

Clean Code

Warum sich sauberer Code lohnt!



JUGS-Referat von Roland Gisler

Inhalt

- Teil 1: Clean Code
- Teil 2: Clean Code Developer
- Teil 3: Clean Code [Developer] – Praxistipps
- Fragen und Diskussion



clean code developer

Teil 1: Clean Code



Woher kommt 'Clean Code'?



Robert C. Martin



Clean Code – das Buch

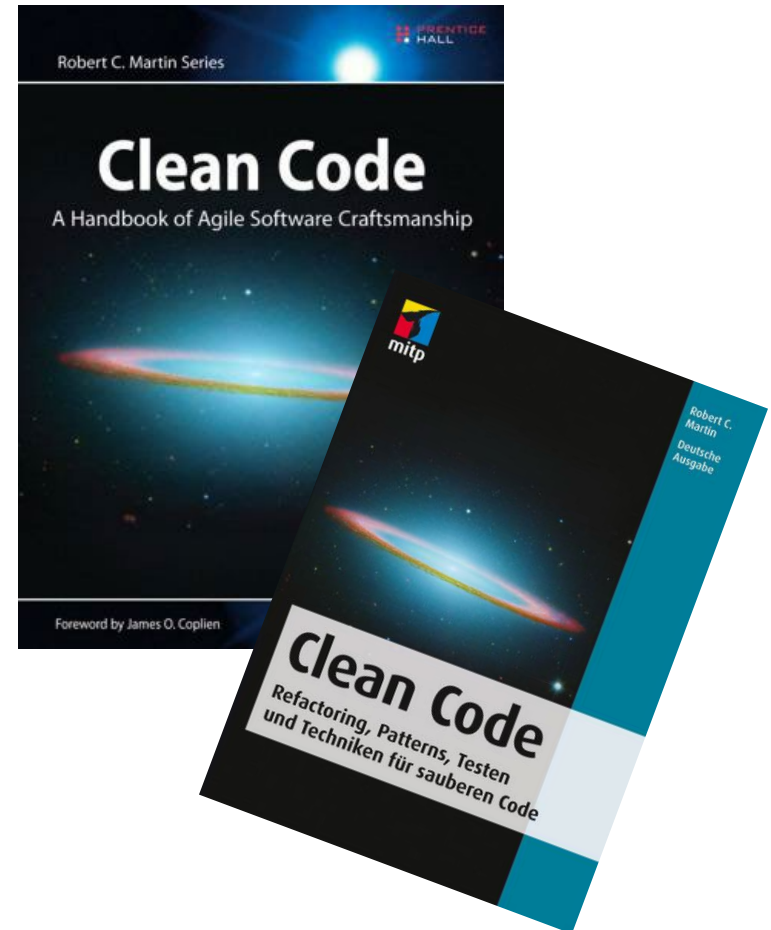
Robert C. Martin:
Clean Code

A Handbook of Agile Software Craftsmanship

Prentice Hall, März 2009,

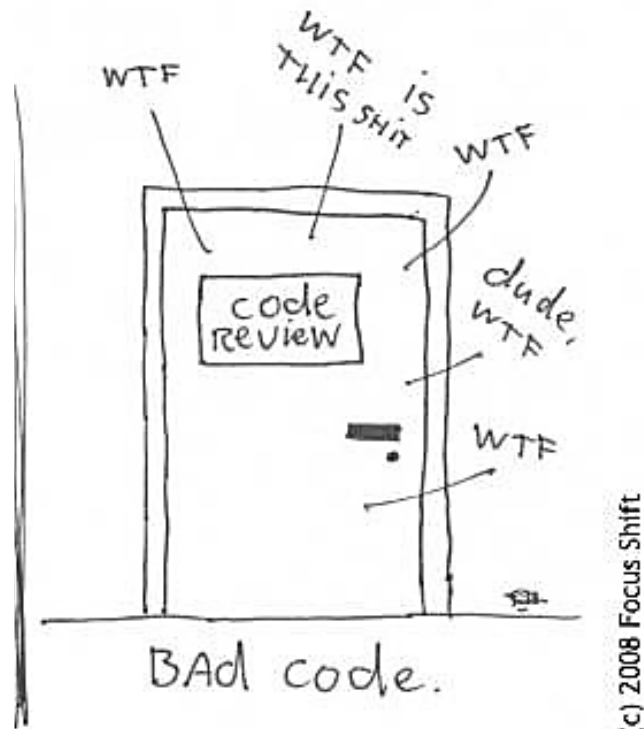
ISBN: 978-0-13-235088-4 (englisch)

ISBN: 978-3-8266-5548-7 (deutsch)



Code-Qualität verbessern!

