Visitor Pattern

Visitor Pattern

- Purpose
  - Allowing one or more operations to be applied to a set of objects at runtime
  - Decoupling the operations from the object structure

- Use when
  - An object structure must have many unrelated operations performed upon it
  - The object structure can't change but operations on it can
  - Operations must be performed on the concrete classes of an object structure
  - Operations should be able to operate on multiple object structures that implement the same interface sets

Example : Abstract Syntax Tree

- Consider a compiler with abstract syntax trees

- Compiler may need to define operations for

    Type-Checking

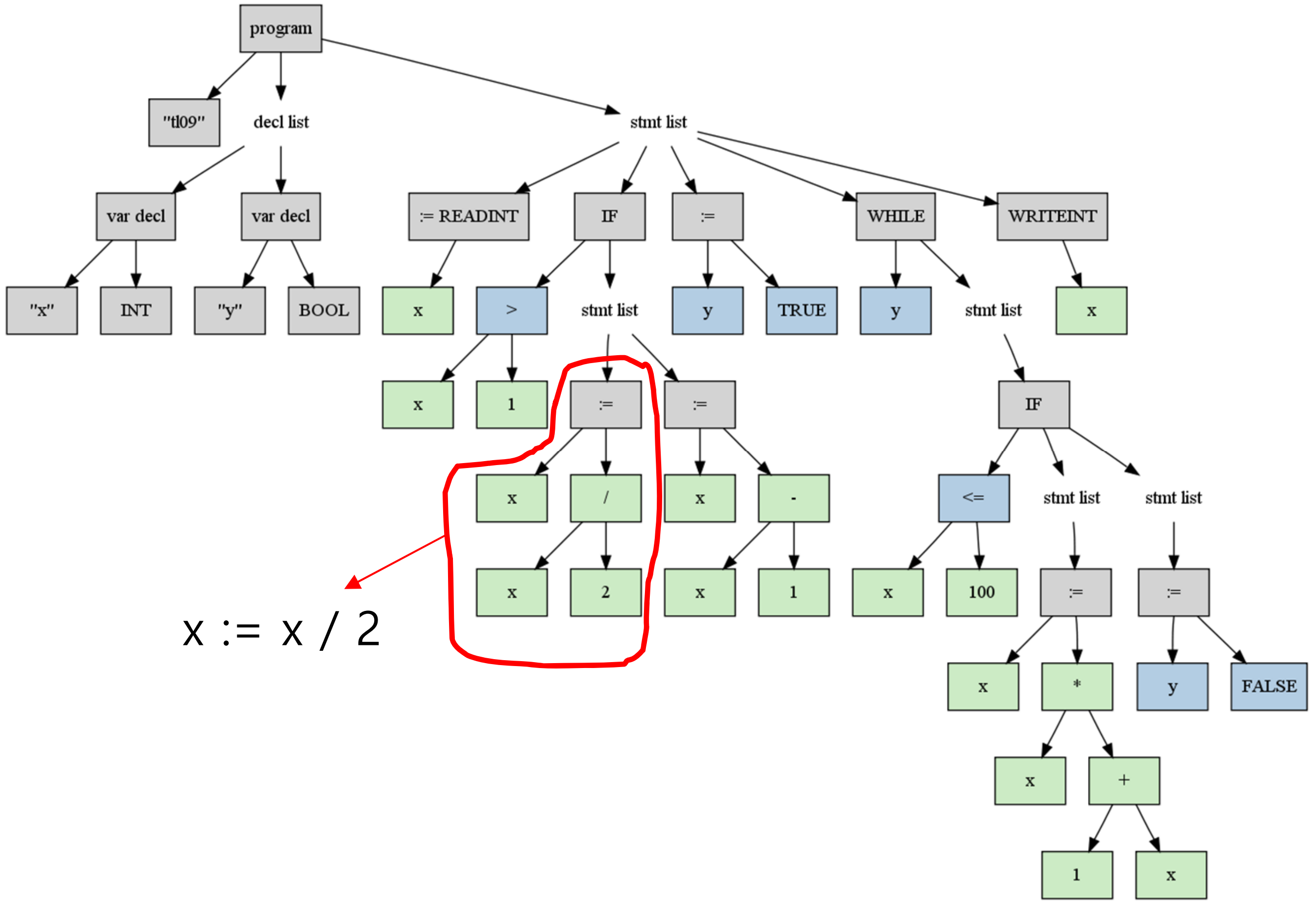
    Code Optimization

    Flow Analysis

    … and more!

