Visitor Pattern

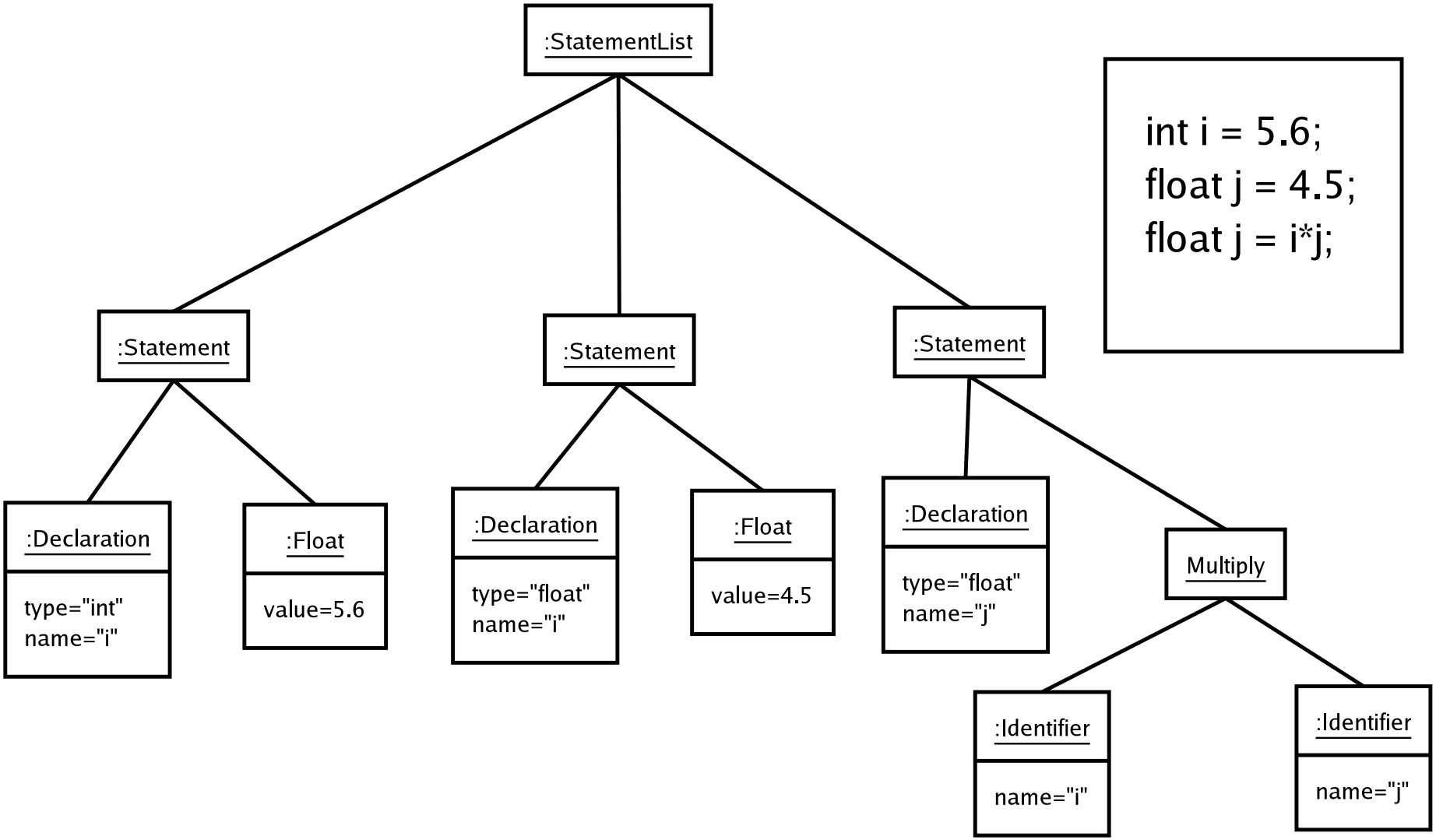
Marc Provost
McGill University
marc.provost@mail.mcgill.ca