The ONLY valid measurement
of code quality: WTFs/minute



SW-Entwicklung als ein «echtes» Handwerk!



Themen von «Clean Code»

Klassen

Fehlerhandling

Emergenz

Namensgebung

Funktionen

Unit-Tests

Nebenläufigkeit

Grenzen

Formatierung

Objekte und Datenstrukturen

Kommentare

Systeme

Soll guter Quellcode kommentiert sein?

Quellcode Kommentare

- Kommentare machen keinen guten Code!

Kommentieren Sie schlechten Code nicht – schreiben Sie ihn um!

Brian W. Kernighan / P. J. Plauger

- Kommentare (und auch Dokumentationen) lügen!
 - Nur im Code liegt die Wahrheit!
- Kommentare sind ein notwendiges Übel!
 - fundamentaler Wechsel des Paradigmas!

Gute Kommentare

- Grundsatz: **Der beste Kommentar ist derjenige, den man gar nicht zu schreiben braucht!**
 - Energie besser in guten, selbsterklärenden Code stecken!
- Notwendige, akzeptable Kommentare können sein:
 - juristische Kommentare (Copyright etc.)
 - **TODO**-Kommentare (aber nur temporär!)
 - verstärkende, unterstreichende Kommentare, welche Dinge hervorheben, die sonst zu unauffällig wären
 - Kommentare zur (Er-)Klärung der übergeordneten Absicht oder zur expliziten Warnung vor Konsequenzen

Schlechte Kommentare (1/2)

- Redundante Kommentare ([CCD:DRY](#))
 - reine Wiederholungen dessen, was schon der Code sagt
- Irreführende Kommentare
 - falsche oder unpräzise Formulierungen
- vorgeschriebene oder erzwungene Kommentare
 - sture JavaDoc Kommentare, nur damit sie da sind.
- Tagebuch- oder Changelog-Kommentare
 - heute haben wir Versionskontrollsysteme!
- Positionsbezeichner und Banner
 - zur optischen Unterteilung von grossen Quellcodedateien

Schlechte Kommentare (2/2)

- Zuschreibungen und Nebenbemerkungen
 - Hinweise auf Autor, sinnlose Zusatzbemerkungen
- Auskommentierter Code
 - Ein Todsünde! Kurioserweise traut sich (fast) niemand diesen zu löschen. Wir haben Versionskontrollsysteme!
- HTML(-formatierte) Kommentare
 - Kommentar muss im Code direkt lesbar sein. Nicht erst in der JavaDoc.
- zu viel Kommentar / Information
 - endlose Abhandlungen über Gott und die Welt

Beispiel: Ein guter Kommentar?

```
// Timeout in Millisekunden  
int time = 100;
```

- Der Kommentar kann entfallen, wenn der Code verbessert wird:

```
int timeoutMilliseconds = 100;
```

- ➔ Bessere Namensgebung erspart uns viele Kommentare!
 - Gute Namensgebung ist aber anspruchsvoll!

