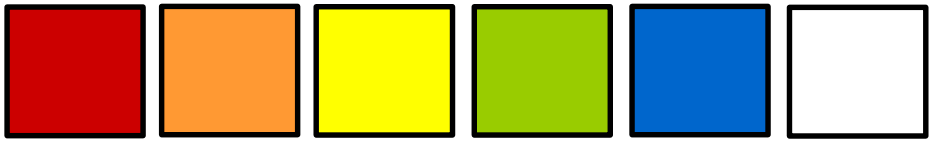


Stufe 1: Roter Grad von CCD

Lutz Prechelt
Institut für Informatik, Freie Universität Berlin

Bevor wir starten:

Aldo Cavini Benedetti, Flickr

4 wichtige Begriffe

1. "Abwägung"

dance-dream.de

fritz-berger.de

- Kosten/Aufwand gegen Nutzen

  oder

- verschiedene Vorteil/Nachteil-Kombinationen gegen einander

Ist schwierig wegen:

2. "Risiko"

- Möglichkeit eines unerwünschten Ereignisses
  - z.B. "Nutzen bleibt aus"
- Eintrittswahrscheinlichkeit
  - bei uns meist sehr unklar
- Schadenshöhe
  - bei uns oft unklar

Benötigt wird:

http://i.imgur.com/B2QZHtr.jpg

3. "Zutrauen" ("confidence")

- Gewissheit von "Eintrittswahrscheinlichkeit ist genügend niedrig"

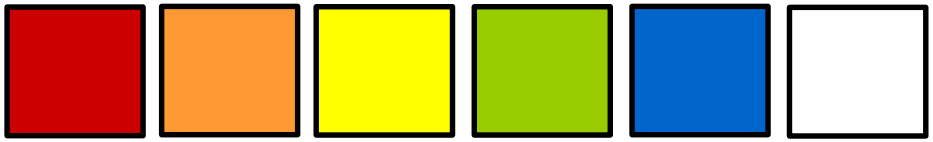
Problem im Kurs dabei oft:

4. "Kurpfuscherei" ("snake oil")

- Behauptete Wirkung einer Technik tritt nicht immer ein

James Gillray - Der Aderlass (um 1805)

Also bitte…
…nicht zu leicht beeindrucken lassen
Äh, wo waren wir? Ach ja:
Stefano Mortellaro, Flickr



Stufe 1: Roter Grad von CCD

Lutz Prechelt
Institut für Informatik, Freie Universität Berlin

Übersicht

Prinzipien:

- Don't Repeat Yourself (DRY)
- Keep it simple, stupid (KISS)
- Vorsicht vor Optimierungen!
- Favour Composition over Inheritance (FCoI)
- Integration Operation Segregation Principle (IOSP)

Gestalt des Codes

Praktiken:

- Pfadfinderregel befolgen
- Root Cause Analysis durchführen (RCA)
- Versionskontrollsystem einsetzen
- Einfache Refaktorisierungen anwenden
  - Extract Method, Rename
- Täglich reflektieren

Eigenes Verhalten

Don't Repeat Yourself (DRY)

- Was?
  - Redundanz im Code vermeiden
    - Logik
    - Daten, Konstanten
    - Annahmen!
- Wofür?
  - Evolvierbarkeit
  - Korrektheit
  - Produktionseffizienz
- Hinderungsgrund:
  - Anfangs ist Code mit Redundanz oft sehr viel schneller zu produzieren

Don't Repeat Yourself (DRY)

- Arten von Verletzungen:
  - Erhöhen den Pflegeaufwand
  - Code-Klone
    - Werden sehr oft inkonsistent geändert (→Defekte) [CloneEvol, ClonesMatter]
  - Verstreut hartkodierte Annahmen aller Art
    - z.B. Stringformate
- Hilfreiche Techniken:
  - Abstraktion
    - Unterprogramme, Klassen, Konstanten etc.
    - Funktionale Programmierg.
    - Aspekt-orientierte Programmierung (manchmal)
  - Convention over configurat.
  - Objekt-relationales Mapping mit Schemamanagement
  - Domänenspezif. Sprachen (DSLs), Generatoren
    - auch Javadoc u.ä.

Don't Repeat Yourself (DRY) nicht übertreiben

- Für alle Regeln in diesem Kurs gilt:
  Maß halten ist wichtig!

