7. Шаблон Посетител (Visitor)

ЛЕКЦИОНЕН КУРС: ШАБЛОНИ ЗА ПРОЕКТИРАНЕ

ДОЦ. Д-Р ЕМИЛ ДОЙЧЕВ

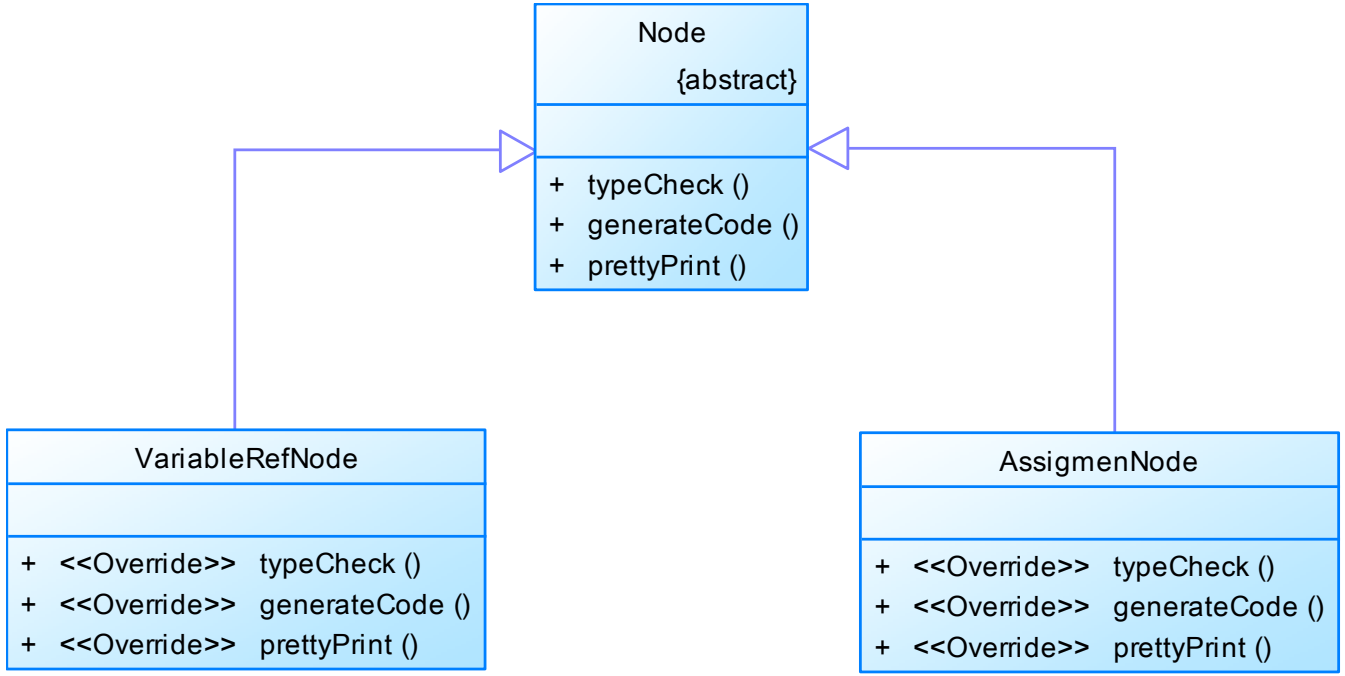
Общи сведения

- ✓ **Вид:** поведенчески за обект
- ✓ **Цел:** Представя операциите, които се извършват върху елементите на обектна структура. Шаблонът *Посетител* позволява добавяне на нова операция, без да се променят класовете на елементите, върху които работи тя.
- ✓ **Известен и като:** *няма алтернативни имена*

Мотивация

- ✓ **Пример:** Компилатор парсира AST (абстрактно синтактично дърво) на сорс код. AST има различни видове възли, като *присвояване*, *променлива, аритметичен израз* и др.
- ✓Операциите, които биха могли да бъдат изпълнявани са разнообразни:
  - Проверка, че всички променливи са дефинирани.
  - Проверка, че на всички променливи е зададена стойност, преди да бъдат използвани.
  - Проверка на типовете.
  - Генериране на код.
  - Форматиране на кода.
- ✓ Тези операции обработват всеки вид възел по различен начин.
- ✓ Единия начин за реализация на такова поведение е да се дефинира всяка операция в специфичния клас за всеки тип възел.

