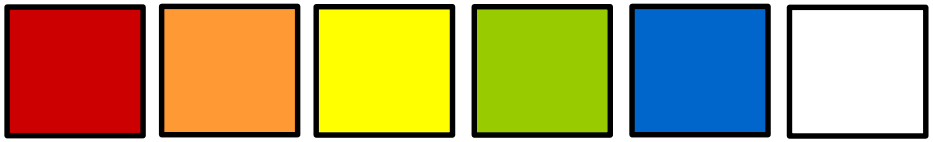


Stufe 2: Oranger Grad von CCD

Lutz Prechelt
Institut für Informatik, Freie Universität Berlin

Übersicht

Prinzipien:

- Single Level of Abstraction (SLA)
- Single Responsibility Principle (SRP)
- Separation of Concerns (SoC)
- Source Code Konventionen

Praktiken:

- Issue Tracking
- Automatisierte Integrationstests
- Lesen, Lesen, Lesen
- Reviews

Single Level of Abstraction (SLA)

- Was?
  - Die Anweisungen einer Methode sollten möglichst einen Abstraktionsgrad genau eine Stufe unter dem des Methodennamens haben
  - [ClC] Kap. 3
    [CoCo] Kap. 7
- Wofür?
  - Evolvierbarkeit
- Hinderungsgrund:
  - Führt bei strenger Auslegung zu vielen winzigen Methoden
    - die sich z.T. auch noch Wissen teilen
  - Abstraktionsgrad ist eine unklare Größe

Single Level of Abstraction (SLA)

- Arten von Verletzungen:
  - Aufrufe von Domänenoperationen und von technischen Detailoperationen mischen sich
  - Aufrufe von umfangreichen und von sehr feinen Operationen mischen sich
- Vorsichtig anwenden!
  - Sonst ist das Ergebnis oftmals sehr dubios
    - Verlust von Lokalität
    - Inflation winziger Methoden
- Hilfreiche Techniken:
  - Extract Method Refactoring

Single Level of Abstraction (SLA): Kritisches Beispiel

Freie Universität Berlin

- Saros: GOTOInclusionTransformation (lokal)
  - letzte Methode:
    transform(DeleteOperation delA, DeleteOperation delB)
  - Definiert die Semantik für wetteifernde Textlöschungen

- Hier müsste für strengen SLA jeder Vergleich und jede Arithmetik a+b, a-b in eine Methode ausgelagert werden
  - Das Resultat wäre unangenehm nichtlokal
  - und viel schwieriger per Durchsicht zu prüfen
- SLA ist oftmals Geschmackssache!

- Sinnvolle Abmilderung: *Similar* Level of Abstraction
  - Hier z.B. könnte man Zwischenvariablen für die Ausdrücke einführen

Das führt zu:

Single Level of Abstraction (SLA): Sinnvolle Abwandlung

Freie Universität Berlin

- Anderer Ansatz: Mische WAS und WIE möglichst wenig
- Positives Beispiel: http://www.markhneedham.com/blog/2009/06/12/coding-single-level-of-abstraction-principle/
  - validPolicyNumber.cs (lokal)

```
private bool validPolicyNumber(string policyNumber)
{
    var hasExpectedPrefix = policyNumber.Substring(0,5) == "POLIC";
    var followedBy7Digits = Regex.IsMatch(policyNumber.Substring(6,7),
                                          "^[0-9]{7}$");
    var hasLengthOf12 = policyNumber.Length == 12;

    return hasExpectedPrefix && followedBy7Digits && hasLengthOf12;
}
```

Single Responsibility Principle (SRP)

