Willkommen zur Vorlesung
Softwarekonstruktion
im Wintersemester 2012 / 2013

Prof. Dr. Jan Jürjens

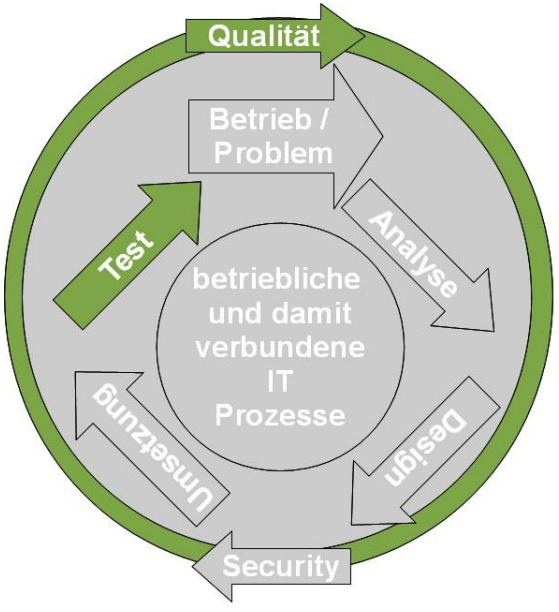
TU Dortmund, Fakultät Informatik, Lehrstuhl XIV

2.2 Testen im Softwarelebenszyklus

Basierend auf dem Foliensatz
„Basiswissen Softwaretest - Certified Tester“ des „German Testing Board“
(nach Certified Tester Foundation Level Syllabus, deutschsprachige Ausgabe, Version 2011) (mit freundlicher Genehmigung)

Einordnung
Testen im Softwarelebenszyklus

- Qualitätsmanagement
- Testen
  - Einführung
  - Grundlagen Softwaretesten
  - Testen im Softwarelebenszyklus
  - Statischer Test
  - Black-Box-Test
  - White-Box-Test
  - Test-Management
  - Testwerkzeuge
- Modellgetriebene SW-Entwicklung



2.2 Testen im Software-Lebenszyklus

In diesem Abschnitt beschäftigen wir uns mit dem Testen im Softwarelebenszyklus, insbesondere den folgenden Inhalten:

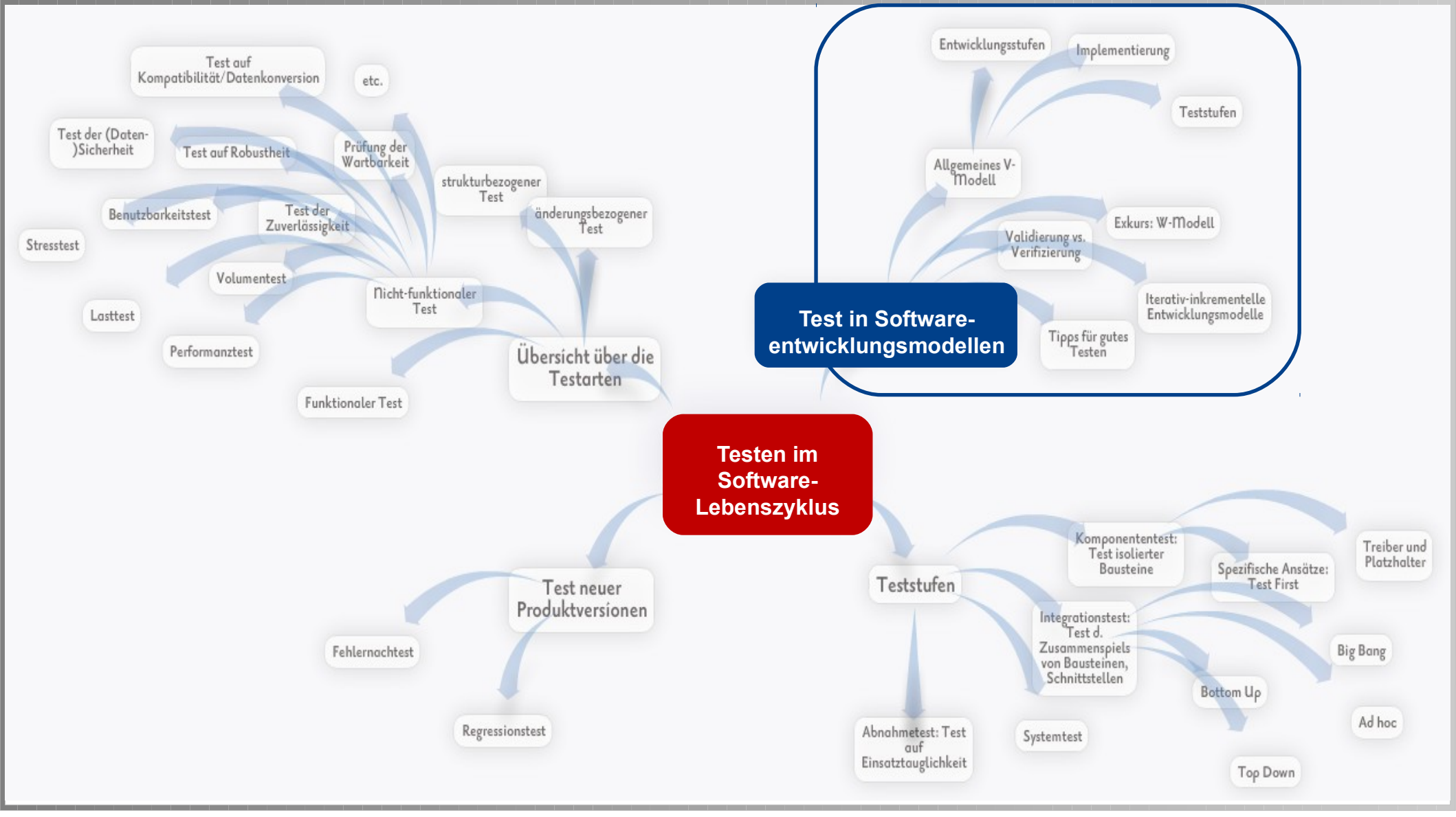
- Zusammenhang zwischen Entwicklungs- und Testaktivitäten,
- Komponenten-, Integrations-, System- und Abnahmetest,
- Testen von neuen Produktversionen,
- Wartungstests,
- Regressionstests und Auswirkungsanalysen.

2.2 Testen im Software-Lebenszyklus

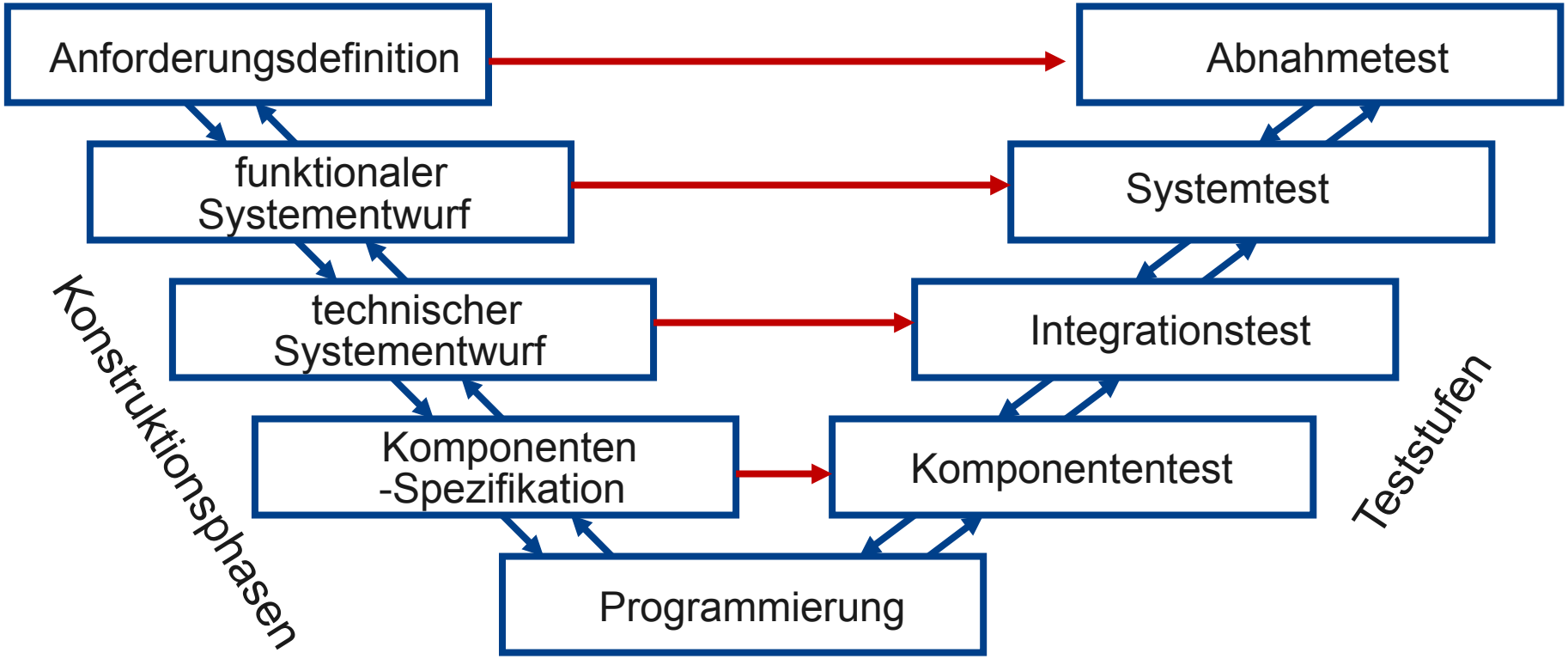
2.2 Testen im Software-Lebenszyklus

- **Testen in Softwareentwicklungsmodellen**
- Teststufen (Komponenten-/Integrations-/System-/Abnahmetest)
- Test neuer Produktversionen
- Übersicht über die Testarten

Big Picture


Testen im Software-Lebenszyklus
Test in Software-entwicklungsmodellen
Entwicklungsstufen
Implementierung
Teststufen
Allgemeines V-Modell
Exkurs: W-Modell
Validierung vs. Verifizierung
Iterativ-inkrementelle Entwicklungsmodelle
Tipps für gutes Testen
Übersicht über die Testarten
Test auf Kompatibilität/Datenkonversion
etc.
Test der (Daten-)Sicherheit
Test auf Robustheit
Prüfung der Wartbarkeit
strukturbezogener Test
Benutzbarkeitstest
Test der Zuverlässigkeit
änderungsbezogener Test
Stresstest
Volumentest
Nicht-funktionaler Test
Lasttest
Performanztest
Funktionaler Test
Test neuer Produktversionen
Fehlernachtest
Regressionstest
Teststufen
Komponententest: Test isolierter Bausteine
Treiber und Platzhalter
Spezifische Ansätze: Test First
Integrationstest: Test d. Zusammenspiels von Bausteinen, Schnittstellen
Big Bang
Bottom Up
Ad hoc
Abnahmetest: Test auf Einsatztauglichkeit
Systemtest
Top Down

Allgemeines V-Modell (sequentielles Entwicklungsmodell)

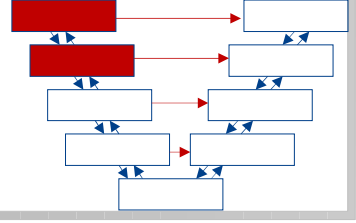


Legende
Testfälle basieren auf den entsprechenden Dokumenten

Allgemeines V-Modell - Konstruktionsphasen

Die konstruktiven Aktivitäten im linken Ast sind die Konstruktionsphasen; das Softwaresystem wird zunehmend detaillierter beschrieben.

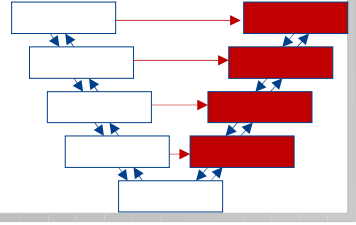
- **Anforderungsdefinition:** Die Wünsche und Anforderung des Auftraggebers oder des späteren Systemanwenders werden gesammelt, spezifiziert und verabschiedet. Zweck und gewünschte Leistungsmerkmale des zu erstellenden Softwaresystems liegen damit fest.
- **Funktionaler Systementwurf:** Die Anforderungen werden auf Funktionen und Dialogabläufe des neuen Systems abgebildet.
- **Technischer Systementwurf:** Die technische Realisierung des Systems wird entworfen. Hierzu gehören u. a.:
  - Definition der Schnittstellen zur Systemumwelt
  - Zerlegung des Systems in überschaubarere Teilsysteme (Systemarchitektur), die möglichst unabhängig voneinander entwickelt werden können.
- **Komponentenspezifikation:** Für jedes Teilsystem werden Aufgabe, Verhalten, innerer Aufbau und Schnittstelle zu anderen Teilsystemen definiert.
- **Programmierung:** (Implementierung) jedes spezifizierten Bausteins (Modul, Unit, Klasse o. Ä.) in einer Programmiersprache.



Allgemeines V-Modell - Teststufen

Da Fehler am einfachsten auf derselben Abstraktionsstufe gefunden werden, auf der sie entstanden sind, ordnet der rechte Ast nun jedem Spezifikations- bzw. Konstruktionsschritt eine korrespondierende Teststufe zu:

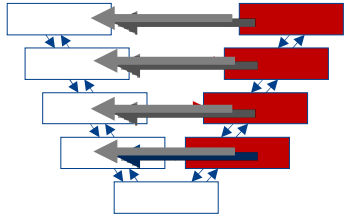
- **Komponententest**
  prüft, ob jeder einzelne Softwarebaustein (Komponente) für sich die Vorgaben seiner Spezifikation erfüllt.
- **Integrationstest**
  prüft, ob Gruppen von Komponenten wie im technischen Systementwurf vorgesehen zusammen spielen.
- **Systemtest**
  prüft, ob das System als Ganzes die spezifizierten Anforderungen erfüllt.
- **Abnahmetest**
  prüft, ob das System aus Kundensicht die vertraglich vereinbarten Leistungsmerkmale aufweist.



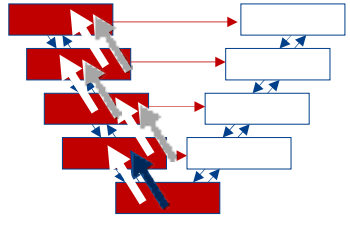
Validierung vs. Verifikation

In jeder Teststufe ist also zu überprüfen, ob die Entwicklungsergebnisse diejenigen Anforderungen erfüllen, die auf der jeweiligen Abstraktionsstufe relevant bzw. spezifiziert sind.

- Dieses Prüfen der Entwicklungsergebnisse gegen die ursprünglichen Anforderungen wird _____ genannt.
  - Beim _____ bewertet der Tester, ob ein (Teil-)Produkt eine festgelegte (spezifizierte) Aufgabe tatsächlich löst und deshalb für seinen Einsatzzweck tauglich bzw. nützlich ist.
  - Untersucht wird, ob das Produkt im Kontext der beabsichtigten Produktnutzung sinnvoll ist.



- Neben _____ Prüfungen fordert das V-Modell auch sogenannte _____ Prüfungen.
  - _____ ist im Gegensatz zur _____ auf eine einzelne Entwicklungsphase bezogen und soll die Korrektheit und Vollständigkeit eines Phasenergebnisses relativ zu seiner direkten Spezifikation (Phaseneingangsdokumente) nachweisen.
  - Untersucht wird, ob die Spezifikationen korrekt umgesetzt wurden, unabhängig von einem beabsichtigten Zweck oder Nutzen des Produkts.

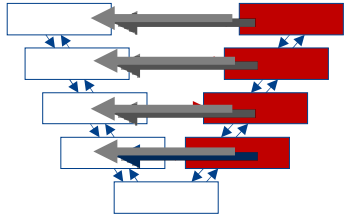


Anmerkung: In der Praxis beinhaltet jeder Test beide Aspekte, wobei der _____anteil mit steigender Teststufe zunimmt.

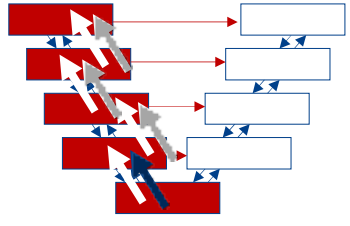
Validierung vs. Verifikation

In jeder Teststufe ist also zu überprüfen, ob die Entwicklungsergebnisse diejenigen Anforderungen erfüllen, die auf der jeweiligen Abstraktionsstufe relevant bzw. spezifiziert sind.

- Dieses Prüfen der Entwicklungsergebnisse gegen die ursprünglichen Anforderungen wird Validierung genannt.
  - Beim Validieren bewertet der Tester, ob ein (Teil-)Produkt eine festgelegte (spezifizierte) Aufgabe tatsächlich löst und deshalb für seinen Einsatzzweck tauglich bzw. nützlich ist.
  - Untersucht wird, ob das Produkt im Kontext der beabsichtigten Produktnutzung sinnvoll ist.



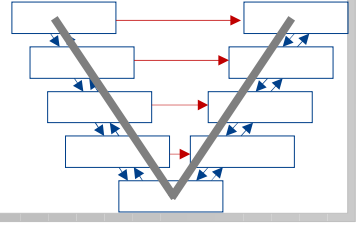
- Neben validierenden Prüfungen fordert das V-Modell auch sogenannte verifizierende Prüfungen.
  - Verifizierung ist im Gegensatz zur Validierung auf eine einzelne Entwicklungsphase bezogen und soll die Korrektheit und Vollständigkeit eines Phasenergebnisses relativ zu seiner direkten Spezifikation (Phaseneingangsdokumente) nachweisen.
  - Untersucht wird, ob die Spezifikationen korrekt umgesetzt wurden, unabhängig von einem beabsichtigten Zweck oder Nutzen des Produkts.



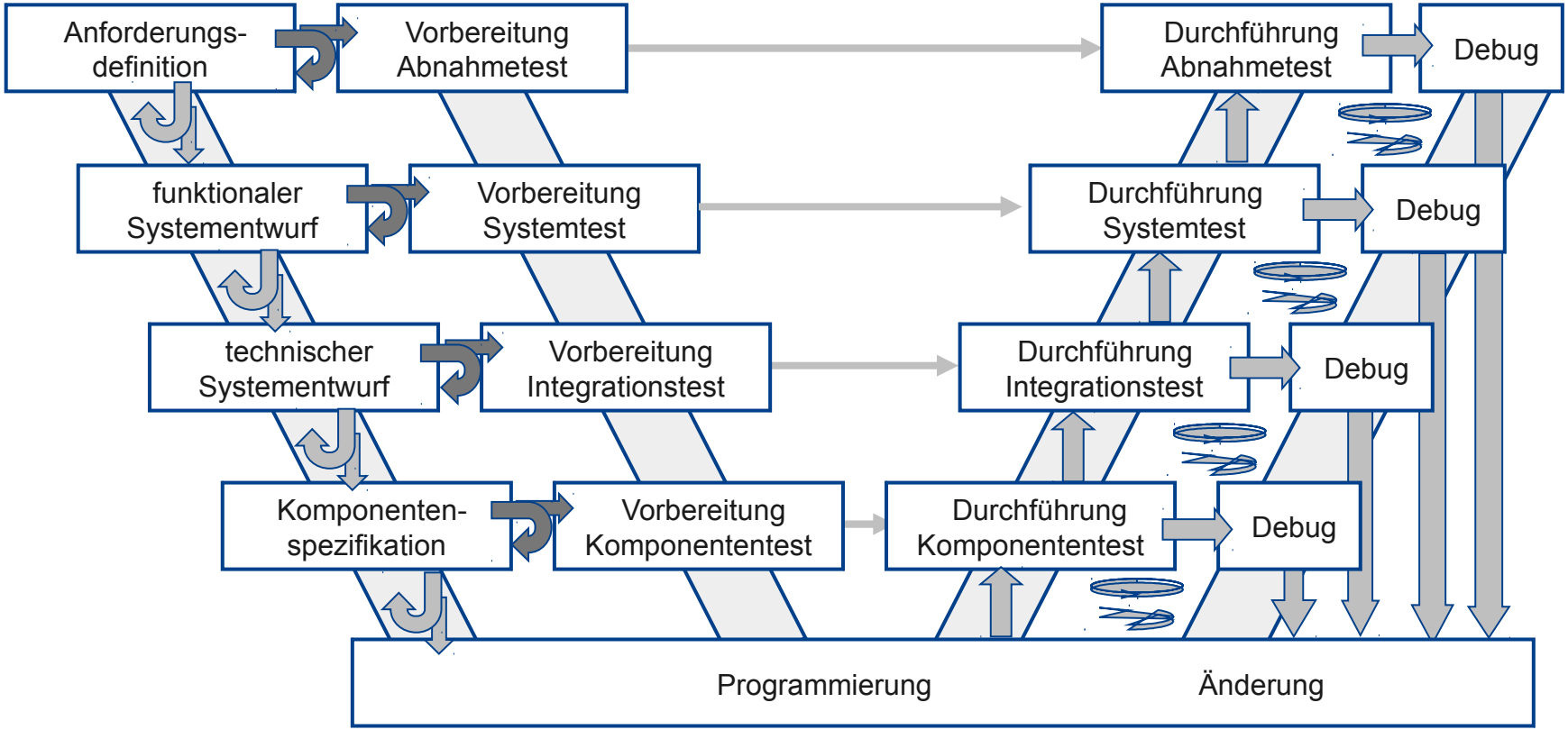
Anmerkung: In der Praxis beinhaltet jeder Test beide Aspekte, wobei der Validierungsanteil mit steigender Teststufe zunimmt.

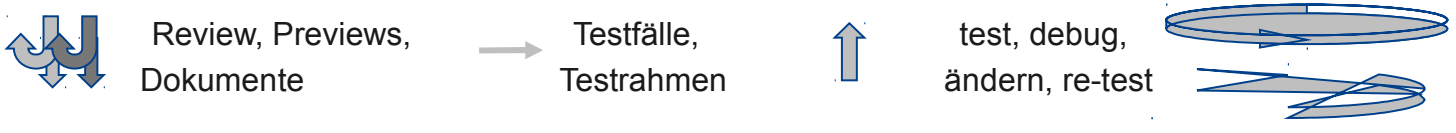
V-Modell

- Konstruktions- und Testaktivitäten sind getrennt, aber gleichwertig (linke Seite/rechte Seite).
- »V« veranschaulicht die Prüfaspekte Verifizierung und Validierung.
- Es werden arbeitsteilige Teststufen unterschieden, wobei jede Stufe »gegen« ihre korrespondierende Entwicklungsstufe testet.
- V-Modell erweckt den Eindruck, dass Testen erst relativ spät, nämlich nach der Implementierung beginnt. Dies ist falsch.
- Die Teststufen im rechten Ast des Modells sind als Phasen der Testdurchführung und -auswertung zu verstehen. Die zugehörige Testvorbereitung (Testplanung, Testspezifikation) startet früher und wird parallel zu den Entwicklungsschritten im linken Ast durchgeführt (siehe W-Modell, folgende Folie).

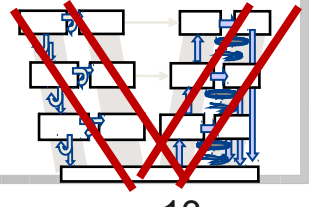


Exkurs: W-Modell - Weiterentwicklung des V-Modells



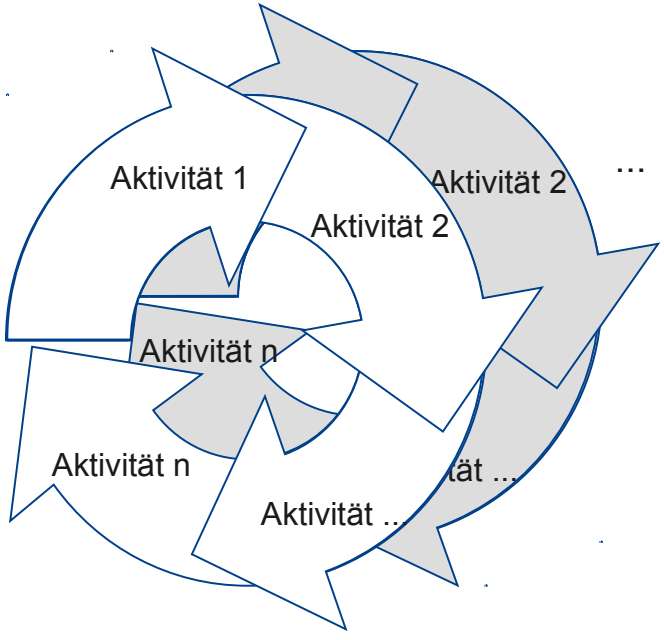


Spillner/Roßner/Winter/Linz: Praxiswissen Softwaretest -Testmanagement, dpunkt, 2006, Kap. 3.4



Iterativ-inkrementelle Entwicklungsmodelle

- Es werden kleine Entwicklungsschritte durchlaufen.
- Das System wird nicht »am Stück« erstellt, sondern in einer geplanten Abfolge von Versionsständen und Zwischenlieferungen (Inkrementen).
- Bei jeder Iteration werden Erweiterungen vorgenommen.
- Jede Iteration ergibt ein wachsendes (noch) unvollständiges System.



Testen in iterativ-inkrementellen Entwicklungsmodellen

- Testen ist dem Entwicklungsablauf anzupassen.
- Für jedes Inkrement (Zwischenlieferung) sind wiederverwendbare Tests vorzusehen.
- In jeder Iteration werden die vorhandenen Tests wiederverwendet.
- Für neue Funktionalität werden zusätzliche Tests benötigt.
- Für jedes Inkrement innerhalb einer Iteration können die verschiedenen Teststufen durchlaufen werden.
- Notwendig sind kontinuierliche Integrationstests und Regressionstests.
- Verifizierung und Validierung können für jedes Inkrement durchgeführt werden.

Beispiele: iterativ-inkrementelle Entwicklungsmodelle

- Prototyping
- Rapid Application Development (RAD)
- Rational Unified Process (RUP)
- agile Entwicklungsmodelle
  - Extreme Programming (XP),
  - Dynamic Systems Development Method (DSDM)
  - SCRUM
  - ...

