Some Language Engineering patterns

Tijs van der Storm
(storm@cwi.nl / @tvdstorm)

CWI
Centrum Wiskunde & Informatica

University of Amsterdam

Recap

- Parsing: turns text into tree
- Grammars *describe* syntax
- Generate parser from grammar
- Generated code *creates* AST nodes
- Abstract Syntax Tree: tree without syntactic noise (layout, comments, keywords, ...)

Grammars are recursive

```
expr   : '+' expr
       | '-' expr
       | '!' expr
       | expr '*' expr
       | expr '/' expr
       | expr '+' expr
       | expr '-' expr
       | expr EQ expr
       | expr NEQ expr
       | expr '>' expr
       | expr '<' expr
       | expr GEQ expr
       | expr LEQ expr
       | expr AND expr
       | expr OR expr
       ;
```

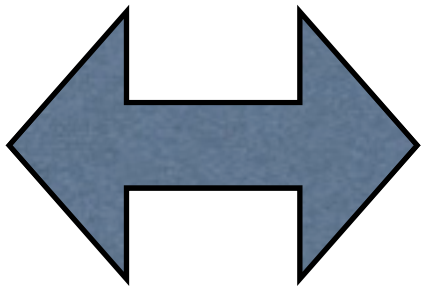
a type (syntactic category)

variants of the type

arguments (children) of each variant

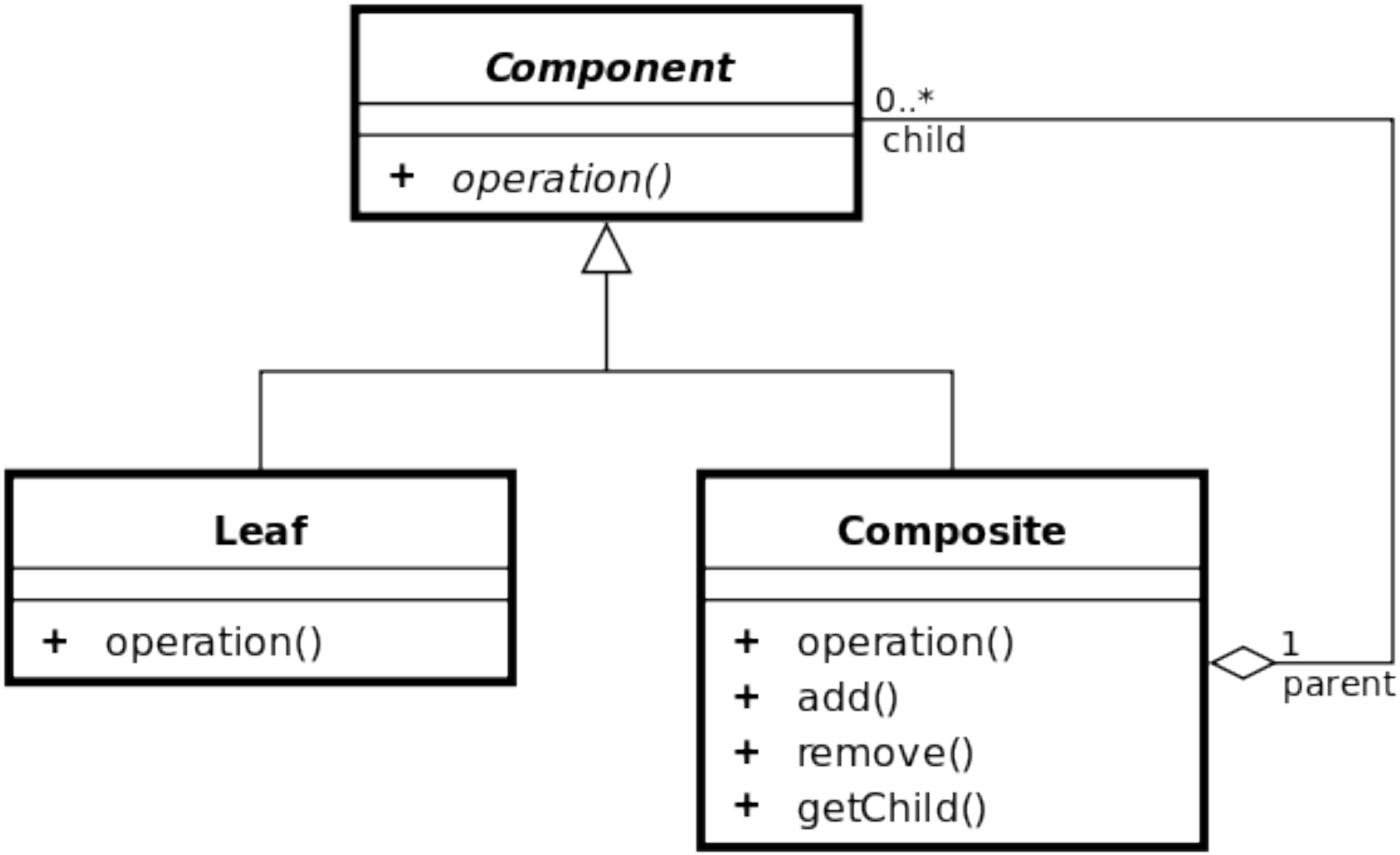
Algebraic data type

```
expr  : '+' expr
      | '-' expr
      | '!' expr
      | expr '*' expr
      | expr '/' expr
      | expr '+' expr
      | expr '-' expr
      | expr EQ expr
      | expr NEQ expr
      | expr '>' expr
      | expr '<' expr
      | expr GEQ expr
      | expr LEQ expr
      | expr AND expr
      | expr OR expr
      ;
```

