

Arten von Software:

- Produkts / Individualentwicklung
- Systemsoftware / Anwendungssoftware
- Produktintegriert / reine Systeme
- Daten- / Berechnungsintensiv
- monolithisch / Verteilt
- Standalone / Integriert

Eigenschaften von Software:

- > immateriell
- Verschleißfrei
- Software veraltet
- Es gilt keine Ersatzteile: Pfeekte sind immer Konstruktionsfehler
- Schwer messbar
- leicht änderbar
- ständiger Anpassungsdruck

Themen des Software Engineering:

- (- Projektmanagement)
- Vorgehensmodellierung
- Software - Entwicklungsmethoden:
  - > Requirements Engineering
  - > Software - Architektur + Entwurf
  - > Software Wartung
  - > Re- Engineering / Sanierung
- Qualitätsmanagement (inkl. Testverfahren)
- (- Notationen + Sprachen (UML, Java, ...))
- Werkzeugunterstützung (Case, SVN, ...)

- = Nicht relevant für den Kurs

Kommunikation:

- Austausch / Übertragung von Informationen

Hauptgründe für Projektabbruch:

1. Änderungen der Anforderungen / des Umfangs
2. Mangelnde Einbindung des höheren Managements
3. Anpass im Budget
4. Fehlende Projektmanagementfähigkeiten

Nicht: Technische Probleme

4-Seiten-Modell:

Bsp:  
Streitendes  
Ehepaar

- Sachebene: „Inhalt“ der Nachricht
- Selbstkundgabe: Was der Sender von sich kundgibt
- Beziehungsseite: Wie Sender/Empfänger zueinander stehen
- Appellseite: Was der Empfänger tun soll

Weitere Aspekte der Kommunikation:

- Explizite und implizite Botschaften
- Nonverbale Kommunikation
- Kongruente / inkongruente Nachrichten
- „Man kann nicht nicht kommunizieren“

Kommunikationsprobleme:

- Einseitige Empfangs- / Sendegewohnheiten (z.B. „Senden“ nur auf Beziehungsebene)
- „Bullshit Bingo“ und „KO boy to much terms“

Zu viele  
Fachbe-  
griffe

Killerphrasen:

- Berufung auf vergangene Lösungen / Tradition
- Zeitaspekt
- Thematische Verwirrung ("Das gehört nicht hierher")
- Mangelndes Wissen des Kommunikationspartners
- Frage der Realisierbarkeit
- Frage der Zuständigkeit

Kennzeichen gelungener Kommunikation:

- Verständlichkeit
- ziel-/lösungsorientierung
- persönliche Transparenz
- wechselseitige Beziehungsgestaltung

Kategorisierung von Konflikten:

- Nach Streitgegenständen
- " Erscheinungsformen
- " Eigenschaften der Konfliktparteien

Verschiedene Handlungstypen als Konfliktursache:

- Verständorientiert:
  - ⊕ > Autonomie
  - ⊖ > Isolation / Distanz, Angst vor Nähe / Abhängigkeit
- Risikoorientiert:
  - ⊕ > Hohe Veränderungskraft / Tempo / Angst am Neuen
  - ⊖ > Respektlosigkeit / Unverbüdlichkeit, Angst vor Grenzen
- Ordnungsorientiert
  - ⊕ > Beständigkeit, Stabilität, Genügigkeit
  - ⊖ > Starrheit, Intoleranz / Angst vor Chaos / Unsicherheit

## Verschiedene Handlungstypen als Konfliktursache (2):

- ⊕ - Nähe / Beziehungsfähigkeit / Kontaktfreude
- ⊖ - Angst vor Isolation, Selbstentgabe, Abhängigkeit

## Escalationsmodell:

1. Verhärtung
2. Debatten
3. Taten statt Worte
4. Sorge um Image / Koalitionen
5. Gesichtsverlust
6. Proliferation
7. Begrenzte Vernichtungsschläge
8. Zersplitterung
9. Gemeinsam in den Abgrund (sovietische Methode)

Hohe  
Schwierigkeit

## Schichtenmodell zur Bearbeitung von Konflikten:

- > Arbeitsorganisation: Hilfsmittel, Infrastruktur, Bedingungen
- > Rollen: Arbeits- / Kompetenzverteilung, Fähigkeiten, Fertigkeiten
- > Verhalten: Verhaltensmuster, informelle Rollen
- > Werte und Normen: Persönliche Überzeugungen, Einstellungen, Weltanschauungen
- > Persönlichkeit

Tiefe  
↓  
Schwierigkeit

## Softwarequalität ⇒ Qualitätsmerkmale:

- Funktionalität
- Effizienz
- Zuverlässigkeit
- Benutzbarkeit,

Kundenorientiert

- Portabilität
- Wartbarkeit
- Transparenz
- Testbarkeit

Herstellerorientiert

## Korrelationen von Softwarequalitätsmerkmalen:

- Effizienz: Negative Korrelation mit fast allen anderen Merkmalen  $\Rightarrow$  Mit Bedacht optimieren
- Benutzbarkeit: Keine Korrelation mit anderen Merkmalen  $\Rightarrow$  Benutzertreue möglich, ohne andere Merkmale zu beeinträchtigen

## Magische Dreiecke des Projektmanagements:



## Warum ist Softwarequalität oft schlecht?

- Wachsende Komplexität
- Neue Anwendungsbereiche
- Vollständige Tests praktisch unmöglich
- Produktlebensdauer >> Projektdauer
- Erwartungshaltung / Fehlerakzeptanz des Kunden

## Was kann man gegen schlechte Softwarequalität tun?

- Produktqualität:
  - > Konstruktive Qualitäts sicherung: Softwarerichtlinien, Typisierung, Vertragsbasierte Programmierung, Portabilität, Dokumentation
  - > Analytische Qualitäts sicherung: Tests, Statische Analyse, Verifikation
- Prozessqualität:
  - > Softwareinfrastruktur: Konfigurationsmanagement, Build-Automatisierung, Test-Automatisierung

# Was kann man gegen schlechte Softwarequalität tun? (2): ⑥

## - Prozessqualität (2):

> Managementprozesse: Vorgehensmodelle, Reifegradmodelle

## lexikalische und syntaktische Fehler:

- Fehler, die der Compiler nicht erkennt
- z. B. falsches Verwenden von Schleifenzählern, ...

## Parallelität als Fehlerquelle:

- z. B. Zugriff auf gleiche Ressource zur gleichen Zeit

## Numerische Fehler:

- Rechnungsgenauigkeiten bei double z. B.

