**Zusammenfassung MST**

**Kapitel 2 - Kommunikation**

**Verschiedene Arten der Kommunikation:**

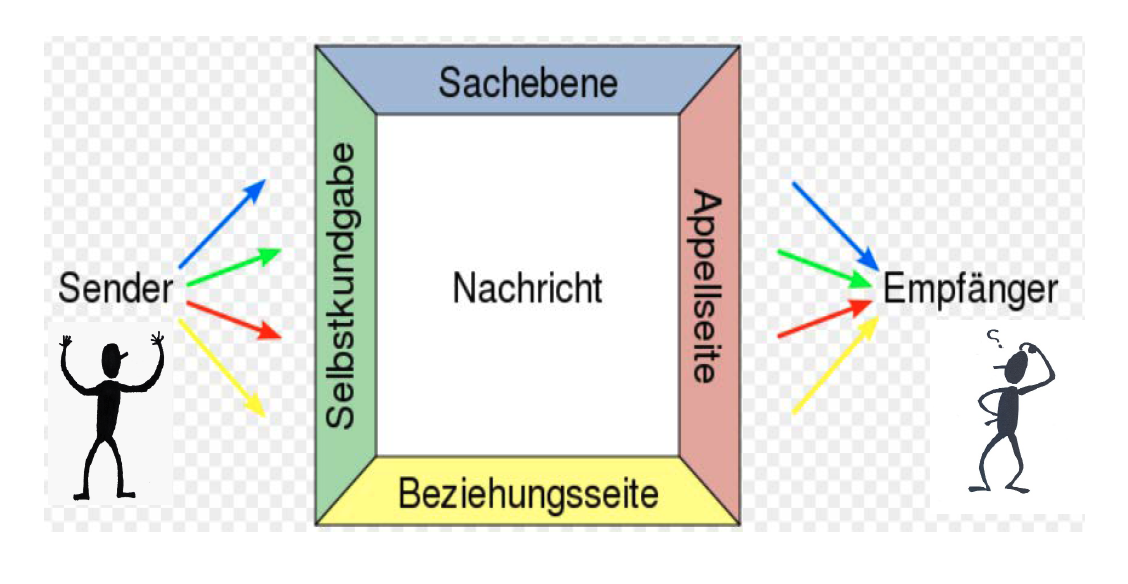
* Formal – infomell
* Mündlich – schriftlich
* Ein Senser- ein oder mehrere Empfänger

**Hauptgründe für Projektabbruch:**

1. Änderungen der Anforderungen und des Umfangs
2. Mangelnde Einbindung des höheren Managements
3. Engpass im Budget
4. Fehlende Projektmanagement Fähigkeiten

**--> Kommunikationprobleme**

**Kommunikationsmodell von Friedemann Schulz von Thun (Vier-Seiten-Modell)**



**Sachebene:** „Inhalt“ der Nachricht

**Sebstkundgabe:** Was der Sender von sich kundgibt

**Beziehungsseite:** Wie Sender / Empfänger zueinander stehen

**Appellseite:** Was der Empfänger tun soll

**Weitere Aspekte der Kommunikation**

* Explizite Botschaften (Direkt ausgedrückt, offen, wie man es meint)
* Implizite Botschaften (Zwischen den Zeilen, durch die Blume, diplomatisch)
* Nonverbale Kommunikation ( Gestik, Mimik)
* Kongruente Nachrichten (Gesagtes mit Körpersprache eindeutig)
* Inkongruente Nachrichten (Person sagt was, passt aber net zur Körpersprache)

**Kommunikationsprobleme**

* Durch stark ausgeprägtes Hören nur eines Aspekts der Nachricht (Hörer)
* Einseitige Sendegewohnheiten (Sender)
* Zu viele Fachbegriffe (Bullshitbingo)

**Killerphrasen**

Killerphrasen sind Sätze der Abwehr, der Ablehung oder der Herabsetzung, die keinen konstruktiven Beitrag leisten. Killerphrasen zielen oft auf die Person und nicht auf die Sache. Killerphrasen killen neue Ideen. Bsp: „Das geht sowieso nicht“, „Dazu fehlt uns die Zeit“ etc.

**Kennzeichen einer gelungenen Kommunikation**

* Verständlichkeit
* Ziel und Lösungsorientierung
* Persönliche Transparenz
* Wertschätzende Beziehungsgestaltung

**Verschiedene Handlungstypologien als Konfliktursache**

* Verstandesorientierung
* Risikoorientierung
* Ordnungsorientierung
* Beziehungsorientierung

**Schichtenmodell zur Bearbeitung von Konflikten**

Konfliktthemen können folgenden Schichten entstammen:

* Arbeitsorganisation (Bedingungen, Infrastruktur) **(Grund)**
* Rollen (Fähigkeiten, Kompetenzverteilung)
* Verhalten (Verhaltensmuster etc.)
* Werte und Normen (Einstellungen, Leitsätze, Überzeugung)
* Persönlichkeit **(Spitze)**

**Kapitel 3 – Software Qualität**

**Qualitätsmerkmale bei Software**

|  |  |
| --- | --- |
| **Kundenorientiert** | **Herstellerorientiert** |
| * Funktionalität * Zuverlässigkeit * Effizienz * Benutzbarkeit | * Portabilität * Wartbarkeit * Transparenz * Testbarkeit |

**Korrelation von Qualitätskriterien**

**Effizienz (Laufzeit):**

Negative Korrelation mit fast allen anderen Qualitätsmerkmalen 🡪 mit Bedacht optimieren

**Benutzbarkeit:**

Keine Korrelation mit anderen Merkmalen 🡪 Benutzerfreundliche Programme sind möglich, ohne die anderen Merkmale zu beeinträchtigen

**Gründe für schlechte Softwarequalität**

* Wachsende Komplexität
* Neue Anwendungsgebiete
* Vollständige Tests praktisch unmöglich
* Produktlebensdauer >> Projektdauer
* Erwartungshaltung / Fehlerakzeptanz des Kunden

**Welche Möglichkeiten haben Sie, um SW Qualität zu sichern?**

**Produktqualität sichern durch:**

Konstruktive und analytische Qualitätssicherung

**Prozessqualität sichern durch:**

Software Infrastruktur und Management Prozesse

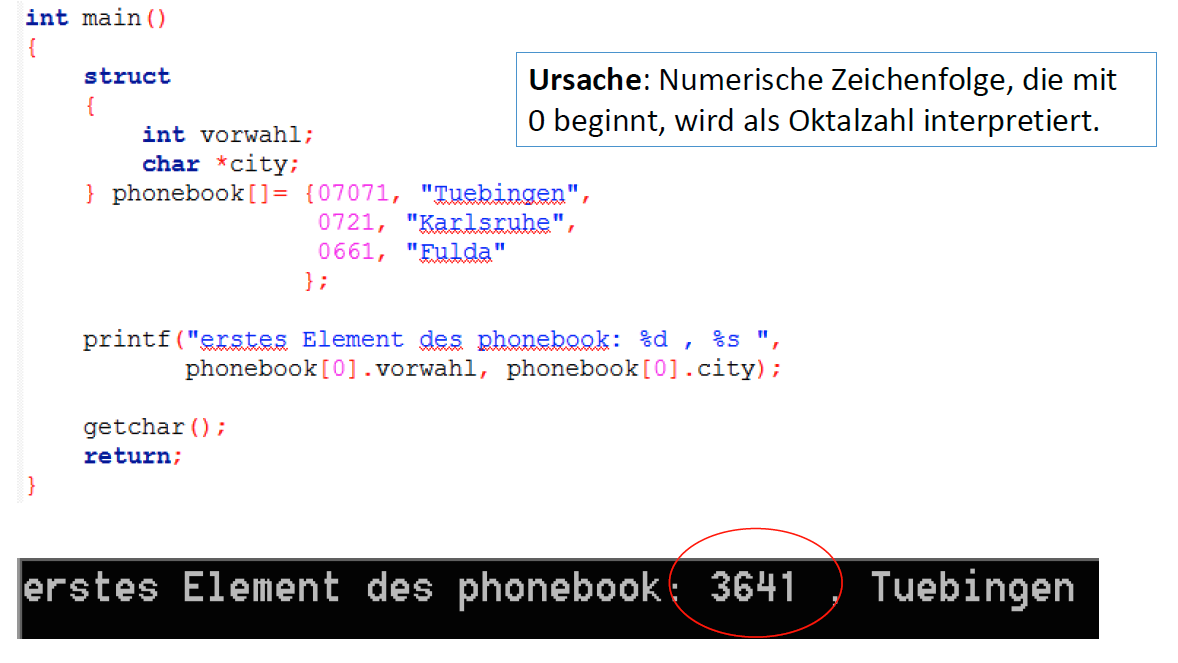
**Kapitel 3.1 – Software Fehler**

**Software Fehlerquellen**

**Lexikalische und syntaktische Fehlerquellen**

Fehler, die der Compiler nicht erkennt und die auf mangelndes Sprachverständnis zurückzuführen sind.

**Beispiel: Numerische Zeichenfolge als Oktalzahl interpretiert**



**Semantische Fehlerquellen**

Wenn Bedeutung von einzelnen Konstrukten fehlinterpretiert wird 🡪 Falsche Verwendung der Konstrukte 🡪 Fehlerverhalten

**Beispiel: AT&T Fehler in Fehlerbehandlung 🡪 break hat gesamten Switch-Konstrukt beendet 🡪 Systeme abgestürzt**

**Parallelität als Fehlerquelle**

Zugriff auf gleiche Ressource zur gleichen Zeit 🡪 Falsche Werte gelesen / geschrieben

**Numerische Fehlerquellen**

Rechenungenauigkeiten, Abweichungen bei Berechnungen 🡪 Fehlverhalten (z.B. Raketenabwehrsystem)

**Portabilitätsfehler**

Wenn Software von einer auf die andere Plattform übernommen wird. Beispielsweise Typkonvertierung beim ersten System noch okay, beim zweiten jedoch Fehlerhaft

**Optimierungsfehler**

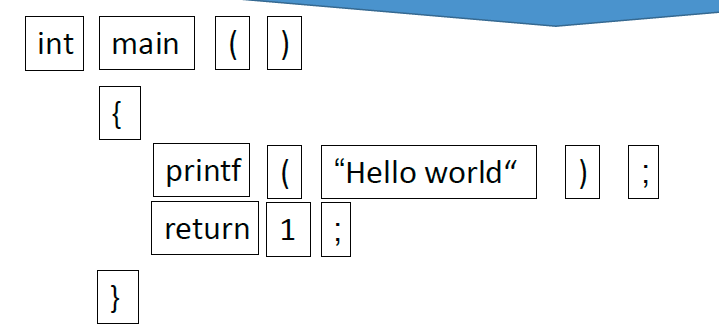
Bei Optimierungsversuchen (Laufzeitverhalten, Code optimieren) können Fehler in den Code eingebaut werden. (Beispielsweise bei einem alten mail Befehl in UNIX mal gewesen)

**Spezifikationsfehler**

Falsche Angaben von Spezifikation an die Software (z.B. Flugzeugsoftware mit falsch erkannter Formel 🡪 Zu späte Aktivierung der Schubumkehr)

**Die Greedy Srategy**

Zeichenstrom wird sequentiell in einen Symbolstrom umgewandelt. Ein neues Symbol wird erst begonnen, wenn das alte nicht mehr vergrößert werden kann. In diesem Falle auch der fehler mit (a--)-1. Hier ein weiteres Beispiel:



**Prioritätsinversion**

Wenn mehrere Prozesse mit unterschiedlicher Priorität und eine Ressource beteiligt sind. Ein Prozess mit hoher Priorität will auf eine Ressource zugreifen, kann dies aber nicht, da die Ressource bereits von einem niedriger priorisierten Prozess belegt ist. Der hoch priorisierte Prozess muss warten, bis der andere Prozess die Ressource wieder freigibt. Dadurch sind die Prozess-Prioritäten in einer ersten Form der Prioritätsinversion vertauscht.

