Programmieren in Java

Vorlesung 06: Das Visitor Pattern

Prof. Dr. Peter Thiemann
(vertreten durch Luminous Fennell)

Albert-Ludwigs-Universität Freiburg, Germany

SS 2015

Executive Summary — Visitor Pattern

- Design-Pattern, dass bei **rekursiven Klassen** Verwendung findet
  - Es erlaubt **einfaches und modulares Hinzufügen von Operationen** für ein Datenmodell
  - **Erweitern** des Datenmodells ist schwieriger
  - Gegenspieler zum **Composite Pattern**, wo Operationen schwerer hinzuzufügen sind.
- Geeignete Modelle sind z.B.
  - Arithmetische/Boolesche/. . . Ausdrücke
  - Repräsentation von Programmen einer Programmiersprache (z.B. beim Compilerbau)
- Operationen werden als **Subklassen** eines Visitor-Interfaces per **Fallunterscheidung** implementiert
- Es genügt eine Methode im Datenmodell (accept) um eine Vielzahl von Visitor-basierten Operationen zu Unterstützen
- Laufendes Beispiel heute: **Arithmetische Ausdrücke**

Inhalt

Wiederholung: Arithmetische Ausdrücke

Arithmetische Ausdrücke

Datenmodell

Ein arithmetischer Ausdruck hat eine der folgenden Formen:

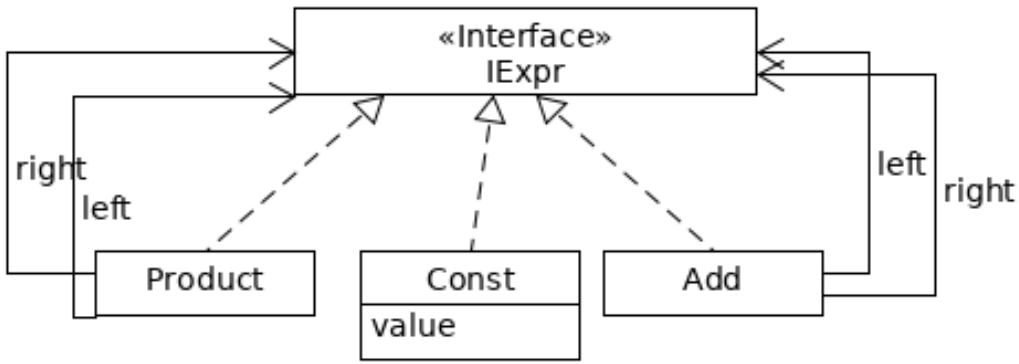
- eine Konstante (eine ganze Zahl),
  Beispiele: 0, 51, −42
- eine Summe von zwei arithmetischen Ausdrücken,
  Beispiele: 3+4, 17+4, 17+ (2+2)
- ein Produkt von zwei arithmetischen Ausdrücken,
  Beispiele: 3∗4, 2 ∗ (17+4), (2 ∗ 3) ∗ 4

Operationen für arithmetische Ausdrücke

- Berechnung des Werts: $\text{eval}(3 + 4) \rightsquigarrow 7$
- Berechnung seiner Größe: $\text{size}(3 + 4) \rightsquigarrow 3$

