

14

visitor

Diseño del Software

Grado en Ingeniería Informática del Software

2024-2025

Patrón de comportamiento, de objetos

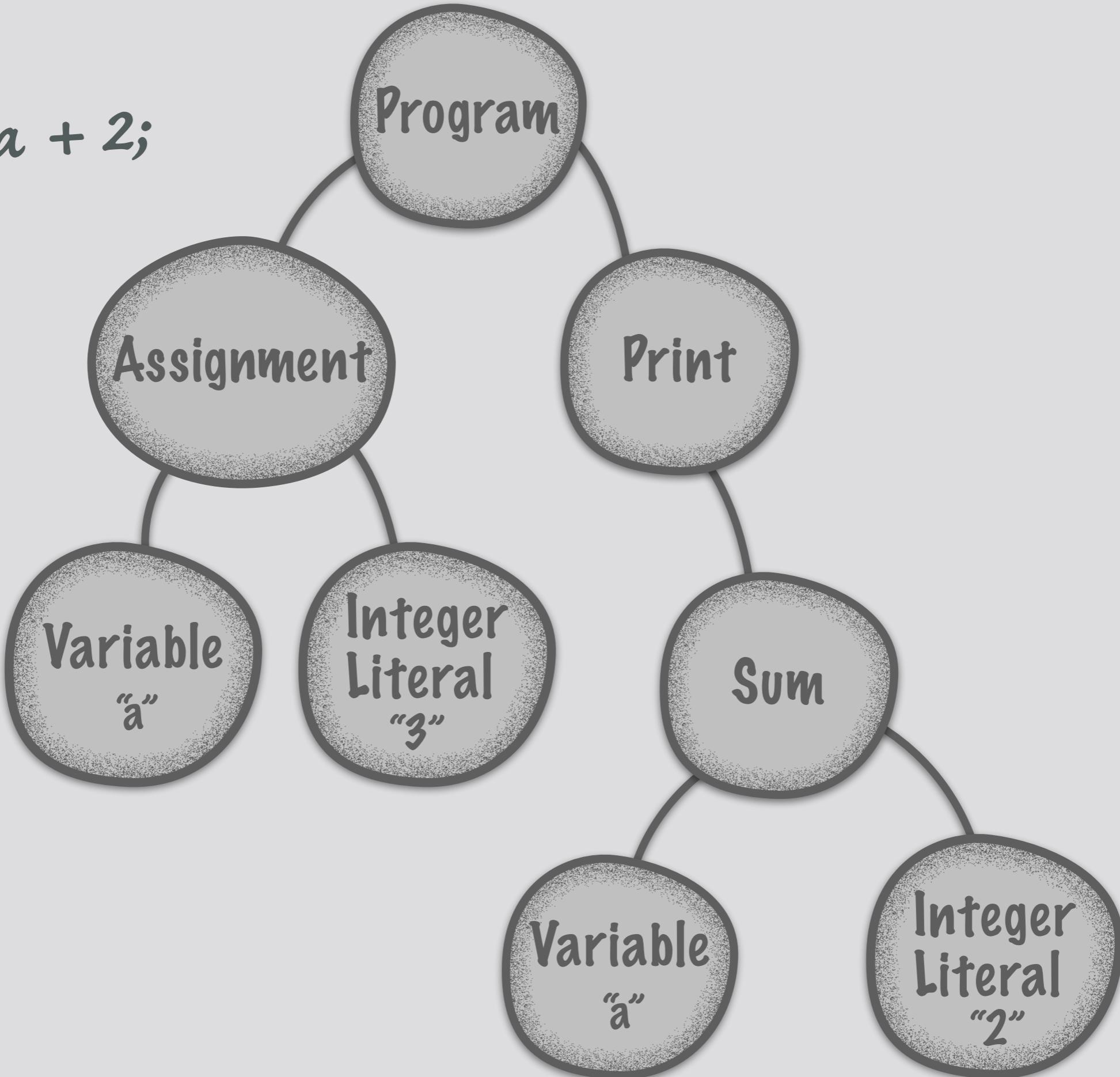
Representa una operación a realizar sobre una estructura de objetos.

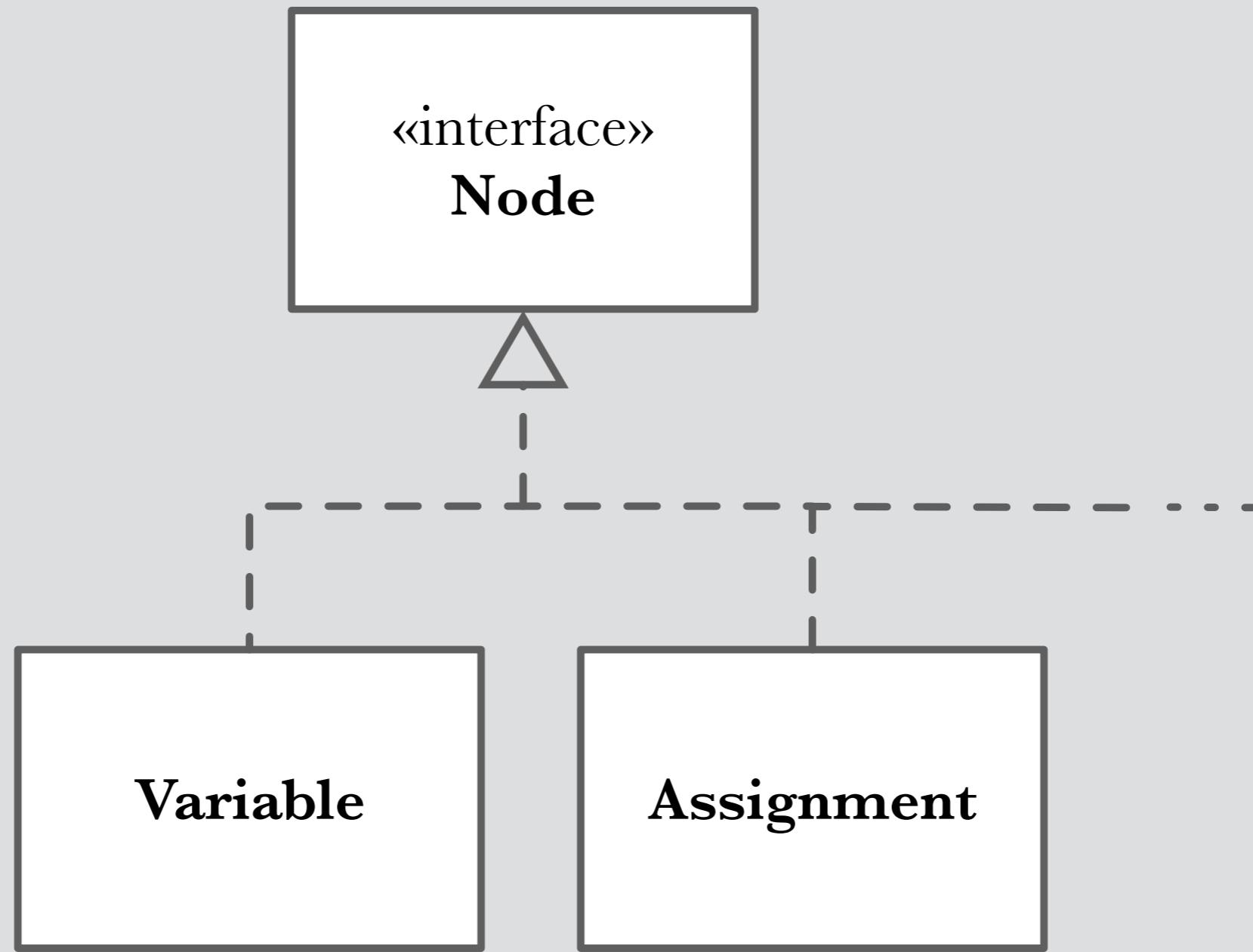
Permite definir nuevas operaciones sin modificar las clases de los elementos sobre los que opera.

Motivación

Un compilador suele representar los programas como árboles sintácticas abstractos (AST), donde cada sentencia y expresión representan distintas clases de nodos.

```
a = 3;  
print a + 2;
```





Necesitaremos llevar a cabo distintas operaciones sobre el árbol.

Resaltado de sintaxis

Análisis semántico

Comprobación de errores

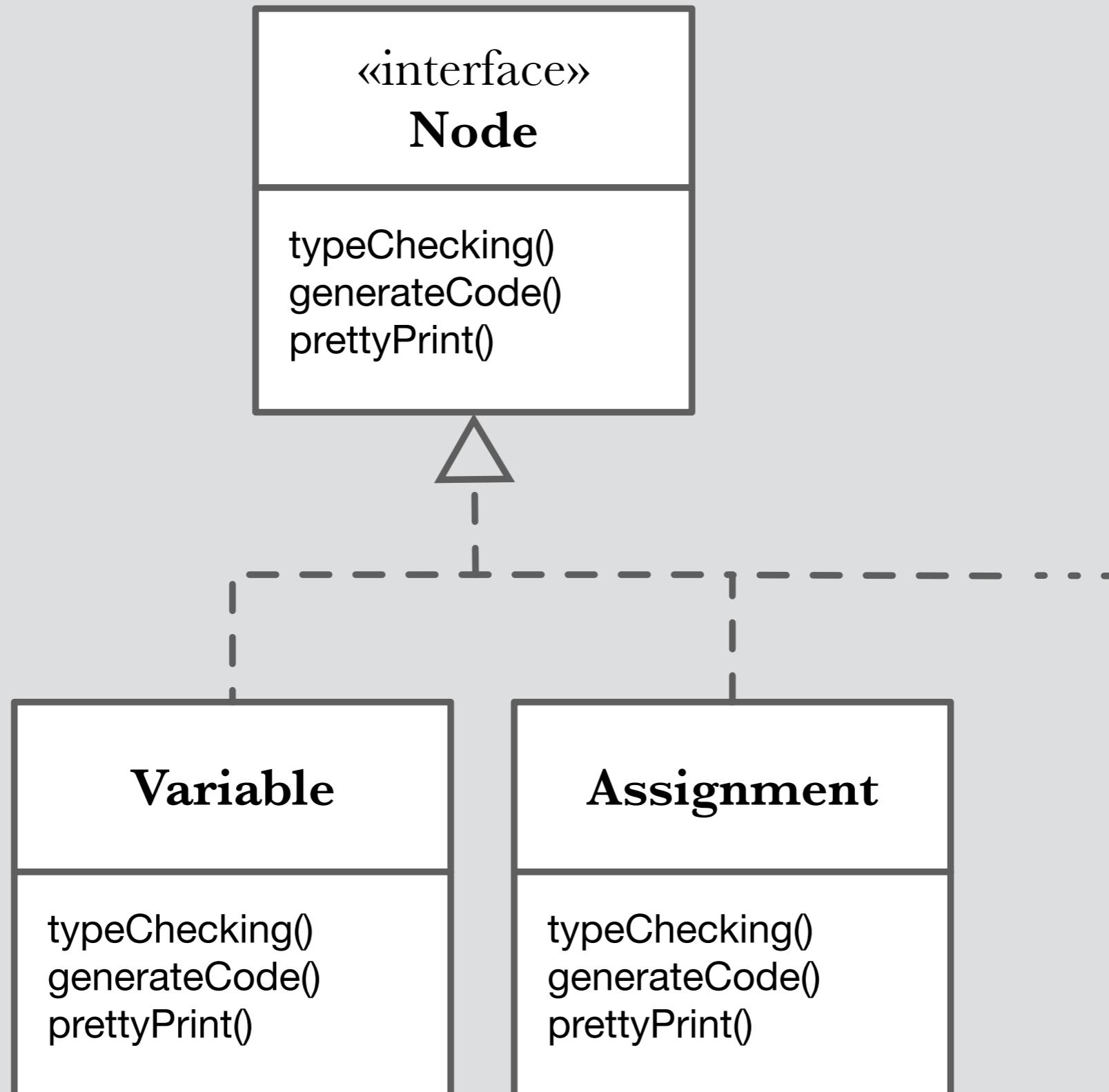
Generación de código

Etcétera.

El enfoque típico orientado a objetos
consistiría en implementar esas operaciones
directamente en los propios nodos.

Es como lo hace el patrón Interpreter.

Lo que hicimos en la práctica 2 de la máquina virtual.



¿Qué problemas tiene ese enfoque?

**Estamos mezclando
responsabilidades que nada
tienen que ver entre sí.**

Violando así el principio de responsabilidad única.

El código de comprobación de tipos aparecerá mezclado con el de resaltar la sintaxis o el de generación de código, dando lugar a clases muy poco cohesivas.

**Cada operación está diseminada
por todas las clases de nodos.**

Lo que conduce a un sistema difícil de entender, mantener
y cambiar.

Dificulta añadir nuevas operaciones.

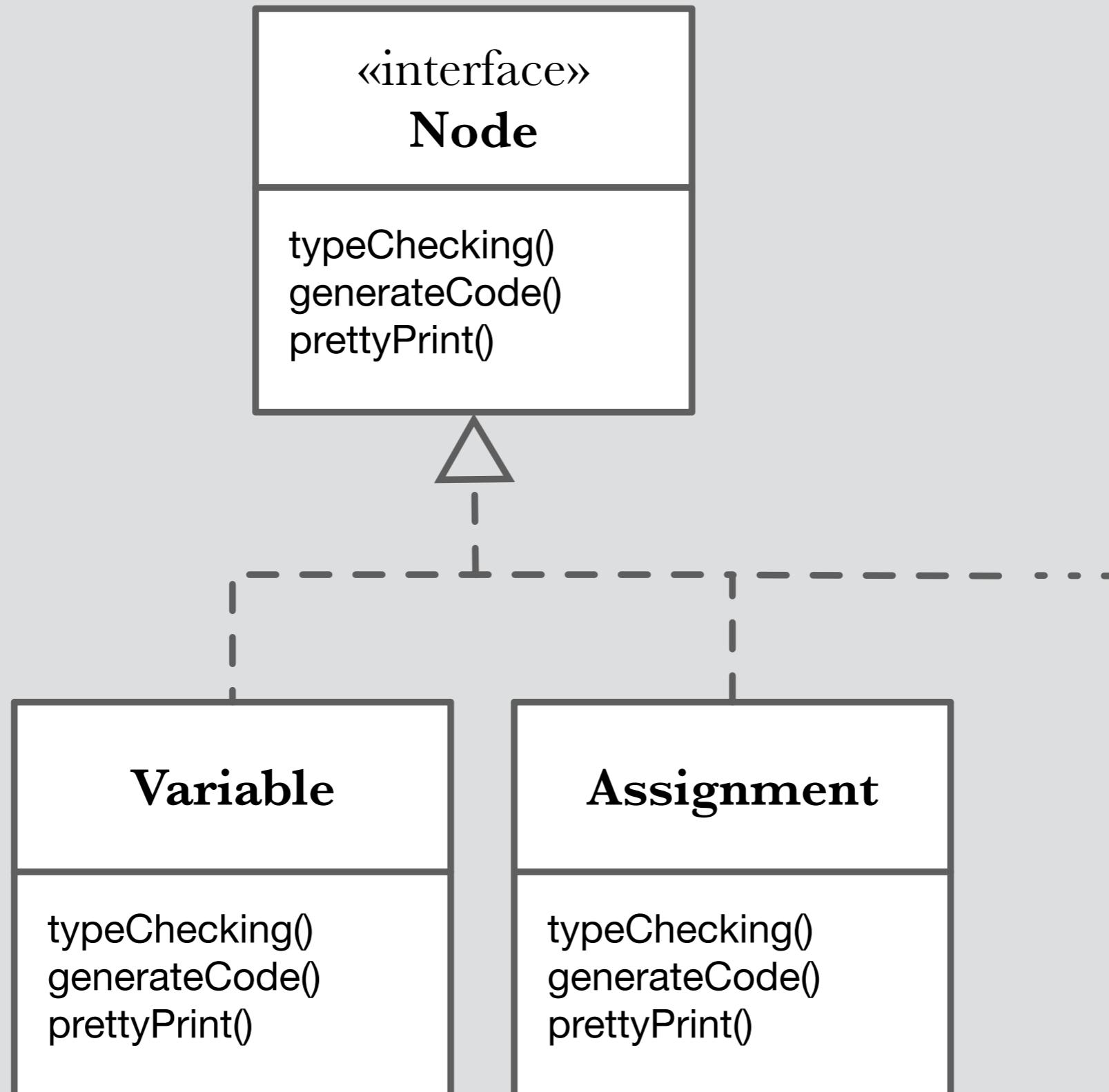
Requiere cambiar la interfaz raíz de los nodos y modificar todas las clases.

