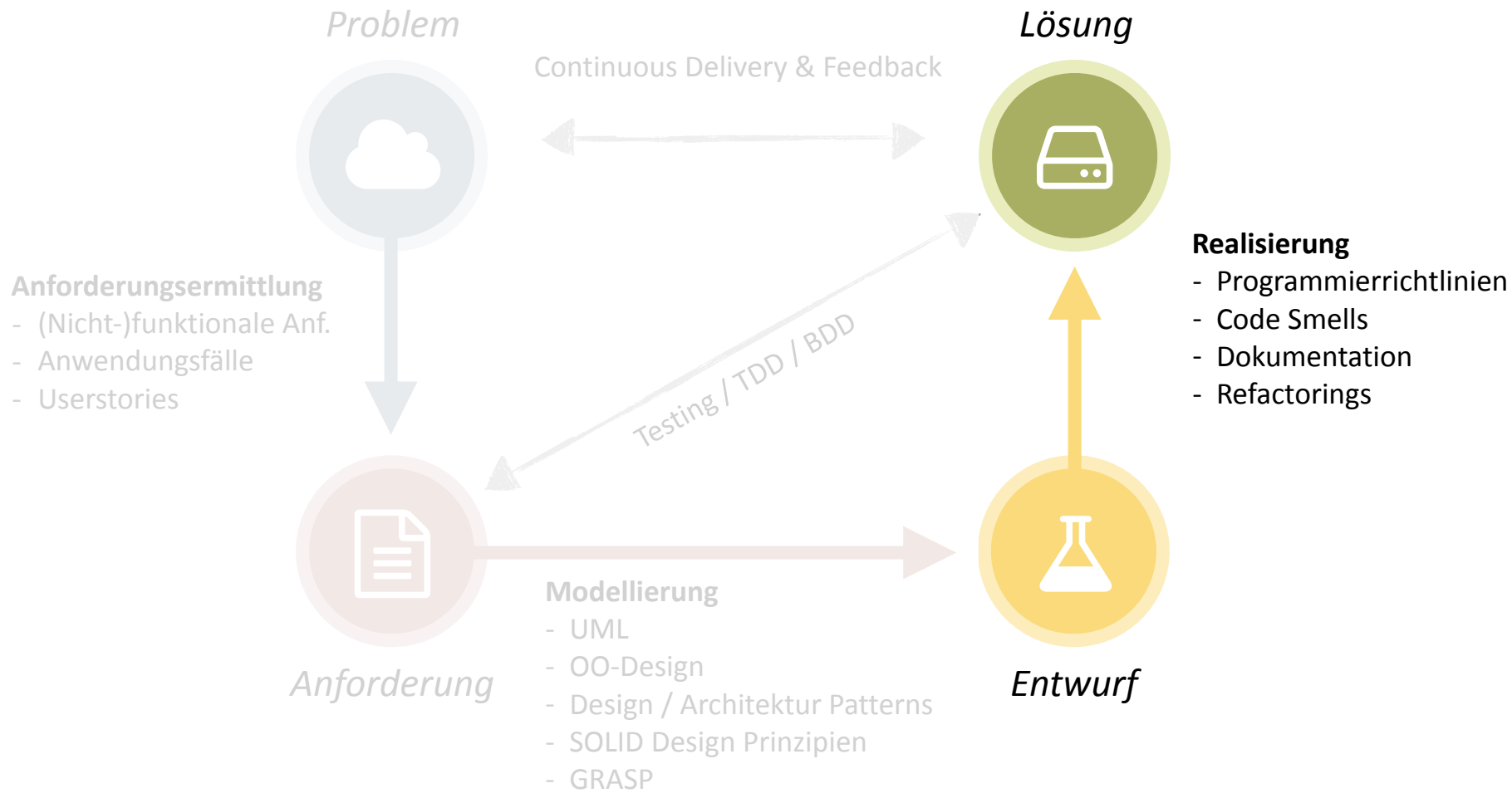
Software Engineering

2. Best Practices - Lesbaren Code schreiben

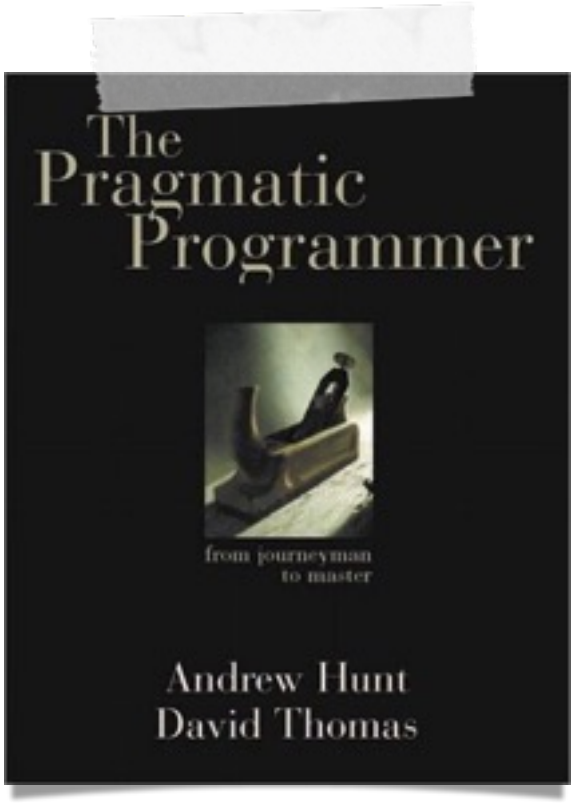
Jonathan Brachthäuser

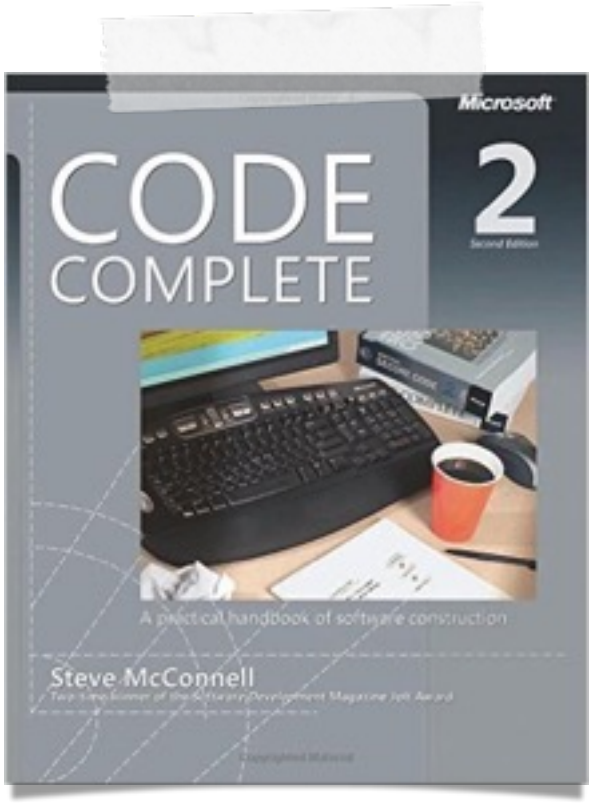
Teilweise basierend auf Folien von Gabriele Täntzer und Klaus Ostermann

Einordnung


Problem
Lösung
Continuous Delivery & Feedback
Anforderungsermittlung
- (Nicht-)funktionale Anf.
- Anwendungsfälle
- Userstories
Testing / TDD / BDD
Realisierung
- Programmierrichtlinien
- Code Smells
- Dokumentation
- Refactorings
Modellierung
- UML
- OO-Design
- Design / Architektur Patterns
- SOLID Design Prinzipien
- GRASP
Anforderung
Entwurf

Literaturhinweis





Warum ist lesbarer Quelltext wichtig?

Programme für Menschen

- Privater und öffentlicher Quelltext
- Team
- Wartung
- Wiederverwendung

Warum

- Der Evolutionsanteil an Softwareentwicklung wird immer höher
  - "80% der Kosten im Lebenszyklus einer Software entfallen auf die Wartung."
  - "Faktor 2 bis 100, je nach Anwendung" (Sommerville, 2001)
  - "1976 – 1998: Wartungskosten mehr als 67%" (Schach, 2003)
- Softwareentwickler im Projekt wechseln und neue Entwickler müssen sich einarbeiten
  - Bevor eine Software neu geschrieben wird sitzen durchschnittlich 10 “Generationen” Wartungsprogrammierer daran. (Parikh, Zvegintzov 1983)
  - Wartungsprogrammierer verbringen durchschnittlich 50% ihrer Zeit damit Quelltext zu verstehen. (Fjeldstad & Hamlen, 1983; Standish, 1984)

Wartungskosten

Year	Proportion of software maintenance costs	Definition	Reference
2000	>90%	Software cost devoted to system maintenance & evolution / total software costs	Erlikh (2000)
1993	75%	Software maintenance / information system budget (in Fortune 1000 companies)	Eastwood (1993)
1990	>90%	Software cost devoted to system maintenance & evolution / total software costs	Moad (1990)
1990	60-70%	Software maintenance / total management information systems (MIS) operating budgets	Huff (1990)
1988	60-70%	Software maintenance / total management information systems (MIS) operating budgets	Port (1988)
1984	65-75%	Effort spent on software maintenance / total available software engineering effort.	McKee (1984)
1981	>50%	Staff time spent on maintenance / total time (in 487 organizations)	Lientz & Swanson (1981)
1979	67%	Maintenance costs / total software costs	Zelkowitz *et al.* (1979)

Jussi Koskinen, http://users.jyu.fi/~koskinen/smcosts.htm

Warum verständlichen Code schreiben?

