”Design Patterns in C#”

Lab Manual

Wincubate ApS

01-10-2023

V1.1-PROSA

Table of Contents

Exercise types

The exercises in the present lab manual differs in type and difficulty. Most exercises can be solved by applying the techniques from the presentations in the slides in a more or less direct manner. Such exercises are not categorized further.

However, the remaining exercises differs slightly in the sense that they are not necessarily easily solvable. These are categorized as follows:

Labs marked with a single star denote that the corresponding exercises are a bit more loosely specified.

Labs marked with two stars denote that the corresponding exercises contain only a few hints (or none at all!) or might be a bit more difficult or nonessential. They might even require additional searches for information elsewhere than in the slide presentations.

Labs marked with three stars denote that the corresponding exercises are not expected in any way to be solved. These are difficult, tricky, or mind-bending exercises for the interested participants – mostly for fun! ☺

Prerequisites

The present labs require the course files accompanying the course to be extracted in some directory path, e.g.

```
C:\Wincubate\DesignPatternsInCS
```

with Visual Studio 2022 and .NET 6 installed on the PC.

We will henceforth refer to the chosen installation path containing the lab files as *PathToCourseFiles* .

Day 1 – Module 02: "Abstract Factory"

Lab 02.1: "Tasty Factories and Products"

This exercise implements all aspects of the Abstract Factory Pattern in an example involving foreign cuisines. The overall structure of the solution will proceed in a manner similar to examples in the module presentation.

- Open the starter project in
  PathToCourseFiles`\Labs\02 – Abstract Factory\Lab 02.1\Starter` ,
  which contains a project called `Cuisines`.

Here you fill in all the additional code needed for implementing Abstract Factory.

Throughout this exercise a "foreign cuisine" (such as Italian or Indian) is an abstract factory interface letting the client create

1. A main course (e.g. pizza)
2. A dessert (e.g. tiramisu)

Consequently, there are two kinds of abstract products in the cuisine abstract factory: MainCourse objects and Dessert objects. These are already defined in the existing projects via the following two definitions:

```
interface IMainCourse
{
    void Consume();
}

interface IDessert
{
    void Enjoy();
}
```

Main courses should have a `void Consume()` method. The intention here is that concrete products should print to the console what is being consumed by the client.

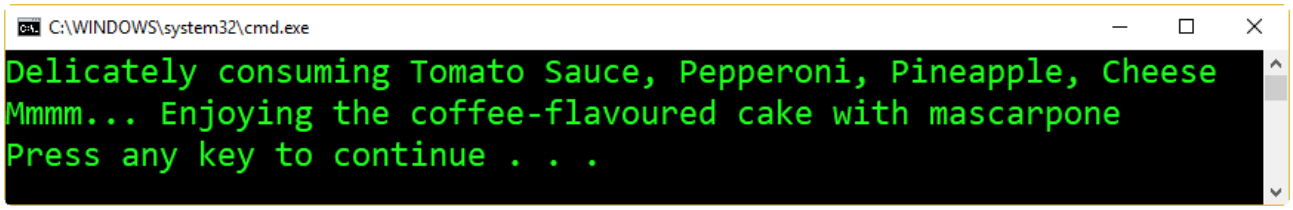
Desserts should have a `void Enjoy()` method. When invoked it should print to the console reflect what is being enjoyed by the client.

You will start by implementing an Italian cuisine using the Abstract Factory Pattern

- Implement a concrete main course product called `Pizza`
  - Its constructor should accept a sequence of topping strings.
- Implement a concrete dessert product called `Tiramisu` (without additional members)
- Create the appropriate abstract factory interface for cuisines called `IMealFactory`.
- Create a concrete factory class for the Italian cuisine, where
  - the main course being created is a pizza with "Tomato Sauce", "Pepperoni", "Pineapple", and "Cheese"

    - the dessert is a tiramisu,
- Test your implementation by adding the appropriate client code in Program.cs.
    - Invoke `IMainCourse.Consume()` on the created main course object.
    - Invoke `IDessert.Enjoy()` on the created dessert object.

