

# Software Engineering Design & Construction

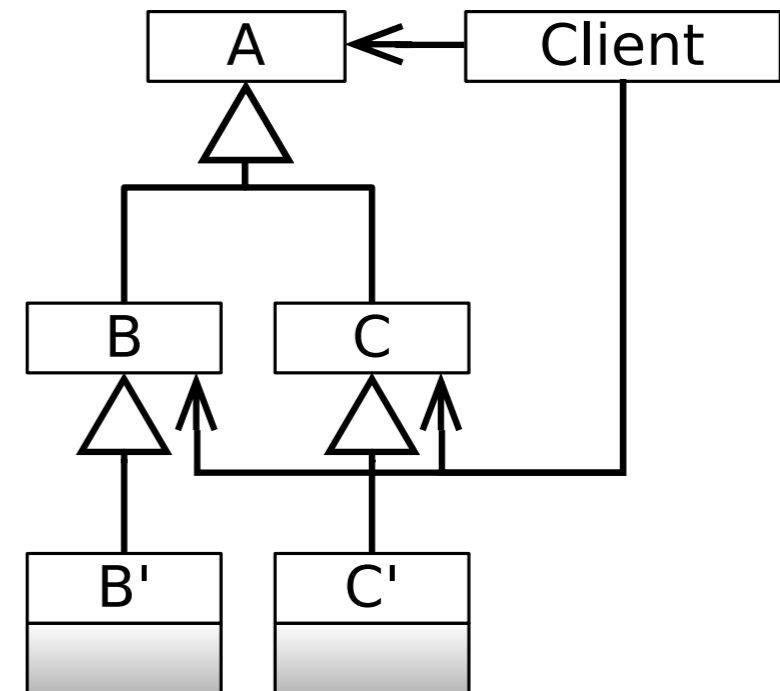
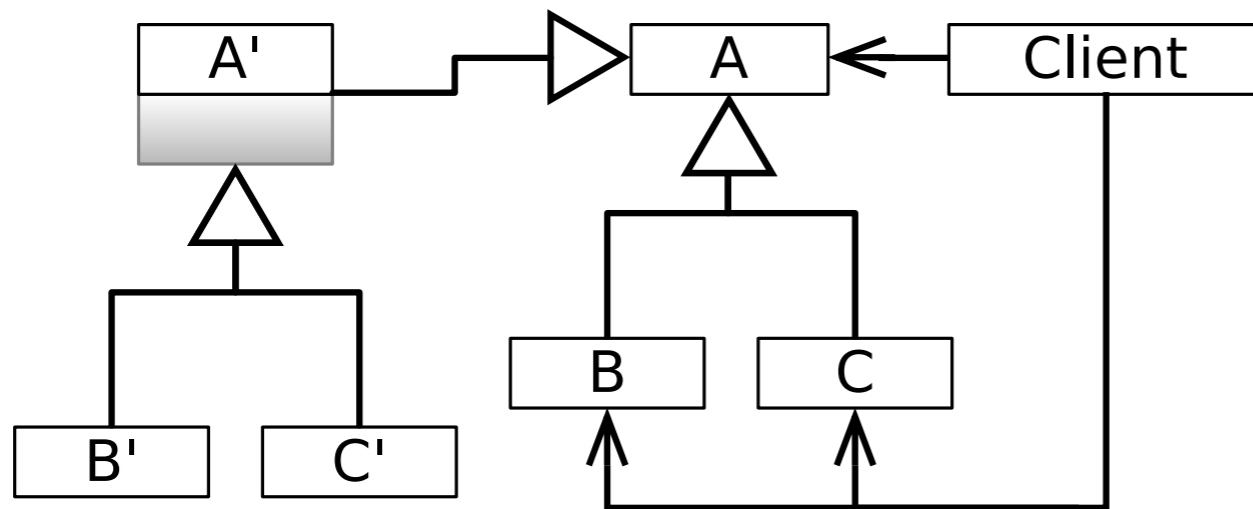
Dr. Michael Eichberg  
Fachgebiet Softwaretechnik  
Technische Universität Darmstadt

---

Visitor Pattern

---

Recall the problems of inheritance with modeling variations at the level of multiple objects (object composites).



## Solution Idea

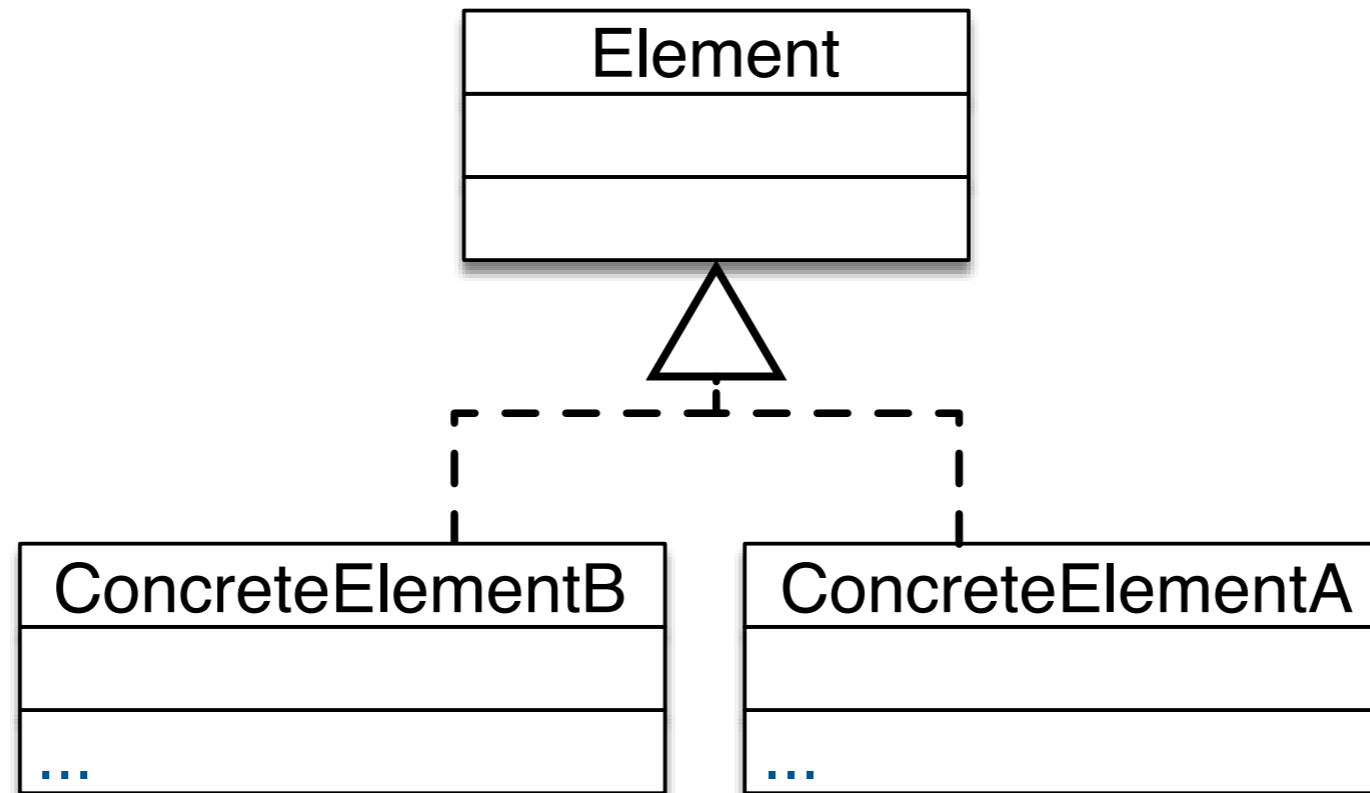
Represent the additional operations to be performed on the elements of an object structure (*additional features*) as objects (of type **Visitor**).

# Visitor Design Pattern

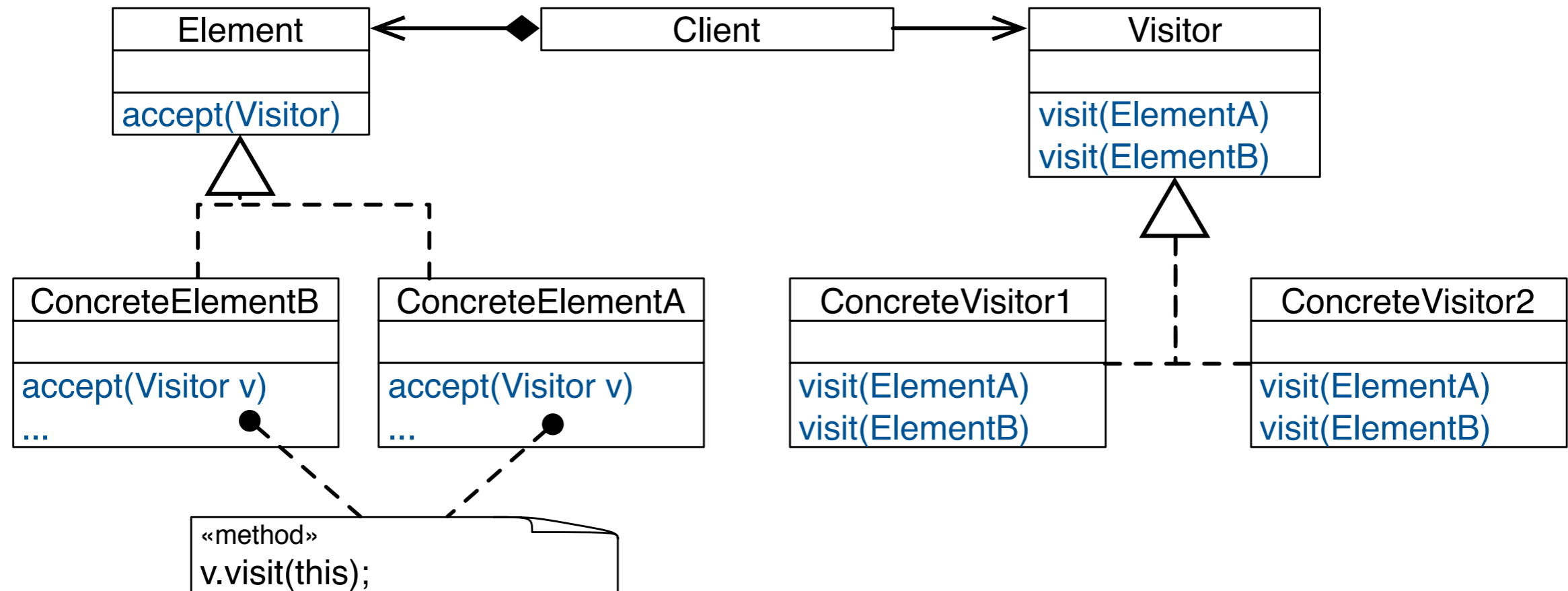
---

The Visitor Pattern enables to add new behavior to existing classes in a **fixed class hierarchy** without changing this hierarchy.

