**COURSEWORK SUBMISSION FORM**

|  |  |  |  |
| --- | --- | --- | --- |
| **STUDENT USE** | | **STAFF USE** | |
| Module Name | Object Oriented Programming | First Marker’s  (acts as signature) |  |
| Module Code | 5COSC018C | Second Marker’s  (acts as signature) |  |
| Lecturer Name | Nargiza Akramova | Agreed Mark |  |
| UoW Student IDs |  | **For Registrar’s office use only (hard copy submission)** | |
| WIUT Student IDs | 00010881 |
| Deadline date | December 24, 2020 |
| Assignment Type | 🗌Group☑Individual |

**SUBMISSION INSTRUCTIONS**

**COURSEWORKS *must* be submitted in *both* HARD COPY (to the Registrar’s Office) *and* ELECTRONIC unless instructed otherwise.**

For hardcopy submission instructions refer to: [http://intranet.wiut.uz/Shared%20Documents/Forms/AllItems.aspx](http://charlotte.wiut.uz:36765/Shared%20Documents/Forms/AllItems.aspx) - Coursework hard copy submission instructions.doc

For online submission instructions refer to: [http://intranet.wiut.uz/Shared%20Documents/Forms/AllItems.aspx](http://charlotte.wiut.uz:36765/Shared%20Documents/Forms/AllItems.aspx) - Coursework online submission instructions.doc

|  |
| --- |
| **MARKERS FEEDBACK (Continued on the next page)** |
|  |

Оглавление

[Introduction 3](#_Toc59678882)

[Main Section 3](#_Toc59678883)

[Case 1 3](#_Toc59678884)

[Description of the scenario 3](#_Toc59678885)

[Naive approach code 3](#_Toc59678886)

[Builder Design Pattern 4](#_Toc59678887)

[UML Diagram 5](#_Toc59678888)

[Case 2 6](#_Toc59678889)

[Description of the scenario 6](#_Toc59678890)

[Naïve Approach Code 6](#_Toc59678891)

[Adapter Design Pattern 7](#_Toc59678892)

[UML Diagram 8](#_Toc59678893)

[Case 3 8](#_Toc59678894)

[Description of the scenario 8](#_Toc59678895)

[Naïve Approach Code 8](#_Toc59678896)

[Observer Design Pattern 9](#_Toc59678897)

[UML Diagram 10](#_Toc59678898)

[Conclusion 10](#_Toc59678899)

[References 11](#_Toc59678900)

# Introduction

As a business domain, local “HamkorBank” bank has been chosen. This bank provides common services to its customers just like other banks, like giving loan, sending and receiving money from abroad, making deposits. Besides that, it provides an online services as well from its website as well as from its recently developed mobile applications. However, its services might vary depending on its customers. For example, citizens of this country and non-citizens will pay different amount of commissions for transferring money. Bank has defined its customers into several groups. They are individuals, legal entities, students and non-citizens. To be more exact, when it comes to get a loan, it will depend on the group from which customer comes from. If he or she is from students group, then interest-rate will be lower than others, but amount of the loan also will be less than others. If a non-citizen wants to get a loan, the length will not be more than 3 months and interest rate will be higher than average. In some other cases like making deposit, using online services via mobile application or website, currency exchange and consulting will be the same for all groups.

# Main Section

# Case 1

## Description of the scenario

Banks give loans to people depending on many different aspects like for what purpose this loan is being taken, how much, for how long, by whom and a lot others. Hence, for each person, a Bank has to come up with individual loan type depending on this person’s desires and other aspects mentioned above. For instance, an individual is taking loan to buy some product and he needs only 3 million uzbek sums for 6 months, and there is a legal entity who wants to get a loan of tens or even hundreds of million uzbek sums for 5, 10 or even more period of time. Due to these factors length, interest rate, type of a loan (differential, annuity or others) will be different for each loaner. In order to save time and make this process much easier, there should be created an application, which will differentiate all loaners and make a loan offer for each of them based on their wishes.

## Naive approach code

public class Loan

{

//properties

//constructor if needed

}

class Program

{

static void Main(string[] args)

{

//here we call one of the methods created below depending what type of loaner we need

SetStudentLoanProperties(); // this method isn’t created below, but could be

Console.ReadLine();

}

public static void SetIndividualLoanerProperties()

{

//loaner instantiation

// set properties depending on loaner and his desires

//like interest rate, length and others

}

public static void SetLegalEntityLoanerProperties()

{

//loaner instantiation

// set properties depending on loaner and his desires

//like interest rate, length and others

}

//Below here can be a lot other methods created depending on loaners

//And that will a lot of work, like A LOT

}

In this naïve code, there are too many duplications, which violates the SOLID principles. Which also makes the code not reusable and maintainable. If a developer wants to add new groups of customers, he would need to change the existing code and add new methods.

