Лабораторна робота №5
Тема: «Формування структури та взаємодії між класами за допомогою шаблонів проектування Facade та Decorator»

Decorator

Decorator (Декоратор) - структурний шаблон проектування, призначений для динамічного підключення додаткової поведінки до об'єкта. Шаблон Декоратор надає гнучку альтернативу практиці створення підкласів з метою розширення функціональності. Відомий також під менш поширеною назвою Обгортка (Wrapper), яке багато в чому розкриває суть реалізації шаблону.

Проблема

Об'єкт, який передбачається використовувати, виконує основні функції. Однак може знадобитися додати до нього деяку додаткову функціональність, яка буде виконуватися до, після або навіть замість основної функціональності об'єкту.

Спосіб вирішення

Декоратор передбачає розширення функціональності об'єкта без визначення підкласів.

Facade

Facade (Фасад) - шаблон проектування, що дозволяє приховати складність системи шляхом зведення всіх можливих зовнішніх викликів до одного об'єкту, який делегує їх відповідними об'єктами системи.

Проблема

Як забезпечити уніфікований інтерфейс з набором розрізнених реалізацій або інтерфейсів, наприклад, з підсистемою, якщо небажано високе зв'язування з цією підсистемою або реалізація підсистеми може змінитися?

Спосіб вирішення

Визначити одну точку взаємодії з підсистемою - фасадний об'єкт, що забезпечує загальний інтерфейс з підсистемою і покласти на нього обов'язок по взаємодії з її компонентами. Фасад - це зовнішній об'єкт, що забезпечує єдину точку входу для служб підсистеми. Реалізація інших компонентів підсистеми закрита і не видна зовнішнім компонентам.

Приклад:

Міська кав'ярня працює декілька років і спеціалізується на виготовленні какових напитків. Коли бізнес лише відкривався в кав'ярні готували 4 види кави: Espresso, HouseBlend, DarkRoast, Decaf. Кожен напій має свою схему розрахунку вартості. Для функціонування такої системи була розроблена наступна ієрархія класів:

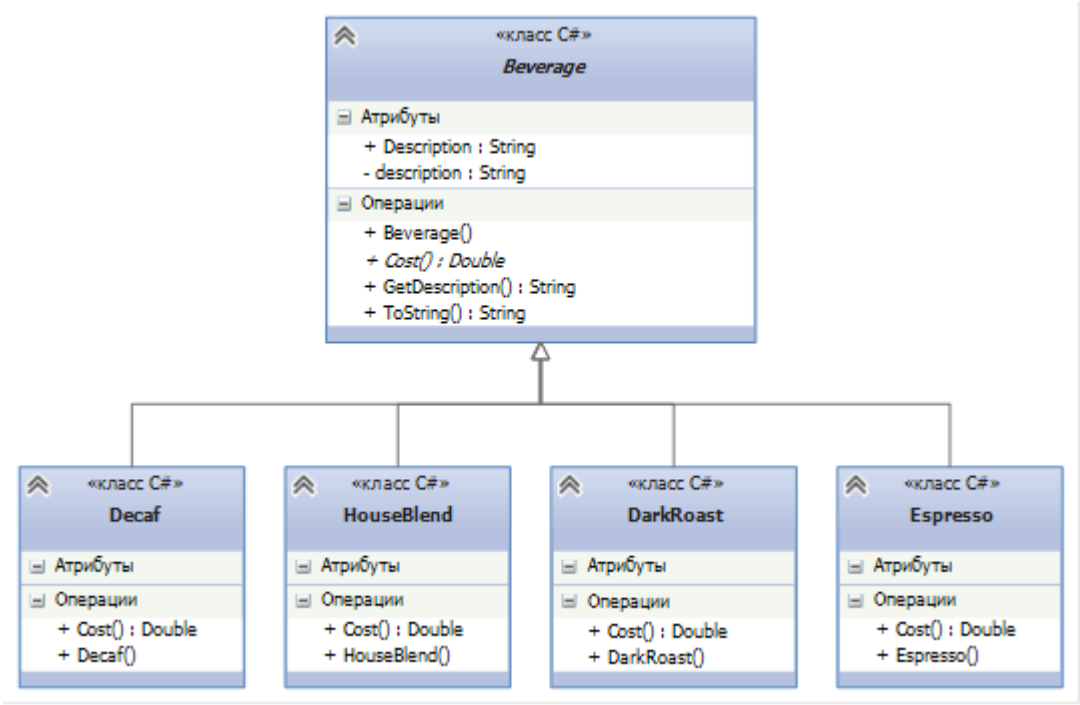


Рис.1. Діаграма класів для магазину кави

Абстрактний дочірній клас Напій (Beverage) містить поле з описом напою, метод який повертає опис напою та абстрактний метод, який повинен повертативартість (вартість розраховується для кожного напою за окремою схемою).