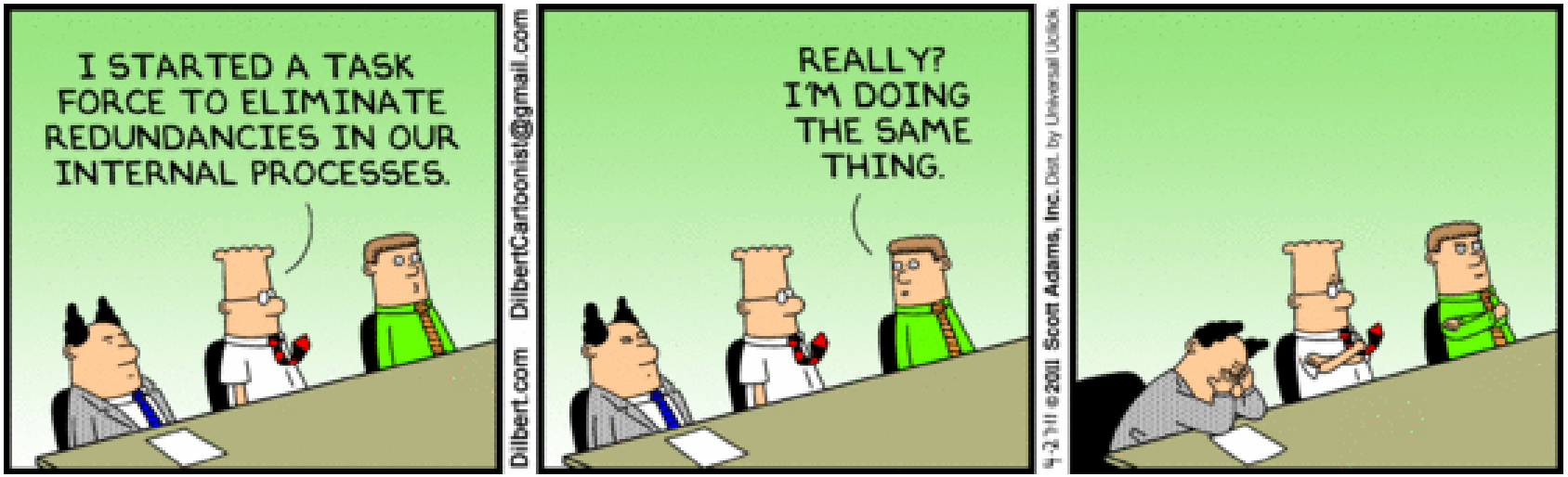
- DRY übertreiben heißt:
  Zu viel Aufwand für die Redundanzvermeidung treiben
  - Ziel ist Komplexitäts-*Reduktion*!

- Beispiel für Übertreiben:
  - Clean Code, Ch.14, p.195:
    - Bibliothek zur Verarbeitung v. Kommandozeilen-Args.
  - Was, wenn man diese erlaubten Formate in einer Fehlermeldung auflisten möchte?
  - Falsch:
    - Formate in Konstanten-Array speichern
    - Indices als symbolische Konstanten definieren
  - Richtig:
    - Redundanz in Kauf nehmen
    - Kommentar an Else-If-Kette kleben, der darauf hinweist

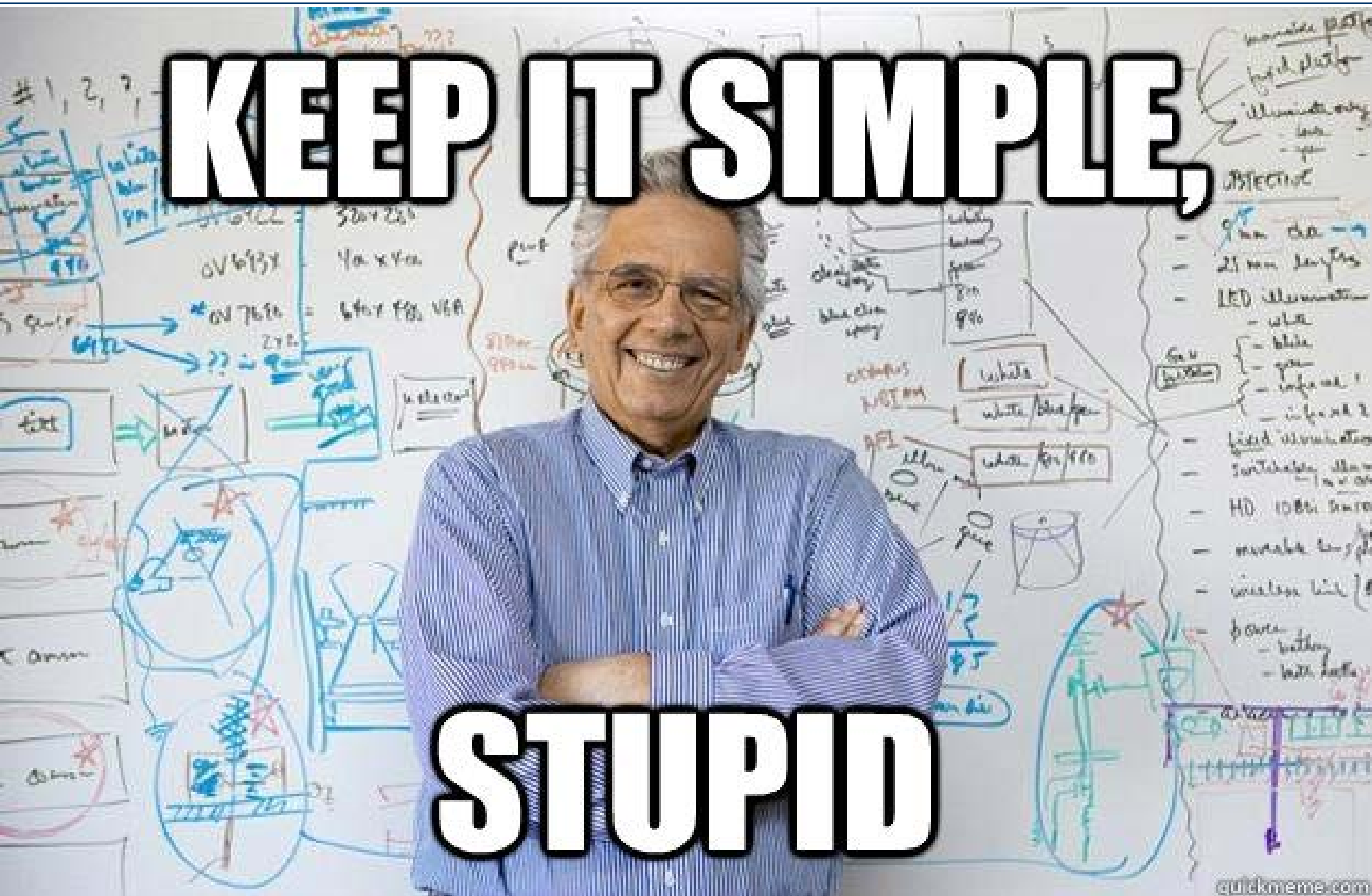
Ansichtssache!

