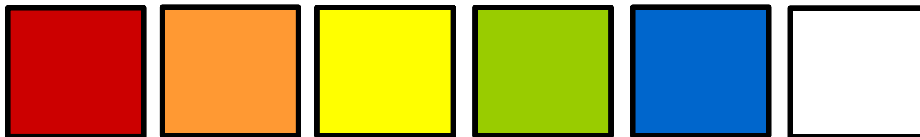


Stufe 5: Blauer Grad von CCD

Lutz Prechelt
Institut für Informatik, Freie Universität Berlin

Übersicht

Prinzipien:

- Entwurf und Implementation überlappen nicht
- Implementation spiegelt Entwurf
- You Ain't Gonna Need It (YAGNI)

Praktiken:

- Continuous Delivery
- Iterative Entwicklung
- Komponentenorientierung
- Test-First-Entwicklung

Entwurf u. Impl. überlappen nicht

- Was?
  - Entwurfsdokumente sollten möglichst wenig Information enthalten, die in der Implementation enthalten ist (und umgekehrt). DRY!
    - Gilt erst ab Beginn des Implementierens
  - Also: Entwurfsdokumente wegwerfen oder auf sehr grobem Niveau halten
    - Komponentenarchitektur
- Wofür?
  - Evolvierbarkeit
  - Produktionseffizienz
- Hinderungsgrund:
  - Ganz ohne Entwurfsdokumente ist das Leben anstrengend, aber
  - vorhandene Dokumente muss man meist erst mal verschlanken

