# VISITOR

# MOTIVATIONAL EXAMPLE

# EXPRESSION EVALUATION

```
abstract class Expression {
    public abstract int Eval();
}

class ValueExpression : Expression {
    public int Value { get; }
    public override int Eval() {
        return Value;
    }
}

abstract class UnaryExpression : Expression {
    Expression Child { get; }
}

sealed class UnaryMinusExpression : UnaryExpression {
    public override int Eval() {
        return -Child.Eval();
    }
}

abstract class BinaryExpression : Expression {
    public Expression LeftChild { get; }
    public Expression RightChild { get; }
}
```

```
sealed class AddExpression : BinaryExpression {
    public override int Eval() {
        return LeftChild.Eval() + RightChild.Eval();
    }
}

sealed class MinusExpression : BinaryExpression {
    public override int Eval() {
        return LeftChild.Eval() - RightChild.Eval();
    }
}

sealed class MultiplyExpression : BinaryExpression {
    public override int Eval() {
        return LeftChild.Eval() * RightChild.Eval();
    }
}

sealed class DivideExpression : BinaryExpression {
    public override int Eval() {
        return LeftChild.Eval() / RightChild.Eval();
    }
}
```

# ADDING NEW FUNCTION

```
abstract class Expression {
    public abstract int Eval();
    public abstract double EvalDouble();
}

class ValueExpression : Expression {
    public int Value { get; }

    public override int Eval() {
        return Value;
    }

    public override double EvalDouble() {
        return Value;
    }
}
```

```
sealed class AddExpression : BinaryExpression {
    public override int Eval() {
        return LeftChild.Eval() + RightChild.Eval();
    }

    public override double EvalDouble() {
        return LeftChild.Eval() + RightChild.Eval();
    }
}
```

# ADDING MORE FUNCTIONS

```
abstract class Expression {
    public abstract int Eval();
    public abstract double EvalDouble();
    public abstract string PrintInInfixNotation();
}

class ValueExpression : Expression {
    public int Value { get; }

    public override int Eval() {
        return Value;
    }

    public override double EvalDouble() {
        return Value;
    }

    public override string PrintInInfixNotation() {
        return Value.ToString();
    }
}
```

```
sealed class AddExpression : BinaryExpression {
    public override int Eval() {
        return LeftChild.Eval() + RightChild.Eval();
    }

    public override double EvalDouble() {
        return LeftChild.Eval() + RightChild.Eval();
    }

    public override string PrintInInfixNotation() {
        // some code
    }
}
```

# THE SOLUTION – 1st STEP

```
class Algorithm
{
    public void Process(UnaryMinusExpression e) {...}
    public void Process(AddExpression e) {...}
    public void Process(MinusExpression e) {...}
    ...

    public void Start(Expression e)
    {
        Process(e);
    }

    // public void Process(Expression e) {...}

}
```

- ❑ Separate the algorithm from the object structure
- ❑ Add method Process() for each expression
- ❑ Add entry method

# THE SOLUTION – 1st STEP

```
class Algorithm
    {
        public void Process(UnaryMinusExpression e) {...}
        public void Process(AddExpression e) {...}
        public void Process(MinusExpression e) {...}
        ...

        public void Start(Expression e)
        {
            if (e is UnaryMinusExpression)
                Process((UnaryMinusExpression)e);
            else if (e is AddExpression)
                Process((AddExpression)e);
            else if (...){
                ...
            }
            else
                throw new NotImplementedException();
        }

        // public void Process(Expression e) {...}

    }
```

- What happens if we add new ModuloExpression?

# THE SOLUTION – 2ND STEP

```
abstract class Expression {
    public abstract void Method(Algorithm alg);
}

sealed class UnaryMinusExpression : UnaryExpression {
    public abstract void Method(Algorithm alg);{
        alg.Process(this);
    }
}

sealed class AddExpression : BinaryExpression {
    public abstract void Method(Algorithm alg);{
        alg.Process(this);
    }
}

sealed class MinusExpression : BinaryExpression {
    public abstract void Method(Algorithm alg);{
        alg.Process(this);
    }
}
```

```
class Algorithm
{
    public void Process(UnaryMinusExpression e) {...}
    public void Process(AddExpression e) {...}
    public void Process(MinusExpression e) {...}
    ...

    public void Start(Expression e)
    {
        if (e is
            Pro
        else if (
            Process
        else if (
            ...
        }
        else
            throw new NotImplementedException();
    }
}
```