# Structure - the Basic Class Hierarchy

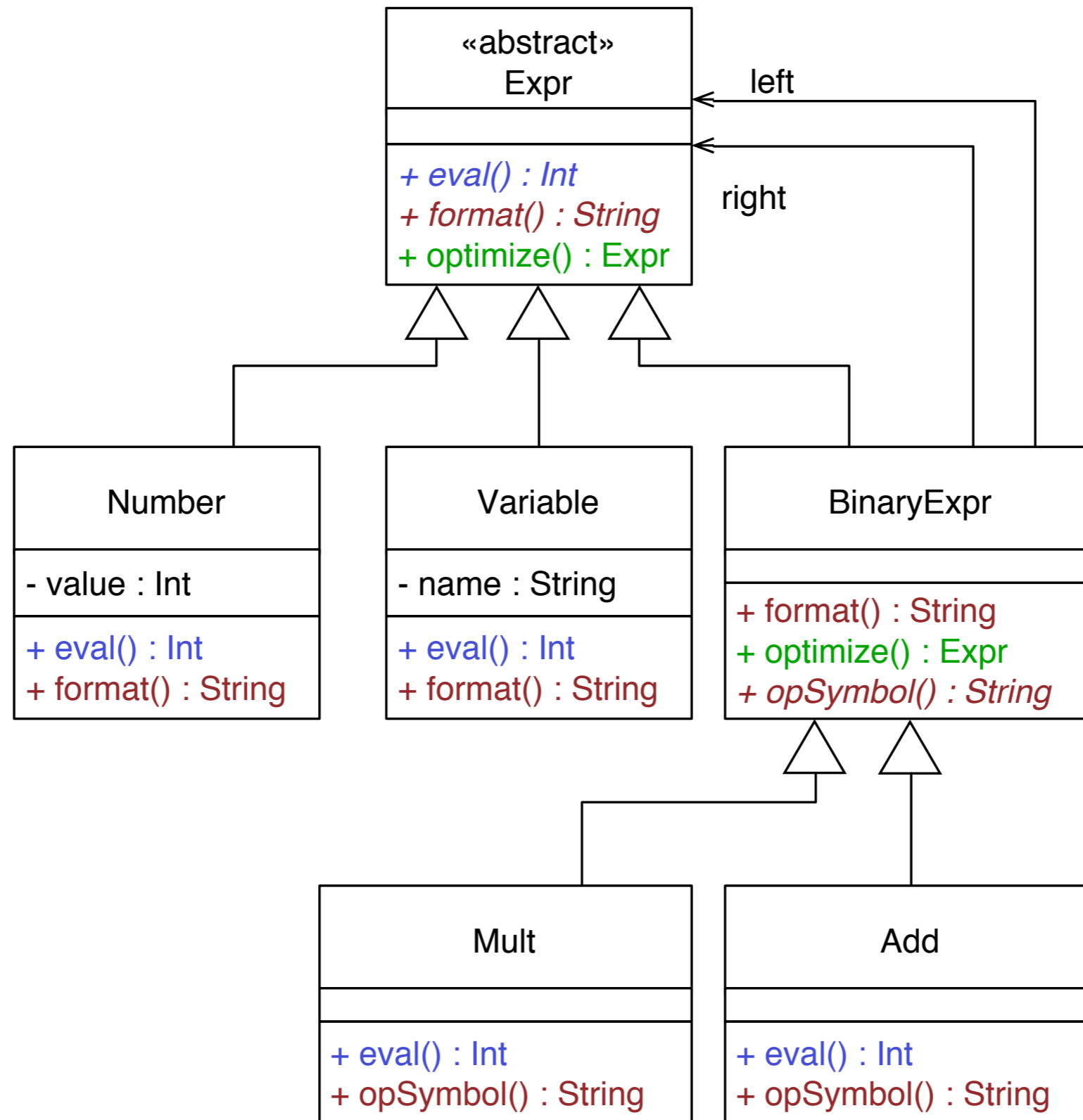


# Structure

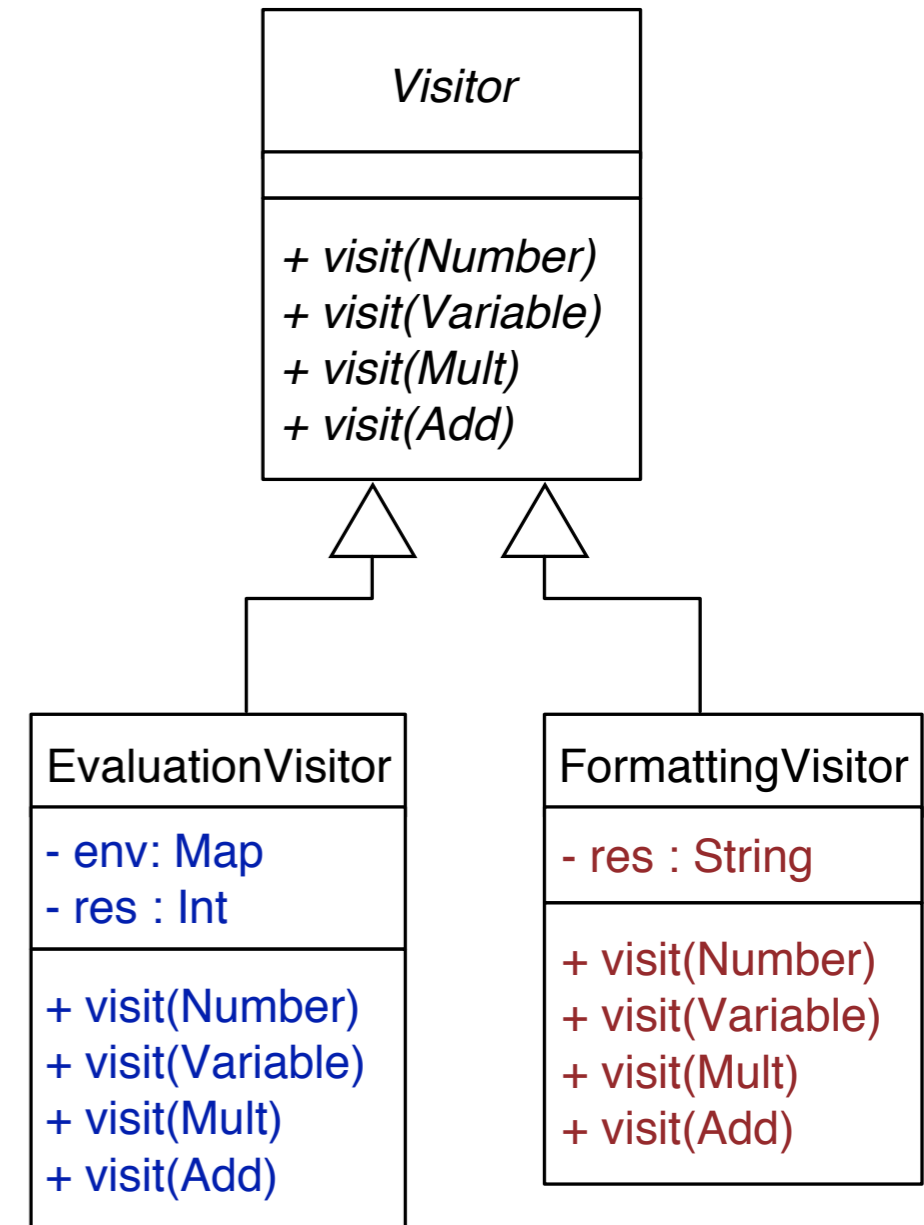
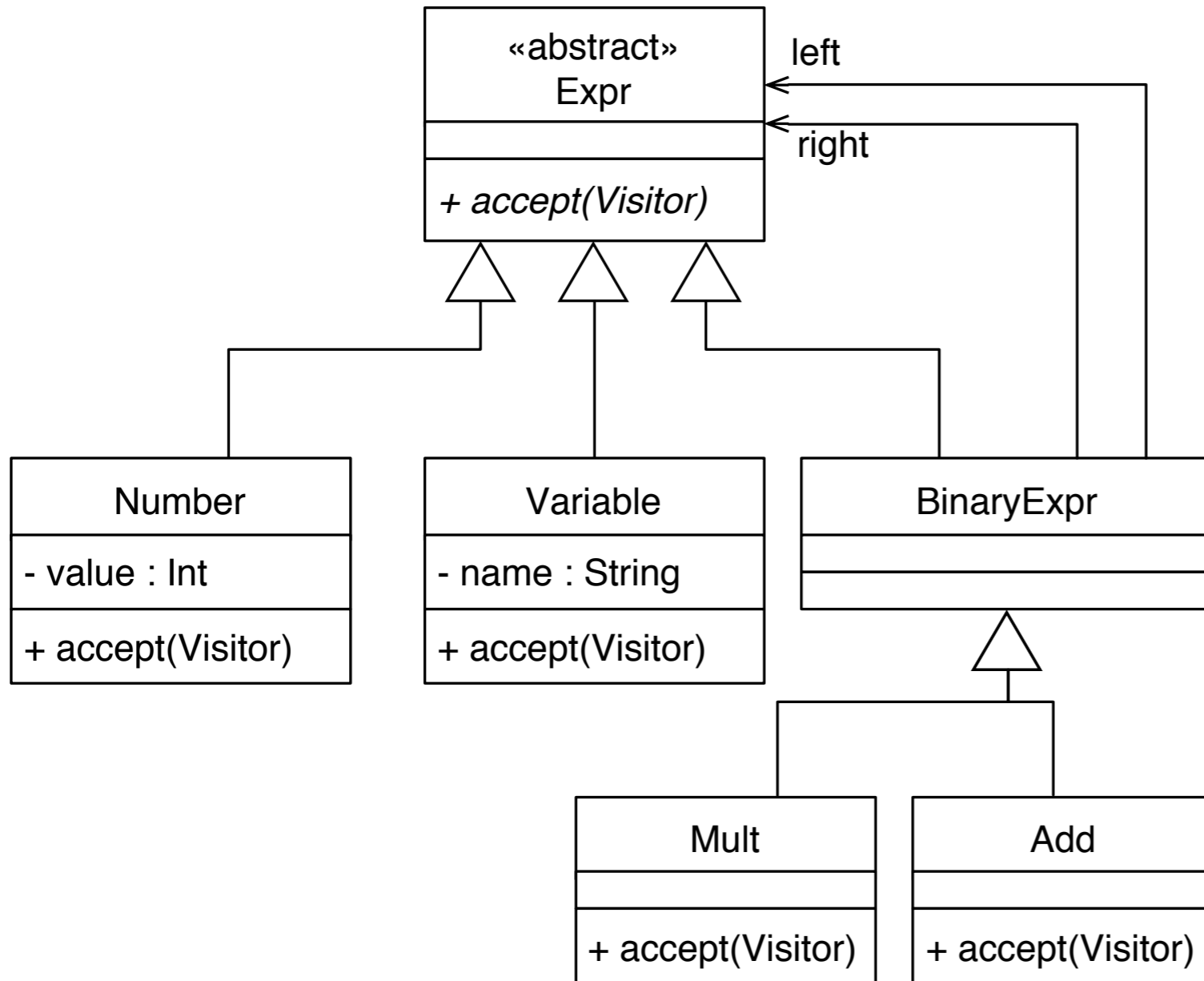


```
Element e = new ConcreteElementA(...);  
Visitor v = new ConcreteVisitor1(...);  
e.accept(v);
```