## Portabilitätsfehler:

- z. B. Probleme mit Typkonvertierung von 1 auf die andere Plattform

## Optimierungsfehler:

- Diverse Verkürzungen, die nicht das tun, was gewünscht ist

## Spezifikationsfehler:

- Falsche Angaben in Spezifikationen

## Fehlerbewertung:

> Anwender: Interesse an Abwesenheit von Fehlern

> Entwickler: " " am Aufdecken " " "

## Softwarerichtlinien:

- Regeln Gebrauch der Programmiersprache über Syntax + Semantik hinaus  $\Rightarrow$  Vereinheitlichung + Fehlerreduktion
- $\Rightarrow$  unterteilbar in Notations- und Sprachkonventionen

## Notationskonventionen:

- Definition auf verschiedenen Ebenen (Projekt, Betriebssystem, ...)
- Auswahl + Schreibweisen von Bezeichnern
- Einrückung / Verwenden von Kennzeichen
- Dokumentation
- Notationsstile:
  - > Pascal Case: some Method
  - > Camel Case: someMethod
  - > Uppercase: SOME...
  - > lowercase: some ...

## ungarische Notation: $\rightarrow$ Totaler Bullshit...!

- > Apps Hungarian: < Präfix > < Datentyp > < Bezeichner >
  - > Systems " " : < Präfix > < Bezeichner >
- ↓  
Name
- z.B. i  $\dagger$  index  
p  $\dagger$  pointer  
⋮
- z.B. d  $\dagger$  double  
i  $\dagger$  int  
⋮

## Sprachkonventionen:

- Semantik einer Sprache (z.B. MISRA-C)
- MISRA-C unterstützt C-Entwickler (sicherheitskritische Systeme)
- Umfang:
  - Empfehlungen zu: Tool Selection, Projektaktivitäten, Implementierung von MISRA-Compliance
  - Guidelines  $\rightarrow$  z.B. Kein GOTO verwenden

## Code reviews (Gefahren):

- Die, die die Konventionen kennen, sind die "Bösen"
- Gezwistete Person fühlt sich angegriffen
- Reviewer will deswegen nichts mehr annehmen

## Durchsetzung von Konventionen:

- Alle Beteiligten bei Erarbeitung beteiligen
- Gemeinsame Weiterentwicklung
- Ausnahmen dokumentieren (Begründet)
- Toolgestützt rüsten
- Regeln zu Beginn vereinbaren
- mit Bedacht ändern

## Typisierung:

- Compiler kann korrekte Verwendung der Datentypen rüsten
- Frühzeitige Erkennung von Fehlern
- Verständlichkeit des Codes wird verbessert

Typumwandlungen + Erkennung von Typverletzungen

## ⇒ Typprüfung:

- statische: zur Compilezeit → Hardwareschicht Einsatz}
  - dynamische: während der Ausführung
- Höhere Pro-grammier-sprachen

## Typisierung:

- statische: Declaration des Typs im Code bei der Variable
- dynamische: Variablen sind nicht typpräzise → z.B. PHP
- schwache: Typecasts zur Uminterpretierung beliebig möglich
- starke: stets typkonforme Zugriffe

## Generics:

- dynamischer Typ, hierbei Erhöhung der Typsicherheit durch Containerdatentypen

## Vertragsbasierte Programmierung (Design by contract): (9)

- Vereinbarungen werden vor Implementierung festgeschrieben (für Modulschnittstellen)
- Vertrag zwischen Auftragendem und Auftrüger, der Bedingungen + Fehlbehandlung abdeckt → Invarianten
- Grundprinzipien:
  - > Vorbedingungen: Zusicherungen, die der Auftrüger zu beachten hat
  - > Nachbedingungen: Zusicherungen, die der Auftrüger zu beachten hat
  - > Invarianten: Gesundheitszustand der Klasse (logische Aussagen, die immer gelten für diese Klasse)

Verletzung der Bedingungen führt zum Abbrechen des Programms

### ⇒ Nutzen:

- Qualitätssteigerung durch genaue Schnittstellendefinition
- Dokumentation
- Unterstützung beim Testen / Fehlerfinden
- In manchen Sprachen nativ, oft per Erweiterung

## DBC und Vererbung:

- Vorbedingungen können in geerbten Methoden gelöst werden
- Nachbedingungen + Invarianten können stärker gemacht werden in geerbten Methoden
- Zusätzliche Einschränkungen sind möglich !! !! !!

## Fehlertolerante Programmierung:

Möglichst kein kompletter Softwareabzug

Softwareredundanz:

- Funktionale Redundanz: zusätzliche Funktionen zur Erhöhung der Fehlertoleranz
- Informationelle Redundanz: Nutzdaten um zusätzliche Informationen anzusiedeln
- Temporale Redundanz: Zeitanforderungen werden überstellt, Wiederholung wäre möglich im Notfall
- Strukturelle Redundanz: Mehrfachauslegung von Komponenten
  - > Homogene: n Komponenten gleicher Bauart  $\rightarrow$  Nur Hardware
  - > Heterogene: n Komponenten unterschiedlicher Bauart
  - > Statische: alle Komponenten aktiv, Ergebnisse werden durch Voter verglichen
  - > Dynamische: Ersatzkomponenten, die bei Bedarf aktiviert werden

Selbstüberwachende Systeme:

$\Rightarrow$  Reaktionsszenarien:

- Fail-Safe-Reaktion: Wechsel in einen sicheren Zustand, z.B. Selbstaktivierung einer Notbremsung eines Autoges
- Selbstreparatur: Selbstdiagnose und Reparatur, z.B. automatische Dateiwiederherstellung
- Reaktivierung: watchdoglogik

Exceptions:

- Benutzerfehler: Client errors
- Programmierfehler: Internal errors
- Ressourcenzugriffsfehler: Service errors
- Philosophien: Delegation aus Betriebssystem
  - $\Rightarrow$  Behandlung in der Programmiersprache

Ausnahmehandlung:

1. hoggen der Ausnahme nahe der Stelle, an der sie auftritt.
2. An den Aufrufer weiterreichen
3. Behandlung der Ausnahme an der Stelle, an der man die Folgen + angemessene Reaktion erkennen kann
4. Angemessene Reaktion:
  - Erwartete Eingabe durch Benutzer
  - Datei schließen
  - Programm beenden
  - :

Vorteile: - Trennung Anwendung  $\Leftrightarrow$  Exceptionhandlung

- Kein Ignorieren von Exceptions möglich
- Weitergabe an Aufrufer einfach möglich
- Saubere Gruppierung von Fehlern

Java Exceptions:

- Checked: Compiler prüft, ob Code zum Auffangen der Exception vorhanden ist
  - Unchecked: können behandelt werden, müssen aber nicht  $\rightarrow$  Sinn ??
- $\hookrightarrow$  können schwere Fehler verursachen

Gutes/schlechtes Exceptionhandling:

- Gute:
  - > einfache Entwicklung + Wartung
  - > Weniger Bugs  $\rightarrow$  ? Nicht widelich
  - > einfache Anwendung
- Schlechtes:

> Verwirrung der Benutzer

> Schwere Wartung

## Perspektiven beim Exceptionhandling:

- Klasse, die die Exception wirft
- " " " " " fängt
- Benutzer, der mit dem Fehler umgehen muss
- Entwickler / Support