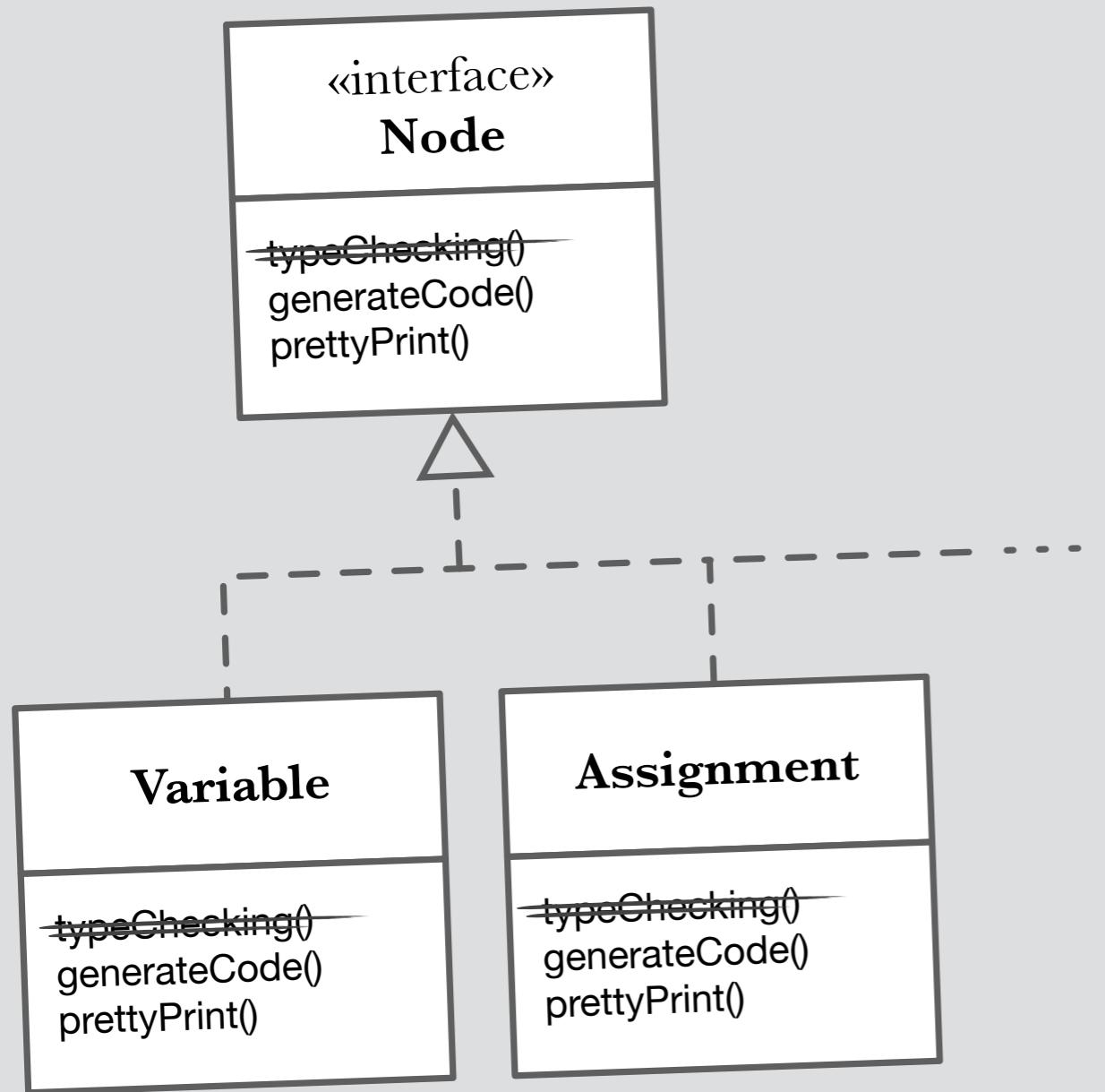
Solución

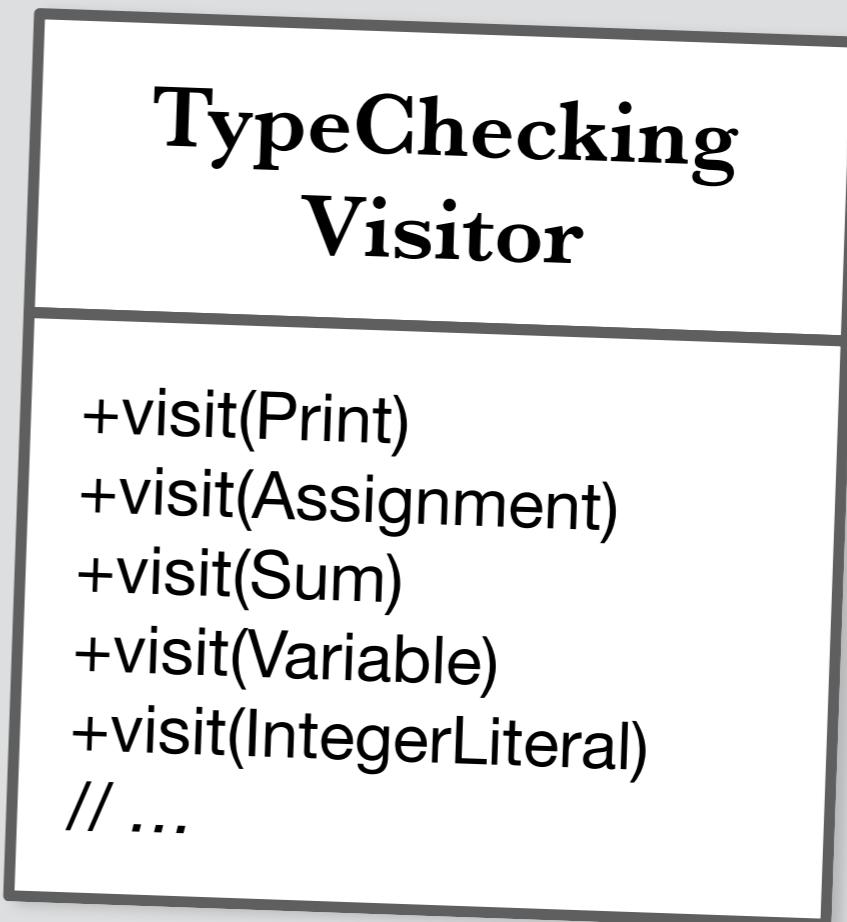
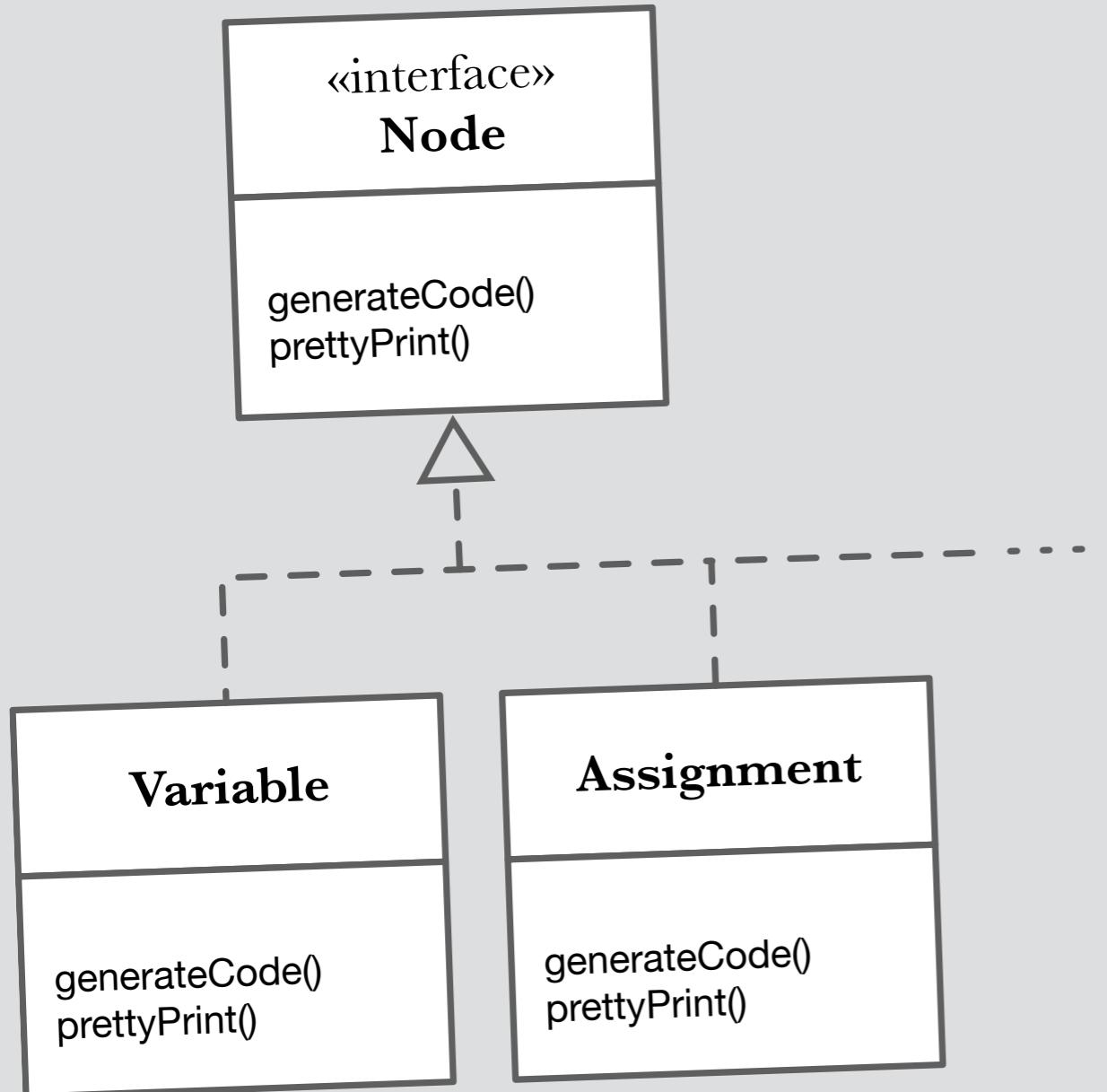
Encapsular cada operación y extraerlas de cada clase de nodo a una clase aparte.

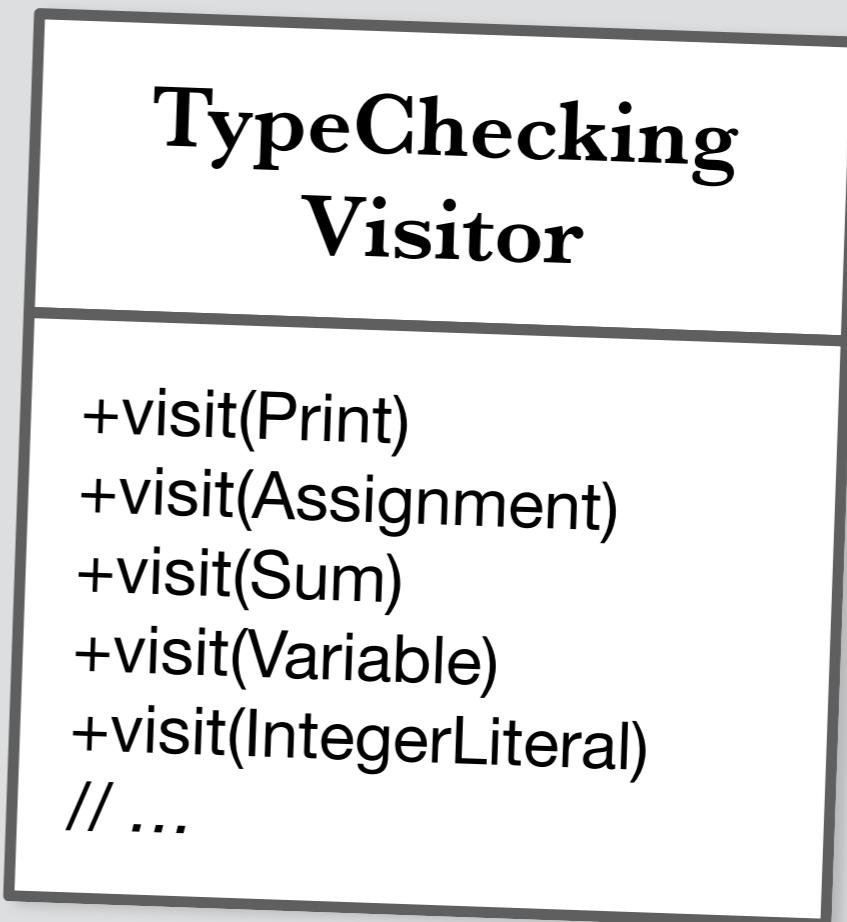
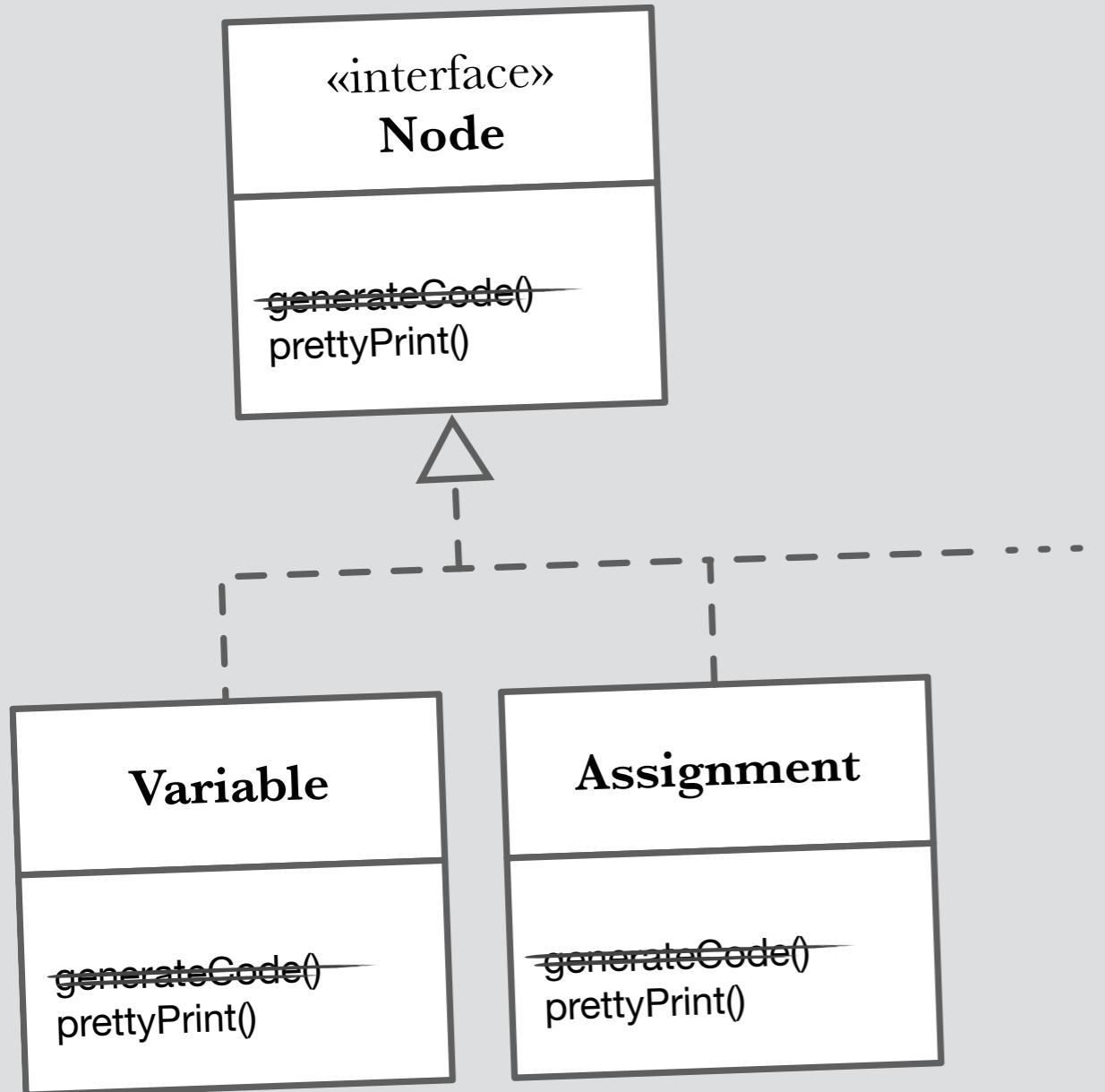
A dicha clase se la conoce como visitante.