У зв'язку з розширенням мережі кав'ярня закупила нове обладнання і весь час намагається впроваджувати нові напої і радувати клієнтів. Але що стосується внутрішньої структури ієрархії класів, то тут не все так гладко як у бізнесі. Покупець може замовити еспрессо з молоком, а ще з шоколадом і збитими вершками, вартість такої кави бути вираховуватись із всіма додатками, а їх можна комбінувати різними способами, все залежить від вподобання клієнта. В результаті на кожен вид фіксованої кави розробляється новий клас із власною реалізацією методу Cost().

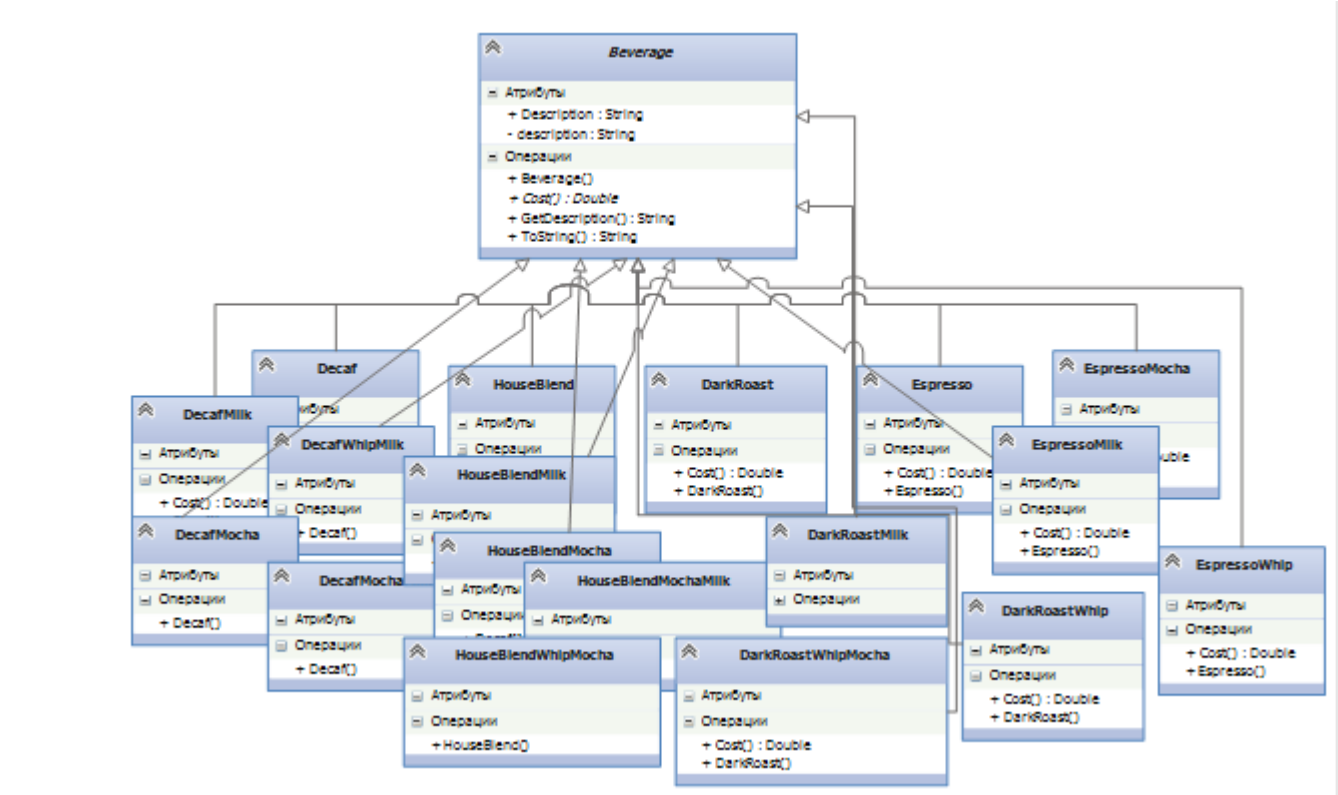


Рис.2. Діаграма класів після додавання функціональності шляхом наслідування.

Супровід системи з такою кількістю класів буде нереально складним. А якщо з'являться нові додатки до кави, то кількість класів буде тільки рости.

Можна піти іншим шляхом (Рис.3). Для кожного додатку створити окреме булівське поле в батькіському класі. Метод Cost() в дочірніх класах буде визначати вартість напою, а потім викликати батьківську версію методу для обрахунку вартості усіх додатків, встановлених користувачем.