# Case-Study: Arithmetic Expressions



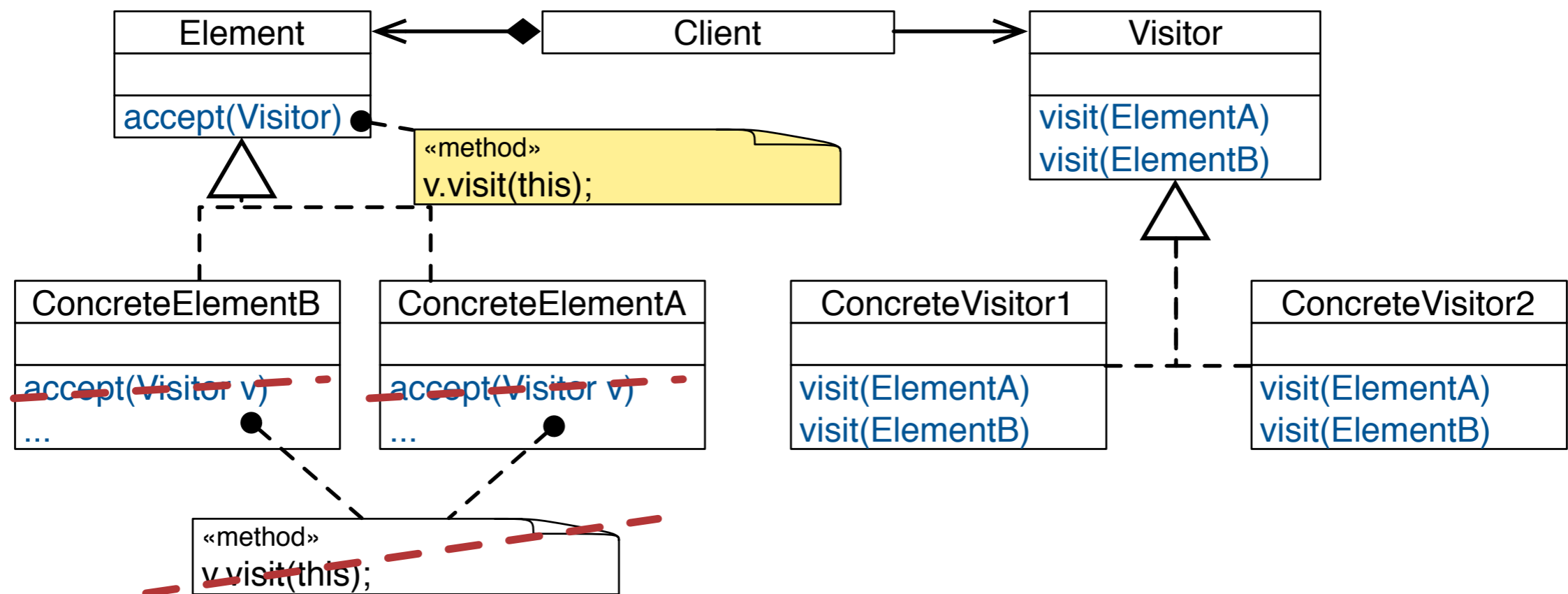
# Visitor Based Design





# Reflections on the Visitor Structure

Can we move the implementation of accept higher up the Element hierarchy?



# Double Dispatch

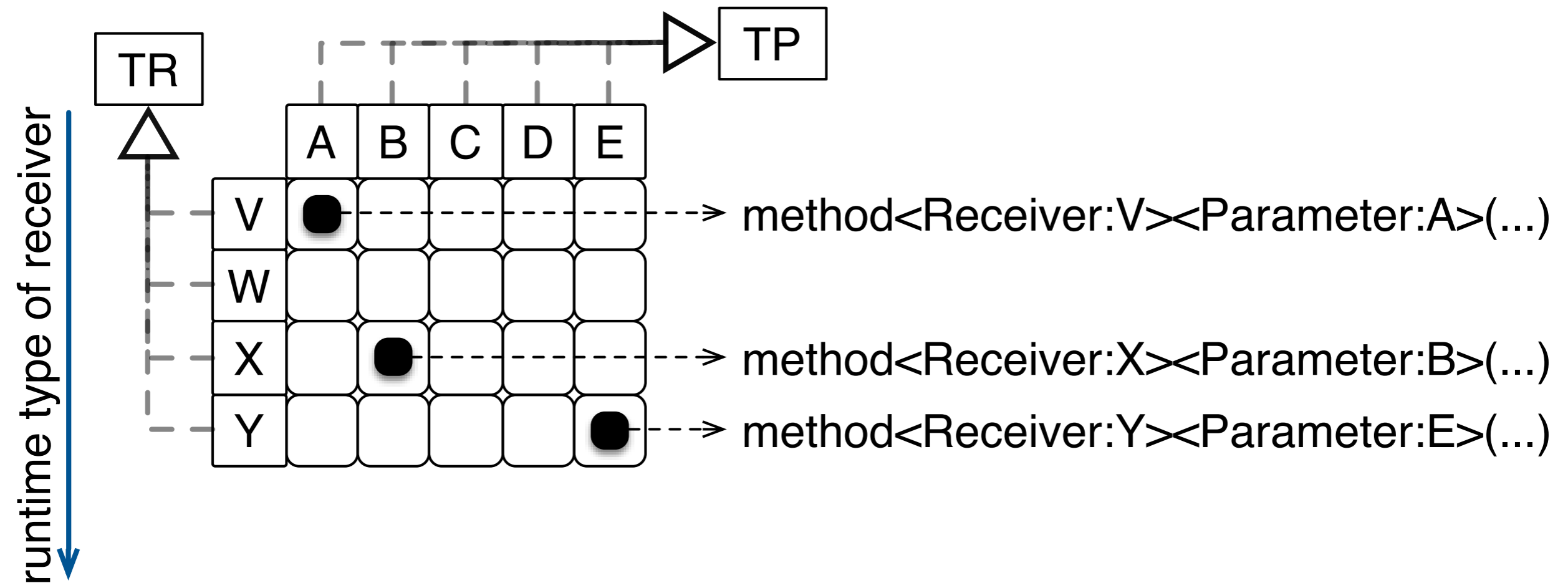
---

Dispatching an operation based on the **dynamic type** of two objects is called double dispatch.

Double dispatch is not supported in mainstream OO languages, e.g., Java, C#, Scala,.....

