Bad Smells and Internal Quality

Michael Inden

Table of Contents

- Basics: Why This Talk?
- What are Bad Smells?
- Refactorings as an Answer!?
- What are Characteristics of Good Software?
- Internal Quality In Depth
- Q & A

Basics: Why This Talk?

Basics: Why this talk?

- **In programmers heaven …**
  - Code is wonderful, efficient and simply beautiful
  - Code is readable and understandable
  - You can always implement new features
  - You don't make mistakes, so you don't have to test
  - Your code works right from the start
  - You are productive and have a lot of fun

- **Come back to reality**

Basics: Why this talk?

- **You know it better from your daily business**
  - Coding can be great – mostly if you implement something new
  - Coding can be frustrating when finding ugly pieces of code (**bad smells**)
  - Finding a bug can be difficult
  - And fixing may be even more difficult
  - So … **maintenance** can be a **nightmare**

- **Are you sure that you REALLY fixed the bug? Or introduced a new one?**

Basics: Why this talk?

- There are some answers to these questions
  - **Don’t start to code before you understood the problem or your tasks**
  - Write supplementary unit tests
  - **Know common bad smells, pitfalls and traps**
  - Use small steps to implement functionality (TDD: test, code, refactor)
  - Do some internal QA

- What can you do?

Basics: Why this talk?

- **Learn to follow good habits:**
  - meaningful names
  - proper algorithms and abstractions
  - Documentation of the right things (but don't overdocument)
  - Being careful, open-minded and self-critical
  - Read books

- **Improve your coding skills and do the best you can to write code that is**
  - Clear and Readable
  - **Understandable** and free of redundancy

Basics: Why this talk? Tricky Assignment

- What about this piece of code?

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


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

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

- What does it print out???

0, 50, throws Exception?

Basics: Why this talk? Tricky Assignment

- The result is

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

- What?

```
tricky += ++tricky;       =>     tricky = tricky + ++tricky;
tricky += tricky++;       =>     tricky = tricky + tricky++;
```

- Was genau macht?

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

- Different semantics (who knows it?)

Basics: Why this talk?
Tricky Assignment

- Different semantics (increment then use vs. use then increment)

```
++tricky                        tricky++;

tricky = tricky + 1;            temp = tricky;

                                tricky = tricky + 1;

                                return temp
```

- We learned till now

```
• trickyPre += ++trickyPre;    =>    value is incrementing
• trickyPost += trickyPost++;  =>    value stays 0
```

- How can we get -1 if we start with zero and increment?

Basics: Why this talk? Tricky Assignment – Conclusion

- Let’s do some System.out-Debugging:

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

- **PAY ATTENTION FOR 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**

Basics: Why this talk?

- **Don’t underestimate small code changes and their impact**
  - Switching between pre- and post-increment can be tricky
  - Silent overflows may occur and cause unexpected calculation results
  - A semicolon at the wrong position od missing { } change behavior

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

```
if (condition)
        doSomething();
        doOther();
```

- **Knowledge of programming problems, common pitfalls and language abnormalities/traps is helpful**
- **Because of this we will explore some of this stuff in this talk**

zühlke
empowering ideas

What are Bad Smells?

What are Bad Smells?

My findings: Bad Smells are pieces of code that …

- potentially contain errors
- are misleading or hard to understand
- are suspicious
- give you a bad feeling when looking at them
- make you fear if you have to integrate new features

=> code that's ugly, hard to enhance and tends to be unmaintainable

=> but most of all: code that isn't reliable

Bad Smells – A first example

- Just two lines of code can’t be too bad, or??
- Here they are:

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

- But what’s wrong with these 2 lines of code?
- A lot – right! Okay, let’s focus on the various problems …

Bad Smells – A first example Possible Problems

- **Misleading names** `CmdExe, Printer`
- **Abbreviation with no or little meaning** `ce, reg`
  - nearly no information about semantics
  - in general abbreviations are often confusing (not only for project newbies)
- **Magic Number** `4711`
  - Not transporting semantics
  - Hard to check if they are valid
- **The passed value may be illegal (out of valid range)**
  - What will happen? Exception? Wrong or no execution?