Implementing Type-Checking Operation

- Seemingly not an issue
  - Just implement at each AST node

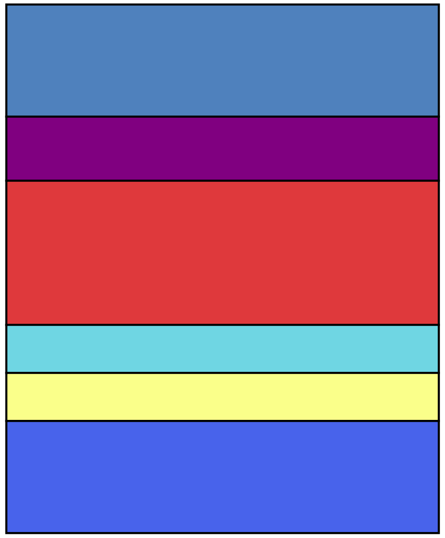


More operation types are expected

- There will be one class for every single thing that needs to be performed.
- Therefore, distributing operations amongst various node classes is hard to understand, maintain and change!
  - Adding a new operation would be painful

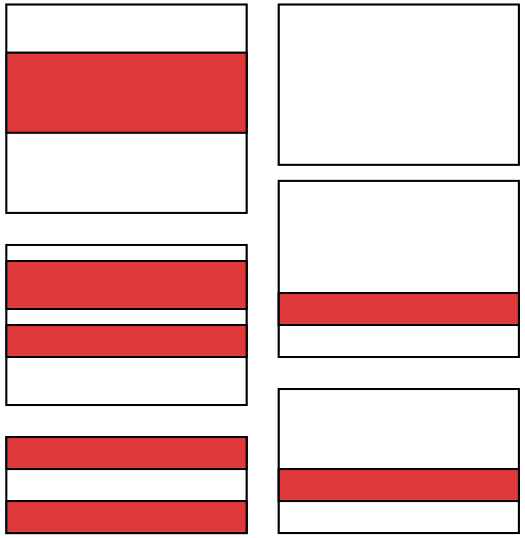
code tangling:

one module many concerns



code scattering:

