

Stufe 5: Blauer Grad von CCD

Lutz Prechelt

Institut für Informatik, Freie Universität Berlin

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

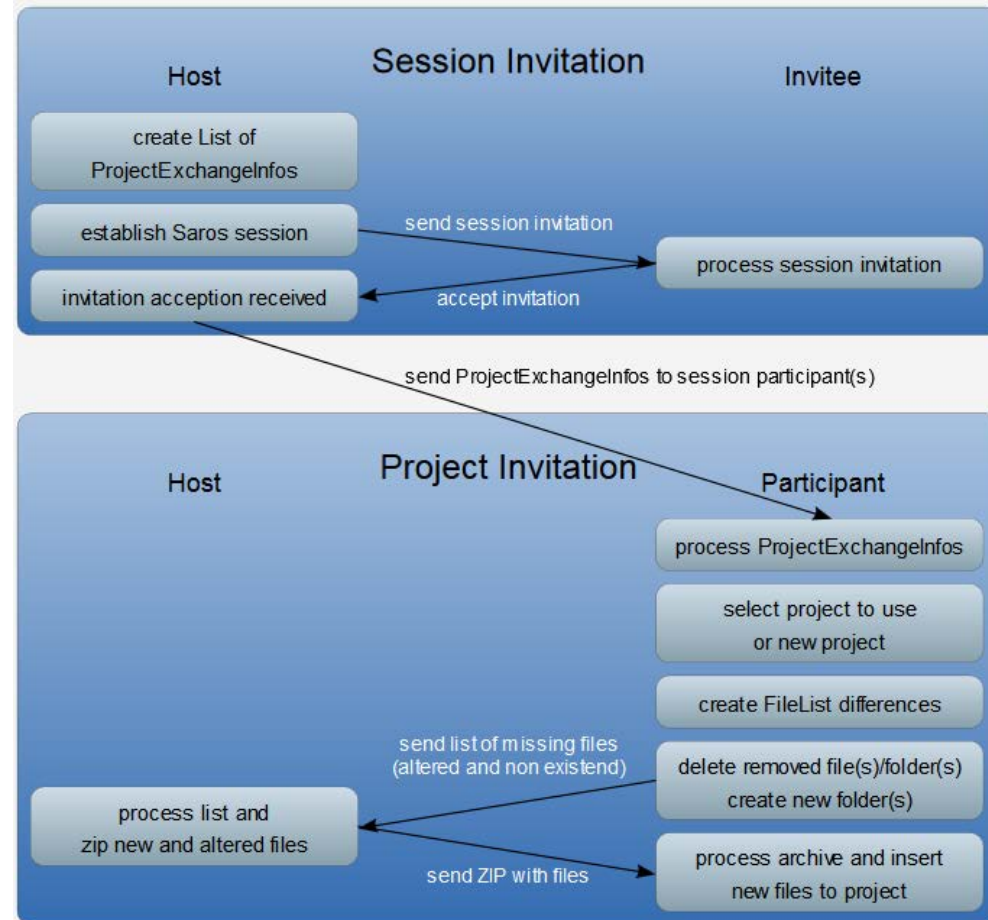
- Was?
 - Entwurfsdokumente sollten möglichst wenig Information enthalten, die in der Implementation enthalten ist (und umgekehrt). DRY!
 - Gilt erst ab Beginn des Implementierens
 - Also: Entwurfsdokumente wegwerfen oder auf sehr grobem Niveau halten
 - Komponentenarchitektur
- Wofür?
 - Evolvierbarkeit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

- 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

- 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 Plugins etc.)
 - 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 Entwurfsschritten, sondern
 - auf KISS im Produkt
 - aber nur unter Wahrung einer agilen Codestructur!
 - 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 verfrüht gefällt
 - also unter größerer Unsicherheit
 - 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 vorauszudenken
 - 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)
 - z.B. vermutlich dann, wenn zu einer komplexen Methode A die duale oder inverse komplexe Methode A' eigentlich erst später gebraucht wird.
A' sofort zu schreiben ist aber viel einfacher als später.