Все ніби нічого, кількість класів лише 5. Виглядає досить пристойно. Але, щоб оцінити адекватність такої архітектури, варто задати запитання : Як ця архітектура буде змінюватись в майбутньому?

Наступні пункти викривають усі недоліки:

- при появі нових додатків доведеться вносити нові поля та методи а базовий клас для встановлення додатку та модифікувати метод розрахунку вартості;
- для деяких нових напитків (наприклад холодний чай) доповнення можуть здаватися недоцільними (наприклад шоколад), проте вони все одно реалізуватимуть такі методи.
- що буде якщо клієнт захоче подвійну порцію вершків?

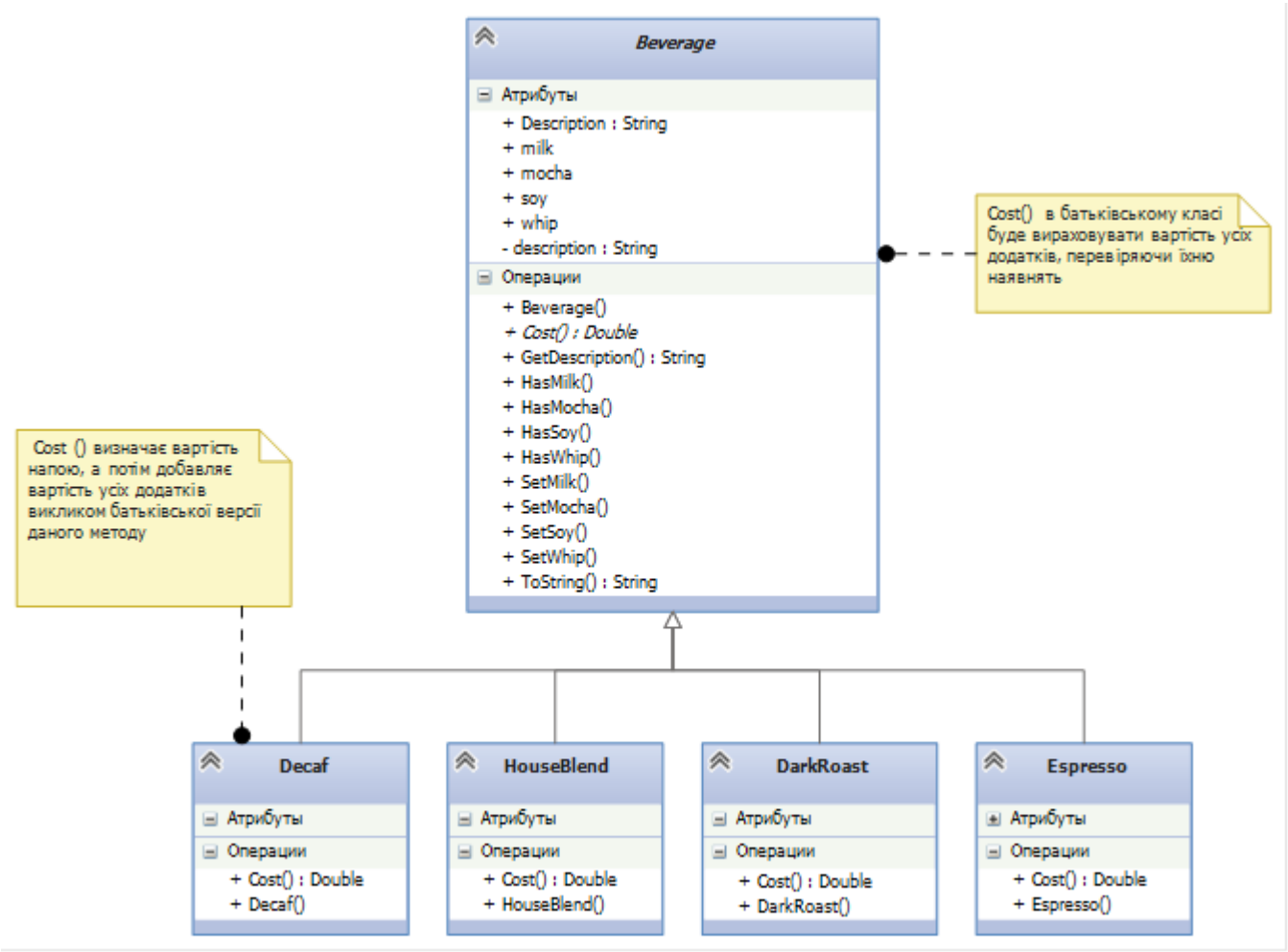


Рис.3. Діаграма класів після модифікації батьківського класу.

