Bonnes pratiques en Java

Table des matières

Einleitung

Java ist eine erste Wahl für die Realisierung sehr großer Softwareprojekte. Der Nutzen der folgenden Tipps wird auch erst auf diesem Hintergrund ersichtlich: „guter Code“, ist als solcher der gut gekapselt ist, leicht lesbar und damit auch mit geringen Kosten wartbar ist (siehe Tipp 4, 5 und 6). Die Semantik der zwischen den Objekten ausgetauschten Information sollte in den Methodennamen Ausdruck finden und deren Syntax so weit wie möglich durch Typisierung dem Compiler überprüfbar bleiben (Tipp 5).

Viele Richtlinien zur sicheren Programmierung (Tipp 1,2,3,7) erschließen sich dem Leser erst, wenn man sich Objekte als Speicher und Kommikationsmittel von parallelen asynchronen Workflows vorstellt. Workflows können z. B. als Threads oder Beans realisiert sein. Die hier vorgestellten Grundregeln zur sicheren Programmierung sind so einfach, dass sie nicht alle Anwendungsfälle der Praxis abdecken können. Zu jeder Regel wird man eine fortgeschrittene Anwendung finden, die einer Ausnahme bedarf. Bis unsere Schüler allerdings mit solch komplexen Fällen konfrontiert werden, sind sie lange nicht mehr bei uns im Unterricht. Dann sei alles erlaubt, „if you know what you are doing...“.

Die hier vorgestellten Tipps, sind als Hilfe zu verstehen um mit einfachen Mitteln – einer Hand voll Regeln - sicheren und guten Code[1] zu schreiben. Aus pädagogischen Gründen kann es Sinn machen manchmal davon abzuweichen.

1 Siehe auch *Java Programming Style Guidelines:* http://geosoft.no/development/javastyle.html

Tipp 1: Gültige Objekte erzeugen

Bei gültiger Eingabe durch den Benutzer[2] darf kein Objekt zur keiner Zeit ungültige Attribute haben.[3] Gültig meint hier im Sinne der OO, dass das Objekt zu jeder Zeit mit allen seinen Attributen einem Objekt der Wirklichkeit entspricht.
Synchronisationsmechanismen zwischen der View-Schicht und der Model-Schicht - wie `updateView`() usw. - müssen Transaktions-Charakter haben.

// ungünstig

```
Fraction f = new Fraction(1,1);
//ungültiges Objekt
int n = Integer.valueOf( nTextField.getText() );
int d = Integer.valueOf( dTextField.getText() );
//ungültiges Objekt
f.set(n,d) ;
//gültiges Objekt
...
```

// noch ungünstiger

```
Fraction f = new Fraction(1,1);
//ungültiges Objekt
int n = Integer.valueOf( nTextField.getText() );
int d = Integer.valueOf( dTextField.getText() );

f.setN(n);
//ungültiges Objekt
f.setD(d);
//gültiges Objekt
...
```

// richtig

```
Fraction f = null;
//kein Objekt
int n = Integer.valueOf( nTextField.getText() );
int d = Integer.valueOf( dTextField.getText() );

f = new Fraction(n,d);
//gültiges Objekt
...
```

2 Die Beschränkung auf gültige Benutzerdaten gilt nur, da wir in der 11TG- 12GE keine Gültigkeitstests vornehmen. I. A. sollten Objekte immer gültig oder „null“ sein.

3 Wichtigster Grund : Objekte müssen immer gültig sein, da i. A. jederzeit zum Beispiel durch andere Threads darauf zugegriffen werden kann.

