

# Projektowanie obiektowe oprogramowania

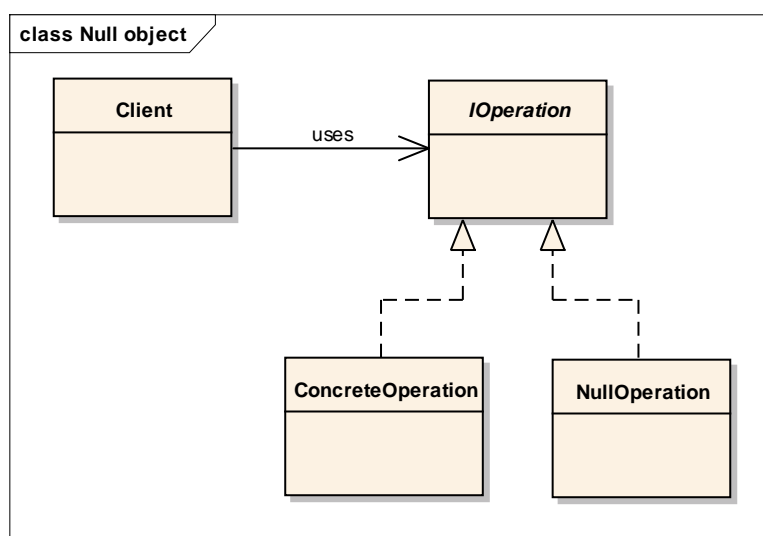
## Wykład 6 – wzorce czynnościowe

### Wiktor Zychła 2025

---

#### 1 Null Object

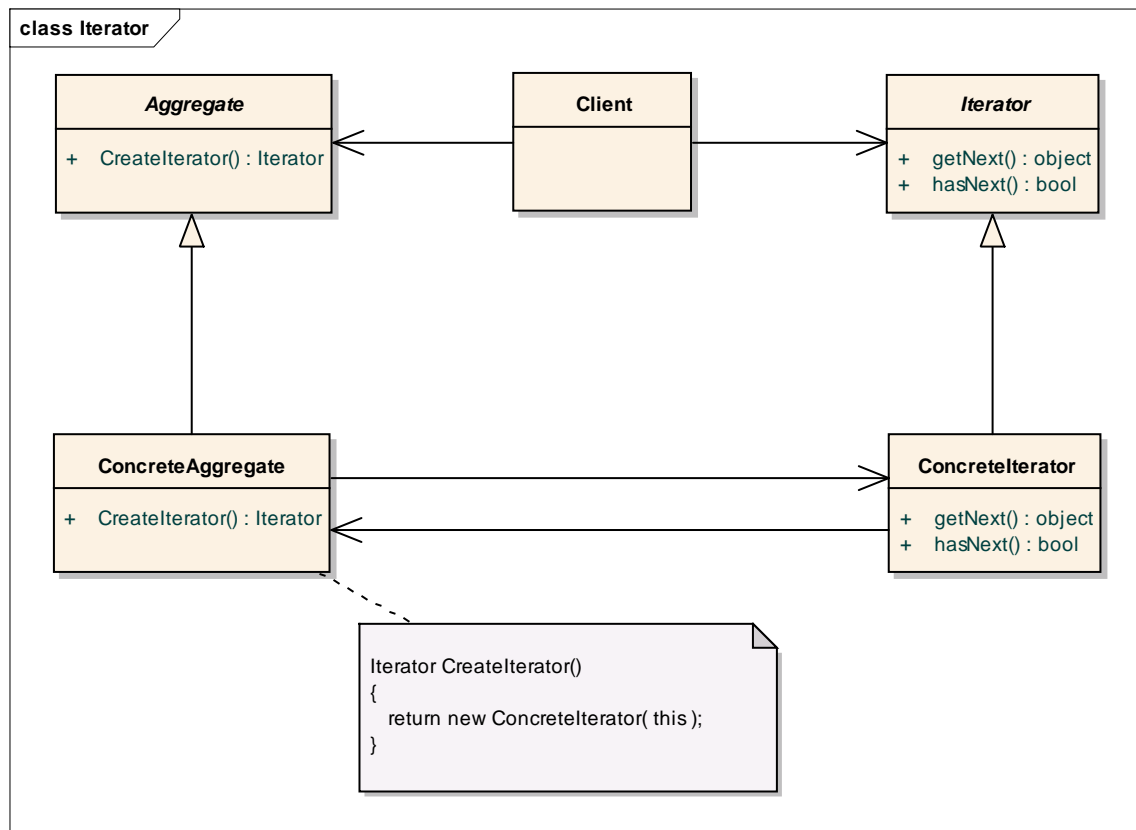
Motto: pusta implementacja zwalniająca klienta z testów **if** na *null*