RAD: Martin, J.: Rapid Application Development. Macmillan, 1991
RUP: Kruchten, P.: Der Rational Unified Process – Eine Einführung. Addison-Wesley-Longman, 1999.
XP: Beck, K.: Extreme Programming. Addison-Wesley, München, 2000
DSDM: Stapleton, J. (Hrsg.): DSDM: Business Focused Development (Agile Software Development Series). Addison-Wesley, 2002
SCRUM: Beedle, M.; Schwaber, K.: Agile Software Development with Scrum. Prentice Hall, 2001

Tipps für gutes Testen

Unabhängig vom gewählten Softwareentwicklungsmodell haben sich folgende Aspekte in Bezug auf das Testen als sinnvoll herausgestellt:

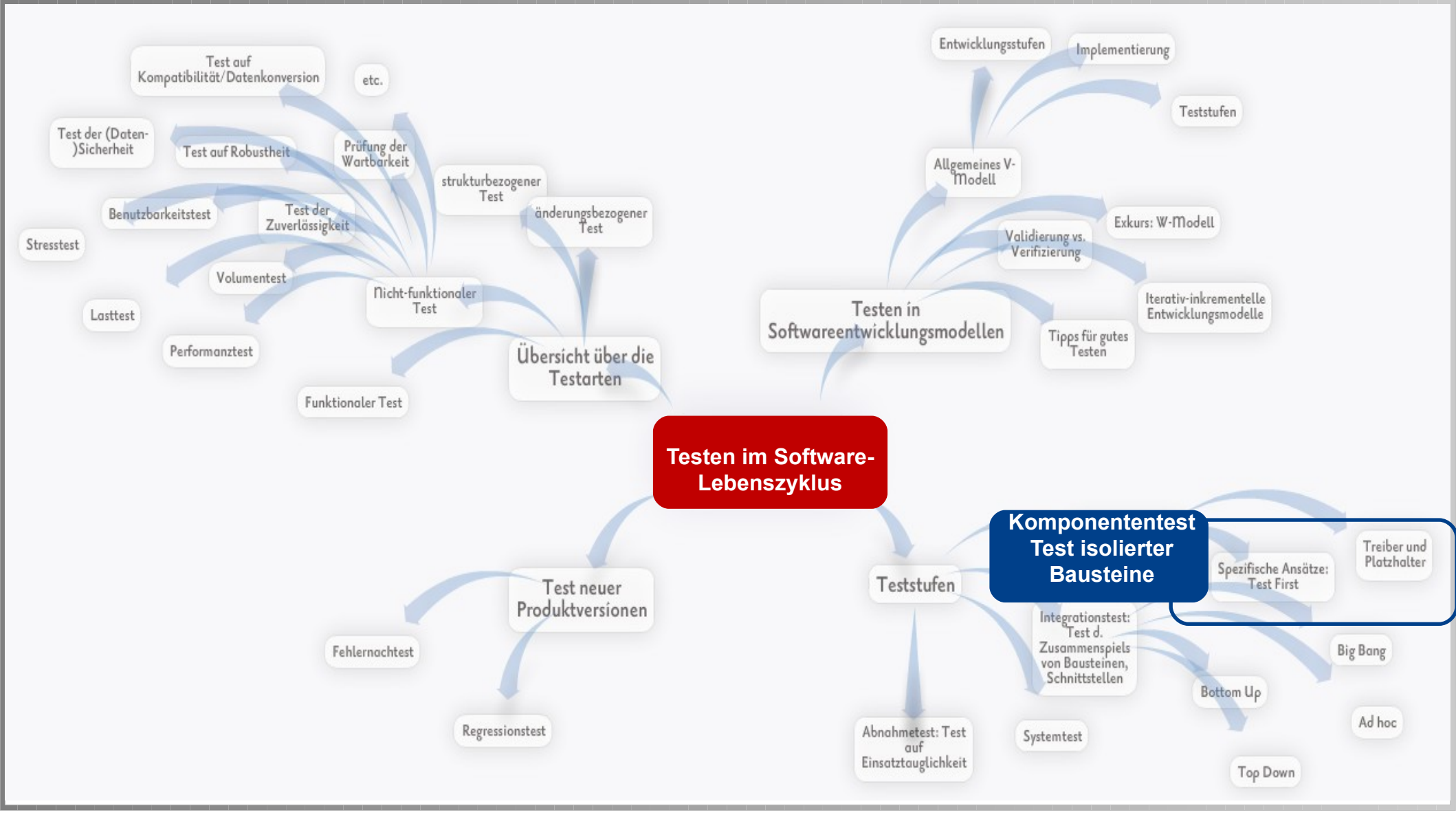
- Zu jeder Entwicklungsaktivität gibt es eine entsprechende Testaktivität.
- Testaktivitäten sollten früh im Entwicklungszyklus beginnen. Testanalyse und Testentwurf sollten parallel zur entsprechenden Entwicklungsstufe beginnen.
- Tester früh im Review-Prozess der Entwicklungsdokumente einbinden.
- Softwareentwicklungsmodelle sollten nicht „Out of the Box“ verwendet werden. Sie müssen an Projekt- und Produktcharakteristika angepasst werden (z.B. Anzahl der anzuwendenden Teststufen, die Anzahl und Länge der Iterationen, etc. müssen je Projektkontext angepasst werden).

2.2 Testen im Software-Lebenszyklus

2.2 Testen im Software-Lebenszyklus

- Testen in Softwareentwicklungsmodellen
- Teststufen → Komponententest
- Test neuer Produktversionen
- Übersicht über die Testarten

Big Picture


Testen im Software-Lebenszyklus
Übersicht über die Testarten
Nicht-funktionaler Test
Test auf Kompatibilität/Datenkonversion
etc.
Test der (Daten-)Sicherheit
Test auf Robustheit
Prüfung der Wartbarkeit
strukturbezogener Test
Benutzbarkeitstest
Test der Zuverlässigkeit
änderungsbezogener Test
Stresstest
Volumentest
Lasttest
Performanztest
Funktionaler Test
Testen in Softwareentwicklungsmodellen
Entwicklungsstufen
Implementierung
Teststufen
Allgemeines V-Modell
Exkurs: W-Modell
Validierung vs. Verifizierung
Iterativ-inkrementelle Entwicklungsmodelle
Tipps für gutes Testen
Test neuer Produktversionen
Fehlernachtest
Regressionstest
Teststufen
Komponententest
Test isolierter Bausteine
Spezifische Ansätze: Test First
Treiber und Platzhalter
Integrationstest: Test d. Zusammenspiels von Bausteinen, Schnittstellen
Big Bang
Bottom Up
Ad hoc
Top Down
Abnahmetest: Test auf Einsatztauglichkeit
Systemtest

Anmerkungen

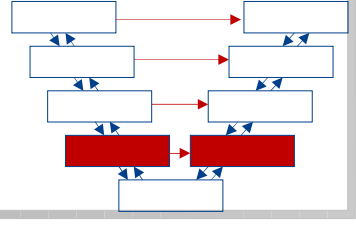
- Die einzelnen Teststufen Komponententest, Integrationstest, Systemtest und Abnahmetest werden am allgemeinen V-Modell erläutert.
- Die Stufen kommen aber auch analog in anderen Entwicklungsmodellen vor, wobei deren Bezeichnung unterschiedlich sein kann.
- Teststufen sind zu unterschieden, die mit dem Test von kleinen Einheiten beginnen und dann schrittweise das System integrieren, bis es vollständig ist und vom Kunden abgenommen wird.
- Weitere mögliche Stufen sind:
  - **Komponentenintegrationstest**
    wird nach dem Komponententest durchgeführt und testet das Zusammenspiel der Softwarekomponenten.
  - **Systemintegrationstest**
    wird nach dem Systemtest durchgeführt und testet das Zusammenspiel zwischen verschiedenen Systemen oder zwischen Hardware und Software.

Teststufen

- Unterscheidung verschiedener Teststufen ist mehr als nur eine zeitliche Unterteilung von Testaktivitäten.
- Unterschiede sind gegeben durch bzw. ergeben sich aus:
  - unterschiedliche Testobjekte
  - unterschiedliche Testbasis
  - unterschiedliche Teststrategie
  - Anwendung unterschiedlicher Testverfahren
  - Einsatz unterschiedlicher Testwerkzeuge
  - unterschiedliche Ziele und Arbeitsergebnisse
  - unterschiedliche Verantwortlichkeiten
  - unterschiedlich spezialisiertes Testpersonal
- Falls Konfigurationsdaten Teil des Systems sind, soll auch der Test dieser Daten in der Testplanung berücksichtigt werden.

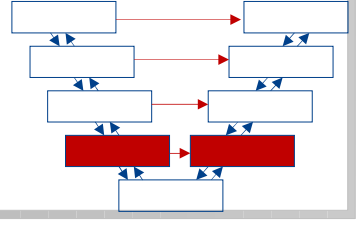
Komponententest - Begriffsklärung

- Im **Komponententest** (erste Teststufe) werden die erstellten Softwarebausteine unmittelbar nach der Programmierphase erstmalig einem systematischen Test unterzogen.
- Abhängig von der eingesetzten Programmiersprache werden diese kleinsten Softwareeinheiten unterschiedlich bezeichnet:
  - zum Beispiel als Module, Units oder Klassen (im Fall objektorientierter Programmierung).
  - Die entsprechenden Tests werden Modul-, Unit- bzw. Klassentest genannt.
- Von der verwendeten Programmiersprache abstrahiert, wird von Komponente oder Softwarebaustein gesprochen. Der Test eines solchen einzelnen Softwarebausteins wird als Komponententest bezeichnet.



Komponententest - Testobjekte

- Jeweils ein einzelner Softwarebaustein wird überprüft, isoliert von anderen Softwarebausteinen des Systems.
- Die Isolierung soll komponentenexterne Einflüsse beim Test ausschließen.
- Deckt der Test eine Fehlerwirkung auf, lässt sich deren Ursache klar der getesteten Komponente zuordnen.
- Die zu testende Komponente kann auch eine aus mehreren Bausteinen zusammengesetzte Einheit sein. Entscheidend ist, dass komponenteninterne Aspekte geprüft werden, jedoch nicht die Wechselwirkung mit Nachbarkomponenten.
- Testbasis beim Komponententest sind in erster Linie die Spezifikation der Komponente (im Design festzulegen) sowie deren Programmcode. Darüber hinaus alle anderen Dokumententeile, die sich auf die zu testende Komponente beziehen.



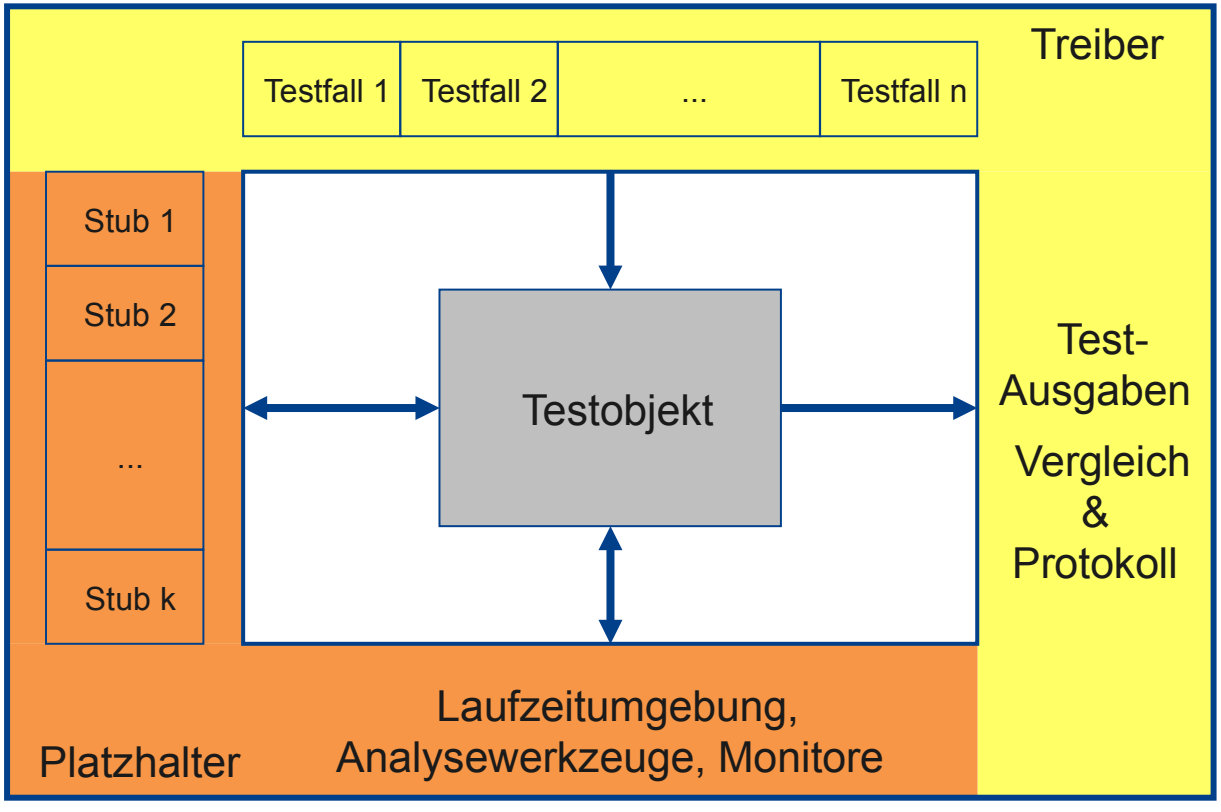
Komponententest - Testumgebung (1)

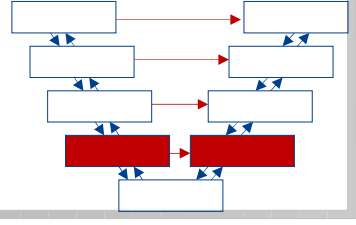
Testumgebung besteht aus

- **Treiber/Testtreiber (Driver)**
  Aufruf der Dienste des Testobjekts

  und/oder

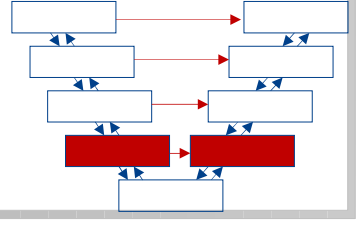
- **Platzhalter (Stubs, Dummy)**
  Simulation der Dienste, die das Testobjekt importiert





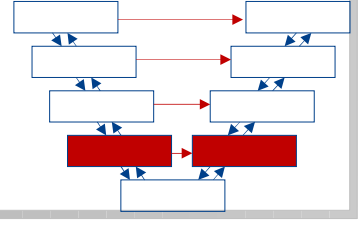
Komponententest - Testumgebung (2)

- In dieser Teststufe wird sehr entwicklungsnah gearbeitet.
- Zur Erstellung der Testumgebung ist Entwickler-Know-how notwendig.
- Der Programmcode des Testobjekts – zumindest der Code der Schnittstelle – muss verfügbar und verstanden sein, damit im Treiber der Aufruf des Testobjekts korrekt programmiert werden kann.
- Komponententests werden deshalb sehr oft von den Entwicklern selbst durchgeführt.
- Es wird dann vom Entwicklertest gesprochen, obwohl ein Komponententest gemeint ist.



Komponententest - Testziele (1)

- Wichtigste Aufgabe des Komponententests ist die Sicherstellung, dass das jeweilige Testobjekt die laut seiner Spezifikation geforderte **Funktionalität** korrekt und vollständig realisiert.
- Funktionalität ist dabei gleichbedeutend mit dem Ein-/Ausgabe-Verhalten des Testobjekts.
- Um Korrektheit und Vollständigkeit der Implementierung zu prüfen, wird die Komponente einer Reihe von Testfällen unterzogen, wobei jeder Testfall eine bestimmte Ein-/Ausgabe-Kombination (Teilfunktionalität) abdeckt.
- Typische Fehlerwirkungen, die beim funktionalen Komponententest aufgedeckt werden, sind Berechnungsfehler oder fehlende und falsch gewählte Programmpfade (z. B. vergessene Sonderfälle).
- **Test auf Robustheit:** Jede Softwarekomponente muss später beim Betrieb des Gesamtsystems mit einer Vielzahl von Nachbarkomponenten zusammenarbeiten und Daten austauschen. Dabei ist nicht auszuschließen, dass die Komponente unter Umständen auch falsch, d. h. entgegen ihrer Spezifikation angesprochen oder verwendet wird. In solchen Fällen sollte die falsch angesprochene Komponente nicht gleich den Dienst einstellen und das Gesamtsystem zum Absturz bringen. Vielmehr sollte sie die Fehlersituation abfangen und »vernünftig« bzw. robust reagieren.



Komponententest - Testziele (2)

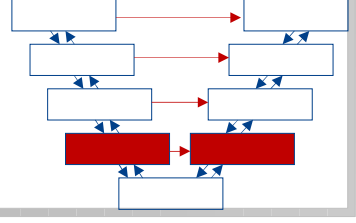
Neben Funktionalität und Robustheit sollten im Komponententest des Weiteren alle diejenigen Komponenteneigenschaften überprüft werden, die die Qualität der Komponente maßgebend beeinflussen und die in höheren Teststufen nicht mehr oder nur mit wesentlich höherem Aufwand geprüft werden können wie Effizienz und Wartbarkeit.

Effizienz: gibt an, wie wirtschaftlich die Komponente mit den verfügbaren Rechnerressourcen umgeht. Teilkriterien (z. B. Speicherverbrauch in Kilobyte, Antwortzeit in Millisekunden) können im Test exakt gemessen werden.

Wartbarkeit umfasst all diejenigen Eigenschaften eines Programms, die Einfluss darauf haben, wie leicht oder schwer es fällt, das Programm zu ändern oder weiterzuentwickeln. Entscheidend dabei ist, wie viel Aufwand es kostet, das vorhandene Programm und dessen Kontext zu verstehen. Folgende Prüfaspekte stehen im Vordergrund: Codestruktur, Modularität, Kommentierung des Codes, Verständlichkeit und Aktualität der Dokumentation usw.

Wartbarkeit lässt sich natürlich nicht durch dynamische Tests überprüfen.

- Notwendig sind Analysen der Programmtexte und Spezifikationen. Mittel hierzu ist der statische Test und insbesondere das (Code-) Review.
- Da Eigenschaften der einzelnen Komponente untersucht werden, werden solche Analysen aber zweckmäßigerweise im Rahmen des Komponententests durchgeführt.



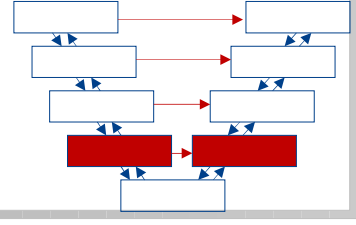
Komponententest - Teststrategie

- Der Tester hat in der Regel Zugang zum Programmcode, folglich kann der Komponententest als White-box Test (s. Abschnitt 2.5) durchgeführt werden.
- Der Tester kann Testfälle unter Ausnutzung seines Wissens über komponenteninterne Programmstrukturen, Methoden und Variablen entwerfen.
- Auch bei der Testdurchführung ist das Vorliegen des Programmcodes nützlich. Mit Werkzeugen können während des Testablaufs Programmvariablen beobachtet werden, um auf ein korrektes oder fehlerhaftes Verhalten der Komponente zu schließen.

In der Praxis wird aber in vielen Fällen auch der Komponententest »nur« als Black-box-Test (s. Abs. 2.4) durchgeführt (d. h., die innere Struktur zur Auswahl der Testfälle nicht herangezogen).

- Reale Softwaresysteme bestehen oft aus hunderten oder tausenden Komponenten. Ein Einstieg in den Code ist hier sicher nur bei ausgewählten Komponenten praktikabel.
- Zum anderen werden im Verlauf der Integration die elementaren Programmbausteine zu größeren Einheiten zusammengesetzt.

Oft gibt es auch in der ersten Teststufe nur zusammengesetzte Programmbausteine als testbare Einheiten. Diese Testobjekte sind schon zu groß, um mit vertretbarem Aufwand Beobachtungen oder Eingriffe auf Codeebene verstehen und vornehmen zu können. Ob beim Komponententest elementare oder bereits zusammen-gesetzte Programm-bausteine getestet werden, wird in der Integrations- und Testplanung festgelegt.

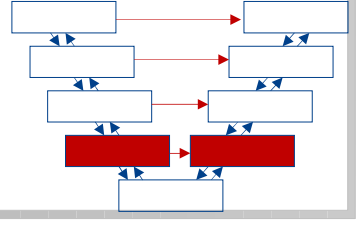


»Test-first«-Ansatz im Komponententest

Idee: Zuerst sind die Testfälle zu erstellen und die Testausführung zu automatisieren und danach sind die gewünschten Komponententeile zu programmieren.

Iterativer Ansatz: Programmtext wird mit Testfällen überprüft und solange verbessert, bis die Tests keine Fehlerwirkungen mehr aufzeigen.

Testgetriebene Entwicklung (test-driven development): Zyklen aus Testfallentwicklung, Programmierung und Integration von kleinen Programmstücken und der Ausführung von Komponententests, bis diese bestanden sind.

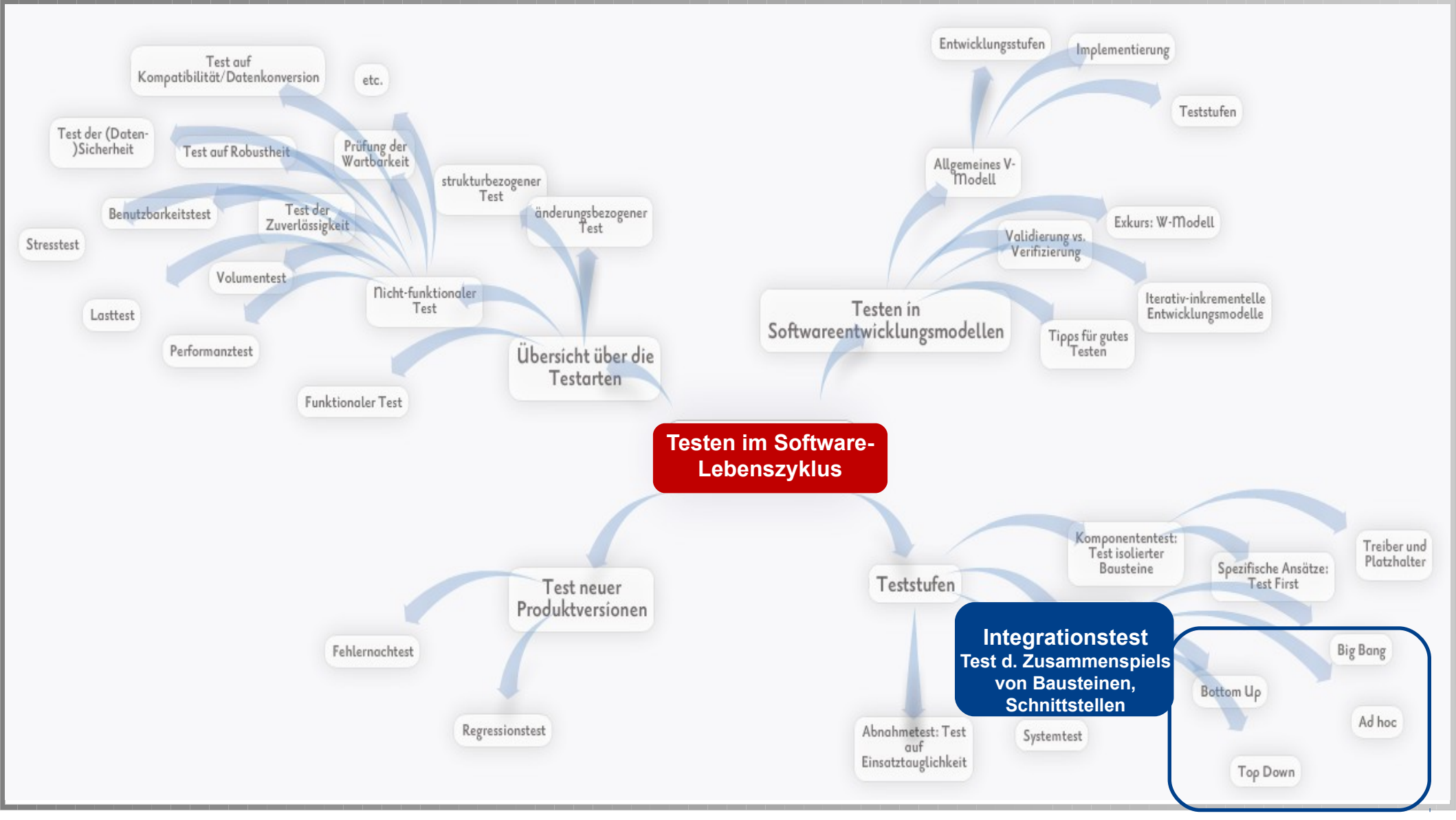


2.2 Testen im Software-Lebenszyklus

2.2 Testen im Software-Lebenszyklus

- Testen in Softwareentwicklungsmodellen
- Teststufen → Integrationstest
- Test neuer Produktversionen
- Übersicht über die Testarten

Big Picture


Testen im Software-Lebenszyklus
Übersicht über die Testarten
Nicht-funktionaler Test
Test auf Kompatibilität/Datenkonversion
etc.
Test der (Daten-)Sicherheit
Test auf Robustheit
Prüfung der Wartbarkeit
strukturbezogener Test
änderungsbezogener Test
Benutzbarkeitstest
Test der Zuverlässigkeit
Stresstest
Volumentest
Lasttest
Performanztest
Funktionaler Test
Testen in Softwareentwicklungsmodellen
Entwicklungsstufen
Implementierung
Teststufen
Allgemeines V-Modell
Exkurs: W-Modell
Validierung vs. Verifizierung
Iterativ-inkrementelle Entwicklungsmodelle
Tipps für gutes Testen
Test neuer Produktversionen
Fehlernachtest
Regressionstest
Teststufen
Komponententest: Test isolierter Bausteine
Spezifische Ansätze: Test First
Treiber und Platzhalter
Integrationstest
Test d. Zusammenspiels von Bausteinen, Schnittstellen
Big Bang
Bottom Up
Ad hoc
Top Down
Systemtest
Abnahmetest: Test auf Einsatztauglichkeit

Integrationstest - Begriffsklärung

Zweite Teststufe nach dem Komponententest.

Voraussetzung:

- Die übergebenen Testobjekte (d. h. einzelne Komponenten) sind bereits getestet und
- aufgezeigte Fehlerzustände möglichst korrigiert.

Integration:

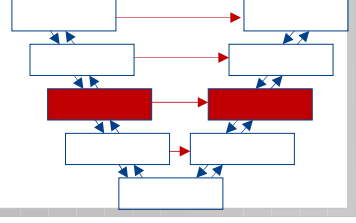
- Gruppen dieser Komponenten werden dann von Entwicklern, Testern oder speziellen Integrationsteams zu größeren Baugruppen bzw. Teilsystemen verbunden.

Integrationstest*:*

- Es muss getestet werden, ob das Zusammenspiel aller Einzelteile miteinander richtig funktioniert.

Ziel:

- Fehlerzustände in Schnittstellen und im Zusammenspiel zwischen integrierten Komponenten zu finden.

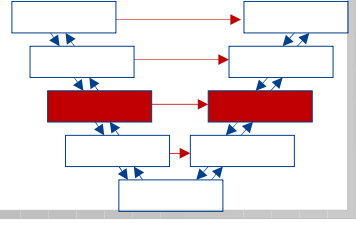


Integrationstest - Testobjekte

- Einzelbausteine werden schrittweise zu größeren Einheiten zusammengesetzt (Integration) und sollten einem Integrationstest unterzogen werden.
- Jedes entstandene Teilsystem kann anschließend Basis für die weitere Integration noch größerer Einheiten sein.
- Testobjekte des Integrationstests können also auch mehrfach zusammengesetzte Einheiten sein.

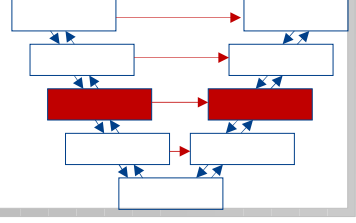
In der Praxis

- wird ein Softwaresystem selten auf der grünen Wiese entwickelt, sondern ein vorhandenes System wird verändert, ausgebaut oder mit anderen Systemen gekoppelt.
- Auch sind viele Systemkomponenten Standardprodukte, die am Markt dazu gekauft werden. Im Komponententest werden solche Alt- oder Standardkomponenten vermutlich nicht beachtet. Im Integrationstest müssen diese Systemteile jedoch berücksichtigt und deren Zusammenspiel mit anderen Teilen überprüft werden.



Integrationstest - Testumgebung

- Beim Integrationstest werden ebenfalls Treiber benötigt, die die Testobjekte mit Testdaten versorgen und Ergebnisse entgegennehmen sowie protokollieren.
- Es können die vorhandenen Treiber des Komponententests wiederverwendet werden.
- Da Schnittstellenaufrufe und der Datenverkehr über die Treiberschnittstellen getestet werden müssen, werden im Integrationstest als zusätzliches Diagnoseinstrument oft so genannte Monitore benötigt, die Datenbewegungen zwischen Komponenten mitlesen und protokollieren.



Integrationstest - Testziele

Die Testziele der Teststufe Integrationstest sind klar:

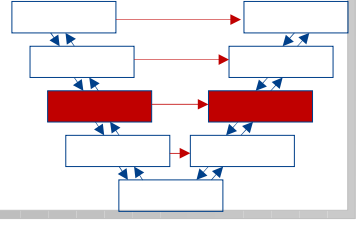
- Fehlerzustände in den Schnittstellen aufdecken.
- Fehlerzustände im Zusammenspiel zwischen Komponenten aufdecken.
- Auch der Test nicht-funktionaler Eigenschaften (z.B. Performanz) möglich.

Probleme können schon beim Versuch der Integration zweier Bausteine auftreten, wenn diese sich nicht zusammenbinden lassen,

- weil ihre Schnittstellenformate nicht passen,
- weil einige Dateien fehlen oder
- die Entwickler das System in ganz andere Komponenten aufgeteilt haben, als spezifiziert war.