Don't Repeat Yourself (DRY) auf Prozessebene

- Auch mehrfach wiederholte *Tätigkeiten* verletzten DRY
- Dagegen hilft
  - Automatisierung (z.B. Build, Tests, Release/Deployment)
  - Teamkoordination
  - Firmenweite Koordination



Keep it simple, stupid (KISS)



Keep it simple, stupid (KISS)

- Was?
  - Versuche überall (Architektur, Klassenentwurf, Logik, Datenstrukturen, Infrastruktur usw. usf.) einfache Lösungen zu finden
    - Verwende kompliziertere nur, wenn es gewichtige Gründe für sie gibt
- Wofür?
  - Evolvierbarkeit
  - Korrektheit
  - Produktionseffizienz
- Hinderungsgrund:
  - Einfach ist oft schwierig
    - Saint-Exupéry: *"Perfektion ist nicht dann erreicht, wenn es nichts mehr hinzuzufügen gibt, sondern wenn man nichts mehr weglassen kann."*
    - Unbedingt Maß halten!
  - Ob ein Grund gewichtig ist, ist oft unklar
  - Ob etwas einfach wirkt, hängt vom Vorwissen ab
    - Siehe "Simplicity by Design" in Java EE 6

Keep it simple, stupid (KISS)

- XP1's rules of simple design [DesignDead]:
  1. Runs all the tests
     - tests are comprehensive and do not fail
  2. Reveals all the intention
  3. Contains no duplication
     - OAOO: Do it once and only once (=DRY)
  4. Has the smallest number of classes or methods

- 4 ist das Gegengift gegen übertriebene Anwendung der hiesigen Ideen

- Hilfreiche Techniken:
  - Funktionalität ganz weglassen
  - Inkrementelle Entwicklung
  - Mehrere Entwerfer fragen

Keep it simple, stupid (KISS): vorher/nachher Beispiel

Vorher: schlecht

- Saros hat Operationen auf Dateiebene (FileUtils.java).
- Deren Teile wurden intern mittels WorkspaceRunnables zu atomaren Transaktionen zusammengefasst.
- Tatsächlich enthielt aber jede Transaktion ohnehin nur 1 Operation
- (Plus noch zwei andere überflüssige Sachen)

Nachher: gut

- → 230 LOC eingespart, auch mehrere komplexe Methoden
- https://github.com/saros-project/saros/commit/3adb19a

Keep it simple, stupid (KISS) auf Prozessebene

- KISS gilt nicht nur für das Produkt, sondern auch für alle Methoden und Arbeitsprozesse

Und für die Berührstellen (Abhängigkeiten) der beiden:

- "Simple" heißt auch, die Qualitätsmaßstäbe geeignet an die Umstände anzupassen
- Beispiel: Unsauberkeit im Code belassen, wenn
  - das Aufräumen viel Arbeit macht
  - aber die Unsauberkeit fast keine Probleme erwarten lässt.
  - Beispiel im Artikel [SuffDesign]:
    - einen "return null"-Smell im Code belassen, weil nur wenige alltägliche Änderungen dem Smell begegnen, aber viele Änderungen nötig wären, ihn zu beseitigen

Keep it simple, stupid (KISS) auf Metaebene

- Viele der Prinzipien in diesem Kurs sind Konkretisierungen von KISS
  - z.B. SLA, SRP, SoC, LSP, YAGNI
- Die Mehrzahl von diesen führt jedoch von KISS weg, wenn man die Anwendung übertreibt
  - z.B. alle obigen

- Eine zusätzliche, sehr wichtige Konkretisierung lautet deshalb:

  Sei nicht dogmatisch
  bei der Anwendung von Prinzipien!

KISS: The Zen of Python

Freie Universität Berlin

- http://legacy.python.org/dev/peps/pep-0020/
- Beautiful is better than ugly.
- Explicit is better than implicit.
- Simple is better than complex.
  - Complex is better than complicated.
  - Flat is better than nested.
  - Sparse is better than dense.
- Readability counts.