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

Minibeispiel von Jeremy Miller:

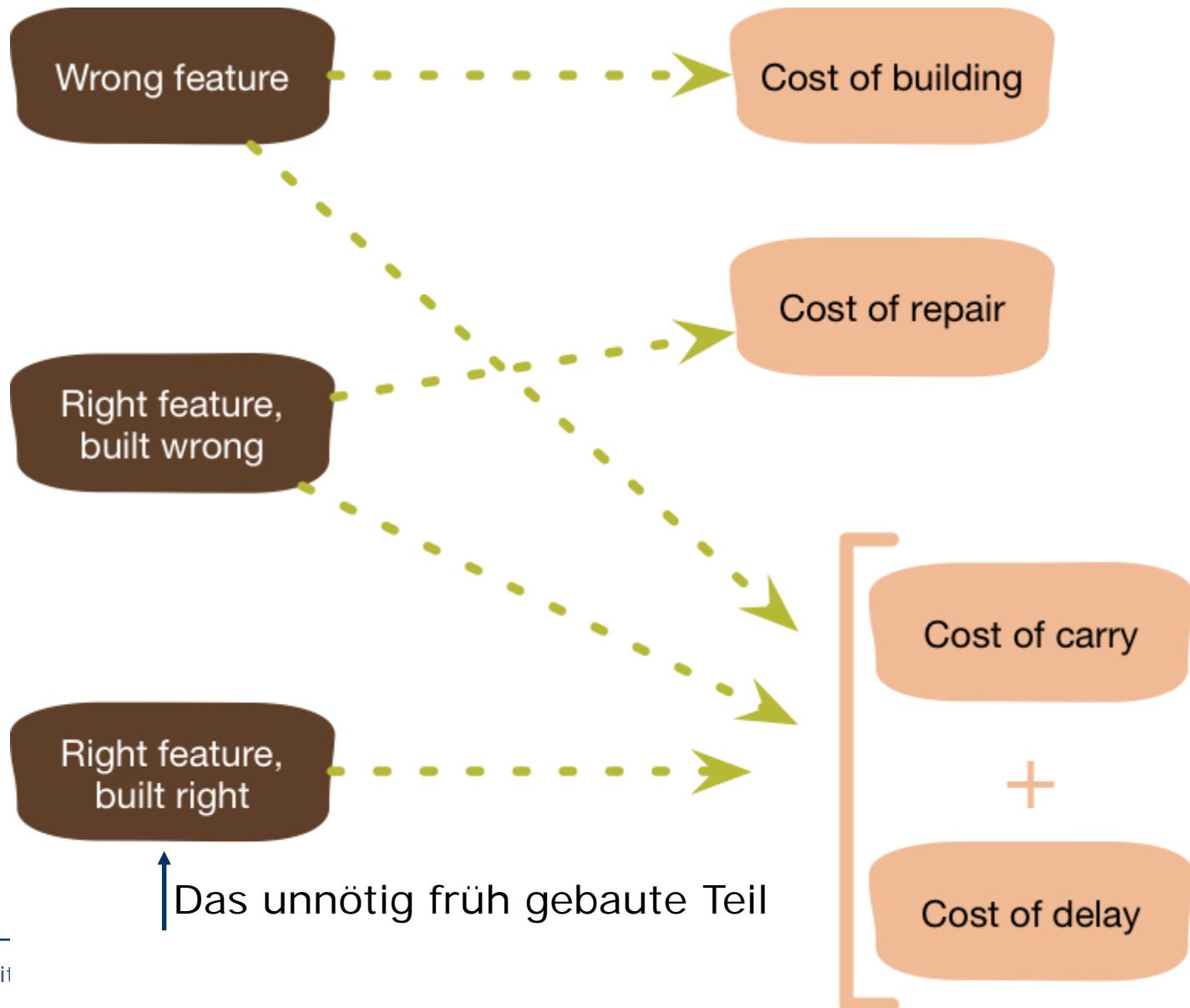
- Wenn die Webanwendung heute nur ein festes 2x2 Layout von Dashboard Panels braucht
 - und es keine Mobil-Anwender gibt
- dann baue nicht jetzt schon die Flexibilität für konfigurierbare Layouts ein
 - Und benutze auch ruhig erst mal `<table>` statt CSS, wenn Du CSS nicht genauso schnell kannst.