Entwurf u. Impl. überlappen nicht

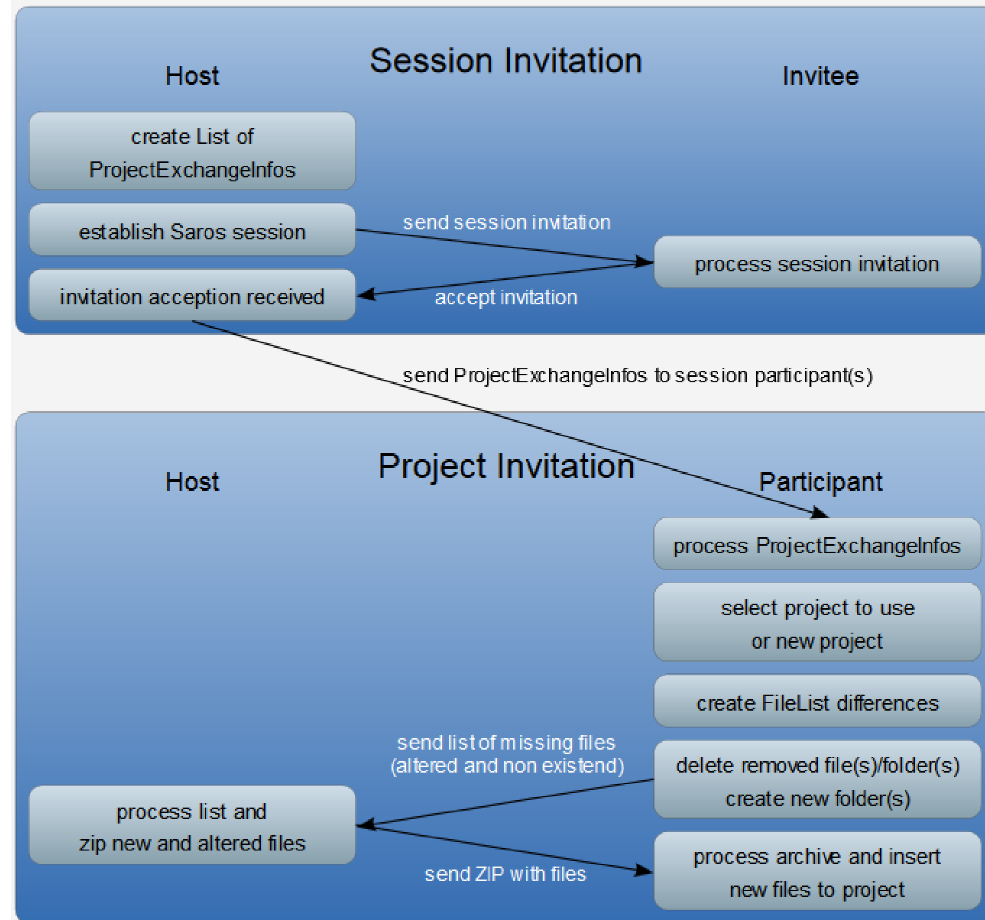
Freie Universität Berlin

- Wirkungen von Verletzungen:
  - Entwurfsdokumente veralten schnell
    - und sind dann oft falsch.
    - Gelegentlich passieren Defekte nur deshalb.
  - ➔ Entwickler nutzen sie nicht mehr oder viel Pflegeaufwand ist nötig
- Hilfreiche Techniken:
  - Grobkörnige Architektur definieren
    - Services oder
    - echte Komponenten
  - Schnittstellen, die Abhängigkeiten einschließen
  - Echte Komponenten-Zerlegung
    - mit deklarativen Abhängigkeiten
    - z.B. OSGi "Bundles"
      - Basis v. Eclipse
  - Echte REST-Dienste-Zerlegung
    - "REST Level 3"
      - d.h. inkl. Reflektion!

Entwurf u. Impl. überlappen nicht: Beispiel: Saros

Freie Universität Berlin

- Die Saros-Entwurfsdokumentation beschränkt sich weitgehend auf langlebige Fakten
  - wie gut das klappt ist aber nicht ganz klar
- Saros besteht aus mehreren Komponenten
  - diese Zerlegung ist aber nicht sehr weit getrieben



Implementation spiegelt Entwurf

- Was?
  - Die Strukturelemente der Architektur sollten sich in der *physischen(!)* Organisation des Codes wiederfinden
    - separate Codebasen
      - oder wenigstens Executables o.ä.
    - z.B. damit Architekturänderungen nicht aus Versehen passieren
- Wofür?
  - Evolvierbarkeit
- Hinderungsgrund:
  - Solange die Lösung noch klein ist und eine Trennung technisch nicht erzwungen wird, ist dies umständlich
    - und wird deshalb gern ausgelassen.
  - Und dann fängt die SW an zu wachsen...

Implementation spiegelt Entwurf

- Wirkung von Verletzungen:
  - Code ist schwieriger zu verstehen
    - insbes. f. Neueinsteiger
    - weil einzelne Codebasis umfangreicher
  - Architekturelle Entscheidungen lassen sich zu leicht ändern
    - und werden deshalb auch geändert, ohne expliziten Beschluss
    - Stichwort: Fehlende Fassaden, zu hohe Sichtbarkeit von Klassen

- Hilfreiche Techniken:
  - Komponenten
  - separierte Dienste (Microservices)
    - denn die sind beide in der SW-Struktur recht starr
  - in Java, im Kleinen: package-scope (statt public) für Klassen

Implementation spiegelt Entwurf: Beispiel: Saros

Wie vor:

- Saros besteht aus mehreren physischen Komponenten (separate Eclipse-Plugins)
  - diese Zerlegung ist aber nicht sehr weit getrieben.
  - Unterhalb der Komponenten-Ebene sind viele eigentlich architekturelle Entscheidungen prompt erodiert
    - und mussten schon mehrfach nach Entdecken mühsam wieder hergestellt werden.

