Visitor Pattern

Software Design (v1.12)

Raúl Izquierdo Castanedo

Example

Example

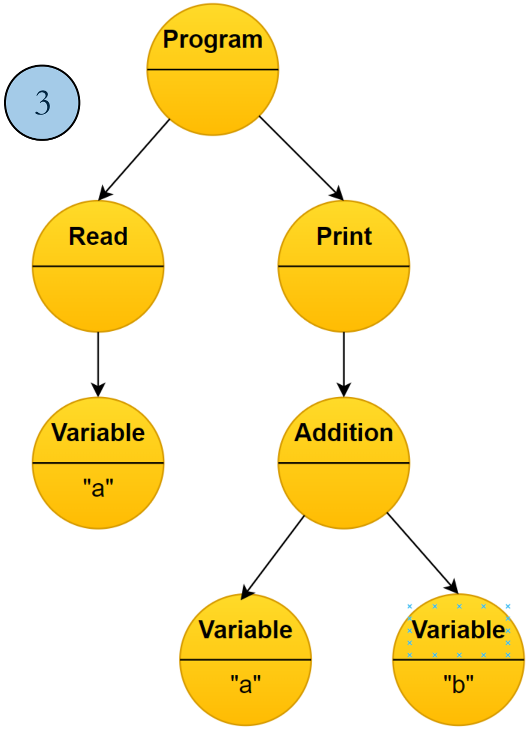
1

```
read a;
print a + b;
```

2 **Modeling the tree nodes**

```
interface Node { }

class Program implements Node {
    List<Statement> statements;
}


interface Statemen extends Node { }

class Read implements Statemen {
    Variable var;
}
class Print implements Statement {
    Expression expr;
}


interface Expression extends Node { }

class Addition implements Expression {
    Expression left, right;
}
class Variable implements Expression {
    String name;
}
```

3



Implementing the Browser Model

It is desirable to browse the programs with different objectives :

- To **print the program** (format and coloring)
- To do the **Semantic Analysis** (checking errors)
- To **compile** (generate code)
- To **document** (*javaDoc*)
- And in the future…

How/where to implement the code for each tree browse?

- Alternative 1. Decentralized Implementation
- Alternative 2. Centralized Implementation

Decentralized Implementation

Alternative 1. Decentralized Implementation

- *Interpreter* pattern
- this is based on distributing the code to browse the tree among the node classes. Every class performs an operation on the tree.
  - Each node will have a method for EVERY TREE BROWSE (for each operation)

```
class Print implements Statement {
        void testErrors() { … };
        void generateCode() { … };
}

class Adition implements Expressión {
        void testErrors() { … };
        void generateCode() { … };
}

// And so on in the rest of the classes…
```

- Drawbacks?

- It is appropriate only if…
  - Tree browse (operations) are more stable than the nodes.

Centralized Implementation

Alternative 2. Centralized Implementation

- The full code for an operation is applied in a single class
  - This code should indicate what to do with each node.
- Advantage
  - Add/remove operations do not affect nodes
- Drawbacks?

- It is appropriate only if…
  - Nodes are more stable than tree browses

The conditions in our case are

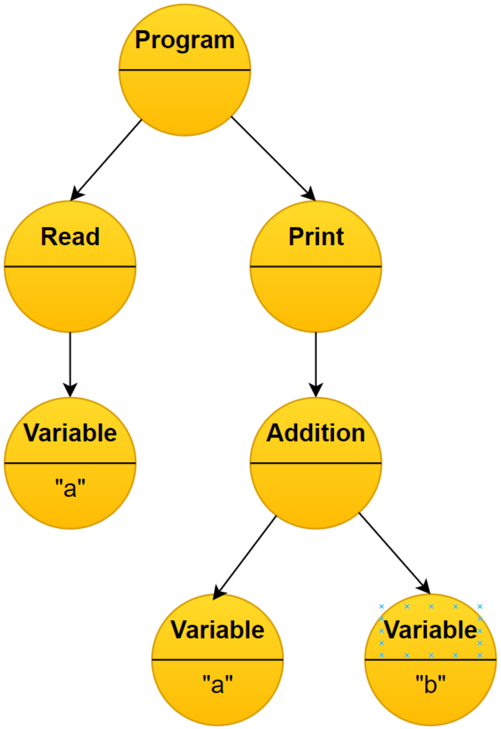
  - Nodes are more stable
  - And different tree browses will be added and removed.
    - We want to be able to do this without modifying the nodes every time.!!!

