# VISITOR

Lukáš Holub

# DEFINICE

Návrhový vzor, který **odděluje** algoritmus od objektů, na kterých operuje.

# Motivační příklad

# EXPORTOVÁNÍ GEOMETRIE

```
interface Shape
{
    void Move(int x, int y);
    void Draw();
}

class Dot : Shape
{
    public void Move(int x, int y)
    {
        // Implementation
    }
    public void Draw()
    {
        // Implementation
    }
}
```

```
class Circle : Shape
{
    public void Move(int x, int y)
    {
        // Implementation
    }
    public void Draw()
    {
        // Implementation
    }
}

class Rectangle : Shape
{
    public void Move(int x, int y)
    {
        // Implementation
    }
    public void Draw()
    {
        // Implementation
    }
}
```

# PŘIDÁNÍ FUNKCE

```
interface Shape
{
    void Move(int x, int y);
    void Draw();
    void Export();
}

class Dot : Shape
{
    public void Move(int x, int y) {...}
    public void Draw() {...}
    public void Export() {...}
}

class Circle : Shape
{
    public void Move(int x, int y) {...}
    public void Draw() {...}
    public void Export() {...}
}
```

```
class Rectangle : Shape
{
    public void Move(int x, int y) {...}
    public void Draw() {...}
    public void Export() {...}
}
```

## PROBLÉMY

- Dává smysl mít exportovací funkcionalitu v Shape třídě?
- Co když chceme přidat další funkcionalitu?
- Co když nechceme neustále měnit třídy, jelikož už jsou otestované?

# PŘIDÁME TŘÍDU

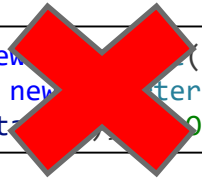
```
class Exporter {
    public void Export(Dot dot) {
        Console.WriteLine("DOT");
    }

    public void Export(Circle circle) {
        Console.WriteLine("CIRCLE");
    }

    public void Export(Rectangle rectangle) {
        Console.WriteLine("RECTANGLE");
    }

    public void Export(Shape shape) {
        Console.WriteLine("UNKNOWN SHAPE");
    }
}
```

```
Shape rectangle = new Rectangle();
Exporter exporter = new Exporter();
exporter.Export(rectangle); // Output: UNKNOWN SHAPE
```



```
Shape[] shapes = new Shape[] {
    new Dot(),
    new Circle(),
    new Rectangle()
};
Exporter exporter = new Exporter();
foreach (Shape shape in shapes) {
    if (shape is Dot) {
        exporter.Export((Dot)shape);
    }
    else if (...) {
        ...
    }
    else {
        exporter.Export(shape);
    }
}
```

Co když přidáme novou třídu?

# DISPATCH

= problém párování dat a kódu, který na nich zavolat

## Static Dispatch

- Forma polymorfismu vyřešena během **compile timu**
- Templates v C++, generické programování, overload funkcí

## Dynamic Dispatch

- Forma polymorfismu vyřešena během **run timu**
- **Virtuální funkce**
- Single x Multiple - výběr je založen na základě jednoho, nebo více objektů
- Typicky v OOP jazycích
  - Single Dispatch -> C++, C#, ...
  - Multiple Dispatch -> CLOS (Common Lisp Object System), Julia

# SINGLE DISPATCH

= zavolaná metoda je vybrána na základě **jednoho** typu

```
class Program
{
    static void Speak(Pet pet)
    {
        pet.Speak();
    }
    static void Main(string[] args)
    {
        Pet dog = new Dog();
        Pet cat = new Cat();
        Speak(dog);
        Speak(cat);
    }
}
```

```
abstract class Pet
{
    public abstract void Speak();
}

class Dog : Pet
{
    public override void Speak()
    {
        Console.WriteLine("Woof!");
    }
}

class Cat : Pet
{
    public override void Speak()
    {
        Console.WriteLine("Meow!");
    }
}
```

```
Output:
Woof!
Meow!
```

# 🔀 DOUBLE DISPATCH

- zavolaná metoda je vybrána na základě **dvou** typů
- Visitor ho implementuje

## PŘÍKLAD V C++

- **std::visit** je způsob, jak zkoumat alternativy dané **std:variant**
- std::visit potřebuje, aby každá alternativa (datový typ) ve variantě byl podporován visitorem předávaným do std::visit

```
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"; }
};


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

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

```
Output:
banana apple grapes
```

# 🔀 DOUBLE DISPATCH PRO PŘÍKLAD

KROK 1:

- Vytvořit Visitor **interface** s Visit metodou pro **každý** konkrétní element v programu (Dot, Circle, Rectangle)

