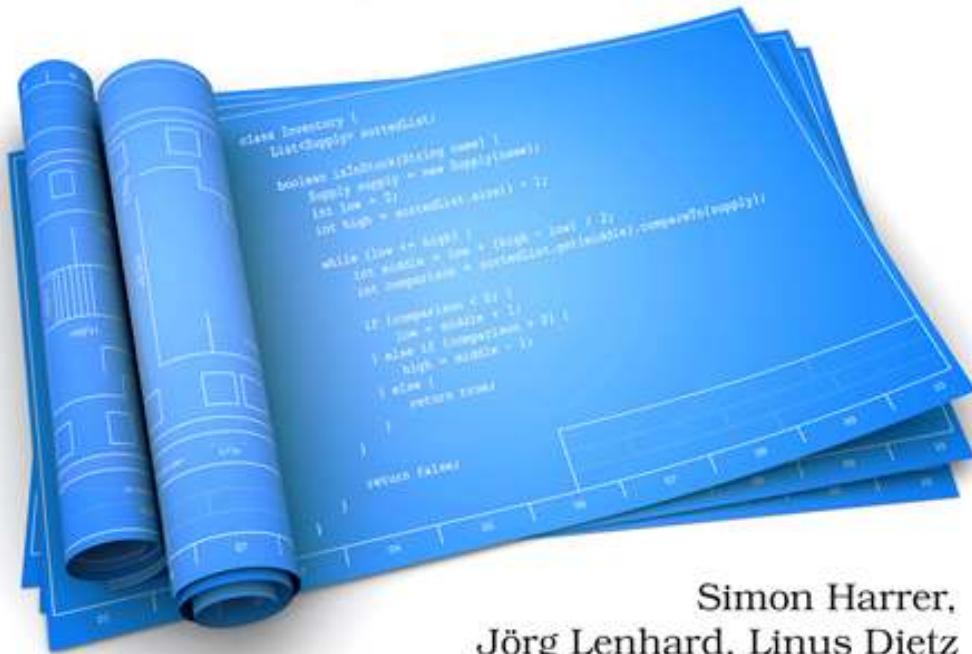


The
Pragmatic
Programmers

Java by Comparison

Become a Java Craftsman
in 70 Examples



Simon Harrer,
Jörg Lenhard, Linus Dietz
Foreword by Venkat Subramaniam
Edited by Andrea Stewart

25.03.2019

München / Coding Dojo @ BMW

Die Geschichte(n) von Java by Comparison

Dr. **Simon** Harrer
Linus Dietz

INNOQ

Technische
Universität
München **TUM**



Dr. Simon Harrer

Senior Consultant
innoQ Deutschland GmbH

Simon wirbt für
passende Architekturen



Linus Dietz

Wissenschaftlicher Mitarbeiter & Doktorand
Technische Universität München

Linus forscht über
Empfehlungsdienste für Reisende

Agenda

- 1. Geschichte(n) des Buches (30 min)**
- 2. Comparison Jeopardy (15 min)**
- 3. Einführung Kata (15 min)**
- 4. Pause (30 min)**
- 5. Java by Comparison Kata (120 min)**
- 6. Retro (15 min)**

2011



Typische Lehrveranstaltungen

Theoretische Vorlesung

Plenarübung

Hausaufgaben

Schriftliche Klausur

Aufgabenstellung



Lösung



Note



**Wie bringt man anderen bei
sauberen Code
zu schreiben?**

1. **Code Reviews** interaktiv miterleben
2. **Code Reviews** schriftlich bekommen
3. **Code Reviews** selbst durchführen

16 – Wurde der Code ausreichend dokumentiert?

17 Ja

18
19 – Code Conventions eingehalten?

20 Nein. Ihr solltet Klassen- und VariablenNamen in CamelCase schreiben. Im Fall von Findspots ist das nicht der Fall. Klassennamen sollten auch generell Nomen sein. Also in diesem Fall `SpotFinder`.

21
22
23 **##Code**

24
25 – Ressourcen korrekt geschlossen?

26 Ja

27
28 – Exceptions korrekt behandelt (inkl. weiterwerfen mit Cause)?

29 Nein. Wenn der Validator fehlschlaegt darf das Programm nicht mit `System.exit()` beendet werden! Stattdessen soll eine `TokenfinderException` mit einer Nachricht geworfen werden. Fehler sollten frueh geworfen werden. Das bedeutet, dass sobald im `Validator` ein Check fehlschlaegt, sofort eine passende Exception geworfen werden sollte und der Rest nicht mehr ueberprueft werden muss.

30 Ihr verwendet die `javax.xml.bind.ValidationException`. Das ist schlechte Praxix, da sich diese auf Fehler im Kontext von XMLvalidierung bezieht. Hier haettet ihr einfach die `TokenFinderException` verwenden koennen. Weiterhin habt ihr im `Validator` die Methoden von `java.nio` noch einmal in eigene gleichnamigen Methoden gesteckt nur um jeweils eine `javax.xml.ValidationException` werfen zu koennen. Das ist sehr viel unnoetiger und falscher Code, weil diese Methoden ja ihre eigenen Exceptiontypen (meist `IOException`) werfen, die ihr dann jeweils fangen sollt.