Komentarz: Null object dobrze sprawdza się w połączeniu z fabryką – przy specyficznych lub niedostatecznych parametrach inicjalizacyjnych fabryka zwraca Null object zamiast referencję **null**. Klient ma więc gwarancję otrzymania zawsze obiektu implementującego oczekiwany kontrakt, co najwyżej jest to jednak obiekt o pustej implementacji metod.

## 2 Iterator

Motto: dostęp do obiektu jak do „kolekcji”, bez ujawniania jego struktury  
Kojarzyć: *IEnumerator*, *IEnumerable*



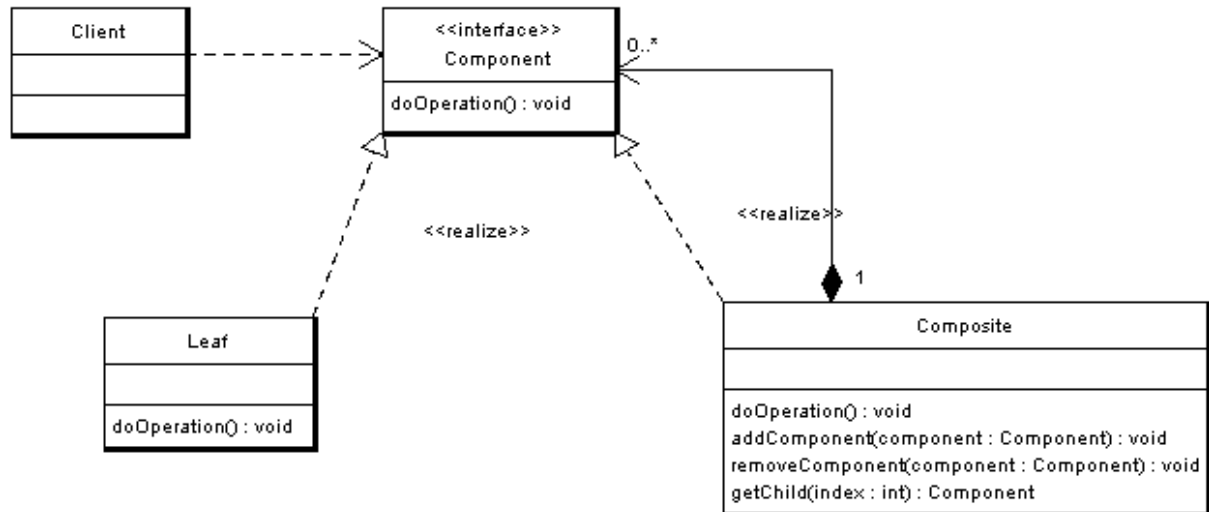
Komentarz: ten wzorec został z powodzeniem włączony do nowoczesnych języków programowania (Java – **Iterator**<>, C# - **IEnumerator**<>) stanowiąc podstawę dla lukru syntaktycznego (Java – **for:** C# - **foreach**).

To kolejny przykład jak wzorce projektowe wprost wpływają na języki/technologie – poprzednim tak jaskrawym przykładem był wzorec Dekorator i jego implementacja w bibliotekach strumieni w platformach przemysłowych.