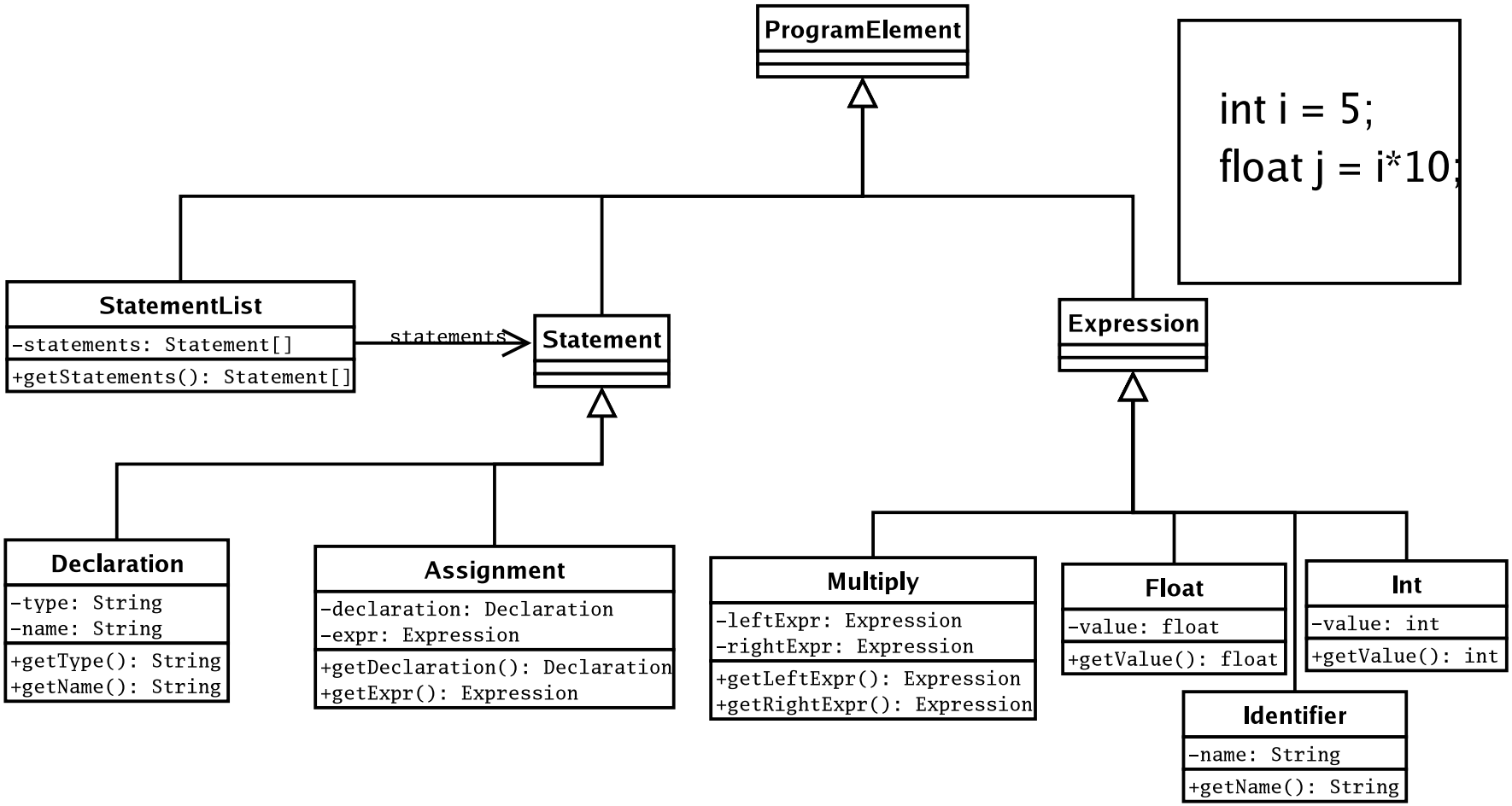
April 1, 2005

Visitor Pattern

- Purpose : encapsulate an operation you want to perform on the elements of a data structure