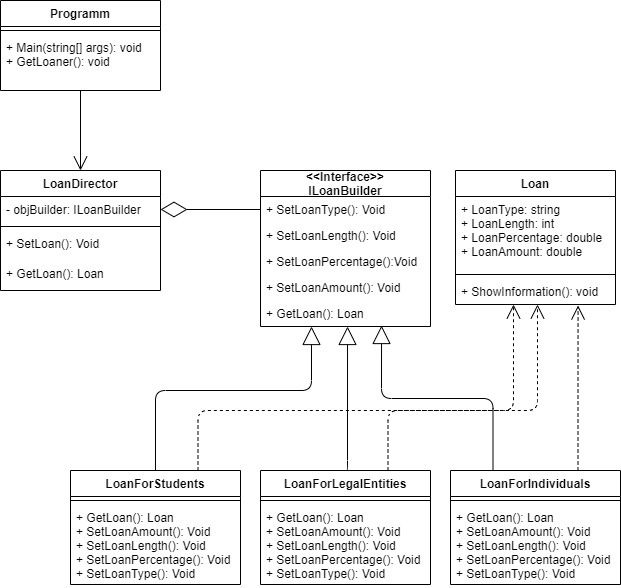
## Builder Design Pattern

One of the advantages of this design pattern is that it can build a product individually for anyone. What is meant here is that in current situation where we have too many variations of loans where some aspects like amount of money that is being lent, length of the loan, interest rate and others. This pattern just combines all those attributes and makes a ready loan offer for any loaner. Besides that, it helps developers to make code DRY by creating a few classes for loaner types, one builder class, one interface, which contain of methods to build a loan offer depending on loaner types.

One of the obvious disadvantages if this pattern is not going to be applied is that there is no way to make the code DRY. For each loaner, even if there were a lot of same loaners but with different amount of loan or length, developer would have to create new methods with the almost same properties. Besides that, according to the single responsibility principle (part of SOLID principles), our classes should have only one reason to change or in other words, it should have only one responsibility (code-maze.com). In naïve approach code written above, there is a violation of this principle. It is not only showing the result, but also setting the properties for different objects, which in theory can be thousands.

If we compare this pattern to the naïve code written above, then we can notice that in naïve code there is too many repeating of the same code. And the main problem here is reusability and maintainability of this code, which is impossible. When it comes to the code written based on the builder design pattern, it can be reused and easy to maintain. There can be deleted loaner types if needed, and can be added new loaner types like loan for non-citizens if bank implements this type of loans.

## UML Diagram



# Case 2

## Description of the scenario

A bank now made it possible to make deposits not only in local currency, which is Uzbek soms but also in USD as well. However, these deposits will vary from each other by their interest rate, length and other attributes. Now bank has to make sure that the old version of application, which was designed to make deposits only in Uzbek soms will work with new deposit type which is meant to be made in the US dollars. In this case, a user will be able to choose which type of deposit he or she wants to make and call the needed methods. In other words, if a person comes to the bank and wishes to make a deposit, he or she will be offered two types of deposit. One in USD currency and second in local UZS. Properties like length, interest rate and the minimum amount of time of the deposit will change depending on the currency of a deposit that a person wants to make.

## Naïve Approach Code

public abstract class Deposit

{

//common properties that subclasses will inherit

public abstract void MakeDeposit();

}

public class DepositInUSD : Deposit

{

//individual properties like interest rate and length

public override void MakeDeposit()

{

//make deposit in USD

}

}

public class DepositInUZS : Deposit

{

//individual properties like interest rate and length

public override void MakeDeposit()

{

//make deposit in UZS

}

}

Class program

{

static void Main(string[] args)

{

//instantiate a needed class and call needed method to make a deposit

Console.ReadLine();

}

}

Despite the fact that current naïve code works well, there are a few drawbacks of this solution. Firstly, the problem here is the reusability of this code. For example, here a developer has to change subclasses whenever he needs to insert the missing functions into the new created subclasses. Thus he will duplicate the code in all of the new classes.

## Adapter Design Pattern

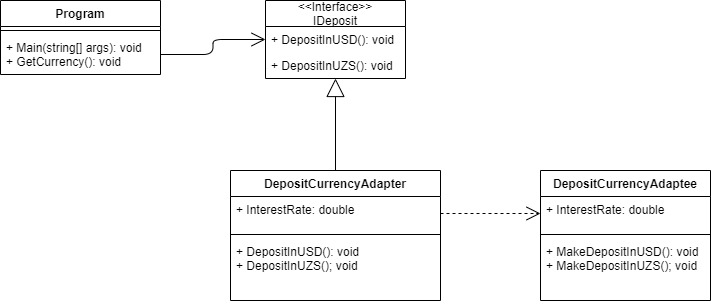
In order to solve the problem mentioned above an Adapter Design patter has been applied. One of the advantages of this design pattern is that it meets the SOLID principles like Single Responsibility and Open/Closed Principle. Which means that interface or data conversion can be separated from the primary business logic of the program. In case of Open/Closed principle a new types of adapters can be introduced into the program without breaking the existing client code while they are working with the adapters through the client interface (refactoring-guru).