**Fehlerbewertung:**

* **Aus Sicht des Anwenders:** Interesse an Abwesenheit von Defekten
* **Aus Sicht des Entwicklers:** Interesse am Aufdecken von Defekten

**Kapitel 3.2 – Konstruktive Qualitätssicherung**

**Software Richtlinien**

Richtlinien regeln den Gebrauch einer Programmiersprache über die syntaktischen und semantischen Regeln hinaus. Motivation ist die **Fehlerreduktion** und **Vereinheitlichung**

**Notationskonventionen**

Definition auf verschiedenen Ebenen (Projekt, Sprache, Betriebssystem).

**Folgende Dinge sind betroffen:**

* Auswahl und Schreibweise von Bezeichnern
* Einrückungen, Verwendung von Leerzeichen
* Dokumentation

**Notationsstile:**

* **Pascal Case** (Bezeichner bzw. jedes enthaltene Wort startet mit Großbuchstaben z.B. SomeMethod)
* **Camel Case** (Bezeichner startet mit klein, jedes weitere Wort groß z.B. someMethod)
* **uppercase** (Alles groß z.B. SOMEMETHOD)
* **lowercase** (alles klein z.B. somemethod)´

Des Weiteren existieren Coding Styles, in denen die Notationskonventionen und weitere Standards definiert werden (Java Coding Style, C# etc.)

**Sprachkonventionen**

Nun semantische Besonderheiten einer Sprache (z.B. **MISRA-C: Prorammierstandard)**

**MISRA-C: Prorammierstandard**

C wird in sicherheitskritischen Bereichen verwendet, aber:

* Verhalten teils undefiniert
* C macht es leicht, die Sprache falsch zu gebrauchen
* C erlaubt schwer verständliche Konstrukte etc.

🡪 MISRA unterstützt die Entwickler

**Umfang von MISRA**

* Empfehlungen zu Tool Selection, Projekt Aktivitäten, Implementierung der MISRA Cpmliance
* Guidelines (z.B. kein goto verwenden, keine Rekursion, magic numbers vermeiden)

**Gefahren bei der Durchsetzung von Konventionen durch Codereviews**

* Diejenigen, die die Konventionen am besten kennen sind die Buhmänner
* Gereviewte fühlt sich persönlich angegriffen
* 🡪 Reviewer will deshalb nichts mehr bemängeln

**Durchsetzung der Konventionen (Abhilfe für Gefahren)**

* Alle Beteiligten mit an Konvention arbeiten lassen
* Konvention gemeinsam weiterentwickeln
* Toolgestützt prüfen
* Regeln zu Beginn vereinbaren

**Typisierung**

Durch die Typisierung der Programmiersprachen werden Daten kategorisiert und gleichartige Objekte zu einem Datentyp zusammengefasst.

Der Compiler kann die korrekte Verwendung der Datentypen prüfen (**Typprüfung**), was zu folgenden Vorteilen führt:

* Frühzeitige Erkennung von Fehlern / Inkonsistenzen
* Verständlichkeit des Codes wird verbessert

**Bestandteile des Typsystems**

* Typen
* Möglichkeit, Variablen, Funktionsparameter etc. mit einem bestimmten Typ zu deklarieren
* Regeln für eine bestimmte Typzuweisung eines Werts
* Regeln zur Prüfung der Zuweisungskompatibilität von Typen

**Aufgaben eines Typsystems**

* Erkennen von Typverletzung
* Typumwandlungen

**Arten von Typprüfungen**

* **Statische Typprüfung** 🡪 Prüfung zur Compilezeit (Hauptsächlich bei hardwarenaher Programmierung)
* **Dynamische Typprüfung** 🡪 Prüfung während der Programmausführung 🡪 Erhöht Sicherheit des Programms und kostet Performance (bei Programmsprachen der höheren Abstraktionsebene
* **Kombinierte Typprüfung** 🡪 Statisch + Dynamisch

**Einteilung der Typisierung nach Prinzipien**

* **Statisch:** Deklaration des Typs im Code bei der Variable
* **Dynamisch:** Variablen sind an bestimmten Datentyp gebunden
* **Schwach:** Datentypen eines Objektes dürfen beliebig uminterpretiert werden
* **Stark:** Stets typkonforme Zugriffe

**Generische Schnittstellen**

Manchmal ist eine strenge Typsicherheit nicht gewünscht (z.B. Kommunikation mit Gerätetreibern über eine einheitliche Schnittstelle)

Für diesen Anwendungsfall werden dynamische Typen verwendet. Erhöhung der Typsicherheit erfolgt durch die Verwendung von Container Typen (z.B. VARIANT)

**Vertragsbasierte Programmierung (Design by Contract)**

**Idee von Design by Contract**

* Vor der Implementierungen werden Vereinbarungen für Schnittstellen zwischen Modulen festgeschrieben
* Es wird ein Vertrag zwischen Aufrufer und Aufgerufenem vereinbert, der Vor- und Nachbedingungen sowie Invarianten festlegt

**Grundprinzipien**

* **Vorbedingungen (Preconditions):** Zusicherungen, die der Aufrufer zu beachten hat
* **Nachbedingungen (Postcond.):** Zusicherungen, die der Aufgerufene zu beachten hat
* **Invarianten:** Gesundheitszustand der Klasse (logische Aussagen, die für alle Instanzen einer Klasse über den gesamten Objektlebenszyklus hinweg gelten)

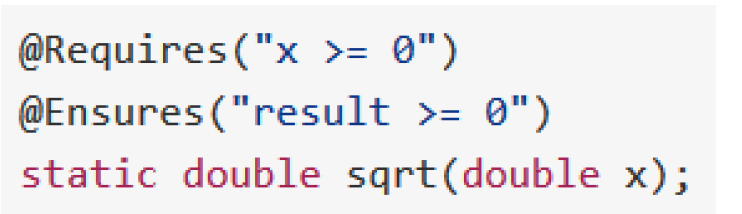
🡪 Verletzung der Bedingung führt zu Abbruch des Programms

**Nutzen**

* Steigerung der Qualität durch genaue Spezifikation der Schnittstellen
* Contracts als Dokumentation
* Unterstützung beim Testen/Fehlerfinden

In manchen Sprachen ist Design by Contract nativ verankert, aber auch oft per Erweiterung.

**Beispiel von DbC in Java**



**DbC und Vererbung**

* Vorbedingungen können in den vererbten Methoden gelockert werden
* Nachbedingungen können strenger gemacht werden
* Invariants können strenger gemacht werden
* Zusätzliche Einschränkungen sind bei den abgeleiteten Methoden möglich

**Fehlertolerante Programmierung**

Verbesserung des Reaktionsverhaltens eines Programms im Fall eines Fehlers 🡪 Möglichst kein kompletter Softwareabsturz

**Software Redundanz**

* **Funktionale Redundanz:** zusätzliche Funktionen zur Erhöhung der Fehlerzoleranz
* **Informelle Redundanz**: Nutzdaten um zusätzliche Informationen angereichert
* **Temporale Redundanz**: Zeitanforderungen werden übererfüllt, Wiederholung wäre im Notfall möglich
* **Strukturelle Redundanz**: Mehrfachauslegung von Komponenten
  + **Homogen**: n Komponenten gleicher Bauart 🡪 Im Hardware-Bereich
  + **Heterogen**: n Komponenten unterschiedlicher Bauart
  + **Statisch**: Alle Komponenten aktiv, Ergebnisse werden durch Voter verglichen
  + **Dynamisch**:Ersatzkomponenten, die bei Bedarf aktiviert werden

**Selbstüberwachende Systeme**

**Reaktionsszenarien**

* **Fail-Safe Reaktion:**  Wechsel in einen sicheren Zustand (z.B. Selbstaktivierende Notbremse eines Aufzugs
* **Selbstreperatur**: Selbstdiagnose und Reperatur (z.B. automatische Dateiwiederherstellung)
* **Reaktivierung:** Watchdog Logik z.B. Explorer Prozess Windows XP)

**Ausnahmebehandlung**

In C erreicht man Ausnahmebehandlungen durch if, else Konstrukte. In Java hat man das Exception Handling.

**Mögliche Fehler die auftreten können**

* Benutzerfehler (Client Errors)
* Programmierfehler (Internal Errors)
* Fehler beim Zugriff auf Ressourcen (Service Errors)

**Philosophien der Ausnahmebehandlung**

* Delegation der Ausnahmebehandlung ans Betriebssystem
* Behandlung der Ausnahmen in dem Konzept der Programmiersprache verankert

**Verschiedene Ausnahmen**

* **Geplante Ausnahme:** Datei nicht gefunden, Ungültige Eingabe etc.
* **Ungeplante Ausnahme:** Speichermangel, Division durch Null etc.

**Auswirkungen von fehlerhaften Exceptionhandlings**

* Dateien werden nicht geschlossen
* Datenbankverbindungen bleiben offen
* Gui: Sanduhr verschwindet nicht

**Strukturierte Ausnahmebehandlung**

**Vorgehen bei der strukturierten Ausnahmebehandlung**

1. Loggen der Ausnahme nahe der Stelle, an der sie auftritt
2. And den Aufrufer weiterreichen
3. Behandlung der Ausnahme an der Stelle, an der man die Folgen erkennen kann und eine angemessene Reaktion erfolgen kann
4. Angemessene Reaktion
   1. Erneute Eingabe durch Benutzer
   2. Datei schließen
   3. Programm beenden etc.

**Vorteile der strukturierten Ausnahmebehandlung**

* Trennung Anwendungscode von Ausnahmebehandlung
* Exceptions können nicht ignoriert werden
* Weitergabe an Aufrufer einfach möglich
* Saubere Gruppierung von Fehlern möglich

**Java Exceptions**

* **Checked Exceptions:** Compiler prüft, ob Code zum Abfangen der Exception vorhanden ist 🡪 Entwickler wird gezwungen, alle Ausnahmen wirklich zu behandeln
* **Unchecked Exceptions:**  Können behandelt werden, müssen aber nicht 🡪 Können schwere Fehler verursachen
* **Errors:** Unchecked Exceptions, die auf ein wirklich ernsthaftes Problem hinweisen und nicht gefangen werden sollten

**Gutes / Schlechtes Exception Handling**

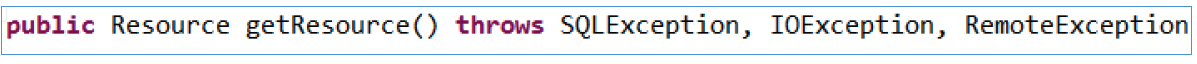
|  |  |
| --- | --- |
| **Gutes Exception Handling** | **Schlechtes Exception Handling** |
| * Einfachere Entwicklung * Einfachere Wartung * Weniger Bugs * Einfachere Anwendung | * Verwirrung der Benutzer * Schwere Wartung |

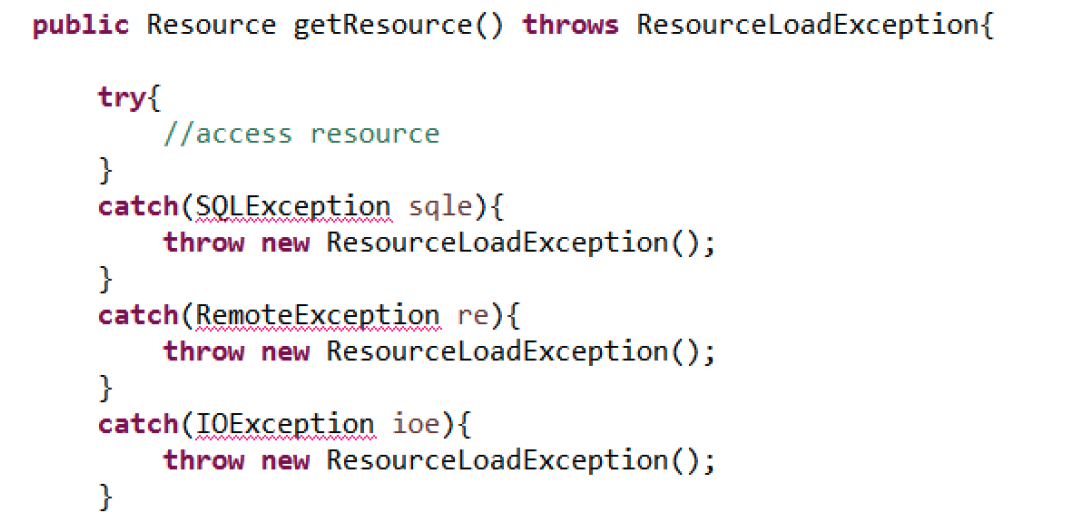
**Perspektiven des Exceptions Handlings**

