

Verteilte Systeme und Komponenten

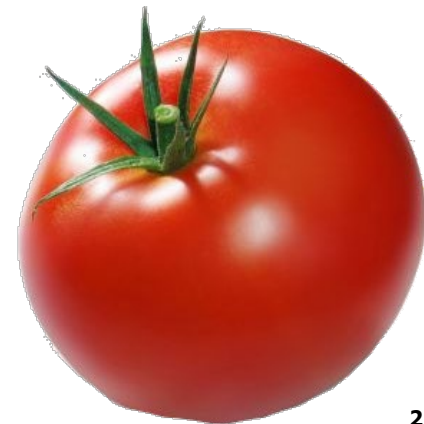
Clean Code: Funktionen

Roland Gisler



Gute Funktionen nach Clean Code

- Clean Code: Kapitel 3 – Funktionen; Smells: F1 – F4
- Sehr viele Anforderungen an «gute» Funktionen:
 - Klein sein und nur eine Aufgabe erfüllen.
 - Nur eine Abstraktionsebene enthalten.
 - Geringe Einrücktiefe, möglichst kein **switch** enthalten.
 - Einen guten Namen haben.
 - Möglichst wenig (keine!) Funktionsargumente haben.
 - Keine Flag-Argumente verwenden.
 - Keine Nebeneffekte aufweisen.
 - Mit Exceptions (statt Fehlercodes) arbeiten.
 - u.v.a.m.



Funktionen sollen klein sein

- Grundregel 1: Funktionen sollen **klein** sein!
- Grundregel 2: Funktionen sollen **noch kleiner** sein!
- Eine Funktion sollte auf einen Blick überschaubar sein.
 - Auch ohne AGB-Schriftgrösse und ohne Scrollen.
- Vorteil: Kleine Funktionen sind (einzeln) viel besser und schneller verständlich (→ man muss weniger lesen!)
- Konsequenz von kleinen Funktionen:
 - Es gibt insgesamt viel mehr Funktionen.
 - Da eine Klasse nicht zu viele Methoden haben soll, gibt es auch viel mehr (aber kleinere) Klassen.
 - Beides ist **positiv**, nur schon z.B. bezüglich Testbarkeit!

Pro Funktion nur eine Aufgabe

- Eine Funktion sollte
 - **Eine** Aufgabe erledigen.
 - Diese Aufgabe **gut** erledigen.
 - **Nur** diese Aufgabe erledigen.
- siehe OOP: **S**ingle **R**esponsibility **P**inciple (**SRP**)
- Wie erkennen wir was genau eine Aufgabe ist?
Trick: Einen «**to**»-Absatz («um zu») bilden!
 - Entlehnt aus der Sprache LOGO
(in welcher alle Funktionsnamen mit einem «to» beginnen!)
- Beispiel:
 - **T0 caculateSumme()** → addiere die zwei Summanden.
- Abschnitte in Funktionen: Hinweis auf Verletzung!
 - Vermutlich mehrere Aufgaben in einer Funktion enthalten.

Douglas McIlroy:
„Mache nur eine Sache
und mache sie gut.“
Bekannt als die «Unix-Philosophie»

Nur eine Abstraktionsebene pro Funktion

- **Single Level of Abstraction (SLA)**
 - Schwierig zu erlernen und zu erreichen, braucht Disziplin!
- Beispiel:

```
...  
buffer.append(getPageHeader());  
buffer.append(getPageContent());  
buffer.append(' \n ');  
buffer.append(getPageFooter());  
...
```



- Grosse Gefahr: Wird **SLA** in Funktionen nicht eingehalten, wirken diese wie Magnete für den weiteren Zerfall.
 - Codeerosion im wahrsten Sinn des Wortes!

Generell: **switch**-Anweisungen vermeiden!

- Mit **switch**-Anweisungen...
 - werden Funktionen gross (und werden weiter wachsen).
 - erfüllt eine Funktion typisch mehr als eine Aufgabe.
- Mit einem **switch** verletzt man viele Clean Code-Regeln:
 - **Single Responsibility Prinzip** (siehe OOP: SRP)
 - Es gibt mehr als einen Grund für Änderungen (jeder **case**).
 - **Open Closed Prinzip** (OCP)
 - **switch**-Anweisung muss für Erweiterung geändert werden.
 - **Don't repeat yourself** (siehe OOP: DRY)
 - Ein **switch**-Konstrukt kann sich im Code beliebig oft wiederholen.
➔ Eine **subtile** Form der Duplikation.
- Lösung: **switch** durch polymorphe Konstrukte (z.B. Strategy-Pattern nach GoF) ersetzen.