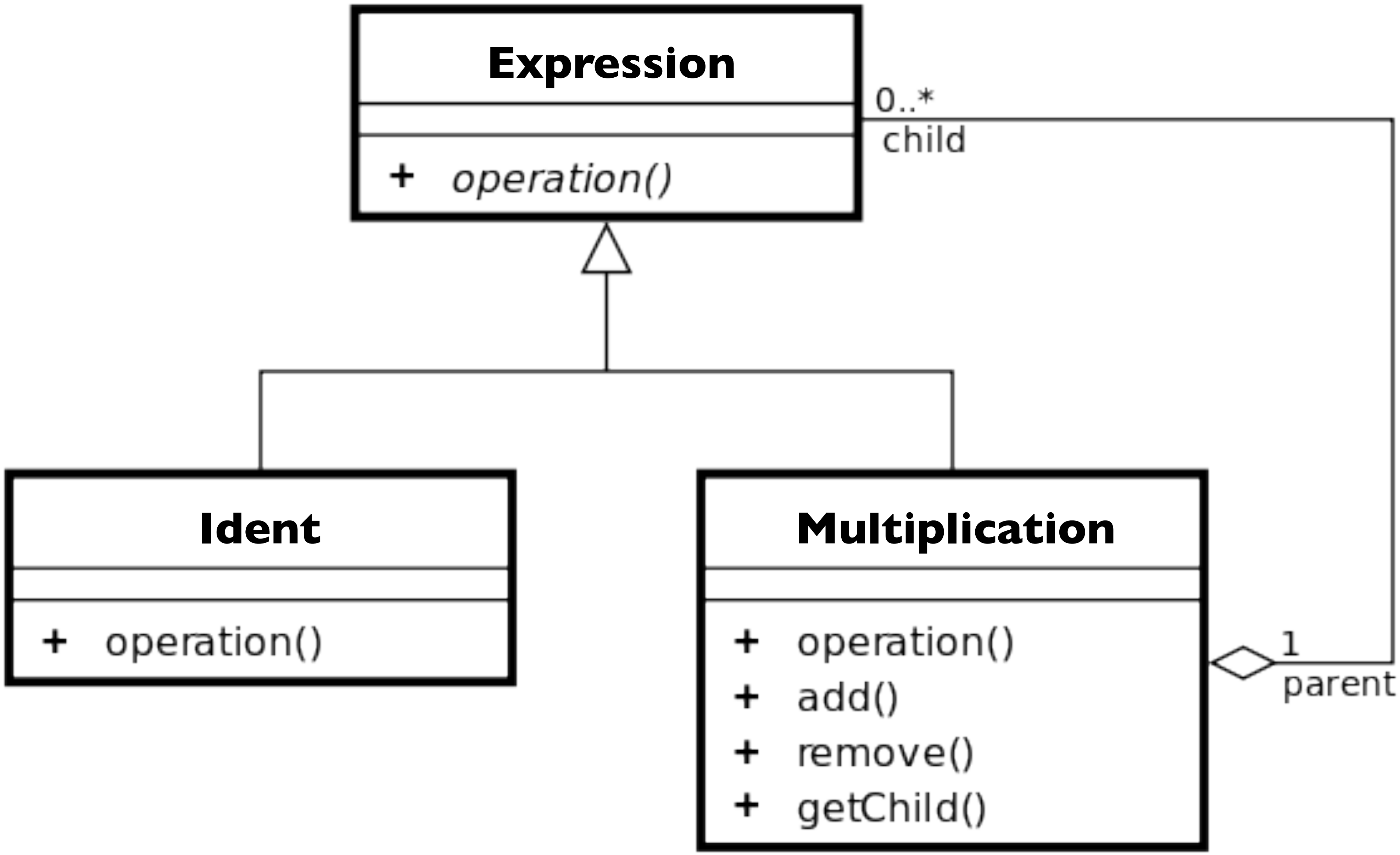


```
data Expr
  = not(Expr arg)
  | mul(Expr lhs, Expr rhs)
  | div(Expr lhs, Expr rhs)
  | add(Expr lhs, Expr rhs)
  | sub(Expr lhs, Expr rhs)
  | lt(Expr lhs, Expr rhs)
  | leq(Expr lhs, Expr rhs)
  | gt(Expr lhs, Expr rhs)
  | geq(Expr lhs, Expr rhs)
  | eq(Expr lhs, Expr rhs)
  | neq(Expr lhs, Expr rhs)
  | and(Expr lhs, Expr rhs)
  | or(Expr lhs, Expr rhs)
  ;
```

OO version: composite



OO version: composite



Expressions

Abstract class represents syntactic category

```
public abstract class Expr {

}
```

Expression variants

```
public class Mul extends Expr {
   private final Expr lhs;
   private final Expr rhs;

   public Mul(Expr lhs, Expr rhs) {
      this.lhs = lhs;
      this.rhs = rhs;
   }

   public Expr getLhs() {
      return lhs;
   }

   public Expr getRhs() {
      return rhs;
   }
}
```

Contains Exprs

Extends Expr type

Expression variants

```
public class Add extends Expr {
   private final Expr lhs;
   private final Expr rhs;

   public Add(Expr lhs, Expr rhs) {
      this.lhs = lhs;
      this.rhs = rhs;
   }

   public Expr getLhs() {
      return lhs;
   }

   public Expr getRhs() {
      return rhs;
   }
}
```

Expression variants

```
public class Sub extends Expr {
   private final Expr lhs;
   private final Expr rhs;

   public Sub(Expr lhs, Expr rhs) {
      this.lhs = lhs;
      this.rhs = rhs;
   }

   public Expr getLhs() {
      return lhs;
   }

   public Expr getRhs() {
      return rhs;
   }
}
```

Expression variants

```
public class Div extends Expr {
   private final Expr lhs;
   private final Expr rhs;

   public Div(Expr lhs, Expr rhs) {
      this.lhs = lhs;
      this.rhs = rhs;
   }

   public Expr getLhs() {
      return lhs;
   }

   public Expr getRhs() {
      return rhs;
   }
}
```

Intermediate classes

```
public abstract class Binary extends Expr {
    private final Expr lhs, rhs;

    protected Binary(Expr lhs, Expr rhs) {
        this.lhs = lhs;
        this.rhs = rhs;
    }

    public Expr getLhs() {
        return lhs;
    }

    public Expr getRhs() {
        return rhs;
    }
}
```

```
public class Mul extends Binary {
   public Mul(Expr lhs, Expr rhs) {
      super(lhs, rhs);
   }
}
```

```
public class Add extends Binary {
   public Add(Expr lhs, Expr rhs) {
      super(lhs, rhs);
   }
}
```

```
public class Sub extends Binary {
   public Sub(Expr lhs, Expr rhs) {
      super(lhs, rhs);
   }
}
```

```
public class Div extends Binary {
   public Div(Expr lhs, Expr rhs) {
      super(lhs, rhs);
   }
}
```

Etc.

- All *expressions* are subclass of Expr
- Concrete classes represent *variants*
- Fields represent AST *children* in a typed way
- Extract intermediate classes to share code (e.g., Binary, Unary)
- Terminals represent *leaves* in the composite pattern (e.g., Ident, Int, String, Bool etc.)

Type checking expressions

- What are types?
- What is the type of an expression?
- When are types compatible?
- What does checking type correctness of expressions mean?
- What does it mean for statements?

