
Bad Smells and more

Michael Inden

Table of Contents



- **Basics: Why This Talk?**
- **What are Bad Smells?**
- **What are Characteristics of Good Software?**
- **Internal Quality**
- **Q & A**

Basics: Why This Talk?

Basics: Why this talk?



- **Im Programmierhimmel und am perfekten Tag ...**

- Code ist schön, effizient und funktioniert
- Code ist lesbar und verständlich
- Fällt es leicht, neue Features zu integrieren
- Muss man nichts testen, da man ja keine Fehler macht ;-)
- Funktioniert alles gleich von Anfang an
- Macht alles viel Spaß



- **Aber mal ehrlich ...**

- **Wir kennen es anders aus der täglichen Arbeit**
 - Coding kann toll sein, wenn man was ganz Neues anfängt
 - Coding wird frustrierend, wenn man auf hässliche Stellen stößt
 - Fehler zu finden ist schwierig
 - **Und schlimmer noch, das Beheben ist meistens noch schwieriger**
 - **Maintenance** wird schnell zum **Albtraum**



-
- **Wie kommen wir aus der Wartungsabtraum-Falle?**
 - **Nicht sofort vom Gehirn in die Tastatur hacken**
 - **Problem verstehen, Lösung (grob) designen**
 - Entwickle in kleinen Schritten (XP, TDD: test, code, refactor)
 - Prüfe die Realisierung durch Unit Tests
 - Mehr interne QA => Verantwortung nicht an Testabteilung abgeben
 - **Ganz wichtig: Gutes Wissen über Bad Smells und Fallstricke, das JDK, externe Bibliotheken usw.**

-
- **Befolge Grundregeln und trainiere gute Angewohnheiten**
 - Aussagekräftige Namen
 - Geeignete Algorithmen und Abstraktionen
 - Dokumentiere die *richtigen* Dinge (aber keine Über-/Pseudo-Doku, bitte)
 - Lies Fachbücher und diskutiere mit Kollegen
 - Sei aufmerksam und vorsichtig, aber offen für neue Impulse und selbstkritisch
 - **Verbessere deine Fähigkeiten, guten Code zu schreiben**
 - Klar und lesbar
 - **Verständlich** und frei von Redundanzen

Basics: Why this talk?

Tricky Assignment



- Was machen diese paar Zeilen?

```
int trickyPre = 0;
int trickyPost = 0;

for (int i = 0; i < 50; i++)
{
    trickyPre += ++trickyPre;
    trickyPost += trickyPost++;
}

System.out.println("trickyPre = " + trickyPre + " / " +
    "trickyPost = " + trickyPost);
```

- Wie lautet die Ausgabe???

0, 1, 49, 50, Exception?

Basics: Why this talk?

Tricky Assignment



- Das Ergebnis ist

```
trickyPre = -1 / trickyPost = 0
```

- Was? Wieso?

| | | |
|----------------------------------|--------------------|--|
| <code>tricky += ++tricky;</code> | <code>=></code> | <code>tricky = tricky + ++tricky;</code> |
| <code>tricky += tricky++;</code> | <code>=></code> | <code>tricky = tricky + tricky++;</code> |

- Was genau macht?

`++tricky` bzw. `tricky++;`

- Unterschiedliche Semantik (**wer kennt sie?**)

Basics: Why this talk?

Tricky Assignment



- erst zählen dann nutzen bzw. erst nutzen dann zählen

```
++tricky
```

```
tricky = tricky + 1;
```

```
tricky++;
```

```
temp = tricky;  
tricky = tricky + 1;  
return temp
```

- Dann gilt also

```
trickyPre += ++trickyPre;  
trickyPost += trickyPost++;
```

```
=> Wert wird hochgezählt  
=> Wert bleibt 0
```

- ABER: Wie kriegen wir -1 als Ergebnis, wenn wir hochzählen?

Basics: Why this talk?