Дана архітектура порушує один з основних принципів проектування: *клас повинен бути відкритий для розширення, але закритий для модифікації.*

Наша ціль полягає в тому, щоб класи можна було легко розширювати новою функціональність без зміни існуючого коду. Дану ціль можна досягти скориставшись паретном Декоратор.

Почнемо з базового напитку і декоруємо його на стадії виконання. Наприклад, якщо замовили каву темного б смаження (DarkRoast) зі збитими вершками (Whip) і шоколадом(Mocha), то

- Беремо об’єкт DarkRoast (створюємо об’єкт DarkRoast);
- Декоруємо його як Mocha (створюємо об’єкт Mocha і загортаємо в нього об’єкт DarkRoast);
- Декоруємо ще раз як Whip (створюємо об’єкт Whip і загортаємо в нього об’єкт Mocha);
- Викликаємо метод Cost() і користуємось делегуванням для додавання вартості доповнень. (спершу викликається метод Cost() для об’єкта Whip і додає свою вартість, той в свою чергу викликає метод Cost() для б смаж

Mocha і додає свою вартість, а об’єкт Mocha викликає метод для об’єкта DarkRoast)

Для реалізації такої схеми необхідно створити наступну структуру:

- створити базовий клас декоратора, який буде реалізувати той самий батьківський клас що і елемент, який декорується.
- клас декоратора містить в собі елемент, який декорується (посилання на об’єкт являється полем класу)
- уся необхідна функціональність додається до класу декоратора.

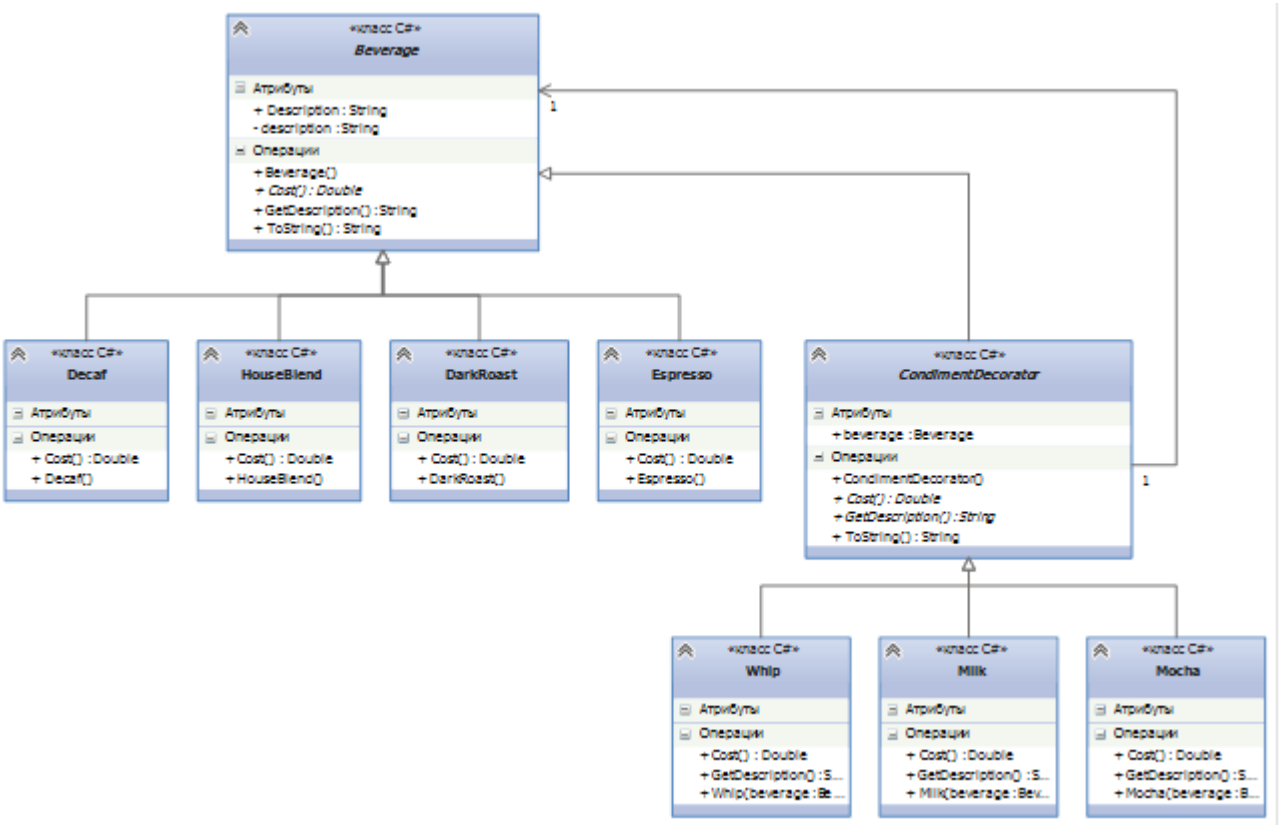


Рис.4. Діаграма класів для магазину кави з використанням Декоратора.