- Was?
  - Jede Klasse sollte nur eine Zuständigkeit haben
    - Und was heißt das, bitte?
    - Jede Klasse sollte nur einen Grund haben, aus dem sie geändert werden muss
      - Und was heißt *das*?
  - [ClC] Kap. 10 sagt:
    - If a 25-word summary includes 'and', be wary!
- Wofür?
  - Evolvierbarkeit
  - Korrektheit
- Hinderungsgrund:
  - Umsetzung evtl. mühsam und die Vielzahl kleiner Klassen evtl. unerwünscht
  - "Zuständigkeit" ist ein unklares Konzept

Single Responsibility Principle (SRP)

- Arten von Verletzungen:
  - Früh in der Entwicklung: Große Klassen mit vielen Zuständigkeiten
    - Z.B. durch Bottom-Up-Vorgehen
  - Später: Kleine Funktionserweiterungen mit in eine vorhandene Klasse packen
    - auch solche mit anderer Zuständigkeit
- Hilfreiche Techniken:
  - Zweck einer Klasse im Namen möglichst vollständig widerspiegeln
  - Refactoring, refactoring, refactoring
  - Keine Scheu vor kleinen Klassen
  - IDE-Funktionen zum Browsen gut beherrschen

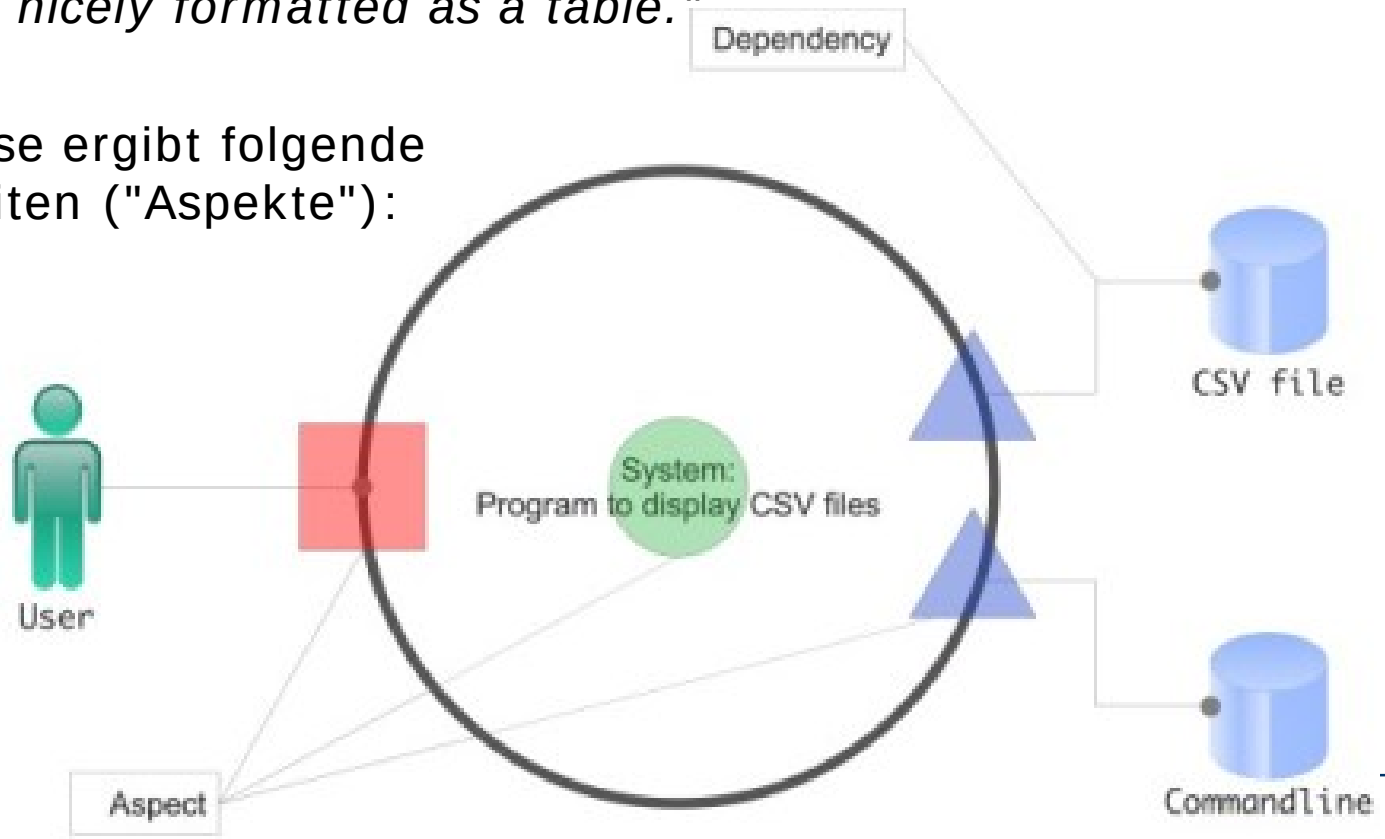
Single Responsibility Principle (SRP): Gilt nicht nur für Klassen