You ain't gonna need it (YAGNI): "Du wirst es nicht brauchen"

- Was?
  - Baue extern sichtbare Funktionalität nur, wenn sie auch (schon) verlangt ist
  - Baue interne Funktionalität und Flexibilität nur, wenn sie
    - jetzt schon benötigt wird oder
    - garantiert nichts kostet (ob jetzt oder später)
- Wofür?
  - Evolvierbarkeit
  - Produktionseffizienz
- Hinderungsgrund:
  - Verlangt die Unterdrückung von Informatiker-Reflexen
    - Eleganz! Flexibilität! Abstraktion! Entwurfsmuster!
- Häufiges Missverständnis:
  - YAGNI zielt nicht auf das Weglassen von Entwurfssschritten, sondern
  - auf KISS im Produkt
    - aber nur unter Wahrung einer agilen Codestruktur!
  - Fowler [DesignDead]:
    - "you shouldn't add any code today which will only be used by [a] feature that is needed tomorrow"

You ain't gonna need it (YAGNI)

- Wirkung von Verletzungen:
  - Jetzt verlangte Funktionen werden später fertig
  - Entscheidungen werden unter größerer Unsicherheit gefällt
    - d.h. verfrüht
  - Die SW schleppt Komplexität mit sich herum, die gar nicht nötig wäre
- Hilfreiche Techniken:
  - KISS-Attitüde
    - aber unterscheide "simple" klar von "simplistic"
  - Beherrschung von Refactoring
  - Simple Design (XP1, p.57):
    1. Runs all the tests
    2. Contains no duplicated logic
    3. States every intention important to the programmers
    4. **Has the fewest possible classes and methods**

You ain't gonna need it (YAGNI): Genauere Definition

- Sergey Ignatchenko: YAGNI-C ("clarified")
  - für Entwicklung von Bibliotheken:
  1. Thinking ahead is good, implementing ahead is bad
  2. If nobody in the team can describe a very specific use case for a problem, the problem doesn't exist
  3. If in doubt how it should behave, *prohibit* that case

- Viel Diskussion auf c2.com

You ain't gonna need it (YAGNI): Negative Übertreibung (zu viel YAGNI)

Nochmal:

- Viele Leute missverstehen YAGNI so, dass es verboten sei, in die Zukunft vorauszu**denken**
  - Das führt zu sehr unklugen Entwürfen
  - YAGNI warnt nur vor dem **Implementieren** von Funktionen oder Mechanismen, die noch nicht gebraucht werden.
    - Und, wie überall:
      selbst dabei mag gelegentliche Missachtung sinnvoll sein
      (wenn man die Wahrscheinlichkeiten gut abschätzen kann)

You ain't gonna need it (YAGNI): Positivbeispiel (gelungen)