```
interface IVisitor {
    void Visit(Dot dot);
    void Visit(Circle circle);
    void Visit(Rectangle rectangle);
}
```

# 🔀 DOUBLE DISPATCH PRO PŘÍKLAD

KROK 2:

- Vytvořit interface pro element (Shape)
- Interface by měl mít metodu Accept, který přijímá Visitor interface jako argument

```
interface Shape
{
    void Move(int x, int y);
    void Draw();
    void Accept(IVisitor visitor);
}
```

# 🔀 DOUBLE DISPATCH PRO PŘÍKLAD

KROK 3:

- Implementovat všechny Accept metody pro konkrétní element třídy (Dot, Circle, Rectangle)
- Accept metoda pouze přesměruje volání do Visit metody Visitora

```
class Dot : Shape
{
    // ...
    public void Accept(IVisitor visitor)
    {
        visitor.Visit(this);
    }
}
```

```
class Circle : Shape
{
    // ...
    public void Accept(IVisitor visitor)
    {
        visitor.Visit(this);
    }
}
```

```
class Rectangle : Shape
{
    // ...
    public void Accept(IVisitor visitor)
    {
        visitor.Visit(this);
    }
}
```

# 🔀 DOUBLE DISPATCH PRO PŘÍKLAD

KROK 4:

- Implementovat konkrétní Visitory a implementovat všechny metody interfacu

```
class ExportVisitor : IVisitor {
    public void Visit(Dot dot)
    {
        Console.WriteLine("DOT");
    }

    public void Visit(Circle circle)
    {
        Console.WriteLine("CIRCLE");
    }

    public void Visit(Rectangle rectangle)
    {
        Console.WriteLine("RECTANGLE");
    }
}
```

# 🔀 DOUBLE DISPATCH PRO PŘÍKLAD

KROK 5:

- Client vytváří konkrétní Visitor objekty a posílá je elementům do Accept metody

```
Shape[] shapes = new Shape[] {
    new Dot(),
    new Circle(),
    new Rectangle()
};
ExportVisitor exportVisitor = new ExportVisitor();