## Möglichkeiten des Designs von Exceptionhandlings:

- Methodensignatur: ... throws IOException, ...
- Eigene Exception, die die anderen Exceptions als „InnerException“ kapselt. (Exception chaining)

## Unterteilung von Exceptions:

- ApplicationException: Fehler auf Geschäftslogikebene
- SystemException: Fehler auf Systemebene

## Exceptions für Benutzer:

- Keine technischen Details
- Sprechende Meldung
- Globalisiert

## Anforderungen an Exceptionhandling:

- Primär: > Kein Absturz der Applikation → Fehler fangen + sauber schließen
  - > Information des Entwicklers
  - > Fehlerdiagnose + Reproduktion → Wo ist der Fehler aufgetreten + Welcher Kontext?
- Sekundär: > Abstraktion
  - > Lesbarer + Wartbarer Code → Welcher Weg hat dorthin geführt
  - Verbergen (z.B. Custom exceptions)

Localization: Wo ist Fehler aufgetreten?

\* Context: Ausführungsplatz

## Typische Reaktionen beim Exceptionhandling:

(13)

- Abbruch der Reaktion
- Schließen geöffneter Ressourcen
- Freigeben von Speicher
- Nutzer benachrichtigen
- Loggen des Fehlers

## Strategie beim Exceptionhandling:

- Fehler entdecken
  - Informationen über den Fehler sammeln
  - Exception wofan
  - Weitergeben an Aufrufer + Zusatzinfos
  - Fangen + Reaktion:
    - > Neuer Versuch?
    - > Information an Benutzer / Entwickler?
- ⇒ Benutzung von Templates sinnvoll.

## Vernwendung der AppException:

- Vernwendung von Severity und ErrorType
- Bei mehreren „Inner Exceptions“ letzte verwenden
- Beschreibung für Benutzer:
  - > Client error: Beschreibung + Anleitung zur Korrektur
  - > Internal error: Keine Beschreibung, Standardfehlermeldung
- Beschreibung für Entwickler: z.B. XML

## Portierung: → Portabilität = Plattformunabhängigkeit

- Architekturenportierung (auf andere Hardware)
- Betriebssystemportierungen
- Systemportierungen (auf andere Geräteliste)

Documentation:

- External Documentation (Für Kunden)

- Internal " "

  - > Konventionen, Richtlinien

  - > Programm documentation

    - Spezifikationsdocumentation

    - Implementierungsdokumentation

  - > Marktstudien / Strategiepapiere

Spezifikationsdokumentation:

- Criteria:

  - > Vollständig

  - > eindeutig

  - > Widerspruchsfrei

  - > Verständlich

- Variants:

  - > Informal ⇒ implizite Annahmen, Ignorieren von Sonderfällen, Sprachliche Ungenauigkeiten

  - > Semiformal: „mathematische“ Definition, Verwendung definierter Formalismen, gut lesbar, wenig (bis keine) Zweideutigkeiten, Beispiel UML

  - > Formal: Kein Interpretationsspielraum, oft sehr komplex, In der Praxis nahezu irrelevant, z.B. Z

  - > Referenzimplementierung: Programm wird so spezifiziert, dass es das gleiche tun soll wie die Referenz. D.h. „neues Verhalten = altes Verhalten“. } Oft nicht ausreichend

## UML-Diagrammtypen:

### - Strukturdigramm

> Klassendiagramm

> Objekt- "

> Paket - "

> Architekturdiagramm

→ Kompositionsdigramm

→ Komponentendiagramm

→ Einsatz- und Verteilungsdiagramm

### - Verhaltensdiagramm

> Usecasediagramm

> Aktivitäts- "

> Zustands - "

### - Interaktionsdiagramm

> Sequenzdiagramm

> Kommunikations- "

> Interaktionsübersicht

> Zeitdiagramm

## Implementierungsdokumentation:

- "Our code is our documentation" ⇒ Oft nicht gut

- Regeln für Kommentare:

1. Code so klar wie möglich ⇒ Wenig Kommentare nötig.

2. Niemals das gleiche in den Kommentaren schreiben,  
was bereits im Code steht.

3. Kommentare präzise und vollständig formulieren.

(4. Grammatik und Buchstabenbildung richtig verwenden)

5. Kommentare sollten vom Code visuell abgeho-

ben werden → Macht doch die IDE? \*

6. Strukturierte Kommentare → Automatisierung ✓

Sollte  
selbstver-  
ständlich  
sein?

Dokumentation:

- Relevant für Entwickler, Benutzer, „Weiterentwickler“
- Dokumentation für alle lesbar (auch, wer den Code nicht hat) → ohne Code lesbar
- Dokumentation in aufbereiteter Form (HTML, PDF, ...)

Javadoc:

- Integriert Dokumentation in Java-Code
  - Automatische Generation der Dokumentation möglich
- ```
/**  
 * ...  
 */
```

Doxygen:

- Output in Text-Format
- Klassendiagramme
- Source-Code-Browsing
- :

Testtypen:

- Prüfebene:
  - > Anwendungsebene
  - > Systemebene
  - > Integrationsebene
  - > Unit-Ebene
- Prüfmethode:
  - > Black-box
  - > White-box
  - > Grey-box
- Prüfleiterium:
  - > Funktional
  - > Operational
  - > Temporal

Prüfebenen:

- **Unitests:** Test von eigenständigen atomaren Einheiten
- **Integrationstests:** Zusammenspiel mehrerer Einheiten wird getestet
- **Systemtests:** Test des Systems als Ganzes
- **Ablaufmetests:** Systemtest in der Umgebung des Auftraggebers  
↳ sind juristisch relevant

Unitest:

- 2 ATen: "richtige" Eingaben vs. "falsche" Eingaben

Integrationsstrategien:

- **Big - Bang - Integration:**

- Integration auf einmal
- Vorteil: Fehler werden schnell gefunden
- Nachteil: Fehlersuche schwierig

- **Strukturorientierte Integration:**

- Incrementelle Integration der Module
- Bottom - Up: Ausgang von Basiskomponenten
- Top - Down: " von „hohen“ Modulen
- Outside - In: Integration von beiden Seiten nach innen
- Inside - Out. " " innen " außen