"pythonic style"

- Special cases aren't special enough to break the rules.
  - Although practicality beats purity.
- Errors should never pass silently.
  - Unless explicitly silenced.
- In the face of ambiguity, refuse the temptation to guess.
- There should be one-- and preferably only one --obvious way to do it.

(plus 6 other,
less important rules)

Das vielleicht schwierigste Prinzip von allen

Vorsicht vor Optimierungen!

- Was?
  - Optimiere nur, wo das wirklich nötig ist
    - Denn unnötige Optimierung kostet Aufwand und macht den Code meistens fehleranfällig, schwer zu verstehen und schwer zu ändern
  - (Donald Knuth: *"Premature optimization is the root of all evil"*)
    - Diskussion
- Wofür?
  - Evolvierbarkeit
  - Korrektheit
  - Produktionseffizienz
- Hinderungsgrund:
  - Manchmal juckt es einen halt…
  - Gelegentlich ist es schwierig herauszufinden, wo eine Optimierung hin muss

(Dies gehört eigentlich nach "Praktiken")

Vorsicht vor Optimierungen!

- Arten von Verletzungen:
  - Unnötig komplizierte Algorithmen/Verfahren
  - Unnötiges Caching
  - Verzicht auf Fehlerprüfungen
  - Verzicht auf Abstraktion
  - Einsparen weniger, schneller Anweisungen
  - etc.

- Hilfreiche Techniken:
  - Rules of Optimization
    1. Don't do it.
    2. For experts only: Don't do it yet

    (Michael A. Jackson)
  - Überschlagsrechnungen
  - Teile *ganz* weglassen
    - insbes. Ein-/Ausgaben: Platte, Netz, Bildschirm
  - Profiling
  - Lazy Optimization [LazOpt]
    - Eine Mustersprache

Vorsicht vor Optimierungen!: Negativ- und Positivbeispiel

Freie Universität Berlin

- Negativ:
  - Oracle-Transaktionsanwdg. lief zu langsam
  - Berater empfahl die Entfernung von Dutzenden "if global_debug_flag then call trace routine"
    - Quatsch!, denn:
  - Jede Transaktion umfasste ~200 solche Abfragen, aber auch ~20 DB-Abfragen
- Positiv anschließend:
  - Was wirklich half: Nicht mehr zu Beginn 1 Satz in leere Markertabelle schreiben und am Ende wieder löschen
    - Datensatz 1 ist bei Oracle sehr langsam
  - Dadurch wurde die Anwdg. um Faktor 3 schneller!

Favour Composition over Inheritance (FCoI)

- Was?
  - Verwende Vererbung nur für *is-a*-Beziehungen
    - nicht zur Wieder-verwendung von Methoden
  - In allen anderen Fällen bette hilfreiche Objekte lieber ein (Komposition) und schreibe einzeilige Methoden zur Verwendung (Delegation)
- Wofür?
  - Evolvierbarkeit
  - Korrektheit
- Hinderungsgrund:
  - Faulheit
  - Zufügungen zu einer Oberklasse
    - und dabei übersehen: Nicht alle Unterklassen brauchen dies!
    - aktuelle oder künftige

Favour Composition over Inheritance (FCoI)

- Effekte von Verletzungen:
  - Verwirrende Klassenhierarchie
  - Hohe Kopplung (zur Oberklasse)
    - evtl. ungünstige Methodennamen
    - Schlechtere Testbarkeit
  - Unsinnige Methoden mit geerbt

