Clean Code

HERKUNFT

TPM

Total Productive Maintenance

- Qualitätsansatz
- ~1960, japanische Autoindustrie
- Konzentration auf Instandhaltung des Arbeitsplatzes
- ähnlich Lean Produktion
- Fundament: 5S-Prinzipien

Seiri
: Aussortieren; Übersicht schaffen - wo finde ich Dinge wieder, Namensgebung

Seiton
: Ordentlichkeit; Code sollte da stehen, wo ich ihn erwarte

Seiso
: Säubern; Abfall und Einzelteile entfernen

Seiketsu
: Standardisierung; konsistenter Codierstil

Shutsuke
: (Selbst-) Disziplin & ständige Verbesserung

Qualität ist das Ergebnis einer Million selbstloser Akte der Sorgfalt.

— Robert “Uncle Bob” Martin

> *Wir (Entwickler) sind Autoren. Ein Merkmal von Autoren ist es, dass sie Leser haben.*
>
> — Robert “Uncle Bob” Martin

Sauberer Code kann von anderen Entwicklern gelesen und verbessert werden.

— Dave Thomas

CHAOS IM CODE

- je älter ein Projekt, desto höher der Aufwand, neue Funktionen hinzuzufügen
- damit die Produktivität wenigstens annähernd gleich bleibt, wird (leider) der Fokus der Arbeit auf neue Funktionen gelegt
  - Folge: Code verrottet - wichtige Basis-Arbeiten werden vernachlässigt
    - keine neuen Tests
    - Konzepte werden durch Ausnahmen aufgeweicht
    - Dokumentation wird nicht nachgezogen
- Gesetz von LeBlanc: Später heißt niemals

CHAOS IM CODE

- Schlussfolgerung:
  - es reicht nicht aus, guten Code zu schreiben
  - Code muss auch *sauber gehalten* werden
  - sofort & kontinuierlich

Leave the campground cleaner than you found it

— Robert “Uncle Bob” Martin

AUSSAGEKRÄFTIGE NAMEN

ZWECKBESCHREIBENDE NAMEN

```
public List<int[]> getThem() {
   List<int[]> list1 = new ArrayList<int[]>();
   for (int[] x : theList)
      if (x[0] == 4)
         list1.add(x);
   return list1;
}
```

- Kontext geht nicht aus dem Code hervor
- Code ist implizit, sollte aber explizit sein

```
public List<int[]> getFlaggedCells() {
   List<int[]> flaggedCells = new ArrayList<int[]>();
   for (int[] cell : this.gameBoard)
      if (cell[STATUS_VALUE] == FLAGGED)
         flaggedCells.add(cell);
   return flaggedCells;
}
```

FEHLINFORMATIONEN VERMEIDEN

- keine irreführenden Hinweise, z.B. für eine Gruppe von Konten:

```
private Map accountList;
```

- zwei Namen sollten sich nicht geringfügig unterscheiden, z.B.

```
XYZControllerForEfficientHandlingOfStrings
XYZControllerForEfficientStorageOfStrings
```

UNTERSCHIEDE DEUTLICH MACHEN

```
public static void copyChars(char c1[], char c2[]) {
   for (int i=0; i < c1.length; i++) {
      c2[i] = c1[i];
   }
}
```

```
public static void copyChars(char source[], char destination[]) {
   for (int i=0; i < source.length; i++) {
      destination[i] = source[i];
      }
}
```

- Namen wie `c1` sind nicht irreführend, sondern informationsleer
- zusammengesetzte Klassennamen können auch informationsleer sein
  - `Product`
  - `ProductInfo`
  - `ProductData`

AUSSPRECHBARE NAMEN VERWENDEN

```
class DtaRcrd102 {
   private Timestamp genymdhms;
   private Timestamp modymdhms;
}
```

```
class DtaRcrd102 {
   private Timestamp genymdhms;
   private Timestamp modymdhms;
}
```

ymdhms

Year, Month, Day, Hours …

```
class DtaRcrd102 {
   private Timestamp genymdhms;
   private Timestamp modymdhms;
}
```

ymdhms

Year, Month, Day, Hours …

```
class Customer {
   private Timestamp generationTimestamp;
   private Timestamp modificationTimestamp;
}
```

SUCHBARE NAMEN VERWENDEN

```
int s = 0;
for (int j=0; j<34; j++) {
   s += (t[j]*4)/5;
}
```

- Die Länge eines Namens sollte der Größe seines Geltungsbereichs entsprechen
- Suche nach *t* oder *5* ergibt in der gesamten Codebasis viele Treffer

```
int realDaysPerIdealDay = 4;
const int WORK_DAYS_PER_WEEK = 5;
int sum = 0;
for (int j=0; j < NUMBER_OF_TASKS; j++) {
   int realTaskDays = taskEstimate[j] * realDaysPerIdealDay;
   int realTaskWeeks = (realTaskDays / WORK_DAYS_PER_WEEK);
   sum += realTaskWeeks;
}
```