Types = *semantic* categories

- Type analysis != syntactic analysis
- Types are semantic, not syntactic
- But there may be syntactic representation of types

Syntactically ok, but not semantically

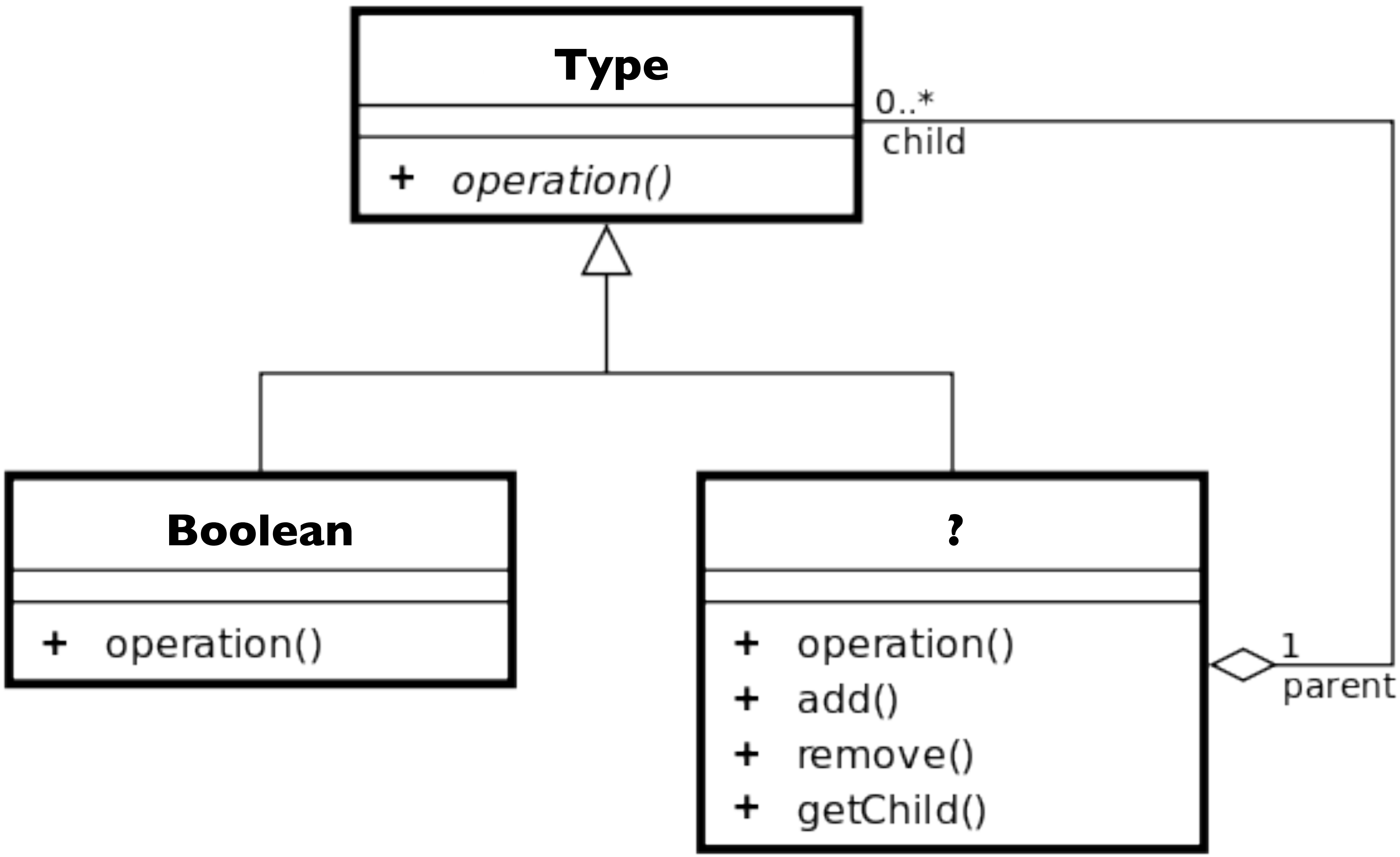
- 1 + “abc”
- 1 && true
- if (1 + 2) ...
- 3 == true
- true > false
- ...

Important: not a given! All depends on how *you* define your language...