31 Was auch unschoen ist, dass ihr in `SimpleTokenFinder.search()` #55–74 einen `try-catch-block` habt, der eine `TokenFinderException` faengt. Diese sollte einfach durch throws nach oben weitergegeben werden anstatt die Nachricht auszugeben.

32
33
34 – Wird die Java NIO Bibliothek verwendet?

35 Ja

36
37 – Wurde die Aufgabe sinnvoll (bspw. in mehrere Klassen) strukturiert?

38 Ja, alledings haben eure Methoden teilweise noch viele Einrueckungen. Da solltet ihr noch etwas mehr

2016

Common Stuff

- Warnings still in the Code
- FIXME and TODO in the Code
- Commented out code
- Do not use `System.exit()`

Issues

- Always use braces `if(CONDITON) { STATEMENT; }`, never `if(condition) STATEMENT;` (see <http://embeddedgurus.com/barr-code/2014/03/apples-gotofail-ssl-security-bug-was-easily-preventable/>)
- Do never catch `Exception` and `NullPointerException`, except you have an extremely good reason for it
- Instead of `if (CONDITION) { return true; } else { return false; }` simply write `return CONDITION;`
- Avoid unnecessary documentation such as `// ... return true ...` to document `return true;`. Focus on the important things instead.
- Do not document your name in the classes to avoid marking bias
- Use `final` as much as possible for your fields - this simplifies your code tremendously

Siegerehrung

Zum ersten Mal seit 6 Jahren AJP hat ein Team
die volle Punktzahl erreicht. Es ist möglich.

Herzlichen Glückwunsch
Gruppe ?



Siegerehrung

Zum ersten Mal seit 6 Jahren AJP hat ein Team
die volle Punktzahl erreicht. Es ist möglich.

Herzlichen Glückwunsch
Gruppe ?



In 6 Jahren ...

... **500+ Assignments** korrigiert

... **1.000.000+ Java Codezeilen** gelesen

... **1000de der gleichen Fehlern** immer wieder angemerkt

... gelernt andere zu überzeugen “Clean Code” zu schreiben

... neue Herausforderungen gesucht.



War's das jetzt?

Common Stuff

- Warnings still in the Code
- FIXME and TODO in the Code
- Commented out code
- Do not use `System.exit()`

Issues

- Always use braces `if(CONDITION) { STATEMENT; }`, never `if(condition) STATEMENT;` (see <http://embeddedgurus.com/barr-code/2014/03/apples-gotofail-ssl-security-bug-was-easily-preventable/>)
- Do never catch `Exception` and `NullPointerException`, except you have an extremely good reason for it
- Instead of `if (CONDITION) { return true; } else { return false; }` simply write `return CONDITION;`
- Avoid unnecessary documentation such as `// ... return true ...` to document `return true;`. Focus on the important things instead.
- Do not document your name in the classes to avoid marking bias
- Use `final` as much as possible for your fields - this simplifies your code tremendously

Na klar!

Actionable Name

Favor Format Over Concatenation

Bad Code

```
1 class SearchResult {
2     Path file;
3     int lineNumber, startIndex;
4     String token;
5
6     void print() {
7         System.out.println(file + ":[\" + lineNumber + \";\"
8             + startIndex + \"-\" + (startIndex +
9             token.length() - 1) + \"]> \"
10            + token + System.lineSeparator());
11     };
12 }
```

The problem with this code is that it is *very* hard to see how the output will look like.

Because there are so many plusses, quotes and spaces that obfuscate the code it takes longer to read and the chance of making mistakes is higher. Additionally, performing inline computations like summing ints is not really acceptable any more as the plus operator has different semantics in the very same line what makes it ever harder to read for us.

Problem Description

Better Code

```
1 class SearchResult {
2     Path file;
3     int lineNumber, startIndex;
4     String token;
5
6     void print() {
7         System.out.printf(\"%s:[%d;%d-%d]> %s\\n\",
8             file,
9             lineNumber,
10            startIndex,
11            startIndex + token.length() - 1,
12            token);
13     };
14 }
```

By using `System.out.printf()` (or `String.format()` for working with Strings without printing them) we achieve enhanced separation of syntax (how the String is printed) and semantics (which data is used for printing). Furthermore, it is more acceptable to make inline computations, as we can format the code such that each parameter is in its own line.