foreach (Shape shape in shapes)
{
    shape.Accept(exportVisitor);
}
```

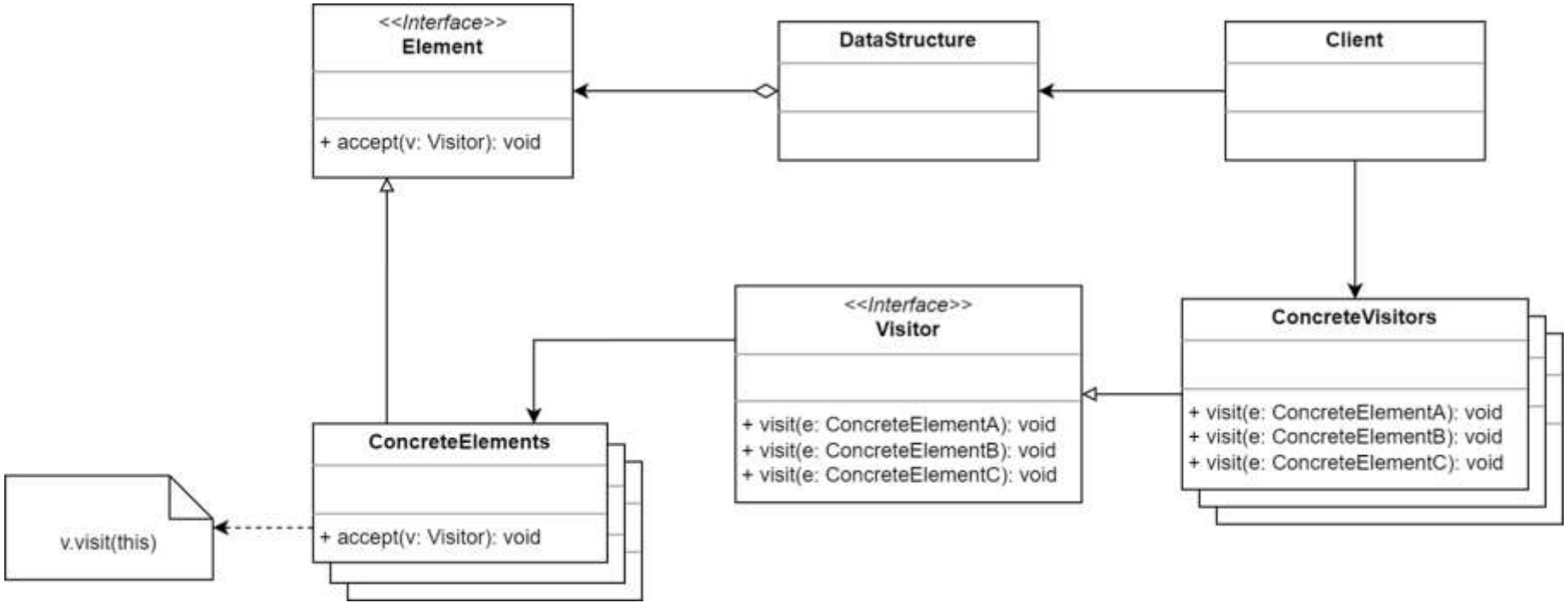
```
Output:
DOT
CIRCLE
RECTANGLE
```

# SPECIFIKACE

“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



# ELEMENTY

## CLIENT

Volá Accept(visitor). Většinou ani neví o všech konkrétních elementech, protože pracuje pouze s interfacem.

## VISITOR

Deklaruje množinu navštěvujících metod, které berou konkrétní elementy objektové struktury jako argumenty.

## CONCRETE VISITOR

Implementuje několik verzí stejného chování, přizpůsobených pro různé konkrétní element třídy.

## OBJECT STRUCTURE

Schopnost enumerovat přes elementy. Může mít high-level interface, které povoluje visitoru navštěvovat prvky. Může být composite nebo nějaká kolekce.

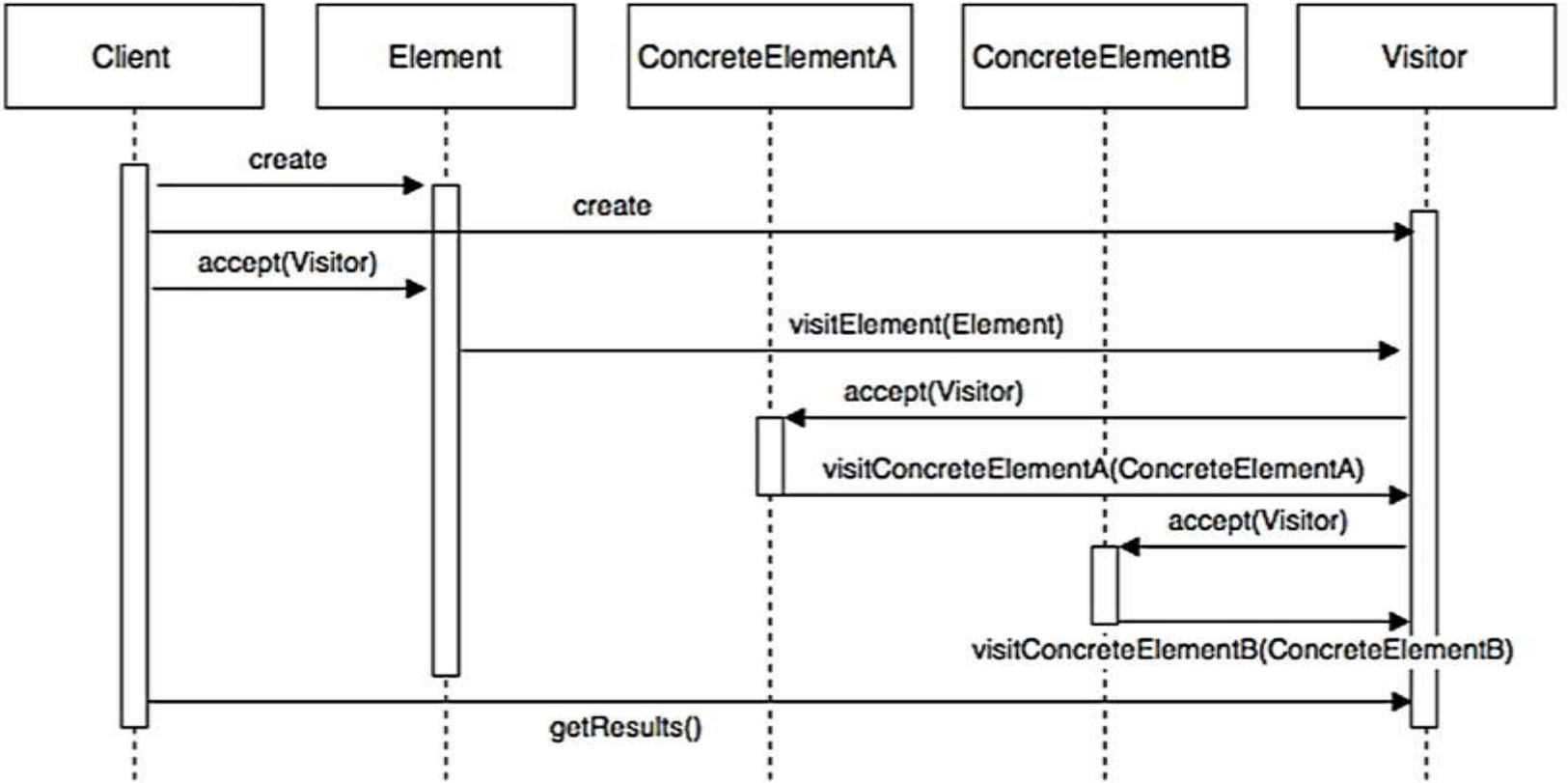
## ELEMENT

Definuje přijímající metodu s visitorem jako argumentem.

## CONCRETE ELEMENT

Definuje přijímající metodu s visitorem jako argumentem, která přesměruje volání do správné metody visitora.

# SEQUENTIAL DIAGRAM



# JAK IMPLEMENTOVAT

**01** Vytvořit Visitor interface s Visit metodou pro každý konkrétní element v programu

**02** Declarovat element interface. Případně přidat abstraktní přijímající metodu do základní třídy existující struktury

**03** Implementovat všechny přijímající metody ve všech konkrétních element třídách, které rozšiřují element interface.

**04** Vytvořit konkrétné visitor a implementovat všechny navštěvující metody.

**05** Klient vytváří visitor objekty a předává je elementům pomocí přijímajících metod.

# IMPLEMENTAČNÍ PROBLÉM

KDO MÁ ZA ÚKOL PROCHÁZET OBJEKTOVOU STRUKTUROU?

- Visitor musí navštívit všechny elementy objektové struktury. Jak se tam ale dostane?
- 3 místa, kam můžeme dát odpovědnost:
  1. Objektové struktuře
  2. Do zvláštního iterátoru
  3. Do visitora

- **Objektová struktura** - často odpovědná
  - Jednoduše přes sebe iteruje a volá přijímající metodu na každém elementu
  - Composite rekurzivně zavolá přijímající metodu na každého potomka