Themen von Clean Code

Klassen

Fehlerhandling

Emergenz

Namensgebung

Funktionen

Unit-Tests

Nebenläufigkeit

Grenzen

Formatierung

Objekte und Datenstrukturen

Kommentare

Systeme

Gute Namensgebung als Herausforderung


- Überlege dir den Namen einer Klasse so gut, wie den Namen eines Kindes
 - Ein Kind (und die Klasse) trägt ihn ein Leben lang!
 - Besonders heikel: Interfaces!
- Ein richtig guter Name sollte
 - absolut zweckbeschreibend sein
 - Fehlinformationen vermeiden
 - Unterschiede deutlich machen (differenzierend sein)
 - gut aussprechbar und gut suchbar sein
 - möglichst keine Codierungen enthalten

Namensgebung: Heuristiken N1 – N7

- **N1:** Beschreibende Namen wählen
- **N2:** Namen passend zur Abstraktionsebene wählen
- **N3:** Standardnomenklatur verwenden
- **N4:** Eindeutige Namen wählen
- **N5:** Namenlänge abhängig vom Geltungsbereich
- **N6:** Codierungen vermeiden
- **N7:** Namen sollten auch Nebeneffekte beschreiben

Beispiel 1: Gute Namensgebung?

```
...
String bar = getKnownHosts();
if(bar != null) {
    boolean foo = true;
    File goo = new File(bar);
    if(!goo.exists()) {
        foo = false;
        if(userinfo != null) {
            foo = userinfo.promptYesNo(bar);
            goo = goo.getParentFile();
            if(foo && goo != null && !goo.exists()) {
                ...
            }
        }
    }
}
...
}
```



Quellcodeausschnitt aus JSch-Library (KnownHosts.java); Copyright (c) 2002-2012 ymnk, JCraft,Inc. All rights reserved.

Beispiel 2: Gute Namensgebung!

```
String removeLastClosingCurlyBrace(String message) {  
    ...  
}
```

Themen von Clean Code

Klassen

Fehlerhandling

Emergenz

Namengebung ✓

Unit-Tests

Funktionen

Nebenläufigkeit

Grenzen

Formatierung

Objekte und Datenstrukturen

Kommentare ✓

Systeme

Teil 2: Clean Code Developer



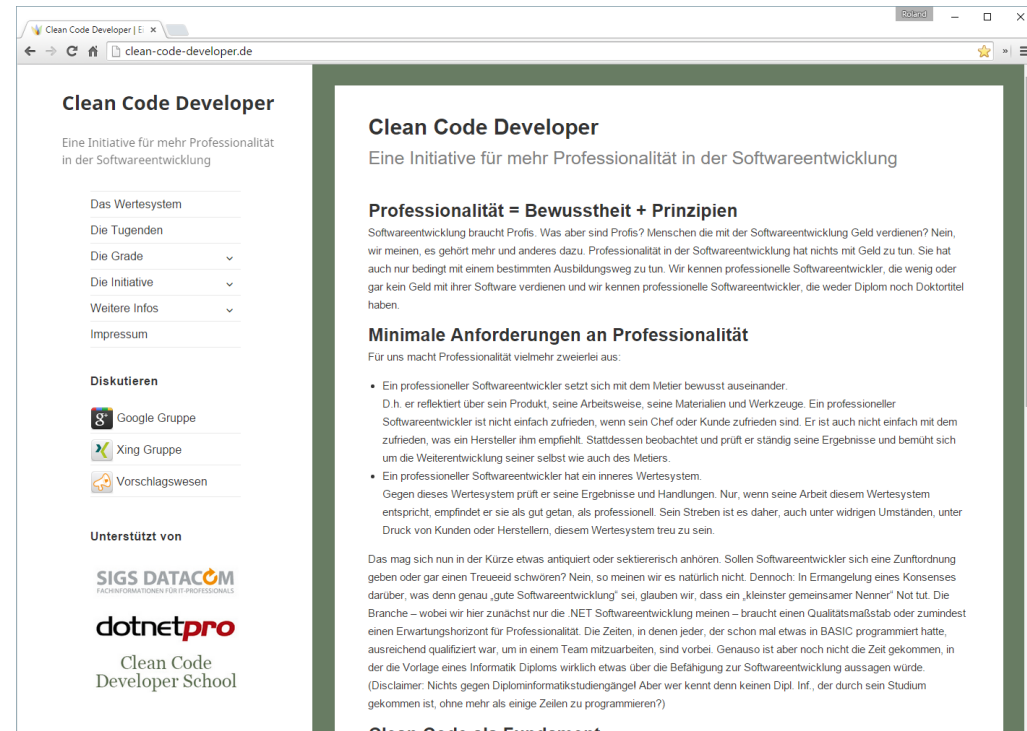
Woher kommt 'Clean Code Developer'?



