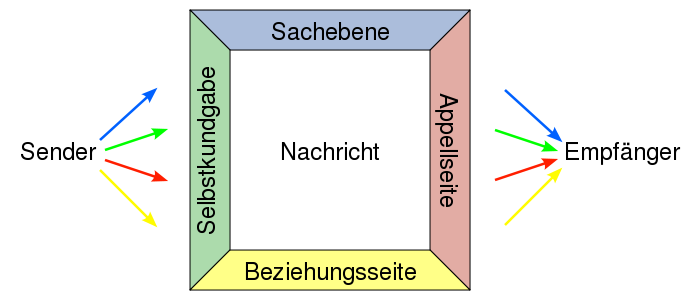
**Zusammenfassung**

**Moderne Software Techniken**

**Kommunikation**



Modell von Friedemann Schulz von Thun

Sach -> Worüber ich informiere

Selbst -> Was ich kundgebe (über mich)

Beziehung -> Wie wir zueinander stehen

Appell -> Wozu ich veranlassen möchte

Problem -> Einseitige Wahrnehmung:

Nur hören/senden **eines** Aspekts

Weitere Aspekte:

Explizite / Implizite Botschaft

Nonverbale Kommunikation (Gestik + Mimik)

Man kann **nicht nicht** kommunizieren

„Killerphrasen“

„Das haben wir schon immer so gemacht“

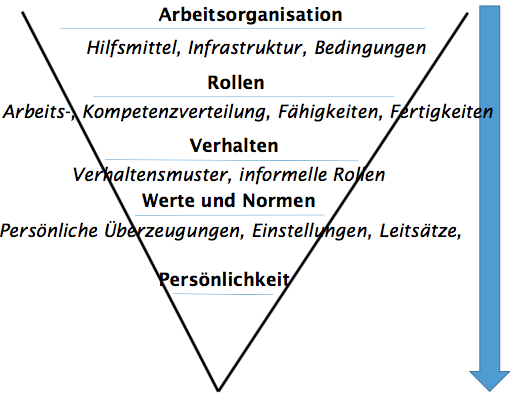
„Da haben Sie keine Ahnung“

„Das ist doch gar nicht machbar“

Konflikt**lösung**smöglichkeiten nach

* Streitgegenständen
* Erscheinungsformen
* Eigenschaften der Konfliktparteien

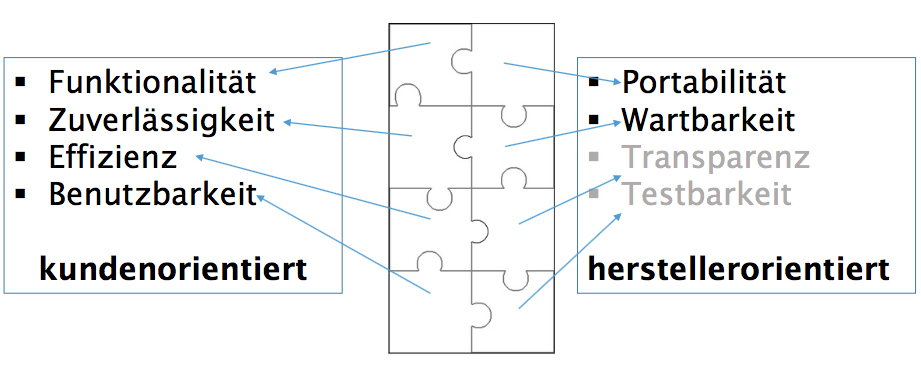
**Schichtenmodel** zur Bearbeitung von Konflikten

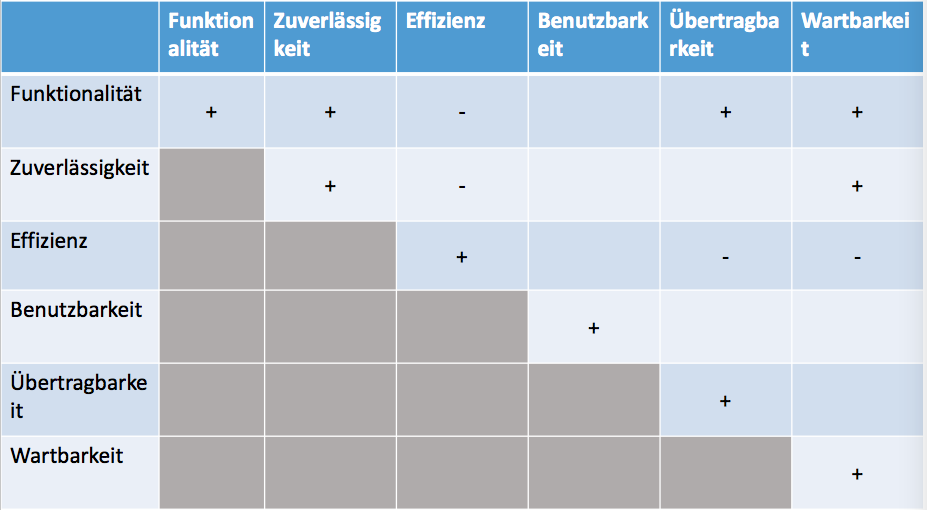


**Software Qualität**

Definition nach **DIN-ISO 9126**:

Software Qualität ist die Gesamtheit der Merkmale und Merkmalswerte eines Software Produkts, die sich auf dessen Eignung beziehen, festgelegte Erfordernisse zu erfüllen.

****



**Korrelation** von Qualitätskriterien:

**Effizienz**

Negative Korrelation mit vielen anderen Bereichen

* Mit Bedacht optimieren

**Benutzbarkeit**

Keine Korrelation

* Benutzerfreundliche Programme ohne beeinträchtigung möglich! ☺

**Gründe** für **schlechte** SW **Qualität**

Wachsende **Komplexität**

**Vollständige** Tests praktisch **unmöglich**

Produktlebensdauer **>>** Projektdauer

Möglichkeiten zur **Qualitätssicherung**

**Produkt**qualität

SW Richtlinien

Dokumentation

SW **Test** / Analyse / Verifikation

**Prozess**qualität

Konfigurationsmanagement

Build / Test **Automatisierung**

Vorgehensmodelle

Reifegradmodelle

Fehler**quellen**

**Lexikalisch/syntaktische**

* **Nicht(!) Compiler** Fehler
* MangelndesSprach**verständnis**
* Bsp.: c = a---b -> c = (a--) – b ODER c = a – (--b)
* **Greedy** Strategy: Zeichenstrom -> Neues Symbol, wenn nicht mehr vergrößert werden kann

**Semantische**

* Fehlerbehandlung schwer zu testen
* Einheitenumrechnungstabelle (Fehlinterpretation)
* Vermeidung über Standards (MISRA-C)

**Parallelität**

* Ausführung höher **priorisierter** Tasks in falschem Zyklus
* Prioritätsinversion
  + Niedriger Prio Task lockt Ressource
  + Hoher Prio Task muss warten -> Umkehrung der Priorität
    - Lsg.: Prioritäts**vererbung**

**Numerisch**

* **Umrechnung**sfehler

**Portabilität**

* Zahlenbereich zu klein bei **neuer** Anwendung

**Optimierung**

* Spät in Entwicklung
* Fehler **einbauen** beim Versuch zu optimieren (Laufzeit, Platz usw.)

**Spezifikation**

* Fehlerhafte Formel(n)

Fehler**bewertung**

Sichtweisen

* **Anwender**
  + Abwesenheit von Defekten
* **Entwickler**
  + Aufdecken von Defekten

**Konstruktive** Qualitätssicherung

SW **Richtlinien** (mehr als Syntaktik + Semantik)

Notations-

**-konvention**

Bezeichner, Einrückung, Leerzeichen, Kommentare, ...

-**stile**

camelCase, PascalCase, upper-/lowercase

Coding**sytle** pro Programmiersprache

Sprach**konvention**

MISRA-C

Oder in Firmen / Projekten -> **bestehende** verwenden!

Gefahr: **Soziale** Aspekte bei der Umsetzung (Angriff auf Person)

Behebung: **Gemeinsames** Entwickeln der Richtlinien, **Tool**gestützte Prüfung

**Typisierung**

Gleich**artige** Objekte zusammenfassen zu einem Daten**typ**

Erkennen von Typverletzungen, Umwandlungen zw. Typen

**Statisch** -> Zur Compilezeit

**Dynamisch** -> Bei Ausführung (Kombination beider möglich!)

Performanceverlust ☹

Starke: **Umwandlung** erlaubt (casts)

Schwache: Zugriff nur **typkonform**

**Generische Schnittstellen**

Strenge Typsicherheit nicht gewünscht (bsp.: Treiber)

Erhörung der Sicherheit durch **Container** Typen (Variant in C)

**Design by Contract**

Definition von **formalen, messbaren** Vereinbarungen für Schnittstellen zw. Modulen

* Überprüfung der Vereinbarung, Festlegung vor Implementierung

Vertrag zwischen **Aufrufer** und **Aufgerufenem** mit **Vor-** und **Nach**bedingungen sowie **Invarianten**

Vorbedingung

Zusicherung die **Aufrufer** beachten muss  
Nachbedingung

Zusicherung die **Aufgerufener** beachten muss

Invarianten

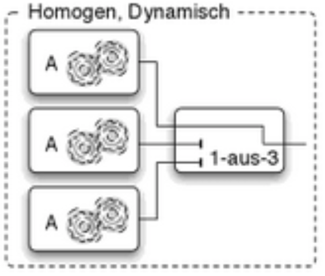
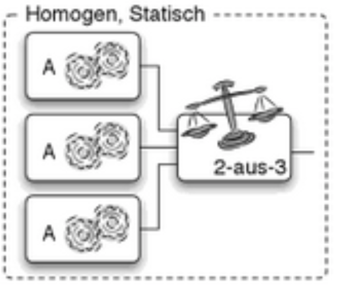
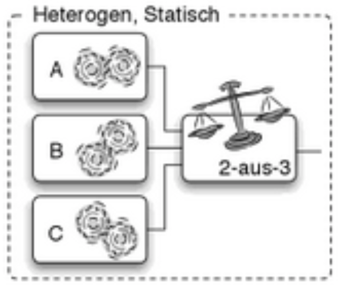
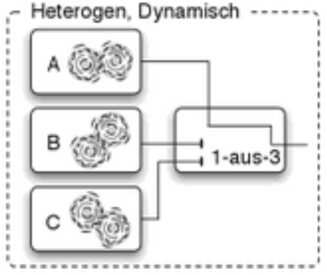
Gesundheitszustand der Klasse

-> Logische Aussagen für alle Objekte der Klasse die für gesamten Lebenszyklus gelten

