

## Stufe 2: Oranger Grad von CCD

Lutz Prechelt

Institut für Informatik, Freie Universität Berlin

## Prinzipien:

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

## Praktiken:

- Limit Work-in-Progress (WIP)
- 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
  - [CIC] Kap. 3  
[CoCo] Kap. 7
- Wofür?
  - Evolvierbarkeit
- Hinderungsgrund:
  - Abstraktionsgrad ist eine unklare Größe
  - Führt bei strenger Auslegung zu vielen winzigen Methoden

# 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
    - Abstraktion für 1 oder mehrere Statements einführen
  - Extract Variable Refactoring
    - Mini-Abstraktion für einen komplexen (Teil)Ausdruck einführen

- 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 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*?)
  - [CIC] 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 *übertriebene* Scheu vor kleinen Klassen
  - IDE-Funktionen zum Browsen gut beherrschen

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

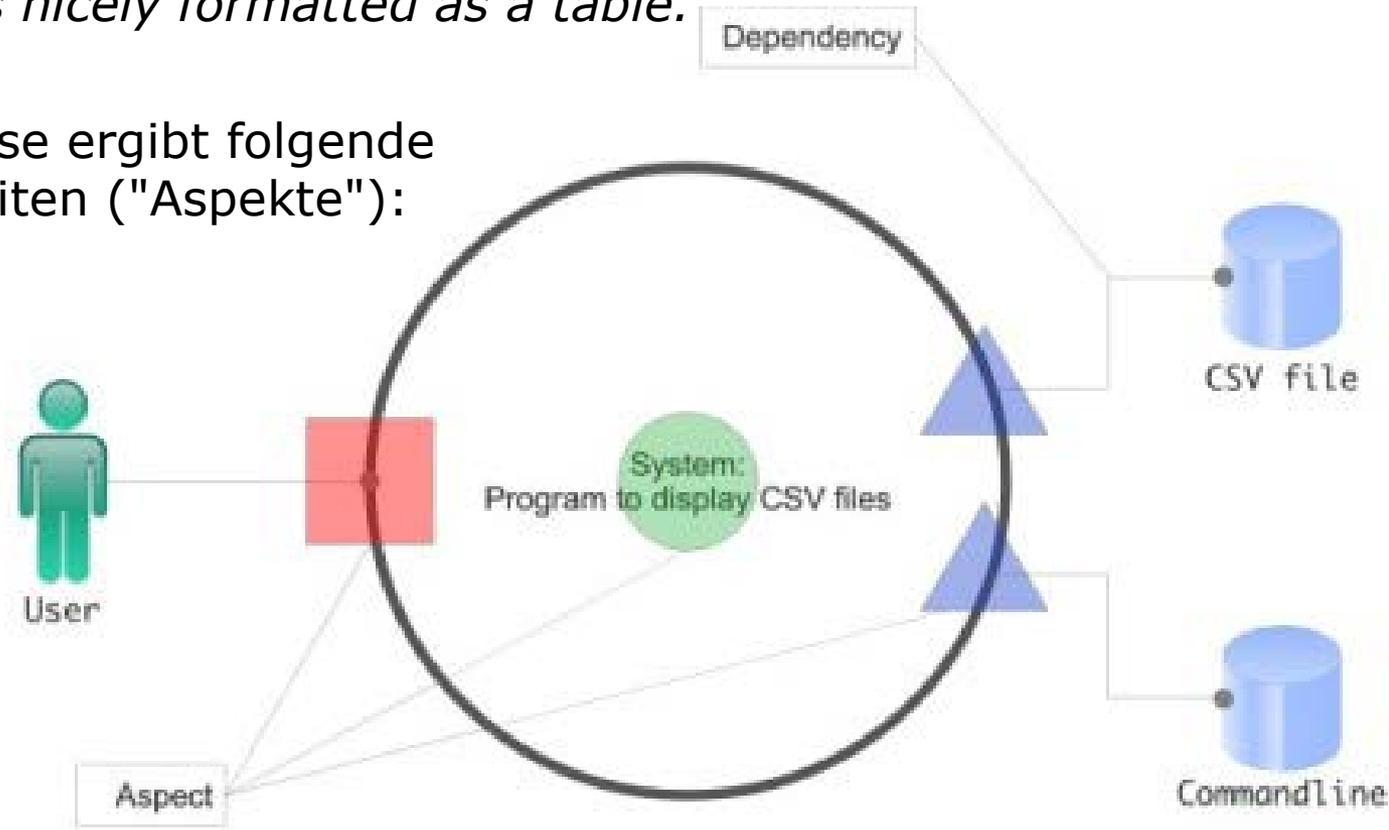
- SRP ist eine Daumenregel für sinnvoll aufgeteilte Modularität
- Sie ist auf mehreren Ebenen anwendbar:
  - Subsysteme
  - Module, Komponenten, Pakete
  - Klassen
  - Methoden
- D.h. z.B.: Mehrere Zuständigkeiten einer Klasse können OK sein, wenn sie klar auf Methoden aufgeteilt sind
  - aber genug 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
  - nachvollziehbare Konzepte
  - z.B. wie folgt:

