Hes·so VALAIS WALLIS
Haute Ecole de Gestion & Tourisme
Hochschule für Wirtschaft & Tourismus

Filière Informatique de gestion
Studiengang Wirtschaftsinformatik

634-1 / COMPONENTS AND PATTERNS

Patterns project

VSFly patterns implementation

Students: Brice Berclaz, Coline Fardel

Class: 604 F

Teacher: Prof. Dr. Michael Ignaz Schumacher

Hand Back: 03.06.2020

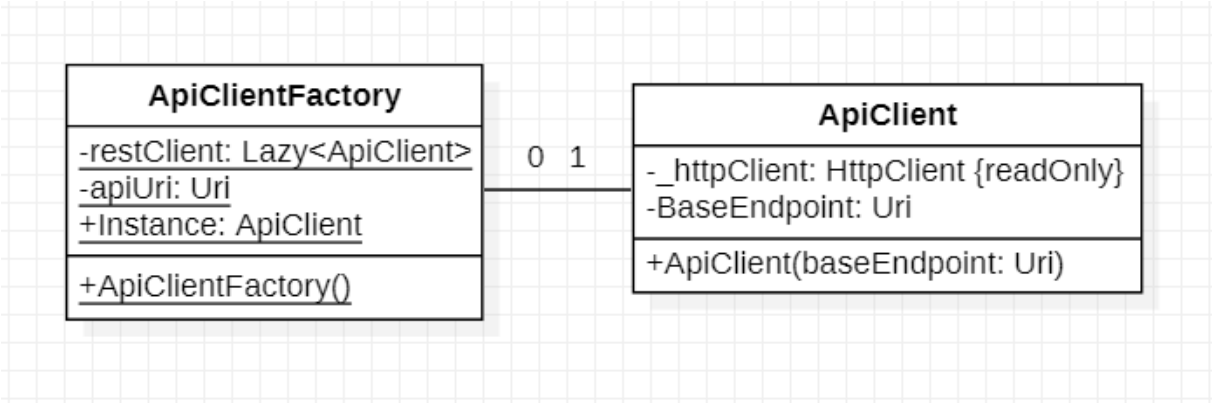
TABLE OF CONTENTS

1. FACTORY METHOD

1.1 Why

A factory method handles object creation and encapsulates it in a subclass. This decouples the client code in the superclass from the object creation code in the subclass.

1.2 UML



1.3 Code

The factory pattern is used for creating new objects. We can use this to determine which object gets returned. Here we can see that the object returned is an ApiClient.

1.3.1 Internal static class ApiClientFactory

```
internal static class ApiClientFactory
{
    private static Uri apiUri;

    private static Lazy<ApiClient> restClient = new Lazy<ApiClient>(
      () => new ApiClient(apiUri),
      LazyThreadSafetyMode.ExecutionAndPublication);

    static ApiClientFactory()
    {
        apiUri = new Uri(ApplicationSettings.WebApiUrl);
    }

    public static ApiClient Instance
    {
        get
        {
            return restClient.Value;
        }
    }
}
```

1.3.2 Partial class ApiClient

```
6 références | Bearbrice, Il y a 15 heures | 1 auteur, 2 modifications
public partial class ApiClient
{
    private readonly HttpClient _httpClient;
    2 références | Bearbrice, il y a 5 jours | 1 auteur, 1 modification
    private Uri BaseEndpoint { get; set; }

    1 référence | Bearbrice, il y a 5 jours | 1 auteur, 1 modification
    public ApiClient(Uri baseEndpoint)
    {
        if (baseEndpoint == null)
        {
            throw new ArgumentNullException("baseEndpoint");
        }
        BaseEndpoint = baseEndpoint;
        _httpClient = new HttpClient();
    }
```

In the next line we call the factory object. We never call ApiClient in our project for business, it will always be managed by the ApiClientFactory class.

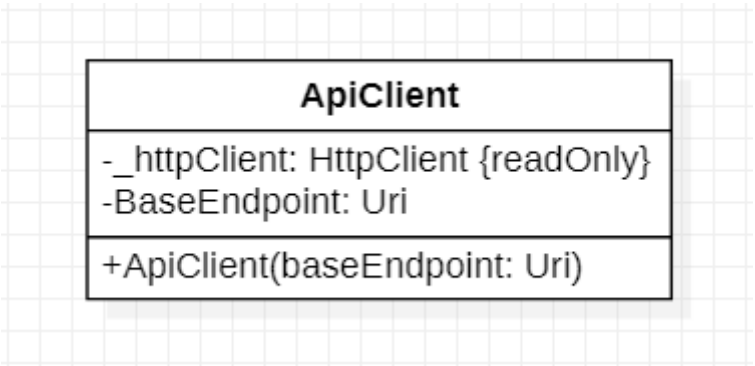
```
var flights = await ApiClientFactory.Instance.GetAllFlights();
```

2. SINGLETON

2.1 Why

The singleton pattern can prevent more than one object from being instantiated. We used it for the class ApiClient which manages the client.

2.2 UML



2.3 Code

The ***readonly*** modifier prevents the field from being replaced by a different instance of the reference type.