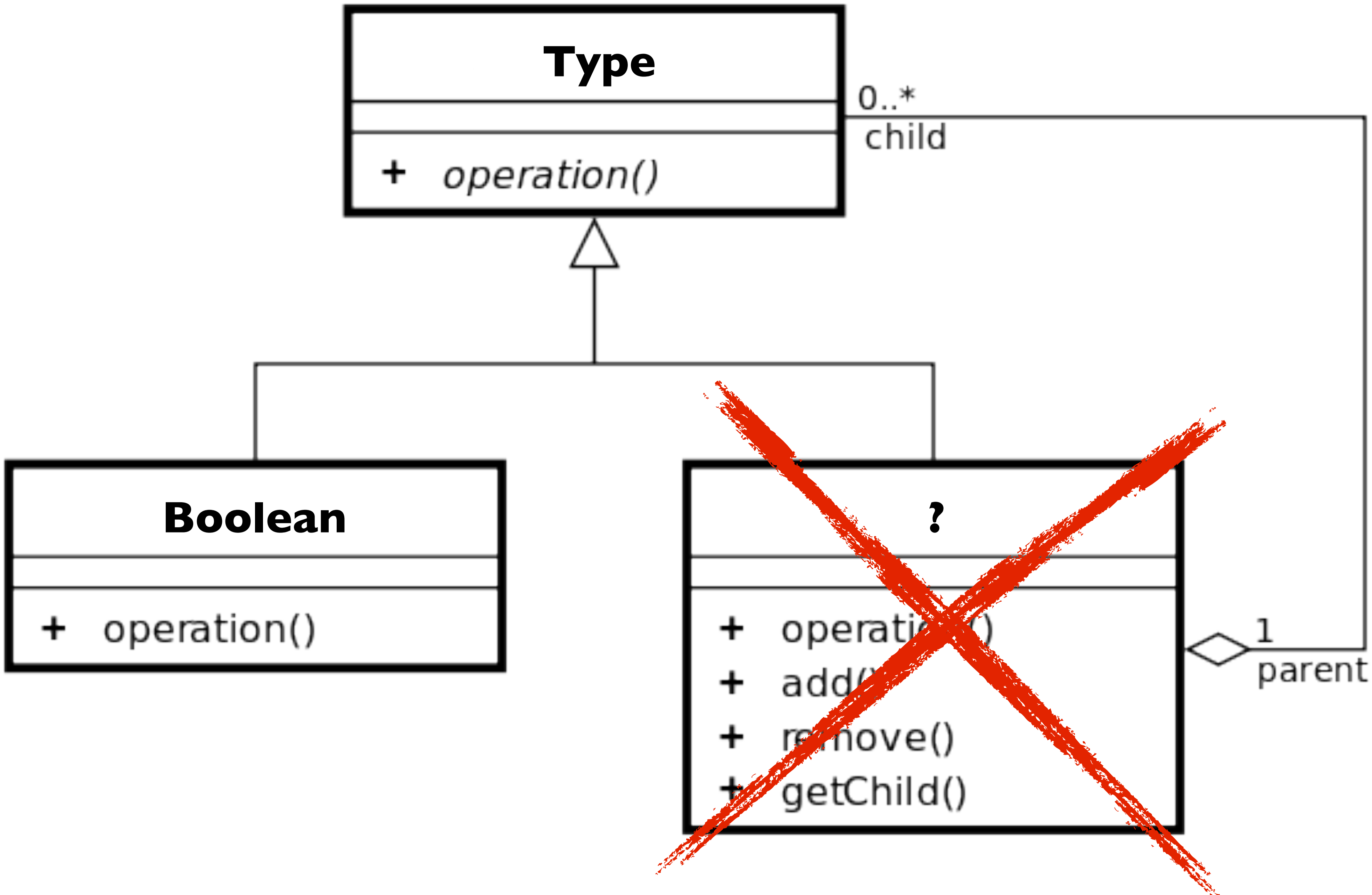
Expressions have types

- 1 + 2: Integer
- 1 + 3.0: Numeric (or Money? or Float?)
- true: Boolean
- a && b: Boolean
- ...

ASTs for Types



ASTs for Types



Type hierarchy

```
public abstract class Type {
}
```

```
public class Numeric extends Type {
}
```

```
public class Money extends Numeric {
}
```

```
public class Int extends Numeric {
}
```

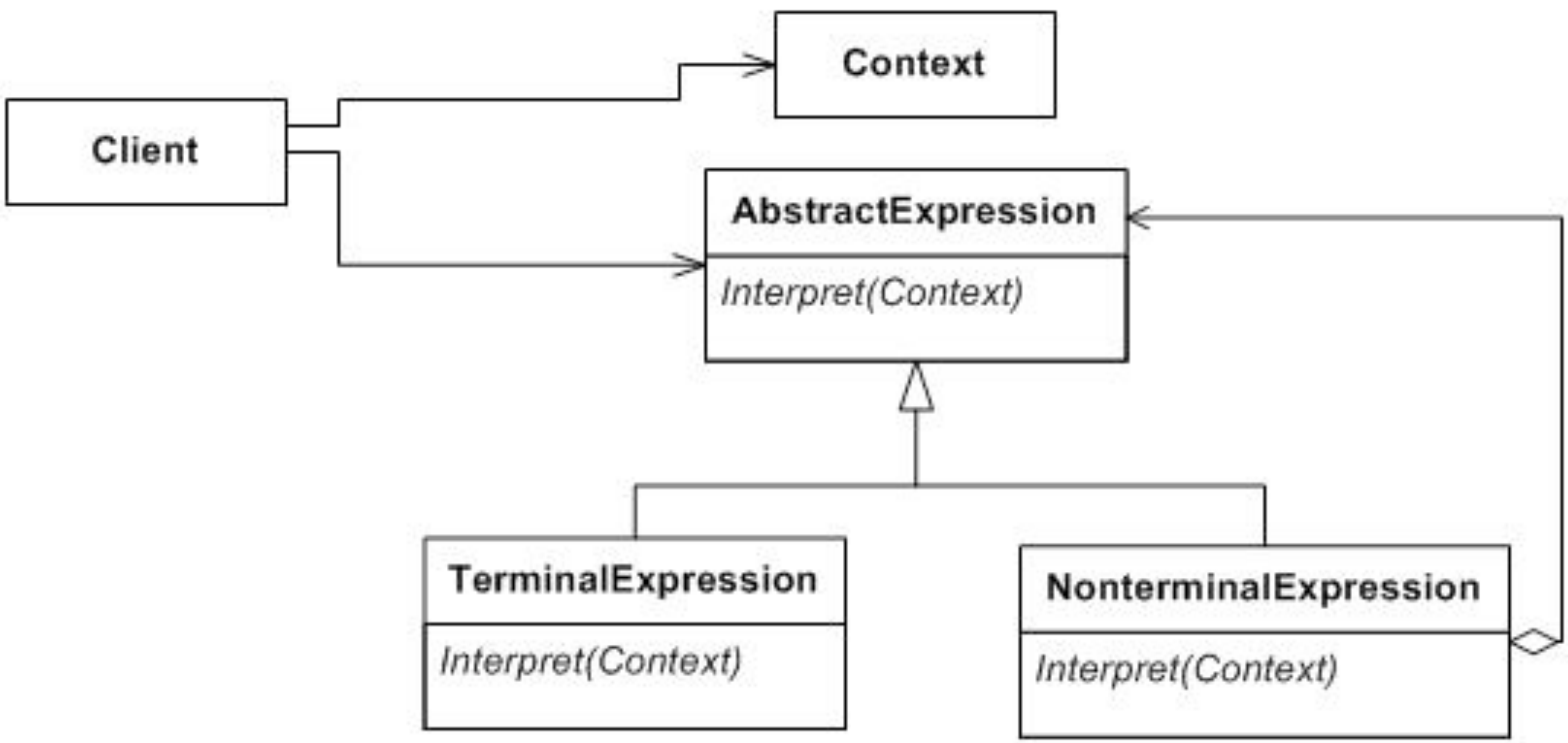
```
public class Bool extends Type {
}
```

```
public class Str extends Type {
}
```