# Single Responsibility Principle (SRP): Beispiel

- Beispiel von Ralph Westphal:

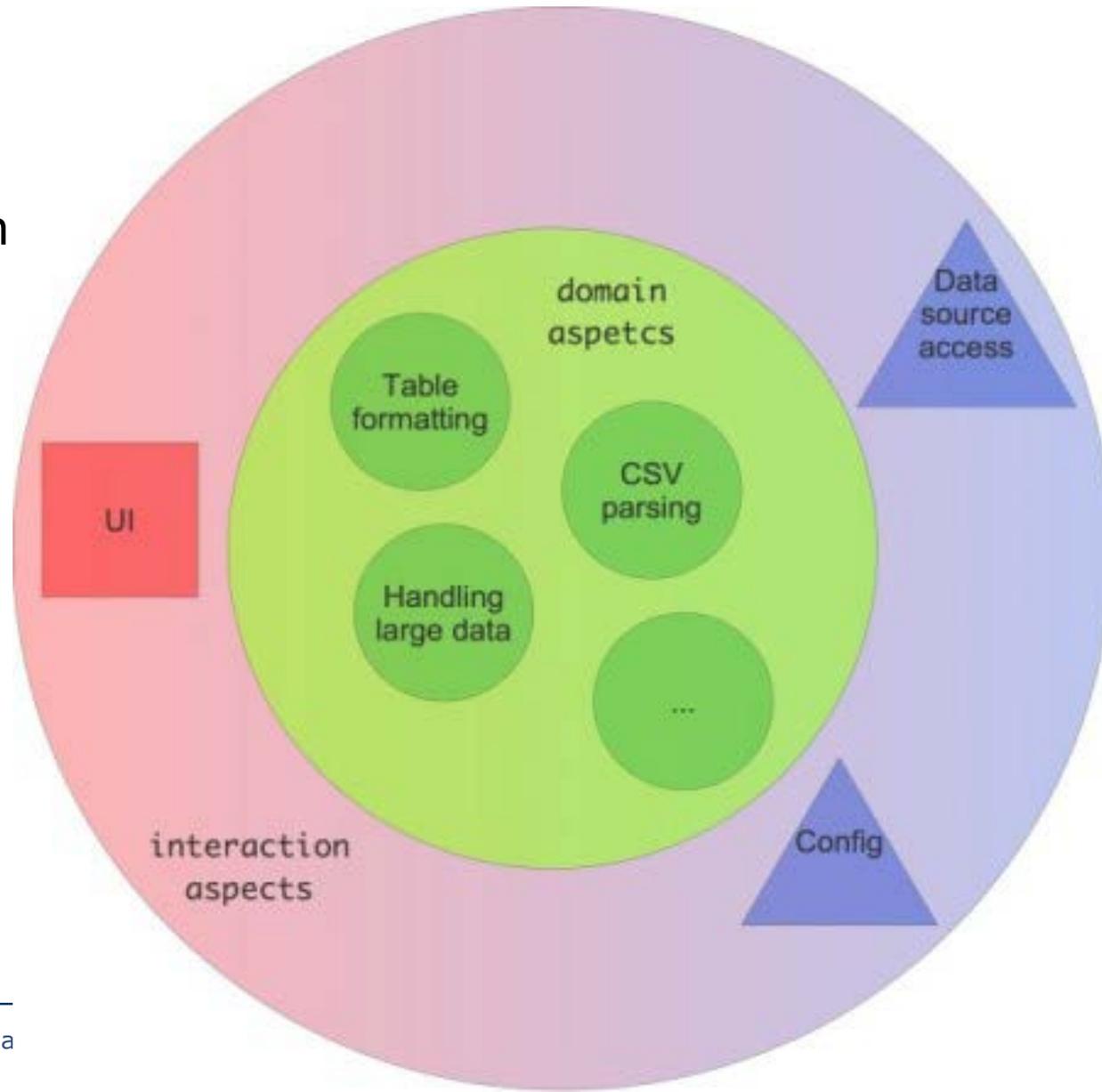
- Anforderungen sind: *"The program reads in CSV data files and displays them in a page-wise manner. The name of the file to display as well as the number of lines per page is passed in on the command line. Each page is nicely formatted as a table."*