Нижче наведена програмна реалізація даного прикладу.

Створений абстрактний клас для напоїв.

```
Abstract class Beverage
    {
        String description = «Unknown Beverage»;

        public String Description  {set { description = value; }}
        public virtual String GetDescription()
        {
            return description;
        }
        public abstract double Cost();
        public override string ToString()
        {
            return GetDescription() + «: « + Cost()+»$»;
        }
    }
```

Далі чотири класи які представляються уже конкретні напої:

```
    class Espresso : Beverage
    {
        public Espresso()
        {
            Description = «Espresso»;
        }
        public override double Cost()
        {
            return .88;
        }
    }
    class HouseBlend : Beverage
    {
        public HouseBlend()
        {
            Description = «HouseBlend»;
        }
        public override double Cost()
        {
            return .92;
        }
    }
    class DarkRoast : Beverage
    {
        public DarkRoast()
        {
            Description = «DarkRoast»;
        }
        public override double Cost()
        {
            return 1.01;
        }
    }
    class Decaf : Beverage
    {
        public Decaf()
        {
            Description = «Decaf»;
        }
        public override double Cost()
        {
            return 1.10;
        }
    }
```

Декоратор представлений абстрактним класом, який наслідує клас напоїв:

```
abstract class CondimentDecorator: Beverage
    {
        public Beverage beverage;
        public abstract override String GetDescription();
        public abstract override double Cost();

        public override string ToString()
        {
            return GetDescription()+»: «+Cost()+ «$»;
        }
    }
```

Далі слідує реалізація кожного конкретного декоратора:

```
    class Mocha : CondimentDecorator
    {
        public Mocha(Beverage beverage)
        {
            this.beverage = beverage;
        }
```

```
        public override string GetDescription()
        {
            return beverage.GetDescription() + «, Mocha»;
        }
        public override double Cost()
        {
            return (beverage.Cost() + 0.25);
        }
    }

    class Milk : CondimentDecorator
    {
        public Milk(Beverage beverage)
        {
            this.beverage = beverage;
        }
        public override string GetDescription()
        {
            return beverage.GetDescription() + «, Milk»;
        }
        public override double Cost()
        {
            return (beverage.Cost() + 0.10);
        }
    }

    class Whip : CondimentDecorator
    {
        public Whip(Beverage beverage)
        {
            this.beverage = beverage;
        }
        public override string GetDescription()
        {
            return beverage.GetDescription() + «, Whip»;
        }
        public override double Cost()
        {
            return (beverage.Cost() + 0.15);
        }
}
```

Створення різних об’єктів кави має наступний вигляд:

```
class Program
    {
        static void Main(string[] args)
        {
            //Спершу готуємо звичайне еспрессо
             Beverage espresso = new Espresso();
            Console.WriteLine(espresso.ToString());

            //Створюємо звичайну каву темного  обсмаження
            Beverage blackWithCondiment = new DarkRoast();

            //Декоруємо її молоком
            blackWithCondiment = new Milk(blackWithCondiment);

            //та вершками
            blackWithCondiment = new Whip(blackWithCondiment);

            Console.WriteLine(blackWithCondiment.ToString());
        }
}
```

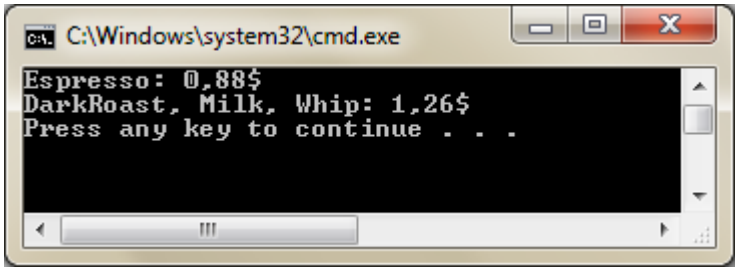


Рис.5. Перевірка виконання програми.

Тепер слід звернути увагу на взаємодію з користувачем. В наведеному прикладі ми безпосередньо в коді прописали створення відповідного напитку. Але таке рішення може бути лише демонстративним. Якщо ми хочемо протестувати виконання усіх методів і можливість створення усіх видів напоїв нам потрібно буде написати доволі багато хаотичного і негнучкого коду.