Bad Smells – A first example
Magic Number / Illegal value range

- Illegal value range

```
CommandExecutor(final int registrationStrategy)
{
        if (registrationStrategy == 0){

                switchToAddAsLast()
        }
        if (registrationStrategy == 1){
                switchToAddAsFirst()
        }
        …
}
```

- You never notice that 4711 is invalid!
- Illegal value range – What can we do?

Bad Smells – A first example Define constants

```
public static final int ADD_AS_LAST = 0;
public static final int ADD_AS_FIRST = 1;
public static final int REPLACE_FIRST = 2;

CommandExecutor(final int registrationStrategy)
{
        if (registrationStrategy == ADD_AS_LAST) {
                switchToAddAsLast()
        }
        if (registrationStrategy == ADD_AS_FIRST){
                switchToAddAsFirst()
        }
        …
}
```

+ is more readable, better understandable

- **but still can’t prohibit passing wrong numbers**

Bad Smells – A first example
Add a range check

```
CommandExecutor(final int registrationStrategy)
{
    if (registrationStrategy < ADD_AS_LAST ||
        registrationStrategy > REPLACE_FIRST )
    {
       throw new IllegalArgumentException(“value out of range”);
    }

    if (registrationStrategy == ADD_AS_LAST) {
    …
```

- Seems to be a lot of work to check all input parameters especially when things get more complicate => use frameworks / utility classes
- It’s worth it => rest of the code operates on valid data and don’t have to check again and again

Bad Smells – A first example
Illegal value range – What can we do?

- But think about the erroneous caller?
- Is the warning message `“value out of range”` really helpful?

- Seems to be like these messages:

  Any (bad) program: “Unexpected error 1234 occurred.”

  Deutsche Bahn: “We stopped unexpectedly.”
- Thanks a lot! ;-) Yes, I realized it, too
- But: What is the reason, what can I change, do I get my connection trains?

- => Communicate errors clearly and with helpful information for the caller

Bad Smells – A first example
Illegal value range – What can we do?

zühlke
empowering ideas

- Communicate errors clearly and with helpful information for the caller
  - Error Message should show the valid range

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

  - Looks good. But what was the value that was passed? 4713? 0815?
  - Error Message should show actual value

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

  - Much better. But there is still room for improvement …

Bad Smells – A first example
Provide a list of valid values

```
List<Integer> VALID_VALUES = Arrays.asList(0, 1, 2, 4, 7, 9);

...

throw new IllegalArgumentException("parameter” +
         “‘registrationStrategy’ is invalid: value = “ +
        registrationStrategy + " is not in range “ +
        VALID_VALUES);
```

Console output for input 6:

```
parameter ‘registrationStrategy’ is invalid: value = 6 is not in range [0, 1, 2, 4, 7, 9]
```

Bad Smells – A first example
Final Correction

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

- **What we achieved**
  - Code is more readable and understandable
  - Transports semantics and communicates more clearly what it will do
  - Code is robust and more reliable
  - Predictable behaviour even in error situations: object stays in valid state
  - A lot easier to maintain and to enhance

Bad Smells – A first example
Parameter Checks – Design By Contract

Conclusion

- Use meaningful names
- Avoid magic numbers – use constants instead
- Ensure that all parameters are valid when passed to your public methods
- Handle errors and communicate well

- Design By Contract: Pre-/Post-Conditions and Invariants

Refactorings as an Answer!?

What are Refactorings?

- Martin Fowler says …

  “Refactoring is a change made to the internal structure of a software component to make it easier to understand and cheaper to modify without changing the observable behaviour of that software component.”

- My (less strict) definition:

  “A small change that improves the quality of that piece of code”

When do we need Refactorings?

zühlke
empowering ideas

When do we need Refactorings?

When do we need Refactorings?