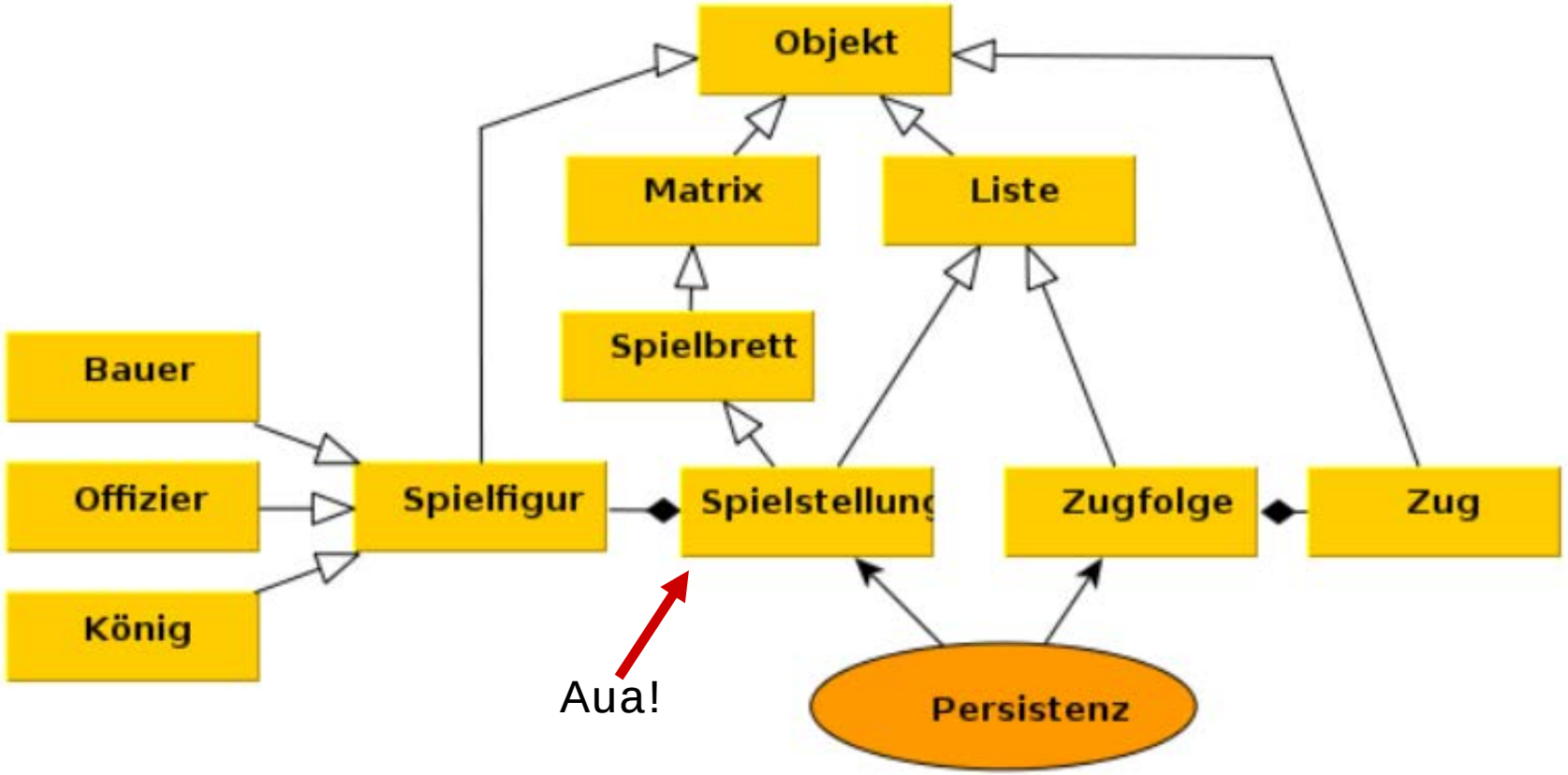
- Hilfreiche Techniken:
  - Komposition und Delegation
    - die GoF-Entwurfsmuster basieren oft darauf
    - Erlaubt Auswählen/Aus-tauschen zur Laufzeit

- Aber:
  - Evtl. ist Vererbung einfacher zu verstehen
    - Abwägung nötig
  - Beispiel von Melle Koning

Favour Comp. over Inheritance (FCoI): Negativbeispiele (zu wenig)

Freie Universität Berlin

1. FCoI/apache-aries-MDBStats.java (lokal)
   - Singleton-Klasse!
   - benötigt von Oberklasse nur: manche Leseoperationen
     - aber z.B. nicht: containsValue, put, putAll
2. Beispiel von https://de.wikipedia.org/w/index.php?title=Mixin&oldid=130573817



Integration Operation Segregation Principle (IOSP)

- Was?
  - Unterscheide Methoden streng in zwei Sorten:
  - *Operationen:* Kontrollstrukturen und Fremd-API-Aufrufe
    - aber keine Aufrufe der eigenen Codebasis
  - *Integrationen:* nur Aufrufe anderer Methoden dieser Codebasis

  oder:
  - Aufrufer einer Methode verarbeitet nie selbst einen Rückgabewert
    - [MsgProgModel]

- Wofür?
  - Evolvierbarkeit
  - Korrektheit

Häh? Wie soll das denn gehen?

- Spezialfall des Single Responsibility Principle (SRP)
  - → Orangener Grad

Übersicht

Prinzipien:

- Don't Repeat Yourself (DRY)
- Keep it simple, stupid (KISS)
- Vorsicht vor Optimierungen!
- Favour Composition over Inheritance (FCoI)
- Integration Operation Segregation Principle (IOSP)

Gestalt des Codes

Praktiken:

- Pfadfinderregel befolgen
- Root Cause Analysis durchführen (RCA)
- Versionskontrollsystem einsetzen
- Einfache Refaktorisierungen anwenden
  - Extract Method, Rename
- Täglich reflektieren

Eigenes Verhalten

Pfadfinderregel befolgen

- Was?
  - [ClC], S.14:
    "The Boy Scout Rule [...]
  - If we all checked-in our code a little cleaner than when we checked it out, the code simply could not rot.
  - The cleanup doesn't have to be something big.
    - Change one variable name for the better,
    - break up one function that's a little too large,
    - eliminate one small bit of duplication,
    - clean up one composite if statement."
- (Anmerkung:
  - in Wirklichkeit heißt es bei den Scouts nur:
    "Leave No Trace")
- Wofür?
  - Evolvierbarkeit
- Hinderungsgrund:
  - zu fokussiert auf aktuelles eigenes Ziel
  - zu faul