- **Funktionsorientierte Integration:**

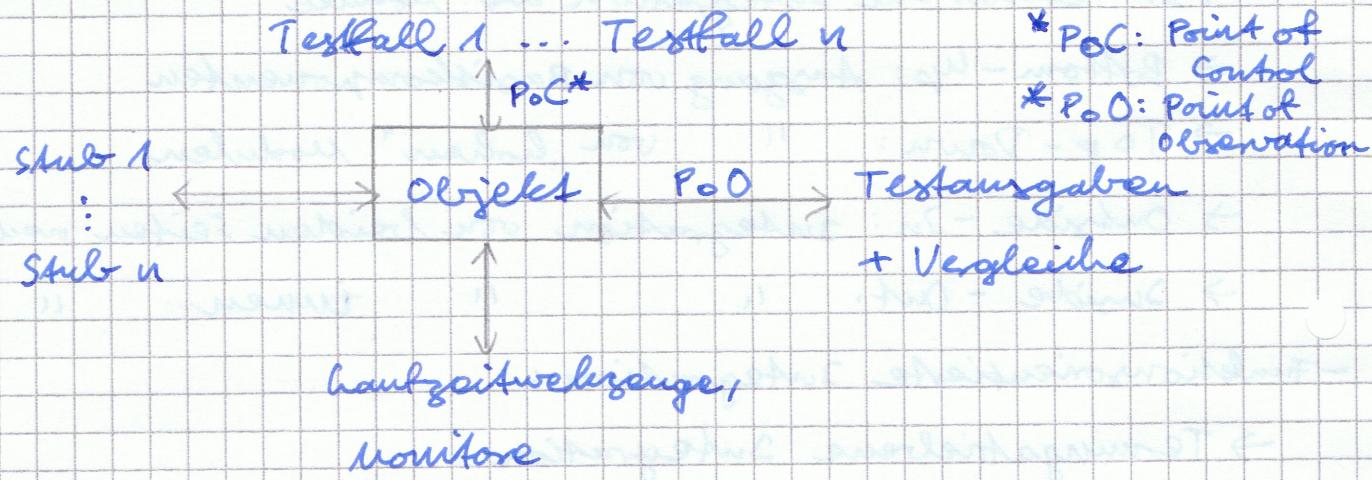
- Termingetriebene Integration
- Risikogetriebene " "
- Testgetriebene " "
- Anwendungsgetriebene " "

Felddat - Produkte für Massenmarkt:

- **Alpha - Tests:** Beim Hersteller. Durchführung durch ausgewählte Anwender
- **Beta - Tests:** In der Umgebung des Kunden. Häufig inkrementell durchgeführt mit immer größerem Nutzerkreis

Prüfkriterien:

- Funktional:
  - Funktions test
  - Trivial test
  - Grahl test
  - Kompatibilitäts test
  - zufallstest
- Operational:
  - Installationstest
  - Ergonomietest
  - Sicherheitstest
- Temporal:
  - Komplexitätstest
  - Laufzeit test
  - Last test
  - Stress test

Testrahmen:Prüftechniken:

- Blackbox - Test: Tests aus Schnittstellenbeschreibung
- Whitebox - Test: Tests werden aus innerer Programmstruktur abgeleitet
- Greybox - Test: „Beides zusammen“

## Blackbox - Testtechniken:

(19)

- Äquivalenzklassentest: Bilden von Testklassen  
(z.B. [0, 10] und [11, 23] ...)

⇒ Es gibt auch mehrdimensionale Äquivalenzklassen

⇒ Vollständige Partitionierung: ungültige Werte mit einbeziehen (Partielle Aut dies nicht)

- Grenzwertbefragung:

→ Ähnlich Äquivalenzklassentest

→ Unterschied: Auswahl der Randwerte, eines Wertes in der Klasse und eines Wertes außerhalb der Klasse

- Zustandsbasierter Test:

→ Alle möglichen Übergänge testen

→ Übergangsbaum (für Zustände) zeichnen

→ GUI-Masken können in Zustände eingeteilt werden

⇒ eignet sich gut für GUIs

- Use-Case-Test: Use-Case-basierter Blackbox-Test

- Entscheidungstabellen-basierter Test:

→ Ursache-Wirkungsgraph erstellen

→ Entscheidungstabelle ableiten

→ Beispiel Bankautomat:

| Tabelle         |                  | TF 1 | TF 2 | ... | TF 5 |
|-----------------|------------------|------|------|-----|------|
| Bedingungen     | Karte<br>gültig? | Nein | Ja   | --  | --   |
| PIN<br>korrekt? | Ja               |      | --   | --  | Nein |

## Blackbox - Testtechniken (Fortsiedlung):

(20)

### - Paarweises Testen:

→ Statt alle Möglichkeiten zu testen: Auswahl von bestimmten Kombinationen, die repräsentativ sind

→ Wird sehr schnell sehr komplex

### - Diversifizierende Tests:

→ Vergleich verschiedener Programmversionen

### → Back-to-Back-Tests:

- Vergleicht mehrere Varianten nach der Modifikation
- Ziel: Abweichungen + deren Ursache finden

### → Regressionstests:

- Wiederholter Test eines Programmteils nach Änderung
- Ziel: Zeigen, dass keine neuen Fehler eingeschaut wurden

### → Mutationstests:

- Technik, um Testwahrscheinlichkeiten zu verdeutlichen

## Whitebox - Testtechniken:

→ Kontrollflussorientierte Tests: Konstruktion der Testfälle aufgrund der internen Berechnungsstruktur

→ Datenflussorientierte Tests: Heranziehen zusätzlicher Kriterien

### ⇒ Kontrollflussgraphen erstellen:

– Knotenmarkierte Kontrollflussgraphen

(- Knoten " "

" "

) ⇒ für Required-2-Tupel-Tests

### ⇒ Weitere Unterteilung:

– Expandiert: Je ein Befehl pro Knoten

– Teilkollabiert: Teilweise 2 oder mehr Befehle pro Knoten\*

– Kollabiert: überall " " " " " " " "

schlüsselblöcke

\* müssen verzweigungsfrei sein

## White-Box - Testvaloren (Fortschreibung):

(21)

- Überdeckungsarten:

→ Anweisungsüberdeckung:

⇒ jeder Knoten des Kontrollflussgraphen\* wird mindestens einmal durchlaufen

⇒ „ $C_0$ -Test“

⇒ schwer erreichbar, da aufwendig (Bei großen Projekten)

⇒ 100% Abdeckung nötig, sonst nicht aussagekräftig

→ Zweigüberdeckung:

⇒ „ $C_1$ -Test“

⇒ jede Kante des KFG\* muss mindestens einmal durchlaufen werden

⇒ „Minimalkritium“

→ Pfadüberdeckung:

⇒ Für jeden Pfad vom Eingangsknoten zum Ausgangsknoten 1 Testfall

⇒ Pfadanzahl steigt oft exponentiell

⇒ Varianten:

- Boundary-Interior-Pfadüberdeckung\*\*:

- 3 Gruppen von Testfällen:

- > äußere Pfade: Schleifen nicht betreten

- > Grenzpfade: genau eine Iteration

- > innere Pfade: mindestens eine weitere Iteration

- Strukturierte Pfadüberdeckung:

- > Verallgemeinerung von BIP\*\*

- > Alle möglichen Ablaufpfade bis zur  $k$ -ten Schleifeniteration durchlaufen

- ↓
  - > am besten nur für Schleifen, die keine inneren Schleifen mehr enthalten