- **Iterator**
  - Hodně záleží na tom, co je dostupné a efektivní
  - Může fungovat jak interní, tak I externí iterator
- **Ve visitoru**
  - Dojde k duplikaci procházejícího kódu v každém konkrétním visitoru pro každý konkrétní element
  - Často pro velmi komplexní procházející kód, který záleží na výsledcích operací objektové struktury

# KDY POUŽÍT

- Máme hodně tříd s odlišnými interfacy, na kterých chceme provádět operace, které záleží na jejich konkrétních třídách
- Chceme zabránit znečišťování tříd mnohou různých a nesouvisejících operací
- Určité chování má smysl jen v některých třídách, ale ne ve zbytku
- Přidávání hodně nových operací na existující málo se měnící objektovou strukturu
  - Pokud se struktura mění velmi často, tak je pravděpodobně lepší operace definovat přímo ve třídách
  - Stačí přidat nového visitora bez změnění objektové struktury
- Pokud máme hodně různých aplikací, ale určité aplikace nepotřebují všechna chování, tak nemusíme předávat visitora

# PROS

- Splňuje **open/closed principle**
  - Nová operace != úprava objektů
- Splňuje **single responsibility principle**
  - Stejné chování ve stejné třídě
- Managování algoritmu z jedné lokace
- Shromažďuje příbuzné operace
- **Type safe**
  - Přidání nového elementu způsobí compilation error
- Akumulace stavu
  - Visitor může sbírat užitečné informace o objektech v objektové struktuře během toho, co ji prochází
    - Bez něj by se stav musel předávat jako dodatečný argument nebo globální proměnná

# CONS

- Nutnost aktualizovat všechny visitory, když se přidá nebo odstraní třída ze struktury
- Zničení enkapsulace
  - Visitor potřebuje dělat svojí práci, tak často potřebuje přístup k internímu stavu elementu – private na public

# VZTAH S JINÝMI VZORY

## ITERATOR

Může být použit společně s iteratorem na procházení data struktur a spouštět operace na elementech

## COMPOSITE

Visitor může spouštět operace přes objektovou strukturu definovanou jako Composite

## INTERPRET

Visitor může být použit na dělání interpretace