Datenmodell

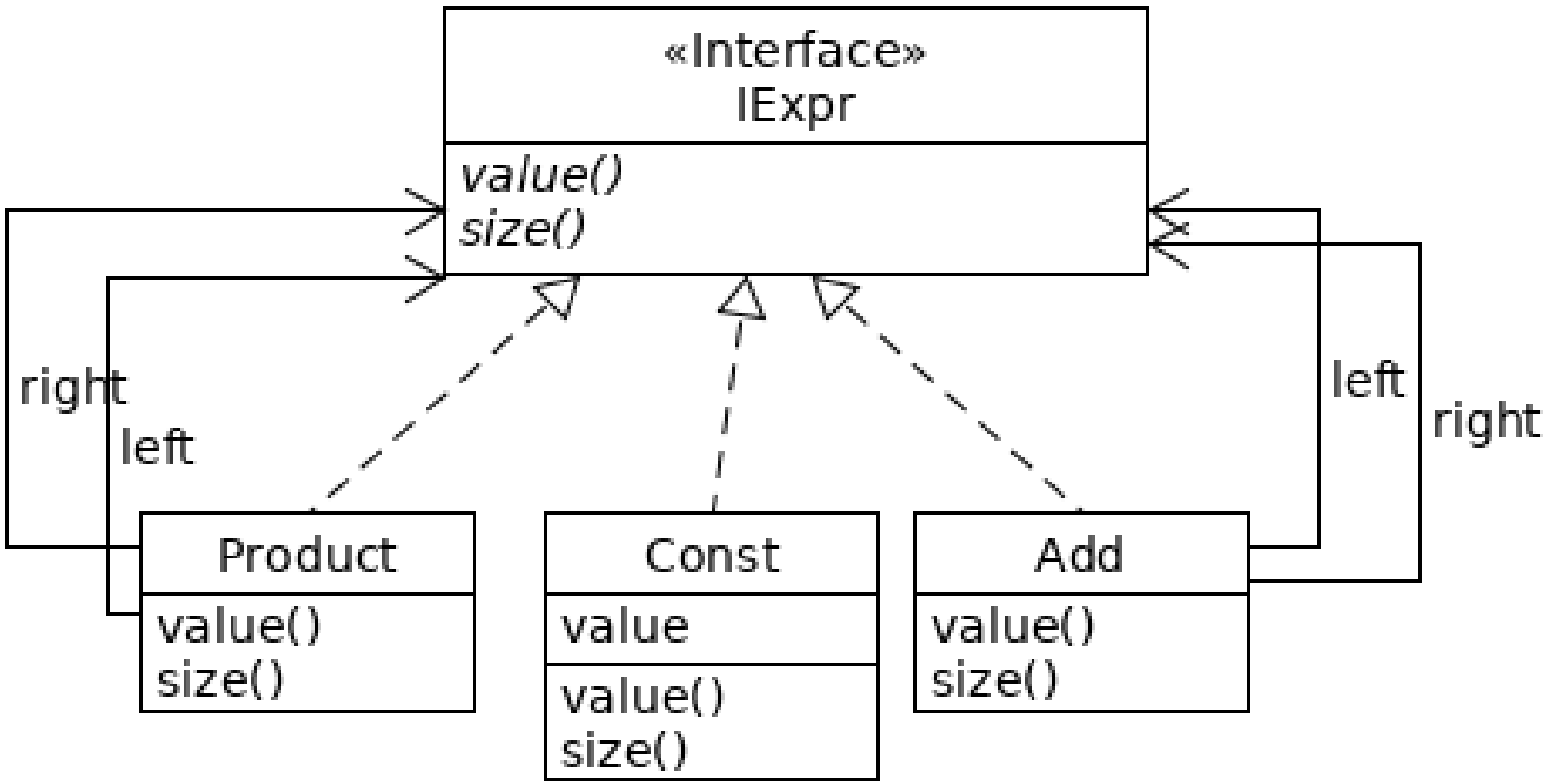
Arithmetische Ausdrücke sind
rekursive, unveränderliche Datenstrukturen



- Basisfall: Konstanten
- Rekursive Fälle: Addition und Produkt
- Hier: Zugriff auf Bestandteile durch **public final** Felder.

Operationen

Modellierung mit Composite Pattern



Static Imports

Erzeugen von Beispiel-Ausdrücken

Mit Java Konstruktoren

```
1  @Test
2   public void testEval() {
3     // (42 + (5 * 4)) * (2 + 1)
4     IExpr e = new Product(new Add(new Const(42), new Product(new Const(5),
5                         new Const(4))), new Add(new Const(2), new Const(1)));
6     assertEquals(168, e.eval());
7   }
```

Java's Syntax für Konstruktoraufrufe ist für das
Erstellen verschachtelter Strukturen sehr unleserlich.