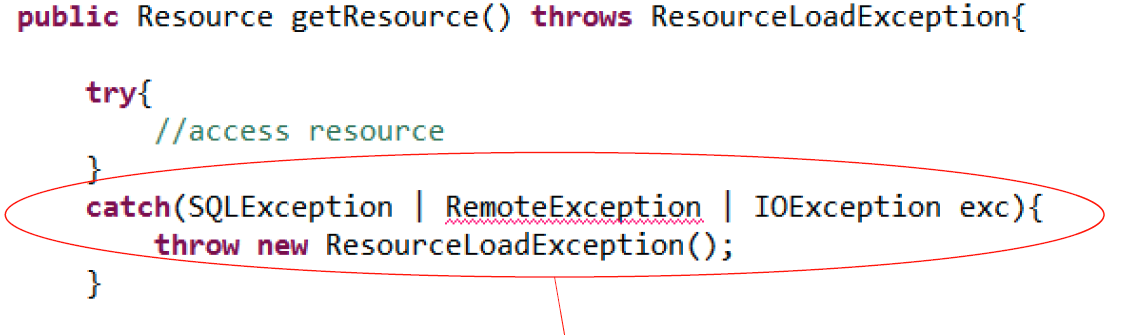
* Klasse, die die Exception wirft
* Klasse, die die Exception fängt
* Der Benutzer, der mit dem Resultat umgehen muss
* Entwickler bzw. Support

**Möglichkeiten des Designs von Exceptionshandlings**

* **throws:**  Die geworfenen Exception Types einer Klasse
* **NICHT machen:**  Deklaration aller möglichen Exceptions im throws



* **Hier wird die Information der Implementierung offenbart bzw. Info nicht für Aufrufer relevant**
* **Methodensignatur:** zusammenfassen der einzelnen Exceptions zu einer übergeordneten (jedoch gehen hier noch Infos verloren)
* 
* **Java Multicatch:**



* **Exception Chaining:** Eigene Exception, die die anderen Exceptions als „Inner Exception“ kapselt.

**Unterteilung von Exception**

* **ApplicationException:**  Fehler auf Geschäftslogikebene
* **SystemException:** Fehler auf Systemebene

**Exceptions für den Benutzer**

* Keine technischen Details
* Sprechende Meldung, mit Anweisungen, wie er weiter vorgehen soll
* Lokalisiert

**Best Practices mit Exceptions**

* Verwenden Sie spezifische Exceptions
* Exceptions erst dann fangen, wenn sie auch ordentlich behandelt werden können
* Loggen Sie alle Exceptions (aber nur einmal!)

**Anforderungen ans Exception Handling**

* **Primär:**
  + Kein Absturz der Applikation (Fehler fangen und sauber schließen)
  + Information der relevanten Stellen an Entwickler, Admin etc.
  + Fehlerdiagnose und Reproduktion (Wo ist der Fehler aufgetreten + Welcher Kontext?, Welcher Weg hat dorthin geführt?)
* **Sekundär:**
  + Abstraktion (Verbergen z.B. Custom Exceptions)
  + Les- und Wartbarer Code

**Error Location und Error Context**

* **Error Location:** Wo genau ist der Fehler aufgetreten?
* **Error Context:** Ausführungspfad zu Errorlocation 🡪 Auswirkung auf Behebung

**Fehlerarten und Reaktionen**

* Abbruch der aufgerufenen Aktion **(Client, Service, Internal)**
* Schließen der geöffneten Ressourcen **(Client, Service, Internal)**
* Freigeben der allokierten Ressourcen **(Client, Service, Internal)**
* Nutzer benachrichtigen **(Client, Service, Internal)**
* Loggen des Fehlers **(Client, Service, Internal)**
* Benachrichtigung des Betriebs 🡪 Maßnahmen können ergriffen werden **(Service, Internal)**
* Benachrichtigung der Entwickler zur Behebung **(nur Internal)**

**Strategielemente des Exception Handlings**

* Fehler entdecken
* Informationen über Fehler sammeln
* Exception werfen
* Weitergeben an Aufrufer + Zusatzinfos hinzufügen
* Fangen + Reaktion:
  + Wenn möglich neuer Versuch
  + Relevante Parteien informieren
* **Die Benutzung von Templates wäre sinnvoll !**

**Beispiel für ein Exception Template – AppException**

* Verwendung von Severity und ErrorType
* Bei mehreren ErrorInfoObjekten 🡪 letzte verwenden
* Beschreibung für Benutzer:
  + Client error: Beschreibung + Anleitung zur Korrektur
  + Internal error: Keine Beschreibung, Standardfehlermeldung
* Beschreibung für Entwickler: z.B. XML

**Portabilität (Protabilität entspricht Plattformunabhängigkeit)**

**Portierungsszenarien in der Praxis**

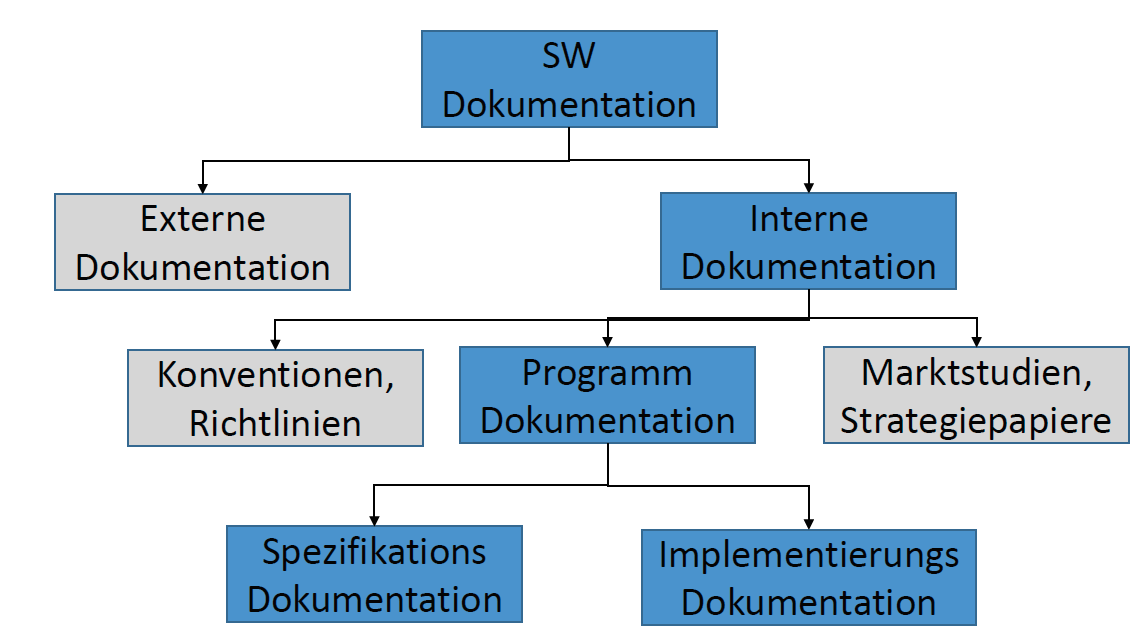
* Architekturportierung (auf andere Hardware)
* Betriebssystemportierung
* Systemportierung (Portierung auf andere Geräteklasse)
* Sprachportierung

**Verschiedene Ebenen um Portabilität von SW zu erhöhen:**

* Portabilität auf Implementierungsebene
* Auf Sprachebene
* Auf Systemebene (Anpassung der Umgebung an das Programm)

**Dokumentation**

**Arten von SW-Dokumentationen**

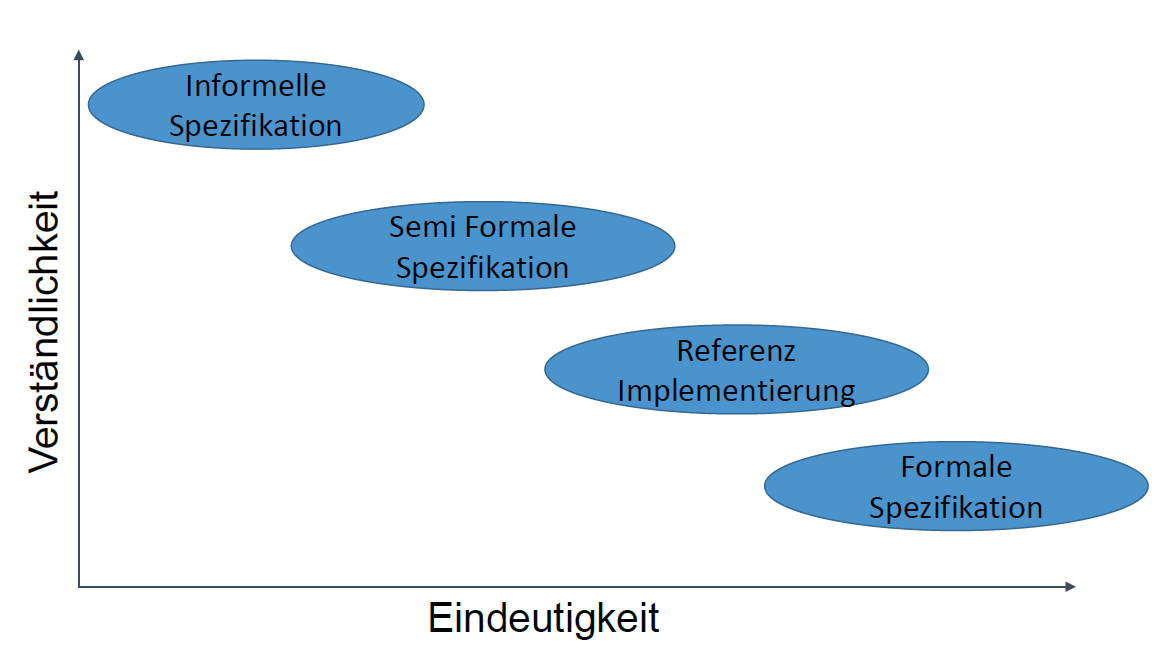


**Die Spezifikationsdokumentation**

Die Drei Varianten sind:

* **Informal:** Implizite Annahmen, ignorieren von Sonderfällen, sprachliche Ungenauigkeiten
* **Semiformal:**  „mathematische Definition, Verwendung definierter Formalismen, gut lesbar, wenig (bis keine) Zweideutigkeiten (Beispiel UML)
* **Formal:** Lässt keinen Interpretationsspielraum, schnell extrem komplex, spielt in der Praxis kaum eine Rolle (z.B. Spezifikationssprache 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 – Eine semiformale Spezifikationsdokumentation**

**UML-Diagrammtypen**

|  |  |
| --- | --- |
| **Strukturdiagramm:**   * Klassendiagramm * Objektdiagramm * Paketdiagramm | **Architekturdiagramm**   * Kompositionsdiagramm * Komponentendiagramm * Einsatz- und Verteilungsdiagramm |
| **Verhaltensdiagramm:**   * Usecase Diagramm * Aktivitätsdiagramm * Zustandsdiagramm | **Interaktionsdiagramm:**   * Sequenzdiagramm * Kommunikationsdiagramm * Zeitdiagramm |

**Implementierungsdokumentation**

* „Our Code is our documentation“ 🡪 Oft nicht ausreichend!!!
* Aus diesem Grund muss der Code integriert Dokumentiert werden!