## Whitebox - Testverfahren (Fortsetzung):

- Überdeckungsarten:

→ Bedingungsüberdeckung:

⇒ zusätzlich zum Kontrollflussgraph: Einbeziehung der logischen Struktur der if-Bedingungen

⇒ einfache Bedingungsüberdeckung:

"Alle atomaren Prädikate müssen mindestens einmal beide Wahrheitswerte annehmen"

$$A=0 \quad B=1$$

$$A=1 \quad B=0$$

⇒ minimale Mehrfachbedingungsüberdeckung:

"Alle atomaren und zusammengesetzten Prädikate müssen mindestens einmal beide ..."

$$A=0 \quad B=0$$

$$A=0 \quad B=1$$

$$A=1 \quad B=0$$

⇒ mehrfachbedingungsüberdeckung:

"Alle Kombinationen"

$$A=0 \quad B=0$$

$$A=0 \quad B=1$$

$$A=1 \quad B=0$$

$$A=1 \quad B=1$$

→ McCabe - Überdeckung: Kontrollflussonieniert

→ Defs - Uses - " : Ableitung der Testfälle aus dem Datenfluss

→ Required - k - Typel - " : Ebenfalls " "

→ sehr komplex, werden selten verwendet.

## Textmetriken:

- Wird ein Modul ausreichend getestet?
- Wie viele unentdeckte Fehler gibt es?
- Wie gut ist das Testverfahren?

⇒ Wesentlich relevante Metriken:

> Anweisungsüberdeckung:

$$MC_0 = (\text{Anzahl überdeckte Knoten} / \text{Anzahl Knoten}) * 100$$

> Zweigüberdeckung:

$$MC_1 = (\text{Anzahl überdeckte Kanten} / \text{Anzahl Kanten}) * 100$$

- Verwendung: Modulabschreben, Modulvergleiche

- z.B. GCOV, LCOV, Cobertura, ...

## Mutationstests:

→ Zufällig Fehler einfügen und prüfen, wie viele davon im Test gefunden werden

## Grenzen der Softwaretests:

→ Oft schwierig und aufwändig

⇒ Fehlende Anforderungen

⇒ Komplexität des Programms

⇒ Mangelnde Werkzeugunterstützung

⇒ Zeitprobleme

:



Gründe

## JUnit:

- Testautomatisierung

- Test Suites: Aufführen von Tests aus mehreren Klassen

- Fixtures: Vorberichtungs- / Nachbereitungscode für Tests

↓  
@after: Jedes nach dem Test (auch wenn Exception auftritt)

@before: " vor " "

- Rules: Wiederverwendbare Fixtures ("Temporärer Ordner")

- Categories: Gruppierung von Testmethoden

## JUnit (Fortszung):

- (24)
- Theories: Allgemeine Aussage formulieren und durch Reihe von Testwörtern überprüfen
  - Runner:
    - Block JUnit 4 Class Runner: Default
    - Suite: Für mehrere Klassen
    - Parameterized: Runner für parametrisierte Tests
    - Categories: Runner für Sammlungen von Tests
    - Theories: Runner für Theories

## Statistische Codeanalyse:

- Automatisiert und manuell möglich
- Vorteile:
  - + untestiger Code kann getestet werden
  - + Beim Testen (Unitests) können sich Fehler überdecken
  - + Prüfung von Qualitätsmerkmalen zusätzlich
- Nachteile:
  - Test der Spezifikation, nicht der Benutzeranforderungen
  - Performance / Usability, ... schwer überprüfbar

## ⇒ Manuelle Analyse:

1. Planning: Was wird gescannt? Wer reviewt? Kriterien?
2. Einführung: Alle erhalten Infos + Zweck erklären
3. Vorbereitung: Reviewer bereitet sich auf Sitzung vor
4. Reviewsituation: Moderator moderiert diese, Prüfobjekt wird bewertet ⇒ Ergebnis ⇒ alle Gutachter müssen das Ergebnis tragen
5. Überarbeitung: Autor arbeitet gewünschte Änderungen ein
6. Nachbereitung: Kontrolle der Änderungen

- ⇒ Arten:
- |               |                      |
|---------------|----------------------|
| - Walkthrough | - Technisches Review |
| - Inspektion  | - Informelles Review |

## Softwaremetriken:

### - Prozessmetriken:

- > Zeitverbrauch
- > Ressourcenverbrauch
- > Häufigkeit von Ereignissen

### - Produktmetriken:

> dynamisch

> statisch:

- konventionell :

→ Umfangmetriken

→ Logische Strukturmetriken

→ Datenstrukturmetriken

→ Stilmetriken

- objektorientiert:

→ Vererbungshierarchien

→ Methodenebene

→ Klassenebene

→ Aggregationshierarchien

### Probleme:

- Nutzen in der Praxis unklar (oft)
- Keine Standards

## Gütekriterien für Softwaremetriken:

- Objektivität : Objektiv
- Robustheit : Bei Wiederholung gleiches Ergebnis
- Vergleichbar : Vergleichbarkeit der Metriken
- Ökonomisch : Billig
- Korrelation : Aussage im Bezug auf Kenngröße Handels

# Goal - Question - Metric - Metrics:

(40)

Goal  $\longleftrightarrow$  Goal Attainment

Question  $\longleftrightarrow$  Answer

Metric  $\longleftrightarrow$  Measurement

## Einfache Metriken:

- LOC: Lines of Code
- NCSS: Non Commented Source Statements  $\equiv$  LOC-Kommentare  
 $\Rightarrow$  schlecht vergleichbar

## Halsteadmetriken:

- Basieren auf lexikalischer Struktur des Programms auf
- Berechnungen auf Basis empirischer Zusammenhänge
- zerlegen Quelltext in Operatoren und Operanden
- Basisgrößen:

$\rightarrow N_1 = \text{Anzahl unterschiedlicher Operatoren}$

$N_2 = \text{Anzahl Operanden}$

$N_1 = \text{Gesamtzahl der Vorkommen aller Operatoren}$

$N_2 = \text{Anzahl Operanden}$

$N_2^* = \text{Anzahl der Ein- und Ausgabeoperanden}$

$$\frac{N_1 + N_2}{2}, \frac{N_1 + N_2}{2}$$

$\Rightarrow$  Metriken wie: program volume:  $V = N \cdot \log_2 (n)$

$\vdots$

$\Rightarrow$  Vorteile: + Einfach

+ Automatisierbar

+ Empirisch als gut bestätigt

$\Rightarrow$  Nachteile: - Ausschließlich lexikalische Analyse

- Aufteilung Operator / Operand sprachabhängig

## McCabe-Metrik:

(L+)

