Tecniche di Progettazione: Design Patterns

GoF: Visitor

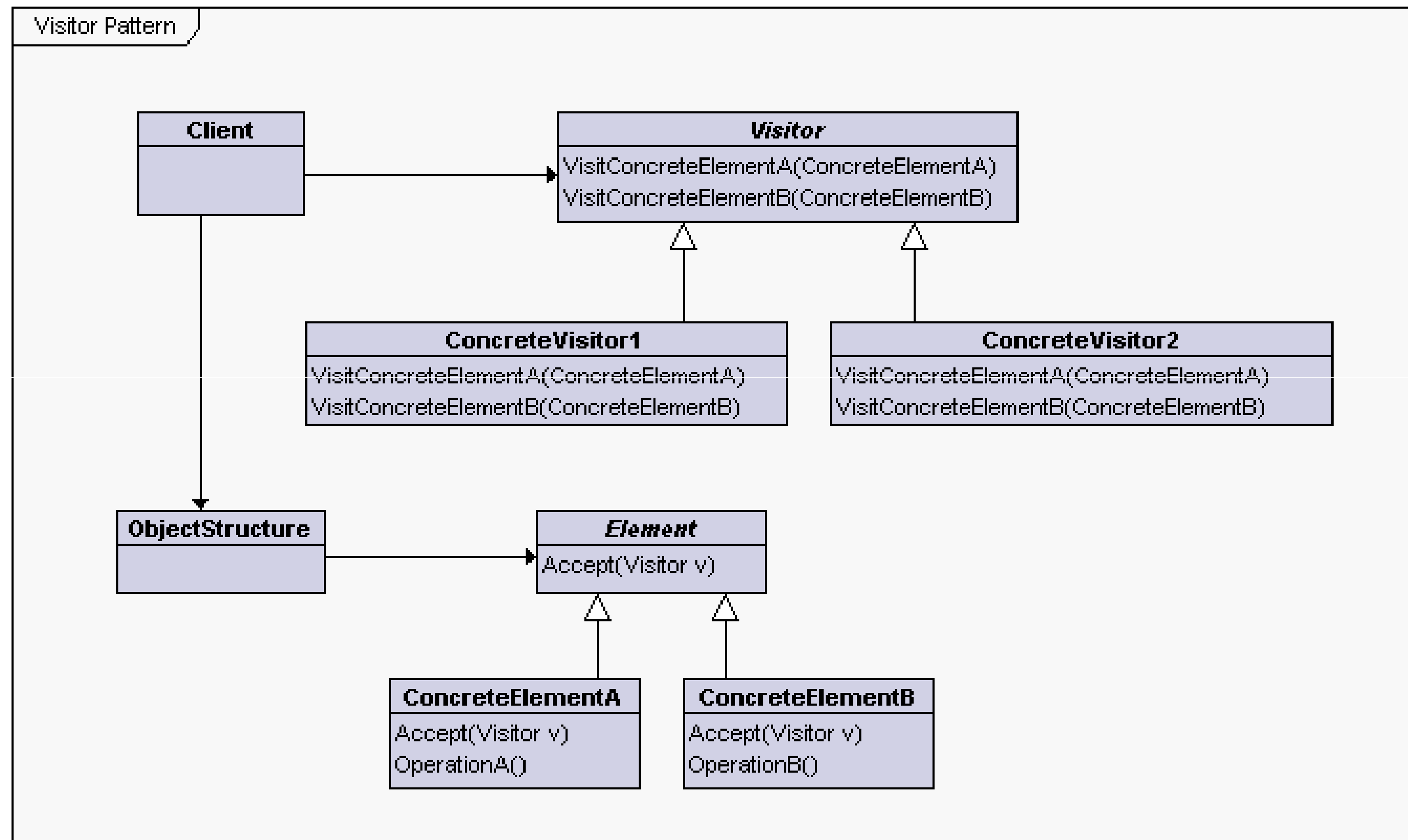
Visitor Pattern

- Intent
  - Lets you define a new operation without changing the classes on which they operate.
- Motivation
  - Allows for increased functionality of a class(es) while streamlining base classes.
  - A primary goal of designs should be to ensure that base classes maintain a minimal set of operations.
  - Encapsulates common functionality in a class framework.

Visitor Pattern: Applicability

- The following situations are prime examples for use of the visitor pattern.
  - When an object structure contains many classes of objects with different interfaces and you want to perform functions on these objects that depend on their concrete classes.
  - When you want to keep related operations together by defining them in one class.
  - When the class structure rarely change but you need to define new operations on the structure.

Structure



Visitor Pattern: Participants

- Visitor
  - Declares a Visit Operation for each class of Concrete Elements in the object structure.
- Concrete Visitor
  - Implements each operation declared by Visitor.
- Element
  - Defines an Accept operation that takes the visitor as an argument.
- Concrete Element
  - Implements an accept operation that takes the visitor as an argument.
- Object Structure
  - Can enumerate its elements.
  - May provide a high level interface to all the visitor to visit its elements.
  - May either be a composite or a collection.