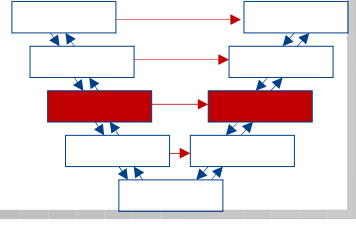
Die schwerer zu findenden Probleme betreffen allerdings die Ausführung der miteinander in Wechselwirkung stehenden Programmteile.

- Dies sind Fehlerzustände im Datenaustausch bzw. in der Kommunikation zwischen den Komponenten, die nur durch einen dynamischen Test aufgedeckt werden können.



Integrationstest – Testziele: Fehlerzustände in der Kommunikation

Zu welchen Typen von Fehlerzuständen könnte es Ihrer Meinung nach in der Kommunikation zwischen zwei Komponenten kommen ?

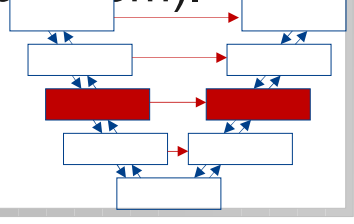


Integrationstest – Testziele: Fehlerzustände in der Kommunikation

Zu welchen Typen von Fehlerzuständen könnte es Ihrer Meinung nach in der Kommunikation zwischen zwei Komponenten kommen ?

Folgende Typen des Fehlerzustands können grob unterschieden werden:

- Eine Komponente übermittelt **keine oder syntaktisch falsche Daten**, so dass die empfangende Komponente nicht arbeiten kann oder abstürzt (funktionaler Fehler einer Komponente, inkompatible Schnittstellenformate, Protokollfehler). (Sonderfall: Übermittlung keiner Daten wegen Deadlock (Verklemmung).)
- Die Kommunikation funktioniert, aber die beteiligten Komponenten **interpretieren übergebene Daten unterschiedlich** (funktionaler Fehler einer Komponente, widersprüchliche oder fehlinterpretierte Spezifikationen). Beispiele:
  - „Several functions inside the OpenSSL library incorrectly check the result after calling the EVP_VerifyFinal function.“ http://www.ocert.org/advisories/ocert-2008-016.html
  - Verlust des Mars Climate Orbiters: "The 'root cause' ... was the failed translation of English units into metric units“ http://www5.in.tum.de/~huckle/bugs.html
- Die Daten werden richtig übergeben, aber **zum falschen oder verspäteten Zeitpunkt** (Timing-Problem) oder in zu kurzen Zeitintervallen (Durchsatz- oder Lastproblem).



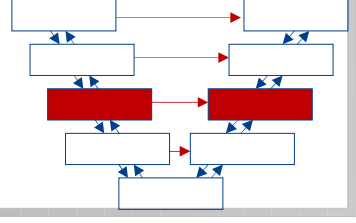
Keiner dieser Typen von Fehlerzuständen kann im Komponententest gefunden werden, denn die Fehlerwirkung äußert sich erst in der Wechselwirkung zwischen zwei Softwarebausteinen.

Integrationstest - ohne Komponententest ?

Kann auf den Komponententest verzichtet werden, sodass alle Testfälle erst nach erfolgter Integration durchgeführt werden ?

Das ist natürlich möglich und leider auch eine in der Praxis oft anzutreffende Vorgehensweise.

Welche Nachteile könnten damit verbunden sein ?



Integrationstest - ohne Komponententest ?

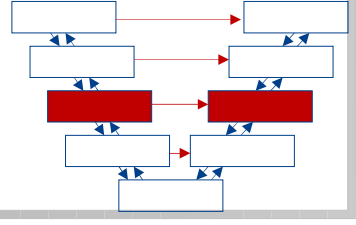
Kann auf den Komponententest verzichtet werden, sodass alle Testfälle erst nach erfolgter Integration durchgeführt werden ?

Das ist natürlich möglich und leider auch eine in der Praxis oft anzutreffende Vorgehensweise.

Welche Nachteile könnten damit verbunden sein ?

Damit verbunden sind gravierende Nachteile:

- Die meisten Fehlerwirkungen, die in derart angelegten Tests auftreten werden, sind durch funktionale Fehlerzustände einzelner Komponenten verursacht. Es wird also ein impliziter Komponententest in einer dazu nicht geeigneten Testumgebung durchgeführt, die den Zugang zur Einzelkomponente erschwert.
- Weil kein geeigneter Zugang zur Einzelkomponente möglich ist, können manche Fehlerwirkungen nicht provoziert und viele Fehlerzustände deshalb nicht gefunden werden.
- Wenn eine Fehlerwirkung oder ein Ausfall im Test auftritt, kann es schwierig oder sogar unmöglich sein, seinen Entstehungsort und damit seine Ursache einzugrenzen.

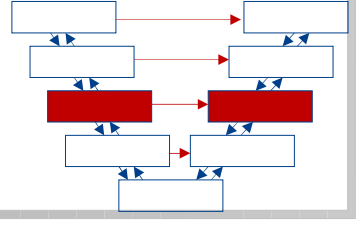


Integrationstest - Integrationsstrategie

In welcher Reihenfolge sind die Einzelkomponenten zu integrieren, damit die notwendigen Testarbeiten möglichst einfach und schnell durchgeführt werden ?

- Die verschiedenen im Projekt entstehenden Software-komponenten sind zu unterschiedlichen Zeitpunkten fertig, die eventuell Wochen oder Monate auseinander liegen können.
- Kein Projektmanager und auch kein Testmanager kann es tolerieren, dass seine Tester so lange untätig warten, bis alle Komponenten fertig sind und gemeinsam integriert werden können.

Im Folgenden werden wir uns einige Grundstrategien ansehen, an denen sich der Testmanager bei seiner Planung orientieren kann.



Integrationsstrategie: Inkrementelles Testen

Vorgehen:

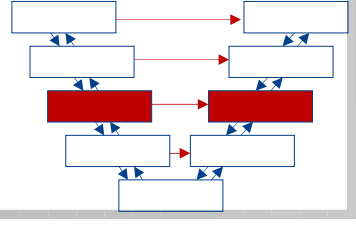
1. Der Test beginnt mit einem Modul X des Systems. X ist das bisherige Teilsystem. Der Rest wird durch Treiber bzw. Platzhalter ersetzt.
2. Füge Modul M zu bisherigem Teilsystem hinzu, wenn für M gilt:
   - M benutzt keine anderen Module oder
   - ein von M benutztes Modul gehört zum bisherigen Teilsystem oder
   - M wird von einem Modul benutzt, das zum bisherigen Teilsystem gehört
   - Der Rest wird wieder durch Treiber bzw. Platzhalter ersetzt.
3. Wiederhole Schritt 2 bis das „Teilsystem“ das ganze System ist.

Vorteile

- Schnittstellenfehler werden bei jedem Dazubinden sofort entdeckt.
- Fehlerlokalisierung wird vereinfacht.
- Zuerst ausgewählte Module werden gründlich getestet.

Nachteile

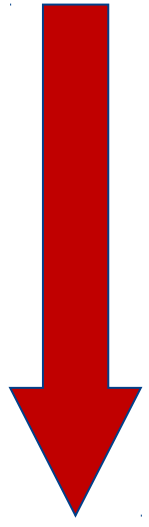
- Testaufwand höher als beim nicht-inkrementellen Testen.
- Bei Schritten 2 und 3 kann wenig parallel gearbeitet werden.



Inkrementelle Integrationsstrategie: Top-down-Integration

Top-down-Integration:

- Der Test beginnt mit der Komponente des Systems, die weitere Komponenten aufruft, aber selbst (außer vom Betriebssystem) nicht aufgerufen wird. Die untergeordneten Komponenten sind dabei durch Platzhalter ersetzt. Sukzessive werden die Komponenten niedrigerer Systemschichten hinzu integriert. Die getestete höhere Schicht dient dabei jeweils als Treiber.

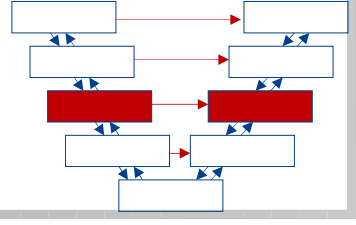


Vorteil:

- Es werden keine bzw. nur einfache Test-Treiber benötigt, da übergeordnete, bereits getestete Komponenten den wesentlichen Teil der Ablaufumgebung bilden.

Nachteil:

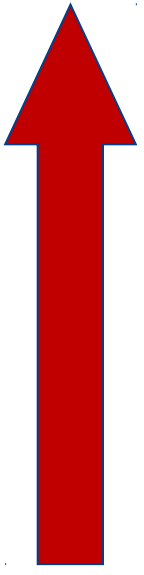
- Untergeordnete, noch nicht integrierte Komponenten müssen durch Platzhalter ersetzt werden, was sehr aufwändig sein kann.



Inkrementelle Integrationsstrategie: Bottom-up-Integration

Bottom-up-Integration:

- Der Test beginnt mit den elementaren Komponenten des Systems, die keine weiteren Komponenten aufrufen (außer Funktionen des Betriebssystems). Größere Teilsysteme werden sukzessive aus getesteten Komponenten zusammengesetzt, mit anschließendem Test dieser Integration.

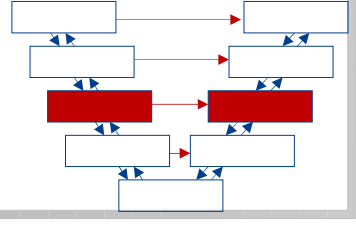


Vorteil:

- Es werden keine Platzhalter benötigt.

Nachteil:

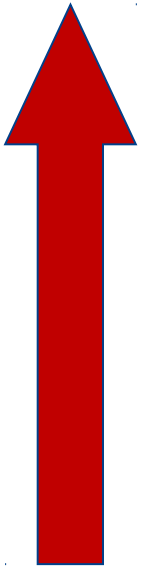
- Übergeordnete Komponenten müssen durch Test-Treiber simuliert werden.

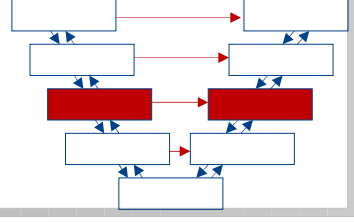


Inkrementelle Integrationsstrategie: Anmerkungen

Anmerkungen:

- Top-down- oder Bottom-up-Integration lassen sich in Reinform nur bei streng hierarchisch gegliederten Programmsystemen einsetzen (in der Praxis selten).
- Daher wird in der Realität immer eine mehr oder minder individuelle Mischung der beiden Integrationsstrategien gewählt.
- Je größer der Umfang einer Integration ist, um so schwieriger ist die Isolation von Fehlerzuständen und um so höher ist der Zeitbedarf zur Fehlerbehebung.

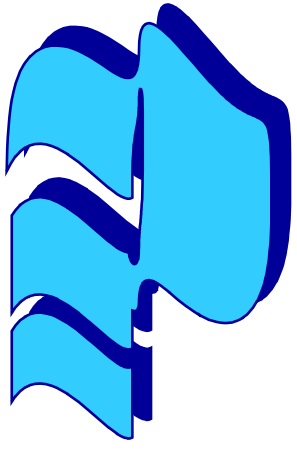




Integrationsstrategie: Ad-hoc-Integration

Ad-hoc-Integration:

- Die Bausteine werden z. B. in der (zufälligen) Reihenfolge ihrer Fertigstellung integriert. Sobald eine Komponente ihren Komponententest absolviert hat, wird geprüft, ob sie zu einer anderen schon vorhandenen und bereits getesteten Komponente oder zu einem teilintegrierten Subsystem passt. Wenn ja, werden beide Teile integriert und der Integrationstest zwischen beiden wird durchgeführt.

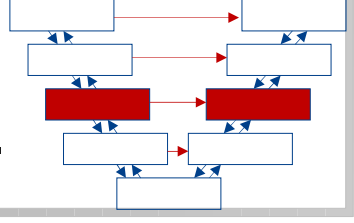


Vorteil:

- Zeitgewinn, da jeder Baustein frühestmöglich in seine passende Umgebung integriert wird.

Nachteil:

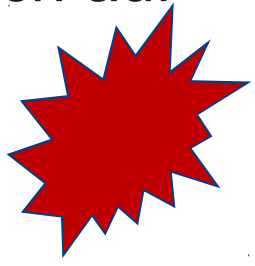
- Es werden sowohl Platzhalter als auch Treiber benötigt.



Integrationsstrategie: *big-bang*-Integration

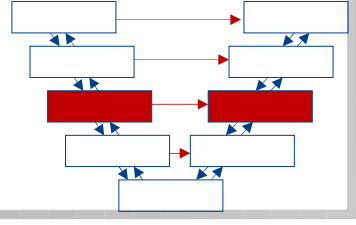
Nicht inkrementelle Integration – *big-bang*-Integration:

- Es wird hierbei mit der Integration gewartet, bis alle Softwarebauteile entwickelt und getestet sind, und dann wird alles auf einmal zusammengeworfen. Im schlimmsten Fall wird auch auf vorgelagerte Komponententests verzichtet.



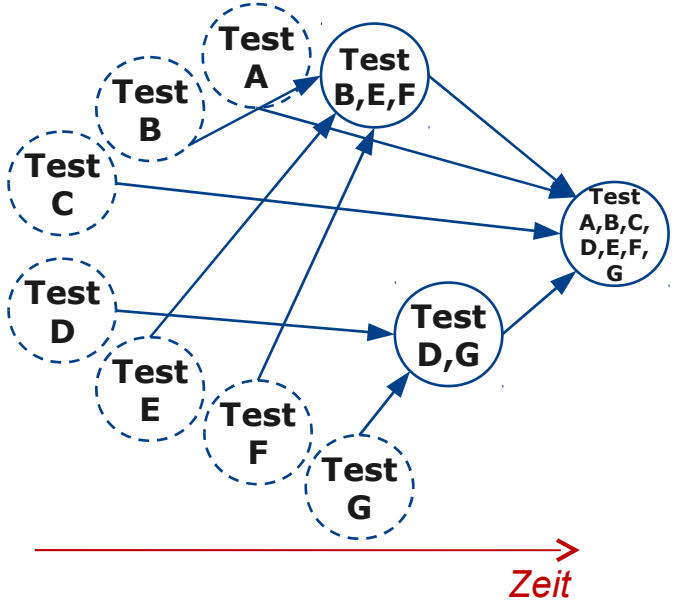
Nachteile:

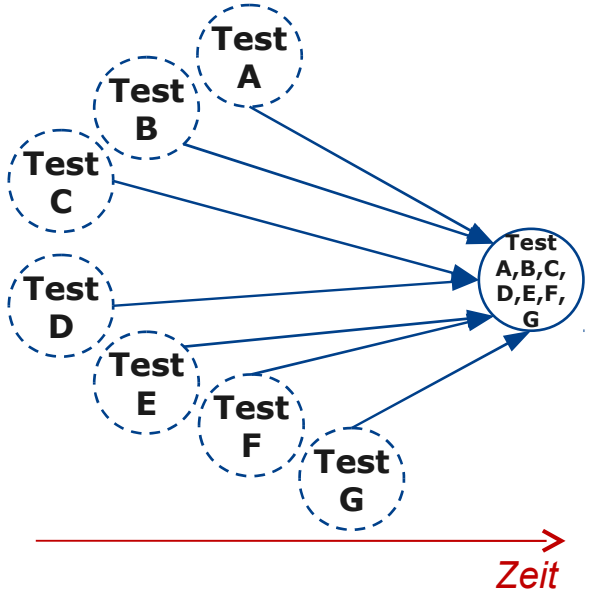
- Die Wartezeit bis zum *big-bang* ist leichtfertig verlorene Testdurchführungszeit. Da Testen ohnehin immer unter Zeitmangel leidet, sollte kein einziger Testtag verschenkt werden.
- Alle Fehlerwirkungen treten geballt auf; es wird schwierig oder unmöglich sein, das System überhaupt zum Laufen zu bringen.
- Die Lokalisierung und Behebung von Fehlerzuständen gestaltet sich schwierig und zeitraubend.

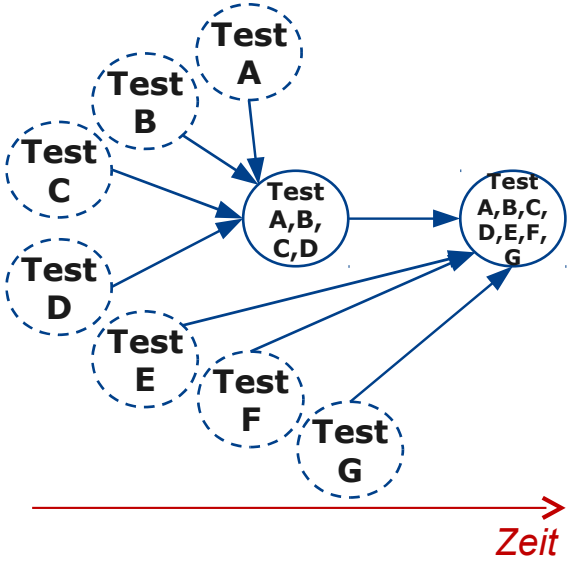


Integrationsstrategien am Beispiel

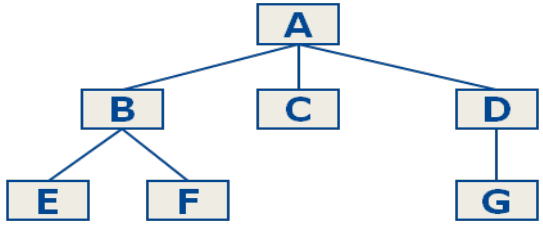
Welche Strategie liegt jeweils vor ?
(big-bang, bottom-up, top-down)





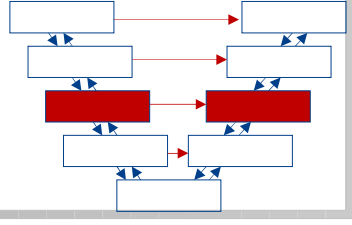


Beispielhierarchie:



Komponententest

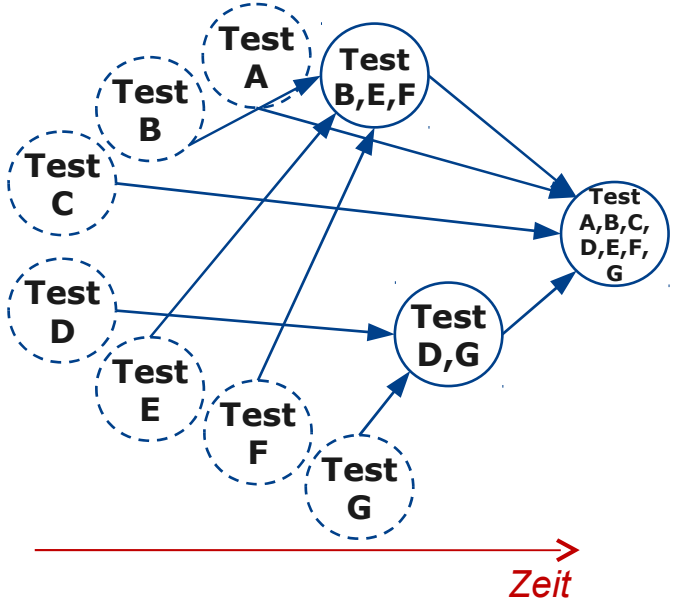
Integrationstest

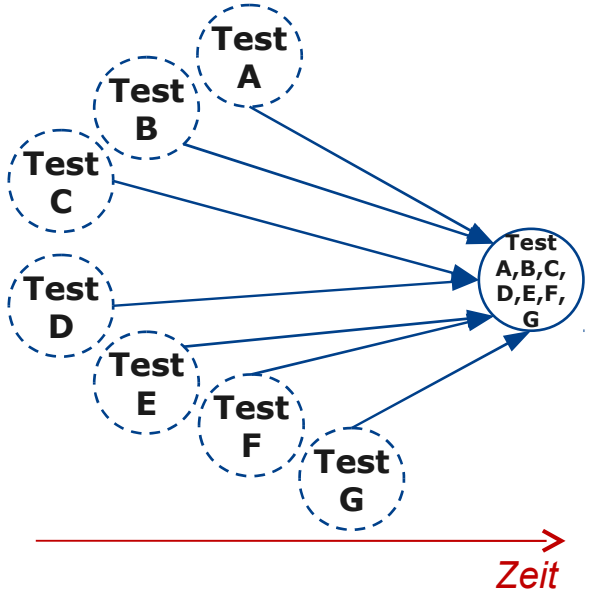


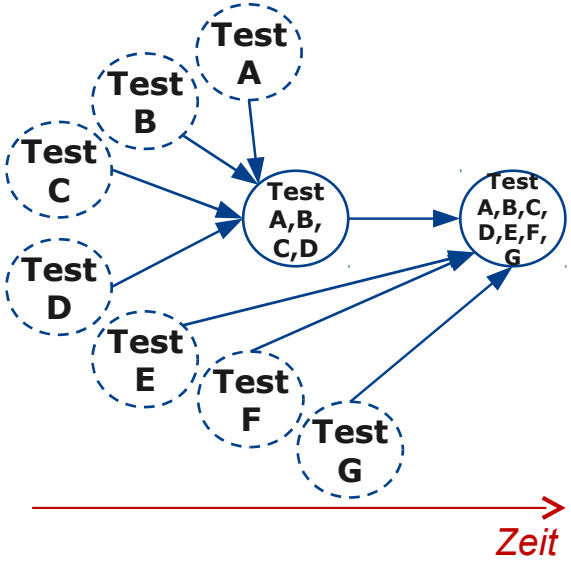
Integrationsstrategien am Beispiel

Bottom-up Integration: Integrationstest eines Teilsystems (bottom-up), sobald Komponententests aller enthaltenen Knoten vorliegen.

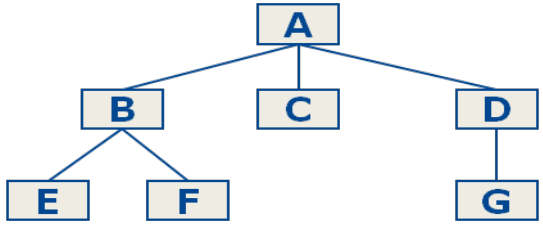
Bottom-up Integration:

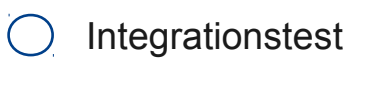


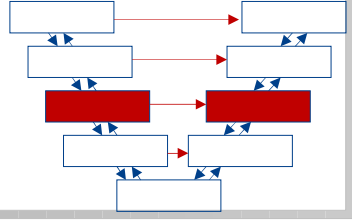




Beispielhierarchie:



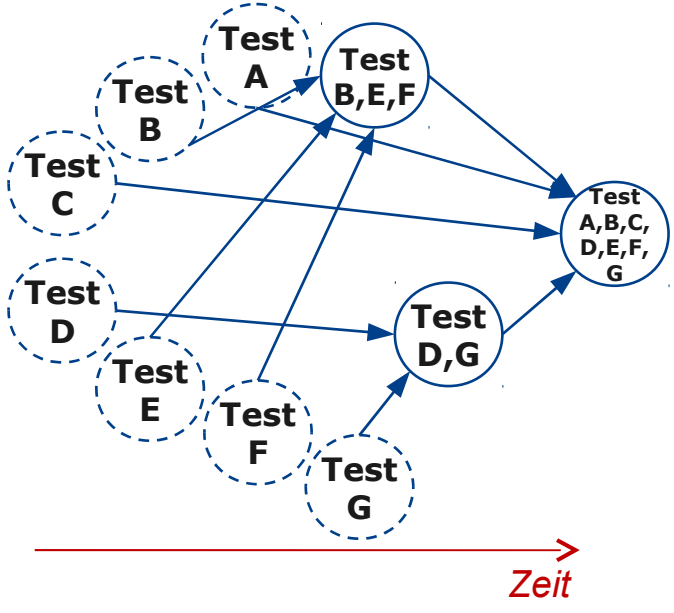




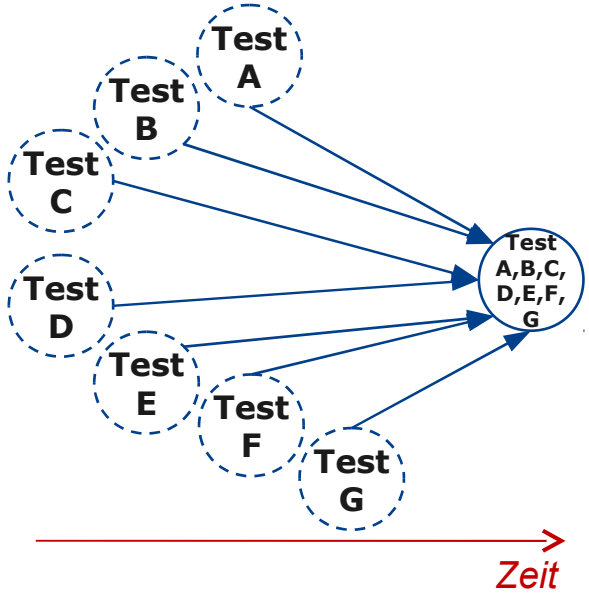
Integrationsstrategien am Beispiel

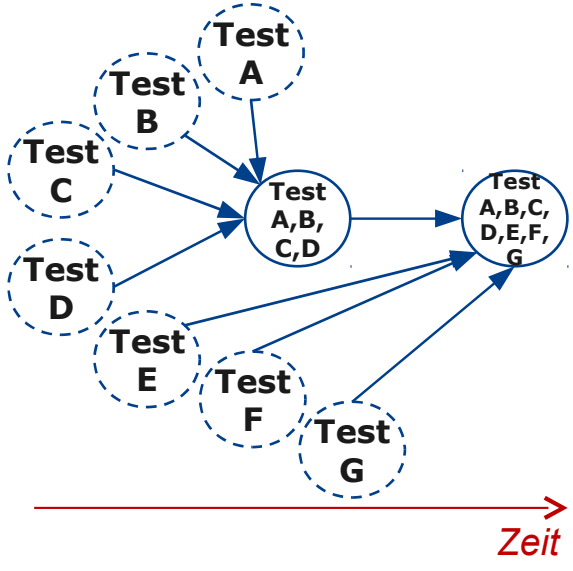
Big-Bang Integration: Integrationstest des Gesamtsystems nach Vorliegen aller Komponententests

Bottom-up Integration:

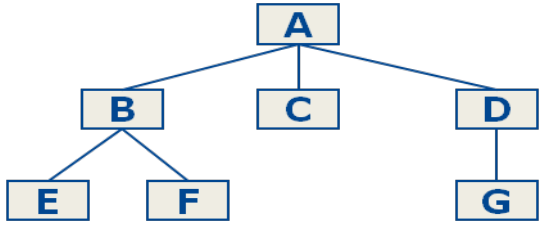


Big-Bang Integration:



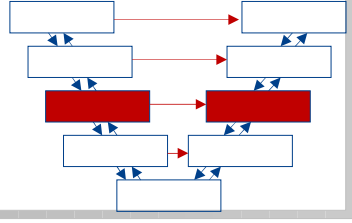


Beispielhierarchie:



Komponententest

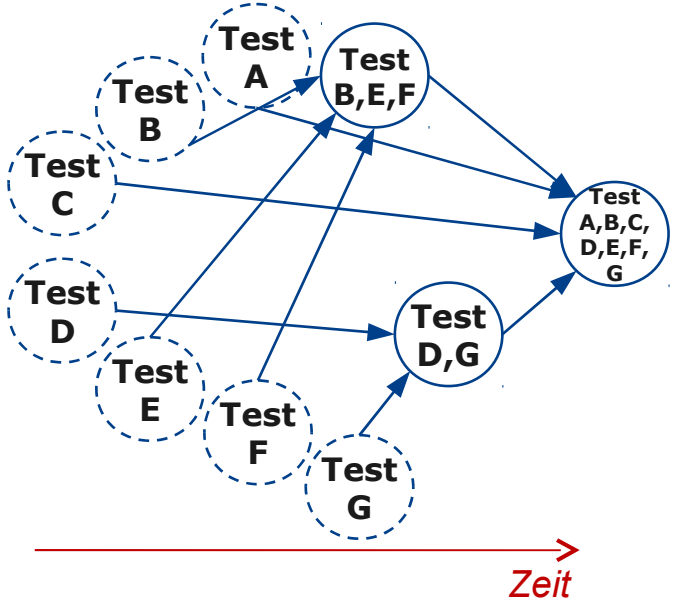
Integrationstest



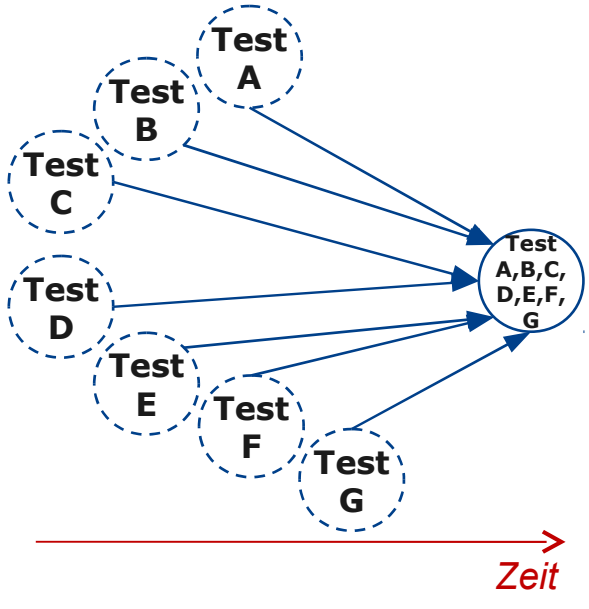
Integrationsstrategien am Beispiel

Top-down Integration: Integrationstest eines Teilsystems (top-down), sobald Komponententests der direkten Tochterknoten vorliegen.