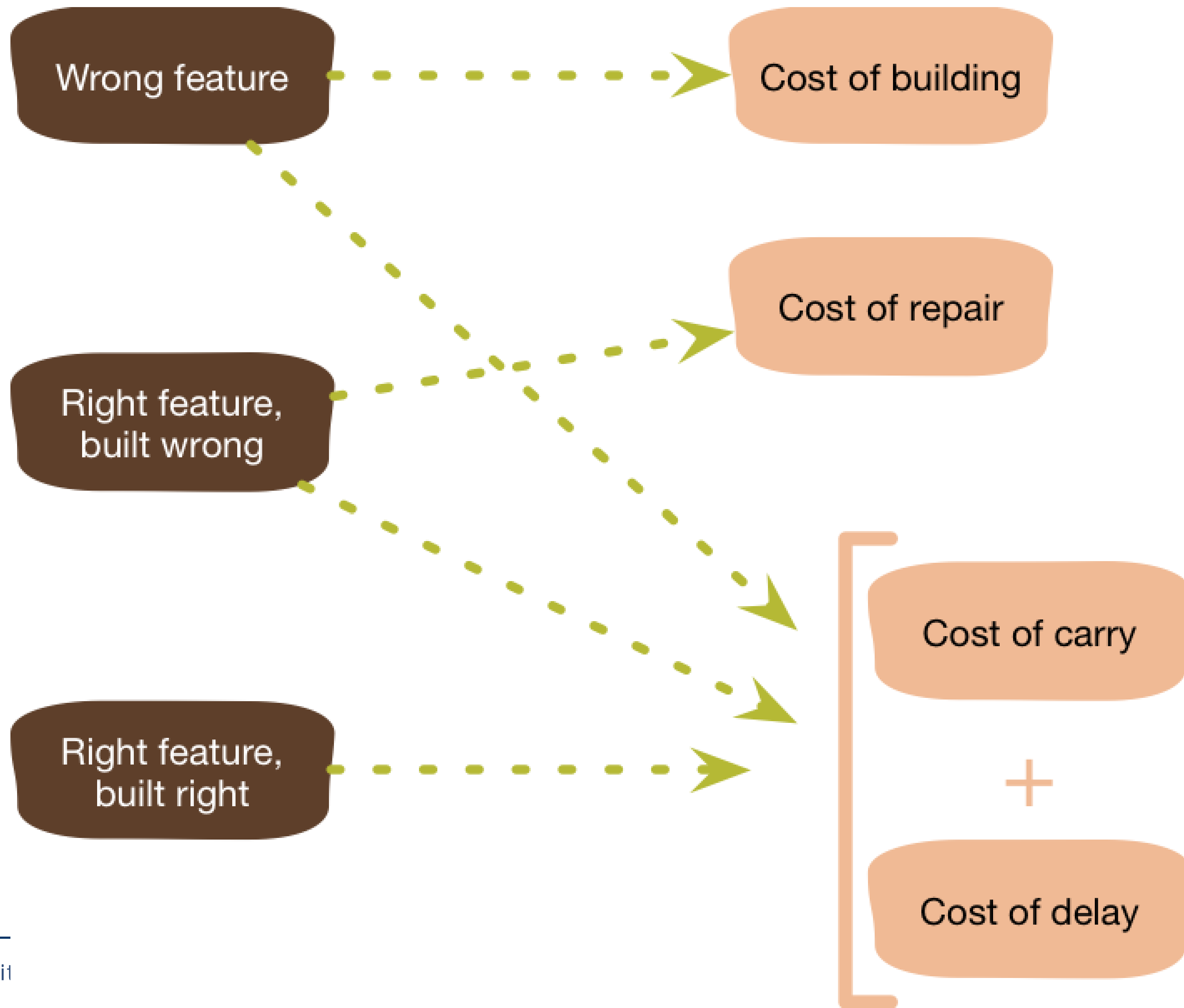
Minibeispiel von Jeremy Miller:

- Wenn die Webanwendung heute nur ein festes 2x2 Layout von Dashboard Panels braucht
  - und es keine Mobil-Anwender gibt
- dann baue nicht jetzt schon die Flexibilität für konfigurierbare Layouts ein
  - Und benutze ruhig erst mal <table> statt CSS, wenn Du CSS nicht genauso schnell kannst.

- C/C++
  - z.B. im Gegensatz zu Algol 68

You ain't gonna need it (YAGNI): Konsequenzen von zu wenig YAGNI

- martinfowler.com/bliki/Yagni.html



Übersicht

Prinzipien:

- Entwurf und Implementation überlappen nicht
- Implementation spiegelt Entwurf
- You Ain't Gonna Need It (YAGNI)

Praktiken:

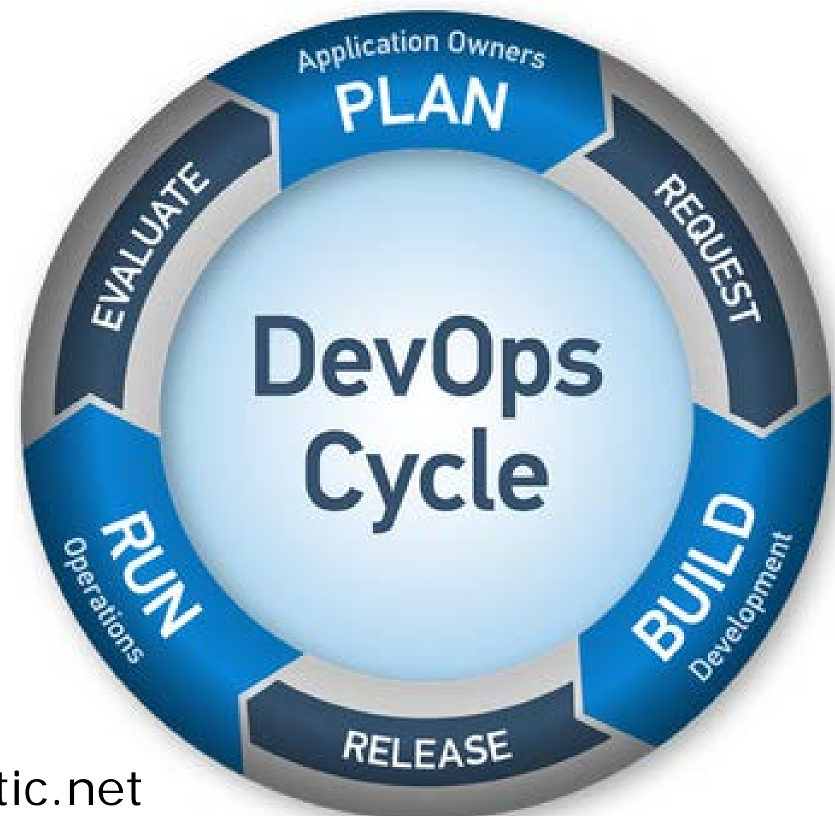
- Continuous Delivery
- Iterative Entwicklung
- Komponentenorientierung
- Test-First-Entwicklung

Continuous Delivery (CD)

- Was?
  - Automatisiere auch die Bereitstellung und Inbetriebnahme der SW ("Continuous Deployment")
  - und sei jederzeit in der Lage, Änderungen tatsächlich in Betrieb zu nehmen
- Wofür?
  - Produktionseffizienz
- Hinderungsgrund:
  - Verlangt kurze Entscheidungswege
    - Ermächtigung des Entwicklungsteams
  - Verlangt ein hohes Niveau automatisierter Tests
    - Wirksamkeit beim Defektefangen
  - Ist in manchen Domänen zu riskant

Continuous Delivery (CD)

- Wirkung von Verletzungen:
  - Der ganze Prozess wird erheblich schwerfälliger
- Hilfreiche Techniken:
  - Lean SD, Kanban, DevOps



networkstatic.net

Continuous Delivery (CD): Positivbeispiel (gelungen)

Modular Architecture

influence type

Empowered To Deploy

sociol.

psychol.

engineering

Held Responsible

Automated Deployment

Automated Tests

Direct Feedb.

Rapid Repair

Feel Responsible

Realistic Feedback

Quick Feedb.

Co-define Requirements

High Motivation

Frequent Deployments

Prechelt, Schmeisky, Zieris;
Studie bei SoundCloud u. Immob.Scout24

2

Continuous Delivery (CD): Negativbeispiel (zu wenig)

- Das dritte Team in obiger Studie hatte kein CD
  - Seine Motivation war wesentlich niedriger
  - und der ganze Entwicklungszyklus langsamer
    - kurzfristige Reaktionen (auf Geschäftsebene) waren dadurch unmöglich

Iterative Entwicklung

- Was?
  - Setze Anforderungen stets nur in kleinen Happen von max. 2-4 Wochen um
  - und verschaffe Dir dann Feedback
- Wofür?
  - Evolvierbarkeit
    - der Funktionalität!
  - Korrektheit
  - Produktionseffizienz
    - der *sinnvollen* Funktionalität!
  - Reflexion

