Software-Engineering

Fachhochschule Niederrhein

SS 2006

Prof. Dr. Rethmann

Inhalt

- Software-Qualität: Test und Verifikation
- Design von Software verbessern: Refactoring
- Lernfähige Systeme: Neuronale Netze
- Unscharfes Wissen: Fuzzy-Logik
- Expertensysteme

Software-Qualität

Test und Verifikation

Literatur

- Spillner, Linz: Basiswissen Softwaretest. dpunkt.verlag
- Sneed, Winter: Testen objektorientierter Software. Hanser Verlag.
- Ehrenberger: Software-Verifikation. Hanser Verlag.
- Zeller, Krinke: Programmierwerkzeuge. dpunkt.verlag.

Software-Fehler haben Konsequenzen

- 1979: erste Venussonde flog am Ziel vorbei, weil im Fortran-Programm Punkt mit Komma vertauscht wurde
  Kosten: mehrere hundert Millionen Dollar
- 1984: Überschwemmung in Frankreich, da Computer Überlaufgefahr nicht erkannte und Schleusen öffnete
- Ariane 5 verglüht
  Grund: Software-Problem im Trägernavigationssystem
  Kosten: ca. 1 Milliarde Euro
- Bank of New York buchte 32 Milliarden Dollar zuviel, weil ein 16-Bit Zähler überlief
  Kosten: 5 Millionen Dollar (Zinsverlust)

Minimiere die Komplexität: the art of programming is the art of organizing complexity (Dijkstra)

Fehlerursachen

Viele Fehlerursachen treten nicht erst bei der Programmierung auf, sondern sind Folgen von Fehlern in früheren Phasen:

- Anforderungsdefinition: fehlerhafte Anforderungen
- Systemspezifikation: Spezifikationsfehler
- Entwurf: Entwurfsfehler
- Implementierung: Programmfehler
- Test und Integration: korrigierte Fehler

Fehlerursachen (2)

Planungsfehler:

- Anforderungen sind zu schwammig formuliert
- Terminvorgaben sind unrealistisch
- kein Risikomanagement (Projektverfolgung)
- Auswahl/Einsatz von Tools unzureichend vorbereitet
- Aus-/Weiterbildung der Mitarbeiter unzureichend
- zu früh mit Codierung begonnen
- mangelndes Qualitätsmanagement
- Vorgehensmodell wird nicht befolgt

Fehlerursachen (3)

Problem- und Systemanalyse:

- Anforderungen/Qualitätsmerkmale unzureichend
- Begriffsdefinition mangelhaft
- semantisches Datenmodell mangelhaft

Systementwurf:

- kein modularer Aufbau
- mangelhafte Datenkapselung
- Systemarchitektur zu kompliziert
- falsche/unvollständige ER-/UML-Diagramme
- falsche Strukturbeschreibungen

Fehlerursachen (4)

Kodierung:

- nichtbefolgen von Programmierstandards/-richtlinien
- keine modulare Programmierung (top down/bottom up)
- schlechte Namensvergabe
- nicht objektorientiert

Verifizierung und Validierung:

- unsystematisches Testen
- keine Abnahme der Phasenergebnisse

Fehlerursachen (5)

Betrieb und Wartung:

- Dokumentation fehlt, nicht adäquat oder veraltet
- mangelnde Schulung der Anwender
- unzureichendes Konfigurationsmanagement

⇒ In jeder Phase sind Qualitätsprüfungen durchzuführen!

- **Spiralmodell:** mehrfaches, zyklisches Durchlaufen der Teilphasen *Analyse*, *Entwurf*, *Kodierung* und *Testen* erkennt frühzeitig evtl. vorhandene Mängel
- **Refactoring:** Design vorhandener Software verbessern und Entwurfsfehler korrigieren

Software-Qualität

Was ist das? [ISO 9126]

- Funktionalität
- Zuverlässigkeit
- Benutzbarkeit
- Effizienz
- Änderbarkeit
- Übertragbarkeit

Auftraggeber muss festlegen, welche Qualitätsmerkmale für ihn eine hohe Bedeutung haben

Software-Qualität (2)

Funktionalität

- geforderte Fähigkeit des Systems ist beschrieben durch
  * spezifiziertes Ein-/Ausgabeverhalten oder
  * Reaktion bzw. Wirkung des Systems auf Eingaben
- Funktionalität gliedert sich in
  * Angemessenheit
  * Richtigkeit
  * Interoperabilität
  * Ordnungsmäßigkeit
  * Sicherheit

