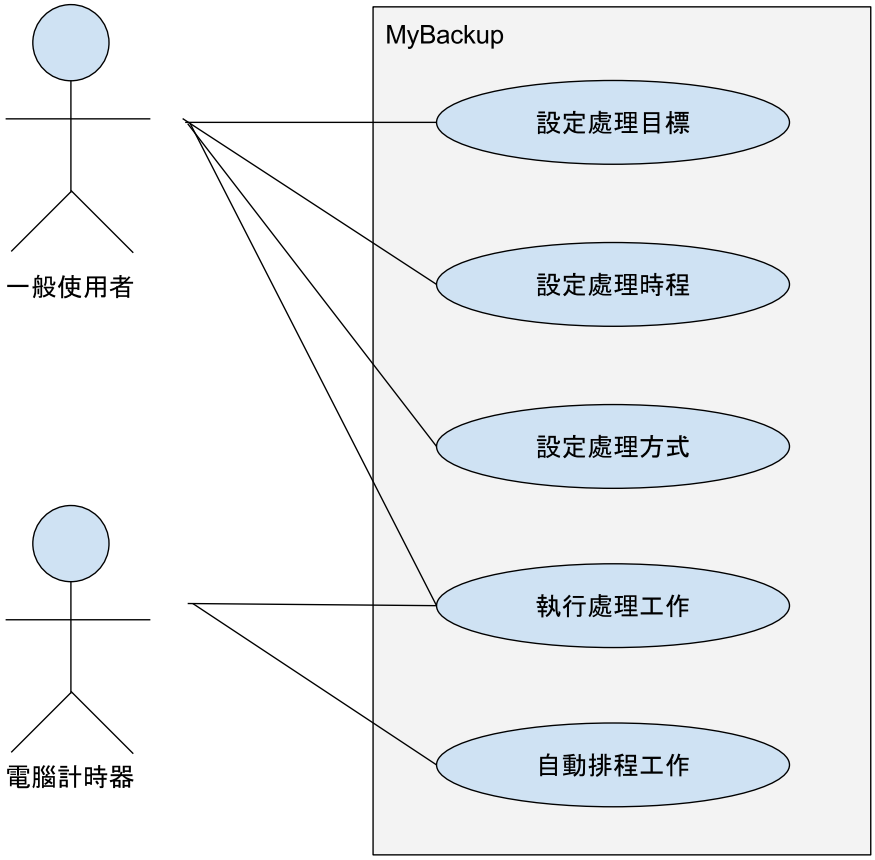
Others
Example
Schedule Class
ScheduleManager Class
Fields
Properties
Methods
Others
Example
Conclusion

User Story

Sam 的資料越來越多，雖然 SSD 越來越便宜，但現在 SSD 普遍都是 TLC，讀寫次數越來越少，因此想寫一個備份程式，能手動、也能自動備份檔案。

- 程式碼檔案 (*.cs) : 壓縮後存到特定目錄下
- Word 檔案 (*.docx) : 加密後存進資料庫
- 圖片檔 (*.jpg) : 直接存到特定目錄下

Use Case



Traditional Method

```
if (IsCSharpFile(theFile))
    HandleCSharpFile();

if (IsWordFile(theFile))
    HandleWordFile();

if (IsJpgFile(theFile))
    HandleJpgFile();
```

- 每加一種新的檔案格式，就要去加一個 `if`
- 新加入的程式碼需要再次測試
- 新加入的程式碼會不會把原來的程式碼改壞？

> 是否有一致方法處理各種不同格式檔案？不用每加一種格式就必須修改程式碼？
>
> 有什麼方法可以確保程式碼品質，盡量降低新加入程式碼對於原有程式碼的影響？

```
switch theDestination
{
    case ToDatabase :
        SaveToDatabase();
        break;
    case ToDirectory :
        SaveToSpecificDirectory();
        break;
    case ToCompress:
        DoCompressFile();
        break;
}
```