**Regeln für Kommentieren der Software (Muss nicht auswendig gelernt werden)**

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 vollständige fotmulieren
4. Grammatik und Buchstabierung richtig anwenden 🡪 Sprache muss verständlich sein
5. Kommentare sollten vom Code visuell abgehoben werden
6. Strukturierte Kommentaren in den Headern, sodass diese automatisch über ein Tool extrahiert werden können

**Weitere Aspekte der Dokumentation**

* **Wer benötigt Code Dokumentation?:** Die Dokumentation ist relevant für den Entwickler, Benutzer und Weiterentwickler
* Dokumentation soll für alle lesbar sein (auch, wer den Code nicht hat) 🡪 Ohne Code lesbar!
* Dokumentation ist in aufbereiteter Form lesbar (HTML oder PDF)

**Javadoc**

* Ein Werkzeug, das eine standardisierte Dokumentation für Java unterstützt
* Integriert Dokumentation in Java Code
* Automtatische Generierung der Dokumentation möglich durch das lesen der Doc-Comments aus den Quellcodedateien
* Daraus wird eine HTML basierte Dokumentation erzeugt
* Doc Comments können vor Klassen, Konstruktoren, Datentypen und Methdoen stehen **!!Müssen aber genau vor einer Deklaration stehen!!“**

**Doxygen**

* Output in TEX-Format
* Klassendiagramme
* Source-Code-Browsing

**Kapitel 3.3 – Software Test**

**Testklassifikation**

|  |  |
| --- | --- |
| **Prüfebene**   * Abnahmeebene * Systemebene * Integrationsebene * Unit-Ebene | **Prüfkriterium**   * Funktional * Operational * Temporal |
| **Prüfmethodik**   * Black-box * White-box * Gray-box |  |

**Prüfebenen**

* **Unit Tests:** Tests von eigenständigen atomaren Einheiten
* **Integrationstests:** Zusammenspiel mehrerer Einheiten wird getestet
* **Systemtests:** Test des Systems als Ganzes
* **Abnahmetests:** Systemtest in der Umgebung des Auftraggebers 🡪 sind juristisch relevant

**Unit Tests**

Einheiten (Units) sind z.B. einzelne Methoden, Klassen mit mehreren Attributen und Methoden etc.

Es gibt zwei Arten von Unit Test Cases:

1. Normale Programmausführung 🡪 Komponente tut, was sie soll
2. Ungewöhnliche Eingaben, falsche Eingaben

**Automatisierte Unit Tests**

* Ziel: Möglichst viele Modultests automatisieren
* Durchführung: Verwendung eines Testautomatisierungs-Frameworks (z.B. JUnit)
* Bei jeder Änderung können alle Tests nochmal laufen gelassen werden

**Modultests**

Testfälle definieren (Eingaben, erwartete Ergebnisse) 🡪 Aufruf der zu testenden Objekte / Methoden 🡪 Auswertung der Ergebnisse

**Integrationstests**

**Big Bang Integration**

Entwicklung sämtlicher Module, anschließend Integration auf einen Schlag

|  |  |
| --- | --- |
| **Vorteile** | **Nachteile** |
| * Testtreiber und Mocks sind nicht nötig | * Beginn erst, wenn alle Module fertig sind * Schwierige Fehlersuche (da gleichzeitig integriert wird) |

**Strukturorientierte Integration**

Inkrementelle Integration der Module zum Gesamtsystem (Reihenfolge richtet sich nach Abhängigkeit der Module).

|  |  |
| --- | --- |
| **Bottom Up –** Ausgangspunkt: Basiskomponenten, Verwendung von Testtreibern. | **Outside-In** – Integration von beiden Seiten nach innen |
| **Top Down** – Ausgangspunkt: Module der höchsten Schicht, Vewendung von Stubs | **Inside Out –** Integration von innen nach außen |

**Funktionsorientierte Integration**

|  |  |
| --- | --- |
| **Termingetriebene Integration -**  entsprechend der Verfügbarkeit | **Testgetriebene Integration -**  Integration für bestimmte Testfälle |
| **Risikogetriebene Integration –** riskanteste als erstes | **Anwendungsgetriebene Integration -**  Integration für bestimmte Usecases |

**Systemtests**

**Erschwerende Faktoren**

* Unvorhergesehene Fehler - Unklare Anforderungen
* Testumgebung - Eingeschränkte Debug Möglichkeiten
* Eingeschränkte Handlungsunfähigkeit

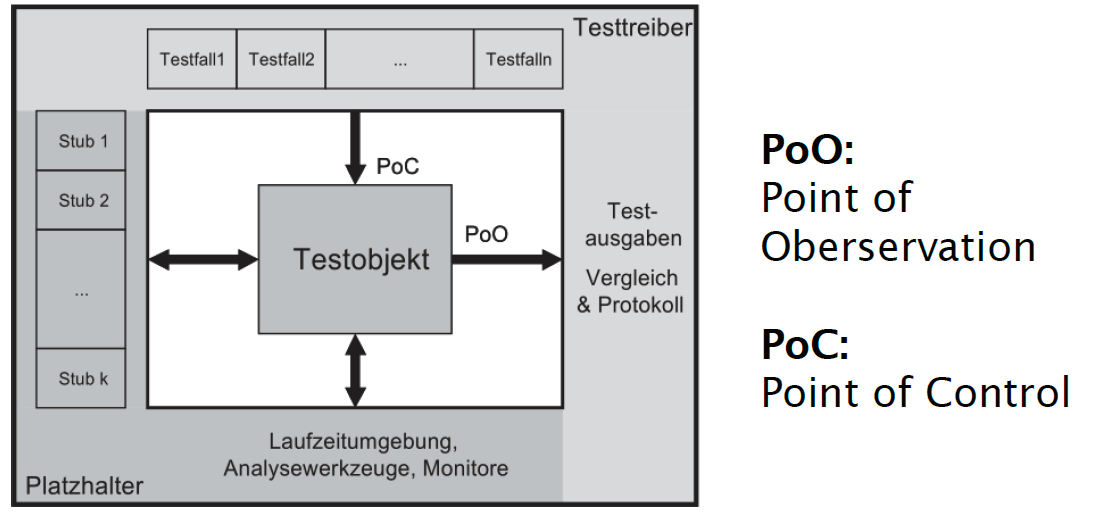
**Abnahmetests**

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

**Prüfkriterien**

|  |  |
| --- | --- |
| **Funktional:**   * Funktionstest * Trivialtest * Crashtest * Kompatibilitätstest * Zufallstest | **Temporal:**   * Komplexitätstest * Laufzeittest * Lasttest * Stresstest |
| **Operational:**   * Installationstest * Ergonomietest * Sicherheitstest |  |

**Testrahmen**



**Vorgehen beim Erstellen von Testfällen**

1. Die durch den Test verfolgten Ziele / Bedingungen /Voraussetzungen festlegen
2. Testfälle spezifizieren
3. Testausführung festlegen

**Prüftechniken im Vergleich**

* **Blackbox Test:** Tests aus Schnittstellenbeschreibungen hergeleitet
* **Whitebox Test:** Tests werden aus innerer Programmstruktur hergeleitet
* **Greybox Test:** Kombination Whitebox / Greybox

**Blackbox – Testverfahren**

**Äquivalenzklassentest**

**Äquivalenzklassen:** Teilmengen der möglichen Eingabewerte, die intern eine identische Verarbeitung erwarten lassen.

**Beispiele für ÄK:** [minLong, 0] und [1, maxLong] oder [minLong, -1] und [0, maxLong]

**Vorgehen:** 1. Aquivalenzklassenbildung für jeden Eingabeparameter suchen

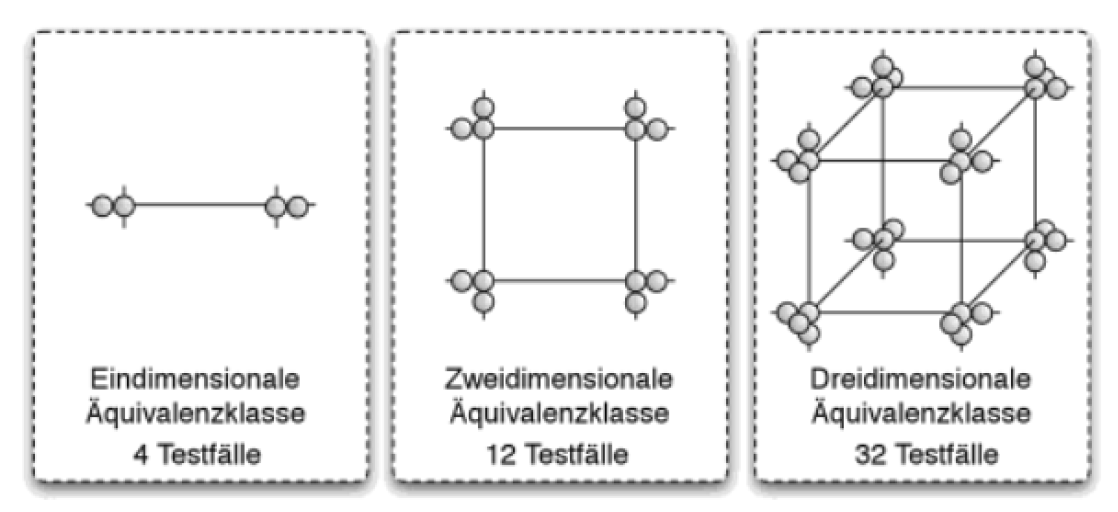
2. Zusammengehörige Äquivalenzklassen verschmelzen

**Vollständige Partitionierung**

Äquivalenzklassenbildung bezieht die ungültigen Werte mit ein

**Grenzwertbetrachtung**

* Partitionierung ähnlich zum Äquivalenzklassentest
* Unterschied: Auswahl der Randwerte eines Wertes innerhalb der Äquivalenzklasse und eines Wertes außerhalb der Äquivalenzklasse (Grenzwerte)



**Zustandsbasierter Software Test**

* Alle möglichen Übergänge testen
* Übergangsbaum (für Zustände zeichnen
  + Der Anfangszustand ist die Wurzel des Baums
  + Testfälle = Anzahl der finalen Blätter (die, die nicht mehr weitergehen)
* GUI-Masken können in Zustände eingeteilt werden
  + 🡪 Zustandsbasierte Tests eignen sich gut für GUI’s

**Use Case Test**

Der Use Case Test ist speziell für die Systemebene (Systemtest, Abnahmetest).

**Usecase basierter Black Box Test:** Alle möglichen Szenarien eines Usecases werden durch mindestens einen Testfall abgedeckt.

**Paarweises Testen**

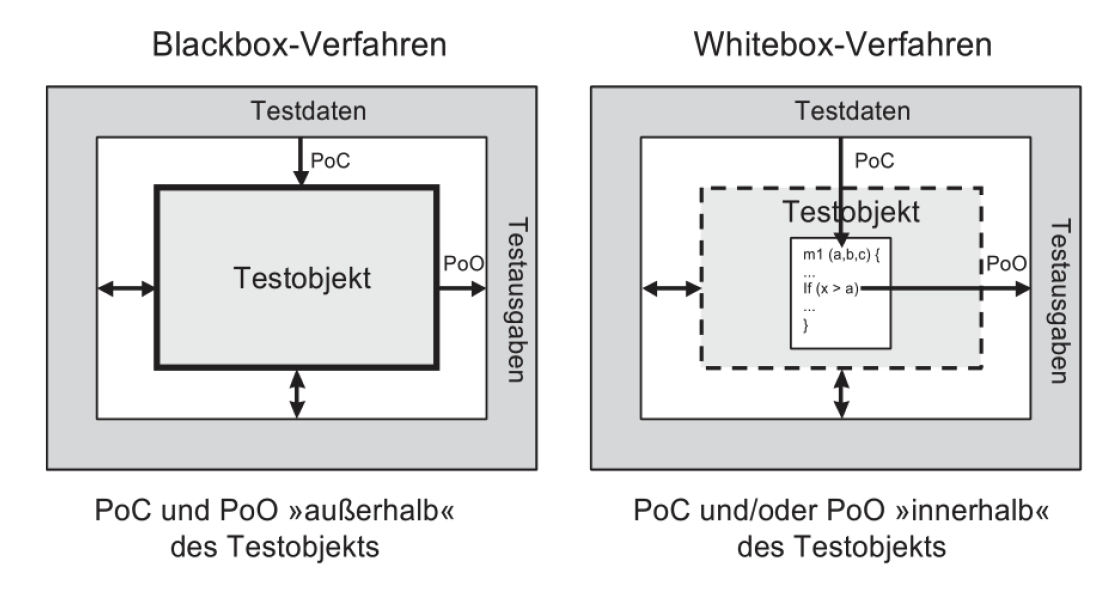
* Statt alle Möglichkeiten zu testen: Auswahl von bestimmten Kombinationen, die repräsentativ sind
* Wird jedoch sehr schnell sehr komplex !!