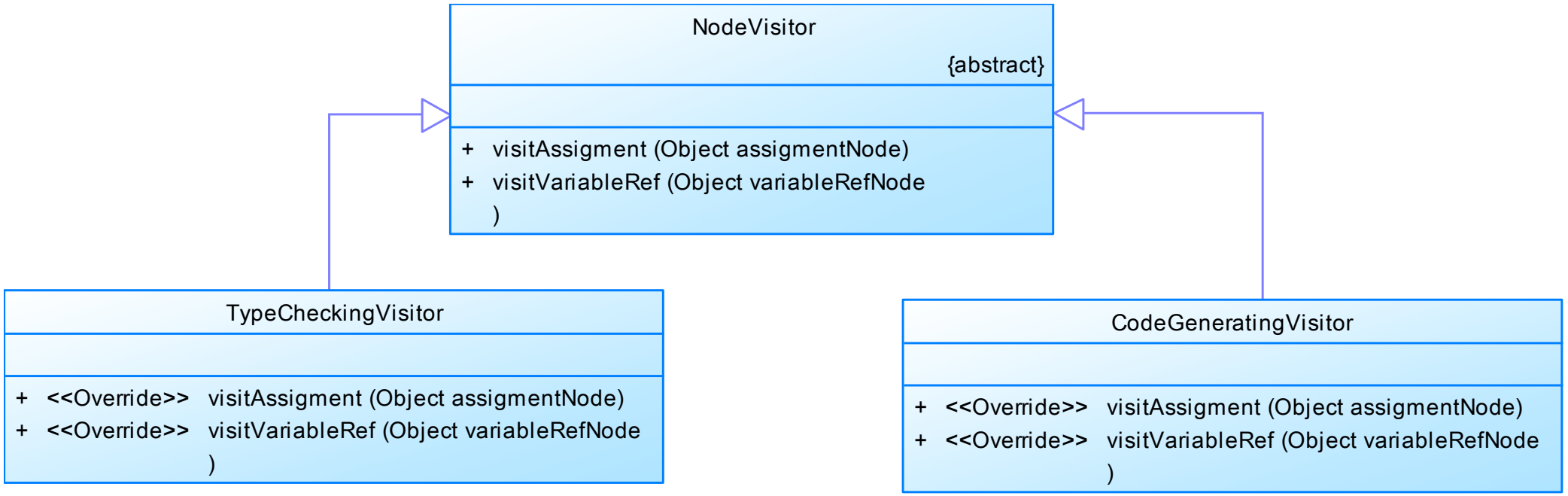
Мотивация


Node
{abstract}
+ typeCheck ()
+ generateCode ()
+ prettyPrint ()
VariableRefNode
+ <<Override>> typeCheck ()
+ <<Override>> generateCode ()
+ <<Override>> prettyPrint ()
AssigmenNode
+ <<Override>> typeCheck ()
+ <<Override>> generateCode ()
+ <<Override>> prettyPrint ()