**Fehlertolerante** Programmierung

Software Redundanz

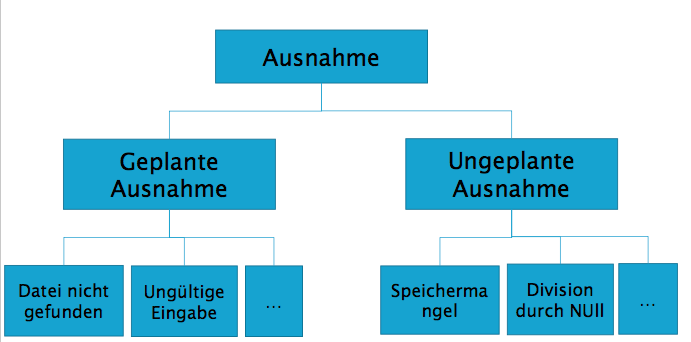
* Funktional -> Funktionserweiterung zur **Absicherung**
* Informationell -> Anreicherung Nutzen um **Zusatzinfos**
* Temporale -> Zeitanforderung **über**erfüllen -> evtl. **erneute** Ausführung
* Strukturelle -> **Mehrfach**auslegung von Komponenten
  + Homogene: n Komponenten **gleicher** Bauart
  + Heterogene: n Komponenten **unterschiedlicher** Bauart
  + Statische: **Alle** aktiv, **Voter** vergleicht Ergbnisse
  + Dynamische: **Ersatz**komponenten, Aktivierung bei **Bedarf**



**Selbstüberwachte Systeme**

* Fail Safe Reaktion: Wechsel in **sicheren** Zustand (Notbremse Aufzug)
* Selbstreparatur: Selbst**diagnose** und Reparatur (autom. Datenwiederherstellung)
* Reaktivierung: Watchdog (Explorer in Windows)

**Ausnahmebehandlung**

* Client Errors: **Benutzer**fehler
  + Aktion abbrechen
  + Ressource schließen
  + Speicher freigeben
  + Benutzer informieren
  + Error **loggen**
* Internal Errors: **Programmier**fehler
  + Wie Client Error
  + **+ Nachricht an Admin**
* Service Errors: Zugriff auf **Ressourcen**
  + Wie Internal Error
  + **+ Entwickler informieren**

Entweder **Delegation** der Ausnahmebehandlung an **OS**

Oder **Behandlung** der Ausnahme in **Programmiersprache**

**Strukturierte** Ausnahmebehandlung

**Trennung** von **Ausnahme**behandlung von **Anwendung**scode

* Loggen nahe Ursache
* Ggf. an Aufrufer weiterreichen
* Behandlung an angemessener Stelle
  + Benutzer zu erneuter Eingabe auffordern
  + Datei schließen
  + Programm beenden, ...

Pros: Trennung Codes, Exceptions nicht ignorieren, Fehler hochpropagieren, Gruppierung / Differenzierung

Exception Handling

* Gut
  + Einfachere Entwicklung/Wartung
  + Weniger Bugs
* Schlecht
  + Verwirrung der Benutzer
  + Schwere Wartung

Berücksichtigung im **Design**!

Exception **chaining**

Speicherung **ursprünglicher** Fehlermeldung in **neuer** Meldung

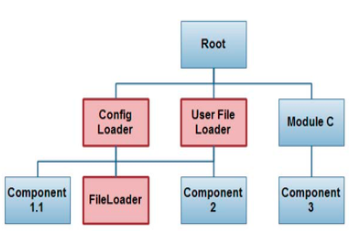
Best Practices

* **Spezifische** Exceptions verwenden
* Erst fangen, wenn **ordentlich** Behandlung möglich ist
* Ressourcen im **finally** Block schließen
* **Alle** Exceptions **einmalig** loggen
* **Dokumentation** Exceptions
* **Nicht** für **Kontroll**fluss verwenden
* **Keine** Exceptions **ignorieren**

Exception Handling Strategie

Anforderungen

* Primär
  + **Überleben** der App
    - App läuft weiter
    - Ressourcen sauber schließen
  + Info der **relevanten** Stellen
    - Unterschiedlich für Benutzer, Admin, Betreiber, Entwickler
  + Diagnose
    - **Wo** aufgetreten?
    - **Kontext** des Aufrufs
    - Beschreibung + Variablen**werte**
  + Reproduktion
    - Welcher **Weg** führt zu Fehler?
* Sekundär
  + Abstraktion
    - Verberge **unnötige** Infos für **höhere** Schichten
  + Les- und wartbarer Code

Error

* Location
  + Wo ist Fehler aufgetreten?
* Context
  + Ausführungspfad 🡪
  + (Hat Auswirkung auf Behandlung)

**Strategie** Elemente

* Fehler entdecken
* Infos sammeln
* Exception werfen + proparieren (+ Kontextinfos)
* Fangen + Reaktion

**Portabilität** (= Plattformunabhängigkeit)

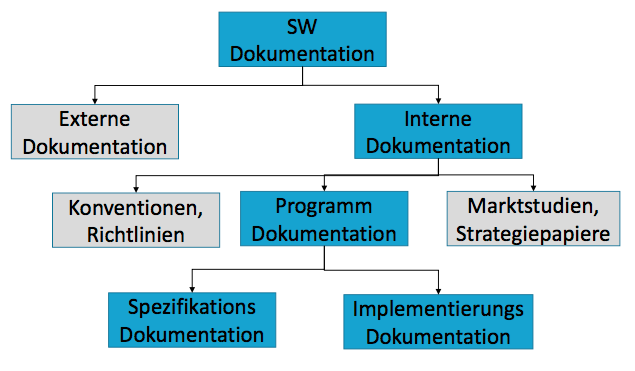
Portierungen

* Architektur: andere **HW**
* OS
* System: andere Geärte**klasse**
* Sprach

Ebenen der Portabilität

* Implementierung: Programm selbst
* Sprach: Compiler + Sprache selbst
* System: Umgebung an Programm anpassen

**Dokumentation** (hier nur Programmdoku -> Teil der internen Doku -> NICHT für Kunden)



**Spezifikation**

Varianten (Sortierung: Verständlich -> Eindeutig)

* Informal
  + Umgangssprachlich
  + Sonderfälle **ignorieren**
  + Sprachliche **Ungenauigkeit**
* Semiformal (UML)
  + Verwendung def. Formalismen
  + Mehrdeutigkeit **vermeiden**, dennoch **gut lesbar**
* Referenzimplementierung
  + **Neues** entspricht **altem** Verhalten
  + **Nicht ausreichend,** da Funktionalität zwar **überprüfbar** aber **nicht verstanden**
* Formal
  + Kein Interpretationsspielraum
  + Schnell extrem komplex
  + Keine Relevant für die Praxis

**Implementierung**

Regeln für Quellcode**kommentare (ANWENDEN!!)**

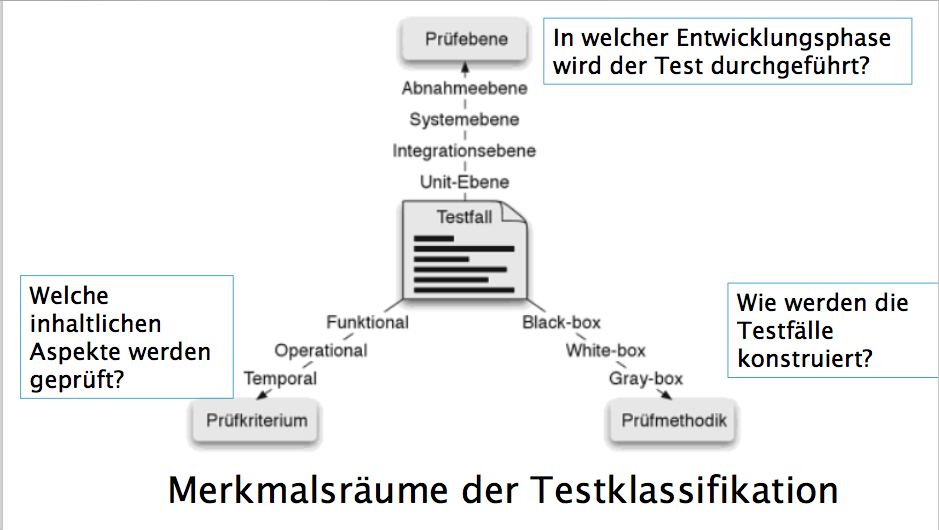
* Code so **verständlich wie möglich**, um **Notwendigkeit** an Kommentaren zu **minimieren**
* **Keine** Information **wiederholen** die bereits im Code vorhanden ist
* **Kurz** aber **vollständig**
* Visuelle **Abgrenzung** vom Code
* Kommentare durch **Tool** **auswertbar** machen (im Header strukturieren)

Achtung

* Dokumentation **NICHT** als Selbstzweck
* Adressaten berücksichtigen
* Dokumentation warum etwas **NICHT** implementiert wurde

Extraktion von Dokumentation (z. B. JavaDoc)

* **Lesbar** von Personen **ohne** Code**zugriff**
* Ohne Code lesbar
* In aufbereiteter Form (PDF, HTML)
* Querverweise einfach nachzuverfolgen

**Software** **Test**

Klassifikation ->

Prüfebenen

* Unit
  + Atomare Einheiten
* Integration
  + Zusammenfassung zu Komponenten
* System
  + Alles zusammen
* Abnahme
  + System vom **Auftrag-geber** durchgeführt

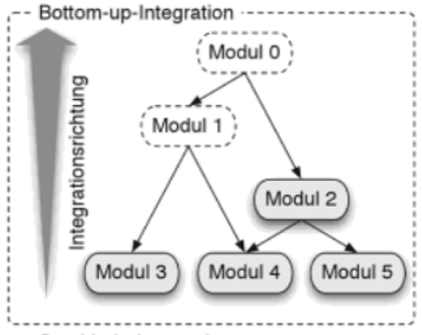
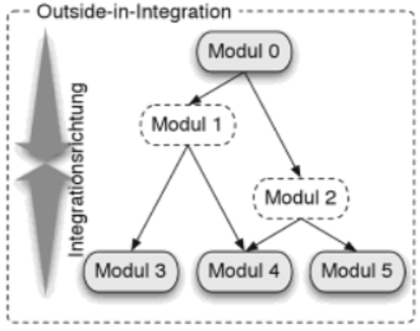
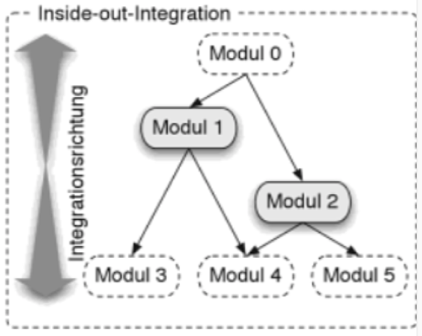
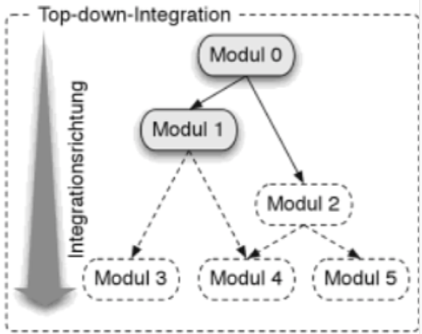
**Unit Tests**