Pfadfinderregel befolgen

- Martin Fowler nennt das "Opportunistic Refactoring"
  - dann sind eigene Zeitabschnitte für Refactoring unnötig
  - weil sich Technische Schulden (technical debt) gar nicht erst ansammeln
    - siehe Dilbert
- [HowRefac]: Refactoring nebenbei (*"Zahnseide"*) ist viel häufiger als massenhaftes Refactoring (*"Wurzelbehandlung"*)

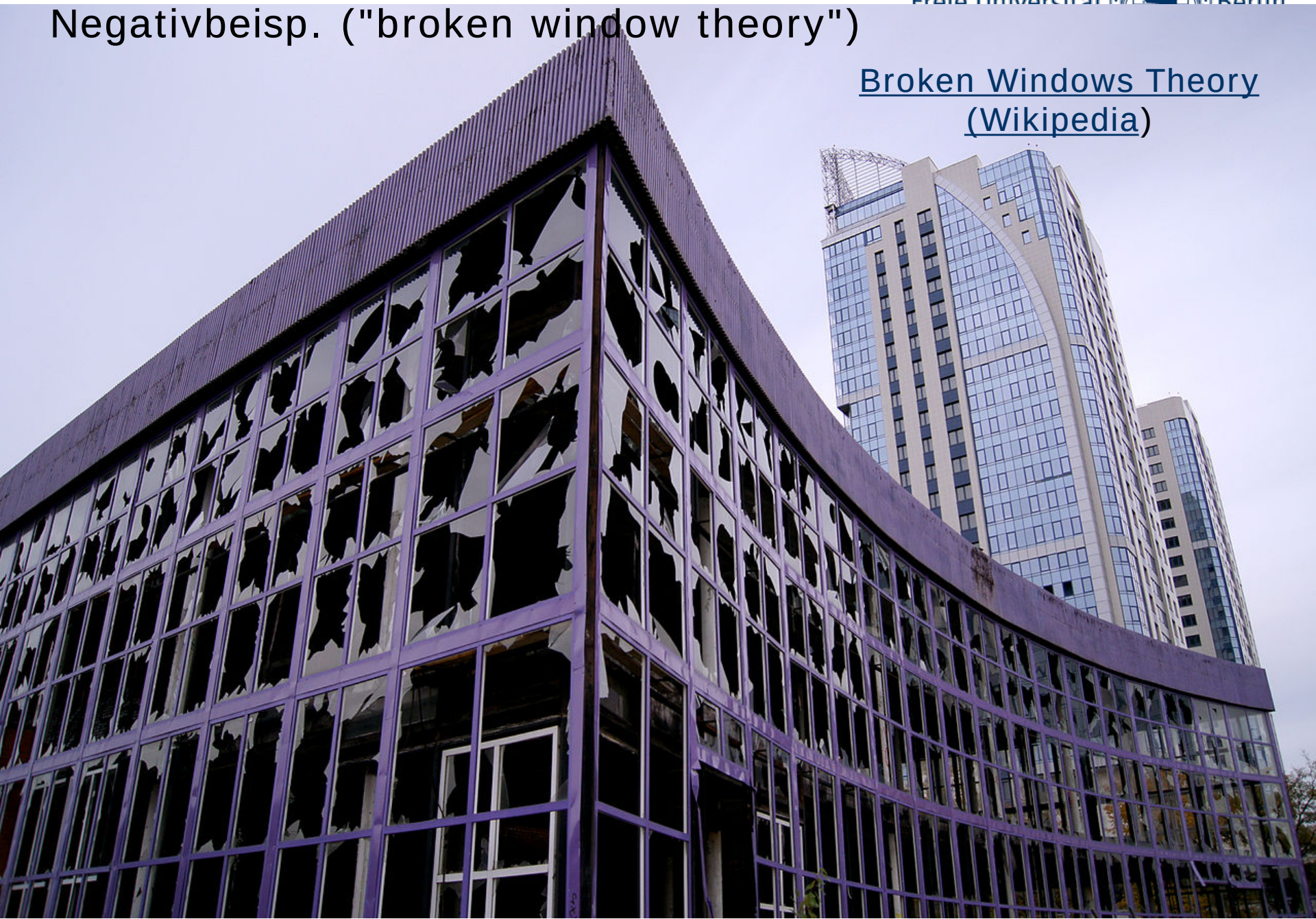
- Hilfreiche Techniken:
  - Code smells kennen [Smells]
    - Aber immer schön locker bleiben, bitte!
  - IDE gut beherrschen
  - Nicht ärgern, sondern handeln

Pfadfinderregel befolgen: Positivbeispiel

Freie Universität Berlin

- Saros benutzt eine ganze Reihe Threads
- Für das Debugging ist es hilfreich, deren Zweck leicht zuordnen zu können.
- Leider waren die Namen unsystematisch.
- Diese Änderung räumt (nur) diese Namen auf: https://github.com/saros-project/saros/commit/7c9bae

Pfadfinderregel befolgen: Negativbeisp. ("broken window theory")



Broken Windows Theory (Wikipedia)

Pfadfinderregel befolgen: Negativbeispiel (zu wenig)