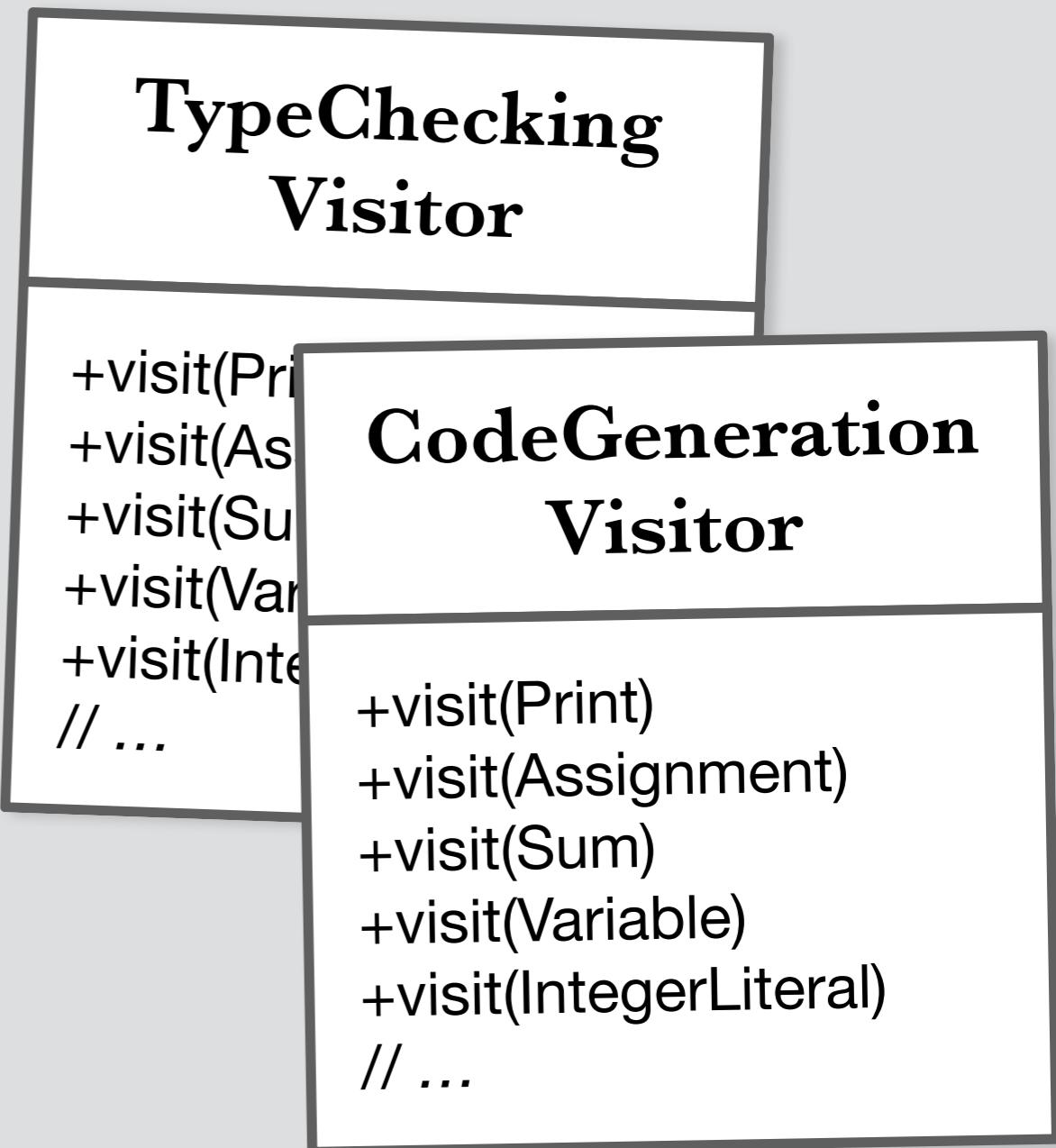
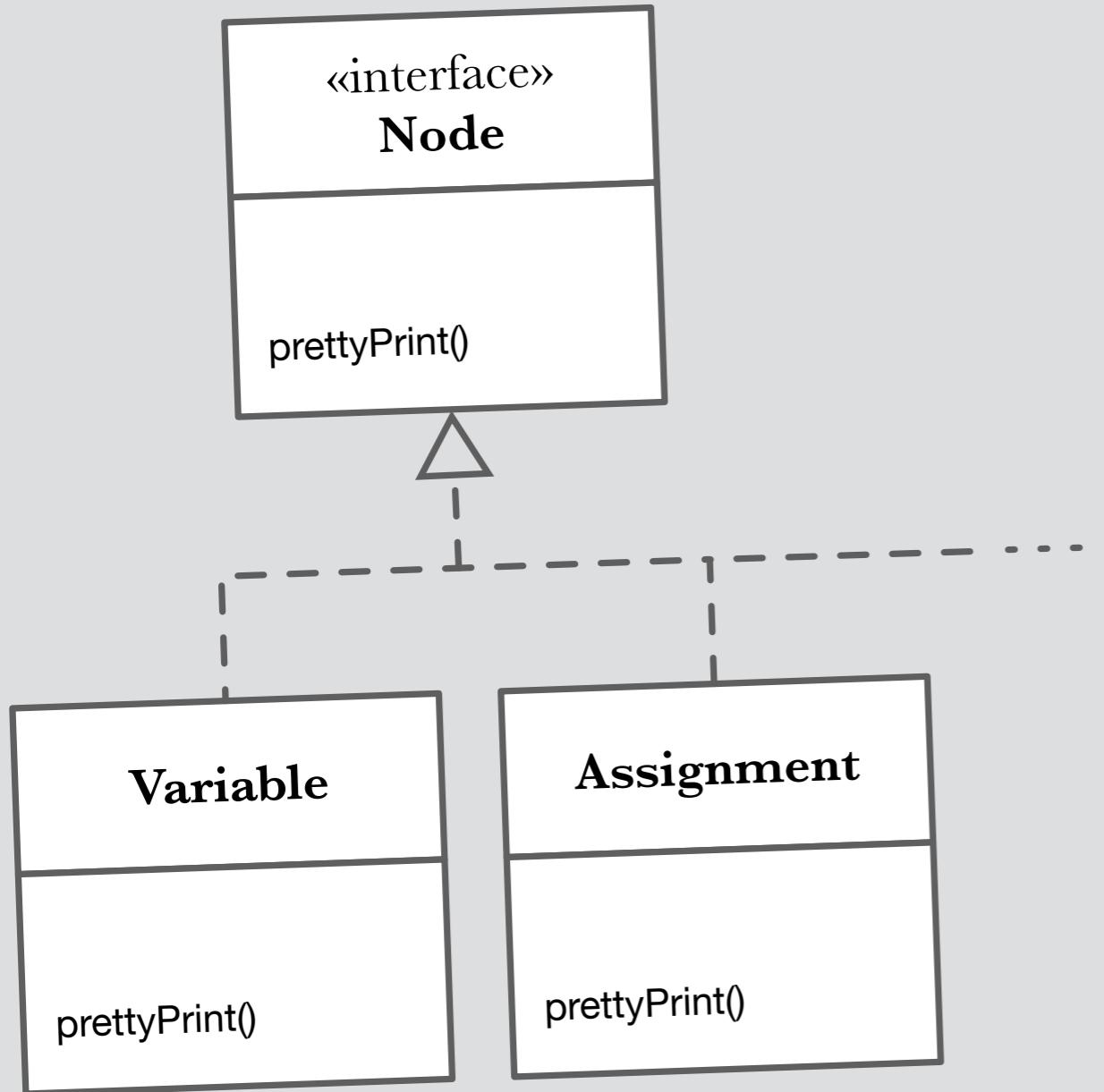
Recorrerá la estructura de nodos aplicando su operación a cada elemento del árbol.

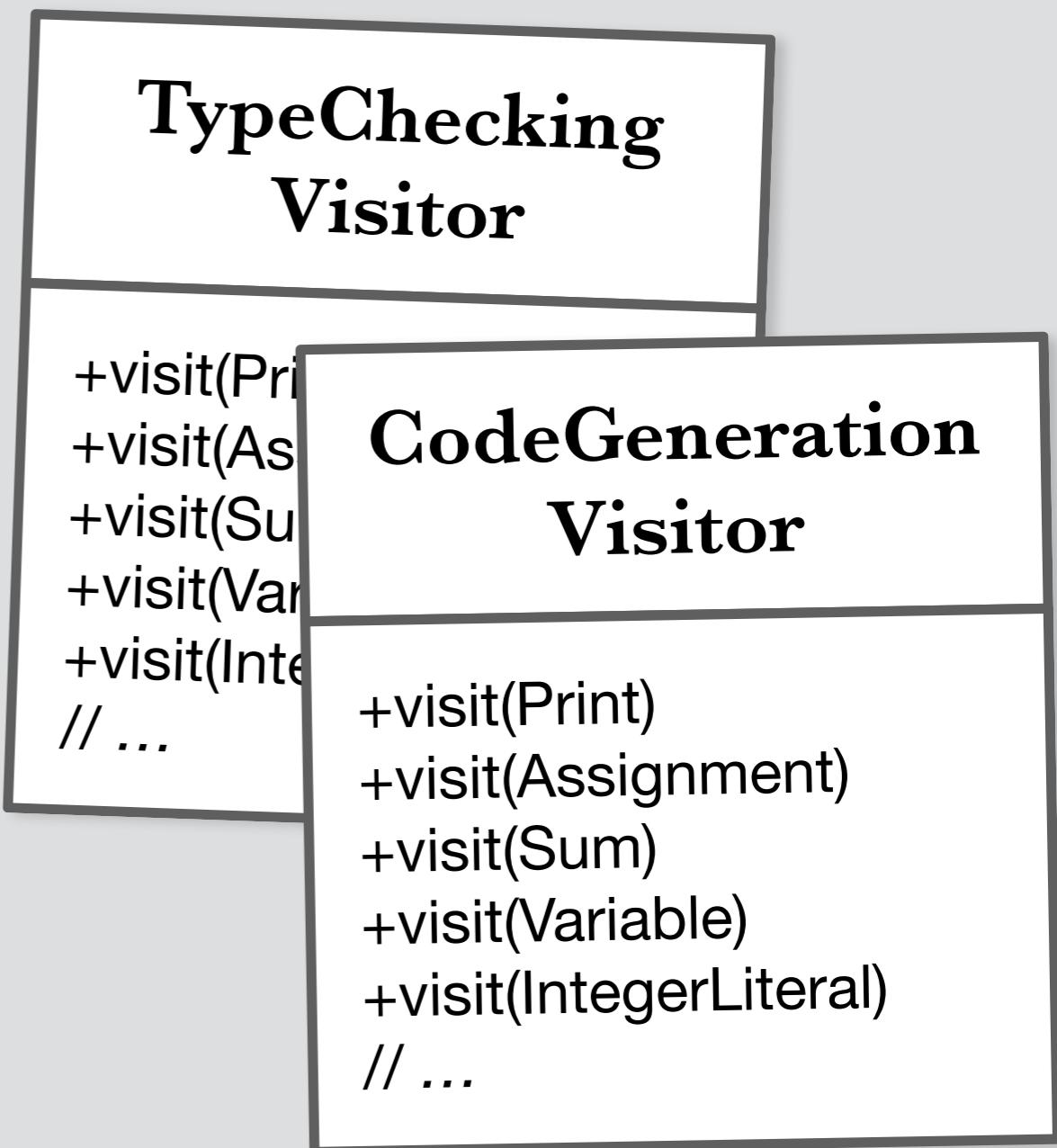
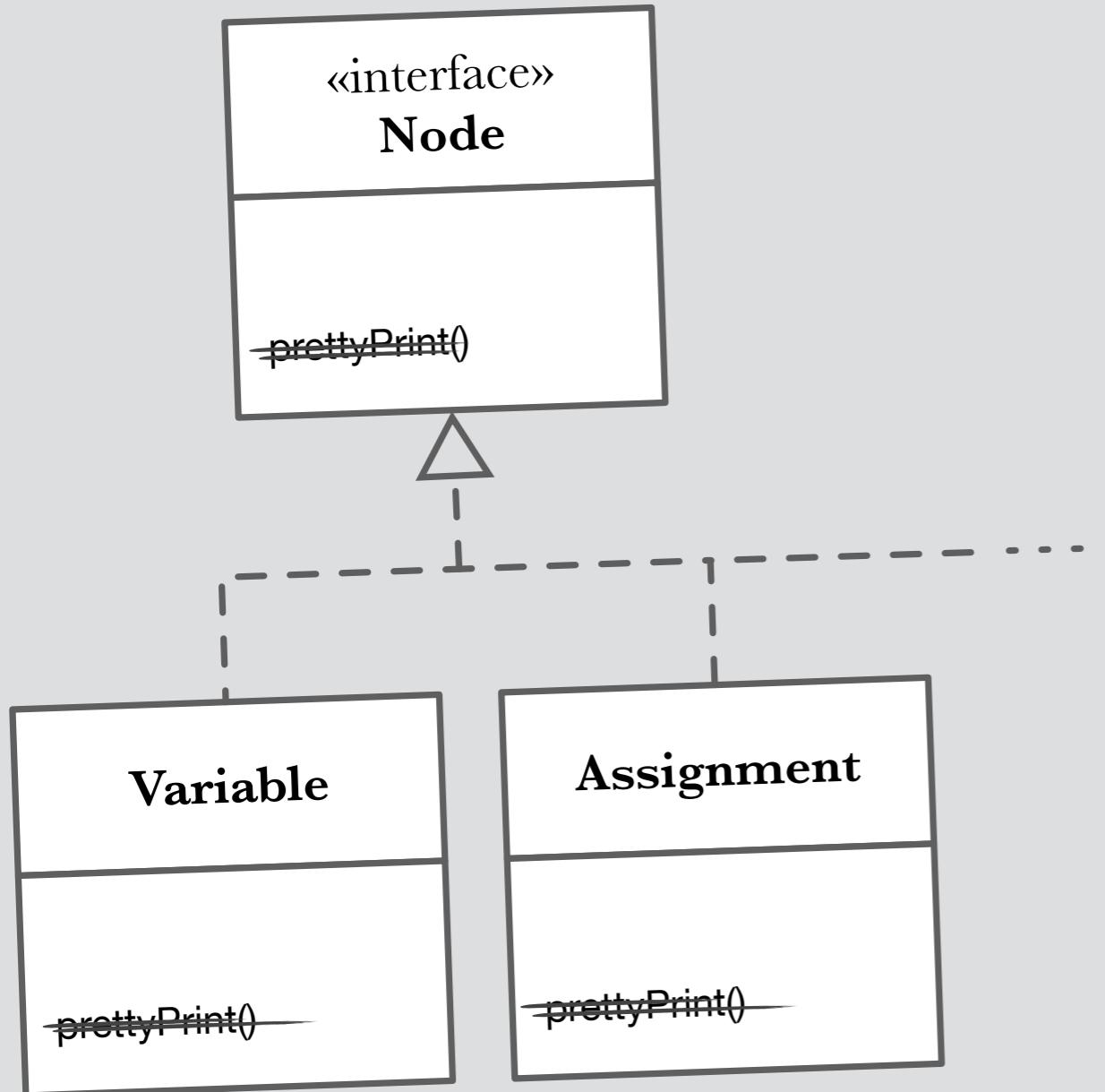


