2

(XI)

Visitor

(Patrones de diseño)

Diseño del Software

Grado en Ingeniería Informática del Software

Curso 2017-2018

Escuela de Ingeniería Informática
Universidad de Oviedo

César Acebal

Visitor
(Visitante)

- **Patrón de comportamiento (ámbito de objetos)**
- **Propósito:**

> *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 mediante una estructura de árbol**
  - Árboles sintácticos abstractos (AST, «abstract syntax trees»)
- **Necesitará realizar operaciones como:**
  - Análisis sintáctico
  - Análisis semántico
  - Generación de código
  - etcétera

También podríamos definir otras operaciones sobre el árbol, como depuración, o resaltar el código en el editor.

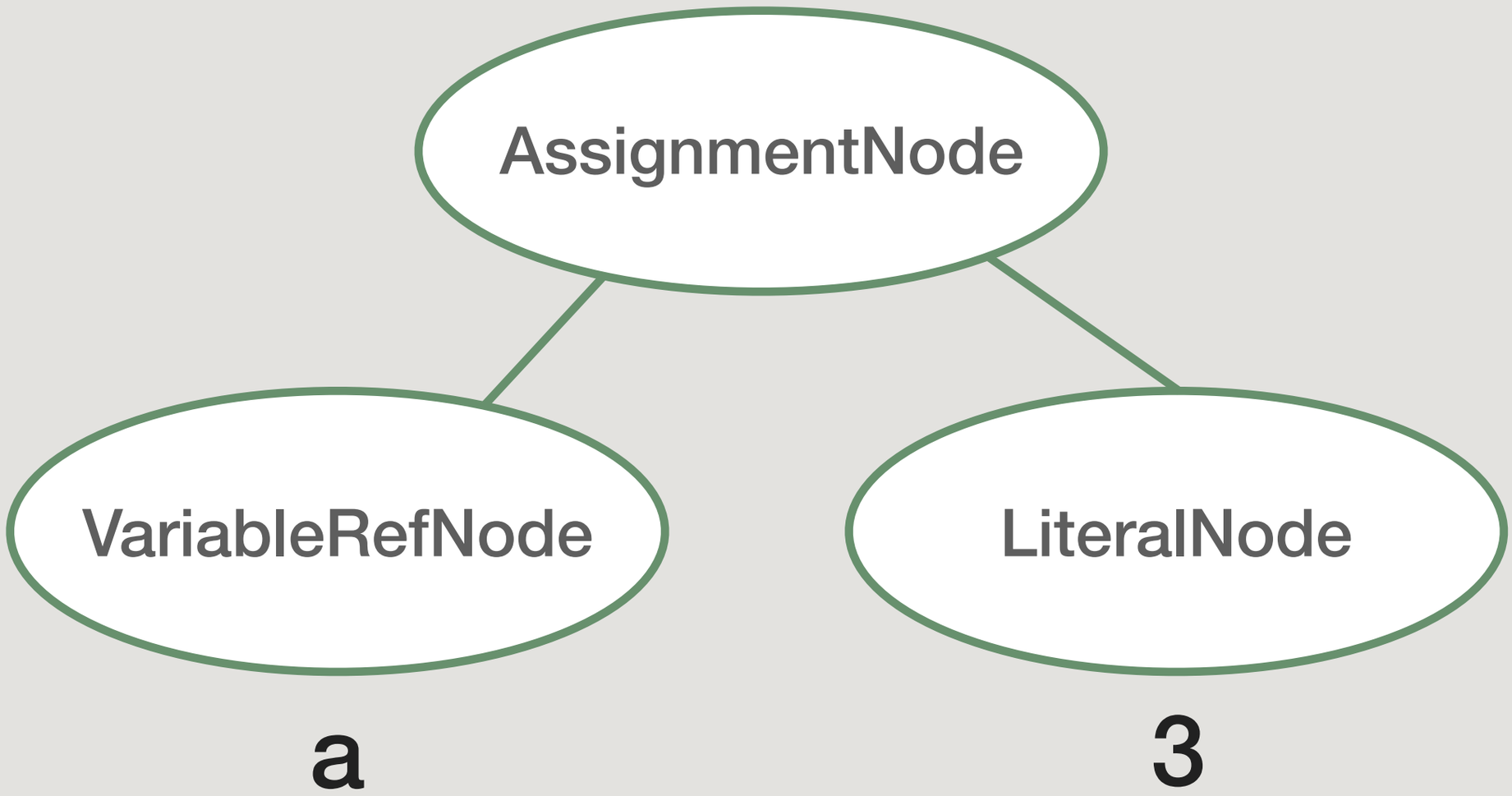
Motivación

- **Normalmente tendremos clases distintas para las distintas construcciones del lenguaje**
  - Referencias a variables
  - Sentencias de asignación
  - ...
- **Serán los nodos del árbol**

  Los tipos de nodos dependen del lenguaje que está siendo analizado, pero normalmente serán bastante estables para un determinado lenguaje (mientras éste no cambie, no cambiarán).

