

МІНІСТЕРСТВО ОСВІТИ І НАУКИ, МОЛОДІ ТА СПОРТУ
УКРАЇНИ
ДНІПРОДЗЕРЖИНСЬКИЙ ДЕРЖАВНИЙ ТЕХНІЧНИЙ
УНІВЕРСИТЕТ

МЕТОДИЧНІ ВКАЗІВКИ

до лабораторних занять з дисципліни
„Моделювання та аналіз програмного забезпечення”
для студентів напрямку
6.050103 „Програмна інженерія”

Затверджено редакційно-видавничою
Секцією науково методичної ради ДДТУ
протокол № 5 від 17.01.2013 р.,

Дніпродзержинськ
2013

Методичні вказівки до лабораторних занять з дисципліни „Моделювання та аналіз програмного забезпечення” для студентів напрямку 6.050103 „Програмна інженерія” / К.М. Ялова, В.В.Завгородній // Дніпродзержинськ: ДДТУ, 2013. – 51 с.

Укладачі: к.т.н., доц. Ялова К.М.
ст. викл. Завгородній В.В.

Рецензент: проф. Самохвалов С.Є.
(Дніпродзержинський державний технічний університет)

Затверджено на засіданні кафедри ПЗС
(протокол № 1 від 31 вересня 2012 р.)

Наведено зміст та описання лабораторних занять з дисципліни „Моделювання та аналіз програмного забезпечення”, в яких розглянуто основні принципи об’єктно-орієнтованого моделювання програмного забезпечення з використанням паттернів проектування на основі аналізу предметної області.

ЗМІСТ

1. ВСТУП.....	4
2. ЛАБОРАТОРНІ ЗАНЯТТЯ.....	5
2.1. Лабораторне заняття Л1. Розв’язання задач із використанням основних принципів об’єктно-орієнтованого програмування	5
2.2. Лабораторне заняття Л2. Введення в проектування програмного забезпечення з використанням шаблонів. Породжувальний паттерн Абстрактна Фабрика (Abstract Factory).....	17
2.3. Лабораторне заняття Л3. Приклади використання породжувального паттерна Фабричний Метод (Factory Method).....	21
2.4. Лабораторне заняття Л4. Розв’язання задач із використанням структурного паттерна Адаптер (Adapter).....	25
2.5. Лабораторне заняття Л5. Побудова програмних додатків із використанням структурного паттерна Фасад (Facade).....	30
2.6. Лабораторне заняття Л6. Загальні принципи побудови програмних додатків із використанням шаблонів поведінки. Паттерн Ітератор (Iterator).....	34
2.7. Лабораторне заняття Л7. Розробка програмних додатків із використанням паттерна поведінки Спостерігач (Observer).....	40
2.8. Лабораторне заняття Л8. Принципи використання паттерна поведінки Стратегія (Strategy), програмна реалізація...	44
3. РЕКОМЕНДОВАНА ЛІТЕРАТУРА.....	48
4. МЕТОДИЧНЕ ЗАБЕЗПЕЧЕННЯ.....	50

ВСТУП

Дисципліна „Моделювання та аналіз програмного забезпечення” викладається для студентів напряму 6.050103 „Програмна інженерія”.

Дисципліна спрямована на оволодіння навичками використання архітектурного стилю проектування, ґрунтованого на визначенні основних об’єктів системи, їх атрибутів, поведінки та взаємодії; розробки програмного забезпечення при застосуванні паттернів проектування об’єктів і класів: структурних, породжувальних та паттернів поведінки.

Мета дисципліни – формування знань про способи проектування програмного забезпечення на основі моделі предметної області, вміння ефективного застосування компонентів багаторазового використання – паттернів проектування при використанні технології об’єктно-орієнтованого програмування.

Задачею вивчення дисципліни є набуття:

- теоретичних знань щодо моделювання якісного програмного забезпечення з використанням паттернів проектування;
- практичних навичок та умінь вибору доцільних паттернів проектування, їх програмної реалізації з використанням принципів об’єктно-орієнтованого програмування засобами мов програмування високого рівня.

На лабораторних заняттях студенти закріплюють теоретичні знання, що отримують на лекціях і під час самостійної роботи.

У результаті вивчення навчальної дисципліни студент повинен **знати**:

- основні положення теорії проектування програмного забезпечення на основі аналізу предметної області;
- характеристики та признаи якісного програмного забезпечення;
- способи та засоби представлення описів об’єктів системи;

- основні прийоми об'єктно-орієнтованого проектування програмних додатків та систем;
- механізми повторного використання програмного коду;
- види паттернів створення класів та об'єктів GoF;
- задачі паттернів проектування;
- невід'ємні складові частини описання паттернів проектування;
- способи використання та практичні підходи при розробці програмного забезпечення з використанням шаблонів проектування;
- переваги використання паттернів проектування.

вміти:

- аналізувати предметну область у рамках поставленої задачі на основі сформульованих вимог до програмного забезпечення;
- представляти опис об'єктів предметної області, їх структури і поведінки засобами уніфікованої мови моделювання UML;
- створювати діаграму класів у рамках поставленої задачі;
- визначати способи підвищення якості програмного коду;
- знаходити найдоцільніші паттерни проектування для розв'язання поставленої задачі;
- застосовувати надбані навички об'єктно-орієнтованого програмування при розробці програмного забезпечення з використанням паттернів проектування.

2. ЛАБОРАТОРНІ ЗАНЯТТЯ

Виконання завдань лабораторного заняття передбачає:

- словесне описання предметної області в рамках поставленої задачі (постановка задачі);
- створення діаграми класів предметної області;
- розробку програмного забезпечення по діаграмі класів;
- представлення результату виконання програми;
- висновки.

2.1. Лабораторне заняття Л1. Розв'язання задач із використанням основних принципів об'єктно-орієнтованого

програмування.

Мета лабораторного заняття: повторення загальних принципів моделювання програмного забезпечення з використанням принципів об'єктно-орієнтованого проектування: механізми спадкування, композиції, інкапсуляції, поліморфізму.

Приклад завдання 1.

У заданій предметній області створити 2 абстракції, відтворити їх у 2 класах, що знаходяться у відношенні композиції („клієнт-постачальник”). Результат взаємодії класів відтворити в консольному додатку із використанням мови програмування високого рівня.

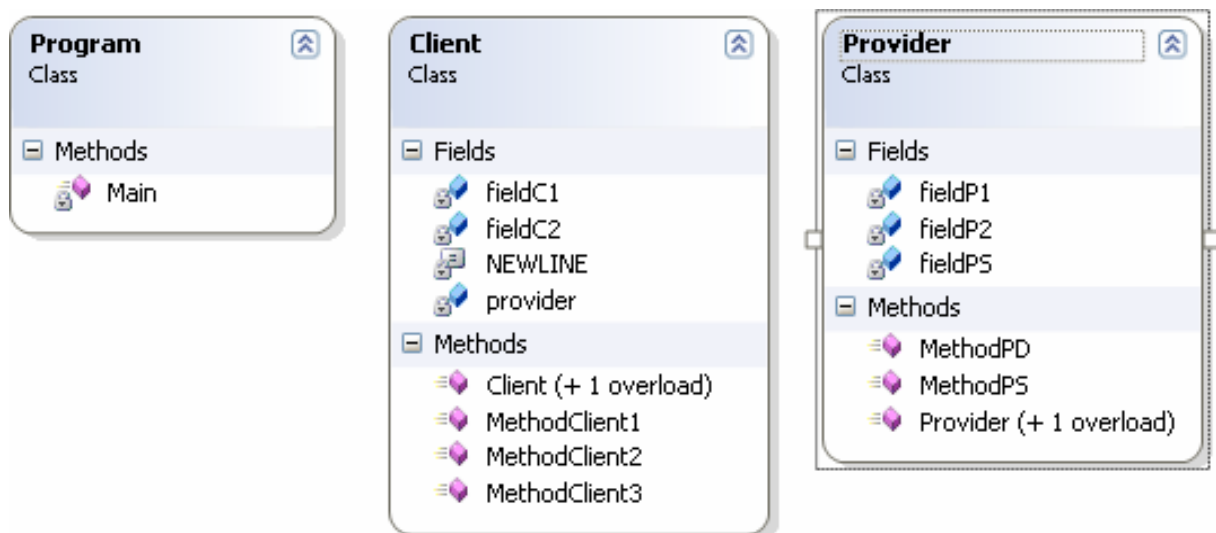
Розв'язання:

Описання предметної області. Нехай в предметній області існує два класи Provider і Clent, що пов'язані відношенням композиції. Клас постачальник Provider має три поля, одне з яких статичне, два конструктора, два методи – один статичний і один динамічний. Поля класу – закриті для клієнтів (по принципу інкапсуляції). У класі, як і належить, є конструктор без аргументів, що ініціалізує поля класу відповідними константами, і конструктор з аргументами, який перетворює передані йому значення, перш ніж записати їх у поля класу. Методи класу дозволяють отримати інформацію, що зберігається в полях. Динамічний (екземплярний) метод MethodPD, якому доступні поля класу, повертає рядок з інформацією про збережені значення в полях. Одночасно цей метод збільшує значення, збережене в статичному полі, яке можна розглядати як лічильник загального числа викликів динамічного методу всіма об'єктами даного класу. Статичний метод MethodPS, якому доступне тільки статичне поле, повертає в якості результату рядок з інформацією про кількість викликів динамічного методу.

Клас Client – клієнт класу Provider. Клас буде влаштований схожим чином. Істотне доповнення полягає в тому, що одним із

полів є об'єкт provider класу Provider. Конструктори класу Client створюють об'єкт постачальника (класу Provider), викликаючи конструктор постачальника. Для створення об'єктів постачальника можуть бути необхідні аргументи, тому вони передаються конструктору клієнта. Створюючи об'єкт класу Client, конструктори цього класу створюють і об'єкт класу Provider, пов'язуючи його посиланням з полем provider. Всі динамічні методи клієнтського класу можуть використовувати цей об'єкт, викликаючи доступні клієнту методи і поля класу постачальника. Метод класу Client – MethodClient1 починає свою роботу з виклику: provider.MethodPD(), викликаючи сервіс, що поставляється методом класу Provider.

Діаграма класів предметної області



Лістинг створеного програмного забезпечення (C#)

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace lab1
{
    class Provider
    {
        //fields
        string fieldP1;
        int fieldP2;
        static int fieldPS;
        //Конструкторы класса
        /// <summary>

```

```

    /// Конструктор с аргументами
    /// </summary>
    /// <param name="p1">аргумент, инициализирующий поле
класса</param>
    /// <param name="p2">аргумент, инициализирующий поле
класса</param>
    public Provider(string p1, int p2)
    {
        fieldP1 = p1.ToUpper(); fieldP2 = p2 * 2;
        fieldPS = 0;
    }
    public Provider()
    {
        fieldP1 = ""; fieldP2 = 0; fieldPS = 0;
    }
    //Динамический (Экземплярный) метод
    public string MethodPD()
    {
        fieldPS++;
        string res = "Объект класса Provider" + "\n\r";
res += string.Format("Мои поля: поле1 = {0}, поле2 = {1}",
        fieldP1, fieldP2);
        return res;
    }
    // Статический (Модульный) метод
    public static string MethodPS()
    {
        string res = "Модуль класса Provider" + "\n\r";
res += string.Format("Число вызовов метода MethodPD = {0}",
        fieldPS.ToString());
        return res;
    }
}
class Client
{
    //fields
    Provider provider;
    string fieldC1;
    int fieldC2;
    const string NEWLINE = "\n\r";
    //Конструкторы класса
    public Client(string p1, int p2, string c1, int c2)
    {
        fieldC1 = c1.ToLower(); fieldC2 = c2 - 2;
        provider = new Provider(p1, p2);
    }
    public Client()
    {
        fieldC1 = ""; fieldC2 = 0;
        provider = new Provider();
    }
    /// <summary>
    /// Метод, использующий поле класса provider

```

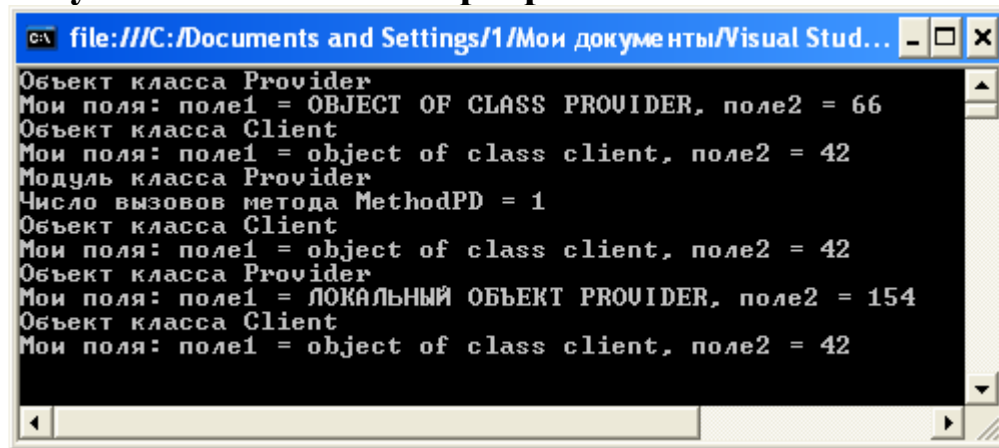