### 3 Composite

Motto: składanie obiektów w struktury „drzewiaste”

Kojarzyć: Tree, Expression



Wzorzec **Composite** jako taki nie wnosi wiele wartości sam w sobie. Stanowi raczej bazę dla implementacji kolejnych dwóch: **Interpreter** i **Visitor**. We wzorcu Composite chodzi o reprezentowanie struktury rekursywnej, w której struktura komponentu może zależeć od jednego lub wielu komponentów.

W ten oczywisty sposób implementuje się na przykład drzewa. Przykładowo dla drzewa binarnego, abstrakcyjna klasa reprezentująca dowolny węzeł (**Tree**) miałaby dwie podklasy, **TreeLeaf** dla reprezentacji liścia i **TreeNode** dla reprezentacji węzła z podwęzłami:

```
public abstract class Tree
{
}

public class TreeLeaf : Tree
{
}

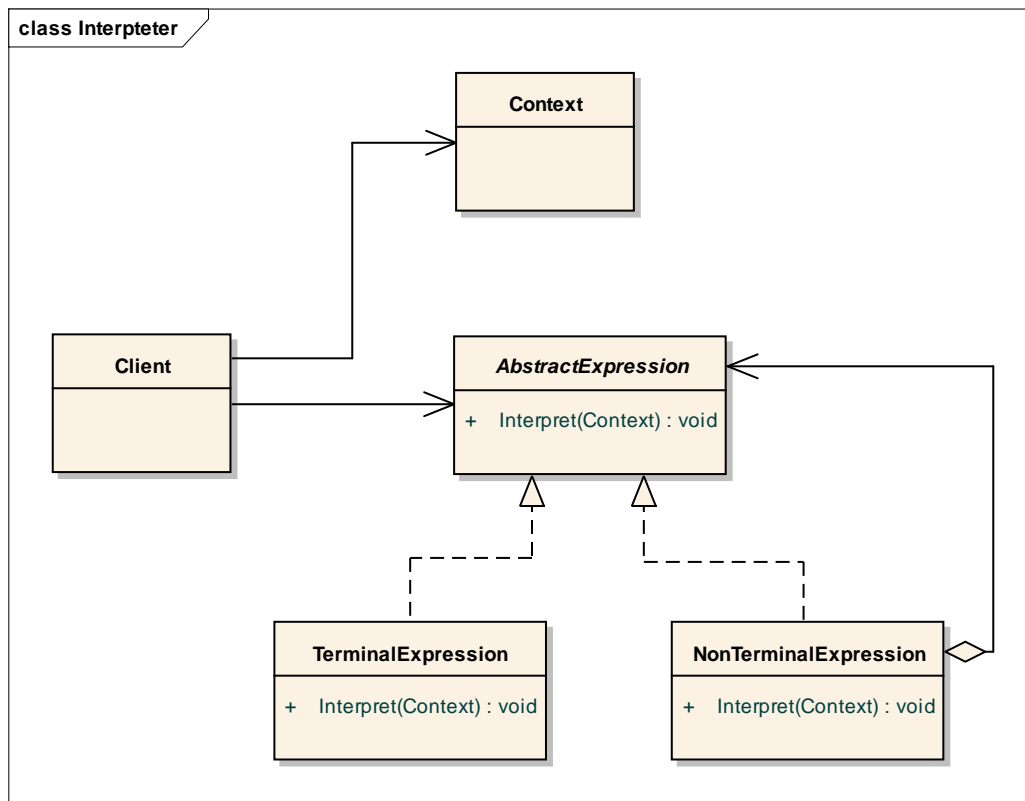
public class TreeNode : Tree
{
    public Tree Left;
    public Tree Right;
}
```

Jak widać, podstawowy pomysł polega tu na wykorzystaniu typu bazowego w odwołaniu rekurencyjnym. W ten sposób z jednej strony można reprezentować dowolnie złożoną strukturę (lewy/prawy podwęzeł węzła może być liściem albo kolejnym węzłem) ale też zdefiniować jakąś operację na poziomie klasy bazowej a potem tylko implementować ją w kolejnych podklasach.