There are several ways to implement the centralized solution

- Recursive browse
- Visitor

Centralized Implementation. Recursive Browse (I)

```
public static void main(String[] args) {
  Program prog = new Program … // Build the tree here

  RecursiveBrowse browse = new RecursiveBrowse();
  browse.visit(prog);
}

class RecursiveBrowse {

    public void visit(Node node) {
       if (node instanceof Program) {
             for (Statement statement : ((Program) node).statement)
               visit(statement);

       } else if (node instanceof Print) {
            System.out.println("print ");
            visit( ((Print)node).expr );
            System.out.println(";");

        } else if (node instanceof Read) {
            System.out.println"read ");
            visit( ((Read)node).var );
            System.out.println(";");

        } else if (node instanceof Addition) {
            visit( ((Addition)node).left );
             System.out.println(“+");
            visit( ((Addition)node).right );

        } else if (node instanceof Variable)
            System.out.println( ((Variable)node).name );
    }
}  (*)
```



read a;
print a + b;

Centralized Implementation. Recursive Browse (II)

```
class RecursiveBrowse {

    public void visit(Node node) {
       if (node instanceof Program) {
            for (Statement statement : ((Program) node).statement)
              visit(statement);

       } else if (node instanceof Print) {
            System.out.println("print ");
            visit( ((Print)node).expr );
            System.out.println(";");

        } else if (node instanceof Read) {
            System.out.println"read ");
            visit( ((Read)node).var );
            System.out.println(";");

        } else if (node instanceof Addition) {
            visit( ((Addition)node).left );
            System.out.println(“+");
            visit( ((Addition)node).right );

        } else if (node instanceof Variable)
            System.out.println( ((Variable)node).name );
    }
```

Is there a problem with this implementation?

Centralized Implementation. Ideal Version

```
public class PrintProgram        // Ideal Version
{
    public void visit(Program program) {
       for (Statement statement : program.statements)
           visit(statement);
    }

    public void visit(Print print) {
       System.out.println("print ");
       visit(print.expr);
       System.out.println(";");
    }

    public void visit(Read read) {
       System.out.println("read ");
       visit(read.var);
       System.out.println(";");
    }

    public void visit(Addition addition) {
        visit(addition.left);
        System.out.println(" + ");
        visit(addition.right);
    }

    public void visit(Variable var) {
        System.out.println(var.name);
    }
}
```

It does not compile!

Ideal Version. Problem (I)

```
interface Figure
{

}

class Circle implements Figure
{

}
```

```
class Test
{
  void print(Figure f) {
      System.out.println(“Figure");
  }

  void print(Circle c) {
      System.out.println(“Circle");
  }

  public static void main(String[] args){
      Figure circle = new Circle();
      print(circle);   // What is printed?
  }
}
```

There are languages that support this feature:

- Multiple dispatch

Ideal Version. Problem (II)

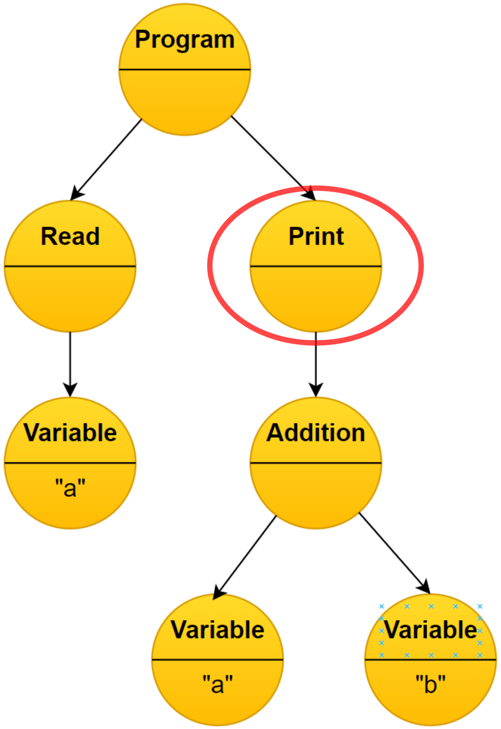
```
public class PrintProgram         // Ideal Version
{
    public void visit(Program prog) {
        for (Statement statement : prog.statements)
            visit(statement);
    }

    public void visit(Print print) {
       System.out.println("print ");
       visit(print.expr);
       System.out.println(";");
    }

    public void visit(Read read) {
       System.out.println("read ");
       visit(read.var);
       System.out.println(";");
    }

    public void visit(Addition addition) {
        visit(addition.left);
        System.out.println(" + ");
        visit(addition.right);
    }

    public void visit(Variable var) {
        System.out.println(var.name);
    }
}
(*)
```