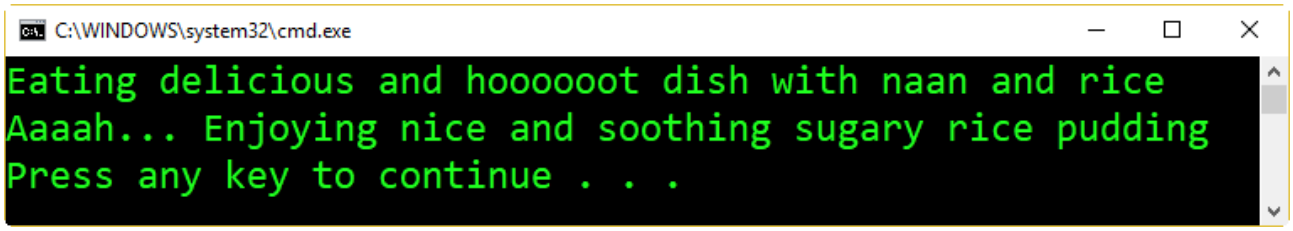
When you run the program, the output should be the following (or equivalent):



You have now implemented the Italian cuisine. You will then proceed to implementing the Indian cuisine as follows.

- Implement a concrete main course product called `ChickenCurry`
    - Its constructor should accept an integer indicating spicyness.
- Implement a concrete dessert product called `Kheer` (without additional members)
- Create the corresponding concrete factory class for the Indian cuisine, where
    - the main course being created is with a spicyness of 5.
- Test your implementation by changing only the Italian cuisine to the Indian cuisine in Program.cs.

When you run the program, the output should now be the following (or equivalent):



Day 1 – Module 03: "Builder"

Lab 03.1: "Creating Very Simple Fluent APIs" (★)

This exercise illustrates how to create a Fluent API for building pizza products using a variation of the Builder Pattern. Fluent APIs are quite popular in .NET for configuring the Builder instances in a "fluent" fashion, which is reminiscent of the flow in natural, spoken languages.

Consider the `Pizza` class defined as

```
class Pizza
{
    public CrustKind Crust { get; set; }
    public bool HasSauce { get; set; }
    public IEnumerable<ToppingKind> Toppings { get; set; }
    public CheeseKind? Cheese { get; set; }
    public bool Oregano { get; set; }
}
```

Then the well-known Hawaii pizza manually constructed in the following manner:

```
Pizza hawaii = new Pizza
{
    Crust = CrustKind.Classic,
    HasSauce = true,
    Cheese = CheeseKind.Regular,
    Toppings = new List<ToppingKind>
    {
        ToppingKind.Ham,
        ToppingKind.Pineapple
    },
    Oregano = true
};
```

could be built using an appropriate fluent API Builder as follows:

```
FluentPizzaBuilder builder = new FluentPizzaBuilder();
Pizza hawaii = builder
    .Begin()
    .WithCrust(CrustKind.Classic)
    .Sauce
    .AddCheese()
    .AddTopping(ToppingKind.Ham)
    .AddTopping(ToppingKind.Pineapple)
    .Oregano
    .Build();
```

Your task is now to create this `FluentPizzaBuilder` class.

- Open the starter project in
  PathToCourseFiles`\Labs\03 - Builder\Lab 03.1\Starter` ,

which contains a project with the `Pizza` class and related types.

- Create the `FluentPizzaBuilder` class.
- Test that your class in the Fluent API definition correctly build a Hawaii pizza instance equivalent to the manually created instance above.

Lab 03.2: "A Much Better Fluent API" ()

This exercise examines how to create a better Fluent API for building pizza products.

The Fluent API solution to Lab 03.1 is simple and not too difficult to create with a little bit of practice. But it is too simplistic for professional purposes due to a number of problems:

1. Any order of invoking the fluent methods is allowed
2. Repetitions of the fluent methods are allowed
3. All methods are essentially optional (as well as repeatable)
4. It uses properties containing getters with side effects.

Your task is now to remedy all these deficiencies.

- Open the starter project in
  PathToCourseFiles`\Labs\03 – Builder\Lab 03.2\Starter` ,
  which contains a project with the solution to Lab 03.1.
- You should create a better fluent API solution, which is statically safe in the sense that the compiler will only allow fluent method sequences which are legal.

