MINISTRY OF EDUCATION AND TRAINING

HCMC UNIVERSITY OF TECHNOLOGY AND EDUCATION

FACULTY FOR HIGH-QUALITY TRAINING

Final Project

Dietic Nutrition Application

Course: Object-oriented software design

Lecturer: Huỳnh Xuân Phụng

Group 1:

Đinh Văn Trường - 18110060

Trần Đăng Khoa - 18110024

Dương Võ Nhật Duy - 18110006

HCMC, May, 2021

SCORE

Criteria	Content	Presentation	Total
Point			

REMARKS OF TEACHERS

……

……

……

……

……

Ho Chi Minh City, May, 2021

Teacher's score

(Signature and full name)

Huynh Xuan Phung

THANK YOU

In order to successfully complete this topic and this report, we would like to extend our sincere thanks to the lecturer, Dr. Huỳnh Xuân Phụng, who directly supported us throughout the process of making the topic. We thank the teacher for giving advice from his practical experience to guide us in the right direction with the requirements of the selected topic, always answer questions and give suggestions and corrections. time to help us overcome our shortcomings and complete it well as well as on schedule.

We also would like to express our sincere thanks to the teachers in the High Quality Education Department in general and the Information Technology industry in particular for their dedicated knowledge to help us have a foundation to make. This topic has created conditions for us to learn and perform well on the topic. Along with that, we would like to thank our classmates for providing useful information and knowledge to help us improve our topic.

The topic and report are made by us in a short time, with limited knowledge and many other limitations in terms of technical and experience in implementing a software project. Therefore, in the process of creating a topic with shortcomings is inevitable, we look forward to receiving valuable comments from the teachers to make our knowledge more complete and we can do even better next time.
We sincerely thank you.

At the end, we would like to wish all of you teachers, ladies and gentlemen, always having abundant health and more success in the career of growing people.
Once again we sincerely thanks.

SUBJECT DESCRIPTION OF THE SUBJECTS OF DATABASE MANAGEMENT

Implementing student: **Đinh Văn Trường** ID's student: **18110060**

Implementing student: **Dương Võ Nhật Duy** ID's student: **18110006**

Implementing student: **Trần Đăng Khoa** ID's student: **18110024**

Field: **Information Technology**

Project: **Nutrition Application**

Instructor: **Dr. Huỳnh Xuân Phụng**

DESIGN PATTERN

- Based on our sequence, class and state diagrams, our group choose 5 patterns to implement: Builder, Command, Factory, Façade and Bridge.

1.Sign Up New Account using Builder Pattern:
Why we choose this Pattern and it characteristic:

- We use Builder Pattern for this function. Builder is a creational design pattern that lets us to construct complex objects step by step. The pattern allows us to produce different types and representations of an object using the same construction code.
- The Builder design pattern uses the Factory Builder pattern to decide which concrete class to initiate in order to build the desired type of object.

C# code:

- The solution here is we extend the base SignUp class and create a set of subclasses to cover all combinations of the parameters.

```
public Signup(string Username, string Password, string NameOfUser, string Address, string
Phone)
        {
            this.Username = Username;
            this.Password = Password;
            this.NameOfUser = NameOfUser;
            this.Address = Address;
            this.Phone = Phone;
        }
```

- After that in the base class, we will extend busclasses of it:

```
    public class SignupBuilder
        {
            private string Username;
            private string Password;
```

```
            private string NameOfUser;
            private string Address;
            private string Phone;
            public SignupBuilder(string NameOfUser)
            {
                this.NameOfUser = NameOfUser;
            }
            public SignupBuilder withUsername(string Username)
            {
                this.Username = Username;
                return this;
            }
            public SignupBuilder withPassword(string Password)
            {
                this.Password = Password;
                return this;
            }
            public SignupBuilder withAddress(string Address)
            {
                this.Address = Address;
                return this;
            }
            public SignupBuilder withPhone(string Phone)
            {
                this.Phone = Phone;
                return this;
            }
            public Signup build()
            {
                validateUserObject();
                Signup account = new Signup(this.Username, this.Password,
this.NameOfUser, this.Address, this.Phone);
                return account;
            }
            private void validateUserObject()
            {

            }
        }
```

2.Log Out of App using Command Pattern:
Why we choose this Pattern and it characteristic:

- We use Command Pattern for this function. The Command design pattern encapsulates commands (method calls) in objects allowing us to issue requests without knowing the requested operation or the requesting object. Command design pattern provides the options to queue commands, undo/redo actions and other manipulations.

- The classes participating in the pattern are:
    + Logout - declares an interface for executing an operation.
    + openNoti and closeNoti - extends the Command interface, implementing the Execute method by invoking the corresponding operations on OpenNoti and

CloseNoti.

+ NotiCommand - creates a NotiCommand object and sets its receiver.
+ OpenNoti - asks the command to carry out the request.
+ OpenNoti and CloseNoti - knows how to perform the operations.