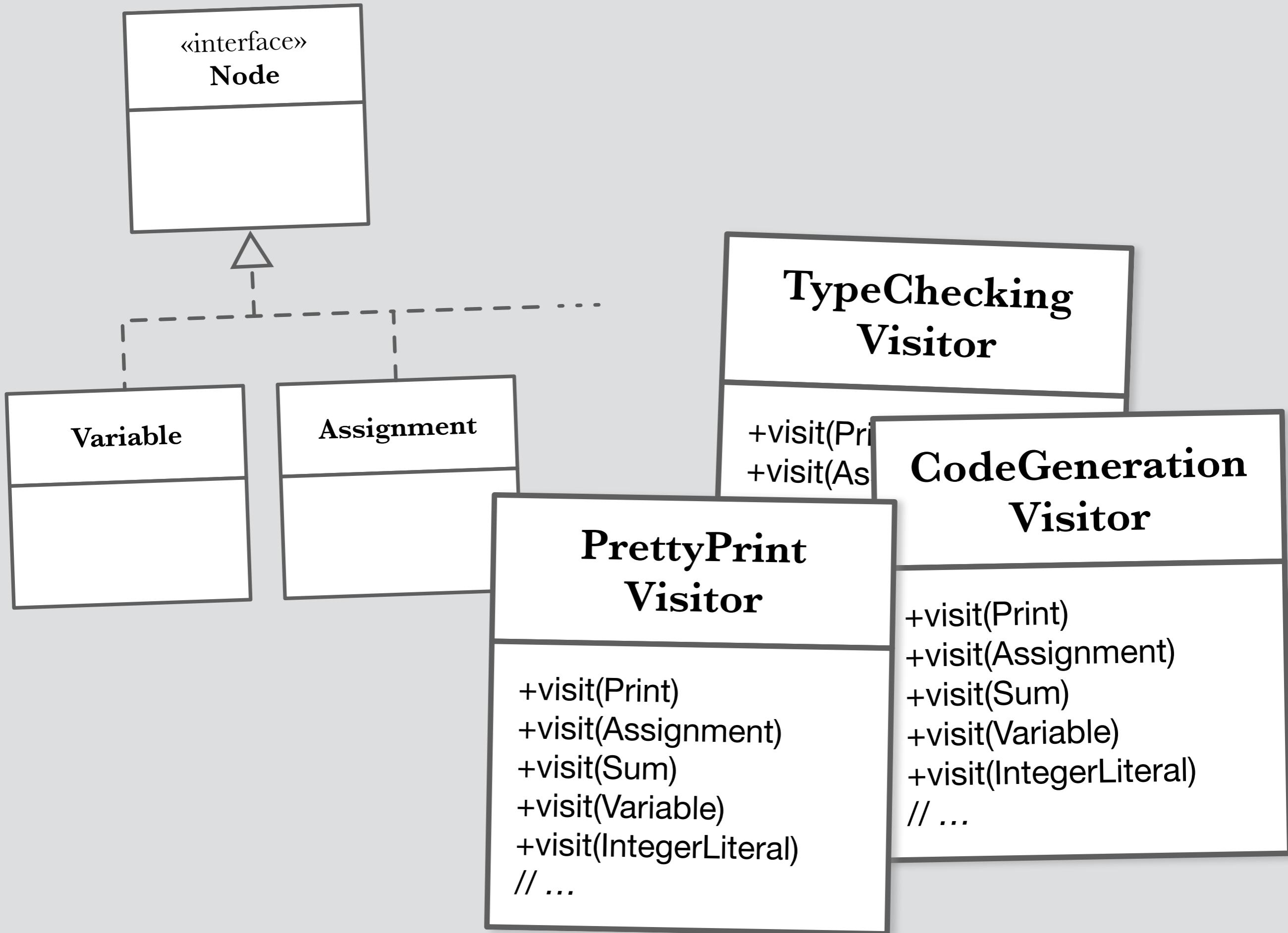


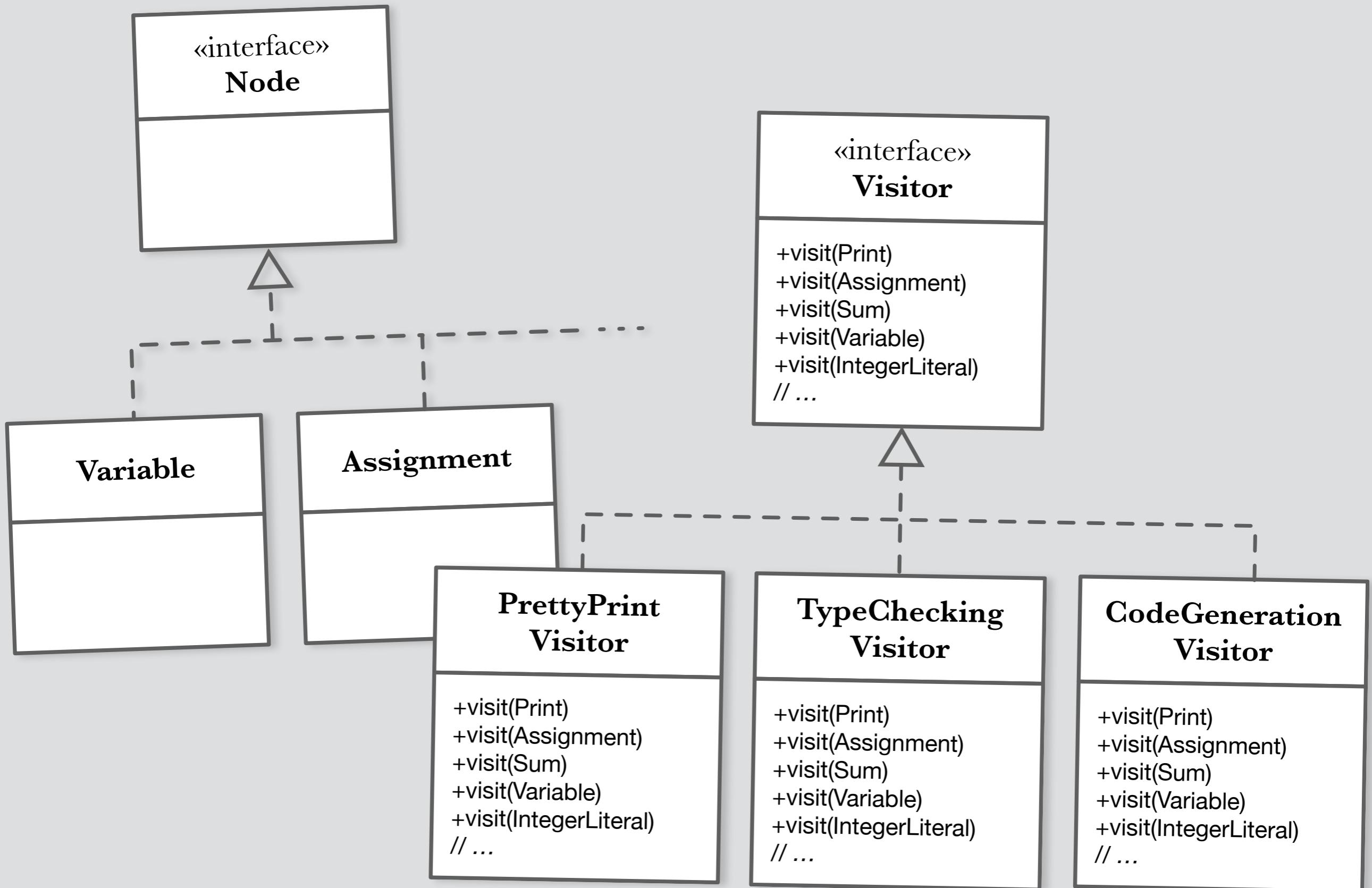








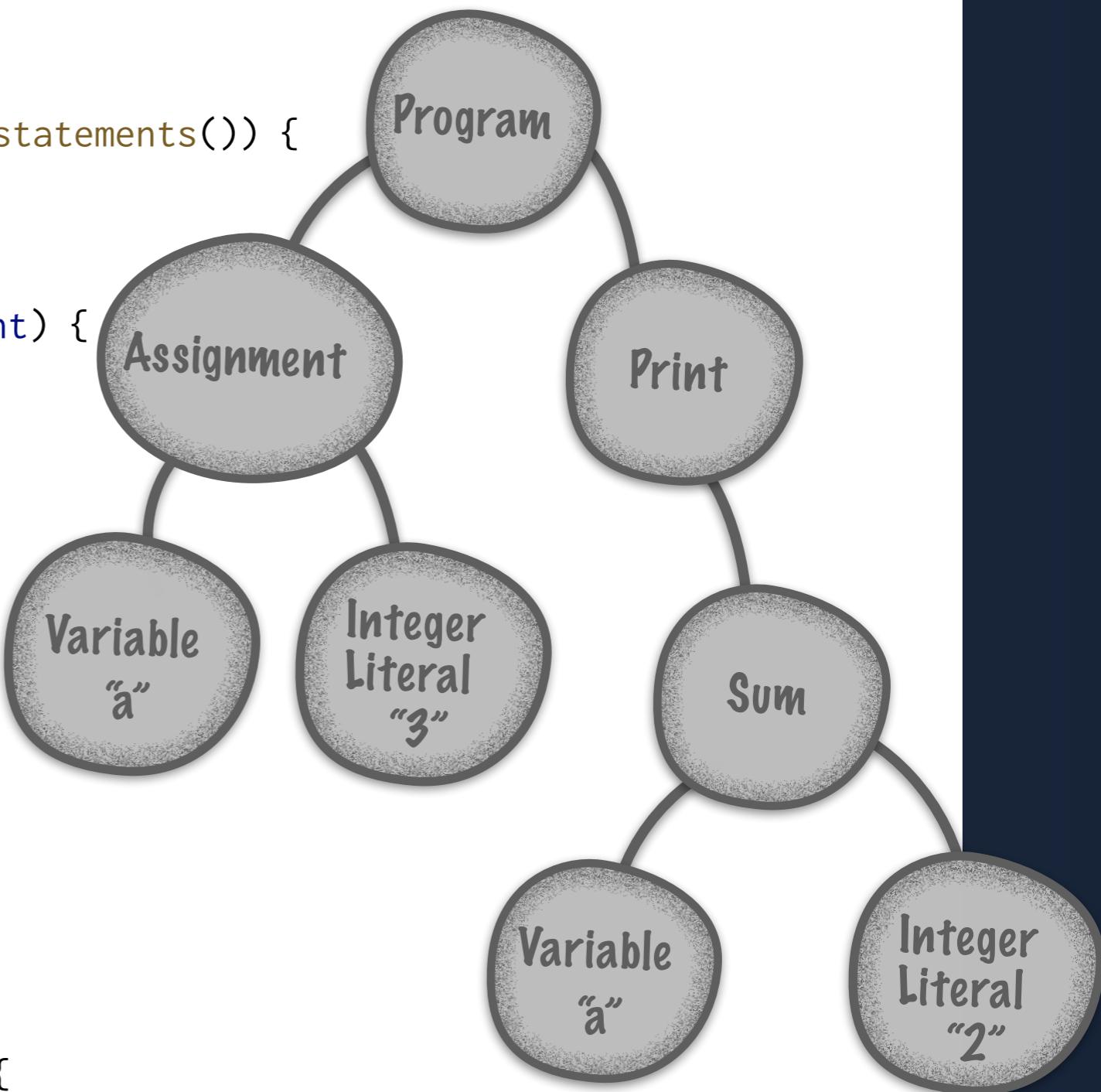




Esto es lo que nos gustaría



```
public class IdealPrint {  
    public void visit(Program program) {  
        for (Statement statement : program.statements()) {  
            visit(statement);  
        }  
    }  
  
    public void visit(Assignment assignment) {  
        visit(assignment.variable());  
        System.out.print(" = ");  
        visit(assignment.expression());  
        System.out.println(";");
    }  
  
    public void visit(Print print) {  
        System.out.print("print ");  
        visit(print.expression());  
        System.out.println(";");
    }  
  
    public void visit(Sum sum) {  
        visit(sum.left());  
        System.out.print(" + ");
        visit(sum.right());
    }  
  
    public void visit(Variable variable) {  
        System.out.print(variable.name());
    }  
  
    public void visit(IntegerLiteral number) {  
        System.out.print(number.value());
    }
}
```



*a = 3;
print a + 2;*

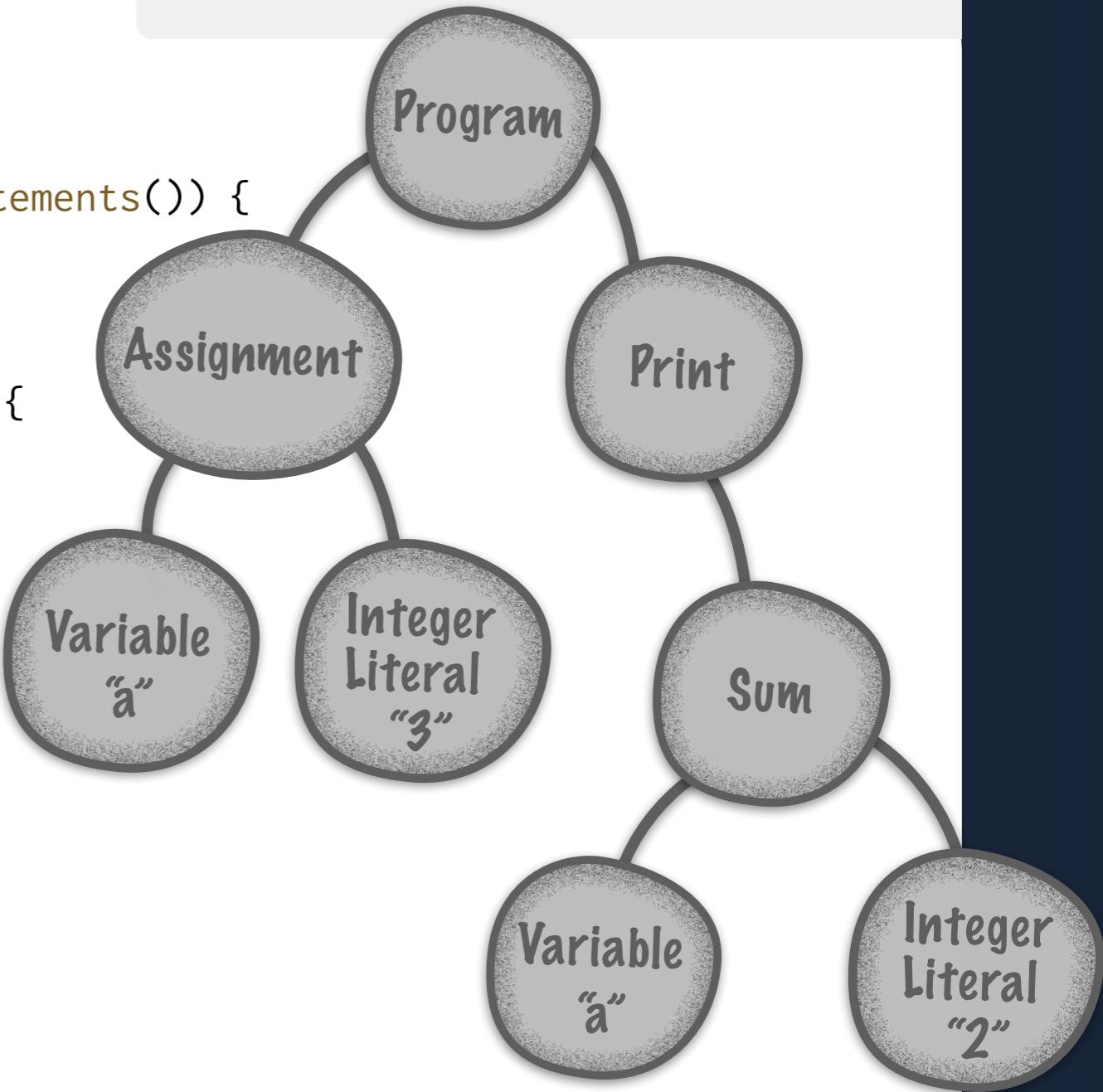
El problema es que

¡No compila!

¿Por qué?



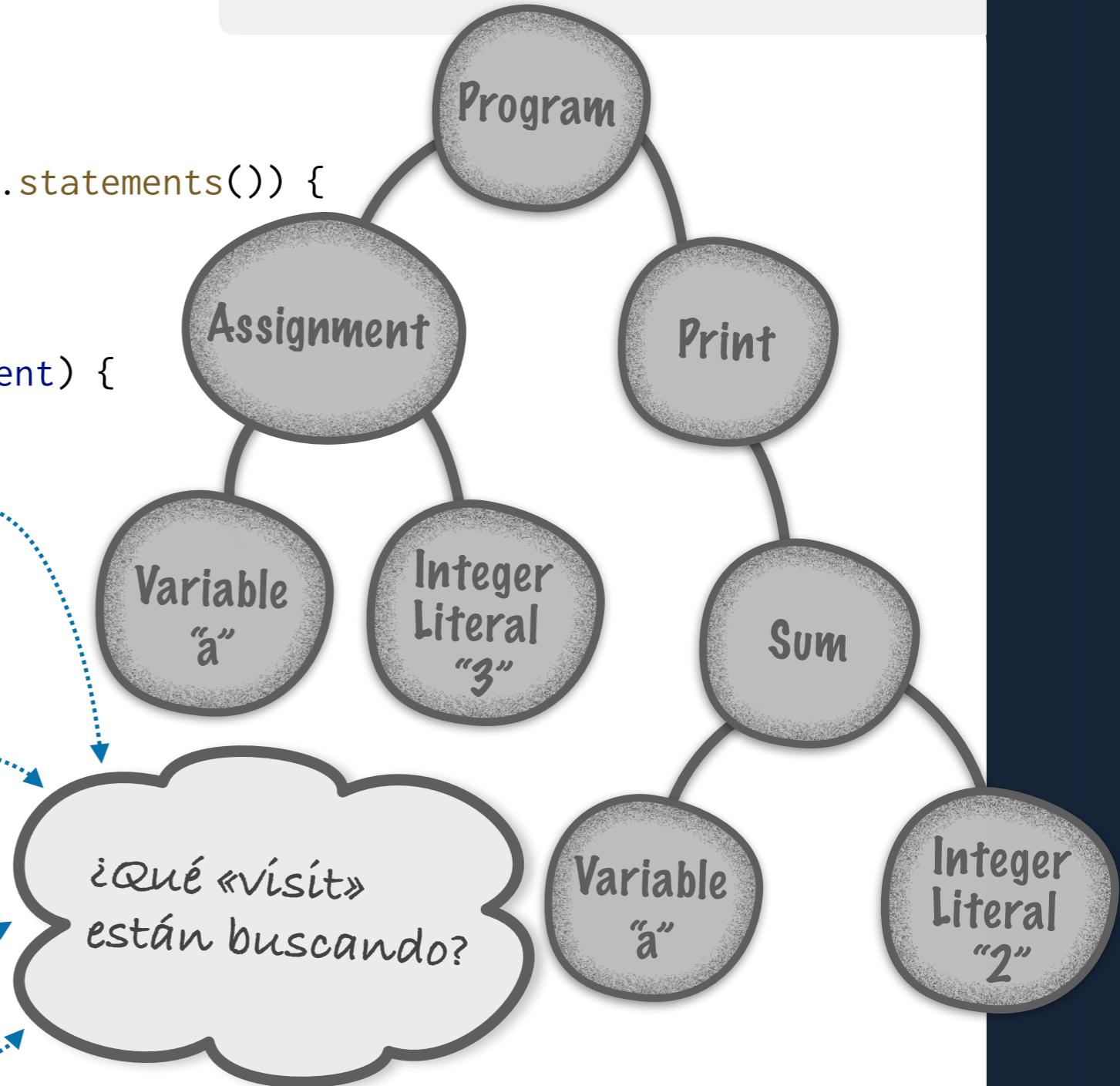
```
public class IdealPrint {  
    public void visit(Program program) {  
        for (Statement statement : program.statements()) {  
            visit(statement);  
        }  
    }  
  
    public void visit(Assignment assignment) {  
        visit(assignment.variable());  
        System.out.print(" = ");  
        visit(assignment.expression());  
        System.out.println(";");
    }  
  
    public void visit(Print print) {  
        System.out.print("print ");  
        visit(print.expression());  
        System.out.println(";");
    }  
  
    public void visit(Sum sum) {  
        visit(sum.left());  
        System.out.print(" + ");
        visit(sum.right());
    }  
  
    public void visit(Variable variable) {  
        System.out.print(variable.name());
    }  
  
    public void visit(IntegerLiteral number) {  
        System.out.print(number.value());
    }
}
```