Einzelne Einheiten isoliert betrachten

Mit **richtigen** oder **falschen** **Eingabe**werten

Automatisierung, ermöglicht regelmäßige Ausführung ALLER Tests + **grafische** Auswertung der Ergebnisse

* Aufbau
  + **Definition** Testfälle mit **Eingaben** und erwarteten **Ergebnissen**
* Aufruf
* Auswertung



* + **Vergleich** wirkliches mit erwartetem **Ergebnis**

**Integrationstests**

Module aus einzelnen Units **zusammengesetzt**

Strategien

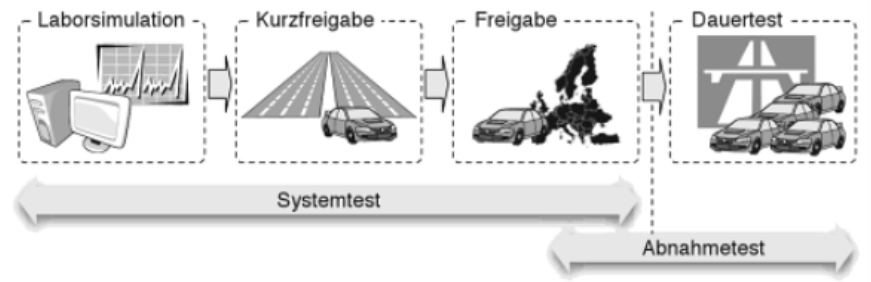
* Big Bang
  + - Alles entwickeln, einmalig integrieren
      * Pros: Testtreiber und Mocks nicht nötig
      * Cons: Erst **am** **Schluss** möglich, schwierige Fehlersuche
* Strukturorientierte (inkrementell)
  + Bottom-Up
    - Basiskomponenten, Verwendung Testtreiber
  + Top-Down
    - Module höchster Schicht, Verwendung von Stubs
  + Outside-In
    - Von beiden Seiten nach innen
  + Inside-Out
    - Von innen nach außen
* Funktionsorientierte
  + Termin
    - Entsprechend Verfügbarkeit
  + Risiko
    - Riskanteste als erstes
  + Test
    - Anhand der Testfälle
  + Anwendung
    - Anhand Usecases

**Systemtests**

Nach Integration ALLER Komponenten, **Funktionale** Sichtweise

Erschwerende Faktoren

* Unvorhergesehene Fehler
* Unklare Anforderungen
* Testumgebung
* Eingeschränktes Debugging
* Eingeschränkte Handlungsfähigkeit

Bsp.: KFZ-Steuergerät:

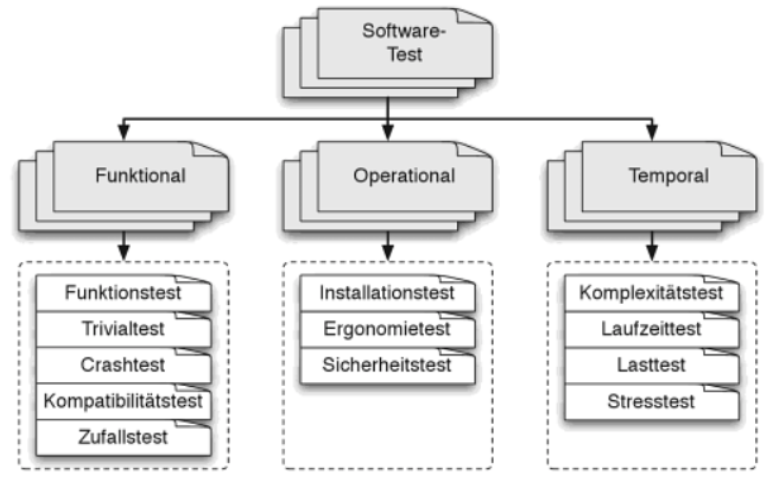
**Abnahmetests**

Findet beim **Kunden** statt, wird vom **Kunden** durchgeführt

Alternativ: **Feldtests**

* Alpha
  + Durchführung bei **Hersteller**, **Ausgewählter** Anwenderkreis
* Beta
  + Durchführung beim **Kunden**, **inkrementell** durchgeführt, **zunehmender** Nutzerkreis

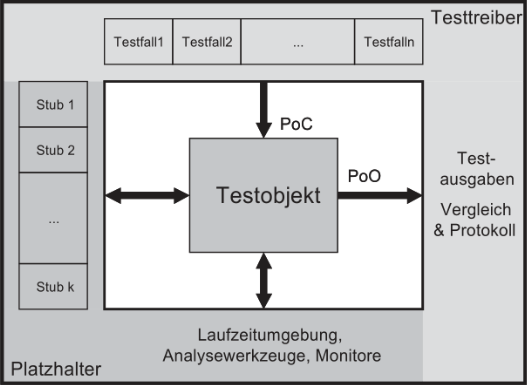
**Prüfkriterien**



**Testklassifikation**

Es muss **ablauffähiges** Programm vorliegen

Für **Unit/Integration**stests wird es in Testrahmen eingebettet



PoO = Point of **Observation**

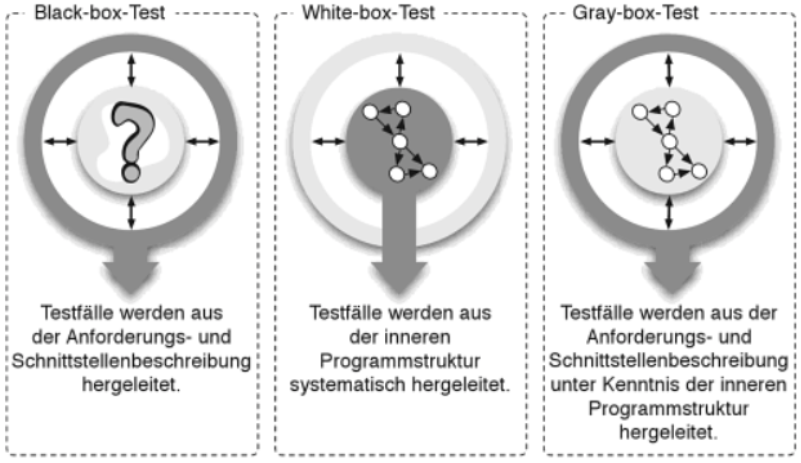
PoC = Point of **Control**

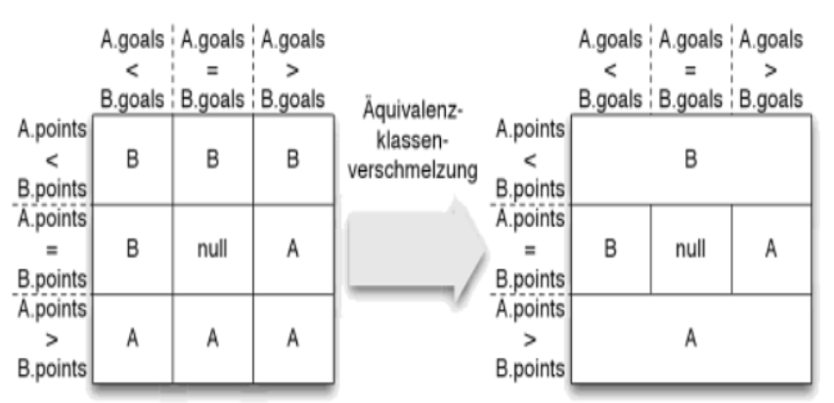
**Erstellen von Testfällen**

* Systematisches Vorgehen
* Geringer Aufwand
* Überprüfung vieler Anforderungen
* Fehler finden

Vorgehen

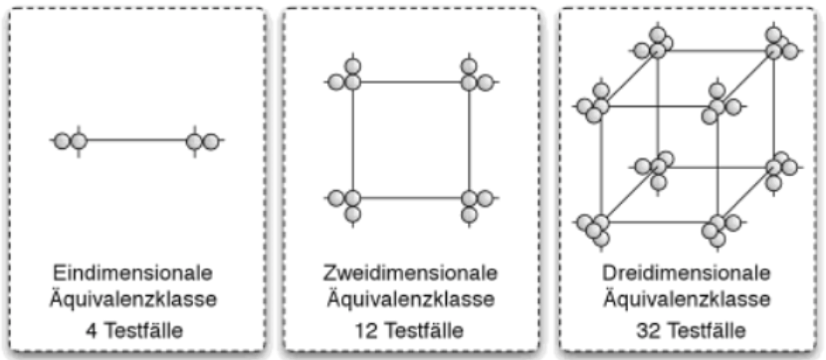
* Festlegen von
  + Zielen, Bedingungen, Voraussetzungen
* Fälle spezifizieren
* Ausführung festlegen