Anzahl der Funktionsargumente minimieren

- Funktionsargumente sind «schwer» (im Sinn von gewichtig).
- Aus der Perspektive des Aufrufers (Nutzer) betrachten!
 - Bereits ab zwei Argumenten kann man diese vertauschen!
 - Beispiel aus JUnit-Framework:

```
assertEquals(int expected, int actual);
```



- Mal ehrlich: Wer hat die beiden Parameter noch nie versehentlich verwechselt? (Und hat es bemerkt?)
- Bei mehr als drei Argumenten hört der Spass bereits auf!
 - Ohne Konsultation der Dokumentation kaum mehr nutzbar.
 - Aufrufe werden mühsam, lang und unübersichtlich!
 - Horizontales Scrolling oder Zeilenumbrüche? ☹

Empfehlungen aus Clean Code

- **Niladische** Funktionen - **keine** Argumente.
 - Ideal! Funktion ist sehr einfach aufzurufen.
 - Man kann praktisch keine Fehler machen.
- **Monadische** Funktionen – **ein** einziges Argument.
 - Ist ok, kann man machen wenn es nötig ist.
- **Dyadische** Funktionen - **zwei** Argumente*.
 - Wenn es gute Gründe gibt wird es akzeptiert.
- **Triadische** Funktionen - **drei** Argumente.
 - Sollte man wenn immer möglich vermeiden!
- **Polyadische** Funktionen - **mehr** als drei Argumente.
 - Selbst bei sehr guter Begründung: **Nein!** Nicht machen!

Reduktion der Argumente

- Grundsätzlich: Mehr und kleinere Klassen, mit Attributen!
- Häufig können dyadische Funktionen ganz einfach auf monadische Funktionen zurückgeführt werden!
 - Kann schöneres Design zur Folge haben (muss aber nicht).
- Beispiel: Dyadische (reine) Funktion:

```
summe = addiere(summand1, summand2);
```
- Reduziert auf objektorientierte, monadische Form:

```
summe = summand1.addiere(summand2);
```
- Reduktion verbessert auch die Lesbarkeit / Verständlichkeit.
 - Beispiel: AssertJ-Library (mit Fluent-API) für JUnit-Tests.

Keine Flag-Argumente (formale Parameter)

- Reine Flag-Argumente sollte man möglichst vermeiden!
 - Besser die Methode in verschiedene, aber eindeutig benannte Varianten aufteilen.
 - ➔ Auch monadische Funktionen sind nicht immer gut!
- Beispiel:

```
writeFile(myFile, true);
```



- myFile ist klar, aber für was ist das boolean-Flag?
- Besser zwei alternative Methoden:


```
writeFileOverwrite(myFile);  
writeFileAppend(myFile);
```



Keine Nebeneffekte einbauen

- Beispiel einer Funktion:

```
boolean checkPassword(String username, String password) {  
    final User user = UserDirectory.findByName(username);  
    if (user != User.NULL) {  
        final String hash = user.getPasswordHash();  
        final String input = Cryptographer.hash(password);  
        if (input.equals(hash)) {  
            Session.initialize();  
            return true;  
        }  
    }  
    return false;  
}
```



- Gibt es darin einen unerwünschten Nebeneffekt?

Nebeneffekte vermeiden

- Nebeneffekte sind Lügen!
 - Funktionen versprechen eine Aufgabe zu erfüllen, aber sie erledigen noch andere, verborgene Aufgaben.
- Führen zu seltsamen Kopplungen und Abhängigkeiten.
 - Im Fehlerfall nur schwer aufzudecken.
- Möglichkeiten zur Verbesserung:
 - 1. Nebeneffekt entfernen:
Aufruf von **Session.initialize()** entfernen.
 - 2. Der Funktion einen «ehrlichen» Namen geben:
Beispiel: **checkPasswordAndInitializeSession(...)**

Output-Argumente vermeiden

- Output Argumente (call by reference) sind in OO-Sprachen weitgehend überflüssig geworden.
 - Wenn **ein** Rückgabewert/-objekt nicht ausreicht, ist es die Aufgabe von **this** als Output zu agieren!