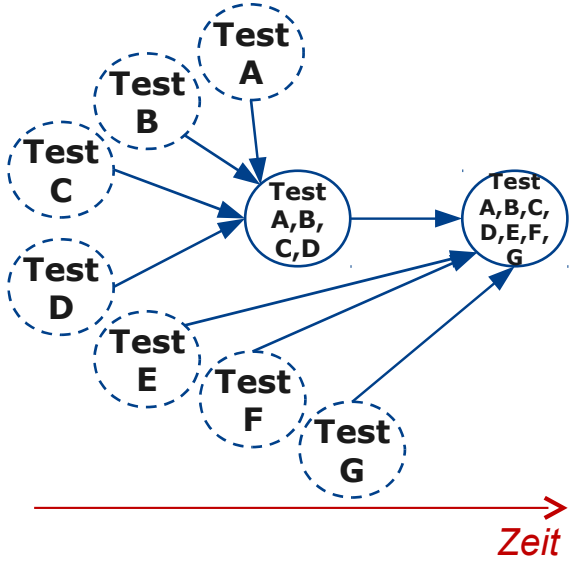
Bottom-up Integration:



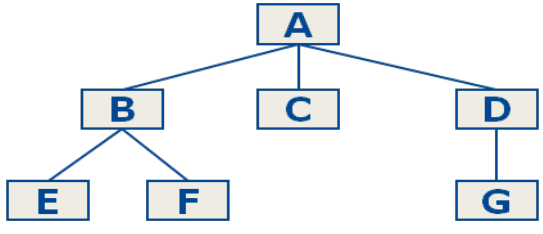
Big-Bang Integration:



Top-down Integration:

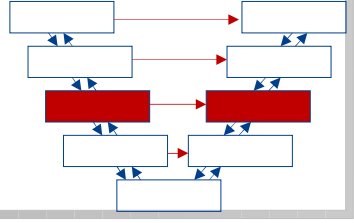


Beispielhierarchie:



Komponententest

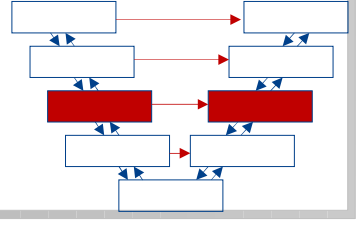
Integrationstest



Integrationstest - Auswahl Integrationsstrategie

Welche Integrationsstrategie optimal ist (zeitsparend, kostengünstig), hängt von Randbedingungen ab, die in jedem Projekt individuell analysiert werden müssen:

- Systemarchitektur
  - bestimmt, aus welchen und wie vielen Komponenten das Gesamtsystem besteht und wie diese voneinander abhängen.
- Projektplan
  - legt fest, zu welchen Zeitpunkten im Projekt einzelne Systemteile entwickelt werden und wann diese testbereit sein sollen.
- Testkonzept / Mastertestkonzept
  - legt u.a. fest, welche Systemaspekte wie intensiv getestet werden müssen und auf welcher Teststufe das jeweils geschehen soll.
- Testmanager
  - muss aus diesen Randbedingungen die für sein Projekt passende Integrationsstrategie aufstellen, in Absprache mit dem Projektmanager.



Integrationstest – Testobjekte

Kleine Systeme: **Mikroskopische** Betrachtung:

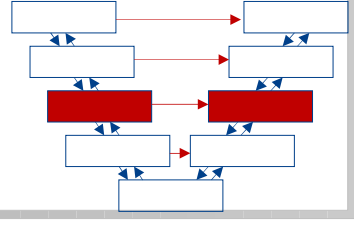
- Anweisungen, Daten, Kontroll- und Datenfluss

Große Systeme: **Makroskopische** Betrachtung:

- **Operationen / Funktionen / Methoden**
- **Klassen / Module / Teilsysteme**
- **Schnittstellen**
- **Operationsaufrufgraph**
- **Benutzbeziehungen**: Operationsaufrufe und benutzte Typdefinitionen
- **Modulgraph:** Benutzbeziehungen zwischen Modulen

Nichtmonolithischer Test: nicht Alles-auf-einmal-Testen !

Auf den nächsten Folien sprechen wir kurz einige statische und dynamische Integrationstestmethoden an; mehr technische Einzelheiten zu den Methoden gibt es dann in Abs. 2.3-2.5.



Integrationstest – Testobjekte: Kontrollflussgraph

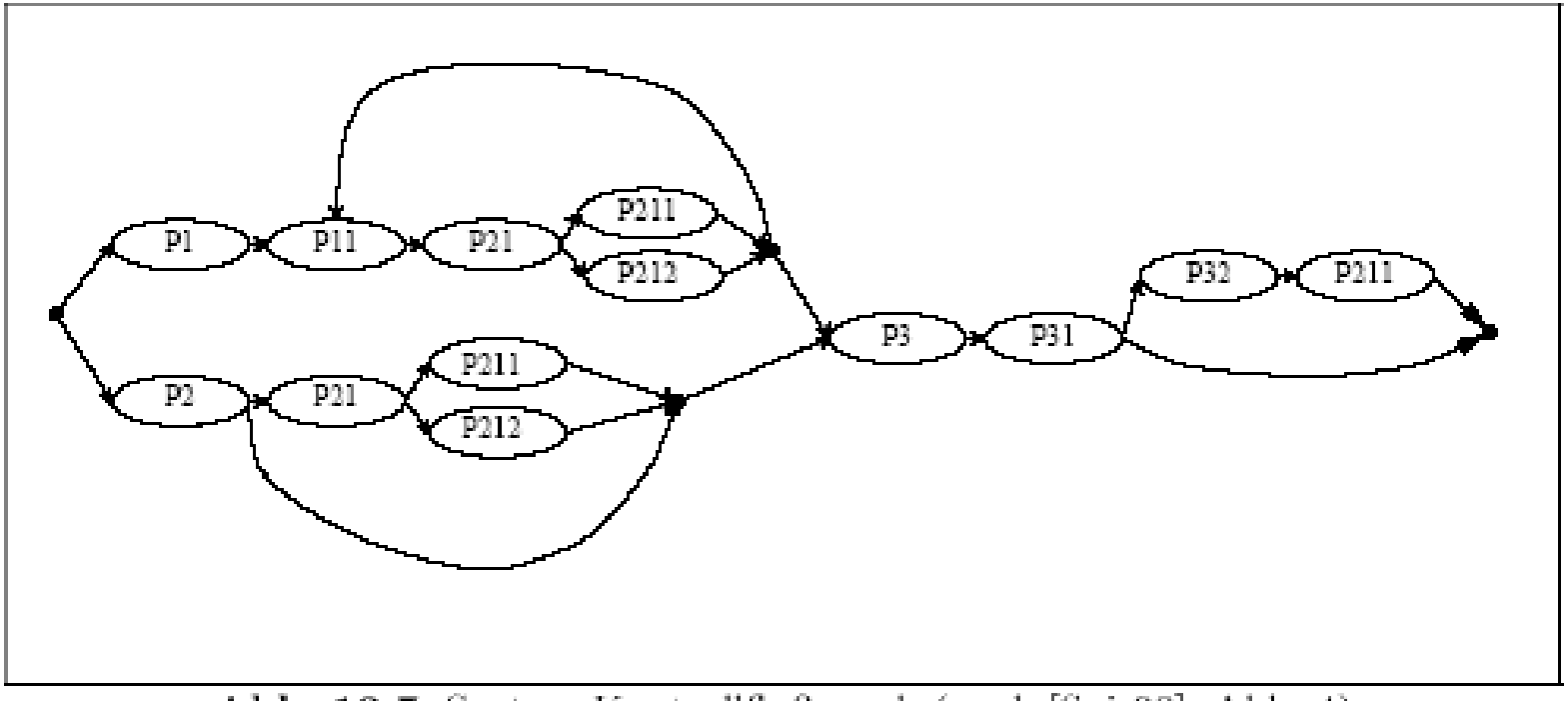
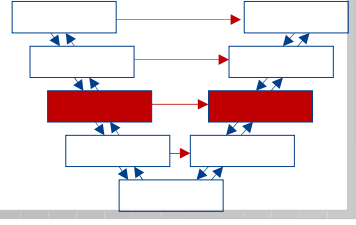


Abb. 13.7: System-Kontrollflußgraph (nach [Spi 90], Abb. 4)



Integrationstest – Testobjekte: Modulgraph

MODUL IM1

```
import P1;
...
P1(j);
...
```

MODUL IM2

```
import P1;
...
P1(x);
...
P1(y);
...
```

MODUL IM3

```
import P1;
...
P1(i);
...
if z > 0
then P1(z);
else ...
...
while k > j
do P1(k);
...
```

MODUL EX

```
export P1;
...
procedure P1(var x: integer) ;
...
end P1;
...
```

Abb. 13.1: Programmcode von vier Modulen (s. [Spi 95], Fig. 2)

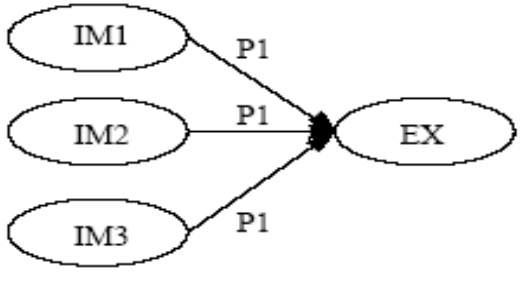
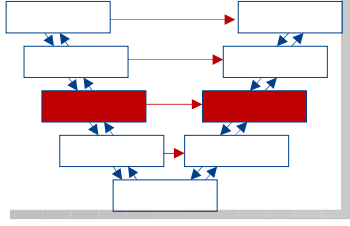


Abb. 13.2: Modulgraph zur Struktur von Abb. 13.1



Integrationstest – Testobjekte: Modulgraph

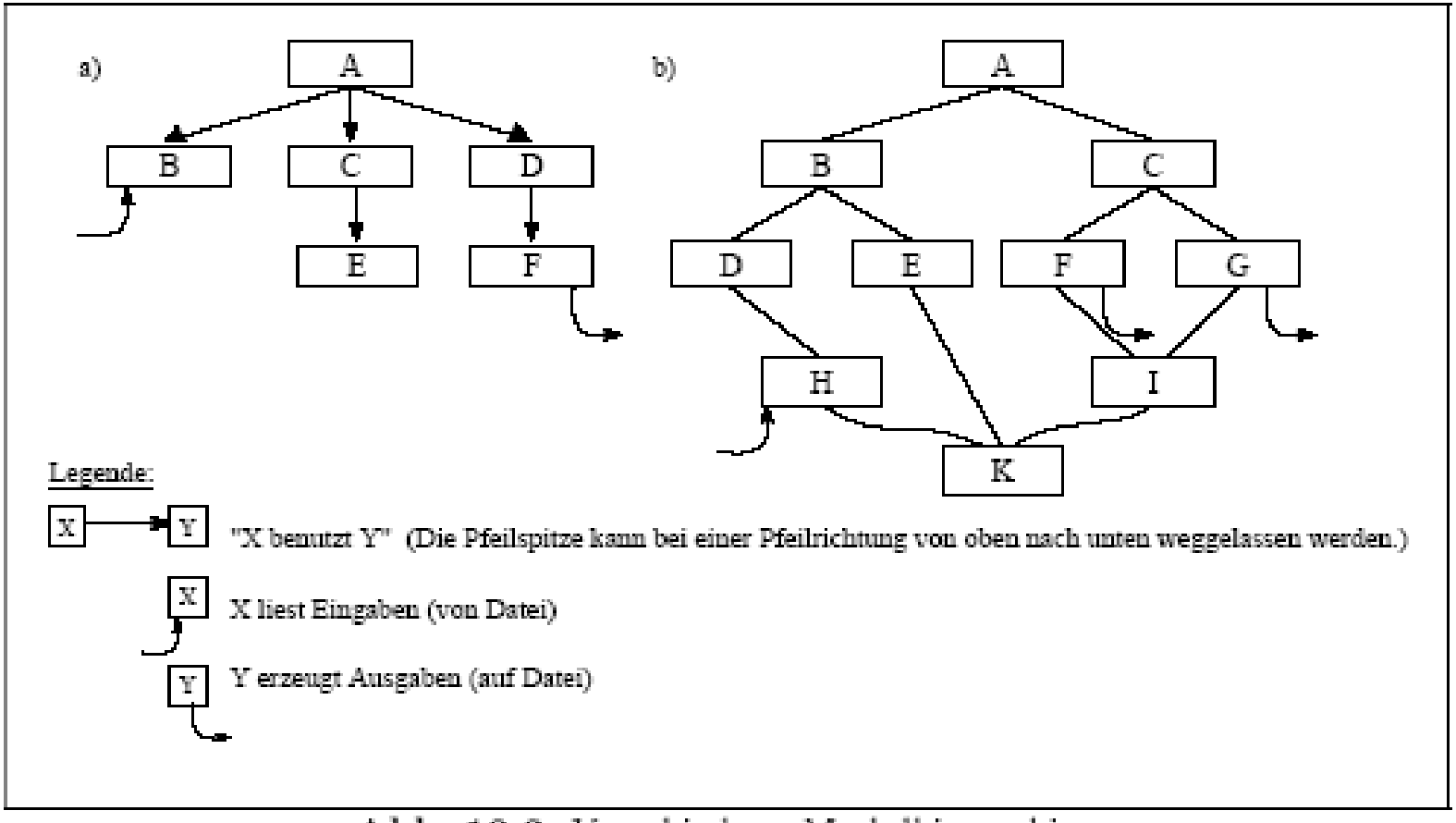
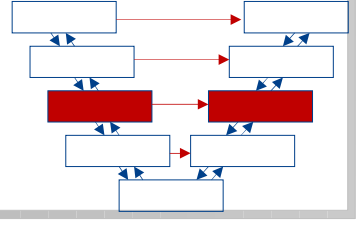
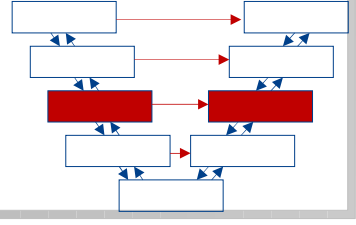


Abb. 13.3: Verschiedene Modulhierarchien



Statische Integrationstestmethoden

- **Syntaxprüfung der Schnittstellen** (durch Übersetzer bzw. Binder[1]):
  Damit können grobe Fehler wie falsche Parameteranzahl oder -typen bei streng getypten Programmiersprachen erkannt werden.
- **Vergleich der** (realisierten) **Modulkopplungen** mit dem Entwurf (d.h. den geplanten Modulkopplungen):
  Die Abweichungen können gewollt oder unabsichtlich - also vermutlich fehlerhaft - sein; das ist zu überprüfen, und zwar mit statischer Analyse.
- **Anzeige verdeckter Abhängigkeiten:**
  Verwaltet und exportiert z.B. ein Modul A eine (globale) Variable, die von zwei anderen Modulen B und C verwendet wird, so besteht zwischen B und C eine Abhängigkeit, die aus dem Programmtext nicht direkt zu entnehmen ist.
- **Intermodulare Datenflussanalyse:**
  Bei dieser Vorgehensweise werden Anomalien im Datenfluss untersucht.
  Beim Integrationstest wird speziell der Datenfluss bei der Übergabe und Verarbeitung von **Schnittstellenvariablen** betrachtet (das sind die Parameter und globalen Variablen).



[1] Ein Binder ist ein Programm, das vordefinierte und in einem Programm aufgerufene Funktionen aus bestimmten Programmbibliotheken zum eigentlichen Programm hinzufügt.

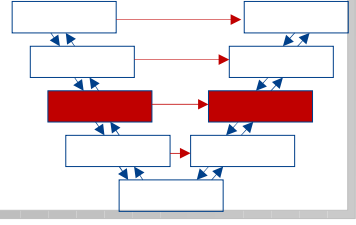
Intermodulare Datenflussanalyse:

Im allgemeinen Fall (wenn die Operationsparameter nicht genau als Eingabe- oder Ausgabeparameter deklariert sind) muss eine kombinierte Analyse der beiden Module bzw. Operationen erfolgen.

Für einen **Operationsaufruf** bedeutet dies:

- Im **aufrufenden Modul** müssen alle möglichen *letzten* Aktionen bzw. Zustände der Schnittstellenvariablen *vor* dem Aufruf ermittelt werden (define, reference oder undefine).
- In der **aufgerufenen Operation** müssen alle möglichen *ersten* Aktionen der Schnittstellenvariablen ermittelt werden (define, reference oder undefine).

Welche Anomalien sind bei der Verwendung von define, reference oder undefine im aufrufenden bzw. aufgerufenen Modul möglich ?

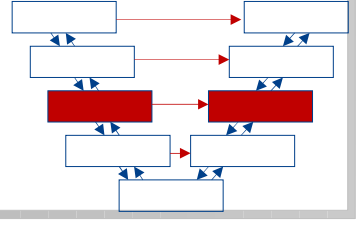


Intermodulare Datenflussanalyse:

Im allgemeinen Fall (wenn die Operationsparameter nicht genau als Eingabe- oder Ausgabeparameter deklariert sind) muss eine kombinierte Analyse der beiden Module bzw. Operationen erfolgen.

Für einen **Operationsaufruf** bedeutet dies:

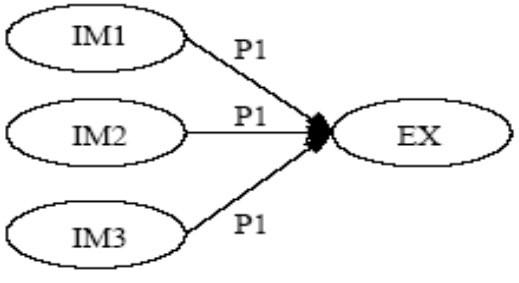
- Im **aufrufenden Modul** müssen alle möglichen *letzten* Aktionen bzw. Zustände der Schnittstellenvariablen *vor* dem Aufruf ermittelt werden (define, reference oder undefine).
- In der **aufgerufenen Operation** müssen alle möglichen *ersten* Aktionen der Schnittstellenvariablen ermittelt werden (define, reference oder undefine).
- Kann als letzte Aktion vor dem Aufruf ein define vorkommen und als erste Aktion in der aufgerufenen Prozedur ebenfalls ein define, nennen wir dies eine **potenzielle dd-Anomalie**. (Entsprechendes gilt für **du-** und **ur-Anomalien**.)
- Mit vertauschten Rollen muss eine entsprechende Analyse für die *letzten* Aktionen in der **aufgerufenen Operation** und die *ersten* Aktionen nach dem Aufruf im **aufrufenden Modul** erfolgen !

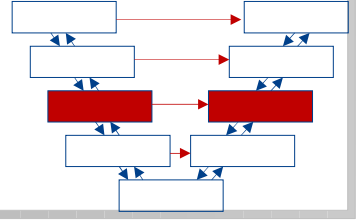


Ablaufbezogener Integrationstest: Kontrollflusskriterien (1):

Kontrollflussbasierte Vorgehensweise mit Orientierung am Modulgraphen:

- **Alle Module**:
  Jedes Modul muss mindestens einmal bei einem Test aufgerufen werden.
- **Alle Relationen**:
  Jedes Modul muss mindestens einmal von jedem Modul (von dem es überhaupt aufgerufen wird) bei einem Test aufgerufen werden - dies entspricht der Ausführung jeder Kante im Modulgraphen.
- **Alle Relationen mehrfach**:
  Zu jeder Operation P, die von einem anderen Modul M aufgerufen wird, muss es einen Test geben, bei dem M Operation P mindestens einmal aufruft - dies entspricht der Ausführung jeder Kante im Modulgraphen mit jedem als Markierung angegebenen Aufruf (vgl. F. 58).

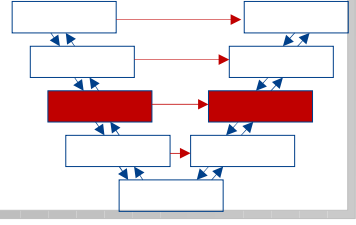




Ablaufbezogener Integrationstest: Kontrollflusskriterien (2):

Kontrollflussbasierte Vorgehensweise mit Orientierung am Modulgraphen:

- **Alle Importe mehrfach**:
  Zu jeder Operation P, die von einem anderen Modul M aufgerufen wird, muss es Tests geben, bei denen jede Aufrufstelle von P in M mindestens einmal ausgeführt wird (dies erfordert eine Ergänzung des Modulgraphen um die Lage und Anzahl der Aufrufstellen von externen Operationen in den Modulen).
- **Alle Aufrufreihenfolgen**:
  Jede mögliche Reihenfolge von Operationsaufrufen über Modulgrenzen hinweg ist bei einem Test auszuführen (d.h. alle Wege im Modulgraphen sind auszuführen).



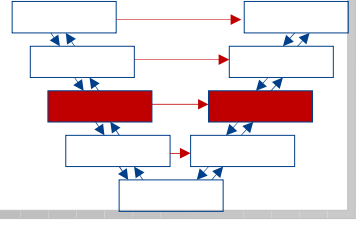
Ablaufbezogener Integrationstest: Datenflusskriterien:

Kontrollflussbasierte Vorgehensweise mit Orientierung am Modulgraphen: Statisch festgestellte **potenzielle Datenflussanomalien** sollen ausgeführt werden. Dazu ist der **Datenfluss bei Operationsaufrufen an der Schnittstelle** von Modulen zu testen (analoge Vorgehensweise wie beim Ermitteln von Datenflussanomalien):

a) Im ***aufrufenden*** Modul müssen die letzten [bzw. ersten] Aktionen auf den Schnittstellenvariablen vor [bzw. nach] dem Aufruf ermittelt werden (kritisch sind **define** [bzw. **reference**]-Aktionen).

b) In der ***aufgerufenen*** Operation müssen ebenfalls die ersten [bzw. letzten] Aktionen auf den Schnittstellenvariablen ermittelt werden (kritisch ist eine **reference** [bzw. **define**]).

Dann sind **alle (bzw. einige) Wege** von den ermittelten letzten define-Aktionen im aufrufenden Modul zu den ermittelten ersten reference-Aktionen in der aufgerufenen Operation **beim Test auszuführen.** Die **Wegeauswahl** kann sich an **Datenflusskriterien** aus Abs. 2.5 orientieren. Entsprechend für Wege von den letzten define-Aktionen der aufgerufenen Operation zu den ersten reference-Aktionen des aufrufenden Moduls nach dem Aufruf.



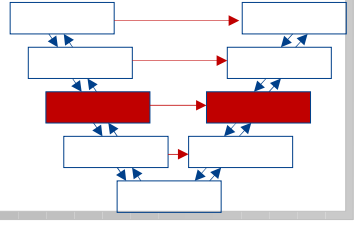
Wertbezogener Integrationstest:

Bestimmte Werte für Parameter verwenden, mit denen mit größerer Wahrscheinlichkeit Fehler aufgedeckt werden können.

Um **Sonderfälle** zur Ausführung zu bringen, die beim Modultest wegen der Platzhalterproblematik bisher nicht getestet wurden, kommen die folgenden herausragenden Werte in Frage:

- Grenzwerte
- Extremwerte (0, 1, -MAXINT, +MAXINT, etc.)
- Fehlerfälle (z.B. negative Werte)

Beim Integrationstest sind außerdem Fehler aufzudecken, die dadurch entstehen, dass eine Operation P mit **falschen Werten der Schnittstellenvariablen** aufgerufen wird.



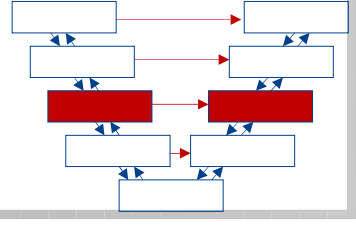
Funktionsbezogener Integrationstest:

- Spezifikationsorientierter Test, der das korrekte Zusammenspiel der von den einzelnen Modulen realisierten Teilfunktionen bzw. Abweichungen von der spezifizierten Funktionalität feststellen soll.

Abweichungen können von folgender Art sein:

- **Mangelnde Funktionalität**, d.h. Modul liefert erwartete Teilfunktionen nicht.
- **Zuviel Funktionalität**, d.h. eine Teilfunktion wird vom Modul zusätzlich ausgeführt, obwohl dies nicht spezifiziert bzw. erwartet ist.
- Die Funktionalität des Moduls ist **falsch.**

Für **jede exportierte Operation P** eines Moduls sollte überprüft werden, ob die Funktionalität mit den Erwartungen aller Module, die P importieren, übereinstimmt bzw. davon abweicht.
Diese Überprüfung sollte durch eine **informelle Analyse (Inspektion)** und durch **Tests** erfolgen, wobei die Testdaten aus den ablauf- und wertbezogenen Tests verwendet werden können.



Integrationstest-Werkzeug: CruiseControl

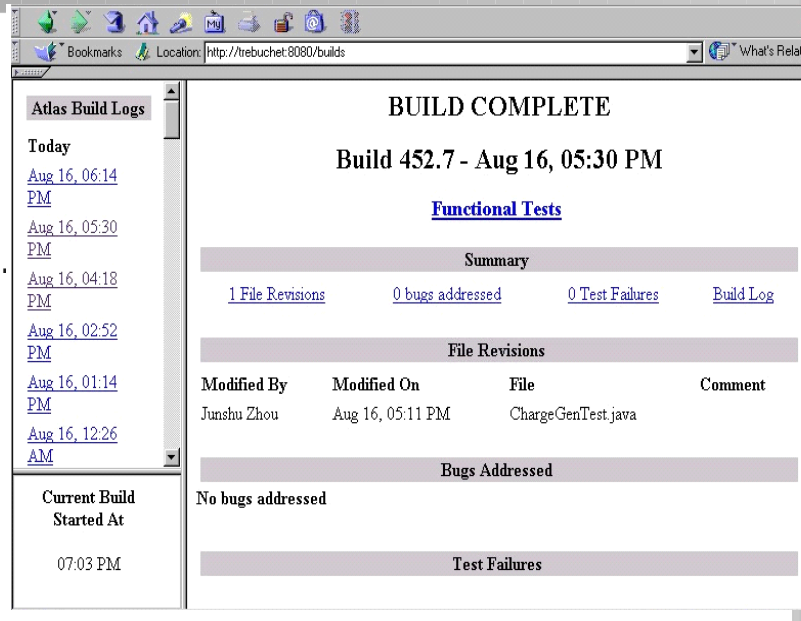
CruiseControl: Open Source Werkzeug zur kontinuierlichen Integration von Systemen (ThoughtWorks).

Kontinuierliche Integration (Prinzip von XP):

- (Mehrmals) tägliches Build der fertigen Systemeinheiten.

