Unveränderliche Klassen

Softwareentwicklung II (IB)

Prof. Dr. Oliver Braun

Letzte Änderung: 05.04.2018 07:02

Idee

- Früheres Beispiel zeigt Problem: Methode kann unerkannt fremdes Objekt manipulieren
- Ursache: Freier Zugriff auf Objektvariablen
- Lösungsidee: **Unveränderliche Klassen** (engl. immutable classes) lassen keine Änderungen an Objektvariablen zu
- Folge: Lesen von Informationen aus Objekten ok, Ändern nicht möglich
- Ausblick: Dadurch auch Parallelisierung / Nebenläufigkeit sehr viel einfacher

`final`-Objektvariablen

- Modifier `final` für Objektvariablen blockiert Änderungen (wie bei lokalen Variablen)
- `final`-Objektvariable muss einmal mit einem Wert versorgt werden
- Nachfolgende Änderung unzulässig $\Rightarrow$ erster (und einziger) Wert bleibt für die gesamte Lebensdauer des Objektes

Zuweisung des Wertes wahlweise … (1/2)

- bei der Definition

```
class Foo {
  final int n = 1;
  Foo() {}
}
```

- in einem Konstruktor

```
class Foo {
  final int n;
  Foo() {n = 1;}
}
```

Zuweisung des Wertes wahlweise ... (2/2)

- in einem verketteten Konstruktor

```
class Foo {
  final int n;
  Foo()       {this(1);}
  Foo(int n)  {this.n = n;}
}
```

- in einem Initializer

```
class Foo {
  final int n;
  {n = 1;}
  Foo() {}
}
```

Nicht ausreichend ...

- Defaultwert

```
class Foo {
  // Defaultwert 0, wird nicht übersetzt
  final int n;
  Foo() {}
}
```

- anderer Methodenaufruf

```
class Foo {
  final int n;
  Foo() {set(0);}
  // wird nicht übersetzt
  void set(int n) {this.n = n;}
}
```

Beispiel

- Unveränderliche Klasse `Rational`:

```
class Rational {
  final int numer;
  final int denom;

  Rational(int n, int d) {
    numer = n;
    denom = d;
  }
  ...
}
```

Rückgabe neuer Objekte

- Problem: Änderungen von `this` nicht mehr möglich
- Konsequenz: Frühere Fassung der `Rational`-Methode `mult` nicht mehr möglich
- Ausweg: Ergebnis in neuem Objekt zurückliefern, statt `this` zu modifizieren

Beispiel (1/3)

- Neue Fassung von `mult`:

```
class Rational {
   ...
  Rational mult(Rational that) {
    int n = numer*that.numer;
    int d = denom*that.denom;
    Rational result = new Rational(n, d);
    return result;
  }
}
```

Beispiel (2/3)

- Arbeitsweise:

    1. Zähler und Nenner des Produkts einzeln berechnen, in den lokalen Variablen `n` und `d` zwischenspeichern
    2. Drittes `Rational`-Objekt erzeugen, unabhängig von `this` und vom Parameterobjekt `that`
    3. Neues `Rational`-Objekt mit Zähler und Nenner des Produkts initialisieren
    4. An die lokale Variable `result` zuweisen
    5. `result` als Methodenergebnis zurückgeben

Beispiel (3/3)

- Syntaktisch kürzer, gleichwertig:

```
class Rational {
  ...
  Rational mult(Rational that) {
    return new Rational(numer * that.numer,
                        denom * that.denom);
  }
}
```

- Idee auf alle ändernden Methoden übertragen

Beispiele (1/3)

- Operationen primitiver Typen lassen Operanden unverändert, Ergebnis = neuer Wert

```
int a = ...;
int b = ...;
int c;
// a und b unverändert, Ergebnis in c
c = a * b;
```

- Vergleichbares Verhalten mit unveränderlichen Klassen:

Beispiele (2/3)

```
Rational a = ...;
Rational b = ...;
Rational c;
// a und b unverändert, Ergebnis in c
c = a.mult(b);
```

Beispiele (3/3)

- Gegensatz: Früheren Fassung der `mult`-Methode:

```
Rational a = ...;
Rational b = ...;
// b unverändert, Ergebnis in a, altes a ersetzt
a.mult(b);
```

entspricht etwa

```
int a = ...;
int b = ...;
// b unverändert, Ergebnis in a, altes a ersetzt
a *= b;
```

Wertesemantik

- **Wertesemantik** (engl. value semantics) einer Klasse: Methoden lassen `this` und Parameter unberührt
- Verhalten entspricht dem von primitiven Typen: Operationen lassen Operanden unberührt
- Gegensatz: **Referenzsemantik** (engl. reference semantics): Operationen können `this` oder Parameter-Objekte oder beide verändern
- Zusammenfassung

Primitive Typen	Wertesemantik
Unveränderliche Klassen	Wertesemantik
Veränderliche Klasse	Referenzsemantik

Vorteil Wertesemantik

- Wertesemantik ist vorteilhaft
- Beispiel: Suche nach der Ursache für fehlerhaftes Objekt

  Referenzsemantik Jeder Methodenaufruf kommt in Betracht, muss überprüft werden; Beispiel: `a.mult(b);` könnte `a` oder `b` oder keines oder beide verändern

  Wertesemantik Nur Konstruktoraufrufe zu prüfen; Methodenaufrufe können ein korrekt erzeugtes Objekt nicht mehr stören

- Definieren Sie Klassen mit Wertesemantik (= unveränderlich), wo immer möglich!

Unveränderlichkeit auf logischer Ebene

- Beispiel: Brüche $\frac{6}{8}$ und $\frac{3}{4}$ logisch (mathematisch) gleichwertig $\Rightarrow$ Kürzen eines `Rational`-Objektes zulässig, „Wert“ bleibt unverändert
- Trotz Wertesemantik: Modifier `final` eventuell zu strikt
- Technisch: (Kontrollierte) Veränderungen am Objekt zulassen, dennoch Wertesemantik eingehalten
- Weitere Schutzmaßnahmen folgen