Freie Universität Berlin

- SRP ist eine Daumenregel für sinnvoll aufgeteilte Modularität
- Sie ist auf mehreren Ebenen anwendbar:
  - Subsysteme
  - Module, Komponenten, Pakete
  - Klassen
  - Methoden
- D.h. z.B.: Mehrere Zuständigkeiten einer Klasse können OK sein, wenn sie klar auf Methoden aufgeteilt sind
  - aber genug Kohäsion sollte natürlich da sein
  - nicht wie z.B. hier
- Am besten denkt man über Zuständigkeiten auf einem abstrakten Niveau nach
  - nicht auf der Ebene von Sprachkonstrukten
  - z.B. wie folgt:

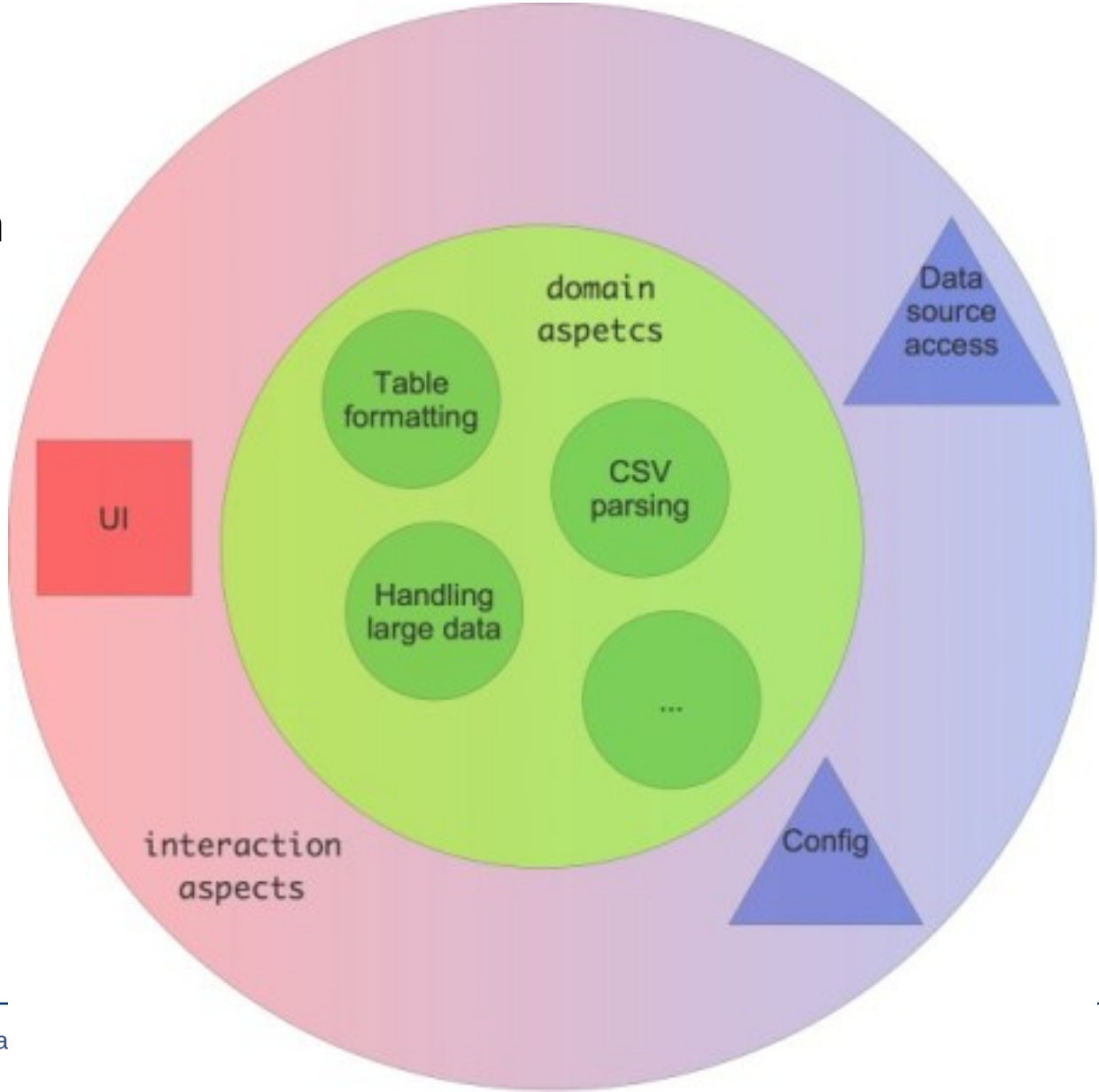
Single Responsibility Principle (SRP): Beispiel