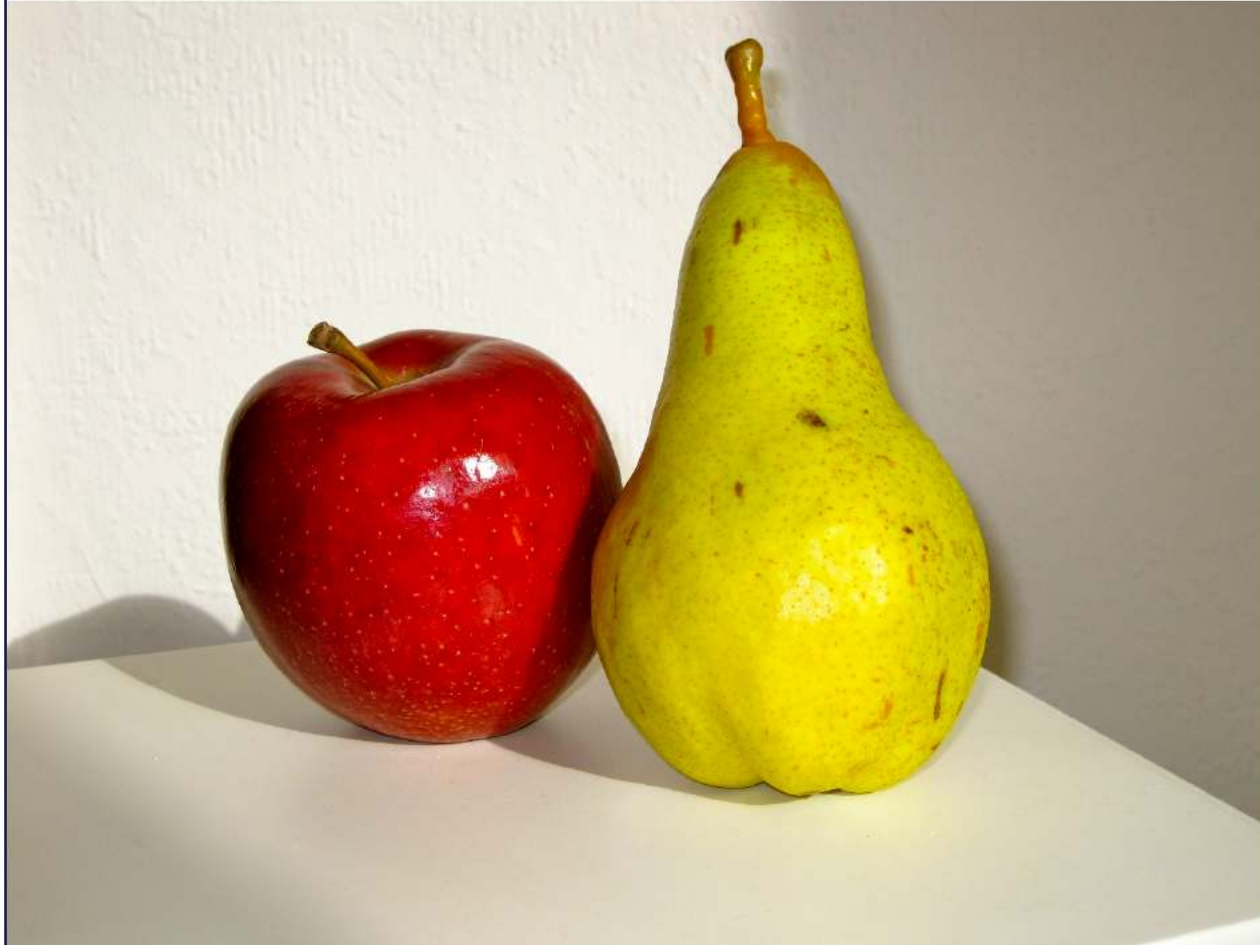
On a side note it should be mentioned that this way of formatting Strings is very common in most imperative programming languages and works the very same way in C/C++ and other programming languages.

