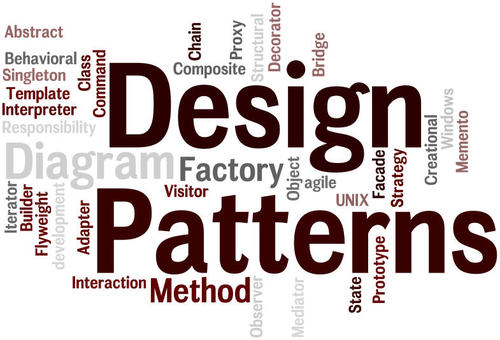


Design Patterns - VSFly

634-1



Students: May Aurélien & Favre Léonard

Class: 604\_F

Professors: Schumacher Michael Ignaz

Widmer Antoine

Russo David

Table des Matières

[Decorator 3](#_Toc42100327)

[1. Use 3](#_Toc42100328)

[2. UML 3](#_Toc42100329)

[3. Implementation 4](#_Toc42100330)

[3.1. Pattern call 4](#_Toc42100331)

[3.2. Component 4](#_Toc42100332)

[3.3. Decorator 5](#_Toc42100333)

[3.4. Display 5](#_Toc42100334)

[3.5. Result 5](#_Toc42100335)

[Singleton 6](#_Toc42100336)

[1. Use 6](#_Toc42100337)

[2. UML 6](#_Toc42100338)

[3. Implementation 6](#_Toc42100339)

[Factory 7](#_Toc42100340)

[1. Use 7](#_Toc42100341)

[2. UML 7](#_Toc42100342)

[3. Implementation 8](#_Toc42100343)

Decorator

## Use

**The Decorator Pattern** attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing.

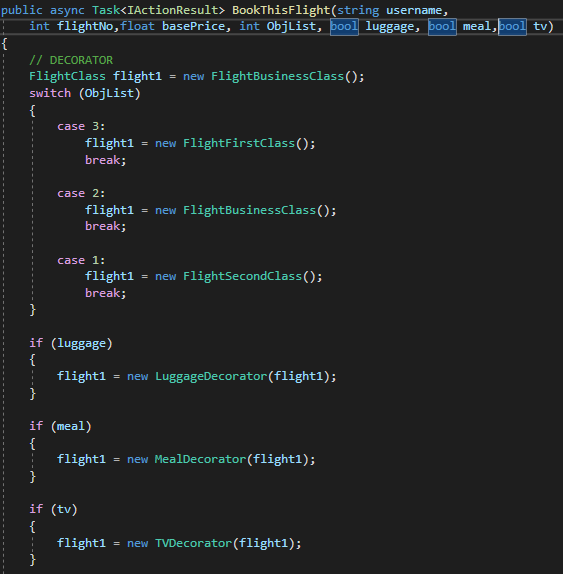
This pattern proposes a solution to modify an object in run time, applying independent features or behaviors which can be combined and accumulated in any order.

We thought the decorator was the perfect pattern for ordering a plane ticket. Because there are different classes, first class, business class and second class used in this project (components). And in addition to these classes, there is the possibility to have extras/options such as luggage, meal and TV (decorators).

We have implemented this pattern only on the client side, so information such as the chosen class (flight type), luggage, meal, TV, including the price of these extras are not stored in the server's database. The pattern is still used, however only in the display, as a console execution. The data saved in the database relates to the other part of the project (component).

## UML

## Implementation

* 1. Pattern call

It is in this part of the code that the plane ticket is instantiated, as if it were a "FlightDecoratorLauncher" (main).

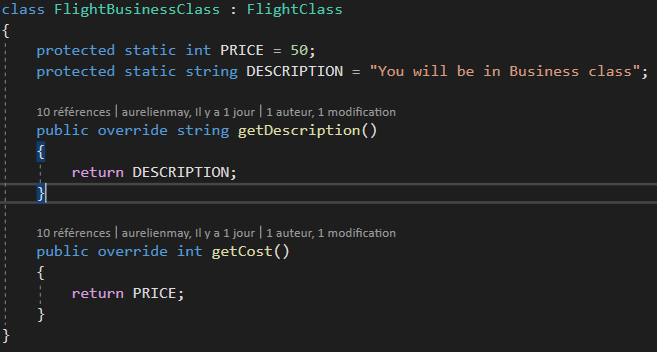
So, when the user books a flight, this method “BookThisFlight” is called in the Controller to do the reservation and contains many parameters used for the pattern.

The “int ObjList” is the class (first, second or business class) choose by the user (return the number in the list).

The “bool luggage” is the value of the checkbox (ticked or not). Same for “bool meal” and “bool tv”.

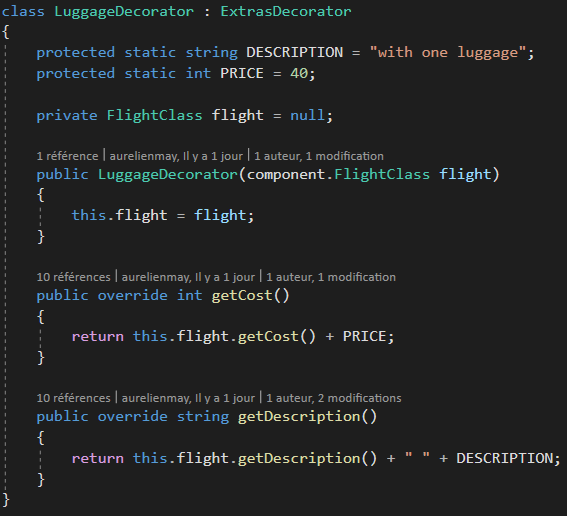
Once the method is called, there is a first switch that instantiates the concrete component (first, business or second class) according to the user's choice

Then there are the “if” for every extra chosen by the user (luggage, meal and tv).