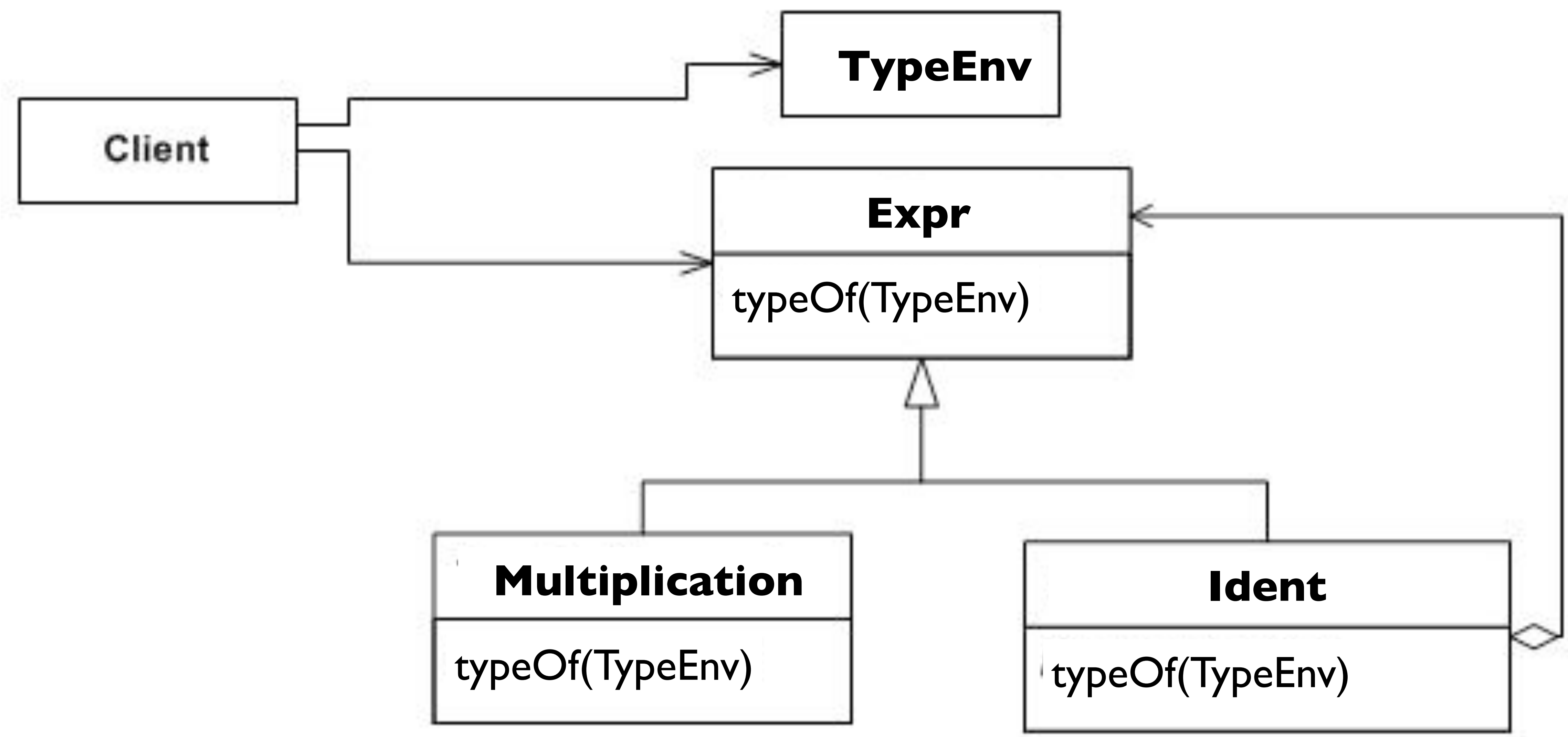
So now we can represent types

- How do we compute the type of an expression?

Interpreter pattern



Interpreter pattern



Type environments

- is map/table (e.g. java.util.Map<K,V>)
  - from *Identifiers*
  - to *Types*
- Used to lookup the type of an identifier
- *Declarations* update this table

Extend AST classes

```
public abstract class Expr implements ASTNode {

   public abstract Type typeOf(Map<Ident, Type> typeEnv);

}
```

Ensures that all Exprs have the typeOf method

Syntactic Bool
(a literal)

```
public class Bool extends Expr {
   private final Boolean value;

   @Override
   public Type typeOf(Map<Ident, Type> typeEnv) {
      return new org.uva.sea.ql.ast.types.Bool();
   }

   ...
}
```

Type Bool
(e.g. "boolean")

```java
public class Int extends Expr {

   @Override
   public Type typeOf(Map<Ident, Type> typeEnv) {
      return new org.uva.sea.ql.ast.types.Int();
   }

}
```

Overloading

```
public class Mul extends Binary {

   @Override
   public Type typeOf(Map<Ident, Type> typeEnv) {
      return new Numeric();
   }

}
```

NB: Numeric, (captures both Int and Money)

Numeric has no syntactic representation

Looking up identifiers

```
public class Ident extends Expr {

    @Override
    public Type typeOf(Map<Ident, Type> typeEnv) {
        if (typeEnv.containsKey(this)) {
            return typeEnv.get(this);
        }
        return new Error();
    }

}
```

Check if the variable is defined, if so, return its declared type.

Otherwise, return special “Error” type

Now

- We know how to represent types
- And how to get the type of an expression,
- But how to check if two types are compatible?

Type compatibility

- Equality is not enough: overloading
- We don't want if-then-else/switch with:
  - instanceof
  - enums
  - strings
- Can we encapsulate the logic of compatibility using classes?

Double dispatch

```java
public abstract class Type {
   public abstract boolean isCompatibleTo(Type t);

   public boolean isCompatibleToInt() { return false; }
   public boolean isCompatibleToNumeric() { return false; }
   public boolean isCompatibleToStr() { return false; }
   public boolean isCompatibleToBool() { return false; }
   public boolean isCompatibleToMoney() { return false; }
}
```

Subclasses override where needed

Booleans

```
public class Bool extends Type {

   @Override
   public boolean isCompatibleTo(Type t) {
      return t.isCompatibleToBool();
   }

   @Override
   public boolean isCompatibleToBool() {
      return true;
   }

}
```

Ask the argument if it's compatible to Bool

And yes, "I am compatible to Bool"...