**Prüftechniken**



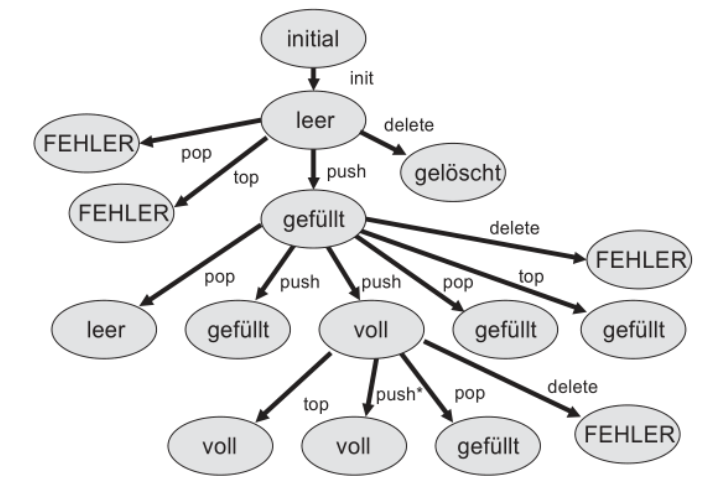
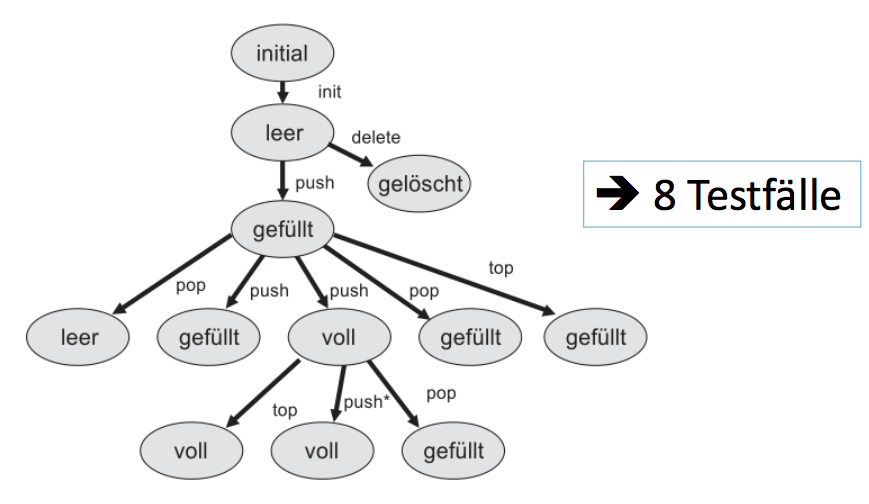
**Black Box**

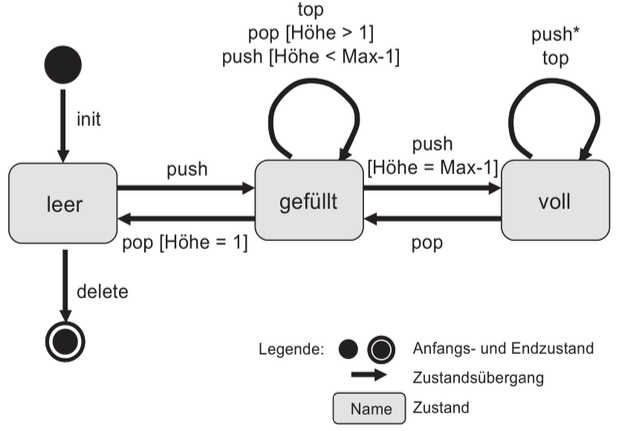
**Äquivalenzklassen** (Vermeidung exp. Testfälle)

* [minLong,0] und [1,maxLong]
* oder [minLong,-1] und [0,maxLong]
* n Übergabeparameter
  + n-dimensionale Äq.Klassen
* Vollständige Partitionierung
  + Einbeziehen aller (auch ungültiger) Werte

**Grenzwertbetrachtung**

* Partitionierung wie bei Äquivalenzklassen
* Auswahl der **Randwerte**

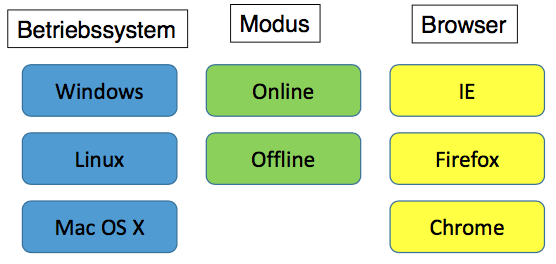
**Zustandsbasierter** SW Test

* Mögliche (**Zustands**-)Übergänge prüfen
* Bsp.: GUI Masken jeweils als Zustand

**Use Case** Test

* Speziell für **System-/Abnahme**tests
* **Alle möglichen** Szenarien eines Use Cases werden durch **einen** Testfall **abgedeckt**

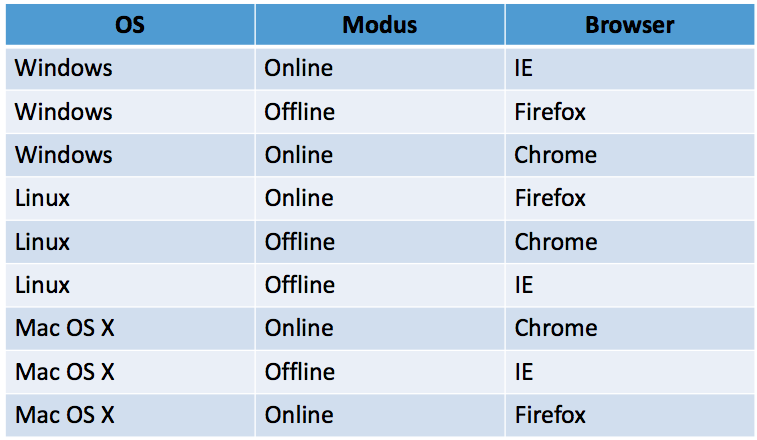
**Paarweises** Testen

* **Anzahl** der Fälle durch Kombinatorik **reduzieren**
* **Annahme**: Nicht alle, aber Kombinationen testen

Bsp.: Web Apps

18 Konfigurationen ->

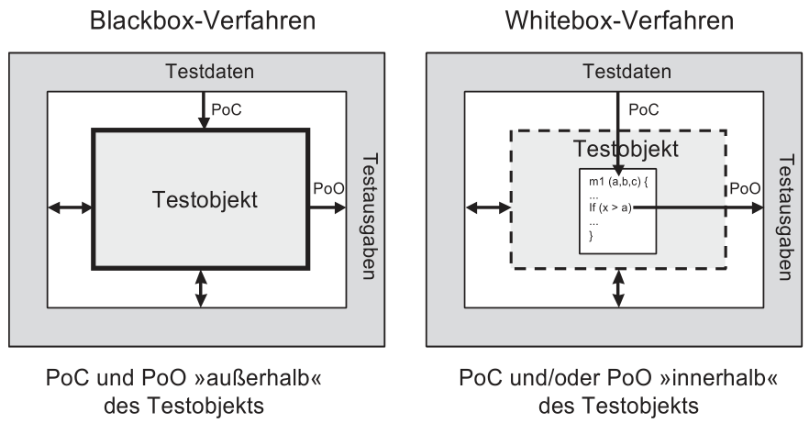
(Bei verschiedenen Versionen entsprechend mehr)

Nicht: Alle kombinatorisch möglichen Konfigurationen!

9 statt 18, siehe Tabelle ->

**Diversifizierende** Verfahren

* Vergleich verschiedener **Versionen**
* Verfahren
  + Back to back
    - **Mehrere** Varianten **ausführen**
    - Ergebnisse **vergleichen**
    - Abweichung **analysieren**
  + Regression
    - Getestetes Programm **erneut** testen
    - Vermeidung **neu** **eingebauter** Fehler
  + Mutation
    - **Beurteilung** von Testverfahren

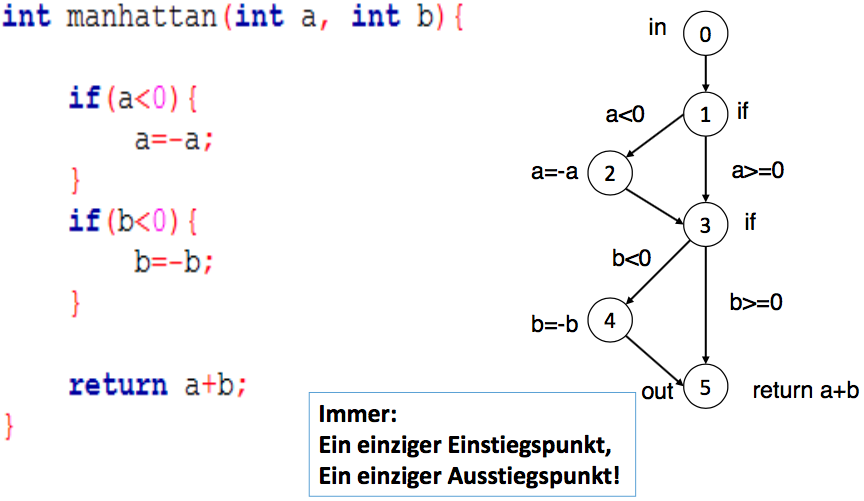
**White Box** (Strukturtests)

Unterschied **Eingabe** basiert auf

* Black Box: **Funktionaler** Beschreibung
* White Box: **Inneren** Programmstruktur

1. Struktur**analyse**
2. Test**konstruktion**
3. Test**durchführung**

Kontrollflussgraphen

* Expandierte ->
  + Jeder Befehl ein Knoten
* Teilkollabiert
  + Zwei oder mehr Befehle ein Knoten
* Kollabiert
  + Verzweigungsfreie Blöcke ein Knoten

**Überdeckungskriterien**

* Anweisungsüberdeckung (C0 Test)
  + Alle **Knoten** min. **einmal** durchlaufen
    - manhattan(-1,-1)
    - Überdeckung: {0,1,2,3,4,5}
  + Mit **wenigen** Testfällen zu erfüllen
  + Wichtige Fälle bleiben **unberücksichtigt**
  + Häufig **schwer** zu erreichen
  + Risiko **nicht kalkulierbar**, wenn **nicht 100%ig** erreicht
* Zweiüberdeckung (C1 Test)
  + Jede **Kante** min. **einmal** durchlaufen
    - manhattan(-1,1) + manhattan(1,-1)
  + Häufig **Minimalkriterium**
* Pfadüberdeckung
  + Jeder **Pfad** vom Eingangs- zum Ausgangsknoten
  + Anzahl Pfade explodiert, v. a. bei Schleifen
    - manhattan(-1,-1) + manhattan(-1,1) + manhattan(1,-1) + manhattan(1,1)
  + **Praktisch** (meist) nicht realisierbar
    - **Varianten**, um Fälle zu **reduzieren**
    - **Schleifen** nur min. **eine** Iteration durchlaufen zur Fehlerdetektion
* Bedingungsüberdeckung
  + **Zusätzlich** zu Kontrollflussgraph
  + Einbeziehung **logischer** Struktur von **Bedingungen**
    - Einfache
      * ->
    - Minimale
      * -->
    - Mehrfache
      * --->

**Testmetriken**

Anweisungsüberdeckung: MC0 = (Anzahl überdeckter Knoten/ Alle Knoten) \* 100%

Zweigüberdeckung: MC1 = (Anzahl überdeckter Kanten/ Alle Kanten) \* 100%

**Verwendung**

* Abnahme für Module (Schwellwerte)
* Vergleich von Modulen

**Mutationstest**

Quantitative Bewertung von Testverfahren

Idee: Füge an **zufälliger** Stelle **Fehler** ins Programm ein und prüfe, **wie viele** davon gefunden werden

**Grenzen** Des SW Tests

Schwierig und aufwendig, aufgrund von

* Unklaren, fehlenden **Anforderungen**
* Programm**komplexität**
* Mangelnde Werkzeugunterstützung
* Aus- und Fortbildungsdefizite
* Zeit

**Testautomatisierung**

**JUnit**

Ziele

* **Einfach** zu schreiben
* **Schnelle** Ausführung
* **Isolierte** Ausführung

Features

* Infrastruktur für **automatisierte** Tests
  + Schreiben, durchführen, auswerten
* Vorbereiten, Nachbereiten, Organisieren
* **Parametrisierbare Test**

Test **Suites**

* Ausführung **mehrerer** Tests aus verschiedenen Testklassen
* Über **Annotationen** im Code

Test **Fixtures**

* **Vor**- und **Nachbereitung**scode für Testmethoden
* **Isolation** des eigentlichen Testcodes
* **Vermeidung** redundanter Vor- und Nachbereitungscodes

JUnit **Annotationen**

* **@Before[Class]**
* **@After[Class]** <- wird **immer** aufgerufen!
* **@Test**(expected = Exception.class)
* **@Test**(timeout=100), ...