Erzeugen von Beispiel-Ausdrücken

Mit statischen Factory-Methoden

```
1  @Test
2  public void testEvalStatic() {
3          // (42 + (5 * 4)) * (2 + 1)
4          IExpr e = product(add(42, product(5, 4)), add(2, 1));
5          assertEquals(168, e.eval());
6  }
```

Erzeugen von Beispiel-Ausdrücken

Mit statischen Factory-Methoden

```
1  @Test
2  public void testEvalStatic() {
3          // (42 + (5 * 4)) * (2 + 1)
4          IExpr e = product(add(42, product(5, 4)), add(2, 1));
5          assertEquals(168, e.eval());
6  }
```

- Die statischen Methoden product und add sind **statische Methoden** aus einer Klasse Expressions.
- Durch **statische Imports** kann der Klassenname beim Aufruf weggelassen werden

```
import composite.Expressions;
```

normaler import, erlaubt: Expressions.add(42, 5);

```
import static composite.Expressions.*;
```

statischer import, erlaubt: add(42,5)

Hinzufügen weiterer Operationen

Weitere Operationen für Arithmethische Ausdrücke

- prettyprint(IExpr)

  prettyprint(add(add(0, 5), 8)) $\rightsquigarrow$ "((0 + 5) + 8)"

Weitere Operationen für Arithmethische Ausdrücke

- prettyprint(IExpr)

  prettyprint(add(add(0, 5), 8)) $\rightsquigarrow$ "((0 + 5) + 8)"
- depth(IExpr)

  depth(add(add(0, 5), 8)) $\rightsquigarrow$ 3

Weitere Operationen für Arithmethische Ausdrücke

- prettyprint(IExpr)

  prettyprint(add(add(0, 5), 8)) $\rightsquigarrow$ "((0 + 5) + 8)"
- depth(IExpr)

  depth(add(add(0, 5), 8)) $\rightsquigarrow$ 3
- simplify()

  simplify(add(add(0, 5), 8)) $\rightsquigarrow$ add(5, 8)

Weitere Operationen für Arithmethische Ausdrücke

- prettyprint(IExpr)

  prettyprint(add(add(0, 5), 8)) $\rightsquigarrow$ "((0 + 5) + 8)"
- depth(IExpr)

  depth(add(add(0, 5), 8)) $\rightsquigarrow$ 3
- simplify()

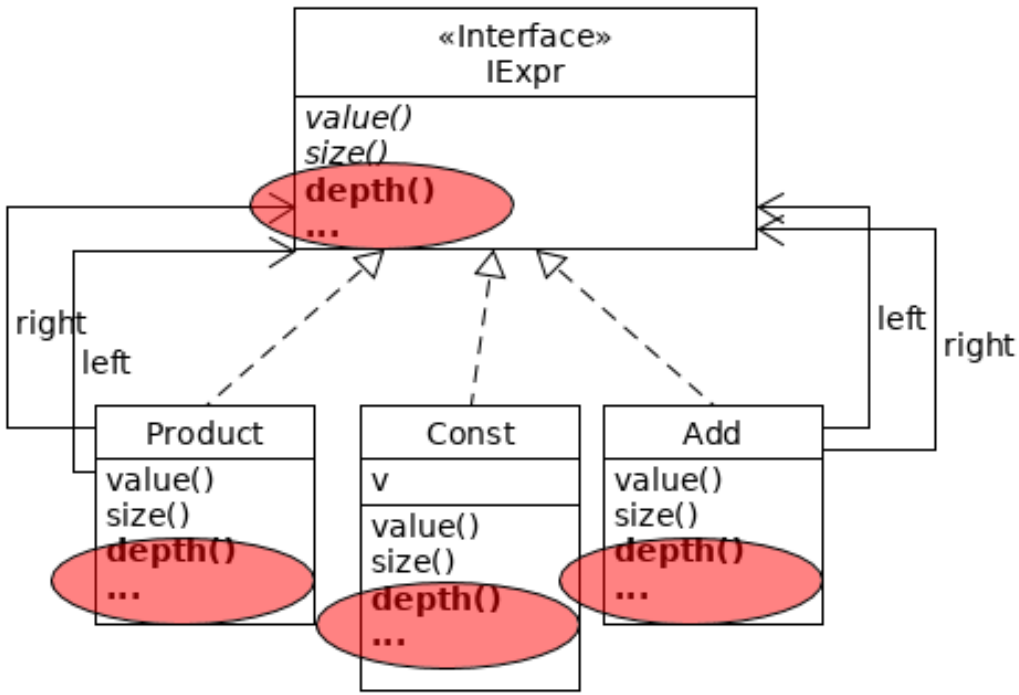
  simplify(add(add(0, 5), 8)) $\rightsquigarrow$ add(5, 8)