## 4 Interpreter (Little Language)

Motto: reprezentacja gramatyki języka i jego interpretera  
Kojarzyć: kompozyt z interpreterem



To wzorec który odwołuje się wprost do zasady **Inheritance** z GRASP. Przez delegowanie implementacji do podklasy, cała hierarchia uzyskuje jednolity sposób implementacji konkretnego algorytmu – tu chodzi o „interpretowanie”.

Dodatkowy **Context** może nieść ze sobą informację, której nie posiada interpretowany element. Na przykład, hierarchię interpretowalnych obiektów zawierałaby klasę **VariableExpression** reprezentującą zmienną. Ale zmienna sama z siebie nie podlega interpretacji, do tego potrzebny jest kontekst, w którym nazwom zmiennych przyporządkowane są wartości. W prostym przykładzie kontekst zawierałby więc właśnie mapowanie zmiennych lokalnych na wartości, w bardziej rozbudowanym mógłby mieć jeszcze mapowanie identyfikatorów zmiennych globalnych na wartości.

Przy takich założeniach interpretacja konkretnego typu wyrażenia jest więc zwykle łatwa, na przykład:

```
public class Context
{
    // tu słownik - mapowanie nazw zmiennych na wartosci
    private Dictionary<string, double> _localVariables =
        new Dictionary<string, double>();

    public double Get(string variableName)
    {
```

```

        if ( _localVariables.ContainsKey( variableName ) )
        {
            return _localVariables[variableName];
        }
        else
        {
            throw new ArgumentException();
        }
    }

    public void Put(string variableName, double variableValue)
    {
        if ( _localVariables.ContainsKey( variableName ) )
        {
            _localVariables.Remove(variableName);
        }

        _localVariables.Add(variableName, variableValue);
    }
}

public abstract class AbstractExpression
{
    public abstract double Interpret(Context context);
}

public class BinExpression : AbstractExpression
{
    public AbstractExpression Left;
    public AbstractExpression Right;

    public string Operator;

    /// <summary>
    /// Interpretacja rekursywna, delegująca interpreter
    /// do przeciążonej metody węzła
    /// </summary>
    public override double Interpret(Context context)
    {
        switch ( Operator )
        {
            case "+":
                return
                    this.Left.Interpret(context) +
                    this.Right.Interpret(context);

            default:
                throw new ArgumentException();
        }
    }
}

public class VariableExpression : AbstractExpression
{
    public string VariableName;

    public override double Interpret(Context context)
    {

```

```
        return context.Get(VariableName);
    }
}

class Program
{
    static void Main(string[] args)
    {
        // reprezentuj drzewo wyrażenia x + y
        var expression =
            new BinExpression()
            {
                Left = new VariableExpression() { VariableName = "x" },
                Right = new VariableExpression() { VariableName = "y" },
                Operator = "+"
            };

        // kontekst
        var context = new Context();
        context.Put("x", 1);
        context.Put("y", 2);

        Console.WriteLine(expression.Interpret(context));
    }
}
```

## 5 Visitor

Motto: definiowanie nowej operacji bez modyfikowania interfejsu  
Kojarzyć: ExpressionVisitor z biblioteki standardowej .NET

Motywacja: **Interpreter** pokazuje że na strukturze rekursywnej można bez trudu zaimplementować jakąś operację (tu: interpretację) zgrabnie wykorzystując polimorfizm. Problem w podejściu Interpretera polega na tym że dodanie operacji oznacza konieczność dodania jej „na trwałe” do całej hierarchii.

W przykładzie jest to jedna metoda, Interpret, ale co gdyby takich metod miało być więcej?

Niech to będzie Print, do wypisywania, PrettyPrint do alternatywnego wypisywania, SymbolicInterpret, w którym możliwa jest interpretacja symboliczna (czyli podstawienie za token nie tylko konkretnej wartości ale również wartości symbolicznej), itd.