可是 (*.cs) 是 `壓縮` 後存到特定目錄下，而 (*.docx) 是 `加密` 後存進資料庫

假如將來有需求是 `壓縮` 後再 `加密` 呢？

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。

> 有沒有一致的方式來處理檔案的目的地？
>
> 能不能夠讓不同的目的地能夠具有彈性的結合在一起？例如結合壓縮到特定目錄，或壓縮再加密才存進資料庫？
>
> 若有新的處理方式也能夠新增，能不用修改原本的程式碼？

Task

所有的檔案格式、處理方式與排程，都使用 JSON 檔案設定。

- 建立 JSON 檔案
  - `config.json`：設定檔案格式與處理方式
  - `schedule.json`：設定自動排程時間
- `MyBackup` 讀取 `config.json` 與 `schedule.json`

Specification

config.json

- `configs`：以陣列儲存多筆 `config` 物件
- `ext`：設定檔案格式
- `location`：設定要備份檔案的目錄
- `subDirectory`：是否處理子目錄，`true`：處理子目錄；`false`：不處理子目錄
- `unit`：設定備份單位，`file`：以單一檔案為處理單位；`directory`：以整個目錄為處理單位
- `remove`：處理完是否刪除檔案，`true`：刪除；`false`：不刪除
- `handler`：`zip`：壓縮；`encode`：加密
- `destination`：處理後要儲存到什麼地方，`directory`：目錄；`db`：資料庫
- `dir`：處理後的目錄
- `connectionString`：設定資料庫連接字串

config.json

```
{
    "configs": [
        {
            "ext": "cs",
            "location": "c:\Projects",
            "subDirectory": true,
             "unit": "file",
             "remove": false,
             "handler": "zip",
             "destination": "directory",
             "dir": "c:\MyArchieves",
             "connectionString": ""
        },
        {
            "ext": "DOCX",
            "location": "c:\Documents",
            "subDirectory": true,
            "unit": "file",
            "remove": false,
            "handler": "encode",
            "destination": "db",
            "dir": "",
            "connectionString": "MyConnectionString"
        },
        {
            "ext": "jpg",
            "location": "c:\Pictures",
            "subDirectory": true,
            "unit": "file",
            "remove": false,
            "handler": "",
            "destination": "directory",
            "dir": "c:\MyArchieves",
            "connectionString": ""
        }
```

```
    ]
}
```

schedule.json

- `shedules`：以陣列儲存多筆 `shedule` 物件
- `ext`：設定此排程所處理的檔案格式
- `time`：設定此排程所處理的時間
- `interval`：設定此排程執行的間隔

schedule.json

```
{
    "schedules": [
        {
            "ext": "cs",
            "time": "12:00",
            "interval": "Everyday"
        },
        {
            "ext": "docx",
            "time": "20:00",
            "interval": "Everyday"
        },
        {
            "ext": "jpg",
            "time": "7:00",
            "interval": "Monday"
        },
    ]
}
```

Architecture

若不使用 OOP，直覺會寫出以下的程式碼：

```
ProcessConfigFile();
ProcessSheduleFile();
DoBackup();
```

也就是都以 function 的方式去實踐。

從真實事物找出 Object

> OOP
>
> 模擬世界，加以處理

所以第一步就是要從 `真實世界` 中找出 object，再由 object 歸納出 class。

在真實世界，我們看到了 `config.json` 與 `schedule.json`：

- `config.json`：在 `configs` 中，封裝了每種檔案格式的組態資訊，因此可以將每組 `{}` 看成 `config` 組態物件
- `schedule.json`：在 `schedules` 中，封裝了每個種排程的組態資訊，因此可以將 `{}` 看成 `schedule` 組態物件