- Verständlichkeit (Teamwork, Wartung)
- Fehlerrate (nachvollziehen was passiert = Fehler gleich vermeiden)
- Debugging (leichter verstehen = leichter Fehler finden)
- Änderbarkeit (Änderungen verlieren ihren Schrecken)
- Wiederverw. (lesbaren Code kann man leichter wiederverwenden)
- Entwicklungszeit (aus allem hiervor)
- Produktqualität (aus allem hiervor)

Zitat

“Falls du glaubst du brauchst keinen lesbaren Quelltext schreiben, weil ihn eh niemand anderes angucken wird, verwechsle nicht Ursache und Wirkung.”
(Steve McConnell)

Lesbaren Code Schreiben

Developer debugging his own code after a month
Sir Joseph Noel Paton, 1861
Oil on canvas
Frei nach http://classicprogrammerpaintings.com/

Lesbaren Code Schreiben

- Entwicklungsrichtlinien
- Namenswahl
- Magische Zahlen
- Selbstdokumentierender Code

Entwicklungsrichtlinien

- Konstruktive Qualitätssicherungsmaßnahme
- Für alle Phasen des Entwicklungsprozesses nötig
  - Programmtextebene (Namenskonventionen, Code Layout, ...)
  - Auf Ebene der Modellierung (Modellierungskonventionen, Architekturentscheidungen)
  - Gestaltungsrichtlinien (für graphische Oberflächen)
  - Auf Prozessebene (Pull-Request Workflow, ...)
- Werden beeinflusst von:
  - Firmenphiliosophie
  - Entwicklerteam
  - Entwicklungsumgebung (d.h. verwendete Sprachen, Plattformen)
- Können teilweise automatisch erzwungen werden (Linter)

Namenskonventionen

- Konventionen erhöhen Lesbarkeit
- Erlauben direkte Unterscheidung von Klassen, Methoden, Variablen, Konstanten, ...

```
package MeinPackage;

public class calculate {
      final static int zero=0;
      public void POWER(int x, int y) {
         if (y<=zero) return 1;
         return x*this.POWER(x,y-1);
      }
      public void handleincomingmessage() {}
}
```

Beispiel: Java-Namenskonventionen

- Klassennamen:
  - Substantive in "CamelCase"
  - Klassennamen sollten einfach und beschreibend sein
  - Ganze Wörter verwenden, keine Abkürzungen, außer gebräuchliche, dabei erster Buchstabe groß (z.B. Gui, Html, Uml)
- Methodennamen:
  - Verb oder Verbalphrase in "camelCase"
  - Schwache Verben vermeiden (z.B. "handleCalculation")
  - Prädikate beginnen mit "is", z.B. "isEmpty()"

Namenswahl

- Konventionen abhängig vom Ökosystem
- Tipp: Etablierte Bibliotheken als Inspiration

Bedeutung	**Möglicher Name**	**Negativ Beispiel**
Anzahl Studenten pro Vorlesung	numberOfStudents	i,c,spc,students, students1
Aktuelles Datum	currentDate	cd,c,date,current,x
Pausieren und Wiederaufnahme eines Spieles	pause/resume	pause/unpause,stop/ unpause,off/start
Datenbankergebnis	studentList, students	databaseresult, data, input

Namenswahl

- So spezifisch wie möglich
- Problemorientiert, statt Lösungsorientiert
  - "Was" statt "Wie"
  - z.B. `employee` statt `inputRecord`, `printerReady` statt `bitFlag`
- Namenskonventionen können helfen
  - Aus dem Ökosystem, sowie Projektspezifische
  - Sollten dokumentiert sein
- Mehrdeutigkeiten vermeiden
- Paare von Methoden sollten sprachlich passen
  - on/off, pause/resume, undo/redo
- Namenslänge proportional zum Scope
- Methodennamen sollten die vollständige Tätigkeit abdecken
  - Wenn mehrere Tätigkeiten, dann Indiz für Refactoring

Magische Zahlen

```
for (Event event : events) {
  if (event.startTime > now &&
      event.startTime < now + 86400) {
    event.print();
  }
}
```

```
... status == 1 ...
```

```
if (key == 'D') key = 'A';
```

```
for (int i=0; i < min(20, data.length); i++)
```

Entzauberte Zahlen

```
for (Event event : events) {
  if (event.startTime > now &&
      event.startTime < now + SECS_PER_DAY) {
    event.print();
  }
}
```

```
... status == OPEN ...
```

```
if (key == DELETE) key = APPROVE;
```

```
for (int i=0; i < min(MAX_ROWS, data.length); i++)
```

Selbstdokumentierender Code

- Guter Quellcode benötigt gar keine oder nur wenige Kommentare
  - Zumindest innerhalb von Methoden/Funktionen
  - Bei der Dokumentation von APIs kann jedoch eine informelle Spezifikation sinnvoll sein.
- Kommentare können sogar schaden
- Beste Quellcode-Dokumentation durch
  - gute Variablen- und Methodennamen
  - geringe Komplexität

Selbstdokumentierender Code vs. Kommentare

```
/* if operation flag is 1 */
if (opFlag == 1) ...
```

```
/* if operation is "delete all" */
if (opFlag == 1) ...
```

```
/* if operation is "delete all" */
if (operationFlag == DELETE_ALL) ...
```

```
if (operationFlag == DELETE_ALL) ...
```

Selbstdokumentierender Code (Beispiel 2)

```
for (Event event : events) {
  if (event.startTime > now &&
      event.startTime < now + SECS_PER_DAY) {
    event.print();
  }
}
```

Selbstdokumentierender Code (Beispiel 2)

```
boolean isInFuture(Event event) {
  return event.startTime > now;
}
boolean isWithinOneDay(Event event) {
  return event.startTime < now + SECS_PER_DAY;
}


for (Event event : events) {
  if (isInFuture(event) && isWithinOneDay(event)){
    event.print();
  }
}
```

Selbstdokumentierender Code (Beispiel 2)

```
boolean isInFuture(Event event) {
  return event.startTime > now;
}
boolean isWithinOneDay(Event event) {
  return event.startTime < now + SECS_PER_DAY;
}

events.filter(e -> isInFuture(e))
      .filter(e -> isWithinOneDay(e))
      .forEach(e -> print(e));
```

Noch mehr sprechende Abstraktion mit Java-8 Streams.

Was denken Sie über die Wahl des Variablennamens "e"?

Selbstdokumentierender Code (Beispiel 3)

```
void Update () {
  if (!paused) {
    Time.timeScale = Time.timeScale * timeModificator;
    if (Time.timeScale > 1 ||
        Time.timeScale < minTimescale) {
      timeModificator = 1;
    }
  }
}
```

`timeModificator` liegt entweder zwischen 0 und 1 (verlangsamen), oder > 1 (beschleunigen)

Selbstdokumentierender Code (Beispiel 3)

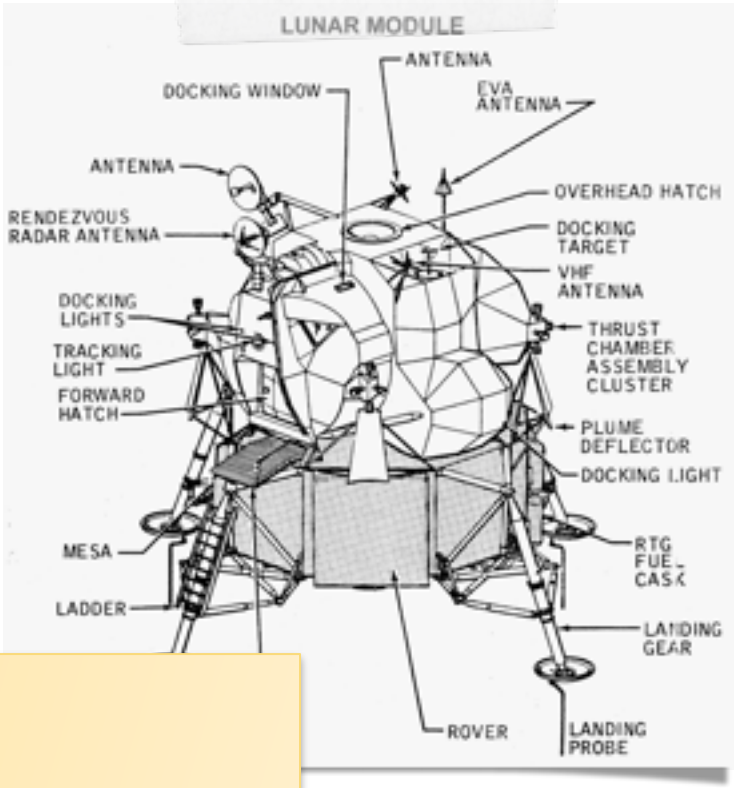
```
void Update () {
  if (paused) return;

  if (tooFast() || tooSlow()) {
    stopAcceleration()
  }
}
```