- Beispiel:

```
appendFooter(s);
```



- Wie ist das jetzt gemeint? Ist **s** ein Input oder Output?
 - Ohne Dokumentation nicht verständlich!
- Erkenntnis: Funktionsargumente werden am natürlichsten als **Input** einer Funktion interpretiert.
 - Darum: Auch immer **nur als Input** verwenden!
 - Nebenbei: Java kennt **kein** «call by reference»!

Anweisungen und Abfragen trennen

- Funktionen sollten entweder...
 - etwas **tun** und damit den Zustand eines Objekts ändern.
 - oder **antworten** und Informationen eines Objekts liefern.
- Aber **nie** beides, weil das verwirrt!
- Beispiel einer Funktion welche diese Forderung verletzt:

```
/*  
 * Setzt den Wert eines benannten Attributes.  
 * Liefert true zurück wenn es geklappt hat.  
 * Liefert false zurück wenn Attribut nicht existiert.  
 */  
public boolean set(String attribute, String value) {  
    ...  
}
```



Anweisungen und Abfragen trennen

- Rückgabewerte einer Funktion «provozieren» den Einsatz in Bedingungen, wo es leicht zu Fehlinterpretationen kommen kann:

```
if (set("username", "Uncle Bob")) { ... }
```

- Wie interpretiert man nun diesen Code?
 - Prüft ob das Attribut **username** auf "Uncle Bob" gesetzt werden konnte *oder* ob dieses existiert?
 - Rein sprachliches Verständnis:
Prüft ob **username** auf "Uncle Bob" gesetzt ist...?
- Lösung: Aufbrechen in zwei Methoden, Abfrage und Anweisung auftrennen:
 - **attributeExists(...)** und **setAttribute(...)**

Keine Fehlercodes, besser Exceptions!

- Grundsätzlich: Fehlercodes sind eine Verletzungen der Anweisungs-/Abfrage-Trennung!
 - Fördern den Einsatz von Funktionen in **if**-Anweisungen.
- Die Fehlercodes müssen sofort ausgewertet werden.
 - Führt zu tief verschachtelten (eingerückten) Strukturen.
 - Sehr unübersichtlich und wartungsunfreundlich!
 - Gefahr der Code-Duplikationen beim Fehlerhandling.
- **Fehlercodes sind Abhängigkeitsmagneten!**
- Darum: Fehlerbehandlung besser mit Exceptions!
 - Ist auch nicht «wirklich» schön, macht den Code aber übersichtlicher und besser wartbar!


Fehlerverarbeitung ist eine Aufgabe

- Nochmal: Eine Funktion sollte nur **eine** Aufgabe haben.
- Fehlerverarbeitung und -behandlung **ist eine** Aufgabe.
- Darum: Fehlerbehandlung in getrennte Methoden auslagern.
 - **try/catch-Konstrukte sind nicht wirklich «schön».**
- Empfehlung: Wenn eine Funktion ein **try/catch**-Konstrukt enthält dann sollte...
 - sie sonst **nichts** anderes machen.
 - das **try** immer das erste Statement sein.
 - der **catch-** (bzw. **finally-**)Block der letzte Block sein.
- So bleibt der Blick immer auf das Wesentliche fokussiert.


Gute Namensgebung von Funktionen

- Gute Namensgebung ist und bleibt eine Kunst!
 - Tendenziell werden die Namen länger und sprechender...
 - ...und bleiben trotzdem kurz weil es mehr Packages gibt!
- Man kann es so machen:

```
package org.framework42;  
public class Universal {  
    public static Object doIt(Object arg1, Object arg2)  
    { ... }  
}
```



```
// Hier berechnen wir etwas!  
x = Double.getValue(Universal.doIt(new Double(230),  
                                   new Double(3.1)));
```



- Was macht dieser Code?