one concern many modules

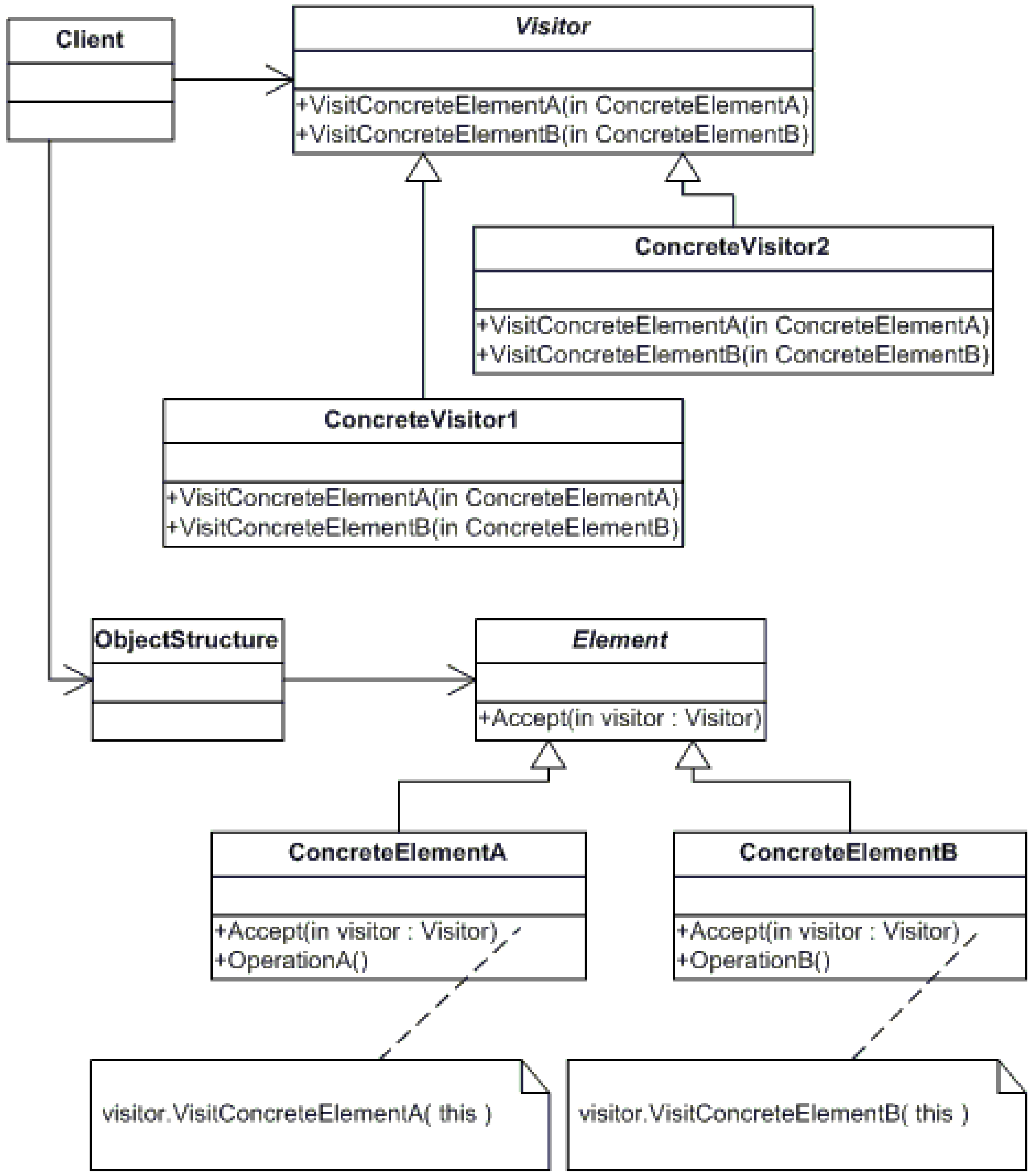


Motivations and Benefits

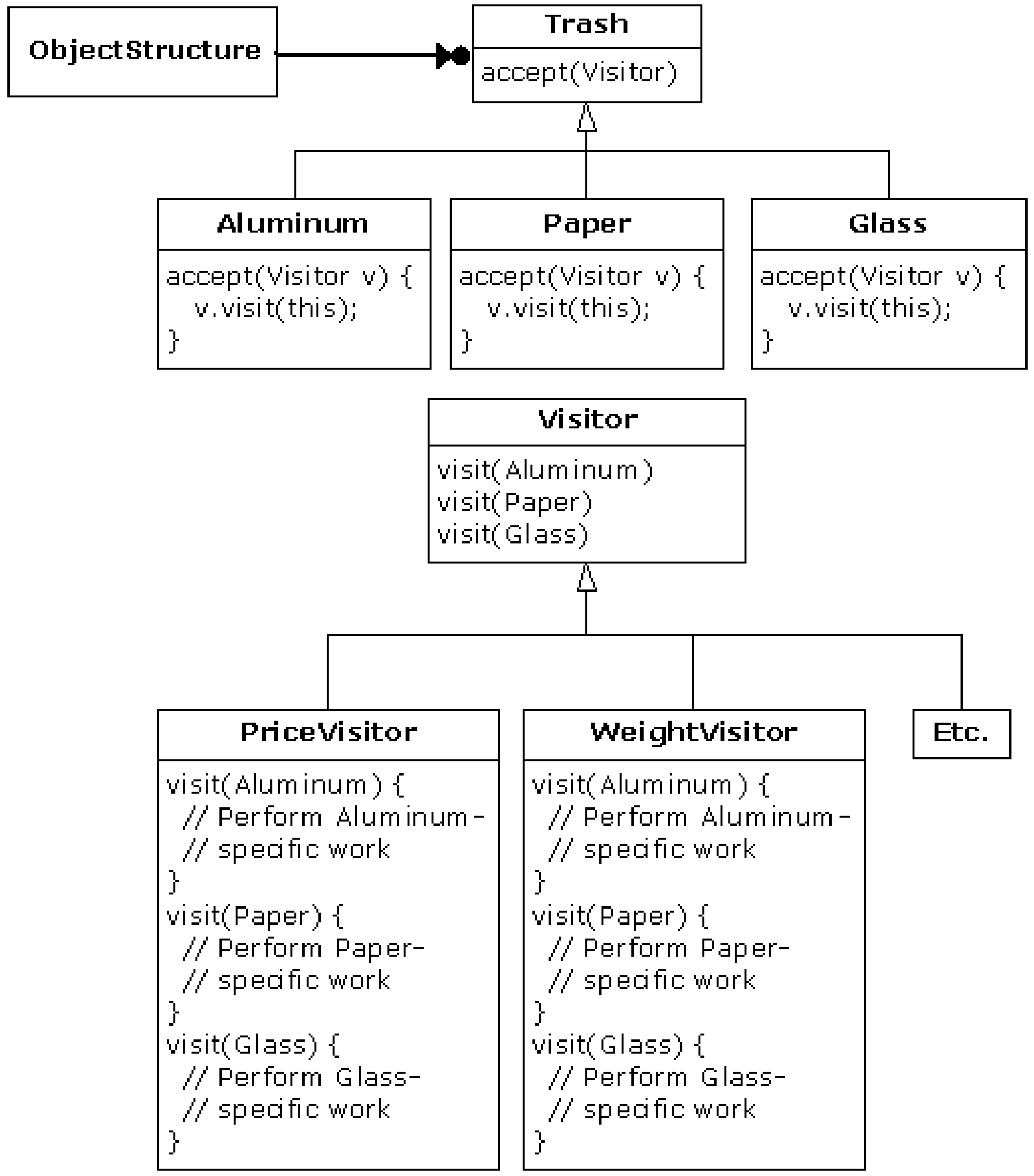
- Motivations
  - An object structure contains many classes of objects
  - Many distinct and unrelated operations on these objects
    → We want to avoid “polluting” their classes
  - Classes defining the object structure rarely change, but operations change frequently