# THE SOLUTION

```
interface IVisitor {
    void Visit(ValueExpression expression);
    void Visit(UnaryMinusExpression expression);
    void Visit(AddExpression expresson);
    ...
}

class Visitor1 : IVisitor {
    public void Visit(ValueExpression expression) {
        //some code
    }
    public void Visit(UnaryMinusExpression expression) {
        //some code
    }
    public void Visit(AddExpression expresson) {
        //some code
    }
    ...
}
```

```
abstract class Expression {
    public abstract void Accept(IVisitor visitor);
}

class ValueExpression : Expression {
    public int Value { get; }
    public override void Accept(IVisitor visitor) {
        visitor.Visit(this);
    }
}

sealed class UnaryMinusExpression : UnaryExpression {
    public override void Accept(IVisitor visitor) {
        visitor.Visit(this);
    }
}

sealed class AddExpression : BinaryExpression {
    public override void Accept(IVisitor visitor) {
        visitor.Visit(this);
    }
}

sealed class MinusExpression : BinaryExpression {
    public override void Accept(IVisitor visitor)
    {
        visitor.Visit(this);
    }
}
```

# DISPATCH

- **Dynamic dispatch** (or *virtual method call)*
  - Selecting implementation of a *polymorphic* method or function at *run time*
  - A prime characteristic of object-oriented languages (C++, Java, C#, …)
  - Used for dynamic loading of DLL files (Windows) or SO files (Unix, Linux)
- **Static dispatch** (or *early binding*)
  - Fully resolved selection of implementation of method or function during compile time
  - Overloading of function

```
abstract class BinaryExpression : Expression {
    public Expression LeftChild { get; }
    public Expression RightChild { get; }
}

sealed class AddExpression : BinaryExpression {
    public override int Eval(){
        return LeftChild.Eval() + RightChild.Eval();
    }
}
```

```
class Algorithm
{
    public void Process(UnaryMinusExpression e) {...}
    public void Process(AddExpression e) {...}
    public void Process(MinusExpression e) {...}
    ...

    public void Start(Expression e)
    {
        Process(e);
    }

    // public void Process(Expression e) {...}

}
```

# SINGLE DISPATCH

- **Single dispatch**
  - The operation that is executed depends on the name of the request, and the type of the receiver
  - Supported by languages like Java, JavaScpript, C++, Python, …

```
public interface DiscountPolicy {
    double discount(Order order);
}

public class FlatDiscountPolicy implements DiscountPolicy {
    @Override
    public double discount(Order order) {
        return 0.01;
    }
}

public class CostDiscountPolicy implements DiscountPolicy {
    @Override
    public double discount(Order order) {
        if(order.totalCost() > 500)
            return 0.1;
        else
            return 0.0
    }
}
```

```
Order orderWorth501 = orderWorthNDollars(501);

DiscountPolicy flatPolicy = new FlatDiscountPolicy();
System.out.println(flatPolicy.discount(orderWorth501));

costPolicy = new CostDiscountPolicy();
System.out.println(costPolicy.discount(orderWorth501));
```

```
Output:
0.01
0.1
```

# DOUBLE DISPATCH

- The operation executed depends on the name of the request, and the type of TWO receivers (the type of visitor and the type of element it visits)
- Use cases:
  - Sorting a mixed set of objects
  - Adaptive collision algorithm
  - Lock and key systems

```
interface IVisitor {
    void Visit(AddExpression e);
    void Visit(ValueExpression e);
}
class Visitor1 : IVisitor {
    public void Visit(AddExpression e) {
        Console.WriteLine("Add expression");
    }
    public void Visit(ValueExpression e) {
        Console.WriteLine("Value expression");
    }
}
```

```
abstract class Expression {
    public abstract void Accept(IVisitor visitor);
}
sealed class AddExpression : BinaryExpression {
    public override void Accept(IVisitor visitor) {
        visitor.Visit(this);
    }
sealed class ValueExpression : Expression {
    public override void Accept(IVisitor visitor) {
        visitor.Visit(this);
    }
}
```

```
 static void Main(string[] args) {
    IVisitor visitor = new Visitor1();
    Expression add = new AddExpression();
    Expression value = new ValueExpression();

    add.Accept(visitor); // prints "Add expression"
    value.Accept(visitor);  // prints "Value expression"
}
```