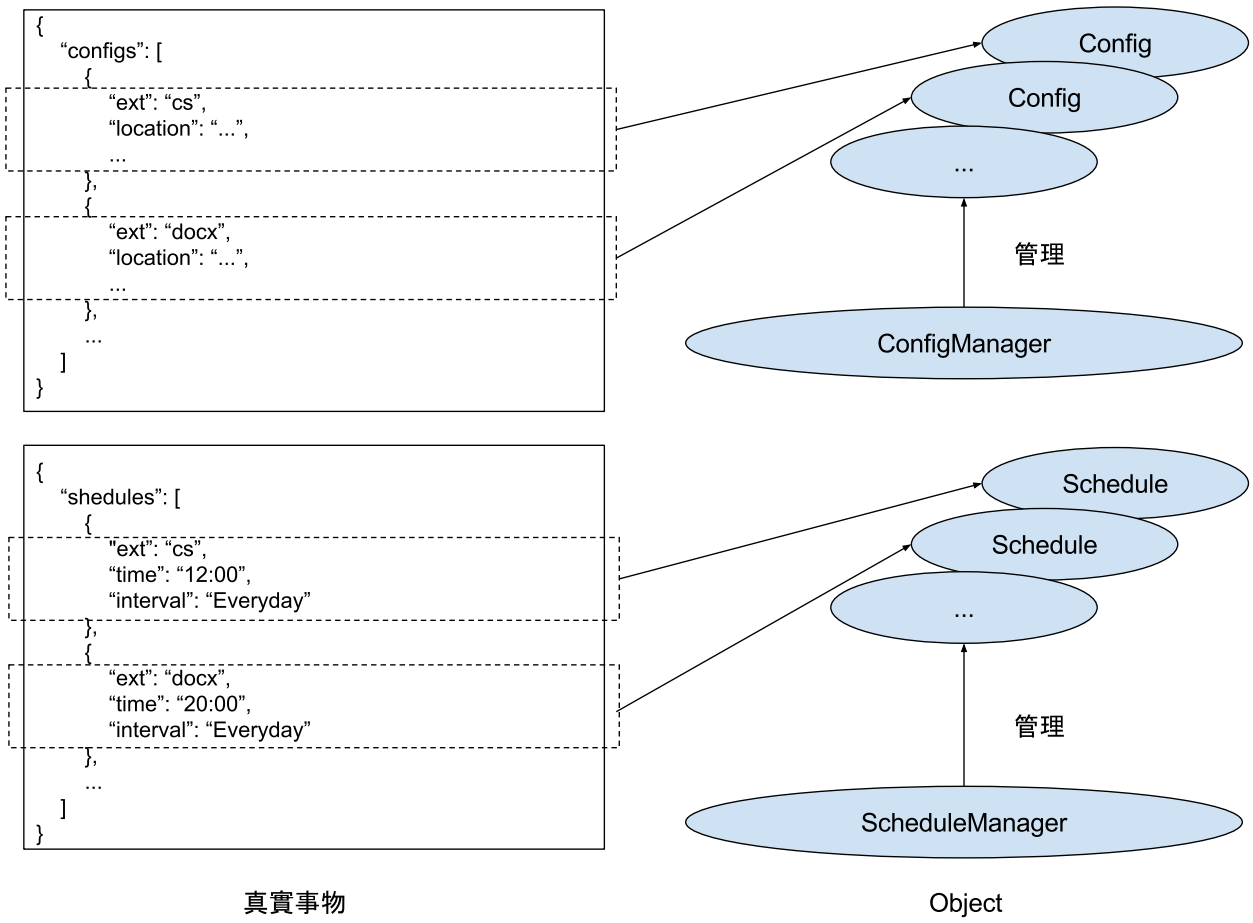
在 `物件的五大特性中` 談到：

> 物件必須有一個資料結構存放資料
>
> 物件必須要有行為

以 `config.json` 為例，`config.json` 包含多筆 `config` 物件與 ，因此我們還需要 `ConfigManager` 物件來存放多筆 `config` 物件 `資料`，也必須有 `行為` 將 `config.json` 轉換成 `config` 物件。