What *visit* do they look for?

```
class Program implements Node {
    List<Statement> statements;
}

class Print implements Statement {
    Expression expr;
}

class Addition implements Expression {
    Expression left, right;
}
```

Centralized Implementation. Main goal

```
void visit(Print print) {
  System.out.println("print ");
  visit(print.expr);
  System.out.println(";");
}

void visit(Addition Addition){
  visit(Addition.left);
  System.out.println(" + ");
  visit(Addition.right);
}

void visit(Variable var) {
   System.out.println(var.name);
}
```



Should we return to the *if/else* with *instanceof*?

Yes… please…!

Solution: Visitor Pattern

```
public static void main(String[] args) {
   Program prog = new Program … // Bild the tree here

   PrintVisitor visitor = new PrintVisitor();
   prog.accept(visitor);
}
```

Interface with a method for each Node

```
public interface Visitor {
   void visitProg(Program p);
   void visitPrint(Print p);
   void visitRead(Read r);
   void visitAddition(Addition s);
   void visitVariable(Variable v);
}
```

It is the Nodes that choose the appropriate method

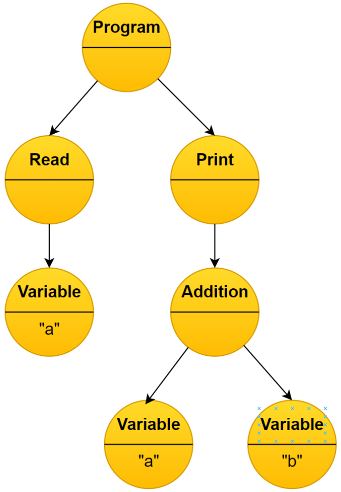
```
public interface Node {
     void accept(Visitor v);
}
```

By redefining the *accept* method, the corresponding *visit* to the Node is chosen

```
public class Print implements Node {
      …
      public void accept(Visitor v) {
         v.visitPrint(this);
      }
}

public class Read implements Node {
      …
     public void accept(Visitor v) {
         v.visitRead(this);
      }
}
```

```
public class PrintVisitor implements Visitor {

    public void visitProg(Program prog) {
        for (Instance instance : prog.Instances)
            instance.accept(this);
     }


    public void visitPrint(Print print) {
         System.out.print("print ");
         print.expr.accept(this);
         System.out.println(";");
     }

    public void visitRead(Read read) {
         System.out.print("read ");
         read.var.accept(this);
         System.out.println(";");
     }


    public void visitAddition(Addition addition)
         Addition.left.accept(this);
         System.out.print(" + ");
         Addition.right.accept(this);
     }

    public void visitVariable(Variable var) {
         System.out.print(var.name);
     }
}
```

From a visit method an accept method is always called (another visit method should never be called)



Optional: Unify names (overload)

The visit method names do not need to be different

```
public interface Visitor {
   void visit~~Prog~~(Program p);
   void visit~~Print~~(Print p);
   void visit~~Read~~(Read r);
   void visit~~Addition~~(Addition s);
   void visit~~Variable~~(Variable v);
}
```

The Node does not change

```
public interface Node {
    void accept(Visitor v);
}
```

But now all *accept* methods are the same.!!

```
public class Print implements Node {
   …
   public void accept(Visitor v) {
      v.visit~~Print~~(this);
   }
}

public class Read implements Node {
   …
   public void accept(Visitor v) {
      v.visit~~Read~~(this);
   }
}
```

The implementation can be copied and pasted to all nodes.

```
public class PrintVisitor implements Visitor {

   public void visit~~Prog~~(Program prog) {
      for (Instance instance : prog.Instances)
          instance.accept(this);
   }

   public void visit~~Print~~(Print print) {
       System.out.print("print ");
       print.expr.accept(this);
       System.out.println(";");
   }

   public void visit~~Read~~(Read read) {
       System.out.print("read ");
       read.var.accept(this);
       System.out.println(";");
   }

   public void visit~~Addition~~(Addition Addition) {
       Addition.left.accept(this);
       System.out.print(" + ");
       Addition.right.accept(this);
   }

   public void visit~~Variable~~(Variable var) {
       System.out.print(var.name);
   }
}
```

Generalizing the Visitor Pattern

The Node must be traversable for any task.