JUnit **Assertions**

* fail() -> Lässt Methode fehlschlagen
* assertTrue() -> Prüft ob Boolean == true
* assertFalse() -> Prüft ob Boolean == false
* ...

**Namenskonventionen** (JUnit)

Test für Klasse -> Gleicher Name wie Klasse + „Test“ -> Car + CarTest

Test Methoden -> Beginne Namen mit „test“ (seit JUnit 4 nicht zwingend) -> add() + test\_add()

**Parametrisierbare** Tests

Parameterliste statt einzelner, fixer Werte bei mehrmaligem Aufruf

JUnit **Rules**

* @Before / @After, öfter benötigt
* Einmalige Implementierung ODER Verwendung bereits mitgelieferter Rules
* Entsprechungen
  + @Before == @Rule
  + @BeforeClass = @ClassRule
* Rule**Chain**
  + Hintereinanderausführung von Rules
* Eigene Rules durch Implementierung des Interfaces „TestRule“

JUnit **Categories**

* Gruppierung von Testmethoden
* Nur Test bestimmter **Kategorie** laufen lassen

JUnit **Theories**

* Allgemeingültige Aussage als Test formulieren und durch eine Reihe von Testwerten prüfen
* Trennung Methode | Daten

JUnit **Custom Runners**

* Für die **Ausführung** von Tests zuständig (Annotation mit @RunWith() )
  + Testklassen Instanziieren
  + Ausführen
  + Reporting der Resultate
* Normalerweise sind die **bereits vorhandenen** ausreichend

Junit**3** vs. Junit**4**

In 3

* **Ableitung** von TestCase
* Methoden **setUp**() und **tearDown**() statt **annotierte** Methoden
* Name muss mit “**test**” beginnen

**Statische Analyse** (Sichtung der Quelltexte)

Pros

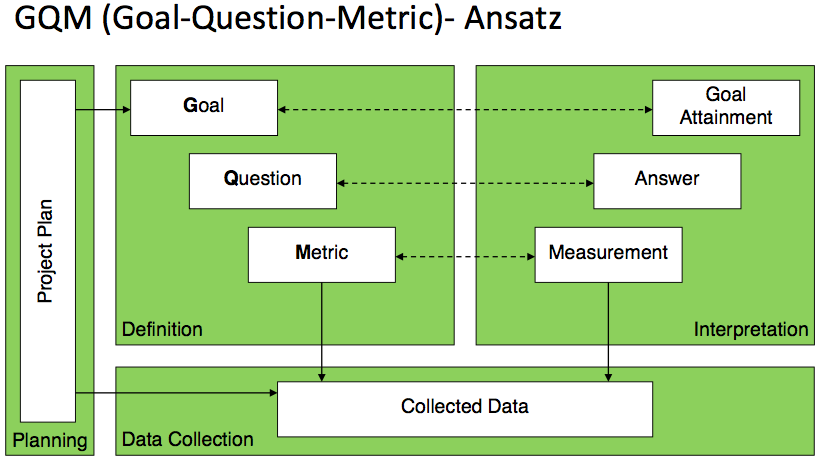
* Beim Testen können Fehler andere Fehler **überdecken**
* Inspektion **unvollständiger** Versionen
* **Qualität**smerkmale **prüfen**

Cons:

* Nur Einhaltung der **Spezifikation** prüfbar, **nicht** Benutzer**anforderung**
* Schwierige Überprüfbarkeit von **nicht-funktionalen** Charakteristiken (**Performance**, **Usability**)

**Manuelle** Prüfung

Reviews Vorgehensweise

1. Planung, Definition der
   1. Review**objekte**
   2. **Reviewer**
   3. Eingangs- und Ausgangs**kriterien**
2. Einführung
   1. Alle erhalten nötige Infos
   2. Bedeutung, Sinn, Zweck und Ziel des Reviews
3. Vorbereitung
   1. **Individuelle** Vorbereitung
      1. Mängel, Fragen, Kommentare notieren
4. Reviewsitzung
   1. Durchführung durch **Moderator**
   2. Ziel: Beurteilung Prüf**objekt** nach
      1. Umsetzung Vorgaben / Richtlinien
      2. Aufzeigen von Fehlern, Abweichungen, Unstimmigkeiten
   3. Ergebnis: Objekt wird
      1. Akzeptiert
      2. Akzeptiert, mit Änderungen
      3. Nicht akzeptiert

SW **Metriken**

Probleme in der Praxis

* Nutzen oft unklar
* Standards fehlen
* Programmierer wehren sich

Durchführung von Messungen

* Definition + Sammeln von Messwerten
* Analyse + Interpretation + Beurteilung
* Gefahr: Schwer interpretierbare Zahlen -> **Zielorientiertes** Messen!

Gütekriterien von Metriken

* Objektivität -> **Keine** **Subjektiven** Einschlüsse
* Robustheit -> Immer **gleiches** Ergebnis bei **Wiederholung**
* Vergleichbarkeit -> **Verschiedene** Messungen müssen in **Relation** gesetzt werden können
* Verwertbarkeit -> Unterschiedliche **Ergebnisse** führen zu unterschiedlichem **Handeln**
* Korrelation -> Aussage der Metrik im Bezug auf die Kenngröße

Verwendung von Metriken

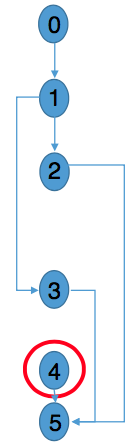
* Beurteilung von eigener/fremder SW,
* Kostenabschätzung + finden kritischer Stellen im Code

Metriken – **Beispiele**

* Lines of Code (**LoC**) / Non Commented Source Statements (**NCSS**) – wie LoC **ohne** Kommentare
  + Pros: Einfach, schnell automatisiert zu erheben
  + Cons: Mangelnde Vergleichbarkeit zw. Programmiersprachen
* **Halstead** – Lexikalische Struktur -> Aufteilung in **Operatoren** und **Operanden**
  + Pros: Einfach, automatisiert auswertbar, für alle Programmiersprachen geeignet
  + Cons: Nur lexikalische Komplexität, Sichtbarkeit / Namensräume **nicht** berücksichtigt
    - Verwendung:
      * Bewertung der **Wartbarkeit** eines Moduls
      * Beurteilung der **Entwicklung** über die Zeit eines Moduls
* **McCabe** – Zyklomatische Komplexität (Graphentheorie) -> Komplexität in **einer Zahl**
  + Pros: Einfach, Gute Korrelation zw. Zahl und Verständlichkeit
  + Cons: Berücksichtigt nur **Kontroll**- nicht Datenfluss
    - (Anzahl Kanten) – (Anzahl der Knoten) + 2
* **Objektorientierte** – Berücksichtigt Vererbung
  + **Komponenten**
    - AnzahlderObjekt-/Klassen**variablen** bzw. Summe aus beiden
    - Umfangsmetrik
      * Hinweis auf
        + Funktionale Bedeutsamkeit
        + Potentielle Monolithen
    - Weighted Methods per Class (**WMC**)
      * Falls hoch -> schwer **wartbar**
    - Depth of Inheritence Tree (**DIT**)
      * Falls hoch -> schwer **verständlich**
    - Number of Descendants (**NOD**)
      * Falls hoch -> potentiell falsche **Abstraktion**
    - Lack of Cohesion (**LCOM**)
      * Falls hoch -> schlechte **Kohäsion**

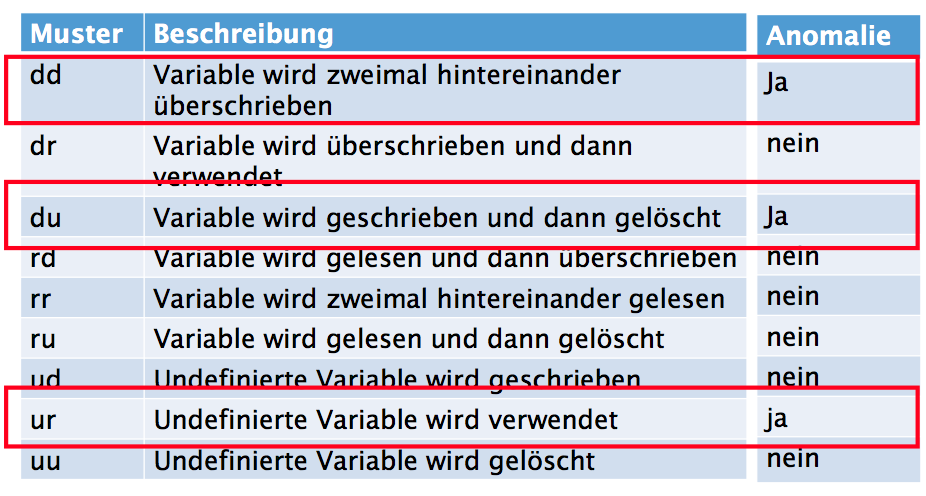
**Konformitätsanalyse**

* Syntax -> In Compiler integriert
* **Semantik**
  + -Wall flag für gcc
  + Secure Programming Lint (**Splint**)
  + „False Negatives“
    - Verstehen
    - Durch Änderung vermeiden
    - Oder Meldung ausblenden