Tricky Assignment – Conclusion



- Machen wir ein wenig System.out-Debugging:

```
...  
i = 6 / trickyPre = 127 / trickyPost = 0  
  
...  
i = 29 / trickyPre = 1073741823 / trickyPost = 0  
i = 30 / trickyPre = 2147483647 / trickyPost = 0  
i = 31 / trickyPre = -1 / trickyPost = 0  
i = 32 / trickyPre = -1 / trickyPost = 0
```

Basics: Why this talk?

Tricky Assignment – Conclusion



- **ACHTUNG: SILENT OVERFLOW IN JAVA**

```
Integer.MAX_VALUE + 1 == Integer.MIN_VALUE  
2147483647 + 1 == -2147483648
```

```
i=31 2147483647 + (2147483647 + 1) = 2147483647 - 2147483648 = -1  
i=32 -1 + (-1 + 1) = -1 + 0 = -1 ...
```

- Was haben wir gelernt?
 - *kleine* Änderungen **GROSSE** Auswirkungen
 - Stille Überläufe => merkwürdige Ergebnisse

- **Semikolon an falscher Position oder fehlende { } ändern Verhalten massiv:**

```
if/while(condition);    if (condition)
{                        doSomething();
    doSomething();       doOther();
}
```

- **Beginnen wir die weitere Entdeckungsreise ...**



What are Bad Smells?

Think about it ...

What are Bad Smells?

Bad Smells sind Code-Abschnitte des Codes, die...

- Potenziell Fehler enthalten
- Irreführend oder Missverständlich
- Irgendwie verdächtig
- Ein komisches, ungutes Gefühl in der Magengegend geben
- Furcht auslösen, wenn man dort erweitern soll



⇒ Hässlich und hartnäckig gegen Änderungen

⇒ Das Schlimmste: Der Code ist nicht verlässlich

Bad Smells – A first example



- Zwei Zeilen Code können doch nun wirklich nicht so schlimm sein, oder?

```
CmdExe ce = new CmdExe(4711);  
ce.reg(new Printer("Hi Bad Smell World"));
```


Bad Smells – A first example

3 or more possible problems



- **Irreführende Namen** `CmdExe`, `Printer`
- **Abkürzungen mit keiner oder kaum Bedeutung** `ce`, `reg`
 - Nahezu keine Informationen über die Semantik
 - Generell sind Abkürzungen eher irritierend (nicht nur für Project Newbies)
- **Magic Number** `4711`
 - Wieder keine Semantik
 - Schwierig zu beurteilen, ob gültig, oder nicht
 - Könnte außerhalb des erlaubten Bereichs sein, was dann?
 - Exception? Falsches Ergebnis oder keine Ausführung?

Bad Smells – A first example

Final Correction



```
CommandExecutor executor = new CommandExecutor(ADD_AS_LAST);  
executor.register(new PrintToConsole("Hi Bad Smell World"));
```

- **Was haben wir erreicht?**
 - besser lesbar und verständlich
 - robust und besser verlässlich
 - transportiert Semantik (drückt aus, was passiert)
 - Vorhersagbares Verhalten selbst in Fehlersituationen
 - leichter zu warten und zu erweitern

Bad Smells – A first example

Bad error message(s)



- Ist die Warnungsmeldung **"value out of range"** hilfreich?
- Man kennt und ärgert sich über solche Mitteilungen:
Schlechte Programme: "Unexpected error 1234 occurred."
Deutsche Bahn: "Wir haben unerwartet gehalten."
- **Abhilfe: Kommuniziere Fehlersituationen klar und stelle nützliche Informationen für Aufrufer bereit**

Bad Smells – A first example

Communicate errors clearly



- **3 Tipps für hilfreiche Warnmeldungen**

1. **Nachricht sollte den erlaubten Wertebereich enthalten**

```
"parameter XYZ is not in range [0-2]"
```

2. **Nachricht sollte den momentanen Wert zeigen**

```
"parameter XYZ is invalid: value = " +  
registrationStrategy + " is not in range [0-2]"
```

3. **Und eine Liste gültiger Werte liefern**

```
List<Integer> VALID_VALUES = Arrays.asList(0, 1, 2, 4, 7);  
"parameter 'XYZ' is invalid: value = " + XYZ  
+ " is not in range " + VALID_VALUES);
```

=> parameter 'XYZ' is invalid: value = 6 is not in range [0, 1, 2, 4, 7]

Characteristics Of Good Software

Martin Fowler says ...