Vorteile der kontinuierlichen Integration:

- Fehler werden direkt nach der Integration der neuen / eänderten Komponente mit dem restlichen System identifiziert.
- Auch wenn der Fehler nicht genau lokalisiert werden kann, kann entschieden werden, ob der betreffende Code wieder entfernt wird oder nicht (falls das Feature wichtiger ist als der Fehler).



Prinzipien:

- Single Source Point (z.B. CVS)
  - Gesamter Quellcode wird an einer zentralen Stelle verwaltet.
  - Von dort auch kann jeder Entwickler Quellcode ein-, auschecken.
- Automatisiertes Build-Skript (z.B. Ant)
- „Selbst-testender“ Code (z.B. JUnit)

Ausreden: Projekt ist zu groß für ein tägliches Build.

- Gilt da nicht, da beispielsweise Microsoft tägliche Builds erstellt (für Projekte mit 10 Mio Zeilen Code).

- Gefährlich, da Integrationsaufwand exponentiell steigt mit dem Integrationsintervall (für einwöchigen Abstand 25-fachen Aufwand).

Gegenargument: Aufwand zum Aufsetzen einer kontinuierlichen Integrationsumgebung ist gering, da anschließend der gesamte Compile-, Build und Testprozess automatisch abläuft.

Buildprozess:

- Buildroutine startet zeitabhängig (z.B. jede Stunde) oder ereignisabhängig (z.B. neue Codeversionen).

- Gesamten Quellcode auschecken, kompilieren und deployen.

- Bei Fehlern wird die entsprechende Datei identifiziert und der Entwickler per Mail benachrichtigt.

- Dieser muss den Fehler in einer festzulegenden Zeit beheben und die Datei neu einchecken oder die Datei zurückziehen.

- Hierzu kann der Entwickler den gesamten Quellcode auf seine lokale Entwicklungsumgebung auschecken.

- Entwickler checken Code in festgelegten Zeitabständen ein (z.B. täglich).

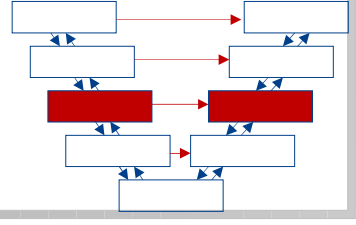
Testen objektorientierter Systeme

Objektorientierung ist technologisch gesehen ein komplexeres Paradigma als die traditionellen strukturierten Ansätze der Softwareentwicklung

Andere, neue Herausforderungen an Testverfahren

Unterschiede beim Modul- und Integrationstest:

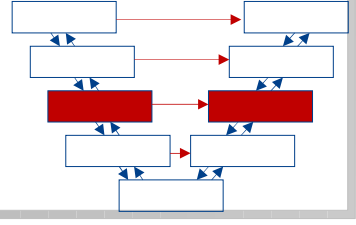
- Klassen mit Vielzahl „verquickter“ Methoden
- Offenheit der Methoden für diverse Zwecke
  - → Vielzahl möglicher Zustände
  - → komplexe Objekte mit vielen Zuständen



Testen objektorientierter Systeme

Fehleranfälligkeit durch:

- Polymorphie & dynamische Bindung
  → Vermehrung potenzieller Ablaufpfade
- Kapselung → beschränkte Sicht für Fehlererkennung
- viele Kollaborationen → viele Schnittstellen → viele Abstimmungen → Missverständnisse → Fehler
- Vererbung → subtile Abhängigkeiten
- Klassen in Vererbungshierarchien dreimal so fehleranfällig wie Klassen ohne Vererbung [ShCa 97]



Merkmale objektorientierter Systeme

Stärkere Modularisierung führt zu mehr intermodularen Abhängigkeiten.

- Methode einer Klasse ist oft auf Methoden anderer Klassen angewiesen.
- Für die Klassen sind verschiedene Entwickler zuständig, d.h. auch Abhängigkeiten zwischen den Entwicklern.
- Arbeitsteilung nicht mehr funktional, sondern objektorientiert ausgerichtet.
- Mechanismen wie Vererbung und Verwendung öffentlicher Methoden erhöhen Abhängigkeiten.
- Redundanz wird auf Kosten der gegenseitigen Abhängigkeit eliminiert.

Abhängigkeit einzelner Methoden über den Zustand gemeinsamer Objektattribute

- Methode hinterlässt Objektzustand, der das Verhalten der Nachfolgemethode beeinflusst.

Klassen oft funktional umfangreicher als nötig

- Oft nicht bekannt, zu welchem Zweck die Methoden der Klassen verwendet werden.
- Daher so programmiert, dass jeder potentielle Zweck erfüllt werden kann.
- Diese Offenheit führt zu einer Vielzahl möglicher Zustände, die nur bei hohen Kosten getestet werden können.
- Diskrepanz zwischen funktionsbezogener und objektorientierter Software.

Merkmale objektorientierter Systeme

Zustandsvielfalt eines Objekts erfordert viele Tests.

- Je komplexer ein Objekt, desto mehr Attribute und Methoden, desto mehr Zustände.
- D.h. alle möglichen Zustände und Zustandsübergänge zu testen.
- Exponentielle Steigerung der Anzahl von Testfällen.

Klassische Testverfahren können nicht ohne weiteres auf OO-Systeme angewendet werden.

- Klasse enthält eine Menge von Methoden.
- Von einer Anwendung wird jedoch nur ein Teil der Methoden benutzt.
- Es muss daher nicht der gesamte Code analysiert werden, sondern nur die Methoden, die im Rahmen einer Transaktion beansprucht werden.
- Dieser selektive Test ist technisch anspruchsvoller als der blinde Test des gesamten Codes (bei klassischen Verfahren).

Merkmale objektorientierter Systeme

OO-Systeme fehleranfälliger als konventionell funktionsorientierte Systeme:

- Polymorphie und dynamische Bindung führen zu einer Vermehrung der potentiellen Ablaufpfade und damit der potentiellen Fehler.
- Vererbung schafft viele subtile, unsichtbare Abhängigkeiten.
- Kapselung beschränkt die Sicht auf die Objektzustände und erschwert dadurch die Fehlererkennung.
- Zahlreiche Kollaborationen zwischen Objekten schafft viele Schnittstellen, die wiederum zu Fehlern führen können.
- Studie: Klassen in Vererbungshierarchien sind dreimal so fehleranfällig wie Klassen ohne Vererbung [SC97].
- Moderne Technologie wird oft nicht beherrscht und führt daher zu Fehlern.

[SC97]: Sheppard, M.; Cartwright, M.: An empirical study of object-oriented metrics, TR97/01, Bournemouth University, UK, 1997

[SW02]: Sneed, H.; Winter, M.: Test objektorientierter Software, Hanser, 2002

Methoden:

- sind aufgrund ihrer Spezifikation (klar spezifizierte Eingangs- und Ausgangsdaten) einzeln testbar.
- Wechselwirkungen zwischen Methoden einer Klasse und die Abhängigkeit der Verarbeitung vom Zustand der nächst größeren Einheit, der Klasse, erschweren allerdings den unabhängigen Test.

Klassen:

- kapseln Methoden.
- können bzgl. Ihrer Schnittstellen einzeln getestet werden.

Pakete:

- Mengen von Klassen

Komponenten:

- Klassenmengen, in denen nur wenige Abhängigkeiten zu Klassen in anderen Komponenten bestehen (sollten).
- Komponenten daher zum Testen gut geeignet.

Systeme

Folgen der Kapselung

Vorteile:

- Objekte abgeschlossen mit fest definierten Schnittstellen.
- Objekt kann unabhängig von anderen Objekten getestet werden.
- Fehlerursachen leichter lokalisierbar.

Nachteile:

- Alle Zustände eines Objekts müssen getestet werden.
- Schnittstellen komplex.
- Schnittstellen müssen in allen Varianten getestet werden.
- Ausführung von Methoden hängt vom aktuellen Zustand des Objekts ab, welcher ggf. von vorher ausgeführten Methoden abhängt.
- Kombinationen von Methodenausführungsfolgen testen.
- Kontrolle der Objektzustände wegen Kapselung schwierig.
- Umgehung durch z.B. friend-Methoden (C++) möglich, die die Kapselung aufheben.

Folgen der Vererbung

Vorteile:

- Komplexität wird durch Reduktion des Quellcodes bekämpft.

Nachteile:

- „Vererbung ist GOTO-Anweisung der Objektorientierung“ [SW02], da Unterklassen direkter Zugriff auf Attribute und Operationen der Oberklassen gewährt wird:
  - Abhängigkeit der Unterklassen von den Oberklassen.
  - Unklar, ob und welche der geerbten Methoden getestet werden müssen.
  - Überschreiben geerbter Funktionen (overloading) erhöht die Komplexität der Abläufe.
  - Ändern von Parametertypen (overriding) verschleiert Schnittstellen.
- Fehler in Basisklassen werden auf abgeleitete Unterklassen übertragen, d.h. Fehler werden vererbt.
  - Basisklassen gründlich testen, damit Fehler nicht an Unterklassen vererbt werden.
- Unterklassen nicht unabhängig von Oberklassen testbar, da in der Klassenhierarchie Verweise nach oben existieren.
  - Beispiel: Klasse A erbt von Klasse B, Klasse B erbt von Klasse C, Test von A bezieht daher B und C im Test mit ein.
  - Hierarchien schwer testbar, stellen hohe Anforderungen an den Testrahmen.

Struktur des Quelltexts spiegelt nicht den Kontrollfluss wider, da Ablauflogik über Vererbungshierarchie verteilt ist.

- Erhöhte Komplexität, obwohl strukturelle Programmierung das Gegenteil zu erreichen sucht!

Folgen der Vererbung

Mehrfachvererbung:

- Klasse erbt von zwei oder mehr Oberklassen
- Nachteile
  - Vererbung gleichnamiger Methoden
    - Klasse C erbt von Klassen A und B. C nutzt zunächst Methode A.m, wird dann aber mit B.m realisiert.
    - Nach dieser Änderung muss C erneut getestet werden.
    - Verschiedene Testfälle für A.m und B.m notwendig.

Folgen der Polymorphie

Polymorphismus:

- Dieselbe Botschaft wird an Objekte verschiedener Klassen einer Vererbungshierarchie gesendet; die Objekte können diese Botschaft unterschiedlich interpretieren

Probleme:

- Programmablauf nicht aus dem Quellcode ableitbar
- Alle dynamischen Abläufe müssen getestet werden
- Wird Polymorphie mehrfach wiederholt, explodiert die mögliche Anzahl der Ablaufpfade

Klassentest

Unterschiede zwischen Klassentest und Modultest:

- Module besitzen größere interne Komplexität.
- Klassen besitzen größere externe Abhängigkeit.
- Methoden einer Klasse sind leichter zu erreichen und ihre Schnittstellen leichter zu bedienen.
  - Aber: Umso schwieriger, Klasse getrennt von ihrer Umgebung zu testen.
- Instanziierung der Objektdaten mit unterschiedlichen Zuständen erschwert den Test.

Zweck des Klassentests:

- Ermöglichen dem Entwickler, eigene Fehler zu finden.
- Rückkopplung über Lauffähigkeit und Korrektheit des Codes.
- Allerdings: Generell nur ein Drittel der potentiellen Fehler erkannt, nämlich Codier- und Logikfehler.
- Aber: Klassentest unabdingbar, da Probleme sonst nur hinausgeschoben werden.

Klassentest

Unterschiedliche Klassenarten:

- Nonmodale Klassen:
  - Keine Abhängigkeiten zwischen den Methoden, d.h. können in beliebiger Reihenfolge aufgerufen werden.
  - Zustand der Objekte hat keinen Einfluss auf Verhalten der Methoden.
  - Am einfachsten zu testen.
- Unimodale Klassen:
  - Methoden müssen in gewisser Reihenfolge ausgeführt werden.
  - Zustand ist Voraussetzung für Ausführung der nächsten Methode (z.B. Konto eröffnen und dann einzahlen).
- Quasimodale Klassen:
  - Sind vom Zustand der Objekte abhängig, d.h. Ergebnis der Methodenausführung wird je nach Zustand anders ausfallen (z.B. werden bei Kontosperre alle Kontobewegungen abgelehnt).
- Modale Klassen:
  - Sind sowohl von der Reihenfolge der Methodenaufrufe als auch vom Zustand der Objekte abhängig (Abbuchung kann nur auf Einzahlung folgen, bei einem Kontostand größer als der Abbuchungsbetrag).
  - Am schwierigsten zu testen.

Klassentest

Klassentest und Vererbung:

- Erbende Klassen können nicht alleine, sondern müssen mit den Oberklassen getestet werden.
- Testreihenfolge:
  - Klassen an der Spitze der Klassenhierarchie, d.h. die Wurzeln.
  - Dann die jeweils nächste Schicht von abgeleiteten Klassen bis hin zu den Blättern.
- Es reicht nicht, bei abgeleiteten Klassen nur jeweils die neu hinzugekommenen Methoden zu testen, da die geerbten Methoden unter geänderten Umgebungsbedingungen ausgeführt werden („Class flattening“).

Klassentest und Polymorphie:

- Dynamische Bindung erschwert den Klassentest, da Ausprägung der Methode erst zu Laufzeit bestimmt wird.
- Daher: alle Ausprägungen sind zu testen oder Stichproben.
- Achtung: bei verschachtelter Polymorphie Explosion von Testfällen.

Klassentest

Klassentest und Überladen von Parametern:

- Hoher Testaufwand, da alle potentiellen Parameterkombinationen getestet werden müssen.
- Generierung eines Testtreibers erschwert, da der Testtreiber nur eine Standard-Parameterliste kennt.

Klassentest und Wiederverwendung:

- Fremde Klassen sind durch Stubs zu simulieren oder mitzutesten.
- Beim Mittesten müssen Rückgabewerte kontrolliert werden, damit durch fehlerhafte Werte nicht die Ergebnisse der Klasse verfälscht werden.

Klassentest

Klassentestarten:

- Implementierungsbezogen:
  - Ausgehend vom Quellcode werden Methoden, Verzweigungen, Operationsaufrufe und Parameter identifiziert.
  - Test wird so konstruiert, dass:
    - die Methoden mit allen Parameterkombinationen aufgerufen,
    - alle Verzweigungen durchlaufen,
    - alle Operationsaufrufe betätigt werden.
- Spezifikationsbezogen:
  - Klasse zweimal schreiben: Mit formaler Spezifikationssprache und mit Implementierungssprache.
  - Beide Repräsentationen dann miteinander vergleichen:
    - Statisch: Signaturen, Bedingungen, …
    - Dynamisch: Ableitung modellhafter Ablaufpfade und Vergleich mit tatsächlichen Ablaufpfaden

Klassentest

Praktische Ansätze zum Klassentest:

- Klassentesttreiber:
  - Kennt Operationsschnittstellen der zu testenden Klasse.
  - Testet alle Operationen.
  - Parameterwerte werden manuell eingegeben oder per Skript festgelegt und von dort ausgelesen.
  - Ergebnisse werden angezeigt oder in Datei ausgegeben.
  - Ist-Werte werden manuell oder automatisch übeprüft.
- Build-In Tests:
  - Tests werden direkt in die Klasse miteingebaut.
  - Pro Klasse gibt es dann ein oder mehrere Testoperationen, von wo aus die „echten“ Methoden aufgerufen werden.
  - Vorteile:
    - Testoperationen haben Zugang zu den Objektzuständen und können diese manipulieren.
    - Statt zweier Dateien (Quellcode, Testtreiber) nur eine Datei fortzuschreiben.
  - Nachteil:
    - Sourcecode wird aufgebläht, Testoperationen per Compiler-Option abschalten.

Klassentest

Praktische Ansätze zum Klassentest:

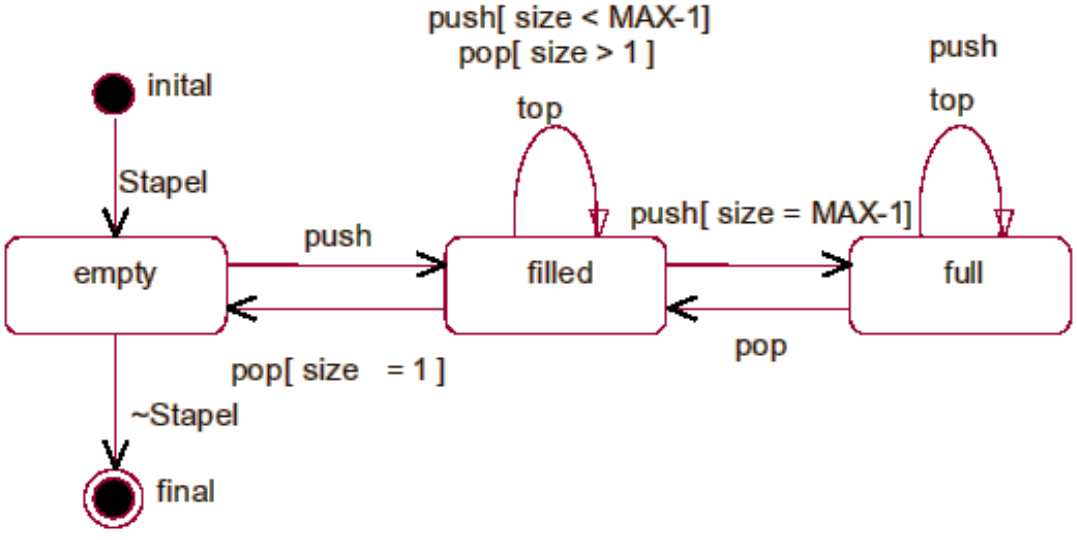
- Zusicherungstest (insbesondere bei Eiffel):
  - Voraussetzung: Vorliegen von Zusicherungen (constraints, assertions) für jede Operation der zu testenden Klasse:
    - Vorbedingung z.B. Alter>29
    - Nachbedingung z.B. Gehalt = old.Gehalt + old.Gehalt * Prozentsteigerung
    - Invariante z.B.: Alter => 18 and Alter <= 60
  - Testfallerzeugung aus den Vorbedingungen (z.B. per minimaler Mehrfachbedingungsüberdeckung).
  - Invarianten müssen vor und nach jeder Operationsausführung erfüllt sein.
  - Vergleich der Ergebnisse mit den Nachbedingungen.
  - Zusätzlich: Test von Ausnahmesituationen durch gezielte Verletzung der Vorbedingungen einer Operation.

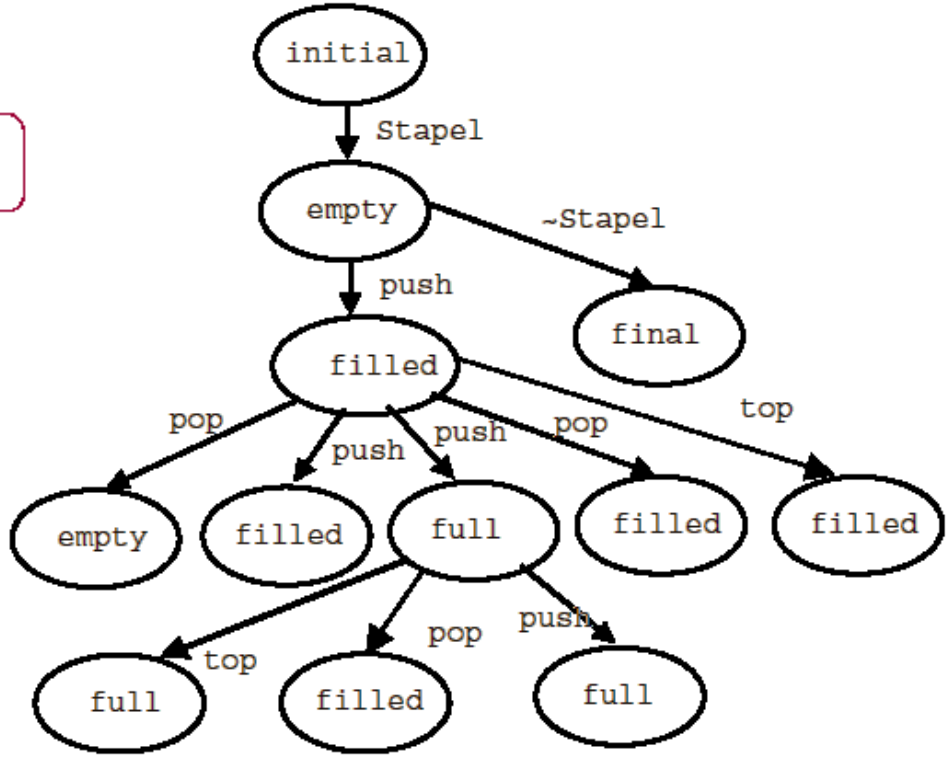
Praktische Ansätze zum Klassentest:

- Zustandstest:
  - Test, ob Klasse konform zu dem Zustandsdiagramm ist, welches das zustandsabhängige Verhalten der Klasse beschreibt.
  - Ableitung eines Übergangsbaums vom Zustandsdiagramm, der bestimmte Folgen von Zustandswechseln repräsentiert:
    - Schritt 1: Anfangszustand: Wurzel des Baums.
    - Schritt 2: Für jeden möglichen Übergang vom Anfangszustand zu einem Folgezustand im Zustandsdiagramm erhält der Baum von der Wurzel aus eine Verzweigung zu einem Knoten, der den Nachfolgezustand repräsentiert.
    - Schritt 3: Wiederhole Schritt 2 für jedes Blatt, bis eine der folgenden Bedingungen eintritt:
      - Der dem Blatt entsprechende Zustand ist auf dem Weg von der Wurzel zum Blatt bereits einmal im Baum enthalten.
      - Der dem Blatt entsprechende Zustand ist ein Endzustand.
  - Ggf. Erweiterung des Übergangsbaums um Spezifikationsverletzungen.
  - Testfälle nach Pfaden von der Wurzel zu den Blättern bilden.

Praktische Ansätze zum Klassentest

- Zustandstest: Beispiel





Klassentest-Werkzeug: JUnit

JUnit - Open Source Java Test-Framework zur Erstellung und Durchführung wiederholbarer Tests

- Erdacht von Erich Gamma und Kent Beck
- Beispiel: Rechnen mit Währungen

Klasse Money repräsentiert Betrag in einer bestimmten Währung

Methode add addiert Beträge derselben Währung

```
class Money {
    private int fAmount;
    private String fCurrency;

    public Money(int amount, String currency) {
        fAmount= amount;
        fCurrency = currency;
    }

    public int amount() {
        return fAmount;
    }

    public String currency() {
        return fCurrency;
    }

    public Money add(Money m) {
        return new Money(amount()+m.
    }
}
```

Klassentest-Werkzeug: JUnit

- Definition von Testklassen (hier: MoneyTest) für jede Klasse
- Definition von Testmethoden (hier: testSimpleAdd) zum Test der Klassenmethoden

```
public class MoneyTest extends TestCase {
    //...
    public void testSimpleAdd() {
        Money m12CHF = new Money(12, "CHF");
        Money m14CHF = new Money(14, "CHF");
        Money expected = new Money(26, "CHF");
        Money result = m12CHF.add(m14CHF);
        Assert.assertTrue(expected.equals(result));
    }
}
```

Erzeugung von Testobjekten

Vergleich mit erwartetem Ergebnis

Klassentest-Werkzeug: JUnit

- Ausführung der Tests.
- Statisch: Methode runTest wird mit der Testmethode überschrieben:
  - Benamung („simple add“) zum Zwecke der Identifikation.
- Dynamisch: Dynamischer Aufruf der Testmethode:
  - Quellcode kompakter.
  - Fehler in der Benamung fällt allerdings erst zur Compilezeit auf.

```
TestCase test= new MoneyTest("simple add") {
    public void runTest() {
        testSimpleAdd();
    }
};

TestCase test= new TestSuite().createTest(
```

Statische Ausf

Dynamische A

Klassentest-Werkzeug: JUnit

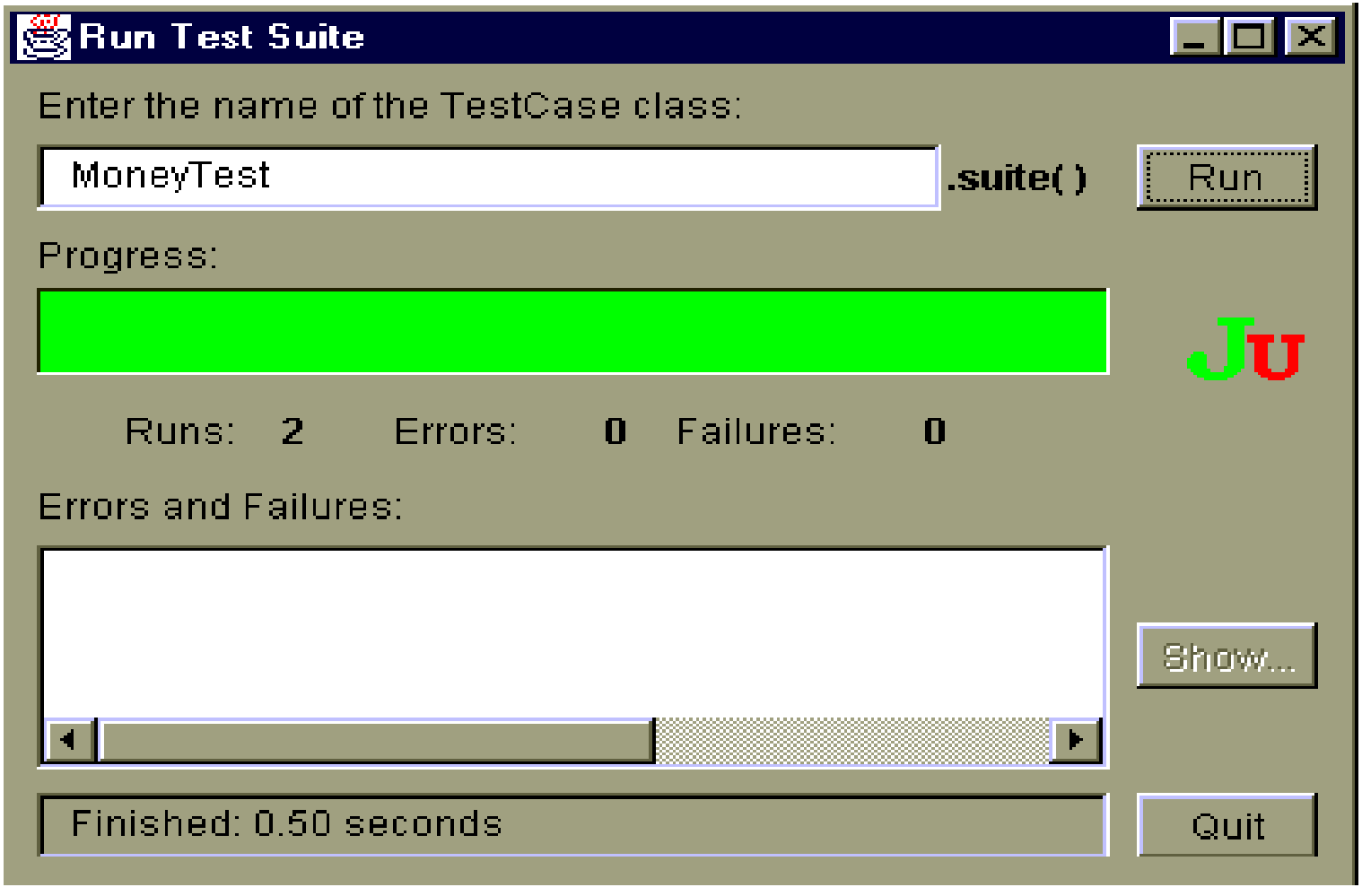
- Definition einer TestSuite zur Ausführung einer Menge von Testfällen:
  - Statisch: Hinzufügen der Testmethoden zur TestSuite.
  - Dynamisch: Testmethoden werden automatisch aus zu testender Klasse extrahiert.

```
public static Test suite() {
    TestSuite suite = new TestSuite();

    suite.addTest(
        new MoneyTest("simple add") {
            protected void runTest() { testSimpleAdd(); }
        }
    );

    ....
    return suite;
}

public static Test suite() {
    return new TestSuite(MoneyTest.class);
}
```

