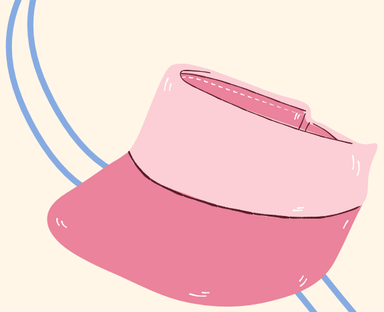


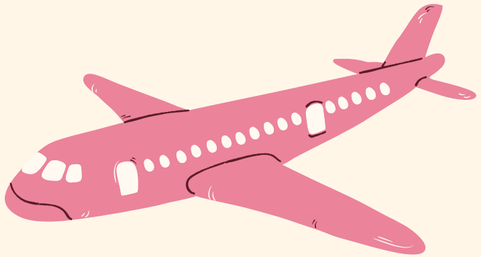
VISITOR

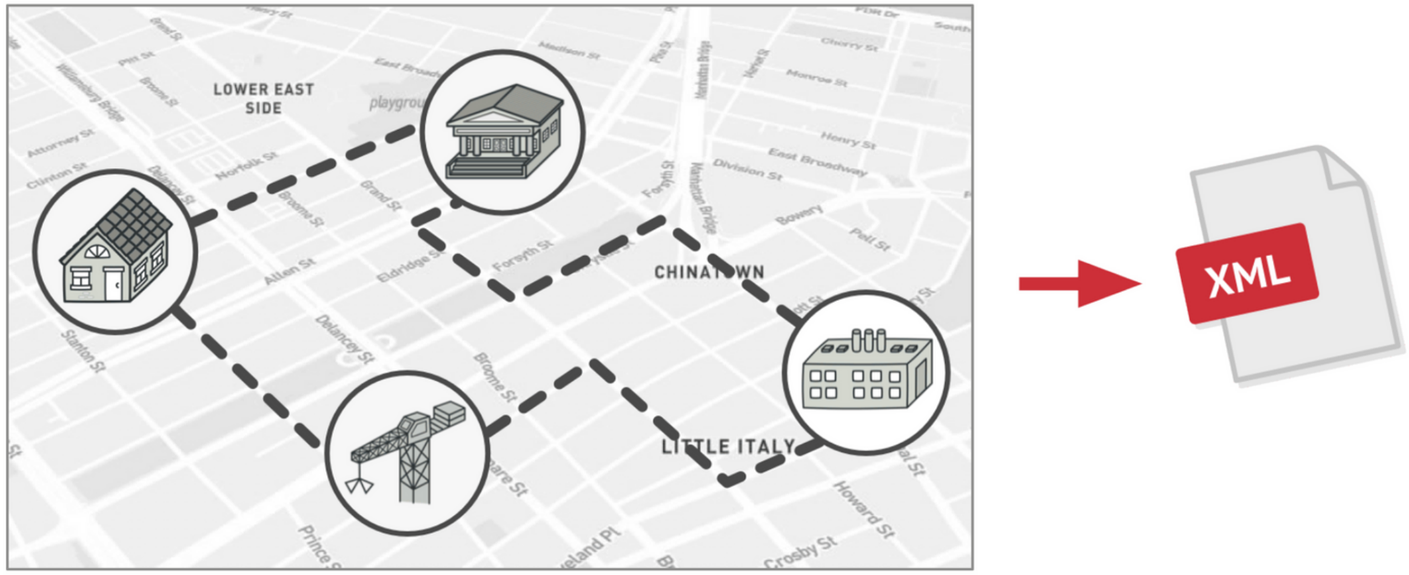
LUIS DIEGO BARRANTES
FRANCISCO MORA

CONTENIDO

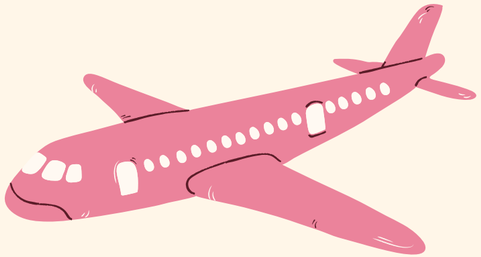
- Problema
- Solución
- Implementación
- Ejemplo en código
- Consecuencias
- Relaciones