$a = 3;$
 $print a + 2;$



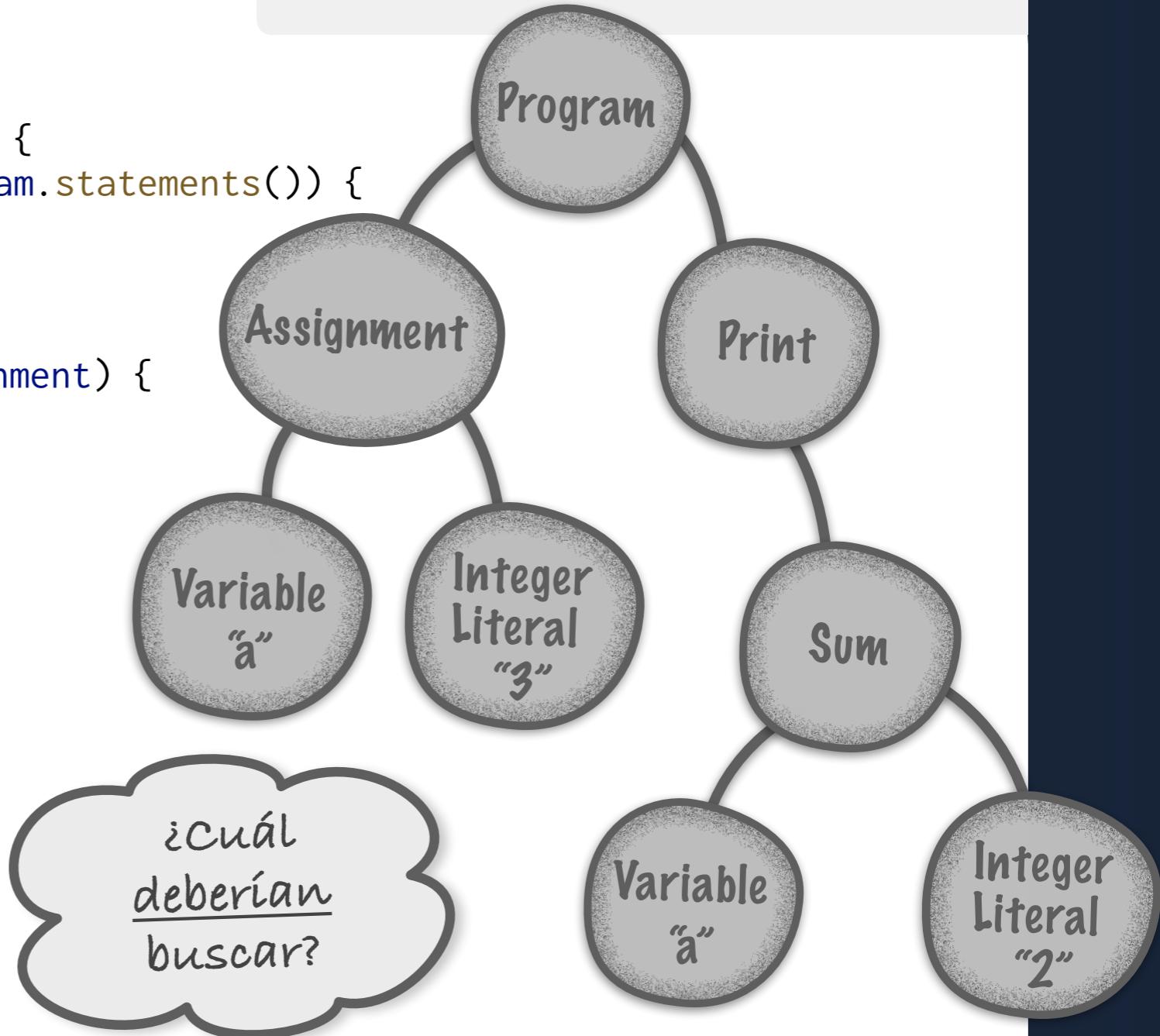
```
public class IdealPrint {  
    public void visit(Program program) {  
        for (Statement statement : program.statements()) {  
            visit(statement);  
        }  
    }  
  
    public void visit(Assignment assignment) {  
        visit(assignment.variable());  
        System.out.print(" = ");  
        visit(assignment.expression());  
        System.out.println(";");
    }  
  
    public void visit(Print print) {  
        System.out.print("print ");  
        visit(print.expression());  
        System.out.println(";");
    }  
  
    public void visit(Sum sum) {  
        visit(sum.left());  
        System.out.print(" + ");  
        visit(sum.right());
    }  
  
    public void visit(Variable variable) {  
        System.out.print(variable.name());
    }  
  
    public void visit(IntegerLiteral number) {  
        System.out.print(number.value());
    }
}
```



$a = 3;$
 $print a + 2;$



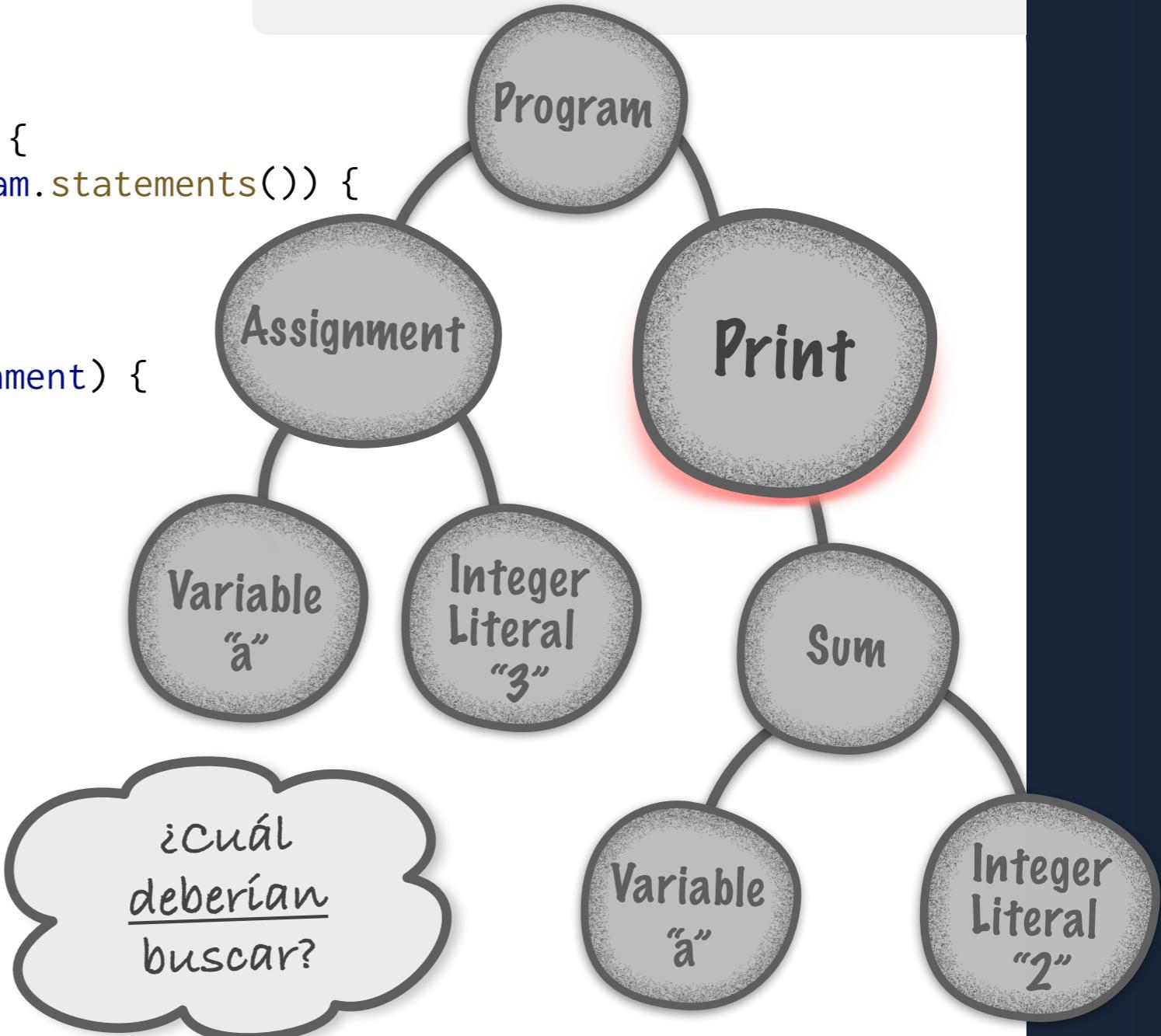
```
public class IdealPrint {  
    public void visit(Program program) {  
        for (Statement statement : program.statements()) {  
            visit(statement);  
        }  
    }  
  
    public void visit(Assignment assignment) {  
        visit(assignment.variable());  
        System.out.print(" = ");  
        visit(assignment.expression());  
        System.out.println(";");
    }  
  
    public void visit(Print print) {  
        System.out.print("print ");  
        visit(print.expression());  
        System.out.println(";");
    }  
  
    public void visit(Sum sum) {  
        visit(sum.left());  
        System.out.print(" + ");
        visit(sum.right());
    }  
  
    public void visit(Variable variable) {  
        System.out.print(variable.name());
    }  
  
    public void visit(IntegerLiteral number) {  
        System.out.print(number.value());
    }
}
```



$a = 3;$
 $print a + 2;$



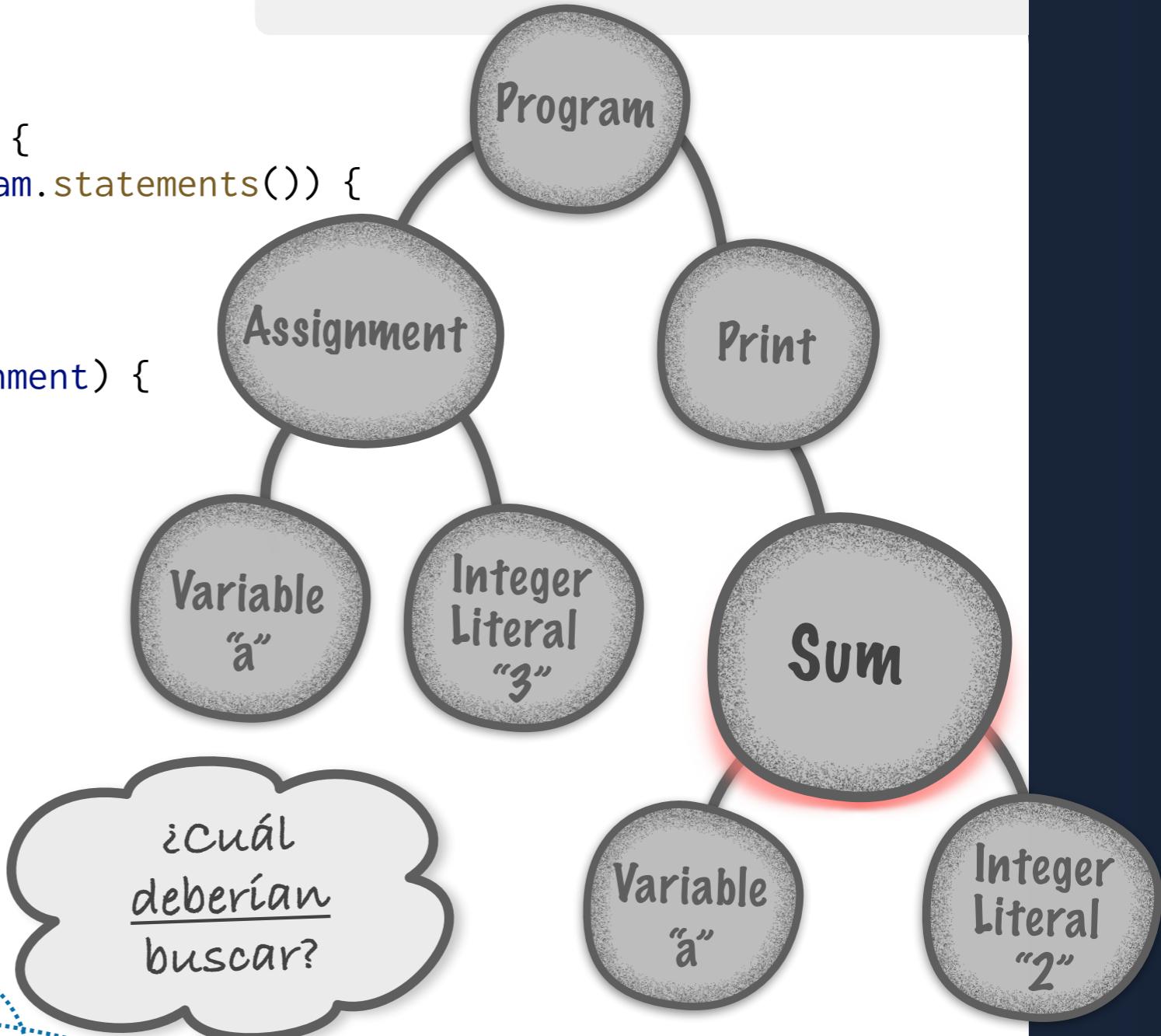
```
public class IdealPrint {  
    public void visit(Program program) {  
        for (Statement statement : program.statements()) {  
            visit(statement);  
        }  
    }  
  
    public void visit(Assignment assignment) {  
        visit(assignment.variable());  
        System.out.print(" = ");  
        visit(assignment.expression());  
        System.out.println(";");
    }  
  
    public void visit(Print print) {  
        System.out.print("print ");  
        visit(print.expression());  
        System.out.println(";");
    }  
  
    public void visit(Sum sum) {  
        visit(sum.left());  
        System.out.print(" + ");  
        visit(sum.right());
    }  
  
    public void visit(Variable variable) {  
        System.out.print(variable.name());
    }  
  
    public void visit(IntegerLiteral number) {  
        System.out.print(number.value());
    }
}
```



$a = 3;$
 $print a + 2;$



```
public class IdealPrint {  
    public void visit(Program program) {  
        for (Statement statement : program.statements()) {  
            visit(statement);  
        }  
    }  
  
    public void visit(Assignment assignment) {  
        visit(assignment.variable());  
        System.out.print(" = ");  
        visit(assignment.expression());  
        System.out.println(";");
    }  
  
    public void visit(Print print) {  
        System.out.print("print ");  
        visit(print.expression());  
        System.out.println(";");
    }  
  
    public void visit(Sum sum) {  
        visit(sum.left());  
        System.out.print(" + ");  
        visit(sum.right());
    }  
  
    public void visit(Variable variable) {  
        System.out.print(variable.name());
    }  
  
    public void visit(IntegerLiteral number) {  
        System.out.print(number.value());
    }
}
```



¿cuál
deberían
buscar?

$a = 3;$
 $print a + 2;$

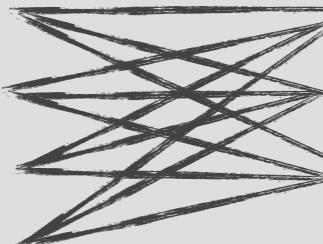
El problema

Supongamos que tenemos una jerarquía de figuras y otra de impresoras.

Supongamos que la impresora tiene un método para imprimir cada figura.

Y que, a su vez, cada impresora las imprime de formas distintas.

figuras	impresoras
rectángulo	láser
círculo	chorro de tinta
óvalo	matricial
línea	...
...	





LaserPrint.java

```
class LaserPrinter implements Printer {  
    public void print(Rectangle rectangle) { ... }  
    public void print(Circle circle) { ... }  
    // ...  
}
```



InkjetPrint.java

```
class InkjetPrinter implements Printer {  
    public void print(Rectangle rectangle) { ... }  
    public void print(Circle circle) { ... }  
    // ...  
}
```



```
for (Figure figure : figures) {  
    for (Printer printer : printers) {  
        printer.print(figure);  
    }  
}
```

Esto es lo que nos gustaría hacer

¿Funciona?



```
for (Figure figure : figures) {  
    for (Printer printer : printers) {  
        printer.print(figure);  
    }  
}
```

¿Funciona?



```
for (Figure figure : figures) {  
    for (Printer printer : printers) {  
        printer.print(figure);  
    }  
}
```



LaserPrint.java

```
class LaserPrinter implements Print {  
    public void print(Rectangle rectangle) {  
        public void print(Circle circle) {  
            // ...  
        }  
    }  
}
```



InkjetPrint.java

```
class InkjetPrinter implements Print {  
    public void print(Rectangle rectangle) {  
        public void print(Circle circle) {  
            // ...  
        }  
    }  
}
```

The screenshot shows a Java code editor interface with a dark theme. The top bar includes standard window controls (red, yellow, green), a back/forward button, a search bar containing "double-dispatch", and a tab bar with multiple tabs. The left sidebar contains icons for Explorer, Search, Problems, File Explorer, Run, and Java Projects. The Explorer view shows a project structure under "DOUBL...": .vscode, bin, src (containing figures and printers packages), main (containing Main.java), printers (containing InkjetPrinter.java, LaserPrinter.java, and Printer.java), and .gitignore. The bottom status bar displays file navigation, SonarLint status, and code analysis details.

```
4
5 import figures.*;
6 import printers.*;
7
8 public class Main {
9
10    public static void main(String[] args) {
11
12        List<Printer> printers = List.of(new InkjetPrinter(), new LaserPrinter());
13        List<Figure> figures = List.of(new Rectangle(), new Circle());
14
15        // Prints every figure in every printer
16        for (Figure figure : figures) {
17            for (Printer printer : printers) {
18                printer.print(figure);
19            }
20        }
21    }
22}
23
```

The screenshot shows a Java code editor interface with a dark theme. The top bar includes standard window controls (red, yellow, green dots), a back/forward button, a search bar containing "double-dispatch", and a tab bar with multiple tabs. The left sidebar contains icons for Explorer, Search, Issues, Outline, Timeline, SonarLint, and Java Projects. The Explorer view shows a project structure under "DOUBL...": .vscode, bin, src (containing figures and printers packages), main (containing Main.java, Circle.java, Figure.java, Rectangle.java, and Printer.java), and .gitignore. The main editor area displays Main.java:

```
4  
5 import figures.*;  
6 import printers.*;  
7  
8 public class Main {  
9  
10    public static void main(String[] args) {  
11        list<Printer> printers = list.of(new InkjetPrinter(), new LaserPrinter());  
12        printers.print(rectangle);  
13    }  
14    void printers.Printer.print(Rectangle rectangle) {  
15        printer.print(figure);  
16    }  
17    Rectangle rectangle = new Rectangle();  
18    printer.print(figure);  
19    }  
20    }  
21    }  
22    }  
23 }
```

A SonarLint tooltip is displayed over the line "printers.print(rectangle);", indicating a warning: "The method print(Rectangle) in the type Printer is not applicable for the arguments (Figure) Java(67108979)". It also shows code completion suggestions: "void printers.Printer.print(Rectangle rectangle)", "View Problem (F8)", "Quick Fix... (⌘.)", and "Fix using Copilot (⌘I)".