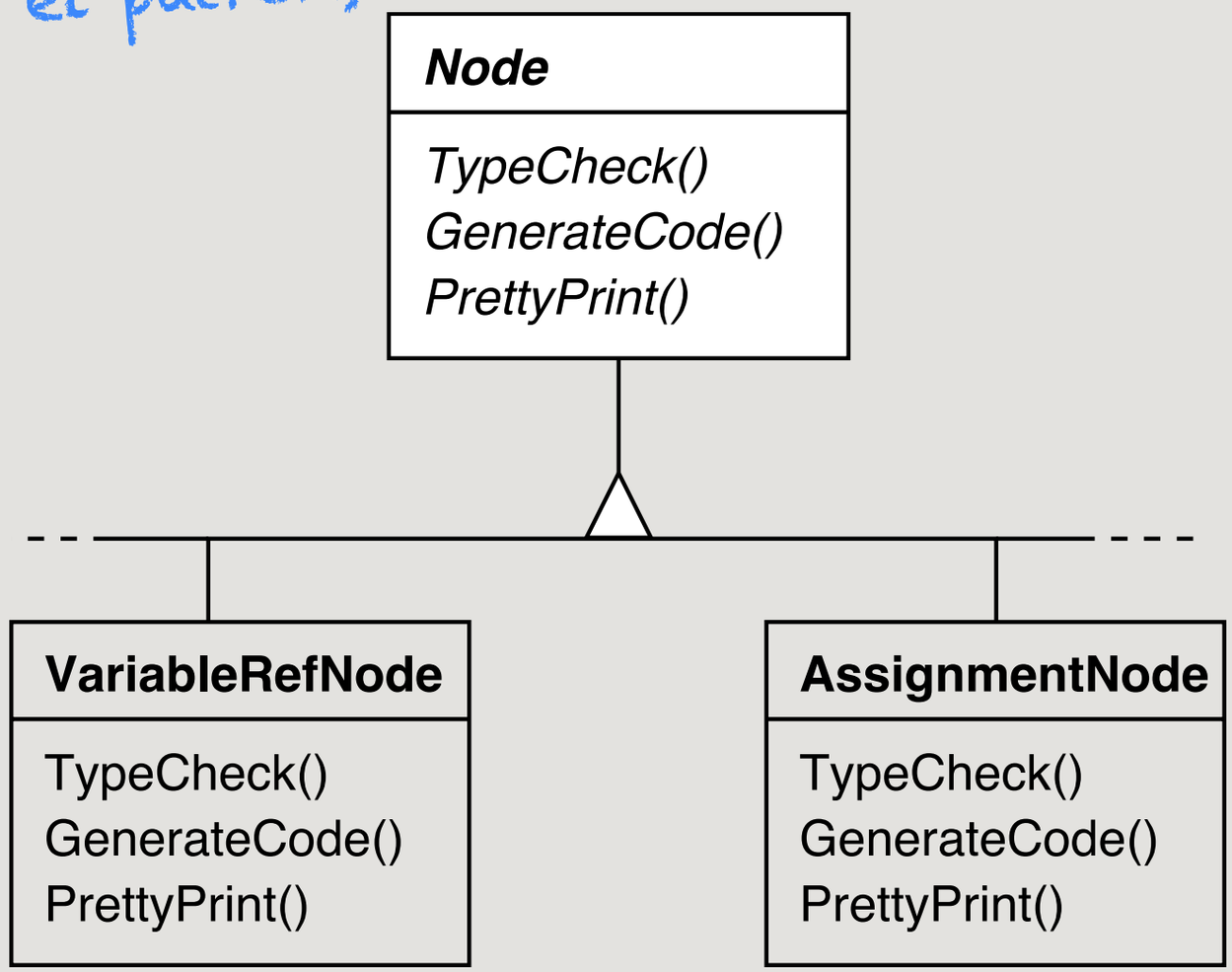
Ejemplo

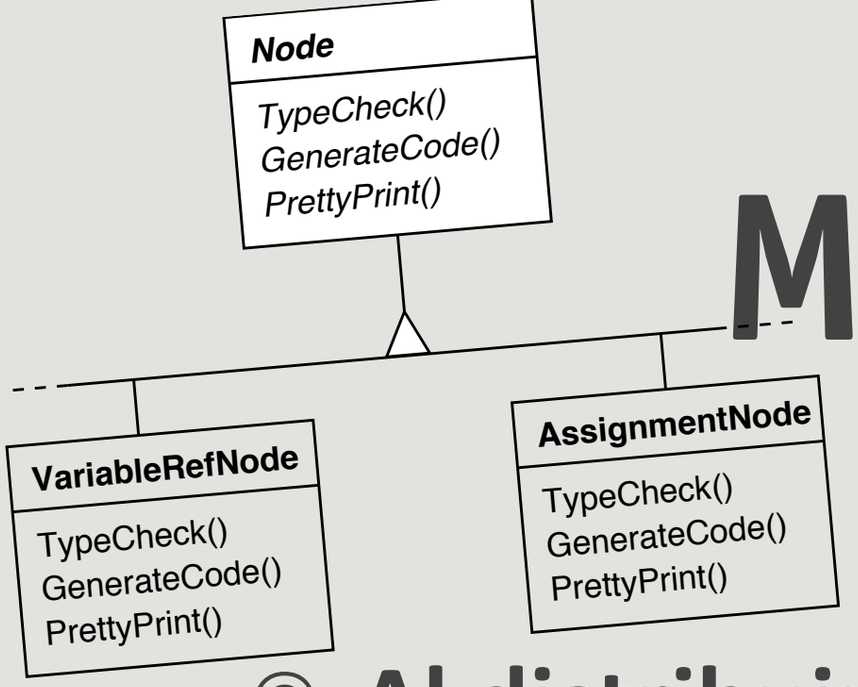
```
a = 3;
```



Motivación

Una posibilidad
(antes de aplicar el patrón)





Motivación

Problema

- **Al distribuir esas operaciones sobre todas las clases tenemos un sistema que es más difícil de comprender, mantener y cambiar**
  - Estamos mezclando responsabilidades que nada, o muy poco, tienen que ver entre sí

    clases muy poco cohesivas

¿Qué habría que hacer para añadir una nueva operación (y eso sí es probable que ocurra)?