**Anomalie**analyse

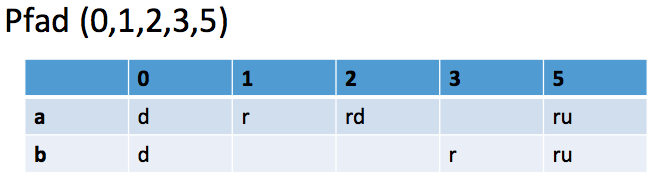
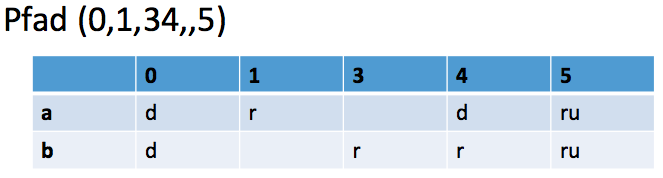
**Kontrollfluss**anomalie -> siehe Grafik links, Graph checken, Unerreichbarer Knoten

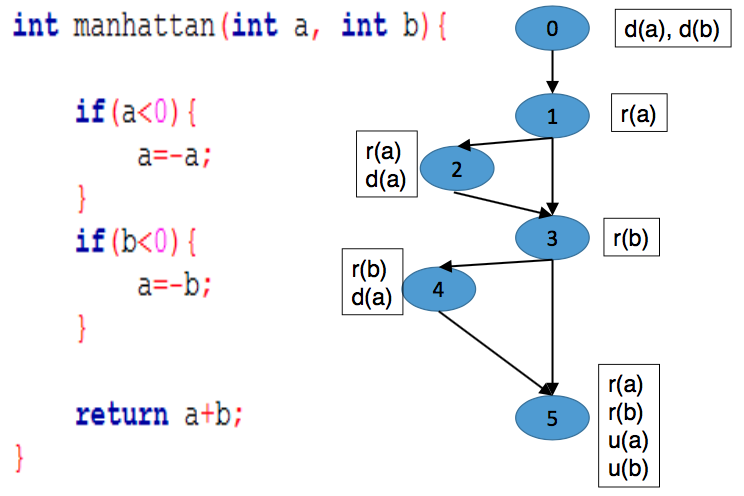
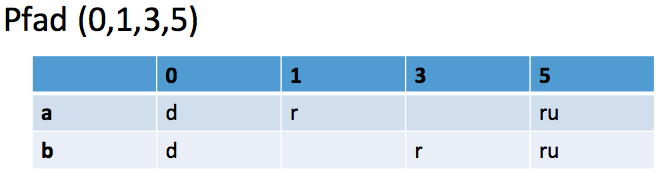
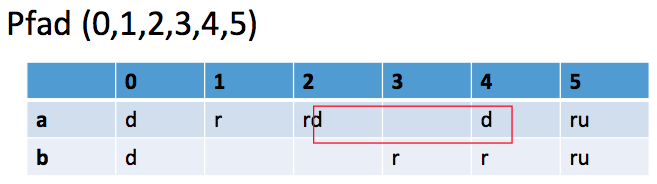
**Datenfluss**anomalie



**Notation**

d(x): x wird definiert r(x): x wird referenziert u(x): x ist undefiniert





**Tools** zur statischen Code Analyse

* Einhaltung von Konventionen
* Erfassung von Metriken
* Lokalisierung von potentiellen Bugs
* Bsp.:
  + Findbugs
  + CheckStyle
  + JDepend

**Vorgehensmodelle**

Beschreibt:

* Menge von Aktivitäten
* Produkte
* Rollen
* Vor- und Nachbedingungen

Wichtige Modelle – **nicht agil**

* Code and Fix
* Wasserfall
* Prototyping
* V-Modell
* Rational Unified Process (**RUP**)

Kritik

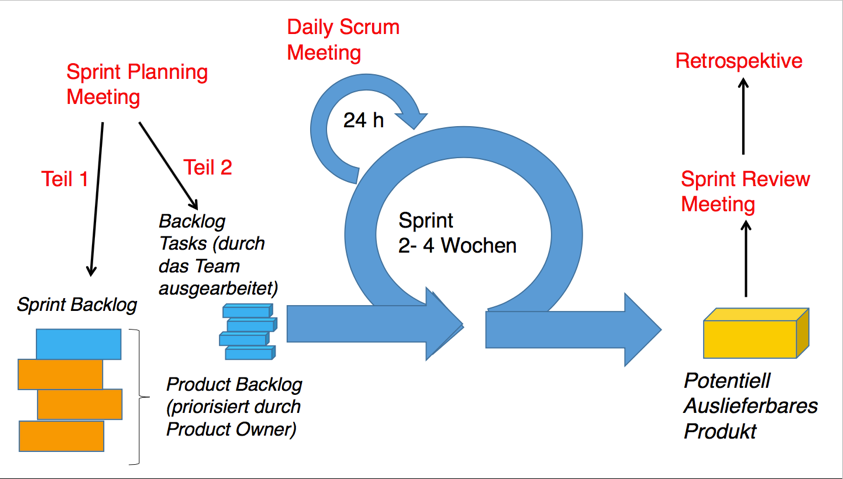
* Viele **Dokumente** erzeugen + pflegen
* Prozessbeschreibungen **hemmen Kreativität**

**Agile** Methoden – nicht auswending!

* **Individuen** und **Interaktionen** mehr als **Prozesse** und **Werkzeuge**
* **Funktionierende** SW mehr als **umfassende** Doku
* **Zusammenarbeit** mit dem **Kunden** mehr als Vertragsverhandlung
* **Reagieren** auf **Veränderungen** mehr als das Befolgen eines Plans

**Prinzipien** – nicht auswending!

* Kunde durch **frühe, kontinuierliche** Auslieferung wertvoller SW **zufrieden** zu **stellen**
* Anforderungs**änderungen** auch **spät** willkommen
* Liefere **funktionierende** SW regelmäßig, innerhalb **weniger Wochen**
* **Fachexperten** und **Entwickler** müssen **zusammenarbeiten**
* **Motivierte Individuen -> Erledigen Aufgabe!**

****

**Scrum** – empirisch, inkrementell, iterativ

Transparenz -> Überprüfung -> Anpassung

**Rollen**

* Product Owner
  + Wertmaximierung Produkt
  + Management **Product Backlog**
* Scrum Master
  + **Verständnis + Durchführung** Scrum
* Entwicklungs Team
  + **selbstorganisiert + interdisziplinär**
  + Flexibilität, Kreativität, Produktivität

Nach Gloger:

* Customer
  + Auftraggeber, finanziert Projekt
* Manager
  + Stellt Ressourcen und Ritchlinien bereit
* User
  + Informationsquelle

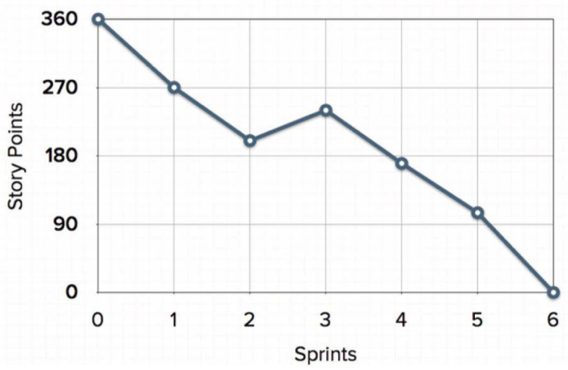
**Meetings** (alle im **Sprint**)

* Sprint Planning
  + Was **enthält** nächstes Product-Increment?
  + Wie wird die nötige Arbeit **erreicht**?
    - 1
      * Anforderungs Workshop
      * Ergebnis: **Sprint Backlog**
    - 2
      * Design Workshop
      * Ergebnis: **Liste der Tasks** am Taskboard
* Daily Scrum
  + Was habe ich **gestern** gemacht?
  + Was werde ich **heute** tun?
  + Was **hindert** mich bei meiner Arbeit?
* Sprint Review
  + **Prüfung** Product-Increment
  + Ggf. **Anpassung** Product Backlog
* Sprint Retrospective
  + Selbst**kontrolle**
  + **Verbesserung**splan

Nach Gloger:

* Estimation Meeting
  + **Schätzen** der Backlog Items -> Keine Aussage über **Art / Einheit**!
    - **Storypoints** anhand
      * Geschäftsnutzen / Aufwand
  + Ggf. Releaseplan anpassen

**Artefakte**

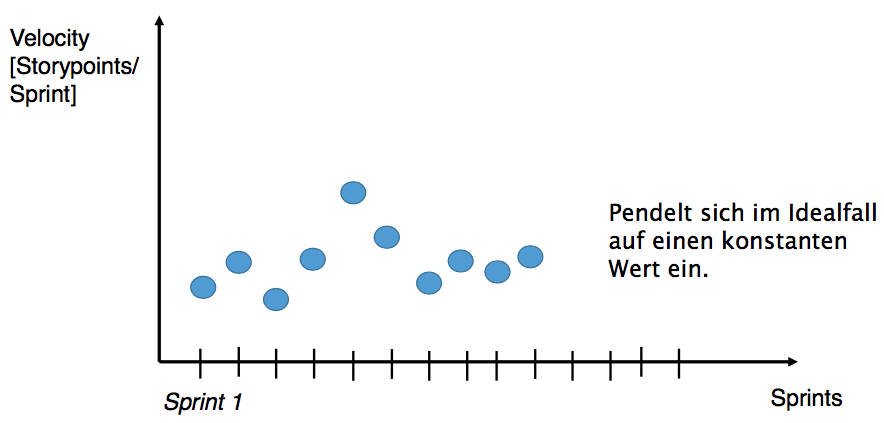
* Product Backlog
  + **Geordnete** Liste von **allem** was im Produkt enthalten sein **kann**
* Sprint Backlog
  + Auswahl Backlog Items für Sprint
* Sprint Ziel
  + Zusammenhängende Funktionalität
* Product Increment == Ergebnis

Nach Gloger: Vision, Sprint Goal, Tasks, Releaseplan,

Impediment Backlog

**Definition of Done**

Gemeinsames Verständnis von „Done“

**Reporting**

* Taskboard (Scrum Board)
  + Täglicher Fortschritt
  + Impediments sichtbar
* Charts
  + Burndown Chart
* Berechnung der Velocity