- Nearly all the time while coding
- Refactorings are small changes done in little steps
- Should be accompanied by unit tests

Refactorings by Example
A short method to refactor

```
public static boolean isNumber(final String strText)
{
        if (Character.isDigit(strText.charAt(0)))
        {
                for (int i = 1, n = strText.length(); i < n; i++)
                {
                        if (!(Character.isDigit(strText.charAt(i))))
                        {
                                return false;
                        }
                }
        }
        else
        {
                return false;
        }
        return true;
}
```

Refactorings by Example

- **Problems to solve:**
  - **Unexpected exception**
  - **Unclear behaviour**
  - **Error prone**
  - **A lot of returns**
  - **A little complicated**

- **But wait! Before we start changing the method what we should do?**

Refactorings by Example
Step 1: Improve the existing unit tests

- Assume that there are some unit tests for normal inputs and invalid inputs and they are showing green

- We asked the RE team and they say a more general solution is desired
  - we add unit tests for signed numbers and fraction
  - we add unit tests for corner cases like empty inputs or null values

- We ran the tests and some of them fail and that's absolutely correct because we wanted them to fail

Refactorings by Example
Step 2: Combine the two conditions

```
public static boolean isNumber(final String strText)
{
        for (int i = 0, n = strText.length(); i < n; i++)
        {
                if (!(Character.isDigit(strText.charAt(i))))
                {
                        return false;
                }
        }

        return true;
}
```

Now it’s more clear that it tests every char to be a digit

Refactorings by Example
Step 3: Use a method from the jdk

```
public static boolean isNumber(final String input)
{
        try
        {
                Double.parseDouble(input); // just parse
                return true;
        }
        catch (final NumberFormatException ex)
        {
                return false;
        }
}
```

Now it's clear what we want to do

But: We changed the observable behaviour! In this case it's okay

Refactorings by Example
Step 4: Check input parameter

```
public static boolean isNumber(final String input)
{
        if (input== null)
                throw new IllegalArgumentException("parameter " +
                                             "'input' must not be null");
        try
        {
                Double.parseDouble(strText);
                return true;
        }
        catch (final NumberFormatException ex)
        {
                return false;
        }
}
```

Now it's understandable, readable and communicates errors clearly

Characteristics Of Good Software

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.”

Good Software – Internal Quality

Internal Quality is about CODE:

- readable
- Easy to understand
- Without traps, no Easter eggs
- Extensible, Maintainable
- Well documented
- Good Test/Code coverage
- …

Good Software – External Quality

External Quality is a user centred view (FEATURES and BEHAVIOUR):

- Works as expected
- Provides all desired functionality
- Correctness
  - Nearly no (observable) bugs
  - Well tested
- Usability
- Reliability
- …

Good Software – Common design principles

Common design principles

1. KISS – Keep It Simple and Short
2. DRY – Don't Repeat Yourself
3. YAGNI – You Ain't Gonna Need It
4. SOLID – 5 Principles

SOLID – At a glimpse

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

Internal Quality In Depth

Michael Inden

Internal Quality – What really matters

1. Shortness – KISS
2. Meaningful names
3. Structure Of Code
4. Find proper abstractions
5. Avoid side effects and other surprises
6. Separation Of Concerns
7. Control of State / Immutability

Internal Quality in Depth
1. KISS– Keep it simple and short

- **KISS –Keep your code base as small as possible**
- **Use existing the JDK and 3rd party libraries wheneve**

- **SRP – Single Responsibility Principle states that**
  - methods and classes should have one responsibility
  - methods and classes should be as short or simple as possible

- **My advice**
  - methods => max. 50 – 100 lines, preferable 10 – 20 lines
  - classes => max. 1000 – 2000 lines, preferable up to 500 lines

Internal Quality in Depth
1. Shortness – KISS

- Keep your code base as small as possible, crispy and precise
- ?-Operator ??? With this, the following is true, right?

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

=>

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

Internal Quality in Depth
1. Shortness – KISS

- Keep your code base as small as possible
- Example: We can get rid of `if`'s when we use conditional operators

```
(x % 2 == 0) ? "even" : "odd"
```

- But short is not always preferable ... take a look at this

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

- Can you figure out, what the code does? Immediately? For sure?
- Bad Smell: Complex logic in conditional operator

Internal Quality in Depth
1. Shortness – KISS

- Keep your code readable and understandable at first
- What about the following?

```
  Double value = nullsafeAdd(value1, value2);



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

  return 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;
}
```