Motivación

- **¿No sería mejor, en este caso, tener toda la lógica de análisis sintáctico, semántico, resaltado de código, etcétera, agrupada en sus propias clases?**
  - Y que las clases de los nodos del árbol fuesen independientes de las operaciones que se les aplican

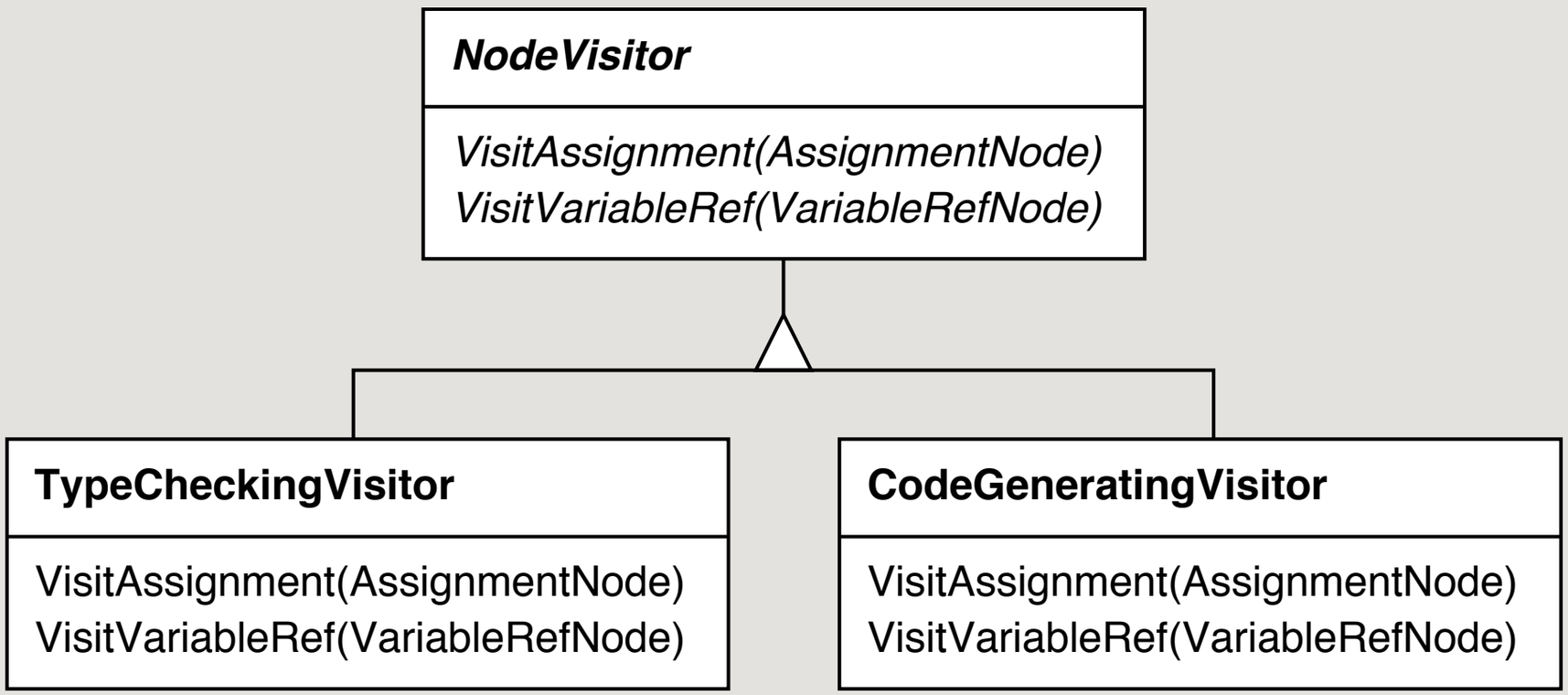
Ahora tenemos que resolver cómo «recorrer» y procesar esa estructura, llamando al método apropiado para cada tipo de nodo

Motivación

- **Encapsulamos cada tipo de operación en una clase «visitor» y se la pasamos al árbol**
  - Los nodos definirán una operación para «aceptar» visitantes
  - Y llamarán a su vez a la operación apropiada del visitante pasándose a sí mismos como parámetros