Numeric

```
public class Numeric extends Type {
   public boolean isCompatibleTo(Type t) {
      return t.isCompatibleToNumeric();
   }

   public boolean isCompatibleToInt() {
      return true;
   }

   public boolean isCompatibleToMoney() {
      return true;
   }

   public boolean isCompatibleToNumeric() {
      return true;
   }
}
```

Money

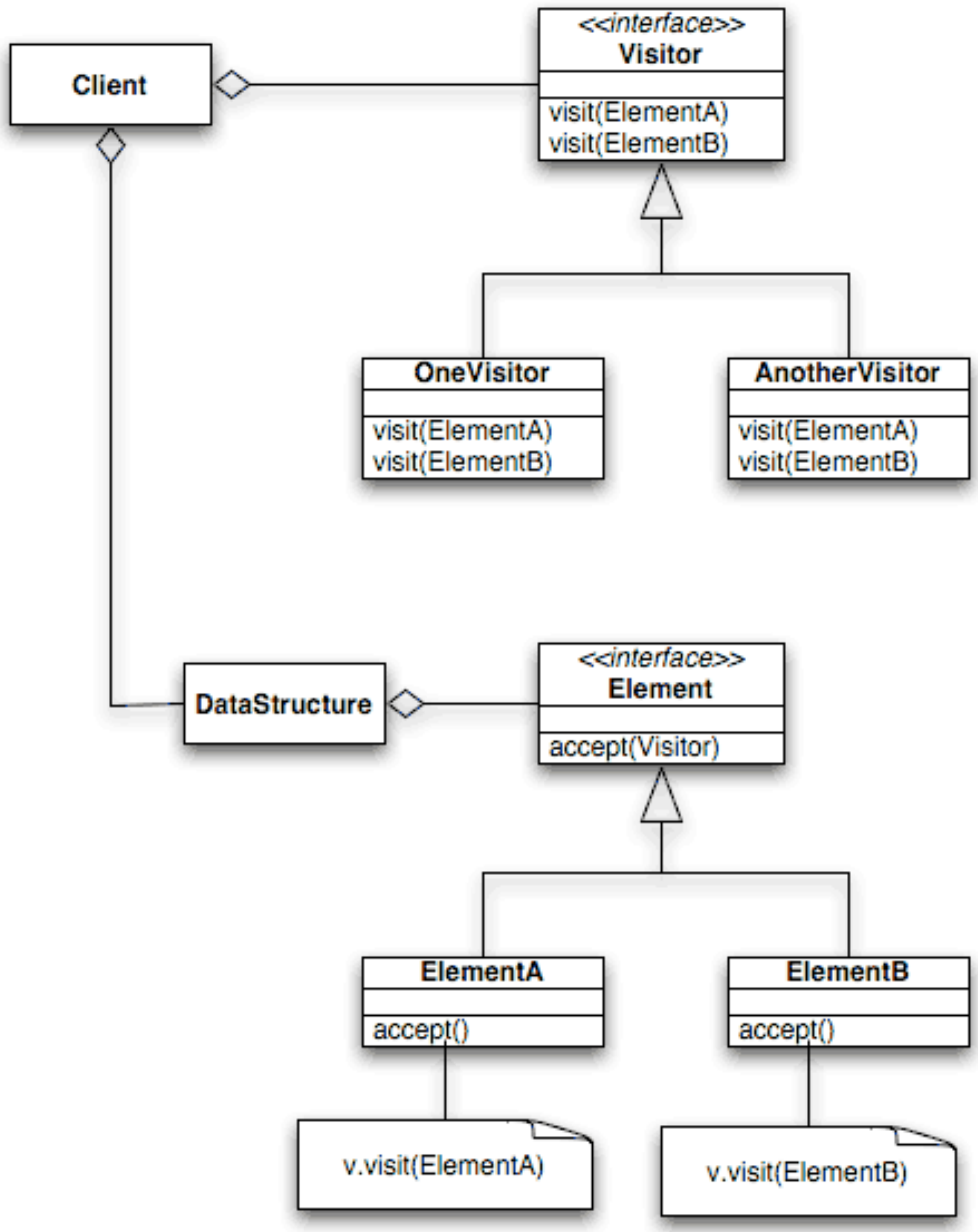
```
public class Money extends Numeric {

   @Override
   public boolean isCompatibleTo(Type t) {
      return t.isCompatibleToMoney();
   }

}
```

The rest is inherited from Numeric...

So, now:

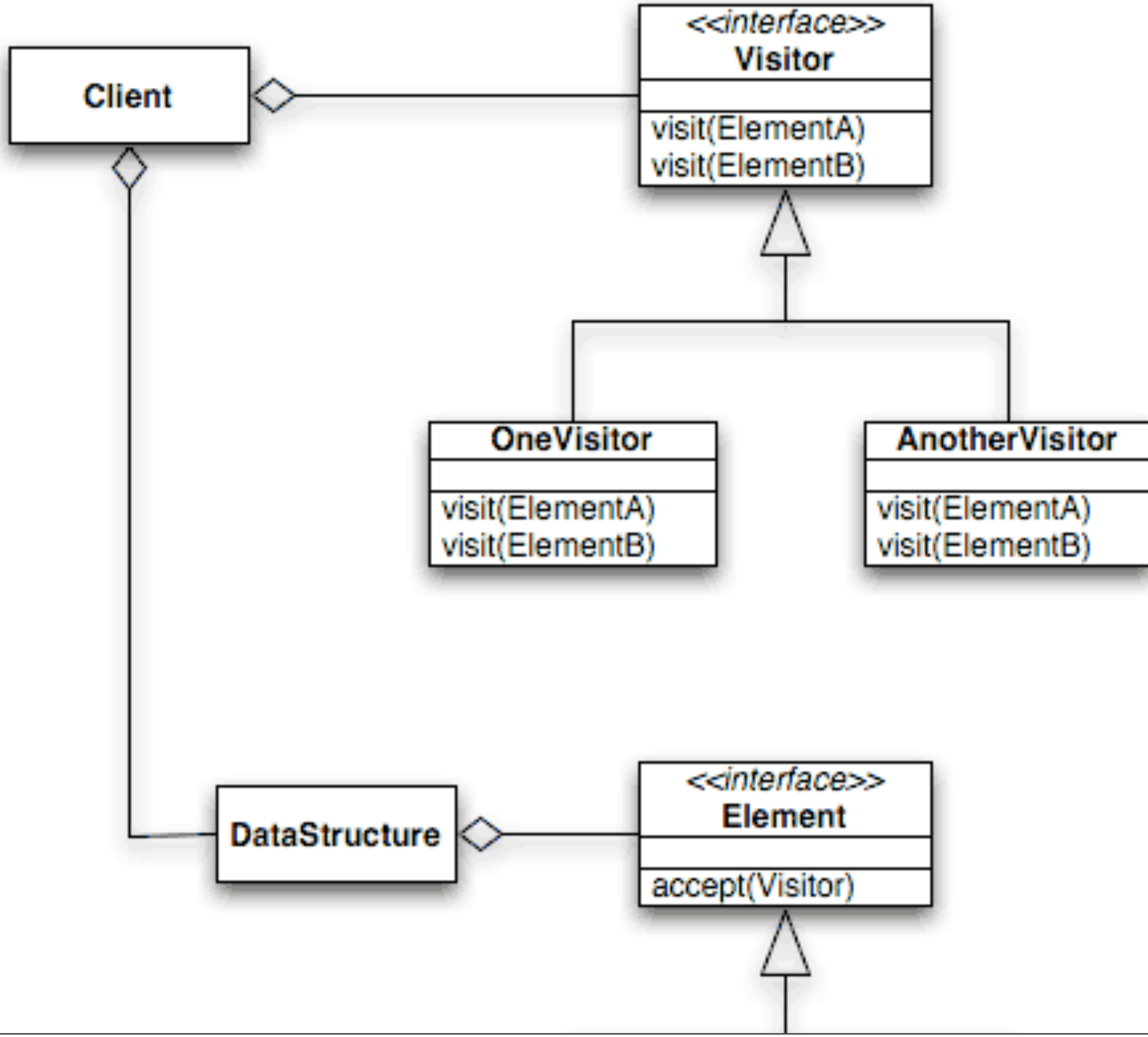
- We can represent types
- We can compute types of expressions
- We can compute compatibility of types
- But how do we get a type *checker*?

