Problemstellung „Backend:“ Knotentyp- oder Algorithmen-orientierte Struktur?

Mehrere Baumtraversierungen nötig

- Baumausgabe (Serialisierung)
- Symboltabellenaufbau
- Semantische Prüfung
- Memory-Adressen berechnen
- Assemblercode generieren

Jeder Algorithmus Knotentyp-spezifisch

Bei jeder Baumtraversierung muss eine Knotentyp-spezifische Verarbeitung durchgeführt werden

Standard OO-Technik: Methoden in der Klassendefinition

- Jeder Knotentyp ist eine Subklasse von *Absyn*: *ProcDec*, *ArrayTy*, *OpExp*, *IfStm*, ...
- Insgesamt 27 Knotentypen
- Knotentyp-spezifischen Methoden in der Klassendefinition → pro Baumtraversierung eine Methode

Problem

- Jeder Algorithmus ist auf 27 Klassendefinitionen verteilt.
- Programmierung umständlich
- Programm unübersichtlich

Lösung: Entwurfsmuster „Visitor“

Merkmale

- Algorithmen-orientierte Struktur
- *Visitor*-Objekt repräsentiert eine Algorithmus
- **visit** verarbeitet einen Knoten

Java-Problem

- n Algorithmen, m Knotentypen $\rightarrow$ $n \times m$ visit-Methoden
- kein „Multipolymorphismus“ zur dynamischen Zuordnung anhand von Knotentyp **und** Algorithmus

Lösung

- Zuordnung des Algorithmus dynamisch mit Polymorphismus
- Zuordnung des Knotentyps statisch durch Überladungsauflösung
- Baumabstieg erfordert indirekte Rekursion über *accept*-Methode

Beispiel: Lösung ohne Visitor mit 3 Algorithmen

```
public abstract class AST {
  public abstract void prettyPrint();
  public abstract Type typeCheck();
  public abstract void codeGen();
}
```

```
class BinOpExpr extends AST {
  public void prettyPrint () { ... }
  public Type typeCheck ()   { ... }
  public void codeGen ()     { ... }
}

class IntConst extends AST {
  public void prettyPrint () { ... }
  public Type typeCheck ()   { ... }
  public void codeGen ()     { ... }
}
```

Visitor 1: Algorithmus in nur einer Methode

```
public abstract class Visitor {
  abstract void visit (AST t);
}
```

```
public class PrettyPrintVisitor extends Visitor {
  void visit (AST t){
    if      (t instanceof BinOpExpr) { ... }
    else if (t instanceof IntConst)  { ... }
    else if (t instanceof Variable)  { ... }
  }
}
```

Fallunterscheidung nach Knotentyp zur Laufzeit: unschön und ineffizient!

Visitor 2: Pro Knotentyp eine Methode

```
public abstract class Visitor {
  abstract void visitBinOpExpr (BinOpExpr e);
  abstract void visitIntConst  (IntConst c);
  abstract void visitVariable  (Variable v);
}
```

```
public class PrettyPrintVisitor extends Visitor {
  void visitBinOpExpr (BinOpExpr e) { ... }
  void visitIntConst  (IntConst c)  { ... }
  void visitVariable  (Variable v)  { ... }
}
```

Visitor 3: Pro Knotentyp eine Überladung von `visit`

```
public abstract class Visitor {
  abstract void visit (BinOpExpr e);
  abstract void visit (IntConst c);
  abstract void visit (Variable v);
}
```

```
public class PrettyPrintVisitor extends Visitor {
  void visit (BinOpExpr e) { ... }
  void visit (IntConst c)  { ... }
  void visit (Variable v)  { ... }
}
```

Bei n Algorithmen und m Knotentypen: $n \times m$ `visit`-Methoden

accept - Verbindung von Knotenklasse und Visitor

```
public abstract class AST {
  public abstract void accept (Visitor v);
}
```

```
public class BinOpExpr extends AST {
  public void accept (Visitor v){
     v.visit(this);
  }
}
```

Welche visit-Methode?

- Knotentyp: Auflösung der Überladung zur Übersetzungszeit: `void visit(BinOpExpr)`
- Algorithmus: Zuordnung der Visitorklasse zur Laufzeit (Polymorphismus)

Beispiel für die Baumtraversierung

```
public class PrettyPrintVisitor extends Visitor {

  ...

  void visit (BinOpExpr e) {
    e.operand1.accept(this); // linken Operanden ausgeben
    System.out.print(e.operator.toString());
    e.operand2.accept(this); // rechten Operanden ausgeben
  }

  ...
}
```