Visitor Pattern: Collaborations

anObjectStructure
aConcreteElementA
aConcreteElementB
aConcreteVisitor

Accept(aVisitor)
VisitConcreteElementA(aConcreteElementA)
OperationA()
Accept(aVisitor)
VisitConcreteElementB(aConcreteElementB)
OperationB()

Visitor Pattern: Consequences

- Makes adding new operations easier.
- Collects related functionality.
- Adding new Concrete Element classes is difficult.
- Can "visit" across class types, unlike iterators.
- Accumulates states as they visit elements.
- May require breaking object encapsulation to support the implementation.

Visitor: Related Patterns

- Composites
  - Visitors can be used to apply an operation over an object structure defined by the composite pattern.
- Interpreter
  - Visitors may be applied to do the interpretation.

Visitor Pattern

- Motivation (cont)
  - Visitors avoid type casting that is required by methods that pass base class pointers as arguments. The following code describes how a typical class could expand the functionality of an existing composite.

```
Void MyAddition::execute( Base* basePtr) {
        if( dynamic_cast<ChildA*>(basePtr)){
                        // Perform task for child type A.
        } else if ( dynamic_cast<ChildB*>(basePtr)){
                        // Perform task for child type B.
        } else if( dynamic_cast<ChildC*>(basePtr)){
                        // Perform task for child type C.
        }
}
```

Double dispatch

- Visitor pattern is a very natural solution to double dispatch problems.
- Double dispatch problem is a subset of *dynamic dispatch* problems and it stems from the fact that method overloads are determined statically at compile time, unlike virtual(overriden) methods, which are determined at runtime.

Double dispatch.
Ex code:

```
public class CarOperations {
      void doCollision(Car car){}
      void doCollision(Bmw car){}
    }


public class Car {    public void doVroom(){}     }

public class Bmw extends Car {    public void doVroom(){}   }


public static void Main() {
    Car bmw = new Bmw();

    //calls Bmw.doVroom() - single dispatch, works out that car is actually Bmw at runtime.
    bmw.doVroom();

     //calls CarOperations.doCollision(Car car) because compiler chose doCollision overload
    based on the declared type of bmw variable
    CarOperations carops = new CarOperations();
    carops.doCollision(bmw);
```

Double dispatch. Solution

- In the java project

Deprecated
Ex.Visitor 1/2: the visitor and main

//This is the car operations interface. It knows about all the different kinds of cars it support
//and is statically typed to accept only certain ICar subclasses as parameters

```
public interface CarVisitor {
    void StickAccelerator(Toyota car);
    void ChargeCreditCardEveryTimeCigaretteLighterIsUsed(Bmw car);
}

public class Program {
  public static void Main() {
    Car car = carDealer.getCarByPlateNumber("4SHIZL");
    CarVisitor visitor = new SomeCarVisitor();
    car.performOperation(visitor)
```

Deprecated
Ex. Visitor 2/2 (elements):

```
//Car interface, a car specific operation is invoked by calling PerformOperation
public interface Car {
    public void performOperation(CarVisitor visitor);
    }


public class Toyota implements Car {
    public void performOperation(CarVisitor visitor) { visitor.StickAccelerator(this); }
    }


public class Bmw implements Car{
        public void performOperation(ICarVisitor visitor) {
                visitor.ChargeCreditCardEveryTimeCigaretteLighterIsUsed(this);
        }
    }
```

Another point of view
nice example

- The issue addressed by the Visitor pattern is the manipulation of composite objects
  - Without visitors, such manipulation runs into several problems as illustrated by considering an implementation of integer lists, written in Java

```
interface List {}

class Nil implements List {}

class Cons implements List {
    int head;
    List tail;
}
```

  - What happens when we write a program which computes the sum of all components of a given List object?

First Attempt: Instanceof and Type Casts

```
List l;
// The List-object we are working on.
int sum = 0;
// Contains the sum after the loop.
boolean proceed = true;
while (proceed) {
    if (l instanceof Nil)
      proceed = false;
    else if (l instanceof Cons) {
      sum = sum + ((Cons) l).head; // Type cast!
      l = ((Cons) l).tail;         // Type cast!
    }
}
```

Second Attempt: Dedicated Methods

```
interface List {
   int sum();
}
class Nil implements List {
   public int sum() { return 0; }
}
class Cons implements List {
   int head;
   List tail;
   public int sum() {
       return head + tail.sum();
   }
 }
```

Dedicated Methods

- Can compute the sum of all components of a given List-object l by writing l.sum().
- Advantage: type casts and instanceof operations have disappeared, and that the code can be written in a systematic way.
- Disadvantage: Every time we want to perform a new operation on List-objects, say, compute the product of all integer parts, then new dedicated methods have to be written for all the classes, and the classes must be recompiled

Third attempt: Visitor (1/2)

```
interface List {
void accept(Visitor v);
}
class Nil implements List {
    public void accept(Visitor v) {       v.visitNil(this); }
}
class Cons implements List {
   int head;
    List tail;
    public void accept(Visitor v) {       v.visitCons(this);
   } }
```

Third attempt: Visitor (2/2)

```
interface Visitor {
  void visitNil(Nil x);
  void visitCons(Cons x);
}
class SumVisitor implements Visitor {
    int sum = 0;
    public void visitNil(Nil x) {}
    public void visitCons(Cons x){
        sum = sum + x.head;
        x.tail.accept(this); } }

SumVisitor sv = new SumVisitor();
l.accept(sv);
System.out.println(sv.sum);
```