```

    /// для работы с методами класса Provider
    /// </summary>
    /// <returns>композиция строк провайдера и клиента
</returns>
    public string MethodClient1()
    {
        string res = provider.MethodPD() + NEWLINE;
        res += "Объект класса Client" + NEWLINE;
        res += string.Format("Мои поля: поле1 = {0}, поле2 = {1}",
            fieldC1, fieldC2);
        return res;
    }
    public string MethodClient3()
    {
        Provider local_provider =
            new Provider("Локальный объект Provider", 77);
        string res = local_provider.MethodPD() + NEWLINE;
        res += "Объект класса Client" + NEWLINE;
        res += string.Format("Мои поля: поле1 = {0}, поле2 = {1}",
            fieldC1, fieldC2);
        return res;
    }
    public string MethodClient2()
    {
        string res = Provider.MethodPS() + NEWLINE;
        res += "Объект класса Client" + NEWLINE;
        res += string.Format("Мои поля: поле1 = {0}, поле2 = {1}",
            fieldC1, fieldC2);
        return res;
    }
}
class Program
{
    static void Main(string[] args)
    {
        Client client = new Client("object of class
Provider", 33, "object of class Client", 44);
        string res = client.MethodClient1();
        Console.WriteLine(res);
        res = client.MethodClient2();
        Console.WriteLine(res);
        res = client.MethodClient3();
        Console.WriteLine(res);
        Console.ReadKey();
    }
}
}

```

Результат виконання програми:



```

Объект класса Provider
Мои поля: поле1 = OBJECT OF CLASS PROVIDER, поле2 = 66
Объект класса Client
Мои поля: поле1 = object of class client, поле2 = 42
Модуль класса Provider
Число вызовов метода MethodPD = 1
Объект класса Client
Мои поля: поле1 = object of class client, поле2 = 42
Объект класса Provider
Мои поля: поле1 = ЛОКАЛЬНЫЙ ОБЪЕКТ PROVIDER, поле2 = 154
Объект класса Client
Мои поля: поле1 = object of class client, поле2 = 42
  
```

Приклад завдання 2. У заданій предметній області створити 2 абстракції, відтворити їх у 2 класах, що знаходяться у відношенні спадковості. Продемонструвати перевантаження, приховування та перевизначення батьківських методів. Результат взаємодії класів відтворити у Windows-додатку з використанням мови програмування високого рівня.

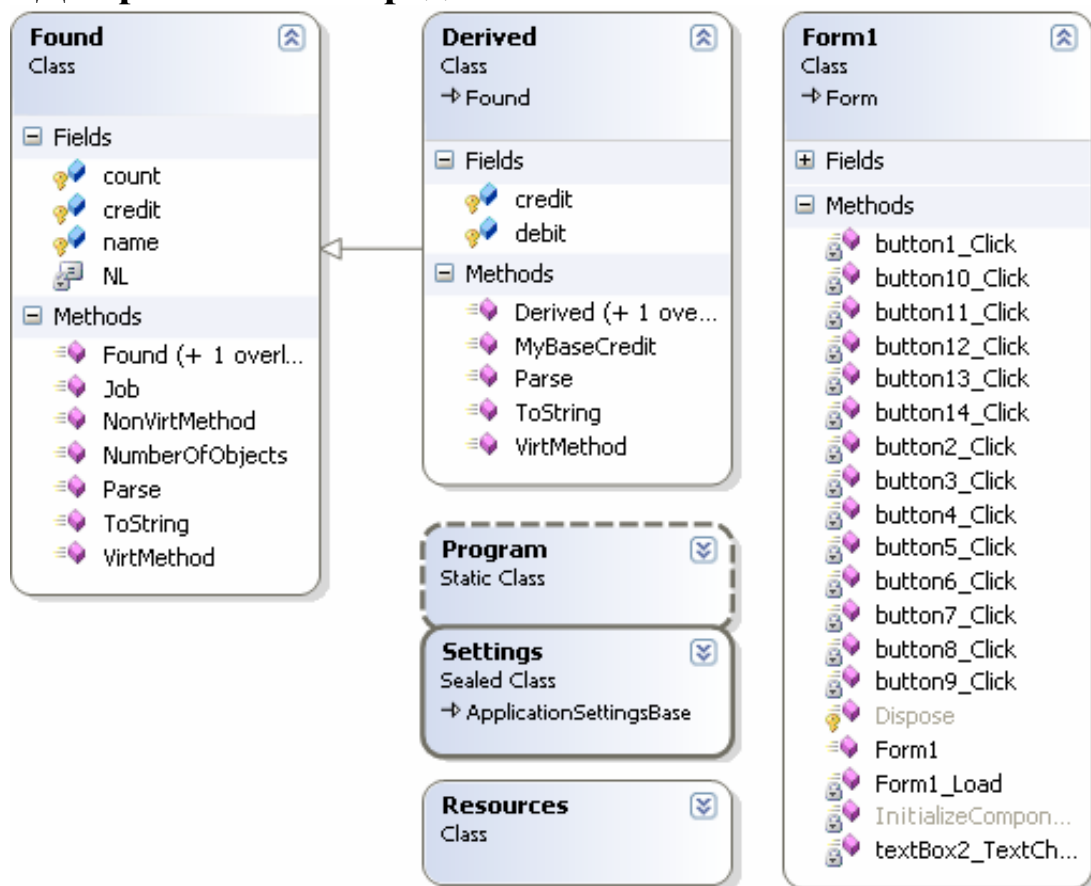
Розв'язання:

Описання предметної області. Клас, названий Found, буде грати роль батьківського класу. Клас Found має три поля. Поля закриті для клієнтів класу, але відкриті для нащадків. Одне з полів є статичним, змістовно воно буде використовуватися для підрахунку числа створених об'єктів класу Found клієнтами класу. Як і належить, крім конструктора з аргументами, переданими для ініціалізації екземплярних полів класу, в класі є конструктор без аргументів, що ініціалізує поля класу деяким заданим за умовчанням способом. Обидва конструктора в процесі роботи збільшують на одиницю значення статичного поля, яке до цього часу вже створене та ініціалізоване. Клас Found перевизначає метод ToString. Він повертає деякий рядок, одержаний конкатенацією константи і рядка, що є результатом виклику тільки що перевизначеного методу ToString. Ім'я методу уточнено ім'ям поточного об'єкта this, щоб підкреслити той факт, що саме поточний об'єкт викликає метод класу ToString. Метод Job по черзі

викликає методи класу – `VirtMethod`, `NonVirtMethod`, `Parse`. Рядки, що задають результати цих методів, з'єднуються, і отриманий рядок повертається в якості результату методу `Job`. Останній метод, який додамо в клас `Found` – це статичний метод, який повертає інформацію із статичного поля класу.

Клас `Derived` – нащадок класу `Found`. Будемо також вважати, що поля `debit` і `credit` повинні мати тип `double`, а не `int`. З цієї причини приховаємо батьківське поле `credit`, додавши власне поле з тим же ім'ям, але змінивши його тип. Клас `Derived` розширив можливості методу `Parse` класу `Found`, додавши перевірку в метод розбору. Клас `Derived` перевизначає методи `VirtMethod` та `ToString`, дотримуючи контракт, що укладається в цьому випадку між батьком і нащадком.

Діаграма класів предметної області



Лістинг створеного програмного забезпечення (C#)

```

using System;
using System.Collections.Generic;
using System.ComponentModel;

```

```

using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
namespace _1_2
{
    public class Found
    {
        //fields
        protected string name;
        protected int credit;
        static protected int count;
        const string NL = "\r\n";
        //Constructors
        public Found()
        {
            name = "Nemo";
            credit = 0;
            count++;
        }
        public Found(string name, int credit)
        {
            this.name = name;
            this.credit = credit;
            count++;
        }
        public override string ToString()
        {
            string s = "Поля: name = {0}, credit = {1}";
            return String.Format(s, name, credit);
        }
        public string NonVirtMethod()
        {
            return "Found: " + this.ToString();
        }
        public virtual string VirtMethod()
        {
            return "Found: " + this.ToString();
        }
        public string Parse()
        {
            return "Выполнен разбор кода!";
        }
        public string Job()
        {
            string res = "";
            res += "VirtMethod: " + VirtMethod() + NL;
            res += "NonVirtMethod: " + NonVirtMethod() + NL;
            res += "Parse: " + Parse() + NL;
            return res;
        }
        public static string NumberOfObjects()

```

```

        {
            return "Объектов создано: " + count;
        }
    }
    public class Derived : Found
    {
        //добавление и скрытие полей
        protected double debit;
        new protected double credit;
        //Конструкторы
        public Derived(): base()
        {
            debit = 0; credit = 0;
        }
        public Derived(string name, double debit,
double credit): base(name, (int)credit)
        {
            this.debit = debit;
            this.credit = credit;
        }
        //Методы
        public string MyBaseCredit()
        {
            return base.credit.ToString();
        }
        new public string Parse()
        {
            string res = base.Parse();
            res += "Выполнена проверка кода!";
            return res;
        }
        public override string ToString()
        {
            string s = "Поля: name = {0}, Basecredit = {1}" +
"credit = {2}, debit = {3}";
            return String.Format(s, name, base.credit, credit, debit);
        }
        public override string VirtMethod()
        {
            return "Derived: " + this.ToString();
        }
    }
    public partial class Form1 : Form
    {
        public Found found;
        public Derived derived;
        public Form1()
        {
            InitializeComponent();
            found = new Found();
            derived = new Derived();
        }
        private void Form1_Load(object sender, EventArgs e)

```

```

    {
    }
private void button1_Click(object sender, EventArgs e)
{
    string name = textBoxFound_Name.Text;
    int credit = 0;
    try
    {
        credit = int.Parse(textBoxF_Credit.Text);
    }
    catch (Exception)
    {
        textBoxResult.Text = "Некорректно заданы числовые данные!";
    }
    if (name != "")
        found = new Found(name, credit);
    textBoxResult.Text = "Объект класса Found успешно создан!";
}
private void textBox2_TextChanged(object sender, EventArgs e)
{
}
private void button3_Click(object sender, EventArgs e)
{
    string res = found.VirtMethod();
    textBoxResult.Text = res;
}
private void button6_Click(object sender, EventArgs e)
{
    string res = found.NonVirtMethod();
    textBoxResult.Text = res;
}
private void button4_Click(object sender, EventArgs e)
{
    string res = found.ToString();
    textBoxResult.Text = res;
}
private void button7_Click(object sender, EventArgs e)
{
    string res = found.Parse();
    textBoxResult.Text = res;
}
private void button5_Click(object sender, EventArgs e)
{
    string res = found.Job();
    textBoxResult.Text = res;
}
private void button8_Click(object sender, EventArgs e)
{
    string res = Found.NumberOfObjects();
    textBoxResult.Text = res;
}
private void button2_Click(object sender, EventArgs e)
{
    string name = textBoxD_Name.Text;
    double credit = 0, debit = 0;
}

```