# DOUBLE DISPATCH IN C++

- **std::visit** way to examine the alternatives of a given **std:variant**
- std::visit requires that every alternative (data type) in the variant is supported by the visitor passed to std::visit

```
Output:
banana apple grapes

yellow red green
```

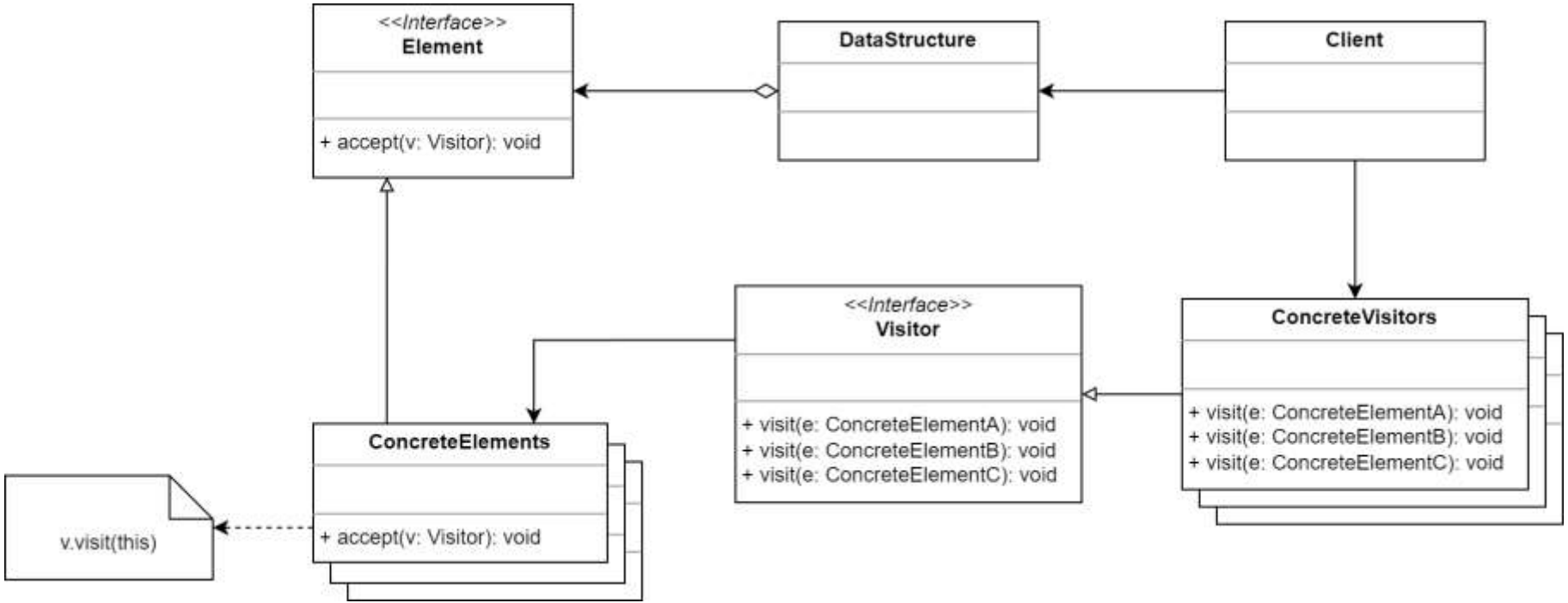
```
struct Apple { };
struct Banana { };
struct Grapes { };

struct VisitorNames {
    std::string operator()(Apple&) { return "apple"; }
    std::string operator()(Banana&) { return "banana"; }
    std::string operator()(Grapes&) { return "grapes"; }
};

struct VisitorColor {
    std::string operator()(Apple&) { return "red"; }
    std::string operator()(Banana&) { return "yellow"; }
    std::string operator()(Grapes&) { return "green"; }
};

int main(){
    std::vector<std::variant<Apple, Banana, Grapes>>
        fruits = { Banana(), Apple(), Grapes()};

    for (auto f : fruits) {
        std::cout << std::visit(VisitorNames(), f) << ' ';
    }

    std::cout << '\n';

    for (auto f : fruits) {
        std::cout << std::visit(VisitorColor(), f) << ' ';
    }
}
```

# SPECIFICATION

“Represents an operation to be performed on elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.”