- Grobe Analyse ergibt folgende Zuständigkeiten ("Aspekte"):



# Single Responsibility Principle (SRP): Beispiel (2)

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



- 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
  - **O**: weder A noch T
    - z.B. Behälterklassen, Standardalgorithmen
  - **AT**: unerwünscht!
  - **R**: Repräsentationsanpassungen
    - 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

- 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:
  - Bemerkten, 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): Geht durch alle Ebenen

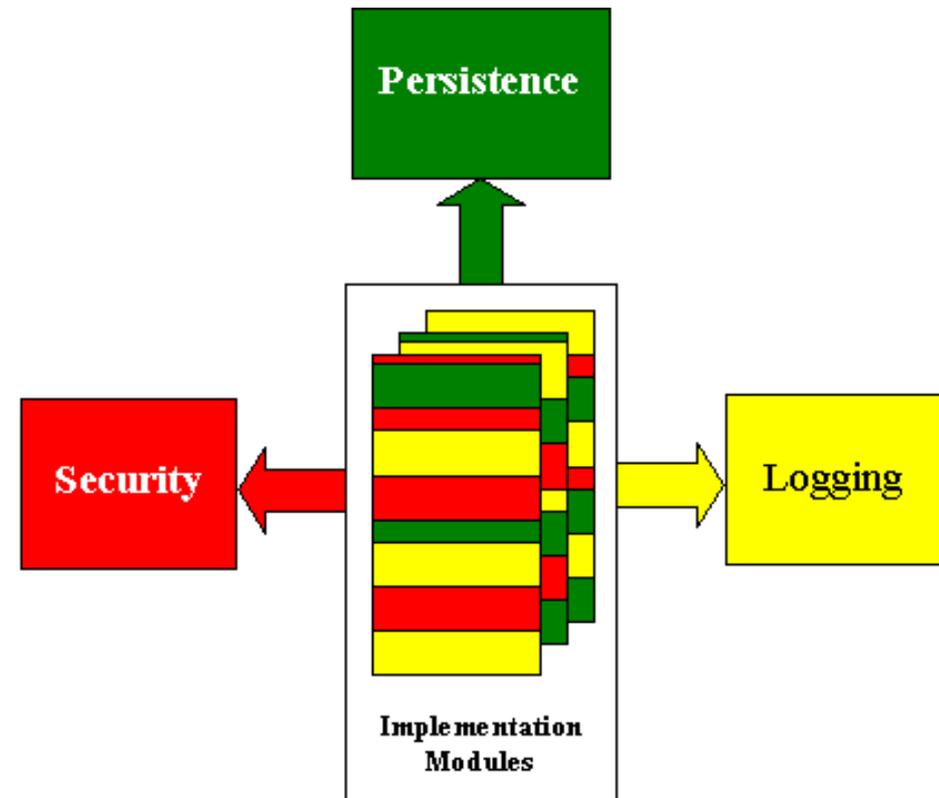
- Kann das ein sinnvolles Programm werden?:  
`constant1 = 3`  
`constant2 = 3`  
`numberOfThreads = 3`  
`afterJobWaitSeconds = 3`
- Jede Anforderung liefert einen Belang (oder mehrere)
  - Oft (z.B. bei nichtfunktionalen Anforderungen) relevant über das gesamte Produkt hinweg.
    - Das macht die Trennung schwierig!
- Bei Konstanten u.ä. fängt SoC an
  - feinstgranulare Belange
- Sie endet bei riesigen Teilprodukten
  - z.B. der SAP ERP-Suite.