More specifically, this sequence should be allowed by the compiler:

```
Pizza hawaii = new FluentPizzaBuilder()
    .Begin()
    .WithCrust(CrustKind.Classic)
    .WithSauce()
    .AddCheese()
    .AddTopping(ToppingKind.Ham)
    .AddTopping(ToppingKind.Pineapple)
    .WithOregano()
    .Build();
```

This sequence should also be allowed by the compiler:

```
Pizza hawaii = new FluentPizzaBuilder()
    .Begin()
    .WithCrust()
    .WithoutSauce()
    .AddCheese()
    .AddTopping(ToppingKind.Ham)
    .AddTopping(ToppingKind.Pineapple)
    .Build();
```

However, this sequence should **not** be allowed by the compiler:

```
Pizza hawaii = new FluentPizzaBuilder()
    .Begin()
    .WithOregano()
    .WithCrust(CrustKind.Classic)
    .WithSauce()
    .AddCheese()
```

```
        .AddTopping(ToppingKind.Ham)
        .AddTopping(ToppingKind.Pineapple)
        .Build();
```

Make sure that:

- Building always begins with `Begin()`
- Building always completes with `Build()`
- Proper defaults are chosen, e.g for `WithCrust()`
- Some methods are optional choices, e.g. `WithOregano()`
- Some choices are mandatory, e.g. `WithSauce()` vs. `WithoutSauce()`
- The compiler allows only correct sequences, which are in the “usual” order:
    - Crust,
    - Sauce,
    - Cheese,
    - Any sequence of toppings, and finally
    - Oregano

Day 2 – Module 07: "Adapter"

Lab 07.1: "Adapting to a Simple Web Shop"

This exercise implements an Adapter for a pre-specified API definition for a very simple web shop.

- Open the starter project in
  PathToCourseFiles`\Labs\07 - Adapter\Lab 07.1\Starter` ,
  which contains a project with the `InventoryClient` class.

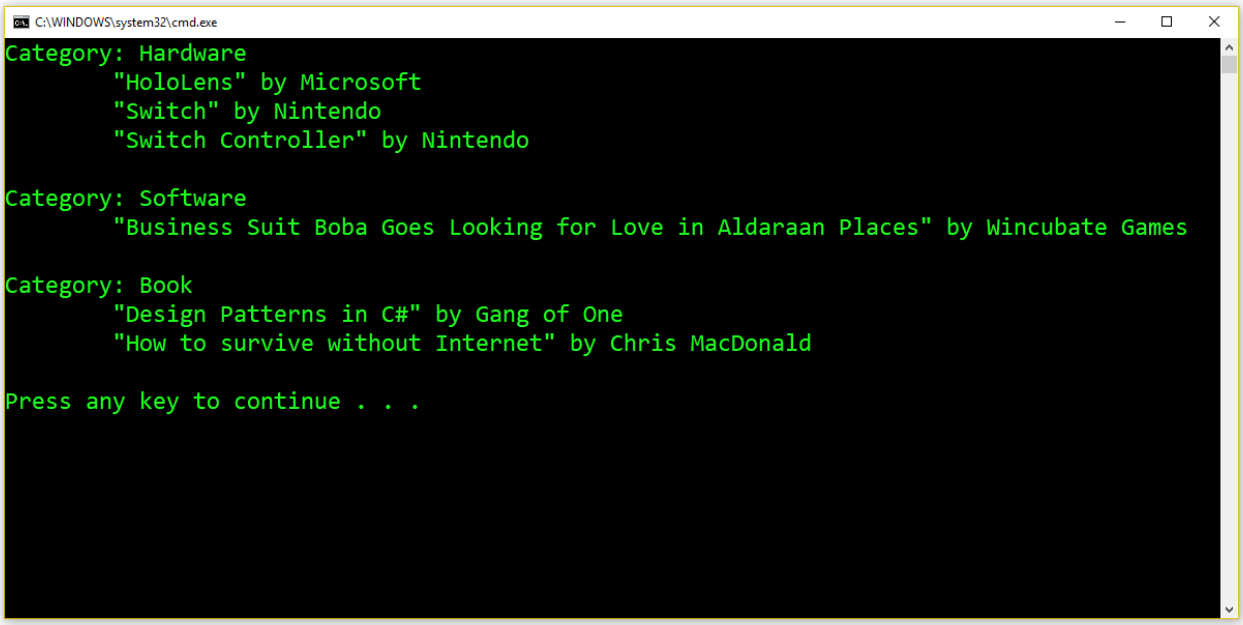
The `InventoryClient` class makes use of an `IInventoryRepository` instance in order to retrieve and display inventory information from a web shop back-end. Unfortunately, the web shop back-end is based on a commercial-off-the-shelves product with an API that cannot be changed.

