MINISTRY OF EDUCATION AND TRAINING
HCMC UNIVERSITY OF TECHNOLOGY AND EDUCATION
FACULTY FOR HIGH-QUALITY TRAINING

HCMUTE

Final Project
Health checking daily app

Course: Object-Oriented Software Design

Group 6:

NAME	STUDENT'S ID
Thieu Quang Khanh	18110022
Nguyen Dinh Long	18110027

HCMC May, 2021

Object-oriented software
Design pattern

1. Register function

Name of function: Register function
Name of design pattern: Builder
Reason: Create a complex object: have many properties (more than 4) and some are required, some are optional.

- Too many constructors
- Want to decouple the construction of a complex object from the parts that make up the object.
- Want to control the build process.
- The user (client) expects different ways for the object to be constructed.

Characteristic: Support, eliminate the need to write multiple constructors.

- Code is easier to read and maintain when the number of properties required to create an object is 4 or 5 properly.
- Reduce the number of constructors, no need to pass null values for unused parameters.
- Less error due to wrong parameter assignment when there are many parameters in the constructor:
- Constructed object is safer: because it is fully created before use.
- Gives you better control over the construction process: we can add binding check handling before the object is returned to the user..

C# code:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace healthcheckapp
{
  public class Account
  {
     private string Name;
     private string Address;
     private string Mail;
     private string Phone;
     public Account(string Name, string Address, string Mail, string Phone)
     {
```

```
        this.Name = Name;
        this.Address = Address;
        this.Mail = Mail;
        this.Phone = Phone;
    }
    public class AccountBuilder
    {
        private string Name;
        private string Address;
        private string Mail;
        private string Phone;
        public AccountBuilder(string Name)
        {
            this.Name = Name;
        }
        public AccountBuilder withAddress(string Address)
        {
            this.Address = Address;
            return this;
        }
        public AccountBuilder withMail(string Mail)
        {
            this.Mail = Mail;
            return this;
        }
        public AccountBuilder withPhone(string Phone)
        {
            this.Phone = Phone;
            return this;
        }
        public Account build()
        {
            validateUserObject();
            Account account = new Account(this.Name, this.Address,
this.Mail, this.Phone);
            return account;
        }
        private void validateUserObject()
        {
            if (this.Mail == null)
```

```
            {
               //Message mail cannot be null
            }

         }
      }
      public override string ToString()
      {
         return "Account name= " + this.Name + " account Address= " +
this.Address + " account Mail= " + this.Mail + " account Phone= " +
this.Phone;
      }
   }
}
```

2. User login/logout

Name of function: User login/logout
Name of design pattern: Command
Reason: need to parameterize objects according to an action performed.

- need to create and execute requests at different times.

Characteristic: Easily add new Commands in the system without making changes in existing classes. Ensure Open/Closed Principle.

- Separate the object that calls the operation from the object that actually performs the operation. Reduce connection between Invoker and Receiver.
- Allows to parameterize different requests with one action to perform.
- Allows to save requests in the queue.
- Encapsulate a request in an object. Easily transfer data as objects between system components.

c# code:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace healthcheckapp
{
  public class AccountCommand
```

```
{
    private string Name;
    public AccountCommand(string Name)
    {
        this.Name = Name;
    }
    public string SignOut()
    {
        return "Account" + this.Name + "Sign Out";
    }
    public string Close()
    {
        return "Account" + this.Name + "Closed";
    }

}
public interface Command
{
    string excute ();

}
public class SignOut : Command
{
    private AccountCommand account;
    public SignOut(AccountCommand account)
    {
        this.account = account;
    }
    public string excute ()
    {
        return account.SignOut();
    }
}
public class CloseAccount : Command
{
    private AccountCommand account;
    public CloseAccount(AccountCommand account)
    {
        this.account = account;
    }
```

```
    public string excute ()
    {
        return account.Close();
    }
  }
  public class HealthApp
  {

    private Command signOut;
    private Command closeAccount;

    public HealthApp(Command signOut, Command closeAccount)
    {
        this.signOut = signOut;
        this.closeAccount = closeAccount;
    }

    public string clickSignOutAccount()
    {

        return signOut.Excute();
    }

    public string clickCloseAccount()
    {
        //print("User clicked close an account");
        return closeAccount.Excute();
    }
  }
}
```

3. Purchase product

Name of function : Purchase product
Name of design pattern: Facade
Reason: the system has many layers that make it difficult for the user to understand the program's process. And when there are many subsystems, each of which has its own interfaces, it is difficult to use it together. The Facade Pattern can then be used to create a simple user interface for a complex system.