- Beispiel von Ralph Westphal:
  - Anforderungen sind: "*The program reads in CSV data files and displays them in a page-wise manner.*
    The name of the file to display as well as the number of lines per page is passed in on the command line.
    Each page is nicely formatted as a table."
  - Grobe Analyse ergibt folgende Zuständigkeiten ("Aspekte"):



Single Responsibility Principle (SRP): Beispiel (2)

- Einige weitere Überlegungen (inkl. denkbarer Änderungen) führen dann zu:



Single Responsibility Principle (SRP): Sinnvoller Mindeststandard

Freie Universität Berlin

- Was man wirklich immer zu trennen versuchen sollte, sind ein paar grobe Bereiche
- Zum Beispiel

  (sinnvoll jedenfalls bei Informationssystemen):
  - Komponenten verschiedener "Software-Blutgruppen" [Quasar]
  - Ports & Adapters-Architektur
    - (hier sind die T's besonders klar von einander getrennt
    - aber T und R vermengt)
- Die Blutgruppen:
  - **A**: nur Anwendungslogik
  - **T**: nur technische APIs
    - und möglichst je nur eines
  - **0**: weder A noch T
    - z.B. Behälterklassen, Standardalgorithmen
  - **AT**: unerwünscht!
  - **R**: Repräsentationsanpass-ungen
    - insbesondere Adapter von A nach T, eine unvermeidbare und milde Sorte von AT
    - z.B. von/zur GUI, von/zur DB

Separation of Concerns (SoC)

- Was?
  - Trenne einzelne **Belange** eines Programms möglichst so von einander, dass sie sich nicht gegenseitig beeinflussen und bequem einzeln ändern lassen
    - Modularität!
- Wofür?
  - Evolvierbarkeit
  - Korrektheit
  - Produktionseffizienz
- Hinderungsgrund:
  - Verlangt Disziplin
  - Belange erkennen ist nicht immer einfach
  - Für viele Belange gibt es zu wenig technische Trennungs-Unterstützung
  - KISS

Separation of Concerns (SoC)