Namensgebung und Grenzwerte

- Oder vielleicht besser so?

```
public class Elektrotechnik {  
    public double berechneLeistung(double spannung,  
                                   double strom) {...}  
}
```



```
leistung = berechneLeistung(230, 3.1);
```

- Empfohlene Grenzwerte für Funktionen:
 - Zeilenlänge maximal 150 Zeichen.
 - Weniger als 20 Zeilen Code (im Schnitt).
 - Keine Funktion sollte jemals länger als 100 Zeilen sein.
 - Möglichst geringe Verschachtelung, maximal zwei Ebenen!

Uncle Bobs Funktions Smells F1 – F4

- **F1:** Zu viele Argumente.
 - Zu viele Argumente verderben den Brei!
 - 0 sind ideal, 1 ist ok, 2 in Ausnahmen, ab 3 sind es **zu viel**.
- **F2:** Output-Argumente.
 - Widersprechen der Intuition / dem was der Leser erwartet.
 - Stattdessen den Zustand des Objektes verändern!
- **F3:** Flag-Argumente.
 - Flag-Argumente zur Steuerung beweisen, dass in einer Funktion mehr als eine Aufgabe realisiert wird.
- **F4:** Tote Funktionen
 - Methoden, die nie aufgerufen werden löschen!
 - Wir haben doch alles im Versionskontrollsystem...

Zusammenfassung – 1/3

- Funktionen sollten möglichst klein sein.
 - Wenige Zeilen, keine Abschnitte, auf einen Blick erfassbar.
- Nur eine einzige Aufgabe (Konzept) pro Funktion.
 - Unix-Philosophie: "Mach nur eine Sache und mach sie gut!"
 - Trick: «To»-Formulierung verwenden.
- Nur eine Abstraktionsebene pro Funktion (SLA).
 - Alles andere macht sie zu einem Magnet für Codeerosion.
- **switch**-Anweisungen wenn immer möglich vermeiden.
 - Potentielle Verletzung von **SRP**, **OCP** und **DRY**.

Zusammenfassung – 2/3

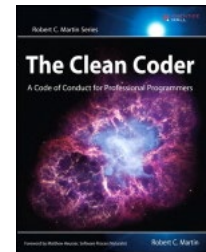
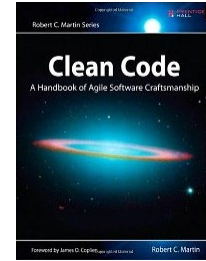
- Anzahl der Funktionsargumente möglichst reduzieren.
 - Bessere Verständlichkeit, weniger Fehler.
- Möglichst auf Flag-Argumente verzichten.
 - Weil sie sind sehr selten selbsterklärend.
- Keine unerwarteten Nebeneffekte einbauen.
 - Separation of Concerns (SoC).
- In objektorientierten Sprachen möglichst auf Output-Argumente verzichten.
- **Allgemein:** Alles Vermeiden was den Nutzer einer Funktion dazu «zwingt» eine Deklaration nachschlagen zu müssen!
 - Kognitive Unterbrechung, stört die Konzentration.

Zusammenfassung – 3/3

- Eine Funktion sollte genau nur eine Aufgabe haben.
- Fehlerhandling ist auch eine Aufgabe!
 - Darum Fehlerhandling in eigene Methoden separieren.
- Möglichst keine Fehlercodes verwenden (besser Exceptions).
- Anweisungen und Abfragen strikte trennen.
- Gute Namensgebung von Methoden.
- Funktion Smells F1 bis F4.
- Perspektive Wechseln: Eine Funktion nicht als AutorIn schreiben, sondern als zukünftige(r) NutzerIn!
 - TDD / Test-First hilft sehr gut diesen Ansatz zu Verfolgen.

Literaturhinweise / Quellen

- Robert C. Martin: **Clean Code**
A Handbook of Agile Software Craftsmanship
Prentice Hall, März 2009
ISBN: 0132350882 / EAN: 9780132350884
- Robert C. Martin: **The Clean Coder**
A Code of Conduct for Professional Programmers
Prentice Hall International, Mai 2011
ISBN: 0137081073 / EAN: 9780137081073
- Joshua Bloch: **Effective Java**
Best practices for the Java Plattform
Pearson Addison-Wesley, Third Edition, Dezember 2017
ISBN: 978-0-13-468599-1



Fragen?