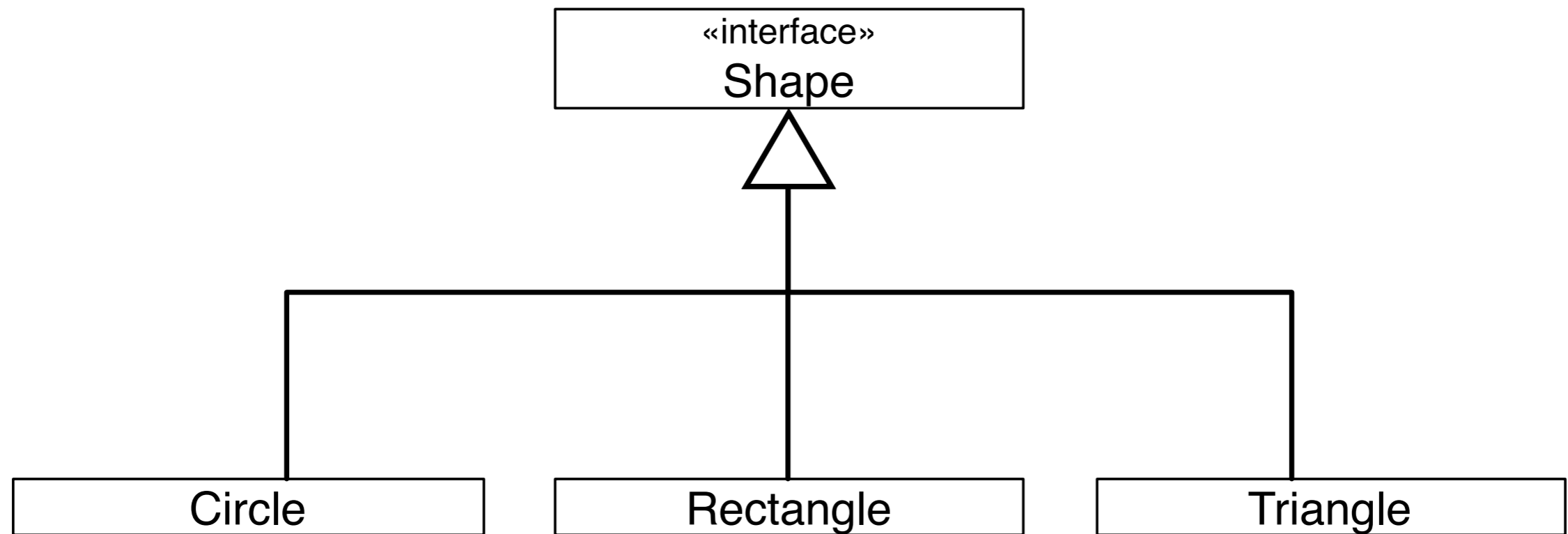
# Double-Dispatch

runtime type of first parameter



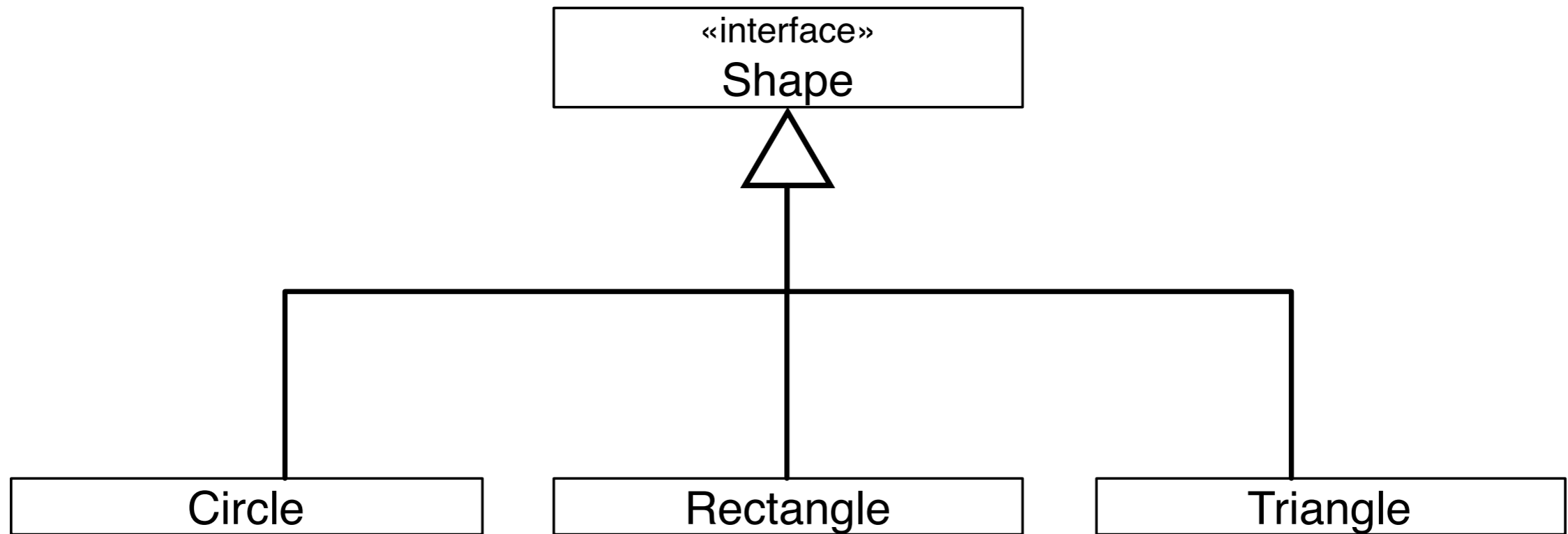
# Case-Study: Calculating Shape Intersection

Task: Implement an intersect operation that calculates whether two given shapes intersect.



# Case-Study: Calculating Shape Intersection

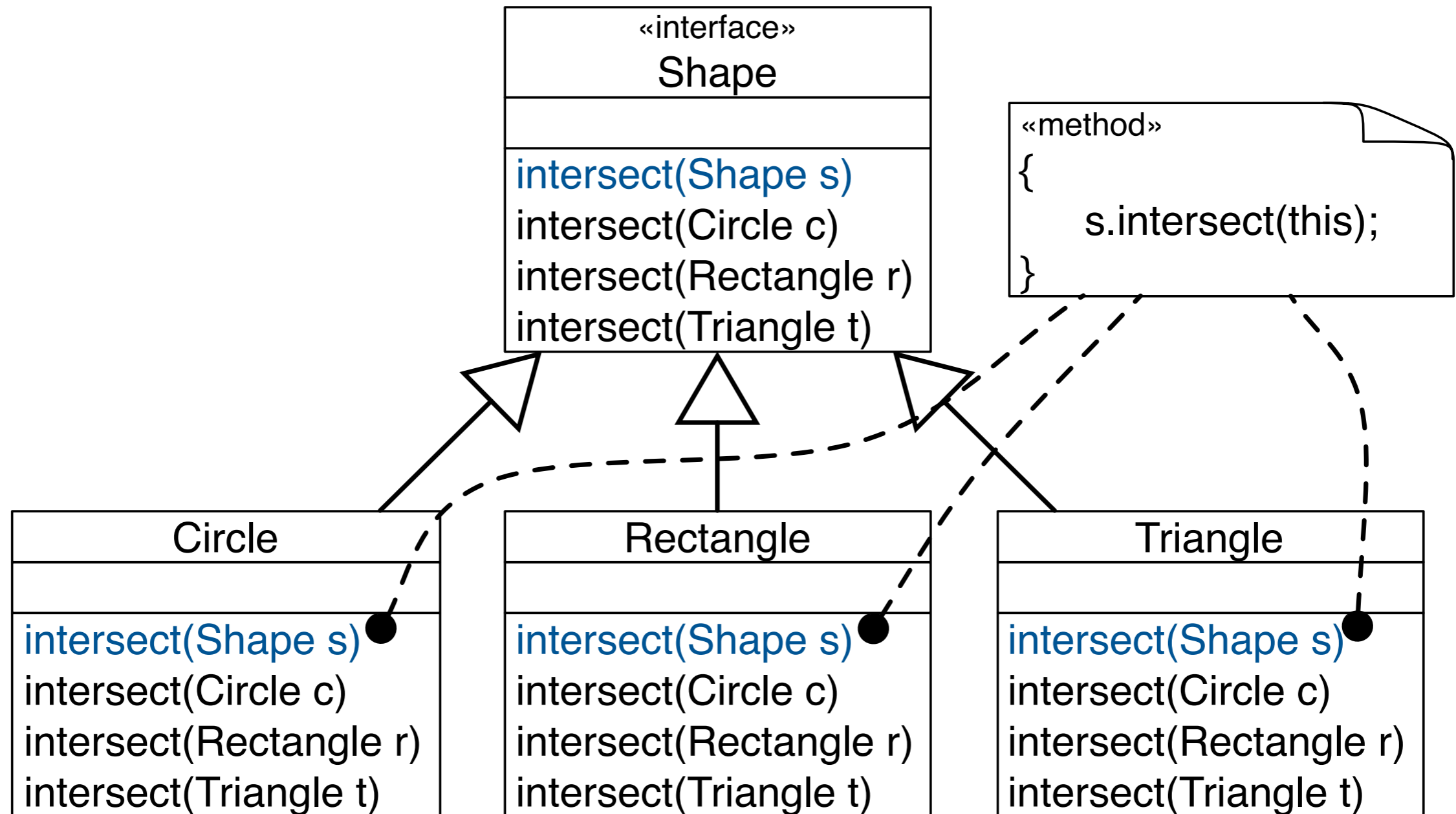
Task: Implement an intersect operation that calculates whether two given shapes intersect.



```
Shape t = new Triangle(...);
Shape r = new Rectangle(...);
if (t.intersect(r)) {...}
```