- Hinderungsgrund:
  - Auftraggeber sind an so enger Zusammenarbeit evtl. wenig interessiert

<u>Die</u> Einstiegsdroge für agile Entwicklung

- alles andere folgt daraus.
  - Continuous Deployment ist quasi die Turboversion hiervon

Iterative Entwicklung

Freie Universität Berlin

- Wirkungen von Verletzungen:
  - Unnötige Funktionen werden gebaut
  - Nötige Funktionen werden in ungünstiger Form gebaut
    - oder ganz falsch verstanden
  - Änderungswünsche werden abgewiesen *oder* die Codebasis ist in schlechterem Zustand:
    - Es wird zu wenig getestet,
    - ungünstig abstrahiert,
    - zu wenig refaktorisiert.

- Hilfreiche Techniken:
  - Scrum:
    - Anforderungssammlung
      - "product backlog"
    - Iterationsplanung
      - → "sprint backlog"
      - striktes Time-Boxing
    - Präsentation des Produkts
      - "sprint review": Feedback
      - evtl. Inbetriebnahme
    - Reflektionsschritt
      - "restrospective"

Iterative Entwicklung: Interessantes Beispiel: Saros

Freie Universität Berlin

- Saros hat einige Zeit lang monatliche Releases gemacht,
- ist dann aber davon abgegangen, weil die Voraussetzungen ungünstig waren:
  - Zu wenig automatisierte Integrationstests
    - ist für Saros nämlich ziemlich schwierig

  - Dadurch aufwändiger Releaseprozess
  - Ungleichmäßiger Entwicklungsfortschritt durch unerfahrenes Personal
    - auch Time-Boxing will gelernt sein
- Aber: Saros hat überwiegend Junior-Entwickler und ca. 100% Fluktuation/Jahr
  - Dass es so gut funktioniert ist eine super Leistung.

Komponentenorientierung

- Was?
  - Zerlege Deine SW in grobe, separat freigebbare Komponenten
    - mit expliziten Verträgen,
    - um unabhängige Entwicklung zu fördern
- Wofür?
  - Evolvierbarkeit
  - Produktionseffizienz
- Hinderungsgrund:
  - Stabile Verträge zu finden ist nicht einfach
  - Separierung ist mühselig, wenn man nicht gleich so angefangen hat

Komponentenorientierung

- Wirkungen von Verletzungen:
  - In größeren Teams ist der Entwicklungsfortschritt viel langsamer
    - viel Koordinationsaufwand
  - Die Softwarestruktur verfällt schneller
    - weil neue Kopplungen zu spät bemerkt werden

- Hilfreiche Techniken:
  - Passende technische Formen
    - COM, DCOM, ActiveX, COM+
    - Enterprise JavaBeans
    - OSGI bundles
    - Mozilla XPCOM
    - technisch+organisatorisch separierte Services
      - z.B. "Microservices" [MicroSvc]
  - Dateiformate wie DLLs, .NET Assemblies, Java JARs

https://en.wikipedia.org/wiki/Component-based_software_engineering#Technologies

Komponentenorientierung: Beispiel: SoundCloud

- SoundCloud begann als monolithische Ruby-on-Rails-Anwendung
  - genannt "Mothership"
  - Anfangs war das zufriedenstellend
- Als Last (Skalierung!) und Team wuchsen, wurde diese Form zunehmend ungeeignet
  1. Inbetriebnahmen immer nur im Ganzen möglich
  2. Skalierung immer nur im Ganzen möglich
- Lösung: Microservices und Ereignissystem einführen
    - https://developers.soundcloud.com/blog/building-products-at-soundcloud-part-1-dealing-with-the-monolith
  - Keine Ergänzungen mehr am Mothership
  - Neue Funktionalität nur noch als separater Dienst
  - Bei Modifikationen am Mothership ggf. Dienst extrahieren.