**Diversifizierende Verfahren**

* Vergleich verschiedener Programmversionen gegeneinander
* Eine zweite Implementierung übernimmt die Aufgabe der Spezifikation
* **Back to back Tests:** Vergleicht mehrere Varianten einer Komponente / Systems nach der Modifikation. **Ziel:** Abweichungen und deren Ursache finden
* **Regressionstests:** Erneuter Test eines Programmteils nach der Änderung. **Ziel:** Zeigen, dass keine neuen Fehler eingebaut wurden
* **Mutationstests**: Technik, um Testverfahren zu beurteilen

**Whitebox – Testverfahren (Strukturtest)**

**Blackbox vs Whitebox**



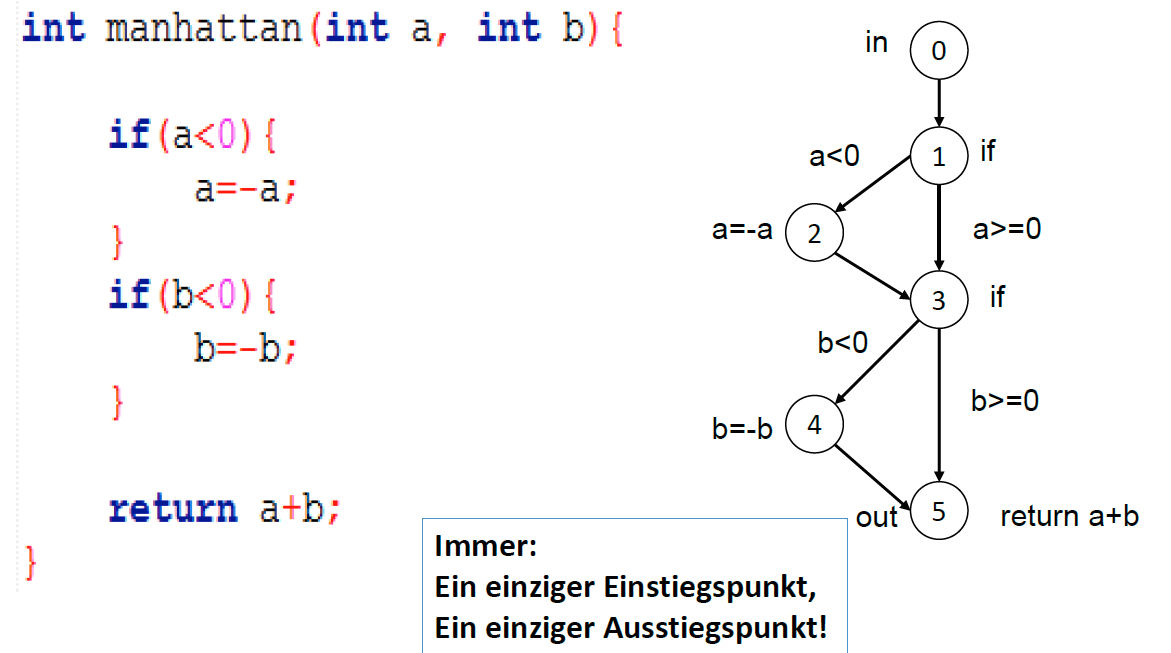
**Einteilung von Whitebox Tests**

* **Kontrollflussorientierte Tests**: Konstruktion der Testfälle aufgrund der internen Berchungspfade
* **Datenflussorientierte Tests:** Heranziehen zusätzlicher Kriterien

**Ablauf eines typischen Whitebox Tests**

1. **Strukturanalyse:** Aus Programmcode wird Kontrollflussgraph extrahiert
2. **Testkonstruktion:** Ableitung der Testfälle durch Überdeckungskriterium
3. **Testdurchführung**

**Beispiel für eine Kontrollflussmodelierung**



**Arten von Kontrollflussgraphen**

|  |  |
| --- | --- |
| **Kantenmarkierte Kontrollflussgraphen:** Anweisungen werden den Knoten, Verzweigungsbedingungen den Kanten zugeordnet. | **Knotenmarkierte Kontrollflussgraphen:** Verzweigungsbedingungen ebenfalls Knoten zugeordnet |
| **Expandierte Kontrollflussgraphen:** Jeder Befehl ein separater Knoten | **Teilkollabierte Kontrollflussgraphen:** Zwei oder mehrere sequenzielle Befehle in einem einzigen Knoten |
| **Kollabierte Kontrollflussgraphen:** Verzweigungsfreie Befehlsblöcke in einem einzigen Knoten |  |

**Überdeckungskriterien**

**Anweisungsüberdeckung (auch C0 Test genannt)**

* Jeder Knoten des Kontrollflussgraphen wird mindestens einmal durchlaufen
* Mit wenig Testfällen zu erfüllen
* Wichtige Fälle bleiben unberücksichtigt
* Schwer erreichbar, da aufwendig (Bei großen Projekten)
* 100% Abdeckung nötig, sonst nicht aussagekräftig
* Beispiele für Schwierigkeiten: Testen des Fehlers malloc

**Anweisungsüberdeckung (auch C1 Test genannt)**

* Jede Kante des Kontrollflussgraphen muss mindestens einmal durchlaufen werden
* Minimalkriterium

**Pfadüberdeckung**

* Für jeden Pfad vom Eingangsknoten zum Ausgangsknoten soll ein Testfall existieren
* Anzahl der Pfade explodiert schnell, vor allem wenn Schleifen vorhanden sind
* Pfadüberdeckung in der Praxis normalerweise nicht realisierbar

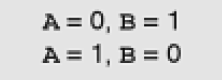
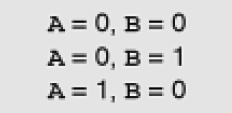
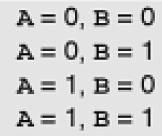
**Varianten, um die Anzahl der Testfälle zu reduzieren**

* **Boundary Interior Pfadüberdeckung:** Für jede Schleife existieren drei Gruppen von Testfällen:
  + **Äußere Pfade:** Schleifen werden nicht betreten
  + **Grenzpfade:** Genau eine Iteration
  + **Innere Pfade:** Mindestens eine weitere Iteration
* **Strukturierte Pfadüberdeckung:** Verallgemeinerung des Boundary-Interior Tests. Es werden alle möglichen Ausführungspfade bis zur k-ten Schleifeniterat. durchlaufen. 🡪 Modifizierung: k Iterationen nur für Schleifen, die keine inneren Schleifen mehr enthalten.

**Bedingungsüberdeckung**

Zusätzlich zum Kontrollflussgraphen: Einbeziehung der logischen Struktur der if-Bedingunge.

Folgende Varianten existieren:

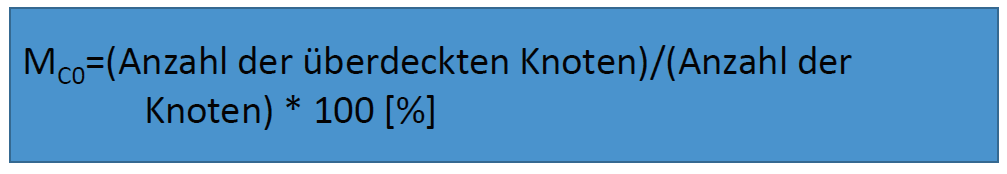
* **Einfache Bedingungsüberdeckung:** „Alle atomaren Prädikate müssen mindestens einmal beide Warheitswerte annehmen“
* **Minimale Mehrfachbedingungsüberdeckung:**  „Alle atomaren und zusammengesetzten Prädikate müssen mindestens einmal beide Wahrheitswerte annehmen“
* **Mehrfachbedingungsüberdeckung:** „Alle Wahrheitskombinationen müssen getestet werden“

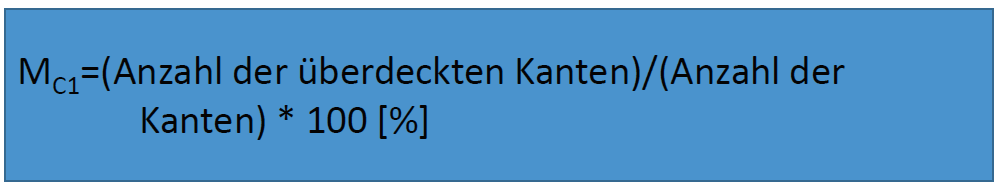
**Testmetriken**

Testmetriken quantifizieren folgende Fragen bzgl. des Testverfahrens / Testfälle:

* Wird ein Modul ausreichend getestet?
* Wie viele unentdeckte Fehler enthält das Programm?
* Wie leistungsfähig ist ein gegebenes Testverfahren?

Folgende Metriken sind im industriellen Bereich relevant:

* **Anweisungsüberdeckung**
* **Zweigüberdeckung**



* Verwendung bei Modulabnahmen und Vergleich von Modulen

**GCOV – Toolunterstützung bei Überdeckungsmetriken**

* Der Profiler gcov ist ein Teil des gcc
* Hiermit lässt sich der Sourcecode analysieren 🡪 Detaillierte Überdeckungsinformation der Testläufe
* Gcov ermittelt die Zeilenüberdeckung, nicht die Anweisungsüberdeckung
* Weitere Tools sind Lcov, Cobertura und Maven

**Mutationstest**

Zufällige Fehler einfügen und prüfen, wie viele davon mit den Tests gefunden werden.

**Grenzen des Software Tests**

Folgende Gründe führen dazu, dass Tests oft schwierig und aufwändig sind:

* Unklare oder fehlerhafte Anforderungen
* Programmkomplexität
* Mangelnde Werkzeugunterstützung bei der Konstruktion
* Ausbildungs- und Fortbildungsdefizite
* Zeitprobleme

**Testautomatisierung**

Folgende Argumente sprechen für die Testautomatisierung:

* Vertrauen in die eigene Arbeit gestärkt
* Zeitersparnis beim Fehlerfinden und Korrigieren (kurzfristig für Entwickler)
* Sicherheit, den Code langfristig warten zu können (langfristig für Entwickler)
* Einfache Integration von gutgetesteten Code

**JUnit**

**Prinzip**

* Lässt die Tests automatisiert laufen.
* Lässt auch viele Tests gemeinsam laufen und fasst das Ergebnis zusammen.
* Vergleicht die Ergebnisse mit Erwartungen und teilt Unterschiede mit

**Ziele**

* Tests sollen einfach zu schreiben sein
* Es soll einfach sein, das Schreiben von Tests zu lernen
* Schnelle Testausführung
* Einfache Testausführung
* Tests sollen zusammensetzbar sein

**Testsuites**

Wenn man nur einen Teil der Tests laufen lassen möchte (z.B. alle Smoketests, alle Tests zu einer Komponente etc. )

**Fixtures**

Vorbereitungscode und Nachbereitungscode für Testmethoden (Deklaration, Test setup, Aufräumen) 🡪 Nurten: Isolation des eigentlichen Testcodes, Vermeidung von Redundanz

**Assertions**

Beispielsweise fail, assertTrue, assertFalse, assertEquals sind Methoden mit denen sich gewisse Eigenschaften testen lassen.