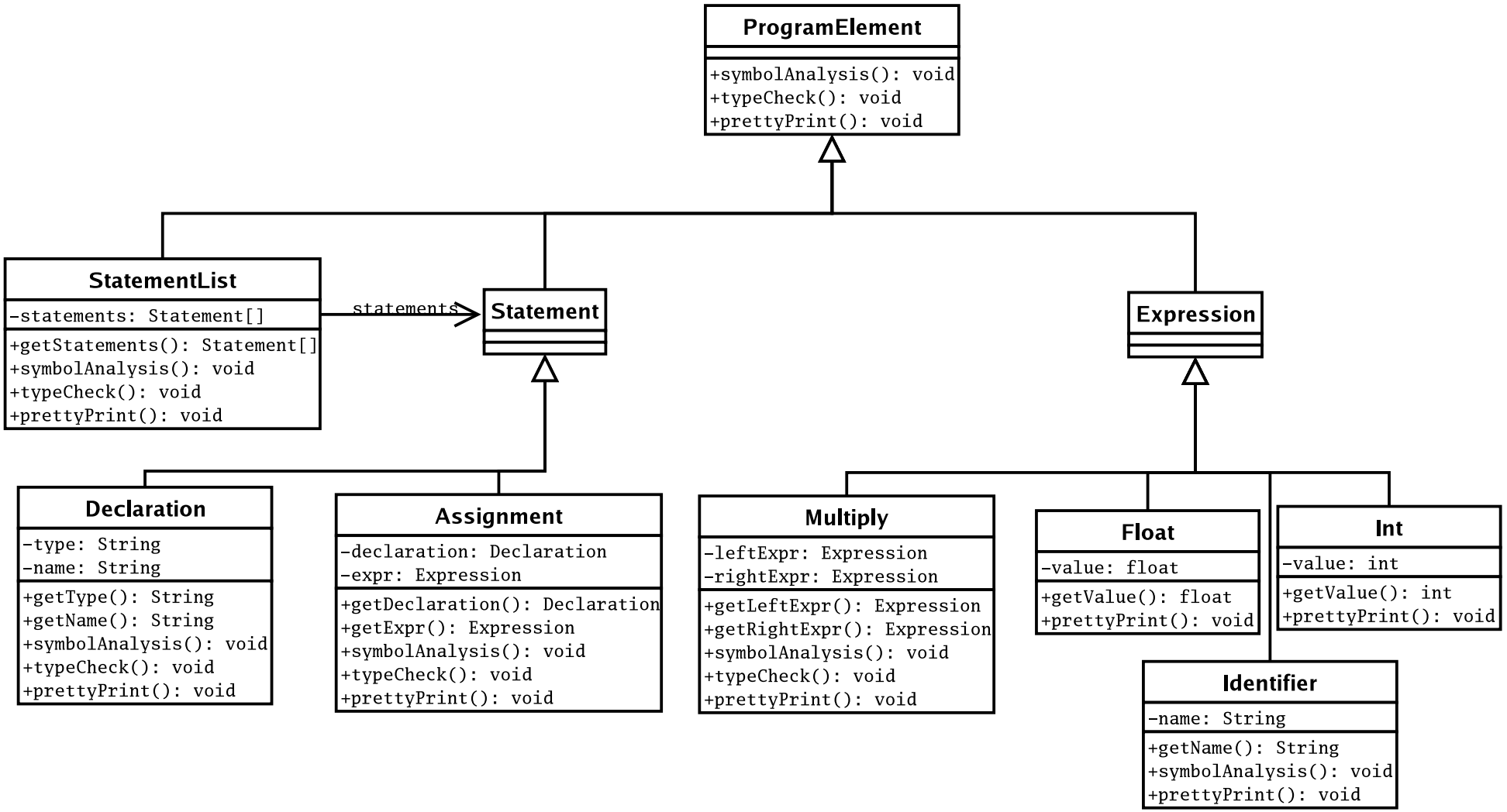
- Main Advantage : The classes implementing the data structure onto which the operation is applied are not modified


