# QA und VVT

Bettina Buth[1]

[1]Fachbereich Elektrotechnik und Informatik
HAW Hamburg

Sommersemester 2005

# Teil I

## Einleitung

# Übersicht

- Einführung und Motivation
- Software Qualitätssicherung
- Software Verifikation, Validation und Test

# Motivation und Hintergrund



- Entwicklung von Softwareprodukten
  - als allgemeines Produkt

    Bsp

    Datenbanken, Textverarbeitungsprogramme, Grafikpakte, Softwareentwicklungsumgebungen

  - als bestelltes oder angepasstes Produkt

    Bsp

    Fahrstuhlsteuerung, ISS Mission Control Center, Flugsicherungssoftware, Airbus Control System, spezielle Entwicklungswerkzeuge

$\Rightarrow$ verschiedene Anwendungsdomänen mit verschiedenen Anforderungen an SWE

# Merkmale guter SW – Qualität

- funktional
  - Korrektheit
  - Vollständigkeit ( von Requirements, gegen Requirements, gegen Design)
  - Testbarkeit
- nicht-funktional
  - Wartbarkeit (Lesbarkeit, Kommentardichte und -güte, Fehlerrate, ...)
  - Zuverlässigkeit (Verlässlichkeit, Zugriffsschutz, Betriebssicherheit = Safety, Security, Availability, Reliability)
  - Effizienz (Performanz, Speicherverbrauch,
  - Benutzerfreundlichkeit (Usability, Ergonomieaspekte)
  - Portabilität

# Literatur

- Allgemein
  - Ian Sommerville: Software Engineering, Pearson und Addison-Wesley
  - Andreas Spillner, Tilo Linz: Basiswissen Softwaretest; dpunkt.verlag
  - Peter Liggesmeyer: Software Qualität; Spektrum Akademischer Verlag
- Standards
  - ISO 9001
- Spezielle Lektüre
  - Bernd Österreich: Analyse und Design mit UML 2 Objektorientierte Softwareentwicklung; Oldenbourg Verlag
  - Uwe Vigenschow: Objektorientiertes Testen und Testautomatisierung in der Praxis; dpunkt.verlag
  - Peter Hruschka, Chris Rupp: Agile Software-Entwicklung für Embedded Real-Time Systems mit der UML; Hanser Verlag
  - Norman E. Fenton, Shari L. Pfleeger: Software Metrics - A Rigorous and Practical Approach; PWS Publishing Company
  - Kent Beck: Test-Driven Development by Example; Addison-Wesley - Pearson Education
  - David Astels: Test-Driven Development - A Practical Guide, Prentice Hall (Coad Series)
- WebLinks (auch via Google)
  - JUnit `http://www.junit.org`
  - NUnit `http://www.nunit.org`

# Motivation und Hintergrund

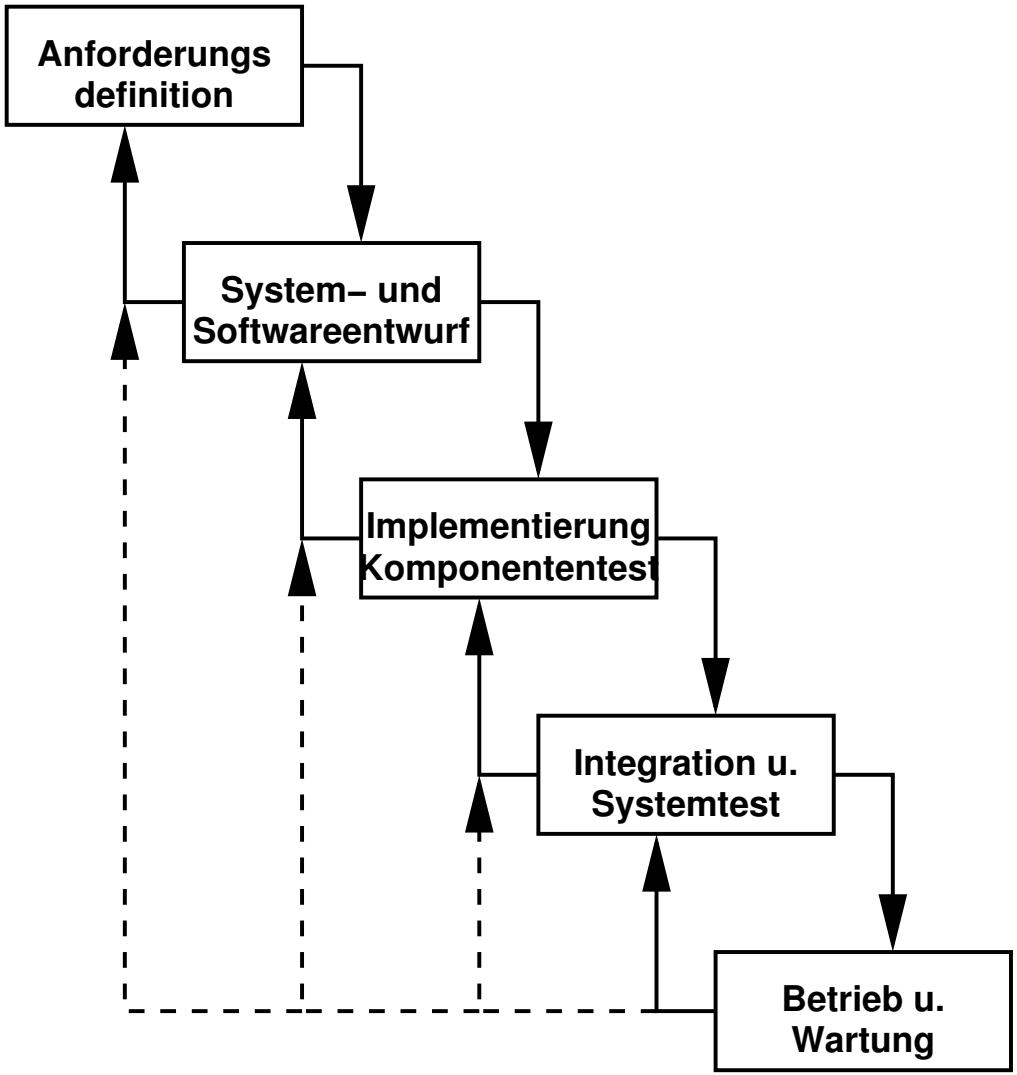
- Einführung und Motivation
  - Literatur
- Lifecycle Modelle
- Prozesse in Projekten Bsp ECSS

# Gebräuchliche Varianten

- Wasserfallmodell (Waterfall Model)
- Evolutionäre Modelle
- V-Modell
- Formale SWE
- Iterative Modelle - als Meta-Modell
- Spiralmodell - als Metamodell
- Agile Softwareentwicklung (XP, FDD, TDD, ...)
- auch: spezielle Entwicklungsmodelle für Wiederverwendung