> Manager Object
>
> 提供 container 存放 value object，並提供 method 用來管理 value object

白話就是：我們需要一個 `主管` 來 `管理` 純粹的 value object。

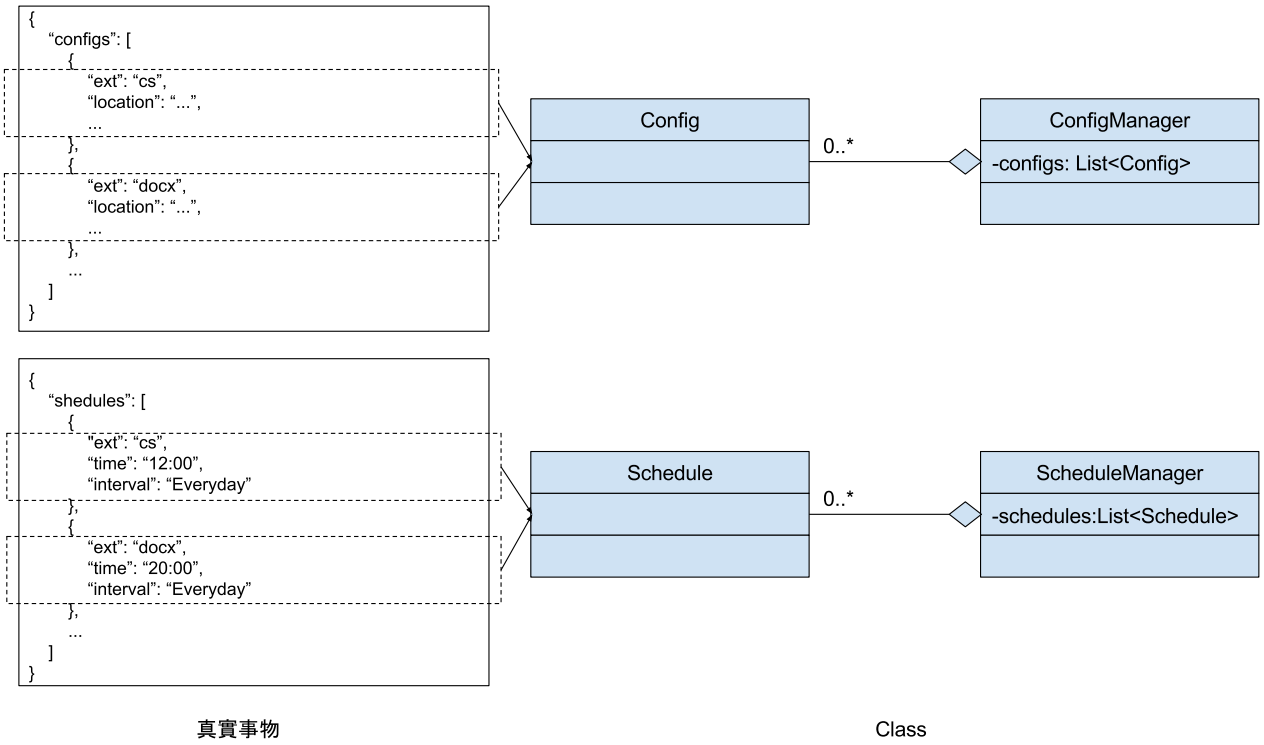


將 `真實事物` 完全使用 `物件` 來模擬。

從 Object 抽象化成 Class

> Class
>
> 將 object 加以分類
>
> 對 object 的抽象化

找出物件之後，我們發現存在著多個 `資料格式` 相同的 `Config` 與 `Schedule` 物件，因此我們想繼續將 object 抽象化成 class。



以 `Config` 物件為例，被抽象化為 `Config` class，至於多個 `Config` 物件，則被封裝在 `ConfigManager` 的 `List<Config>` 內。

> 物件 5 大特性
>
> 物件必須有一個資料結構存放資料

如此我們就將 `真實事務` 完全改用 class 加以模擬，也就是 objcect 的進一步抽象化。

> 實務上 OOP 就是這樣先由 `真實事物` 找出 object，再由 object 抽象化成 class，也就是所謂的 OOA/D (Object Oriented Analysis/Design)