Dokumentation

Developers look for documentation in legacy system
Jean-François Millet, 1857
Oil on canvas
http://classicprogrammerpaintings.com/

Exkurs: Kommentare im Apollo 11 Lander




```
# Page 801
   CAF   TWO          # WCHPHASE = 2
   TS    WCHPHOLD
   TS    WCHPHASE
   TC    BANKCALL     # TEMPORARY, I HOPE HOPE HOPE
   CADR  STOPRATE     # TEMPORARY, I HOPE HOPE HOPE
```

Quelle: https://github.com/chrislgarry/Apollo-11/blob/master/Luminary099/LUNAR_LANDING_GUIDANCE_EQUATIONS.agc

Die 7 Regeln für gute Dokumentation

- (1) Schreibe aus der Sicht des Lesers
- (2) Vermeide unnötige Wiederholungen
- (3) Vermeide Mehrdeutigkeiten
- (4) Verwende eine Standardstrukturierung
- (5) Halte Begründungen für Entscheidungen fest
- (6) Halte Dokumentation aktuell, aber nicht zu aktuell
- (7) Überprüfe Dokumentation auf ihre Gebrauchstauglichkeit

Quelle: http://www.embarc.de/die-sieben-regeln-fuer-gute-dokumentation-in-stein-gemeisselt-tafel-1-von-3
nach Documenting Software Architectures: Views and Beyond – Clements et al. (2002)

(1) Schreibe aus der Sicht des Lesers

- Wer sind die Leser (Zielgruppenanalyse)?
- Was interessiert die Leser?
  - z.B. Nutzer einer API
    - Ersten Schritte (Installation, Konfiguration, notwendige Abhängigkeiten etc.)
    - Grundlegende Benutzung (Häufigste Nutzung anhand von Beispielen)
    - Wichtige Domänenkonzepte und Zusammenhänge, sowie Übersetzung in Source Code
    - API Dokumentation (Nachschlagewerk, z.B. was sind die Randbedingungen für jede einzelne Klasse / Methode)

(1) Schreibe aus der Sicht des Lesers

- Wer sind die Leser (Zielgruppenanalyse)?
- Was interessiert die Leser?
  - z.B. Entwickler, verantwortlich für Wartung
    - Ersten Schritte (Installation, Konfiguration, notwendige Abhängigkeiten etc.)
    - Hinweise zum Debugging (Komandozeilenbefehle, Konfigurationen, Beispielcode)
    - Getroffene Designentscheidungen (z.B. Abweichungen vom Standard, in Kauf genommene Nachteile)
    - API Dokumentation

(2) Vermeide unnötige Wiederholungen

- Spart dem Leser Zeit und unnötigen Wartungsaufwand
- Über Wiederholungen (textuell) Abstrahieren (DRY)
- Wiederholungen an einer Stelle ausreichend erklären, dann nur noch darauf verweisen
  - Die meisten Dokumentationsgeneratoren unterstützen Verweise
  - z.B. Wenn es Standards im Projekt gibt, diese an zentraler Stelle erläutern und nur Abweichungen vom Standard dokumentieren
  - Macros erlauben Textausschnitte an verschiedenen Stellen wiederzuverwenden, ohne diese zu kopieren

(3) Vermeide Mehrdeutigkeiten

- Kategorisieren des Wortschatzes in drei Teilbereiche
  - Allgemeiner Wortschatz
  - Domänenspezifischer Wortschatz
  - Technischer Wortschatz
- Zuweisung in einen Teilbereich sollte stets eindeutig sein
- Für die gleiche Bedeutung sollte immer das gleiche Wort gewählt werden (insb. für Domäne und Technik)
- Standardbegriffe vorziehen
- Leser hat i.d.R. keine Zeit: subtile Unterschiede vermeiden
- Möglichkeit zum Nachlesen bieten (z.B. Glossar, Verweise)

(4) Verwende eine Standardstrukturierung

- Standards helfen dem Lesenden Zeit zu sparen
- Standards machen dem Autor evtl. notwendige Details bewusst
- Gilt auf verschiedenen Ebenen
  - Externe Architektur Dokumentation (z.B. arc42)
  - API Dokumentation (z.B. Javadoc)
  - Sogar auf Satzbauebene ("Instances of <CLASS> represent ...")

```
 * Returns a list iterator over the elements in this
 * list (in proper sequence).
 *
 * <p>The returned list iterator is
 * <a href="#fail-fast"><i>fail-fast</i></a>.</p>
 *
 * @see #listIterator(int)
 */
public ListIterator<E> listIterator() {...}
```

Kurzbeschreibung (ca. 1 Satz)

Optionale längere Beschreibung

Verweise

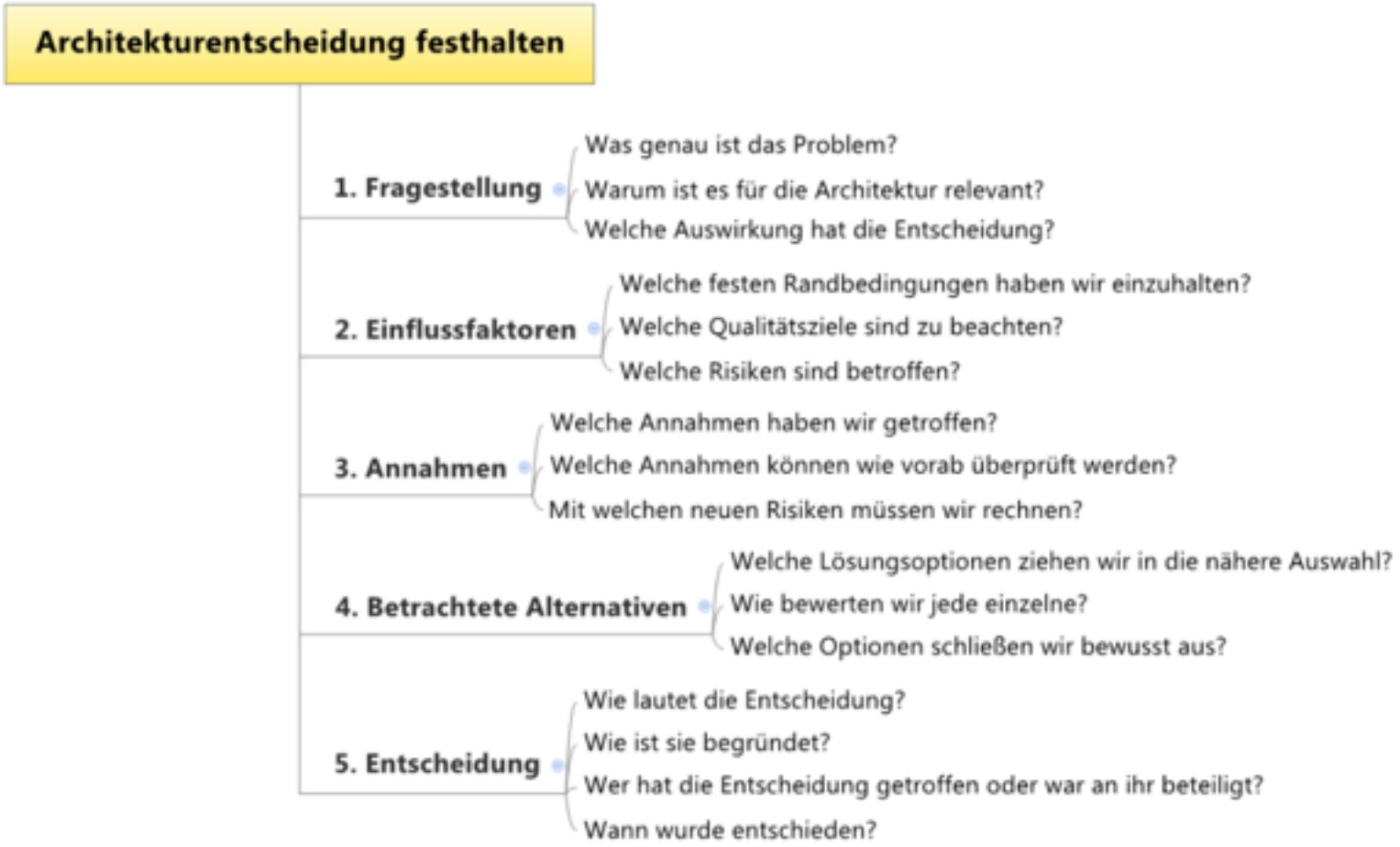
Exkurs: Kommentare für Methoden

- Wenn sinnvoll (siehe auch selbstdokumentierender Code)
- 1 bis 2 Sätze
  - Intention der Methode beschreiben
  - "Was macht die Methode", nicht "Wie macht sie es"
  - lieber Methode teilen, wenn mehrere Verantwortlichenkeiten
- Grenzen der Methoden dokumentieren
  - Vor- und Nachbedingungen
  - z.B. nur positive Eingabewerte