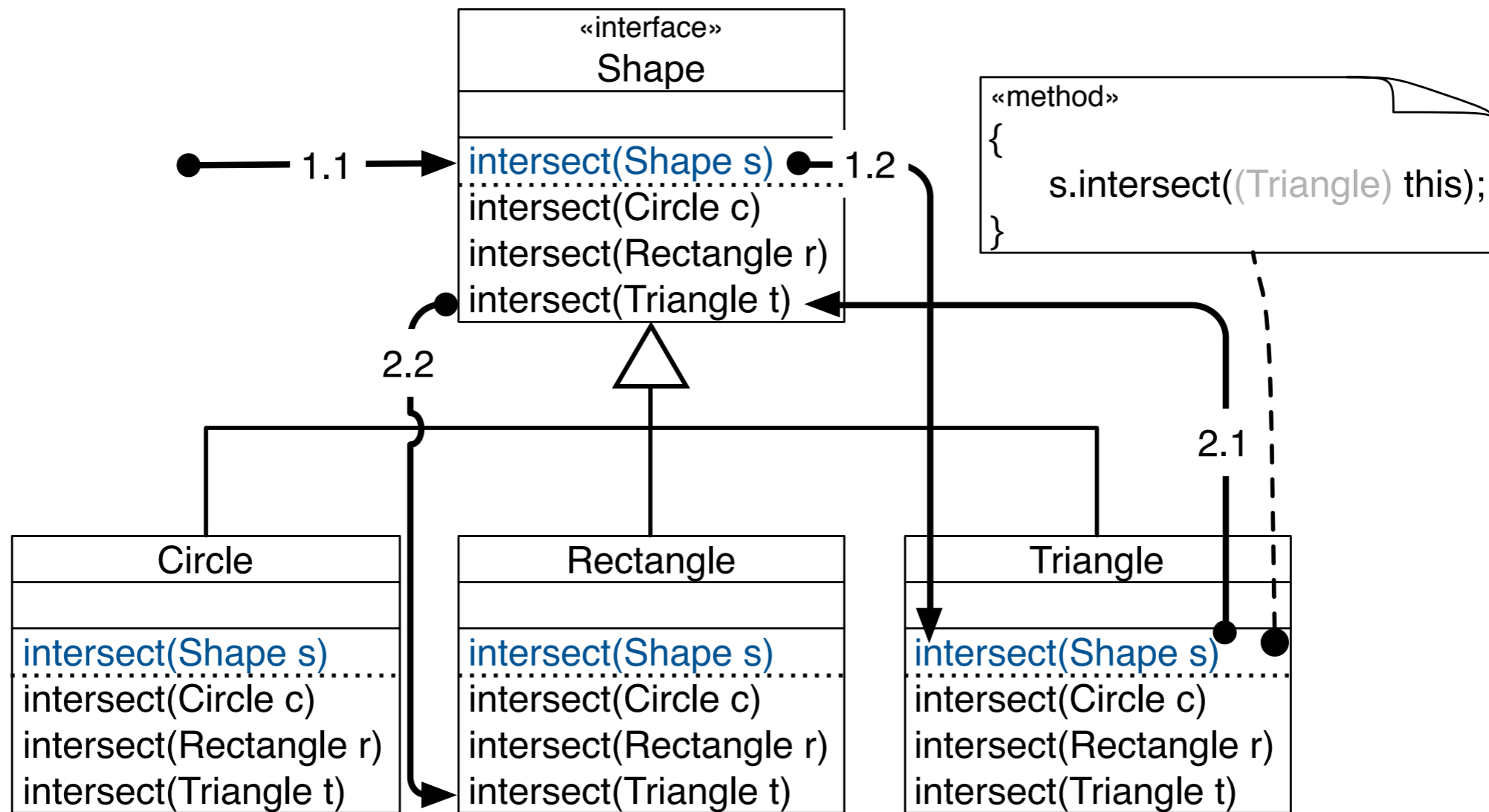
# Case-Study: Calculating Shape Intersection

## Simulating Double Dispatch



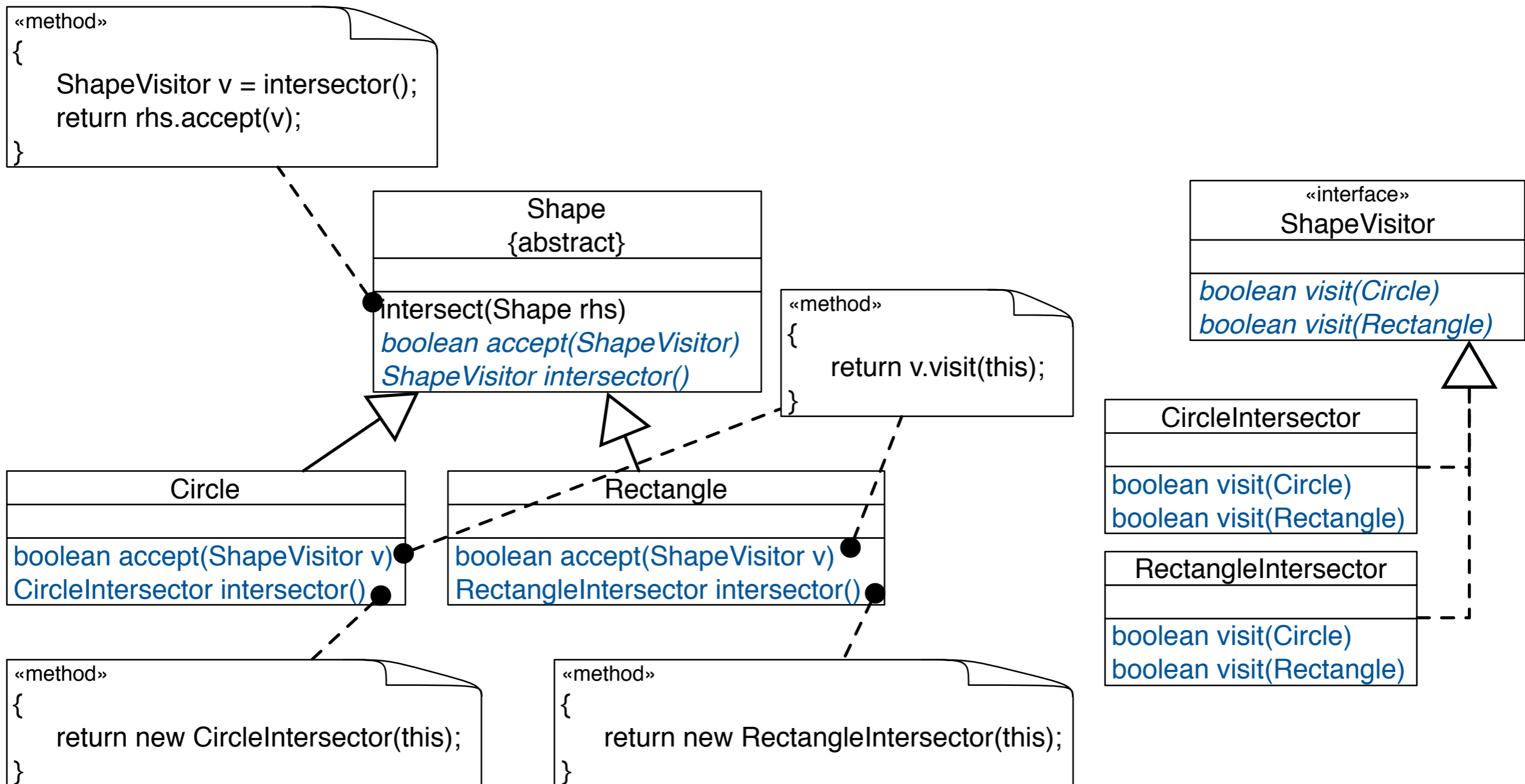
# Case-Study: Calculating Shape Intersection

## Simulating Double Dispatch



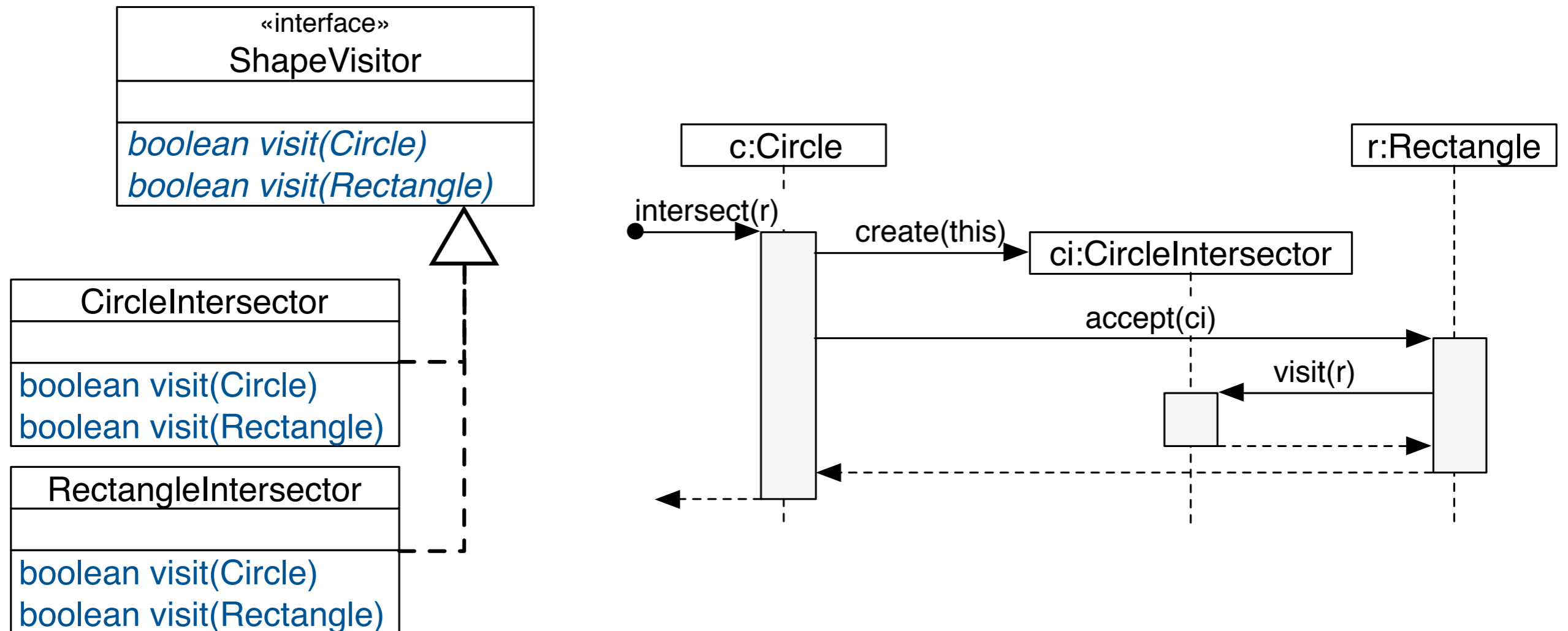
```
Shape t = new Triangle(...);  
Shape r = new Rectangle(...);  
if (t.intersect(r)) {...}
```

# Case-Study: Shape Intersection Using Visitor





# Case-Study: Shape Intersection Using Visitor



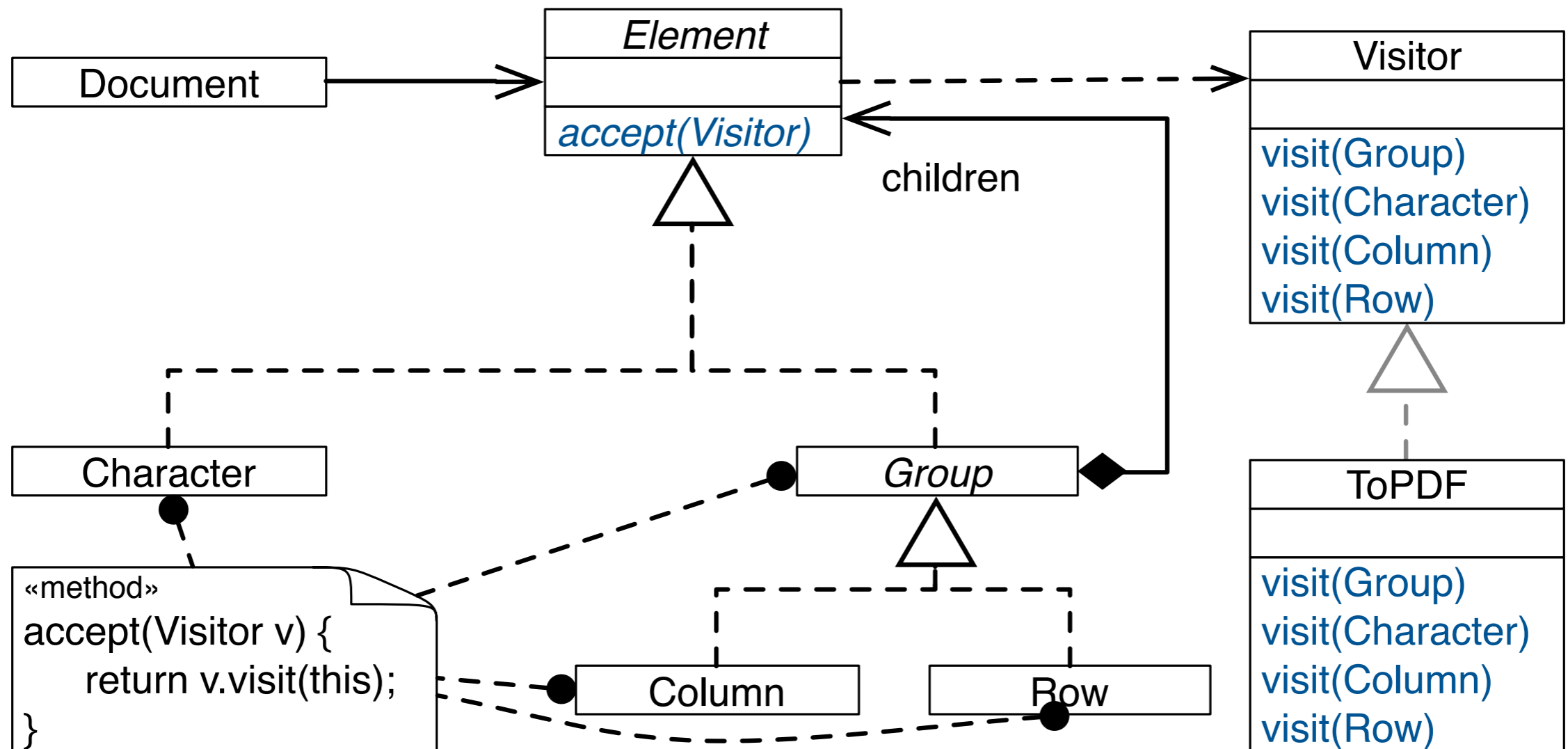
```
Shape c = new Circle(...);  
Shape r = new Rectangle(...);  
if (c.intersect(r)) {...}
```

# Advantages of the Visitor Design Pattern

- **New operations are easy to add** without changing element classes (add a new concrete visitor).  
Different concrete elements do not have to implement their part of a particular algorithm.
- Related behavior focused in a single concrete visitor.
- **Visiting across hierarchies:** Visited classes are not forced to share a common base class.
- **Accumulating state:** Visitors can accumulate state as they visit each element, thus, encapsulating the algorithm and all its data.

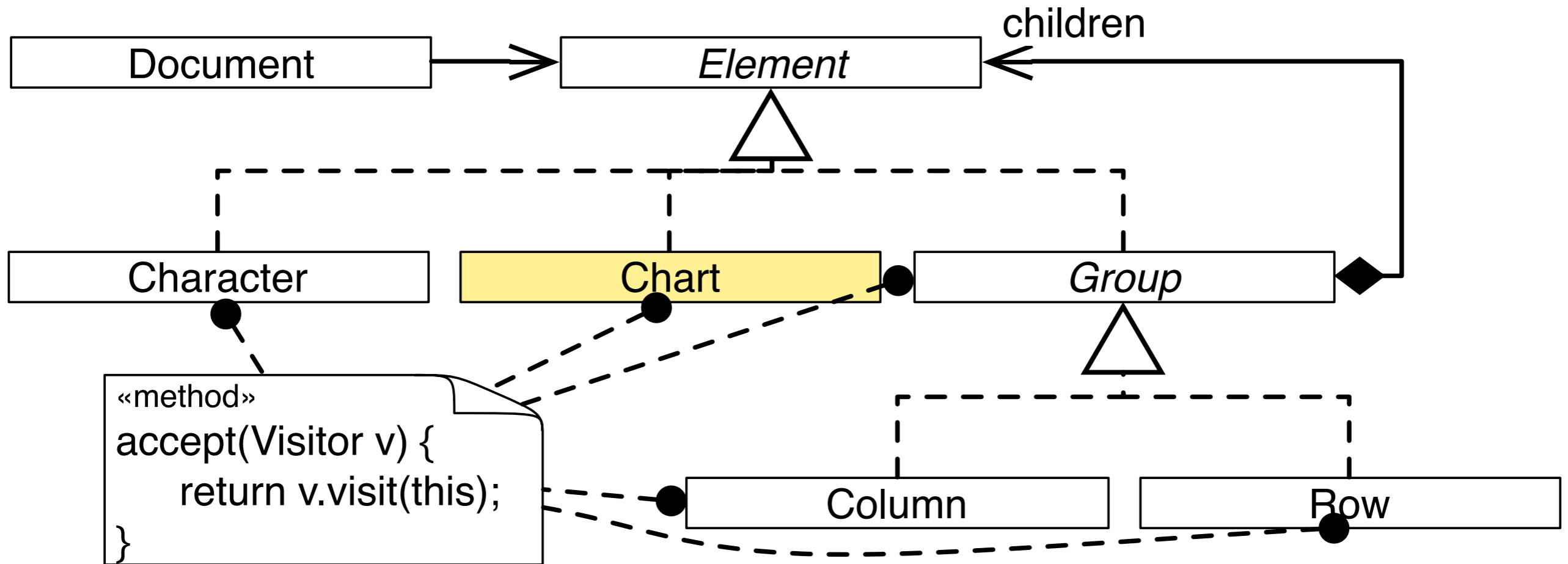
# Issues of the Visitor-Based Design

What happens if we want to add a new element?



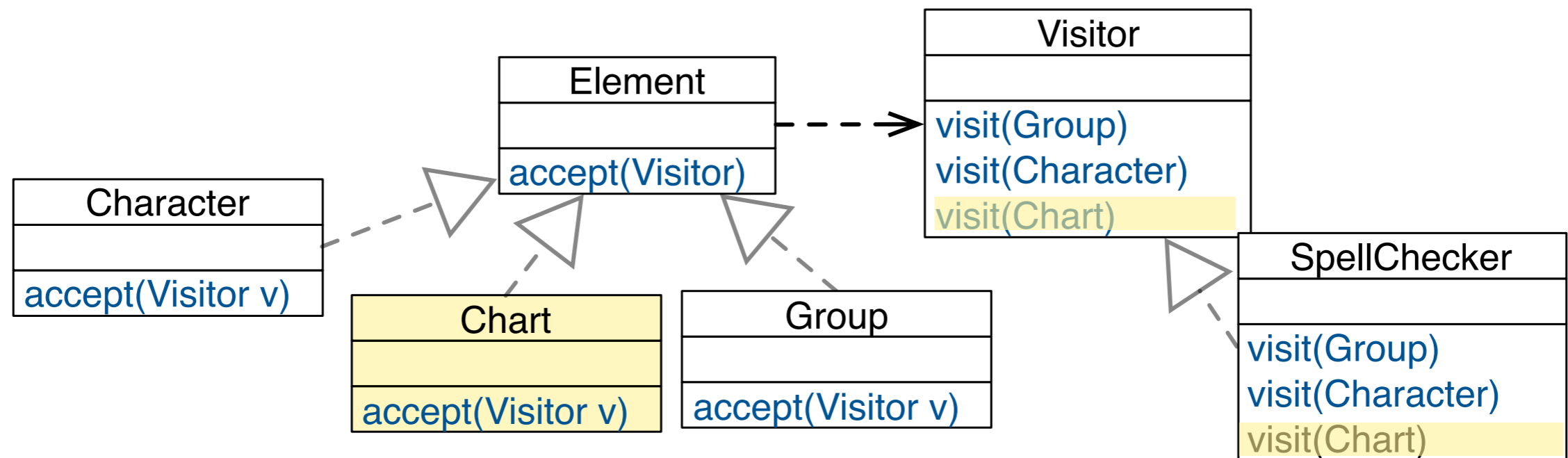
# Issues of the Visitor-Based Design

E.g., adding **Chart** (*adding new kinds of Elements*)



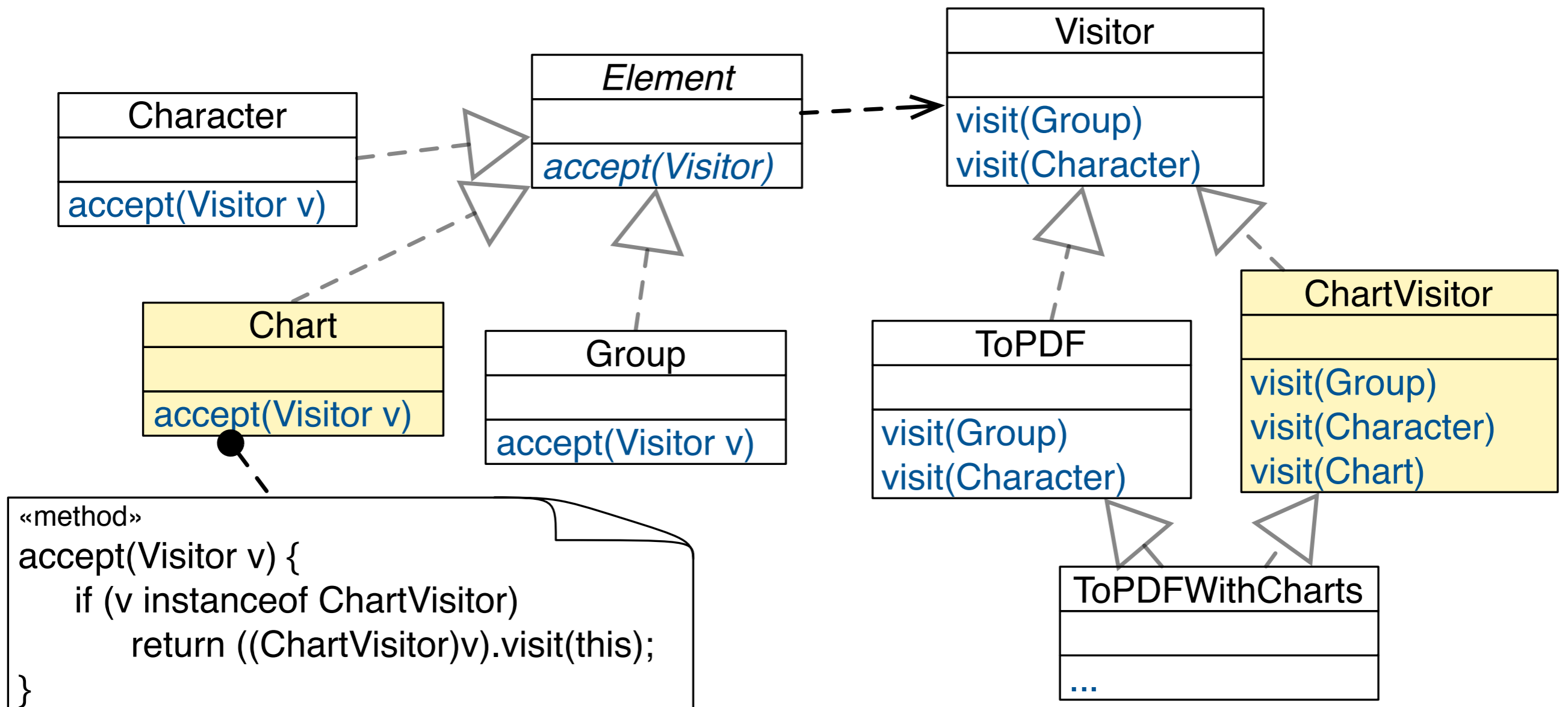
# Issues of the Visitor-Based Design

E.g., adding `Chart` and updating `Visitor`



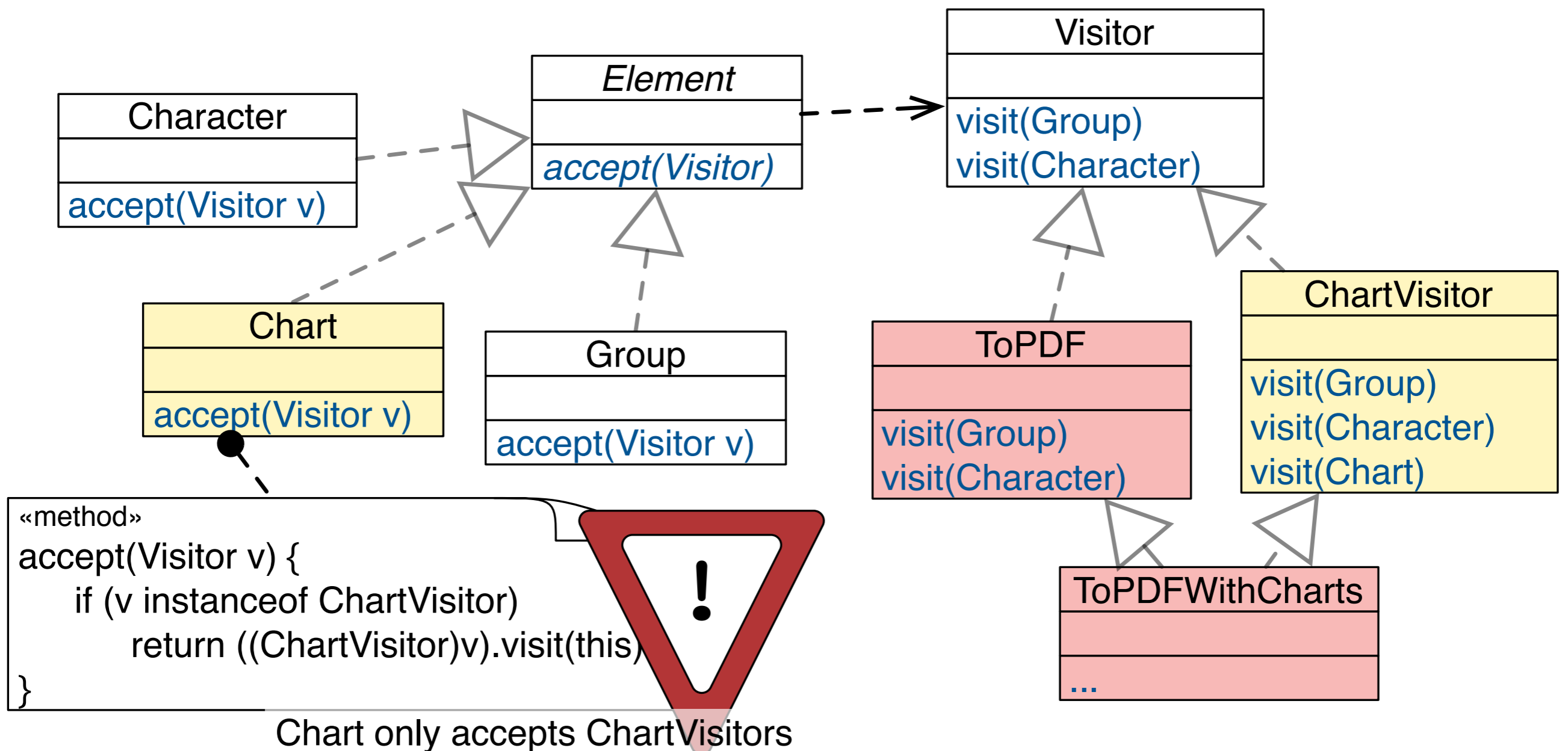
# Issues of the Visitor-Based Design

E.g., adding **Chart** and keeping **Visitor** unchanged



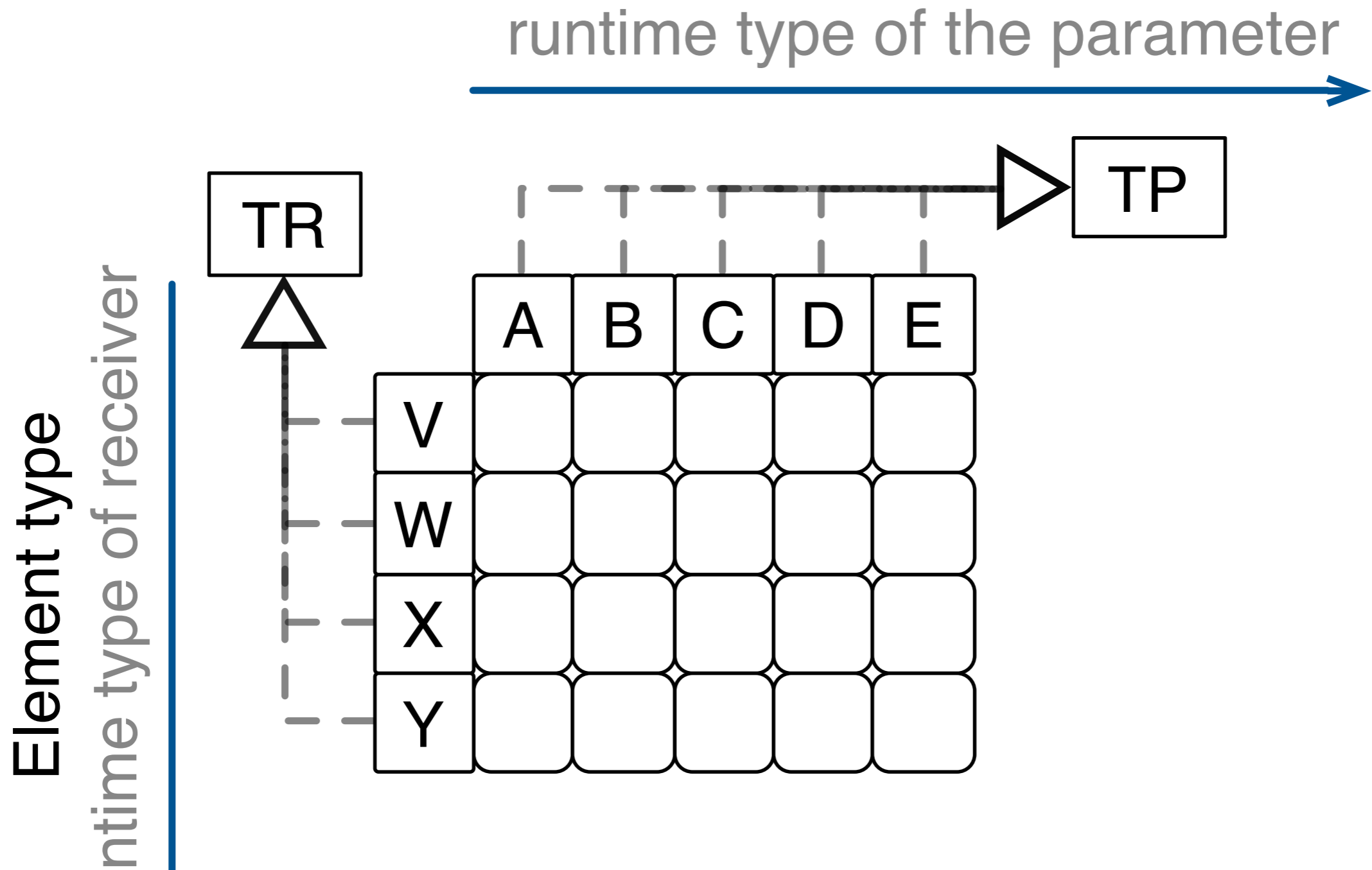
# Issues of the Visitor-Based Design

E.g., adding `Chart` and keeping `Visitor` unchanged



# Issues of the Visitor-Based Design

Partial Visiting Is Not Supported






# Takeaway

- Visitor brings functional-style decomposition to OO designs.
- **Use Visitor for stable element hierarchies.**  
Visitor works well in data hierarchies where new elements are never or at least not very often added.
- **Do not use it, if new elements are a likely change.**
- Visitor only makes sense if we have to add new operations often! In this case Visitor closes our design against these changes.

# Solving the Expression Problem in Scala

The base trait.

```
trait Expressions {  
  type Expression <: TExpression  
  trait TExpression {  
    def eval: Double  
  }  
  
  trait Constant extends TExpression {  
    val v: Double  
    def eval = v  
  }  
}
```



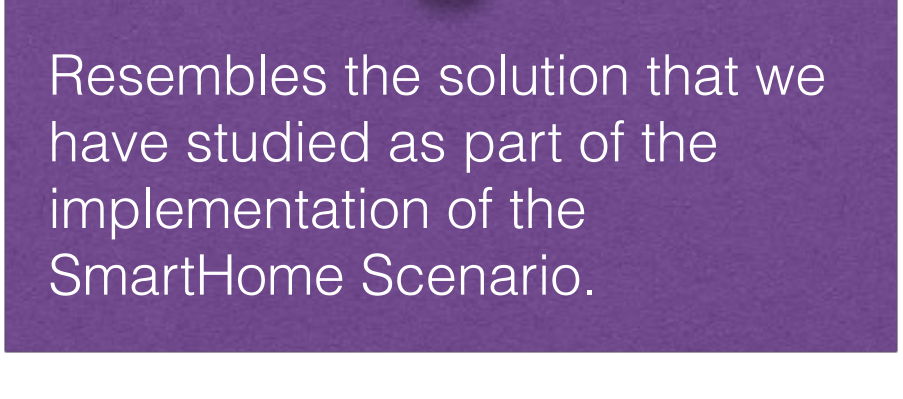
Resembles the solution that we have studied as part of the implementation of the SmartHome Scenario.

Without Visitors

# Solving the Expression Problem in Scala

Adding a new data-type.

```
trait AddExpressions extends Expressions {  
  trait Add extends TExpression {  
    val l: Expression  
    val r: Expression  
    def eval = l.eval + r.eval  
  }  
}
```



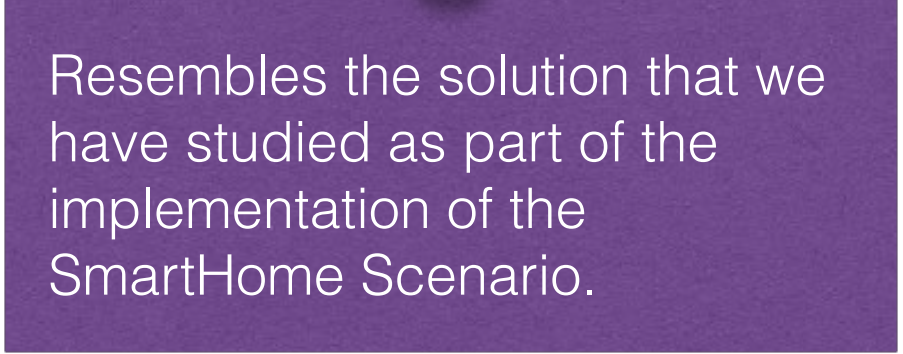
Resembles the solution that we have studied as part of the implementation of the SmartHome Scenario.

Without Visitors

# Solving the Expression Problem in Scala

Adding new functionality.

```
trait PrefixNotationForExpressions extends AddExpression {  
  type Expression <: TExpression  
  trait TExpression extends super.TExpression {  
    def prefixNotation: String  
  }  
  
  trait Constant extends super.Constant with TExpression {  
    def prefixNotation = v.toString  
  }  
  
  trait Add extends super.Add with TExpression {  
    def prefixNotation = "+" + l.prefixNotation + r.prefixNotation  
  }  
}
```



Resembles the solution that we have studied as part of the implementation of the SmartHome Scenario.

Without Visitors

# Solving the Expression Problem in Scala

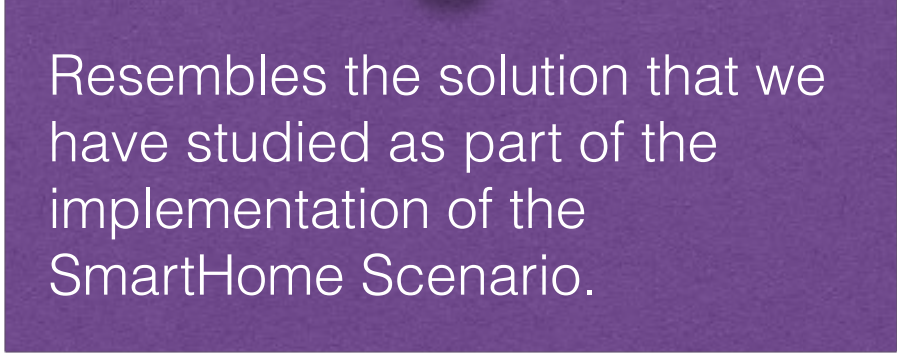
Bringing everything together.

```
object ExpressionsFramework
  extends PrefixNotationForExpressions
  with PostfixNotationForExpressions {

  type Expression = TExpression
  trait TExpression
    extends super[PrefixNotationForExpressions].TExpression