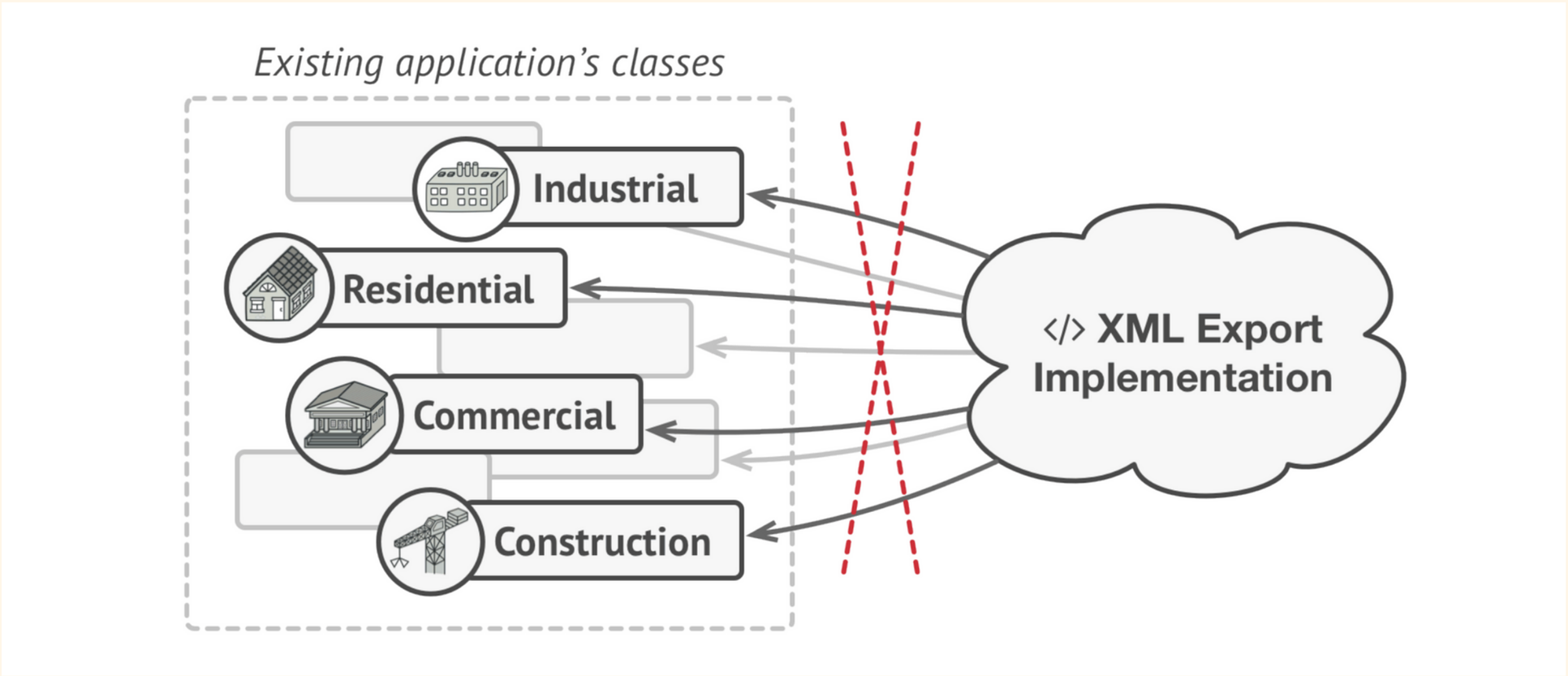
PROBLEMA



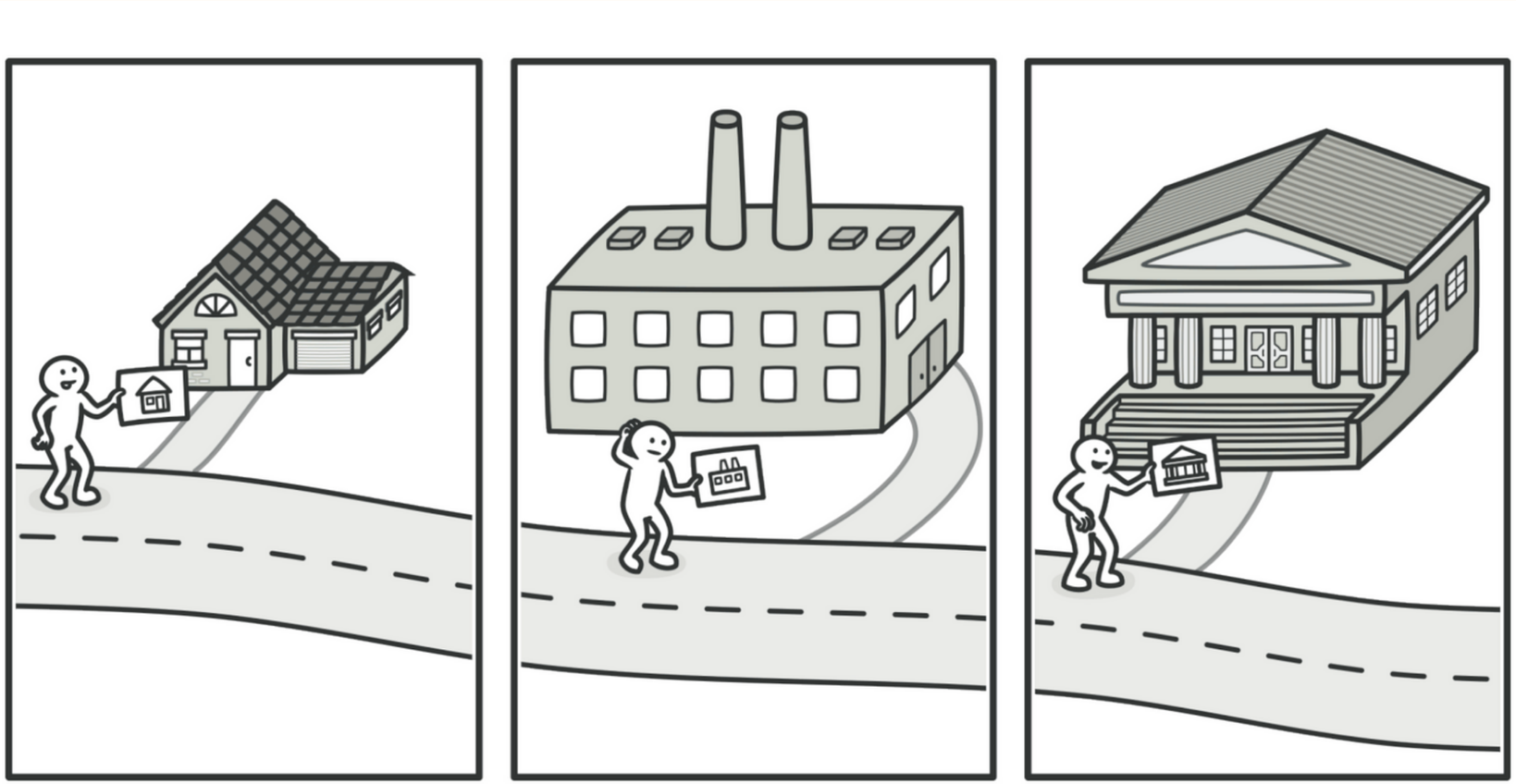


PROBLEMA




Existing application's classes
Industrial
Residential
Commercial
Construction
</> XML Export Implementation