(5) Halte Begründungen für Entscheidungen fest

- Source Code Intern (z.B. API Dokumentation)
- Source Code Extern
  - Architekturdokumentation (z.B. Confluence, github Wiki)
  - Diskussionen im Issuetracker (z.B. JIRA , github Issues)

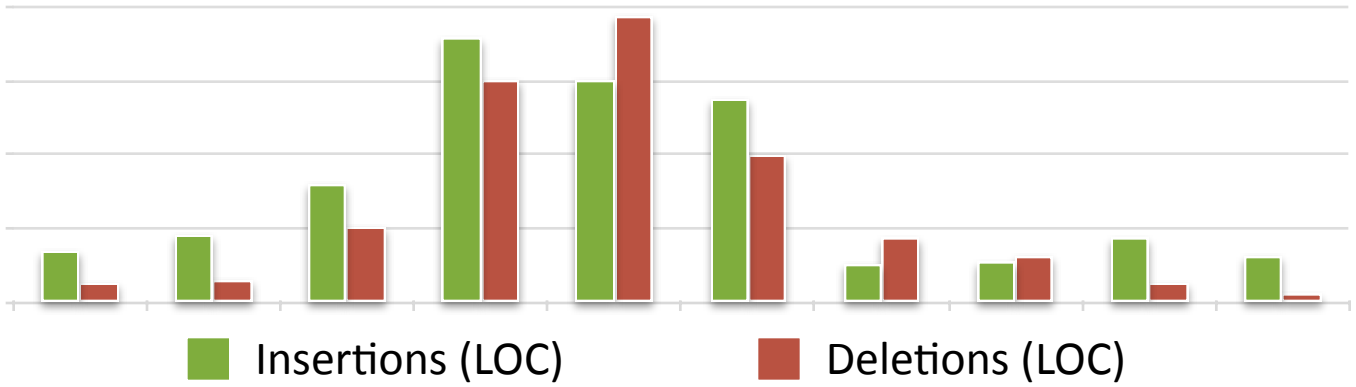
(5) Halte Begründungen für Entscheidungen fest



Quelle: Softwarearchitekturen dokumentieren und kommunizieren – Stefan Zörner (2012)

(6) Halte Dokumentation aktuell, aber nicht zu aktuell

- Dokumentation veraltet sehr schnell
  - Designentscheidungen werden revidiert
  - Refactorings werden durchgeführt
  - Neue Anwendungsfälle werden implementiert
  - Fehler werden behoben und der Code wird verändert
- Unterschiedliche Entwicklungsphasen -> Unterschiedlicher Dokumentationsanspruch



(6) Halte Dokumentation aktuell, aber nicht zu aktuell

- Keine Ausrede, um gar nicht zu dokumentieren
  - Eher Argument für knappe (DRY) und treffende Dokumentation
  - Besser abstrahieren: Details die schnell veralten nicht erwähnen, eher Struktur beschreiben
  - Durchgehend wichtig: Selbstdokumentierender Code

Dokumentation aktuell halten

- Jede Änderung am Code kann Dokumentation invalidieren
- Deshalb Aktualisierungen so einfach wie möglich machen:
  - Dokumentation so nah wie möglich am Objekt der Beschreibung halten (z.B. im Quelltext)
  - Rückverweise im Quelltext zu externer Dokumentation (z.B. "Diese Klasse kollaboriert mit Klasse B, wie in Dokument X beschrieben.")
- Dokumentation automatisch generieren
  - Aus Kommentaren im Source Code (z.B. Javadoc, ScalaDoc, ...)
  - Aus struktierten Textdateien (Literate Programming, readthedocs.io)
  - Erzeugen und Veröffentlichen der Dokumentation am besten vollautomatisch (bei jedem Commit / PR).

Dokumentation automatisch generieren

Dokumentationsgeneratoren

- Dokumentation aus Sourcecode extrahieren
  - Source Code und Dokumentation (Schnittstellendokumentation) zusammen in einer Datei
- Codestruktur beeinflusst Text
- Idee: Source Code Artefakte wie Klassen und Methoden werden mit besonderen Kommentaren dokumentiert
  - Spezielle Syntax erlaubt automatisches Generieren
- z.B. Javadoc, ScalaDoc, RDoc, ...

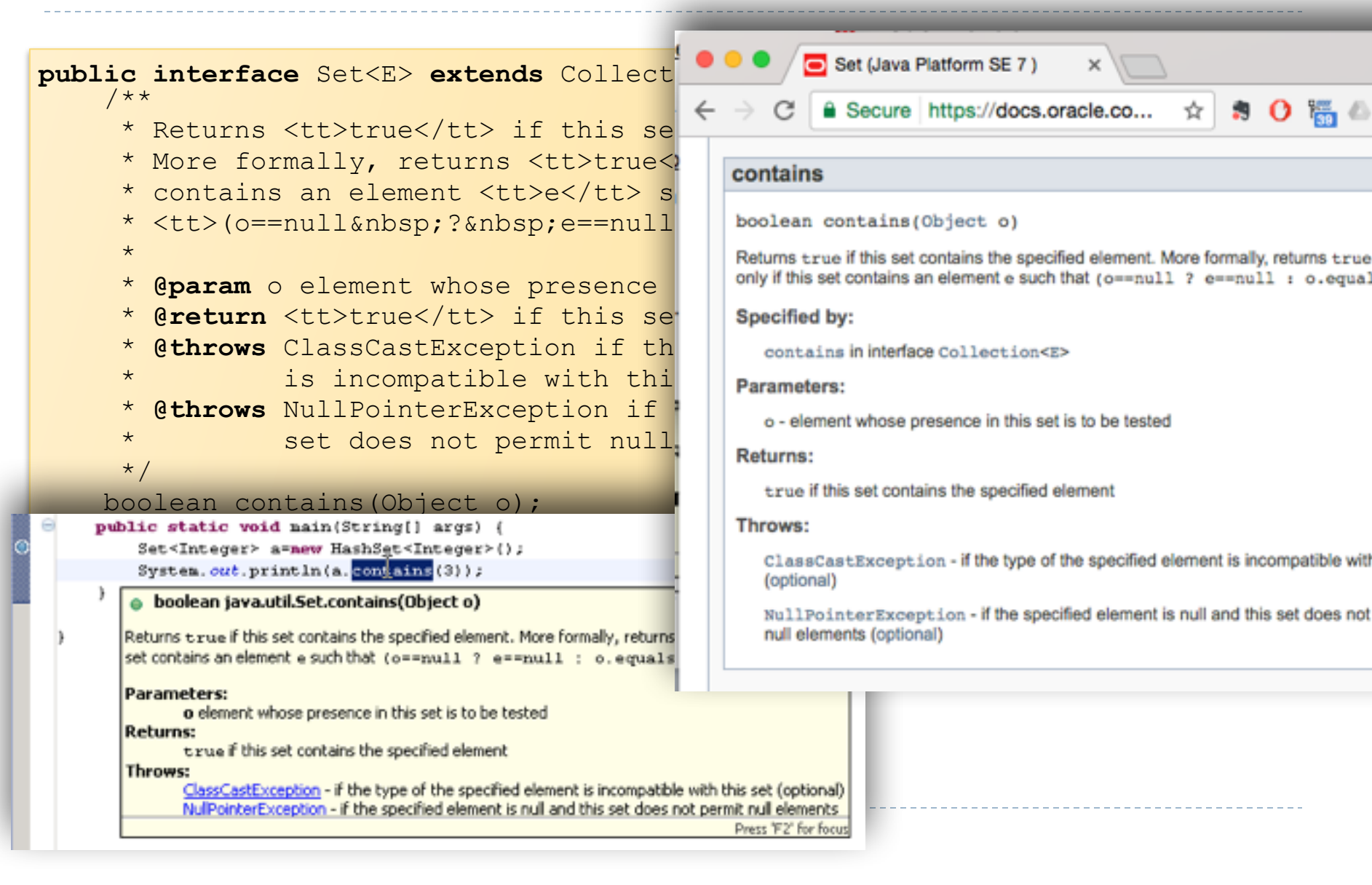
Literate Programming

- Source Code aus Dokumentation extrahieren
  - Source Code und Dokumentation (Endbenutzerdoku) zusammen in einer Datei
- Textstruktur beeinflusst Code
- Idee: Quelltext der Dokumentation anpassen
  - Primär für den menschlichen Leser schreiben
  - Sourcecode ist Sekundärartefakt
- z.B. Rocco, Docco, lhs2tex, tut, ...