- Handles all the special cases upfront
- Communicate clearly, directly understandable
  (blue line not necessary, but easier to understand)
- logic can be kept as simple as possible => that's KISS

Internal Quality in Depth
1. Shortness – KISS

Solution / Refactoring(s)

- Use the proper abstractions
- Avoid duplication with helper methods (DRY)
- Use existing the JDK and 3rd party libraries whenever possible
- Introduce Convenience Methods

Internal Quality in Depth
! KISS – Builder but irritating Code

// We want the following output: 13.03.2014 17:41:22

- BAD

```
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();
```

- A LOT BETTER

```
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
2. Meaningful names

- Hard to find
- But it's worth the effort
- Names shouldn't be to long but expressive enough
- Avoid Abbreviations like AAA/AAI and other nice things ;-)
- Abbreviations like Db, Html, Xml are common and should be used to keep names shorter

Internal Quality in Depth
2. Meaningful names

zühlke
empowering ideas

- What about these few lines of code?

```
// 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());
}
```

- What can be improved??

Internal Quality in Depth
2. Meaningful names

- These lines don’t communicate what they do
- The comments aren’t useful too (!DRY – Don‘t Repeat Yourself)

```
// 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

- Use meaningful names and see what happens:

```
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());
}
```

- Okay: We are calculating the age of oldest person listed in the table.
- Pretty clear now!

Internal Quality in Depth
2. Meaningful names for Collections

- **BAD**
  - name consisting just of data type: 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>();
```

- **IMPROVED**
  - Use additional "s" for collections of elements: person**s**, figure**s**

```
final Vector<File> files = new Vector<File>();
```

  - Repetition of type may be helpful: listOfImages, idToPersonMap

```
final List<File> listOfFiles = new ArrayList<File>();
```

Internal Quality in Depth
2. Meaningful names

- **EVEN BETTER**
  - What about to communicate what is stored?

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

Solution / Refactoring(s)

- Think twice or more about a name
- Try to communicate the purpose …
  What It stores (for attributes) or it does (for methods)
- Discuss names of business methods with Requirements Engineers

Internal Quality in Depth
2. Meaningful names: Communicate well

- BAD:

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

- A BIT BETTER

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

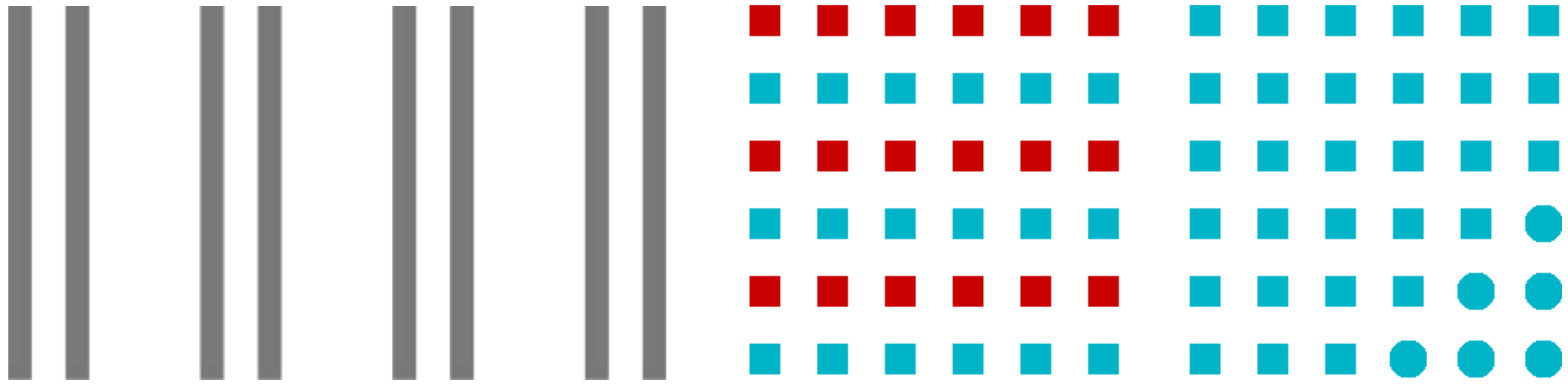
- BETTER

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

Internal Quality in Depth
3. Structure Of Code

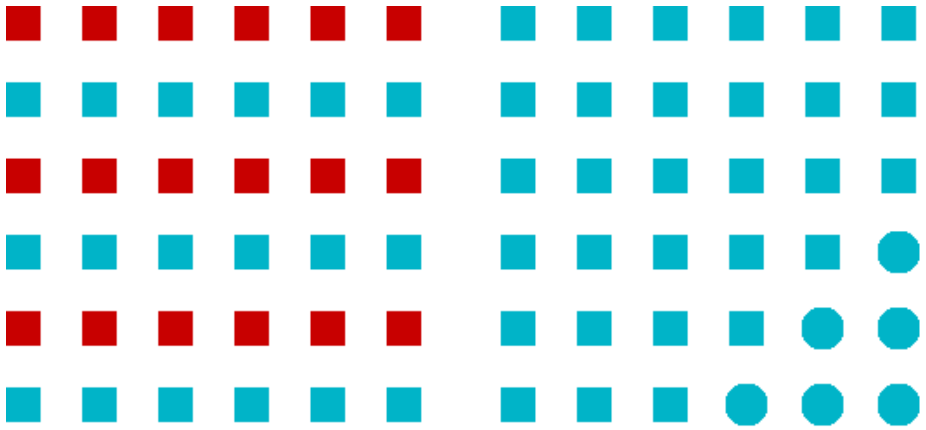
If you look at code there is a lot about psychology:

- when looking at something you notice combinations by grouping related things, known as “Gestalt der Nähe”, “Gestalt der Ähnlichkeit”, …



Internal Quality in Depth
3. Structure Of Code

If you look at code there is a lot about psychology:



- **recognising structures is easier when**
  - The layout is clear
  - Curly braces on a separate line (controversy)
  - Spaces and blank lines separate blocks of functionality

Internal Quality in Depth
3. Structure Of Code – Ugly