Software-Qualität (3)

Angemessenheit jede geforderte Fähigkeit ist im System vorhanden und geeignet realisiert

Richtigkeit die richtigen bzw. spezifizierten Reaktionen oder Wirkungen werden vom System geliefert

Interoperabilität bezeichnet das Zusammenspiel des zu testenden Systems mit anderen Systemen

Ordnungsmäßigkeit Software erfüllt anwendungsspezifische Normen, Vereinbarungen, gesetzliche Bestimmungen oder ähnliche Vorschriften

Sicherheit unberechtigten Zugriff (versehentlich/vorsätzlich) auf Programme und Daten verhindern → Zugriffs- und Datensicherheit

Software-Qualität (4)

Zuverlässigkeit

- Leistungsniveau unter festgelegten Bedingungen über einen definierten Zeitraum bewahren
- **Reife** wie oft kommt ein Versagen der Software durch Fehler vor?
- **Fehlertoleranz** die Software erlangt nach einem Defekt (falscher Schnittstellenaufruf, falsche Bedienung) sein spezifiziertes Leistungsniveau wieder
- **Wiederherstellbarkeit** wie einfach/schnell kann das geforderte Leistungsniveau nach einem Versagen/Ausfall wieder erreicht werden?

Software-Qualität (5)

Benutzbarkeit

- spielt entscheidende Rolle für Akzeptanz des Systems
- Aufwand, der zur Benutzung des Systems erforderlich ist unter Berücksichtigung unterschiedlicher Benutzergruppen?
- Aspekte: Verständlichkeit, Erlernbarkeit, Bedienbarkeit

Effizienz

- benötigte Zeit und Verbrauch an Betriebsmitteln für Erfüllung einer Aufgabe

Software-Qualität (6)

Softwaresysteme werden oft über einen langen Zeitraum und auf unterschiedlichen Plattformen (Betriebssystem und Hardware) eingesetzt

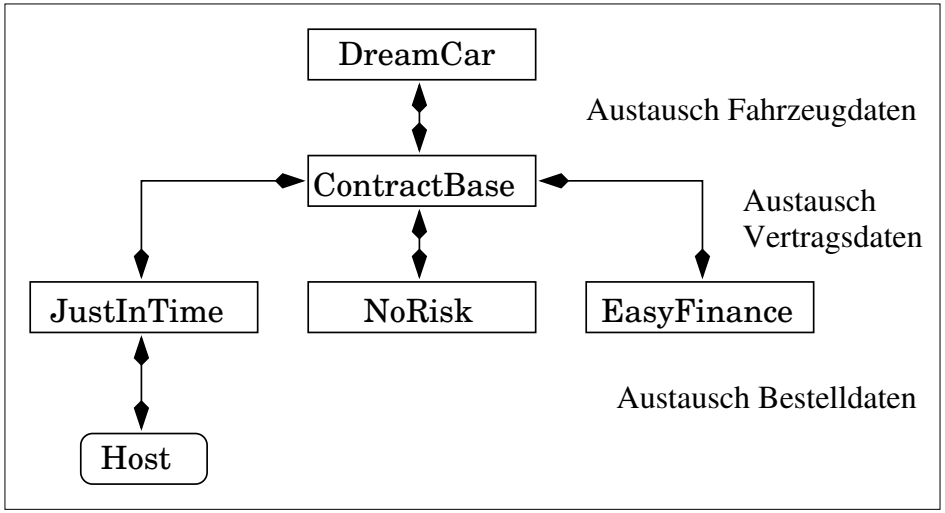
- **Änderbarkeit:** Analysierbarkeit, Modifizierbarkeit, Stabilität und Prüfbarkeit
- **Übertragbarkeit:** Anpassbarkeit, Installierbarkeit, Konformität und Austauschbarkeit

Ein Softwaresystem kann nicht alle Qualitätsmerkmale gleich gut erfüllen! → Priorisierung
Beispiel: effizient → nur eingeschränkt übertragbar (spezielle Eigenschaften der Plattform ausnutzen)

Fallbeispiel

- Autokonzern entwickelt elektronisches Verkaufssystem
- soll weltweit bei allen Händlern installiert werden
- Kunde kann Fahrzeug am Bildschirm konfigurieren
- System zeigt mögliche Modelle und Ausstattungsvarianten an und ermittelt zu jeder Auswahl sofort den Listenpreis → `DreamCar`
- Kunde kann Finanzierung kalkulieren → `EasyFinance`
- Kunde kann Fahrzeug online bestellen → `JustInTime`
- Kunde kann Versicherung abschließen → `NoRisk`
- System verwaltet Kundeninformationen und Vertragsdaten → `ContractBase`