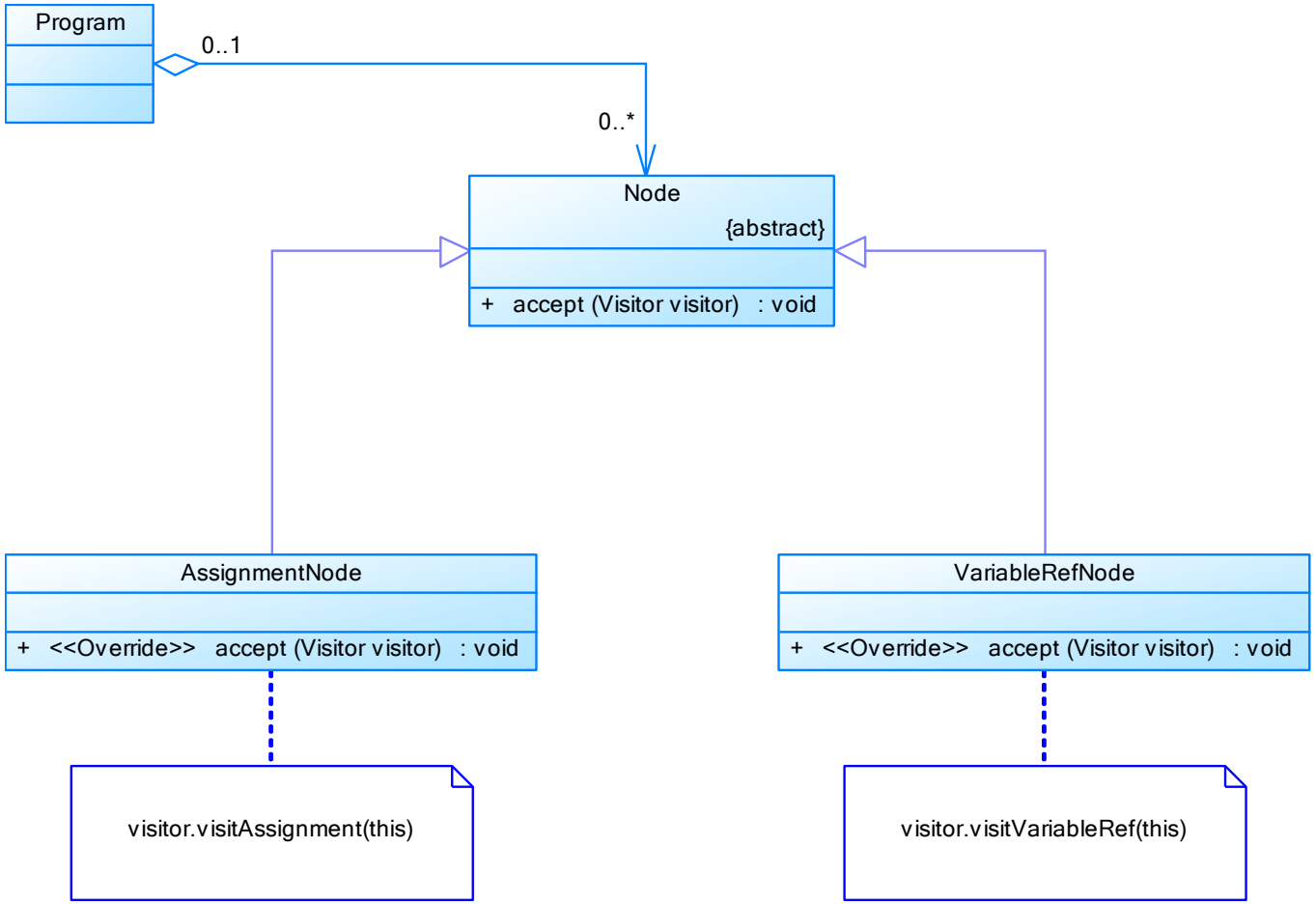
Мотивация

- ✓ **Проблеми:**
  - Добавяне на нова операция изисква промяна във всички класове на възли
  - Объркващо е да има такова разнообразие от операции във всеки клас на възел. Например, смесването на код за проверка на типовете с код за форматиране на кода е трудно за разбиране.
- ✓ **Друго решение:** капсулира се нужната операция в отделен обект, наречен *посетител*. Посетителя „обхожда“ възлите на дървото. Когато възел „приеме“ посетителя всъщност му извиква метод, който приема като аргументи типа на възела и самия възел. Така посетителя изпълнява необходимата функционалност за този конкретен възел.