```
public final class FormatingExample {
   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 FormatingExample
{
     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

Solution / Refactoring(s)

- Use a Code Checker
- Use a Layout Formatter

Internal Quality in Depth
4. Find proper abstractions

- Almost 3 different abstraction layers in code
  - high– readable business methods (**public**)

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

  - medium– a few technical or implementation details (**public – private**)

```
SystemStateService.isAlive(System.RBL)
```

  - low– the level of statements (**private**)

```
((getState() >> 10) & SystemStateService.STATE_ALIVE) ==
                       SystemStateService.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);
}
```

Avoid to mix different levels of abstraction (harder to read)

Internal Quality in Depth
4. Find proper abstractions

Solution / Refactoring(s)

- Stick on the same abstraction level in one method
- Use (and create) helper methods

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

        paintFigures(graphics);
}
```

Internal Quality in Depth
5. Avoid side effects and other surprises

- Let's look at the following few apparently innocent lines:

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

- Doesn't look bad at first glance!
- What can be wrong with it?

Internal Quality in Depth
5. Avoid side effects and other surprises

- **Application code executed in an assert!**
- Assertions can be turned on and off and are disabled by default => payload code is NOT executed!

- Correction:

```
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;
}
```

Internal Quality in Depth
5. Avoid side effects and other surprises

- 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
5. Avoid side effects and other surprises

```
List<Person> getPersons(String containing)
{
        setAttributeXyz();

        modifiyAddresses();


        List<Person> filtered = new ArrayList<>();
        for (Person person : this.persons)
        {
                if (person.getName().contains(containing))
                {
                        filtered.add(person);
                }
        }
        return filtered;
}
```

A call to this get method changes state! But GETTERs should be READ ONLY

Internal Quality in Depth
5. Avoid side effects and other surprises

- Let’s come back to our overly complex conditional statement:

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

- Okay, it’s ugly, of course
- The result is hard to figure out, it is everything but obvious

- Did you noticed the `new Double(value1 + value2)`?

- Should we remove it and replace it with `value1 + value2`?

Internal Quality in Depth
5. Avoid side effects and other surprises

- Okay, let’s remove the `new Double()`

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

+ a little shorter

+ a little more obvious

+ seems to work well

+ so it get’s deployed …

Internal Quality in Depth
5. Avoid side effects and other surprises

- … B A N G . . . We get a NPE

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

- What? A NPE? Ridiculous?
- What can cause a NPE? => value1 and value2 == null
- But why? The problem is, that there is some Auto-Unboxing-Magic
- Remember the original line won't cause a NPE:

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

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**

Internal Quality in Depth
6. Separation Of Concerns

Hints for this kind of problem

- Long methods
- A lot of parameters
- Strange or long method names like
  …And… => retrieveAndFilter
  …After… => searchAfterDbConnect

Solution / Refactoring(s)

Split information retrieval and processing

Internal Quality in Depth
6. Separation Of Concerns