“Any fool can write a program a computer can understand,
Good programmers write code that humans can understand.”

Internal Quality bestimmt durch CODE:

- Lesbarkeit
- Verständlichkeit
- Keine Fallstricke und Hindernisse
- Erweiterbar, pflegbar
- Gut/sinnvoll dokumentiert
- Gute Test/Code Coverage
- ...



External Quality entspricht Benutzersicht (**FEATURES** und **BEHAVIOUR**):

- Arbeitet wie erwartet
- Stellt alle benötigte Funktionalität bereit
- Korrektheit
 - Nahezu keine (beobachtbaren) Bugs
 - Gut getestet
- Benutzbar
- Verlässlich
- ...



Good Software – Common design principles



Common design principles

1. KISS – Keep It Simple and Short
2. DRY – Don't Repeat Yourself
3. LoD – Law of Demeter
4. SOLID – 5 Principles

| | |
|----------|---|
| S | – SRP – Single Responsibility Principle |
| O | – OCP – Open/Closed Principle |
| L | – LSP – Liskov Substitution Principle |
| I | – ISP – Interface Segregation Principle |
| D | – DIP – Dependency Inversion Principle |

1. Shortness – KISS
2. Meaningful names
3. Structure Of Code
4. Find proper abstractions
5. Avoid side effects and other surprises

Internal Quality In Depth

Internal Quality in Depth

1. Shortness – KISS



- **KISS – Keep it simple and short**
- **SRP – Single Responsibility Principle** besagt
 - methods and classes should have (only) one responsibility
 - methods and classes should be as short or simple as possible
- **Faustregeln**
 - Methoden => max. 50 – 100, besser < 10 – 20 Zeilen
 - Klassen => max. 1000 – 2000, besser maximal um die 500 Zeilen



Internal Quality in Depth

Gute Bekannte ... Wölfe im Schafspelz



Sind Fluent Interfaces gut?

Sind Builder wirklich immer gut?

Sind sehr kurze Methoden gut?



```
new PizzaBuilder().mitSalami().small().bitteBeachten("Ohne Mais!") .create();  
=> "Kleine Pizza Salami ohne Mais "
```

Internal Quality in Depth

! KISS – Fluent **but** unreadable Code



- Wir wollen (z.B. in JavaFX) eine Berechnung anstellen
- Es existiert ein Fluent Interface mit `add()`, `multiply()` usw.

```
x.add(y).multiply(z).subtract(7).divideBy(2)
```

- Es ist echt gut zu lesen, aber nicht gut zu verstehen
- Was ist die Semantik?

$(x + (y * z) - 7) / 2$ oder $(x + y) * z - (7 / 2)$...

Internal Quality in Depth

! KISS – Builder **but** irritating Code



// Wir wünschen folgende Ausgabe: 13.03.2014 17:41:22

- **SCHLECHT**

```
final DateTimeFormatter OUTPUT_TIMESTAMP_FORMATTER =  
    new DateTimeFormatterBuilder()  
        .appendDayOfMonth(2) .appendLiteral('.').appendMonthOfYear(2) .appendLiteral('.')  
        .appendYear(4, 4).appendLiteral(' ').appendHourOfDay(2).appendLiteral(':')  
        .appendMinuteOfDay(2).appendLiteral(':') .appendSecondOfMinute(2).toFormatter();
```

- **DEUTLICH BESSER**

```
final SimpleDateFormat sdfInput = new SimpleDateFormat("dd.MM.yyyy HH:mm:ss");
```

Internal Quality in Depth

! KISS – Unused Code



```
private static String getEncoding(final String frequency) {  
    String value = "";  
    if (frequency.startsWith("M")) {  
        return "710";  
    } else if (frequency.startsWith("Q")) {  
        return "708";  
    // } else if (frequency.startsWith("H")) {  
    // return "704";  
    // } else if (frequency.startsWith("T")) {  
    // // trimester?  
    } else if (frequency.startsWith("A")) {  
        return "702";  
    // } else if (frequency.startsWith("D")) {  
    // return "711";  
    } else if (frequency.startsWith("W")) {  
        return "716";  
    }  
    return value;  
}
```