Мотивация


NodeVisitor
{abstract}
+ visitAssigment (Object assigmentNode)
+ visitVariableRef (Object variableRefNode
)
TypeCheckingVisitor
+ <<Override>> visitAssigment (Object assigmentNode)
+ <<Override>> visitVariableRef (Object variableRefNode
)
CodeGeneratingVisitor
+ <<Override>> visitAssigment (Object assigmentNode)
+ <<Override>> visitVariableRef (Object variableRefNode
)

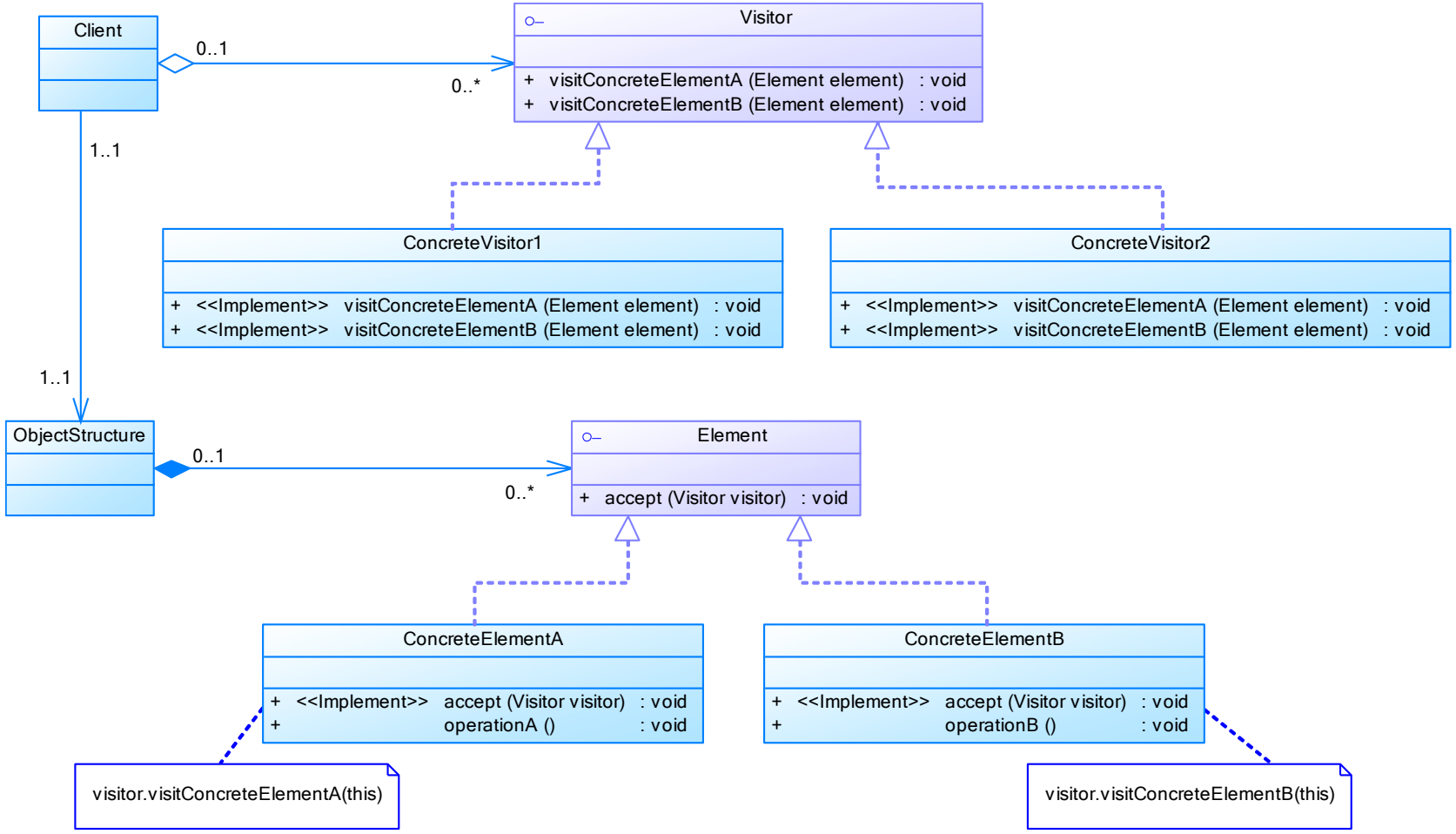
Мотивация


Program
0..1
0..*
Node
{abstract}
+ accept (Visitor visitor) : void
AssignmentNode
+ <<Override>> accept (Visitor visitor) : void
VariableRefNode
+ <<Override>> accept (Visitor visitor) : void
visitor.visitAssignment(this)
visitor.visitVariableRef(this)

