Kapitel 15

Virtuelle Klassen

Grundidee: Ein Member kann nicht nur Instanzvariable oder Methode, sondern auch Klasse sein

Anders als bei Inner classes gibt es Redefinition dieser *Virtuellen Klassen* in Unterklassen + dynamische Bindung!

⇒ neue Kompositionsmechanismen, Klassen höherer Ordnung

Experimentelle Sprache von Ernst/Cook [POPL 06]:

- virtuelle Klassen sehen aus wie innere Klassen, können aber mittels Bezugsobjekt als Member angesprochen werden. Bsp: `o.C x;`
- Variablen/Methodenparameter/ Ergebnisse können als Typ eine virtuelle Klasse, sogar aus einem anderen Parameter, haben. Bsp: `o.C f(K o; o.C x){...}`
- ist `C` eine virtuelle Klasse, die als Member in einer Unterklasse redefiniert wird, so unterliegt `o.C` der dynamischen Bindung ⇒ in `o.C x;` hat `x` “generischen” Typ

Tatsächlich hängt der Typ von `x` vom Laufzeitwert/typ von `o` ab!! („value-dependent types“)

Damit ist rein statische Typisierung sehr erschwert, dafür gewint man dynamisch Flexibilität

- virtuelle Klasse kann durch Schlüsselwort `out` auf das Objekt verweisen, in dem sie Member ist (vgl `outer` bei dynamic inner classes), aber dyn. Bindung über `out`, sowohl für Methoden als auch virtuelle Klassen!
  ⇒ Typ einer Variablen ist nicht statisch fix, sondern hängt vom Wert eines Bezugsobjektes ab!

  Solche „value-dependent types“ wurden zuerst in der funktionalen Programmierung untersucht

Weitere Details:

- ein `field` ist eine unveränderliche Instanzvariable (vgl `final`)
- Typen sind *Pfade* von (geschachtelten) Klassennamen

Beispiel: Implementierung arithmetischer Ausdrücke ($\rightsquigarrow$ abstrakter Syntaxbaum)

Herausforderung: Erweiterbarkeit sowohl von Ausdrucksarten als auch von Operationen auf Ausdrücken unter Beachtung des Lokalitätsprinzips (Vermeidung der „Tyrannei der dominanten Dekomposition“)

Dieses Beispiel würde man klassisch mit dem Visitor-Pattern behandeln, was aber zwei gewöhnliche Hierarchien erfordert und zur TddD führt

Realisierung in der Ernst/Cook Sprache:

```
class Base { // umfassende Klasse für Ausdrücke
//enthält 2 virtuelle Klassen
  class Exp {} // Basisklasse fur Ausdrucke

  // zunächst nur "Literale"
  // als einzige Art Ausdrücke
  class Lit extends Exp {
     int value;
  }

  Lit zero; //Konstante "0" als Member
  out.Exp TestLit() {//erzeugt Lit-Objekt
    // Ergebnis ist "generisch", da virtuelle Klasse
    // Achtung: Lit wird später redefiniert
    out.Lit l;
    l = new out.Lit();
    l.value = 3;
    return l;
  }
}

// Erweiterung: Negativer Ausdruck ("unäres minus") als
  weitere Art Ausdrücke
class WithNeg extends Base { //
  class Neg extends Exp { // Unterklasse einer virtuellen
     Klasse
     Neg(out.Exp e) {this.e = e; } // Konstruktor
     field out.Exp e;
  }
  out.Exp TestNeg () {
    return new out.Neg(TestLit());// dyn. Bindung!!
  }
}
```

```
// Hinzunahme einer Auswertungsfunktion ("Visitor")
class WiThEval extends Base {
  class Exp { // Redefinition  Virtuelle Klasse
     int eval() {return 0;}
  class Lit { // Redefinition
     int eval() {return value;}
  int testEval() {
     return out.TestLit().eval();//dyn. Bindung!
     // im Rumpf von TestLit wird redefiniertes Lit
       verwendet!
  }
}

// Kombination von allem
class NegAndEval extends WithNeg, WithEval {
  // Mehrfachvererbung
  class Neg { // Redefinition!!
    Neg(out.Exp e) {this.e = e;}
    int eval() { -e.eval();}
  }
  int TestNegAndEval() {
    return out.TestNeg().eval();
    // im Rumpf von TestNeg wird out.Neg(...) dynamisch
      gebunden!
  }
}
```

Vorteil: einfache Erweiterbarkeit sowohl um Unterklassen als auch um neue Operationen, keine TddD, Lokalitätsprinzip gewahrt

⇒ neuer Mechanismus zur Softwarekomposition: nicht nur Mehrfachvererbung (= Komposition zweier Oberklassen), sondern diese setzt sich auch auf die enthaltenen virtuellen Klassen fort

Wenn Klassen, die als Oberklassen verwendet werden, neue Unterklassen bekommen, erhalten deren virtuelle Klassen ebenfalls neue Oberklassen

Bsp: 1. Nach Komposition ist `NegAndEval.Neg` auch Unterklasse von `WithNeg.Neg`, weil `NegAndEval` Unterklasse von `WithNeg` ist;

2. `WithNeg.Neg` hat als Oberklasse `WithEval.Exp`, weil `Exp` in `WithEval` redefiniert wurde

Es ergibt sich eine Art dynamischer Mixin-Mechanismus, der – anders als Traits – sich rekursiv über Unterklassen/virtuelle Klassen fortpflanzt. Insbesondere ist der Typ einer Variablen nicht mehr statisch, sondern hängt von den Werten von Bezugsobjekten ab!

Das dynamische Verhalten wird allerdings schnell undurchschaubar (vgl. Kritik an AspectJ)

Virtuelle Klassen erlauben Generizität:

```
class Test {
  field out.WithNeg f1;
  f1.Exp n; // kann redefiniertes Exp sein!
  field out.NegAndEval f2;
  f2.Exp n2;
  ne.Neg buildNeg(out.out.WithNeg ne, ne.Exp ex) {// ex
    hat generischen Typ, abhängig von ne!
    return new ne.Neg(ex);
  }

  int Test(out.WithNeg f1, out.NegAndEval f2) {
    this.f1 = f1; this.f2 = f2;
    n = buildNeg(f1, n); // ok
    n.eval(); // Typfehler, da WithNeg.Exp kein eval hat
    f2.zero = new f2.Lit();
    n2 = buildNeg(f2, f1.zero); //Typfehler!?
    n2 = buildNeg(f2, f2.zero); //ok
    n2.eval();//ok, da NegAndEval.Exp=WithEval.Exp eval
      hat
  }
}
Test t = new Test(new NegAndEval(), new NegAndEval());
```

Zuweisung an `n2` geht, obwohl Return Type von `buildNeg` kein `eval` kennt!