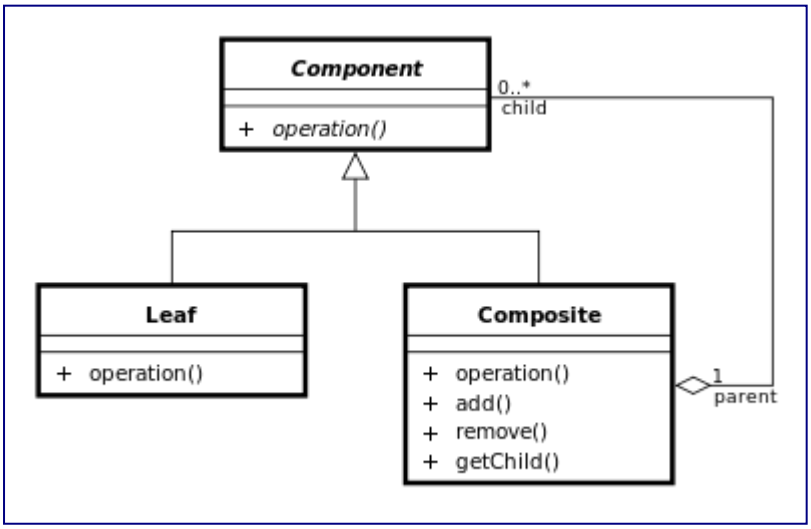
69. Реалізувати шаблони безпечний Composite та Visitor для представлення та обчислення арифметичних виразів.

Компонувальник **Composite** — структурний шаблон який об'єднує об'єкти в ієрархічну деревовидну структуру, і дозволяє уніфіковане звертання для кожного елемента дерева.

Призначення
Дозволяє користувачам будувати складні структури з простіших компонентів. Проектувальник може згрупувати дрібні компоненти для формування більших, які, в свою чергу, можуть стати основою для створення ще більших.

Структура



Ключем до паттерну компонувальник є абстрактний клас, який є одночасно і примітивом, і контейнером(Component). У ньому оголошені методи, специфічні для кожного виду об'єкта (такі як Operation) і загальні для всіх складових об'єктів, наприклад операції для доступу і управління нащадками. Підкласи Leaf визначає примітивні об'єкти, які не є контейнерами. У них операція Operation реалізована відповідно до їх специфічних потреб. Оскільки у примітивних об'єктів немає нащадків, то жоден з цих підкласів не реалізує операції, пов'язані з управління нащадками (Add, Remove, GetChild). Клас Composite складається з інших примітивніших об'єктів Component. Реалізована в ньому операція Operation викликає однойменну функцію відтворення для кожного нащадка, а операції для роботи з нащадками вже не порожні. Оскільки інтерфейс класу Composite відповідає інтерфейсу Component, то до складу об'єкта Composite можуть входити і інші такі ж об'єкти.

Учасники

- Component (Component)

Оголошує інтерфейс для компонуемих об'єктів; Надає відповідну реалізацію операцій за замовчуванням, загальну для всіх класів; Оголошує єдиний інтерфейс для доступу до нащадків та управління ними; Визначає інтерфейс для доступу до батька компонента в рекурсивної структурі і при необхідності реалізує його (можливість необов'язкова);

- Leaf (Leaf_1, Leaf_2) — лист.

Об'єкт того ж типу що і Composite, але без реалізації контейнерних функцій; Представляє листові вузли композиції і не має нащадків; Визначає поведінку примітивних об'єктів в композиції; Входить до складу контейнерних об'єктів;

- Composite (Composite) — складений об'єкт.

Визначає поведінку контейнерних об'єктів, у яких є нащадки; Зберігає ієрархію компонентів-нащадків; Реалізує пов'язані з управлінням нащадками (контейнерні) операції в інтерфейсі класу Component;

Приклад реалізації

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

namespace DoFactory.GangOfFour.Composite.Structural
{
  class MainApp
  {
    static void Main()
    {
      // Create a tree structure
      Composite root = new Composite("root");
      root.Add(new Leaf("Leaf A"));
      root.Add(new Leaf("Leaf B"));

      Composite comp = new Composite("Composite X");
      comp.Add(new Leaf("Leaf XA"));
      comp.Add(new Leaf("Leaf XB"));

      root.Add(comp);
      root.Add(new Leaf("Leaf C"));

      // Add and remove a leaf
      Leaf leaf = new Leaf("Leaf D");
      root.Add(leaf);
      root.Remove(leaf);

      // Recursively display tree
      root.Display(1);

      // Wait for user
      Console.ReadKey();
    }
  }

  abstract class Component
  {
    protected string name;

    // Constructor
    public Component(string name)
    {
      this.name = name;
    }

    public abstract void Add(Component c);
    public abstract void Remove(Component c);
    public abstract void Display(int depth);
  }

  class Composite : Component
  {
    private List<Component> _children = new List<Component>();