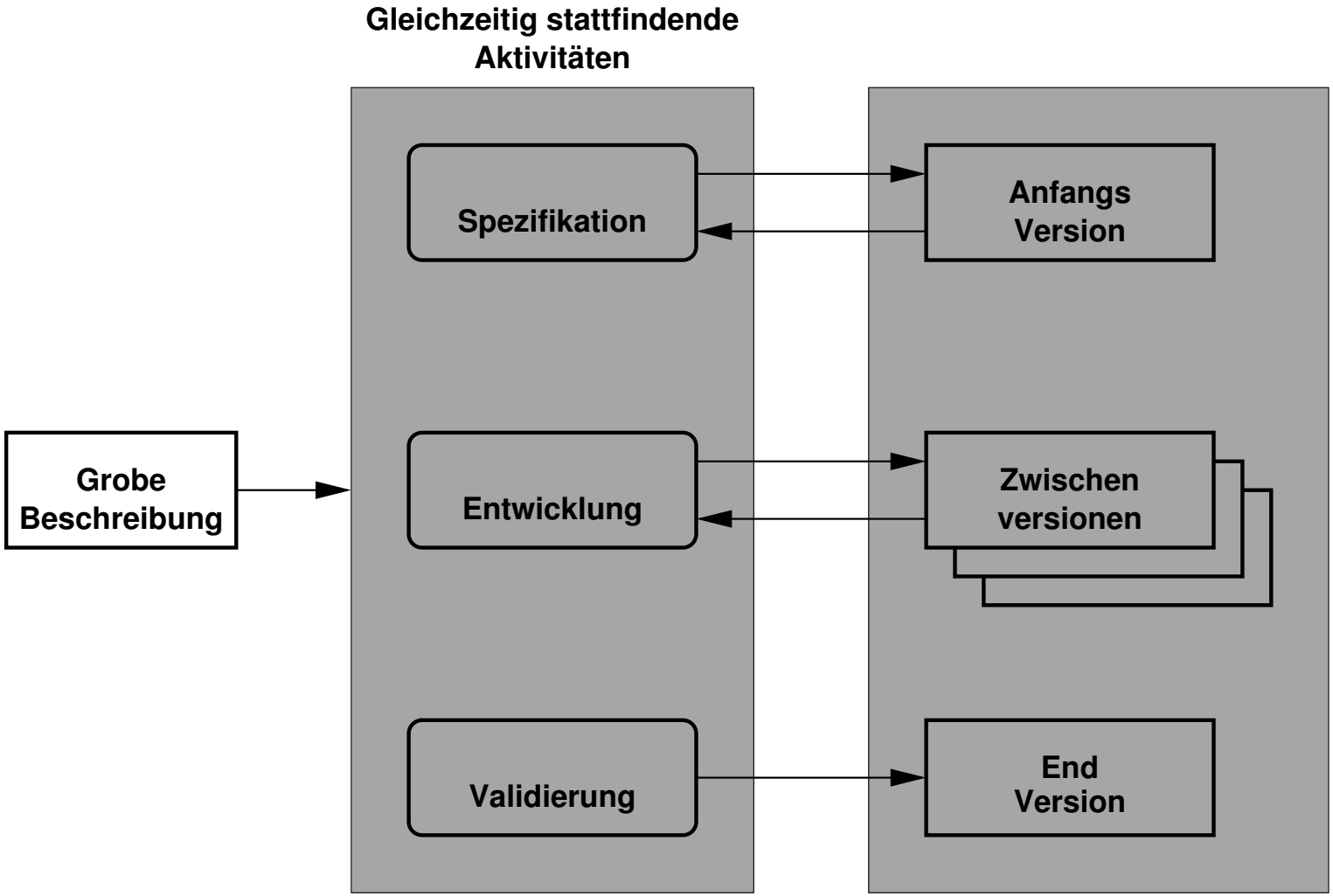
# Wasserfallmodell

## nach Sommerville, Software Engineering, Abbildung 3.1



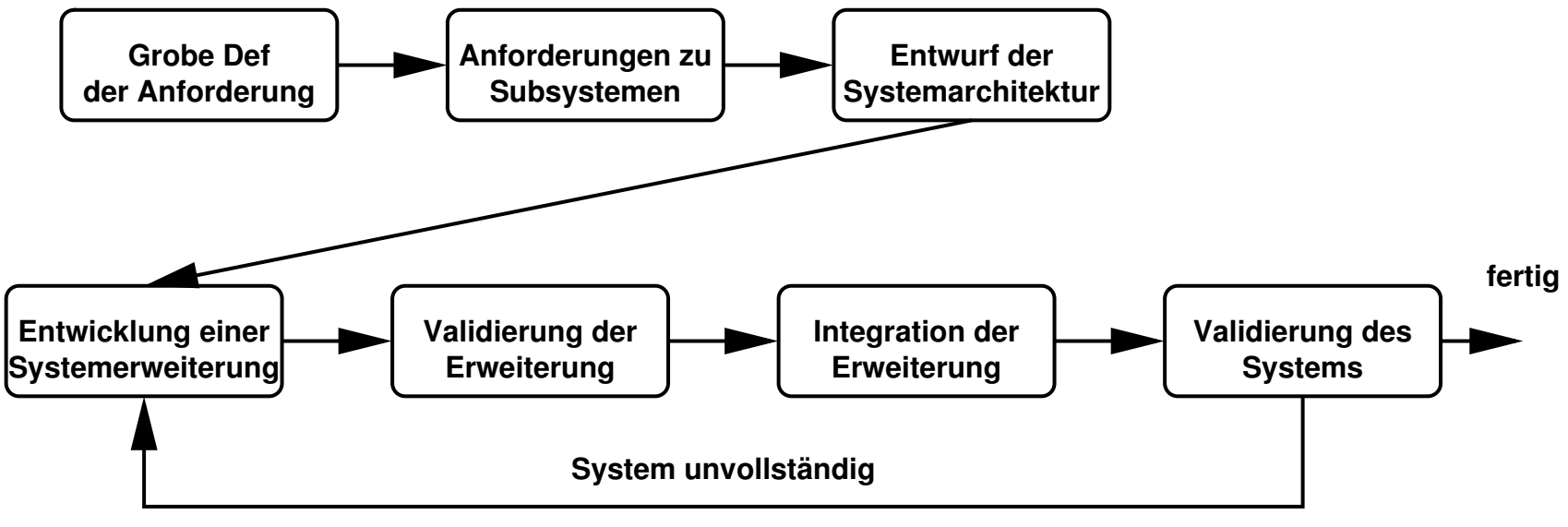
# Inkrementelle SWE

## nach Sommerville, Software Engineering, Abbildung 3.2



# Iterative SWE

## nach Sommerville, Software Engineering, Abbildung 3.6



# Agile SWE

- Alternativer Prozess für Softwareentwicklung
- Motivation: Prozess der Probleme bei SWE in den Griff bekommt
  - Nicht-Einhaltung von geplanten Entwicklungszeiten
  - zu hohe Kosten durch Revisionen von Requirements, Design etc.
  - Am Kunden vorbei entwickelt
  - Fehler werden zu spät gefunden
  - Reviewgetriebene Prozesse unpraktikabel
- Agile SWE = Familie von Ansätzen - bekanntester: XP
- Gleichartige Konzepte : meist iterative Entwicklung
- Agile Alliance – Agenda
- überwiegend OO und Java - aber nicht nur

# Agile Ansätze

- XP (Extreme Programming)
- RUP
- Cockburn's Crystal Family
- Highsmith's Adaptive Software Development
- Scrum
- Feature Driven Development (FDD)
- DSDM (Dynamic System Development Method)
- AM (Agile Modelling)
- TDD (Test Driven Development)

# Motivation und Hintergrund

- Einführung und Motivation
  - Literatur
- Lifecycle Modelle
- Prozesse in Projekten Bsp ECSS

