

# Patrón Visitor

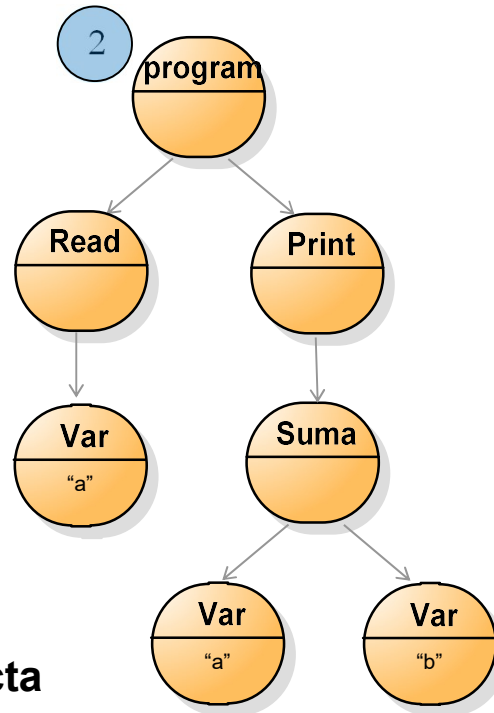
---

**Diseño de Lenguajes de Programación (v1.8)**  
**Raúl Izquierdo Castanedo**

# Modelo de ejemplo

## Ejemplo

1 read a;  
print a + b;



## Gramática Abstracta

3

programa  $\rightarrow$  sentencia\*

read:sentencia  $\rightarrow$  variable

print:sentencia  $\rightarrow$  expr

suma:expr  $\rightarrow$  expr expr

variable:expr  $\rightarrow$  string

## 4 Modelado de los Nodos del Árbol

```
interface Nodo { }
```

```
class Programa implements Nodo {  
    List<Sentencia> sentencias;  
}
```

```
interface Sentencia extends Nodo { }
```

```
class Read implements Sentencia {  
    Variable var;  
}
```

```
class Print implements Sentencia {  
    Expresion expr;  
}
```

```
interface Expresion extends Nodo { }
```

```
class Suma implements Expresion {  
    Expresion left, right;  
}
```

```
class Variable implements Expresion {  
    String name;  
}
```

# Implementación Centralizada. Versión Ideal

```
public class PrintPrograma          // Versión ideal
{
    public void visit(Programa programa) {
        for (Sentencia sent : programa.sentencias)
            visit(sent);
    }

    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(Suma suma) {
        visit(suma.left);
        System.out.println(" + ");
        visit(suma.right);
    }

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

¿Algún problema con esta implementación?

# Versión Ideal. Problema (I)

```
interface Figura
{
}
```

```
class Circulo implements Figura
{
}
```

```
class Prueba
{
    void imprime(Figura f) {
        System.out.println("Figura");
    }

    void imprime(Circulo c) {
        System.out.println("Circulo");
    }

    public static void main(String[] args){
        Figura circulo = new Circulo();
        imprime(circulo);    // ¿Qué sale?
    }
}
```

Hay lenguajes que disponen de esta característica:

- Multiple dispatch

# Versión Ideal. Problema (II)

```
public class PrintPrograma    // versión ideal
{
    public void visit(Programa prog) {
        for (Sentencia sent : prog.sentencias)
            visit(sent);
    }

    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(Suma suma) {
        visit(suma.left);
        System.out.println(" + ");
        visit(suma.right);
    }

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

```
class Programa implements Nodo {
    List<Sentencia> sentencias;
}
```

```
class Print implements Sentencia {
    Expression expr;
}
```

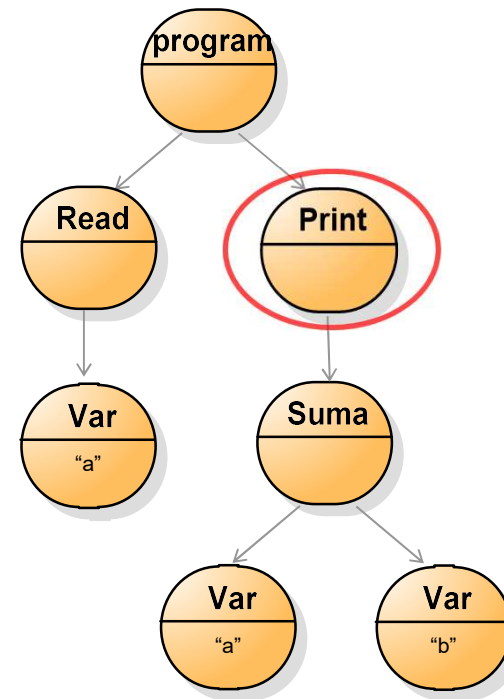
```
class Suma implements Expression {
    Expression left, right;
}
```



¿Qué visit  
buscan?