```

        try
        {
            credit = double.Parse(textBoxD_Credit.Text);
            debit = double.Parse(textBoxD_Debit.Text);
        }
        catch (Exception)
        {
            textBoxD_Result.Text = "Некорректно заданы числовые данные!";
        }
        if (name != "")
            derived = new Derived(name, debit, credit);
        textBoxD_Result.Text = "Объект класса Derived успешно создан!";
    }
    private void button9_Click(object sender, EventArgs e)
    {
        string res = derived.VirtMethod();
        textBoxD_Result.Text = res;
    }
    private void button12_Click(object sender, EventArgs e)
    {
        string res = derived.NonVirtMethod();
        textBoxD_Result.Text = res;
    }
    private void button10_Click(object sender, EventArgs e)
    {
        string res = derived.ToString();
        textBoxD_Result.Text = res;
    }
    private void button13_Click(object sender, EventArgs e)
    {
        string res = derived.Parse();
        textBoxD_Result.Text = res;
    }
    private void button11_Click(object sender, EventArgs e)
    {
        string res = derived.Job();
        textBoxD_Result.Text = res;
    }
    private void button14_Click(object sender, EventArgs e)
    {
        string res = Derived.NumberOfObjects();
        textBoxD_Result.Text = res;
    }
}

```

Результат виконання програми:

The screenshot shows a Windows application window titled "Form1" with two main sections: "Found" and "Derived".

Found Section:

- Имя: RTE
- Кредит: 100
- Buttons: Создать объект, VirtMethod, NonVirtMethod, ToString, Parse, Job, NumberOfObject
- Результаты:


```
VirtMethod: Found: Поля: name = RTE, credit = 100
NonVirtMethod: Found: Поля: name = RTE, credit = 100
Parse: Выполнен разбор кода!
```

Derived Section:

- Имя: QAZ
- Кредит: 13
- Дебит: 12
- Buttons: Создать объект, VirtMethod, NonVirtMeth, ToString, Parse, Job, NumberOfObject
- Результаты:


```
VirtMethod: Derived: Поля: name = QAZ, Basecredit = 13
credit = 13, debit = 12
NonVirtMethod: Found: Поля: name = QAZ, Basecredit = 13
credit = 13, debit = 12
Parse: Выполнен разбор кода!
```

Висновки по лабораторному заняттю 1:

- Клієнт класу може створювати об'єкти класу постачальника, а потім, використовуючи створені об'єкти, отримувати доступ до сервісів, що надаються класом постачальника.

- Клієнти класу отримують доступ до методів класу постачальника. Сукупність цих методів визначає інтерфейс класу постачальника і ті сервіси, які постачальник надає своїм клієнтам.

- Нащадок класу, успадкувавши від батьківського класу який-небудь метод, може його перевизначити, задавши власну реалізацію, відмінну від реалізації, яка використовується батьком. Перевизначений батьківський метод повинен забезпечуватися модифікатором `override`.

- Якщо прямий батько не заданий, то таким є клас `object`.

- Якщо батьківський метод не є віртуальним або абстрактним, то нащадок може використати ім'я батьківського методу для створення свого власного, що називається приховуванням батьківського методу. Для викликів батьківського методу використовується ключове слово `base`.

2.2. Лабораторне заняття Л2. Введення в проектування програмного забезпечення з використанням шаблонів. Породжувальний паттерн Абстрактна Фабрика (Abstract Factory).

Мета лабораторного заняття: ознайомитися з різновидами породжувальних паттернів. Вивчити діаграму класів паттерна Абстрактна фабрика, способи та мету його застосування. Навчитися використовувати цей шаблон при розробці програмного забезпечення в рамках обраної предметної області.

Приклад завдання 1. У заданій предметній області реалізувати шаблон Абстрактна Фабрика для створення об'єктів сімейств класів.

Розв'язання:

Описання предметної області. Нехай в предметній області створені абстракції „Марка виробника автомобілів та запчастин”, „Двигун” та „Автомобіль”. Марки можуть бути „BMW” та „Audi”, що спеціалізуються на створенні як двигунів, так і автомобілів в цілому. Необхідно застосувати паттерн Абстрактна фабрика для створення екземплярів сімейств класів. Для цього:

1. Опишемо класи предметної області:

1.1. Реалізуємо абстрактний клас для автомобілів. В даному випадку у них буде один метод, що дозволяє дізнатися максимальну швидкість машини. З його допомогою ми звернемося і до другого об'єкту – двигуна.

1.2. Всі двигуни, у свою чергу, будуть містити один параметр – максимальну швидкість. Ця проста загальнодоступна змінна дозволить скоротити обсяг програми в даному прикладі.

1.3. Реалізуємо класи для автомобіля марки BMW та автомобіля марки Audi, що є нащадками абстрактного класу автомобілів.

1.4. Реалізуємо класи для двигуна марки BMW та двигуна марки Audi, що є нащадками абстрактного класу двигунів.

2. Реалізуємо складові частини паттерна Абстрактна Фабрика:

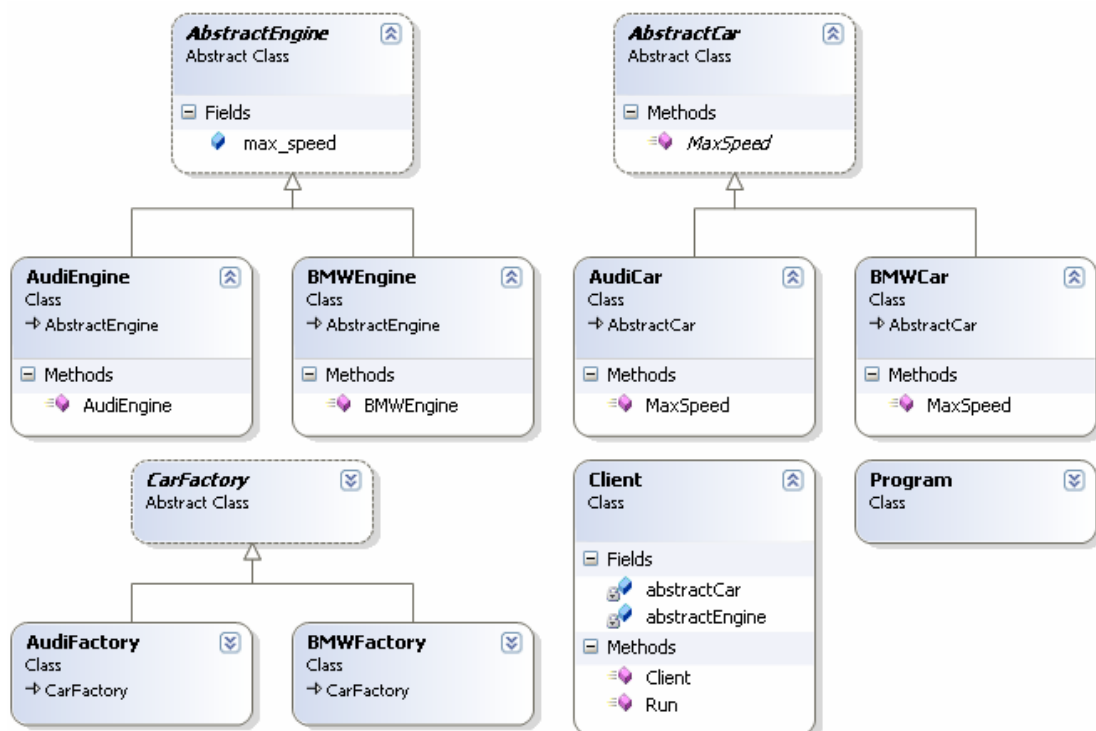
2.1 Створимо абстрактну фабрику CarFactory, що містить сімейство з двох об'єктів – автомобіля і двигуна для нього. В результаті у нас з'явився абстрактний клас із двома методами, що забезпечують отримання відповідних абстрактних об'єктів.

2.2. Реалізуємо першу конкретну фабрику, яка створює клас, що описує автомобіль BMW і двигун для нього та є нащадком абстрактної фабрики.

2.3. Реалізуємо другу конкретну фабрику, яка створює клас, що описує автомобіль Audi і двигун для нього та є нащадком абстрактної фабрики.

3. Реалізовуємо клас-клієнт де покажемо, як здійснюється робота з Абстрактною Фабрикою. У конструктор класу-клієнта будуть передаватися всі конкретні фабрики, які і почнуть створювати об'єкти „Автомобіль” і „Двигун”. Отже, в конструктор класу Client допустимо передати будь-яку конкретну фабрику, що працює з будь-якими марками автомобілів. А метод Run дозволить дізнатися максимальну швидкість конкретної машини.

Діаграма класів предметної області



Лістинг створеного програмного забезпечення (C#)

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace lab2_1
{
    abstract class CarFactory
    {
        public abstract AbstractCar CreateCar();
        public abstract AbstractEngine CreateEngine();
    }
    abstract class AbstractCar
    {
    }
    public abstract void MaxSpeed(AbstractEngine engine);
    }
    abstract class AbstractEngine
    {
        public int max_speed;
    }
    class BMWFactory : CarFactory
    {
        public override AbstractCar CreateCar()
        {
            return new BMWCar();
        }
        public override AbstractEngine CreateEngine()
        {
            return new BMWEngine();
        }
    }
    class AudiFactory : CarFactory
    {
        public override AbstractCar CreateCar()
        {
            return new AudiCar();
        }
        public override AbstractEngine CreateEngine()
        {
            return new AudiEngine();
        }
    }
    class BMWCar : AbstractCar
    {
    }
    public override void MaxSpeed(AbstractEngine engine)
    {
        Console.WriteLine("Максимальная скорость:" + engine.
max_speed.ToString());
    }
    }
    class BMWEngine : AbstractEngine
    {

```

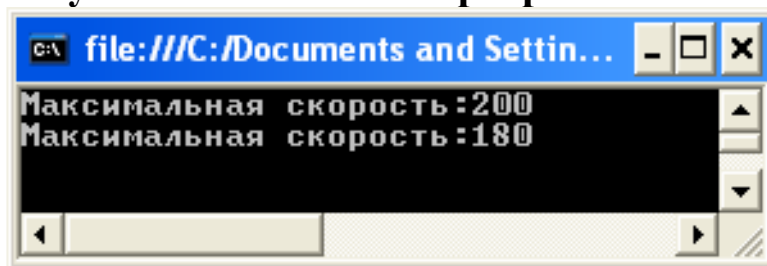
```

        public BMWEngine()
        {
            max_speed = 200;
        }
    }
    class AudiCar : AbstractCar
    {
    public override void MaxSpeed(AbstractEngine engine)
    {
        Console.WriteLine("Максимальная скорость:" + engine.
max_speed.ToString());
    }
    }

    class AudiEngine : AbstractEngine
    {
        public AudiEngine()
        {
            max_speed = 180;
        }
    }
    class Client
    {
        private AbstractCar abstractCar;
        private AbstractEngine abstractEngine;
    public Client(CarFactory car_factory)
        {
            abstractCar = car_factory.CreateCar();
            abstractEngine = car_factory.CreateEngine ();
        }
    public void Run()
        {
            abstractCar.MaxSpeed(abstractEngine);
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            // Абстрактная фабрика № 1
            CarFactory bmw_car = new BMWFactory();
            Client c1 = new Client(bmw_car);
            c1.Run();
            // Абстрактная фабрика № 2
            CarFactory audi_factory = new AudiFactory();
            Client c2 = new Client(audi_factory);
            c2.Run();
            Console.Read();
        }
    }
}

```

Результат виконання програми



Висновки по лабораторному заняттю 2:

- Паттерн Абстрактна Фабрика – породжувальний паттерн, що надає інтерфейс для створення сімейств, зв'язаних між собою класів.