**Konventionen**

Trennung von Testcode und dem Code under Test. Auch Namenskonventionen einhalten 🡪 Klasse, die eine andere Klasse testet, hat den Namen der zu testenden Klasse + „ Test“ (z.B. zu testen: Car.java, Testklasse: CarTest.java)

**Parametriesierter Test**

Instanzen für das Kreuzprodukt aus Testdaten, Elementen, und Testmethoden

**JUnit-Rules**

Mithilfe von Rules lässt sich die Ausführung von Tests beeinflussen.

**Categories**

JUnit Categories dienen dazu, Testmethoden zu gruppieren und nur Tests einer bestimmten Kategorie laufen zu lassen. Dies geschieht zusätzlich zu den TestSuites.

**Theories**

Idee: Allgemeingültige Aussage als Test formulieren und durch eine Reihe von Testwerten überprüfen.

**Runner**

Z.B. Test Runner sind für die Ausführung der Tests zuständig

**Kapitel 3.4 – Statische Analyse**

**Arten von statischen Analysen**

* **Automatisiert:**
  + Code
  + Dokumente, die formaler Struktur unterliegen
* **Manuell:**
  + Code
  + Dokumente, die formaler Struktur unterliegen
  + Dokumente, die keiner formalen Struktur unterliegen

**Vor- und Nachteile von Software Inspektionen**

|  |  |
| --- | --- |
| **Vorteile:**   * Beim Testen können Fehler andere Fehler überdecken * Sie können unvollständige Versionen inspizieren aber ohne weiteres testen * Sie können über die Fehlersuche hinaus Qualitätsmerkmale prüfen | **Nachteile:**   * Man kann prüfen, ob Spezifikationen eingehalten werden, aber nicht, ob Benutzeranforderungen erfüllt sind * Es können nicht / schwer Performance, Usability überprüft werden |
|  |  |

* **Inspektionen und Tests ergänzen sich!**

**Manuelle Prüfungen**

1. **Planung**

Definition der Reviewobjekte, Reviewer und Eingangs- bzw. Ausgangskriterien

1. **Einführung**

Alle Reviewer erhalten die nötigen Infromationen. Klarstellung von Bedeutung, Sinn, Zweck und Ziele des durchzuführenden Reviews

1. **Vorbereitung**

Mängel, Fragen, Kommentare notieren sich die Reviewer für die Sitzung

1. **Reviewsitzung**
   * **Ziel:** Beurteilung des Prüfobjekts in Bezug auf die Einhaltung und Umsetzung der Vorgaben / Richtlinien. Ebenfalls Aufzeigen von Fehlern/Unstimmigkeiten
   * **Ergebnis:** Objekt wird akzeptiert / nicht akzeptiert ggf. Änderungen
2. **Überarbeitung**
3. **Nachbereitung**

**Reviewarten**

|  |  |
| --- | --- |
| **Walkthrough** | **Inspektion** |
| **Technisches Review** | **Informelles Review** |

**Software Metriken**

**Definition nach IEEE**

Eine Softwarequalitätsmetrik ist eine Funktion, die eine Software-Einheit in einen Zahlenwert abbildet. Dieser berechnete Wert ist interpretierbar als der Erfüllungsgrad einer Qualitätseigenschaft der Software-Einheit.

**Arten von Metriken**

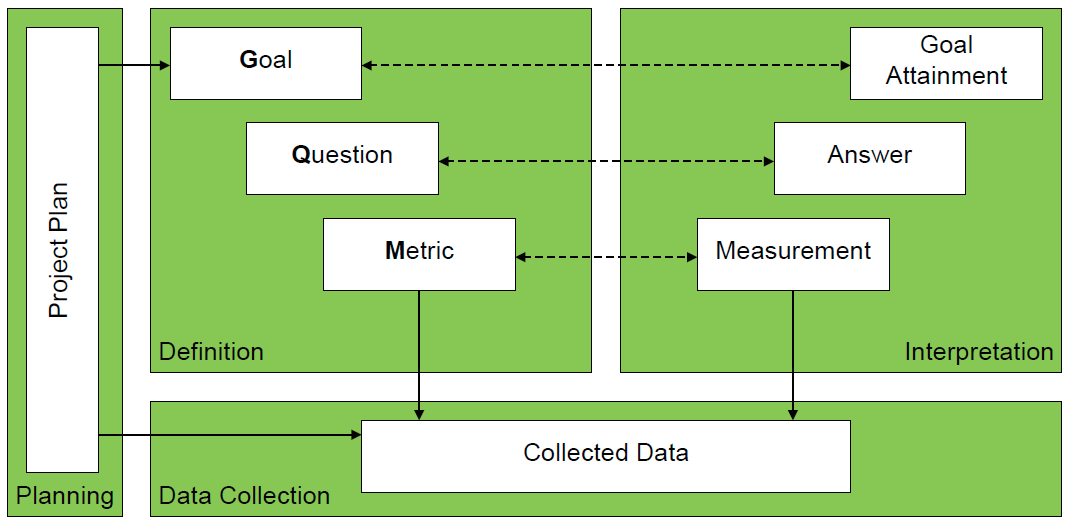
Metriken beziehen sich auf das **Software Management** und das **Software Produkt**. Es existiert folgende Klassifikation von Software-Metriken:

* **Prozess-Metriken**
* **Produkt-Metriken**
  + Statisch
    - Konventionell
    - objektorientiert
  + Dynamisch

**Auftretende Probleme in der Praxis**

* Nutzen von Metriken oft unklar
* Fehlen von Standards
* Programmierer wehren sich dagegen

**GQM (Goal-Question-Metric)-Ansatz**



**Gütekriterien von Software Metriken**

|  |  |  |
| --- | --- | --- |
| **Objektivität:** frei von subjektiven Einflüssen | **Robustheit:** Bei Wiederholung immer gleiches Ergebnis | **Korrelation:** Aussage der Metrik in Bezug auf die relevante Kenngröße |
| **Ökonomie:** billig | **Vergleichbarkeit:** Verschiedene Messungen der gleichen Komponente müssen in Relation gesetzt werden können | **Verwertbarkeit:**  Unterschiedliche Messergebnisse führen zu unterschiedlichem Handeln |

**Warum Metriken verwenden?**

* Beurteilung von selbst entwickelter Software
* Beurteilung von fremder Software / neuer Technologien
* Kostenabschätzung
* Finden von kritischen Stellen im Code

**LoC und NCSS**

* LoC (Lines of Codes)
* NCSS (Non Commented Source Statements)
* Vorteil: Einfach, schnell automatisiert zu erhalten
* Nachteil: Mangelnde Vergleichbarkeit für Module unterschiedlicher Sprachen (z.B. Java vs. Scala)
* Dennoch Basisparameter und für die Abschätzung der Verantwortung eines Coders

**Halstead Metriken**

* Metriken bauen auf der lexikalischen Struktur eines Programms auf
* Berechnung und Interpretation beruht auf empirischen Zusammenhängen
* Verwendet zur Bewertung der Wartbarkeit, oder zur Beurteilung der Entwicklung

**Prinzip**

Aufteilung des Sourecodes in:

* **Operatoren**: Schlüsselwörter, Operatioren, Klammern, …
* **Operanden**: Bezeichner, Konstanten

Anschließende Berechnung von Metriken anhand verschiedener Basisgrößen (Anzahl unterschiedlicher Operatoren / Operanden, Gesamtzahl der Vorkommen aller Operatoren, ..)

Laut Halstead besteht die Bezehung Aufwand E ist proportional zum Codevolumen V2

**Beispiel**

* Programm Volume: 1590,423
* Difficulty Level: 44,733
* Effort to implement: 71144,908
* Implementation time: 3952,495 (≈ 1h)

**Vor- und Nachteile:**

|  |  |
| --- | --- |
| **Vorteile:**   * Einfach zu ermitteln * Automatisiert auswertbar * Für alle (textuellen) Programmiersprachen geeignet * Empirisch bestätigt als gutes Maß für Komplexitat | **Nachteile:**   * Berücksichtigung ausschließlich lexikalischer Komplexität * Konzepte, wie Sichtbarkeit, Namensräume nicht gegeben * Aufteilung Operator / Operand sprachabhängig |

**McCabe Metrik**

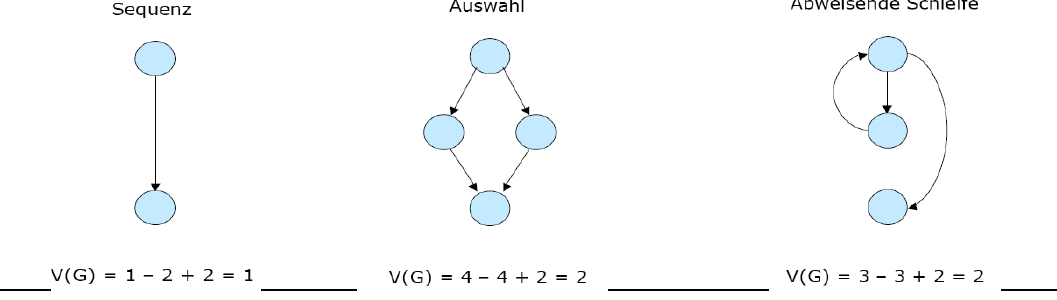
* McCabe Metrik = zyklomatische Komplexität
* Ziel ist es, die Komplexität einer Methode in einer Zahl auszudrücken
* Basiert auf Graphentheorie
* Entspricht der Anzahl linear unabhängiger Pfade auf dem Kontrollflussgraphen eines Moduls
* Entspricht der mindestens nötigen Anzahl von Testfällen für vollständige Zweigüberdeckung

**Definition aus der Graphentheorie**

**V(G) = e – n + 2 \* p** (e = Anzahl der Kanten, n = Anzahl der Knoten, p = unabhängige Teile des Graphen, p = 1, wenn Graph zusammenhängend ist)

**Definition für zusammenhängende Graphen**

**V(G) = e – n + 2** (e = Anzahl der Kanten, n = Anzahl der Knoten)



**Bedeutung der McCabe Metrik in der Praxis**

|  |  |
| --- | --- |
| **V(G)** | **Risiko** |
| 1 – 10 | Einfaches Programm, geringes Risiko |
| 11 – 20 | Komplexeres Programm, erträgliches Risiko |
| 21 – 50 | Komplexes Programm, hohes Risiko |
| > 50 | Untestbares Programm, extrem hohes Risiko |

* **Bei Umstrukturiereung, mit der Komponente beginnen, die die höchste zykolomatische Komplexität hat!**

**Vor- und Nachteile:**

|  |  |
| --- | --- |
| **Vorteile:**   * Einfach zu berechnen * Gute Korrelation zwischen zyklomatischer Zahl und Verständlichkeit der Komponente * Für Berechnung des Testaufwands geeignet | **Nachteile:**   * Metrik berücksichtigt nur Kontrollfluss * Komplexität von Datenfluss nicht berücksichtigt * Metrik vs. Subjektives Empfinden stimmt nicht überein (teilweise) * Teilweise unterschiedlich implementiert |

**McCabe in der Objektorientierung**

* **Methodenbasiert (Normalfall):** Kontrollstrukturen innerhalb von Methoden untersucht
* **Prozessbasiert:**
  + Methoden eines Prozesses werden als unabhängige Teilgraphen betrachtet
  + e = Kanten der Kontrollgraphen aller Methoden
  + n = Knoten der Kontrollgraphen aller Methoden
  + p = Anzahl der beteiligten Methoden

**🡪 Aussage über Gesamtkomplexität von Prozessen**

**Objektorientierte Metriken**

* **Komponentenmetriken:** Bewertung einzelner Komponenten (Klassen, Paket, …)
* **Strukturmetriken:** Bewertung des Klassenverbunds als Ganzes

**Beispiele für Komponentenmetriken**

* **OV (Anzahl Object Variables)**
* **CV (Anzahl Class Variables)**
* **NOA (Number of Attributes OV + CV)**
* **Hierbei handelt es sich um Umfangsmetriken mit hoher Signifikanz**
* **Weighted methods per class (WMC):** Summe der Gewichtung der einzelnen Methoden
* **Depth of Inheritence Tree (DIT):** Größter Abstand von der Wurzel des Vererbungsbaums bis zur betrachteten Klasse (wird von 0 an gezählt)
* **Number of Descendants (NOD):** Zahl der Kinder, die von der betrachteten Klasse direkt erben
* **Coupling between Object Classes:** Anzahl der Klasse, die an die betrachtete Klasse gekoppelt sind

**Konformitätsanalyse**

* **Syntaxanalyse:** In Compiler integriert 🡪 keine Details
* **Semantik Analyse:** Fragwürdige / fehleranfällige Codestellen identifizieren

**False Negatives**

Hierbei handelt es sich um Probleme, die Tools melden, die aber keine Probleme sind.