```
String getResultsAndFormat(final OutputFormat format)
{
        String representation = "";
        if (format == HTML)
        {
                List results = retrievePersonsFromDb();
                representation = convertToHtml(results)
        }
        if (format == XML)
        {
                List results = retrievePersonsFromDb();
                representation = convertToXml(results)
        }

        return representation;
}
```

Internal Quality in Depth
6. Separation Of Concerns

Analysis

+ Names are okay, but can be improved

+ code structure is okay too

+ some functionality is already extracted into separate methods

- Method does a lot / has too many responsibilities

- Not ideal Separation Of Concerns: Retrieval and Conversion to output format bundled and combined together

- Hard to enhance … new types of representation, what about json

(Compile type dependency and using strings / enums)

Internal Quality in Depth
6. Separation Of Concerns

```
List getPersonsAndCreateOutput(final OutputFormat format)
{
        // RETRIEVAL
        List persons = retrievePersonsFromDb();


        // OUTPUT FORMATTING
        String representation = “”;
        if (format == HTML)
        {
                representation = convertToHtml(results)
        }
        if (format == XML)
        {
                representation = convertToXml(results)
        }
        return representation;
}
```

Internal Quality in Depth
6. Separation Of Concerns

```
List getPersonsAndCreateOutput(final OutputFormat format)
{
        List persons = retrievePersonsFromDb();
        return createRepresenation(persons, format);
}

String createRepresenation(List persons, OutputFormat format) {
        String representation = “”;
        if (format == HTML)
        {
                representation = convertToHtml(results)
        }
        if (format == XML)
        {
                representation = convertToXml(results)
        }
        return representation;
}
```

Internal Quality in Depth
6. Separation Of Concerns

```
String createRepresenation(List persons, OutputFormat format)
{
        IOutputFormatter formatter = getByFormat(format);
        if (formatter == null) {
                // warning or exception
                return "";
        }

        return formatter.format(persons);
}
```

Use OO-Design Principles:

- **Use Abstractions** `(IOutputFormatter)`
- **Use Polymorphism** `(HtmlFormatter, XmlFormatter)`

Can be easily enhanced `(e.g. new JsonFormatter + getByFormat())`

Internal Quality in Depth
7. Control Of State / Immutabilty

- Always ensure that your objects are in a valid state
- Goal: Reduce the possibility to change the state (directly) from outside by calling setters

```
Person readPersonFromDb(ResultSet rs)
{
      Person newPerson = new Person();
      try
      {
            newPerson.setName(rs.getString("NAME"));
            …
            newPerson.setHeight(rs.getInt("HEIGHT"));
      }
      catch (SQLException ex)
      { // not correctly handled here (wait a minute) }
      return newPerson;
}
```

Internal Quality in Depth
7. Control Of State / Immutabilty

Problem with the code:

- Every single db access may cause a sql exception and the object is in an undefined state

What can we do?

- Provide a business behaviour driven interface
- Avoid mutable attribute, prefer immutability
- Have well defined state transitions => Multithreading is getting easier

Internal Quality in Depth
7. Control Of State / Immutabilty

- Read data into temporary variables
- Create Person instance only when no error has occured

```
Person readPersonFromDb(ResultSet rs)
{
      try
      {
            String name = rs.getString("NAME");
            …
            int height = rs.getInt("HEIGHT");
            return new Person(name, …, height);

      }
      catch (SQLException ex)
      { // not correctly handled here (wait a minute) }
      return null;
}
```

The End

Thank you for your attention