# Implementación Centralizada. Objetivo

```
void visit(Print print) {  
    System.out.println("print ");  
    visit(print.expr);  
    System.out.println(";");  
}  
  
void visit(Suma suma) {  
    visit(suma.left);  
    System.out.println(" + ");  
    visit(suma.right);  
}  
  
void visit(Variable var) {  
    System.out.println(var.name);  
}
```



- ¿Volvemos a los *if/else* con *instanceof*?

Sí... por favor...

# Solución: Patrón Visitor

```
public static void main(String[] args) {
    Programa prog = new Programa ... // Construir aquí el árbol

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

## Interfaz con un método para cada nodo

```
public interface Visitor {
    void visitProg(Programa p);
    void visitPrint(Print p);
    void visitRead(Read r);
    void visitSuma(Suma s);
    void visitVariable(Variable v);
}
```

## Son los nodos los que eligen el método adecuado

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

## Redefiniendo el método accept se elige el visit correspondiente al nodo

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

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

```
public class PrintVisitor implements Visitor {
```

```
    public void visitProg(Programa prog) {
        for (Sentencia sent : prog.sentencias)
            sent.accept(this);
    }
```

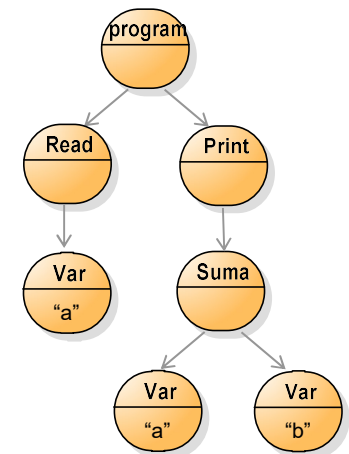
```
    public void visitPrint(Print print) {
        System.out.print("print ");
        print.expr.accept(this);
        System.out.println(";");
    }
```

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

```
    public void visitSuma(Suma suma) {
        suma.left.accept(this);
        System.out.print(" + ");
        suma.right.accept(this);
    }
```

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

Desde un método *visit* siempre se llama a *accept* (nunca a otro *visit*)



# Opcional: Unificar nombres (sobrecarga)

No se necesita que los nombres sean distintos

```
public interface Visitor {  
    void visitProg(Programa p);  
    void visitPrint(Print p);  
    void visitRead(Read r);  
    void visitSuma(Suma s);  
    void visitVariable(Variable v);  
}
```

El Nodo no cambia

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

Pero ahora todos los métodos **accept** son iguales!!

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

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

Se puede copiar y pegar  
en todos los nodos

```
public class PrintVisitor implements Visitor {
```

```
    public void visitProg(Programa prog) {  
        for (Sentencia sent : prog.sentencias)  
            sent.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 visitSuma(Suma suma) {  
        suma.left.accept(this);  
        System.out.print(" + ");  
        suma.right.accept(this);  
    }
```

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



# Generalizando el Patrón Visitor

## El nodo debe poder ser recorrido para cualquier tarea

- Alguna podría requerir parámetros y/o valores de retorno

### Generalizando los nodos...

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

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

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

### Generalizando el Visitor...

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

## Implementando el nuevo Visitor...

- Ejemplo de cómo implementarlo cuando no se necesiten el nuevo parámetro y el valor de retorno

```
public class PrintVisitor implements Visitor {
    public Object visit(Programa prog, Object param) {
        for (Sentencia sent : prog.sentencias)
            sent.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(Suma suma, Object param) {
        suma.left.accept(this, null);
        System.out.print(" + ");
        suma.right.accept(this, null);
        return null;
    }

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

# Resumen

## a) Pasos para implementar el patrón *Visitor* (se hacen *una sola vez*)

1) Hacer un interfaz **Visitor** con un método *visit* por cada tipo de nodo del árbol.

```
public interface Visitor {  
    public Object visit(Programa p, Object param);  
    public Object visit(Print p, Object param);  
    ...  
}
```

2) Añadir un método **accept** al interfaz **Nodo** (así se obliga a que lo implementen *todos* los nodos).

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

3) Hacer que todos los nodos implementen el método **accept**. En él solo tienen que invocar al método *visit*.

```
class Print implements Sentencia { // Sentencia extends Nodo
```

```
    ...  
    Object accept(Visitor v, Object param) {  
        return v.visit(this, param);  
    }  
}
```

```
class Read implements Sentencia { // Sentencia extends Nodo
```

```
    ...  
    Object accept(Visitor v, Object param) {  
        return v.visit(this, param);  
    }  
}
```

Se puede copiar y pegar en todos los nodos

## b) Para implementar un nuevo recorrido del árbol

La clase que implemente el recorrido solo tiene que derivar de **Visitor** y dar implementación a todos sus métodos.

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

Pero no hace falta tocar los nodos!!!