- Indiz von Verletzungen:
  - Eine an sich kleine Anforderungsänderung benötigt Codeänderungen an sehr vielen Stellen.
    - Verstreuen, "scattering"
  - Eine Stelle im Code (Methode) hat mit mehreren ganz verschiedenen Belangen zu tun
    - Verwickeln, "tangling"

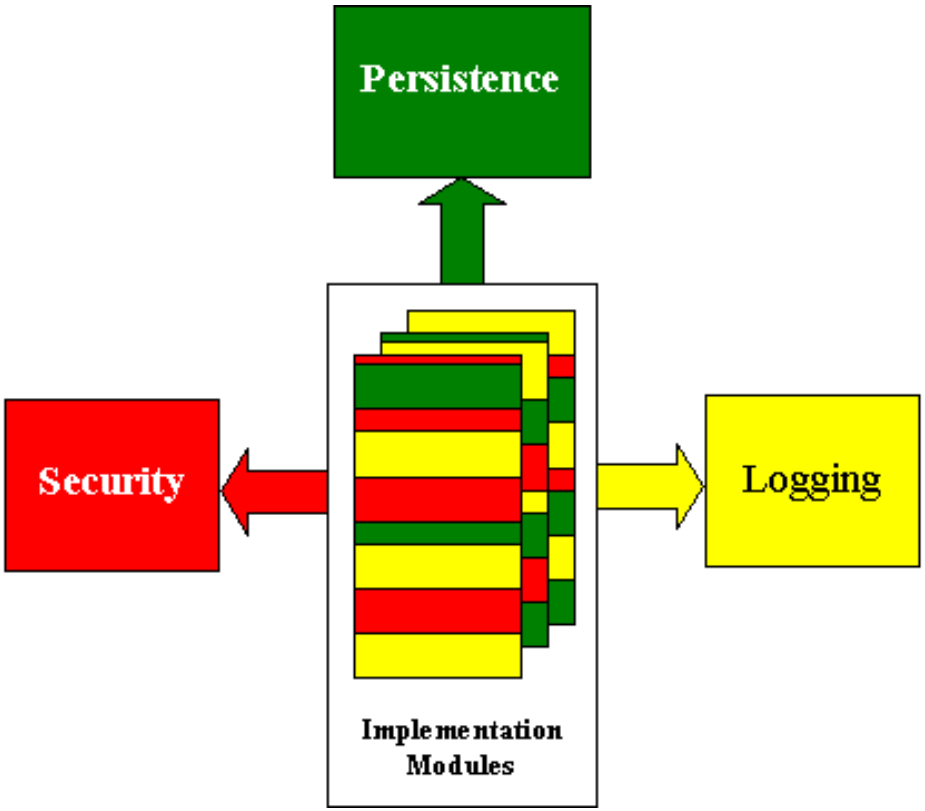
- Hilfreiche Techniken:
  - Bemerken, wenn Wissen an vielen Stellen im Code immer wieder auftaucht
  - Module, Schnittstellen, Wiederverwendung, DRY
  - Aspekt-orientierte Programmierung (AOP)
    - Abtrennen von Querschnittsbelangen (cross-cutting concerns)
      - z.B. Logging, Zugriffsschutz u.ä.
  - Anforderungsverfolgung (requirements tracing)
    - Um Verwickeln zu bemerken (→ SRP)

Separation of Concerns (SoC): AOP-Beispiel

- Grundproblem siehe Bild
  - aus [AOP]
- Beispiele siehe [AOP]:
  - 1. Teil 1, Seite 1: SomeBusinessClass
  - 2. Teil 1, Seite 2:
    - CreditCardProcessor (Zielklasse)
    - Logger (Kern von Aspekt)
    - Beschreibungstext darunter (Weaving-Spezifikation)
    - CreditCardProcessorWithLogging (Resultat)
    - aspect LogCreditCardProcessorOperations (AspectJ-Spezif. v. Aspekt)

2

- Man kann das auch z.B. mit Entwurfsmustern lösen
  - Stets prüfen, wo AOP lohnt!