Приложимост

- ✓ **Приложимост:** Шаблонът *Посетител* се използва в следните случаи:
  - Някоя структура от обекти съдържа обекти с различни интерфейси и е необходимо да се извършват операции върху тях, които зависят от конкретния им клас.
  - Много различни и несвързани операции трябва да се извършат над обектите от дадена структура и не трябва да се натоварват класовете им с тези операции.
  - Класовете, дефиниращи структурата от обекти рядко се променят, но често се дефинират нови операции върху тази структура.

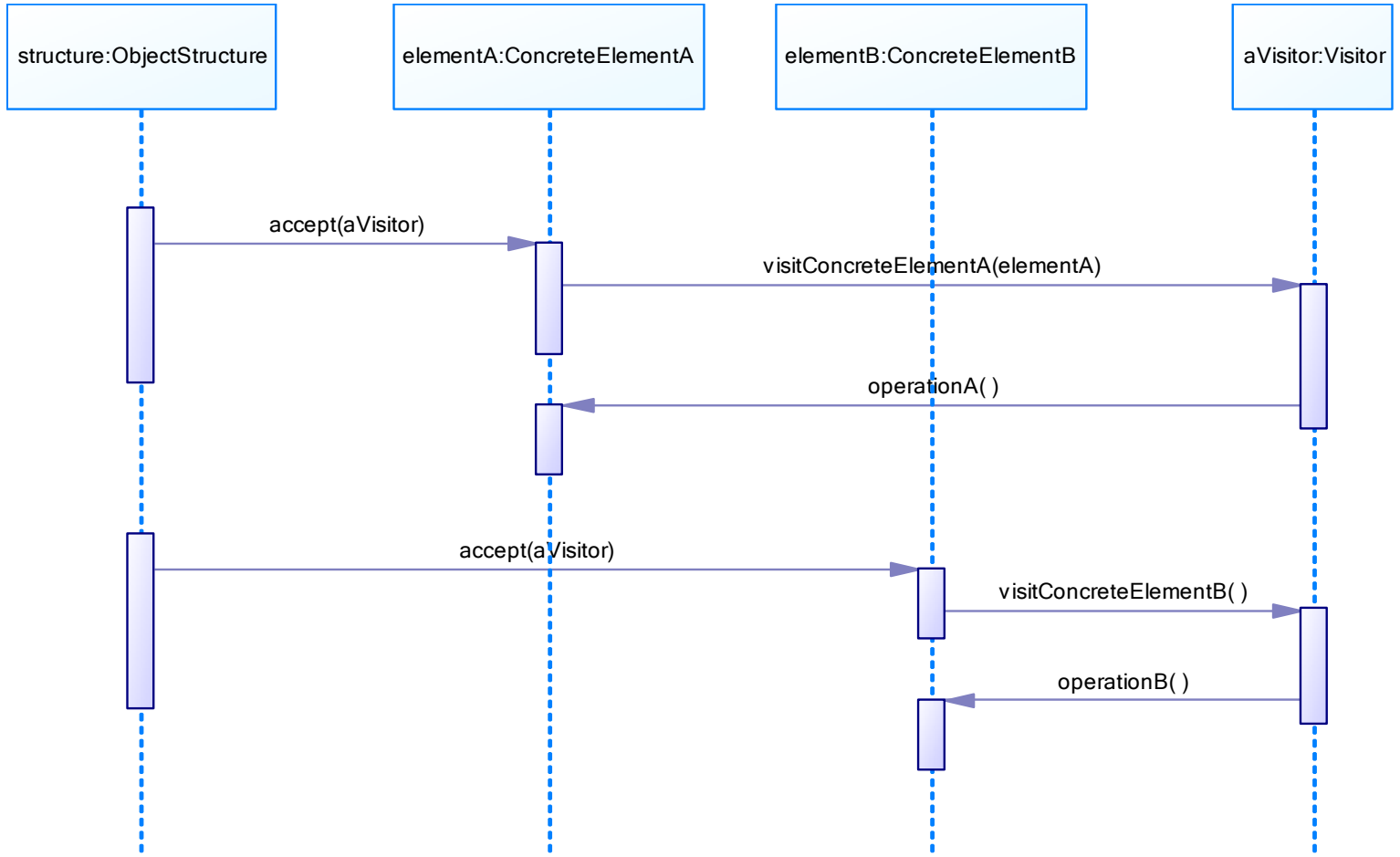
Структура


Client
0..1
0..*
Visitor
+ visitConcreteElementA (Element element) : void
+ visitConcreteElementB (Element element) : void
1..1
ConcreteVisitor1
+ <<Implement>> visitConcreteElementA (Element element) : void
+ <<Implement>> visitConcreteElementB (Element element) : void
ConcreteVisitor2
+ <<Implement>> visitConcreteElementA (Element element) : void
+ <<Implement>> visitConcreteElementB (Element element) : void
1..1
ObjectStructure
0..1
0..*
Element
+ accept (Visitor visitor) : void
ConcreteElementA
+ <<Implement>> accept (Visitor visitor) : void
+ operationA () : void
ConcreteElementB
+ <<Implement>> accept (Visitor visitor) : void
+ operationB () : void
visitor.visitConcreteElementA(this)
visitor.visitConcreteElementB(this)

Участници

- ✓ **Visitor** (NodeVisitor) – дефинира операция *visit* за всеки клас *ConcreteElement* в структурата от обекти. Името и сигнатурата на операцията идентифицират класа изпращащ заявката *Visit* до посетителя.
- ✓ **ConcreteVisitor** (TypeCheckingVisitor) – имплементира всички операции, декларирани от *Visitor*.
- ✓ **Element** (Node) – дефинира операция *accept*, приемаща за аргумент някой посетител.