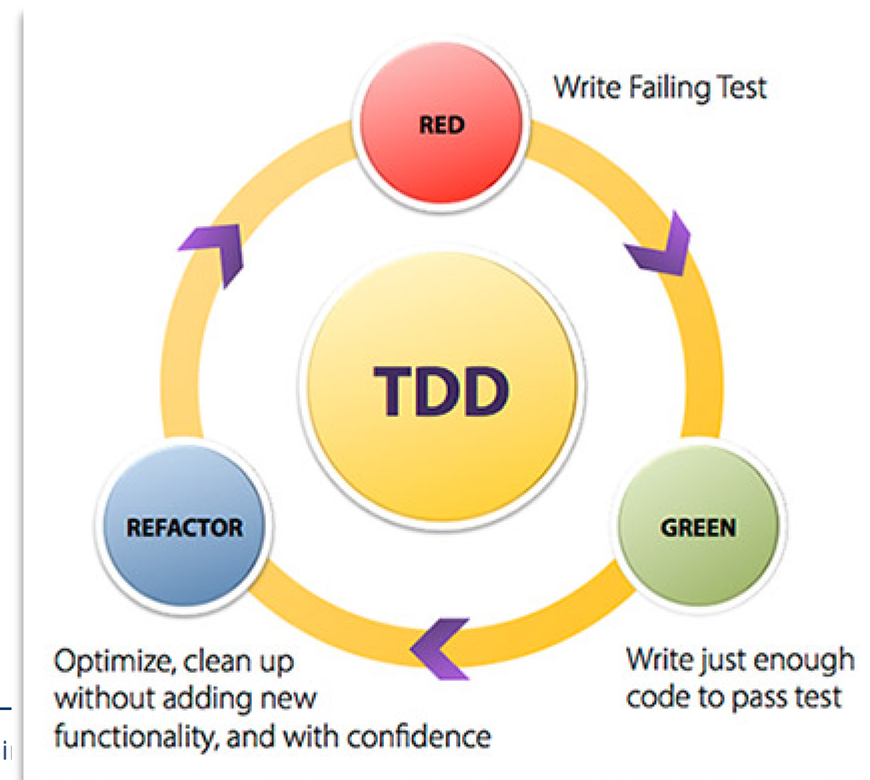
Test-First-Entwicklung

- Was?
  - Schreibe Deine Unittests und Akzeptanztests *vor* der Software, die sie testen,
    - um eine gut benutzbare und gut testbare API zu erhalten.
    - Akzeptanztest: Integrationstest aus Endbenutzerperspektive
- Wofür?
  - Evolvierbarkeit
  - Korrektheit
  - Produktionseffizienz
  - Reflexion
- Hinderungsgrund:
  - Erfordert oft mehrfaches Refactoring von Unittests
    - scheint umständlich
  - Insbesondere, wenn man APIs einsetzt, deren Konzepte man noch nicht versteht

Test-First-Entwicklung

- Wirkungen von Verletzungen:
  - Testen wird viel mühsamer
    - wg. schlechterer Testbarkeit
  - Debugging dauert oft länger
    - weil man meist mehr Code schreibt, bevor man ihn testet → mehr mögliche Defektstellen
    - weil man kaum Unittests, sondern nur Integrations-tests hat → mehr mögliche Defektstellen
  - Anwenden der Klassen wird evtl. mühsamer
    - wg. schlechterer Benutzbarkeit

- Hilfreiche Techniken:
  - ATDD, Storytests
    - und outside-in-Testreihenfolge
  - TDD
    - red, green, refactor



Test-First-Entwicklung: Positivbeispiel (gelungen)

- !!!

Übersicht

Prinzipien:

- Entwurf und Implementation überlappen nicht
- Implementation spiegelt Entwurf
- You Ain't Gonna Need It (YAGNI)

Praktiken:

- Continuous Delivery
- Iterative Entwicklung
- Komponentenorientierung
- Test-First-Entwicklung

Thank you!