Motivación

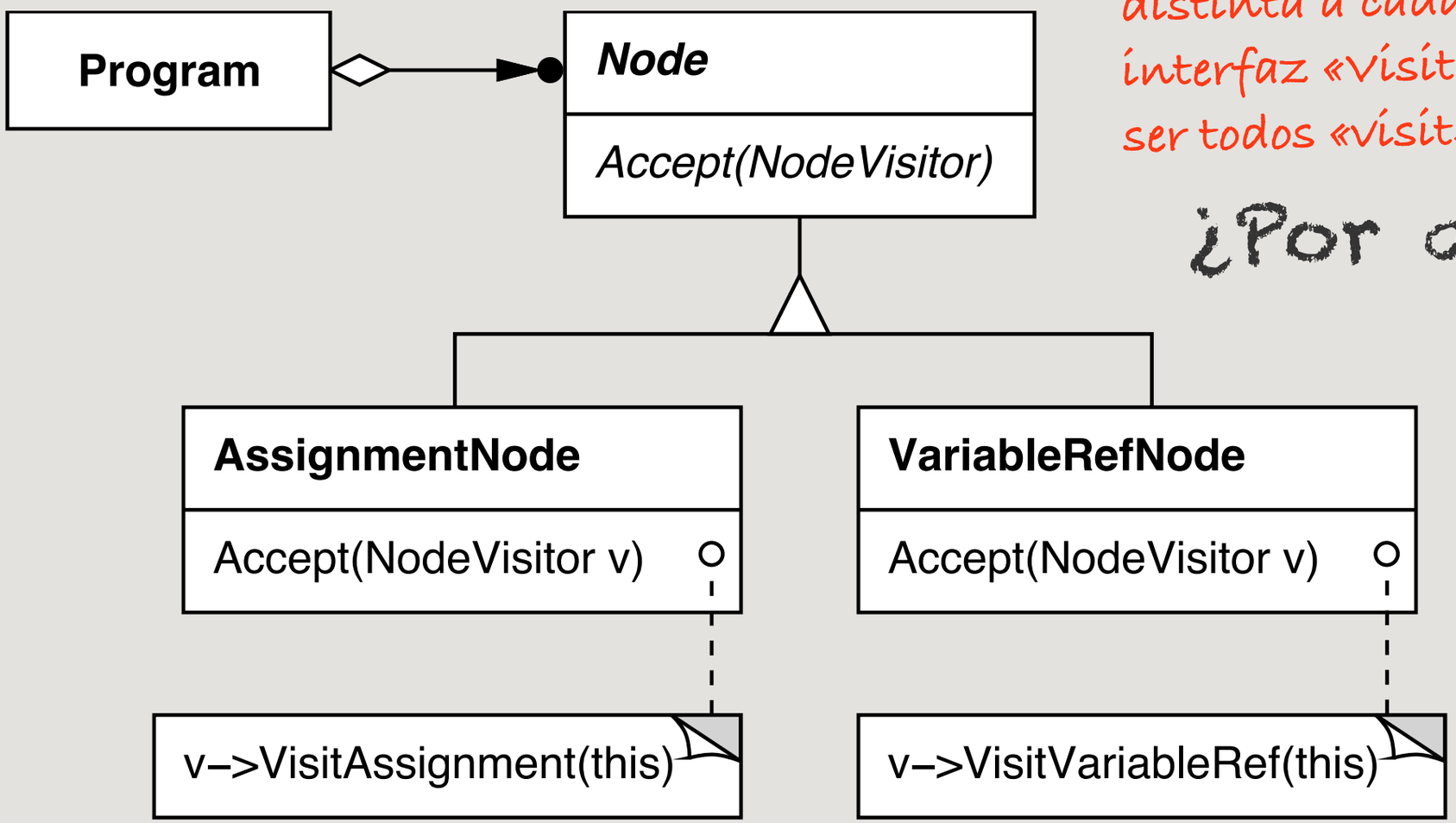
Los «visitantes»



Cada clase «visitor» (una para cada tipo de operación distinta a realizar sobre el árbol: sintáctico, semántico, etc.) debe definir una operación para tratar cada tipo de nodo: sentencia de asignación, if, referencia a variable, valor literal, comentario...

Motivación

El árbol



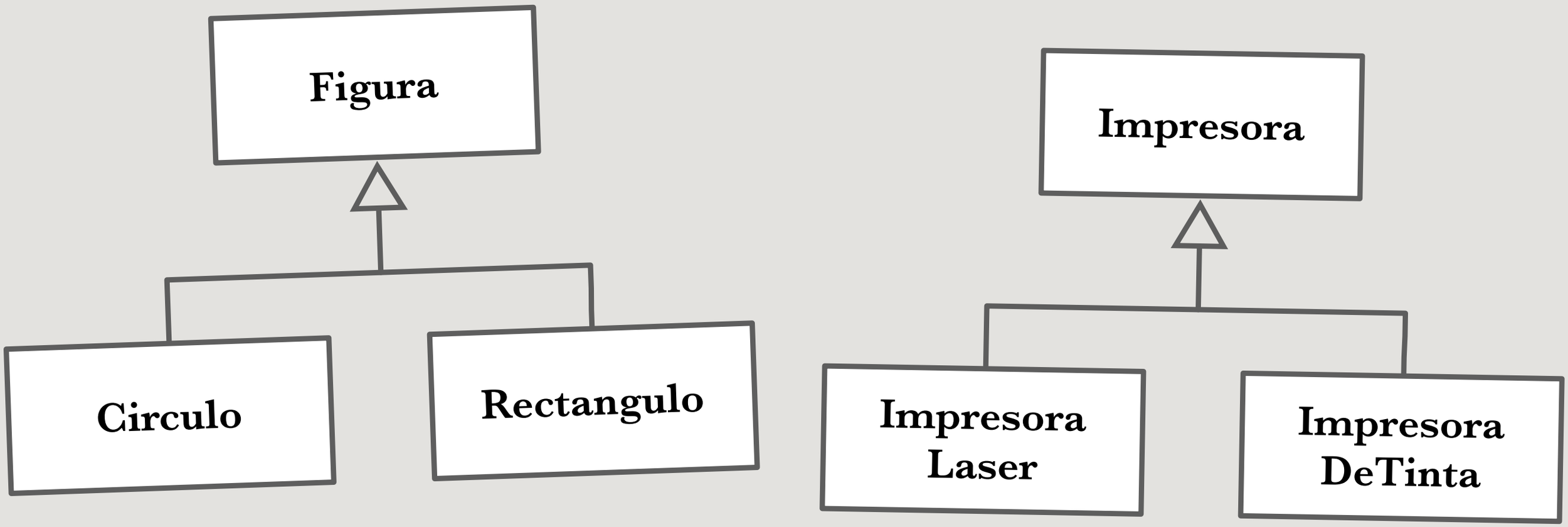
Realmente, al hacerlo así no haría falta llamar de forma distinta a cada método de la interfaz «visitor»: podrían ser todos «visit», a secas.

