

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ ІМЕНІ ІГОРЯ
СІКОРСЬКОГО»

Факультет інформатики та обчислювальної техніки
Кафедра інформатики та програмної інженерії

Комп'ютерний практикум №6

з курсу «Основи розробки програмного забезпечення на платформі
Microsoft.NET»

на тему: «Шаблони проектування. Поведінкові шаблони»

Перевірила:
доцент
Ліщук К. І.

Виконав:
студент 2 курсу
групи ІІІ-21 ФІОТ
Гриценко А. В.

Київ 2024

Варіант 19

Мета: ознайомитися з основними шаблонами проектування, навчитися застосовувати їх при проектуванні і розробці ПЗ.

Постановка задачі комп'ютерного практикуму № 6

При виконанні комп'ютерного практикуму необхідно виконати наступні дії:

- 1) Вивчити поведінкові патерни. Знати загальну характеристику та призначення кожного з них, особливості реалізації кожного з поведінкових патернів та випадки їх застосування.
- 2) Реалізувати задачу згідно варіанту, запропонованого нижче у вигляді консольного застосування на мові C#. Розробити інтерфейси та класи з застосування одного або декількох патернів. Повністю реалізувати методи, пов'язані з реалізацією обраного патерну.
- 3) Повністю описати архітектуру проекту (призначення методів та класів), особливості реалізації обраного патерну. Для кожного патерну необхідно вказати основні класи та їх призначення.
- 4) Навести UML-діаграму класів
- 5) Звіт повинен містити:
 - a. обґрунтування обраного патерну (чому саме він);
 - b. опис архітектури проекту (призначення методів та класів);
 - c. UML-діаграму класів
 - d. особливості реалізації обраного патерну
 - e. текст програмного коду
 - f. скріншоти результатів виконання.

Варіант індивідуального завдання:

- 19) Реалізувати модель, в якій існує лінійний список об'єктів та реалізується можливість послідовного обходу у прямому та зворотному напрямках оминаючи порожні елементи цієї структури та не розкриваючи

її сутності перед користувачем.

Було обрано патерн "Ітератор", що дозволяє послідовно обходити елементи колекції без розкриття її внутрішнього представлення. Перш за все, він (патерн) забезпечує інкапсуляцію внутрішньої структури колекції, що дозволяє клієнтському коду не залежати від деталей реалізації колекції, які можуть змінюватися. Використовуючи патерн "Ітератор", логіка обходу колекції зберігається всередині ітератора, що дозволяє клієнтському коду взаємодіяти лише з інтерфейсом ітератора, не знаючи, як саме реалізована колекція. По-друге, патерн "Ітератор" надає гнучкість обходу колекції, дозволяючи реалізувати різні стратегії обходу. У даному випадку ітератор забезпечує як прямий, так і зворотний обхід колекції, оминаючи порожні елементи. Це робить патерн дуже гнучким і розширюваним. Ще однією важливою перевагою патерну "Ітератор" є можливість роботи з різними типами колекцій через однорідний інтерфейс. Це дозволяє клієнтському коду працювати з різними типами колекцій однаковим чином, підвищуючи узгодженість і гнучкість системи.

Проект складається з наступних основних компонентів:

- Client (клієнтський код)
- Iterator (інтерфейс ітератора)
- ConcreteIterator (конкретний ітератор)
- IterableCollection (інтерфейс колекції)
- ConcreteCollection (конкретна колекція)

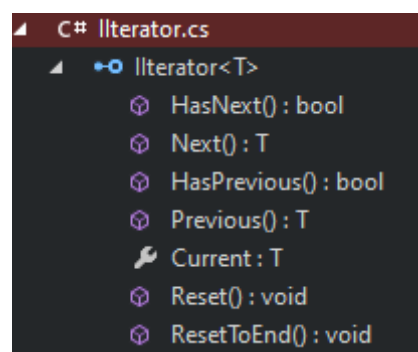
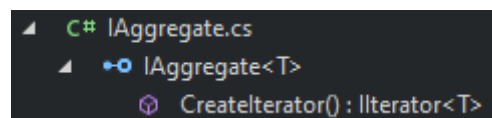


Рисунок 1. Інтерфейс ітератора

Визначає інтерфейс для ітераторів, що підтримують обход елементів колекції в обох напрямках. Методи:

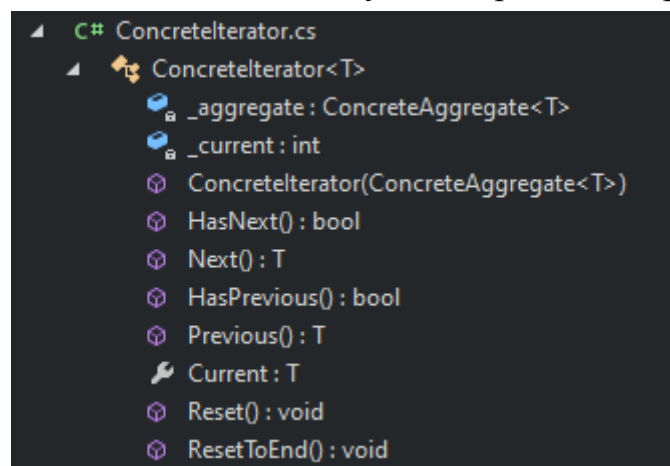
- `bool HasNext()` - перевірка наявності наступного елемента
- `T Next()` - повернення наступного елемента
- `bool HasPrevious()` - перевірка наявності попереднього елемента
- `T Previous()` - повернення попереднього елемента
- `void Reset()` - скидання ітератора до початку колекції
- `void ResetToEnd()` - скидання ітератора до кінця колекції



```
C# IAggregate.cs
IAggregate<T>
CreateIterator() : IEnumerator<T>
```

Рисунок 2. Інтерфейс колекції

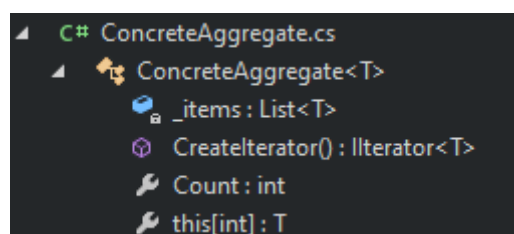
Визначає інтерфейс для колекцій, які можуть створювати ітератори.



```
C# ConcreteIterator.cs
ConcreteIterator<T>
    _aggregate : ConcreteAggregate<T>
    _current : int
    ConcreteIterator(ConcreteAggregate<T>)
    HasNext() : bool
    Next() : T
    HasPrevious() : bool
    Previous() : T
    Current : T
    Reset() : void
    ResetToEnd() : void
```

Рисунок 3. Конкретний ітератор

Реалізує інтерфейс `IEnumerator<T>` та надає механізм для обходу колекції в обох напрямках, оминаючи порожні елементи.



```
C# ConcreteAggregate.cs
ConcreteAggregate<T>
    _items : List<T>
    CreateIterator() : IEnumerator<T>
    Count : int
    this[int] : T
```

Рисунок 4. Конкретна колекція

Реалізує інтерфейс `IAggregate<T>` та представляє конкретну колекцію, яка може створювати ітератори.

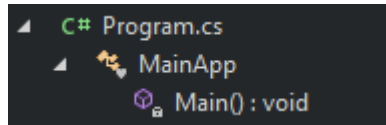


Рисунок 5. Клієнтський код

Особливості реалізації обраного патерну "Декоратор"

Об'єкт Ітератор буде відстежувати стан обходу, поточну позицію в колекції і скільки елементів ще залишилося обійти. Одну і ту ж колекцію зможуть одночасно обходити різні ітератори, а сама колекція не буде навіть знати про це. Колекція не розкриває своє внутрішнє представлення. Ітератор здійснює обхід колекції, не розкриваючи її структури. Легко додати нові способи обходу колекції, просто реалізувавши новий ітератор. Клієнтський код використовує інтерфейси `Iterator<T>` та `IAggregate<T>`, що дозволяє працювати з різними типами колекцій та ітераторів. Ітератор надає методи для обходу в обох напрямках та оминає порожні елементи, що спрощує роботу з колекцією.

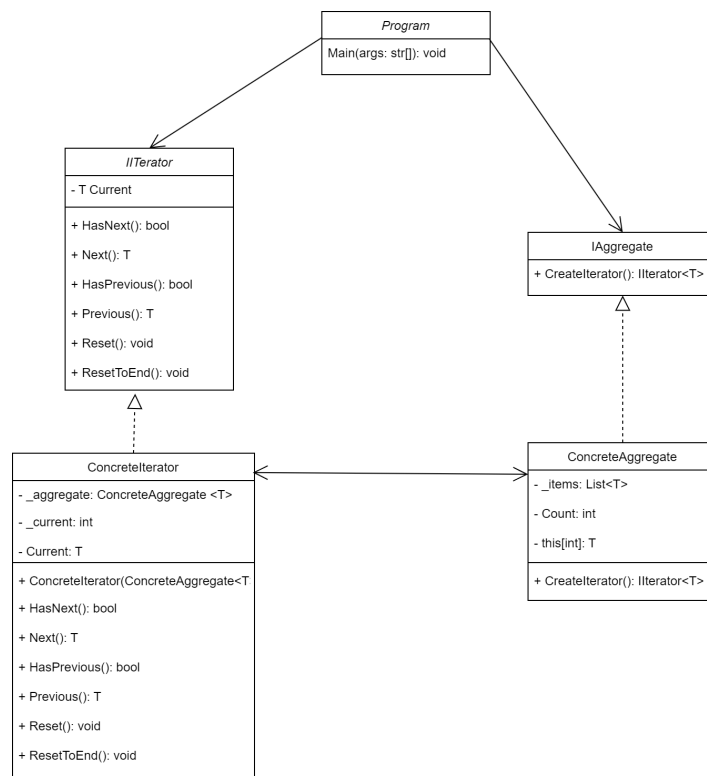


Рисунок 6. UML-діаграма класів

Скріншоти результатів виконання

```
Forward iterating over collection:  
Item A  
Item C  
Item D  
Item F  
  
Backward iterating over collection:  
Item F  
Item D  
Item C  
Item A
```