Visitor pattern

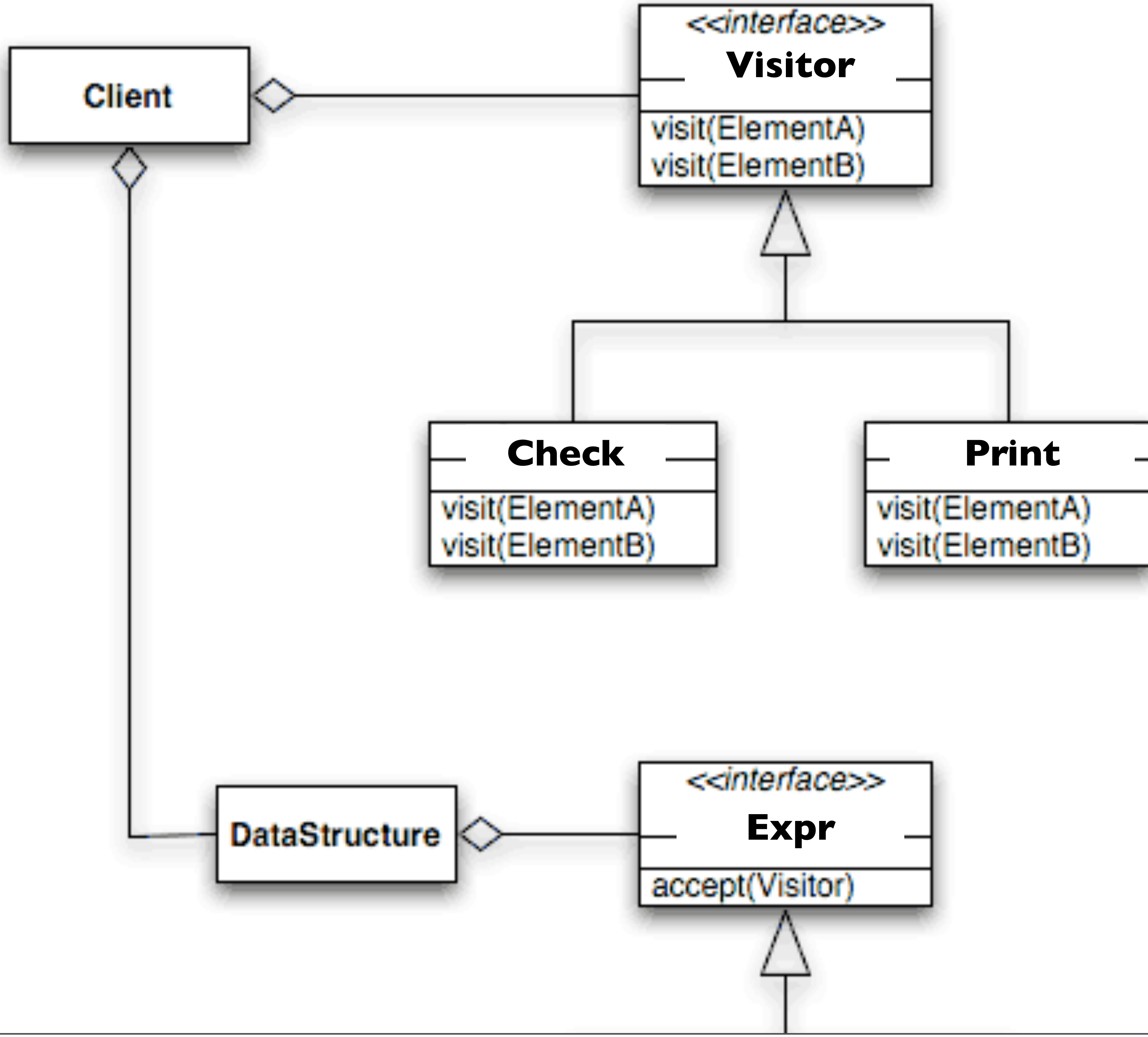


Visitor pattern

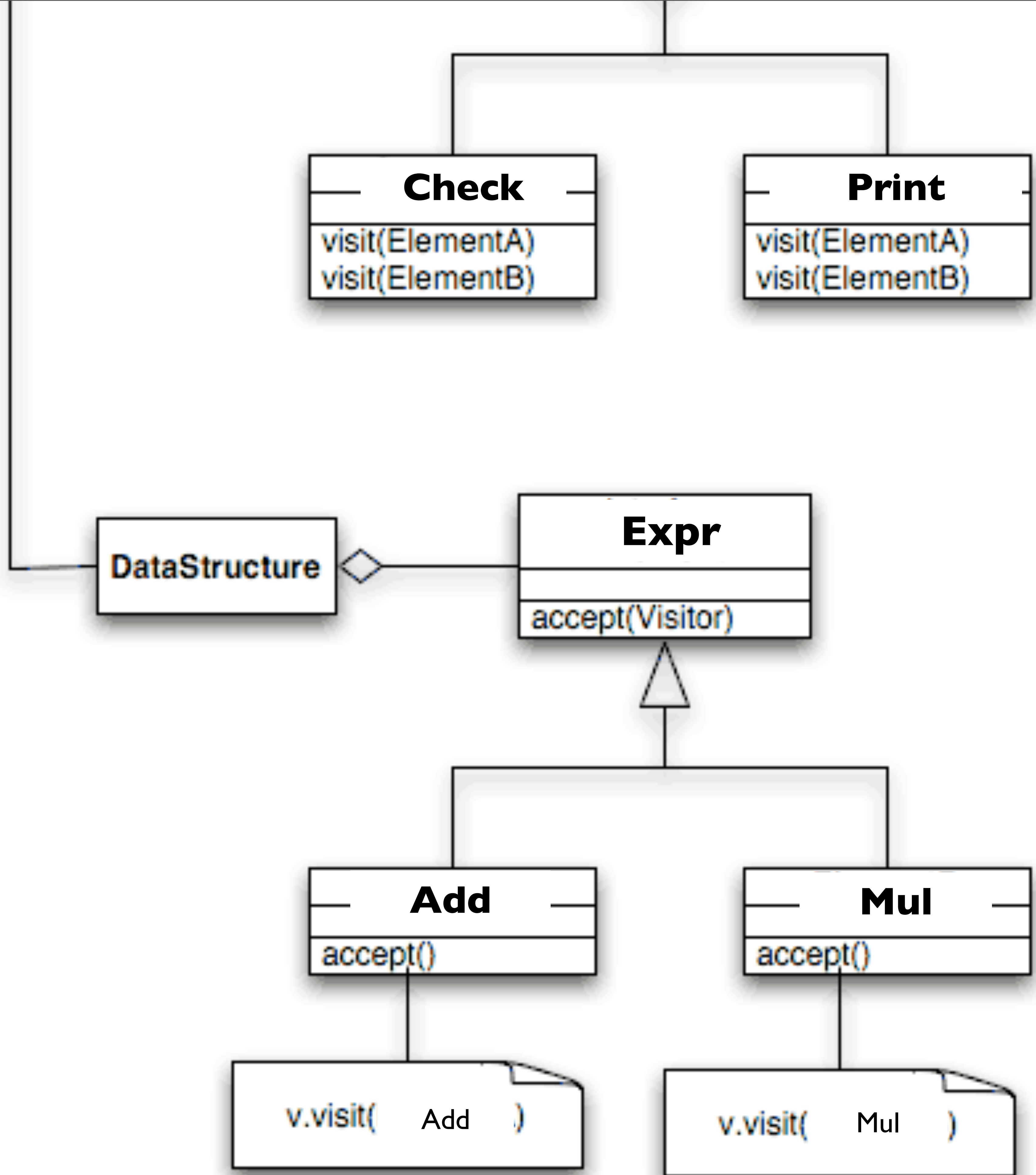
- Allows separation of data structure and traversal
- IOW: can define multiple operations/ interpretations on AST, without changing the AST classes
- Essentially double dispatch



Client
<<interface>>
Visitor
visit(ElementA)
visit(ElementB)
OneVisitor
visit(ElementA)
visit(ElementB)
AnotherVisitor
visit(ElementA)
visit(ElementB)
DataStructure
<<interface>>
Element
accept(Visitor)



<<interface>>
Visitor
visit(ElementA)
visit(ElementB)
Client
Check
visit(ElementA)
visit(ElementB)
Print
visit(ElementA)
visit(ElementB)
DataStructure
<<interface>>
Expr
accept(Visitor)


Check
visit(ElementA)
visit(ElementB)
Print
visit(ElementA)
visit(ElementB)
DataStructure
Expr
accept(Visitor)
Add
accept()
Mul
accept()
v.visit(Add)
v.visit(Mul)

```java
public interface Visitor<T> {
   T visit(Add ast);
   T visit(And ast);
   T visit(Div ast);
   T visit(Eq ast);
   T visit(GEq ast);
   T visit(GT ast);
   T visit(Ident ast);
   T visit(Int ast);
   T visit(LEq ast);
   T visit(LT ast);
   T visit(Mul ast);
   T visit(Neg ast);
   T visit(NEq ast);
   T visit(Not ast);
   T visit(Or ast);
   T visit(Pos ast);
   T visit(Sub ast);
   T visit(Bool bool);
}
```

Type parameter allows us to return different things in different visitors

```java
public abstract class Expr {
   public abstract <T> T accept(Visitor<T> visitor);
}
```

Accepting visitors

```
public class Add extends Binary {

  @Override
  public <T> T accept(Visitor<T> visitor) {
    return visitor.visit(this);
  }

}
```

```
}
```

```
}
```

```
}
```

```
}
```

```
}
```

Etc.

```java
public class CheckExpr implements Visitor<Boolean> {

   private final Map<Ident, Type> typeEnv;
   private final List<Message> messages;

   private CheckExpr(Map<Ident, Type> tenv, List<Message> messages) {
      this.typeEnv = tenv;
      this.messages = messages;
   }

   public static boolean check(Expr expr,
              Map<Ident, Type> typeEnv, List<Message> errs) {