- Учасниками паттерна є: AbstractFactory (CarFactory) – абстрактна фабрика – клас, який задає інтерфейс для операцій, що створюють абстрактні об'єкти-продукти; ConcreteFactory (BMWFactory та AudiFactory) – конкретна фабрика – класи, які реалізують операції, що створюють конкретні об'єкти-продукти; AbstractProduct (AbstractCar, AbstractEngine) – абстрактний продукт – класи, які задають інтерфейс для типу об'єкта-продукту; ConcreteProduct (BMWCar, BMWEngine, AudiCar, AudiEngine) – конкретний продукт; Client – клієнт.

- Перевага даного паттерна полягає в тому, що він ізолює конкретні класи, завдяки чому легко замінити сімейства продуктів.

- До недоліків паттерна Абстрактна Фабрика варто віднести те, що при розширенні можливостей фабрики шляхом додавання нового типу продуктів прийдеться редагувати всі конкретні реалізації Abstract Factory, а це часом буває неприпустимо, наприклад, якщо вже створено 100 конкретних фабрик.

2.3. Лабораторне заняття Л3. Приклади використання породжувального паттерна Фабричний Метод (Factory Method).

Мета лабораторного заняття: поглибити знання стосовно породжувальних паттернів. Вивчити діаграму класів паттерна

Фабричний Метод, мету, способи та випадки доцільного використання. Навчитися використовувати паттерн Фабричний Метод для створення екземплярів класів в обраній предметній області.

Приклад завдання 1. Застосувати паттерн Фабричний метод у заданій предметній області.

Розв'язання.

Описання предметної області. Нехай необхідно створити абстракцію „Товари”, які можуть бути „CD-плеєрами” і „Комп'ютерами” та описуються властивостями „Назва”, „Ціна закупки” та „Ціна продажу”. Необхідно застосувати паттерн Фабричний Метод для створення екземплярів цих класів. Для цього:

1. Опишемо класи предметної області:

1.1. Реалізуємо абстрактний клас, що задає інтерфейс для товарів. Містить властивості „Назва”, „Ціна закупки”, „Ціна продажу”.

1.2. Реалізуємо два нащадки абстрактного класу, для описання „CD-плеєрів” і „Комп'ютерів”.

2. Реалізуємо складові частини паттерна Фабричний Метод:

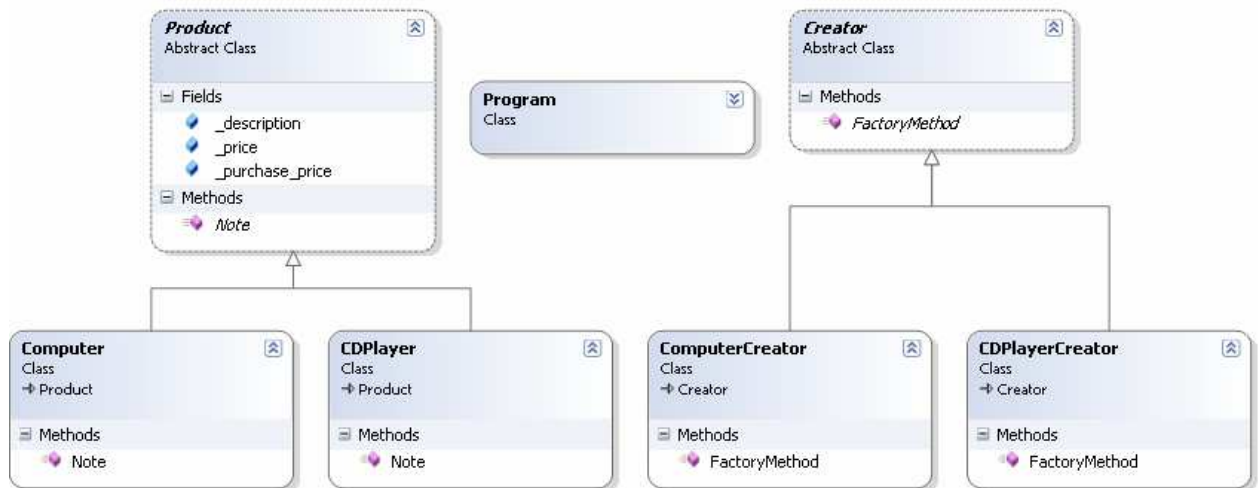
2.1. Дотримуючись шаблону, реалізуємо абстрактний клас, в якому і буде фабричний метод. Це як би фабрика, яка створює продукти конкретного виду. Коли абстрактний клас Creator з фабричним методом визначено, ми можемо створити свій успадкований клас для конкретного продукту.

2.2. Створюємо конкретний клас-творець конкретного продукту „CD-плеєрів” (успадкований від Creator).

2.3. Створюємо конкретний клас-творець конкретного продукту „Комп'ютерів” (успадкований від Creator).

3. В якості класу-клієнта використовуємо клас Program, та в його методі Main демонструємо використання паттерна Фабричний Метод.

Діаграма класів предметної області



Лістинг створеного програмного забезпечення (C#)

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace lab3_3
{
    abstract class Product
    {
        public decimal _purchase_price;
        public decimal _price;
        public string _description;
        public abstract void Note();
    }
    class Computer : Product
    {
        public override void Note()
        {
            Console.WriteLine("Переопределение родительского метода потомком Компьютер");
        }
    }
    abstract class Creator
    {
        public abstract Product FactoryMethod();
    }
    class ComputerCreator : Creator
    {
        public override Product FactoryMethod()
        {
            return new Computer();
        }
    }
    class CDPlayer : Product
    {

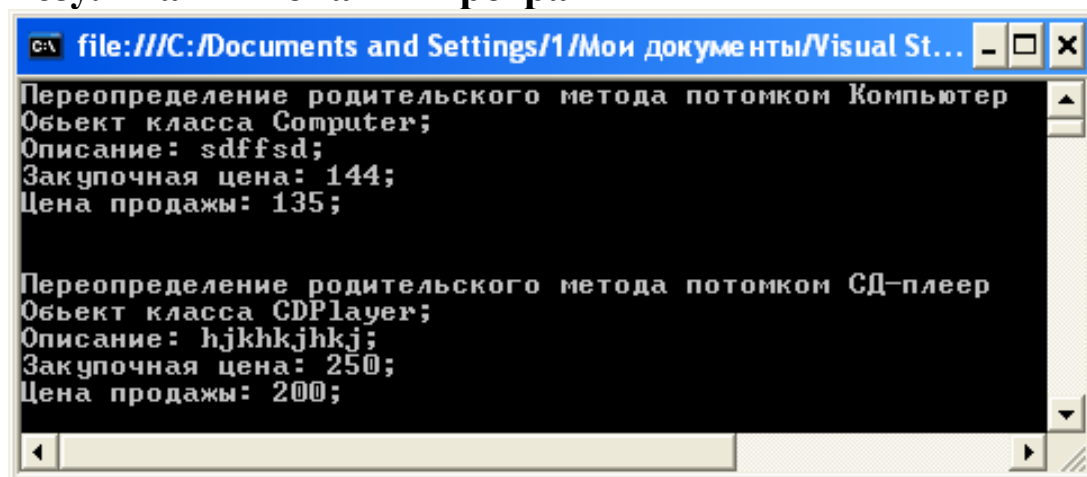
```

```

        public override void Note()
        {
            Console.WriteLine("Переопределение родительского
метода потомком СД-плеер");
        }
    }
    class CDPlayerCreator : Creator
    {
        public override Product FactoryMethod()
        {
            return new CDPlayer();
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Creator cr1=new ComputerCreator();
            Creator cr2=new CDPlayerCreator();
            Product pr = cr1.FactoryMethod();
            pr._description="sdffsd";
            pr._price=135;
            pr._purchase_price=144;
            Product pr2 = cr2.FactoryMethod();
            pr2._description = "hjkjhkhkj";
            pr2._price = 200;
            pr2._purchase_price = 250;
            pr.Note();
            Console.WriteLine("Объект класса {0};\n" +
                "Описание: {1};\n" +
                "Закупочная цена: {2};\n" +
                "Цена продажи: {3};\n",
                pr.GetType().Name,
                pr._description,
                pr._purchase_price,
                pr._price);
            Console.ReadLine();
            pr2.Note();
            Console.WriteLine("Объект класса {0};\n" +
                "Описание: {1};\n" +
                "Закупочная цена: {2};\n" +
                "Цена продажи: {3};\n",
                pr2.GetType().Name,
                pr2._description,
                pr2._purchase_price,
                pr2._price);
            Console.ReadLine();
        }
    }
}

```


Результат виконання програми



```

Переопределение родительского метода потомком Компьютер
Объект класса Computer;
Описание: sdfsd;
Закупочная цена: 144;
Цена продажи: 135;

Переопределение родительского метода потомком CD-плеер
Объект класса CDPlayer;
Описание: hjkhkjkhkj;
Закупочная цена: 250;
Цена продажи: 200;
  
```

Висновки по лабораторному заняттю 3:

- Паттерн Фабричний метод – породжувальний паттерн, що дає можливість підкласам створювати деякі класи за допомогою загального інтерфейсу. Причому саме нащадки визначають який батьківський об'єкт треба реалізувати, паттерн же призначений для того, щоб делегувати їм такі повноваження. Завдяки цьому в тексті програми можна оперувати не якимиись конкретними класами, а їхніми абстрактними представленнями.

- Учасниками паттерна є: Product (Product) – продукт; ConcreteProduct (Computer, CDPlayer) – конкретний продукт; Creator (Creator) – творець; ConcreteCreator (ComputerCreator, CDPlayerCreator).

- До достоїнств паттерна варто віднести можливість створювати об'єкти більш універсально, не орієнтуючись на конкретні класи й оперуючи загальним інтерфейсом.

- Самий очевидний недолік паттерна – необхідність створювати нащадка Creator завжди, коли планується одержати новий тип продукту (тобто новий ConcreteProduct).

2.4. Лабораторне заняття Л4. Розв'язання задач із використанням структурного паттерна Адаптер (Adapter).

Мета лабораторного заняття: отримати загальні відомості

стосовно структурних паттернів. Вивчити діаграму класів, мету, способи та випадки доцільного використання паттерна Адаптер. Навчитися використовувати шаблон у заданій предметній області.

Приклад завдання 1. Застосувати паттерн Адаптер у заданій предметній області.

Розв’язання.

Описання предметної області. Нехай в предметній області існує абстракція, що задає інтерфейс для різних видів качок, з методами, які описують процес польоту і процес видачі звуків. Нащадки цієї абстракції реалізують заданий інтерфейс. У програмі необхідно забезпечити можливість привести інтерфейс класу, що описує види півнів, до інтерфейсу класу качок. Приведення інтерфейсу необхідно здійснити засобами паттерна Адаптер. Для цього:

1. Опишемо класи предметної області:

1.1. Реалізуємо інтерфейс для видів качок, що містить 2 методи: польоту та крякання.

1.2. Реалізуємо нащадка описаного інтерфейсу.

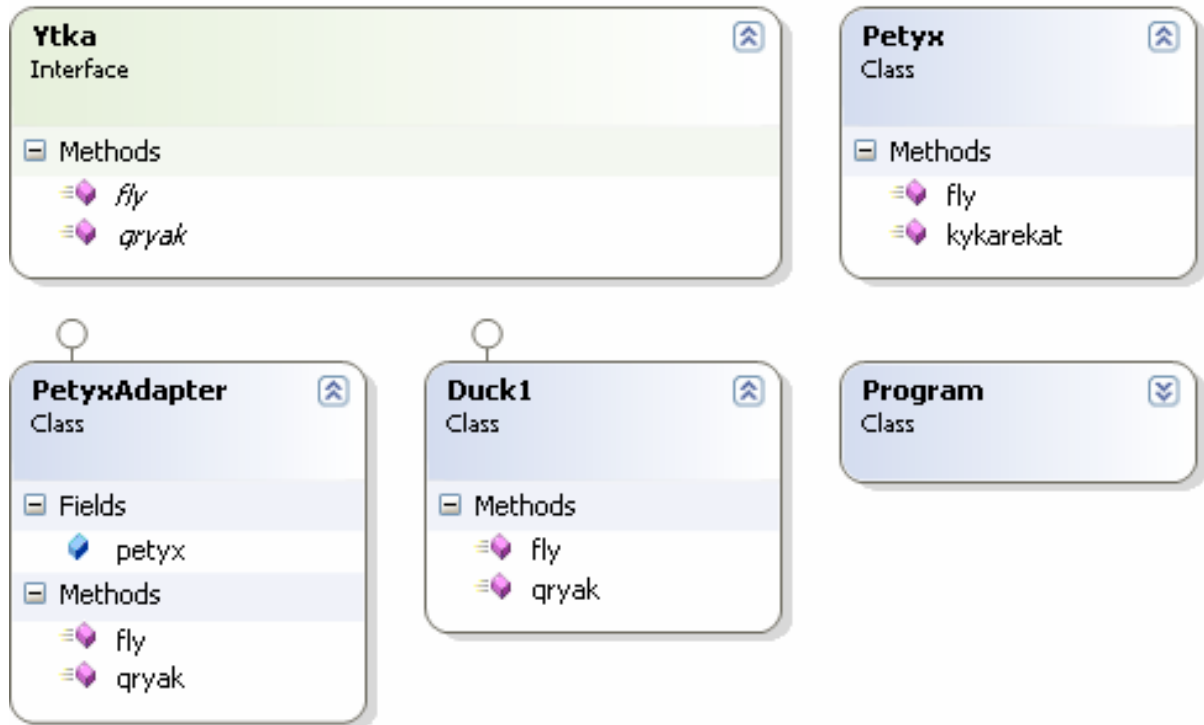
1.3. Створюємо клас видів півнів, що містить 2 методи: польоту та кукурікання.

2. Реалізуємо складові частини паттерна Адаптер:

2.1. Дотримуючись шаблону, реалізуємо клас-адаптер, який з одного боку є нащадком інтерфейсу видів качок, а з іншого – клієнтом класу видів півнів.

3. В якості класу-клієнта використовуємо клас Program, та в його методі Main демонструємо використання паттерна Адаптер.

Діаграма класів предметної області



Лістинг створеного програмного забезпечення (C#)

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace lab4_2
{
    //Target
    public interface Ytka
    {
        void qryak();
        void fly();
    }
    public class Duck1 : Ytka
    {
        public void qryak()
        {
            Console.WriteLine("Крякает");
        }
        public void fly()
        {
            Console.WriteLine("Утка летит");
        }
    }
    //Adaptee
    public class Petyx