- Zyklotomatische Komplexität
  - Basierend auf Graphentheorie
  - $\geq$  Anzahl unabhängiger Pfade im Kontrollflussgraphen
  - $\geq$  mindestens nötige Anzahl an Testfällen zur vollen Überdeckung
  - $V(G) = e - n + 2 * k$
- ↓              ↓              → Unabhängige Teile des Graphen  
 Anzahl Kanten    Anzahl Knoten

|            |         |              |      |
|------------|---------|--------------|------|
| ⇒ Klassen: | 1 - 10  | ✓ Einfach    | V(G) |
|            | 11 - 20 | komplexer    |      |
|            | 21 - 50 | sehr komplex |      |
|            | 50 +    | "Untestbar"  |      |

- ⇒ Pro+:
- + einfach zu berechnen
  - + gut geeignet zur Berechnung des Testaufwands
- Con:
- Berücksichtigt nur Kontrollfluss
  - Unterschiedliche Implementierungen

⇒ 2 Arten:

- Methodenbasiert: Kontrollflussgraph in Methode wird untersucht (Normalfall)
- Prozessbasiert: " in Prozess wird  
pist dann Anzahl analysiert ⇒ gest. Unterteilung nötig  
der Methoden

## Objektorientierte Metriken:

- Komponentenmetriken: Bewertung einzelner Klassen, Pakete
- Systemmetriken: Bewertung des ganzen Projekts

⇒ Komponentenmetriken:

- z.B.  $OV$ : Anzahl Objektvariablen }  $NOT$ : Summe davon  
 $CV$ : " Klassen " } umfangsmetriken

## Komponentenmetriken (Fortsetzung):

(48)

- Weighted methods per class: Summe der Gewichtung der Einzelmethoden
- Depth of inheritance tree (DIT): Größter Abstand der Klasse von der Wurzel des Vererbungsbaums
- Number of descendants (NOD): Zahl der Kinder, die von der betrachteten Klasse erben
- Coupling between object classes (CBO): Anzahl der Klassen, die an die betrachtete Klasse gekoppelt sind
- :

## Strukturmetriken:

- ⇒ Fan-In: Anzahl der Module, die auf das betrachtete Modul zugreifen
- ⇒ Fan-Out: Anzahl der Module, auf die dieses Modul zugreift
- ⇒ Bsp:
  - Henry und Kafura:  $C_m = (\text{Fan-In} * \text{Fan-Out})^2$   
Komplexität des Moduls
  - Henry und Selig:  $C_m = C_{int} * (\text{Fan-In} * \text{Fan-Out})$   
Intern Komplexität  $\Rightarrow$  Bestimmung über z.B. Halstead

## Konformitätsanalyse:

- ⇒ Auch semantische Analyse des Codes
- ⇒ z.B. Splint

## Exploitanalyse:

- ⇒ Analyse für sicherheitskritische Anwendungen

## Anomalienanalyse:

- ⇒ Suche nach auffälligen Codesequenzen
- ⇒ Anomalien:
  - Kontrollflussanomalien:  $\Rightarrow$  Unstimmigkeiten im Programmablauf
  - Datenflussanomalien: zwei Rollback Variablenau-

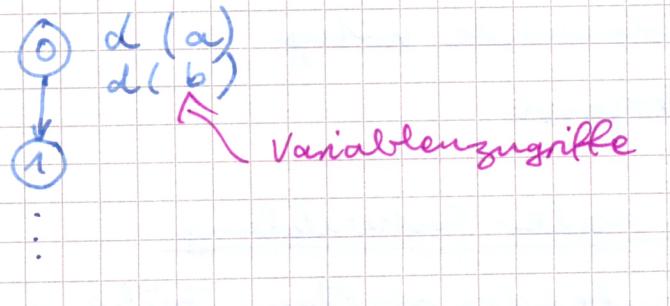
## Datenflussanomalieerkennung:

→ Datenflussgraphen erstellen:

d = schreiben

r = lesen

u = unbesetzt



Kritische: dd : 2 mal Schreiben hintereinander auf gleiche Variable

du : Variable wird geschrieben und dann gelöscht

ur : Lesen von undefinierter Variable

## Tätigkeiten / Abläufe in einem Software-Projekt:

1. Analyse: Systemspezifikation

2. Design: Technisches Design

3. Implementierung + Modultests: Code, Testklassen, Doku, ...

4. Test + Integration: Testberichte, lauffähige Software

5. Abnahme + Einführung: Abnahmegerüste / Testprotokolle

6. Betrieb + Wartung

⇒ es gibt kein ideales Vorgehensmodell für alle Projekte

## Wichtige Vorgehensmodelle:

- Code and Fix
- - Agile Methoden
- Wasserfallmodell
- Prototyping
- V-Modell
- Rational Unified Process (RUP)
- Spiralmodell
- Iterative Entwicklung

## Kritik an klassischen Modellen:

(30)

- Oft schwer verständlich
  - Viele Dokumente nötig
- ⇒ Agile Prozesse

## Prinzipien agiler Entwicklung:

- Kunden zufriedenstellen: Frühe + kontinuierliche Testlieferungen von Software
- Selbst große Änderungen sind spät noch möglich
- Tägliche Zusammenarbeit Entwickler + "Fachexperten"
- Motivation der Individuen durch Vertrauen
- Informationsweiterleitung durch persönliche Gespräche
- Wichtigstes Fortschrittsmaß: Funktionierende Software
- Nachhaltige Entwicklung
- Gleichmäßiges Entwicklungstempo
- Gutes Design

## ⇒ Kritik:

- Wenig Regeln
- Angst, sich zu blamieren
- „Save your ass“ - Mentalität

## Scrum:

- Transparenz / Überprüfung / Anpassung
- Rollen: Product Owner, Scrum Master, Entwicklungsteam, (Customer, Manager, User)
- Meetings: Sprint Planning (1+2), Daily Scrum, Sprint Review, Sprint Retrospektive, (Estimation meeting)
- Artefakte: Product Backlog, Sprint Backlog, Product Increment, (Vision, Sprint Goals, Tasks, Releaseplan, Impediment Backlog)
- Dokumentation & Tools

## Scrum (Fortszung):

- Selbstorganisation
- Interdisziplinarität
- Flexibilität
- Kreativität
- Produktivität

## 4 Arbeitsprinzipien:

- Selbstorganisation
- Pull Prinzip
- Timebox
- Nutzbare Funktionalität

## Product Owner:

- Verantwortlich, dass die Entwickler gut arbeiten
- Management des Product Backlogs

## Entwicklungsteam:

- Entwickelt bis zum Sprint ein Increment der Software (lauffähig)
- Erstellt Product - Increment

## Scrum Master:

- Sorgt dafür, dass die Scrum - Regeln eingehalten werden

## Customer:

- Auftraggeber
- Finanziert das Projekt

## Manager:

- Stellt Ressourcen / Richtlinien innerhalb der Organisation bereit

## User:

- Wesentliche Informationsquelle für das Scrum Team

## Scrum (Ablauf):

→ Sprint Planning 1 und 2:

⇒ Sprint Backlog füllen (aus Product Backlog)

→ Daily Scrum

→ Sprint Meeting, 2 bis 4 Wochen

⇒ Potentiell auslieferbares Produkt

⇒ Sprint Review Meeting

⇒ Retrospektive

## Sprints:

- Umbau: > Sprint Planning

> Daily Scrums

> Entwicklung

> Sprint Review

> Sprint Retrospektive

- während des Sprints:

→ keine Änderungen, die das Sprintziel gefährden

→ keine Qualität schmälen

## Sprint Planning:

⇒ Planung der Arbeit für nächsten Sprint

⇒ oft zweigeteilt:

1. „Anforderungsworkshop“ ⇒ Selected Product Backlog

2. „Designworkshop“ ⇒ Liste der Tasks für nächsten Sprint

## Daily Scrum:

- Was habe ich gestern getan?