Source-Code-Konventionen

- Was?
  - Vereinbare projektweit oder unternehmensweit gültige Vorgaben über den Aufbau von Quellcode
    - insbes. Namens- und Kommentarregeln
- Wofür?
  - Evolvierbarkeit
- Hinderungsgrund:
  - störrische Teammitglieder
    - z.B. übersparsame (lokal)

Source-Code-Konventionen

- Wirkung von Verletzungen:
  - Details: Lesen wird durch ständige Ablenkung erschwert, weil der meiste Code optisch einfach nicht "richtig" aussieht.
  - Namensregeln: Man muss viel öfter etwas selbst nachprüfen, anstatt einem Namen vertrauen zu können.
  - Kommentarregeln: Wichtige Fragen bleiben offen oder Informationen sind unnötig mehrdeutig.
- Hilfreiche Techniken:
  - Für einige Sprachen gibt es leidlich standardisierte Konventionen, z.B.
    - Java
    - Python
    - Liste bei Wikipedia
    - deren Übernahme spart viel Diskussionsaufwand ein
  - Namens- und Kommentarkonventionen siehe [ClC], Kap. 2 bis 4
  - Details wie Spacing siehe die gängige Praxis

Source-Code-Konventionen: Namensregeln

- Namen sollten sinnhaltig sein, aussprechbar und sollten Semantik beschreiben
  - Konsistenz beachten
  - zufällige Ähnlichkeiten meiden
  - keine Wortspiele!
- Größere Sichtbarkeits-bereiche verlangen längere Namen
  - Domänenbegriffe für Problembereichskonzepte
  - Technische Begriffe für Lösungsbereichskonzepte

Source-Code-Konventionen: Kommentarregeln

Freie Universität Berlin

- Kommentarregeln sind je nach Kontext sehr variabel
  - z.B. braucht die Java SE API sehr viel Dokumentation
  - hingegen eine rein interne Codebasis eines stabilen und kompetenten Teams recht wenig
    - seltener Fall!
- Es gibt inzwischen Leute, die ernsthaft glauben, Kommentare seien stets ein Indiz für schlechten Code
  - selbst-dokumentierender Code kann aber realistisch nur beschreiben WAS er tut, nicht warum
- Mindestens sollte dokumentiert werden:
  - Zweck von Klassen
    - also deren Rolle im Gesamtgefüge;
    - nicht zu verwechseln mit der Semantik
      - ein langer Name beschreibt meist bestenfalls Semantik
  - Wichtige Entwurfsüberlegungen
    - "Warum so und nicht anders?"

Übersicht

Prinzipien:

- Single Level of Abstraction (SLA)
- Single Responsibility Principle (SRP)
- Separation of Concerns (SoC)
- Source Code Konventionen

Praktiken:

- Issue Tracking
- Automatisierte Integrationstests
- Lesen, Lesen, Lesen
- Reviews

Issue Tracking

- Was?
  - Aufgaben, die zurückgestellt werden oder länger dauern, müssen aufgeschrieben werden
    - insbesondere (aber nicht nur) Defekte
- Wofür?
  - Korrektheit
  - Produktionseffizienz
  - Reflexion
- Hinderungsgrund:
  - erfordert Disziplin

Issue Tracking

- Phänomene ohne:
  - Defekte werden lange Zeit nicht bereinigt
  - Mäßig wichtige Defekte werden nicht vor unwichtigen bereinigt
  - Zwischenstände bei der Defektbereinigung gehen verloren
  - aktuelle Aufgaben werden nicht erledigt
    - oder manchmal doppelt
  - Funktionalitäten werden vergessen
- Hilfreiche Techniken:
  - Story-Backlog
    - z.B. als Wikiseite
  - Story-Karten, Task-Karten
    - sehr gut an der Wand (wenn Team in 1 Raum)
  - Bugtracker-SW
    - Achtung: Nur einsetzen wo nötig

Issue Tracking: Negativbeispiel? (zu viel?)