- The user is highly dependent on the implementation classes. The application of the Facade Pattern separates the user's subsystem from other subsystems, thus increasing independence and portability the subsystem's, making it easier to switch for future upgrades.
- want to classify subsystems. Use the Facade Pattern to define a common interface for each subsystem, thus reducing the dependency of subsystems since these systems only communicate with each other through those common interface ports.
- want to wrap, hide the complexity in subsystems for the Client side.

Characteristic: Makes your system simpler to use and understand, since a Facade pattern has convenient methods for common tasks.

- Reduces the dependency of external code on the internal implementation of the library, since most of the code uses Facade, thus allowing flexibility in the development of systems.
- Encapsulating a set of poorly designed API functions equals a single better designed API function.

C# code :

```
using System;
using System.Collections.Generic;
using System.Text;

namespace healthcheckapp
{
  public class AccountService
  {
     public string getAccount(string Email)
     {
        return "Account with" + Email;
     }
  }
  public class PaymentService
  {
     public string paymentByPaypal()
     {
        return ("Payment by Paypal");
     }

     public string paymentByCreditCard()
     {
        return ("Payment by Credit Card");
```

```
    }

    public string paymentByEbankingAccount()
    {
        return ("Payment by E-banking account");
    }

    public string paymentByCash()
    {
        return ("Payment by cash");
    }
}
public class EmailService
{

    public string sendMail(string mailTo)
    {
        return ("Sending an email to " + mailTo);
    }
}
public class SmsService
{

    public string sendSMS(string mobilePhone)
    {
        return ("Sending an message to " + mobilePhone);
    }
}
public class AccountFacade
{
    public static AccountFacade INSTANCE = new AccountFacade();
    private AccountService accountService;
    private PaymentService paymentService;
    private EmailService emailService;
    private SmsService smsService;

    private AccountFacade()
    {
        accountService = new AccountService();
        paymentService = new PaymentService();
```

```
        emailService = new EmailService();
        smsService = new SmsService();
    }

    public static AccountFacade getInstance()
    {
        return INSTANCE;
    }

    public string buyProductByCash(String email)
    {
        return accountService.getAccount(email)+
        paymentService.paymentByCash()+
        emailService.sendMail(email);

    }

    public string buyProductByPaypal(String email, String mobilePhone)
    {
        return accountService.getAccount(email)+
        paymentService.paymentByPaypal()+
        emailService.sendMail(email)+
        smsService.sendSMS(mobilePhone);
        //print("Done\n");
    }

  }
}
```

4. Initialize member type

Name of function : Initialize member type
Name of design pattern : Factory
Reason: We have a super class with many subclasses and based on input we need to return a subclass. This model helps us to transfer the responsibility of instantiating a class from the client side to the Factory class.

- We don't know what subclasses will be needed in the future. When needing to extend, create subclass and implement additional factory method for instantiating this subclass.

Characteristic: Factory Pattern helps to reduce dependency between modules (loose coupling): provides an approach to interfaces instead of implementations.

Make the program independent of the specific classes that we need to create an object, the code on the client side is not affected when changing logic in the factory or subclass.

- Extending your code is easier: when you need to extend it, just create a subclass and implement it in a factory method.
- Initialize Objects that hide the logic of the initialization. The user doesn't know the real logic that is initialized below the factory method.
- Easily manage the life cycle of Objects created by the Factory Pattern.
- Unified naming convention: helps developers understand the structure of source code.

c# code:

```
public interface Account{
  string getTypeName();
}
public class SMember : Account{
  public override string getTypeName(){
    return "SMember";
  }
}
public class SSMember:Account{
  public override string getTypeName(){
    return "SSMember";
  }
}
public class AccountFactory{
  private AccountFactory(){
  }
  public static Account getAccount(accountTypeAccountType ){
    switch(accountType){
      case SMember:
        return new SMember();
      case SSMember:
        return new SSMember();
      default:
        return " this type is not support";
    }
  }
}
public enum AccountType{
```

```
  SMember, SSMember,
}
```

5. Provide clients

Name of function : Provide clients with different account types
Name of design pattern : Bridge
Reason : want to separate the constraint between Abstraction and Implementation, so that it can be easily extended independently of each other .

- Both their Abstraction and Implementation should be extended by subsclass.
- Use where changes made in the implementation do not affect the client side.

