3. Übungsbesprechung Programmkonstruktion

Karl Gmeiner

karl@complang.tuwien.ac.at

November 16, 2011

Rückblick und Ausblick

Abgaben Übungsblatt 2

439 Abgaben. 878 gewertet. 226 Einzelgruppen.

Häufiger Fehler:

- `boolean arbeitstag = !(wochentag == 6 || 7);`
- `boolean arbeitstag = !(wochentag == (6 | 7));`
- Korrekt:
  `boolean arbeitstag = !(wochentag == 6 || wochentag == 7);`

2. Test

- Praxis: Stoff von 3. Übungsblatt inkl. Baum als rekursive Datenstruktur.
- Theorie: Kapitel 3 + 4.
- Einteilung wie beim 1. Test.

Klasse Linie

```
public class Punkt { ... }

public class Linie {
    // Gerade von p1 nach p2
    private Punkt p1;
    private Punkt p2;

    public Linie(Punkt initP1, Punkt initP2) {
        p1 = initP1;
        p2 = initP2;
    }
}
```

Klasse Linie mit Verschiebbar (1)

```
public class Punkt {
    public double x, y; ...
    // Schlechte Design-Entscheidung, da Implementierung
    // nicht geaendert werden kann.
}

public class Linie implements Verschiebbar {
    // Gerade von p1 nach p2
    private Punkt p1;
    private Punkt p2;

    public void verschiebe(double deltaX, double deltaY) {
        p1 = new Punkt(p1.x + deltaX, p1.y + deltaY);
        p2 = new Punkt(p2.x + deltaX, p2.y + deltaY);
    }
}
```

Klasse Linie mit Verschiebbar (2)

```
public class Punkt {
    public double getX() { ... }
    public double getY() { ... }
}

public class Linie implements Verschiebbar {
    // Gerade von p1 nach p2
    private Punkt p1;
    private Punkt p2;

    public void verschiebe(double deltaX, double deltaY) {
        p1 = new Punkt(p1.getX()+deltaX, p1.getY()+deltaY);
        p2 = new Punkt(p2.getX()+deltaX, p2.getY()+deltaY);
    }
}
```

Klasse Linie mit Verschiebbar (3)

```
// Punkt erfuellt alle Voraussetzungen, um
// auch Verschiebbar zu sein.
public class Punkt implements Verschiebbar { ... }

public class Linie implements Verschiebbar {
    // Gerade von p1 nach p2
    private Punkt p1;
    private Punkt p2;

    public void verschiebe(double deltaX, double deltaY) {
        p1.verschiebe(deltaX, deltaY);
        p2.verschiebe(deltaX, deltaY);
    }
}
```

Klasse Linie mit Skalierbar (1)

```
public class Punkt implements Verschiebbar, Skalierbar {...}

public class Linie implements Verschiebbar, Skalierbar {
    // Gerade von p1 nach p2
    private Punkt p1;
    private Punkt p2;

    public void skaliere(double faktor) {
        p1.skaliere(faktor);
        p2.skaliere(faktor);
    }
}
```

switch-case ⇒ dynamisch

```
public class Begruessung {
    public static final int FRAU = 0, MANN = 1, KIND = 2;

    public static String begruesse(int person, String name) {
        String gruss = "";
        String anrede = "";
        switch(person) {
            case KIND : gruss = "Hallo";
                        anrede = name; break;
            case FRAU : gruss = "Sehr geehrte";
                        anrede = "Frau " + name; break;
            case MANN : gruss = "Sehr geehrter";
                        anrede = "Herr " + name; break;
        }
        return gruss + " " + anrede;
}   }
```

switch-case ⇒ dynamisch

Interface Person

```
public interface Person {
    String gruss();
    String anrede();
}
```

Methode begruesse

```
public static String begruesse(Person person) {
    return person.gruss() + " " + person.anrede();
}
```

Implementierung von Person

name als Objektvariable.

```
public class Frau implements Person {
    private String name;

    public Frau(String initName) {
        name = initName;
    }

    public String gruss() {
        return "Sehr geehrte";
    }

    public String anrede() {
        return "Frau " + name;
}   }
```

IntStack - Teil 1

```
public class IntStack {
    /* ... */
    public boolean isEmpty() {
        return head == null;
    }

    public void plus() {
        int arg0 = pop();
        int arg1 = pop();
        push(arg0 + arg1);
    }

    public void neg() {
        int arg = pop();
        push(-arg);
}   }
```

toString() für IntStack

```
public String toString() {
    String s = "[";

    Node n = head;

    while(n != null) {
        if(n != head) s += ",";

        s += " " + n.getValue() + " ";

        n = n.getNext();
    }

    return s + "]";
}
```