Fallbeispiel (2)



Testaufwand

- vollständiger Test umfasst Kombination aller möglichen Eingaben unter Berücksichtigung aller unterschiedlichen Randbedingungen, die Einfluss auf den Systemablauf haben
- **Beispiel:** in einer Schleife gäbe es 5 unterschiedliche Wege durch Verzweigungen → $5^{20} + 5^{19} + 5^{18} + \ldots + 5^{1}$ unterschiedliche Abläufe des Programms bei max. 20 Schleifendurchläufen
  pro Test 5Mikrosekunden ⇒ 19 Jahre Testen
- der Testaufwand muss im vernünftigen Verhältnis zum erzielbaren Ergebnis stehen

Testaufwand (2)

- Testen ist ökonomisch sinnvoll, solange die Kosten für das Finden und Beseitigen von Fehlern im Test niedriger sind als die Kosten, die mit dem Auftreten des Fehlers bei Nutzung verbunden sind.
- im Fallbeispiel: fehlerhafte Preisberechnung bei Online-Bestellung führt dazu, dass tausende von Fahrzeugen zu einem zu geringen Preis verkauft werden
- Test muss strukturiert und systematisch durchgeführt werden: viele Fehler mit angemessenem Aufwand finden, unnötige Tests vermeiden

Testprozess

Testplanung

- Planung des Testprozesses zu Beginn des SWE-Projekts
- regelmäßige Kontrolle der Planung, Aktualisierung und Anpassung notwendig
- planen der Ressourcen: welche Mitarbeiter, wieviel Zeit, welche Hilfsmittel, Mitarbeiterschulung → Testkonzept
- lege Teststrategie fest: vollständiger Test nicht möglich, daher Prioritäten setzen anhand Risikoeinschätzung
- kritische Systemteile → intensiver Test
- Ziel: optimale Verteilung der Tests auf richtige Stellen
- SWE-Projekte oft unter Zeitdruck abgewickelt → durch Priorisierung kritische Softwareteile zuerst testen

Testprozess (2)

Testplanung im Fallbeispiel: in Absprache mit Auftraggeber wird festgestellt:

- Fehlverhalten/Ausfall von `DreamCar` oder `ContractBase` → größte schädliche Auswirkung
- online-Bestellung durch `JustInTime` → weniger kritisch aber: Daten des Bestellvorgangs dürfen nicht verfälscht oder verloren gehen
- für `EasyFinance` und `NoRisk`: alle Hauptfunktionen testen (Tarif kalkulieren, Vertrag erfassen/abschließen), aber aus Zeitgründen nur die am häufigsten vorkommenden Tarifkonstellationen konzentrieren

⇒ Festlegen der Testintensität sowohl für Teilsysteme, als auch für einzelne Aspekte sinnvoll

Testprozess (3)

Testspezifikation

- zuerst logische Testfälle definieren, dann konkrete Eingabewerte festlegen
- Auswahl der Testfälle auf Grundlage der Testbasis: aus Spezifikation der Testobjekte ableiten (Black-Box) oder auf Basis des Programmtextes erstellen (White-Box)
- für jeden Testfall Ausgangssituation (Vorbedingung), Randbedingungen und Sollwerte beschreiben
- prüfe spezifizierte Testfälle/Behandlung von Ausnahme- und Fehlersituationen
- prüfen auf ungültige/unerwartete Eingaben bzw. Randbedingungen, für die keine Ausnahmebehandlungen spezifiziert wurden

Testprozess (4)

Testdurchführung

- direkt nach der Programmierung erfolgt Übergabe der zu testenden Systemteile an den Tester
- Prüfung auf Vollständigkeit → Testobjekt in Testumgebung installieren, prinzipielle Ablauffähigkeit überprüfen
- beginne mit Hauptfunktionen → bei Fehlerwirkungen Abbruch (muss in Teststrategie festgelegt sein)

Testprozess (5)