Statische Ausfü

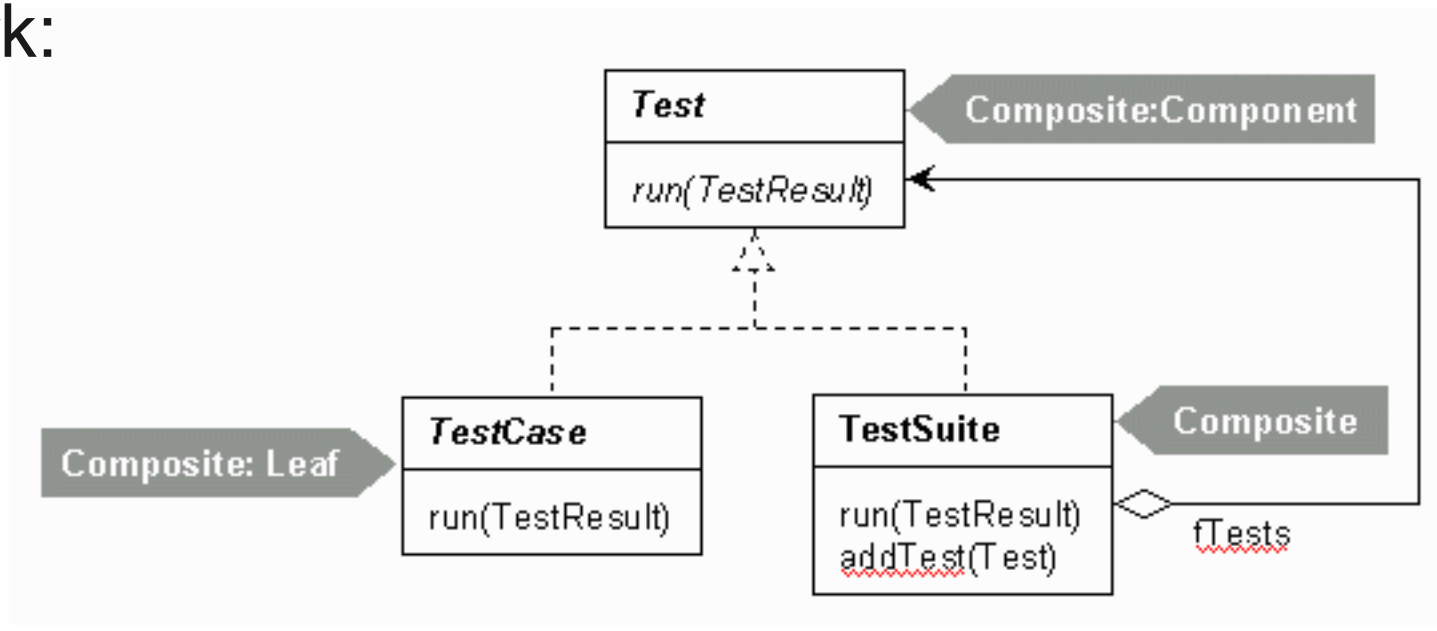
Dynamische Au

Ausführung der Tests mit der Junit-GUI



Klassentest-Werkzeug: JUnit

- Framework:



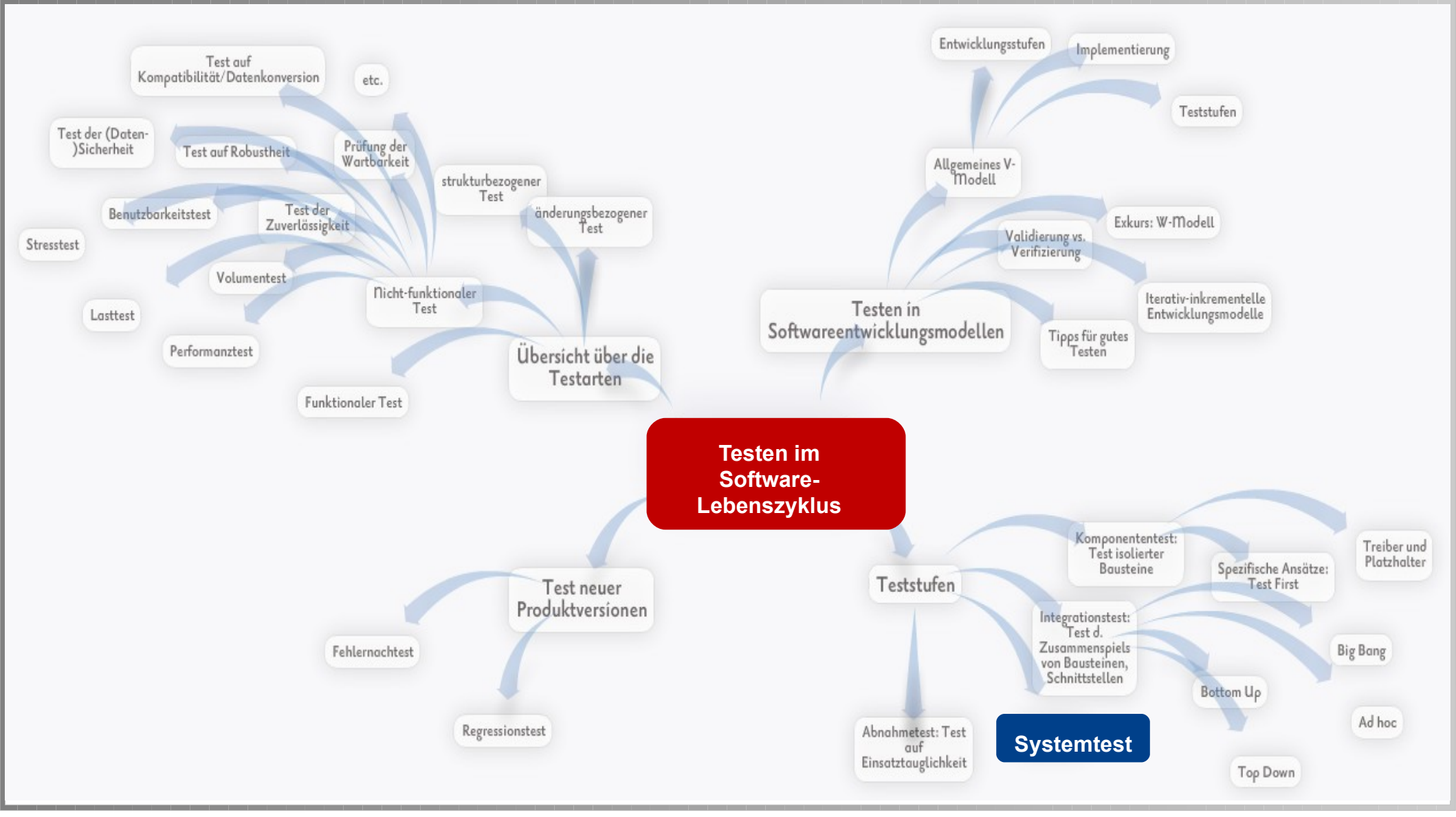
- Vorgehen:
  - Testmethoden immer gleichzeitig (oder sogar zuerst) mit richtigen Methoden schreiben.
  - Text fixtures (Objekte, die von mehreren Testmethoden verwendet werden) zur Codeeinsparung verwenden.

2.2 Testen im Software-Lebenszyklus

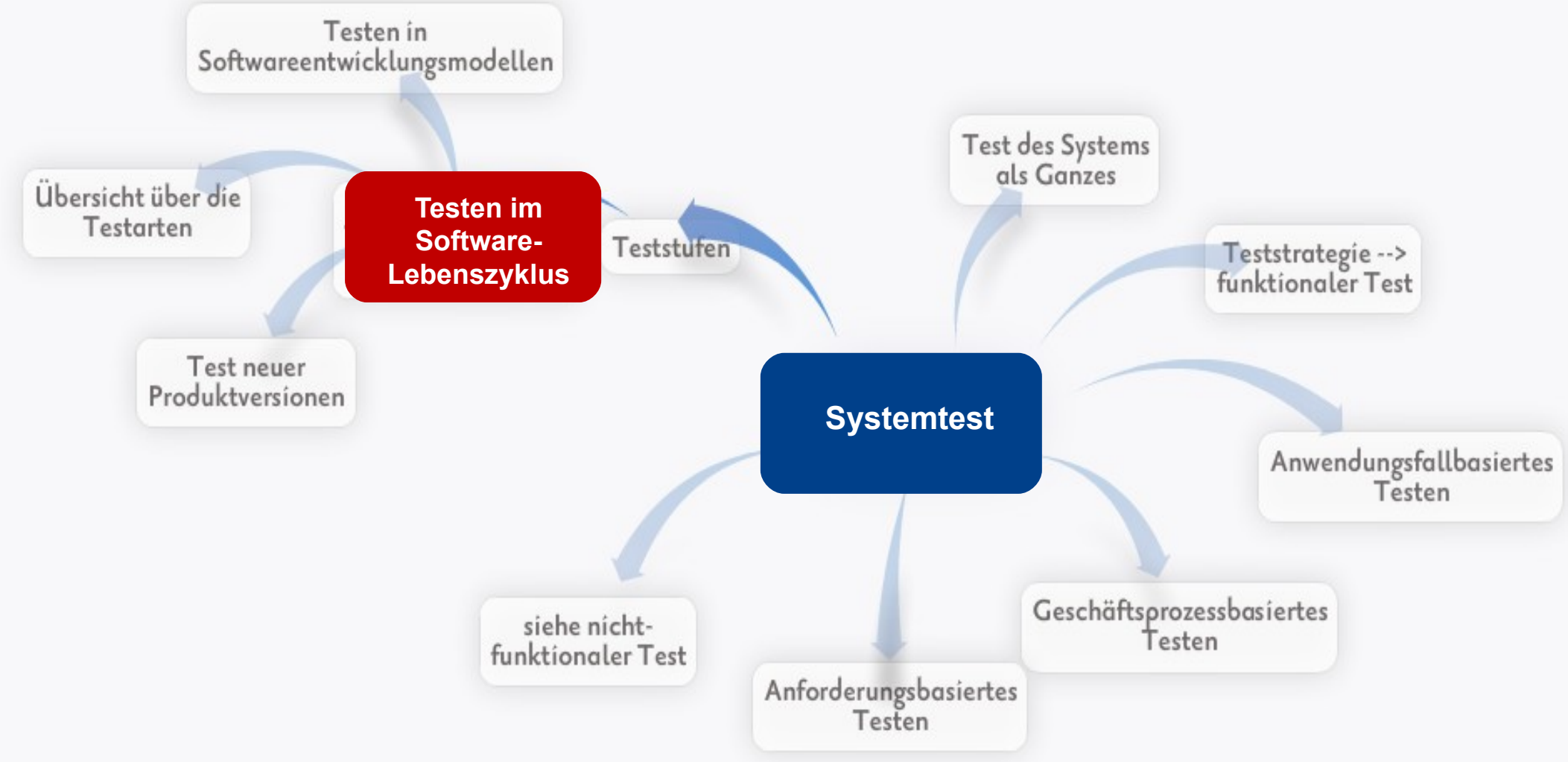
2.2 Testen im Software-Lebenszyklus

- Testen in Softwareentwicklungsmodellen
- Teststufen → Systemtest
- Test neuer Produktversionen
- Übersicht über die Testarten

Big Picture


Testen im Software-Lebenszyklus
Übersicht über die Testarten
Nicht-funktionaler Test
Test auf Kompatibilität/Datenkonversion
etc.
Test der (Daten-)Sicherheit
Test auf Robustheit
Prüfung der Wartbarkeit
strukturbezogener Test
Benutzbarkeitstest
Test der Zuverlässigkeit
änderungsbezogener Test
Stresstest
Volumentest
Lasttest
Performanztest
Funktionaler Test
Testen in Softwareentwicklungsmodellen
Entwicklungsstufen
Implementierung
Teststufen
Allgemeines V-Modell
Exkurs: W-Modell
Validierung vs. Verifizierung
Iterativ-inkrementelle Entwicklungsmodelle
Tipps für gutes Testen
Test neuer Produktversionen
Fehlernachtest
Regressionstest
Teststufen
Komponententest: Test isolierter Bausteine
Treiber und Platzhalter
Spezifische Ansätze: Test First
Integrationstest: Test d. Zusammenspiels von Bausteinen, Schnittstellen
Big Bang
Bottom Up
Ad hoc
Top Down
Abnahmetest: Test auf Einsatztauglichkeit
Systemtest

Big Picture


Testen in Softwareentwicklungsmodellen
Übersicht über die Testarten
Testen im Software-Lebenszyklus
Teststufen
Test neuer Produktversionen
Test des Systems als Ganzes
Teststrategie --> funktionaler Test
Systemtest
Anwendungsfallbasiertes Testen
siehe nicht-funktionaler Test
Geschäftsprozessbasiertes Testen
Anforderungsbasiertes Testen

Systemtest - Begriffsklärung

Nach abgeschlossenem Integrationstest folgt vor Auslieferung des Systems der Systemtest (dritte Teststufe).

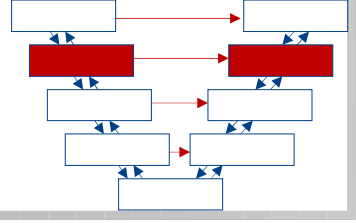
Ein **Systemtest** überprüft, ob die spezifizierten Anforderungen vom Produkt erfüllt werden.

Gründe für den Systemtest:

- In den niedrigeren Teststufen wurde gegen technische Spezifikationen geprüft, aus der Perspektive des Softwareherstellers. Der Systemtest betrachtet das System hingegen aus der Perspektive des Kunden und des späteren Anwenders. Die Tester validieren, ob die Anforderungen vollständig und angemessen umgesetzt wurden.
- Viele Funktionen und Systemeigenschaften resultieren aus dem Ineinandergreifen aller Systemkomponenten und sind
  somit erst auf Ebene des Gesamtsystems beobachtbar
  und testbar.

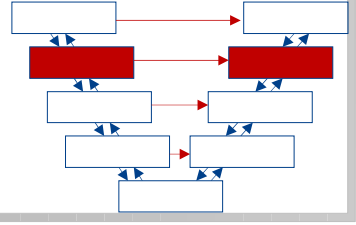
Testobjekte:

- Mit abgeschlossenem Integrationstest liegt das komplett zusammengebaute Softwaresystem vor.
- Im Systemtest wird dieses System als Ganzes betrachtet.



Systemtest - Testumgebung (1)

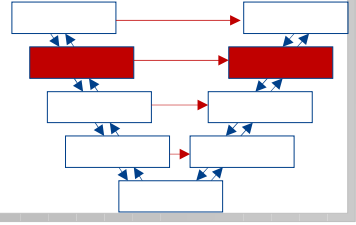
- Testumgebung soll der späteren Produktivumgebung möglichst nahe kommen.
- Statt Treibern und Platzhaltern sollen auf allen Ebenen möglichst die später tatsächlich zum Einsatz kommenden Hard- oder Softwareprodukte in der Testumgebung installiert sein (Hardwareausstattung, Systemsoftware, Treibersoftware, Netzwerk, Fremdsysteme usw.).



Systemtest - Testumgebung (2)

Um Kosten und Aufwand zu sparen, wird oft der Systemtest statt in einer separaten Testumgebung in der Produktivumgebung des Kunden durchgeführt.

Aus welchen Gründen könnte das schädlich sein ?

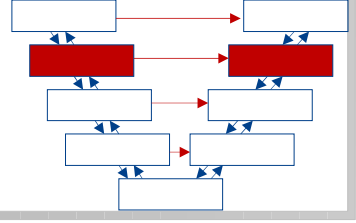


Systemtest - Testumgebung (2)

Um Kosten und Aufwand zu sparen, wird oft der Systemtest statt in einer separaten Testumgebung in der Produktivumgebung des Kunden durchgeführt.

Dies ist aus folgenden Gründen schädlich:

- Im Systemtest werden Fehlerwirkungen auftreten. Dabei besteht immer die Gefahr, dass die Produktivumgebung des Kunden beeinträchtigt wird. Teure Systemausfälle und Datenverluste im produktiven Kundensystem können die Folge sein.
- Die Tester haben keine oder nur geringe Kontrolle über Parameter und Konfiguration der Produktivumgebung. Durch den gleichzeitig zum Test weiterlaufenden Betrieb der anderen Kundensysteme werden die Testbedingungen unter Umständen schleichend verändert. Die durchgeführten Systemtests sind schwer oder nicht mehr reproduzierbar.



Systemtest - Testziele

Wie gut erfüllt das fertige System die gestellten Anforderungen ?

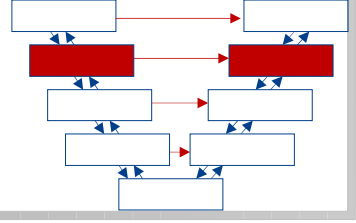
Zwei Klassen von Anforderungen sind zu unterscheiden:

Funktionale Anforderungen...

- … spezifizieren das Verhalten, welches das System oder Systemteile erbringen müssen. Sie beschreiben »was« das (Teil)System leisten soll. Ihre Umsetzung ist Voraussetzung dafür, dass das System überhaupt einsetzbar ist.
- Merkmale der Funktionalität nach [ISO 9126] sind: Angemessenheit, Richtigkeit, Interoperabilität, Ordnungsmäßigkeit, Sicherheit.

Nicht-funktionale Anforderungen...

- … beschreiben Attribute des funktionalen Verhaltens, also »wie gut« bzw. mit welcher Qualität das (Teil)System seine Funktion erbringen soll. Ihre Umsetzung beeinflusst stark, wie zufrieden der Kunde bzw. Anwender mit dem Produkt ist und wie gerne er es einsetzt.
- Merkmale nach [ISO 9126] sind: Zuverlässigkeit, Benutzbarkeit, Effizienz. Indirekt haben auch die Änderbarkeit und Übertragbarkeit Einfluss auf die Kundenzufriedenheit.
- Bei manchen Projekten stehen die Daten im Mittelpunkt (z.B. Datenkonvertierungsprojekte und Anwendungen wie Data Warehouses). Anforderungen an Datenqualität müssen im Systemtest ebenfalls berücksichtigt werden



Systemtest - Teststrategie Funktionale Anforderungen (1)

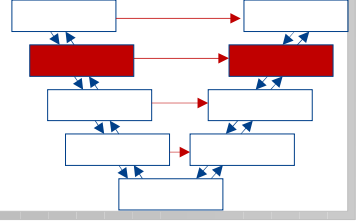
Funktionale Anforderungen werden in der Projektphase *Anforderungsdefinition* gesammelt und in einem Anforderungsdokument (oft auch als Lasten- oder Pflichtenheft bezeichnet) dokumentiert.

Anforderungsbasiertes Testen:

- Das freigegebene Anforderungsdokument wird als Testbasis herangezogen.
- Auch die Systemtestspezifikation wird durch ein Review verifiziert.
- Zu jeder Anforderung (funktional aber auch nicht-funktional) wird mindestens ein Systemtestfall abgeleitet und in der Systemtestspezifikation dokumentiert.
- In der Regel wird mehr als nur ein Testfall benötigt, um eine Anforderung zu testen.

Anwendungsfallbasiertes Testen:

- Systemtestfälle ableiten, wie der Anwender mit dem System umgeht und welche Aktionen er dann typischerweise ausführt.
- Verschiedene Anwendergruppen besitzen jeweils ihre eigenen Benutzerprofile. Für sie lassen sich typische Aktionsmuster oder Anwendungsfälle (*use cases*) in typischer Häufigkeit identifizieren.
- Aus diesen Aktionsmustern lassen sich wieder Testszenarien ableiten.
- Anhand der Häufigkeit, mit der die entsprechenden Aktionen im späteren Betrieb der Software ausgelöst werden, ermittelt der Tester, wie wichtig das zugehörige Testszenario ist und mit welcher Priorität es deshalb im Testplan aufgenommen werden sollte.

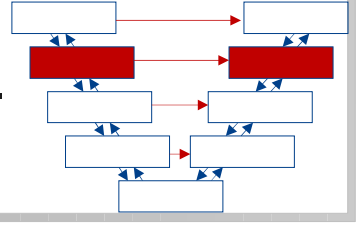


Systemtest - Teststrategie Funktionale Anforderungen (2)

Geschäftsprozessbasiertes Testen:

- Situation: Ein Softwaresystem hat den Zweck, einen bestimmten Geschäftsprozess des Kunden zu automatisieren oder zu unterstützen.
- Eine Geschäftsprozessanalyse (meist Teil der Anforderungsanalyse) zeigt, welche Geschäftsprozesse relevant sind, wie häufig und in welchem Kontext sie auftreten, welche Personen, Firmen, Fremdsysteme daran beteiligt sind usw.
- Anschließend werden auf Grundlage dieser Analyse als Testbasis Testszenarien (Aneinanderreihung von Testfällen) aufgestellt, die typische Geschäftsvorfälle nachbilden.
- Die Priorität der Testszenarien richtet sich nach Häufigkeit und Relevanz der entsprechenden Geschäftsprozesse.

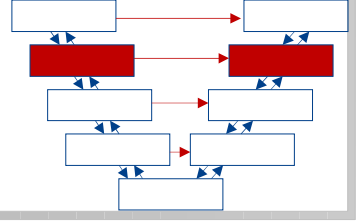
Während beim anforderungsbasierten Testen einzelne Systemfunktionen im Blickpunkt stehen, sind dies beim geschäftsprozessbasierten Test Abläufe, also hintereinander geschaltete Tests.



Systemtest - Teststrategie
Nicht-funktionale Anforderungen (1)

Genauso wichtig und berechtigt wie funktionale Anforderungen sind nicht-funktionale Anforderungen, denn sie legen wichtige qualitative Aspekte der Systemleistung fest. Folgende nicht-funktionale Systemeigenschaften in entsprechenden Tests (in der Regel im Systemtest) sollen berücksichtigt werden:

- **Lasttest:** Messung des Systemverhaltens in Abhängigkeit steigender Systemlast (z.B. Anzahl parallel arbeitender Anwender, Anzahl Transaktionen).
- **Performanztest:** Messung der Verarbeitungsgeschwindigkeit bzw. Antwortzeit für bestimmte Anwendungsfälle, in der Regel in Abhängigkeit steigender Last.
- **Volumen- / Massentest:** Beobachtung des Systemverhaltens in Abhängigkeit zur Datenmenge (z. B. Verarbeitung sehr großer Dateien).
- **Stresstest:** Beobachtung des Systemverhaltens bei Überlastung.
- **Test der (Daten-) Sicherheit:** gegen unberechtigten Systemzugang oder Datenzugriff.
- **Test der Zuverlässigkeit:** im Dauerbetrieb (z. B. Ausfälle pro Betriebsstunde bei gegebenem Benutzungsprofil).
- **Test auf Robustheit:** gegenüber Fehlbedienung, Fehlprogrammierung, Hardwareausfall usw. sowie Prüfung der Fehlerbehandlung und des Wiederanlaufverhaltens (*recovery*).
- **Prüfung auf Änderbarkeit / Wartbarkeit:** Verständlichkeit und Aktualität der Entwicklungsdokumente; modulare Systemstruktur usw.

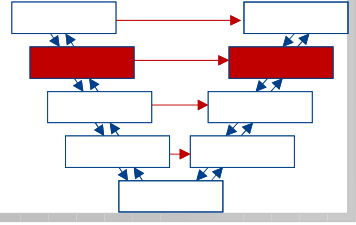


Systemtest - Teststrategie
Nicht-funktionale Anforderungen (2)

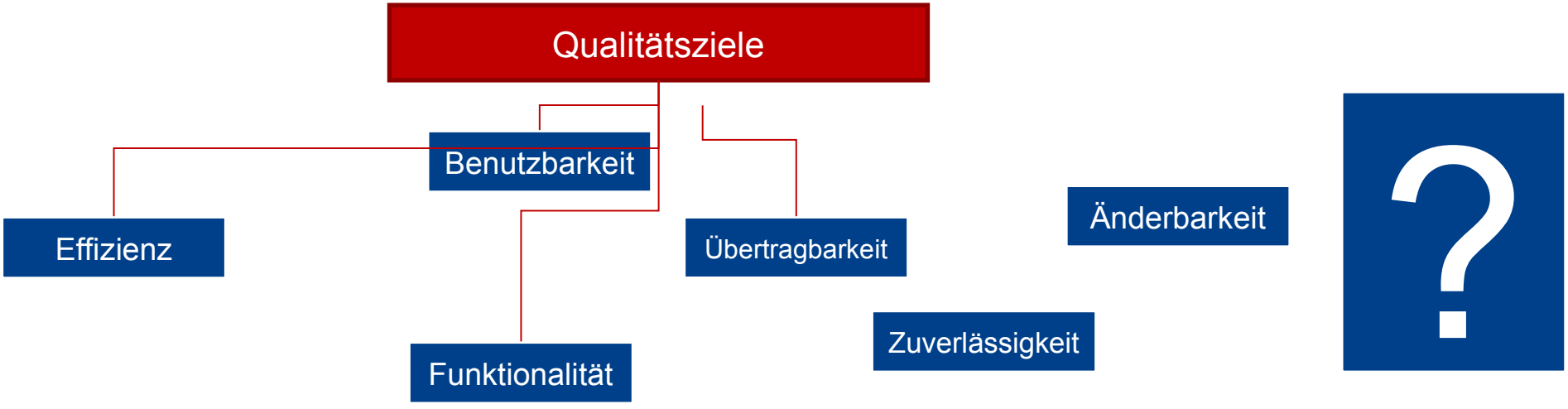
- **Test auf Kompatibilität / Datenkonversion:** Prüfung der Verträglichkeit mit vorhandenen Systemen. Import/Export von Datenbeständen usw.
- **Test unterschiedlicher Konfigurationen:** des Systems, z. B. unterschiedliche Betriebssystemversion, Landessprache, Hardwareplattform usw.
- **Test auf Benutzungsfreundlichkeit / Benutzbarkeit:** Prüfung der Angemessenheit der Bedienung; Verständlichkeit der Systemausgaben usw., jeweils bezogen auf die Bedürfnisse einer bestimmten Anwendergruppe.
- **Prüfung der Dokumentation:** auf Übereinstimmung mit dem Systemverhalten (z.B. Bedienungsanleitung und GUI).

Beim Test nicht-funktionaler Anforderungen stellt sich meist das Problem, dass diese lückenhaft und »schwammig« formuliert sind.

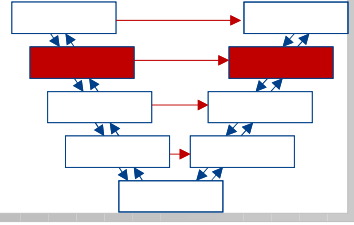
- Formulierungen wie »Das System soll leicht bedienbar sein« oder »schnell reagieren« sind in dieser Form nicht testbar.
- Des Weiteren gelten viele nicht-funktionale Anforderungen als derart selbstverständlich, dass sie nicht im Anforderungsdokument erwähnt werden (vorausgesetzte Anforderungen).
- Auch solche nicht spezifizierten, aber dennoch relevanten Eigenschaften gilt es zu validieren.