When it comes to the disadvantages when current design pattern is not used, there are a few of them. Probably the main disadvantage of it is that a developer would have to break an existing code, which violates the SOLID principles. Besides that, the code will lose its reusability.

If it is compared to the naïve approach code above, then there are several advantages of using adapter design instead of naïve code. The first one is that the Adapter pattern lets us to create a middle layer class, which serves as a translator between code and legacy class, a third party class or any other class with an interface. In naïve code, developer can extend each subclass and insert the missing functions into the new subclasses. Nevertheless, he will need to duplicate the code across all of these new classes, which is not even close to keeping the code DRY.

## 

## UML Diagram



# Case 3

## Description of the scenario

Hamkor Bank has finished developing its mobile application, and it wants to notify all customers who have used the current bank’s services, which means that bank has their personal data including email and all those customers must be subscribed to the emails from bank. However, those people who already installed this mobile application will get different type of notifications related to the app like rating it in play store or app store. Developers’ job here is to write code which performs sending notifications to two types of its customers. First who do not know about this mobile application should get a notification about installing this app, second those who have installed the app should get a notification about rating the app in Play Store or App Store.

## Naïve Approach Code

public class Customer

{

//properties like firstname and last name

public bool HasApp {get; set;} /also property

// constructor

//methods

public void notify()

{

if(HasApp == true)

{

//notifies the customer to rate the app, because he already installed it

}

else

//notifies the customer about installing the app

}

}

class Program

{

static void Main(string[] args)

{

Customer customer = new Customer("adasd", "adsasd", true);

customer.Notify();

}

}

This pseudocode may cause several problems. First, it is not reusable at all. Whenever a developer wants to add new customer he or she has to change the existing code. In this case for example, if there is a new customer with some different properties, a developer would have to create a new class for that customer which will be similar to the existing class. Besides that, a new class also should be instantiated alongside with existing one all the time.

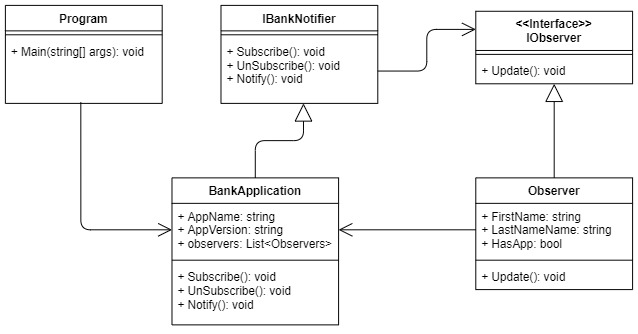
## Observer Design Pattern

To solve the problem listed above Observer design pattern has been applied. One of the advantages of this design pattern is following the SOLID principles like Open Closed principle. A new observer or subscriber class can be created even without changing the publisher’s code. With the same principle observers can be removed from changing the code in other classes. Besides that, we can establish relations between classes (objects) at runtime.

However, if current design pattern is not applied, it would violate the SOLID principles. For example, in order to add new subscriber we would have to change the code in other classes as well. For removing a subscriber it is the same.

If it is compared to the naïve code written above, the most obvious advantages of the observer design pattern is that it is reusable, maintainable and does not violate SOLID principles.

## UML Diagram



# Conclusion

Among software developers Design patterns are known as a toolkit of a tried and tested solutions, which now called templates as well. It helps developers to communicate via their code, which means that design patterns make the communication between software developers easier. For this reason, it is also called a common vocabulary. For example to solve some problems it is enough to say use Façade design pattern and everyone will understand. It is much easier that way, because instead of explaining your thoughts and drawing diagrams, you can just show such templates.

When it comes to my personal experience of using Design patterns. First, it helps to understand and use OOP principles when it is needed. Besides that, it helps to keep the code DRY and understand the importance of SOLID principles.

# 

# References

Refactoring.guru. (2014). *Builder*. [online] Available at: <https://refactoring.guru/design-patterns/builder> [Accessed 18 Dec. 2020].

‌Sep 4, M.S.| U.D. and 8, 2020 (2018). *SOLID Principles in C# - Single Responsibility Principle*. [online] Code Maze. Available at: https://code-maze.com/single-responsibility-principle/ [Accessed 19 Dec. 2020].

Refactoring.guru. (2014). *Adapter*. [online] Available at: <https://refactoring.guru/design-patterns/adapter> [Accessed 19 Dec. 2020].

Refactoring.guru. (2014). *Observer*. [online] Available at: <https://refactoring.guru/design-patterns/observer> [Accessed 20 Dec. 2020].

Refactoring.guru. (2014). *Design Patterns*. [online] Available at: https://refactoring.guru/design-patterns [Accessed 21 Dec. 2020].

‌