

Lab 1: Refactoring

Code Refactoring is a process of restructuring existing computer code without changing its external behavior. Refactoring is intended to improve the design, structure, and/or implementation of the software (its non-functional attributes), while preserving its functionality.

There is badly written class...

It is a class responsible for calculating a discount for the customer while he is buying some product in online shop:

```
public class Class1
{
    public decimal Calculate(decimal amount, int type, int years)
    {
        decimal result = 0;
        decimal disc = (years > 5) ? (decimal)5/100 : (decimal)years/100;
        if (type == 1)
        {
            result = amount;
        }
        else if (type == 2)
        {
            result = (amount - (0.1m * amount)) - disc * (amount - (0.1m * amount));
        }
        else if (type == 3)
        {
            result = (0.7m * amount) - disc * (0.7m * amount);
        }
        else if (type == 4)
        {
            result = (amount - (0.5m * amount)) - disc * (amount - (0.5m * amount));
        }
        return result;
    }
}
```

Tasks to Do

Rewrite the code (VS project **Lab_Refactoring** is included) according to 'Refactoring' rules:

1. **Naming** - only changing names of method, parameters and variables we may exactly know what a class is responsible for.

Example 1: The names of class and method in the code

```
public class Class1
{
    public decimal Calculate(decimal amount, int type, int years)
```

maybe renamed to:

```
public class DiscountManager
{
    public decimal ApplyDiscount(decimal price, int accountStatus,
        int timeOfHavingAccountInYears)
```

Example 2: The variable `result` maybe renamed to `priceAfterDiscount`

Example 3: Using of full names is much more understandable for nontrivial code

```
decimal disc = (years > 5) ? (decimal)5/100 : (decimal)years/100;
```

maybe changed to

```
decimal discountForLoyaltyInPercentage = (timeOfHavingAccountInYears > 5) ?
(decimal)5/100 : (decimal)timeOfHavingAccountInYears/100;
```

2. **‘Magic’ numbers** - One of techniques of avoiding magic numbers is replacing it by **enums**. Something like that:

```
public enum AccountStatus
{
    NotRegistered = 1,
    SimpleCustomer = 2,
    ValuableCustomer = 3,
    MostValuableCustomer = 4
}
```

3. **Make code more readable** - improve readability of our class by replacing **if-else if** statement with switch-case statement. Also, it is preferable to divide long one-line statement into two separate statements.

For example, the operator

```
priceAfterDiscount = (price - (0.5m * price)) -
    (discountForLoyaltyInPercentage * (price - (0.5m * price)));
```

may be replaced by:

```
priceAfterDiscount = (price - (0.5m * price));
priceAfterDiscount = priceAfterDiscount - (discountForLoyaltyInPercentage
    * priceAfterDiscount);
```

4. **Not obvious bug** - Imagine that there is a new customer account status added to our system - **GoldenCustomer**. Now our method will return 0 as a final price for every product which will be bought from a new kind of an account. Why? Because if none of our **if-else if** conditions will be satisfied (there will be unhandled account status) method will return 0. Obviously, this result is not correct.

How can you fix it? By throwing **NotImplementedException**.

5. **Analyse calculations** - In our example we have two criteria of giving a discount for our customers:

- a) Account status
- b) Time in years of having an account in our system

All algorithms look similar in case of discount for time of being a customer:

```
(discountForLoyaltyInPercentage * priceAfterDiscount)
```

but there is one exception in case of calculating constant discount for account status:

```
0.7m * price
```

so, it may be changed to look the same as in other cases:

```
price - (0.3m * price)
```

6. **Get rid of ‘magic’ numbers – another technique** - look at static variable which is a part of discount algorithm for account status: (static_discount_in_percentages/100)

Concrete instances are:

0.1m

0.3m

0.5m

These numbers are ‘magic’ as well – they are not telling anything about themselves. There is the same situation in case of converting “time in years of having an account” to discount a “discount for loyalty”. Number “5” is making our code mysterious enough.

It is strongly recommended to create one static class for constants to have them in one place of the application. For example:

```
public static class Constants
{
    public const int MAXIMUM_DISCOUNT_FOR_LOYALTY = 5;
    public const decimal DISCOUNT_FOR_SIMPLE_CUSTOMERS = 0.1m;
    public const decimal DISCOUNT_FOR_VALUABLE_CUSTOMERS = 0.3m;
    public const decimal DISCOUNT_FOR_MOST_VALUABLE_CUSTOMERS = 0.5m;
}
```

7. **Don't repeat code** - Instead of this “extract” possible methods to be recall several times. For example, in the following way:

```
public static class PriceExtensions
{
    public static decimal ApplyDiscountForAccountStatus(this decimal price,
        decimal discountSize)
    {
        ...
    }
    public static decimal ApplyDiscountForTimeOfHavingAccount(this decimal
        price, int timeOfHavingAccountInYears)
    {
        ...
    }
}
```

8. **Remove few unnecessary lines** - Notice that there is the same method call for 3 kinds of customer account:

ApplyDiscountForTimeOfHavingAccount(timeOfHavingAccountInYears). Do it once!

Note: There is exception for **NotRegistered** users because discount for years of being a registered customer does not make any sense for unregistered customer. True, but what time of having account has unregistered user? 0 years. Discount in this case will always be 0, so it's possible safely add this discount also for unregistered users.

9. **Run the final solution and present it to the teacher.**

Appendix: Example of code refactoring

```
public enum AccountStatus
{
    NotRegistered = 1,
    ...
}
public static class Constants
{
    public const int MAXIMUM_DISCOUNT_FOR_LOYALTY = 5;
    public const decimal DISCOUNT_FOR_SIMPLE_CUSTOMERS = 0.1m;
    ...
}
public static class PriceExtensions
{
    public static decimal ApplyDiscountForAccountStatus(this decimal price, decimal discountSize)
    {
        ...
    }

    public static decimal ApplyDiscountForTimeOfHavingAccount(this decimal price, int timeOfHavingAccountInYears)
    {
        ...
    }
}

public class DiscountManager
{
    public decimal ApplyDiscount(decimal price, AccountStatus accountStatus, int timeOfHavingAccountInYears)
    {
        ...
    }
}

class Program
{
    static void Main(string[] args)
    {
        ...
        Console.WriteLine("Price after discount: " + priceAfterDiscount);
    }
}
```