- ✓ **ConcreteElement** (AssignmentNode, VariableRefNode) – имплементира операция *accept*, приемаща за аргумент някой посетител.
- ✓ **ObjectStructure** (Program)
  - Може да изброи своите елементи
  - Може да бъде или композиция (шаблон) или колекция – списък или множество.

Взаимодействия



Следствия

- ✓ **Предимства**
  - Добавяне на нова операция е лесно.
  - Свързаните операции не се разпростират сред класовете, дефиниращи обектната структура – локализирани са в посетителя.
  - Посетителя може да акумулира състояние.
- ✓ **Недостатъци**
  - Добавяне на нов *ConcreteElement* клас е трудно. Всеки нов *ConcreteElement* води до поява на нова абстрактна операция във *Visitor* и съответната имплементация във всеки *ConcreteVisitor* клас.
  - Интерфейса на *ConcreteElement* трябва да е достатъчен за да може *Visitor* да свърши работата си. Възможно е да се наложи добавяне на публични операции за достъп до вътрешното състояние на елемента, което може да наруши принципа за капсулиране.

Пример

- ✓ Пример за използване на шаблона *Посетител* заедно със шаблона *Композиция.*
- ✓ Следва *Component* класа от шаблона *Композиция* (за отбелязване е метода *accept()*)

```
public abstract class Component {
  protected String name;

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

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

  public abstract double getPrice();

  public abstract void accept(ComponentVisitor v);
}
```

Пример

- ✓ Следва компонент *листо*, наречен Widget.

```
public class Widget extends Component {
  protected double price;

  public Widget(String name, double price) {
    super(name);
    this.price = price;
  }

  public void setPrice(double price) { this.price = price; }
  public double getPrice() { return price; }

  public void accept (ComponentVisitor v) { v.visit(this); }
}
```