Ralf Westphal und Stefan Lieser

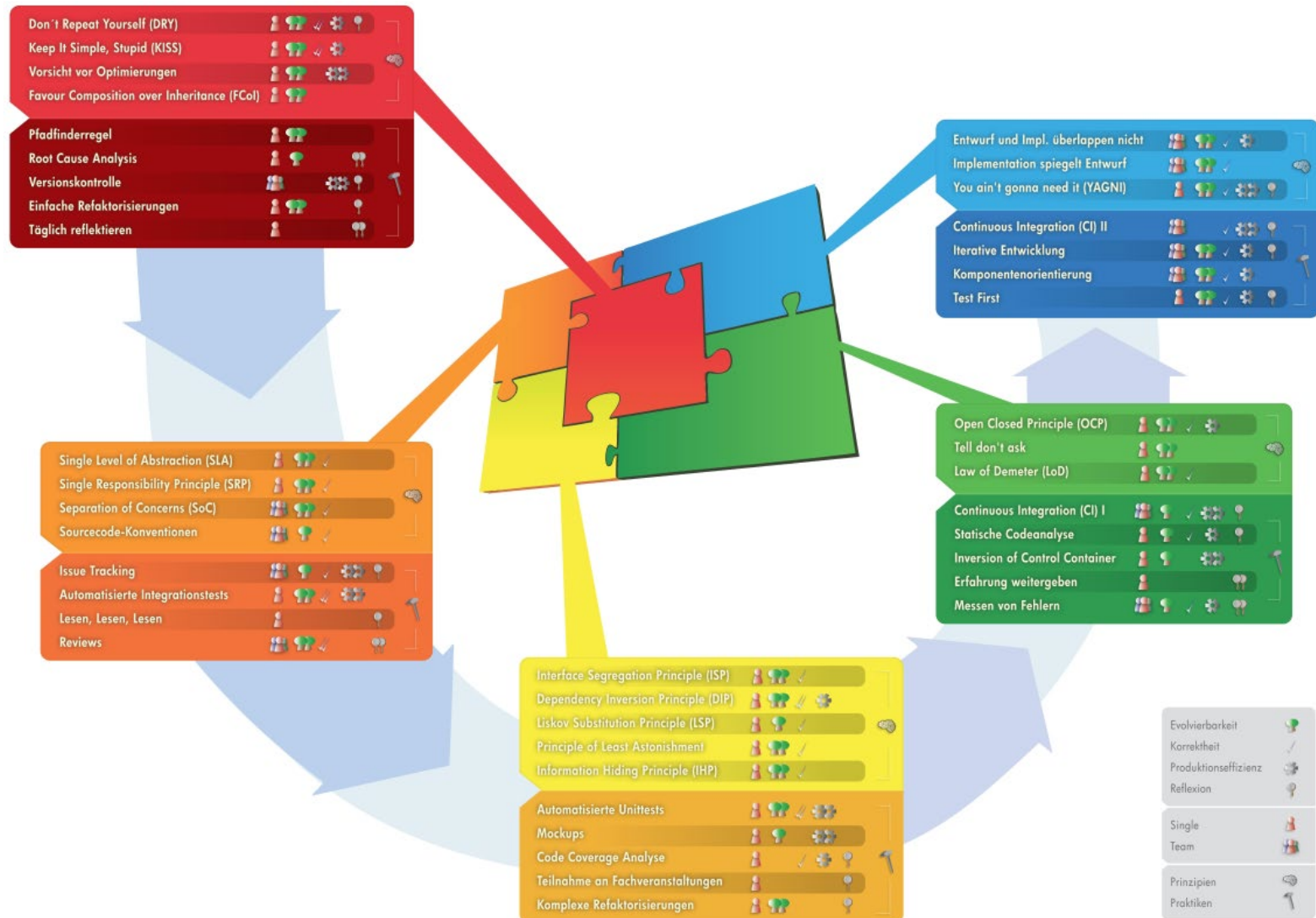
Was ist 'Clean Code Developer'?