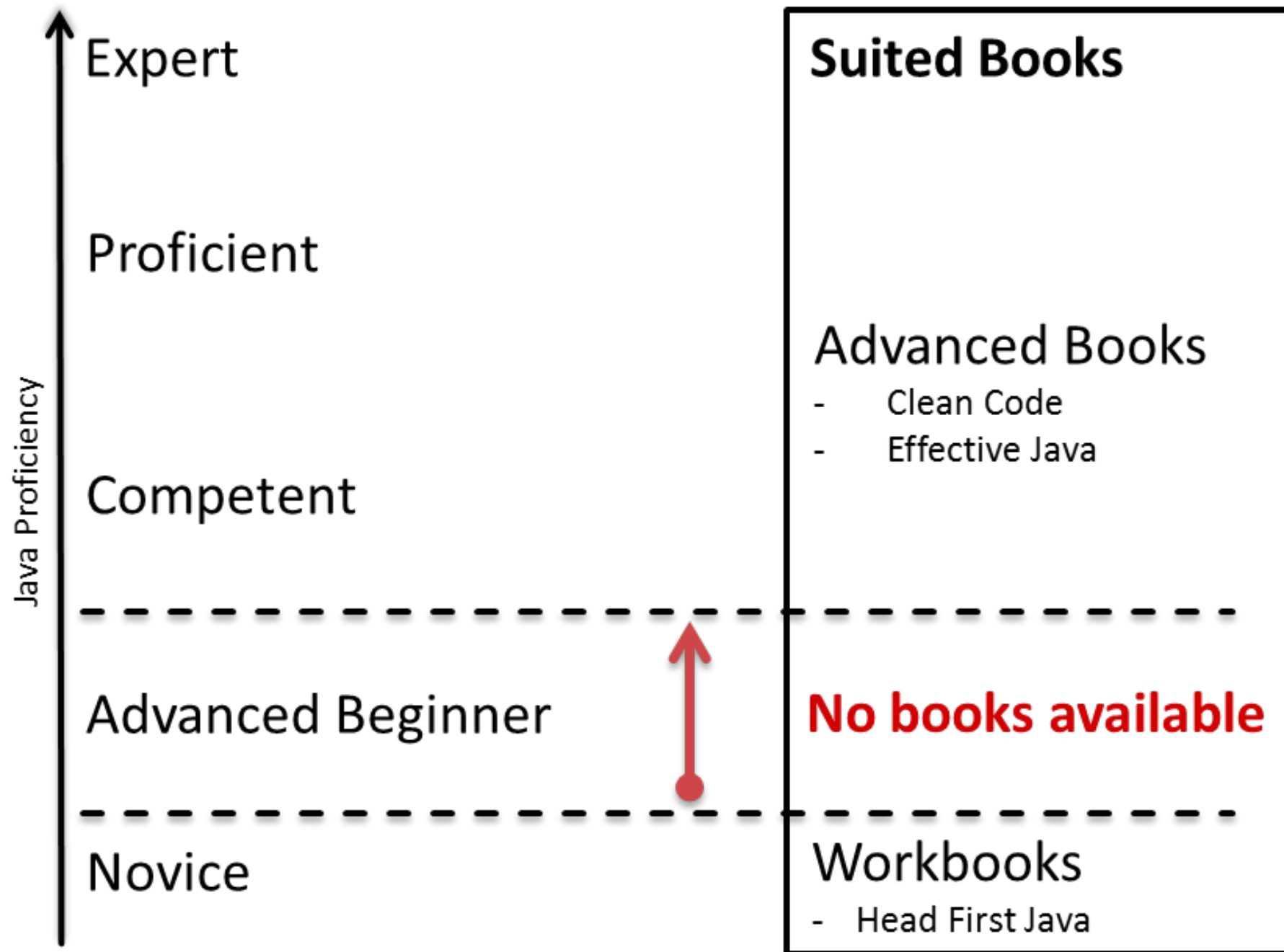
Solution Description

Java by Comparison

A Software Craftsman's First Book

Simon Harrer and Linus Dietz



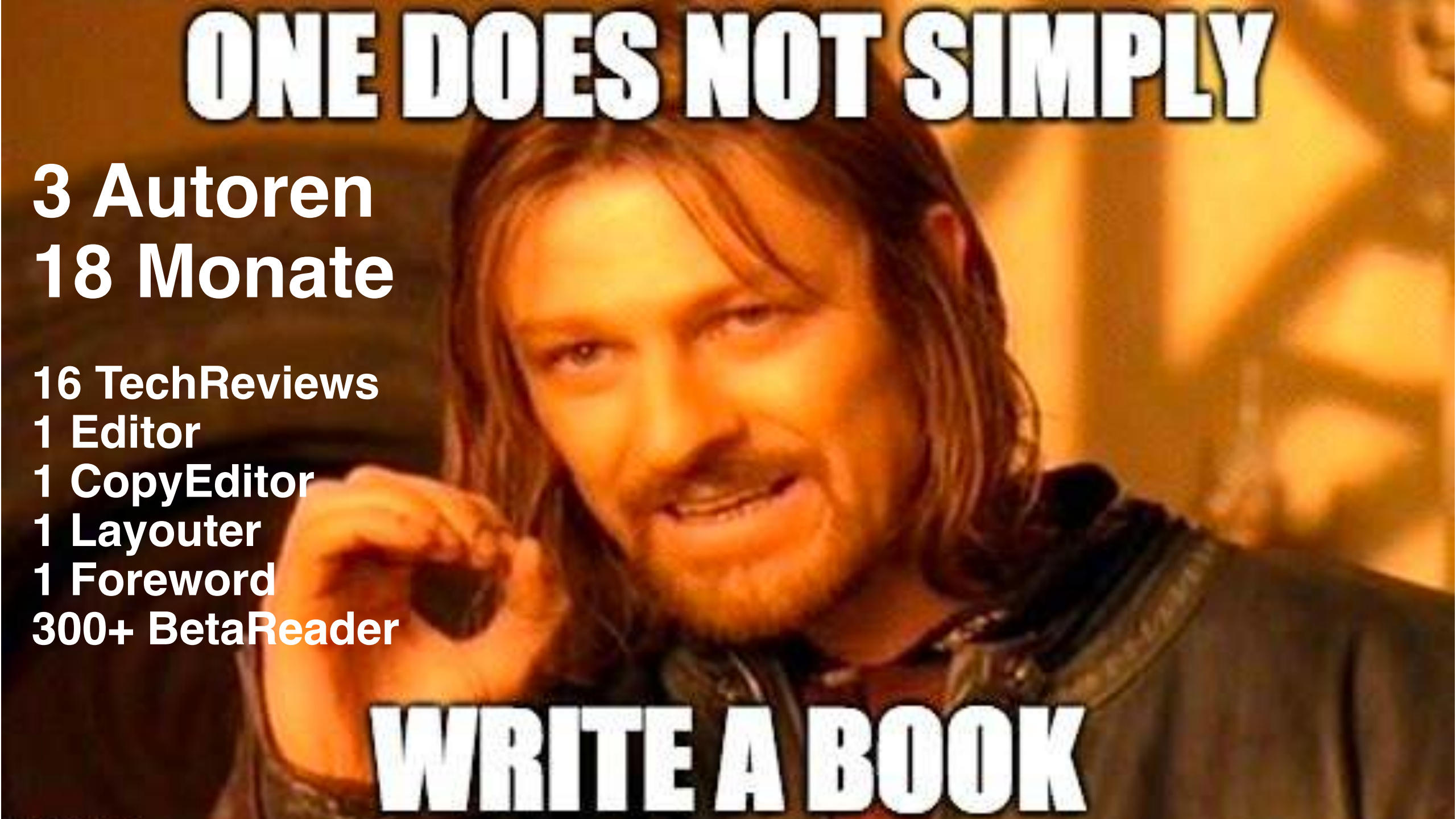




...a ut eu. Ad ...
...e, voluptas ...
...ariatur. Magna, in ...
...molit adipiscit ...
Duis excepteur sint ut, conetur, ...
incididunt et ad ea occaecat ...
ipsum non. Adipiscing ...
velit ut in, velit quis id. Tunc ...
Nulla deserunt amet quis est conetur, ...
enim sed ea eu conetur ut, in ...
pariatur in ea dolor velit, excepteur ...
tempor velit, adipiscing, excepteur ...
ipsum conetur dolor non occaecat ...
Anim laboris ullamco velit tempor velit tempor
culpa deserunt sint elit. Ullamco minim conetur
commodo elit fugiat magna occaecat euismod
Esse ea esse.

[Handwritten signature]

Signature



ONE DOES NOT SIMPLY

3 Autoren
18 Monate

16 TechReviews
1 Editor
1 CopyEditor
1 Layouter
1 Foreword
300+ BetaReader

WRITE A BOOK

Favor Format Over Concatenation

```
class Mission {
    Logbook logbook;
    LocalDate start;

    void update(String author, String message) {
        LocalDate today = LocalDate.now();
        String month = String.valueOf(today.getMonthValue());
        String formattedMonth = month.length() < 2 ? "0" + month : month;
        String entry = author.toUpperCase() + ": [" + formattedMonth + "-" +
            today.getDayOfMonth() + "-" + today.getYear() + "](Day " +
            (ChronoUnit.DAYS.between(start, today) + 1) + ")> " +
            message + System.lineSeparator();
        logbook.write(entry);
    }
}
```

Understandability and readability don't just matter in your code—they're also important for the output your code produces.

If you have to build large strings, you can use format strings to make them more readable.

Here, we're revisiting the log book we already mentioned in [Avoid NullPointerException in Conditionals, on page 10](#). The problem is that it's quite hard to see what the output will actually look like. There are multiple String variables, some of them using Java's in-line notation.

The code isn't really complex. There's no deep nesting, no conditional branching, and even the names of variables and methods are meaningful. Despite all this, it's comparably hard to read. The number of plusses, quotes, and spaces obfuscates the actual processing going on (humans are generally better at understanding words in a natural language than combinations of symbols). This makes the code harder to read and increases the chance of making mistakes when writing it.