SOLUCIÓN: CLASE VISITOR



```
1   foreach (Node node in graph)
2     if (node instanceof City)
3       exportVisitor.doForCity((City) node)
4     if (node instanceof Industry)
5       exportVisitor.doForIndustry((Industry) node)
6     // ...
7   }
```

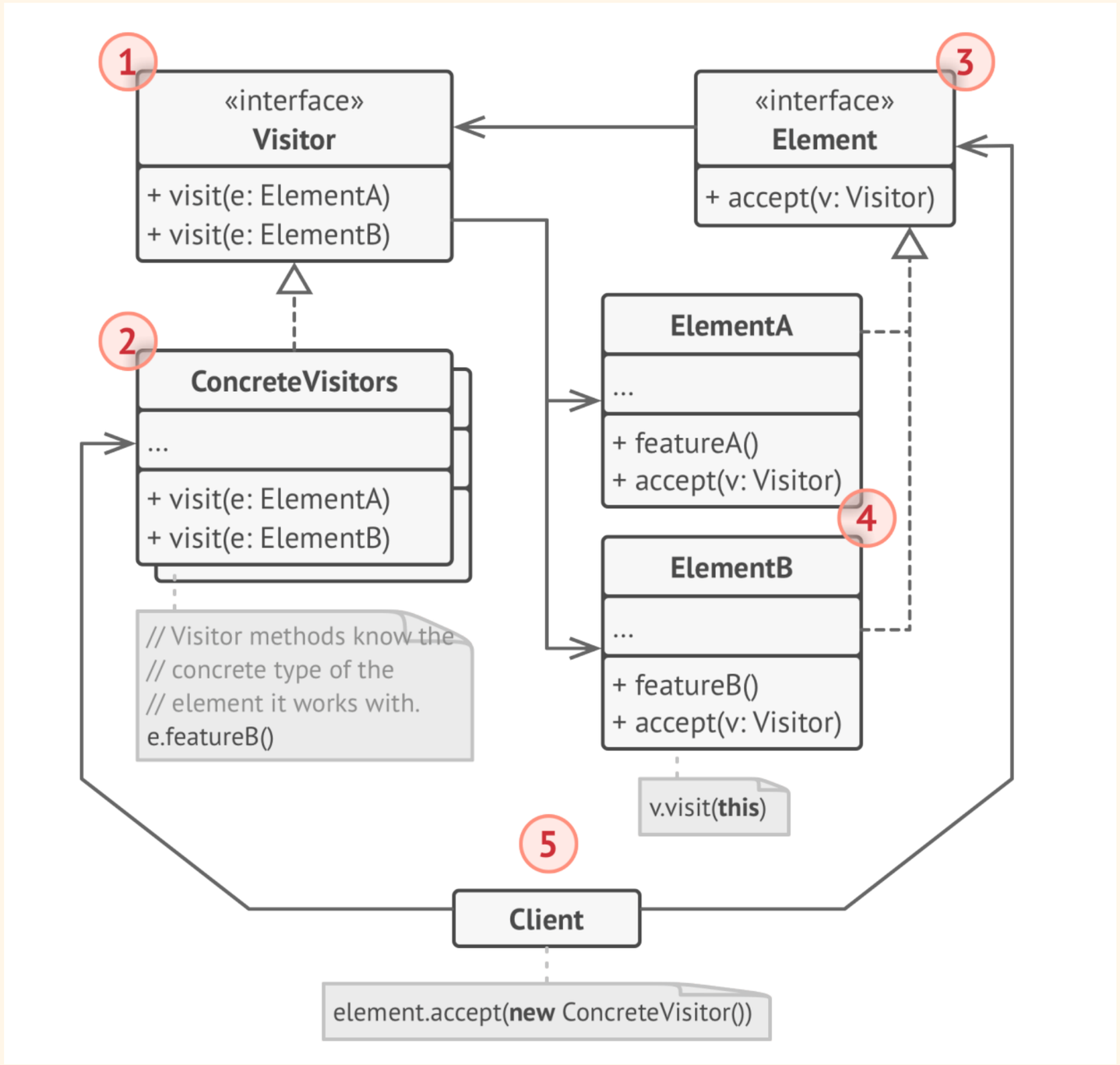
DOUBLE DISPATCH

```
// Client code
foreach (Node node in graph)
  node.accept(exportVisitor)

// City
class City is
  method accept(Visitor v) is
    v.doForCity(this)
  // ...

// Industry
class Industry is
  method accept(Visitor v) is
    v.doForIndustry(this)
  // ...
```