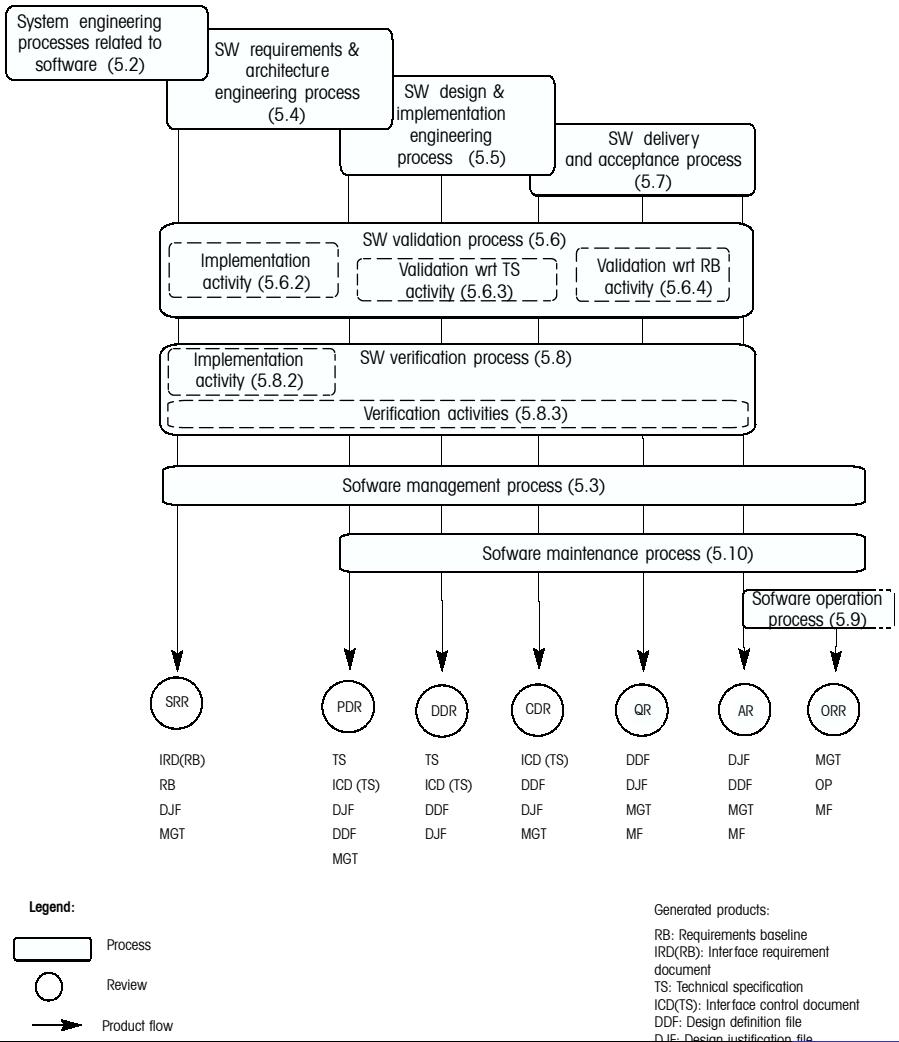
# ECSS Standards Overview

- ECSS = European Cooperation for Space Standardization
- Industriestandards für Entwicklungen im Raumfahrtbereich
- 3 Branches: M – E – Q
- Relevant für SW (aktueller Stand 2003/04)
  - ECSS-E-40 Space Engineering - Software (part1 B, part2 B)
  - ECSS-Q-80B Space Product Assurance - Software Product Assurance
- Level 3 Standards (ECSS-E-40-xy, ECSS-Q-80-xy) für spezielle Aspekte; z.B.
  - ECSS-Q-80-01: SW Reuse
  - ECSS-Q-80-03: SW Dependability Methods
  - ECSS-Q-80-04: SW Metrication
- frei verfügbar nach Registrierung von `http://www.ecss.nl/`

# ECSS Prozesse (1)

## Nach ECSS-E-40





# ECSS Prozesse (2)

- QA als supporting process
- steuert nicht die Entwicklung, sondern an der Entwicklung angehängt
- Verifikation, Validation und Test als Engineering Aufgaben
- in der Praxis oft: QA nur mit CM Aufgaben betraut, eigentliche QA-Aufgaben und speziell VVT nicht durchgeführt

# Teil II

# Qualitätssicherung

# Übersicht

- Einführung und Motivation
- Software Qualitätssicherung
- Software Verifikation, Validation und Test

# Qualitätssicherung

- Qualitätssicherung
  - Aufgaben und Ziele von SW QA
  - SW Metriken und QA
  - Prozessverbesserung
  - Zusammenfassung

# SW QA - Aufgaben und Ziele

- SW QA = Software Quality Assurance (deutsch: Qualitätssicherung, QS)
- Aufgaben innerhalb von Projekten (Produktsicherung)
  - allgemein: Sicherstellung der Qualität des Endproduktes
  - Überwachen der Prozessqualität (abhängig von dem Vorgehensmodell)
  - Prüfen der Produktqualität
  - auch: Risikoprüfung
  - zum Teil: Verifikation und Validation
  - Prüfung und Freigabe von Dokumenten und (Zwischen-)produkten
  - auch Wareneingangsprüfung - zB. für gelieferte SW
- Projektübergreifend
  - Prozessverbesserung
  - Erarbeitung und Wartung von Hausstandardards
  - Unterstützung von Zertifizierung (ISO 9000, CMMI, SPICE)
- Schnittstellen zu SW Requirements Engineering, SW Design, SW Implementierung, SW Verifikation, Validation und Test, SW Konfigurationsmanagement, SW Versionsmanagement

# SW QA - Qualitätskriterien

- Produkt
  - Korrektheit der Anforderungen
  - Korrektheit gegen Anforderungen
  - Vollständigkeit der Anforderungen
  - Vollständige Umsetzung der Anforderungen
  - Zuverlässigkeit: Verfügbarkeit, Sicherheit, Ausfallsicherheit (engl. Dependability = Availability + Safety + Security + Reliability)
  - Wartbarkeit
  - Benutzerfreundlichkeit
  - Portierbarkeit
  - Effizienz
  - auch: Kosten
- Prozess
  - Einhaltung des Prozesszeitplans
  - Einhaltung der Kosten
  - Einhaltung der Standards (Lebenszyklus, Coding Rules, etc.)

# Kleine Aufgabe

## Qualitätskriterium Wartbarkeit

- Welche Eigenschaften müssen Design, Code und Test jeweils haben um wartbar zu sein? Wie prüft man diese Eigenschaften?
- Warum ist Nachvollziehbarkeit von Anforderugnen ein wichtiges Kriterium für die Wartbarkeit in allen Phasen der Entwicklung?

# SW QA Methoden

- Allgemeine Methoden:
  - Reviews
  - Inspektionen
  - Audits
  - Analysen: statisch, dynamisch
- spezielle Methoden
  - SW Metriken: zB über Fehler beim Test, Traceability
  - Sicherheitsanalysen: HAZOP (Hazard and Operability Analysis), FTA (Fault Tree Analysis), FMEA (Failure Modes and Effects Analysis), HSIA (Hardware Software Interaction Analysis)
  - Code Inspektionen
- dynamische Methoden: Simulation und Test
- Achtung: Testen ist i.A. nicht Aufgabe der QA, sondern von Entwicklern!
  Mindestens die Prüfung der Testpläne und Testergebnisse ist aber Aufgabe der QA

# Kleine Übung

## Coding Rules

- Ein Entwickler, der ein guter Programmierer ist, erstellt Software, die üblicherweise nur eine geringe Anzahl von Fehlern enthält; er ignoriert dabei allerdings die nach Firmenstandard und Projektstandard vorgegebenen Qualitätsstandards und Coding Rules. Wie sollten die verantwortlichen Manager (Projektleiter und Abteilungsleiter) auf dieses Verhalten reagieren? Begründen Sie Ihre Entscheidung.
- Welche der Regeln aus dem SUN Java Coding Conventions sind für die Wartbarkeit des Codes relevant?