- Clean Code Developer ist eine wohldefinierte Auswahl von Prinzipien und Praktiken
- Basis ist ein Wertesystem:
 - Evolvierbarkeit
 - Korrektheit
 - Produktionseffizienz
 - Reflexion



<http://www.clean-code-developer.de>

Iteration über sieben Grade



Iteration über sieben Grade



- Grade: **Schwarz**, **Rot**, **Orange**, **Gelb**, **Grün**, **Blau**, **Weiss**
 - In Anspielung an die verschiedenen Gürtel im Judo
- Jeder einzelne Grad fokussiert auf eine relativ kleine, überschaubar ausgewählte Menge von Prinzipien und Praktiken
 - Insgesamt sind es **42** einzelne Items
 - Überblickbare Ziele und doch stetige Verbesserung.

[illegible]

- # Gratulation!

clean code developer

Der zweite Grad – Rot

- Prinzipien
 - **Don't Repeat Yourself (DRY)**
 - **Keep it simple, stupid (KISS)**
 - Vorsicht vor Optimierungen
 - **Favour Composition over Inheritance (FCoI)**
- Praktiken
 - Die Pfadfinderregel beachten
 - **Root Cause Analysis (RCA)**
 - Ein Versionskontrollsystem einsetzen
 - Einfache Refaktorisierungsmuster anwenden
 - Täglich reflektieren

Don't Repeat Yourself (DRY)	1	2	3	4
Keep It Simple, Stupid (KISS)	1	2	3	4
Vorsicht vor Optimierungen	1	2	3	4
Favour Composition over Inheritance (FCoI)	1	2		

Pfadfinderregel	1	2		
Root Cause Analysis	1	2	3	4
Versionskontrolle	1	2	3	4
Einfache Refaktorisierungen	1	2	3	4
Täglich reflektieren	1	2		

* siehe Bloch: Effective Java







Der dritte Grad – Orange

- Prinzipien
 - **Single Level of Abstraction (SLA)**
 - **Single Responsibility Principle (SRP)**
 - **Separation of Concerns (SoC)**
 - Source Code Konventionen: Namensregeln, Kommentare
- Praktiken
 - Issue Tracking
 - Automatisierte Integrationstests
 - Lesen, Lesen, Lesen
 - Reviews



Der vierte Grad – Gelb

- Prinzipien
 - **I**nterface **S**egregation **P**rinciple (ISP)
 - **D**ependency **I**nversion **P**rinciple (DIP)
 - **L**iskov **S**ubstitution **P**rinciple (LSP)
 - Principle of Least Astonishment
 - **I**nformation **H**iding **P**rinciple (IHP)
- Praktiken
 - Automatisierte Unit Tests
 - Mockups (Testattrappen)
 - Code Coverage Analyse
 - Teilnahme an Fachveranstaltungen
 - Komplexe Refaktorisierungen

Interface Segregation Principle (ISP)		/
Dependency Inversion Principle (DIP)		//
Liskov Substitution Principle (LSP)		/
Principle of Least Astonishment		/
Information Hiding Principle (IHP)		/
Automatisierte Unittests		
Mockups		/
Code Coverage Analyse		/
Teilnahme an Fachveranstaltungen		/
Komplexe Refaktorisierungen		/

Der fünfte Grad – Grün

- Prinzipien
 - **O**pen **C**losed **P**rinciple (OCP)
 - Tell, don't ask
 - Law of Demeter
- Praktiken
 - **C**ontinuous **I**ntegration (CI) I
 - Statische Codeanalyse (Metriken)
 - Inversion of Control Container
 - Erfahrung weitergeben
 - Messen von Fehlern



Der sechste Grad – Blau

- Prinzipien
 - Implementation spiegelt Entwurf
 - Entwurf und Implementation überlappen nicht
 - **You Ain't Gonna Need It (YAGNI)**
- Praktiken
 - **Continuous Integration (CI) II**
 - Iterative Entwicklung
 - Komponentenorientierung
 - Test First