    // Constructor
    public Composite(string name)
      : base(name)
    {
    }

    public override void Add(Component component)
    {
      _children.Add(component);
    }

    public override void Remove(Component component)
    {
      _children.Remove(component);
    }

    public override void Display(int depth)
    {
      Console.WriteLine(new String('-', depth) + name);

      // Recursively display child nodes
      foreach (Component component in _children)
```

```
      {
        component.Display(depth + 2);
      }
    }
  }

  class Leaf : Component
  {
    // Constructor
    public Leaf(string name)
      : base(name)
    {
    }

    public override void Add(Component c)
    {
      Console.WriteLine("Cannot add to a leaf");
    }

    public override void Remove(Component c)
    {
      Console.WriteLine("Cannot remove from a leaf");
    }

    public override void Display(int depth)
    {
      Console.WriteLine(new String('-', depth) + name);
    }
  }
}
```

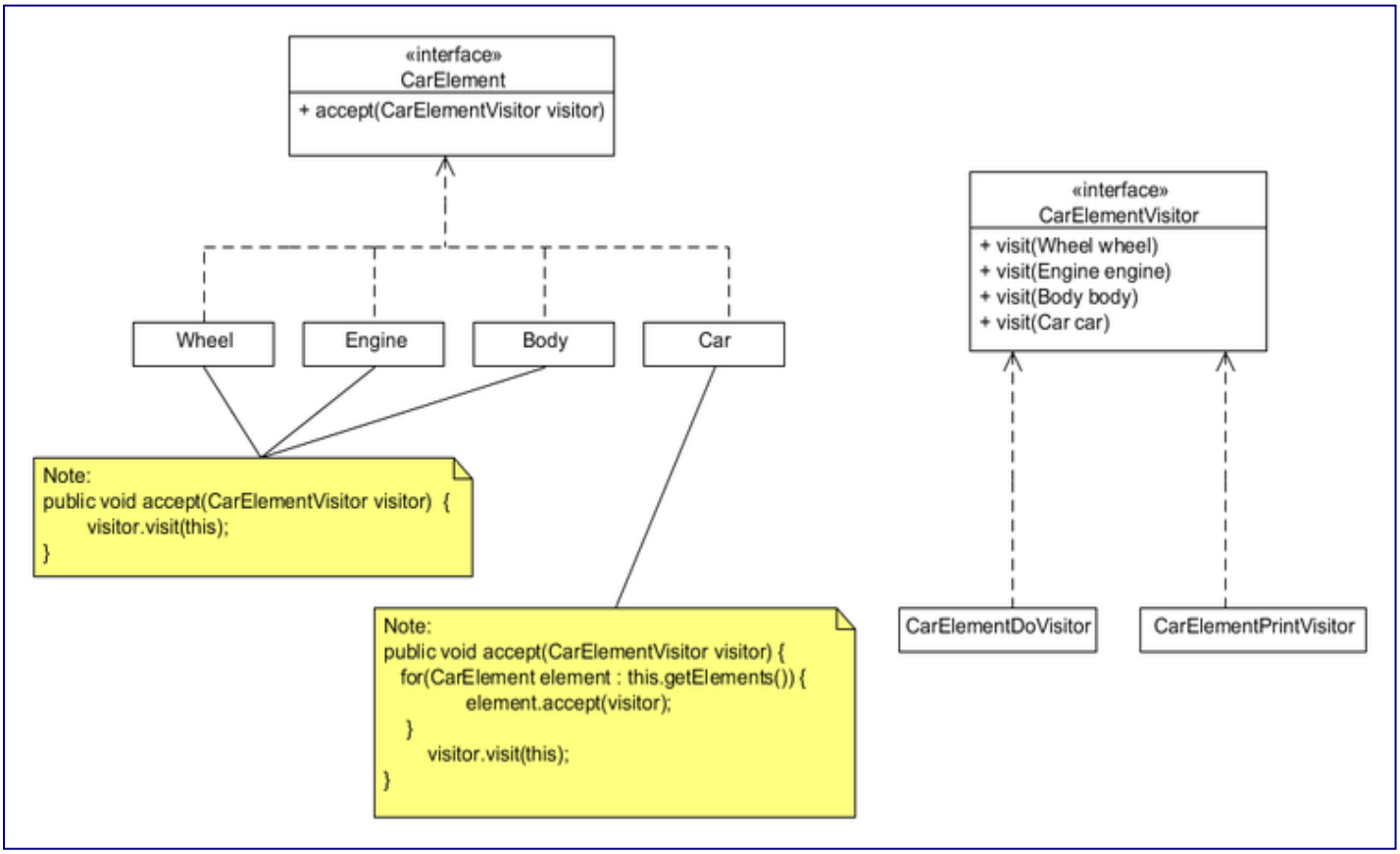
Відвідувач (**Visitor**) - шаблон проектування, який дозволяє відділити певний алгоритм від елементів, на яких алгоритм має бути виконаний, таким чином можливо легко додати або ж змінити алгоритм без змін щодо елементів системи. Практичним результатом є можливість додавання нових операцій в існуючі структури об'єкта без зміни цих структур.
Відвідувач дозволяє додавати нові віртуальні функції в родинні класи без зміни самих класів, натомість, один відвідувач створює клас, який реалізує всі відповідні спеціалізації віртуальної функції. Відвідувач бере приклад посилання в якості вхідних даних і реалізується шляхом подвійної диспетчеризації.Призначення шаблону

- Шаблон Відвідувач визначає операцію, виконувану над кожним елементом деякої структури. Дозволяє, не змінюючи класи цих об'єктів, додавати в них нові операції.
- Є класичною технікою для відновлення втраченої інформації про тип.
- Шаблон Відвідувач дозволяє виконати потрібні дії в залежності від типів двох об'єктів.

Проблема
Над кожним об'єктом деякої структури виконується одна або більше операцій. Визначити нову операцію, не змінюючи класи об'єктів.
Опис шаблону
Основним призначенням патерну Відвідувач є введення абстрактної функціональності для сукупної ієрархічної структури об'єктів "елемент", а саме, патерн відвідувач дозволяє, не змінюючи класи елементів, додавати в них нові операції. Для цього вся обробна функціональність переноситься з самих класів елементів в ієрархію спадкування Відвідувача.
Шаблон відвідувач дозволяє легко додавати нові операції - потрібно просто додати новий похідний від відвідувача клас. Однак патерн Відвідувач слід використовувати тільки в тому випадку, якщо підкласи елементів сукупної ієрархічної структури залишаються стабільними (незмінними). В іншому випадку, потрібно докласти значних зусиль на оновлення всієї ієрархії.
Іноді наводяться заперечення з приводу використання патерну Відвідувач, оскільки він розділяє дані та алгоритми, що суперечить концепції об'єктно-орієнтованого програмування . Однак успішний досвід застосування STL, де поділ даних і алгоритмів покладено в основу, доводить можливість використання патерну відвідувач.
Особливості шаблону

- Сукупна структура об'єктів елементу може визначатися за допомогою патерну Компонувальник (Composite).
- Для обходу може використовуватися Ітератор (Iterator).

- Шаблон Відвідувач демонструє класичний прийом відновлення інформації про втрачені типи, не вдаючись до понижуючого приведення типів(динамічне приведення).



Приклад реалізації