Internal Quality in Depth

1. Shortness – KISS



- Code sollte knackig und präzise sein
- ?-Operator ??? Damit gilt doch folgendes, oder?

```
if (condition)
{
    result = success;
}
else
{
    result = other;
}
```

=>

```
result = (condition) ? success : other
```

Internal Quality in Depth

1. Shortness – KISS



- Für einfache Bedingungen und Auswertungen hilfreich

$(x \% 2 == 0) ? \text{"even"} : \text{"odd"}$

- ABER: Was ist hiermit?

```
Double value = value1 == null ? value2 : value2 == null ? value1  
: new Double(value1 + value2);
```

- Das ist kurz, aber unverständlich – zumindest für mich!
- Wer hilft mir? Weiß jemand, was dieser Ausdruck macht?
- **Bad Smell: Complex logic in conditional operator**

Internal Quality in Depth

1. Shortness – KISS



- **Prio 1 vor Kürze: Verständlichkeit**

```
Double value = value1 == null ? value2 : value2 == null ? value1  
: new Double(value1 + value2);
```

- **Wie wäre es mit dieser Alternative?**

```
public static Double nullsafeAdd(Double value1, Double value2)  
{  
    if (value1 == null)  
        return value2;  
    if (value2 == null)  
        return value1;  
  
    return value1 + value2; // new Double(value1 + value2);  
}  
Double value = nullsafeAdd(value1, value2);
```

Internal Quality in Depth

1. Shortness – KISS



```
public static Double nullsafeAdd(Double value1, Double value2)
{
    if (value1 == null && value2 == null)
        return null;
    if (value1 == null)
        return value2;
    if (value2 == null)
        return value1;

    return value1 + value2;
}
```

- **Behandelt nun alle Spezialfälle direkt am Anfang**
- **Kommuniziert klar und deutlich und ist intuitiv verständlich**
- **That's KISS ...**

Internal Quality in Depth

2. Meaningful names



- Schwierig zu finden
- Sind die Mühe wert, vor allem für später
- Spannungsfeld: nicht zu lang aber aussagekräftig
- Vermeide Abkürzungen wie AAA und andere Batterien ;-)
- Okay sind gängige Abkürzungen wie Db, Html, Xml. In diesem Fall helfen Abkürzungen, den Namen kürzer zu halten



Internal Quality in Depth

2. Meaningful names



- Was machen diese Zeilen?

```
// contains max value  
int val = -1;
```

```
// iterate through all available table rows  
for (int i = 0; i < 50; i++)  
{  
    val = Math.max(val, values[i].getValue());  
}
```

- Was kann verbessert werden??

Internal Quality in Depth

2. Meaningful names



- Die Zeilen kommunizieren nicht, was sie tun
- Die Kommentare sind nahezu überflüssig (!DRY)

```
// contains max value  
int val = -1;
```

```
// iterate through all available table rows  
for (int i = 0; i < 50; i++)  
{  
    val = Math.max(val, values[i].getValue());  
}
```

Internal Quality in Depth

2. Meaningful names – DRY



- Versuchen wir es mal mit sprechenden Namen ...

```
final int PERSON_TABLE_ROW_COUNT = 50;
```

```
int maxAge = -1;  
for (int rowIndex = 0; rowIndex < PERSON_TABLE_ROW_COUNT; rowIndex++)  
{  
    maxAge = Math.max(maxAge, persons[rowIndex].getAge());  
}
```

- Intention nun ziemlich klar und verständlich
- Wir ermitteln das maximale Alter von den Personen aus der Liste

Internal Quality in Depth

2. Meaningful names for Collections



- **SCHLECHT**

- Name besteht nur aus dem Datentyp: map, set, list, vector

```
final List<File> list = new ArrayList<File>();  
final List<File> list2 = new ArrayList<File>();  
final List<File> list3 = new ArrayList<File>();
```