Tipp 2: Objekte nicht wiederverwenden

Im Allgemeinen Objekte nicht wiederverwenden, sondern neu erstellen. Keine `initialize`(), `init`()-Methode o. ä. einführen, stattdessen den Konstruktor benutzen. Je kürzer die Lebensdauer, desto besser. Ist ein Objekt nicht mehr gültig, setzt man dessen Referenz auf Null, den Rest macht der Garbage-Collector. Dieser Tipp betrifft einen häufigen Fehler bei Umsteigern, von prozeduralen Programmiersprachen.

```
// ungünstig
f = new Fraction(x,y);
...

private void enterButtonActionPerformed(
                          java.awt.event.ActionEvent evt) {
   int n = Integer.valueOf( nTextField.getText() );
   int d = Integer.valueOf( dTextField.getText() );
   f.set(n,d) ;   // altes Objekt wiederverwendet
...
}
```

```
// richtig
f = new Fraction(x,y);
...

private void enterButtonActionPerformed(
                          java.awt.event.ActionEvent evt) {
   int n = Integer.valueOf( nTextField.getText() );
   int d = Integer.valueOf( dTextField.getText() );
   f = new Fraction(n,d);   // neues Objekt
...
}
```

Tipp 3: Direkt auf die eigenen Attribute zugreifen

Setter und Getter dienen dem objektorientierten Paradigma der Kapselung. [4] Sie sind damit Teil des Interfaces, welches die Funktion und den Zustand eines Objektes nach außen abstrahiert. Dabei ist prinzipiell jeder Methode erlaubt eigene (auch private) Attribute direkt zu verändern, bzw. deren Inhalt nach außen zu kommunizieren.

Kapselung ist insbesondere nicht als Abstaktionskonzept zu verstehen, welches zwischen den Attributen und den Methoden eines Objektes steht.[5] Setter und Getter welche die Kapselung realisieren, dienen der Kommunikation nach außen und sollten nur in Ausnahmefällen intern verwendet werden. [6]

```
// ungünstig
public class MyNumber
{
    private int val;
    public MyNumber(int number)
    {
        setVal(number);
    }

    public int getVal()
    {
        return val;
    }

    public void setVal(int number)
    {
        val = number;
    }

    public void srt()
    {
        setVal(getVal()*getVal());
    }
}
```

4 Als Kapselung bezeichnet man den kontrollierten Zugriff auf Methoden bzw. Attribute von Klassen. Klassen können den internen Zustand **anderer** Klassen nicht in unerwarteter Weise lesen oder ändern. <http://de.wikipedia.org/wiki/Objektorientierte_Programmierung#Kapselung>.

5 Besonders problematisch ist der Zugriff auf Setter vom Konstruktor aus, welcher nur auf static oder final Methoden zugreifen darf (Constructors shouldn't call overridables). Setter müssen zudem stets ableitbar sein - können daher weder static noch final deklariert werden - was den Zugriff vom Konstruktor aus ausschließt.

6 Soll der Wertebereich von Attributen eingeschränkt und überwacht werden, sind hierfür in Java besondere Verfahren vorgesehen (nicht in 11TG-12GE).

```
// richtig
public class MyNumber
{
    private int val;
    public MyNumber(int number)
    {
        val=number;
    }

    public int getVal()
    {
        return val;
    }

    public void srt()
    {
        val = val * val;
    }
}
```

Tipp 4: Ausgabe-Strings in der View-Schicht generieren

Ausgabe-Strings, die der Benutzer sieht, werden in der *View-Schicht* generiert, nicht in der *Model-Schicht*. Ausnahme: `toString()`, welches in der Praxis hauptsächlich dem Debuggen dient.[7]

```
// ungünstig
public class MainFrame extends javax.swing.JFrame {

    private MyNumber myNumber = null;
...
    private void testButtonActionPerformed(
                    java.awt.event.ActionEvent evt) {
          myNumber = new MyNumber(
                      Integer.valueOf(
                          numberTextField.getText()

                       ));
          messageLabel.setText(myNumber.getMessage());
    }
}
```

```
// ungünstig
public class MyNumber
{
    private int num  = 0;

    public MyNumber(int number)
    {
        num = number;
    }

    public String getMessage()
    {
        String message = "";
        if (num%2 == 0)
            message = "I am even.";

        else
           if (num%2 == 1)
              message = "I am odd";
        return message;
    }
}
```

7 In der View-Schicht. findet dann beispielsweise auch die Lokalisierung statt.

```
// richtig
public class MainFrame extends javax.swing.JFrame {
    private MyNumber myNumber = null;
    ...
    private void testButtonActionPerformed(
                    java.awt.event.ActionEvent evt) {
          myNumber = new MyNumber(
                          Integer.valueOf(
                             numberTextField.getText()));

          if (myNumber.isEven())
                 messageLabel.setText("I am even.");
          if(myNumber.isOdd())
                 messageLabel.setText("I am odd.");
    }
    ...
}
```

```
// richtig
public class MyNumber
{
    private int num  = 0;

    public MyNumber(int number)
    {
        num = number;
    }

    public boolean isEven()
    {
        return num%2 == 0;
    }

    public boolean isOdd()
    {
        return num%2 == 1;
    }
}
```

Tipp 5: Keine Return-Codes

Keine Return-Codes in Methoden verwenden. Stattdessen für jeden einzelnen Code eine eigene bool'sche Methode z. B. `isEven`(), `isOdd`() ... einführen.[8]

Ausnahme: Implementierung des `Comparable` Interfaces, d. h. hier `compareTo()`-Methode mit dessen Algebra[9].

```
// ungünstig
public class MainFrame extends javax.swing.JFrame {

    private MyNumber myNumber = null;

    ...
    private void testButtonActionPerformed(
                    java.awt.event.ActionEvent evt) {
         myNumber = new MyNumber(
                        Integer.valueOf(
                            numberTextField.getText()));
         if (myNumber.analyse() == 1)
             messageLabel.setText("I am even.");
         else if(myNumber.analyse() == 2)
             messageLabel.setText("I am odd.");
    }
```

```
// ungünstig
public class MyNumber
{
    private int num  = 0;
    public MyNumber(int number)
    {
        num = number;
    }

    public int analyse()
    {
        int returnCode = 0;
        if (num%2 == 0)
            returnCode = 1;
        else
           if (num%2 == 1)
              returnCode = 2;
        return returnCode;
    }
}
```

8 Die Semantik der zwischen den Objekten ausgetauschten Information sollte in den Methodennamen Ausdruck finden und deren Syntax so weit wie möglich durch Typisierung dem Compiler überprüfbar bleiben.

9 http://www.javapractices.com/topic/TopicAction.do?Id=10

```
// richtig
public class MainFrame extends javax.swing.JFrame {
    private MyNumber myNumber = null;
    ...
    private void testButtonActionPerformed(
                java.awt.event.ActionEvent evt) {
          myNumber = new MyNumber(
                      Integer.valueOf(
                         numberTextField.getText()));

          if (myNumber.isEven())
              messageLabel.setText("I am even.");
          if(myNumber.isOdd())
              messageLabel.setText("I am odd.");
    }
    ...
}
```

```
// richtig
public class MyNumber
{
    private int num  = 0;

    public MyNumber(int number)
    {
        num = number;
    }

    public boolean isEven()
    {
        return num%2 == 0;
    }

    public boolean isOdd()
    {
        return num%2 == 1;
    }
}
```

Tipp 6: View Schicht initialisiert Model Schicht

und nicht umgekehrt. [10]

```
// ungünstig
public MainFrame() {
        initComponents();
        //model layer initializes view layer
        mem = new DataMemory();
        valueTextField.setText(
            String.valueOf(mem.get());
}
```

```
// richtig
public MainFrame() {
        initComponents();
        //view layer initializes model layer
        mem = new DataMemory(
                Double.valueOf(
                 currentValueTextField.getText()) );

}
```

Anfangswerte werden mit der Initialisierung des GUI geladen, (eventuell) durch User-Einstellungen überschrieben und dann in alle Model-Klassen / DB usw. übertragen.

Anfangswerte von `TextFields` werden vom Programmierer im Properties-Inspektor eingetragen und von einer Methode der Klasse `MainFrame` ins Model übernommen. Das Model reagiert in gleicher Weise wie auf « normale » Benutzereingaben.

10 In der View-Schicht. findet dann beispielsweise auch die Lokalisierung statt.

Tipp 7. Validator und Analyzer Klassen

Seltene Klassen, deren Zweck es ist die Daten **anderer** Klassen zu überprüfen, müssen zustandslos[11] sein, d. h. sie haben keine Attribute und sind meist vollständig static.

```
// ungünstig
public class NumberAnalyser
{
        private int number;

        public NumberAnalyser(int numberToTest)
        {
                number = numberToTest;
        }
        public boolean isPos()
        {
                return number > 0;
        }
}
```

```
// richtig
public class NumberAnalyser
{
        public /* static */ boolean isPos(int numberToTest)
        {
                return numberToTest > 0;
        }
}
```

Hinweis: Hinsichtlich einer guten Kapselung der Objekte finden Tests i. A. in der Klasse statt, die auch die Daten in sich trägt. Dieser Tipp bezieht sich **nicht** auf Gültigkeitstest der **eigenen** Daten einer Klasse (Gültigkeitstests sind nicht vorgesehen 11TG-12GE).

Gültigkeitstests von Parametern finden normalerweise im Konstruktor statt. Ist ein solcher Test nicht erfolgreich, wird im Konstruktor eine Exception geworfen welche verhindert, dass ein ungültiges Objekt erzeugt wird (nicht vorgesehen 11TG-12GE).

11 Zustandslos, weil sich die zu testenden Daten zwischen dem Konstruktor und der Testmethode ändern können. Die Testmethode selbst wird als atomar betrachtet und ggf. über Locking-Mechanismen abgesichert.

Tipp 8: Bei Zuweisung, Parameterübergabe und Vergleich Referenzen von Werten unterscheiden

Zuweisung

Der Operator = verhält sich bei elementaren Datentypen, mutable-Objekten und immutable-Objekten unterschiedlich, was manchmal für Verwirrung sorgt:

int, double, ... (Elementare Datentypen=primitives)

aElem ist 4 bElem ist 5

```
aElem = bElem;            //Inhaltskopie
```

aElem ist 5 bElem ist 5

```
aElem = 3;
```

aElem ist 3 bElem ist 5

String, Integer, Double ...(immutable)

aImmutable ist "do" bImmutable ist "re"

```
aImmutable = bImmutable; //Cloning, “2“ Objekte
```

aImmutable ist "re" bImmutable ist "re"

```
aImmutable = "mi";
```

aImmutable ist "mi" bImmutable ist "**re**"

eigens Objekt (mutable)

aMutable.getVal() ist 4 bMutable.getVal() ist 5

```
aMutable = bMutable;     //Referenzkopie, 1 Objekt
```

aMutable.getVal() ist 5 bMutable.getVal() ist 5

```
aMutable.increment();     //2 Referenzen, 1 Objekt
```

aMutable.getVal() ist 6 bMutable.getVal() ist **6**

Regel: Der Operator = kopiert bei elementaren Datentypen und immutable-Objekten[12] die Inhalte, während bei mutable-Objekten nur die Referenz auf das Objekt kopiert wird.

12 Technisch gesehen, findet bei immutable-Objekten nicht direkt ein `clone`() statt, d. h. aller Wahrscheinlichkeit nach zeigen beide Referenzen auf den selben Speicherbereich, obwohl das nicht garantiert ist. Dennoch kann man beide als unabhängige geklonte Objekte betrachteten, da das Verändern des einen keine Veränderung des anderen nach sich zieht.

Parameterübergabe

Beim Aufruf einer Methode kann man sich die Parameterübergabe wie eine Zuweisung von zwei Variablen vorstellen. D. h. die obige Regel gilt auch für die Parameterübergabe:

übergebener Datentyp	**Beispiel**	**Rückwirkung einer Veränderung innerhalb der aufgerufenen Methode auf die aufrufende Methode.**	**Übergabetyp[13]**
elementar	`int, double, byte`	keine	call by value
immutable-Objekt	`String, Integer, Double...`	keine	call by value
mutable-Objekt	`MyNumber, RunningTrack, ...`	Es findet bei der Parameterübergabe kein `clone`() statt. *Eine Veränderung eines Attributes, des übergebenen Objektes wirkt sich nach außen aus!* Eine Veränderung der übergebenen Referenz innerhalb der Methode wirkt sich nicht nach außen aus.	call by reference