Umgang damit:

1. Verstehen
2. Durch einfache Änderung im Programm vermeiden
3. Oder: Meldung ausblenden

**Exploit Analyse**

Analyse von sicherheitstechnischen Programmschwachstellen

**Anomalieanalyse**

Suche nach ungewöhnlichen oder auffäligen Codesequenzen

**Unterscheidung der unterschiedlichen Anomalien**

* **Kontrollflussanomalien**
  + Unstimmigkeiten im Programmablauf
  + Schwer automatisiert zu finden
* **Datenflussanomalien**
  + Anweisungssequenzen, die zweifelhaft Variablenzugriffe enthalten

**Bestimmung von Datenflussanomalien**

* Zunächst Datenflussgraphen erstellen
* Folgende Notationen gibt es:
  + d(x): x wird definiert (geschrieben)
  + r(x): x wird referenziert (gelesen)
  + u(x): x ist undefiniert (unbesetzt)
* Folgende Operationen in einem programm sind Anomalien:
  + dd: Variable wird zweimal hintereinander überschrieben
  + du: Variable wird geschrieben und dann gelöscht
  + ur: Undefinierte Variable wird verwendet

**Kapitel 4 – Vorgehensmodelle**

**Allgemein**

**Tätigkeiten und Artefakte in einem SW Projekt**

|  |  |
| --- | --- |
| 1. **Analyse:** Systemspezifikation | 1. **Test und Integration:** Lauffähige SW auf produktionsnaher Umgebung, Integrationstests Testberichte |
| 1. **Design:** Technisches Design | 1. **Abnahme und Einführung:** Abnahmeprotokolle, Testprotokolle, Schulungsunterlagen, Installationsanleitung etc. |
| 1. **Implementierung und Modultests:** Programmcode, Modultests, Testberichte, Code Doku etc. | 1. **Betrieb und Wartung** |

* **Es gibt kein ideales Vorgehensmodell für alle Projekte**

**Wichtige Vorgehensmodelle (nicht agil)**

|  |  |  |
| --- | --- | --- |
| Code and Fix | Wasserfallmodell | Prototypen |
| V-Modell | Spiralmodel | Inkrementelle Entwicklung |

**Agile Methoden**

**Kritik an klassischen Vorgehensmodellen**

* Es müssen viele Dokumente erzeugt und gepflegt werden
* Modelle oft schwer verständlich
* Prozessbeschreibungen hemmen Kreativität
* **Alternativer Ansatz: agile Prozesse**

**Prinzipien der agilen Entwicklung**

|  |  |  |
| --- | --- | --- |
| Kunden zufrieden stellen | Anforderungsänderungen selbst spät in der Entwicklung möglich | Liefern von funktionierender Software regelmäßig innerhalb weniger Wochen |
| Wichtigstes Fortschrittsmaß: Funktionierende Software | Motivation der Individuen durch Vertrauen | Informationsverteilung durch persönliche Gespräche |
| Tägliche zusammenarbeit Entwickler + „Fachexperte“ | Nachhaltige Entwicklung | Gleichmäßiges Entwicklungstempo |
| Gutes Design |  |  |

**Kritik an agilen Methoden**

* Bedarf an Regeln - „Save your ass“ - Mentalität
* Angst, sich zu blamieren

**Scrum**

**Prinzipien**

Transparenz, Überprüfung, Anpassung

**Rollen**

* Product Owner - Scrum Master
* Entwicklungsteam - (Customer, Manager, User)

**Meetings**

* Sprint Planning (1&2) - Daily Scrum
* Sprint Review - Retrospective
* (Estimation Meeting)

**Artefakte**

* Product Backlog - Sprint Backlog
* Produkt Inkrement - (Vision, Sprint Goal, Taksk, Releaseplan, ...)
* (Impediment Backlog)

**Definition of Done**

Spezifiziert die Kriterien, ab wann eine Story done ist.

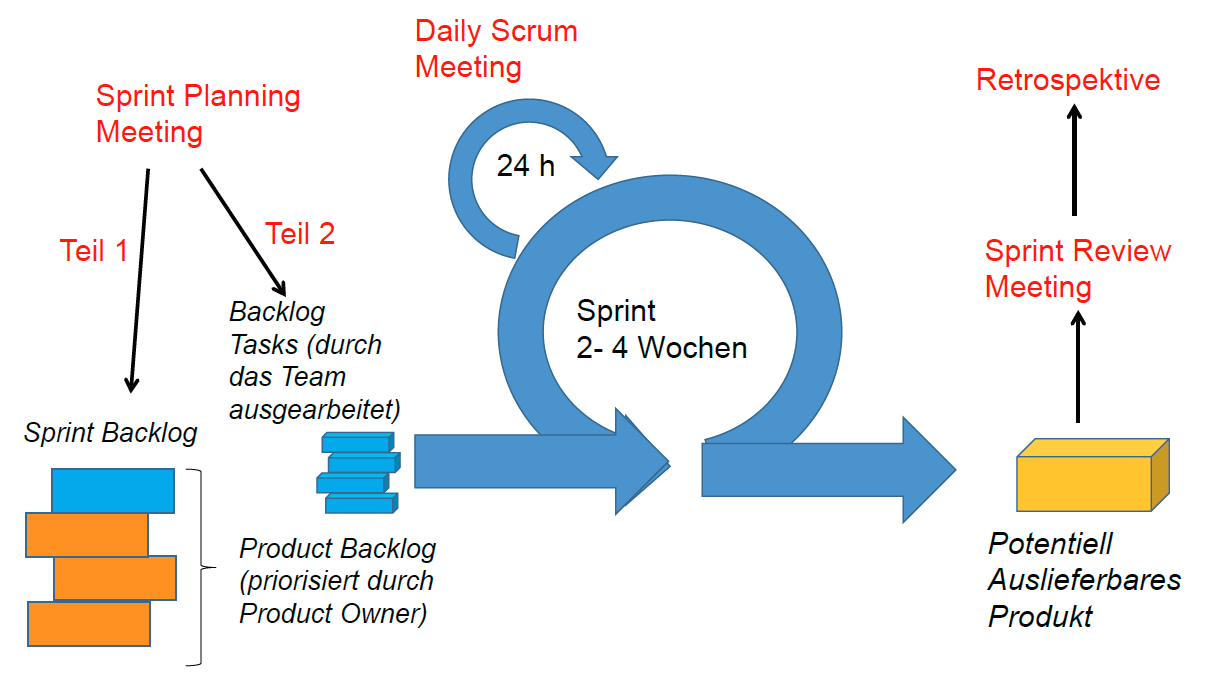
**Arbeitsprinzipien bei Scrum**

* Selbstorganisation - Pull Prinzip
* Timebox - Nutzbare Funktionalität

**Rollen im Detail**

* **Product Owner:** 
  + Verantwirtlich, dass die Entwickler gut arbeiten
  + Managemente des Product Backlogs
* **Entwicklerteam:**
  + Entwickelt bis zum Ende des Sprints ein lauffähiges Software Inkrement
* **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

**Der Scrum Prozess**



**Der Sprint**

Dieser Umfasst das Sprint **Planning,** die **Daily Scrums**, die **Entwicklung**, das **Sprintreview** und die **Sprint Retrospektive**.

Während des Sprints werden keine Änderungen vorgenommen, die das Sprintziel gefährden. Auch der Qualitätsanspruch wird nicht gefährdet.

**Sprint Planning**

* Planung der Arbeit für nächsten Sprint
* Oft zweigeteilt:
  + Anforderungsworkshop: Selected Product Backlog
  + Designworkshop: Liste der Tasks für den nächsten Sprint

**Sprint Planning**

Beantworten der Fragen:

* Was habe ich gestern getan?
* Was werde ich heute tun?
* Was hindert mich an meiner Arbeit?

**Sprint Review**

Diskussion der Ergebnisse des Sprints 🡪 Vorführung des Inkrements

**Retrospektive**

Scrum Team kann sich selbst verbessern 🡪 Was haben wir gut gemacht, was könnten wir für nächsten Sprint verbessern.

**Estimation Meeting**

Backlog wird einmal (oder mehrfach) pro Sprint neu priorisiert

**Schätzen von Story Points**

* Je höher der Aufwand, desto höher der Nutzen
* Einheit ist aber nicht festgelegt
* Die Scala ist meist in Fibonaccizahlen
* Vorgehen:
  + **Planning Poker:** Einigung auf Referenz-Backlog-Item, Schätzen der Storypoints, Wiederholung bis eindeutiges Ergebnis vorliegt
  + **Magic Estimation:** Jeder kann BL-Items einer bestimmten Skala zuordnen und solange verschieben, bis ein eindeutiges Ergebnis vorliegt.
* **Zuverlässiges Schätzen oft schwer bis unmöglich!**

**Reporting des Fortschritts und Prognosen**

* Taskboard (Scrum Board)
* Verschiedene Formen von Charts (Burndown etc)
* Berechnung der Velocity (Geschwindigkeit)

**Scrum in großen Projekten**

* Es existieren mehr als zwei Scrum Teams
* Prinzipiell zwei Arten:
  + **Organisches Wachstum:** Einarbeitung neuer Mitglieder, Team entscheidet wann es sich teilt
  + **Sprunghaftes Skalieren:**  Die Mitglieder des initialen Teams übernehmen die Rolle des Product Owners in den neuen Teams
* Zwei Möglichkeiten der Teamaufteilung
  + **Component Teams:** Verantwortlich für technische Komponenten
  + **Feature Teams:** Verantwortung für fachliche Funktionen
* Synchronisation der Teams durch:
  + Scrum of Scrums, Product Owner Team, Scrum Master Group
  + Virtuelle Teams, Gemeinsame Planning Meetings
* Beispiel für die Organisation:
  + Eigene Teams: Tests, Integration, Features …

**Kapitel 5 – Requirements Engineering**

**Verschiedene Arten von Anforderungen**

* **Funktionale Anforderung**: Anforderungen bezüglich des Ergebnisses eines Verhaltens einer Funktion
* **Qualitätsanforderung:**  Anforderungen, die sich auf ein Qualitätsmerkmal beziehen
* **Randbedingung:** Anforderung, die den Lösungsraum einschränkt, dass funktionale Anforderungen und Qualitätsmerkmale erfüllt werden

**RE in Agilen Projekten**

* **Wasserfall:** Eigene Projektphase
* **Agile Prozesse:** kontinuierlicher phasenübergreifender Prozess in die Systementwicklung integriert 🡪 Verwendung des Product Backlogs

**Anforderungsgegenstände in agilen Projekten**

* **User Story:** 
  + Karte
  + Konversation
  + Akzeptanzkriterien
* **Epic:** Sehr große User Stories, die schwer geschätzt und nicht in einem Sprint bearbeitet werden können
* **Thema:** Zusammenfassung mehrerer User Stories
* **Product Backlog Items:** Themes, Epics, User Stories
* **Für das Sprint Planning:** User Stories

**User Stories**

* In der Sprache des Kunden - Keine Benutzeroberfläche
* Verwendung von Benutzerrollen - Keine Technik
* Verwendung von Sprachtemplates (Als Anwender () benötige ich eine (.), damit ich (-)

**Cohnsche Form für User Stories**

1. Verwendung der Ich-Form führt zur besseren Identifikation mit User Story
2. Einheitliche Struktur hilft beim Priorisieren
3. Sinnvolle Struktur