:StatementList
int i = 5.6;
float j = 4.5;
float j = i*j;
:Statement
:Statement
:Statement
:Declaration
type="int"
name="i"
:Float
value=5.6
:Declaration
type="float"
name="i"
:Float
value=4.5
:Declaration
type="float"
name="j"
Multiply
:Identifier
name="i"
:Identifier
name="j"


ProgramElement
int i = 5;
float j = i*10;
StatementList
-statements: Statement[]
+getStatements(): Statement[]
statements
Statement
Expression
Declaration
-type: String
-name: String
+getType(): String
+getName(): String
Assignment
-declaration: Declaration
-expr: Expression
+getDeclaration(): Declaration
+getExpr(): Expression
Multiply
-leftExpr: Expression
-rightExpr: Expression
+getLeftExpr(): Expression
+getRightExpr(): Expression
Float
-value: float
+getValue(): float
Int
-value: int
+getValue(): int
Identifier
-name: String
+getName(): String

Adding new operations

- Various operations need to be performed: Symbol Analysis, TypeChecking, PrettyPrint
- A different action need to be performed for each type of node.
- Intuitive Solution: Add a method to every class of the AST data structure


ProgramElement
+symbolAnalysis(): void
+typeCheck(): void
+prettyPrint(): void
StatementList
-statements: Statement[]
+getStatements(): Statement[]
+symbolAnalysis(): void
+typeCheck(): void
+prettyPrint(): void
statements
Statement
Expression
Declaration
-type: String
-name: String
+getType(): String
+getName(): String
+symbolAnalysis(): void
+typeCheck(): void
+prettyPrint(): void
Assignment
-declaration: Declaration
-expr: Expression
+getDeclaration(): Declaration
+getExpr(): Expression
+symbolAnalysis(): void
+typeCheck(): void
+prettyPrint(): void
Multiply
-leftExpr: Expression
-rightExpr: Expression
+getLeftExpr(): Expression
+getRightExpr(): Expression
+symbolAnalysis(): void
+typeCheck(): void
+prettyPrint(): void
Float
-value: float
+getValue(): float
+prettyPrint(): void
Int
-value: int
+getValue(): int
+prettyPrint(): void
Identifier
-name: String
+getName(): String
+symbolAnalysis(): void
+prettyPrint(): void

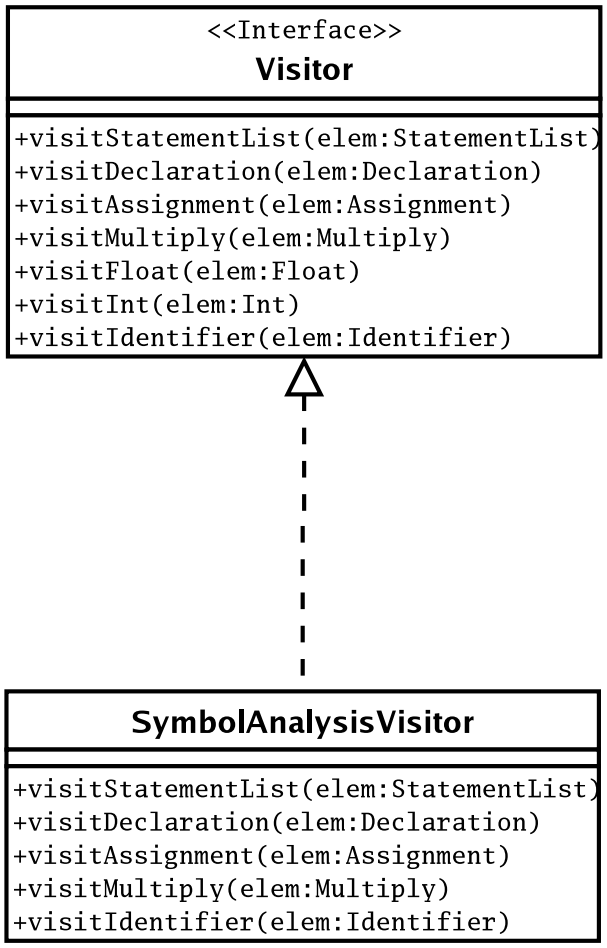
```
#Pseudo-Code implementing symbol analysis
#function of the class 'Declare'
def symbolAnalysis(self):
  #assume access to a global symbol hash table
  if symbolTable.has_key(self.__name):
    raise AlreadyDeclaredError(...)
  symbolTable[self.__name] = self

#function of the class 'Multiply'
def symbolAnalysis(self):
  self.__leftExpr.symbolAnalysis()
  self.__rightExpr.symbolAnalysis()

#function of the class 'Identifier'
def symbolAnalysis(self):
  if not symbolTable.has_key(self.__name):
    raise UndeclaredIdentifierError(...)
```