# SW Metriken

- Quantitative Prüfung Allgemein:

  *Measurement is the process by which numbers or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules.*

  *Fenton*

  Dabei:
  - Entität = Objekt oder Ereignis der realen Welt
  - Attribut = Eigenschaft oder Aspekt einer Entität
- Grundannahme: von der Quantität kann man auf die Qualität schließen

  *You can not control what you can not measure.*

  *De Marco*
- Produkt und Prozessmetriken
- Messbare Eigenschaften bei SW:
  - Größe - Komplexität - Kritikalität - Fehlerrate
- Messbare Eigenschaften des Prozesses (Fortschritt)
  - Mittelabfluss (Zeit, Geld)
  - Abdecken der Anforderungen
  - Erreichen von Planungszielen

# Definition von Messprogrammen - 1

- Verlangt nach zB ISO 9001:200, auch ECSS, CENELEC
- Was soll gemessen werden?
  - Was sind die Qualitätsziele, die geprüft werden sollen?
  - Welche Daten werden benötigt um Aussagen machen zu können

  $\Rightarrow$ GQIM = Goal - Question - Indicator - Metrik
  - Anmerkung: es ist nicht sinnvoll alles zu messen, was gemessen werden kann; Daumenregel: max 10 Metriken
  - Achtung: die Detaildefinition von Daten ist u.U. auch abhängig von Programmiersprache
- Wann soll gemessen werden?
  - In welchen Phasen der Entwicklung (für welche Daten)
  - Mit welcher Frequenz (einmal zum Ende der Phase, regelmässig)?

# Definition von Messprogrammen - 2

- Wer soll Daten erfassen?
  - Verantwortlicher und Ausführende
  - manuell oder automatisch - abhängig von SW Entwicklungsumgebung
- Wie sollen Daten ausgewertet werden?
  - Oft müssen Verhältnisse zwischen Daten hergestellt werden
  - Verlaufsdaten vs. Einzeldaten
  - Festlegen von Kriterien, die auf Probleme hinweisen (Schwellwerte, kritische Verläufe, auch im Vergleich)
- Wo und von wem werden die Daten archiviert?

Allgemeines Problem: Akzeptanz - Messprogamme können leicht als Kontrollmechanismus missbraucht werden!

# Prozessverbesserung - 1

- engl. Process Improvement
- Zyklus Messen - Analysieren - Ändern
  - Idee der kontinuierlichen Verbesserung (von Prozessen und damit Produktqualität)
  - Stichwort: Lessons Learned
  - Erfordert mindestens nachträgliche Bewertung der Projekte zum Identifizieren von Problemquellen
- Prozesscharakterisierung (Stufen)
  - Informal Process: kein Prozessmodell, Verwendung von geeignet scheinenden Methoden und Techniken (zB CM)
  - Managed Process: Definiertes Prozessmodell für den Entwicklungsprozess, inkl. Prozeduren, Scheduling von Aktivitäten, Beziehungen zwischen Prozeduren
  - Methodical Process: Verwendung von wohldefinierten Entwicklungsmethoden und Werkzeugen, auch Analyseprozesse
  - Improving Process: Prozesse und ihre Verbesserungsziele (mittel und langfristig), Zuteilung von Budget für Verbesserungsaktivitäten, explizite Prozeduren für die Einführung von Verbesserungen, ev. auch: quantitative Prozessmessung

# Prozessverbesserung - 2

- Prozessmetriken für Verbesserung:
  - Durchführungsdauer von (Teil-)Prozessen
  - Resourcen für die (Teil-)Prozesse (Zeit, Personal)
  - Überwachung prozessspezifischer Ereignisse (Fehler bei Code Inspektionen, Problemberichte aus Reviews, Anzahl geänderter Code-Zeilen für Anforderungsänderungen)

# Zusammenfassung QA

- Ziel: Sicherstellung der Qualität des Endproduktes
- Ansatz: Prüfung von Produkt, Zwischenprodukten und Prozess
- Methoden:
  - Review, Inspektion und Audits
  - Analysen - statisch und dynamisch
- Definition von Hausstandards, Prüfung der Einhaltung von Haus- und Projektstandards
- Software Metriken als quantitative Daten für Produktqualität und Prozessqualität - müssen an Projekte angepasst werden.
- Prozessverbesserung im Zusammenhang mit Reifemodellen wie ISO 9000 oder CMMI

# Aufgaben

## SW QA allgemein

- Diskutieren Sie, warum ein guter Software Prozess auch einen positiven Einfluss auf die Qualität des Endproduktes hat. Vergleichen Sie dazu den Softwareentwicklungsprozess mit Produktionsprozessen zB von Platinen.
- Diskutieren Sie die Gemeinsamkeiten und Unterschiede von Metriken mit dem Ziel Prozessverbesserung und Metriken, die zur Projektkontrolle verwendet werden. Sammeln Sie Beispiel für beide Varianten.

# Teil III

## Verifikation, Validation und Test

# Übersicht

- Einführung und Motivation
- Software Qualitätssicherung
- Software Verifikation, Validation und Test

# Qualitätssicherung

- Verifikation und Validation
  - Planung von VVT
  - Inspektionen
  - Automatisierte Statische Analysen
  - Formale Verifikation
  - Summary V&V

- Software Test
  - Testen - Übersicht
  - System-Tests
  - Komponenten-Tests
  - Testfall-Design
  - Test-Automatisierung
  - Summary SW Test

# Verifikation und Validation

Nach Boehm:

- Validation: Entwickeln wir das richtige Produkt?
  Prüfen ob die Software die Spezifikation erfüllt - unter Umständen auf unterschiedlichen Ebenen
- Verifikation: Entwickeln wir das Produkt richtig?
  Prüfen ob die Ergebnisse von Entwicklungsschritten korrekt gegen die jeweilige Eingabe sind.

Anmerkung: unterschiedliche Definition in unterschiedlichen Anwendungsbereichen zu berücksichtigen
Immer: Beides zusammen ist die Prüfung ob das Produkt seinen Zweck erfüllt

# Unterschiedliche Aspekte V&V

- Prüfen der Systemfunktion (Korrektheit, gegen Anforderungen)
  Aufwand und Umfang stehen i.a. in Relation zu der Kritikalität der Software für die Firma (z.B. Safety, aber auch Unternehmenskritikalität)
- Prüfen gegen Nutzererwartungen
  Sinkende Akzeptanz von Systemausfällen, Problemen bei der Verwendung führt zu intensiverer V&V
- Prüfen im Rahmen des Marktsegments
  Vergleich mit Konkurrenten (Preis, Nutzerfreundlichkeit, Reife)

# Methoden - allgemein

- statische Analysen
  ohne laufendes System, z.T. auch automatisiert
  - Fagan Inspektion (manuell)
  - Review (manuell)
  - statische Analysen mit Compiler, SW-Entwicklungsumgebung, Spezialwerkzeugen

    Coding Rule Checker, Metrikprogramme, Coverage, Requirementstracing
- dynamische Analysen
  erfordern Prototyp oder andere lauffähige Version der SW
  - meist: Testen
  - Unterschiedliche Testziele:
    Validation Testing: Prüfen gegen Anforderungen
    Defect Testing: Lokalisierung und Behebung von Fehlern (Debugging)
  - auch: Simulation
  - Regression-Testing: Wiederholung von Tests für veränderte Versionen
- statische und dynamische Analysen sollten sich ergänzen