- Was werde ich heute tun?

- Was hindert mich an meiner Arbeit?

## Sprint Review:

⇒ Diskussion der Ergebnisse des Sprints

## Retrospektive:

⇒ Scrum - Team kann sich selbst verbessern  
+ Verbesserungen für nächsten Sprint vorschlagen

## Estimation meeting:

→ Backlog wird einmal (oder mehr) pro Sprint neu priorisiert

## Schätzen von Storypoints:

- Je höher der Aufwand, desto höher der Nutzen
- Einheit aber nicht festgelegt
- Skala: Häufig Fibonacci-Zahlen

⇒ Vorgehen:

> Planning Poker:

1. Einigung auf Referenz - Backlog - Item
2. Schätzen von Storypoints
3. Wiederholung bis eindeutiges Ergebnis vorliegt

> magic Estimation:

1. Backlogitems zu Prioritäten ordnen
2. Jeder kann verschieben
3. Wiederholung bis eindeutiges Ergebnis vorliegt

⇒ Oft unscharf, d.h. deshalb neue Schätzungen in Referenz zu bisherigen Sprints setzen

## Definition of done:

enthält alle fertiggestellten Backlogeinträge

Stimmt nicht, Definition of done wird pro Firma festgelegt und spezifiziert, wann ein Backlogeintrag fertiggestellt ist.

Scrum - Fortschritt:

- Taskboard
- Charts
- Velocity berechnen (Storypoints/Sprint)

Scrum in großen Projekten:

- ⇒ mehr als ein Scrum Team (Vermeiden, wenn es geht)  
 ⇒ 2 Asten:

→ organisches Wachstum:

1. Neue Mitarbeiter einarbeiten

2. Team entscheidet, wann es sich teilt

→ Sprunghaftes Skalieren:

→ Mitglieder des initialen Teams übernehmen die Rolle des Subproduct Owners

⇒ 2 Möglichkeiten zur Teamaufteilung:

- Component Teams: Technische Komponenten
- Feature Teams: Fachliche Funktionen

⇒ Synchronisation der Teams:

> Scrum of Scrums

> Product Owner Team

> Scrum Master Group

> Virtuelle Teams

> Planning Meetings

Vertragsmodelle:

<sup>wahl</sup>

⇒ Hängt von Faktoren ab:

- Projektart (Neuentwicklung, Anpassung)
- Sicherheitsbedürfnis + Planbarkeit (stabiler Preis)
- Konkurrenzfähigkeit (niedriger Preis)

Nicht  
relevant  
!!

## Vertragsmodelle (Fortsetzung):

- Gewinnchance (für Auftragnehmer)
- Nutzenerwartung (Nutzen soll größer als Kosten sein)

Nicht  
rele-  
vant  
!!!

### Prinzipielle Unterscheidung:

- Festpreis
- Aufwandspreis (Time + Material)

### Varianten:

- Aufwandspreis mit Obergrenze
- Phasenfestpreis
- Agiler Festpreis
- Festpreis mit inhaltlichen Spielraum

### Auftraggeber vs. Auftragnehmer:

- Auftraggeberinteressen:
    - ⇒ Tendenz zum Festpreis (Risiko beim Auftragnehmer)
  - Auftragnehmerinteressen:
    - ⇒ Tendenz zu Aufwandsvertrag (Risiko beim Auftraggeber)
- ⇒ Lösung: Agiler Festpreis als neues Vertragsmodell.

### Agiler Festpreis:

- Wasserfall: Plan Driven
- Agil: Value / Vision Driven

### Aufen von Anforderungen:

- Funktionale Anforderungen: Anforderungen bezüglich des Ergebnisses eines Verhaltens einer Funktion
- Qualitätsanforderungen: Anforderungen, die sich auf ein Qualitätsmerkmal beziehen

## Arten von Anforderungen (Fortsetzung):

- Randbedingungen: Anforderung, die den Lösungsraum so einschränkt, dass funktionale Anforderungen und Qualitätsmerkmale erfüllt werden

## Requirements Engineering:

- Systematischer Ansatz zur Spezifikation von Anforderungen
- Die relevanten Anforderungen kennen, Konsens unter den Stakeholdern darüber herstellen
- Standards berücksichtigen
- Bedürfnisse der Stakeholder verstehen und dokumentieren

⇒ Im Vorgehensmodell:

- Wasserfall: Eigene Projektphase
- Agile Methoden: Kontinuierlicher phasenübergreifender Prozess
  - ⇒ Product Backlog Items

## User Stories:

- Karte
- Konversation
- Akzeptanzkriterien

## Epic:

- Sehr große User Stories, die schwer geschwäzt und nicht in einem Sprint bearbeitet werden können

## Thema:

- Zusammenfassung mehrerer User Stories

Backlog Items:

- Product Backlog Items:
  - Themes
  - Epics
  - user Stories
- für das Sprint Planning : User Stories

User Stories:

- In der Sprache des Kunden
- Verwendung von Benutzerrollen
- Keine Technik
- Keine Benutzeroberfläche
- Verwendung eines Templates

Colossale Form für User Stories:

- Verwendung der Ich - Form zur besseren Identifikation mit der User Story
- Einheitliche Struktur hilft beim Priorisieren
- Sinnvolle Struktur

Schneiden von User Stories:

- Vertikales Schneiden
- Schneiden nach Daten
- " " nach Aufwand
- " " " Forschungsanteilen
- " " " Qualität
- " " " Benutzerrolle
- " " " Akzeptanzkriterien
- " " " technischer Voraussetzung

## Anderer Themen in User Stories packen:

- Nichtfunktionale Anforderungen
- Fehler
- Refactorings
- Bugtrackingssystem einsetzen
- ...