Systemtest - Teststrategie
Nicht-funktionale Anforderungen (3)



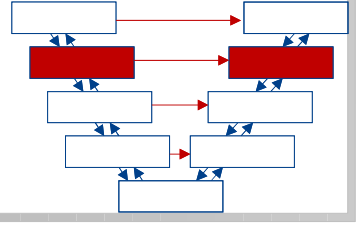
Beispiele: • Funktionaler Test • Test auf (Daten-) Sicherheit • Test auf Daten-kompatibilität / Datenkonver-sion	Beispiele: • Test auf Robustheit • Test auf Wiederherstell-barkeit	Beispiele: • Benutzbar-keitstest • Prüfung der Benutzer-dokumentation inkl. Online-Hilfe	Beispiele: • Lasttest • Performanz-test • Volumentest • Stresstest	Beispiele: • Prüfung auf Wartbarkeit (Code-Review, Architektur-Review) • Prüfung der Dokumentation • Statische Analyse	Beispiele: • Test unterschied-licher Konfigur-ationen



Systemtest - Teststrategie
Nicht-funktionale Anforderungen (3)

Qualitätsziele

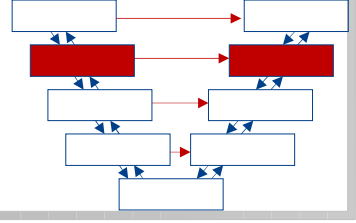
Funktionalität	Zuverlässigkeit	Benutzbarkeit	Effizienz	Änderbarkeit	Übertragbarkeit
• Angemessenheit • Richtigkeit • Interoperabilität • Sicherheit • Ordnungsmäßigkeit	• Reife • Fehlertoleranz • Wiederherstellbarkeit	• Verständlichkeit • Erlernbarkeit • Bedienbarkeit • Attraktivität	• Zeitverhalten • Verbrauchsverhalten	• Analysierbarkeit • Modifizierbarkeit • Stabilität • Testbarkeit	• Anpassbarkeit • Installierbarkeit • Koexistenz • Austauschbarkeit
Beispiele: • Funktionaler Test • Test auf (Daten-) Sicherheit • Test auf Datenkompatibilität / Datenkonversion	Beispiele: • Test auf Robustheit • Test auf Wiederherstellbarkeit	Beispiele: • Benutzbarkeitstest • Prüfung der Benutzerdokumentation inkl. Online-Hilfe	Beispiele: • Lasttest • Performanztest • Volumentest • Stresstest	Beispiele: • Prüfung auf Wartbarkeit (Code-Review, Architektur-Review) • Prüfung der Dokumentation • Statische Analyse	Beispiele: • Test unterschiedlicher Konfigurationen



Systemtest - Probleme (1)

Unklare Kundenanforderungen:

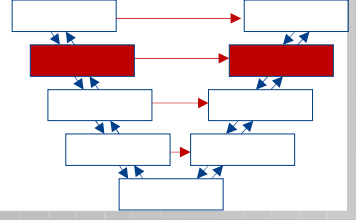
- Wo keine Anforderungen existieren, ist zunächst jedes Systemverhalten zulässig bzw. nicht bewertbar.
- Natürlich wird der Anwender oder Kunde eine gewisse Vorstellung davon haben, was er von »seinem« Softwaresystem erwartet. Es existieren also sehr wohl Anforderungen.
- Nur sind diese eben nirgends nachlesbar, sondern sie sind nur »in den Köpfen« einiger am Projekt beteiligten Personen vorhanden.
- Den Testern fällt dann die undankbare Rolle zu, alle diese Informationen über das gewünschte Soll-Verhalten nachträglich zusammenzutragen.



Systemtest - Probleme (2)

Versäumte Entscheidungen

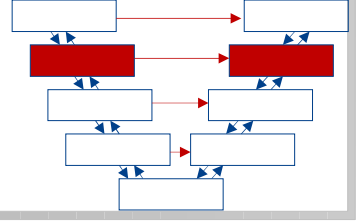
- Wenn die Tester die ursprünglichen Anforderungen identifizieren, werden sie feststellen, dass in den Köpfen der verschiedenen Personen zu ein und derselben Sache ganz unterschiedliche Ansichten und Vorstellungen existieren.
- Da im Projekt versäumt wurde, die Anforderungen schriftlich zu dokumentieren, abzustimmen und freizugeben, ist dies nicht weiter verwunderlich.
- Der Systemtest muss also nicht nur Anforderungen zusammen sammeln, sondern auch noch Klärungs- und Entscheidungsprozesse, die viele Monate unterblieben sind, zu einem eigentlich viel zu späten Zeitpunkt erzwingen.
- Dieses Zusammentragen der Informationen ist sehr zeit- und kostenintensiv. Test und Fertigstellung des Systems werden verzögert.



Systemtest - Probleme (3)

Projekte scheitern

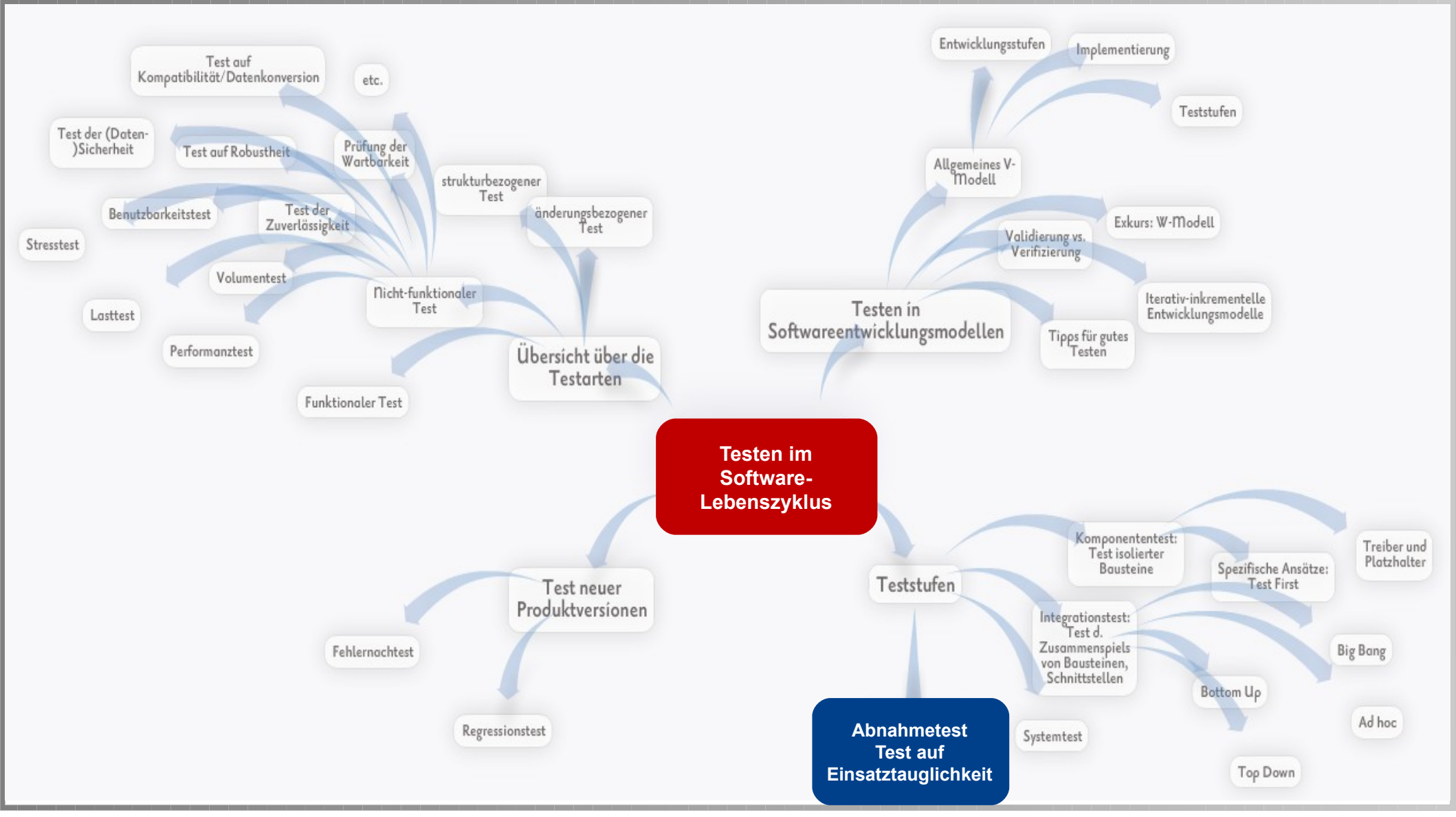
- Wenn Anforderungen nicht dokumentiert sind, fehlen natürlich auch den Entwicklern klare Ziele. Die Wahrscheinlichkeit, dass das konstruierte System die impliziten Kundenanforderungen erfüllt, ist deshalb außerordentlich gering.
- Niemand kann ernsthaft hoffen, dass unter solchen Projekt-bedingungen ein auch nur halbwegs brauchbares System entsteht.
- In derart gelagerten Projekten kann der Systemtest oftmals nur das Scheitern des Projekts »offiziell« attestieren.



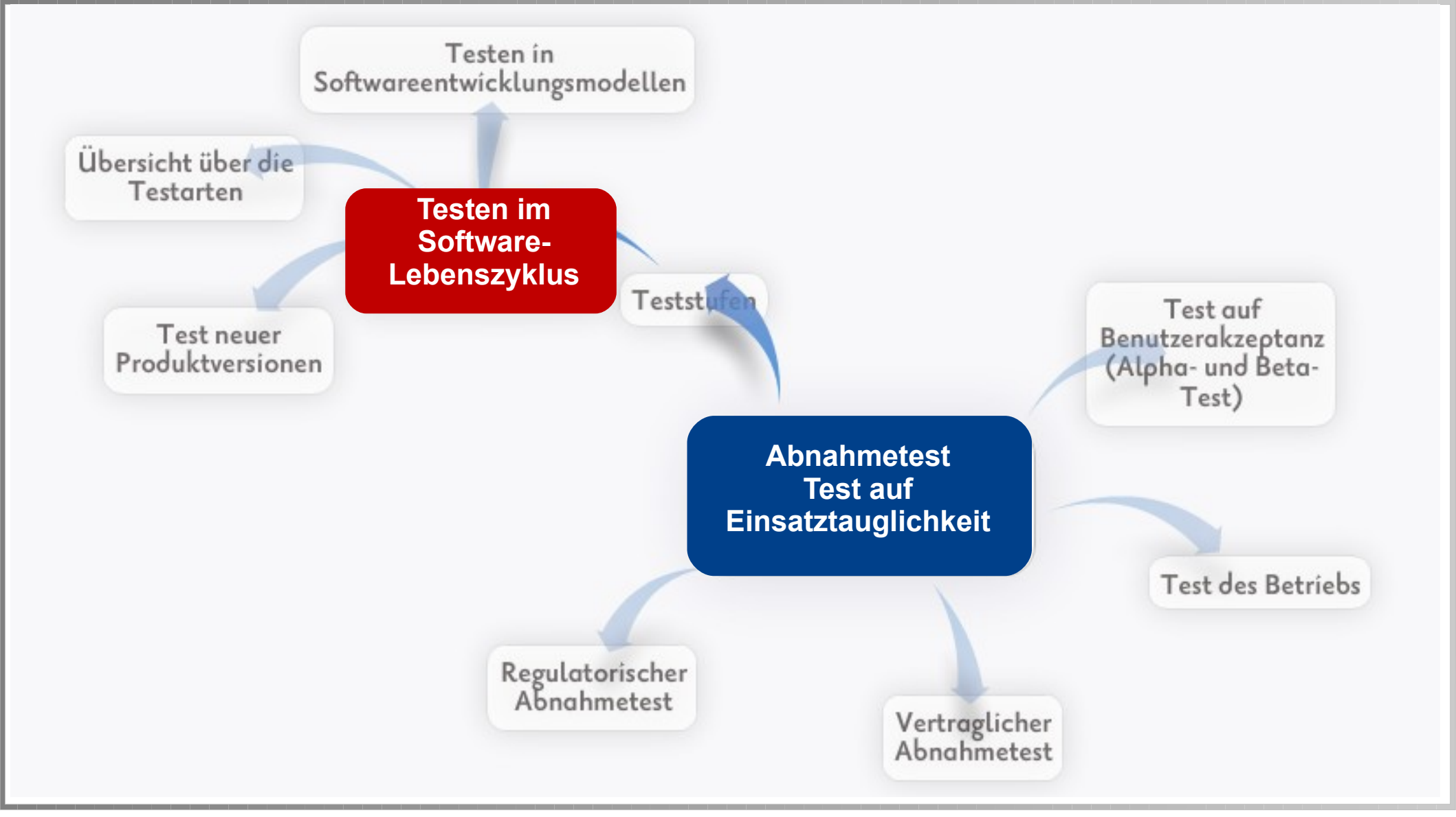
2.2 Testen im Software-Lebenszyklus

- Testen in Softwareentwicklungsmodellen
- **Teststufen → Abnahmetest**
- Test neuer Produktversionen
- Übersicht über die Testarten

Big Picture

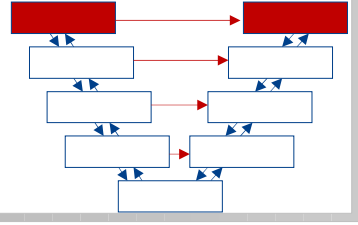

Testen im Software-Lebenszyklus
Übersicht über die Testarten
Nicht-funktionaler Test
Test auf Kompatibilität/Datenkonversion
etc.
Test der (Daten-)Sicherheit
Test auf Robustheit
Prüfung der Wartbarkeit
strukturbezogener Test
Benutzbarkeitstest
Test der Zuverlässigkeit
änderungsbezogener Test
Stresstest
Volumentest
Lasttest
Performanztest
Funktionaler Test
Testen in Softwareentwicklungsmodellen
Entwicklungsstufen
Implementierung
Teststufen
Allgemeines V-Modell
Exkurs: W-Modell
Validierung vs. Verifizierung
Iterativ-inkrementelle Entwicklungsmodelle
Tipps für gutes Testen
Test neuer Produktversionen
Fehlernachtest
Regressionstest
Teststufen
Komponententest: Test isolierter Bausteine
Treiber und Platzhalter
Spezifische Ansätze: Test First
Integrationstest: Test d. Zusammenspiels von Bausteinen, Schnittstellen
Big Bang
Bottom Up
Ad hoc
Top Down
Systemtest
Abnahmetest Test auf Einsatztauglichkeit

Big Picture


Testen in Softwareentwicklungsmodellen
Übersicht über die Testarten
Testen im Software-Lebenszyklus
Teststufen
Test neuer Produktversionen
Test auf Benutzerakzeptanz (Alpha- und Beta-Test)
Abnahmetest Test auf Einsatztauglichkeit
Test des Betriebs
Regulatorischer Abnahmetest
Vertraglicher Abnahmetest

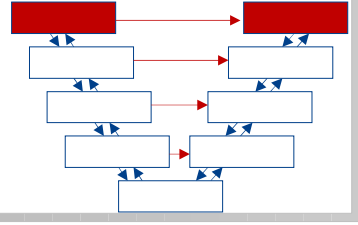
Abnahmetest - Begriffsklärung

- Bei den bisher beschriebenen Teststufen handelt es sich um Testarbeiten, die in Verantwortung des Herstellers oder der entwickelnden Projektgruppe durchgeführt werden, bevor die Software an den jeweiligen Kunden oder Nutzer übergeben wird.
- Vor Inbetriebnahme der Software (insbesondere wenn kundenspezifische Individualsoftware erstellt wurde) erfolgt nun als abschließender Test noch ein so genannter **Abnahmetest**.
- Hierbei stehen die Sicht und das Urteil des Kunden bzw. Anwenders im Vordergrund.
- Dabei zielt der **Abnahmetest** nicht auf das Finden von Fehlerzuständen ab. Vielmehr soll Vertrauen in das Produkt gewonnen werden sowie dessen Eignung für den beabsichtigten Einsatz beurteilt werden.
- Der Abnahmetest ist unter Umständen der einzige Test, den der Kunde nachvollziehen kann oder an dem er direkt beteiligt ist.
- Der Abnahmetest ist eine spezielle Form des Systemtests.
- Der Abnahmetest wird beim Kunden durchgeführt.



Abnahmetest – Regulatorischer und vertraglicher Abnahmetest

- Auf Basis der Ergebnisse des Abnahmetests entscheidet der Kunde, ob er das bestellte Softwaresystem als mangelfrei betrachtet und den Entwicklungsvertrag bzw. die vertraglich geschuldete Leistung als erfüllt ansieht.
- Dies kann auch ein mehr oder weniger formal gestalteter Vertrag oder Projektauftrag sein, der zwischen beauftragender Fachabteilung und realisierender IT-Abteilung einer Firma oder eines Konzerns besteht.
- Beim **vertraglichen Abnahmetest** wird die Software explizit gegen die vertraglichen Abnahmekriterien geprüft.
  - Als Testkriterien gelten demnach die im Entwicklungsvertrag festgeschriebenen Abnahmekriterien, die deshalb klar und eindeutig formuliert werden müssen.
- **Regulatorische Abnahmetests** werden gegen alle Gesetze und Standards durchgeführt, denen das System entsprechen muss.



Abnahmetest – Vertraglicher Abnahmetest

In der Praxis wird natürlich der Softwarehersteller schon in seinem eigenen Systemtest diese Abnahmekriterien auf Erfüllung prüfen und entsprechende Testfälle vorsehen.

Für den Abnahmetest reicht es dann aus, die gemäß Vertrag abnahmerelevanten Testfälle zu wiederholen und so dem Kunden zu demonstrieren, dass die Akzeptanzkriterien des Vertrags erfüllt sind.

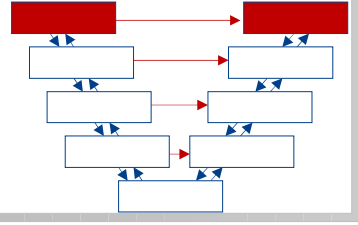
Im Gegensatz zum Systemtest, der in der Systemtestumgebung des Herstellers stattfindet, werden die Abnahmetests in der Abnahmeumgebung des Kunden durchgeführt.

Wegen der unterschiedlichen Testumgebungen kann ein Testfall in der Abnahme durchaus fehlschlagen, der im Systemtest nie Probleme bereitet hat !

Die Abnahmeumgebung soll so weit wie möglich der späteren Produktivumgebung entsprechen. Eine Testdurchführung in der Produktivumgebung selbst ist allerdings zu vermeiden, um den produktiven Betrieb laufender Softwaresysteme nicht zu gefährden.

Zur Ermittlung geeigneter Abnahmetests oder Abnahmekriterien können dieselben Methoden herangezogen werden wie für die Testfallermittlung im Systemtest.

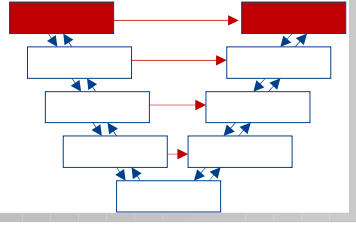
Insbesondere sind auch Geschäftsvorfälle einer typischen Zeit- oder Abrechnungsperiode (z. B. Monatsabschluss) zu berücksichtigen.



Abnahmetest - Test des Betriebs

Der **betriebliche Abnahmetest**, meist durch den Systemadministrator des Kunden durchzuführen, umfasst z.B. Tests:

- zum Erstellen und Wiedereinspielen von Sicherungskopien (Backup / Restore),
- zur Wiederherstellbarkeit des vorherigen Zustands nach Systemausfällen,
- von Funktionen des Benutzermanagements,
- der routinemäßigen Wartungsaufgaben,
- für Datenimport- und Migrationsaufgaben als auch
- die Überprüfung von möglichen Sicherheitslücken.



Abnahmetest - Test auf Benutzerakzeptanz

Ein weiterer Aspekt im Rahmen der Abnahme – als letzte Stufe der Validierung – ist der **Test auf Benutzerakzeptanz** (auch als Benutzerabnahmetest bezeichnet).

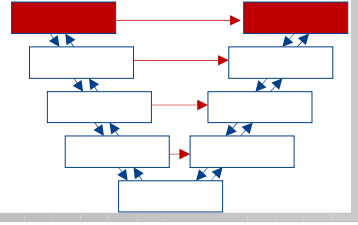
Ein solcher Test ist immer dann zu empfehlen, wenn Kunde und Anwender des Systems verschiedene Personen(gruppen) sind. Die unterschiedlichen Anwendergruppen haben in der Regel ganz verschiedene Erwartungen an das neue System.

Wenn eine Anwendergruppe das System ablehnt, z. B. weil es als »umständlich« empfunden wird, kann dies das Scheitern der gesamten Systemeinführung zur Folge haben, obwohl das System funktional vollkommen in Ordnung ist.

Deshalb ist es wichtig, für jede Anwendergruppe Tests auf Benutzerakzeptanz vorzusehen. Diese Tests werden meist vom Kunden selbst organisiert, der auch die Testfälle auswählt (basierend auf seinen Geschäftsprozessen und typischen Anwendungsszenarien).

Wenn im Abnahmetest gravierende Akzeptanzprobleme sichtbar werden, ist es für über Kosmetik hinausgehende Gegenmaßnahmen allerdings oft zu spät.

Um solchen Desastern vorzubeugen, ist es vernünftig, schon **in frühen Projektphasen Prototypen durch repräsentativ ausgewählte Vertreter** der späteren Anwender begutachten zu lassen.



Alpha- und Beta-Test

Soll die Software in sehr vielen verschiedenen Produktivumgebungen betrieben werden, ist es für den Softwarehersteller sehr kostenintensiv oder gar unmöglich, im Systemtest jede dieser Produktivumgebungen mit einer entsprechende Testumgebung nachzubilden.

In solchen Fällen wird der Softwarehersteller dem Systemtest **Alpha- und Beta-Tests** nachschalten (Test durch repräsentative Kunden). Ziel dieser Tests ist, Einflüsse aus nicht vollständig bekannten oder nicht spezifizierten Produktivumgebungen zu erkennen und ggf. zu beheben.

Der Hersteller liefert hierzu stabile Vorabversionen der Software an einen ausgewählten Kundenkreis, der den Markt für die Software gut repräsentiert oder dessen Produktivumgebungen die verschiedenen möglichen Umgebungen gut abdecken.

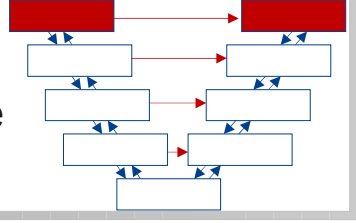
Diese Kunden führen dann entweder vom Hersteller vorgegebene Testszenarien durch oder sie setzen das vorläufige Produkt probehalber unter realistischen Bedingungen ein. Anschließend geben sie ihre Fehlermeldungen (aber auch allgemeine Kommentare und Eindrücke über das neue Produkt) an den Hersteller zurück, der entsprechende Anpassungen vornimmt.

Unterschied von Alpha- und Beta-Tests:

- Alpha-Tests finden beim Hersteller statt.
- Beta-Tests (auch Feldtests genannt) beim Kunden.

Alpha- und Beta-Tests dürfen einen hausinternen Systemtest des Herstellers nicht ersetzen (auch wenn einige Hersteller dies vielleicht so sehen).

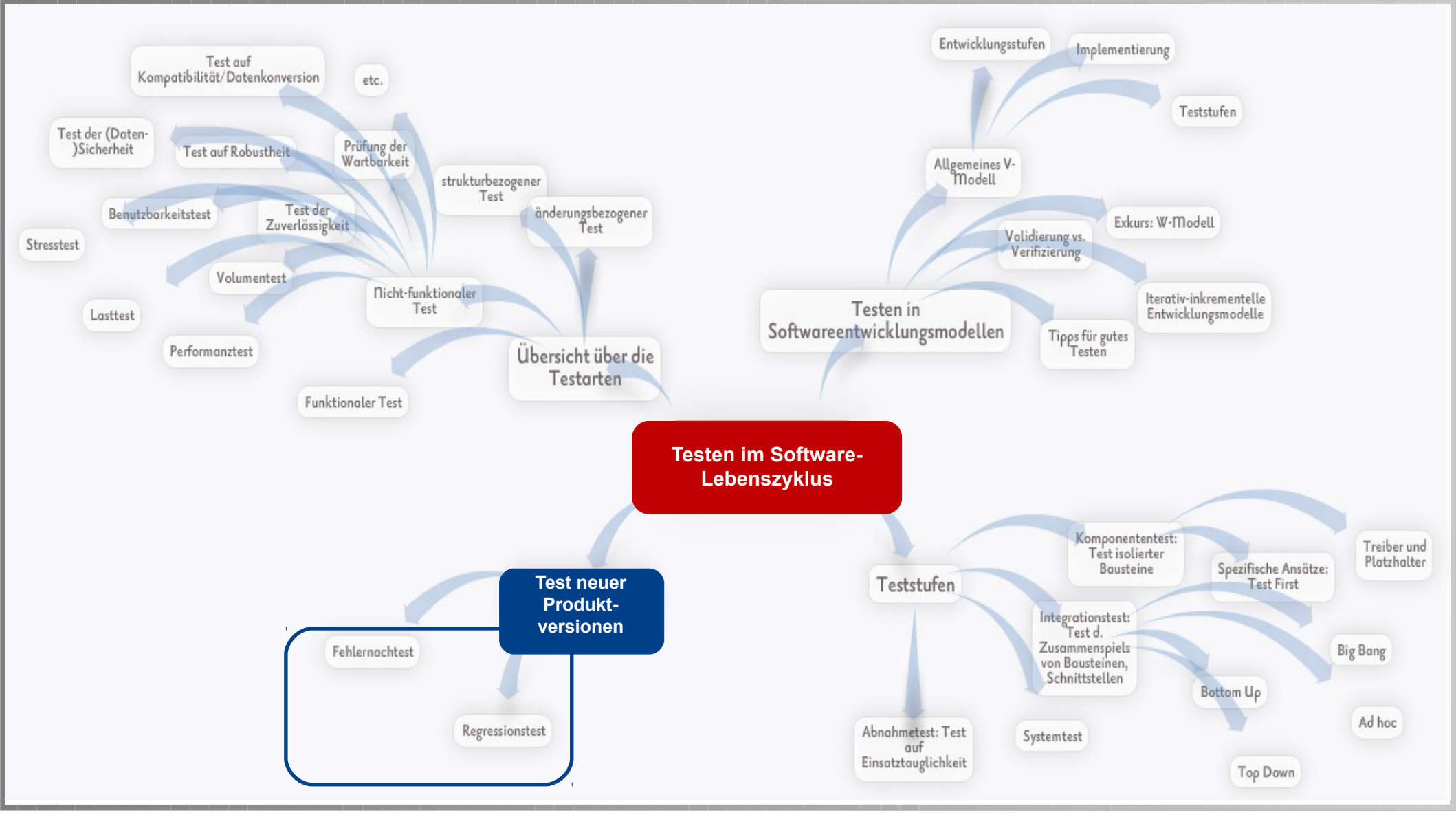
Erst wenn der Systemtest nachgewiesen hat, dass die Software hinreichend stabil ist, sollte das Produkt dem Endkunden für einen Alpha- oder einen Beta-Test zugemutet werden.



2.2 Testen im Software-Lebenszyklus

- Testen in Softwareentwicklungsmodellen
- Teststufen → Abnahmetest
- **Test neuer Produktversionen**
- Übersicht über die Testarten

Big Picture


Testen im Software-Lebenszyklus
Übersicht über die Testarten
Nicht-funktionaler Test
Test auf Kompatibilität/Datenkonversion
etc.
Test der (Daten-)Sicherheit
Test auf Robustheit
Prüfung der Wartbarkeit
strukturbezogener Test
Benutzbarkeitstest
Test der Zuverlässigkeit
änderungsbezogener Test
Stresstest
Volumentest
Lasttest
Performanztest
Funktionaler Test
Testen in Softwareentwicklungsmodellen
Entwicklungsstufen
Implementierung
Teststufen
Allgemeines V-Modell
Exkurs: W-Modell
Validierung vs. Verifizierung
Iterativ-inkrementelle Entwicklungsmodelle
Tipps für gutes Testen
Test neuer Produkt-versionen
Fehlernachtest
Regressionstest
Teststufen
Komponententest: Test isolierter Bausteine
Treiber und Platzhalter
Spezifische Ansätze: Test First
Integrationstest: Test d. Zusammenspiels von Bausteinen, Schnittstellen
Big Bang
Bottom Up
Ad hoc
Top Down
Abnahmetest: Test auf Einsatztauglichkeit
Systemtest

Test neuer Produktversionen (Wartungstest)

- Bisher wurde stillschweigend davon ausgegangen, dass ein Softwareentwicklungsvorhaben mit bestandenem Abnahmetest und der Auslieferung des neuen Produkts beendet ist.
  Die Realität sieht anders aus.
- Mit der erstmaligen Auslieferung steht ein Softwareprodukt erst am Anfang seines Lebenszyklus. Einmal installiert, ist es oft Jahre oder Jahrzehnte lang im Einsatz. Das Softwareprodukt, seine Umgebung sowie seine Konfigurationsdaten werden während dieser Zeitspanne vielfach korrigiert, geändert und erweitert.
- Jedes Mal entsteht eine neue Version des ursprünglichen Produkts, die getestet werden muss.