```

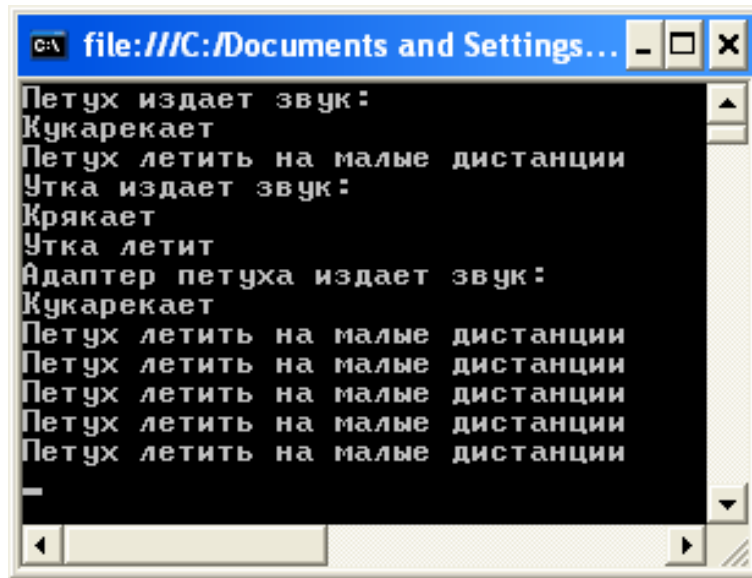
```

{
    public void kykarekat()
    {
        Console.WriteLine("Кукарекает");
    }
    public void fly()
    {
        Console.WriteLine("Петух летить на малые
дистанции");
    }
}
//Adapter
public class PetyxAdapter : Ytka
{
    public Petyx petyx=new Petyx() ;

    public void qryak()
    {
        petyx.kykarekat();
    }
    public void fly()
    {
        for (int i = 0; i < 5; i++)
        {
            petyx.fly();
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
        Duck1 duck1 = new Duck1();
        Petyx petyx = new Petyx();
        Ytka petyxadapter = new PetyxAdapter();
        Console.WriteLine("Петух издает звук:");
        petyx.kykarekat();
        petyx.fly();
        Console.WriteLine("Утка издает звук:");
        duck1.qryak();
        duck1.fly();
        Console.WriteLine("Адаптер петуха издает звук:");
        petyxadapter.qryak();
        petyxadapter.fly();
        Console.ReadKey();
    }
}

```

Результат виконання програми



```

file:///C:/Documents and Settings...
Петух издает звук :
Кукарекает
Петух летить на малые дистанции
Утка издает звук :
Крякает
Утка летит
Адаптер петуха издает звук :
Кукарекает
Петух летить на малые дистанции
Петух летить на малые дистанции
Петух летить на малые дистанции
Петух летить на малые дистанции
Петух летить на малые дистанции

```

Висновки по лабораторному заняттю 4:

- Паттерн Адаптер – структурний паттерн, що перетворює інтерфейс одного класу в інтерфейс іншого, який очікують клієнти. Адаптер забезпечує спільну роботу класів з несумісними інтерфейсами, яка без нього була б неможлива.

- Учасники паттерна: Target (Ytka) – цільовий, що визначає залежний від предметної області інтерфейс, яким користується Client; Client (Program) – клієнт, що вступає у взаємини з об'єктами, які задовольняють інтерфейсу Target; Adaptee (Petyx) – клас, що адаптується, визначає існуючий інтерфейс, що має потребу в адаптації; Adapter (PetyxAdapter) – адаптер, що адаптує інтерфейс Adaptee до інтерфейсу Target.

- Достоїнства паттерна Адаптер: Паттерн Адаптер дозволяє повторно використовувати вже існуючий код, адаптуючи його несумісний інтерфейс до виду, придатному для використання.

- Недоліки паттерна Адаптер: Задача перетворення інтерфейсів може виявитися непростю у випадку, якщо клієнтські виклики і (або) передані параметри не мають функціональної відповідності в об'єкті, що адаптується.

2.5. Лабораторне заняття Л5. Побудова програмних додатків із використанням структурного паттерна Фасад (Facade).

Мета лабораторного заняття: закріпити знання стосовно структурних паттернів. Вивчити діаграму класів, мету, способи та випадки доцільного використання паттерна Фасад. Навчитися використовувати шаблон у заданій предметній області.

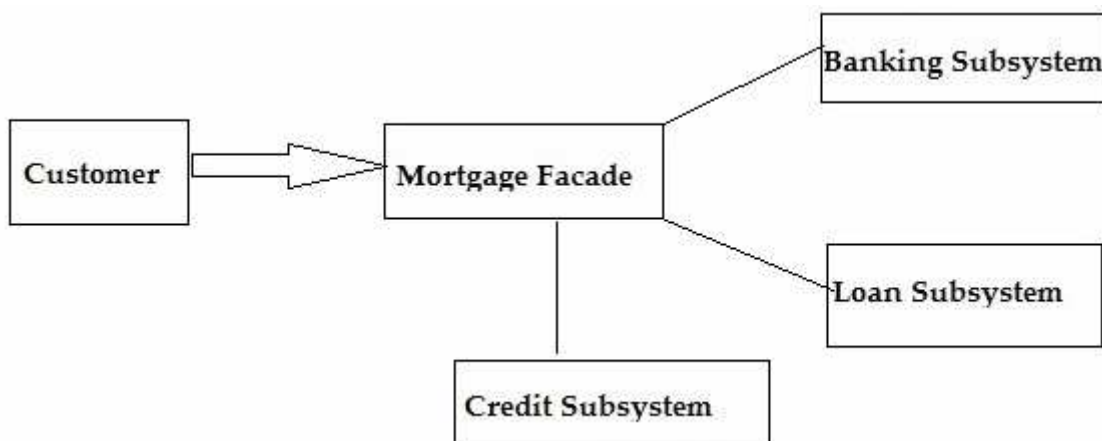
Приклад завдання 1. Застосувати паттерн Фасад у заданій предметній області.

Розв’язання.

Описання предметної області. Нехай у предметній області виділяється абстракція „Клієнт банку”, який хоче взяти іпотечний кредит на будинок. Клієнт описується властивостями „Прізвище”, „Посада”. В банку є кредитний відділ, який надає іпотечні кредити. Клієнт взаємодіє з представником кредитного відділу. Кредитний відділ має три підрозділи:

1. Банківська система, яка визначає чи платоспроможний клієнт для кредиту, що видається;
2. Кредитний сектор, в якому визначають чи має клієнт непогашений або прострочений кредити (стежить за кредитною історією);
- 3) Розрахунковий відділ – розраховує суму кредиту.

Кредитний відділ взаємодіє з банківськими підсистемами для визначення відповіді клієнту за схемою.



Описати взаємодію клієнта банку з кредитним відділом в рамках паттерна Фасад. Для цього:

1. Опишемо класи предметної області:

1.1. Реалізуємо клас для банківської системи з методом, що перевіряє клієнта на платоспроможність;

1.2. Реалізуємо клас кредитного сектору з методом, що перевіряє клієнта по кредитній історії;

1.3. Реалізуємо клас розрахункового відділу з методом, що розраховує суму кредиту;

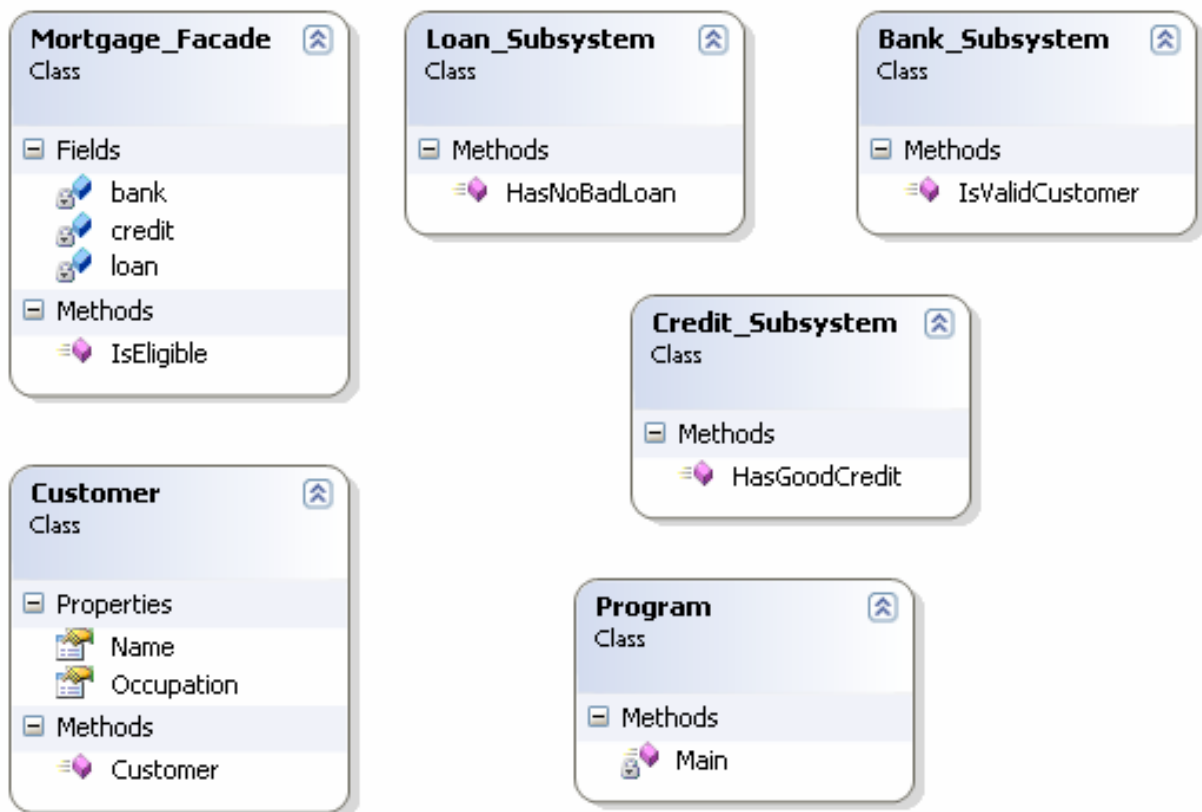
1.4. Реалізуємо клас клієнта з властивостями „Ім'я” та „Посада”.