Programmiersprachen-Beispiel:

- Java (mit viel Evolution)
 - z.B. im Gegensatz zu Algol 68 (Entwurf in einem Rutsch)

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

- martinfowler.com/bliki/Yagni.html
- yagni.net/2015/04/15/bad-things-about-good-design/



Prinzipien:

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

Praktiken:

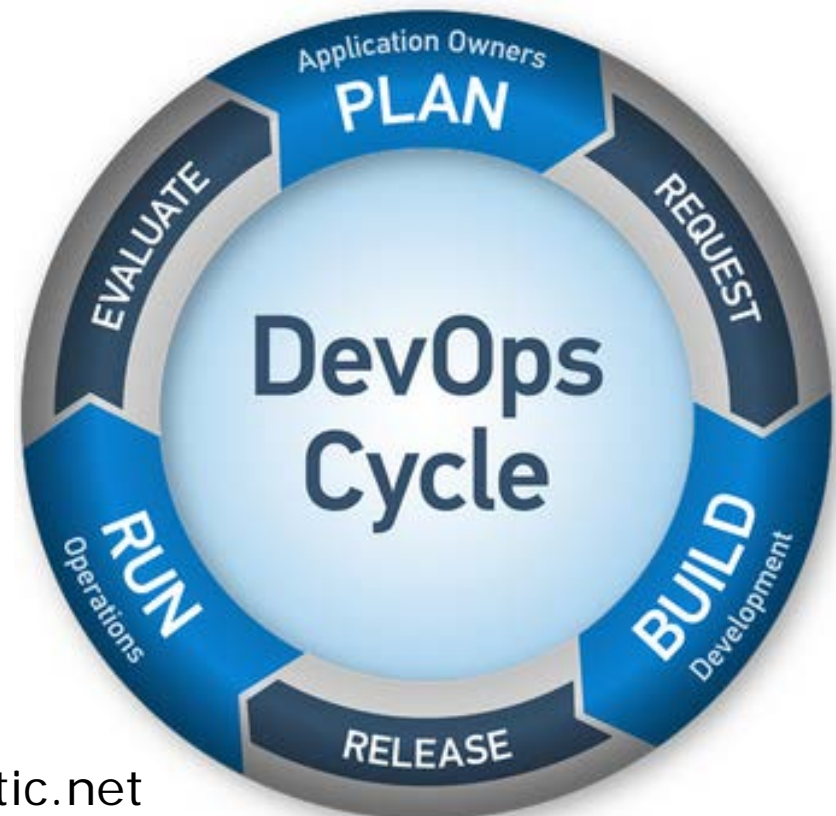
- Continuous Delivery
- Iterative Entwicklung
- Komponentenorientierung
- Test-First-Entwicklung

Continuous Delivery (CD)

- Was?
 - Automatisiere auch die Bereitstellung und Inbetriebnahme der SW ("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
 - mit viel Wirksamkeit beim Defektfangen
 - Ist in manchen Domänen zu riskant
 - das gilt noch viel mehr für "Continuous Deployment", wo die Inbetriebnahme sogar *immer sofort* passiert.