Softwarewartung

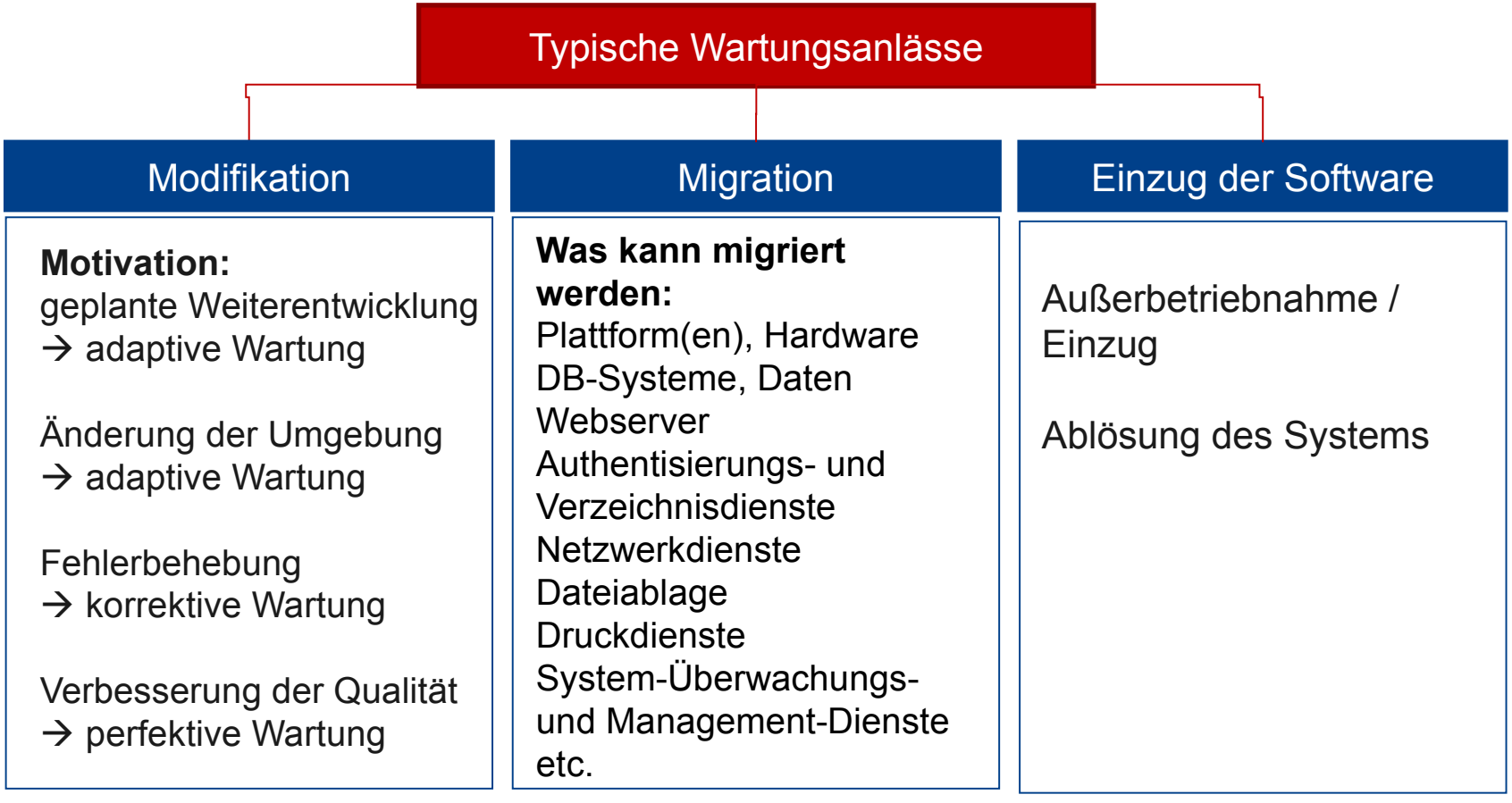
Im Gegensatz zu »klassischen« Industrieprodukten geht es bei der Softwarewartung nicht darum, durch regelmäßige Pflege die Einsatzfähigkeit zu erhalten oder Schäden, die z.B. durch Abnutzung entstehen, zu reparieren. Software nutzt sich nicht ab.

Von **Softwarewartung** wird gesprochen, wenn ein Produkt an geänderte Einsatzbedingungen angepasst wird (Softwarepflege) oder wenn Fehlerzustände beseitigt werden, die schon immer im Produkt enthalten waren (Softwarewartung im engeren Sinne).

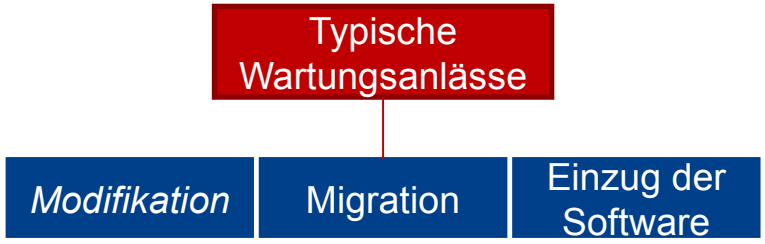
Jedes Softwaresystem bedarf also nach seiner Auslieferung gewisser Korrekturen und Ergänzungen.

Die Tatsache, dass Wartung auf jeden Fall notwendig ist, darf aber nicht als Argument missbraucht werden, um bei Komponenten-, Integrations- oder Systemtests zu sparen (»Wir müssen ja sowieso immer wieder neue Versionen rausbringen; also ist es nicht so schlimm, wenn wir es mit dem Testen nicht so genau nehmen und Fehler übersehen.«).

Typische Wartungsanlässe



Modifikation (1)



Geplante Weiterentwicklung:

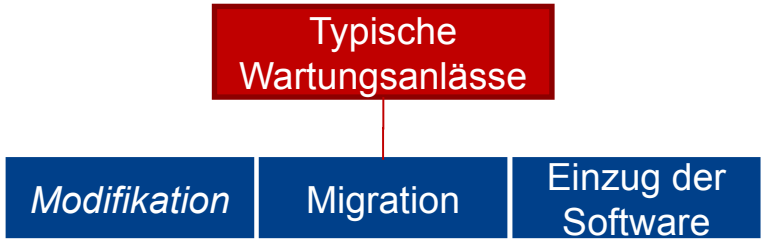
Außer den durch Mängel und Fehler ausgelösten Wartungsarbeiten gibt es auch Änderungs- und Erweiterungsarbeiten, die das Projektmanagement von Anfang an vorgesehen hat. Sie gehören zur normalen Produktweiterentwicklung.

Hierzu gehören beispielsweise:

- Anpassungen durch eine geplante Änderung eines Nachbarsystems.
- Umsetzung einer vorgesehene Funktionalität, die aus Zeitgründen nicht zur Systemeinführung geliefert werden konnte.
- Erweiterungen, die im Zuge einer geplanten Marktausdehnung notwendig werden.

Ein Softwareprojekt ist also keinesfalls mit der Lieferung der ersten Produktversion abgeschlossen.

Modifikation (2)



Fehlerbehebung:

- Fehler aus dem Betrieb
- Notfallkorrekturen („Hot Fixes“)

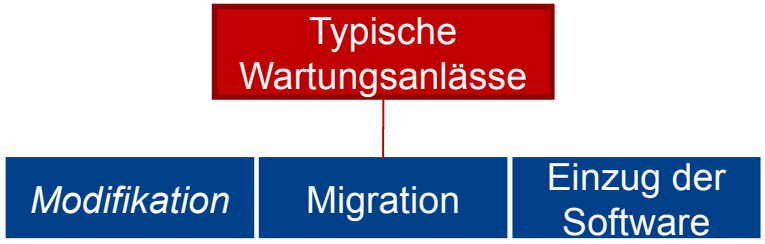
Änderung der Umgebung:

- Aktualisierung des Betriebssystems
- Aktualisierung des DB-Managementsystems
- Upgrades kommerzieller Software
- Patches externer Komponenten
- etc.

Verbesserung der Qualität:

- Verbesserung der Qualitätsfaktoren, z.B. Wartbarkeit, Performanz, Benutzbarkeit ohne Änderung des funktionalen Umfangs

Modifikation (3)



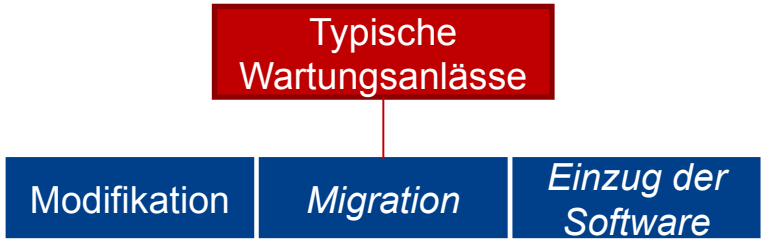
Es findet eine kontinuierliche Weiterentwicklung statt, die z.B. jährlich verbesserte Produktversionen auf den Markt bringt. Zweckmäßigerweise werden diese Lieferungen mit den Wartungsarbeiten synchronisiert, so dass z.B. halbjährlich eine neue Version erscheint: ein Wartungsupdate und ein echtes funktionales Update. Eine vorausschauende Release-Planung ist entscheidend für einen erfolgreichen Wartungstest.

Nach jeder Auslieferung beginnt das Projekt quasi mit einem neuen Durchlauf durch alle Projektphasen. Dieses Vorgehen wird deshalb auch iterativ-inkrementelle Softwareentwicklung genannt. Iterativ-inkrementelle Softwareentwicklung ist heute die Regel.

→ Wie muss das Testen darauf reagieren ?

→ Müssen alle Teststufen in vollem Umfang ihre Tests bei jedem Release des Produkts komplett wiederholen (Wartungstests auf jeder Stufe und für alle Testarten) ?

Wartungstest - Migration und Einzug



Wartungstest bei Migration:

- Umfasst Tests im Betrieb der neuen (Software-/Hardware-) Umgebung als auch der geänderten Software.
- Umfasst Konvertierungstests
  - der Software, die für die Konvertierung der Daten genutzt wird (z.B. von einem vorhandenen System zur Verwendung in einem, das alte System ersetzenden System) bzw.
  - der (neuen) Daten, die Ergebnis der Konvertierung sind.

Wartungstest bei der Außerbetriebnahme des Systems (Einzug):

- Umfasst das Testen der Datenmigration (auf das neue System) oder das Testen der Archivierung (bei langer Aufbewahrungszeit).

Regressionstest (1)

Durch Wartungsarbeiten und auch bei Weiterentwicklung werden Teile vorhandener Software geändert oder neue Softwarebausteine ergänzt.

In beiden Fällen muss die geänderte Software erneut getestet werden. Diese Tests heißen Regressionstests. Der **Regressionstest** ist ein erneuter Test eines bereits getesteten Programms nach dessen Modifikation.

Ziel

- Nachweisen, dass durch die vorgenommenen Änderungen keine neuen Fehlerzustände eingebaut oder bisher maskierte Fehlerzustände freigelegt wurden.

→ Was muss aufgrund einer Änderung getestet werden ?
Wie umfangreich muss ein Regressionstest sein ?

- Eine Auswirkungsanalyse ist durchzuführen, um den Umfang des Regressionstests zu bestimmen.

Regressionstest (2)

Umfang des Regressionstests:

1. Wiederholung aller Tests, die Fehlerwirkungen erzeugt haben, deren Ursache der korrigierte Fehlerzustand war (Fehlernachtest) ?
2. Test aller Programmstellen, an denen korrigiert oder geändert wurde (Test geänderter Funktionalität) ?
3. Test aller Programmteile oder Bausteine, die neu eingefügt wurden (Test neuer Funktionalität) ?
4. Das komplette System (vollständiger Regressionstest) ?

Welche der Strategien ist in der Praxis die richtige ?

Regressionstest (2)

Umfang des Regressionstests:

1. Wiederholung aller Tests, die Fehlerwirkungen erzeugt haben, deren Ursache der korrigierte Fehlerzustand war (Fehlernachtest) ?
2. Test aller Programmstellen, an denen korrigiert oder geändert wurde (Test geänderter Funktionalität) ?
3. Test aller Programmteile oder Bausteine, die neu eingefügt wurden (Test neuer Funktionalität) ?
4. Das komplette System (vollständiger Regressionstest) ?

Sowohl der reine Fehlernachtest (1) als auch Tests nur am »Ort« der Modifikation (2 und 3) sind zu wenig.

In Softwaresystemen können scheinbar simple lokale Änderungen unerwartete Auswirkungen und Seiteneffekte auf beliebige andere (auch weit entfernte) Systemteile haben.

Regressionstest (3)

Vollständiger Regressionstest:

- Zusätzlich zum Test korrigierter Fehlerzustände und zusätzlich zum Test geänderter Funktionen müssten eigentlich alle vorhandenen Testfälle wiederholt werden.
- Erst dann hätte der Test die gleiche Aussagekraft wie der entsprechende Test, der an der Ausgangsversion des Programms durchgeführt wurde.
- Ein solcher vollständiger Regressionstest müsste ebenfalls durchgeführt werden, wenn die Systemumgebung geändert wurde, da dies potentiell auf jedes Systemteil Rückwirkungen haben kann.

In der Praxis ist ein vollständiger Regressionstest aber fast immer zu zeit- und kostenintensiv.

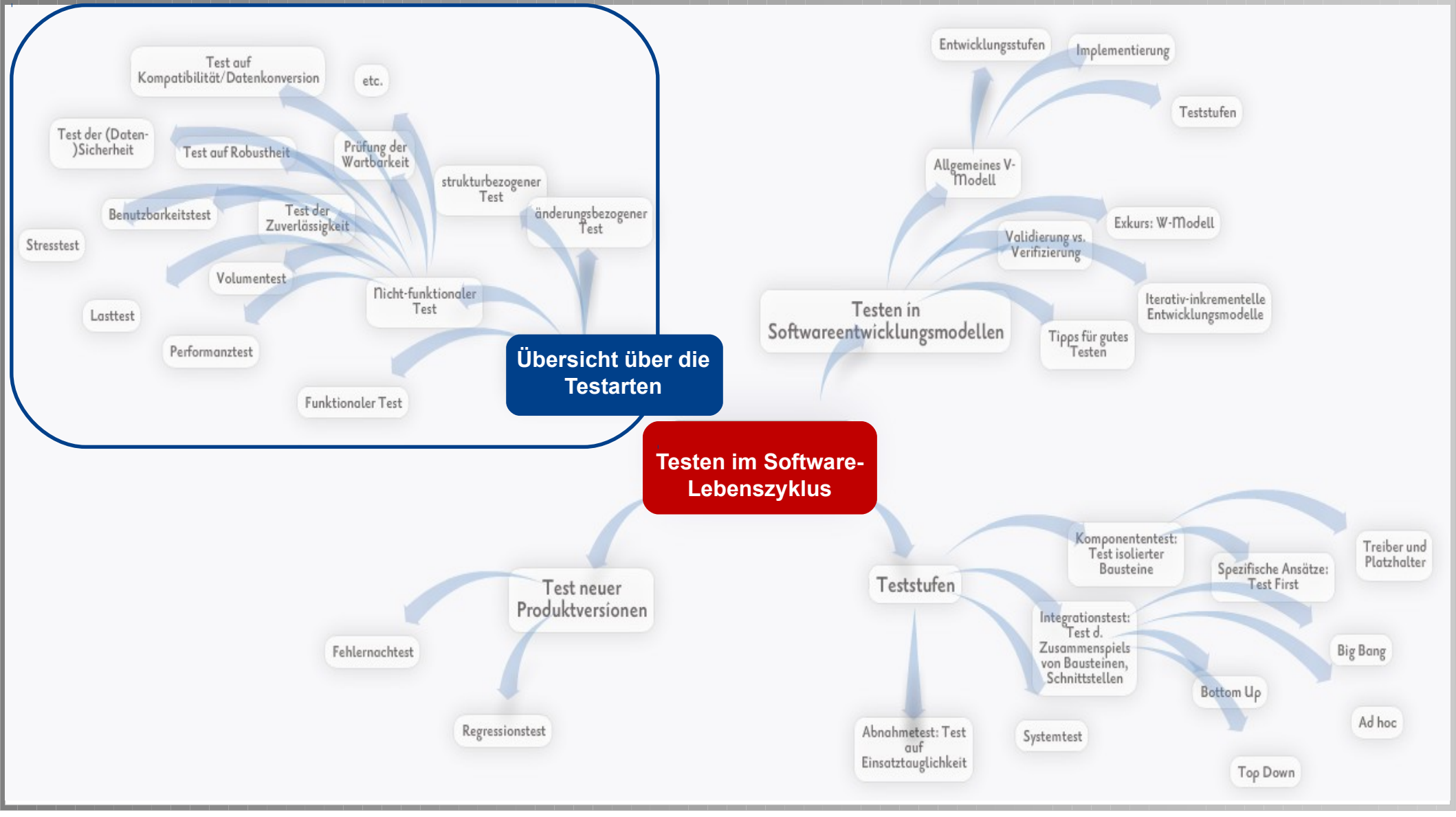
Auswahl von Regressionstestfällen:

- Wiederholung nur von derjenigen Tests aus dem Testplan, denen hohe Priorität zugeordnet ist.
- Bei funktionalen Tests Verzicht auf gewisse Varianten (Sonderfälle).
- Einschränkung der Tests auf bestimmte Konfigurationen
  (z. B. nur Test der englischsprachigen Produktversion, nur Test auf einer bestimmten Betriebssystemversion u. Ä.).
- Einschränkung der Tests auf bestimmte Teilsysteme oder Teststufen.

2.2 Testen im Software-Lebenszyklus

- Testen in Softwareentwicklungsmodellen
- Teststufen → Abnahmetest
- Test neuer Produktversionen
- Übersicht über die Testarten

Big Picture


Testen im Software-Lebenszyklus
Übersicht über die Testarten
Test auf Kompatibilität/Datenkonversion
etc.
Test der (Daten-)Sicherheit
Test auf Robustheit
Prüfung der Wartbarkeit
strukturbezogener Test
Benutzbarkeitstest
Test der Zuverlässigkeit
änderungsbezogener Test
Stresstest
Volumentest
Nicht-funktionaler Test
Lasttest
Performanztest
Funktionaler Test
Testen in Softwareentwicklungsmodellen
Entwicklungsstufen
Implementierung
Teststufen
Allgemeines V-Modell
Exkurs: W-Modell
Validierung vs. Verifizierung
Iterativ-inkrementelle Entwicklungsmodelle
Tipps für gutes Testen
Test neuer Produktversionen
Fehlernachtest
Regressionstest
Teststufen
Komponententest: Test isolierter Bausteine
Treiber und Platzhalter
Spezifische Ansätze: Test First
Integrationstest: Test d. Zusammenspiels von Bausteinen, Schnittstellen
Big Bang
Bottom Up
Ad hoc
Abnahmetest: Test auf Einsatztauglichkeit
Systemtest
Top Down

Vergleich der Teststufen

Kriterium	Komponententest	Integrationstest	Systemtest	Abnahmetest
Testziele	Fehlerzustände in Software (-bausteinen), die separat getestet werden können, finden.	Fehlerzustände in Schnitt-stellen und im Zusam-menspiel zwischen integrierten Komponenten finden.	Prüfung, ob die spezifizierten Anforderungen (funktional, nicht-funktional) vom Produkt erfüllt werden.	Vertrauen in das System oder in bestimmte nicht-funktionale Eigenschaften gewinnen.
Testbasis	→Komponenten-spezifikation →Detaillierter Entwurf →Datenmodell →Programmcode	→Software- und →Systementwurf →Architektur →Nutzungsabläufe, Workflows →Anwendungsfälle	→System- und Anforderungsspezifikation →Anwendungsfälle →funktionale Spezifikation →Geschäftsprozesse →Risikoanalyseberichte	→Benutzer-anforderungen →Systemanforderungen →Anwendungsfälle →Geschäftsprozesse →Risikoanalyseberichte
Typische Test-objekte	Isolierte Softwarebausteine (Klasse, Unit, Modul) →Komponenten, Programme →Datenumwandlungs- / Migrationsprogramme →Datenbankmodule	Zu integrierende Einzel-bausteine, Subsysteme und zugekaufte Standard-Komponenten →Datenbank-implementierungen →Infrastruktur →Schnittstellen →Systemkonfiguration und Konfigurationsdaten	→System-, Anwender- und Betriebshandbücher →Systemkonfiguration und Konfigurations-daten	→Geschäftsprozesse des integrierten Systems →Betriebs- und Wartungsprozesse →Anwenderverfahren →Formulare →Berichte →Konfigurationsdaten
Testwerk-zeuge	Entwicklungsumgebung, Debugging-Unterstützung Stat. Analysewerkzeuge, Komponententestumgebung	Testmonitore zur Überwachung des Datenaustauschs zwischen Komponenten	Testmanagement-Werkzeuge, GUI-Automatisierungs-werkzeuge	
Testumge-bung	Platzhalter, Treiber, Simulatoren	Wiederverwendung / Erweiterung der Platzhalter, Treiber, Simulatoren aus dem Komponententest	Test- und Produktivumgebung sollten so weit wie möglich übereinstimmen.	Test- und Produktivumgebung sollten so weit wie möglich übereinstimmen.

Teststufen und Testarten

Ziele des Testens variieren von Stufe zu Stufe. Es kommen verschiedene Testarten in unterschiedlicher Intensität zur Anwendung.

Die grundlegenden **Testarten** sind:

- funktionaler Test (Testen der Funktionalität)
- nicht-funktionaler Test (Testen der nicht-funktionalen Softwaremerkmale)
- strukturorientierter Test (Testen der Softwarestruktur/Softwarearchitektur)
- änderungsorientierter Test (Testen im Zusammenhang mit Änderungen, auch änderungsbezogener Test genannt)

Alle Arten können auf den verschiedenen Teststufen (Komponententests, Integrationstests, Systemtests, Abnahmetests) zur Anwendung kommen.

Funktionaler Test

Funktionalität beschreibt, »was« das System leisten soll. Funktionalität wird vom System, von einem Teilsystem oder einer Komponente geliefert.

Testbasis sind Anforderungsspezifikation, Anwendungsfälle oder funktionale Spezifikation. Funktionalität kann auch undokumentiert sein (z.B. berechtigte Erwartung des Kunden).

Funktionaler Test prüft das von außen sichtbare Verhalten der Software (Black-box Testentwurfsverfahren, s. Abschnitt 2.4).

Verwendung von spezifikationsorientierten Testentwurfsverfahren, um Ausgangskriterien (auch Test-Ende-Kriterien genannt) und Testfälle aus der Funktionalität der Software oder des Systems herzuleiten.

Nicht-funktionaler Test

Nicht-funktionaler Test prüft anhand von Software- und Systemmerkmalen, »wie gut« das System arbeitet.

Nicht-funktionale Tests sind z.B.:

- Performanztest,
- Lasttest,
- Stresstest,
- Benutzbarkeitstest,
- Wartbarkeitstest,
- Zuverlässigkeitstest und
- Portabilitätstest

Kann in allen Teststufen zur Anwendung kommen.

Nicht-funktionaler Test prüft meist das von außen sichtbare Verhalten der Software (Black-box Testentwurfsverfahren, s. Abschnitt 2.4).

Grundlage sind Qualitätsmodelle (z.B. ISO 9126, s. Abschnitt 1.0).

Strukturorientierter Test

Der **strukturorientierte Test** (White-box Testentwurfsverfahren, s. Abschnitt 2.5) basiert auf der internen Struktur einer Komponente bzw. der Architektur der Software bzw. des Systems.

Grundlage sind z.B.

- der Kontrollfluss innerhalb von Komponenten,
- die Aufrufhierarchie von Prozeduren oder Menüstrukturen oder
- die Struktur abstrakter Modelle der Software (Zustandsautomat)

Ziel ist die Überdeckung durch Tests möglichst alle Elemente der betrachteten Struktur.

Geeignete und ausreichend viele Testfälle sind zu entwerfen.

Strukturorientierte Tests kommen überwiegend im Komponenten- und Integrationstest zum Einsatz, als Ergänzung auch in höheren Teststufen (z.B. zur Abdeckung von Menüstrukturen).

Änderungsorientierter Test

Testen im Zusammenhang mit Änderungen (Fehlernachtest und Regressionstest).

Wurde im Test eine Fehlerwirkung nachgewiesen und korrigiert, muss erneut getestet werden, ob der Fehlerzustand erfolgreich beseitigt wurde (**Fehlernachtest**, auch Nachtest genannt).

Werden bereits getestete Programm(teil)en einer Änderung unterzogen, ist nachzuweisen, dass durch die Änderung keine Fehlerzustände neu eingebaut oder (bisher maskierte) Fehlerzustände dadurch zur Wirkung kommen (**Regressionstest**).

Regressionstests sind auch durchzuführen, wenn sich die Softwareumgebung ändert.

Fehlernach- und Regressionstests werden oft mehrfach ausgeführt und müssen wiederholbar sein.

Regressionstests umfassen funktionale, nicht-funktionale als auch strukturorientierte Tests (auf allen Teststufen).

Zusammenfassung

Das V-Modell definiert grundlegende Teststufen (Komponententest, Integrationstest, Systemtest und Abnahmetest) und unterscheidet zwischen verifizierender und validierender Prüfung.

- Der Komponententest testet einzelne Softwarebausteine.
- Der Integrationstest prüft das Zusammenwirken bereits getesteter Bausteine.
- Funktionaler und nicht-funktionaler Systemtest betrachten das Gesamtsystem aus Sicht des späteren Anwenders.
- Im Abnahmetest prüft der Auftraggeber das erstellte Produkt auf vertragliche Akzeptanz und Benutzerakzeptanz.

Wenn das System in sehr vielen Produktivumgebungen betrieben werden soll, bieten Alpha- und Beta-Tests eine zusätzliche Möglichkeit, Einsatzerfahrungen mit Vorabversionen des Produkts zu sammeln.

Durch Fehlerkorrekturen (Wartung) und geplante Weiterentwicklung (Pflege) wird ein Softwareprodukt im Laufe seines Lebenszyklus immer wieder geändert oder erweitert. Jede dieser geänderten Versionen muss erneut getestet werden. Welchen Umfang solche Regressionstests haben, muss eine individuelle Risikoabschätzung festlegen.

Die grundlegenden Testarten sind funktionaler und nicht-funktionaler Test, sowie struktur- und änderungsorientierter Test.

Sie sollten jetzt...

- wissen, wann das Testen im Softwarelebenszyklus beginnt,
- wissen, wie Entwicklungs- und Testaktivitäten zusammenhängen,
- wissen, dass Softwareentwicklungsmodelle an Projekt- und Produkteigenschaften angepasst werden müssen,
- Komponenten-, Integrations-, System- und Abnahmetest vergleichen können,
- Eigenschaften "guter" Tests nennen können, die in beliebigen Entwicklungszyklen anwendbar sind,
- wissen, wie das Testen von neuen Produktversionen aussehen soll,
- typische Anlässe für Wartungstests kennen,
- die Rolle von Regressionstests und Auswirkungsanalysen in der Softwarewartung beschreiben können,
- die zu unterscheidenden Testarten vergleichen können.

Folgende Fragen sollten Sie jetzt beantworten können

- Erläutern Sie die einzelnen Phasen des allgemeinen V-Modells.
- Definieren Sie die Begriffe Verifizierung und Validierung.
- Begründen Sie, warum Verifizierung sinnvoll ist, auch wenn eine sorgfältige Validierung stattfindet (und umgekehrt).
- Charakterisieren Sie die typischen Testobjekte im Komponententest.
- Nennen Sie die Testziele des Integrationstests.
- Welche Integrationsstrategien lassen sich unterscheiden ?
- Welche Gründe sprechen dafür, Tests in einer separaten Testinfrastruktur durchzuführen ?
- Erläutern Sie anforderungsbasiertes Testen.
- Definieren Sie Lasttest, Performanztest, Stresstest.
  Was sind die Unterscheidungsmerkmale ?
- Worin unterscheiden sich Fehlernachtest und Regressionstest ?
- In welcher Projektphase nach allgemeinem V-Modell sollte das Testkonzept erstellt werden ?
- Was sind die Ziele von Fehlernach- und Regressionstest ?
- Welche Testarten lassen sich unterscheiden ?