2.3.1 Class ApiClient

```
public partial class ApiClient
{
    private readonly HttpClient _httpClient;
```

2.3.2 Class ApiClientFactory

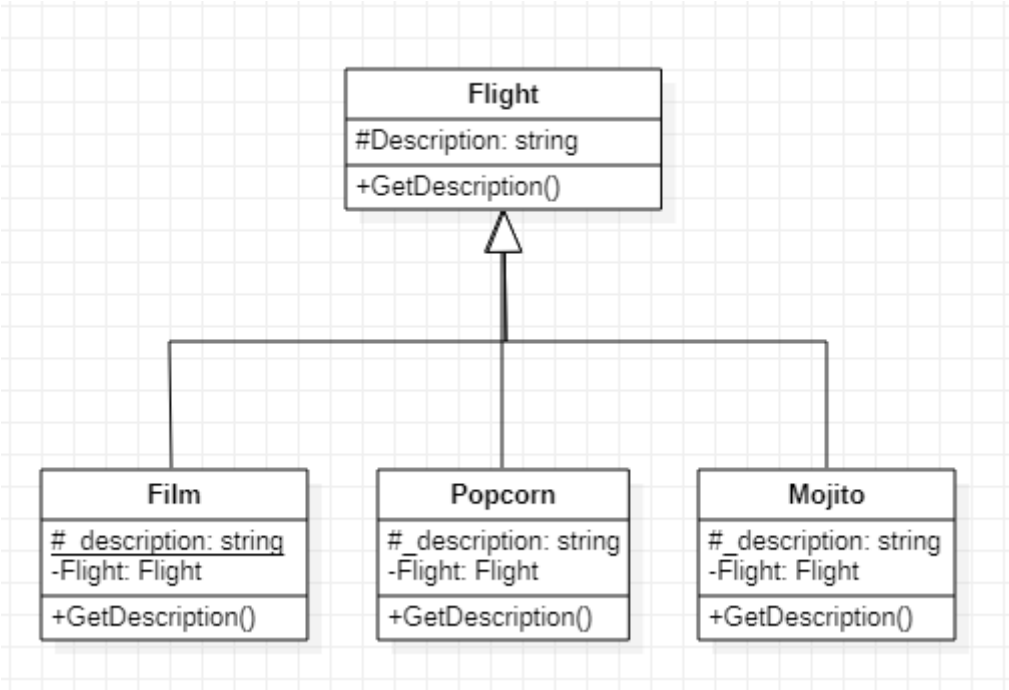
```
internal static class ApiClientFactory
{
    private static Uri apiUri;

    private static Lazy<ApiClient> restClient = new Lazy<ApiClient>(
      () => new ApiClient(apiUri),
      LazyThreadSafetyMode.ExecutionAndPublication);

    static ApiClientFactory()
    {
        apiUri = new Uri(ApplicationSettings.WebApiUrl);
    }

    public static ApiClient Instance
    {
        get
        {
            return restClient.Value;
        }
    }
}
```

3. DECORATOR

3.1 Why

We decided to use this pattern because it is a good way to add some extra features to an object. Any passenger can now add popcorns, films or mojitos to their flight.

3.2 UML



3.3 Code

3.3.1 Concrete component: class Flight

The component is what you want to decor.

```
public class Flight
{
    protected string Description = "You have successfully booked a flight ";

    public virtual string GetDescription()
    {
        return Description;
    }

}
```

3.3.2 Concrete decorators

The concrete decorators are decorating the concrete component. They will override the method GetDescription by returning the current Description of the Flight object created plus the new attribute _description of the concrete decorators.

3.3.2.1 Class Popcorn

```
public class Popcorn : Flight
{
    protected static string _description = "with popcorn ";

    private Flight Flight = null;

    public Popcorn(Flight flight)
    {
        this.Flight = flight;
    }

    public override string GetDescription()
    {
        return this.Flight.GetDescription() + _description;
    }
}
```

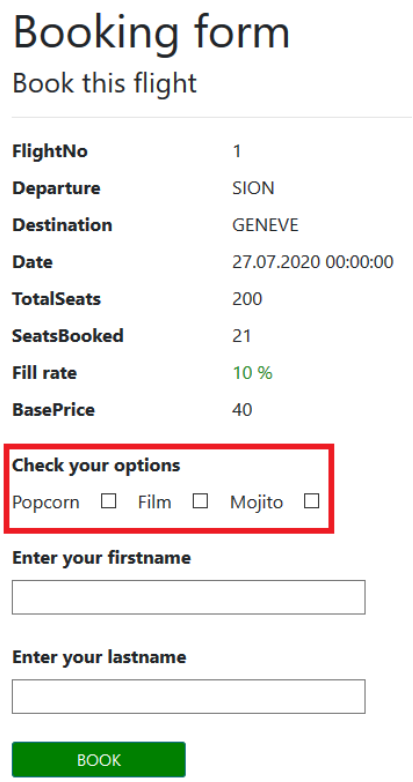
3.3.2.2 Class Film

```
public class Film : Flight
{
    protected static string _description = "with film ";

    private Flight Flight = null;

    public Film(Flight flight)
    {
        this.Flight = flight;
    }

    public override string GetDescription()
    {
        return this.Flight.GetDescription() + _description;
    }
}
```

3.3.2.3 Class Mojito

```
public class Mojito : Flight
{
    protected static string _description = "with mojito ";

    private Flight Flight = null;

    public Mojito(Flight flight)
    {
        this.Flight = flight;
    }

    public override string GetDescription()
    {
        return this.Flight.GetDescription() + _description;
    }
}
```

3.3.3 Implementation: Class FlightController



To implement it, we did some checkboxes with some Booleans attributes in the web client of the API.

We retrieve the value of the Booleans to decide how to decorate our flight. For example, in we tick the boxes popcorn and film, this will go to *"f= new Pocporn(f)"* and *"f=new Film(f)"*.

3.3.3.1 Excerpt from the FlightController class code

```
/* Pattern Decorator*/
Models.Decorator.Flight f = new Models.Decorator.Flight();

if (popcorn)
{
    f = new Popcorn(f);
}

if (film)
{
    f = new Film(f);
}

if (mojito)
{
    f = new Mojito(f);
}
```

3.3.3.2 The client can see the result of our pattern implementation