- Benefits
  - Makes adding new operations easy
  - Gathers related operations and separates unrelated operations
  - Visitors can visit objects that don’t have a common parent class

The Visitor Design Pattern


Client
Visitor
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor2
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor1
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ObjectStructure
Element
+Accept(in visitor : Visitor)
ConcreteElementA
+Accept(in visitor : Visitor)
+OperationA()
ConcreteElementB
+Accept(in visitor : Visitor)
+OperationB()
visitor.VisitConcreteElementA(this)
visitor.VisitConcreteElementB(this)

The Visitor Example



```
accept(Visitor v) {
   v.visit(this);
}
```

```
visit(Aluminum) {
  // Perform Aluminum-
  // specific work
}
visit(Paper) {
  // Perform Paper-
  // specific work
}
visit(Glass) {
  // Perform Glass-
  // specific work
}
```

Participants

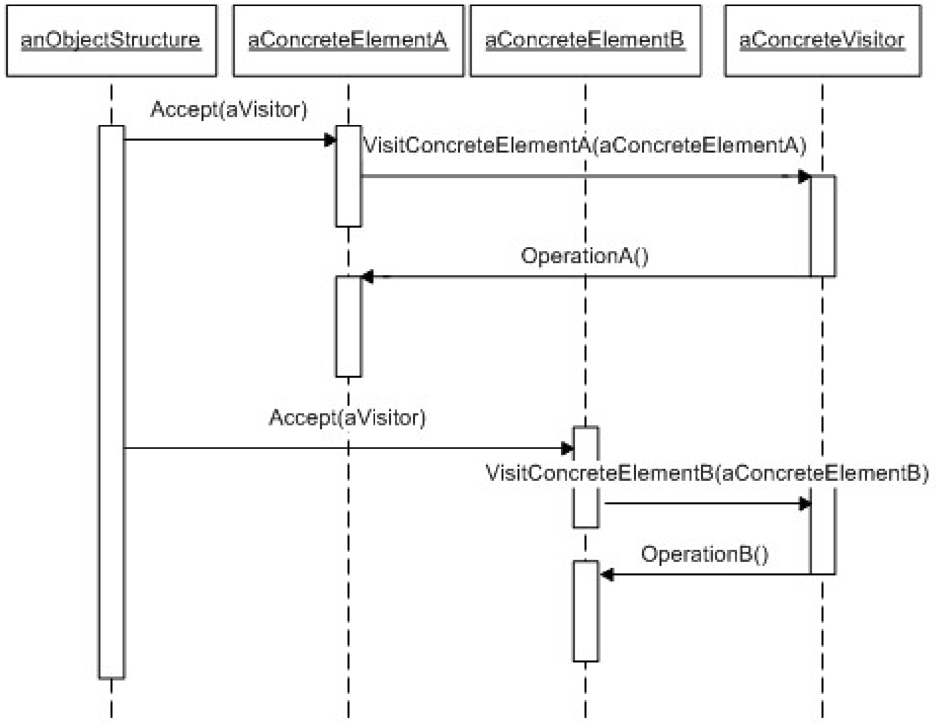
- Visitor
  - declares a visit operation for each class of ConcreteElement in the object structure
- ConcreteVisitor (PriceVisitor, WeightVisitor)
  - implements each operation declared by Visitor
- Element (Trash)
  - defines an Accept operation that takes a visitor as an argument
- ConcreteElement (Aluminum, Paper, Glass)
  - implements an Accept operation that takes a visitor as an argument
- ObjectStructure
  - can enumerate its elements
  - may provide a high-level interface to allow the visitor to visit its elements
  - may be a composite or a collection like a set or list