# Planen von VVT Aktivitäten (1)

- Ziel: So wenig wie möglich, aber soviel wie nötig
- Motivation: Kosten und Zeit im Rahmen des Projektes
- Erfahrung: frühe Prüfungen und Planen von Testaktivitäten sparen Zeit und helfen Fehler so früh wie möglich zu finden

## V-Modell

Planen von Tests gleichzeitig mit den Entwicklungsstufen:

- Anforderungesspezifikation und Abnahmetests
- System Spezifikation und System Integrationstests
- System Architektur und Design und Subsystem-Integrationstests
- Module und Unit-Tests

# Planen von VVT Aktivitäten (2)

- Detailsplanung von Tests/Validation:
  - Festlegen von Testfällen, Analyse und Archivieren von Testergebnissen
  - Entscheidung über Testwerkzeuge
  - auch: Festlegen von Kriterien für das Beenden von Tests (oft Coverage bzgl Anforderungen oder Code)
- Planen von Verifikationsaktivitäten:
  - Relevanz von Reviews und Verifikationen die dazu fertig sein sollten
  - Zeitplan für Codeinspektionen
  - Festlegen von Design und Coding Rules; u.U. Auswahl von Werkzeugen
  - Auch: Festlegung von Tracing Kriterien um den Bezug zu den Anforderungen und/oder vorhergehenden Ebenen nachzuweisen

# Software Inspektionen

- statische Analyse mit dem Ziel Probleme in der SW zu finden
  - Fehler
  - Auslassungen
  - Anomalien
- Meist Source Code, aber auch Modelle oder Anforderungen
  auch: Prototypen
- Vorteile gegen Testen:
  - Beim Testen können weitere Fehler durch gefundene versteckt werden
  - Auch unvollständige - sogar nicht lauffähige - Versionen können inspiziert werden ohne dass zusätzliche Kosten entstehen
  - Inspektionen können einen größeren Umfang an Kriterien (Qualitätskriterien) berücksichtigen

    Bsp: Konformität mit Standards, Wartbarkeit, Portierbarkeit, schlechter Programmierstil

    Bsp aus der funktionalen Sicht: Ineffiziente Algorithmen, ungeeignete Algorithmen
- Erfahrungen mit Inspektionen:
  - gute Ergebnisse - als Ergänzung zum Testen
  - geringe Akzeptanz - zusätzliche Kosten, Einarbeitungszeit

# Motivation für Automatisierung

- Einsparen von Kosten: Input für Inspektionssitzung
- Wiederholbarkeit - vorzugsweise bei jedem Build/jeder Version
- Explizites Aufzeigen von Problemen bei weitestgehendem Ausschluss von menschlichem Versagen
- Aber: Die Ergebnisse der Analysen müssen nachvollziehbar sein!
  - Lokalisierung der Probleme muss gewährleistet sein
  - relevante Outputs müssen von nicht-relevanten leicht getrennt werden können
  - Echte Fehler müssen zB. von Folgefehlern unterscheidbar sein
- ähnliche Überlegungen gelten auch für Testautomatisierung!
- Einige Aspekte sind schon in modernen Compilern enthalten.

# Werkzeuge

- Compiler - ev. spezielle Optionen
- Build-Tools - Ant, automake
- Entwicklungsumgebungen:
  - Together: konfigurierbaren Check für Coding Rules (Audit) und Metriken
  - Eclipse Plug-Ins (CheckStyle, Clover)
- LINT, LClint, SPlint: für C-Programme, mitgeliefert bei Linux;
- (Valgrind ! nicht statisch, sondern Analyse von Executables)

# Formale Verifikation (1)

- allgemein: Voraussetzung formale Spezifikation
  also: eindeutige Semantik, mathematisch/logische Fundierung von Spezifikation und Vergleich zwischen Spezifikationen
- zZ im Forschungs und Industriebereich: auf Basis von UML Modellen
  Aber: Nicht alle Werkzeuge verwenden die gleiche Semantik (Bedeutung) der Modelle, z.T. auch Abweichungen in der Syntax (Darstellung)
- Ministery of Defence, GB: Verwendung von formalen Spezifikationen und formaler Verifikation für sicherheitskritische Anwendungen gefordert
- CENELEC (Europäische Bahn): für höhere SIL (Safety Integrity Level) dringend empfohlen
- Problem: Methoden und Werkzeuge nur bedingt für Industrieeinsatz geeignet:
  - Spezialwissen mit hohem Trainingsaufwand
  - z.T. nicht gut skalierbar
  - z.Z. trotz einigen positiven Beispieln nur gerninge Akzeptanz
  - z.T. nur Spezialaspekte prüfbar

# Zusammenfassung V&V

- Verifikation = zeigen dass ein Produkt seine Spezifikation erfüllt
- Validation = zeigen dass das Produkt den Kunden zufriedenstellt
- Testen = dynamische Analyse
- Ergänzt durch statische Analyse zum Aufdecken von Fehlern
- Inspektionen auf Basis von Checklists als teuere aber zum Teil sehr effektive Ergänzung durch kleines Team nach festgelegtem Prozess
- Automatisierung von Statischen Analysen durch eine ganze Reihe von Werkzeugen möglich. Planung im Vorfeld notwendig
- Formale oder Cleanroom Entwicklung als Alternativen, aber mit wenig Erfahrungen für grosse Projekte

# Qualitätssicherung

- Verifikation und Validation
  - Planung von VVT
  - Inspektionen
  - Automatisierte Statische Analysen
  - Formale Verifikation
  - Summary V&V

- Software Test
  - Testen - Übersicht
  - System-Tests
  - Komponenten-Tests
  - Testfall-Design
  - Test-Automatisierung
  - Summary SW Test

# Testen im Überblick

- Systemtest: von unabhängigem Testteam (u.U. Integration und Test)
- Komponententest: durch Softwareentwickler
- Generelle Ziele:
  - Nachweis, dass die Software die Anforderungen erfüllt ⇒ Validierungstests
  - Aufdecken von Fehlern ⇒ Defect Tests
- Aktivitäten:
  - Testfälle entwicklen
  - Testdaten identifizieren und vorbereiten
  - Tests durchführen (lassen)
  - Testergebnisse auswerten (im Vergleich zu den Testfällen)
- Anmerkung: Erschöpfendes Testen ist i.A. nicht möglich!
- Planen (PM und QA) - Zeitplan und Verantwortlichkeiten

# System-Tests

- während Integration, Planung empfohlen parallel zur Entwicklung
- basierend auf System/SW Spezifikation
- eigenes Team unterschiedliche Aufgabenzuteilung - abhängig zT von Firmenkultur
  - VVT: Team für Verifikation, Validation, Test
  - SIVQ: Software Integration, Verification, Qualification (EADS)
  - Testfactory: Testen losgelöst von anderen Aktivitäten
- Aspekte (beide notwendig)
  - Integrationstests
    Sicherstellen, dass Komponenten bei der Integration vernünftig zusammenarbeiten
    System ist i.a. nicht vollständig, Testfälle aus abgeleiteten Anforderungen, oft: Stubs
  - Release oder Versions Tests
    Validieren des Gesamtsystems gegen Anforderungen vor Auslieferung