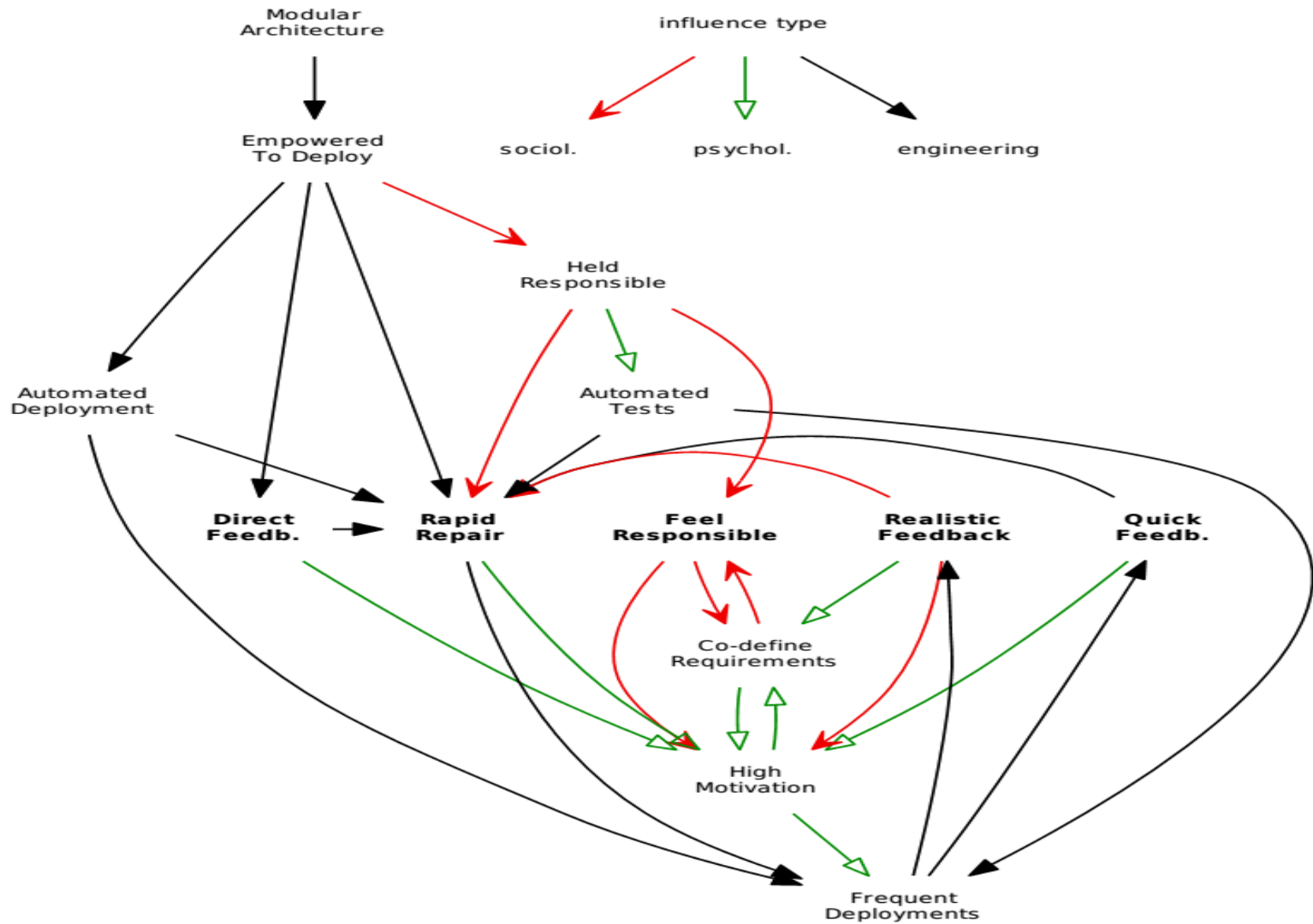
Continuous Delivery (CD)

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



networkstatic.net

Continuous Delivery (CD): Positivbeispiel (gelingen)



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

Continuous Delivery (CD): Negativbeispiel (zu wenig)

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

- 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

Die Einstiegsdroge für agile Entwicklung

- warum das erst hier im Blauen Grad kommt?
Weil es ohne automatische Tests und CI nicht geht!
- alles andere folgt daraus.
 - Continuous Deployment ist quasi die Turboversion hiervon

- Wirkungen von Verletzungen:
 - Unnötige Funktionen werden gebaut
 - Nötige Funktionen werden in ungünstiger Form gebaut
 - oder ganz falsch verstanden
 - Änderungswünsche werden abgewiesen und/oder die Codebasis ist in schlechterem Zustand:
 - Es wird zu wenig getestet,
 - ungünstig abstrahiert,
 - zu wenig refaktoriert.
- Hilfreiche Techniken:
 - Scrum:
 - Anforderungssammlung
 - "product backlog" (wird ständig gepflegt)
 - Iterationsplanung
 - → "sprint backlog"
 - striktes Time-Boxing
 - Präsentation des Produkts
 - "sprint review": Feedback
 - evtl. Inbetriebnahme
 - Reflektionsschritt
 - "retrospective"

Iterative Entwicklung: Interessantes Beispiel: Saros



- Saros hat einige Zeit lang monatliche Releases gemacht,
- ist dann aber wegen schlechter Voraussetzungen davon abgegangen:
 - Zu wenig automatisierte Integrationstests
 - ist für Saros nämlich ziemlich schwierig
 - Dadurch aufwändiger Releaseprozess
 - 4 Spezialrollen: Release-Manager + Vize, Test-Manager + Vize.
 - Viel manuelles Testen
 - 1 Woche Arbeit
- 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.