Problems with this approach

- Classes 'polluted' with several methods
- Implementation of an algorithm spread over all classes
- Must use global variables or arguments passed by reference




```
#sample code for 'StatementList'
def accept(self, v):
  for statement in self.__statements:
    statement.accept(v)
  v.visitStatementList(self)
```

```
#sample code for 'Identifier'
def accept(self, v):
  v.visitIdentifier(self)
```

```
#sample code for 'SymbolAnalysisVisitor'
def visitIdentifier(self, elem):
  if not self.symbolTable.has_key(elem.getName()):
    raise UndeclaredIdentifierError(...)
```

Advantages/Disadvantages

- Algorithm is now located in a single class. All variables needed to execute the algorithm are also in the class. No need for global variables anymore (or variables passed by reference).
- AST class structure was not modified!
- Easy to add new operations.
- A visitor can iterate over elements which are not sharing a common parent class.
- However, if a new subtype of ProgramElement is added, all the visitors must be modified.

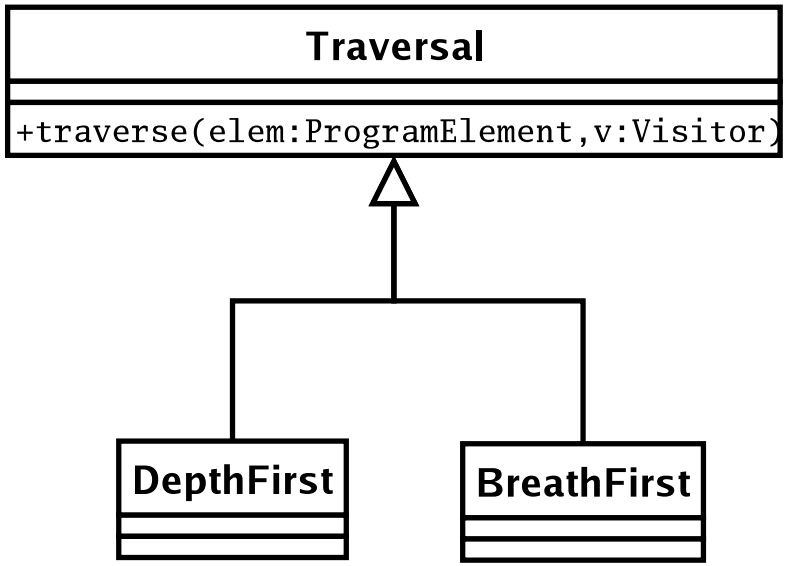
- For instance, we might want to add an 'Addition' node. This would require a new function 'visitAddition' in each visitor.