  ! keine strikte Trennung

# Integrations-Tests

- Test der Interaktion von Komponenten/Subsystemen
- Annahme: einzelne Komponenten sind (weitgehend) funktional korrekt
- Probleme: Lokalisieren von Fehlern
- Schlechter Ansatz: Big Bang Integration (*alles zusammenbauen und dann gleich alles testen*)
- Guter Ansatz: inkrementeller Ansatz für Integration und Test
- Integrationsvarianten:
  - top-down integration: Integration auf Basis der Funktionseinheiten (hierarchisches Vorgehen angelehnt an Architektur)
  - bottom-up integration: Integration durch schrittweises Ergänzen des Systems (starten mit Basisdiensten)
- Planung der Integration:
  - Reihenfolge - Daumenregel: häufig verwendete Funktionalität zuerst
  - Vorteil: relevante Komponenten werden am häufigsten getestet (speziell bei bottom-up)
- Achtung: Fehlerbehebung kann u.U. mehrere Komponenten betreffen, auch solche die noch nicht integriert sind
- Regressiontests nach Fehlerbehebung: Testwiederholung; prüfen ob Fehler behoben und/oder neue entstanden

# Versionstests

- Ziel: dem Kunden die Funkionsfähigkeit des Produktes demonstrieren / nachweisen
- Alternative Bezeichnungen: Funktionstests
- meist als black-box Test (s.u.)
- auch: Usability Tests
- auch: Defect Testen - spezielle Auswahl von Testdaten
  - Eingaben, die das System zwingen alle Fehlermeldungen zu erzeugen
  - Erzwingen von Pufferüberläufen
  - Gleiche Eingabefolgen mehrere Male um Nichtdeterminismus auszuloten
  - Versuch Unzulässige Ausgaben zu erzwingen
  - Versuch Überlauf/Unterlauf von Berechnungen zu erzwingen
- Ansätze:
  - Szenariobasiertes Testen
  - Use-Case-basiertes Testen
  - Modelbasiertes Testen (Sequenzdiagramme, Überdecken von Zustandsautomaten)

# Performanztests

- Zur Abdeckung nicht-funktionaler Anforderungen auf Systemebene
- Basis: Verwendungsprofil (Normale Nutzung, Hochlast)
- auch: Stress-Tests
  - Schrittweise an die Grenzen gehen (zB. Taktung, Datenmenge, Laufzeit)
  - Test des Fehlerverhaltens - Abgleich mit geforderten Einsatzbedingungen
  - Aufdecken von Fehlern im Belastungsfall
- Besonders empfohlen für verteilte Systeme
  entdecken von Bottlenecks und kritischen Schnittstellen

# Komponenten-Tests

- Verantwortung der Entwickler selbst; basiert auf intuitivem Verständnis
- Andere Bezeichnungen: unit Test, Modultest
- Ziel: Defect Tests - Fehler in den Komponenten finden
- Komponenten die getestet werden können
  - Einzelne Funktionen / Prozeduren / Methoden (besonders wenn komplexe Algorithmen darin implementiert)
  - Klassen mit Attributen und Methoden (als Objekt) - Test des Datentyps oder der Zustandsmaschine bis in die Details
    ! Test aller Operationen, Manipulation aller Attribute, alle Methoden in allen Zuständen - als Optimum
  - Zusammengesetzte Komponenten aus mehreren Objekten mit definierter Schnittstelle in die gemeinsame Umgebung
- Auswahl der Testfälle hängt von der Komponenten ab
  - Datenorientierte Tests (Äquivalenzklassenansatz)
  - Ereignisbasierte Tests (Basis: Event Traces)
  - Kontrollstrukturbasierte Tests

  $\Rightarrow$ siehe Testfalldesign

# Schnittstellentests (1)

- engl. Interface Testing
- vorrangig bei zusammengesetzten Komponenten: Testen des Verhaltens der Komponentenschnittstelle zur Umgebung; Interna oft als black-box
- Typen von Schnittstellen
  - Parameterschnittstelle: Übergabe von Daten und Funktionsreferenzen
  - Shared Memory: gemeinsame Nutzung, Sicherstellen der der Konsistenz
  - Prozedurschnittstellen: Sammlung von Prozeduren, die für andere Komponenten verwendbar sind
  - Message Passing: Anforderung von Services und Rücklieferung von Ergebnissen mittels Nachrichten

# Schnittstellentests (2)

- Verschiedene Klassen von Fehlern
  - Schnittstellenmissbrauch: Fehler in der Verwendung der Schnittstelle; z.B. Übergabe von Daten des falschen Typs
  - Missverständnisse an Schnittstellen: Falsche Interpretation der Schnittstellenspezifikation,
    zB. durch falsche Annahmen über das Verhalten der angesprochenen Komponente,
    resultiert in unerwartetem Verhalten
  - Verletzung von Zeitbedingungen
    zB. bei Echtzeitsystemen, Protokollen, Absprachen bei gemeinsamer Nutzung von Ressourcen
    Ergebnis: zu alte Daten, Daten überschrieben, Aktion abgebrochen nach time-out
- Design von Tests ist hier schwierig

# Schnittstellentests (3)

Empfehlungen für das Design von Schnittstellentests

- Teste jeden Aufruf an eine externe Komponente mit allen Extremwerten der Parameter (deckt Inkonsistenzen zu der Spezifikation auf)
- Teste Aufrufe mit Pointern immer auch mit NULL (typischer Fehler)
- Entwickle Tests, die für Prozedurschnittstellen die Fehler in der Komponente erzeugen (deckt Missverständnisse über Fehlerannahmen auf)
- Stresstests für Message Passing: hohe Last an Nachrichten (deckt Zeitprobleme auf, eventuell auch Pufferüberläufe o.ä.)
- Entwickle Tests die bei dem Zugriff auf shared memory die Reihenfolge der Zugriffe variieren. (Deckt Fehler auf, die im weitesten Racing Conditions sind)

Alternative für viele Schnittstellentests: Statische Analysen (LINT, VALGRIND)

# Testfalldesign: Ansätze allgemein

- Ziel: Auswahl geeigneter Eingaben / Ereignisfolgen, die bestimmte Verhalten des Systems / der Komponente erzwingen
- Ansätze:
  - Anforderungsbasiert (überwiegend Systemtests)
  - Partitionstests - Tests von Vertretern einer identifizierten Gruppe von Eingaben / Ereignissen, die sich unterschiedlich verhalten. Bekannteste Variante: Äquivalenzklassentests
  - Strukturelle Tests: basiert auf der Struktur des Programms (meist: Komponententests); Zusammenhang zu Testüberdeckungsforderungen (Pfadüberdeckung, Anweisungsüberdeckung)
- Empfehlung: erst abstraktere Testebenen dann konkretere d.h. Anfoderungen - Partitionen - Struktur: zur schrittweisen Verfeinerung der Testfälle
- Empfehlung: Entwurf von high-level Tests kann schon früh starten

# Basistestmethoden

- Black-Box-Verfahren
  keine Verwendung von Aufbau des Testlings;
  Point of Control/Point of Observation ausserhalb
  meist Test gegen Spezifikation
  - Äquivalenzklassenbildung
  - Grenzwertanalyse
  - Zustandsbezogener Test
- White-Box-Verfahren (Glass-Box)
  strukturelle Testverfahren (basierend auf Komponentenhierarchie, Kontrollfluss, Datenfluss)
  eher Modultests
  - Anweisungsüberdeckung
  - Zweigüberdeckung
  - Test der Bedingungen
  - Pfadüberdeckung
- Intuitive Testfallermittlung