Collaborations

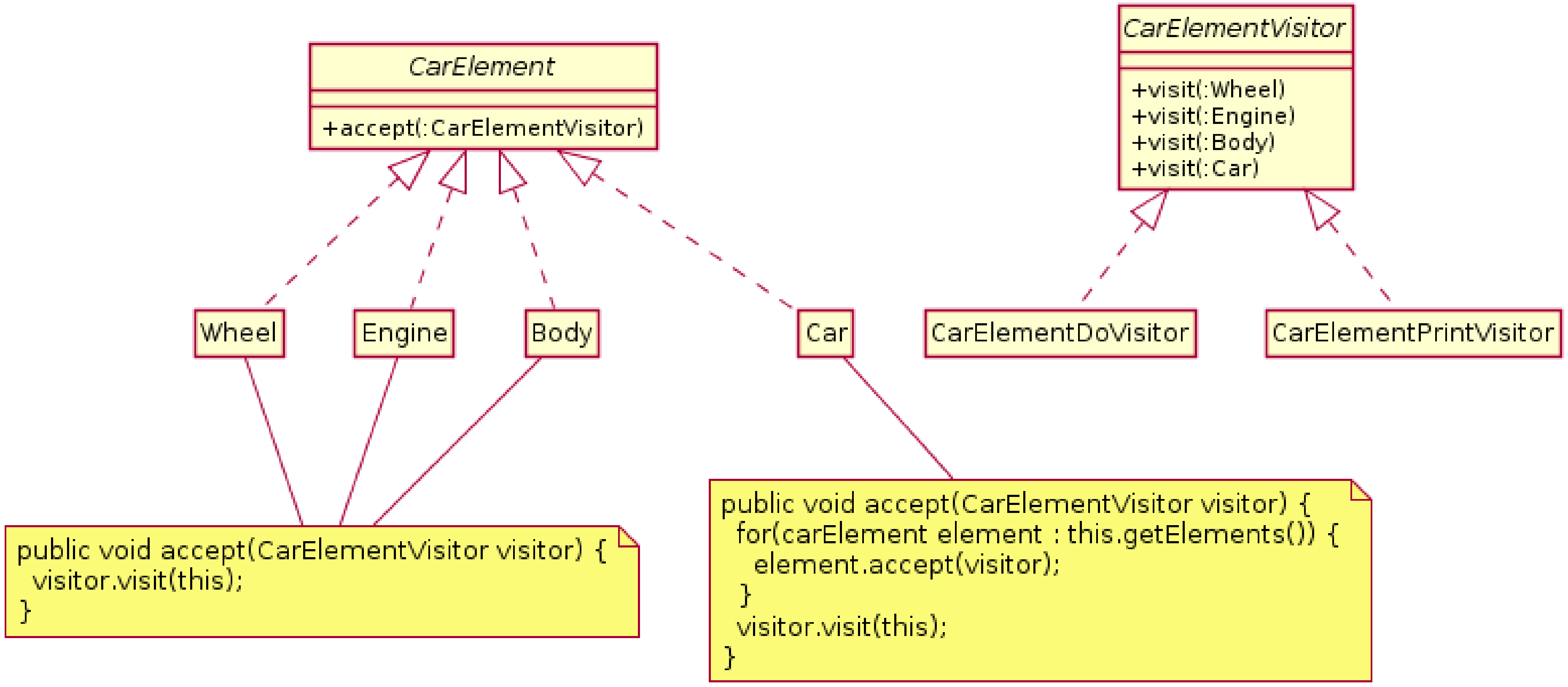
- A client must create a ConcreteVisitor object and then traverse the object structure, visiting each element with the visitor

- When the element is visited, it calls the Visitor operation that corresponds to its class.

- The element supplies itself as an argument to this operation to let the visitor access its state, if necessary