* 1. Component

The “FlightBusinessClass” is implemented almost like the two other concrete components, “FlightFirstClass” and “FlightSecondClass”. The differences between them is the description and the price because each class has its own description and price.

All the concrete components extend the “FlightClass” class (“:” in C#) to have access to “getDescription” and “getCost” and dynamically add new behavior.

* 1. Decorator

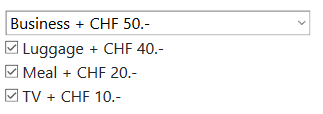
The “LuggageDecorator” is implemented almost like the two other other concrete decorators, “MealDecorator” and “TVDecorator”. The differences between them is the description and the price because each class has its own description and price.

All the concrete decorators extend the “ExtrasDecorator” class which extends the “FlightClass” (component).

The concrete decorator has an instant variable of the thing it decorates (“FlightClass”, the component the decorator wraps).

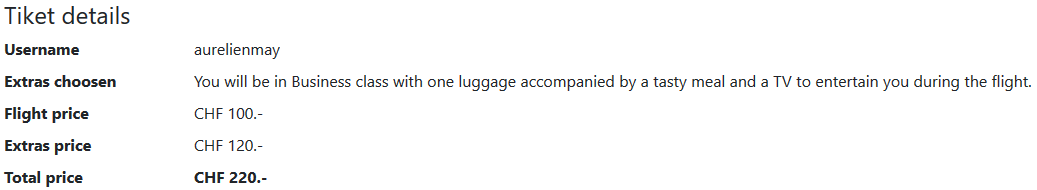
* 1. Display

Extras selected



In this case the user selected the Business class with all the extra.

* 1. Result



As you can see, the various extra have been added up, just like the description.

If the user had chosen a different class and not all the extras, the description and the "Extras price" would not have been the same.

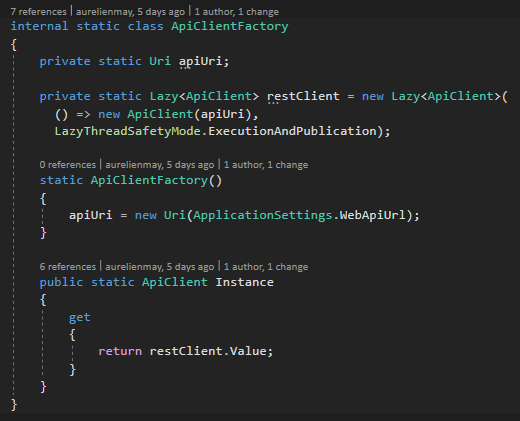
Singleton

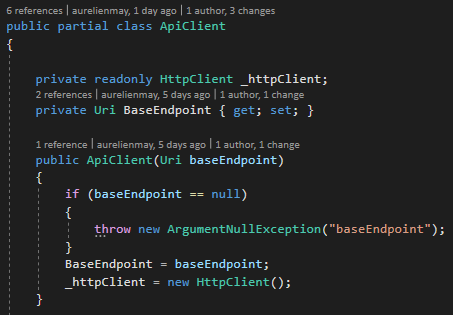
## Use

The Singleton Pattern let a class create a single instance of itself. It Prevent other classes from creating a new instance. To get an instance, you’ll be able to only use the private method from the class, if the object doesn’t exist it will instantiate the object otherwise it will give the reference of the object.

## UML

## Implementation

In C# we can use Lazy to implement the Singleton Pattern. Lazy initialization of an object means that the object creation is deferred until it first use. This is primarily used for optimization purposes. In .Net 4 and above we can use the Lazy class. Lazy is a thread safe, means that you don’t need to worry about multiple threads creating multiple times the same object. You can verify that the instance has been created with the method “[IsValueCreated](http://msdn.microsoft.com/en-us/library/dd642334.aspx" \t "_blank)”.

In the api client class, this is where the Singleton is created.

Factory

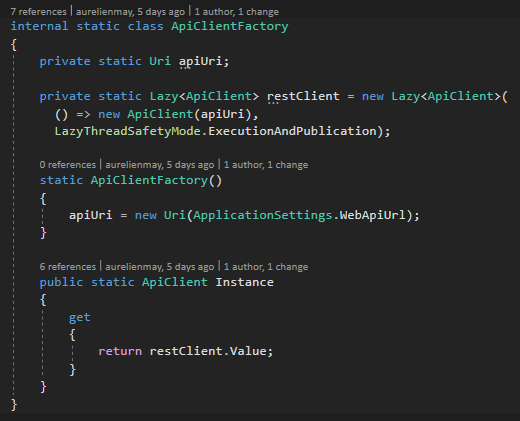
## Use

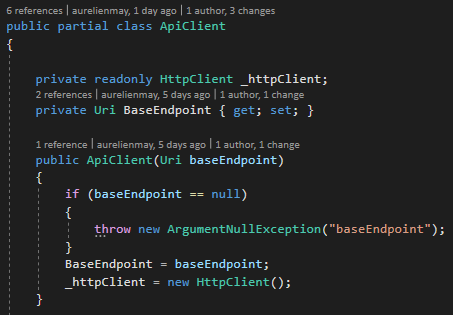
The Factory design pattern provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created.

## UML

## Implementation

ApiClientFactory



ApiClient

ApiClientClientFactory call



In the controller we get access to the factory. The lazy class will manage the instance.