Additionally, performing in-line computations like summing int values adds to the confusion, as the + operator has different semantics when it comes to Strings and ints. Using it with different semantics in the very same line makes it even harder to maintain an overview of what's happening. Don't do this!

We can trim this down a lot. Take a look at the following code:

```
class Mission {
    Logbook logbook;
    LocalDate start;

    void update(String author, String message) {
        final LocalDate today = LocalDate.now();
        String entry = String.format("%S: [%tm-%<te-%<tY](Day %d)> %s%n",
            author, today,
            ChronoUnit.DAYS.between(start, today) + 1, message);
        logbook.write(entry);
    }
}
```

Format strings help to overcome this issue, and they're available in practically any contemporary programming language—not just in Java. The key lies in separating the layout of a String (*how* it is printed) from the data (*what* is being printed). Format strings define a coherent String in a single block using special placeholder characters, marked by %.

Format methods, such as String.format(), or System.out.printf(), accept data using placeholder characters in the order in which they're listed after the String.

In our example here, %S converts an object into an upper case String using its toString() method. The parameter author will be processed this way. The date variable today serves as data for %tm, the month, %te the day of the month, and %tY, the year. The additional character < makes sure that all three placeholders read the same input data. %d processes a decimal value and %s accepts a String. Finally, %n is the symbol for a line break.

The actual data is listed in a nicely formatted fashion after the String layout. This makes it a little more acceptable to perform in-line computations, as we can use one line for each parameter. Point taken that it's not so easy to see what "%S: [%tm-%<te-%<tY](Day %d)> %s%n" will print in the end. But it's a well-documented standard³ and actually a good alternative to the cluttered code from the problem—for large strings we recommend StringTemplate,⁴ a powerful template engine.