Щоб уникнути цієї проблеми досить скористатись шаблоном Фасад. З його допомогою ми отримаємо доступ до всієї системи за допомогою одного об’єкту.

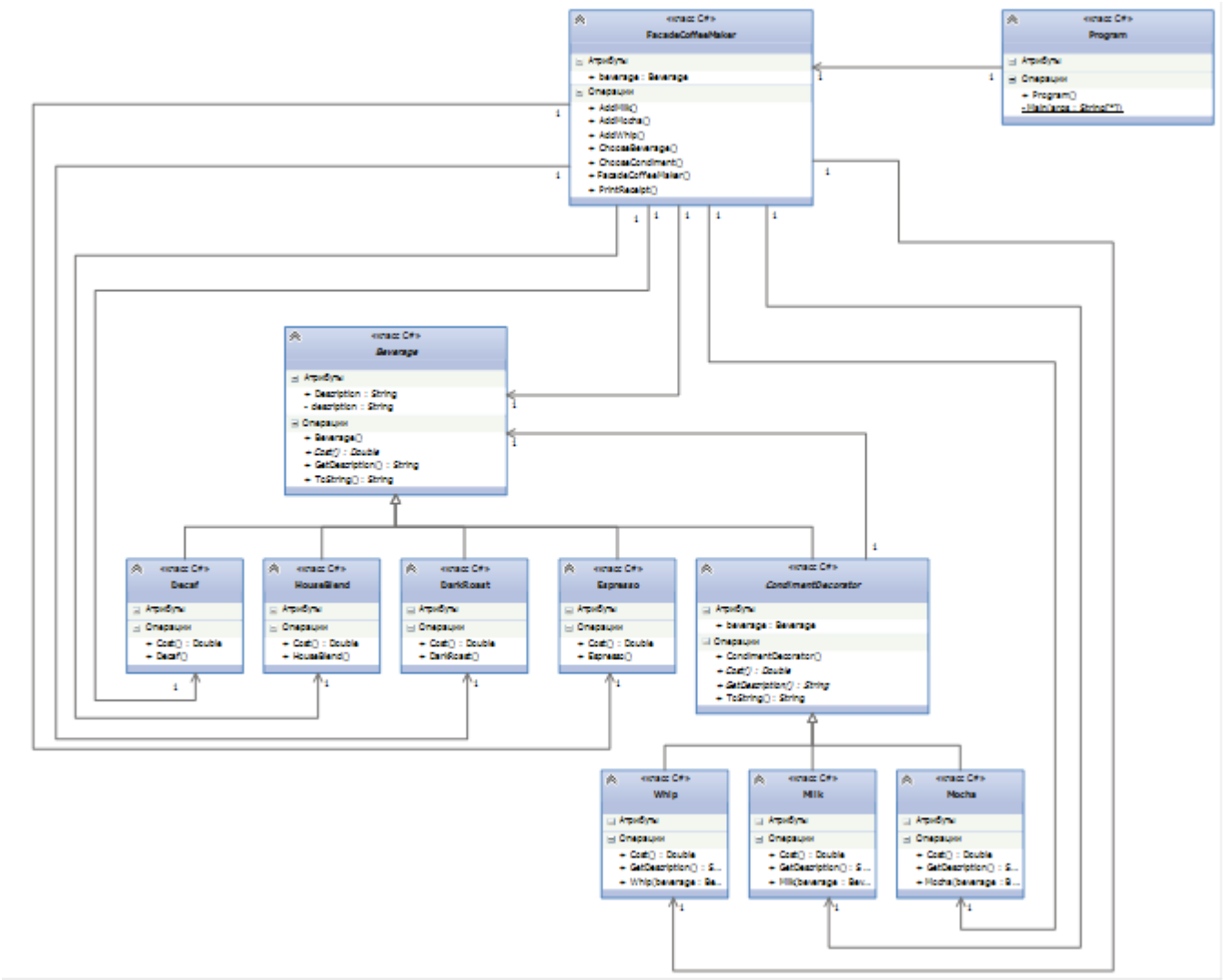


Рис.6. Діаграма класів з використанням фасаду.