Der siebte Grad – Weiss



- Weisser Grad vereinigt alle Prinzipien und Praktiken der farbigen Grade
- Eine gleichschwebende Aufmerksamkeit ist jedoch sehr schwer zu halten
 - darum beginnt der Clean Code Developer im Gradesystem nach einiger Zeit wieder von vorne
- Die zyklische Wiederholung bringt stetige Verbesserung auf der Basis von überschaubaren Schwerpunkten
- **CCD wird somit zur verinnerlichten Einstellung!**

Don't Repeat Yourself (DRY)



Keep It Simple, Stupid (KISS)



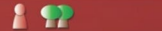
Vorsicht vor Optimierungen



Favour Composition over Inheritance (FCoI)



Pfadfinderregel



Root Cause Analysis



Versionskontrolle



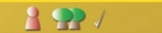
Einfache Refaktorisierungen



Täglich reflektieren



Single Level of Abstraction (SLA)



Single Responsibility Principle (SRP)



Separation of Concerns (SoC)



Sourcecode-Konventionen



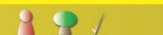
Interface Segregation Principle (ISP)



Dependency Inversion Principle (DIP)



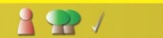
Liskov Substitution Principle (LSP)



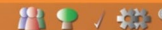
Principle of Least Astonishment



Information Hiding Principle (IHP)



Issue Tracking



Tests



Entwurf und Impl. überlappen nicht



Implementation spiegelt Entwurf



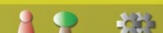
You ain't gonna need it (YAGNI)



Automatisierte Unittests



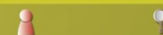
Mockups



Code Coverage Analyse



Teilnahme an Fachveranstaltungen



Komplexe Refaktorisierungen



Continuous Integration (CI) II



Iterative Entwicklung



Komponentenorientierung



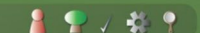
Test First



Continuous Integration (CI) I



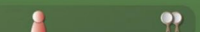
Statische Codeanalyse



Inversion of Control Container



Erfahrung weitergeben



Messen von Fehlern



Teil 3: Praxistipps



Just do it!

Reviews. Reviews? Reviews!

- Mit Abstand am Effizientesten!
- Anfangs alleine oder in kleinen (2er-) Teams!
- Erst später mit mehr Teilnehmern (Clean Coder!)
- Wichtig: Offene, vertrauensvolle Atmosphäre!
 - Nicht als QS-Massnahme!



Erfahrungen weitergeben

- Eigene Clean Code Erfahrungen und Erlebnisse weitergeben
- Beispiel: «Clean Code Snacks»
 - (sehr) kurze Präsentationen (5'–10')
 - z.B. unmittelbar vor oder nach der Znünipause

T8: Hinweise von Coverage Patterns nutzen

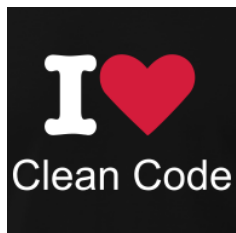
- Wenn ein Testfall scheitert: Codeabdeckung studieren!
 - Manchmal erkennt man aufgrund der Zeilen die **(nicht)** ausgeführt wurden sehr schnell den Fehler!
- **Failure** in JUnit-Test:

Punkt in Quadrant 2
expected:<2> but was:<3>
- «Pattern» der Codeabdeckung:
 - **if-Zweig wurde nicht ausgeführt!**
 - **Bedingung ist falsch formuliert!**
- Vergleiche mit Beispiel zu T2: Nicht nur die grünen und roten Bereiche ansehen, sondern auch die zusätzlichen Informationen studieren und nutzen!

```
57. int quadrant = NO_QUADRANT;
58. if ((x != 0) && (y != 0)) {
59.     if (x > 0) {
60.         if (y > 0) {
61.             quadrant = QUADRANT_1;
62.         } else {
63.             quadrant = QUADRANT_4;
64.         }
65.     } else {
66.         if (y == 0) {
67.             quadrant = QUADRANT_2;
68.         } else {
69.             quadrant = QUADRANT_3;
70.         }
71.     }
72. }
73. return quadrant;
```

Aufmerksamkeit für Clean Code wecken

- Clean Code Ringe tragen
 - Aufmerksamkeit und Interesse wecken
- Zettel mit einzelnen Themen aufhängen
 - Kühlschranktüre?
- Clean Code Buch
 - kaufen und verschenken
- Mausmatten
- Broschüre



Optische Präsenz!