CODIERUNGEN VERMEIDEN

```
// Datentypen
private String szVorname;
private Integer nId;
// Geltungsbereich
private String pri_szVorname;
public Integer pub_nId;
```

- Codierung von Informationen in Namen von Variablen
  - Datentyp oder Geltungsbereich
  - Ungarische Notation
- Nachteile
  - Änderungen müssen überall nachgezogen werden
  - Präfixe und Suffixe werden bald vom Entwickler ignoriert

METHODENNAMEN

- Verben verwenden, z.B.
  - `downloadEmailAttachments()`
- nur ein Wort pro Konzept
  - *fetch*, *retrieve*, *get* … sind Synonyme

DOMÄNEN NAMEN

- Problemdomäne
  - Begriffe/Konzepte des Bereichs, für den die Software bestimmt ist
  - z.B. `BeneficialOwner`
    - Bezug auf wirtschaftlich Berechtigten eines Bankkontos
- Lösungsdomäne
  - Begriffe/Konzepte der Informatik, Algorithmen, Pattern
  - z.B. `AccountVisitor`
    - Bezug auf Visitor-Pattern

FUNKTIONEN

BEISPIEL

HtmlUtil.java SetupTeardownIncluder.java

```
public class HtmlUnit {
  public static String testableHtml(
     PageData pageData,
     boolean includeSuiteSetup
   ) throws Exception
{
  WikiPage wikiPage = pageData.getWikiPage();
  StringBuffer buffer = new StringBuffer();
  if (pageData.hasAttribute("Test")) {
    if (includeSuiteSetup) {
      WikiPage suiteSetup =
        PageCrawlerImpl.getInheritedPage(
            SuiteResponder.SUITE_SETUP_NAME, wikiPage
        );
  ...
```

- Beispiel aus Fitnesse
  - FitNesse begann als ein HTML und Wiki "front-end" für FIT ("Framework for Integrated Testing")
  - Wiki Seite == Page
  - Test-Suite == Zusammenfassung mehrere Tests
  - Teststruktur
    - ggf. Suite Setup
    - Setup
    - Test (== pageDate)
    - TearDown
    - ggf. Suite TearDown

```
public class HtmlUnit {
  public static String testableHtml(
     PageData pageData,
     boolean includeSuiteSetup
    ) throws Exception
    {
      WikiPage wikiPage = pageData.getWikiPage();
    StringBuffer buffer = new StringBuffer();
    if (pageData.hasAttribute("Test")) {
      if (includeSuiteSetup) {
        WikiPage suiteSetup =
          PageCrawlerImpl.getInheritedPage(
              SuiteResponder.SUITE_SETUP_NAME, wikiPage
          );
```

ERSTE VERBESSERUNG

```
public static String renderPageWithSetupsAndTeardowns(
   PageData pageData, boolean isSuite
) throws Exception {

   boolean isTestPage = pageData.hasAttribute("Test");
   if (isTestPage) {
      WikiPage testPage = pageData.getWikiPage();
      StringBuffer newPageContent = new StringBuffer();
      includeSetupPages(testPage, newPageContent, isSuite);
      newPageContent.append(pageData.getContent());
      includeTearDownPages(testPage, newPageContent, isSuite);
      pageData.setContent(newPageContent.toString());

   }

   return pageData.getHtml();
}
```

KLEIN

- Funktionen sollten klein sein

Wie kann das erreicht werden?

- keine verschachtelten Strukturen
- die *einzig erlaubte* Einrückungstiefe sollte dann möglichst nur eine Anweisung enthalten

```
public static String renderPageWithSetupsAndTeardowns(
   PageData pageData, bool isSuite
) throws Exception {
   if (isTestPage(pageData)) {
      includeSetupAndTeardownPages(pageData, isSuite)
   }
   return pageData.getHtml();
}
```

EINE AUFGABE ERFÜLLEN

- **eine** Aufgabe
  - Wenn alle Schritte einer Funktion eine Abstraktionsebene unter dem Zweck liegen, der durch den Namen ausgedrückt wird
- Hilfsmittel
  - einen UM-ZU-Absatz formulieren

> *UM RenderPageWithSetupsAndTeardowns ausZUführen, prüfen wir, ob eine Seite eine Testseite ist, und wenn dies der Fall ist, schließen wir die Setups und Teardowns ein. In beiden Fällen stellen wir die Seite in HTML dar.*

BESCHREIBENDE NAMEN

- gute Namen für kleine Funktionen finden, die eine Aufgabe erledigen
- lange beschreibende Namen sind besser als kurze geheimnisvolle Namen
- lange Namen sind besser als lange Kommentare
- mehrere Wörter per Konvention trennen
  - CamelCaseSchreibweise
- verschiedene Namen ausprobieren und Code lesen
  - IDE unterstützt das
- Namen sollten in einem Modul konsistent sein
  - Synonyme vermeiden