Пример

- ✓ Накрая компонент *композиция*, наречен *WidgetAssembly*.

```
public class WidgetAssembly extends Component
  protected Vector components;

  public WidgetAssembly (String name) {
    super(name);
    components = new Vector();
  }

  public void addComponent (Component c) {
    components.addElement(c);
  }
```

Пример

```
  public void removeComponent (Component c) {
    components.removeElement(c);
  }

  public double getPrice() {

    double totalPrice = 0.0;

    Enumeration e = components.elements();
    while (e.hasMoreElements()) {
      totalPrice += ((Component) e.nextElement()).getPrice();
    }
    return totalPrice;
  }

  public void accept (ComponentVisitor v) { v.visit(this); }
}
```

Пример

- ✓ Абстрактния суперклас от йерархията на шаблона *Посетител*.

```
public abstract class ComponentVisitor {

  public abstract void visit(Widget w);
  public abstract void visit(WidgetAssembly wa);

}
```

- ✓ Ако се добавят нови наследници на *Component* трябва да се добавят и нови методи в *ComponentVisitor.*

Пример

- ✓ Следва прост *посетител*, който просто извежда, че е посетил всеки компонент.

```
public class SimpleVisitor extends ComponentVisitor {

  public SimpleVisitor() {}

  public void visit (Widget w) {
    System.out.println("Visiting a Widget");
  }

  public void visit (WidgetAssembly wa) {
    System.out.println("Visiting a WidgetAssembly");
  }

}
```

Пример

- ✓ И *посетител*, който прави сравнително пазаруване.

```
public class PriceVisitor extends ComponentVisitor {
  private double maxPrice;

  public PriceVisitor(double maxPrice) {
    this.maxPrice = maxPrice; }

  public void visit (Widget w) {
    double price = w.getPrice();
    if (price > maxPrice)
      System.out.println("Don't Buy! Widget price of " +
        price + " exceeds maximum price (" + maxPrice + ").");
    else
      System.out.println("Buy! Widget price of " + price +
        " is less than maximum price (" + maxPrice + ").");
  }
```

Пример

```
  public void visit (WidgetAssembly wa) {
    double price = wa.getPrice();
    if (price > maxPrice)
      System.out.println("Don't Buy! WidgetAssembly price of " +
        price + " exceeds maximum price (" + maxPrice + ").");
    else
      System.out.println("Buy! WidgetAssembly price of " +
        price + " is less than maximum price (" +
        maxPrice + ").");
  }

}
```

Пример

- ✓ Тестовата програма

```
public class VisitorTest {

  public static void main (String[] args) {

    // Create some widgets.
    Widget w1 = new Widget("Widget1", 10.00);
    Widget w2 = new Widget("Widget2", 20.00);
    Widget w3 = new Widget("Widget3", 30.00);

    // Add then to a widget assembly.
    WidgetAssembly wa = new WidgetAssembly("Chassis");
    wa.addComponent(w1);
    wa.addComponent(w2);
    wa.addComponent(w3);
```

Пример

```
    // Visit some nodes with a SimpleVisitor.
    SimpleVisitor sv = new SimpleVisitor();
    w1.accept(sv);
    w2.accept(sv);
    w3.accept(sv);
    wa.accept(sv);

    // Visit some nodes with a PriceVisitor.
    PriceVisitor pv = new PriceVisitor(25.00);
    w1.accept(pv);
    w2.accept(pv);
    w3.accept(pv);
    wa.accept(pv);

  }

}
```

Пример

- ✓ Изхода на тестовата програма

```
Visiting a Widget
Visiting a Widget
Visiting a Widget
Visiting a WidgetAssembly
Buy! Widget price of 10.0 is less than maximum price (25.0).
Buy! Widget price of 20.0 is less than maximum price (25.0).
Don't Buy! Widget price of 30.0 exceeds maximum price (25.0).
Don't Buy! WidgetAssembly price of 60.0 exceeds maximum price
(25.0).
```

Край: Шаблон Посетител

ЛЕКЦИОНЕН КУРС: ШАБЛОНИ ЗА ПРОЕКТИРАНЕ