Characteristic: Reduce dependency between abstraction and implementation (loose coupling): inheritance in OOP often binds abstraction and implementation at build time. The Bridge Pattern can be used to break this dependency and allow us to choose the right implementation at runtime.

- Reduce the number of unnecessary subclasses: some use cases of inheritance will increase the number of subclasses greatly. For example, in case the program views images on different operating systems, we have 6 types of images (JPG, PNG, GIF, BMP, JPEG, TIFF) and 3 operating systems (Window, MacOS, Ubuntu). Using inheritance in this case will make us design 18 classes: JpgWindow, PngWindow, GifWindow, ... While applying Bridge will reduce the number of classes to 9 classes: 6 classes for each implementation of Image and 3 classes for each operating system, each operating system will include a reference to a specific Image object.
- The code will be cleaner and the application size will be smaller: by reducing the number of unnecessary classes.
- Easier to maintain: its Abstractions and Implementations will be easy to change at runtime as well as when it needs to be changed in the future.
- Easy to extend later: usually large applications often require us to add modules to the existing application, but do not modify the existing framework/application because those frameworks/applications can be upgraded by the company to the new version. Bridge Pattern will help us in this case.
- Allows hiding implementation details from the client: since the abstraction and implementation are completely independent, we can change a component without affecting the client side. For example, the classes of the image viewer program will be independent of the drawing

algorithm in the implementation. Thus, we can update the image viewer when there is a new image drawing algorithm without much modification.

C# code :

```
using System;
using System.Collections.Generic;
using System.Text;
public interface Type
{
  string OpenAccount();
}
namespace healthcheckapp
{

  public class CheckingAccount : Type
  {
     public string OpenAccount()
     {
        return "Checking...";
     }
  }
  public class SavingAccount : Type
  {
     public string OpenAccount()
     {
        return "Saving....";

     }
  }
  public abstract class MemberType
  {
     protected Type type;
     public MemberType(Type type)
     {
        this.type = type;
     }
     public abstract string OpenAccount();
  }
  public class : SMemberMemberType
  {
```

```
    protected Type type;
    public SMember(Type type):base(type)
    {
        this.type = type;
    }

    public override string OpenAccount()
    {

        type.OpenAccount();
        return "SMember is Opened";

    }
}
public class SSMember : MemberType
{
    protected Type type;
    public SSMember(Type type):base(type)
    {
        this.type = type;
    }
    public override string OpenAccount()
    {
        //print open a normal account is a
        type.OpenAccount();
        return "SSMember is opened";
    }
}
public class client
{

    public static void main(String[] args)
    {
        MemberType sMember = new SMember(new CheckingAccount());
        sMember.OpenAccount();
        MemberType sSMember = new SSMember(new
CheckingAccount());
        sSMember.OpenAccount();
    }
}
```

```
}
```

6. Template method

Name of function : Provide client with different interfaces

Name of design pattern: Template method

Reason:

- When there is an algorithm with multiple steps and wish to allow customization of them in the subclass.
- Expect to have only a single abstract method implementation of an algorithm.
- Expect common behavior between subclasses should be placed in a common class.
- Superclasses can call behaviors in their subclasses uniformly (step by step).

Characteristic:

- Reuse code (reuse), avoid code duplication (duplicate): put duplicate parts into the parent class (abstract class).
- Allows users to override only certain parts of a large algorithm, making them less susceptible to changes occurring to other parts of the algorithm.

C# code:

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace healthcheckapp
{

    public abstract class PageTemplate
    {

        protected void showHeader(MenuStrip menuStrip)
        {
            menuStrip.Visible = true;
        }
```

```
    protected void showNavigation()
    {

    }

    protected void showFooter(MenuStrip menuStrip)
    {
        menuStrip.Visible = true;
    }

    protected abstract void showImage(PictureBox pictureBox);

    public void showPage(MenuStrip menuStrip, MenuStrip menuStrip1, PictureBox pictureBox)
    {
        showHeader(menuStrip);
        showNavigation();
        showImage(pictureBox);
        showFooter(menuStrip1);
    }
}
public class DietImage : PageTemplate
{


    protected override void showImage(PictureBox pictureBox)
    {
        pictureBox.Visible = true;
        pictureBox.BringToFront();
    }
}

public class HomeImage : PageTemplate
{
    protected override void showImage(PictureBox pictureBox)
    {
        pictureBox.Visible = true;
        pictureBox.BringToFront();
    }
}
}
```