**Einschätzung**

Agiles Vorgehen **!=** Verzicht auf Planung

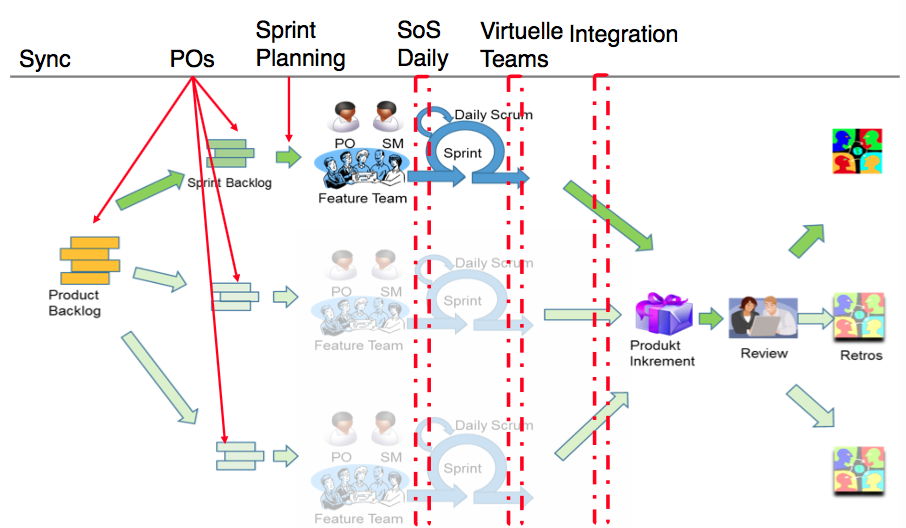
Gute Transparenz

Kein Allheilmittel

Scrum in **großen Projekten** (mehr als ein Team)

* Organisches Wachstum
  + Einarbeitung neuer Mitarbeiter
  + Team entscheidet wann es sich teilt
* Sprunghaftes Skalieren
  + Mitglieder des **Initialen** übernehmen Rolle des (Sub)**Product Owners** in **neuen Teams**

**Schnitt** nach

* Component Team -> Pro Komponente
* Feature Team -> Pro Funktion

**Synchronisation** durch

* Scrum of Scrums
* Product Owner Team
* Scrum Master Group
* Virtuelle Teams
  + Architektur, Doku, Tests, ...
* Gemeinsame Planning Meetings

**Requirements Engineering**

**Anforderungen** legen fest, was man von einem SWsystem als **Eigenschaften** **erwartet**

Arten

* Funktional
* Qualität
* Randbedingung

**Einbettung** in Vorgehensmodelle

* Wasserfall: Eigene Projektphase
* Agile Prozesse: Kontinuierlicher, phasenübergreifender Prozess
  + Product Backlog **!=** Anforderungen
  + Items nicht **vollständig**, bevor nicht **diskutiert**

**User Story -** In der **Sprache** des **Kunden**, verwende **Benutzerrollen**, **Keine** Technik / Benutzeroberfläche

* **Karte**, Konversation, Akzeptanzkriterien -> „Als Coach will ich mein Profil einstellen“
* **Epic**: Große User Story, nicht **innerhalb eines Sprints** umsetzbar
* **Thema**: User Stories, gruppiert um **funktionales** Thema

Nichtfunktionale Anforderungen -> **Constraints** fomulieren

Fehler -> Entweder **sofort beheben** oder **einplanen**

Technisches Backlog -> Owner ist das **Team**

Ggf. -> **Umformulierung** in User Stories

**Sprachtemplate**

„Als Anwender <mit der **Rolle**> benötige ich eine <**Funktionalität**>, damit ich <**Nutzen**> bekomme“

**Schneiden** nach: Vertikal, Daten, Aufwand, Forschungsanteil, Qualität, Benutzerrolle, ...

Acceptance Test Driven Development (**A-TDD**) -> Tests as requirements, requirements as tests

**Konfiguration Management**

Projektergebnisse sicher **verwalten** und den Teammitgliedern **kontrollierten** Zugriff gewähren

Ziele

* **Änderungen** kontrollieren
  + Versionsmanagement
  + Rechtevergabe
* **Kommunikation** vereinfachen / Transparenz verbessern
  + Projekt Homepages
  + Kollaborationstools
* **Qualität** sichern
  + Projektautomatisierung
  + Testautomatisierung
  + Change Management
* **Produktivität** steigern
  + Entwickler können sich aufs **Ziel** konzentrieren

Konfig **Elemente**

* Alle Elemente die zur **Erstellung** der SW **nötig** sind oder **beschreiben**
  + Quelltext
  + Use Cases
  + Architektur / Design Dokumente
  + Schnittstellenverträge
  + Mögliche: Werkzeuge, Libs / Frameworks
  + **Keine**: Protokolle, Projektpläne, binäre Auslieferungsdateien

Projekt**ablage**

* Daten zur **Steuerung / Durchführung** des Projekts

**Beschreibung**

* Inhalt
* Namenstemplate (UC\_Preisliste\_erstellen)
  + Anforderungen
    - Jede Instanz eindeutig **identifizierbar**
    - Name der Datei weist auf **übergeordnetes** Element hin
    - **Beziehungen** zwischen Elementen aus Namen **ersichtlich**

Festlegung der **Projektstruktur** nach

* Konfig Element
* Projekt Struktur
* SW Architektur

**Verwaltung**

* **Verfügbarkeit** sicherstellen
* **Integrität** bei gleichzeitiger Änderung
* **Nachvollziehbarkeit** der Änderungen
* **Verhinderung** unberechtigter Zugriffe

Alles erreichbar durch **Repository**

* **Copy**-Modify-**Merge**
  + Niemand wird **blockiert**
* **Lock**-Modify-**Unlock**
  + **Vermeidung** von **Konflikten**, auch für binäre Dateien gangbar

Tags statt Versionsnummern

Bedeutsames Ereignis == baseline (technisch == Tag)

Baseline pro Release -> Kein Release pro Baseline

**Release** ist Zusammenstellung von

* **Ausführbaren** Elementen
* **Doku**mentation
* Definition des **Installationsvorgangs**
* Alles möglichst **automatisiert**

**Projektautomatisierung**

Builds (Shell Scripte, Ant, Maven)

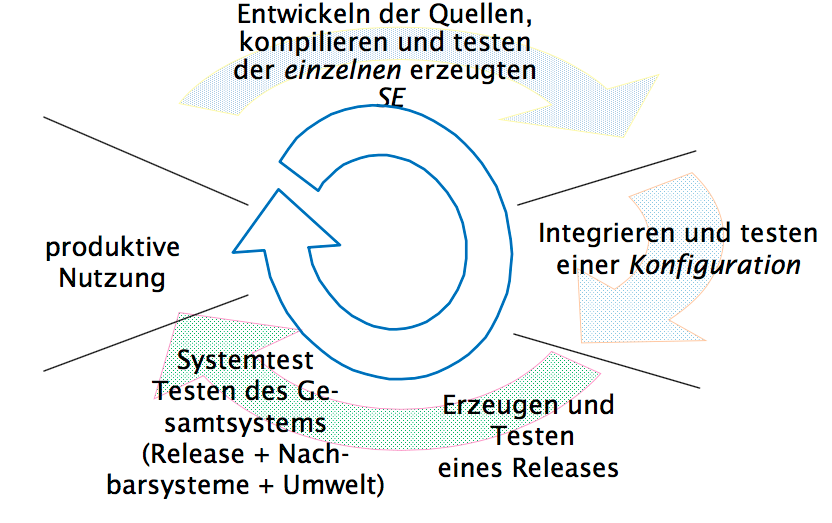
* Entwickler
  + Lokal, **schnell**
* Integration
  + **Regelmäßig** auf Integrationumgebung
* Release
  + Ähnlich Integration, **Tag setzen**

**Änderunges- und Fehlermanagement**

Geordneter, steuernder Umgang mit Anforderungen der **Umwelt** an das System

**C**hange **R**equest (CR)

* Änderungsmanager
  + Vorfilterung
  + Bewertung (Kritisch? Aufwand?)
* Change Control Board
  + Entscheidung über Annahme / Abelehnung

**Workflow ->**

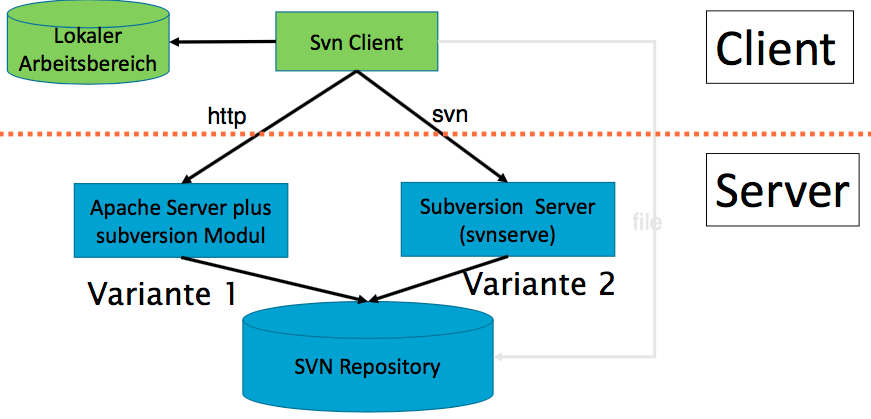
**Raumkonzept (Umgebungen)**

* Entwicklung
* Test
* Integration
* Produktiv

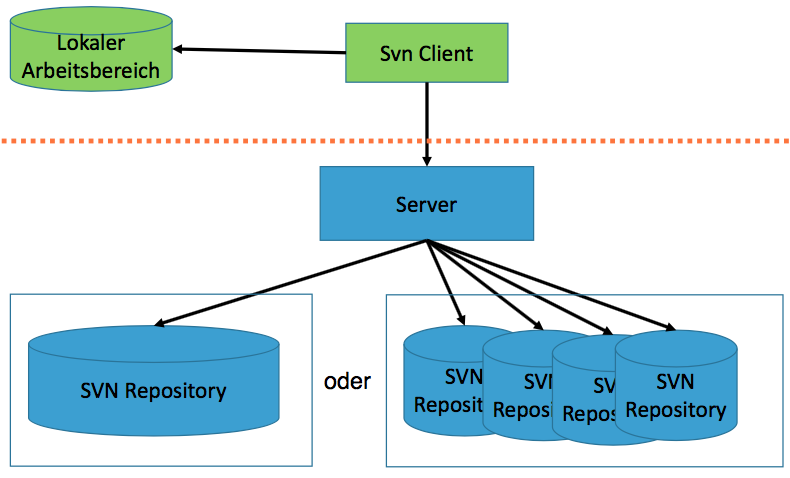
Übergänge zw. Umgebungen stellen

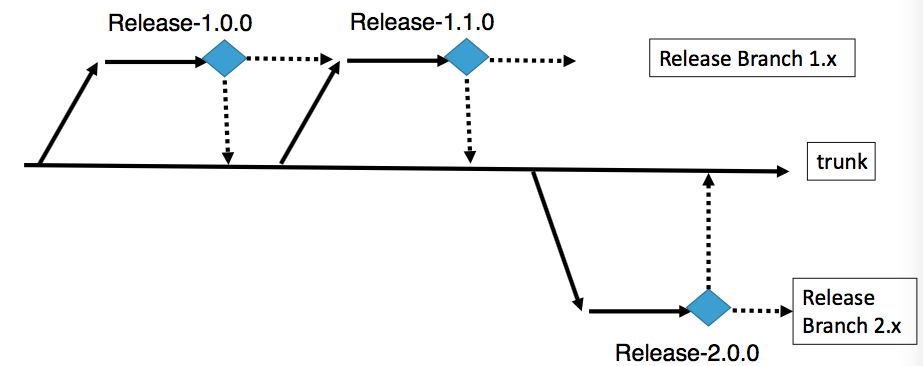
**Planungseinheit** dar

* Konfig Management
  + Release-Notes
  + Archivierung Release
* Qualitätsmanagement