# Black-Box: Äquivalenzklassenbildung (1)

- Menge der möglichen konkreten Eingabewerte wird in Äquivalenzklassen eingeteilt; je Klasse wird mindestens ein Repräsentant getestet
- Äquivalenzklasse: Menge von Werten die ein gleichartiges Sollverhalten zeigt (Eingabe- oder Ausgabewerte des Testobjekts)
- Systematisches Herleitung von Testfällen mit Äquivalenzklassen
  - ermittle für jede Eingabevariable den Definitionsbereich $\Rightarrow$ zulässige Werte; diese müssen gemäß der Spezifikation verarbeitet werden.
  - Verfeinerung der Äquivalenzklassen auf Basis der Spezifikation; Festlegen der erwarteten Ergebnisse und eventuell Vorbedingungen für den Testlauf
  - Auch: Zerlegung des Ausgabebereichs - aber hier ist die Ermittlung geeigneter Startdaten schwieriger
  - Speziell: unzulässige Eingaben und Ausgaben
  - Aussichtsreich: Grenzwerte der Äquivalenzklassen - decken oft Missverständnisse oder Ungenauigkeiten auf

# Black-Box: Äquivalenzklassenbildung (2)

Tipps für die Ermittlung von Äquivalenzklassen:

- Ermitteln von spezifizierten Einschränkungen und Bedingungen für Eingaben und Ausgaben aus der Spezifikation; für jede eine Äquivalenzklasse
- Zusammenhängender Wertebereich: ein gültiger, zwei ungültige Äquivalenzklassen
- Anzahl von Werten einzugeben: Klasse mit richtiger Anzahl richtiger Werte, je eine Klasse für Unter- und Überschreitung der gültigen Anzahl
- Menge von unterschiedlich zu behandelnden Werten führen zu je einer gültigen und einer ungültigen Äquivalenzklasse
- Situationen durch zwingend zu erfüllende Einschränkung / Bedingung beschrieben: je eine gültige und eine ungültige Äquivalenzklasse
- Zweifel an Gleichbehandlung der Werte: weitere Teilung der Äquivalenzklasse

# Black-Box: Äquivalenzklassenbildung (3)

Definition von Testfällen

- Äquivalenzklassen: Fälle für einzelne Eingabeparameter
- Testfälle als Kombination der jeweiligen Repräsentanten:
  - gültige Testfälle: Kombinationen von allen Äquivalenzklassen mit gültigen Werten
  - Negativtestfall: Kombination eines ungültigen Repräsentanten für einen Parameter mit gültigen Repräsentanten der anderen
- Explosion der gültigen Testfälle durch Kreuzprodukt - Regeln zur Reduzierung:
  - Testfälle nach Nutzerprofilen priorisieren
  - Testfälle bevorzugen, die Grenzwerte verwenden
  - Nur zwei Parameter permutieren (paarweise Kombination statt vollständiger Kombination)
  - Repräsentanten ungültiger Äquivalenzklassen nicht mit anderen Repräsentanten ungültiger Klassen kombinieren (Gefahr der Fehlermaskierung!)

# Black-Box: Äquivalenzklassenbildung (4)

Festlegen von Testkriterien

- Endekriterium - Wann ist genug getestet?
- zB. als Verhältnis von Anzahl durchgeführter Tests zu Gesamtzahl von Äquivalenzklassen:

  ÄK-Überdeckung = (Anzahl getesteter ÄK /Gesamtzahl ÄK) * 100%
- Achtung: Fehlen Äquivalenzklassen, so kann ein hoher Überdeckungsgrad irreführen!

Bewertung:

- Sorgfältige Bestimmung der Klassen liefert systematisch relevante Tests; unnütze Tests werden vermieden
- Keine Berücksichtigung von Abhängigkeiten und Wechselwirkungen zwischen Parametern
- In Kombination mit fehlerorientierten Verfahren wie der Grenzwertanalyse wirkungsvoll

# Black-Box: Grenzwertanalyse (1)

- Beobachtung: Fehler treten häufig an den Grenzbereichen der Äquivalenzklassen auf
- Nur möglich bei geordneten Datenmengen, deren Grenzen identifizierbar sind

  Beispiel für geordnete Mengen mit Grenzwerten:

  - INT mit MinInt und MaxInt
  - Integer aus [-5, 5]

  Beispiele für Mengen ohne Grenzwerte

  - reelle Zahlen aus (-5, 5)
  - {ROT, GELB, GRÜN}

- Testfälle:
  - Grenzwerte (gültig und ungültig) zB. der Äquivalenzklassen

# Black-Box: Grenzwertanalyse (2)

Tipps zur Testfallerstellung

- Datenbereich (Eingabe und Ausgabe): Grenzwerte und benachbarte Werte ausserhalb des Bereichs
- Grenzwerte bei Anzahl (Eingabe / Ausgabe): Min, Max, Min - 1, Max + 1
- geordnete Mengen: erstes und letztes Element
- komplexe Datenstrukturen: leere Liste, Nullmatrix, etc.
- Numerische Berechnungen: eng zusammenliegende Werte und weit auseinanderliegende Werte
- ungültige Äquivalenzklassen: nur sinnvoll, wenn Ausnahmebehandlung mit Grenzen assoziiert.

Festlegen der Testendekriterien:

- Analog zu Äquivalenzklassenüberdeckung:

  GW-Überdeckung = (Anzahl getesteter GW /Gesamtzahl GW) * 100%

Bewertung:

- In Verbindung mit Äquivalenzklassen sinnvoll
- erfordert hohe Kreativität, speziell bei komplexen Daten

# Black-Box: Zustandsbasierte Tests (1)

- Basierend auf Zustandsmodellen (Automaten) für die Historie des Testlings
- Startzustand - Folgezustände - Zustandsübergänge
- Verhalten hängt ab vom Zustand $\Rightarrow$ testen des Verhaltens in den jeweiligen Zuständen, die gezielt angesteuert werden müssen.

  Stack mit 3 Einträgen

  Zustände: leer, gefüllt, voll

  Operationen: push, pop, top, delete Transitionen: leer nach gefüllt mit push; etc. Testfälle = ?

- Testkriterien: Basierend auf Übergangsbaum
  - Wurzelknoten: Anfangszustand
  - Berechne Knoten als Folgezustände, Kanten markieren mit Übergangsfunktion
  - Zweig terminiert wenn Knoten auf dem Weg von der Wurzel schon vorhanden oder zugehöriger Zustand ein Endzustand ist (inkl. Fehlerzustände!)

# Black-Box: Zustandsbasierte Tests (2)

Testfälle definieren:

- Ein Testfall ist definiert durch Anfangszustand, Eingabenfolge, erwartete Ausgaben/Aktionen, Endzustand
- Für jeden Zustandsübergang: Vorzustand, auslösendes Ereignis, Reaktion, Folgezustand
- Achtung: Es ist nicht immer offensichtlich in welchen Zuständen sich der Testling befindet! Eventuell ist dafür eine begrenzte Sicht auf interne Variablen nötig.
- Bei Spezifikation schon Testbarkeit prüfen: hohe Anzahl von Zuständen und / oder Übergängen ist problematisch - eventuell Vereinfachung vorschlagen
- Komplexe Zustandsdefinitionen sind problematisch - auch hierauf bei der Spezifikation achten

# Black-Box: Zustandsbasierte Tests (3)

Testendekriterien - mögliche Varianten:

- jeder Zustand mindestens einmal erreicht.
- jeder Zustandsübergang mindestens einmal durchgeführt.
- alle spezifikationsverletzenden Zustandsübergänge geprüft.
- Erweitert:
  - Alle Kombinationen von Zustandsübergängen
  - alle Kombinationen von Zustandsübergängen in jeder beliebigen Reihenfolge, auch mehrfach hintereinander

  $\longrightarrow$ Langzeittests, $\longrightarrow$ Modellchecking

Bewertung:

- relevant wo Zustände eine Rolle spielen
- Auch: OO Klassentests - Zustände entsprechen Kombinationen von Attributwerten einer Klasse bzw ihrer Instanz

# Black-Box-Tests

Allgemeine Bewertung

- Weitere Blackbox-Verfahren:
  - Ursache-Wirkung-Graph-Analyse (Berücksichtigt Abhängigkeiten)
  - Syntaxtest - bei formaler Spezifikation der Syntax zB von Eingabemasken
  - Zufallstest
  - Smoke-Test = Ausprobieren des Testobjektes
- Vorteile Black-Box-Tests
  - Prüfung der Funktionalität unabhängig von Implementierung
- Nachteile Black-Box-Tests
  - Fehlerhafte Spezifikationen werden nicht erkannt
  - Nicht geforderte Funktionalität wird nicht erkannt (auch: toter Code)
- Anwendung bei Abnahmetests, Schnittstellentests - überall dort wo die äussere Funktion relevant ist

# White-Box: Amweisungsüberdeckung (1)

- Testfälle so wählen, dass alle oder eine festgelegte Quote der Anweisungen mindestens einmal ausgeführt werden (C0-Maß)
  Dh. jeder Knoten sollte einmal besucht werden
- Basiert auf Kontrollflussgraph (Darstellung der Kontrollstruktur einer Komponente als Graph mit gerichteten Kanten; Elemente: Sequenz, Verzweigung, Schleifen, Basisanweisungen)
- Testdurchlauf beinhaltet Nachweis, welche Anweisungen ausgeführt wurden.
- Unter Umständen genügt ein Testfall
- Testendekriterium: Quote (analog zu GW, ÄK)
- Bewertung:
  - schwaches Kriterium - muss nicht alle Fälle abdecken
  - identifizierten potentiell unerreichbaren Code
  - erkennt u.U. leere IF-Zweige nicht

# White-Box: Zweigüberdeckung

- Testfälle so wählen, dass alle oder eine festgelegte Quote der Bedingungen mindestens einmal ausgeführt werden (C1 - Maß)
- Basiert auf Kontrollflussgraph
  - bei Bedingungen die Auswertung zu TRUE und zu FALSE
  - alle CASE-Fälle
  - Schleifen: ohne Durchlauf, Durchlauf und Wiederholung
- Testdurchlauf beinhaltet
  - Entscheidung über weiteren Testverlauf basierend auf der Auswertung einer Bedingung
  - Nachweis, welche Bedingungen und Anweisungen ausgeführt wurden (Zweige)
  - Berücksichtigung von leeren IF-Zweigen ergibt sich aus Berücksichtigung der Bedingungen
- Testendekriterien über Quote, empfohlen: 100 %
- Bewertung:
  - mehr Testfälle, da alle Möglichkeiten der Bedingungsauswertung zu berücksichtigen sind
  - Festlegen der erwarteten Ergebnisse analog zu Äquivalenzklassentest
  - Unzureichend für OO (nur innerhalb von Klassen möglich)

# White-Box: Bedingungsüberdeckung

- Testfälle so wählen, dass alle Möglichkeiten der Bedingungsauswertung berücksichtigt werden ausgeführt werden
- Einfache Bedingungsüberdeckung (Branch Condition Testing) Kriterium: jede atomare Teilbedingung wird mit TRUE und FALSE berücksichtigt
- Mehrfache Bedingungsüberdeckung (Branch Condition Combination Testing) Kriterium: jede Kombination der atomaren Bedingungswerte soll berücksichtigt werden
- Minimale Mehrfachbedingungsüberdeckung (Modified Branch Condition Decision Testing) Kriterium: nur solche Varianten von Kombinationen atomarer Bedingungswerte, die sich unterscheiden
- Testfälle: so auswählen, dass alle Bedingungen erreicht werden, u.U. ist dafür eine Rückwärtsanalyse notwendig
- Testendekriterien: basierend auf Quote
- Bewertung
  - Empfehlung: Minimale Bedingungsüberdeckung
  - Einfache Überdeckung: nicht ausreichend, Mehrfachüberdeckung i.A. zu aufwändig
  - u.U. aufwändige Auswahl von Testfällen

# White-Box: Pfadüberdeckung

- Testfälle so wählen, dass alle Pfade inkl. Schleifen / Wiederholungen ausgeführt werden
- Beinhaltet die Kombination aller Pfade durch jeweilige Basisblöcke des Programms; berücksichtigt Abhängigkeiten zwischen Zweigen
- Bewertung:
  - Kombination von anderen White-Box-Verfahren
  - Berücksichtigung aller möglichen Anzahlen von Schleifendurchläufen - i.a. nicht mehr 100 % möglich
  - u.U. sehr aufwändig

# White-Box-Tests

## Allgemeine Bewertung

- Weiter White-Box-Verfahren:
  - Linear Code Sequence and Jump (LCASJ): Folgen von Anweisungen im Vordergrund (Basisblöcke), Kombination aller Basisblöcke sind zu berücksichtigen
  - Datenbasierte Verfahren: basiert auf lesenden und schreibenden Zugriffen auf Variablen/Parameter, berücksichtigt Abhängigkeiten zwischen Variablen/Parameter; Unterscheidung zwischen Verwendung der Variablen/Parameter in und ausserhalb von Bedingungen
- Grundprinzip: Basis ist der Programmtext - dessen Komplexität bestimmt den Aufwand
- Vorteile
  - geeignet für untere Testebenen (Modul, Komponente)
  - guter Support durch Werkzeuge für Testfallgenerierung und Testreporting
- Nachteile
  - nicht umgesetzte Anteile der Spezifikation werden u.U. nicht erkannt
  - instrumentierter Code (zB um Überdeckung zu messen) verhält sich unter Umständen anders als der Code alleine (speziell: Zeitverhalten)
- Empfehlung: Minimum Zweigüberdeckung

# Intuitive Testfallermittlung

- Ergänzung zur systematischen Testfallermittlung (black box, white box)
- Ansatz: intuitives Verständnis des Testers nutzen
  - zB. um Fehler zu finden
  - z.B. um noch unerreichbare Zweige/Pfade zu erreichen
- Dokumentation von Vor- und Nachbedingungen, erwartete Ausgaben, erwartetes Verhalten ist auch hier notwendig
- Empfehlung: nicht als einzigen Testansatz, nicht als erste Testkampagne
- Testendekriterien: können hier nicht formal festgelegt werden.

# Testautomatisierung und Werkzeuge

- Gut beherrscht auf Unit-Ebene
  - Eclipse Plugins
  - JUnit - CPPUnit - NUnit
- Skriptbasiert für Systemtests
  - SQS-Test/Professional
  - Imbus Testbench
  - Daimler CTE Tool für Äquivalenzklassenbildung
- GUI-Testen mit Capture-Replay
  - zB in Together
  - Abbot Java GUI Test Framework `http://abbot.sourceforge.net/`
- neuere Entwicklungen:
  - HASTE (Java) `http://atomicobject.com/haste/`
  - Systir (SOAP, graphical or web controlled applications) : `http://atomicobject.com/systir.page`
- Spezifikationsbasiertes Testen
- Modellbasiertes Testen
  - auch: Test-Driven Development

# Zusammenfassung Software Testen

- Unterscheidung der Testebene ist essentiell
- Frühe Testplanung und frühes Test-Design
  - speziell bei iterativer Entwicklung: Zwischenergebnisse müssen testbar bleiben!
  - Aufwand von Mock-Ups und Stubs nicht unterschätzen
- Automatisierung vor allem für Unit Tests innerhalb moderner Entwicklungsumgebungen