# Separation of Concerns (SoC): AOP-Beispiel

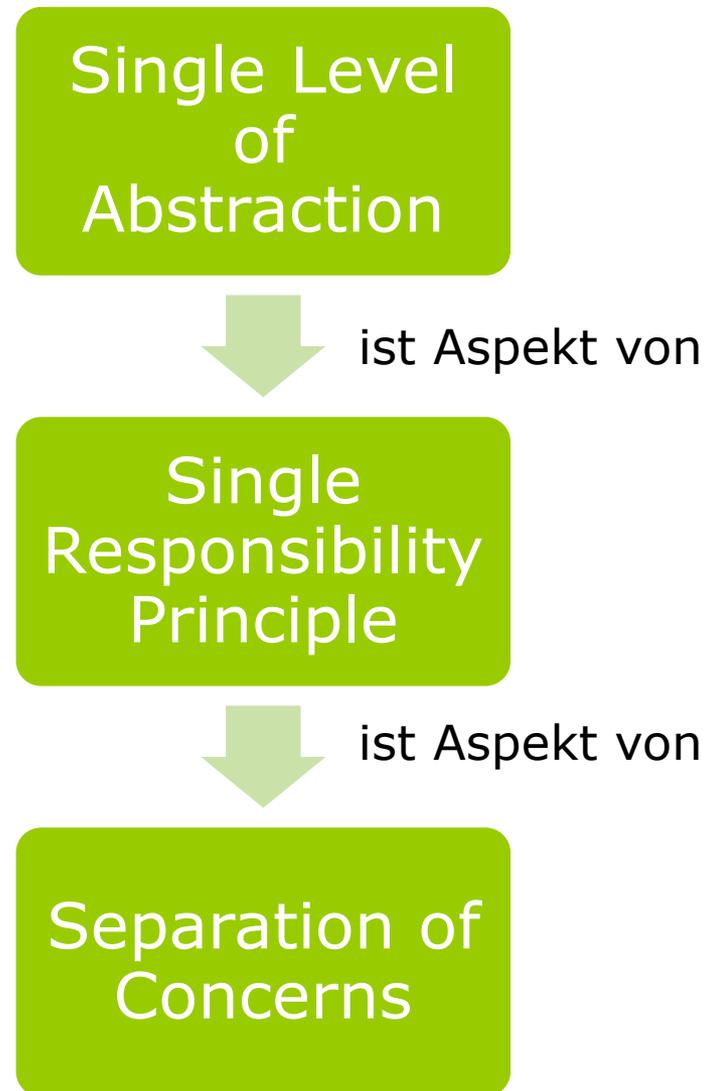


- Grundproblem siehe Bild
  - aus [AOP]
- Beispiele siehe [[AOP](#)]:
  - 1. [Teil 1, Seite 1](#):  
SomeBusinessClass
  - 2. [Teil 1, Seite 2](#):
    - CreditCardProcessor (Zielklasse)
    - Logger (Kern von Aspekt)
    - Beschreibungstext darunter (Weaving-Spezifikation)
    - CreditCardProcessorWithLogging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

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



# Separation of Concerns (SoC): Ein sehr allgemeines Prinzip



- 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)

- 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 oder wg. Geschwätzigkeit schwer zu entdecken.
- Hilfreiche Techniken:
  - Für viele Sprachen gibt es leidlich standardisierte Konventionen, z.B.
    - [Java](#)
    - [Python](#)
    - [Liste bei Wikipedia](#)
    - deren Übernahme spart viel Diskussionsaufwand ein
  - Namens- und Kommentarkonventionen siehe [CIC], Kap. 2 bis 4
  - Orientierung an gängiger Praxis
    - siehe z.B. github

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

- 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!
- Manche Leute glauben angeblich, 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 die Semantik, aber nicht den Zweck
  - Wichtige Entwurfsüberlegungen
    - "**Warum** so und nicht anders?"