Програмний код класу фасаду має наступний вигляд:

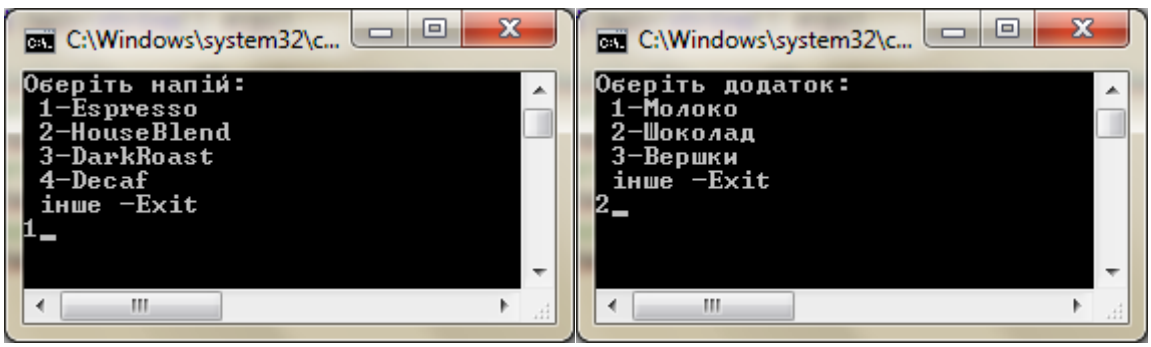
```
/// <summary>
    /// Фасад для роботи з системою приготування кави
    /// </summary>
    class FacadeCoffeeMaker
    {
        public Beverage beverage;
        /// <summary>
        /// Обираємо напій
        /// </summary>
        public void ChooseBeverage()
        {
            Console.Clear();
            Console.WriteLine("Оберіть напій: \n 1-Espresso\n 2-HouseBlend\n
                               3-DarkRoast\n 4-Decaf\n інше -Exit");
            int choose = Convert.ToInt32(Console.ReadLine());
            switch (choose)
            {
                case 1: { beverage = new Espresso(); break; }
                case 2: { beverage = new HouseBlend(); break; }
                case 3: { beverage = new DarkRoast(); break; }
                case 4: { beverage = new Decaf(); break; }
                default: break;
            }
        }
        /// <summary>
        /// Обираємо додатки
        /// </summary>
        public void ChooseCondiment()
        {
            if (beverage != null)
            {
                bool flag = true;
                while (flag)
                {
                    Console.Clear();
                    Console.WriteLine("Оберіть додаток:\n 1-Молоко\n
                                       2-Шоколад\n 3-Вершки \n інше -Exit");
                    int choose = Convert.ToInt32(Console.ReadLine());
                    switch (choose)
                    {
                        case 1: { AddMilk(); break; }
                        case 2: { AddMocha(); break; }
                        case 3: { AddWhip(); break; }
                    }
                    Console.Clear();
                    Console.WriteLine("Бажаєте до кави ще щось?\n 1 - так \n все інше - ні");
                    string temp = Console.ReadLine();
                    if (temp != "1") flag = false;
                }
            }
        }
        /// <summary>
        /// Додаємо молоко
        /// </summary>
        public void AddMilk()
        {
            beverage = new Milk(beverage);
        }
        /// <summary>
        /// Додаємо шоколад
        /// </summary>
        public void AddMocha()
        {
            beverage = new Mocha(beverage);
```

```
        }
        /// <summary>
        /// Додаємо вершки
        /// </summary>
        public void AddWhip()
        {
            beverage = new Whip(beverage);
        }
        /// <summary>
        /// Роздрук чеку
        /// </summary>
        public void PrintReceipt()
        {
            if (beverage != null)
            {
                Console.Clear();
                Console.WriteLine("\n\nВаш рахунок: ");
                Console.WriteLine(beverage.ToString());
                Console.WriteLine("Дякуємо за покупку! Приходьте ще:)\n\n");
            }
            else
            {
                Console.Clear();
                Console.WriteLine("Бувай! Можливо іншим разом:)\n\n");
            }
        }
    }
```

Даний клас передбачений для роботи з консольним додатком. Якщо виникне необхідність за тим самим принципом можна побудувати фасад і для віконного додатку, так як можна описати декілька фасадів для роботи з системою. В даному випадку в класі Program буде написаний наступний код:

```
class Program
{
        static void Main(string[] args)
        {
            FacadeCoffeeMaker coffeeMaker = new FacadeCoffeeMaker();
            //вибрати напій
            coffeeMaker.ChooseBeverage();
            //вибрати додаток
            coffeeMaker.ChooseCondiment();
            //роздрукувати чек
            coffeeMaker.PrintReceipt();
        }
 }
```

В представленій реалізації клієнтський код не цікавить наскільки складною є логіка вибору напоїв, додатків до кави (додатків можна обирати скільки завгодно).