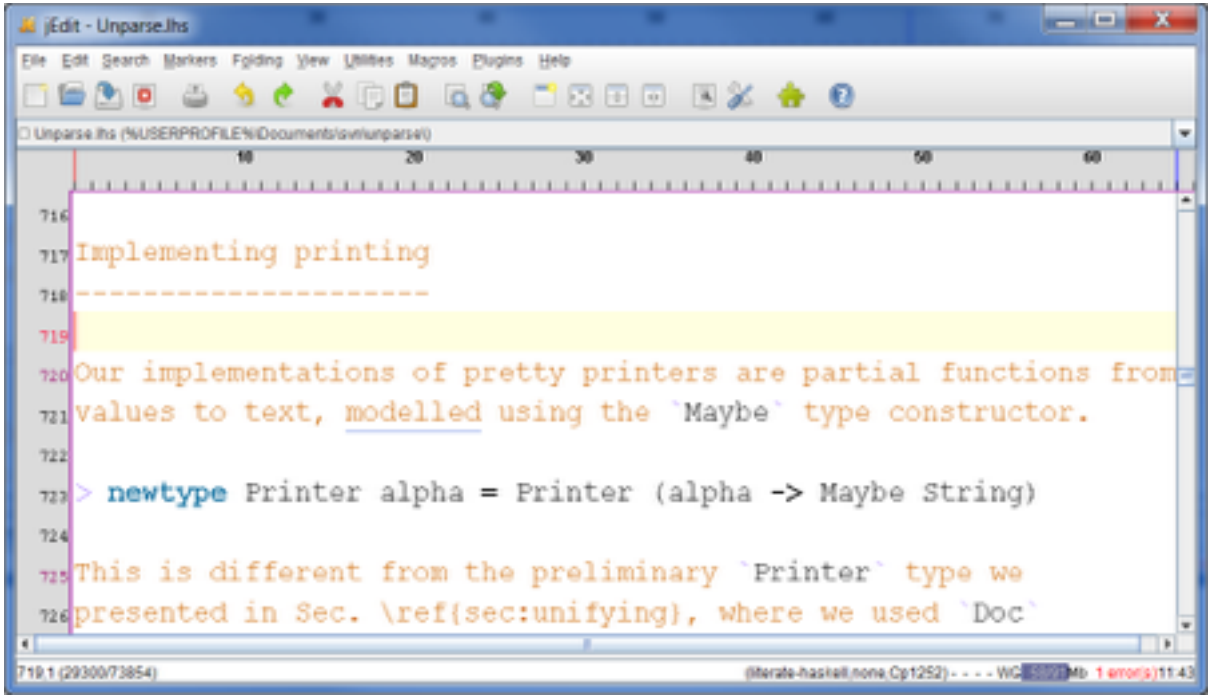
API-Dokumentation (Beispiel Javadoc)

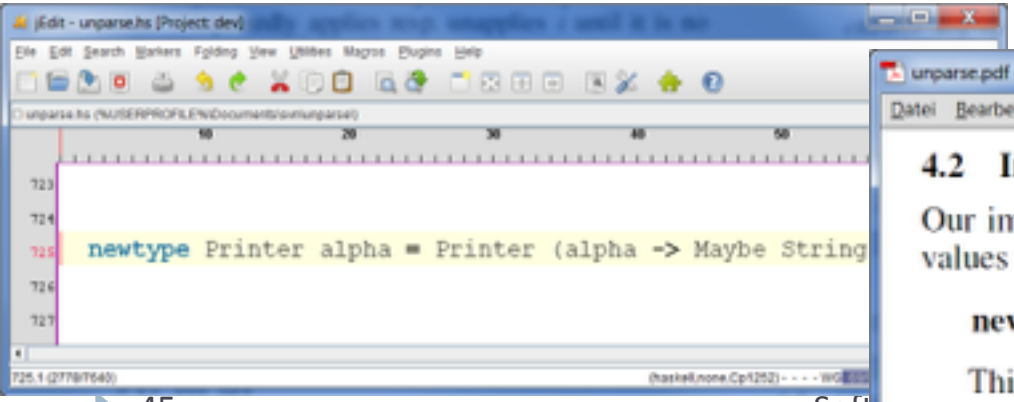
```
public interface Set<E> extends Collection<E> {
    /**
     * Returns <tt>true</tt> if this set contains the specified element.
     * More formally, returns <tt>true</tt> if and only if this set
     * contains an element <tt>e</tt> such that
     * <tt>(o==null ? e==null : o.equals(e))</tt>.
     *
     * @param o element whose presence in this set is to be tested
     * @return <tt>true</tt> if this set contains the specified element
     * @throws ClassCastException if the type of the specified element
     *         is incompatible with this set (optional)
     * @throws NullPointerException if the specified element is null and this
     *         set does not permit null elements (optional)
     */
    boolean contains(Object o);
…
```

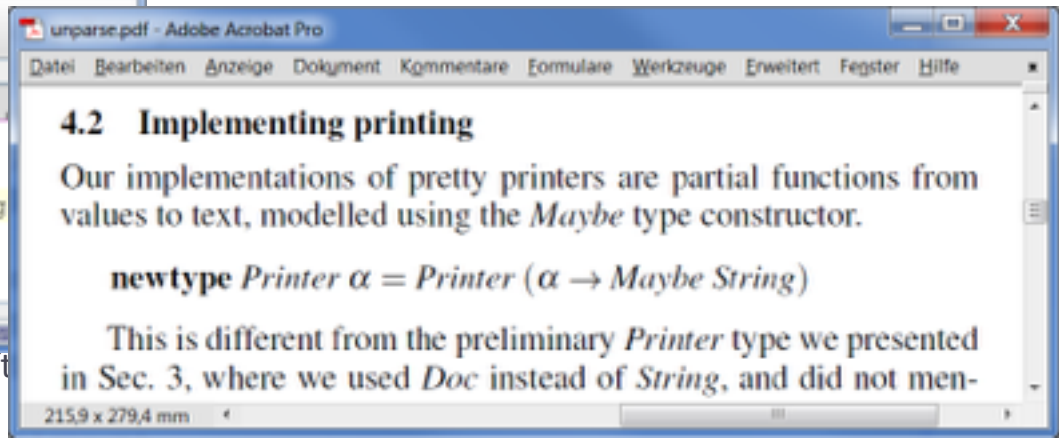
API-Dokumentation (Beispiel Javadoc)



Literate Programming (Haskell Beispiel)







Dokumentation aktuell halten

- Dokumentation automatisch überprüfen
  - Code Beispiele in Dokumentation sollten automatisch kompiliert und ausgeführt werden
  - Änderungen im Code -> ggf. Fehler beim Erzeugen der Doku.
  - z.B. Scala: tut, Haskell: doctest

````
Here is how you add numbers:
```tut
1 + 1
```
````

Eingabe (in Markdown)

````
Here is how you add numbers:
```scala
scala> 1 + 1
res0: Int = 2
```
````

Generierte Ausgabe (in Markdown)

https://github.com/tpolecat/tut

(7) Überprüfe Dokumentation auf ihre Gebrauchstauglichkeit

- Nur die Leserzielgruppe kann über Tauglichkeit entscheiden
- Reviews mit Vertretern aus der Zielgruppe
- Feedback direkt von Lesern erheben, sammeln und auswerten
- Lesern die Möglichkeit bieten, direkt Änderungen vorzuschlagen oder durchzuführen

Zusammenfassung: Dokumentation

- Ziel von Dokumentation: Leser können **effizient** und **effektiv** Informationen extrahieren
- Effizient: Wiederholungen vermeiden, Verwirrung vermeiden
- Effektiv: Notwendige Information ist vorhanden und aktuell

Code Smells & Refactoring

Programmers at work performing a major, unpaid refactoring
Eero Järnefelt, 1893
Oil on canvas
Frei nach http://classicprogrammerpaintings.com/

Was sind Bad Code Smells?

- Kondensierung von Erfahrungswissen
- verdächtige Code-Stellen
- Anhaltspunkte für mögliche Schwachstellen / Verbesserungspotenzial
- sollten evtl. durch Refactoring behoben werden
- Häufig rein syntaktisch oder basierend auf Code-Metriken (z.B. LoC, Anzahl von abhängigen Klassen etc.)
  - automatisiert erkennbar durch statische Analyse

Erweiterte Liste an Code Smells mit automatischer Erkennung durch statische Analyse:
http://findbugs.sourceforge.net/bugDescriptions.html

Was ist ein Refactoring?

*“Refactoring ist der Prozess, ein Softwaresystem so zu verändern, dass das **externe Verhalten unverändert** bleibt, der Code aber eine **bessere Struktur** erhält.”*
(Martin Fowler)

Was ist ein Refactoring?

- "... dass das **externe Verhalten** unverändert bleibt ..."
- Abhängig von der Grenze zwischen intern/extern
- Grenzen häufig zwischen verschiedenen Rollen:
  - Endnutzer(in)/Programmierer(in)
  - API-Nutzer(in)/API-Implementierer(in)
  - Modul-Tester(in)/Modul-Implementierer(in)
- Interfaces (z.B. in Java) ermöglichen Repräsentation der Grenze im Code

Was ist ein Refactoring?