The bottom status bar shows file navigation icons (main, up, down, search), Java status ("Java: Ready"), SonarLint focus ("SonarLint focus: overall code"), and encoding/encoding options ("Spaces: 4", "UTF-8", "LF", "{} Java").

¿Y si añadimos otro método que reciba
un objeto **Figure**?



Printer.java

```
interface Printer {  
    void print(Rectangle rectangle);  
    void print(Circle circle);  
}
```



Printer.java

```
interface Printer {  
    void print(Figure figure);  
    void print(Rectangle rectangle);  
    void print(Circle circle);  
}
```

The screenshot shows a Java project named "double-dispatch-v1" in the VS Code interface. The project structure in the Explorer sidebar includes ".vscode", "bin", "src" (with "figures" and "printers" subfolders containing "Circle.java", "Figure.java", "Rectangle.java", "InkjetPrinter.java", "LaserPrinter.java", and "Printer.java"), and ".gitignore". The "Main.java" file is the active editor, showing the following code:

```
import java.util.List;
import figures.*;
import printers.*;

public class Main {
    public static void main(String[] args) {
        Run | Debug
    }
}
```

The "TERMINAL" tab shows the output of running the application:

```
acebal@Mac-Studio-de-Cesar double-dispatch-v1 % /usr/bin/env /Library/Java/JavaVirtualM  
hines/jdk-23.jdk/Contents/Home/bin/java --enable-preview -XX:+ShowCodeDetailsInExceptionMe  
ssages -cp /Users/acebal/Desktop/Code,double-dispatch-v1/bin main.Main  
Inkjet printer => Printing a figure  
Laser printer => Printing a figure  
Inkjet printer => Printing a figure  
Laser printer => Printing a figure
```

The status bar at the bottom indicates "Java: Ready" and "SonarLint focus: overall code".

¿Qué ocurre?

Que en Java, C# y la mayoría de lenguajes de programación, el enlace dinámico no se aplica a los parámetros, sino solo al receptor del mensaje.

Es una decisión de diseño de dichos lenguajes.

Los parámetros, en estos lenguajes, son «ciudadanos de segunda».

Comparados con el receptor del mensaje (lo que está a la izquierda del punto en una llamada a un método).

Para ellos, solo se tiene en cuenta el tipo estático de la referencia, no de qué clase concreta es el objeto en tiempo de ejecución (no hay, por así decirlo, «polimorfismo de parámetros»).

Una posibilidad



AbstractPrinter.java

```
abstract class AbstractPrinter implements Printer {  
    @Override  
    public void print(Figure figure) {  
        if (figure instanceof Rectangle) {  
            print((Rectangle) figure);  
        } else if (figure instanceof Circle) {  
            print((Circle) figure);  
        }  
    }  
    @Override  
    public abstract void print(Rectangle rectangle);  
    @Override  
    public abstract void print(Circle circle);  
}
```

Pero...

Perdemos la comprobación estática de tipos.

Aun extrayendo el código para cada tipo de nodo a un método abstracto, seguimos dependiendo de ese método general para el nodo raíz, con lógica condicional e `instanceof`, para decidir a qué método llamar.

Otra posibilidad

Añadamos un nuevo método
a `Figure`.

Lo llamaremos `println`.



Figure.java

```
public interface Figure {  
    void draw();  
    // New method for printing  
    void printIn(Printer printer);  
}
```

Ahora, le damos la misma implementación en todas las clases de figuras.



Rectangle.java

```
class Rectangle implements Figure {  
    // ...  
    @Override  
    public void printIn(Printer printer) {  
        printer.print(this);  
    }  
}
```



Circle.java

```
class Circle implements Figure {  
    // ...  
    @Override  
    public void printIn(Printer printer) {  
        printer.print(this);  
    }  
}
```



Rectangle.java

```
class Rectangle implements Figure {  
    // ...  
    @Override  
    public void printIn(Printer printer) {  
        printer.print(this);  
    }  
}
```



Circle.java

```
class Circle implements Figure {  
    // ...  
    @Override  
    public void printIn(Printer printer) {  
        printer.print(this);  
    }  
}
```

Por último, en vez de llamar al imprimir de la impresora directamente, lo haremos a través del método `println` de `Figure`.



```
for (Figure figure : figures) {  
    for (Printer printer : printers) {  
        printer.print(figure);  
    }  
}
```

Antes



```
for (Figure figure : figures) {  
    for (Printer printer : printers) {  
        figure.printIn(printer);  
    }  
}
```

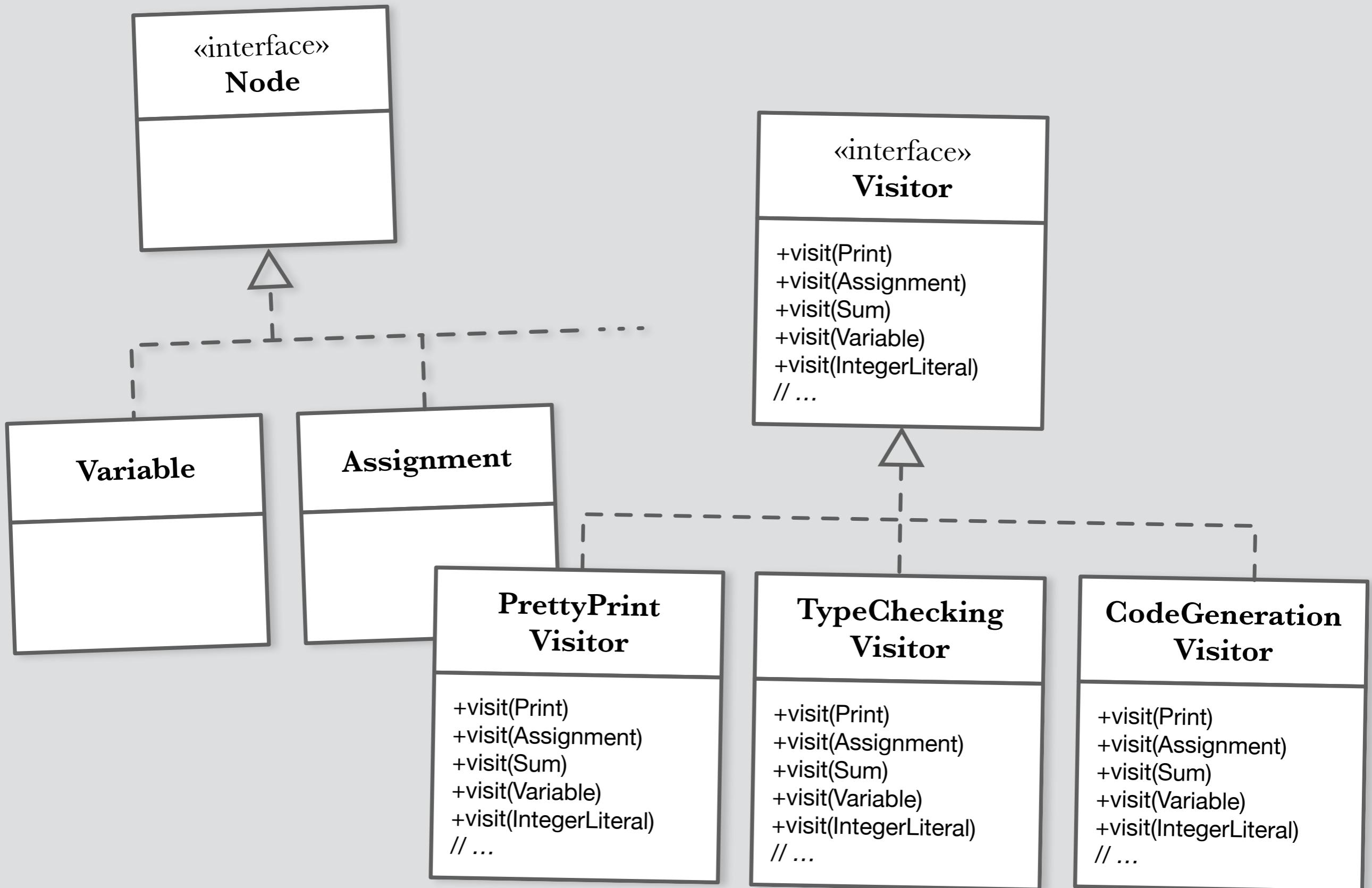
Después

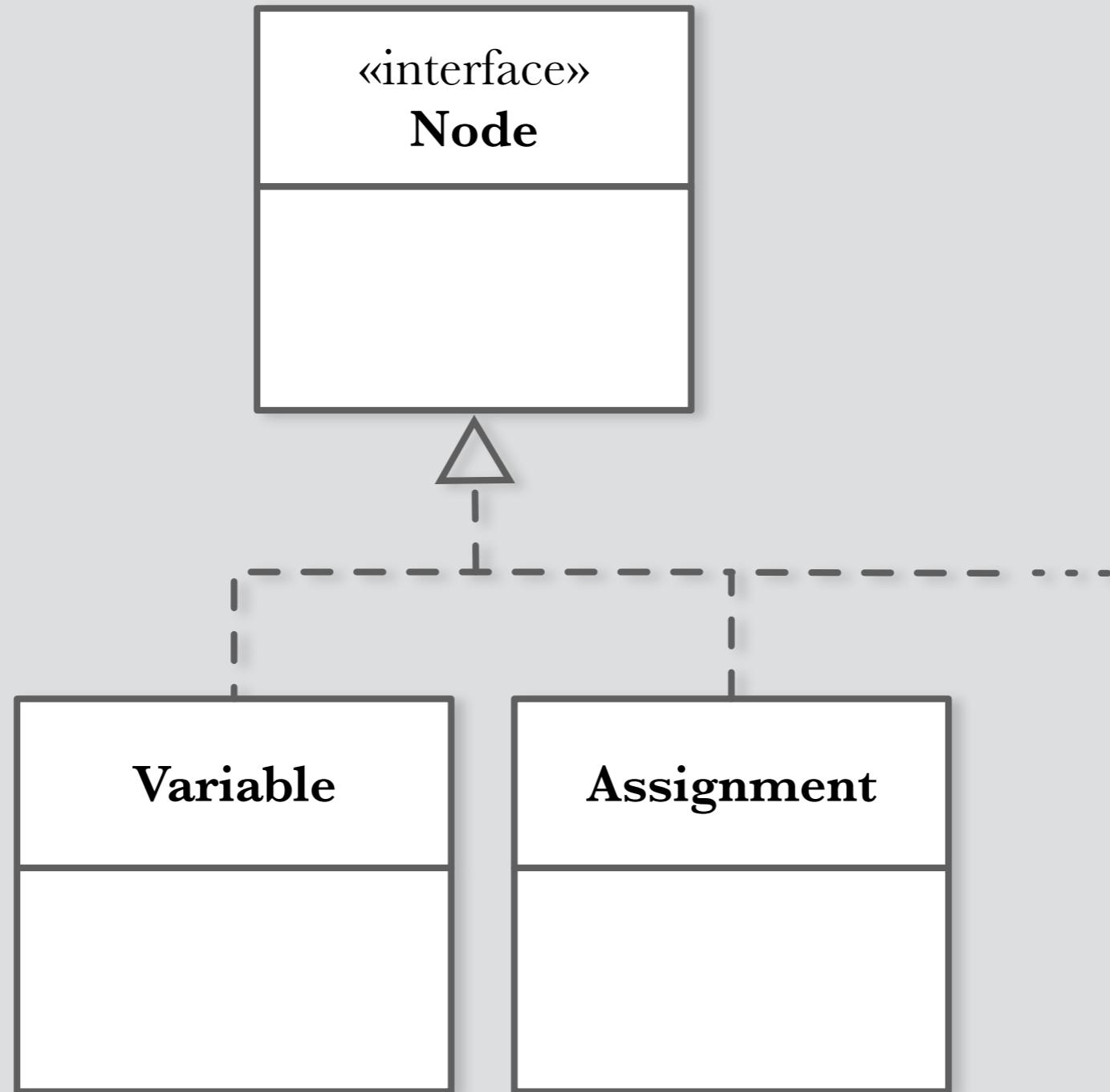
Despacho doble

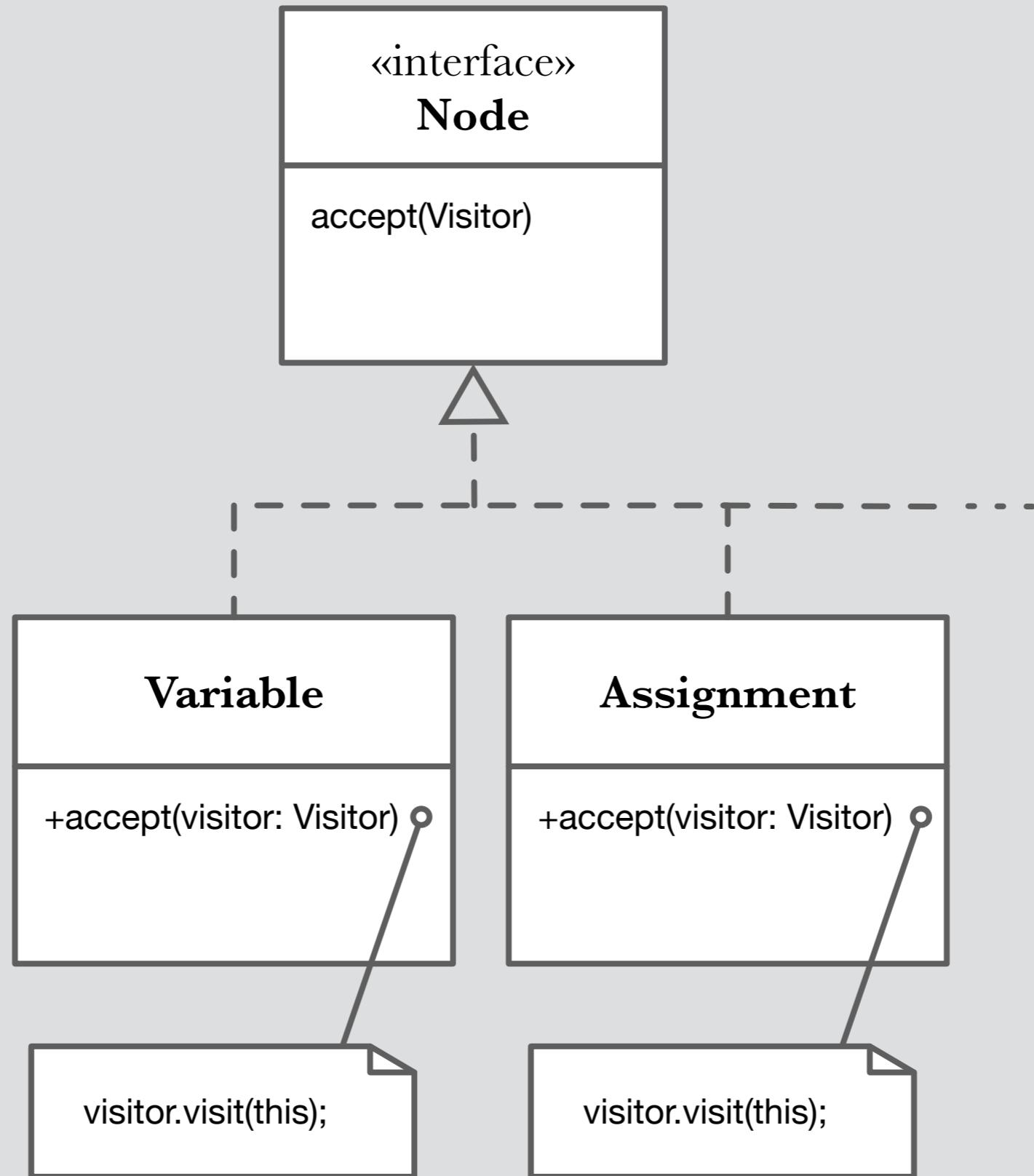
Lo que hemos hecho no es más que una forma de simular el despacho doble.