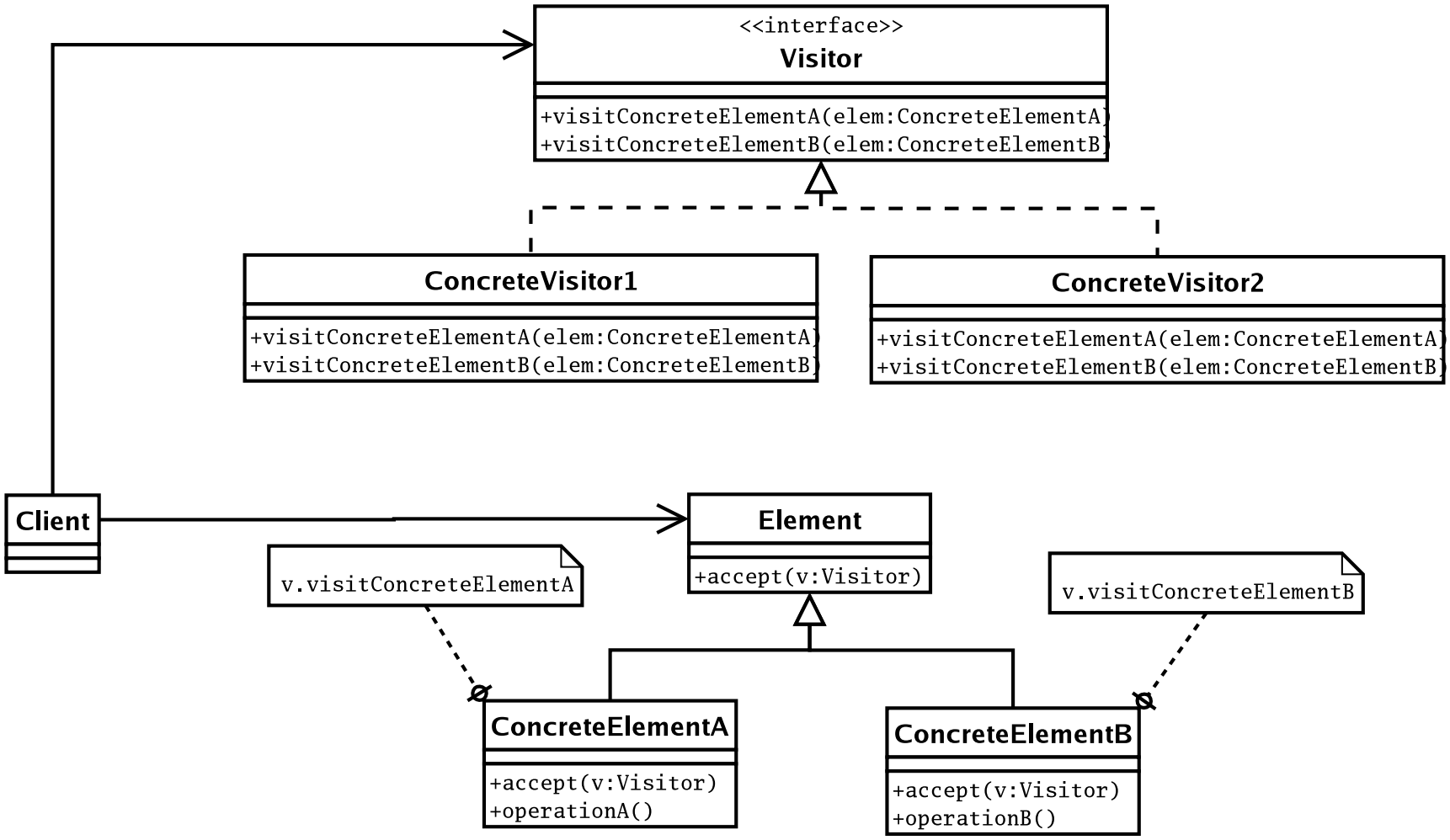
- Encapsulation could be broken if a visitor needs to access an element internal state.

Who should traverse Composite Elements?

- In the previous example: The composite element itself (StatementList).
  - This works if all the visitors need to visit the elements in the same order.
- To allow different traversal orders, the traversal could be in the visitors.
  - This would allow each visitor to use a specific traversal (e.g. Breath First, Depth First).
  - However, a lot of repeated code...
- Or use an external class..

External Classes for traversal




<<interface>>
Visitor
+visitConcreteElementA(elem:ConcreteElementA)
+visitConcreteElementB(elem:ConcreteElementB)
ConcreteVisitor1
+visitConcreteElementA(elem:ConcreteElementA)
+visitConcreteElementB(elem:ConcreteElementB)
ConcreteVisitor2
+visitConcreteElementA(elem:ConcreteElementA)
+visitConcreteElementB(elem:ConcreteElementB)
Client
Element
+accept(v:Visitor)
v.visitConcreteElementA
v.visitConcreteElementB
ConcreteElementA
+accept(v:Visitor)
+operationA()
ConcreteElementB
+accept(v:Visitor)
+operationB()