- “... dass das externe Verhalten **unverändert** bleibt ...”
- Extern nicht *sichtbar* -> kann geändert werden
- "Sichtbar" kann heißen
  - durch manuelles Beobachten in der entsprechenden Rolle
  - durch automatisches Testen (kann auch Zeit-/Speicher/IO-Verhalten einschließen)
- Sicherstellen durch Tests (automatisiert!)
- Bedeutung sollte sich nicht ändern, analog zu algebraischen Umformungen

Warum Refactoring?

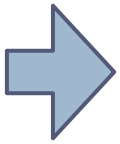
- Beheben von Code Smells
- Lesbarkeit / Übersichtlichkeit / Verständlichkeit
  - Reduktion vom Komplexität, z.B. Aufteilen von Methoden
  - Bewusste Benennung von Variablen, Methoden, …
- Wiederverwendung / Entfernen von Redundanz
  - z.B. Methoden aufteilen um Teile wiederzuverwenden
  - Kopierte Quelltextfragmente in eine Methode extrahieren
- Erweiterbarkeit und Testen
  - später mehr dazu

Literaturhinweis (inkl. Liste an Smells und Refactorings): Martin Fowler, "Refactoring" (2005)

Refactoring Beispiel

```
class Person {
  String name;
  String street;
  String houseNumber;
  String zipCode;
  String city;
  ...
}
```

```
class Company {
  String name;
  String street;
  String houseNumber;
  String zipCode;
  String city;
  ...
}
```



```
class Person {
  Address address;
  ...
}
```

```
class Address {
  String name;
  String street;
  String houseNumber;
  String zipCode;
  String city;
  ...
}
```

```
class Company {
  Address address;
  ...
}
```

Literaturhinweis zu Datenmodellierung:
http://spaceninja.com/2015/12/07/falsehoods-programmers-believe

Code Smell vs. Refactoring

Code Smell

- Schwächen im Quelltext
- Aus Erfahrung festgehalten
- Einteilung in:
  - Klasseninterne Smells
  - Klassenübergreifende Smells

Refactoring

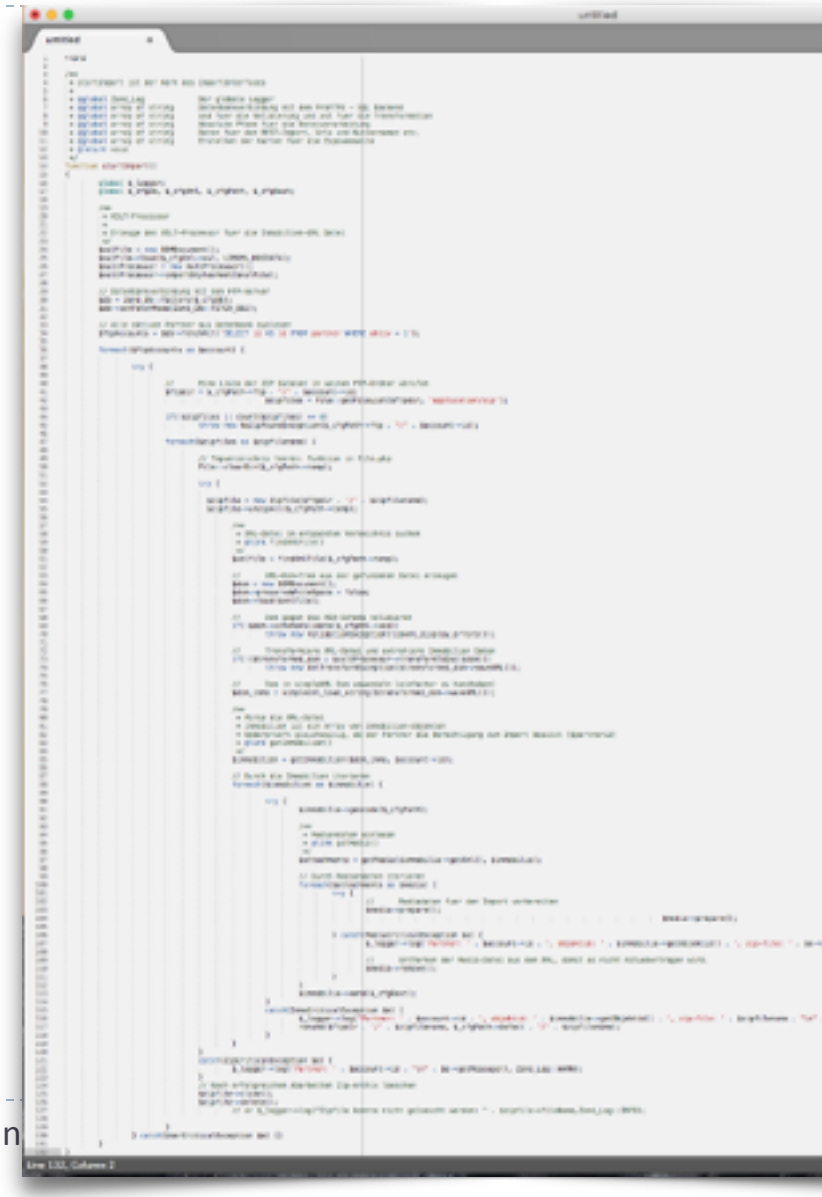
- Rezepte zur Verbesserung
- Manuelles Vorgehen und Automatisierung möglich
- Besteht aus:
  - Name
  - Motivation
  - Vorgehen
  - Beispiele

Klasseninterne Bad Smells

- Beispiele:
  - Lange Methode
  - Doppelter Code
  - Lange Parameterliste
  - Kommentare
  - Temporäre Felder
  - Switch-Befehle
  - ...

Beispiel 1: Lange Methode

- Vermeiden von "Spaghetti Code"
- Extrahieren statt Kommentieren
- Ideale Methodenlänge ist abhängig von:
  - Ökosystem
  - Informationsdichte
- üblicherweise ca. 1-50 Zeilen
- Refactoring: **Methode Extrahieren**



Extract Method – Vorgehen

1. Neue Methode anlegen – sinnvollen Namen vergeben
2. Zu extrahierenden Code in die neue Methode kopieren
3. Zugriffe auf lokale Variablen suchen -> als Parameter übergeben
4. Temporäre Variablen nur in Fragment benutzt -> in neuer Methode anlegen
5. Werden lokale Variablen verändert? -> Rückgabewert der neuen Methode
6. Originalquelltext mit Methodenaufruf ersetzen

Extract Method – Bedingungen (Auszug)

- Extrahierter Code muss ein oder mehrere komplette Statements sein
- Maximal auf eine lokale Variable (die später benutzt wird) wird schreibend zugegriffen
- Bedingtes return-Statement verboten
- Break und Continue verboten, wenn das Ziel außerhalb des zu extrahierenden Code liegt

Beispiel 2: Doppelter Code

- sehr häufiger Code Smell
- Grund häufig: Copy-and-Paste
  - ermöglicht schnelle Wiederverwendung
  - führt evtl. Fehler ein, wenn nicht an Kontext angepasst
  - erhöhter Wartungsaufwand
- Mögliche Refactorings
  - Klasse extrahieren
  - Methode extrahieren
  - Methode nach oben verschieben
  - Template Methode bilden

Beispiel 2: Doppelter Code

```
void valuesChanged() {
  speedModifier = model.speedModifier;
  if (lastSpeedModifier != speedModifier) {
    model.component.speedModifier = speedModifier;
    lastSpeedModifier = speedModifier;
  }

  distance = model.distance;
  if (lastDistance != distance) {
    model.component.distance = distance;
    lastDistance = distance;
  }

  showAnimations = model.showAnimations;
  if (lastShowAnimations != showAnimations) {
    model.component.showAnimations = showAnimations;
    lastShowAnimations = showAnimations;
  }
}
```