⇒ Umformulieren, dass der Zweck klar wird

⇒ Hilfsmittel:

- Zusätzliche künstliche Benutzervollen
- Constraints definieren (Storyübergreifende nicht-funktionale Anforderungen)
- Fehler:
  - Sofort beheben
  - In nächsten Sprint einplanen

→ Technisches Backlog:

- Owner ist das Team
- Einträge werden im Sprint Planning mitbearbeitet

## A-TDD:

- Acceptance Test Driven Development
- Tests als Requirements, Requirements als Test
- Workshop zum klären der Requirements

## Konfigurationsmanagement:

- Projektresultate sicher verwahren
- Teammitgliedern kontrolliert Zugriff geben
- ⇒ Ziele:
  - Änderungen protokollieren
  - Kommunikation vereinfachen (Transparenz)
  - Qualität sicherstellen
  - Dokumentation ...

## Konfigurationselement:

- Typ einer Gruppe von Artefakten, die dem zu entwickelnden Produkt zugeordnet sind
- Beispiele:

- Quelltext
- Anforderungsdokumente (z.B. use cases)
- Architektur- und Designdokumente
- Schnittstellenverträge
- Dokumentationen
- Build-Skripte
- ⋮

⇒ Keine Konfig-Elemente:

- Meetingprotokolle
- Binäre Auslieferungsdateien
- Projektpläne
- ⋮

⇒ Werden im KM (Konfigurationsmanagement)-Handbuch beschrieben:

- Kurze Beschreibung } mindestens
- Namenstemplate } ↓
  - eindeutige Identifikation einer Instanz des Objekts am Namen
  - Name der Datei soll auf übergeordnetes Element verweisen
  - Beziehungen zwischen Konfigelementen sollen aus dem Dateinamen ersichtlich sein

## Projektstruktur festlegen:

- Nach Konigelementen
- Nach Projektstruktur
- Nach Softwarearchitektur

## Repositories:

- Verwaltung der Konigelemente
- Alte Stände wiederherstellen
- Parallelle Änderungen: (2 Möglichkeiten):
  - SVN → lock - Modify - unlock (reserved checkout)
  - Git → Copy - Modify - merge (unreserved " ")
- Tags statt Versionsnummern ⇒ leichter zu merken
- Branches für Features erstellen
- Releases: Was an den Kunden gegeben werden könnte
- Buildprozess:

Art, nutzen, ...

→ Entwicklerbuild: local  
 → Integrationsbuild: regelmäßig ⇒ Qualitäts sicherung  
 → Releasebuild: Analog ⇒ Setzen eines Release Tags

## Änderungs- und Fehlermanagement:

⇒ Daten eines Change Request:

- Id
- Name des Autors
- Datum
- Status
- Beschreibung
- Bewertungen
- Priorisierung
- :

Verwaltung durch Change Control Board

Raumkonzept:

→ Dient zur Unterstützung des Workflows:

- Entwicklungsumgebung
- Testumgebung
- Integrationsumgebung
- Produktivumgebung
- :

Subversion:

- Dateivercionierung
- SVN Client + Server
- SVN vergibt für jede Änderung Versionsnummer (Nicht änderbar)
- SVN kennt Tags und Branches nur als Unterordner
- Arbeiten auf lokaler Kopie → Feststellen von Änderungen
- Aktualisierungen mit commit auf den Server schreiben
- SVN Update zum Updaten vom Server
- zentrales Versionswaltungssystem
- Gut geeignet für Continuous Integration

Revision  
↑

Git:

- Dezentral
- Hohe Performance
- Effizientes Arbeiten
- Offlinefähig
- Wartbarkeit
- Backups leicht möglich
- jede Version als Commit in Repository: Identifiziert durch Hash

git (Fortsetzung):

- Diff: Unterschied der Versionsstände: „Changeset“
- Historie ist lokal (Wird per Merge zentral gespeichert)
- Branches für verschiedene Entwicklungsstände
- Pro Repository genau 1 aktiver Branch
- Merge automatisch, außer es gibt Konflikte
- Rebase: "Historie glätten": Bei Commits im falschen Branch verwenden  $\Rightarrow$  ergibt neue Commits, die zurücksetzen
- Fetch / pull zum holen (zentrales)
- Push zum schicken in anderes Repository
- Remote tracking branch: holen den Stand mit dem Stand eines anderen Repositories vergleichen  
 $\Rightarrow$  Aktualisierung von Fetch / Pull
- Pull: Fetch + Merge
- Upstream Branch: Verknüpfung zwischen einem lokalen und einem Remote-Tracking-Branch
- zentrales Repo: master + Develop-Branch
- lokal: Feature-Branches

Maven:

- Buildsystem
  - Dependency Management
  - Wiederverwendung der Buildlogik
  - Vereinheitlichung
  - einfache Suche nach Artefakten
  - erstellen der Dokumentation
  - Plugins: Erweiterungen, die Funktionen ausführen
- z.B. jar  
komplizieren,  
↑  
..

## Maven (Fortsetzung):

(43)

- Goals: Tasks, die Aufgaben durchführen  
⇒ Werden von Plugins ausgeführt
- POM: Project object model: Deklaration des Projekts
- Eindeutige Definition: groupId, artifactId, version
- Packaging: lifecycle mapping noo package type
- Maven - Repository: Sammlung von Artefakten
- Konzepte:
  - Plugins und Goals
  - lifecycle
  - Koordinaten
  - repositories
  - Dependency Management
  - Site generation and reporting
- POM - Vererbung ist möglich

## Nexus:

- Repository - Manager als Proxy zu den Remote Repositories
- Hostet firmeninterne Artefakte
- Hauptmerkmale:
  - Management von Artefakten + Metadaten
  - „Proxying“
  - Deployment auf gehostete Repositories
  - Suche / Indizierung
- Private Repositories möglich
  - Proxy "": Apache / Codehaus Snapshots, Zentral
  - Hosted "": 3rd Party
  - Virtual Repositories

## Nexus (Fortssetzung):

- Repositories können zu Gruppen zusammengefasst werden

## Continuous Integration:

- Merge - Konflikte }
- Compile - "
- Test - "

} Integrationsprobleme

- Integration ist aufwendig:

→ Hohe Zahl an Metalebenen

→ " " an Fehlern? (eventuell)

⇒ Häufige, kurze Integrationen

- Entwicklung soll von den Integrationen nicht gestört werden

- Vorteile:

→ Bugs werden schneller gefunden

→ Risikominimierung ("Technische Schulden")

→ Häufiges Deployment ⇒ Schnelles Feedback

## Hudson:

- Buildautomatisierung
- Automatisiertes Deployment
- Testautomatisierung
- Wann CI?:

→ Entwicklebuild:

> local

> modultests

> Auf Basis des Maven POM

Hudson (Fortsetzung):

→ Integrationsbuild:

- > Zentrale Rolle im Projekt
- > Deployment
- > Nicht mit Maven alleine möglich
- > Buildprofil im Maven POM anlegen
  - ⇒ Anpassen möglich (z. B. Versionsnummern)
- Weitere Punkte beim Integrieren:
  - > Checkstyle: Codestyle überprüfen
  - > Code Coverage Tools: zur Testabdeckung