¿Por qué?

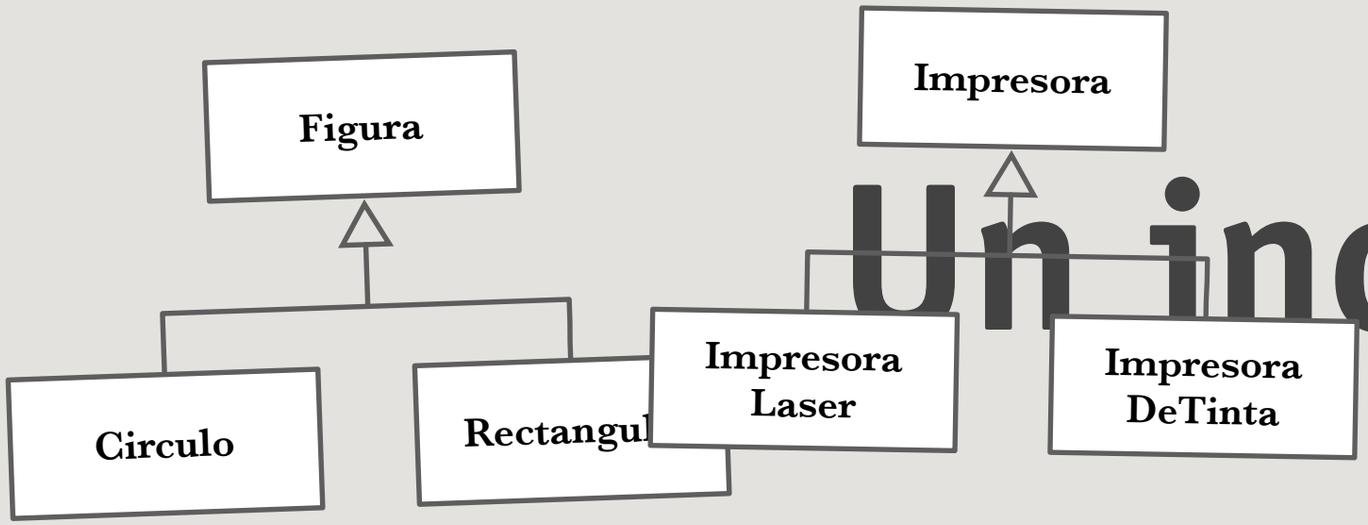
Con la operación «accept», que recibe un «visitor» como parámetro, definida en cada tipo de nodo

Un inciso

¿Nos acordamos de esto?



¿Qué ocurría?



Un inciso

¿Qué ocurría?

- Cada impresora tenía un método para imprimir cada tipo de figura

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

Esto no funcionaba

```
for (Figura figura : figuras) {
    for (Impresora impresora : impresoras) {
        impresora.imprimir(figura);
    }
}
```

```
class ImpresoraLaser
{
  ...
  imprimir(Rectangulo rectangulo) {...}
  imprimir(Circulo circulo) {...}
}
```

```
class ImpresoraDeTinta
{
  ...
  imprimir(Rectangulo rectangulo) {...}
  imprimir(Circulo circulo) {...}
}
```

```
        for (Figura figura : figuras) {
            for (Impresora impresora : impresoras) {
                impresora.imprimir(figura);
            }
        }
    }
```

Problems | Javadoc | Declaration

1 error, 0 warnings, 0 others

Description	Resource	Path	Location	Type
Errors (1 item)				
The method imprimir(Rectangulo) in the type Impresora is not applicable for the arguments (Figura)				Java Problem

Tampoco si añadíamos un método para Figura

```java
public class ImpresoraLaser extends Impresora
{
    @Override
    public void imprimir(Figura figura)
    {
        System.out.println("Imprimiendo una figura en la impresora láser");
    }

    @Override
    public void imprimir(Rectangulo rectangulo)
    {
```