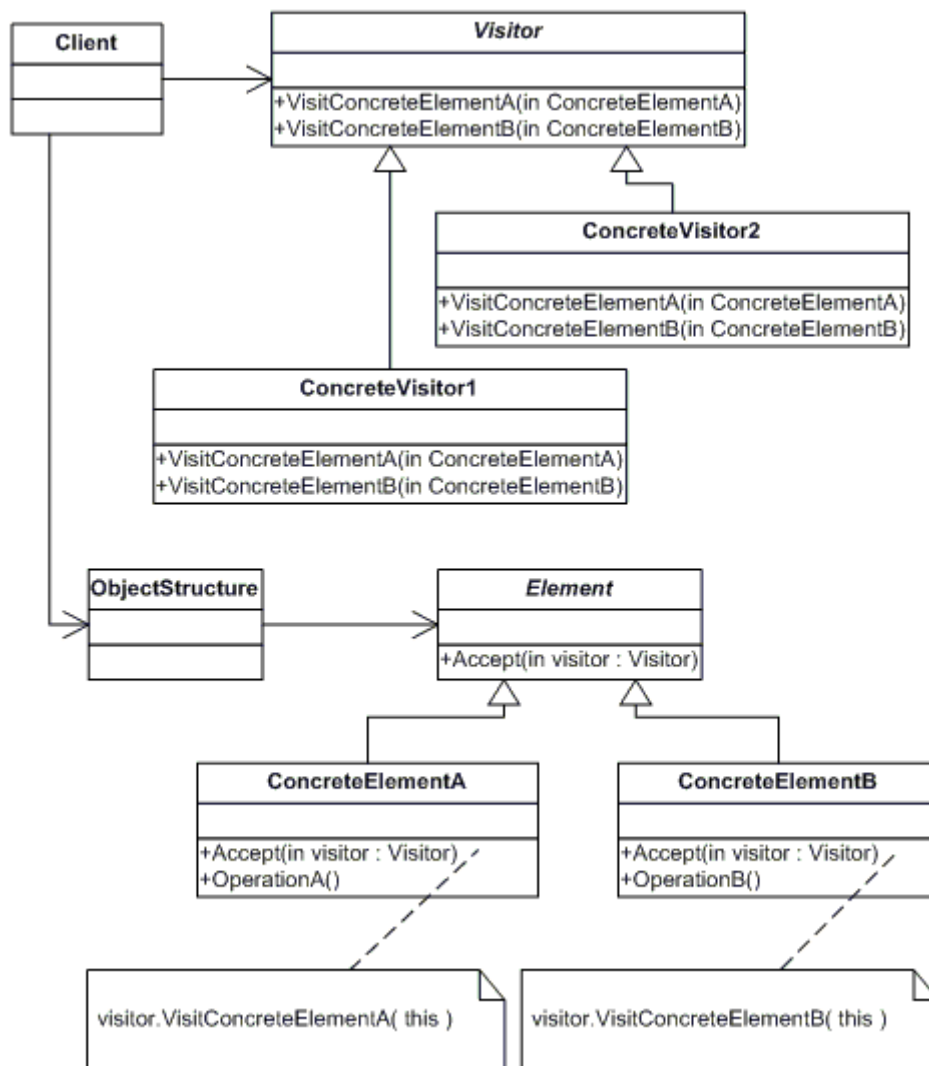
Każda taka nowa operacja to nowa funkcja w całej rekursywnej strukturze. Funkcję trzeba dodać do klasy bazowej, a potem – wielokrotnie przeciążyć ją w podklasach. Pomysł Visitora wychodzi z założenia, że można to zrobić lepiej.

Zamiast tego proponuje się tu wyniesienie operacji „na zewnątrz”, przez zdefiniowanie klasy która jest abstrakcją operacji na strukturze rekursywnej (tak zwanego Visitora) i związanie tej abstrakcji ze strukturą kompozytową raz, bez znajomości szczegółów tego do czego ta operacja będzie służyć.

Za szczegółową implementację operacji odpowiada wtedy zewnętrzna w stosunku do struktury kompozytowej klasa, która jest implementacją tejże abstrakcji (konkretny Visitor). W takim podejściu nowa operacja nie oznacza już nowej metody w całej hierarchii kompozytu, a jedynie nową klasę, implementującą abstrakcję zewnętrznej operacji.

To dobra perspektywa, w której wzorec Visitor rozwiązuje konkretny problem, zaobserwowany na przykładzie wzorca Interpreter.