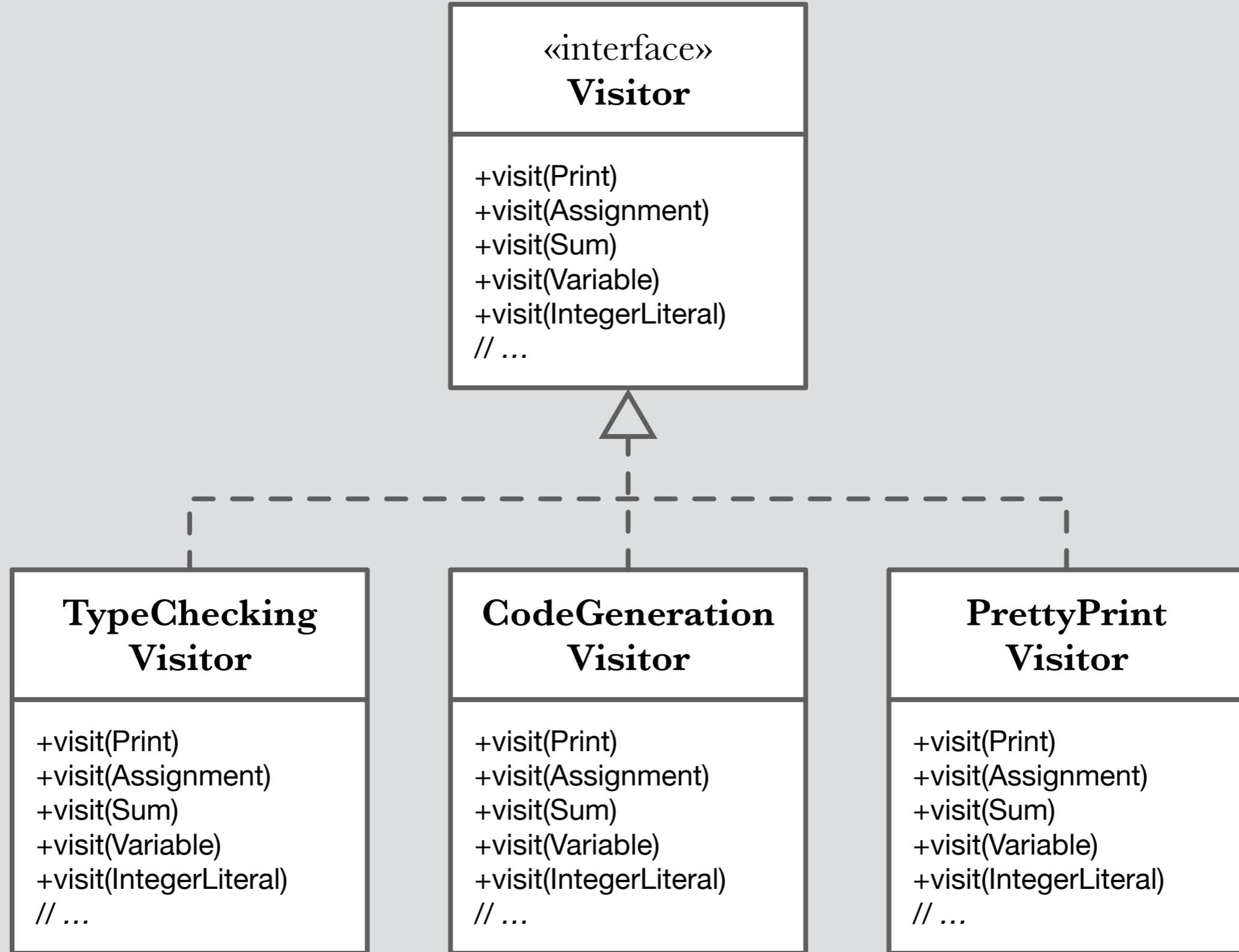
La implementación típica del patrón Visitor se basa en esta misma técnica.

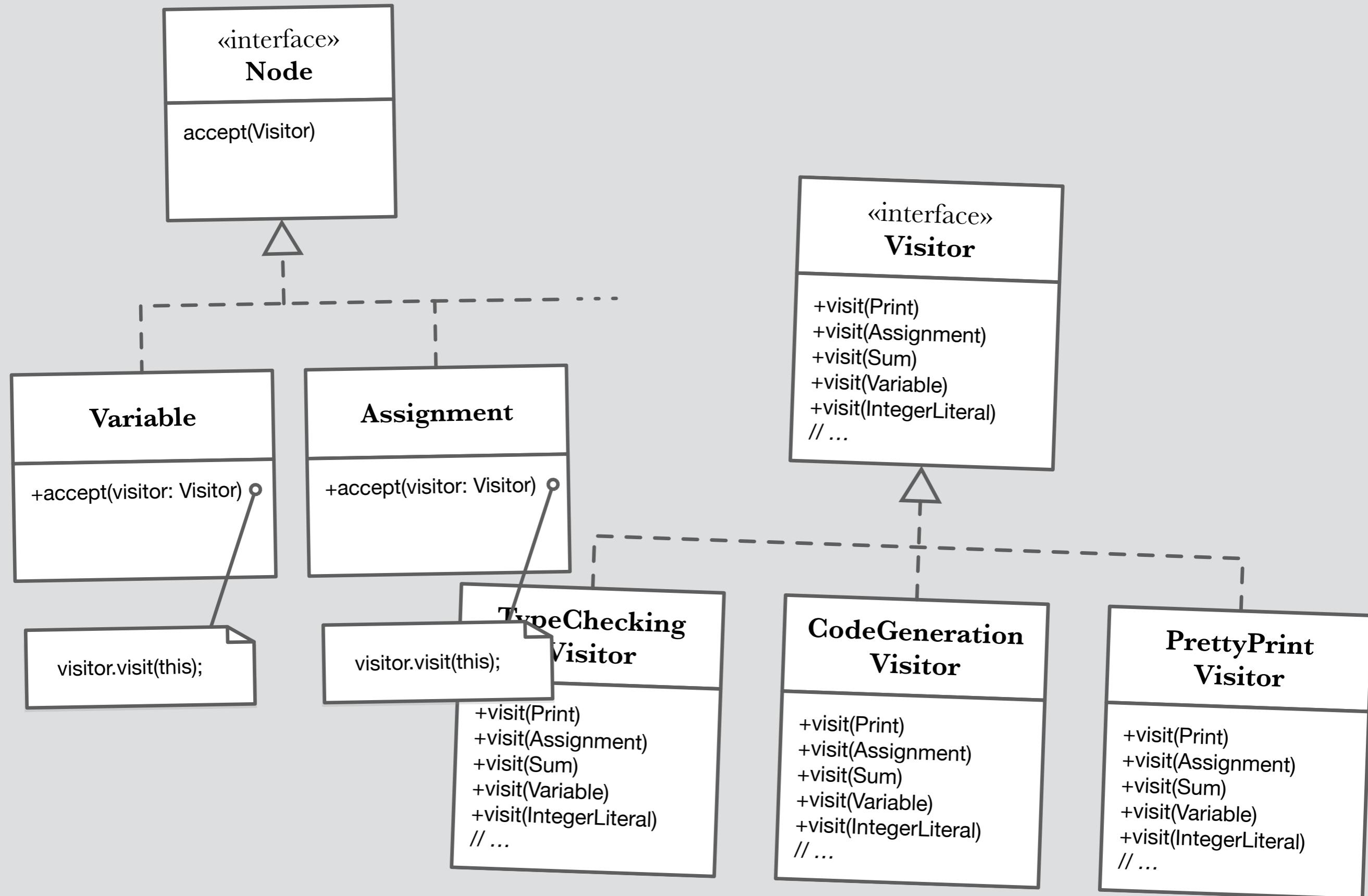
Volvamos a nuestro ejemplo inicial.











Aplicabilidad

Úsese el patrón Visitor cuando

Tenemos una estructura de objetos de distintos tipos y queremos realizar operaciones sobre esos elementos que dependen de su clase concreta.

Se quiere realizar muchas operaciones distintas y no relacionadas sobre objetos de una estructura de objetos.

Y queremos evitar «contaminar» sus clases con dichas operaciones.

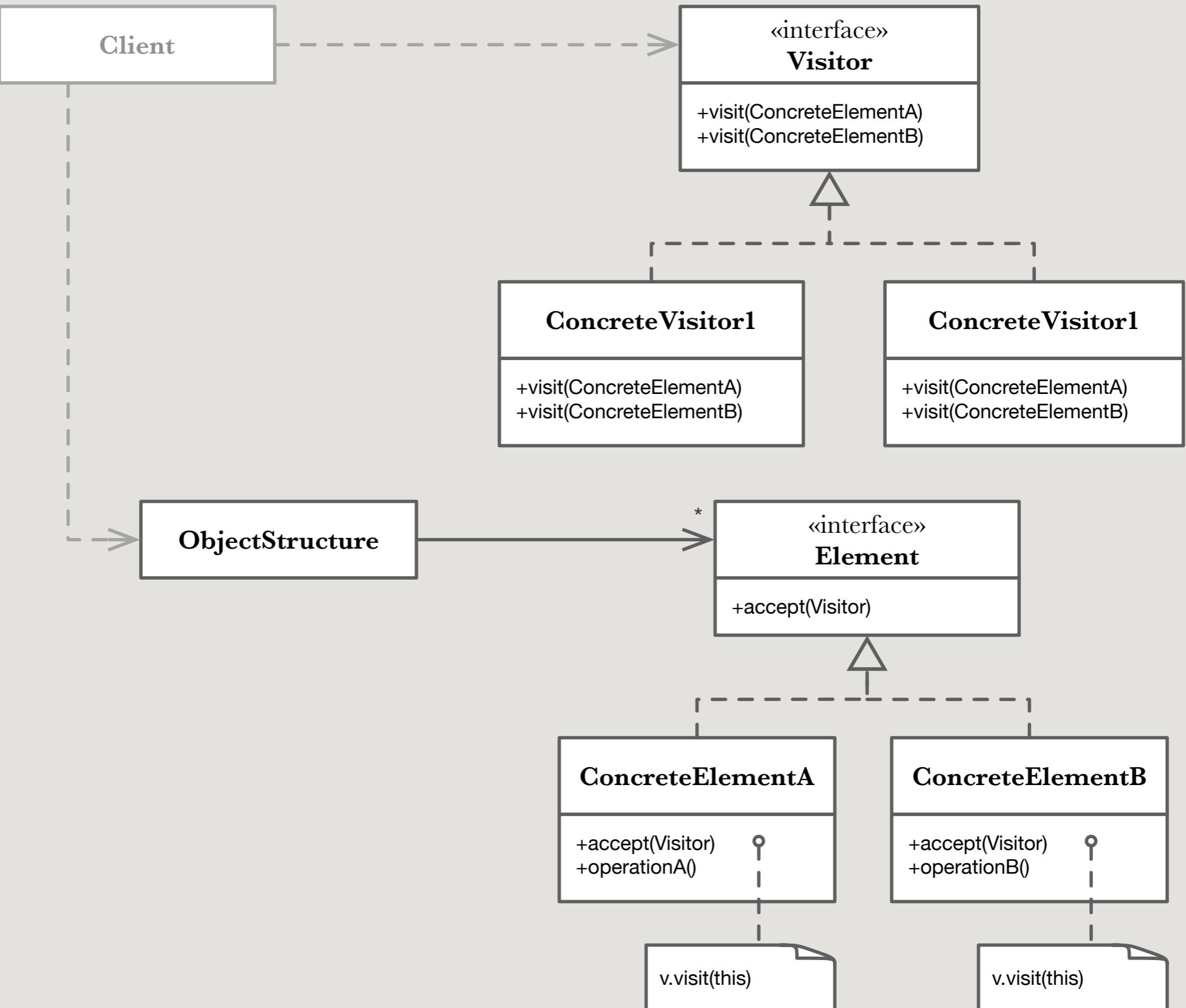
El patrón Visitor permite mantener juntas las operaciones relacionadas definiéndolas en una clase.

Las clases que definen la estructura de objetos rara vez cambian, pero queremos poder definir nuevas operaciones sobre dicha estructura.

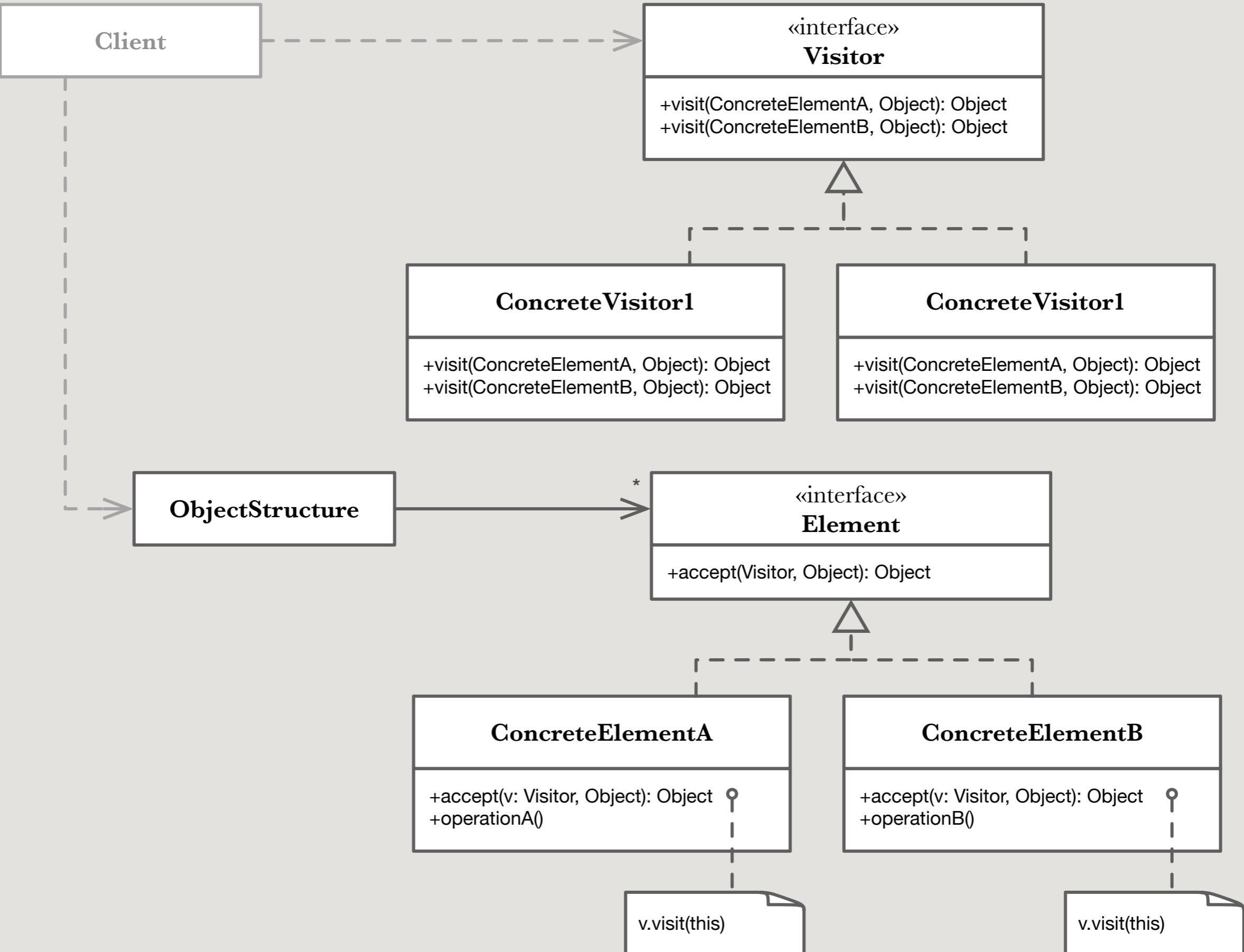
Cambiar las clases de la estructura de objetos requiere redefinir la interfaz para todos los visitantes, lo que es costoso.

Si las clases de la estructura cambian con frecuencia, probablemente sea mejor definir las operaciones en las propias clases.

Estructura



La estructura del patrón Visitor, tal como aparece en el GoF



La versión genérica del patrón Visitor

Participantes

Visitor

(Visitor)

Declara una operación visit para cada clase de elemento concreto de la estructura de objetos.

El visitante puede acceder directamente al elemento a través de su interfaz particular.

ConcreteVisitor

(TypeCheckingVisitor, PrettyPrintVisitor)

Implementa cada operación definida por la interfaz Visitor.

Cada operación implementa un fragmento del algoritmo para cada clase correspondiente de la estructura.

ConcreteVisitor

(TypeCheckingVisitor, PrettyPrintVisitor)

**Proporciona el contexto para el algoritmo y
almacena el estado necesario.**

Dicho estado suele usarse para acumular el resultado
durante el recorrido de la estructura.

Element

(Node)

Define una operación accept que recibe un Visitor como parámetro.

ConcreteElement

(Assignment, Variable)

Implementa la operación accept que recibe un Visitor como parámetro.

Simplemente delega en el método visit del visitante pasándose a sí mismo como parámetro.

ObjectStructure

(Program)

Permite enumerar sus elementos.

