

Konzepte und Werkzeuge für die Softwareentwicklung mit Java:

Java Code Conventions

Von Sebastian Pöttsch

- Einführung
- Motivation
- Ziele
- Anforderungen
- Sun Konventionen
- Die Realität
- Zusammenfassung
- Literatur

1. Einführung

- **Gründe:**
 - Viele Entwickler an einem Projekt
 - Damit viele unterschiedliche Programmierstile
 - Originalentwickler betreut Software nicht lebenslänglich
 - Wiederverwendbarkeit ist abhängig von Softwarequalität
 - Programmcode sollte deshalb lesbar und verständlich sein
 - 80% der Softwarekosten entfallen auf Instandhaltung

„In order to write great software, you have to write software greatly.“ [net]

- **Beispiel ohne Code-Konvention**

```
/* Use the insertion sort technique to sort the "data" array in
ascending order. This routine assumes that data[ firstElement ]
is not the first element in data and that data[ firstElement-1 ]
can be accessed. */ public void InsertionSort( int[] data, int
firstElement, int lastElement ) { /* Replace element at lower
boundary with an element guaranteed to be first in a sorted list.
*/ int lowerBoundary = data[ firstElement-1 ]; data[ firstElement-1 ]
= SORT_MIN; /* The elements in positions firstElement through
sortByoundary-1 are always sorted. In each pass through the loop,
sortByoundary is increased, and the element at the position of
the new sortByoundary probably isn't in its sorted place in the
array, so it's inserted into the proper place somewhere between
firstElement and sortByoundary. */ for ( int sortByoundary =
firstElement+1; sortByoundary <= lastElement; sortByoundary++ )
{ int insertVal = data[ sortByoundary ]; int insertPos = sortByoundary;
while ( insertVal < data[ insertPos-1 ] ) { data[ insertPos ] =
data[ insertPos-1 ]; insertPos = insertPos-1; } data[ insertPos ]
= insertVal; } /* Replace original lower-boundary element */
data[ firstElement-1 ] = lowerBoundary; }
```

- Beispiel nach Anwendung von Code-Konventionen:

```
/* Use the insertion sort technique to sort the "data" array in ascending
order. This routine assumes that data[ firstElement ] is not the
first element in data and that data[ firstElement-1 ] can be accessed. */
public void InsertionSort( int[] data, int firstElement, int lastElement ) {

    // Replace element at lower boundary with an element guaranteed to be
    // first in a sorted list.
    int lowerBoundary = data[ firstElement - 1 ];
    data[ firstElement - 1 ] = SORT_MIN;

    /* The elements in positions firstElement through sortBoundary-1 are
    always sorted. In each pass through the loop, sortBoundary
    is increased, and the element at the position of the
    new sortBoundary probably isn't in its sorted place in the
    array, so it's inserted into the proper place somewhere
    between firstElement and sortBoundary.
    */
    for ( int sortBoundary = firstElement + 1; sortBoundary <= lastElement;
          sortBoundary++ ) {
        int insertVal = data[ sortBoundary ];
        int insertPos = sortBoundary;

        while ( insertVal < data[ insertPos - 1 ] ) {
            data[ insertPos ] = data[ insertPos - 1 ];
            insertPos = insertPos - 1;
        }
        data[ insertPos ] = insertVal;
    }

    // Replace original lower-boundary element
    data[ firstElement - 1 ] = lowerBoundary;
}
```

3. Ziele

- Lesbarkeit verbessern
- Verständnis erleichtern
- Wiederverwendung erleichtern
- Code-Qualität erhöhen
- Instandhaltungskosten verringern

4. Anforderungen

- 1. Genaue Darstellung der logische Struktur des Codes.
 - Das erreicht man z.B. durch den Einsatz von Leerzeichen, Leerzeilen und Einzüge.
- 2. Konsequente Darstellung der logische Struktur.
 - Das bedeutet, die Konventionen können bei jeden auftretenden Fall angewendet werden (ohne viele Ausnahmen).
- 3. Verbesserung der Lesbarkeit des Codes.
 - Gute Konventionen erleichtern das Lesen des Codes und werden es niemals verschlechtern. Sie sollten auf keinen Fall guten Code schlechter machen.
- 4. Resistenz gegenüber Änderungen.
 - Das Ändern von einer Zeile Code sollte nicht das Ändern von vielen Zeilen Code auslösen.

5. Sun Konventionen

- Wurden von den Mitgliedern des Java Communicator Teams verabschiedet
- Zu finden auf <http://java.sun.com/docs/codeconv/>

- **Dateinamen**
 - 1. Für Quellcodedateien: *.java
 - 2. Für die kompilierten Bytecodedateien: *.class
- **Weitere gebräuchliche Dateinamen**
 - 1. Für Information über das Programm: README
 - 2. Für das Makefile: Makefile

5. Sun Konventionen

- **Dateiaufbau**

1. Quellcodedateien sollten mit Kommentar beginnen
 - Informationen zum Klassennamen, zur Version, zum Datum und zum Urheberrecht
2. Paketzugehörigkeit
3. Importstatements für die zu importierenden Klassen
4. Klassen- und Schnittstellendeklarationen (mit Kommentar)
 - Nicht mehr als eine public Klasse pro Datei
 - Ausnahme Helferklassen
5. Statische Klassenvariablen
6. Instanzvariablen (Reihenfolge: public, protected, private)
7. Konstruktor
8. Methoden (geordnet nach Funktionalität)

5. Sun Konventionen

- **Einzug**
 - 4 Leerzeichen
 - Untersuchungen haben ergeben, dass 4 Leerzeichen optimal sind
 - Keinen Tabulator verwenden
- **Zeilenlänge**
 - Nicht mehr als 80 Zeichen
 - Gründe:
 1. Zeilen lassen sich leichter lesen
 2. Lässt Programmierer nochmals über den Sinn ihrer Zeile nachdenken
 3. Zeilen lassen sich besser drucken
 4. Alte Terminals (z.B. bei IBM Großrechnern) sind auf 80 Zeichen begrenzt

5. Sun Konventionen

• Zeilenumbruch

- Umbrüche nach Kommas
- Umbrüche nach Operatoren
- Anpassen des Einzuges auf vorhergehenden Ausdruck
- Klammersausdrücke nicht zerstückeln

```
if ((condition1 && condition2)
    || (condition3 && condition4)
    ||!(condition5 && condition6)) {
    doSomethingAboutIt();
}
```

```
// so nicht
if ((condition1 && condition2)
    || (condition3 && condition4)
    ||!(condition5 && condition6)) { // schlechter Umbruch
    doSomethingAboutIt();           // Zeile ist leicht zu übersehen
}
```

- **Leerraum**

- Zwei Leerzeilen zwischen Klassen-, Schnittstellen-Definitionen und zwischen einzelnen Codeabschnitte
- Eine Leerzeile sollte vor jeder Methode, vor jeden Block- oder Einzeilenkommentar, zwischen logischen Abschnitten innerhalb einer Methode und zwischen den Variablendeklarationen und dem ersten Statement
- Leerzeichen nach jedem Schlüsselwort (for, if, while, usw.)
- Leerzeichen nach Kommas in Parameterlisten, vor und nach binären Operatoren und nach Cast-Konstrukten.
- Ausdrücke in for-Statements sollten durch ein Leerzeichen getrennt sein

5. Sun Konventionen

• Deklarationen

- Eine Zeile pro Deklaration
- Mischen von Datentypen vermeiden

```
int sum;  
int number;          // gut  
int sum, number;    // schlecht  
int sum, number[];  // schlecht
```

- Variablendeklarationen dadurch leichter auszukommentieren.
- Mehr als eine Deklarationen in einer Zeile kann zu Missverständnissen führen
- Variablen gleich bei der Deklaration initialisieren
- Deklarationen am Anfang von Blöcken, nicht mitten drin
- Deklarationen vermeiden, die zu Überdeckung führen
- Methodendeklaration mit einer Leerzeile voneinander trennen
- Bei Methodendeklaration kein Leerzeichen zwischen dem Methodennamen und der aufgehenden Klammer
- Öffnende geschwungene Klammern sollten in der selben Zeile stehen

5. Sun Konventionen

• Namenskonventionen

- Paketnamen:
 - In Kleinbuchstaben
 - Typischen Domainnamen: com, edu, gov, mil, net, org, de (ISO Standard 3166)
 - Sollten innerer Organisation entsprechen
`package com.sun.corba.se.extension;`
- Klassennamen:
 - Sollten Substantive sein
 - In Einzahl stehen
 - Anfangsbuchstabe groß
 - Jedes interne Worte groß
 - Kurz und prägnant
 - Vermeidung von Abkürzungen und Akronyme
 - Bei gebräuchlichen Akronymen wird nur erster Buchstaben groß geschrieben
`class HtmlViewer;`
`class XmlLoader;`
- Schnittstellennamen:
 - Wie Klassennamen
`interface Loader;`

5. Sun Konventionen

• Namenskonventionen (cont.)

• Methodennamen:

- Sollten Verben sein
- Erster Buchstabe klein, jedes interne Wort groß geschrieben
- Objektname im Methodennamen vermeiden
 - `employee.getName();` // nicht `employee.getEmployeeName();`
- Beim Zugriff auf Klassenattribute sollte man die Präfixe `get` und `set` verwenden
- Beim Abfragen von booleschen Variablen sollte man den Präfix `is` verwenden
`boolean isFinished();`
- Bei Funktionen, die etwas berechnen, hat sich der Präfix `compute` durchgesetzt.
`data.computeMean();`

• Variablennamen:

- Klein geschrieben, jedes interne Wort wird groß geschrieben
- Kurz, selbsterklärend und einprägsam
- Einzelne Buchstaben als Variablen vermeiden (Ausnahme: temporäre Variablen)
- Generische Variablen sollten den Namen ihres Typs tragen.
`void connect(Database database);` // gut

`void connect(Database db);` // schlecht
- Wenn möglich Rolle und Typ vereinen
`Name loginName;`
`Matrix rotationMatrix;`
- Konstanten werden komplett groß geschrieben

5. Sun Konventionen

- **Kommentare**

- Blockkommentare:

- Stehen in `/* ... */`
- Gehen über mehrere Zeilen
- Dienen dazu Dateien, Klassen, Datenstrukturen und Methoden zu beschreiben
- Sollten so ausgerichtet sein, wie der Code, den sie beschreiben
- Werden auch verwendet, um ganze Codeabschnitte auszukommentieren

- Zeilenkommentare:

- Beginnen mit `//` und enden mit dem Zeilenende
- Um einzelne Codezeilen zu beschreiben
- Um einzelne Zeilen Code auszukommentieren
- Stehen über oder hinter dem zu beschreibende Code
- Genügend Leerzeichen zum Abtrennen von Code und Kommentar
- Vor Kommentarzeile wird eine Leerzeile eingefügt

5. Sun Konventionen

- **Statements**

- Eine Zeile pro Statement verwenden

```
i = 0; j = 0; k = 0; DestroyBadLoopNames( i, j, k ); // schlecht
```

- Gründe:

- Veranschaulicht Komplexität besser
- Man muss nur von oben nach unten lesen
- Compilerfehler lassen sich leichter finden
- Lässt sich einfacher debuggen
- Mehrere Operationen pro Zeile oft schwieriger zu verstehen

5. Sun Konventionen

- **Statements (cont.)**

- For-Statement

```
for (initialization; condition; update) {  
    statements;  
}
```

- Do-While-Statement

```
do {  
    statements;  
} while (condition);
```

- While-Statement

```
while (condition) {  
    statements;  
}
```

- If-Statement

```
if (condition) {  
    statements;  
} else if (condition) {  
    statements;  
} else {  
    statements;  
}
```

5. Sun Konventionen

- **Referenzen:**

- Keine Objekte anlegen für statische Variablen oder Methoden

```
classMethod();           // gut
AClass.classMethod();   // gut
anObject.classMethod(); // schlecht
```

- **Konstanten:**

- Vermeiden von „Magic Numbers“ (Ausnahme: 1, 0, -1)

```
// gut
private static final int TEAM_SIZE = 11;
Player[] players = new Player[TEAM_SIZE];
// schlecht
Player[] players = new Player[11];
```

- **Klammern:**

- Verwenden von Klammern in logischen Ausrücken, auch bei eindeutiger Operatorpriorität (hilft unerfahrenen Programmierern)

```
if (a == b && c == d) // schlecht
if ((a == b) && (c == d)) // gut
```

6. Die Realität

- Jede größere Firma hat eigene Code-Konventionen
- Einzelne Projekte haben eigene Konventionen
- Vieles ist Geschmackssache

7. Zusammenfassung

- Vorteile:
 - Verbessern Lesbarkeit
 - Erleichtern Verständnis
 - Helfen Kosten zu sparen
 - Erhöhen Qualität
 - Leichteres Einarbeiten in fremden Code
 - Zwingen Programmierer zum sauberen Arbeiten
- Nachteile:
 - Keine einheitliche Regelung
 - Nicht resistent gegenüber „Mode-Erscheinungen“
 - Können zu „Religionskriegen“ führen
 - Führen manchmal zu unnötigen Einschränkungen
 - Wahrung des Aussehens zeitaufwendig

8. Literatur

- Sun: *Code Conventions for the Java Programming Language*.
<http://java.sun.com/docs/codeconv/>
- Netscape: *Netscape's Software Coding Standards Guide For Java*.
<http://www.csa.iisc.ernet.in/oldwebsite/Documentation/Tutorials/StyleGuides/netscape-codestyle.html>
- McConnell, Steven: *Code Complete. 2nd.* Microsoft Press, September 2002. – ISBN 1556154844
- Richard J. Miare, Juan A. N.: *Program indentation and comprehensibility*. In: *Communications of the ACM* (1993)