- **DEUTLICH BESSER**

- Warum nicht einfach das nennen, was gespeichert wird?

```
final List<File> newFiles = new ArrayList<File>();  
final List<File> changedFiles = new ArrayList<File>();  
final List<File> removedFiles = new ArrayList<File>();
```

- **SCHLECHT:**

```
cur.getParent().getChildren().remove(cur);
```

- **SCHON BESSER**

```
selectedItem.getParent().getChildren().remove(selectedItem);
```

- **BESSER:**

```
final TreeItem<String> parentItem = selectedItem.getParent();  
parentItem.getChildren().remove(selectedItem);
```

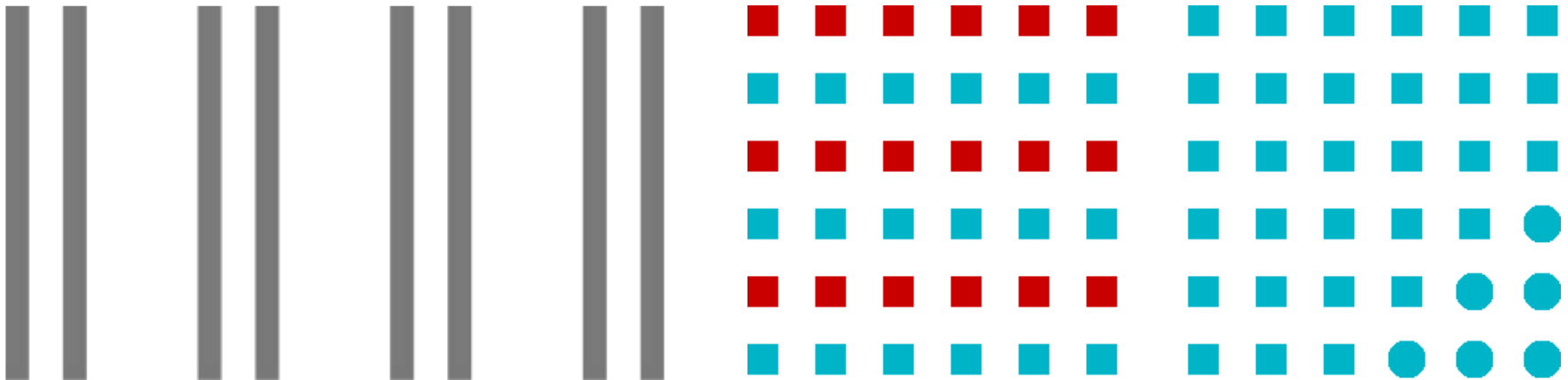
Internal Quality in Depth

3. Structure Of Code



Beim Betrachten von Code spielt die Wahrnehmungspsychologie eine Rolle:

- Ähnliche Dinge werden durch das Gehirn automatisch anhand verschiedener Faktoren gruppiert: “Gesetz der Nähe”, “Gesetz der Ähnlichkeit”, ...



Internal Quality in Depth

3. Structure Of Code – Ugly



```
public final class FormattingExample {
    private static final Logger
                                log =
Logger.getLogger (...);

    public static String asHex( final byte[] tele)
    {
log.info( "asHex("+Arrays.toString(tele)+")");

        final StringBuffer sb = new StringBuffer ( "0x");
for(int i=0;i<tele.length;i++)
    {
        sb.append (Integer.toHexString (tele[i]));
    }
    return sb.toString();
    }
    // ...
}
```

Internal Quality in Depth

3. Structure Of Code – Well Formatted



```
public final class FormattingExample
{
    private static final Logger log = Logger.getLogger(...);

    public static String asHex(final byte[] tele)
    {
        log.info("asHex(" + Arrays.toString(tele) + ")");

        final StringBuffer sb = new StringBuffer("0x");
        for (int i = 0; i < tele.length; i++)
        {
            sb.append(Integer.toHexString(tele[i]));
        }
        return sb.toString();
    }
    // ...
}
```

Internal Quality in Depth

4. Find proper abstractions



- In etwa 3 verschiedene Abstraktionslevel im Code:

- **High** – lesbare Businessmethoden (**public**)

```
isRBLAlive()  
rbl.isAlive()
```