Auf Defektebene:

- http://sourceforge.net/p/dpp/bugs/804/
  - Eine Meldung, die manchmal irreführend ist
  - Eintrag am 24.4.2013 mit Prio 5
  - Prio sofort gesenkt auf 2
  - "Ist erheblich verwirrend" → Prio auf 4 am 25.6.
  - Diskussion am 5.-7.10.
  - Korrekturvorschlag am 10.11.
  - Blieb dennoch offen

Issue Tracking: Positivbeispiel (gelungen)

Freie Universität Berlin

Methodenvorschlag für die Anforderungsebene:

- http://www.jamesshore.com/Agile-Book/iteration_planning.html
  - Abschnitt "Tracking the iteration" und schematische Abbildung
    - erledigte Karten grün umrandet
    - Karten in Arbeit werden mitgenommen und ersetzt durch Entwicklername
  - Angewandtes KISS!

- Reale Kartenwände:
  - http://agilexp.com/informativeworkspace/walls.php

Automatisierte Integrationstests

- Was?
  - Habe automatisierte Tests, die das korrekte Zusammenspiel vieler Module prüfen
  - Und benutze sie häufig
- Wofür?
  - Evolvierbarkeit
  - Korrektheit
  - Produktionseffizienz
- Hinderungsgrund:
  - Auf GUI-Ebene mühselig
    - z.B. Selenium, Watir
  - Also besser darunter bleiben
    - Das geht aber nur mit guter Modularisierung ausreichend!
  - Tests laufen evtl. lange
  - Tests müssen gepflegt werden

Automatisierte Integrationstests

- Phänomene ohne:
  - Angst vor Änderungen
    - Refaktorisierungen unterbleiben, Code verfällt!
  - Strengere Code-Eigentümerschaft
    - → unflexible Teamarbeit
  - Seltene Freigaben
  - Hoher manueller Testaufwand
  - Weniger Spaß am Entwickeln
- Hilfreiche Techniken:
  - Keine Geschäftslogik im GUI!
  - Klare, einfache Dienst-Schnittstellen überall
  - Spezifikation durch Beispiel ("Specification by Example") als Gewohnheit
    - kann viel Dokumentation ersetzen
    - [SpecByEx]
  - Möglichst nicht gegen Ein-/Ausgabeformate testen
    - ändern sich zu oft.
    - Ausnahme: REST API

Automatisierte Integrationstests: Zu viel

- Manche Teams haben so viele Integrationstests, dass deren Durchlauf zu lange dauert (Stunden)
  - viele der positiven Wirkungen schwächen sich dann stark ab

Automatisierte Integrationstests: Positivbeispiel (gelungen)

Freie Universität Berlin

von SoundCloud:

- Teildienst "Buckster"
  - Bezahlfunktionen
- Integrationstest für die zentrale Operation "checkout"
  - ein externer Dienst wird durch Attrappe ersetzt
  - enthält 2 hartkodierte JSON-Objekte
  - 90 Zeilen, Ruby mit RSpec
  - integration_purchase.rb

WP:

- Monatlich Wertpapiere aus einer festen Gruppe kaufen/verkaufen nach rein zahlenmäßigen Kriterien
  - Hier: nur historische Kurse, nur Kauf*entscheidung*, ETFs
  1. Lies WP-Liste aus Exceldatei
  2. Hole aktuelle Kursdaten von Yahoo
  3. Werte Kriterien aus
  4. Drucke Bericht m. Aktionen ggü. Vormonat
- Normalbenutzg↔Integr.test
  - Problem: *aktuelle* Kurse