Warum equals(Object)

```
class Foo {
  boolean equals(Foo other) {
    return true;
  }
}

class Bar extends Foo {
  int a;
  Bar(int initA) { a = initA; }

  boolean equals(Bar other) {
      return this.a == other.a;
  }
}
```

- `Foo foo = new Bar(1);`
- `Bar bar = new Bar(2);`
- Ergebnis von `foo.equals(bar)`?
- Ergebnis von `bar.equals(foo)`?
- Ergebnis von `bar.equals((Bar) foo)`?

Die Methode `equals(Object)`

Vergleichen von Objekten - Vorgaben

- Reflexiv: `this.equals(this)` muss `true` sein.
- Symetrisch: `this.equals(that) == that.equals(this)`
- ... (transitiv, consistent)

Warum nicht über toString()

```
class Foo {
  @Override
  public boolean equals(Object other) {
    return toString().equals(
      other.toString());
} }

class Bar extends Foo {
  /* ... */
  @Override
  public String toString() {
    return "Bla";
  }
} }
```

- `Foo foo1 = new Bar(1);`
- `Foo foo2 = new Bar(2);`
- Ergebnis von `foo1.equals(foo2)`?

Warum nicht instanceof?

```
class Foo {
  @Override
  public boolean equals(Object other) {
    return other instanceof Foo;
} }

class Bar extends Foo {
  /* ... */
  @Override
  public boolean equals(Object other) {
    if(other instanceof Bar) {
      Bar that = (Bar) other;
      return this.a == that.a;
    } else {
      return false;
} } }
```

- `Foo foo = new Foo();`
- `Bar bar = new Bar(2);`
- Ergebnis von `foo.equals(bar)`?
- Ergebnis von `bar.equals(foo)`?

Implementierung mit getClass()

```
class Foo {
    @Override
    public boolean equals(Object other) {
        return this.getClass() == other.getClass();
}   }

class Bar extends Foo {
    /* ... */
    @Override
    public boolean equals(Object other) {
        if(this.getClass() == other.getClass()) {
            Bar that = (Bar) other;
            return this.a == that.a;
        } else {
            return false;
}   }   }
```

Typische equals(Object)-Methode

```
class Bar extends Foo {
  @Override
  public boolean equals(Object other) {
    if (other == null) { return false; }
    if (other == this) { return true; }
    if (other.getClass() != getClass()) {
      return false;
    }

    Bar that = (Bar) other;
    return this.a == that.a;
  }
}
```

equals(Object) in IntStack - Teil 1

```
@Override
public boolean equals(Object other) {
  if (other == null) { return false; }
  if (other == this) { return true; }
  if (other.getClass() != getClass()) {
    return false;
  }

  IntStack that = (IntStack) other;
```

equals(Object) in IntStack - Teil 1

```
    Node n0 = this.head;
    Node n1 = that.head;

    while(n0 != null && n1 != null) {
        if(n0.getValue() != n1.getValue()) {
            // Wenn Werte nicht uebereinstimmen
            break; // beende Schleife.
        }

        n0 = n0.getNext();
        n1 = n1.getNext();
    }

    // Wenn in beiden das Ende der Liste erreicht wurde:
    return n0 == null && n1 == null;
}
```

Laufzeitabschätzung

Ändert sich die Laufzeit mit der Größe des Stacks?

- `pop()`: Gleiche Laufzeit bei Stacks mit unterschiedlicher Länge $\Rightarrow$ konstant
- `plus()`: 2x `pop`, 1x +, 1x `push`. 3x konstant = konstant.
- `toString()`: Jedes Element wird exakt 1x angeschaut $\Rightarrow$ linear (in der Länge des Stacks)
- `equals(Object other)`: Linear in der Länge des kürzeren Stacks.

Quadratische Laufzeit

Üblicherweise doppelt geschachtelte Schleife:

```
public boolean containsAll(IntStack that) {
    // Sind alle Elemente in this auch in that?
    for(Node n = head; n != null; n = n.getNext()) {
        boolean inStack = false;
        for(Node m = that.head; m != null; m = m.getNext()) {
            if(n.getValue() == m.getValue()) {
                inStack = true;
                break; // Element gefunden, beende Schleife.
            }
        }
        if(!inStack) return false; // Element nicht gefunden.
    }
    return true; // Alle Elemente gefunden.
}
```