Problems | @ Javadoc | Declaration

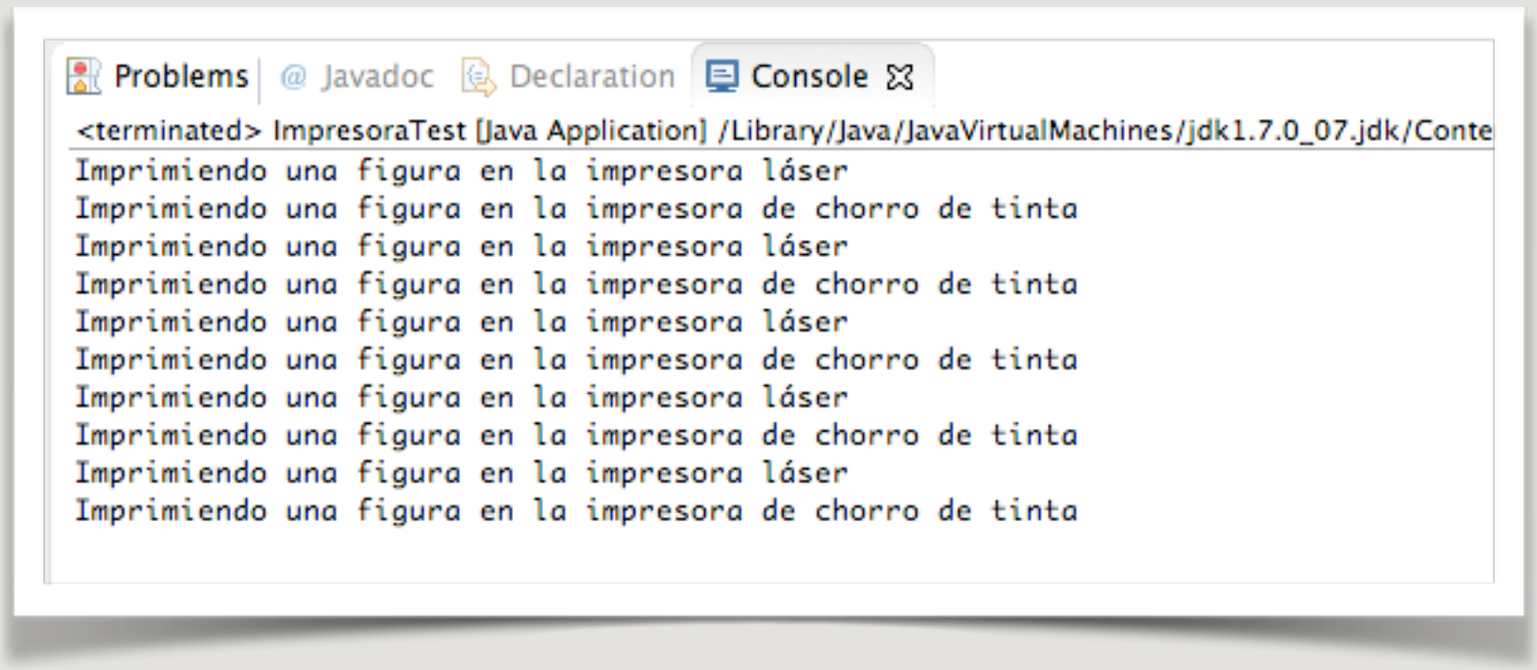
0 items

Description	Resource	Path	Location	Type

Ahora compila sin errores

Pero...

- **No funciona como esperábamos**
  - Siempre se ejecuta el nuevo método
    - El que recibe una figura (`Figura`), independientemente de cuál sea su tipo real en tiempo de ejecución



¿Qué ocurría?

- **Pues que en Java, C++ y muchos otros lenguajes de programación, el polimorfismo no se aplica a los parámetros, sino sólo al receptor del mensaje**
- **Es una decisión de diseño de estos lenguajes**

Los parámetros son ciudadanos de segunda clase, comparados con el receptor del mensaje (lo que aparece a la izquierda del punto en la llamda a un método). Para ellos sólo se tiene en cuenta el tipo estático de la referencia, no el tipo en tiempo de ejecución.

Una posible solución

- Que el cliente compruebe el tipo de cada figura en tiempo de ejecución:

Antes

```
public static void main(String[] args)
{
   ...
   for (Figura figura : figuras) {
     for (Impresora impresora : impresoras) {
       impresora.imprimir(figura);
     }
   }
}
```

Una posible solución

- Que el cliente compruebe el tipo de cada figura en tiempo de ejecución:

Antes

```
public static void main(String[] args)
{
   ...
   for (Figura figura : figuras) {
     for (Impresora impresora : impresoras) {
       impresora.imprimir(figura);
     }
   }
}
```

Una posible solución

- **Que el cliente compruebe el tipo de cada figura en tiempo de ejecución:**

Después

```
public static void main(String[] args)
{
  ...
  for (Figura figura : figuras) {
    for (Impresora impresora : impresoras) {
      if (figura instanceof Rectangulo)
        impresora.imprimir((Rectangulo) figura);
      else if (figura instanceof Circulo)
        impresora.imprimir((Circulo) figura);
      else if ...
    }
  }
}
```

Otra

- Que lo haga la impresora:

```
public abstract class Impresora
{
    public void imprimir(Figura figura)
    {
        if (figura instanceof Rectangulo) {
            imprimir((Rectangulo) figura);
        } else if (figura instanceof Circulo) {
            imprimir((Circulo) figura);
        } else {
            assert false : "Tipo de figura no implementado";
        }
    }

    public abstract void imprimir(Rectangulo rectangulo);
    public abstract void imprimir(Circulo circulo);
}
```

Un poco mejor

- **Esta última es algo mejor que la anterior**
  - La lógica condicional está encapsulada en una clase (en la impresora)
  - Si aparecen nuevos tipos de figuras, habrá que cambiarlo, pero ahora sólo en un sitio
    - No en todos los clientes de la clase, como antes

Otra solución

```
public interfaz Impresora
{
    void imprimir(Rectangulo rectangulo);
    void imprimir(Circulo circulo);
}
```

```
public interfaz Figura
{
    void imprimirEn(Impresora impresora);
}
```

Otra solución

```
public class Rectangulo implements Figura
{
    @Override
    public void imprimirEn(Impresora impresora)
    {
        impresora.imprimir(this);
    }
}
```

```
public class Circulo implements Figura
{
    @Override
    public void imprimirEn(Impresora impresora)
    {
        impresora.imprimir(this);
    }
}
```

```
public interfaz Impresora
{
    void imprimir(Rectangulo rectangulo);
    void imprimir(Circulo circulo);
}
```

```
public interfaz Figura
{
    void imprimirEn(Impresora impresora);
}
```

```
public class Rectangulo implements Figura
{
    @Override
    public void imprimirEn(Impresora impresora)
    {
        impresora.imprimir(this);
    }
}
```

```
public class Circulo implements Figura
{
    @Override
    public void imprimirEn(Impresora impresora)
    {
        impresora.imprimir(this);
    }
}
```

Antes

```
public static void main(String[] args)
{
   ...
   for (Figura figura : figuras) {
     for (Impresora impresora : impresoras) {
       impresora.imprimir(figura);
     }
   }
}
```

Antes

```
public static void main(String[] args)
{
   ...
   for (Figura figura : figuras) {
     for (Impresora impresora : impresoras) {
       impresora.imprimir(figura);
     }
   }
}
```

```
public static void main(String[] args)
{
   ...
   for (Figura figura : figuras) {
     for (Impresora impresora : impresoras) {
       figura.imprimirEn(impresora)
     }
   }
}
```

Despacho doble

- **Lo que hemos hecho no es más que una forma de simular el despacho doble**
  - Una simplificación del despacho múltiple o multimétodos

http://c2.com/cgi/wiki?DoubleDispatch

http://nice.sourceforge.net/visitor.html

La motivación del Visitor es más compleja, pero su implementación se basa exactamente en la misma técnica (en lenguajes que no admitan despacho múltiple de manera nativa).