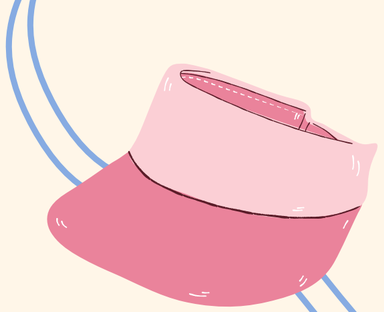
EN RESUMEN

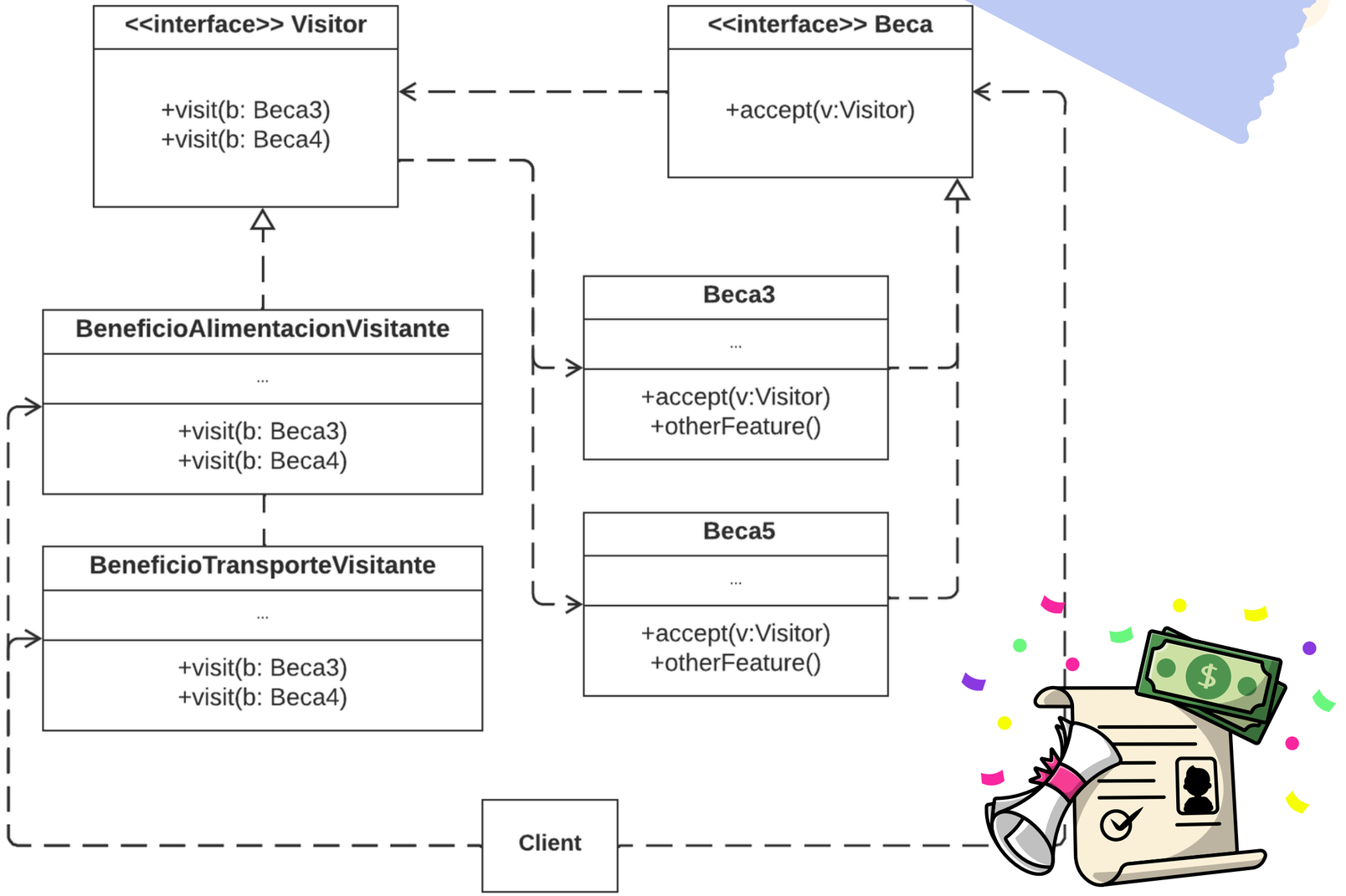
EL PATRÓN VISITOR ES UN PATRÓN DE COMPORTAMIENTO QUE PERMITE SEPARAR LOS ALGORÍTMOS DE LOS OBJETOS QUE LOS APLICAN