      CheckExpr check = new CheckExpr(typeEnv, errs);
      return expr.accept(check);
   }
}
```

```java
public Boolean visit(Add ast) {
   boolean checkLhs = ast.getLhs().accept(this);
   boolean checkRhs = ast.getRhs().accept(this);

   if (!(checkLhs && checkRhs)) {
      return false;
   }

   Type lhsType = ast.getLhs().typeOf(typeEnv);
   Type rhsType = ast.getRhs().typeOf(typeEnv);

   if (!(lhsType.isCompatibleToNumeric()
         && rhsType.isCompatibleToNumeric())) {
      addError(ast, "invalid type for +");
      return false;
   }

   return true;
}
```

check lhs and rhs

return false if there were type errors

get types of lhs and rhs

check required types for "+"

no type errors

Statements

- Now we can represent types
- Compute types of expressions
- Compute type compatibility
- Check for incorrectly typed expressions
- But what about the rest of our language?

Another visitor

```
public interface Visitor {
   void visit(Computed stat);
   void visit(Answerable stat);
   void visit(IfThen stat);
   void visit(IfThenElse stat);
   void visit(Block stat);
}
```

```
public class CheckStat implements Visitor {
   public void visit(Computed stat) {
      checkName(stat, stat.getExpr().typeOf(typeEnv));
      checkExpr(stat.getExpr());
   }
   public void visit(Answerable stat) {
      checkName(stat, stat.getType());
   }
   public void visit(IfThen stat) {
      checkCondition(stat);
      stat.getBody().accept(this);
   }
   public void visit(IfThenElse stat) {
      checkCondition(stat);
      stat.getBody().accept(this);
      stat.getElseBody().accept(this);
   }
   public void visit(Block stat) {
      for (Stat s: stat.getStats()) {
         s.accept(this);
      }
   }
}
```

NB: statement checker depends on expression checker.

Some advice...

- Use Composite for ASTs
- Use Visitor for traversal of ASTs
  - (or Interpreter)
- Don't throw exceptions for type errors
  - (think about why)
- Separate typeOf from type checking
- Separate statement checking from expression checking