Beispiel 2: Doppelter Code

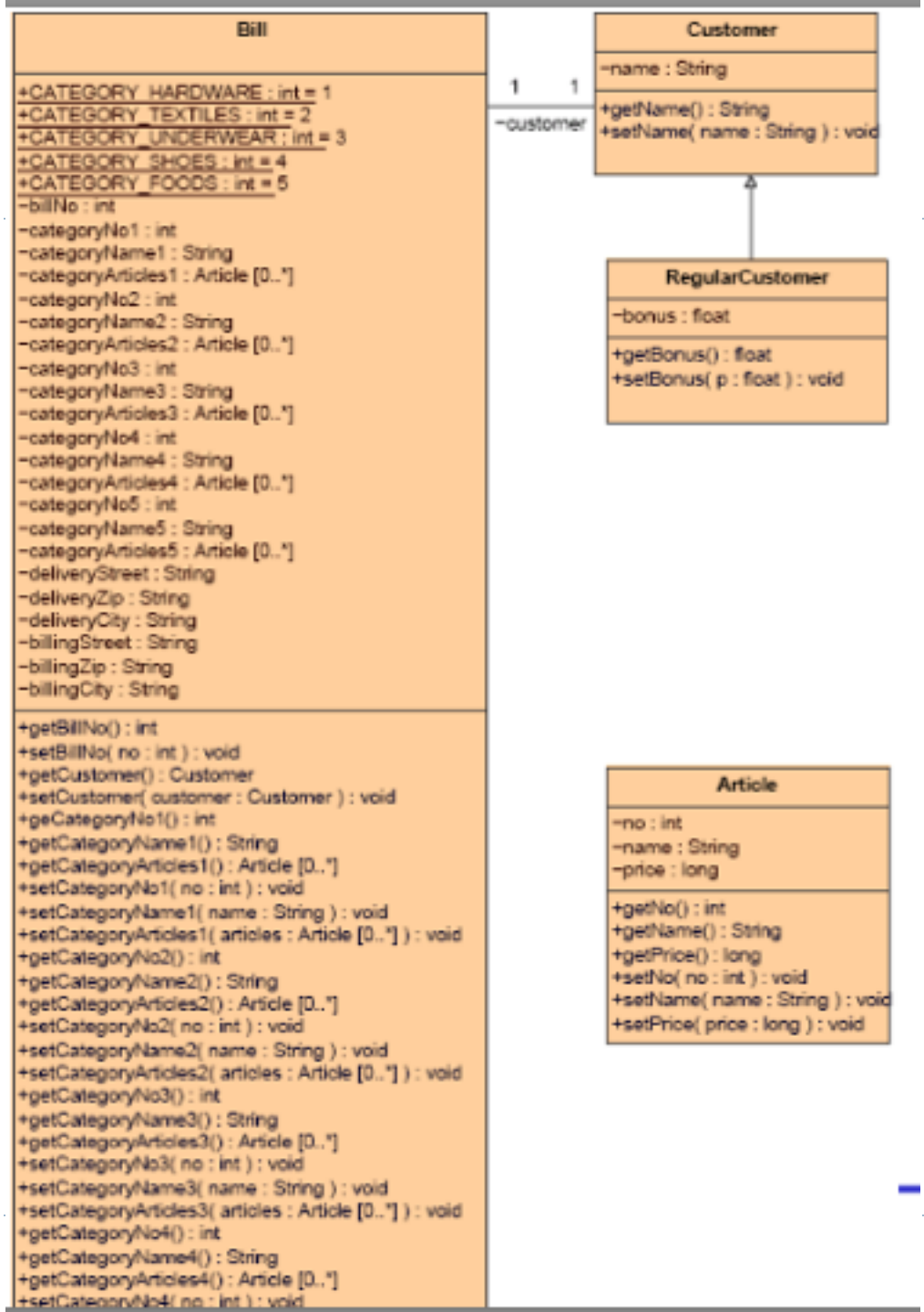
```
abstract class Cache<T> {
  T lastValue;
  void update(T newValue) {
    if (lastValue != newValue) {
      onChange(lastValue, newValue);
      lastValue = newValue;
    }
  }
  abstract void onChange(T oldValue, T newValue);
}
```

Klassenübergreifende Bad Smells

- Beispiele:
  - Große Klassen
  - (Methoden-)Neid
  - Nachrichtenketten
  - Verweigertes Erbe
  - Datenhaufen
  - Faule Klasse
  - ...