Consider documenting the formatted string with a few examples similar to [Document Using Examples, on page 48](#) so that the next developer reading your code doesn't have to look up how %S or any other special formatting syntax behaves to know what the resulting string will be.

1. Start Cleaning Up

2. Level Up Your Code Style

3. Use Comments Wisely

4. Name Things Right

5. Prepare for Things Going Wrong

6. Assert Things Going Right

7. Design Your Objects

8. Let Your Data Flow

9. Prepare for the Real World

JAVAPRO

 heise **Developer**



InfoQ

DAWSON
C O L L E G E

26 Kundenrezensionen

★★★★★ 4,8 von 5 Sternen ▾



10 customer reviews

★★★★★ 5.0 out of 5 stars ▾



Let's Play!

Comparison Jeopardy

Actionable Name

Bad Code

Problem Description

Better Code

Solution Description

Spielregeln

- Wir zeigen euch einen **Codevergleich**
- **Findet einen Namen** der das “Refactoring” beschreibt
 - Name muss mit einem **Verb** anfangen
- **Beispiele**
 - **Simplify Boolean Expression**
 - **Favor Format Over Concatenation**

Kategorie: Code Style

Avoid Negations

```
class Laboratory {  
    Microscope microscope;  
  
    Result analyze(Sample sample) {  
        if (microscope.isInorganic(sample)) {  
            return Result.INORGANIC;  
        } else {  
            return analyzeOrganic(sample);  
        }  
    }  
  
    private Result analyzeOrganic(Sample sample) {  
        if (!microscope.isHumanoid(sample)) {  
            return Result.ALIEN;  
        } else {  
            return Result.HUMANOID;  
        }  
    }  
}
```

```
class Laboratory {  
    Microscope microscope;  
  
    Result analyze(Sample sample) {  
        if (microscope.isOrganic(sample)) {  
            return analyzeOrganic(sample);  
        } else {  
            return Result.INORGANIC;  
        }  
    }  
  
    private Result analyzeOrganic(Sample sample) {  
        if (microscope.isHumanoid(sample)) {  
            return Result.HUMANOID;  
        } else {  
            return Result.ALIEN;  
        }  
    }  
}
```

Kategorie: Kommentare

Document Using Examples

```
class Supply {  
    /**  
     * The code universally identifies a supply.  
     *  
     * It follows a strict format, beginning with an S (for supply), followed  
     * by a five digit inventory number. Next comes a backslash that  
     * separates the country code from the preceding inventory number. This  
     * country code must be exactly two capital letters standing for one of  
     * the participating nations (US, EU, RU, CN). After that follows a dot  
     * and the actual name of the supply in lowercase letters.  
     */  
    static final Pattern CODE =  
        Pattern.compile("^S\\d{5}\\\\(US|EU|RU|CN)\\. [a-z]+$");  
}
```

```
class Supply {  
    /**  
     * The expression universally identifies a supply code.  
     *  
     * Format: "S<inventory-number>\\<COUNTRY-CODE>.<name>"  
     *  
     * Valid examples: "S12345\\US.pasta", "S08342\\CN.wrench",  
     * "S88888\\EU.laptop", "S12233\\RU.brush"  
     *  
     * Invalid examples:  
     * "R12345\\RU.fuel"      (Resource, not supply)  
     * "S1234\\US.light"     (Need five digits)  
     * "S01234\\AI.coconut"  (Wrong country code. Use US, EU, RU, or CN)  
     * " S88888\\EU.laptop " (Trailing whitespaces)  
     */  
    static final Pattern SUPPLY_CODE =  
        Pattern.compile("^S\\d{5}\\\\(US|EU|RU|CN)\\. [a-z]+$");  
}
```

Kategorie: OO-Design

Combine State and Behavior

```
class Hull {
    int holes;
}

class HullRepairUnit {
    void repairHole(Hull hull) {
        if (isIntact(hull)) {
            return;
        }
        hull.holes--;
    }

    boolean isIntact(Hull hull) {
        return hull.holes == 0;
    }
}
```

```
class Hull {
    int holes;

    void repairHole() {
        if (isIntact()) {
            return;
        }
        holes--;
    }

    boolean isIntact() {
        return holes == 0;
    }
}
```