- Some may require parameters and/or return values

Generalizing the Nodes…

```
public interface Node {
   Object accept(Visitor v, Object param);
}


public class Print implements Node {
   …
    public Object accept(Visitor v, Object param) {
        return v.visit(this, param);
    }
}

public class Read implements Node {
   …
    public Object accept(Visitor v, Object param) {
        return v.visit(this, param);
    }
}
```

Generalizing the Visitor…

```
public interface Visitor {
  Object visit(Program p, Object param);
  Object visit(Print p, Object param);
  Object visit(Read r, Object param);
  Object visit(Addition s, Object param);
  Object visit(Variable v, Object param);
}
```

Look at the additional red code with regard to the solution in the previous slide

Implementing the visitor…

- Example of how to implement it when neither the new parameter nor return value is needed

```
public class PrintVisitor implements Visitor {
    public Object visit(Program prog, Object param) {
        for (Instance instance : prog.Instances)
             instance.accept(this, null);
        return null;
    }

    public Object visit(Print print, Object param) {
        System.out.print("print ");
        print.expr.accept(this, null);
        System.out.println(";");
        return null;
    }

    public Object visit(Read read, Object param) {
        System.out.print("read ");
        read.var.accept(this, null);
        System.out.println(";");
        return null;
    }

    public Object visit(Addition Addition, Object param) {
        Addition.left.accept(this, null);
        System.out.print(" + ");
        Addition.right.accept(this, null);
        return null;
    }

    public Object visit(Variable var, Object param){
        System.out.print(var.name);
        return null;
    }
}
```

Summary

a) Steps to implement the Visitor pattern (done only once)

1) Create a **Visitor** interface with a *visit* method for each type of Node in the tree.

```
public interface Visitor {
    public Object visit(Program p, Object param);
    public Object visit(Print p, Object param);
    …
}
```

2) Add an *accept* method to the **Node** interface (thus forcing all **Node**s to implement it).

```
public interface Node {
    Object accept(Visitor v, Object param);
}
```

3) Make all **Node**s implement the *accept* method. Within the *accept* method, only a *visit* method should be called upon.

```
class Print implements Instance {  // Instance implement Node
    …
    Object accept(Visitor v, Object param) {
        return v.visit(this, param);
    }
}

class Read implements Instance {  // Instance implement Node
    …
    Object accept(Visitor v, Object param) {
        return v.visit(this, param);
    }
}
```

Can be copied and pasted to all Nodes

b) To implement a new tree browse (i.e. a new operation on the tree)

The class that implements the operation must only implement Visitor and code all its corresponding methods.

```
public class MiNuevoVisitor implements Visitor {
   …
}
```

But you don't need to modify the Nodes!!!

Summary

a) Steps to implement the Visitor pattern (done only once)

1) M

```
pub

}
```

2) A

```
pub
}
```

3) M

```
cla

}
cla

}
```

Now the “*million dollar*” question. Since all **Nodes** have the same "*accept*" method with the same implementation,…

… would it be possible to create an abstract class "**AbstractNode**", to collect the common code within it, and have **Nodes** inherit from it so that we can remove the repeated implementations in such **Nodes**?

b) To implement a new tree browse (i.e. a new operation on the tree)

The class that implements the operation must only implement Visitor and code all its corresponding methods.

```
public class MiNuevoVisitor implements Visitor {
   …
}
```

But you don't need to touch the Nodes!!!

Summary

a) Steps to implement the Visitor pattern (done only once)

1) M

```
pub
}
```

2)

```
pu
}
```

3)

```
cl
}
cl
}
```

Now the “*million euro*” question. Since all Nodes have the same "accept" method with

repeated implementations on such Nodes?

NOT!, because in that case we return to the original problem, Java has not “<u>Multiple Dispatch</u>”

b) To implement a new tree browse (i.e. a new operation on the tree)

The class that implements the operation must only implement Visitor and code all its corresponding methods.

```
public class MiNuevoVisitor implements Visitor {
   …
}
```

But you don't need to touch the Nodes!!!