- Bisher: Modellierung mit Composite Pattern
- Problem: Composite Pattern erfordert beim Hinzufügen die Modifikation **aller** Ausdruck-Klassen

Weitere Operationen für Arithmethische Ausdrücke

Modellierung mit Composite Pattern

Problem: Composite pattern erfordert Modifikation aller Ausdruck-Klassen



Implementierung der Operation ist über viele Klassen verteilt

Operationen für Arithmethische Ausdrücke

in einer idealen Welt

```
1   public static int eval(IExpr e) {
2    /*
3      – falls e eine Const Instanz ist,
4              dann return e.value
5      – falls e eine Add Instanz ist,
6              dann return eval(e.left) + eval(e.right)
7      – falls e eine Produkt Instanz ist,
8              dann return eval(e.left) * eval(e.right)
9      */
10  }
```

Operationen für Arithmethische Ausdrücke

in einer idealen Welt

```
1  public static int eval(IExpr e) {
2   /*
3     – falls e eine Const Instanz ist,
4             dann return e.value
5     – falls e eine Add Instanz ist,
6             dann return eval(e.left) + eval(e.right)
7     – falls e eine Produkt Instanz ist,
8             dann return eval(e.left) * eval(e.right)
9     */
10 }
```

- Logik der Operation gesammelt an einer Stelle
- Ausdruck-Klassen bleiben unangetastet
- Es kann im Prinzip leicht geprüft werden ob alle Fälle beachtet worden sind

Keine wirkliche Lösung: instanceof

```
1  public static int eval(IExpr e) {
2    if (e instanceof Const) {                    // falls e eine Const Instanz ist
3      Const c = (Const) e;
4      return c.value;
5    } else if (e instance of Add) {              // falls e eine Add Instanz ist
6      Add a = (Add) e;
7      return eval(a.left) + eval(a.right)
8    } else if (...) {                            // ...
9     ...
10   } else {                                     // Fehler, wenn kein Fall zutrifft
11     return IllegalArgumentException...
12   }
13 }
```

Keine wirkliche Lösung: instanceof

```
1  public static int eval(IExpr e) {
2    if (e instanceof Const) {                    // falls e eine Const Instanz ist
3      Const c = (Const) e;
4      return c.value;
5    } else if (e instance of Add) {              // falls e eine Add Instanz ist
6      Add a = (Add) e;
7      return eval(a.left) + eval(a.right)
8    } else if (...) {                            // ...
9     ...
10   } else {                                     // Fehler, wenn kein Fall zutrifft
11     return IllegalArgumentException...
12   }
13 }
```

- Repetitives, fehleranfälliges Muster:

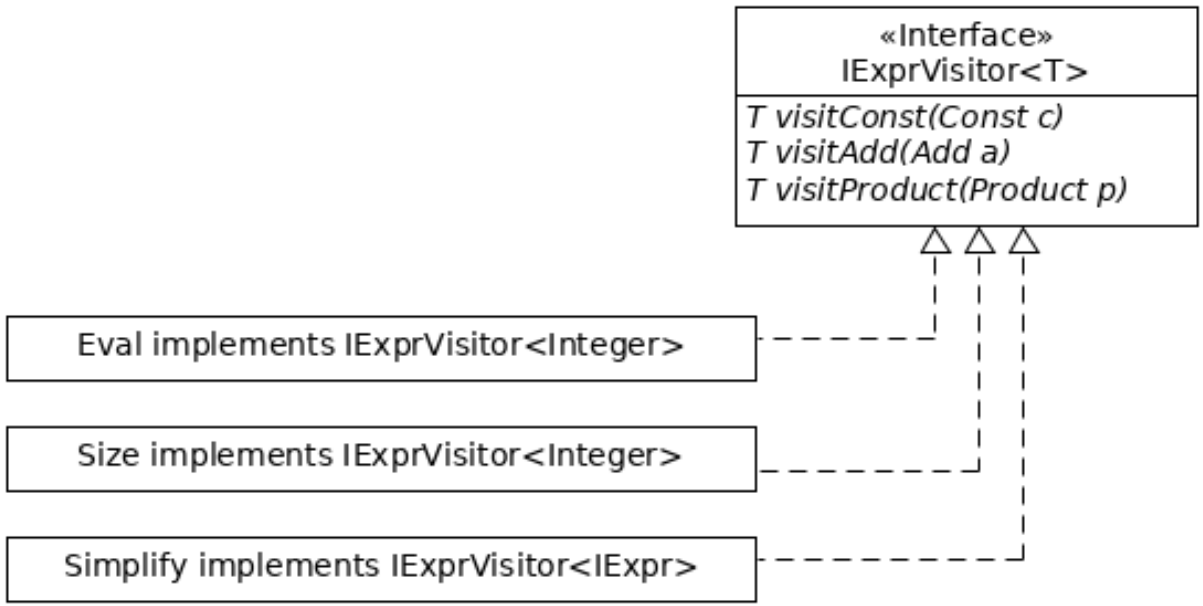
  if(e **instanceof** ⟨*Klasse*⟩) {⟨*Klasse*⟩ x = (⟨*Klasse*⟩) e; ...}

- Laufzeitfehler wenn ein Fall vergessen wird bei falschen Casts

. . . insbesondere wenn doch noch ein Fall nachträglich hinzugefügt wird

Visitors

Überblick



- Die Operationen sind Implementierungen eines **Visitor-Interfaces**
- Rückgabetyp einer Operation ist der **generische Parameter** T
- die verschiedenen Fälle sind auf die visit$*$ Methoden aufgeteilt
- Keine Typecasts und **instanceof** Test nötig

Beispiel: Eval

```
1  class Eval implements IExprVisitor<Integer> {
2  public Integer visitConst(Const c) {
3    return c.value;
4  }
5  public Integer visitAdd(Add a) {
6    return /* a.left + a.right */;
7  }
8  public Integer visitProduct(Product p) {
9    return /* p.left * p.right */;
10 }}
```

- ...
- die verschiedenen Fälle sind auf die visit∗ Methoden aufgeteilt
- Keine Typecasts und **instanceof** Test nötig

Beispiel: Eval

```
1  class Eval implements IExprVisitor<Integer> {
2  public Integer visitConst(Const c) {
3    return c.value;
4  }
5  public Integer visitAdd(Add a) {
6    return /* a.left + a.right */;
7  }
8  public Integer visitProduct(Product p) {
9    return /* p.left * p.right */;
10 }}
```

- ...
- die verschiedenen Fälle sind auf die visit∗ Methoden aufgeteilt
- Keine Typecasts und **instanceof** Test nötig

Es bleibt zu klären: Wie wird Eval aufgerufen?

Anwenden von Visitors

- Das Ausdruck-Interface (IExpr) muss das Visitor-Pattern unterstützen: nur der Ausdruck selber weiß welcher Fall aufgerufen werden muss
- Dazu reicht eine Methode `accept(IExprVisitor v)`
- „Aufruf" der Operation: e.accept(**new** Eval())

Anwenden von Visitors

- Das Ausdruck-Interface (IExpr) muss das Visitor-Pattern unterstützen: nur der Ausdruck selber weiß welcher Fall aufgerufen werden muss
- Dazu reicht eine Methode `accept(IExprVisitor v)`
- „Aufruf" der Operation: e.accept(**new** Eval())

Aufrufe im Vergleich

Composite Pattern

```
IExpr e = add(5, 4);
Integer i = e.eval();
String s = e.prettyPrint();
```