Kategorie: Namen

Avoid Meaningless Terms

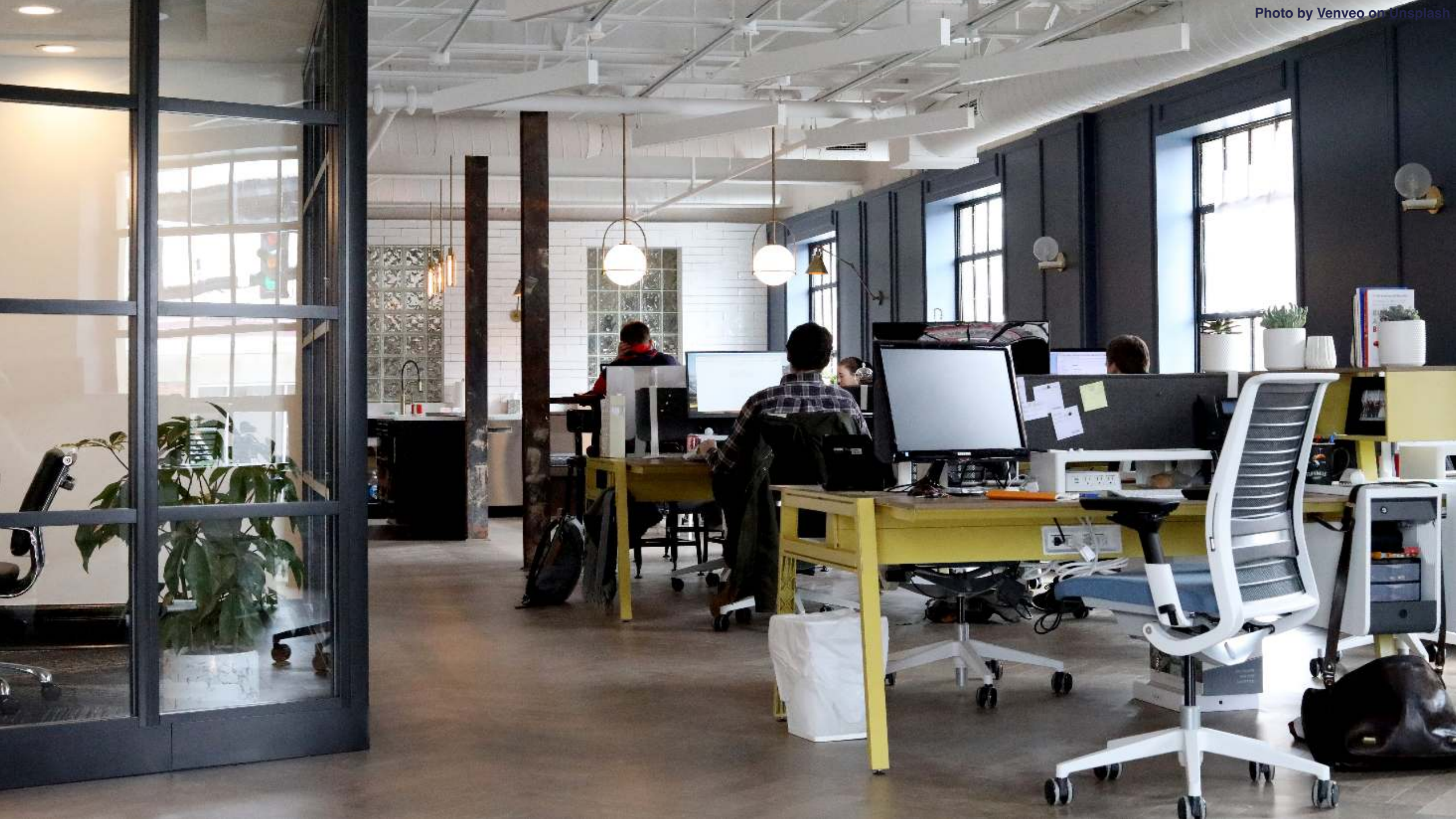
```
class MainSpaceShipManager {  
    AbstractRocketPropulsionEngine abstractRocketPropulsionEngine;  
    INavigationController navigationController;  
    boolean turboEnabledFlag;  
  
    void navigateSpaceShipTo(PlanetInfo planetInfo) {  
        RouteData data = navigationController.calculateRouteData(planetInfo);  
        LogHelper.logRouteData(data);  
        abstractRocketPropulsionEngine.invokeTask(data, turboEnabledFlag);  
    }  
}
```

```
class SpaceShip {  
    Engine engine;  
    Navigator navigator;  
    boolean turboEnabled;  
  
    void navigateTo(Planet destination) {  
        Route route = navigator.calculateRouteTo(destination);  
        Logger.log(route);  
        engine.follow(route, turboEnabled);  
    }  
}
```

BRANDNEU

Java by Comparison Kata





README.md

```
# download kata
```

```
$ git clone https://github.com/javabycomparison/kata
```

```
$ cd kata
```

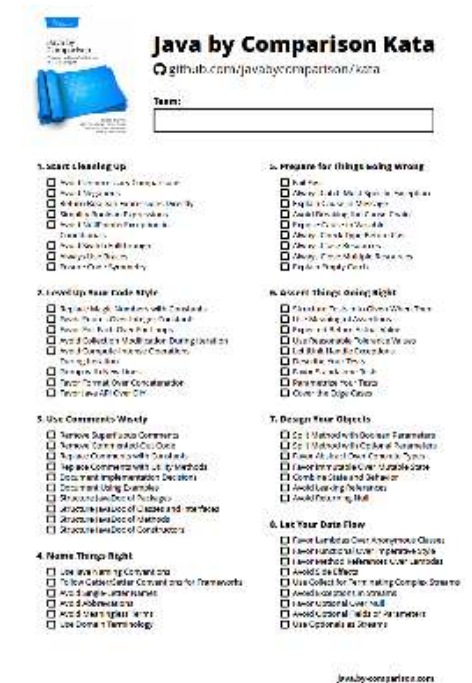
```
# execute tests
```

```
$ ./mvnw test
```

```
# run
```

```
$ ./mvnw exec:java -Dexec.args="src smry"
```