ESTRUCTURA


1
«interface»
Visitor
+ visit(e: ElementA)
+ visit(e: ElementB)
3
«interface»
Element
+ accept(v: Visitor)
2
ConcreteVisitors
...
+ visit(e: ElementA)
+ visit(e: ElementB)
// Visitor methods know the
// concrete type of the
// element it works with.
e.featureB()
ElementA
...
+ featureA()
+ accept(v: Visitor)
4
ElementB
...
+ featureB()
+ accept(v: Visitor)
v.visit(this)
5
Client
element.accept(new ConcreteVisitor())

EJEMPLO




<<interface>> Visitor
+visit(b: Beca3)
+visit(b: Beca4)
<<interface>> Beca
+accept(v:Visitor)
BeneficioAlimentacionVisitante
...
+visit(b: Beca3)
+visit(b: Beca4)
Beca3
...
+accept(v:Visitor)
+otherFeature()
BeneficioTransporteVisitante
...
+visit(b: Beca3)
+visit(b: Beca4)
Beca5
...
+accept(v:Visitor)
+otherFeature()
Client

```java
public interface Beca {
    String getName();
    void accept(Visitante v);
}
```

ELEMENT

```java
public class Beca3 implements Beca{
    public String getName(){
        return "Beca 3";
    }

    public void accept (Visitante visitante){
        visitante.visit(this);
    }

}
```

CONCRETE ELEMENT

```java
public class Beca5 implements Beca{
    public String getName(){
        return "Beca 5";
    }

    public void accept (Visitante visitante){
        visitante.visit(this);
    }
```