VisitorPattern

```
IExpr e = add(5, 4);
Integer i = e.accept(new Eval());
String s = e.accept(new PrettyPrint());
```

Einschub: Generische Methoden

Die Signatur von ,,accept'

```
IExpr e = add(5, 4);
Integer i = e.accept(new Eval());
String s = e.accept(new PrettyPrint());
```

Einschub: Generische Methoden

Die Signatur von ,,accept'

```
IExpr e = add(5, 4);
Integer i = e.accept(new Eval());
String s = e.accept(new PrettyPrint());
```

Die Methode accept muss zu IExpr hinzugefügt werden:

```
interface IExpr {
  ?? accept(IExprVisitor<??> v);


}
```

Einschub: Generische Methoden

Die Signatur von ,,accept'

```
IExpr e = add(5, 4);
Integer i = e.accept(new Eval());
String s = e.accept(new PrettyPrint());
```

Die Methode accept muss zu IExpr hinzugefügt werden:

```
interface IExpr {
  ?? accept(IExprVisitor<??> v);
  // String accept(IExprVisitor<String> v)
  // Integer accept(IExprVisitor<Integer> v)
}
```

Einschub: Generische Methoden

Die Signatur von ,,accept'

```
IExpr e = add(5, 4);
Integer i = e.accept(new Eval());
String s = e.accept(new PrettyPrint());
```

Die Methode accept muss zu IExpr hinzugefügt werden:

```
interface IExpr {
  <T> T accept(IExprVisitor<T> v);
  // String accept(IExprVisitor<String> v)
  // Integer accept(IExprVisitor<Integer> v)
}
```

Einschub: Generische Methoden

Die Signatur von ,,accept'

```
IExpr e = add(5, 4);
Integer i = e.accept(new Eval());
String s = e.accept(new PrettyPrint());
```

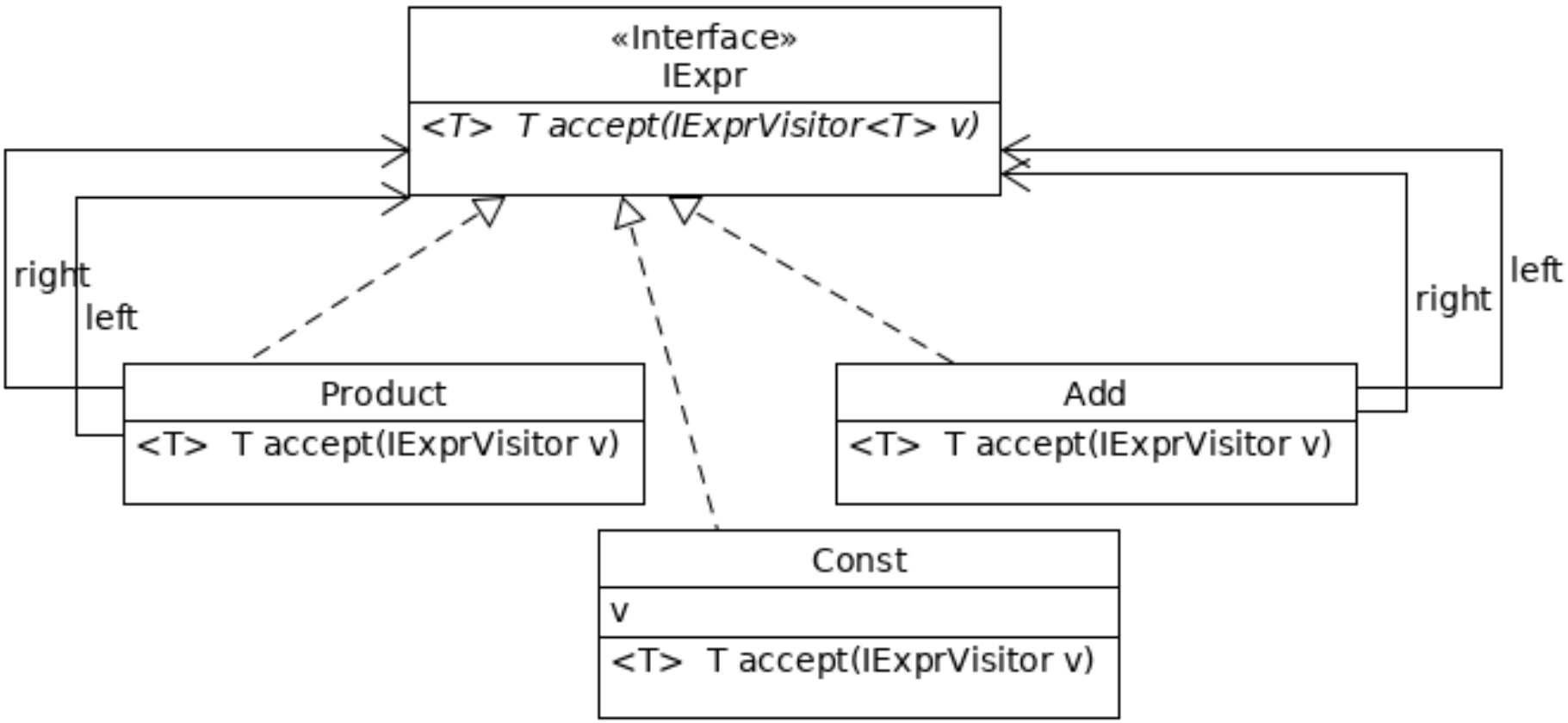
Die Methode accept muss zu IExpr hinzugefügt werden:

```
interface IExpr {
  <T> T accept(IExprVisitor<T> v);
  // String accept(IExprVisitor<String> v)
  // Integer accept(IExprVisitor<Integer> v)
}
```