- 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
 - Vertragsänderungen werden umständlicher

- Wirkungen von Verletzungen:
 - In größeren Teams ist der Entwicklungsfortschritt 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](#)
 - Dateiformate wie DLLs, .NET Assemblies, Java JARs
 - Technisch+organisatorisch separierte "Microservices"
 - [\[MicroSvc\]](#)

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



- 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.

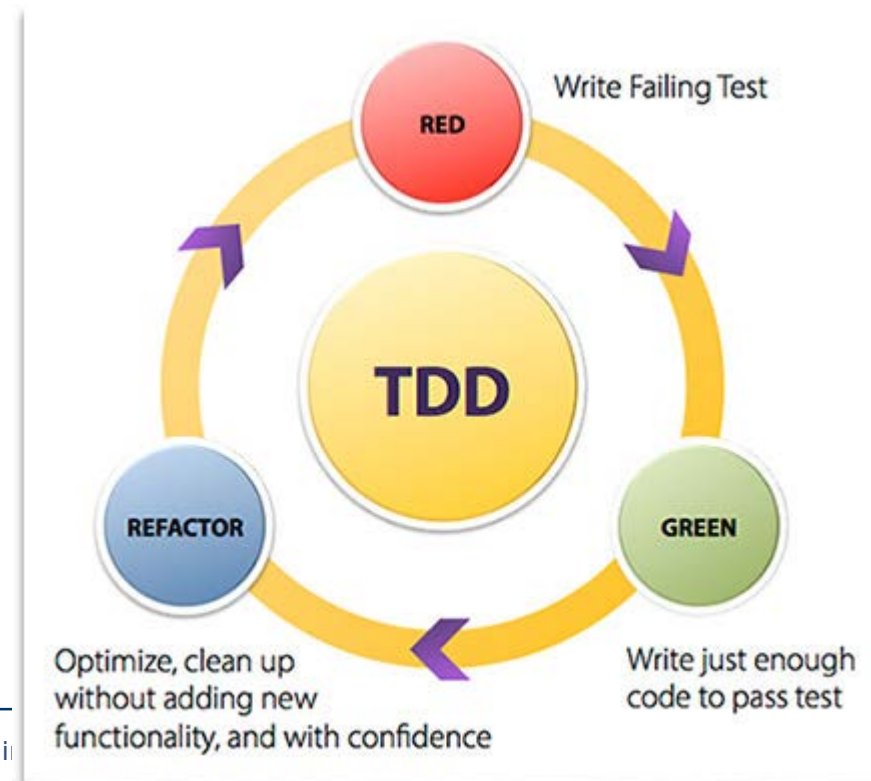
Test-First-Entwicklung

- Was?
 - Schreibe Deine Unittests und Akzeptanztests *vor* der Software, die sie testen,
 - um eine gut benutzbare und gut testbare API zu erhalten.
 - Top-Down (Outside-In), nicht Bottom-Up
 - Akzeptanztest: Integrationstest aus Endbenutzerperspektive
- Wofür?
 - Evolvierbarkeit
 - Korrektheit
 - Produktionseffizienz
 - Reflexion
- Hinderungsgrund:
 - Erfordert oft mehrfaches Refactoring auch von Tests
 - scheint umständlich
 - und ist es oft auch
 - (nicht immer!)
 - Langwierig, wenn man APIs definiert, deren Konzepte man noch nicht gut versteht

- Wirkungen von Verletzungen:
 - Testen wird mühsamer
 - wg. schlechterer Testbarkeit
 - Debugging dauert oft länger
 - weil man meist mehr Code schreibt, bevor man ihn testet → mehr mögliche Defektstellen
 - weil man weniger Unittests, sondern eher 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



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

Danke!