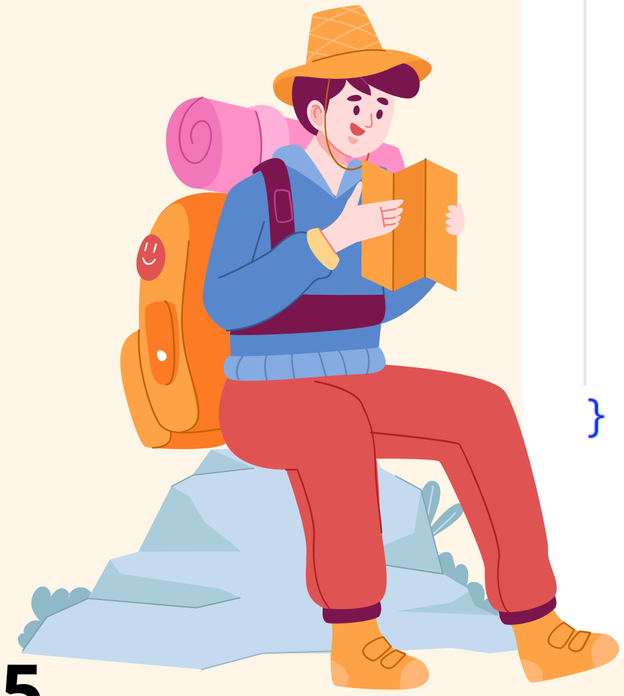
CONCRETE ELEMENT

```
public interface Visitante {

    public void visit(Beca3 beca);

    public void visit(Beca5 beca);

}
```

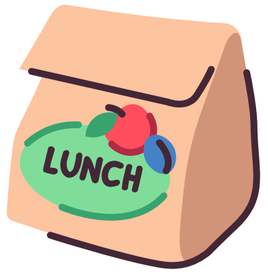
BENEFITS

VISITOR

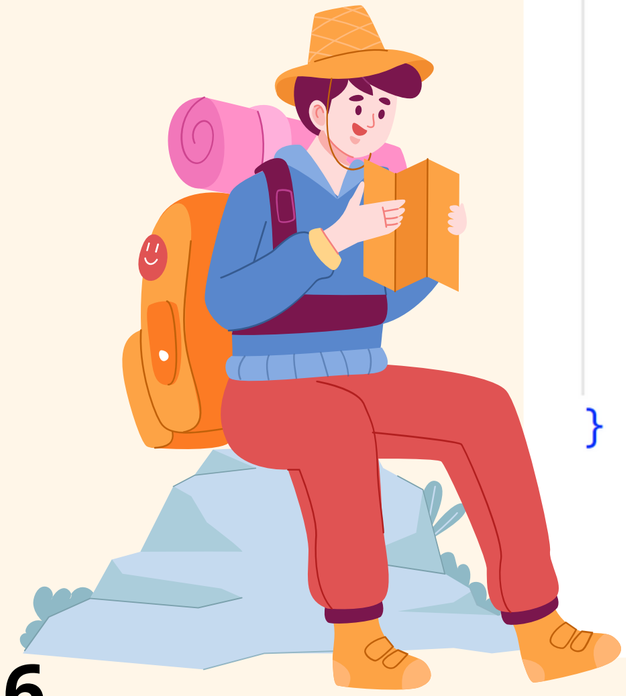
```java
public class BeneficioAlimentacionVisitante implements Visitante {

    public void visit(Beca3 beca){
        System.out.println(x:"70% de descuento en alimentación para Beca 3");
    }

    public void visit(Beca5 beca){
        System.out.println(x:"100% de descuento en alimentación para Beca 5");
    }

}
```



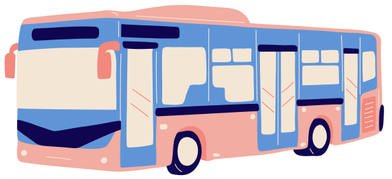
CONCRETE VISITOR



```java
public class BeneficioTransporteVisitante implements Visitante {

    public void visit(Beca3 Beca){
        System.out.println(x:"0% de descuento en transporte para Beca 3");
    }

    public void visit(Beca5 Beca){
        System.out.println(x:"100% de descuento en transporte para Beca 5");
    }

}
```