## Prinzipien:

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

## Praktiken:

- Limit Work-in-Progress (WIP)
- Issue Tracking
- Automatisierte Integrationstests
- Lesen, Lesen, Lesen
- Reviews



- Agile SW-Entwicklung lebt von iterativer Verbesserung.
- Diese basiert auf Feedback.
- Um Feedback zu X zu bekommen, muss X einsatzfähig sein.
- Man bekommt also früher Feedback, wenn man X nicht halbfertig herumliegen lässt.
  - Halbfertige Dinge sind gebundenes Kapital (schon investierte Arbeit), das sich noch nicht verzinst (kein Nutzen, kein Feedback).
- Außerdem arbeitet man dann fokussierter.
- Deshalb:
  - Beschränke die Zahl von Dingen, die begonnen, aber noch nicht beendet sind
    - "work-in-progress"
- Zentrales Prinzip bei Lean SD und Kanban

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

- Phänomene ohne:
  - Defekte werden lange Zeit nicht bereinigt
  - Mäßig wichtige Defekte werden nicht vor unwichtigen bereinigt
  - Zwischenstände bei der Defektbereinigung gehen verloren
    - → Doppelarbeit
  - aktuelle Aufgaben werden nicht erledigt
    - oder doppelt
  - Kleine Funktionalitäten werden ganz vergessen
- Hilfreiche Techniken:
  - Story-Backlog
    - z.B. als Markdown-Datei
  - 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/dp/p/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 (gelingen)

## Methodenvorschlag für die Anforderungsebene:

- [http://www.jamesshore.com/Agile-Book/iteration\\_planning.html](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 Entwicklernamen
  - Angewandtes KISS!

- Reale Kartenwände
  - Bildersuche nach "[Agile Team wall](#)"

- 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. [Cypress](#), [Selenium](#), [Watir](#)
  - Also besser darunter bleiben
    - Das geht aber nur mit guter Modularisierung ausreichend!
  - Tests laufen evtl. lange
  - Tests können sporadisch ohne Grund fehlschlagen
  - Tests müssen gepflegt werden

- Phänomene ohne:
  - Angst vor Änderungen
    - Refaktorisierungen unterbleiben, Code verfällt!
  - → Weniger Spaß am Entwickeln und
  - Strengere Code-Eigentümerschaft
    - → unflexible Teamarbeit
  - Hoher manueller Testaufwand
  - Seltene Freigaben
- Hilfreiche Techniken:
  - Keine Geschäftslogik im GUI!
    - Sonst kann man nur sehr schwer ohne GUI testen
  - Klare, einfache Dienst-Schnittstellen überall
  - Spezifikation durch Beispiel ("Specification by Example") als Gewohnheit
    - kann viel Dokumentation ersetzen
    - [SpecByEx]
  - Ggü. externen Systemen: Teste Verhaltenskonstanz

# 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 (gelingen)

von SoundCloud:

- Teildienst "Buckster"
  - Bezahlungsfunktionen
- 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 *Kaufentscheidung*, ETFs
  - 1. Lies WP-Liste aus Exceldatei
  - 2. Hole aktuelle Kursdaten von Ariva
  - 3. Werte Kriterien aus
  - 4. Drucke Bericht m. Aktionen ggü. Vormonat
- Normalbenutzg ↔ Integr.test
  - Problem: *aktuelle* Kurse

- 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."
    - Die Hälfte wäre auch schon gut
- Wofür?
  - Produktionseffizienz
  - Reflexion
- Hinderungsgrund:
  - Keine Zeit!
    - Material, das man aktuell brauchen kann, darf man während der Arbeitszeit lesen (in Maßen).  
Ansonsten:
    - Schon bloßes Diagonallesen ist oft hilfreich: schafft Gewährsein ("awareness")
    - Interesse an der Materie auch in der Freizeit gehört zu unserem Beruf
  - Keine Lust!
    - Falschen Beruf gewählt?
    - Falschen Arbeitsplatz?

- 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 Technologiekrum im eigenen Dunstkreis
    - Suche nach einer wirklich *guten* Quelle
    - Durcharbeiten, nicht nur Lesen
    - Praktisch anwenden
  - Modus 3: Horizont erweitern
    - Gute Quellen empfehlen lassen
      - z.B. [TechRadar](#)
    - Themenbereiche
      - Anderer Technologiekrum
      - Methodenkrum und Grundlegendes



- 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!

- Phänomene ohne:
  - Hohe Defektdichten
  - Unnötig schwer änderbare Entwürfe
  - Nur schmales Wissen über die Codebasis
- 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](#), [GitHub PRs](#))



# 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.)

## 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

**Danke!**