## 5.1 Implementacja strukturalna

Uwaga! To pierwszy tak złożony wzorec jaki poznajemy, dlatego warto mieć pod ręką przykład implementacji strukturalnej.

```

class Program
{
    static void Main( string[] args )
    {
        CompositeStructure cs = new CompositeStructure();
        cs.AddElement( new ConcreteElementA() );
        cs.AddElement( new ConcreteElementA() );
        cs.AddElement( new ConcreteElementB() );

        ConcreteVisitor1 cv1 = new ConcreteVisitor1();
        ConcreteVisitor2 cv2 = new ConcreteVisitor2();

        cs.Accept( cv1 );
        cs.Accept( cv2 );

        Console.ReadLine();
    }
}

public abstract class Element
{
    public abstract void Accept( Visitor v );
}

public class ConcreteElementA : Element
{
    public override void Accept( Visitor v )
    {
        v.VisitConcreteElementA( this );
    }
}

public class ConcreteElementB : Element
{
    public override void Accept( Visitor v )
    {
        v.VisitConcreteElementB( this );
    }
}

public class CompositeStructure
{
    private List<Element> elements = new List<Element>();
    public void AddElement( Element e )
    {
        this.elements.Add( e );
    }

    public void Accept( Visitor v )
    {
        foreach (var e in elements)
            e.Accept( v );
    }
}

public abstract class Visitor
{

```

```

    public abstract void VisitConcreteElementA( ConcreteElementA elem );
    public abstract void VisitConcreteElementB( ConcreteElementB elem );
}

public class ConcreteVisitor1 : Visitor
{
    public override void VisitConcreteElementA( ConcreteElementA elem )
    {
        Console.WriteLine( "cv1 visiting A" );
    }

    public override void VisitConcreteElementB( ConcreteElementB elem )
    {
        Console.WriteLine( "cv1 visiting B" );
    }
}

public class ConcreteVisitor2 : Visitor
{
    public override void VisitConcreteElementA( ConcreteElementA elem )
    {
        Console.WriteLine( "cv2 visiting A" );
    }

    public override void VisitConcreteElementB( ConcreteElementB elem )
    {
        Console.WriteLine( "cv2 visiting B" );
    }
}

```

Podstawowy problem tak skonstruowanej struktury to jej złożoność. Studenci, którzy już rozumieją na czym polega Visitor pytają często „po co jest cała część akceptowania struktury Visitora w elementach struktury kompozytowej? Dlaczego nie wystarczy sama hierarchia Visitorów?”

Odpowiedzi na to pytanie udzielimy na wykładzie – wszystko zależy od tego **gdzie** znajduje się wiedza o tym jak należy przeglądać strukturę kompozytu. Sam Visitor potrafi obsłużyć różne typy elementów struktury kompozytowej, ale czy nawigacja **pomiędzy** elementami to odpowiedzialność Visitora czy raczej samego kompozytu?

W praktyce dobrze jest rozumieć różnice między tymi dwoma implementacjami i umieć refaktoryzować kod z jednej implementacji do drugiej. Dlatego prześledzimy oba warianty na przykładzie kompozytu - drzewa binarnego.

Punktem wyjścia do obu implementacji jest ten sam początkowy kod:

```

public class MainClass
{
    Tree root = new TreeNode()
    {
        Left = new TreeNode()
        {
            Left = new TreeLeaf() { Value = 1 },
            Right = new TreeLeaf() { Value = 2 },

```

```

    },
    Right = new TreeLeaf() { Value = 3 }
};
}

public abstract class Tree
{
}

public class TreeNode : Tree
{
    public Tree Left { get; set; }
    public Tree Right { get; set; }
}

public class TreeLeaf : Tree
{
    public int Value { get; set; }
}