CONCRETE VISITOR



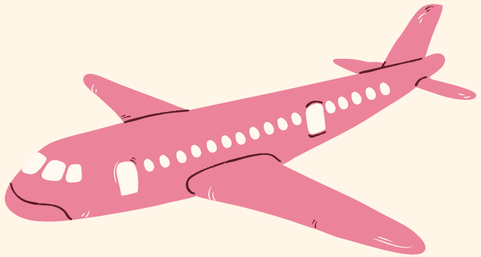
```java
public static void main(String[] args) {

    List<Visitante> visitanteLista = new ArrayList<>();

    List<Beca> becaLista = new ArrayList<>();

    Visitante alimentacion = new BeneficioAlimentacionVisitante();
    Visitante transporte = new BeneficioTransporteVisitante();

    Beca beca3 = new Beca3();
    Beca beca5 = new Beca5();

    visitanteLista.add(alimentacion);
    visitanteLista.add(transporte);
    becaLista.add(beca3);
    becaLista.add(beca5);
```

```
for (Visitante visitante : visitanteLista) {
    for (Beca beca : becaLista) {
        beca.accept(visitante);
    }
    System.out.println();
}
```

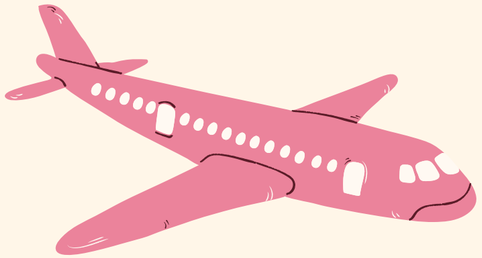
70% de descuento en alimentación para Beca 3
100% de descuento en alimentación para Beca 5

0% de descuento en transporte para Beca 3
100% de descuento en transporte para Beca 5

DOUBLE DISPATCH



"Determina el método a utilizar dependiendo de los objetos involucrados (dos o más)."



```
for (Visitante visitante : visitanteLista) {
    for (Beca beca : becaLista) {
        beca.accept(visitante);
    }
    System.out.println();
}
```

```
for (Visitante visitante : visitanteLista) {
    for (Beca beca : becaLista) {
        visitante.visit(beca);;
    }
    System.out.println();
}
```

```
public interface Visitante {

    public void visit(Beca3 beca);

    public void visit(Beca5 beca);

    public void visit(Beca beca);

}
```

```
public void visit(Beca Beca){
    System.out.println(x:"Descuento en transporte para Beca Genérica");
}
```

Descuento en alimentación para Beca Genérica
Descuento en alimentación para Beca Genérica

Descuento en transporte para Beca Genérica
Descuento en transporte para Beca Genérica

CONSECUENCIAS

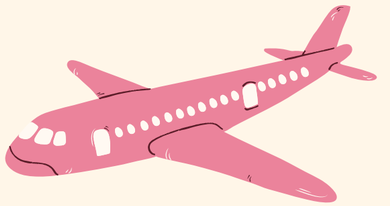
- Single Responsibility ✅
- Open/Closed ✅
- Mantenimiento de visitantes ❌

COMMAND

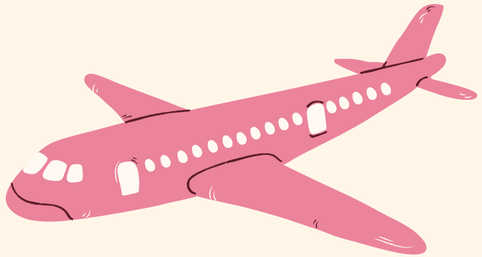
COMPOSITE

ITERATOR

PATRONES RELACIONADOS



REFERENCIAS



- *Geekific. (2021, October 23). The Visitor Pattern Explained and Implemented in Java | Behavioral Design Patterns | Geekific [Video]. YouTube. https://www.youtube.com/watch?v=UQP5XqMqtqQ*
- **Ryan Schachte. (2019, January 20). Understanding The Visitor Design Pattern [Video]. YouTube. https://www.youtube.com/watch?v=TeZqKnC2gvA**
- **Refactoring.Guru. (2023b). Visitor. Refactoring.Guru. https://refactoring.guru/design-patterns/visitor**