Рисунок 7. Результати роботи програми

Висновок

У результаті виконання цієї лабораторної роботи було розроблено консольне застосування на мові C# з використанням патерну "Ітератор". Патерн "Ітератор" дозволив інкапсулювати логіку обходу колекції, забезпечити гнучкість і розширюваність системи, а також спростити клієнтський код і полегшити тестування. Це підкреслює важливість і корисність використання поведінкових патернів при розробці програмного забезпечення.

Программный код

Iterator.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Security.Cryptography;
using System.Text;
using System.Threading.Tasks;
```

```
namespace laba6
{
    public interface IIterator<T>
    {
        bool HasNext();
        T Next();
        bool HasPrevious();
        T Previous();
        T Current { get; }
        void Reset();
        void ResetToEnd();
    }
}
```

IAggregate.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

```
namespace laba6
{
    public interface IAggregate<T>
    {
```

```

        Iterator<T> CreateIterator();
    }
}

```

ConcreteIterator.cs:

```

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

namespace laba6
{
    public class ConcreteIterator<T> : Iterator<T>
    {
        private readonly ConcreteAggregate<T> _aggregate;
        private int _current = -1;

        public ConcreteIterator(ConcreteAggregate<T> aggregate)
        {
            _aggregate = aggregate;
        }

        public bool HasNext()
        {
            int nextIndex = _current + 1;
            while (nextIndex < _aggregate.Count && _aggregate[nextIndex] ==
null)
            {
                nextIndex++;
            }
            return nextIndex < _aggregate.Count;
        }

        public T Next()
        {

```



```

do
{
    _current++;
} while (_current < _aggregate.Count && _aggregate[_current] == null);
return _current < _aggregate.Count ? _aggregate[_current] : default(T);
}

```

```

public bool HasPrevious()
{
    int prevIndex = _current - 1;
    while (prevIndex >= 0 && _aggregate[prevIndex] == null)
    {
        prevIndex--;
    }
    return prevIndex >= 0;
}

```

```

public T Previous()
{
    do
    {
        _current--;
    } while (_current >= 0 && _aggregate[_current] == null);
    return _current >= 0 ? _aggregate[_current] : default(T);
}

```

```

public T Current => _current >= 0 && _current < _aggregate.Count ?
_aggregate[_current] : default(T);

```

```

public void Reset()
{
    _current = -1;
}

```

```

public void ResetToEnd()
{
    _current = _aggregate.Count;
}

```

```
    }  
  }  
}
```

ConcreteAggregate.cs:

```
using System;  
using System.Collections;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
  
namespace laba6  
{  
    using System.Collections;  
  
    public class ConcreteAggregate<T> : IAggregate<T>  
    {  
        private List<T> _items = new List<T>();  
  
        public IIterator<T> CreateIterator()  
        {  
            return new ConcreteIterator<T>(this);  
        }  
  
        public int Count => _items.Count;  
  
        public T this[int index]  
        {  
            get  
            {  
                if (index >= 0 && index < _items.Count)  
                {  
                    return _items[index];  
                }  
                return default(T);  
            }  
        }  
    }  
}
```

```

    }
    set
    {
        if (index >= _items.Count)
        {
            _items.Add(value);
        }
        else
        {
            _items[index] = value;
        }
    }
}
}
}
}

```

Program.cs:

```

using laba6;
using System;

```

```

namespace IteratorPattern

```

```

{

```

```

    class MainApp

```

```

    {

```

```

        static void Main()

```

```

        {

```

```

            ConcreteAggregate<string> aggregate = new
ConcreteAggregate<string>();

```

```

            aggregate[0] = "Item A";

```

```

            aggregate[1] = null;

```

```

            aggregate[2] = "Item C";

```

```

            aggregate[3] = "Item D";

```

```

            aggregate[4] = null;

```

```

            aggregate[5] = "Item F";

```

```

            IIterator<string> iterator = aggregate.CreateIterator();

```

```
    Console.WriteLine("Forward iterating over collection:");
    while (iterator.HasNext())
    {
        Console.WriteLine(iterator.Next());
    }

    iterator.ResetToEnd();

    Console.WriteLine("\nBackward iterating over collection:");
    while (iterator.HasPrevious())
    {
        Console.WriteLine(iterator.Previous());
    }

    Console.ReadKey();
}
}
```