```

## 5.2 Visitor nie znający szczegółów implementacji struktury kompozytowej

Jeżeli odpowiedzialność za przeglądanie struktury kompozytu spoczywa na samym kompozycie, to Visitory nie muszą w ogóle wiedzieć jak wygląda „wnętrze” struktury kompozytowej i wtedy mamy taki podział odpowiedzialności między strukturą kompozytową a Visitorami jak na diagramie strukturalnym, ponieważ implementacja „odwiedzania” struktury, zawarta w kompozycie, deleguje „odwiedzanie” poszczególnych typów struktury do odpowiednich metod visitora (czyli mówiąc technicznie, potrzebne są zarówno metody **Visit** w Visitorach jak i metody **Accept** w elementach struktury kompozytowej).

```

public abstract class Tree
{
    public virtual void Accept( TreeVisitor visitor )
    {
    }
}

public abstract class TreeVisitor
{
    public abstract void VisitNode( TreeNode node );
    public abstract void VisitLeaf( TreeLeaf leaf );
}

```

Pełny przykład:

```

public class MainClass
{
    public static void Main()
    {
        Tree root = new TreeNode()

```

```

        {
            Left = new TreeNode()
            {
                Left = new TreeLeaf() { Value = 1 },
                Right = new TreeLeaf() { Value = 2 },
            },
            Right = new TreeLeaf() { Value = 3 }
        };

        SumTreeVisitor visitor = new SumTreeVisitor();

        root.Accept( visitor );

        Console.WriteLine( "Suma wartości na drzewie to {0}", visitor.Sum );
        Console.ReadLine();
    }
}

public abstract class Tree
{
    public virtual void Accept( TreeVisitor visitor )
    {
    }
}

public class TreeNode : Tree
{
    public Tree Left { get; set; }
    public Tree Right { get; set; }

    public override void Accept( TreeVisitor visitor )
    {
        visitor.VisitNode( this );

        if ( Left != null )
            Left.Accept( visitor );
        if ( Right != null )
            Right.Accept( visitor );
    }
}

public class TreeLeaf : Tree
{
    public int Value { get; set; }

    public override void Accept( TreeVisitor visitor )
    {
        visitor.VisitLeaf( this );
    }
}

public abstract class TreeVisitor
{
    public abstract void VisitNode( TreeNode node );
    public abstract void VisitLeaf( TreeLeaf leaf );
}

public class SumTreeVisitor : TreeVisitor

```

```

{
    public int Sum { get; set; }

    public override void VisitNode( TreeNode node )
    {

    }

    public override void VisitLeaf( TreeLeaf leaf )
    {
        this.Sum += leaf.Value;
    }
}

```

Jak widać, wiedza o szczegółach struktury kompozytowej jest w samej strukturze kompozytowej – tu jest ona częścią implementacji metody **Accept** dla węzła drzewa binarnego, gdzie oprócz wywołania metody VisitNode Visitora dla bieżącego węzła, następuje rekurencyjne wywołanie akceptowania Visitora dla lewego i prawego poddrzewa.

Warto zwrócić uwagę na prostotę implementacji konkretnego Visitora – jego zadaniem jest rzeczywiście już tylko dostarczenie logiki przetwarzania poszczególnych typów elementów struktury kompozytowej.

### 5.3 Visitor znający szczegóły implementacji struktury kompozytowej

Jeżeli jednak odpowiedzialność za przeglądanie struktury kompozytu spoczywa na Visitorze – to wydaje się że nie ma w ogóle potrzeby aby istniała zależność od struktury kompozytowej do Visitorów (czyli mówiąc technicznie, nie są potrzebne metody **Accept** w elementach struktury kompozytowej).

Takie Visitory są jednak bardziej złożone (bo są obciążone tą dodatkową odpowiedzialnością wiedzy o wnętrzu struktury kompozytowej; w dodatku są „przywiązane” do szczegółów implementacji tego wnętrza – nie da się zmienić struktury kompozytu bez wymiany implementacji Visitorów), choć jak pokazują różne przykłady (m.in. **ExpressionVisitor** z biblioteki standardowej), bywają chętniej wybierane przy implementacji.

```

public class MainClass
{
    public static void Main()
    {
        Tree root = new TreeNode()
        {
            Left = new TreeNode()
            {
                Left = new TreeLeaf() { Value = 1 }
                Right = new TreeLeaf() { Value = 2 },
            },
            Right = new TreeLeaf() { Value = 3 }
        };

        SumTreeVisitor visitor = new SumTreeVisitor();

        visitor.Visit( root );
    }
}