Freie Universität Berlin

- Dieser größere Patch für Saros fügt mehrere Tests zu
  - auch ganze Klassen
- und räumt nebenbei einige Kleinigkeiten auf.
  - Pfadfinderregel wird also angewendet!
- Dabei werden in JIDTest.java
  - sowohl 2 neue Tests zugefügt (Z49-78neu) als auch
  - zwei schlechte Namen verbessert (Z83,91).
  - Leider gehören die beiden alten TO DO-Kommentare in Z80,88 jetzt eigentlich vor testMalformedJID (Z73) gezogen.
    - Hier hat der Pfadfinder also geschlafen

Root Cause Analysis durchführen (RCA)

- Was?
  - Wenn ein Problem auftritt (Defekt oder anderes), beseitige es nicht nur, sondern ermittle die ganze Ursachenkette, um den Ur-Grund (eine frühe Ursache) abstellen zu können
- Wofür?
  - Reflexion
- Hinderungsgrund:
  - Kognitiv evtl. schwierig
  - Relevante Fakten oft schwierig zu ermitteln
  - Für die Betroffenen oft peinlich

Root Cause Analysis durchführen (RCA)

- Phänomene ohne:
  - Prozessverbesserung erfolgt nur langsam
  - Erfahrungslernen erfolgt nur langsam
  - Man erkennt Problemlagen wieder, kann sich aber nicht mehr erinnern, wie sie sich beim letzten Mal lösen ließen
- Hilfreiche Techniken:
  - Verletzte Prinzipien aufdecken
    - und dann weiter "Warum?" fragen
  - Gute Bücher oder Artikel zu denkbaren Ursachen lesen
  - Kolleg/inn/en fragen

Root Cause Analysis durchführen (RCA): Nicht-Software-Beispiel

- Nach der Explosion des Space Shuttle Challenger 1986 suchte Richard Feynman in der Untersuchungskommission auch nach soziologischen Ursachen
- Sein Ergebnis:
  - **Direkter Grund:** O-Ring undicht
  - Der war schon früher durch Beschädigungen aufgefallen. Diese waren unerwartet.
    - Für Ingenieure bedeutet so etwas höchste Alarmstufe.
  - Management befand die Konstruktion für flugtauglich, wegen d. "Sicherheitsreserven"
    - Die Reserven kennt man aber nur, wenn man die Konstruktionseigenschftn versteht. Das war hier nicht der Fall.
  - **Auslösender Grund des Unfalls:** Die Start-Entscheidungsprozesse gaben den Ingenieuren zu wenig Stimme.
  - **Fazit (Urgrund):** *"[NASA officials] must be realistic in making contracts, in estimating costs, and the difficulty of the projects. […] For a successful technology, reality must take precedence over public relations, for nature cannot be fooled."*

Versionskontrollsystem einsetzen

- Was?
  - Verwalte allen Quellcode und andere Artefakte des ganzen Teams stets in einer Versionsverwaltung
- Wofür?
  - Korrektheit
  - Produktionseffizienz
  - Reflexion
- Hinderungsgrund:
  - schreiende Ignoranz

Versionskontrollsystem einsetzen

- Symptome ohne:
  - Strenge Eigentümerschaft von Code
    - Manuelle Absprachen über Änderungserlaubnis an Datei X
  - Seltene Integration von Änderungen im Team
  - Defekte, die nach Beseitigung erneut "auftauchen"
  - Verlust von Quellcode
    - ab dann: Änderungen auf Assembler-Ebene
- Hilfreiche Techniken:
  - Just do it
  - Git verstehen
    - Ziemlich kompliziert, aber für Könner unerhört hilfreich
    - Git ist faktisch ein Dateibaum-Änderungs-Editor

Versionskontrollsystem einsetzen: Negativbeispiel

Freie Universität Berlin

Ca. 1992:

- Bei der Entwicklung des Modula-2* Compilers trat ein Defekt auf, dessen Debugging zwei Tage dauerte
- Zwei Wochen später nochmal einer gleicher Schwere
  - Wieder langes Debugging
- Der stellte sich als derselbe heraus:
  Ein Entwickler hatte die letzten Änderungen versehentlich auf einer alten Kopie des Quellcode-Dateibaums gemacht

Versionskontrollsystem einsetzen: Arbeitsstil mit Git

Freie Universität Berlin

- Änderungen werden in die kleinsten sinnhaltigen **Teil-Änderungen** zerlegt,
  - die noch in sich abgeschlossen sind.
- Jede solche wird einzeln eingecheckt
  - Mit einer klaren Beschreibung
  - Jede Änderung zunächst auf einem privaten Zweig
- Defekte beim Integrationstest lassen sich dann ggf. leicht lokalisieren
- Durchsichten sind schnell und effektiv
- Vollendete Teil-Änderungen werden prompt auf den Hauptzweig vereint
  - Häufig, da Teil-Änd. klein
  - Probleme selten, da Vereinigung häufig
  - Bei Problemen ist Lokalisierung einfach
- Nach Abschluss wird der Zweig geschlossen
  - Er dokumentiert ggf., welche Teil-Änderungen zusammen gehörten

http://git-scm.com/about

- analog: Mercurial u.a.