The back-end supplies inventory information in the shape of a `ProductRepository` class which must be instantiated. This is the only way to retrieve information about the current line of products, unfortunately.

- You need to adapt the `ProductRepository` back-end to the `InventoryClient` front-end use without changing either class.
- Modify the code in Program.cs appropriately;

```
InventoryClient client = new InventoryClient( ... );
client.DisplayInventory();
```

When you run the program, the output should be the following:



Day 2 – Module 08: "Composite"

Lab 08.1: "Wedding Gift Sharing with Composite Pattern" ( )

This exercise uses the Composite Pattern for group-based sharing of wedding gift expenses.

- Open the starter project in
  PathToCourseFiles\Labs\08 – Composite\Lab 08.1\Starter ,
  which contains a project with the `Person` class.

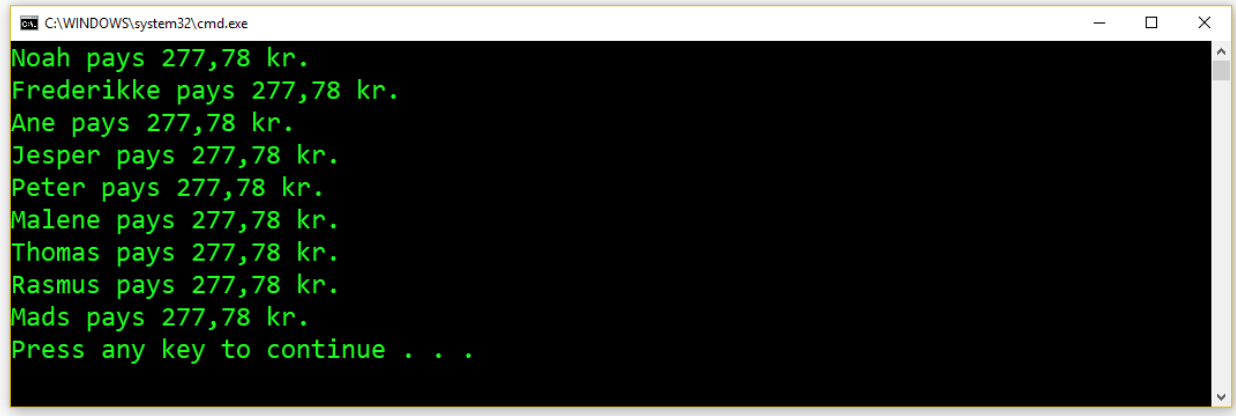
The `Person` class contains information about what the specified person must pay to contribute to the wedding gift. It is specified as follows:

```
class Person
{
    public string Name { get; set; }
    public decimal MustPay { get; set; }

    public override string ToString() =>
        $"{Name} pays {MustPay:c}";

    public Person( string name ) => Name = name;
}
```

Program.cs defines 9 persons and contains a brutally simple sharing algorithm for sharing the wedding gift expenses: The expenses are shared equally among all contributing persons!

```
List<Person> participants = new List<Person>
{
    noah,
    frederikke,
    ane,
    jesper,

    peter,
    malene,

    thomas,
    rasmus,
    mads
};

decimal giftPrice = 2500;

// Equal sharing among all participants
foreach (Person person in participants)
{
    person.MustPay = giftPrice / participants.Count;
}
```

For a gift of DKK 2.500 the algorithm produces the following output:



- Use the Composite Pattern to modify the algorithm such all families can take part in sharing the expenses in such a way that they together contribute as a single individual.
    - Add the appropriate interfaces and class to implement Composite
    - Change Program.cs accordingly to test your implementation.
- Specifically, when splitting the 9 pre-existing persons into 2 groups and 3 individuals as follows:
    - Group 1 consists of Noah, Frederikke, Ane, and Jesper
    - Group 2 consists of Peter and Malene
    - Thomas participates as an individual
    - Rasmus participates as an individual
    - Mads participates as an individual,

  the results of the modified sharing algorithm should be: