DHBW Duale Hochschule Baden-Württemberg Stuttgart Campus Horb

Vorlesung Programmieren
Thema 9: Dokumentation und Programmierstil

Olaf Herden
Fakultät Technik
Studiengang Informatik

:TRICKY:

javadoc

```
/*
 * Dies ist ein Kommentar.
 */
```

Stand: 01/2024

Gliederung

- Motivation/Einleitung
- Organisatorische Grundlagen
- Optik des Quellcodes
- Werkzeug `javadoc`
- Programmierstil

Motivation (I)

- Warum Programmierkonventionen (Code Conventions)?
  - Wartung und Weiterentwicklung von Software nicht ausschließlich von ursprünglichen Entwickler*innen
  - Maßnahmen zur besseren Lesbarkeit und Verständlichkeit notwendig
  - Eine hiervon: Programmierkonventionen
  - Leichtere bzw. schnellere Wartbarkeit rechtfertigt erhöhte Aufwendungen in Entwicklung
- Programmierkonventionen:
  - Offizielle Konventionen sollten als Basis dienen
  - Funktionieren nur, wenn sie von jedem/r Entwickler*in eingehalten werden
  - Möglichkeit projektspezifischer Erweiterungen

Motivation (II)

- Anmerkung: Konventionen sind (leider) keine Norm o.ä.
- Jedoch Reihe von:
  - Firmenspezifischen Konventionen
  - Herstellerspezifischen Dokumenten (z.B. Java Code Style von Oracle)
  - Allgemein anerkannten Konventionen

Lehman‘s Laws

- Reihe von Gesetzen/Aussagen über Software und deren Wartung
- Bekannteste:
  - Law of Uncertainty:
    - Software ändert sich während ihres gesamten Lebenszyklus. Genauere Angaben zum Gegenstand der Änderungen sind nicht vorhersehbar.
  - Law of Increasing Software Entropy:
    - Änderungen einer Software machen ihre Struktur in der Regel unverständlicher. Die innere Struktur einer Software geht mit fortschreitender Software-Lebenszeit verloren.
- Weitere Informationen:
  - https://en.wikipedia.org/wiki/Lehman%27s_laws_of_software_evolution
  - https://www.youtube.com/watch?v=T879CObr2rc
  - Herraiz, Israel; Rodriguez, Daniel; Robles, Gregorio; Gonzalez-Barahona, Jesus M. (2013). "The evolution of the laws of software evolution". ACM Computing Surveys. 46 (2): 1–28.

Empirisches zur Wartung

- Verhältnis Aufwand bis Ersteinführung (d.h. Entwicklungsaufwand) zu Aufwand der Wartung etwa 1:4 bis 1:5
- D.h. 80% der Softwarekosten in Wartungsphase
- Senkung möglich, wenn Software gut wartbar, d.h. leichtere Realisierung Änderungen/Erweiterungen
- Ziel: Senkung Wartungskosten durch Anwendung von Programmierkonventionen und guter Dokumentation während der Entwicklung
- Weitere Maßnahmen zur Erhöhung der Wartbarkeit:
  - Dokumentierte Ergebnisse der objektorientierten Analyse und des Entwurfs (z.B. in Form von UML-Diagrammen)
  - Verzicht auf „Trick“-Programmierung

Maßnahmen für gute Wartbarkeit (I)

- Organisatorischer Rahmen:
  - Struktur/Kultur/Planung
  - Einsatz von Werkzeugen, z.B. IDE, StyleChecker, …
  - Maßnahmen, z.B. Audit, Walk Through, …
- Organisation im Projekt:
  - Aufbau von Paketen, Klassen, Übersetzungseinheiten
- Code-Konventionen:
  - Namenskonventionen
  - Klammerungen
  - Einrückungen
  - Kommentare

Maßnahmen für gute Wartbarkeit (II)

- Regeln:
  - Tipps und Tricks
  - Einhalten Objektorientierung
  - Vermeiden von Hacks
- Entwurfsmuster (siehe VL Software Engineering)

Übersicht

- Motivation/Einleitung
- Organisatorische Grundlagen
- Optik des Quellcodes
- Werkzeug `javadoc`
- Programmierstil

Organisation/Pakete

- Quellcode und ausführbaren Code in separaten Verzeichnissen
- Ebenso eingebundene Bibliotheken

- Pakete:
  - Eigenes Paket für jede funktional zusammengehörende Menge von Klassen
  - Bei großen Paketen Strukturierung mit Unterpaketen
  - Anlegen einer Datei `index.html` mit Beschreibung von Nutzen und Struktur des Paketes

Organisation einzelner Dateien

- Keine Dateien mit mehr als 2000 Zeilen
- Abschnitte durch Leerzeilen trennen
- Pro Datei nur eine öffentliche Klasse bzw. ein öffentliches Interface (geht in Java auch nicht anders)
- Wenn mehrere Klassen/Interfaces in einer Datei, dann öffentliche Klasse/ öffentliches Interface am Dateianfang
- Datei hat vorgegebene Aufteilung:
  - Anfangskommentare
  - `package`- und `import`-Anweisungen
  - Klassen- und Interface-Deklarationen
- Klasse mit `main`-Methode enthält nur diese und steht in separatem Paket
- Grund: Bessere Modularität und Wiederverwendbarkeit

Dateistruktur (I)

- Anfangskommentare in folgendem Format:

```
/*
 * Klassenname
 *
 * Versionsinformation
 *
 * Copyright-Vermerk
 *
 */
```

- `package`- und `import`-Anweisungen

```
package <xy>;
import  <x>.<y>;
```

Dateistruktur (II)

- Klassen- und Interfacedeklarationen in vorgegebener Reihenfolge:
  - Klassen- bzw. Interface-Kommentar (`/** ... */`)
  - `class-` bzw. `interface-`Definition
  - Klassen- bzw. Interface-Implementierungskommentar, falls nötig (`/* ... */`)
  - Attributdeklaration:
    - Klassenattribute (`static`)
    - Instanzattribute
    - Jeweils mit Zeilenendekommentar (falls notwendig)
  - Methoden:
    - Konstruktoren
    - Methoden
      - Nach Zugriffsrechten geordnet (`public`, `protected`, paketsicher (d.h. ohne Schlüsselwort), `private`)
      - Innerhalb der Zugriffsrechte alphabetische Sortierung

Übersicht

- Motivation/Einleitung
- Organisatorische Grundlagen
- Optik des Quellcodes
- Werkzeug `javadoc`
- Programmierstil

Namenskonventionen (I)

- Grundsätzlich:
  - Alle Bezeichner sollten sprechend und selbsterklärend sein
  - Ausnahme: Schleifenzähler (hier üblich: i, j, k, ...)
- Paketnamen:
  - Substantive im Singular, klein geschrieben, z.B.

```
package mathematik;
package personal;
```

- Klassennamen:
  - Substantive im Singular mit großem Anfangsbuchstaben, ganze Worte, innere Worte beginnen mit Großbuchstaben, z.B.

```
class Datum{ ... };
class KundenListe{ ... };
```

Namenskonventionen (II)

- Interfacenamen:
  - Adjektive mit großem Anfangsbuchstaben, ganze Worte, innere Worte beginnen mit Großbuchstaben, z.B.

```
interface Sichtbar{ ... };
interface Invertierbar{ ... };
```

- Methodennamen:
  - Mit Verb beginnend, klein geschrieben, innere Worte beginnen mit Großbuchstaben
  - Beispiele:

```
void print() { ... };
int berechneNaechstenWert(...) { ... };
```

  - Methoden für Kapselung sollten mit Präfix get bzw. set beginnen, gefolgt vom zu ändernden Inhalt, z.B.

```
void setName(...) { ... };
String getName() { ... };
```

Namenskonventionen (III)

- Methoden mit Rückgabetyp `boolean` sollten mit Präfix is oder has beginnen, z.B.

```
boolean isEqual(Auto a, Auto b) { ... };
```

- Keine Methode (außer Konstruktoren) darf den Namen einer Klasse haben
- Bei Substantiven normalerweise Singular wählen
- Ausnahme: Bei Kollektionen (Feldern, Vektoren, etc.) wähle Plural, z.B. `besuchteFelder[]`

- Attributnamen:
  - Wortanfang klein geschrieben, innere Worte beginnen mit Großbuchstaben
  - Namen sollen selbsterklärend sein, z.B.

```
Datum lieferDatum = new Datum("19.01.2024");
```

statt

```
Datum d = new Datum("19.01.2024");
```

Namenskonventionen (IV)

- Konstanten:
  - Werden komplett groß geschrieben, Worte durch "_" voneinander getrennt
  - Beispiel:

```
final static int MAXIMALE_ANZAHL = 1000;
```

- Lokale Variable:
  - Gleiche Regeln wie für Attribute
  - Typische Namen für häufig vorkommende Zwecke:
    - Datenströme:
      - `in` und `out` bzw. `inOut` bei kombiniertem Strom
      - Alternativ: `inputStream`, `outputStream` bzw. `ioStream`
    - Schleifenvariablen:
      - Verwende `i`, `j`, `k`, … nicht `lauf1`, `lauf2`, … o.ä.

Namenskonventionen (V)

- Ausnahmeobjekte:
  - Verwende `e`
- Temporäre Variablen dürfen mit einem Buchstaben benannt werden, wobei gilt:
  - `i, j, k, m` und `n` für `integer` Variable
  - `c, d` und `e` für `char` Variable

Gotcha Schlüsselwörter

- Hervorhebung besonderer Quelltext-Eigenschaften zum schnellen Auffinden
- Etablierung sog. Gotcha-Schlüsselwörter innerhalb von Kommentaren

Wort	Bedeutung
`:TODO: topic`	Programmstelle muss noch bearbeitet werden
`:BUG: [bugid] topic`	Bekannter Fehler, muss beschrieben werden, evtl. mit Fehlernummer (Bug-ID) versehen
`:KLUDGE:`	Code an dieser Stelle auf die Schnelle entstanden, muss noch bearbeitet werden
`:TRICKY:`	Lösung ist für Dritten nicht sofort nachvollziehbar, sollte daher erläutert werden
`:WARNING:`	Gibt Warnhinweise auf irgendwelche Nachteile (z.B. hoher Speicherplatzbedarf oder lange Laufzeit bei großen Parametern)
`:COMPILER:`	Code ist Workaround aufgrund eines Compiler- bzw. Bibliotheksfehlers

Kommentare (I)

- Block-Kommentare mit führendem * in folgender Form:

```
/*
 * Dies ist ein Kommentar.
 */
```

- Einzeilenkommentare vor entsprechendem Programmcode mit zugehöriger Einrückung
- Leerzeile vor dem Kommentar
- Beispiel:

```
if (Bedingung) {

   /* Bedingung wird ausgewertet */
   ...
}
```

Kommentare (II)

- Zeilenendekommentare (/ /) am Ende der Zeile genügend einrücken, z.B.

```
(1) if (n > 0){
(2)   return fak_rek(n-1);     // Rekursiver Aufruf
(3) }
(4) else{
(5)   return 1;                // Fakultät von 0 = 1
(6) }
```

statt

```
(1) if (n > 0){
(2)   return fak_rek(n-1);// Rekursiver Aufruf
(3) }
(4) else{
(5)   return 1;// Fakultät von 0 = 1
(6) }
```

Kommentare (III)

- Keine Zeilenkommentare für mehrzeilige Kommentare verwenden, z.B.:

```
(1) i = 6;
(2) /* Im folgenden Programmstück
(3)  * wird das Feld f invertiert
(4)  * und alle Werte transformiert
(5)  */
```

statt

```
(1) i = 6;
(2) // Im folgenden Programmstück
(3) // wird das Feld f invertiert
(4) // und alle Werte transformiert
```

Struktur Quelltext

- Einrückung um feste Breite (2-4 Zeichen)
- Pro Zeile maximal 60-80 Zeichen (wegen Lesbarkeit im Editor)
- Regeln für Zeilenumbrüche:
  - Umbruch auf höchstmöglicher Ebene
  - Umbruch vor Operatoren und nach Kommata
  - Gleiche Hierarchieebenen sollen möglichst in folgender Zeile untereinander stehen, falls nicht möglich, Einrückung um 8 Zeichen
  - Beispiel:

```
int i = f(3+j,
          g(5,
            k+3));
```

Deklarationen (I)

- Nur eine Deklaration pro Zeile, z.B.:

```
int untergrenze;          // Linke Intervallgrenze
int obergrenze;           // Rechte Intervallgrenze
statt
int untergrenze, obergrenze;
```

- Ausnahme bilden hier z.B. mehrere Zählvariablen, erlaubt ist z.B.:

```
int i,j,k;
```

- Keine Deklarationen unterschiedlichen Typs oder Methoden- und Attributdeklarationen in einer Zeile, z.B.:

```
long kundennummer, getKundennummer();
String name, namensliste[];
```

Deklarationen (II)

- Felddeklarationen:

```
<Typ>[] meinFeld;
```

statt

```
<Typ> meinFeld[];
```

verwenden

- Position im Quellcode:
  - Deklarationen immer am Anfang eines Blocks
  - Nicht bis zur ersten Benutzung warten
- Lokale Variablen bei Deklaration auch initialisieren

Deklarationen (III)

- Überdeckungen von Variablen einer höheren Ebene vermeiden (in Java sowieso nicht erlaubt):

```
(1) int i = 1;
(2) ...
(3)   {
(4)     int i = 5;
(5)     ...
(6)   }
(7) ...
```

Ausdrücke (I)

- Ein Ausdruck pro Zeile, z.B.

```
i++;
j++;
```

statt

```
i++; j++;
```

- Komplexe Ausdrücke: Verdeutlichung Auswertungsreihenfolge durch redundante Klammerungen (und evtl. Einrückungen), z.B. statt

```
(b1 && !b2 || b3 && b4)
```

besser

```
(  (b1 && (!b2))
 ||(b3 && b4))
```

Ausdrücke (II)

- Leerzeichen „richtig“ setzen, z.B. suggeriert

```
i = 2* a+b
```

  Ausführung Addition vor Multiplikation

- Konstanten immer über Namen einbauen (außer bedeutungslosen wie z.B. 0, 1, -1)

- Mehrfachzuweisungen z.B.

```
i = j = k;
```

  vermeiden

Ausdrücke (III)

- Unübersichtliche Ausdrücke vermeiden, z.B.

```
j = i-- + ++i + i++ - --i;
```

- Vermeidung durch Aufteilung, Klammerungen oder Vereinfachung

- Methoden nicht „zu lang“ werden lassen:
  - Im Zweifel Aufteilung auf zwei Methoden
  - Generelle Angabe, was „zu lang“ ist, ist schwierig:
    - Lange `switch`-Anweisung z.B. eher unkritisch, da diese trotz ihrer Länge immer noch relativ übersichtlich ist
    - Tief geschachtelte `if`-Anweisung kann schnell unübersichtlich werden

Importe

- Importe möglichst genau halten, z.B. statt

```
import <Paket1>.*
import <Paket2>.<Unterpaket>.*
```

besser

```
import <Paket1>.<Klasse1>
import <Paket1>.<Klasse2>
import <Paket1>.<Klasse3>
import <Paket2>.<Unterpaket>.<Klasse1>
import <Paket2>.<Unterpaket>.<Klasse2>
```

- Grund: Bessere Lesbarkeit bzgl. Kontext und Abhängigkeiten

Entfernen alten Codes

- Auskommentierten, alten Code entfernen, z.B.

```
(1) private boolean inField(int i){
(2)    return ((i>=0) && (i<this.size));
(3) }
```

statt

```
( 1) private boolean inField(int i){
( 2)   return ((i>=0) && (i<this.size));
( 3)   /*
( 4)   if ((i>=0) && (i<=this.size)){
( 5)     return true;
( 6)   }
( 7)   else{
( 8)     return false;
( 9)   }
(10)  */
(11) }
```

Übersicht

- Motivation/Einleitung
- Organisatorische Grundlagen
- Optik des Quellcodes
- Werkzeug `javadoc`
- Programmierstil

Werkzeug `javadoc`

- Werkzeug `javadoc` erzeugt HTML-Dokumentation
- Berücksichtigung von Dokumentationskommentaren (von `/**` bis `*/`) vor Klassen- und Interface-Deklarationen sowie vor Methoden, Konstruktoren und Attributdeklarationen
- Aufnehmen Kommentare in HTML-Dokumentation (HTML-Tags im Kommentar sind erlaubt)
- Erster Satz des Kommentars gesonderter Zusammenfassungssatz (Punkt als Endmarkierung)
- Erzeugung automatischer Querverweise durch @-Zeichen innerhalb eines Kommentars

Tagging (I): @see

- @see-Tag erzeugt allgemeinen Querverweis
- Referenz zu anderer Klasse:
  - Syntax: `@see <Klassenname>`
  - Beispiel: `@see String`
- Referenz zu Methode einer Klasse:
  - Syntax: `@see <Klassenname>#<Methodenname>`
  - Beispiel: `@see String#equals`
- Referenz zu anderer HTML-Datei:
  - Syntax: `@see <Verweis auf HTML-Seite>`
  - Beispiel: `@see <a href=<Dateiname>><Text></a>`

Tagging (II): @author, @version

- `@author`-Tag um Autor zu kennzeichnen:
  - Syntax: `@author <Name>{, <Name>}`
  - Beispiele:
    - `@author Frank Meier`
    - `@author Frank Meier, Steffi Hansen`

- `@version`-Tag kennzeichnet Version
  - Syntax: `@version <Versionsnummer>`
  - Beispiel: `@version 3.03`

Tagging (III): @param, @return

- @param-Tag: Erläuterung Parameter in Methodendeklarationen:
  - Syntax: `@param <Parametername> <Beschreibung>`
  - Beispiel: `@param z Zinssatz`

- @return-Tag: Beschreibung des Rückgabeparameters:
  - Syntax: `@return <Beschreibung>`
  - Beispiel: `@return Wert entspricht der Fakultät, bei negativem Parameter wird 0 zurückgegeben`

Tagging (IV): @exception, @deprecated

- `@exception`-Tag: Erklärung von Ausnahmen in Methoden:
  - Syntax: `@exception <Ausnahme> <Beschreibung>`
  - Beispiel:
```
@exception IndexOutOfBoundsException
           Die Feldgrenze wird überschritten
```

- `@deprecated`-Tag: Kennzeichnung veralteter Methoden
  - Syntax/Beispiel: `@deprecated`

Anwendung (I)

- Aufruf: `javadoc [Optionen] [Klassen]`
- Wichtigste Optionen (insgesamt ca. 50-60, siehe Dokumentation):
  - `-classpath`: Gibt einzubindede Verzeichnisse an
  - `-sourcepath`: Gibt das Verzeichnis der Quelldateien an
  - `-d`: Gibt das Zielverzeichnis an
  - Angabe welche Klassen in die Dokumentation einbezogen werden:
    - `-public`
    - `-protected` (default)
    - `-package`
    - `-private`

Umfassend in diese Richtung, d.h. z.B. `package` umfasst auch alle `protected` und `public` Deklarierten Klassen

Anwendung (II)

- Beispiel:

  `javadoc –sourcepath ./p –d ../Doku –public *.java`

  generiert
  - Dokumentation aller java-Dateien aus dem Unterverzeichnis p
  - Dokumentation wird ins Verzeichnis ../Doku geschrieben
  - Umfasst nur als `public` gekennzeichnete Klassen

Übersicht

- Motivation/Einleitung
- Organisatorische Grundlagen
- Optik des Quellcodes
- Werkzeug `javadoc`
- Programmierstil

Allgemeines

- Ziele: Code sollte
  - Gut lesbar sein
  - Einfach verständlich sein
  - Sicher sein
  - Usw.
- Dazu: Notwendigkeit einiger allgemeiner Richtlinien
- Hier: Kleine Auswahl wichtiger Regeln

Wichtige Schnittstellen

- Schnittstellen `Cloneable`, `Serializable` und `Comparable` in Java von zentraler Bedeutung
- Genaue Überprüfung Implementierung für jede Klasse

Zugriffsmodifier (I)

- Instanzvariable niemals als `public` deklarieren
- Grund: Einhalten des Kapselungsprinzips
- Lösung:
  - Instanzvariablen als `private` deklarieren
  - Zugriff über `get`- und `set`-Methoden