**Subversion (SVN)**

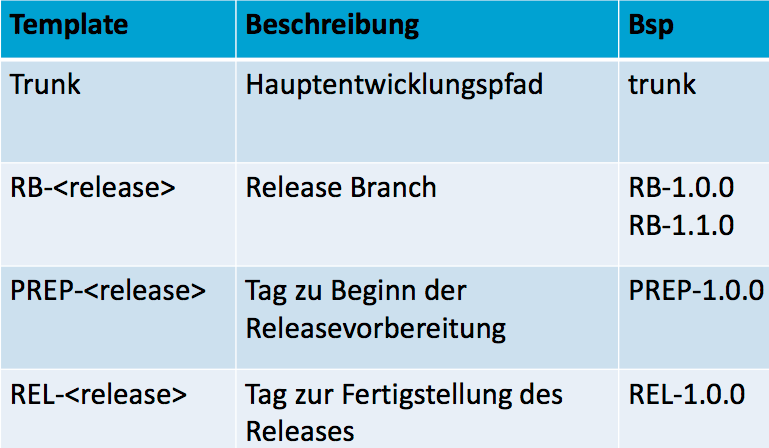
Architektur ->

Repo einrichten:



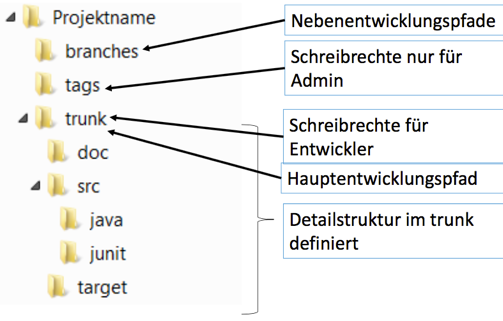
Entscheidung

* Ein Repo für **alle** Projekte
* En Repo für **jedes** Projekt



**Arbeitsweise** Repos

* Revisionen
  + Für **jede** schreibende Transaktion
  + Bezug auf **alle** Elemente (globale Versionierung)
* Changeset
  + **Alle** Änderungen in Transaktion
  + Zuordnung Revision und umgekehrt



SVN kennt **keine** Tags / Branches!

Beides als **Verzeichnisse** im Repo abbilden

**Anlegen** der Struktur + **Import** ins Repo (svn import)

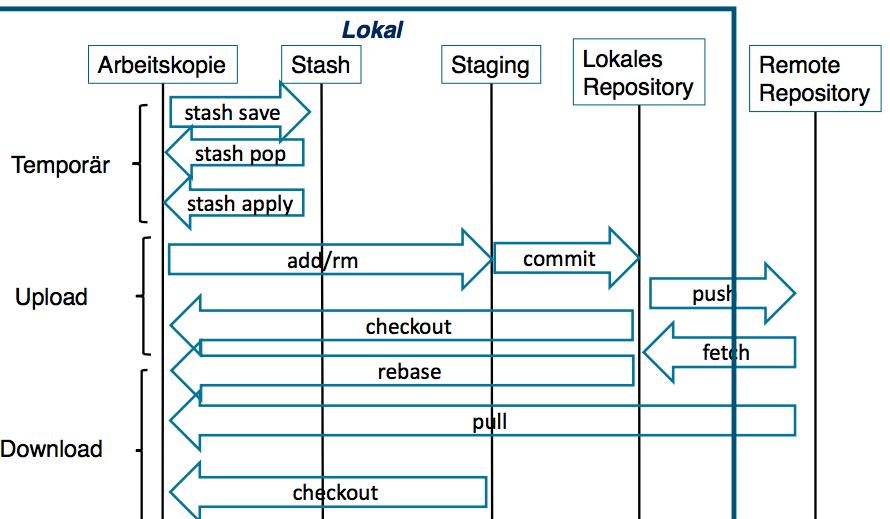
**Arbeitsbereich** (svn checkout svn://localhost/<repo>/trunk)

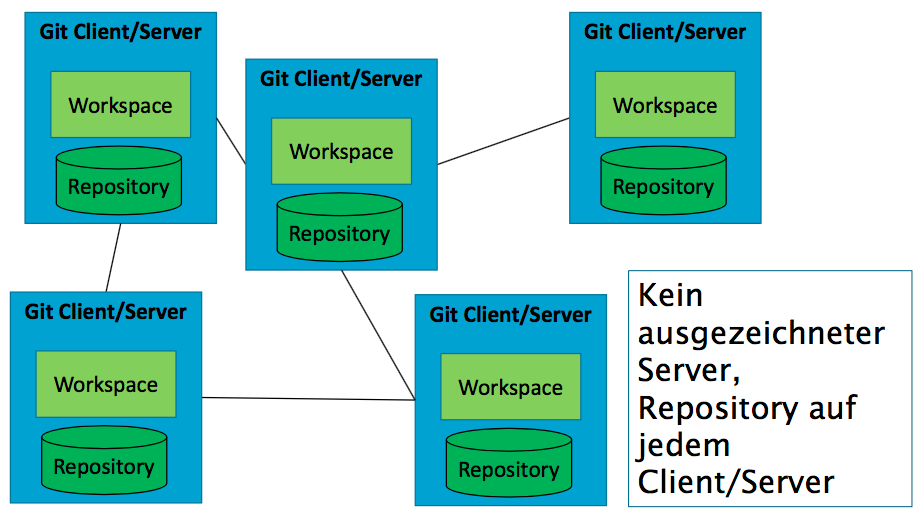
* .svn
  + Auf welcher Revision es aufbaut
  + Zeitstempel der letzten Aktualisierung
* Vergleich lokal <-> Repo
  + Unverändert aktuell / veraltet
  + Lokal geändert aktuell / veraltet

Konflikte automatisch / manuell auflösen!

**Arbeitsprozess**

* Arbeitsbereich aktualisieren
* Prüfung auf Konflikte
* Änderungen durchführen + prüfen + ins Repo schreiben

**GIT**

**Commit**

* Umfasst ganzes Projekt
* Für jede Datei eine Kopie im Repo
* Hash pro Commit

**Historie** -Einrichtung **ausgezeichnetes** Repo, hält **offiziellen** Stand

**Staging**

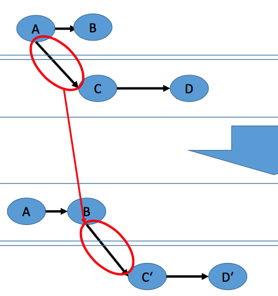
* **Unterschied** Workspace und Staging Area **möglich**
* Detailinfo was commited wird

**Branching** – Entwicklungszweige

* Parallele Entwicklung, Mehrere Entwickler
* Immer **genau ein aktiver** Branch

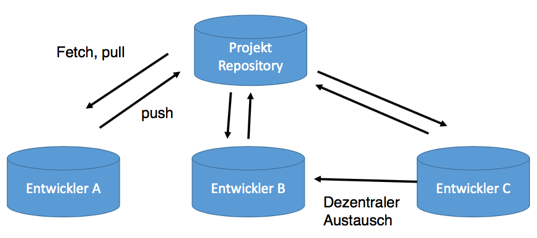
**Merging**

* Zusammenführen + automatischer commit
* Bei nicht lösbaren Konflikten
  + Automatisch **germergte** Dateien im **Workspace** und im **Staging**
  + **Konfliktstellen** markiert nur im **Workspace**

**Rebase**

* Branch an anderen Knoten ansetzen
  + Historie **glätten** (Mehr Entwickler, gleicher Branch)
  + Commits auf **falschen** Branch
* **Folge** von Commits auf **Zielbranch** einspielen

**Repositories**

* Anlegen: git init Klonen: git clone
* Bare: Repo, kein Workspace

Austausch zwischen Repos ->

**Remote tracking Branch** -> Lokal, zeigt wo Remote Branch steht

**Upstream Branch** -> Verknüpfung zw. Lokalem und Remote Tracking Branch

**Workflow**

2 Branches + features

* master
* development
* + feature branches (Quelle development)

**Maven**

**Convention** over **Configuration**

**Einheitliches** Interface um Projekte zu **bauen**

**Standardisierte** Projektstruktur

**Funktionen**

* Durchführung Build **Prozess**
* Verwalten von **Abhängigkeiten**
* Erstelle **Projektdoku**

1. Generiere Projektstruktur -> mvn <**plugin**>:<**goal**> z. B. mvn archetype:generate
   1. Jeweils Plugin pro Aufgabe
   2. Goals sind Tasks die durchgeführt werden
2. Bauen der App -> mvn install
   1. Kompilieren
   2. Testen
   3. Paketieren
   4. Installieren
3. Laufen lassen

Project Object Model (**POM**)

* Deklarative Beschreibung des Projekts
* Unique Identifier für Projekt, dependency oder plugin:
  + **groupId** -> Gruppe, Organisation
  + **artifactId** -> Projekt
  + **version** -> Release
  + **packaging** -> Projekttyp (jar, zip, ...)

**Build Phases**

Lifecycle Phase

* mvn install
  + **Default** -> Deployment
  + **Clean** -> Cleaning
  + **Site** -> Documentation

**Repository**

* Sammlung Projekte / Artefakte als **Ordnerstruktur** nach Maven **Koordinaten**
  + <repo-root>/<groupId>/<artifactId>/<version>/<artifactId>-<version>.<packaging>
* Suche erst **lokal**, dann **remote**

**Dependency Management**

* Abhängigkeiten auflösen und ggf. Nachladen

**Site Generation**

* Generierung **Projekt Webseite**

**Nexus**

Notwendigkeit

* Keine Kontrolle über Remote Repo
* Lizenzierte SW?
* Eigene Artefakte nicht publizieren, innerhalb der Firma zugänglich
* Repo Manager als Proxy zu Remote Repos

**Hudson** (quasi **Jenkins**) – Continuous Integration

Build Profil in Maven POM

Integrationsbuild

* Bei jedem Commit
* Code Metriken
* Tests
* Testmetriken
* Auslieferung

Nach dem Build

* Veröffentlichen der JUnit Testergebnisse