Volvamos al patrón Visitor...

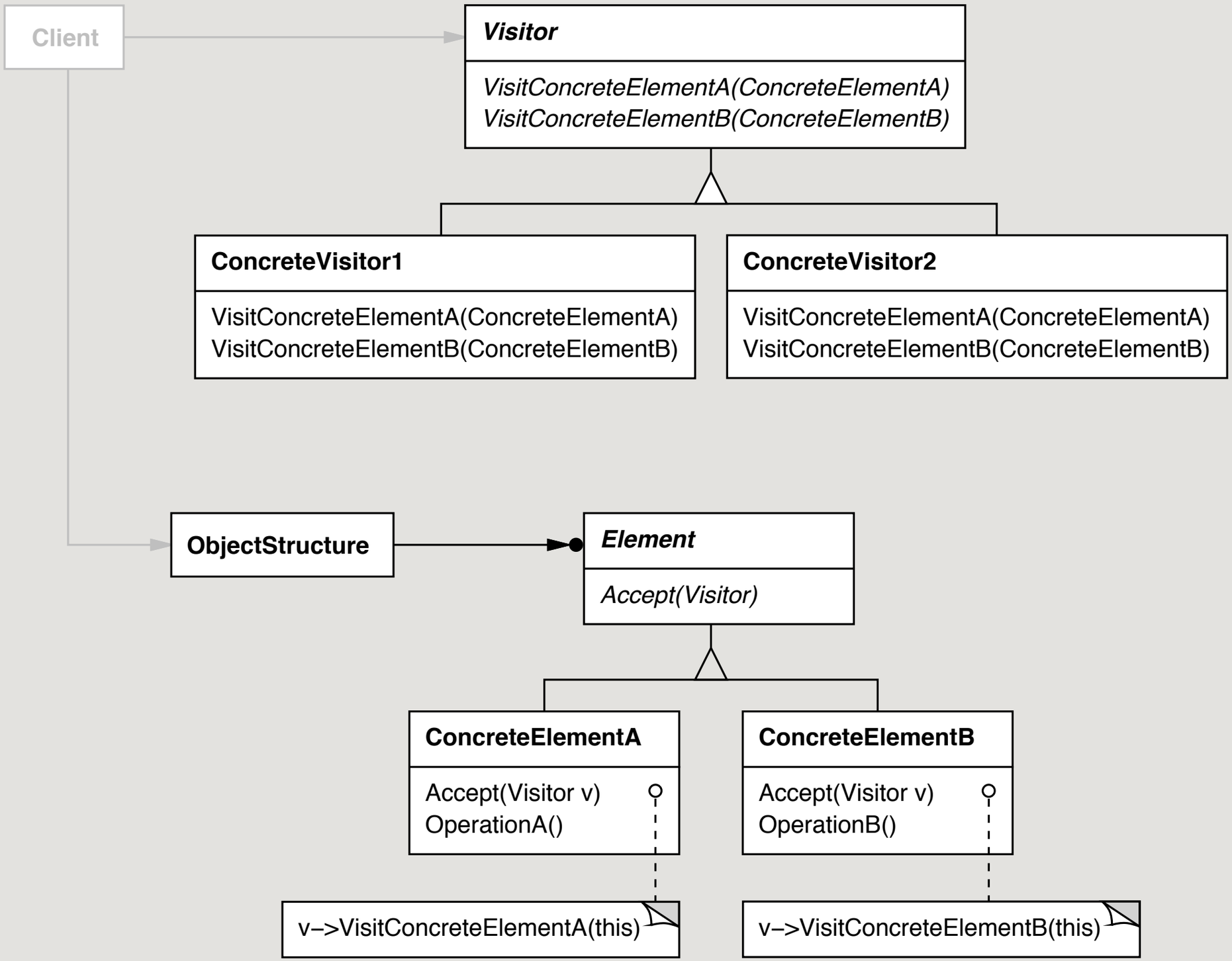
Aplicabilidad

- **Debería aplicarse el patrón Visitor cuando:**
  - Una estructura de objetos contiene muchas clases de objetos con diferentes interfaces, y queremos realizar operaciones sobre esos elementos que dependen de su clase concreta
  - Se necesitan 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 operaciones relacionadas definiéndolas en una clase

Aplicabilidad

- **Debería aplicarse el patrón Visitor cuando (cont.):**
  - Las clases que definen la estructura de objetos rara vez cambian, pero muchas veces queremos definir nuevas operaciones sobre la estructura
    - Cambiar las clases de la estructura de objetos requiere redefinir la interfaz para todos los visitantes, lo que es potencialmente costoso
    - Si las clases de la estructura cambian con frecuencia, probablemente sea mejor definir las operaciones en las propias clases

Estructura


Client
Visitor
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor1
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ConcreteVisitor2
VisitConcreteElementA(ConcreteElementA)
VisitConcreteElementB(ConcreteElementB)
ObjectStructure
Element
Accept(Visitor)
ConcreteElementA
Accept(Visitor v)
OperationA()
ConcreteElementB
Accept(Visitor v)
OperationB()
v->VisitConcreteElementA(this)
v->VisitConcreteElementB(this)