2. Реалізуємо складові частини паттерна Фасад:

2.1. Дотримуючись шаблону, реалізуємо клас-фасад, який є клієнтом класів підсистем кредитного відділу.

3. В якості класу-клієнта використовуємо клас Program, та в його методі Main демонструємо використання паттерна Фасад.

Діаграма класів предметної області



Лістинг створеного програмного забезпечення (C#)

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace kredit
{
    public class Bank_Subsystem
    {
        public bool IsValidCustomer(Customer cust)
        {
            Console.WriteLine("Проверка " + cust.Name + " на
платежеспособность");
            return true;
        }
    }

    public class Credit_Subsystem
    {
        public bool HasGoodCredit(Customer customer)
        {
            Console.WriteLine("Расчет для" + customer.Name + "
суммы кредита");
            return true;
        }
    }

    public class Customer
    {
        public string Name { get; set; }
        public string Occupation { get; set; }
        public Customer(string name, string occupation)
        {
            Name = name;
            Occupation = occupation;
        }
    }

    public class Loan_Subsystem
    {
        public bool HasNoBadLoan(Customer cust)
        {
            Console.WriteLine("Проверка" + cust.Name + " по
кредитной истории");
            return true;
        }
    }

    public class Mortgage_Facade
    {
        private Bank_Subsystem bank;
        private Loan_Subsystem loan;
        private Credit_Subsystem credit;
        public bool IsEligible(Customer cust)
        {

```



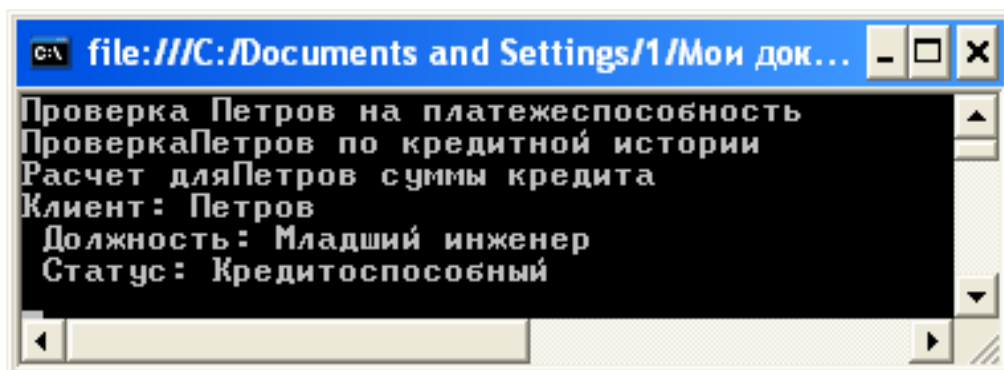
```

        bool isEligible = true;
        bank = new Bank_Subsystem();
        if (!bank.IsValidCustomer(cust))
        {
            isEligible = false;
        }
        loan = new Loan_Subsystem();
        if (isEligible && !loan.HasNoBadLoan(cust))
        {
            isEligible = false;
        }
        credit = new Credit_Subsystem();
        if (isEligible && !credit.HasGoodCredit(cust))
        {
            isEligible = false;
        }
        return isEligible;
    }
}

class Program
{
    static void Main(string[] args)
    {
        Customer cust = new Customer("Петров", "Младший инженер");
        Mortgage_Facade mortgage = new Mortgage_Facade();
        if (mortgage.IsEligible(cust))
        {
            Console.WriteLine(string.Format("Клиент: {0} \n Должность: {1} \n Статус: Кредитоспособный", cust.Name, cust.Occupation));
        }
        else
        {
            Console.WriteLine(string.Format("Клиент: {0} \n Должность: {1} \n Статус: Некредитоспособный", cust.Name, cust.Occupation));
        }
        Console.Read();
    }
}

```

Результат виконання програми



Висновки по лабораторному заняттю 5:

- Паттерн Фасад – структурний паттерн, що надає уніфікований інтерфейс замість набору інтерфейсів деякої підсистеми. Фасад визначає інтерфейс більш високого рівня, що спрощує використання підсистеми.

- Учасниками паттерна Фасад є: класи підсистеми (Bank_Subsystem, Credit_Subsystem, Loan_Subsystem), що реалізують функціональність підсистеми, виконують роботу, доручену об'єктом Facade та при цьому не зберігають посилань на нього; Facade (Mortgage_Facade) – фасад, що знає, яким класам підсистеми адресувати запит та делегує запити клієнтів підходящим об'єктам усередині підсистеми.

- Переваги використання паттерна Фасад: ізолює клієнтів від компонентів підсистеми, зменшуючи тим самим число об'єктів, з якими клієнтам приходится мати справу, і спрощуючи роботу з підсистемою; дозволяє послабити зв'язаність між підсистемою і її клієнтами. Найчастіше компоненти підсистеми сильно зв'язані. Слабка зв'язаність дозволяє видозмінювати компоненти, не торкаючись при цьому клієнтів. Фасади допомагають розкласти систему на шари і структурувати залежності між об'єктами, а також уникнути складних і циклічних залежностей.

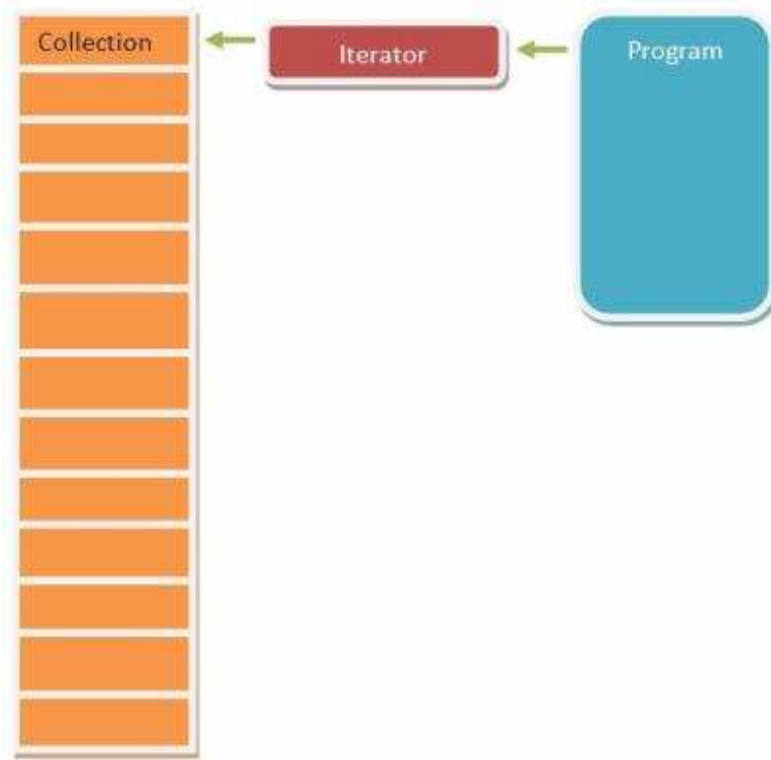
2.6. Лабораторне заняття Л6. Загальні принципи побудови програмних додатків із використанням шаблонів поведінки. Паттерн Ітератор (Iterator).

Мета лабораторного заняття: Отримати загальні відомості стосовно паттернів поведінки. Вивчити діаграму класів, мету, способи та випадки доцільного використання паттерна Ітератор. Навчитися використовувати шаблон у заданій предметній області.

Приклад завдання 1. Застосувати паттерн Ітератор у заданій предметній області.

Розв'язання.

Описання предметної області. Нехай послідовність чисел від 1 до 10 зберігається в колекції даних – список. Навести приклад використання паттерна Ітератор для надання способів доступу до елементів заданої колекції (рисунок).



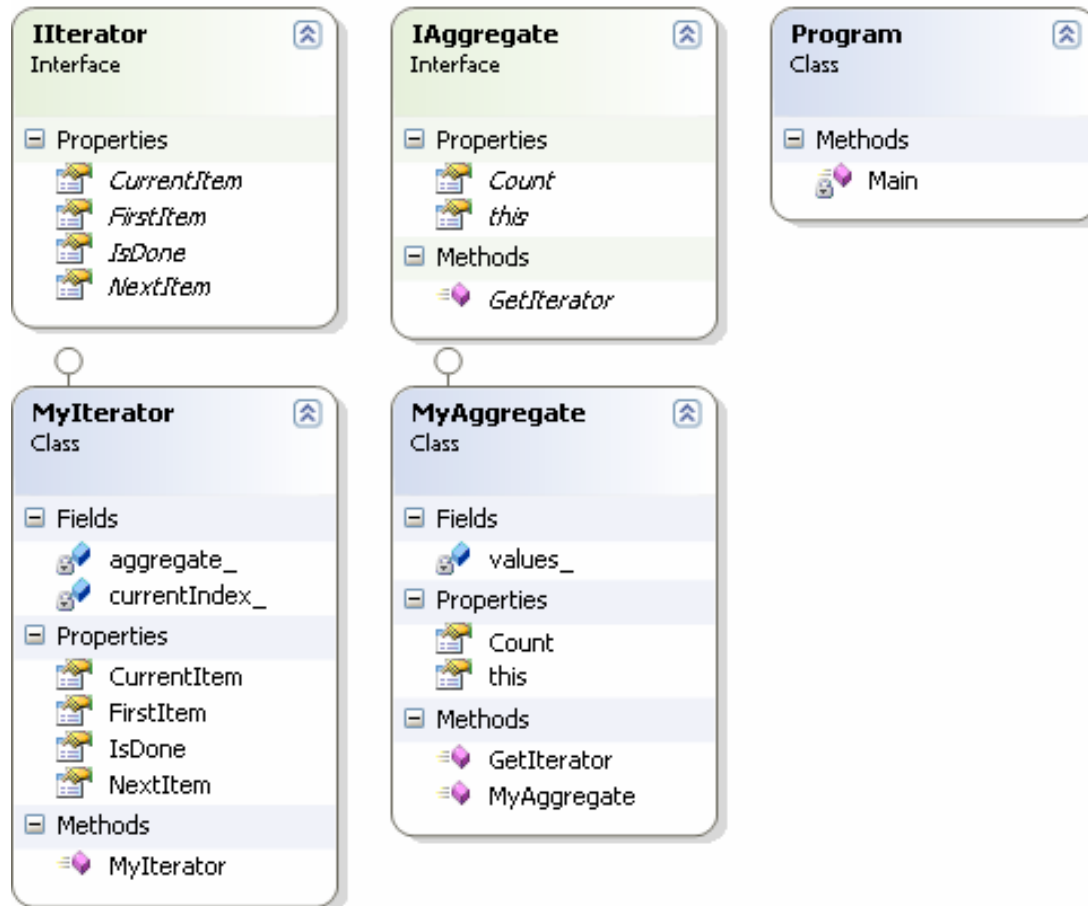
Для цього:

1. Опишемо класи, що задають колекцію даних:
 - 1.1. Реалізуємо абстрактний клас агрегату, що буде задавати інтерфейс для колекції даних;
 - 1.2. Реалізуємо клас конкретного агрегату, в якому опишемо колекцію даних – типу „список”;
2. Реалізуємо складові частини паттерна Ітератор:
 - 2.1. Дотримуючись шаблону, реалізуємо абстрактний клас Ітератор, що задасть інтерфейс для надання способів обходу елементів колекції;
 - 2.2. Реалізуємо клас конкретного Ітератора, що реалізує методи обходу елементів заданої колекції: повернути перший,

повернути наступний, перевірити на порожність, повернути поточний елемент.

3. В якості класу-клієнта використовуємо клас Program, та в його методі Main демонструємо використання паттерна Ітератор.

Діаграма класів предметної області



Лістинг створеного програмного забезпечення (C#)

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace coll2
{
    interface IIterator
    {
        string FirstItem { get; }
        string NextItem { get; }
        string CurrentItem { get; }
        bool IsDone { get; }
    }
    interface IAggregate
    {