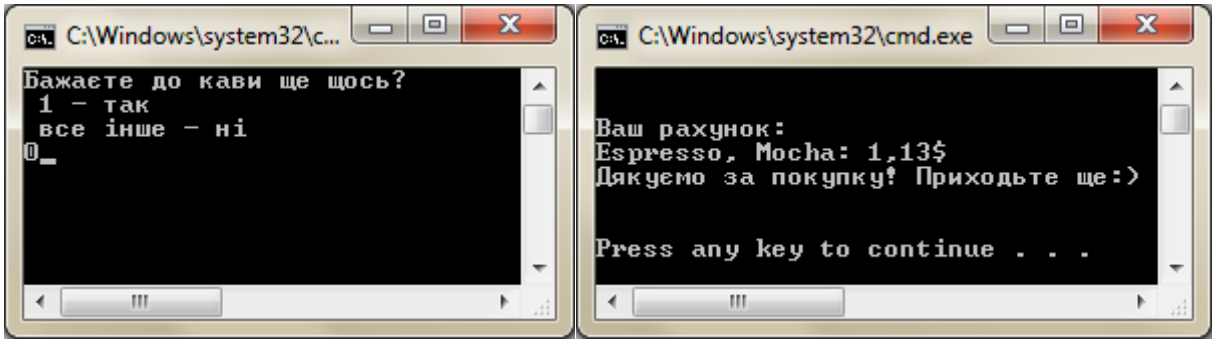


Рис.7. Консольний інтерфейс.

Індивідуальні завдання

Варіант №1

Розробити систему для контролю звітності по заробітній платі для працівників команди фірми-розробника ПЗ. Усі працівники поділяються на штатних і найманих підрядників. Кожна з двох груп має свою систему нарахування заробітної плати і визначення кількості робочих годин в тижнем. Не залежно від групи працівники можуть мати певний рівень кваліфікації (Trainee , Junior, Middle, Senior). Система нарахувань заробітної плати досить гнучка, а керівник команди може визначати ще якийсь проміжних рівень кваліфікації (це залежить від настрою на момент присудження кваліфікації). Також існує система бонусів, якою можуть скористатись всі працівники в якості винагороди. Розробити програму, яка б дозволяла змінювати рівень кваліфікації працівника та відповідно до цього нараховувати йому заробітну плату, формувати список десяти найкращих працівників періоду за кількістю відпрацьованих понаднормових годин та виділяти їм премію у розмірі окладу. Скористатись шаблонами Фасад та Декоратор.

Варіант №2

На військовому складі ведеться облік стрілецької зброї. Там присутні автомати, пістолети, автоматичні гвинтівки певного року випуску. Кожен вид зброї має свою дальність польоту, силу ураження, вагу. До будь-якої з цих видів зброї в якості покращення може бути використано різні аксесуари (приціли, прилади нічного бачення, глушники і т.д.). Вартість усієї додаткової комплектації визначається додатково. При складанні комплекту зброї (наприклад автомат плюс аксесуари) його вага не повинна перевищувати дозволеної для даного виду зброї. Розробити програму, яка б надавала можливість обраховувати вартість зброї, визначати найдорожчий комплект в

арсеналі, здійснювати переоцінку вартості у зв’язку зі зносом (рік випуску). При розробці скористатись шаблонами Фасад та Декоратор.

Варіант №3

Розробити програму, яка керує відображенням рисунків. Рисунки можуть бути кольоровими або чорно-білими, кожен з цих видів характеризується кількістю пік селів та розміром, який він займає на диску. До кожного рисунку можна застосувати фільтр, а їх чимало: розмитість, чіткість, насиченість, художній шум. Кожен фільтр збільшує або зменшує розмір зображення. Також є фільтри які змінюють геометричні розміри рисунку (вирізають у формі кола чи прямокутника) – від цього теж залежить подальший розмір зображення. Розробити програму, яка б зберігала список різних зображень; могла б відбирати їх за видом фільтру та виводити інформацію про назву та розмір, сортувати зображення за розміром. При розробці скористатись шаблонами Фасад та Декоратор.

Варіант №4

Розробити програму, яка б надавала інформацію про персонажей комп’ютерної гри. Вони бувають трьох видів: люди (фізично найслабші), тролі (середні) та орки (найсильніші). Кожен з персонажей має рівень захисту та силу. Сила залежить від початкової величини та він того, яким озброєнням володіє персонаж (луки, мечі, булава, кінжали, сокири). Персонаж може заволодіти будь-яким видом озброєння. Рівень захисту залежить від одягу(обладунки) та щитів(дерев’яні, металеві). Програма повинна створювати персонажей усіх видів. Для вибраного героя здійснювати перевірку, чи є хтось сильніший за нього, та чи є персонажі, яких він може побити за 1, 5 чи 10 ударів (чи перевищує сила його удару чи групи ударів чийсь захист). При розробці скористатись шаблонами Фасад та Декоратор.