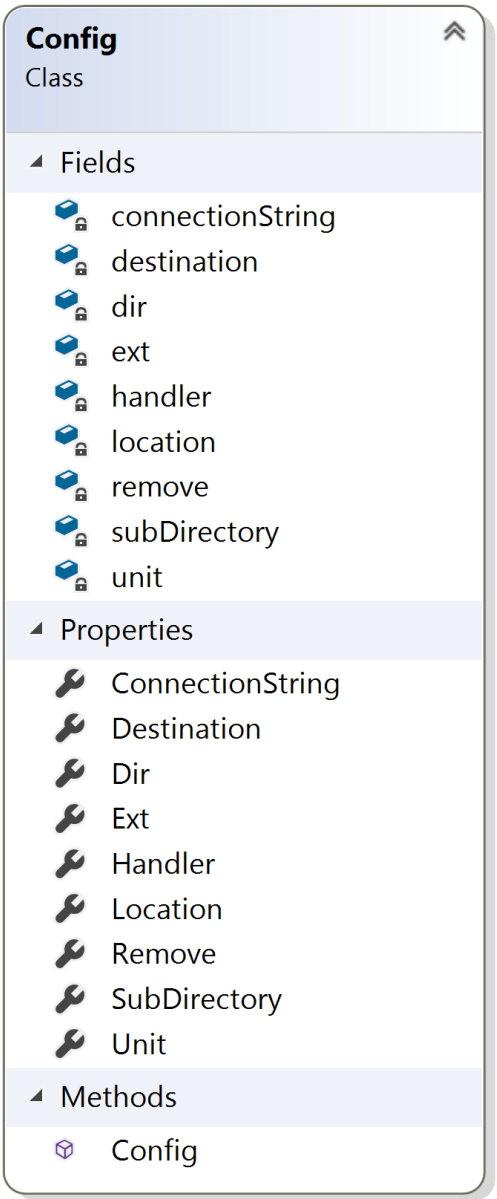
設計 Class 行為

> 物件 5 大特性
>
> 物件必須要有行為

目前只是將 class 的大架構決定，並將 `Config` 物件封裝在 `ConfigManager` 內，但 class 的 `行為` 則尚未設計。

Config Class

`Config` 主要功能是封裝 `設定檔案格式與處理方式` 的 JSON 成為物件，因此沒有 `行為`，有的只是 read only 的 property。



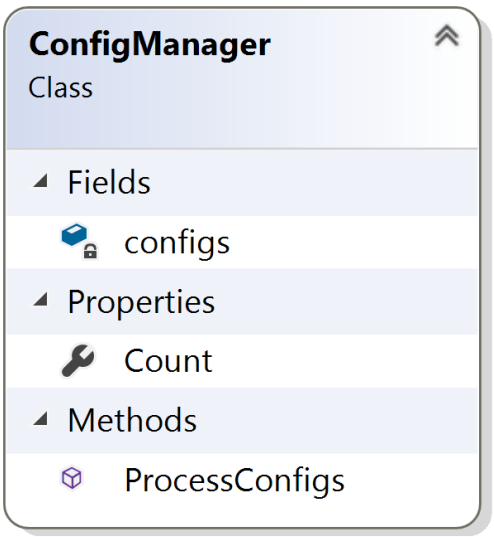
- `Config` 的 constructor 接受使用端的初始化資料，稍後會由 `ConfigManager` 建立
- `Config` 的 property 都是 `read only` ，因為 `Config` 為 `config.json` 的封裝，所有資料應該來自於 `config.json` ，而不該給使用端任意修改，是個純粹的 value object
- constructor 的 signature 如下：

```
public Config(string ext, string location, string subDirectory,
              string unit, string remove, string handler,
              string destination, string dir, string
connectionString);
```

> `Config` 屬於 `易於變動` 的 class，將來只要 `config.json` 有所改變，`Config` 就一定要跟著改變，若使用端直接相依於 `Config` ，將來跟著變動的機率就很大；但 `ConfigManager` 就是比較 `穩定` 的 class，適合用戶端直接相依，這也是另外設計 `ConfigManager` 的理由之一。
>
> OOP 時，要知道每個 class 是否 `易於變動`，對於易於變動的 class，最好另外建立穩定manager class 加以 `封裝`，讓用戶端改相依 manager class。