FUNKTIONSARGUMENTE

- je weniger Argumente, desto besser
  - jedes Argument erfordert konzeptionelle Kraft beim Lesen
  - Name und Typ des Arguments könnten zu anderer Abstraktionsebene gehören
  - das Testen einer Funktion wird aufwändiger
    - die Kombinationen aller Argumente mit allen möglichen Werten

FUNKTIONSARGUMENTE

- Output-Argumente vermeiden, da ungewohnt
  - Input: Argumente
  - Output: Rückgabewert

FUNKTIONSARGUMENTE

- Argument als Output verwendet

```
public static void splitToList(String source, List parameter) {
   String[] array = source.split(",");
   parameter.addAll(Arrays.asList(array));
}
```

FUNKTIONSARGUMENTE

- Argument als Output verwendet

```
public static void splitToList(String source, List parameter) {
   String[] array = source.split(",");
   parameter.addAll(Arrays.asList(array));
}
```

- Rückgabewert als Output

```
public static List splitToList(String source) {
   String[] array = source.split(",");
   return Arrays.asList(array);
}
```

FLAG-ARGUMENTE

- Hinweis darauf, dass mehrere Aufgaben erfüllt werden

```
// Aufruf
  render(true);
// Definition
class Renderer {
   void render(boolean isSuite) {}
}
```

FLAG-ARGUMENTE

- Besser mehrere Methoden

```
// Definition
class Renderer {
   void renderForSuite() {}
   void renderForSingleTest() {}
}
```

DYADISCHE FUNKTIONEN

- Funktionen mit 2 Argumenten
- Verwender muss die Reihenfolge und Bedeutung kennen
  - oder Definition nachschlagen → Aufwand!
- oft unvermeidbar

```
// Aufruf
  int result = getResult(); // 24
  assertEquals(24, result);
// Definition
class Assert {
   void assertEquals(int expected, int actual) {}
}
```

NEBENEFFEKTE VERMEIDEN

```
public boolean checkPassword(String userName, String password){
   User user = UserGateway.findByName(userName);
   if (user != User.NULL) {
      if (user.password.equals(password)) {
         Application.loginUser(user);
         return true;
      }
   }
   return false;
}
```

ANWEISUNG ODER ABFRAGE

- Funktion sollte entweder
  - etwas tun, oder
  - etwas antworten

```
public boolean set(String attribute, String value){
   if (internalList.contains(attribute)) {
      internalList.set(attribute, value);
      return true;
   } else {
      return false;
   }}
// mögliche Verwendung
  if (set("username", "robkle")) ...
```

FEHLERCODE VS EXCEPTIONS

- Fehlercode
  - muss sofort geprüft werden
- Exception
  - kann am Ende behandelt werden
  - ist ebenfalls eine Aufgabe
    - kann in separate Funktion ausgelagert werden

Beispiel mit Fehlercodes inkl. Behandlung

```
if (deletePage(page) == E_OK) {
   if (registry.deleteReference(page.name) == E_OK) {
      if (ConfigKeys.deleteKey(page.name.makeKey()) == E_OK) {
         logger.log("page deleted");
      } else {
         logger.log("config key not deleted");
      }
   } else {
      logger.log("deleteReferences from registry failed");
   }
} else {
   logger.log("delete failed");
}
```

Beispiel mit Exceptionbehandlung

```
try {
   deletePage(page);
   registry.deleteReference(page.name);
   ConfigKeys.deleteKey(page.name.makeKey());
}
catch (Exception e)
{
   logger.log(e.getMessage());
}
```

Exceptionsbehandlung auslagern

```
public void delete(Page page) {
   try {
      deletePageAndAllReferences();
   }
   catch (Exception e)
   {
      logError(e);
   }
}

public void deletePageAndAllReferences(Page page) {...}
public void logError(Exception e) {...}
```

DON’T REPEAT YOURSELF

- Viele Innovationen der Software-Entwicklung haben nur ein Ziel
  - Duplizierung zu vermeiden
  - Wiederverwendung fördern
- Duplikate könnten bei einem Umbau vergessen werden
- Beispiel
  - HtmlUtil.java

KOMMENTARE

ÜBER KOMMENTARE

- Kommentare sind kein Ersatz für schlechten Code
- Kommentare können durch selbsterklärenden Code vermieden werden

```
// Check to see, if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) &&
   employee.age > 65)
   ...
```

Alternative

```
if (employee.isEligibleForFullBenefits())
   ...
```

GUTE KOMMENTARE

- Copyright Header
- nicht-triviale Methoden-Beschreibung
- nicht-triviale Klassen-Beschreibung
- Erklärung der Absichten
- Klarstellungen
- Warnung vor Konsequenzen
- TODO-Kommentare

SCHLECHTE KOMMENTARE

- Geraune
- Redundante Kommentare
  - *Wiederholung* des Codes
- irreführende Kommentare
- Positionsbezeichner
- Kommentare hinter schließenden Klammern
- Auskommentierter Code