—The Gang of Four

# CLASS DIAGRAM



# ELEMENTS

## CLIENT

Calls a dispatching operation accept(visitor) on a top-level element
Is not aware of all the concrete elements

## VISITOR

Declares a set of visiting methods that take concrete elements of an object structure as arguments

## CONCRETE VISITOR

Implements several versions of the same behaviours, tailored for the different concrete element classes

## OBJECT STRUCTURE

Can enumerate its elements
May either be a composite or a collection such as a list or a set

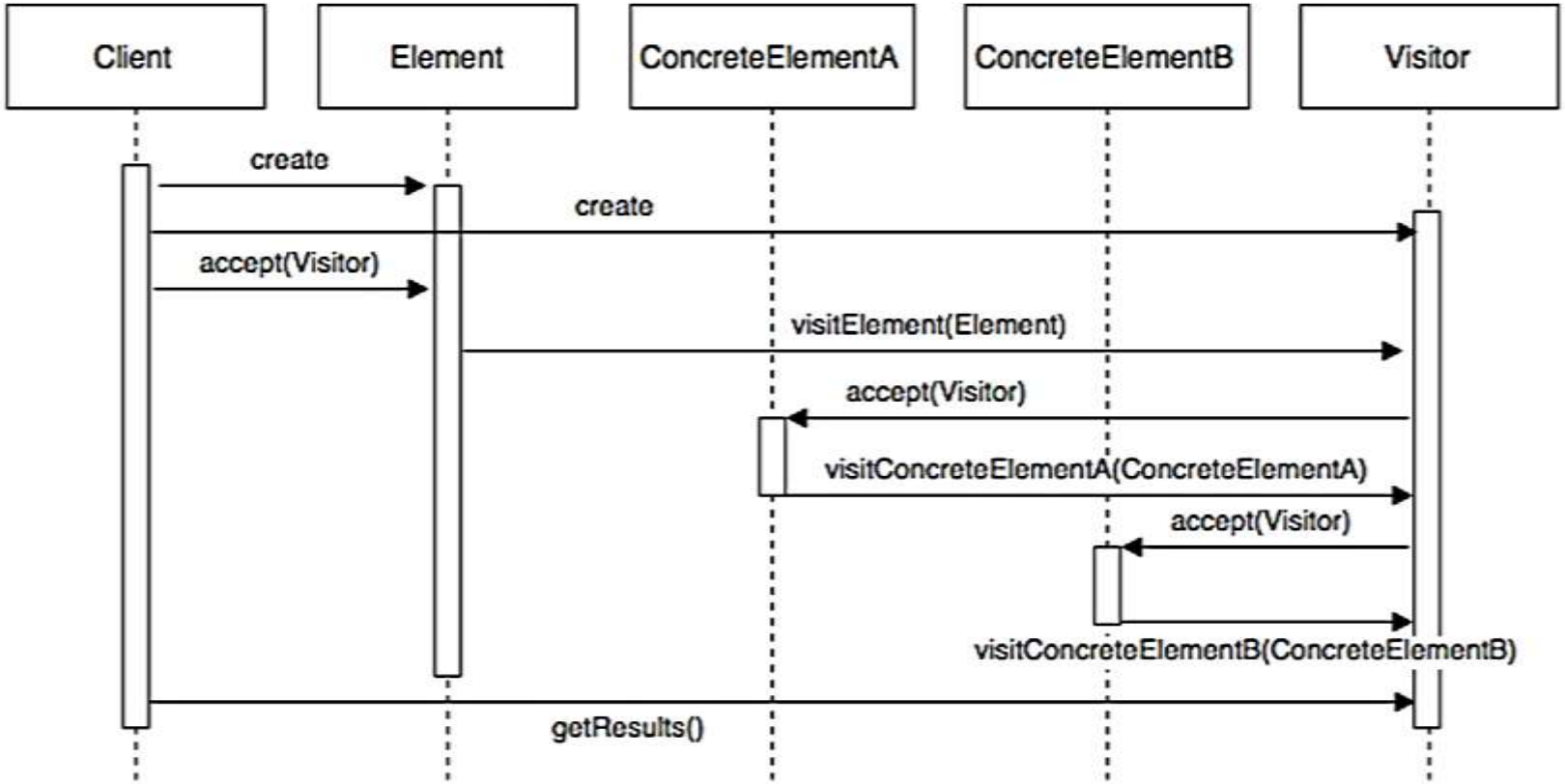
## ELEMENT

Defines the acceptance method with a visitor as an argument

## CONCRETE ELEMENT

Implements the acceptance method with a visitor as an argument to redirect the call to the proper visitor's method

# SEQUENTIAL DIAGRAM



# HOW TO IMPLEMENT

**01** Declare the visitor interface with the “visiting” methods, one per each existing concrete element class

**02** Declare the element interface or add the abstract “acceptance” method to the existing interface

**03** Implement the acceptance methods in all concrete element classes, which extend the element interface

**04** Create a new concrete visitor class and implement all of the visiting methods

**05** The client creates visitor objects and pass them into elements via “acceptance” methods

# IMPLEMENTATION ISSUE

WHO IS RESPONSIBE FOR TRAVERSING THE OBJECT STRUCTURE?

- A visitor must visit each element of the object structure – how does it gets there?
- Where to put responsibility:
  - In the object structure
  - In a separate iterator object
  - In the visitor

- **The object structure** - often responsible
  - Simply iterate over its elements
  - A composite calls Accept method recursively on each of its children
- **Iterator**
  - Depends what is available and efficient
- **In the visitor**
  - Duplicating the traversal code in each concrete visitor for each concrete element
  - Implementation of a complex traversal depending on the operation

# WHEN TO USE

| | |
|---|---|
| Perform operations on one or more object structures containing classes with different interfaces | Execute operation with several variants, which correspond to all target objects |
| Avoid polluting an object structure with many distinct and unrelated operations | The primary classes more focused on their main job |