- **Medium** – technische oder Implementierungsdetails (**public – private**)

```
SystemService.isAlive(System.RBL)
```

- **Low** – Ebene von Statements (**private**)

```
((getState() >> 10) & SystemSystemService.STATE_ALIVE) ==  
    SystemSystemService.STATE_ALIVE;
```

Internal Quality in Depth

4. Find proper abstractions



```
public void paint(final Graphics graphics)
{
    if (showGrid)
    {
        graphics.setColor(Color.DARK_GRAY);
        // Raster zeichnen
        for (int x = 0; x < getSize().width; x += GRID_SIZE_X)
        {
            for (int y = 0; y < getSize().height; y += GRID_SIZE_Y)
            {
                graphics.drawLine(x, y, x, y);
            }
        }
    }
    paintFigures(graphics);
}
```



Vermeide den Mix verschiedener Abstraktionsebenen

Internal Quality in Depth

4. Find proper abstractions



```
public void paint(final Graphics graphics)
{
    if (showGrid)
    {
        paintGrid(graphics) ;
    }

    paintFigures (graphics) ;
}
```



- **Bleibe in einer Methode auf einem Abstraktionslevel**
- **Nutze und erzeuge dazu Hilfsmethoden**

- Schauen wir auf folgende wenigen scheinbar harmlosen Zeilen:

```
private boolean deleteTimeSeries(final String key) {  
    if (exists(key)) {  
        assert delete(key);  
        return true;  
    }  
    return false;  
}
```

- Sieht auf den ersten Blick nicht schlecht aus!
- Was kann daran falsch sein?

Internal Quality in Depth

5. Avoid side effects and other surprises



- Applikationscode in einem assert ausgeführt!
- Assertions können an und ausgeschaltet werden und sind standardmäßig deaktiviert => Nutzcode wird NICHT ausgeführt!
- Korrektur:

```
private boolean deleteTimeSeries(final String key) {  
    if (exists(key)) {  
        final boolean deleted = delete(key);  
        assert deleted : "expected TimeSeries to be deleted";  
        return deleted;  
    }  
    return false;  
}
```

- Analoges Problem Applikationscode in Logging-Code

```
if (log.isDebugEnabled())  
{  
    log.debug("some heavy logging");  
    resetLineCounter();  
}
```

- Sehr schwierig zu finden, wenn Log-Ausgaben mal für Debug und mal nicht konfiguriert sind ... und der Kunde an der Log-Konfiguration herumschauen kann

Internal Quality in Depth – Bonus

5. Avoid side effects and other surprises



- Kommen wir nochmal auf unseren ?-Ausdruck zurück:

```
Double value = value1 == null ? value2 : value2 == null ? value1  
: new Double(value1 + value2);
```

- Sollen wir das nicht kürzer schreiben? Was meint ihr? Sollen wir mal `value1 + value2` nutzen?

```
Double value = value1 == null ? value2 : value2 == null ? value1  
: value1 + value2;
```

- Was passiert, wenn `value1` und `value2` null sind?

Internal Quality in Depth – Bonus

5. Avoid side effects and other surprises



- ... B A N G ... NPE

```
Double value = value1 == null ? value2 : value2 == null ? value1  
: value1 + value2;
```

- **?-Operator besitzt Auto-Unboxing-Magic**, der hintere Ausdruck wird versucht auf den Typ Double zu bringen, dazu muss erst Auto-Unboxing und dann Auto-Boxing stattfinden -- aber nicht nur das!

Internal Quality in Depth – Bonus

5. Avoid side effects and other surprises



```
boolean returnInt = true;  
Object val = returnInt ? 5 : 7.0;  
System.out.println("val is " + val + " of " + val.getClass());  
=> val is 5.0 of class java.lang.Double
```

```
if (returnInt) val = 5;  
else val = 7.0;  
System.out.println("val is " + val + " : " + val.getClass());  
=> val is 5 of class java.lang.Integer
```

=> ACHTUNG: ?-Operator und if sind NICHT 100% äquivalent

The End



Vielen Dank für die Aufmerksamkeit!