Colaboraciones

Un cliente que usa el patrón Visitor debe crear un objeto ConcreteVisitor y a continuación recorrer la estructura, visitando cada objeto con el visitante.

**Cada vez que se visita a un elemento,
éste llama a la operación del Visitor que
se corresponde con su clase.**

El elemento se pasa a sí mismo como parámetro
de la operación para permitir al visitante acceder
a su estado, en caso de que sea necesario.

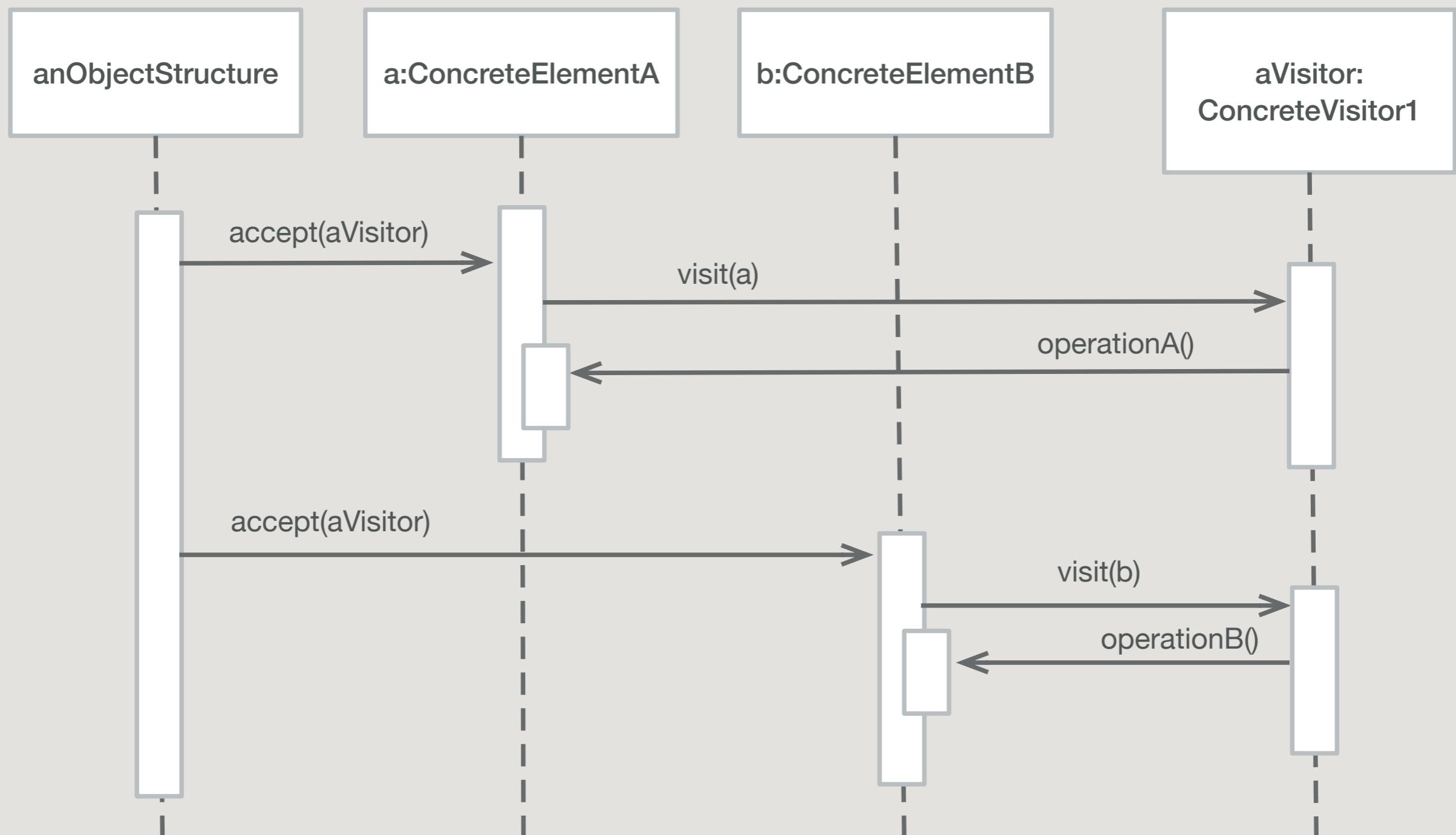
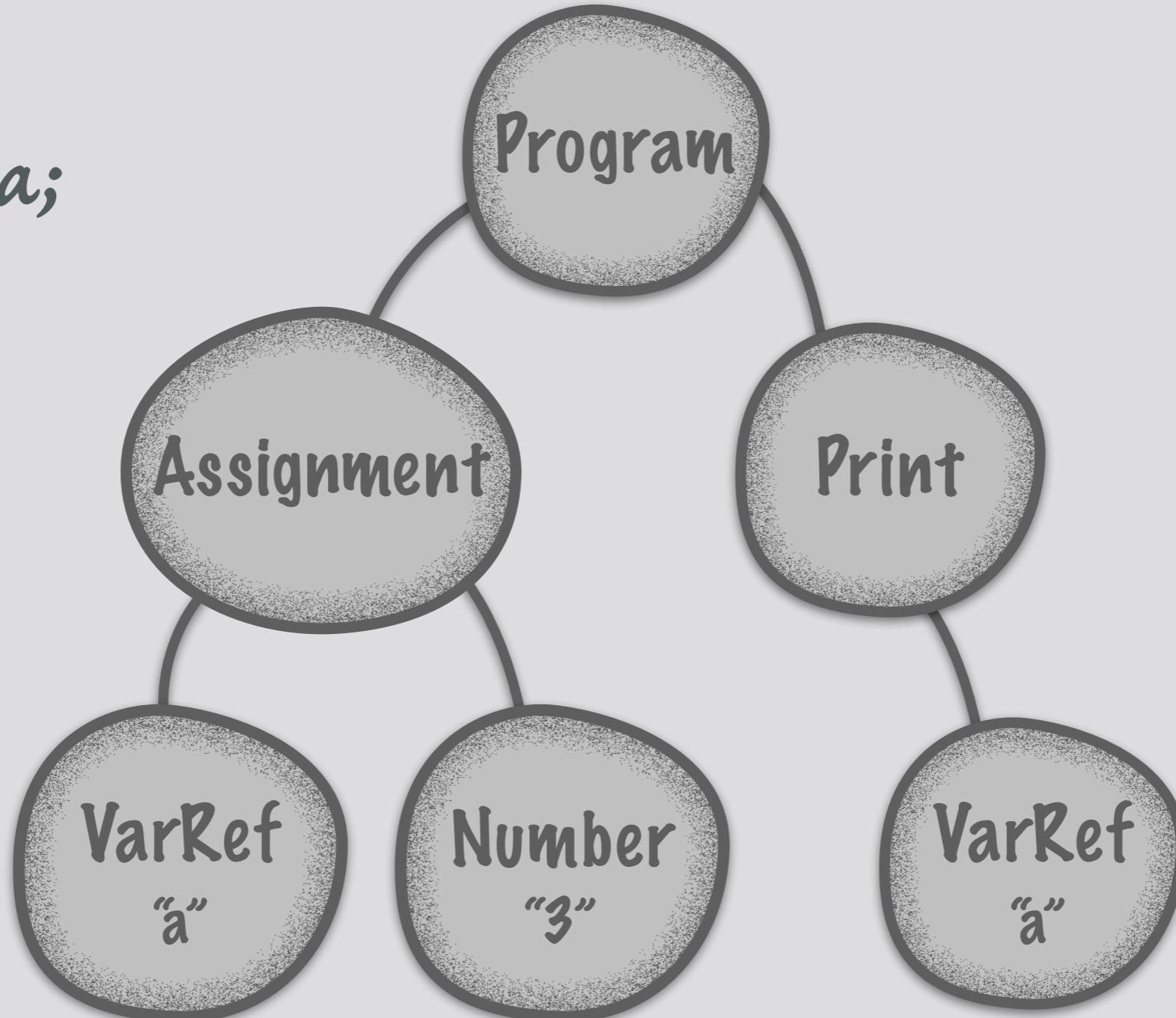
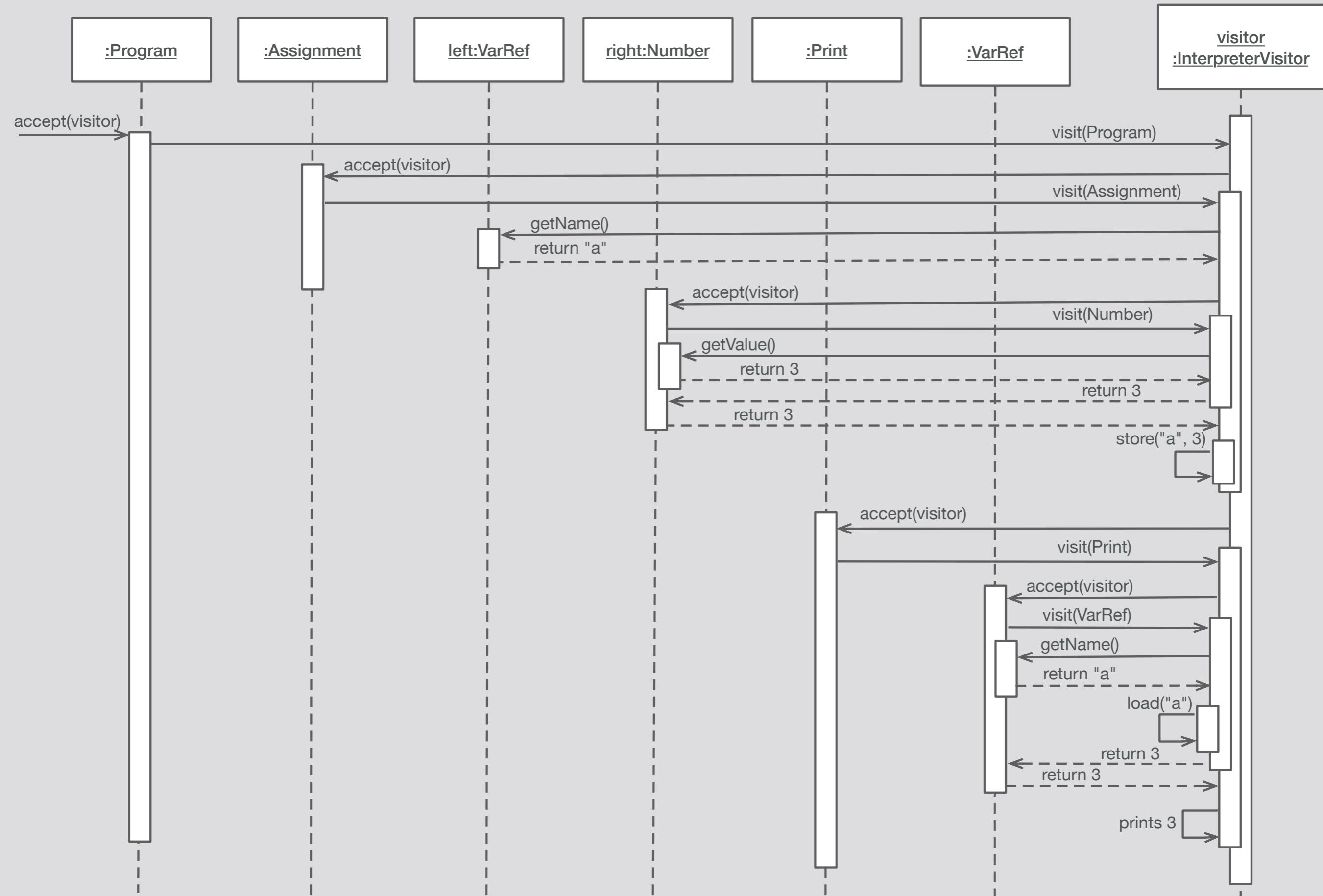


Diagrama de secuencia que ilustra las colaboraciones típicas en tiempo de ejecución del patrón Visitor entre una estructura de objetos, un visitante y dos elementos.

Veamos un caso concreto más elaborado.

*a = 3;
print a;*





Consecuencias

Facilita añadir nuevas operaciones.

Podemos añadir operaciones que dependen de los componentes de objetos complejos.

Simplemente creando una nueva clase de visitante.

Si, por el contrario, dicha funcionalidad estuviese repartida en muchas clases, habría que cambiar cada clase para definir una nueva operación.

Agrupa operaciones relacionadas y separa las que no lo están.

El comportamiento similar no está desperdigado por las clases que definen la estructura de objetos; está localizado en un visitante.

Las partes de comportamiento no relacionadas se dividen en sus propias clases de visitantes.

Esto simplifica tanto las clases que definen los elementos como los algoritmos definidos por los visitantes.

Es difícil añadir nuevas clases de elementos concretos.

Cada `ConcreteElement` nuevo da lugar a una nueva operación abstracta del `Visitor` y a su correspondiente implementación en cada clase `ConcreteVisitor`.

A veces se puede proporcionar una implementación del `Visitor` predeterminada que puede ser heredada por la mayoría de los visitantes concretos, pero esto representa una excepción más que una regla.

Por tanto la cuestión fundamental a considerar a la hora de aplicar el patrón Visitor es si es más probable que cambie el algoritmo aplicado sobre una estructura de objetos o las clases de los objetos que componen la estructura.

La jerarquía de clases Visitor puede ser difícil de mantener cuando se añaden con frecuencia nuevas clases de elementos concretos. En tales casos, probablemente sea más fácil definir operaciones sobre las clases que componen la estructura.

Lo anterior no implica necesariamente que el Visitante nunca sea aplicable si en un futuro pueden aparecer ocasionalmente nuevos elementos.

De hecho, una de las ventajas del patrón es la seguridad de tipos que proporciona en caso de que eso se produzca.

La interfaz Visitante garantiza que no pasemos por alto el procesamiento de ningún nodo de la estructura (como podría ocurrir con el típico recorrido recursivo).

Lo que nos está diciendo simplemente es que debemos preguntarnos cuál es la causa más probable de cambio en nuestro diseño: si nuevos elementos o nuevas operaciones. El patrón Visitor hace que nuestro diseño esté cerrado frente a lo segundo.

Visitar distintas jerarquías de clases.

Puede visitar objetos que no tienen una clase o interfaz padre común.

Podemos añadir cualquier tipo de objeto a una interfaz de Visitante, no tienen por qué ser de la misma jerarquía.

Esto es algo que no se puede conseguir, por ejemplo, simplemente iterando sobre la estructura de objetos.

Implementación

Despacho doble

El patrón Visitor permite añadir operaciones a las clases sin cambiar estas.

Lo consigue mediante una técnica denominada despacho doble.

Es una técnica muy conocida. De hecho, algunos lenguajes de programación la permiten directamente (CLOS, por ejemplo).

Esto no significa que el patrón Visitor no sea aplicable en dichos lenguajes. Lo único que cambia es que se facilita mucho su implementación (no es necesario el método `accept`).



Double Dispatch

Say you have three kinds of figures that you want to print on four kinds of printers. The message

```
circleFigure printOn: laserPrinter
```

must dispatch on both the Figure and Printer hierarchies. One does this by defining methods like

```
printOn: aPrinter
    aPrinter printCircle: self
```

in the Figure hierarchy and methods like

```
printCircle: aCircle
    ... circle printing stuff ...
```

in the Printer hierarchy.

An elaboration of the above example in Java is at [DoubleDispatchExample](#).

Isn't this what the [GangOfFour VisitorPattern](#) does? And [CommonLisp](#) has built into the language via its [GenericFunctions](#)?

[DanIngalls](#) wrote this technique up for one of the early OOPSLAs. The program committee almost rejected the paper because the idea was considered obvious, or insufficiently scientific, or something like that.

It was one of my favorite OOPSLA papers, and it was because OOPSLA doesn't publish papers like that any more that I wanted to start a conference on patterns, where an idea will not be rejected just because it was in [IvanSutherland](#)'s PhD thesis.

-- [RalphJohnson](#)



Visitor Pattern Versus Multimethods

The Visitor Pattern

The visitor pattern is a programming pattern that has been advocated strongly for writing code operating on a hierarchy of classes. [A thorough description is available there](#) from the *Design Patterns* book.

A typical example is the definition of operations on an Abstract Syntax Tree. Here is Java code using a Visitor:

```
package syntax;

abstract class ExpressionVisitor
{
    abstract void visitIntExp(IntExp e);
    abstract void visitAddExp(AddExp e);
}

abstract class Expression
{
    abstract void accept(ExpressionVisitor v);
}

class IntExp extends Expression
{
    int value;

    void accept(ExpressionVisitor v)
    {
        v.visitIntExp(this);
    }
}

class AddExp extends Expression
{
    Expression e1, e2;

    void accept(ExpressionVisitor v)
    {
        v.visitAddExp(this);
    }
}
```

The interest of this construction is that it is now possible to define operations on expressions by subclassing ExpressionVisitor. This can even be done in a different package, without modifying the expression hierarchy classes.

```
// Behaviour can now be defined on Expressions

package tools;

class PrettyPrint extends ExpressionVisitor
{
    void visitIntExp(IntExp e)
    {
        System.out.print(e.value);
    }
}
```

Patrones relacionados

Composite

El Visitor se puede utilizar para aplicar una operación sobre una estructura de objetos definida por el patrón Composite.

Interpreter

Lo que hicimos en la práctica de la máquina virtual.

El visitor es otro enfoque para ejecutar un programa.