**Schneiden von User Stories**

* Vertikales Schneiden
* Schneiden nach Daten, Aufwand, Qualität, Forschungsanteilen, Benutzerrolle

**Andere Themen in User Stories packen**

* Nichtfunktionale Anforderungen
* Fehler
* Refactorings
* Bucktrackingsystem installieren… oder ähnliches

🡪 Umformulieren der Stories, dass Zweck klar wird

Folgende Dinge können durchgeführt werden, um mit der Problematik umgehen zu können:

* Zusätzliche Benutzerrollen (künstlich) einführen
* Storyübergreifende, nicht-funktionale Anforderungen als Constraints formulieren
* Fehler sofort beheben (oder werden sofort in nächsten Sprint eingeplant)
* Führen eines Technischen Backlogs

**A-TTD**

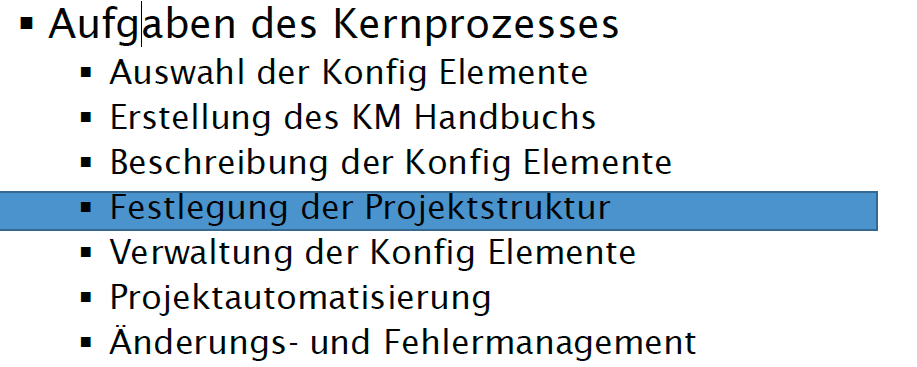
* Acceptance Test Driven Development
* Tests als Requirements, Requirements als Test
* Workhop zum Klären der Requirements

**Kapitel 6.1 – Konfiguration Management**

Man verwendet Konfigurationsmanagement, um Projektergebnisse sicher zu verwalten sowie den Teammitgliedern kontrolliert Zugriff geben.

**Ziele des Konfigurationsmanagements**

* Änderungen kontrollieren - Qualität sicherstellen
* Kommunikation vereinfachen / Transparenz verbessern
* Produktivität steigern

**Aufgaben des Kernprozesses**

**Konfigurationselemente**

Alle Elemente, die zur Erstellung des SW Produgts nötig sind oder dieses beschreiben.

Beispiele hierfür wären:

* Quelltext - Anforderungsdokumente - Schnittstellenverträge
* Dokumentationen - Build Skripte

Keine **Konfigurationselemente wären z.B.:**

* Meetingprotokolle - Binäre Auslieferungsdateien
* Generierte Dateien - Projektpläne

Konfigurationselemnte werden im **KM Handbuch** beschrieben mit mindestens zwei Angaben (kurze inhaltliche Beschreibung bzw. Namenstemplate)

**Anforderungen für das 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 Konfigurationselementen - Nach Projektstruktur
* Nach Softwarearchitektur

**Repositories**

* Verwaltung der Konfigurationselemente
* Alte Stände wiederherstellen
* Sicherstellung der Verfügbarkeit der Dateien
* Parallele Änderungen (2 Möglichkeiten)
  + **SVN** 🡪 Lock – Modify – Unlock (Reserved Checkout)

**Vorteil: Niemand wird blockiert**

* + **GIT**  🡪 Copy – Modify – Merge (Unreserved Ceckout)
    - **Vorteil: Vermeidung von Konflikten auf für binäre Daten**
* **Tags** statt Versionsnummer 🡪 Leichter zu merken
* **Branches** für Features erstellen (anschließend Merging)
* **Releases:** Softwarestand, welcher an den Kunden gegeben werden könnte

**Projektautomatisierung**

Die Schritte in der Projektautomatisierung sehen folgendermaßen aus:

1. Quellelemente ermitteln
2. Produkt aus Quellelementen erstellen
3. Produkt prüfen
4. Produkt ausliefern
5. Ergebnisse dokumentieren

**Varianten der Projektautomatisierung (Buildprozesse)**

* + Entwicklerbuild: lokal
  + Integrationsbuild: regelmäßig 🡪 Qualitätssicherung
  + Releasebuild: Ähnlich Integrationsbuild + Setzten eines Release Tags
  + Umsetzung durch: Shell Skripte, Ant, Maven

**Änderung- und Fehlermanagement**

**Änderungsmanager und CCB (Change Control Board)**

* Verwaltung der Change Requests (CRs) und Fehlermeldung durch einen Änderungsmanager (Vorfilterung, Bewertung)
* **Change Control Board:** Entscheidung über Annahme oder Ablehnung eines CRs / Fehlers

**Workflow- und Raumkonzept**

**Workflow**

Der Workflow sieht folgendermaßen aus:

1. Entwickeln der Quellen, kompilieren und testen der einzelnen Einheiten
2. Integrieren und testen einer Konfiguration
3. Erzeugen und testen eines Releases
4. Systemtest 🡪 Testen des Gesamtsystems
5. Produktive Nutzung

**Raumkonzept**

Hierbei handelt es sich um Systemumgebungen zur Unterstützung des Workflows.

* Entwicklungsumgebung - Testumgebung
* Integrationsumgebung - Produktivumgebung

**Kapitel 6.2 – Subversion**

**Funktionalität**

* **Zentrale Versionsverwaltung (EIN Server, EIN Repo)**
* Versionierung von Dateien und Verzeichnissen
* Versionierung von Kopier, Lösch und Umbennenungsvorgängen
* Atomare Check-Ins
* Unterstützung für Metadaten
* Flexible Client Server Architektur

**Architektur**

* **Client:** Lokaler Arbeitsbereich, SVN Client
* **Server:** Apache Server plus subversion Modul oder Subversion server, SVN Repo

**Arbeitsweise des Repositories**

* **Revisionen:** Für jede schreibende Transaktion 🡪 Revisionsnummer
* **Changeset:** Alle Änderungen im Rahmen der Transaktion
* **SVN kennt keine echten tags und branches 🡪 Als normale Verzeichnisse im Repo abgebildet (SEMANTIK beachten!!)**

**Grundlegende Interaktionen der Arbeitskopie**

* svn checkout 🡪 Arbeitsbereich anlegen
* svn update 🡪 Arbeitskopie aktualisieren
* svn commit 🡪 Änderungen ins Repo schreiben

**Empfohlener Arbeitsprozess**

1. Arbeitsbereich aktualisieren
2. Prüfung auf Konflikte
3. Änderungen durchführen
4. Änderungen ins Repository schreiben

**Kapitel 6.3 – Git**

**Funktionalität**

* **Dezentrale Versionsverwaltung (Kein ausgezeichneter Server, Repo auf jedem Client / Server)**
* Hohe Performance
* Effiziente Arbeitsweisen
* Backup, Wartbarkeit
* Offline Fähigkeiten

**Commit**

* Jede Version wird als Commit ein einem Repository abgelegt
* Ein Commit umfasst immer das ganze Projekt
* Jeder Commit hat einen Commit Hash (SVN nur Nummern)
* Historien sind bei Git lokal 🡪 Offizielle Historie durch Einrichtung eines ausgezeichneten Repos

**Weitere Eigenschaften**

* **Staging:** Commit ist zweistufiger Prozess: 1. In Stage Bereich schreiben 2. Von dort ins repo schreiben.
* **Branching:** Für jeden Entwicklungszweig (Feature, Entwickler) ein Branch. Pro Git Rpository existier immer genau ein aktiver Branch
* **Merging:** Zunächst automatisch, außer bei Konflikten
* **Rebasing:** Historie glätten indem man einen branch an einem anderen Knoten ansetzen lässt (Wenn Commits versehentlich im falschen Branch ausgeführt wurden)
* **Fetch / pull:** Branches aus einem anderen Repo holen
* **Push:** Zum schreiben in ein anderes Repo
* **Remote tracking Branch:** Spezieller lokaler branch, der anzeigt, wo ein remote branch zum Zeitpunkt des letzten updates stand

**Kapitel 6.3 – Maven (Buildsystem)**

**Maven beantwortet folgende Fragen:**

* Was ist nötig, um das Projekt zu bauen?
* Welche Libraries muss ich downloaden?
* Wo lege ich die Libraries ab?
* Was muss ich tun, um den build durchzuführen?

**Allgemein**

* Buildsystem
* Dependency Management
* Widerverwendung der Buildlogik
* Vereinheitlichung /Tool Integration
* Einfache Suche von Artefakten
* Erstellen der Dokumentation

**Funktionsumfang**

1. Durchführung des Build Prozesses
2. Verwaltung der Abhängigkeiten externer Bibliotheken (alle Plugins und libs werden bei Bedarf eingebunden und geladen)
3. Erstellen der Projektdokumentation

**Maven Plugins und Goals**

Ein **Maven Plugin** ist eine Kollektion von goals. **Goals** sind die Tasks, die die Aufgaben durchführen.

Plugins existieren für:

* Erzeugnis von jar Files
* Kompilieren des Source Codes
* Durchführen von Unit Tests etc.

**Aufruf eines Goals**

1. Maven sucht im lokalen Repositore nach dem Plugin. Ist es nicht vorhanden 🡪 Plugin wird aus Remote Repository geladen
2. Goal des Plugins wird ausgeführt

**Project Object Model (POM)**

Eine deklarative Beschreibung des Projekts. Jedes Goal hat Zugriff auf die Information des POMS. **Dieses File ist Dreh- und Angelpunkt!**

Folgende Informationen sind im POM enthalten:

* Verhalten der Plugins wird konfiguriert
* Abhängigkeiten werden konfiguriert
* Informationen zum Projekt gehalten

**Aufbau des POM**

An der Spitze steht das Projekt und dann folgen daraus:

* Allgemeine Angaben - Properties
* Build-Prozess - Build-Varianten
* Repositorys - Abhängigkeiten zu libs
* Projektdokumentation - Auslieferung des Produktes

**Kapitel 6.5 – Nexus**

* Repository Manager (Proxy) zwischen lokalem Rechner und Remote Repos
* Hosten eines Repos für firmeneigene Artefakte
* Core Capabilities:
  + Management der Software-Artefakte
  + Management von Metadaten
  + Proxy von externen Repos
  + Infrastruktur für Management der Artefakte
* Sogar private Repos:
  + Proxy Repos
  + Hosted Repos
  + Virtual Repos
* Repos können zu Gruppen zusammengefasst werden

**Kapitel 6.6 – Integration**

* Integrationsprobleme:
  + Merge Konflikte
  + Compile Konflikte
  + Test Konflikte
* Integration ist aufwendig und Aufwand steigt u.a. mit der Zeit seit der letzten Integration 🡪 **Abhilfe Continuous Integration**

**Continuous Integration**

* Anstelle von großen, langen Integrationsphasen 🡪 Häufige kurze Integrationen
* Workflow:
  + Ausschecken und anschließend Kodieren
  + Automatischer Build auf lokaler Maschine bis Tests grün sind
  + Merge mit den letzen Änderungen bis Tests grün sind
  + Commit
  + Automatisierter Build auf sauberer Integrationsmaschine
  + Ggf. Bugfixes
* Vorteile:
  + Bugs werden schneller gefunden
  + Risiko minimiert
  + Häufiges Deployment möglich 🡪 schnelles Benutzer Feedback
  + Bessere Sichtbarkeit des Projekts

**Kapitel 6.7 – Hudson (Continuous Integration)**

* Default support für CVS und SCN
* Build Automatisierung
* Automatisiertes Deployment in Produktionsumgebung
* Testautomatisierung (XUnit, Funktionale Tests)
* General Purpose Scheduler
* Für Integration ist ein CI Server nötig
* Integrationsbuild bei jedem Commit durchführen