Vergleich

Eine gibt eine Vergleichsart für elementare Datentypen (primitives) == und vier Arten für Objekte: ==, `equals`() , `compareTo`(), `compare`().[14] Beispiele:

Vergleichsart **Datentyp**	**==**	**`equals`() , `compareTo`(), `compare`()**
`int, double, byte`	Wertegleichheit?	N/A
`String, Double, Integer, Byte`	N/A, *nicht benutzen!*[15]	Wertegleichheit?
`MyNumber, RunningTrack`	Referenzengleichheit?	Wertegleichheit? *selbst implementieren!*

Bei eigenen Objekten muss die `equals`() Methode selbst implementiert werden, sonst verhält sie sich fälschlicherweise wie ==.[16] Das gleiche gilt für `compareTo`() und `compare`().

13 Technisch gesehen, werden bei der Parameterübergabe immer entweder Werte oder Referenzen auf dem Stack *kopiert*. Im Gegensatz zu C++ wirkt sich eine Veränderung der Referenz innen nicht nach außen aus, da innen mit einer Kopie der Referenz gearbeitet wird. Dennoch greifen die äußere und die innere Referenz auf das selbe Objekt zu.

14 http://www.ensta-paristech.fr/~diam/java/online/notes-java/data/expressions/22compareobjects.html

15 Aus Optimierungsgründen versucht der Compiler Immutable-Objekte nur einmal im Speicher anzulegen, d. h. häufig sind bei gleichen Werten auch die Zeiger identisch. Darauf kann man sich aber nicht verlassen.

16 Wird die Klasse in hashing-Strukturen wie `HashSet` und `HashMap` benutzt (nicht 11TG-13GE), muss zusätzlich zu `equals`() die Methode `hashCode`() überschrieben werden.

Vergleich	Elementare Datentypen	Objekte
`a == b`, `a != b`	*Wertegleichheit*	*Referenzengleichheit* Vergleicht Referenzen, nicht Werte. Die Anwendung von `==` mit Referenzen beschränkt sich i. A. auf: • Vergleich mit null. • Vergleich von 2 (konstanten) `enum` Objekten. • Zeigen 2 Referenzen auf das selbe (eine) Objekt? Haben immutable-Objekte wie `Integer` und `String` den gleichen Inhalt, legt der Compiler aus Optimierungsgründen das Objekt oft nur einmal im Speicher an, sodass dann auch die Referenzen gleich sind. Verlassen kann man sich darauf nicht. Daher immutable-Objekte nie mit == vergleichen!
`a.equals(b)`	N/A	*Wertegleichheit* Alle Attribute der 2 Objekte werden miteinander verglichen und müssen übereinstimmen. `equals`() ist für alle Standardklassen wie `Integer`, `String` usw. sinnvoll vordefiniert. Für eigene Klassen muss `equals`() selbst implementiert werden.[17] Wird das vergessen, verhält sich `equals`() fälschlicherweise wie ==.[18] Es zeigt sich, dass die korrekte Definition von `equals`() im Falle abgeleiteter Klassen, alles andere als trivial ist.[19]
`a.compareTo(b)`	N/A	*Wertegleichheit* Das Comparable Interface definiert für alle geordneten Datentypen neben der Gleichheit auch größer/kleiner Relationen. Beispiele: `String`, `Double`, `Integer`

Fazit: Mit `==` prüft man ob es sich um ein Objekt mit zwei Namen handelt, und mit `equals`() prüft man ob zwei Objekte (verschiedene Instanzen) die gleichen Attributwerte haben.

17 Wird die Klasse in hashing-Strukturen wie `HashSet` und `HashMap` benutzt (nicht 11TG-13GE), muss zusätzlich zu `equals`() auch die Methode `hashCode`() überschrieben werden.

18 Die Methode `equals`() ist in der Obejct Klasse definiert und wird von abgeleiteten nicht statischen Klassen überschrieben. Damit enthalten alle Klassen eine rudimentäre `equals`() Implementierung, welche in der Grundform aber für nicht statische Objekte ungeeignet ist.

19 Siehe: Horstmann's *Core Java Vol 1*.