```

```

        Console.WriteLine( "Suma wartości na drzewie to {0}", visitor.Sum );
        Console.ReadLine();
    }
}

public abstract class Tree
{
}

public class TreeNode : Tree
{
    public Tree Left { get; set; }
    public Tree Right { get; set; }
}

public class TreeLeaf : Tree
{
    public int Value { get; set; }
}

public abstract class TreeVisitor
{
    // ta metoda nie jest potrzebna ale ułatwia korzystanie z Visitora
    public void Visit( Tree tree )
    {
        if (tree is TreeNode)
            this.VisitNode( (TreeNode)tree );
        if (tree is TreeLeaf)
            this.VisitLeaf( (TreeLeaf)tree );
    }

    public virtual void VisitNode( TreeNode node )
    {
        // tu wiedza o odwiedzaniu struktury
        if ( node != null )
        {
            this.Visit( node.Left );
            this.Visit( node.Right );
        }
    }

    public virtual void VisitLeaf( TreeLeaf leaf )
    {
    }
}

public class SumTreeVisitor : TreeVisitor
{
    public int Sum { get; set; }

    public override void VisitLeaf( TreeLeaf leaf )
    {
        // metoda z klasy bazowej musi być wywołana przy przeciążeniu
        // bo w klasie bazowej Visitora jest wiedza o odwiedzaniu
        // struktury kompozytu
        base.VisitLeaf( leaf );
    }
}

```

```
        this.Sum += leaf.Value;
    }
```

## 5.4 Przykład z biblioteki standardowej

Biblioteka standardowa platformy .NET zawiera ładny przykład implementacji **Visitatora** – to bazowa implementacja Visitatora dla wyrażeń LINQ. Ta implementacja to Visitor który **zna** strukturę odwiedzanego obiektu, dlatego nie ma asocjacji od klasy **Expression** na których operuje Visitor do klasy Visitatora, wystarczy powiązanie od strony Visitatora do klasy Expression.

Przeciążeniu podlega całkiem sporo metod, odpowiadających różnym typom wyrażeń – dla drzewa binarnego mieliśmy tylko dwa konkretne typy struktury kompozytowej (węzeł lub liść), dla wyrażenia jest to m.in. wyrażenie binarne, stała, wyrażenie funkcyjne itd.

Przykład:

```
class Program
{
    static void Main(string[] args)
    {
        Expression<Func<int, int>> f = n => 4 * (7 + n);

        PrintExpressionVisitor v = new PrintExpressionVisitor();
        v.Visit(f);

        Console.ReadLine();
    }
}

public class PrintExpressionVisitor : ExpressionVisitor
{
    protected override Expression VisitBinary( BinaryExpression expression )
    {
        Console.WriteLine("{0} {1} {2}",
            expression.Left, expression.NodeType, expression.Right);

        return base.VisitBinary(expression);
    }

    protected override Expression VisitLambda<T>( Expression<T> expression )
    {
        Console.WriteLine("{0} -> {1}",
            expression.Parameters.Aggregate(string.Empty, (a, e) => a += e),
            expression.Body );

        return base.VisitLambda<T>( expression );
    }
}
```

## 5.5 Visitor a Double Dispatch

O wzorcu Visitor mówi się również w kontekście mechanizmu tzw. **Double Dispatch**

[https://en.wikipedia.org/wiki/Double\\_dispatch](https://en.wikipedia.org/wiki/Double_dispatch)



Chodzi o to że w klasycznym kodzie obiektowym wybór metody która zostanie wywołana przez

```
cs.Accept( cv );
```

zależy tylko od typu **cs**. To klasyczne zachowanie obiektu to **Single Dispatch** (zwykła funkcja wirtualna). Z kolei przy Double Dispatch, to jaki kod zostanie wywołany zależy zarówno od typu **cs** jak i od typu **cv**.