1. **Sucht** den für euch schlimmsten Code Smell für das es eine Comparison gibt
2. **Wendet** Comparison an
3. Tests **Grün?** **Hakt** Comparison ab
4. **REPEAT**

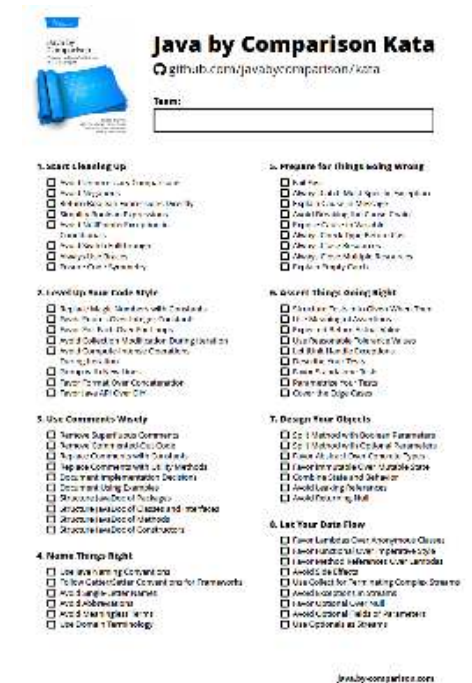


Ihr habt 2 Stunden Zeit. Viel Spaß!

PAUSE bis 14:45

1. **Bildet** Pairs: 2 Personen, 1 Buch, 1 Laptop
2. **Importiert** das Kata in eure IDE
3. **Testet** das Projekt
4. **Verbessert** das Projekt

1. **Sucht** den für euch schlimmsten Code Smell für das es eine Comparison gibt
2. **Wendet** Comparison an
3. Tests **Grün?** **Hakt** Comparison ab
4. **REPEAT**



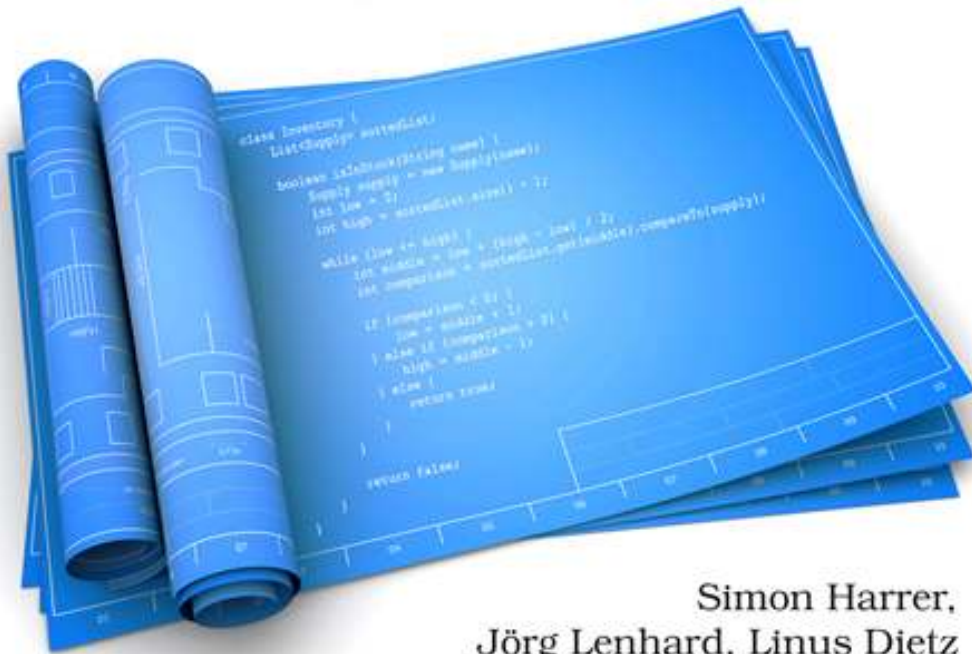
Ihr habt 2 Stunden Zeit. **Viel Spaß!**

Retro

The
Pragmatic
Programmers

Java by Comparison

Become a Java Craftsman
in 70 Examples



Simon Harrer,
Jörg Lenhard, Linus Dietz
Foreword by Venkat Subramaniam
Edited by Andrea Stewart

Danke fürs Teilnehmen!

Euch hat das Buch gefallen?

Wir würden uns über eine Bewertung
auf Amazon und GoodReads freuen!

Dr. **Simon** Harrer
simon.harrer@innoq.com

 [simonharrer](#)

Linus Dietz
linus.dietz@tum.de

 [Lynyus](#)