Another Example of Visitor Pattern



```
public void accept(CarElementVisitor visitor) {
  visitor.visit(this);
}
```

```
public void accept(CarElementVisitor visitor) {
  for(carElement element : this.getElements()) {
    element.accept(visitor);
  }
  visitor.visit(this);
}
```

```java
interface ICarElementVisitor {
    void visit(Wheel wheel);
    void visit(Engine engine);
    void visit(Body body);
    void visit(Car car);
}
interface ICarElement {
    void accept(ICarElementVisitor visitor);
}
public class VisitorDemo {
    public static void main(String[] args) {
        ICarElement car = new Car();
        car.accept(new CarElementPrintVisitor());
        car.accept(new CarElementDoVisitor());
    }
}
```

```
class Wheel implements ICarElement {
    private String name;
    public Wheel(String name) { this.name = name; }
    public String getName() { return this.name; }
    public void accept(ICarElementVisitor visitor) {
        visitor.visit(this);
    }
}
class Engine implements ICarElement {
    public void accept(ICarElementVisitor visitor) {
        visitor.visit(this);
    }
}
class Body implements ICarElement {
    public void accept(ICarElementVisitor visitor) {
        visitor.visit(this);
    }
}
```

```
class Car implements ICarElement {
    ICarElement[] elements;
    public Car() {
        this.elements = new ICarElement[] {
          new Wheel("front left"), new Wheel("front right"),
          new Wheel("back left") , new Wheel("back right"),
          new Body(),
          new Engine() };
    }
    public void accept(ICarElementVisitor visitor) {
        for(ICarElement elem : elements)
            elem.accept(visitor);
        visitor.visit(this);
    }
}
```

```
class CarElementPrintVisitor implements ICarElementVisitor {
    public void visit(Wheel wheel) {
      System.out.println("Visiting " + wheel.getName() + "
wheel");
    }
    public void visit(Engine engine) {
        System.out.println("Visiting engine");
    }
    public void visit(Body body) {
        System.out.println("Visiting body");
    }
    public void visit(Car car) {
        System.out.println("Visiting car");
    }
}
```

```
class CarElementDoVisitor implements ICarElementVisitor {
    public void visit(Wheel wheel) {
        System.out.println("Kicking my " + wheel.getName() + "
wheel");   }
    public void visit(Engine engine) {
        System.out.println("Starting my engine");
    }
    public void visit(Body body) {
        System.out.println("Moving my body");
    }
    public void visit(Car car) {
        System.out.println("Starting my car");
    }
}
```

```
[OUTPUT]
Visiting front left wheel
Visiting front right wheel
Visiting back left wheel
Visiting back right wheel
Visiting body
Visiting engine
Visiting car
Kicking my front left wheel
Kicking my front right wheel
Kicking my back left wheel
Kicking my back right wheel
Moving my body
Starting my engine
Starting my car
```

main()

```
car.accept(new CarElementPrintVisitor());
```

Car::accept()

```
for(ICarElement elem : elements)
    elem.accept(visitor);
```

Engine::accept()

```
public void accept(ICarElementVisitor visitor) {
    visitor.visit(this);
}
```

CarElementVisitor::visit()

```
class CarElementPrintVisitor implements ICarElementVisitor {
    …
    public void visit(Engine engine) {
        System.out.println("Visiting engine");
    }
    …
```

Do We Really Need to Repeat ?

```
class Wheel implements ICarElement {
    public void accept(ICarElementVisitor visitor) {
        visitor.visit(this); }
}
class Engine implements ICarElement {
    public void accept(ICarElementVisitor visitor) {
        visitor.visit(this); }
}
class Body implements ICarElement {
    public void accept(ICarElementVisitor visitor) {
        visitor.visit(this); }
}
class Car implements ICarElement {
    public void accept(ICarElementVisitor visitor) {
        visitor.visit(this); }
}
```

Practice: accept 메쏘드 중복구현 피해보기

- 연습 소스: Visitor.interfaceVisitor
- 각 element 클래스에서 중복 구현되는 accept 메쏘드 없애기
  - ICarElement interface를 abstract class로 변경하고 accept 메쏘드 구현
  - 각 concrete element에서 implement를 extend로 바꾸기
  - 각 concrete element에서 accept 메쏘드 없애기
  - 각 visitor에서 ICarElement를 위한 visit(ICarElement) 추가 (컴파일을 위해 필요)

```
class Wheel implements ICarElement {
    public void accept(ICarElementVisitor visitor) {
        visitor.visit(this); }
}
class Engine implements ICarElement {
    public void accept(ICarElementVisitor visitor) {
        visitor.visit(this); }
}
class Body implements ICarElement {
    public void accept(ICarElementVisitor visitor) {
        visitor.visit(this); }
}
```

Practice: accept 메쏘드 중복구현 피해보기

- 연습 소스: Visitor.interfaceVisitor
- 각 element 클래스에서 중복 구현되는 accept 메쏘드 없애기
  - ICarElement interface를 abstract class로 변경하고 accept 메쏘드 구현
  - 각 concrete element에서 implement를 extend로 바꾸기
  - 각 concrete element에서 accept 메쏘드 없애기

```
interface ICarElement {
    void accept(ICarElementVisitor visitor);
}
```

```
abstract class ICarElement {
    void accept(ICarElementVisitor visitor) {
        visitor.visit(this);
    }
}
```

```
class Wheel implements ICarElement {
    public void accept(ICarElementVisitor
      visitor)
    {
        visitor.visit(this);
    }
}
```

class Wheel extends ICarElement {
~~public void accept(ICarElementVisitor~~
~~visitor)~~
~~{~~
~~visitor.visit(this);~~
~~}~~
}