Einfache Refaktorisierungen anwenden (Extract Method, Rename)

Freie Universität Berlin

- Was?
  - Benenne blöd benannte Pakete, Klassen, Methoden, Variablen, Parameter etc. um
  - Extrahiere kleine Hilfs-methoden aus unschön groß gewordenen Methoden
- Wofür?
  - Evolvierbarkeit
  - Reflexion
- Hinderungsgrund:
  - Faulheit
  - Keine automatisierten Tests vorhanden (→ Angst)
  - Bei eigenem Code:
    Man bemerkt das Problem nicht
  - Bei fremdem Code:
    Man ist evtl. zu sehr vom Verstehen abgelenkt
    - das sollte aber gerade das Signal sein!

Einfache Refaktorisierungen anwenden (Extract Method, Rename)

Freie Universität Berlin

- Arten von Verletzungen:
  - kryptische Namen
  - nichtssagende Namen
  - irreführende Namen
  - mehrdeutige Namen
  - lange Methoden
  - tiefe Verschachtelung in Methoden
  - Verletzungen von DRY, SLA, SRP, LoD
    - für SLA, SRP, LoC siehe nächste Wochen
- Hilfreiche Techniken:
  - Entsprechende IDE-Operationen beherrschen
    - und ihre Grenzen kennen
      - hier sind dynamische Sprachen im Nachteil!
    - und Code so schreiben, dass diese Grenzen wenig stören
  - Kollegen nach Meinung über meinen Code fragen

Einfache Refaktorisierungen anwenden: Beispiele 1, 2

Freie Universität Berlin

1.

- Rename instance variable sarosSession → session
  - 11x in 1 Datei
- Rename method accept → run
  - 2x in 2 Dateien
- Rename method start → run
  - 2x in 2 Dateien

- Dies ist in diesem Commit kombiniert
  - plus: Exemplarvariable zugefügt, toter Code entfernt

2. Extract Method createAccountMenuItem()

- Hier war zuvor das Prinzip *Single Level of Abstraction* verletzt
  - siehe nächste Woche

- Es ist ein schlechtes Zeichen, wenn Methoden mehrmals hintereinander wachsen

Täglich reflektieren

- Was?
  - Reflektiere nach jedem Arbeitspaket:
    - Was war gut?
    - Was nicht? Warum?
    - Was kann ich besser machen? Wie?
  - Bilde genügend kleine Arbeitspakete
    - ca. 0,5 bis 1 Tag
- Wofür?
  - Reflexion
- Hinderungsgrund:
  - Angst vor der eigenen Unvollkommenheit
  - Denkfaulheit
- Zeitnot darf kein Hinderungsgrund sein:
  - Reflektion senkt Stress, weil man sich weniger als Opfer der Umstände fühlt

Täglich reflektieren

- Phänomene ohne:
  - Den "gleichen" Fehler mehrfach machen
  - Dogmatisches Verfolgen von Regeln
  - Langsames Hinzulernen
  - Wenige Lernerfolge werden bemerkt
- Hilfreiche Techniken:
  - Aktuellen CCD-Grad checklistenartig durchgehen
  - Vergleich mit ähnlichen Situationen ein Jahr zuvor
  - Nach längerem Debugging und übersehenen Anforderungen RCA machen
  - Mit Kolleg/inn/en diskutieren

Täglich reflektieren: Positivbeispiel (gelungen)

- Ich habe mich mit dem TDD-Entwicklungsstil schwergetan
  - Das war mir durch Reflektion klar bewusst

- Im Gespräch darüber mit einem TDD-Forscher fiel mir auf, dass das mit meinem Denkstil (MBTI intuitive, d.h. top-down-orientiert) zusammenhängen dürfte

- Beim späteren Lesen über TDD fiel mir deshalb ein Rat sofort als sehr hilfreich auf: Testliste vorab formulieren

Übersicht

Prinzipien:

- Don't Repeat Yourself (DRY)
- Keep it simple, stupid (KISS)
- Vorsicht vor Optimierungen!
- Favour Composition over Inheritance (FCoI)
- Integration Operation Segregation Principle (IOSP)

Gestalt des Codes

Praktiken:

- Pfadfinderregel befolgen
- Root Cause Analysis durchführen (RCA)
- Versionskontrollsystem einsetzen
- Einfache Refaktorisierungen anwenden
  - Extract Method, Rename
- Täglich reflektieren

Eigenes Verhalten

Hausaufgabe
(nächste Wochen gehen analog)

Freie Universität Berlin

- Setzen Sie mindestens zwei der heutigen Elemente in Ihrem Projekt ein
  - möglichst 1 Prinzip und 1 Praktik
- Gern auch mehr, wenn es sich ergibt

- Wir machen später noch einen zweiten Durchgang
  - Dann bitte *andere* Prinzipien/Praktiken auswählen

- Stellen Sie uns das nächste Woche vor
  - **in max. 8 Minuten:**
  - Erinnerung an Projekt
  - Ausgangssituation
    - Probleme, Chancen
  - Praktik
    - Warum gerade diese?
  - Einsatzweise
    - Achtung: Wir interessieren uns für die Praktik, nicht für ihr Projekt
  - Kritik: Was war/ist gut, was noch nicht?
    - Reflektion: Was nehme ich daraus mit?

Thank you!