ConfigManager Class

`ConfigManager` 主要功能是將 `config.json` 轉成 `Config` 物件。



Fields

- `configs` : 提供 List 存放多筆 `Config` 物件

Properties

- `Count` : 提供 List 的 `筆數`

Methods

- `ProcessConfigs()`：將 `config.json` 轉成 `List<Config>`

Others

- 提供類似 array 方式讀取 `List<Config>`

> 若你使用的程式語言沒有 indexer 機制，可不用實作

Example

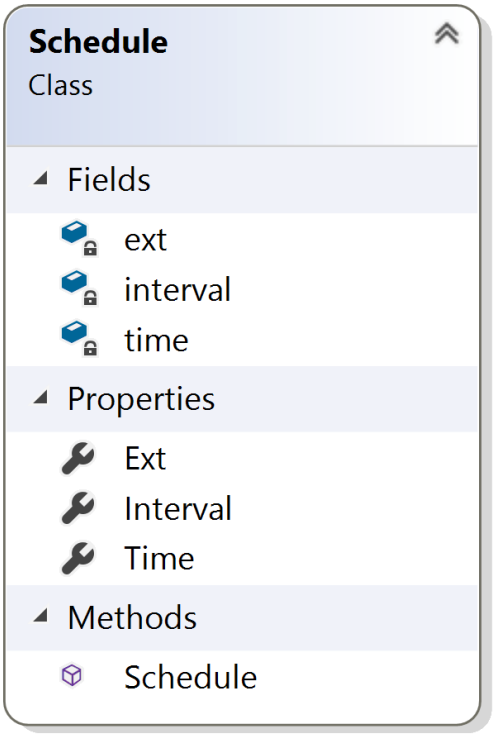
```
ConfigManager configManager = new ConfigManager();
configManager.ProcessConfigs();

for(int i = 0; i < configManager.Count -1; i++)
{
    Config config = configManager[i];
    var ext = config.Ext;
    ....
}
```

> 若你使用的程式語言，沒有 indexer 機制，可將 `ProcessConfigs()` 回傳 container (list、array...)，然後 for 迴圈處理。

Schedule Class

`Schedule` 主要功能是封裝 `設定自動排程時間` 的 JSON 成為物件，因此沒有 `行為`，有的只是 read only 的 property。

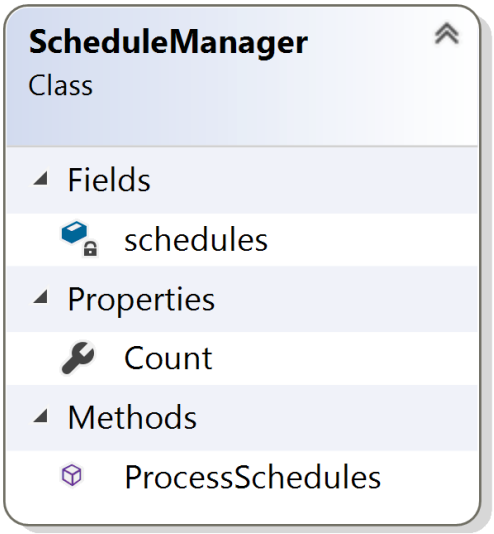


- `Schedule` 的 constructor 接受使用端的初始化資料，稍後會由 `ScheduleManager` 建立
- `Schedule` 的 property 都是 `read only` ，因為 `Schedule` 為 `schedule.json` 的封裝，所有資料應該來自於 `schedule.json` ，而不該給使用端任意修改，是個純粹的 value object
- constructor 的 signature 如下：

```
1 public Schedule(string ext, string time, string interval);
```

ScheduleManager Class

`ScheduleManager` 主要功能是將 `schedule.json` 轉成 `Schedule` 物件。



Fields

- `configs`：提供 `List<Config>` 存放多筆 `Schedule` 物件

Properties

- `Count`：提供 `List<Config>` 的 `筆數`

Methods

- `ProcessSchedules()`：將 `schedule.json` 轉成 `List<Config>`

Others

- 提供類似 array 方式 (indexer) 讀取 `List<Config>`

> 若你使用的程式語言沒有 indexer 機制，可不用實作

Example

```
ScheduleManager scheduleManager = new ScheduleManager();
scheduleManager.ProcessSchedules();

for(int i = 0; i < scheduleManager.Count -1; i++)
{
    Schedule schedule = scheduleManager[i];
    var ext = schedule.Ext;
    ....
}
```

> 若你使用的程式語言，沒有 indexer 機制，可將 `ProcessConfigs()` 回傳 container (list、array...)，然後 for 迴圈處理。

Conclusion

- 程式語言不限，請依照 class diagram 將 `Config`、`ConfigManager`、`Schedule` 與 `ScheduleManager` 4 個 class 實作出來。