```

```

        IEnumerator GetIterator();
        string this[int itemIndex] { set; get; }
        int Count { get; }
    }
    class MyAggregate : IAggregate
    {
        List<string> values_ = null;

        public MyAggregate()
        {
            values_ = new List<string>();
        }
        public IEnumerator GetIterator()
        {
            return new MyIterator(this);
        }
        public string this[int itemIndex]
        {
            get
            {
                if (itemIndex < values_.Count)
                {
                    return values_[itemIndex];
                }
                else
                {
                    return string.Empty;
                }
            }
            set
            {
                values_.Add(value);
            }
        }
        public int Count
        {
            get
            {
                return values_.Count;
            }
        }
    }
    class MyIterator : IEnumerator
    {
        IAggregate aggregate_ = null;
        int currentIndex_ = 0;
        public MyIterator(IAggregate aggregate)
        {
            aggregate_ = aggregate;
        }
        public string FirstItem
        {
            get

```

```

        {
            currentIndex_ = 0;
            return aggregate_[currentIndex_];
        }
    }
    public string NextItem
    {
        get
        {
            currentIndex_ += 1;

            if (IsDone == false)
            {
                return aggregate_[currentIndex_];
            }
            else
            {
                return string.Empty;
            }
        }
    }
    public string CurrentItem
    {
        get
        {
            return aggregate_[currentIndex_];
        }
    }
    public bool IsDone
    {
        get
        {
            if (currentIndex_ < aggregate_.Count)
            {
                return false;
            }
            return true;
        }
    }
}
class Program
{
    static void Main(string[] args)
    {
        MyAggregate aggr = new MyAggregate();
        aggr[0] = "1";
        aggr[1] = "2";
        aggr[2] = "3";
        aggr[3] = "4";
        aggr[4] = "5";
        aggr[5] = "6";
        aggr[6] = "7";
        aggr[7] = "8";
    }
}

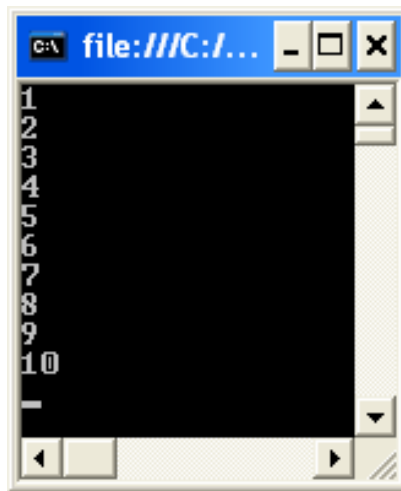
```

```

        aggr[8] = "9";
        aggr[9] = "10";
        IIterator iter = aggr.GetIterator();
        for (string s = iter.FirstItem;
            iter.IsDone == false; s = iter.NextItem)
        {
            Console.WriteLine(s);
        }
        Console.ReadKey();
    }
}

```

Результат виконання програми



Висновки по лабораторному заняттю 6:

- Паттерн Ітератор – паттерн поведінки, що надає спосіб послідовного доступу до всіх елементів складеного об'єкта, не розкриваючи його внутрішнього представлення.

- Учасники паттерна: *Iterator* (*IIterator*) – ітератор – клас, що визначає інтерфейс для доступу й обходу елементів; *ConcreteIterator* (*MyIterator*) – конкретний ітератор – клас, що реалізує інтерфейс класу *Iterator* та стежить за поточною позицією при обході агрегату; *Aggregate* (*IAggregate*) – агрегат – клас, що визначає інтерфейс для створення об'єкта-ітератора та описує колекцію даних; *ConcreteAggregate* (*MyAggregate*) – конкретний агрегат – клас, що реалізує інтерфейс створення ітератора і повертає екземпляр підходящого класу *ConcreteIterator*.

- Паттерн Ітератор підтримує різні види обходу агрегату. Складні агрегати можна проходити по-різному. Наприклад, для

генерації коду і семантичних перевірок потрібно обходити дерева синтаксичного розбору. Генератор коду може проходити дерево у внутрішньому або прямому порядку. Ітератори спрощують зміну алгоритму обходу – досить просто замінити один екземпляр ітератора іншим. Для підтримки нових видів обходу можна визначити підкласи класу `Iterator`.

- Ітератори спрощують інтерфейс класу `Aggregate`. Наявність інтерфейсу для обходу в класі `Iterator` робить зайвим дублювання цього інтерфейсу в класі `Aggregate`. Тим самим інтерфейс агрегату спрощується при цьому одночасно дозволяється здійснювати кілька обходів агрегату.

2.7. Лабораторне заняття Л7. Розробка програмних додатків із використанням паттерна поведінки Спостерігач (`Observer`).

Мета лабораторного заняття: Вивчити діаграму класів, мету, способи та випадки доцільного використання паттерна Спостерігач. Навчитися використовувати шаблон у заданій предметній області.

Приклад завдання 1. Застосувати паттерн Спостерігач у заданій предметній області.

Розв’язання.

Описання предметної області. Нехай необхідно створити програмне забезпечення, що дає змогу стежити за зміною ціни на товар у магазинах. У рамках поставленої задачі найдоцільніше використовувати паттерн Спостерігач. Для цього:

1. Опишемо класи, що задають предметну область:

1.1. Реалізуємо абстрактний клас суб’єкту, що буде задавати інтерфейс для товарів у магазині, в який вбудуємо складові частини паттерна Спостерігач, а саме: методи додавання, видалення та оповіщення всіх спостерігачів. Крім того, створимо

On-процедуру, що складається з виклику оголошеної події, включеної у тест.

1.2. Реалізуємо клас конкретного суб'єкту, в якому опишемо метод зміни ціни на товар.

2. Реалізуємо складові частини паттерна Спостерігач:

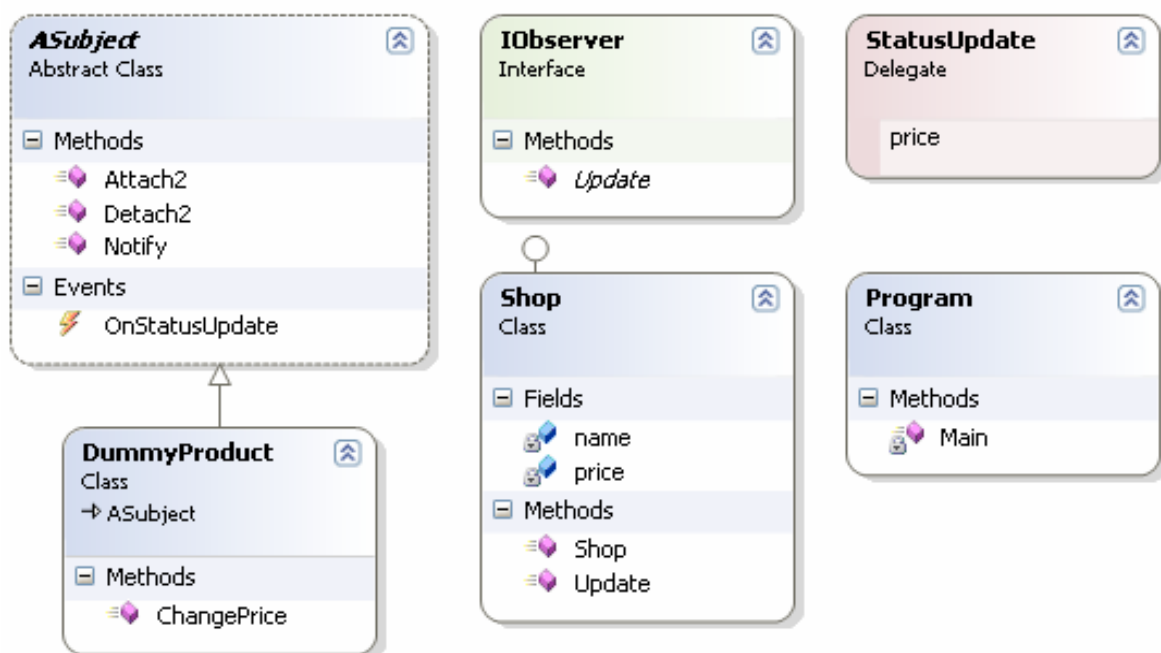
2.1. Дотримуючись шаблону, реалізуємо абстрактний клас Спостерігача, що задасть інтерфейс для оновлення стану всіх спостерігачів.

2.2. Реалізуємо клас конкретного Спостерігача. В рамках описаної предметної області – це магазин, що зберігає назви і ціну товарів. При зміні ціни товару, магазин оповіщається про цю подію і має змогу змінити і свій стан відповідно.

3. Описуємо делегат, що задає сигнатуру методу зміни статусу.

4. В якості класу-клієнта використовуємо клас Program, та в його методі Main демонструємо використання паттерна Спостерігач. Створюємо екземпляри класів суб'єкта та спостерігачів, додаємо створені екземпляри спостерігачів до списку спостерігачів суб'єкта, моделюємо процес зміни ціни на товар.

Діаграма класів предметної області



Лістинг створеного програмного забезпечення (C#)

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace @delegate
{
    public delegate void StatusUpdate(float price);
    abstract class ASubject
    {
        public event StatusUpdate OnStatusUpdate;
        public void Attach2(Shop product)
        {
            // lets assign attach the observers with subjects
            OnStatusUpdate += new StatusUpdate(product.Update);
        }
        public void Detach2(Shop product)
        {
            // lets assign detach the observers with subjects
            OnStatusUpdate -= new StatusUpdate(product.Update);
        }
        public void Notify(float price)
        {
            // lets notify the observers with change
            if (OnStatusUpdate != null)
            {
                OnStatusUpdate(price);
            }
        }
    }
    class DummyProduct : ASubject
    {
        public void ChangePrice(float price)
        {
            Notify(price);
        }
    }
    interface IObservable
    {
        void Update(float price);
    }
    class Shop : IObservable
    {
        //Name of the product
        string name;
        float price = 0.0f; //default
        public Shop(string name)
        {
            this.name = name;
        }
        public void Update(float price)
        {

```

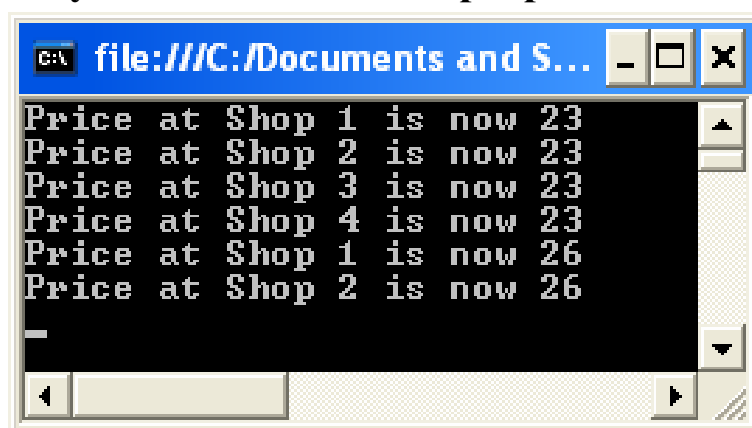
```

        this.price = price;
        //Lets print on console just to test the working
        Console.WriteLine(@"Price at {0} is now {1}", name, price);
    }
}