```
interface CarElementVisitor {
    void visit(Wheel wheel);
    void visit(Engine engine);
    void visit(Body body);
    void visit(Car car);
}

interface CarElement {
    void accept(CarElementVisitor visitor); // CarElements have to provide accept().
}

class Wheel implements CarElement {
    private String name;

    public Wheel(String name) {
        this.name = name;
    }

    public String getName() {
        return this.name;
    }

    public void accept(CarElementVisitor visitor) {
        visitor.visit(this);
    }
}

class Engine implements CarElement {
    public void accept(CarElementVisitor visitor) {
        visitor.visit(this);
    }
}

class Body implements CarElement {
    public void accept(CarElementVisitor visitor) {
        visitor.visit(this);
```

```
    }
}

class Car implements CarElement {
    CarElement[] elements;

    public Car() {
        //create new Array of elements
        this.elements = new CarElement[] { new Wheel("front left"),
            new Wheel("front right"), new Wheel("back left") ,
            new Wheel("back right"), new Body(), new Engine() };
    }

    public void accept(CarElementVisitor visitor) {
        for(CarElement elem : elements) {
            elem.accept(visitor);
        }
        visitor.visit(this);
    }
}

class CarElementPrintVisitor implements CarElementVisitor {
    public void visit(Wheel wheel) {
        System.out.println("Visiting " + wheel.getName() + " wheel");
    }

    public void visit(Engine engine) {
        System.out.println("Visiting engine");
    }

    public void visit(Body body) {
        System.out.println("Visiting body");
    }

    public void visit(Car car) {
        System.out.println("Visiting car");
    }
}

class CarElementDoVisitor implements CarElementVisitor {
    public void visit(Wheel wheel) {
        System.out.println("Kicking my " + wheel.getName() + " wheel");
    }

    public void visit(Engine engine) {
        System.out.println("Starting my engine");
    }

    public void visit(Body body) {
        System.out.println("Moving my body");
    }

    public void visit(Car car) {
        System.out.println("Starting my car");
    }
}

public class VisitorDemo {
    static public void main(String[] args) {
        Car car = new Car();
        car.accept(new CarElementPrintVisitor());
        car.accept(new CarElementDoVisitor());
    }
}
```