Practice: accept 메쏘드 중복구현 피해보기

- 연습 소스: Visitor.interfaceVisitor
- 각 element 클래스에서 중복 구현되는 accept 메쏘드 없애기
  - 각 visitor에서 ICarElement를 위한 visit(ICarElement) 추가 (컴파일을 위해 필요)

```
interface ICarElementVisitor {
    void visit(Wheel wheel);
    void visit(Engine engine);
    void visit(Body body);
    void visit(Car car);
}
```

```
class CarElementPrintVisitor implements
ICarElementVisitor {
    public void visit(Wheel wheel) {
      System.out.println("Visiting " +
wheel.getName() + " wheel");
    }
    public void visit(Engine engine) {
        System.out.println("Visiting engine");
    }
    public void visit(Body body) {
        System.out.println("Visiting body");
    }
    public void visit(Car car) {
        System.out.println("Visiting car");
    }
}
```

```
interface ICarElementVisitor {
    void visit(Wheel wheel);
    void visit(Engine engine);
    void visit(Body body);
    void visit(Car car);
    void visit(ICarElement dummy);
}
```

```
class CarElementPrintVisitor implements
ICarElementVisitor {
    public void visit(Wheel wheel) {
      System.out.println("Visiting " +
wheel.getName() + " wheel");
    }
    public void visit(Engine engine) {
        System.out.println("Visiting engine");
    }
    public void visit(Body body) {
        System.out.println("Visiting body");
    }
    public void visit(Car car) {
        System.out.println("Visiting car");
    }
    public void visit(ICarElement dummy) {
        System.out.println(“!!! Dummy !!!");
    }
}
```

Method Overloading - Static Polymorphism

```
class X
{
   void methodA(int num)
   {
       System.out.println ("methodA:" + num);
   }
   void methodA(int num1, int num2)
   {
       System.out.println ("methodA:" + num1 + "," + num2);
   }
   double methodA(double num) {
       System.out.println("methodA:" + num);
       return num;
   }
}

public class test
{
   public static void main (String args [])
   {
       X Obj = new X();
       double result;
       Obj.methodA(20);
       Obj.methodA(20, 30);
       result = Obj.methodA(5.5);
       System.out.println("Answer is:" + result);
   }
}
```

```
methodA:20
methodA:20,30
methodA:5.5
Answer is:5.5
```

Method Overriding - Dynamic Polymorphism

```
public class X
{
    public void methodA() //Base class method
    {
        System.out.println ("hello, I'm methodA of class X");
    }
}

public class Y extends X
{
    public void methodA() //Derived Class method
    {
        System.out.println ("hello, I'm methodA of class Y");
    }
}
public class Z
{
   public static void main (String args []) {
       X obj1 = new X(); // Reference and object X
       X obj2 = new Y(); // X reference but Y object
       obj1.methodA();
       obj2.methodA();
   }
}
```

```
hello, I'm methodA of class X
hello, I'm methodA of class Y
```

Technical Details

- Double Dispatch
  - special form of multiple dispatch
  - a mechanism that dispatches a method call to different concrete methods depending on the runtime types of two objects involved in the call

- Single Dispatch
  - In most object-oriented systems, the concrete method that is called from a method call in the code depends on the dynamic type of a single object and therefore they are known as single dispatch calls or simply virtual function(method) calls

Double Dispatch is more than Method Overloading

- At first glance, double dispatch appears to be a natural result of method overloading
  - Method overloading allows the method called to depend on the type of the argument
- Method overloading, however, is done at compile time
  - using "name mangling" where the internal name of the method has the argument's type encoded in it
  - Different from method overriding (run-time polymorphism)
- Example
  - method foo(int) → __foo_i
  - method foo(double) → __foo_d

Visitor Pattern and Single Dispatch

- Visitor pattern requires a programming language that supports single dispatch
    - Consider two objects: "element" and "visitor"

- An element has an accept method that can take the visitor as an argument
- The accept method calls a visit method of the visitor; the element passes itself as an argument to the visit method

How Double-dispatch is implemented in Visitor Pattern

- When the accept method is called, its implementation is chosen based on both:
  - The dynamic type of the element
  - The static type of the visitor

```
for(ICarElement elem : elements)
    elem.accept(visitor);
```

- When the associated visit method is called, its implementation is chosen based on both:
  - The dynamic type of the visitor
  - The static type of the element as known from within the implementation of the accept method, which is the same as the dynamic type of the element. (As a bonus, if the visitor can't handle an argument of the given element's type, then the compiler will catch the error)

```
public void accept(ICarElementVisitor visitor) {
    visitor.visit(this);
}
```

How Double-dispatch is implemented in Visitor Pattern

- Consequently, the implementation of the visit method is chosen based on both:
  - The dynamic type of the element
  - The dynamic type of the visitor

- This effectively implements double dispatch
  - Common Lisp language's object system supports multiple dispatch (not just single dispatch), and implementing the visitor pattern in Common Lisp is trivial

Implementation

- Who is responsible for traversing the object structure?
  - A visitor must visit each element of the object structure. How does it get there?