Participantes

- **Visitor (NodeVisitor)**
  - Declara una operación `visit` para cada clase de operación `ConcreteElement` de la estructura de objetos
  - El nombre y signatura de la operación identifican a la clase que envía la petición visit al visitante
    - Eso permite al visitante determinar la clase concreta de elemento que está siendo visitada
    - A continuación el visitante puede acceder al elemento directamente a través de su interfaz particular.
- **ConcreteVisitor (TypeCheckingVisitor)**
  - Implementa cada operación declarada por `Visitor`
  - Cada operación implementa un fragmento del algoritmo definido para la clase correspondiente de la estructura
  - `ConcreteVisitor` proporciona el contexto para el algoritmo y guarda su estado local
    - Muchas veces este estado acumula resultados durante el recorrido de la estructura

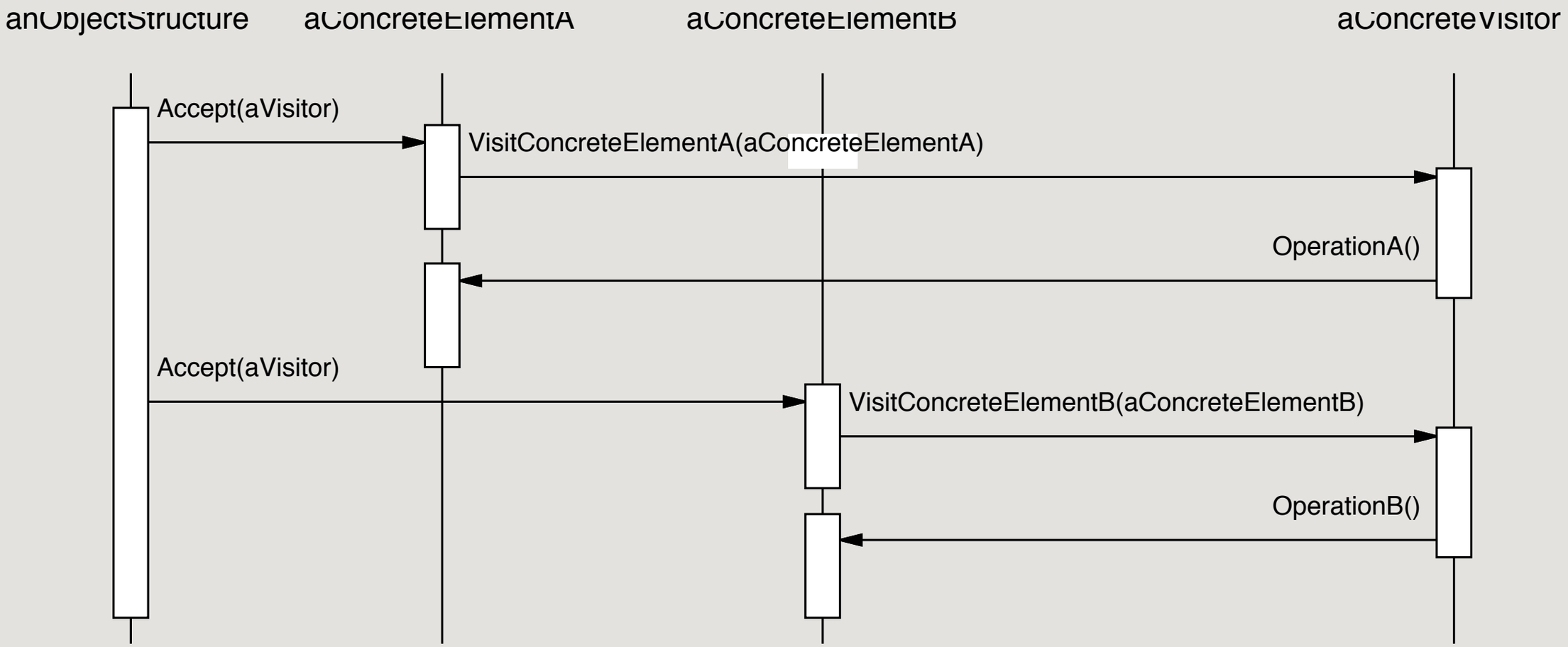
Participantes

- **Element (Node)**
  - Define una operación `accept` que recibe un visitante como argumento
- **ConcreteElement (AssignmentNode, VariableRefNode)**
  - Implementa una operación `accept` que recibe un visitante como argumento
- **ObjectStructure (Program)**
  - Puede enumerar sus elementos
  - Puede proporcionar una interfaz de alto nivel para permitir al visitante visitar a sus elementos
  - Puede ser un compuesto (patrón Composite) o una colección, como una lista o un conjunto

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 argumento de la operación para permitir al visitante acceder a su estado, en caso de que sea necesario

Colaboraciones



Consecuencias

- **El visitante facilita añadir nuevas operaciones**
  - Podemos definir una nueva operación sobre una estructura simplemente añadiendo un nuevo visitante
  - Si, por el contrario, extendiésemos la funcionalidad sobre muchas clases, habría que cambiar cada clase para definir una nueva operación

Consecuencias

- **Un visitante 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 subclases del visitante. Esto simplifica tanto las clases que definen los elementos como los algoritmos definidos por los visitantes

Consecuencias

- **Es difícil añadir nuevas clases de elementos concretos**
  - El patrón Visitor hace que sea complicado añadir nuevas subclases de elementos
    - 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 en Visitor una implementación 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