- <T> T accept(IExprVisitor<T> v) ist eine **generische Methode**.
- anders als bei generischen Klassen (vgl. List<T>),
  muss der Typparameter beim Aufruf meist nicht angegeben werden.
- ansonsten: String s = e.<String>accept(new PrettyPrint())

Anwenden von Visitors

- Das Ausdruck-Interface muss Visitor unterstützen: nur der Ausdruck selber weiß welcher Fall aufgerufen werden muss.
- Dazu reicht eine Methode `accept(IExprVisitor v)`



Beispiel: Eval

```
1  class Eval implements IExprVisitor<Integer> {
2  public Integer visitConst(Const c) {
3    return c.value;
4  }
5  public Integer visitAdd(Add a) {
6    return a.left.accept(this) + a.right.accept(this);
7  }
8  public Integer visitProduct(Product p) {
9    return p.left.accept(this) * p.right.accept(this);
10 }}
```

- ...
- die verschiedenen Fälle sind auf die visit$*$ Methoden aufgeteilt
- Keine Typecasts und **instanceof** Test nötig
- Implementierung von accept ist **mechanisch**.
  (d.h. die Methode könnte automatisch generiert werden)

Beispiel: Depth

Hinzufügen von Fällen

Ein arithmetischer Ausdruck ist entweder

- ...
- oder eine **Variable**, z.B.: **new** Var("x")

Das Visitor-Pattern stellt sicher, dass
alle Visitor-basierten Operationen nachgebessert werden müssen

Hinzufügen von Fällen

Ein arithmetischer Ausdruck ist entweder

- ...
- oder eine **Variable**, z.B.: **new** Var("x")

Das Visitor-Pattern stellt sicher, dass
alle Visitor-basierten Operationen nachgebessert werden müssen

- beim Hinzufügen von Fällen muss die accept Methode implementiert werden (Teil von IExpr)
- dabei fällt auf, dass ein entsprechender visit$*$ Fall fehlt
- der muss zum $*$Visitor Interface hinzugefügt werden
- was zwangsweise eine Anpassung aller konkreten Visitors nach sich zieht

Eigenschaften des Visitor Patterns

Vorteile

- keine Casts oder instanceof checks
- Die Operationen müssen alle Fälle behandeln.

Nachteile

Fälle hinzufügen ist aufwendig
(vgl. “Die Operationen müssen alle Fälle behandeln”)