  - We can put responsibility for traversal in any of three places:
    - In the object structure
    - in the visitor
    - or in a separate Iterator object

  - Having the traversal code in the visitor is the least preferred option, as it forces you to repeat the code in every ConcreteVisitor for each ConcreteElement;
    - so you would only follow this approach for a particularly complex traversal

Benefits

- Visitor makes adding new operations easy
  - You can define a new operation simply by adding a new visitor
  - In contrast, if you spread functionality over many classes, then you must change each class to define a new operation
- A visitor gathers related operations and separates unrelated ones
  - Related behavior is not spread over the classes defining the object structure; it’s localized in a visitor
  - Unrelated sets of behavior are partitioned in their own visitor classes

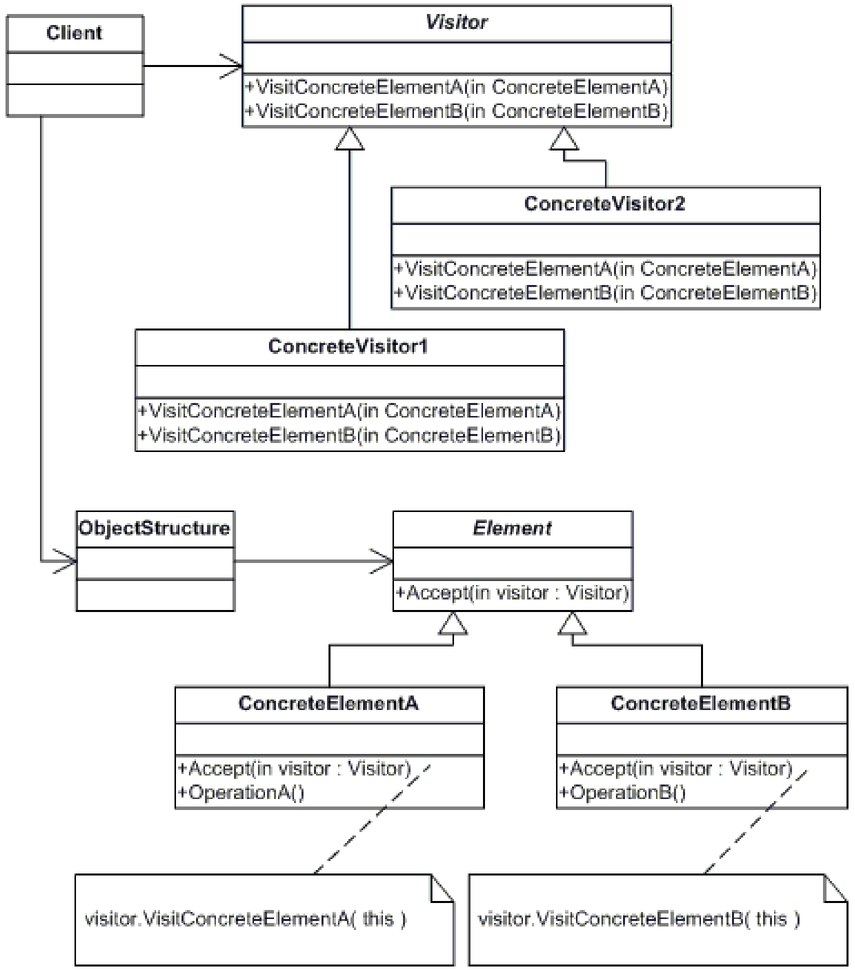
Liabilities

- Adding new ConcreteElement classes is hard
  - The Visitor pattern makes it hard to add new subclasses of Element
  - Each new ConcreteElement gives rise to a new abstract operation on Visitor and a corresponding implementation in every ConcreteVisitor class
- Breaking encapsulation
  - Visitor’s approach assumes that the ConcreteElement interface is powerful enough to let visitors do their job
  - The pattern often forces you to provide public operations that access an element’s internal state, which may compromise its encapsulation

Summary

- The Visitor Design Pattern
  - (+) makes adding new operations easy
  - (+) gathers related operations and separates unrelated ones
  - (-) adding new ConcreteElement classes is hard
  - (-) Breaks encapsulation

- Implementation Issues
  - Who is responsible for traversing the object structure?


Client
Visitor
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor2
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ConcreteVisitor1
+VisitConcreteElementA(in ConcreteElementA)
+VisitConcreteElementB(in ConcreteElementB)
ObjectStructure
Element
+Accept(in visitor : Visitor)
ConcreteElementA
+Accept(in visitor : Visitor)
+OperationA()
ConcreteElementB
+Accept(in visitor : Visitor)
+OperationB()
visitor.VisitConcreteElementA(this)
visitor.VisitConcreteElementB(this)