C# code:

```
public class NotiCommand
    {
        private string Notification;
        public NotiCommand(string Notification)
        {
            this.Notification = Notification;
        }
        public void Open()
        {
            //Open Notification
        }
        public void Close()
        {
            //Close Notificatiom
        }
    }
    public interface Command
    {
        void Excute();
    }
    public class OpenNoti : Command
    {
        private NotiCommand Noti;
        public OpenNoti(NotiCommand Noti)
        {
            this.Noti = Noti;
        }
        public void Excute()
        {
            Noti.Open();
        }
    }
    public class CloseNoti : Command
    {
        private NotiCommand noti;
        public CloseNoti(NotiCommand noti)
        {
            this.noti = noti;
        }
        public void Excute()
        {
            noti.Close();
        }
    }
    public class Logout
    {

        private Command openNoti;
        private Command closeNoti;
```

```
        public Logout(Command openNoti, Command closeNoti)
        {
            this.openNoti = openNoti;
            this.closeNoti = closeNoti;
        }

        public void clickOpenNoti()
        {

            openNoti.Excute();
        }

        public string clickCloseNoti()
        {
            //print("User click close an user");

            closeNoti.Excute();
            return "You have log out of app.";
        }
    }
```

3.Add Weekly Menu using Factory, Bridge, Facade Pattern:

Why we choose these 3 Pattern and it characteristic:

- We use 3 Pattern for this function. A Facade Pattern is an object that serves as a front-facing interface masking more complex underlying or structural code. A facade can improve the readability and usability of a software library by masking interaction with more complex components behind a single. Bridge Pattern is a structural design pattern that lets us split a large class or a set of closely related classes into two separate hierarchies - abstraction and implementation - which can be developed independently of each other. Factory Pattern is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

- The Facade Pattern is for getting meat type and week, day of the menu. Classes in the Pattern:

+ WeeklyMenu – Get the week that we have input.
+ DailyMenu – Get the day that we have input.
+ MeatType - Getting the meat type we have choose.
+ MenuFacade – Contain many subclasses, those to get the week, day and type of meat.

C# code:

```
public class WeeklyMenu
    {
        public string getWeek(string Week)
        {
            return "You have add a new menu to week " + Week;
        }
    }
```

```
public class MeatType
{
    public string Chicken()
    {
        return ("chicken meat");
    }

    public string Pork()
    {
        return ("pork meat");
    }

    public string Beef()
    {
        return ("beef meat");
    }
}

public class DailyMenu
{

    public string getDay(string Day)
    {
        return (", " + Day);
    }
}
public class MenuFacade
{
    public static MenuFacade INSTANCE = new MenuFacade();
    private WeeklyMenu weeklymenu;
    private MeatType meattype;
    private DailyMenu dailymenu;

    private MenuFacade()
    {
        weeklymenu = new WeeklyMenu();
        meattype = new MeatType();
        dailymenu = new DailyMenu();
    }

    public static MenuFacade getInstance()
    {
        return INSTANCE;
    }
    public string getweek()
    {
        return "You have add a new menu to week " + weeklymenu;
    }
        public string getChicken()
    {
        return "Meat type chicken.";
    }
    public string getPork()
    {
        return "Meat type pork.";
    }
    public string getBeef()
    {
```

```
            return "Meat type beef.";
        }
    }
}
```

- The Bridge Pattern is for getting the cooking type for the menu. Classes in the Pattern:

+ ChieckingCookType – For check the cook type.
+ SavingCookType – For save the cook type.
+ Class CookType - For define the subclasses cook type: Grilled, Fry, Boiled.

C# code:

```
public class CheckingCookType : TypeCook
    {
        public string OpenCookType()
        {
            return "Checking";
        }
    }
    public class SavingCookType : TypeCook
    {
        public string OpenCookType()
        {
            return "Saving";

        }
    }
    public abstract class CookType
    {
        protected TypeCook cooktype;
        public CookType(TypeCook cooktype)
        {
            this.cooktype = cooktype;
        }
        public abstract string OpenCookType();
    }
    public class Grilled : CookType
    {

        public Grilled(TypeCook cooktype) : base(cooktype)
        {
            this.cooktype = cooktype;
        }

        public override string OpenCookType()
        {

            cooktype.OpenCookType();
            return "Cook by grilled.";

        }
    }
    public class Fry : CookType
    {
        public Fry(TypeCook cooktype) : base(cooktype)
        {
```

```
            this.cooktype = cooktype;
        }
        public override string OpenCookType()
        {
            //print open a normal account is a
            cooktype.OpenCookType();
            return "Cook by fry.";
        }
    }

    public class Boiled : CookType
    {
        protected Type type;
        public Boiled(TypeCook cooktype) : base(cooktype)
        {
            this.cooktype = cooktype;
        }
        public override string OpenCookType()
        {
            //print open a normal account is a
            cooktype.OpenCookType();
            return "Cook by boiled.";
        }
    }
```

- The Factory Pattern is for checking those cooking type of the menu:

+ Class CookType – to define the subclasses in it, contain: Grilled, Fry, Boiled.

C# code:

```
public class MenuFactory
    {
        private MenuFactory()
        {
        }
        public static CookType getCookType(Cooktypes CookType)
        {
            switch (CookType)
            {
                case Cooktypes.Grilled:
                    return new Grilled(new CheckingCookType());
                case Cooktypes.Fry:
                    return new Fry(new CheckingCookType());
                case Cooktypes.Boiled:
                    return new Boiled(new CheckingCookType());
                default:
                    return null;
            }
        }
    }
    public enum Cooktypes
    {
        Grilled, Fry, Boiled
    }
}
```