Werkzeuge

- Checkstyle, PMD, Spotbugs
 - Namenslängen, Funktions- und Klassengrößen, Formatierung
 - Designprinzipien, Abhängigkeiten
 - Potentielle Programmierfehler
- SonarQube
 - Statistiken zur Verbesserung des Codes
 - Sensoren für 'gefährliche' Verletzungen
- Messung der Codeabdeckung
 - JaCoCo / EcJemma, JCoBERTura, Clover etc.
 - Vereinfachung der Fehlersuche!

Errors	Warnings	Info
1	0	0

Issue	Severity	Message
1	High	...

Issue	Severity	Message
1	High	...

Issue	Severity	Message
1	High	...

Schlusswort

clean code developer

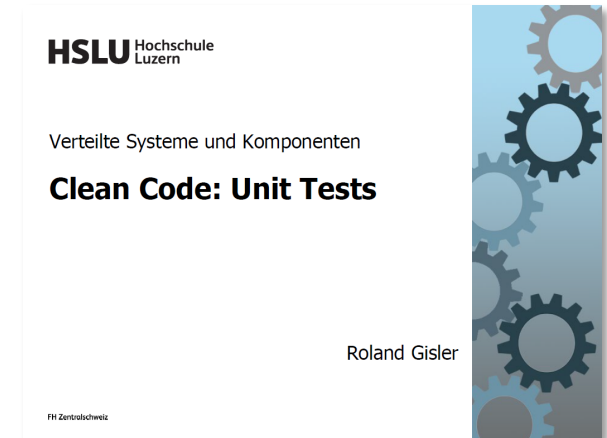
- **Clean Code** und **Clean Code Developer**
 - Verinnerlichte Grundhaltung um **guten** Code zu schreiben!
- SW-Entwicklung als ein **SOLIDES Handwerk**
 - Wertschätzung und Qualitätsbewusstsein
- Pfadfinder-Regel!
 - Schon kleine Verbesserungen sind ein grosser Schritt hin zu besserer Software!



Just do it!

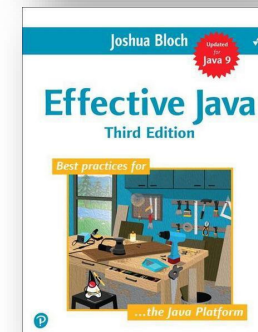
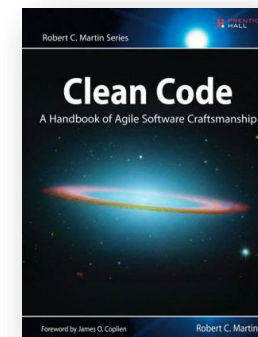
Ergänzende Präsentationen zu Clean Code

- Wie implementieren wir gute (JUnit-)Tests nach Clean Code:
EP_53_CleanCodeUnitTests
- Wie entwirft und implementiert man gute Funktionen (Methoden) nach Clean Code:
EP_54_CleanCodeFunctions



Literaturhinweise

- Robert C. Martin (Uncle Bob):
<https://sites.google.com/site/unclebobconsultingllc/>
- Ralf Westphal & Stefan Lieser:
<http://www.clean-code-developer.de>
- Robert C. Martin: **Clean Code**
A Handbook of Agile Software Craftsmanship
Prentice Hall, März 2009
ISBN: 978-0-13-235088-4 (englisch)
ISBN: 978-3-8266-5548-7 (deutsch)
- Joshua Bloch: **Effective Java**
Best practices for the Java Plattform
Third Edition, Juni 2008
Pearson Addison-Wesley
ISBN: 978-0-13-468599-1





Besten Dank!

Quellen

- <https://sites.google.com/site/unclebobconsultingllc/>
- <http://www.frs.w.de/walz.htm>
- <http://www.lifehack.org/articles/productivity/how-to-do-an-ultimate-gtd-weekly-review-lifehack-lessons.html>