    with super[PostfixNotationForExpressions].TExpression

  case class Constant(v: Double)
    extends super[PrefixNotationForExpressions].Constant
    with super[PostfixNotationForExpressions].Constant
    with Expression

  case class Add(val l: Expression, val r: Expression)
    extends super[PrefixNotationForExpressions].Add
    with super[PostfixNotationForExpressions].Add
    with Expression
}
```



Resembles the solution that we have studied as part of the implementation of the SmartHome Scenario.

Without Visitors

# Solving the Expression Problem in Scala

The base trait.

```
trait Expressions {  
  
  trait Expression { def accept[T](visitor: Visitor[T]): T }  
  
  class Constant(val v: Double) extends Expression {  
    def accept[T](visitor: Visitor[T]): T = visitor.visitConstant(v)  
  }  
  
  type Visitor[T] <: TVisitor[T]  
  trait TVisitor[T] {  
    def visitConstant(v: Double): T  
  }  
  
  trait EvalVisitor extends TVisitor[Double] {  
    def visitConstant(v: Double): Double = v  
  }  
}
```

# Solving the Expression Problem in Scala

Adding a new data-type.

```
trait AddExpressions extends Expressions {  
  class Add(val l: Expression,  
            val r: Expression) extends Expression {  
    def accept[T](visitor: Visitor[T]): T = visitor.visitAdd(l, r)  
  }  
  
  type Visitor[T] <: TVisitor[T]  
  trait TVisitor[T] extends super.TVisitor[T] {  
    def visitAdd(l: Expression, r: Expression): T  
  }  
  
  trait EvalVisitor extends super.EvalVisitor with TVisitor[Double] {  
    this: Visitor[Double] =>  
    def visitAdd(l: Expression, r: Expression): Double =  
      l.accept(this) + r.accept(this)  
  }  
}
```

With Visitors



# Solving the Expression Problem in Scala

Bringing everything together.

```
trait ExtendedExpressions extends AddExpressions with MultExpressions {  
  
  type Visitor[T] = TVisitor[T]  
  trait TVisitor[T]  
    extends super[AddExpressions].TVisitor[T]  
    with super[MultExpressions].TVisitor[T]  
  
  object EvalVisitor  
    extends super[AddExpressions].EvalVisitor  
    with super[MultExpressions].EvalVisitor  
    with TVisitor[Double] {  
    this: Visitor[Double] =>  
  
  }  
}
```

With Visitors



# Solving the Expression Problem in Scala

Adding new functionality.

```
trait PrefixNotationForExpressions extends ExtendedExpressions {  
  object PrefixNotationVisitor extends super.TVisitor[String] {  
    this: Visitor[String] =>  
  
    def visitConstant(v: Double): String = v.toString + " "  
  
    def visitAdd(l: Expression, r: Expression): String =  
      "+ " + l.accept(this) + r.accept(this)  
  
    def visitMult(l: Expression, r: Expression): String =  
      "* " + l.accept(this) + r.accept(this)  
  
  }  
}
```