class Program
{
    static void Main(string[] args)
    {
        DummyProduct product = new DummyProduct();
        // We have four shops wanting to keep updated price
        set by product owner
        Shop shop1 = new Shop("Shop 1");
        Shop shop2 = new Shop("Shop 2");
        Shop shop3 = new Shop("Shop 3");
        Shop shop4 = new Shop("Shop 4");
        //Lets use WAY_2 for other two shops
        product.Attach2(shop1);
        product.Attach2(shop2);
        product.Attach2(shop3);
        product.Attach2(shop4);
        //Now lets try chaging the products price, this should update
        the shops automatically
        product.ChangePrice(23.0f);
        product.Detach2(shop3);
        product.Detach2(shop4);
        //Now lets try changing the products price again
        product.ChangePrice(26.0f);
        Console.Read();
    }
}

```

Результат виконання програми



Висновки по лабораторному заняттю 7:

- Паттерн Спостерігач – це паттерн поведінки, що визначає залежність типу „один до багатьох” між об'єктами таким чином,

що при зміні стану одного об'єкта всі залежні від нього об'єкти оповіщаються про це й автоматично оновлюються.

- Учасники паттерна: **Subject (ASubject)** – суб'єкт – клас, що має інформацію про своїх спостерігачів, надає інтерфейс для додавання, видалення та оповіщення спостерігачів; **Observer (IObserver)** – спостерігач – клас, що визначає інтерфейс оновлення стану для об'єктів, що повинні бути сповіщені про зміну суб'єкта; **ConcreteSubject (DummyProduct)** – конкретний суб'єкт – клас, що зберігає стан, який представляє інтерес для конкретного спостерігача, посилає інформацію своїм спостерігачам, коли відбувається зміна стану; **ConcreteObserver (Shop)** – конкретний спостерігач – клас, що зберігає посилання на об'єкт класу **ConcreteSubject**, зберігає дані, що повинні бути погоджені з даними суб'єкта, реалізує інтерфейс оновлення, визначений у класі **Observer**, щоб підтримувати погодженість із суб'єктом.

- Переваги паттерна Спостерігач: на відміну від звичайного запита, що посилається суб'єктом, не потрібно задавати одержувача. Повідомлення автоматично надходить усім підписаним на нього об'єктам. Суб'єктові не потрібна інформація про кількість таких об'єктів, від нього потрібно всього лише повідомити своїх спостерігачів. Тому ми можемо в будь-який час додавати і видаляти спостерігачів. Спостерігач сам вирішує, обробити отримане повідомлення або ігнорувати його.

- Недоліки паттерна Спостерігач. Нечітко визначені або погано підтримувані критерії залежності можуть стати причиною непередбачених оновлень, відстежити які дуже складно. Ця проблема збільшується ще і тим, що простий протокол оновлення не містить ніяких відомостей про те, що саме змінилося в суб'єкті. Без додаткового протоколу, що допомагає з'ясувати характер змін, спостерігачі будуть змушені проробити складну роботу для непрямого одержання такої інформації.

2.8. Лабораторне заняття Л8. Принципи використання паттерна поведінки Стратегія (Strategy), програмна реалізація.

Мета лабораторного заняття: закріпити знання стосовно способів і випадків використання паттернів поведінки. Вивчити діаграму класів паттерна Стратегія, його учасників, мету, способи найдоцільнішого використання. Отримати навички практичного застосування паттерна Стратегія в заданій предметній області.

Приклад завдання 1. Застосувати паттерн Стратегія в заданій предметній області.

Розв'язання.

Описання предметної області. Нехай в предметній області необхідно описати процес вибору поведінки гри по відношенню до гравця. В залежності від рівня вмінь гравця, у грі реалізуються стратегії: новачок, аматор, професіонал. Щоб мати змогу виконати це завдання з використанням паттерна Стратегія необхідно:

1. Описати класи, що задають предметну область:

1.1. Реалізувати клас, що описує гру та містить поле типу гравця та метод, який моделює хід гри.

2. Реалізувати складові частини паттерна Стратегія:

2.1. Дотримуючись шаблону, реалізуємо абстрактний клас Стратегія, що задає інтерфейс для моделювання ходу гри.

2.2. Реалізувати класи конкретних стратегій, в яких описується вибір рівня складності ходу в залежності від умінь гравця.

3. В якості класу-клієнта використовуємо клас Program, та в його методі Main демонструємо використання паттерна Стратегія. Створюємо екземпляри класів гри та стратегій гри, моделюємо хід гри для різних стратегій.

Діаграма класів предметної області



Лістинг створеного програмного забезпечення (C#)

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace chess
{
    //Стратегія
    public interface IComputerPlayer
    {
        void MakeMove();
    }
    //Конкретна стратегія
    public class EasyComputerPlayer : IComputerPlayer
    {
        void IComputerPlayer.MakeMove()
        {
            Console.WriteLine("Computer вибирає легку стратегію
для новачка.");
        }
    }
    public class MediumComputerPlayer : IComputerPlayer
    {
        void IComputerPlayer.MakeMove()
        {
            Console.WriteLine("Computer вибирає стратегію для аматора.");
        }
    }
}

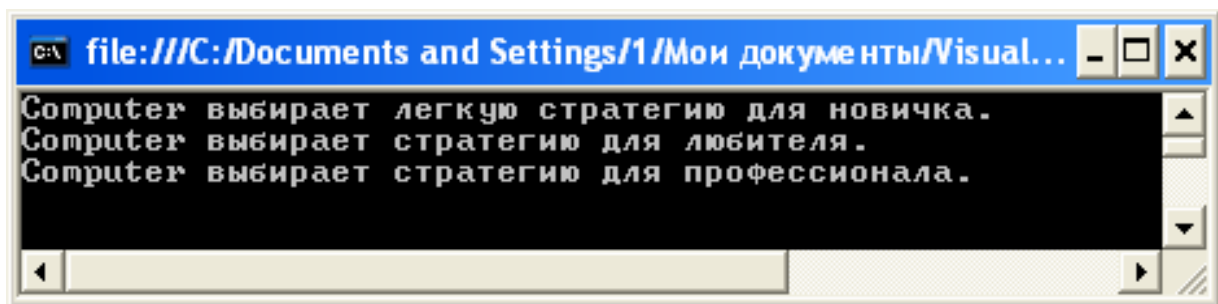
```

```

    }
    public class AdvancedComputerPlayer : IComputerPlayer
    {
        void IComputerPlayer.MakeMove()
        {
            Console.WriteLine("Computer вибирає стратегію для
професіонала.");
        }
    }
}
//Контекст
public class Game
{
    public IComputerPlayer computerPlayer;
    public void Move()
    {
        computerPlayer.MakeMove();    //exhibit the behavior
    }
}
class Program
{
    static void Main(string[] args)
    {
        Game g = new Game();
        g.computerPlayer = new EasyComputerPlayer();
        g.Move();
        g.computerPlayer = new MediumComputerPlayer();
        g.Move();
        g.computerPlayer = new AdvancedComputerPlayer();
        g.Move();
        Console.ReadKey();
    }
}
}

```

Результат виконання програми



Висновки по лабораторному заняттю 8:

- Паттерн Стратегія – це паттерн поведінки, що визначає сімейство алгоритмів, інкапсулює кожний з них і робить їх взаємозамінними. Стратегія дозволяє змінювати алгоритми незалежно від клієнтів, що ними користуються.

- Учасники паттерна: Strategy (IComputerPlayer) – стратегія – клас, що повідомляє загальний для всіх підтримуваних алгоритмів інтерфейс; ConcreteStrategy (EasyComputerPlayer, MediumComputerPlayer, AdvancedComputerPlayer) – класи конкретних стратегій, що реалізують алгоритми; Context (Game) – контекст – клас, що зберігає посилання на об'єкт класу Strategy.

- Достоїнства паттерна Стратегія: Ієрархія класів Strategy визначає сімейство алгоритмів або поведень, які можна повторно використовувати в різних контекстах. Завдяки паттерну вдається відмовитися від умовних операторів при виборі потрібної поведінки. Коли різні алгоритми містяться в одному класі, важко вибрати потрібне без застосування умовних операторів. Інкапсуляція ж кожного з них в окремий клас Strategy, вирішує цю проблему, крім того, це дозволяє змінювати алгоритми незалежно від контексту.

- Потенційний недолік цього паттерна полягає в тому, що для вибору підходящої стратегії клієнт повинен розуміти, чим відрізняються різні стратегії. Тому напевно прийдеться розкрити клієнту деякі особливості реалізації. Звідси виходить, що паттерн Стратегія варто застосовувати лише тоді, коли розходження в поведінці мають значення для клієнта.

3. РЕКОМЕНДОВАНА ЛІТЕРАТУРА

Базова

1. Ванссидес Джон. Применение шаблонов проектирования. Дополнительные штрихи.: Пер. с англ. – М.: Издательский дом "Вильямс", 2003. – 144 с.

2. Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. – СПб: Питер, 2001. – 368 с.

3. Спольски Дж.Х. Лучшие примеры разработки ПО. – СПб: Питер, 2007. – 208.

4. Будай А. Дизайн-паттерни – простіше простого, - Львів, 2012 – 90с.

5. Э. Фримен, Э. Фримен, К. Сьерра, Б. Гейтс. Паттерны проектирования. – СПб.: Питер, 2011. – 656 с.

6. James W. Cooper. Introduction to design patterns in C#. IBM T J Watson Research Center, 2002, - 424 p.

7. Benny G. Serevsen. Design Patterns. BGS Consulting ApS, 2008, - 224 p

8. В.А. Биллиг. Объектное программирование в классах на C#. Интернет-университет ИНТУИТ, 2008, - 360с.

Допоміжна

9. Осваиваем паттерны проектирования на C#. Часть 1 / Антон Гладченко // «Мир ПК» , № 10, 2009, с. 46-50.

10. Осваиваем паттерны проектирования на C#. Часть 2 / Антон Гладченко // «Мир ПК» , № 11, 2009.

11. Осваиваем паттерны проектирования на C#. Часть 3 / Антон Гладченко // «Мир ПК» «Мир ПК» , № 12, 2009.

12. Осваиваем паттерны проектирования на C#. Часть 4 / Антон Гладченко // «Мир ПК» «Мир ПК» , № 01, 2010.

13. Осваиваем паттерны проектирования на C#. Часть 5 «Мир ПК» , № 02, 2010.

Інформаційні ресурси

14. Интернет ресурс: <http://cpp-reference.ru/patterns/>

15. Интернет ресурс:

<http://www.dofactory.com/Patterns/Patterns.aspx>

16. Интернет ресурс: <http://skillcoding.com/Default.aspx?id=222>

17. Интернет ресурс: <http://msdn.microsoft.com/ru-ru/library/dd460654.aspx#Classes>

18. Интернет ресурс:

<http://www.skillcoding.com/Default.aspx?id=123>

4. МЕТОДИЧНЕ ЗАБЕЗПЕЧЕННЯ

1. Методичні вказівки до самостійної роботи для студентів напрямку 6.050103 „Програмна інженерія”/ К.М. Ялова, В.В. Завгородній // Дніпродзержинськ: ДДТУ, 2013. – 20 с.

2. Методичні вказівки до виконання контрольної роботи студентів заочної форми навчання напрямку 6.050103 „Програмна інженерія”/ К.М. Ялова, В.В. Завгородній // Дніпродзержинськ: ДДТУ, 2013. – 72 с.

3. Моделювання та аналіз програмного забезпечення. Конспект лекцій для студентів напрямку 6.050103 „Програмна інженерія”/ К.М. Ялова, В.В. Завгородній // Дніпродзержинськ: ДДТУ, 2013. – 151 с.

Навчальне видання

Методичні вказівки до лабораторних занять з дисципліни
„Моделювання та аналіз програмного забезпечення” для студентів
напряму 6.050103 „Програмна інженерія” / К.М. Ялова,
В.В.Завгородній // Дніпродзержинськ: ДДТУ, 2013. – 51 с.

Укладачі: к.т.н., доц. Ялова К.М.
ст. викл. Завгородній В.В.

Підписано до друку

Формат	Обсяг	др. ар.
Тираж	екз.	Заказ №

м. Дніпродзержинськ,
вул. Дніпробудівська ,2.