Lesen, Lesen, Lesen

- Was?
  - CCD: "Wir schlagen [...] vor, pro Jahr wenigstens 6 Fachbücher zu lesen. Ferner sollten Periodika regelmäßig gelesen werden und darunter verstehen wir neben Fachzeitschriften auch Blogs."
- Wofür?
  - Produktionseffizienz
  - Reflexion
- Hinderungsgrund:
  - Keine Zeit!
    - Material, das man aktuell brauchen kann, darf man während der Arbeitszeit lesen (in Maßen). Ansonsten:
    - Diagonal lesen ist oft schon genug: schafft Gewahrsein ("awareness")
    - Interesse an der Materie auch in der Freizeit gehört zu unserem Beruf
  - Keine Lust!
    - Falschen Beruf gewählt?
    - Falschen Arbeitsplatz?

Lesen, Lesen, Lesen

- Phänomene ohne:
  - Feststecken in alten Gewohnheiten
  - Keine Ahnung von den besten verfügbaren Frameworks und Werkzeugen
  - Routinierte Langeweile; Spaß an der Arbeit sinkt immer weiter ab
- Hilfreiche Techniken:
  - Modus 1: Problem lösen
    - konzentrierte Suche nach einer *ausreichenden* Quelle
      - Nicht pfuschen!
    - Nur das nötige Lesen
  - Modus 2: Dazulernen
    - für Technologiekram im eigenen Dunstkreis
    - Suche nach einer wirklich *guten* Quelle
    - Durcharbeiten, nicht nur Lesen
    - Praktisch anwenden
  - Modus 3: Horizont erweitern
    - Gute Quellen empfehlen lassen
    - Themenbereiche
      - Anderer Technologiekram
      - Methodenkram und Grundlegendes

Reviews

- Was?
  - Code und Entwürfe sollten von einer zweiten Person durchgesehen und diskutiert werden
    - wenigstens alles, was nicht ganz simpel ist
- Wofür?
  - Evolvierbarkeit
  - Korrektheit
  - Produktionseffizienz
  - Reflexion
- Hinderungsgrund:
  - Scheu vor Kritik
    - Das sollte sich ggf. dringend ändern!
  - Keine Zeit
    - Das ist fast immer Einbildung: Klug gemachte Durchsichten sparen Zeit
  - Spezialwissen über diesen Codebereich fehlt anderen
    - Das muss sich dringend ändern!

Reviews

- Phänomene ohne:
  - Hohe Defektdichten
  - Unnötig schwer änderbare Entwürfe

- Hilfreiche Techniken:
  - Selbstdurchsicht
  - Cleanroom Software Engineering
    - systematische Defektvermeidung auf hohem Niveau
      - Top-Down-Entwicklung
      - Korrektheitsbegründg. in jedem Schritt
      - Prüfung im Team
      - → viel KISS
  - Walkthrough
  - Paarprogrammierung
  - Review-Management-SW (z.B. Gerrit)

Reviews: Positivbeispiel (gelungen)

- Saros Change 253
  - bekam im Patch Set 2
    5 Kommentare zu BuddySessionDisplayComposite.java wie folgt:
- Dateiänderung enthielt subtilen Defekt:
  Für Bildschirme geringer Farbtiefe kann und sollte man bei SWT Farbobjekte nach Benutzung wieder freigeben
  - Sowas muss man erst mal überhaupt wissen!
  - **Durch Test kaum zu entdecken**
- Gutachter hatte das Wissen und hat es transportiert
  - Zeile 454neu ff.
  - (Zu Patch Set 1 hatte er erst nur geschrieben:
    "Here is the place to dispose your colors."
    Das war ungenügend verständlich gewesen.)

Übersicht

Prinzipien:

- Single Level of Abstraction (SLA)
- Single Responsibility Principle (SRP)
- Separation of Concerns (SoC)
- Source Code Konventionen

Praktiken:

- Issue Tracking
- Automatisierte Integrationstests
- Lesen, Lesen, Lesen
- Reviews

Thank you!