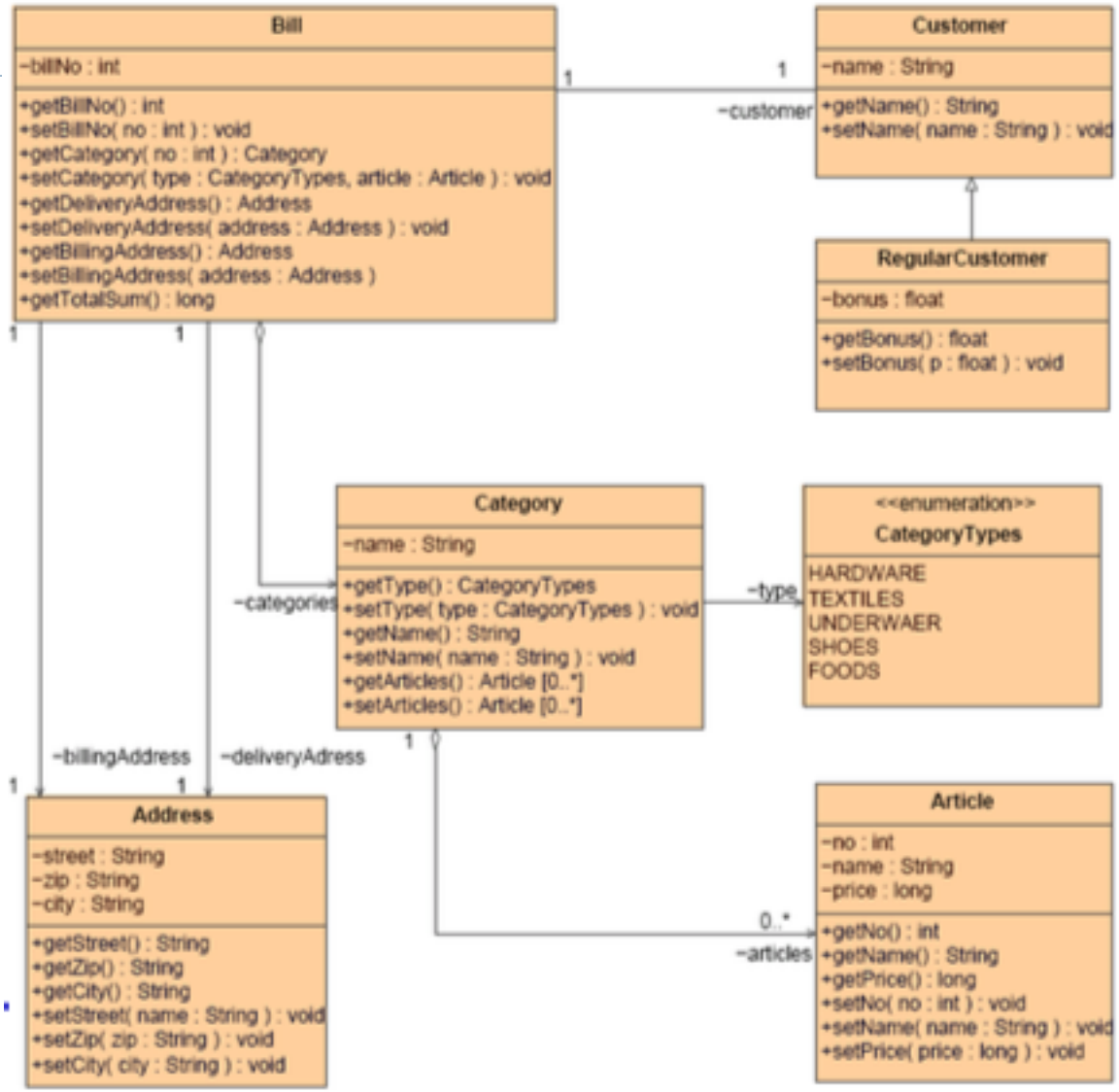
Beispiel 1: Große Klassen

vorher



Beispiel 1: Große Klassen

nachher

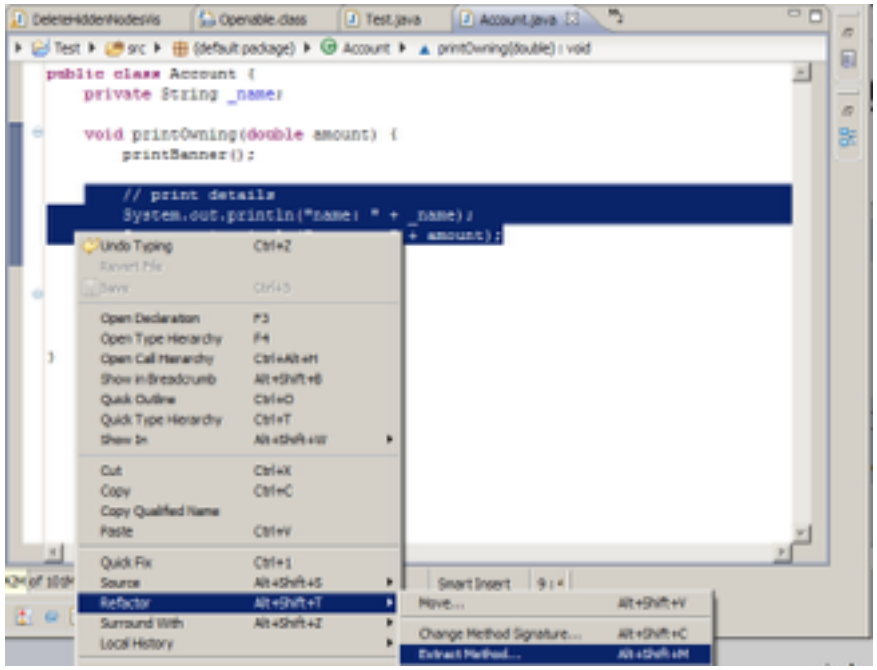


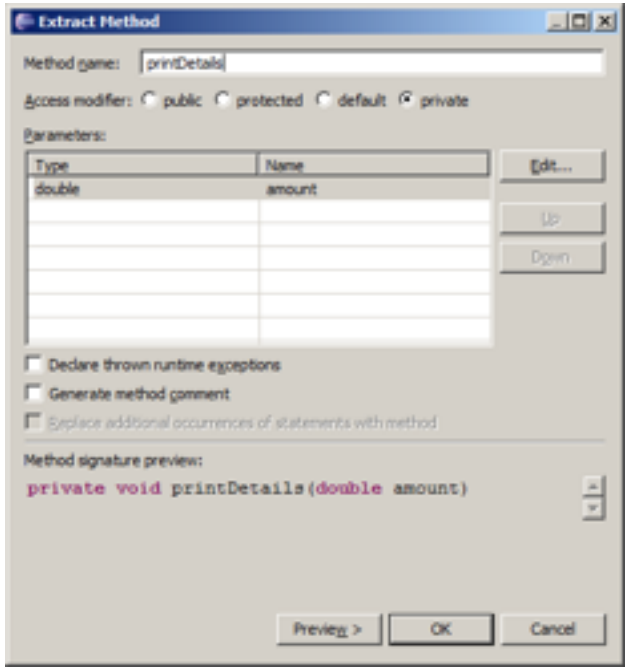
Beispiel 2: (Methoden/Daten-)Neid

- Klassen enkapsulieren Daten und Verhalten
- Verdächtig: eine Methode, die auf mehr auf Daten / Methoden einer anderen Klasse zugreift, als auf die eigenen
- Mögliche Refactorings
  - Methode verschieben
  - Methode aufteilen
  - Feld verschieben

Automatisierte Refactorings

- Viele Entwicklungsumgebungen automatisieren Refactorings
- Ursprung in Smalltalk und IntelliJ





Refactorings Allgemein

- Refactoring als allgemeines Konzept
- Auch große Refactorings in ganzen Klassenhierarchien
- Für viele Sprachen und Modelle, auch sprachübergreifend
- Änderung von Software*design*

- Fundamental für einige Softwaretechnikansätze
  - Schnell erste Quelltextversion schreiben, später umstrukturieren
  - Software wächst, Umstrukturieren wird fast immer nötig sein
  - Nur möglich mit geeigneten Interfaces

Coding Practices

Crowd of maintainers sentencing developers for not following good coding practices
Grigoriy Myasoyedov, 1897
Oil on canvas
Frei nach http://classicprogrammerpaintings.com/

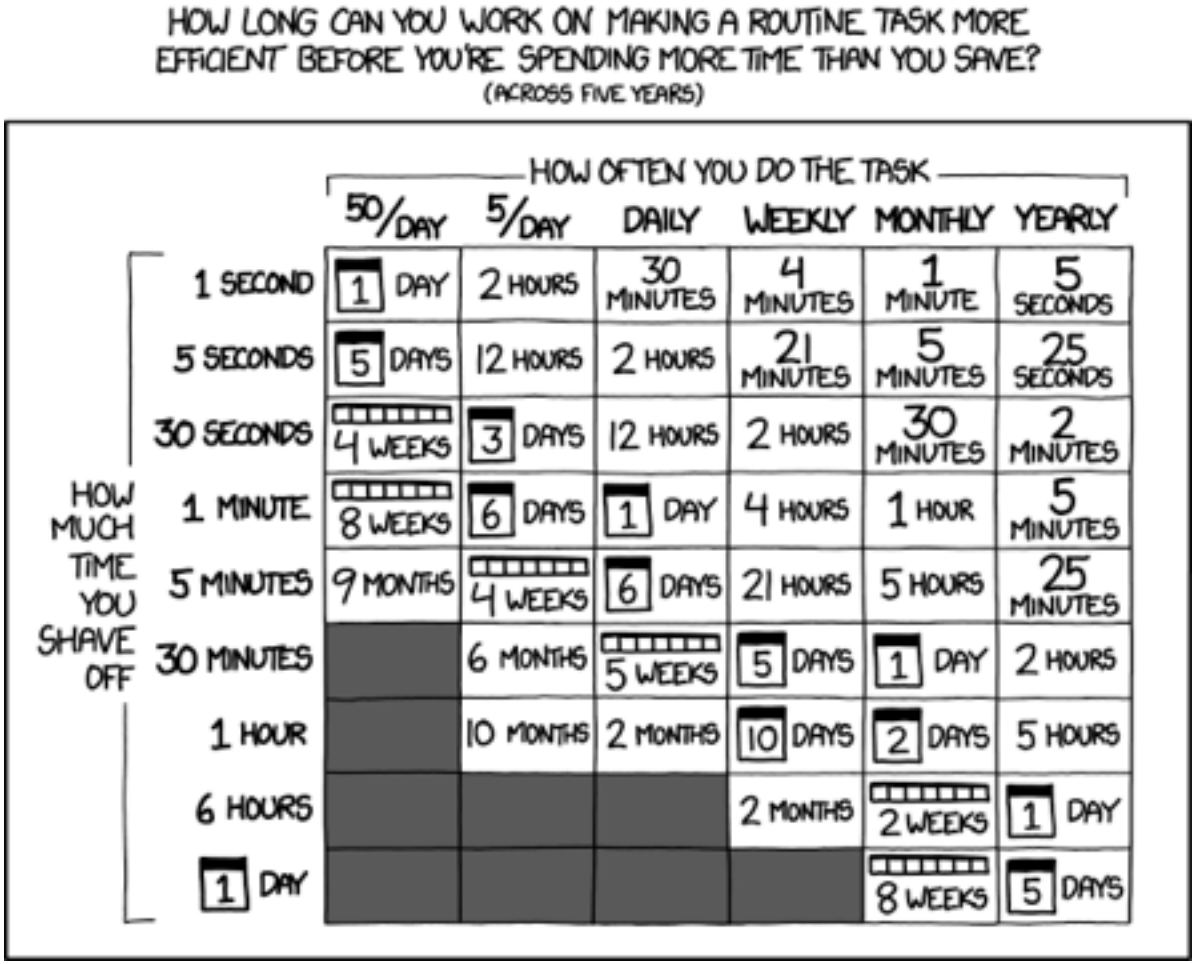
DRY - Don't Repeat Yourself

- Als ProgrammiererIn stehen uns zwei wesentliche Werkzeuge zur Verfügung
  - Abstraktion
  - Automation

- DRY kann man entsprechend auf zwei Weisen interpr.
  - Wiederhole Dich nicht in Programmcode
  - Wiederhole Dich nicht in einer Tätigkeit

- **Wiederholung** ist in beiden Fällen ein Bad Smell

DRY - Don't Repeat Yourself



Quelle: https://xkcd.com/1205/

KISS – Keep it simple stupid (Premature Abstraction)

- Wann ist welche Abstraktion angemessen?
- Abhängig vom Projektumfeld & Anforderungen
- Nicht: Persönlicher Geschmack!

```
fac n = if n == 0
           then 1
           else n * fac (n-1)
```

```
refold c n p f g = fold c n . unfold p f g
fac = refold (*) 1 (==0) id pred
```

```
fac n = product [1..n]
```

Quelle: https://www.willamette.edu/~fruehr/haskell/evolution.html

Premature Optimization

"We should forget about small efficiencies, say about 97% of the time: ***premature optimization is the root of all evil****. Yet we should not pass up our opportunities in that critical 3%."*
(Donald Knuth)

Premature Optimization

- Häufig sind low-level Optimierungen zunächst zu vernachlässigen, erhöhen aber den Wartungsaufwand
- Eine genaue Beschreibung der erwarteten Performance sollte Teil der Anforderungen sein
- Erfüllbarkeit der Anforderungen im Blick behalten
  - "Back of the envelope calculations"
  - Frühe Performancetests der einzelnen Komponenten auf echten Beispielen
  - Frühe (und wiederholte) Lasttests auf einer Testinfrastruktur, die möglichst nah am Produktivsystem ist
- **Nicht:** Optimierungen für Probleme vornehmen, die noch gar nicht aufgetreten sind

NIH – "Not invented here" und Komplizen

"Viel zu kompliziert, so schwer kann das nicht sein..."

"Schneller neu-programmiert, als Doku gelesen..."

- Abzuwägen: In manchen Fällen ist Neuentwicklung tatsächlich wichtig
- Deshalb, stattdesse:
  - "Know your tools" – Lieber existierende Lösungen recherchieren.
  - Existierende Lösungen können angepasst werden: Fragen und/oder Forken
  - Neuentwicklung erst nach ausreichender Recherche
- **Randnotiz**: Bei Wiederverwendung von Bibliotheken immer einen Paketmanager verwenden und das "Bauen" (d.h. den Erstellungsprozess) automatisieren (z.B. mit make, Ant, sbt, rake, ...)

Erinnerung

Nächste Woche: Gastvortrag von Amra Avdic (NovaTec GmbH)