Zugriffsmodifier (II)

- Innerhalb einer Klasse kein direkter Zugriff auf Instanzvariable
- Stattdessen: Verwendung `get`- und `set`-Methoden
- Bsp.:

```
( 1) public class A{
( 2)   private int x;
( 3)   ...
( 4)   public void f(){
( 5)     this.setX(...);       // ok
( 6)     this.x = ...          // Vermeiden!
( 7)     ...
```

Statische Variable

- Anzahl statischer Variablen (Klassenvariablen) minimieren (Ausnahme: mit `static final` deklarierte Konstanten)
- Grund:
  - Klassenvariablen haben Charakter globaler Variabler in Nicht-OO-Sprachen
  - Methoden werden kontext-abhängiger, Seiteneffekte werden versteckt
  - Insgesamt: Code wird unverständlicher, schwerer wartbar, etc.

Datentypen

- Passenden Datentyp wählen, keine Codierungen
- Beispiel: `boolean flag` statt `int flag` und nun der Variablen `flag` nur die Werte `0` und `-1` zuweisen

- Wenn möglich, besser `long` statt `int` und `double` statt `float` verwenden
- Grund:
  - Über- bzw. Unterlauf erheblich unwahrscheinlicher
  - Ebenso Präzisionsprobleme bei Gleitkommatypen
- Gegenargumente:
  - Manchmal anderer Typ notwendig, z.B. `int` bei Felddimensionierung
  - Speicherplatz bei limitierten Ressourcen

Weitere Regeln

- Keine Trick-Programmierung, z.B. kein Shift um zwei Positionen anstelle der Multiplikation mit 4

- Keine Wertzuweisung in Bedingungen bei `if`- oder `while`-Anweisungen
- Grund:
  - Meistens sowieso Fehler (gemeint ist ==)
  - Schlechte Nachvollziehbarkeit

- Zuweisung `null` an nicht mehr benötigte Objekte
- Gründe:
  - Gute Lesbarkeit („Objekt wird hier nicht mehr benötigt“)
  - Garbage Collection wird ermöglicht

Zusammenfassung (I)

- Motivation/Einleitung:
  - Lehman's Laws
  - Wartungsphase
- Organisatorische Grundlagen:
  - Pakete
  - Dateien
  - Dateistruktur

Zusammenfassung (II)

- Optik des Quellcodes:
  - Namenkonventionen
  - Gotcha Schlüsselwörter
  - Kommentare
  - Struktur Quelltext
  - Deklarationen
  - Ausdrücke
  - Importe
  - Alten Code

Zusammenfassung (III)

- Werkzeug `javadoc`:
  - Werkzeug
  - Tagging
  - Anwendung

- Programmierstil:
  - Wichtige Schnittstellen
  - Zugriffsmodifier
  - Statische Variablen
  - Datentypen
  - Weitere Regeln

Aufgaben (I)

- **Aufgabe 1**
  Kopieren Sie Ihre Lösung aus Aufgabe 1 von Übung 8 (Paket `personal`), überprüfen Sie ihren Code auf die Einhaltung von Programmierrichtlinien, erweitern Sie Ihre Lösung um Dokumentationskommentare und erstellen Sie mit `javadoc` eine HTML-Dokumentation.
- **Aufgabe 2**
  Im Moodle-Kurs befindet sich eine Datei `fibonacci.java`.
  Finden sie möglichst viele Punkte, in denen der (lauffähige!) Quelltext gegen die Java Code Conventions verstößt, und markieren sie diese im Quelltext.

Aufgaben (II)

- **Aufgabe 3**

  Zur Untersuchung (optischer Aspekte) von Quellcode stehen eine Reihe von Programmen zur Verfügung, so z.B. das Programm CheckStyle, das Quelltext auf den Programmierstil überprüft und die Regeln der Java Code Conventions beherrscht.

  Laden Sie das Programm CheckStyle herunter, installieren Sie es, und überprüfen Sie den obenstehenden Quelltext.

  Sie finden das Programm unter https://checkstyle.org/.

  Neben dem eigentlichen Programm gibt es auf der Seite auch Beschreibungen und Hinweise über Plugins für gängige IDE.