| A behaviour makes sense only in some objects of the object structure, but not in others | Implement only those visiting methods that accept objects of relevant classes, leaving the rest empty |
| Frequently adding new operations on rarely changed object structure | To add new operation means simply adding a new visitor – you do not need to change each object to define a new operation |

# PROS

- Conforms to **the open/close principle**
  - Adding new operations to existing structures without modifying the object structure
- Conforms to **the single responsibility principle**
  - Implements the visitor pattern in a separate component
- Managing an algorithm from one single location
- Gathers related operation and separates unrelated ones
- **Type safe**
  - Adding new element creates compilation error

# COMS

- Update all visitors each time an element is added or removed from the structure
- Lack of access to the private fields the visitors need to work with
- Can violate the **encapsulation** – the need of public fields and methods
  - The visitor can then modify the concrete elements

# RELATIONS WITH OTHER PATTERNS

## COMMAND

The visitor can execute operations over various objects of different classes and may initiate whatever is appropriate for the kind of object it encounters

## ITERATOR

Iterator can’t work across object structures with different types of elements – can add any type of object to a visitor interface

## COMPOSITE

The visitor can execute an operation over an object structure defined by the composite pattern

# THANK YOU FOR YOUR ATTENTION!

ANY QUESTIONS?

# SOURCES


- https://en.wikipedia.org/wiki/Visitor_pattern
- https://refactoring.guru/design-patterns/visitor
- E. Gamma, R. Helm, R. Johnson, J. Vlissides.Design Patterns Elements of Reusable Object-Oriented Software. *1995*
- https://refactoring.guru/design-patterns/visitor-double-dispatch
- https://en.wikipedia.org/wiki/Dynamic_dispatch
- https://en.wikipedia.org/wiki/Static_dispatch
- https://lukasatkinson.de/2016/dynamic-vs-static-dispatch/
- https://learn.microsoft.com/en-us/dotnet/csharp/advanced-topics/interop/using-type-dynamic?redirectedfrom=MSDN
- https://www.modernescpp.com/index.php/visiting-a-std-variant-with-the-overload-pattern
- https://www.baeldung.com/ddd-double-dispatch
- https://www.youtube.com/watch?v=AFsALrqFy_Q