Testprotokollierung

- Testdurchführung exakt und vollständig protokollieren
- die Testdurchführung muss für nicht direkt Beteiligte (z.B. Kunden) nachvollziehbar sein
- Nachweis, dass geplante Teststrategie tatsächlich umgesetzt wurde
- zur Testdurchführung gehören neben Testobjekt auch Informationen und Dokumente: Testrahmen, Eingabedateien, Testprotokoll usw.
- Informationen und Daten sind so zu verwalten, dass eine Wiederholung des Tests unter gleichen Bedingungen möglich ist

Testprozess (6)

Testbewertung

- bei Abweichungen vom Sollergebnis → liegt eine Fehlerwirkung vor? (fehlerhafte Spezifikation, Testinfrastruktur, Testfälle)
- Fehlerklasse → Priorität der Fehlerbehebung
- nach Fehlerkorrektur: ist Fehler beseitigt? keine neuen Fehler hinzugekommen? ggf. neue Testfälle spezifizieren
- Fehler einzeln korrigieren und erneut testen
  * wünschenswert, um gegenseitige Beeinflussungen der Änderungen zu vermeiden
  * nicht praktikabel, wenn Tests nicht vom Entwickler sondern unabhängigen Testern ausgeführt wird

Testprozess (7)

Testende

- Testkonzept legt für jede Testmethode angemessenes Endekriterium fest (Risiko berücksichtigen), z.B. 90% aller Anweisungen testen (**Überdeckungsmaße**)
- ist ein Kriterium nicht erfüllt → weitere Tests
  * weiterer Aufwand gerechtfertigt? (z.B. Ausnahmesituation kann in Testumgebung nicht ausgelöst werden → andere Prüfverfahren wie statische Analyse)
  * zusätzliche Ressourcen notwendig → Testplanung ist zu überarbeiten
- **Fehlerfindungsrate:** Anzahl gefunder Fehler pro Teststunde (sollte im Projektverlauf gegen Null gehen) → Testende, wenn Rate unter einen gegebenen Wert sinkt

Testprozess (8)

in der Praxis

- Testzyklen einplanen → beim Test von Änderungen werden weitere Fehler entdeckt
- Zeit und Kosten führen zum Abbruch der Tests → Projektplanung stellt nicht ausreichend Ressourcen bereit oder Aufwand ist unterschätzt worden
- erfolgreiches Testen spart Geld ein: Fehler, die erst im Betrieb entdeckt werden, verursachen meist erheblich höhere Kosten als ein überzogenes Budget beim Testen
- Programme leben länger als erwartet → Testware für zukünftige Programmversionen oder ähnliche Projekte konservieren
- Evaluation des Testprozesses wird oft vernachlässigt

Testprozess (9)

Sollwerte

- Sollverhalten des Testobjekts ist vorab zu bestimmen
- Information aus geeigneten Quellen beschaffen
  → **Testorakel**
  * aus Eingabe auf Basis der Spezifikation ableiten
  * ausführbaren Prototyp werkzeuggestützt aus formaler Spezifikation erzeugen
    → wenig verbreitet, da formale Spezifikation selten
  * Testobjekt parallel von unabhängigen Entwicklern erstellen und Versionen mit gleichen Testdaten gegen die anderen testen (**Back-to-back-Test**)
    → sehr aufwändig, oft bei neuen Programmversionen

Testprozess (10)

weitere Orakel

- **Benutzungshandbuch** beschreibt alle Eingabemöglichkeiten und Reaktionen des Programms, auch auf fehlerhafte Eingaben
- **Diagramme und Modelle** in UML dienen als Grundlage für Testüberlegungen (meist nicht detailliert genug)
- System selbst dient als Orakel → **Nutzungsprofile**
  System von unterschiedlichen Personen in typischen Anwendungsfällen zum Sammeln von (Test-)Daten nutzen
- einfache Berechnungen vom Tester selbst durchführen
- Sollwerte berechnen mit Tabellenkalkulationsprogramm
- zusätzlich immer Plausibilitätsprüfungen anwenden

Testprozess (11)

Anmerkungen

- werkzeuggestützter Vergleich der Ist- und Sollwerte
- Sollwerte lassen sich nicht immer exakt vorhersagen (Ergebnisse liegen in einem Toleranzbereich)
- Erfahrung des Testers spielt wichtige Rolle beim Erstellen der Testfälle und Festlegen der Sollergebnisse
- meistens ist Fachwissen aus dem Anwendungsbereich notwendig, um Testfälle zu erstellen

Testprozess (12)

Priorisierung der Tests

- in der Praxis: Zeit zur Durchführung aller spezifizierten Testfälle nicht vorhanden → sinnvolle Auswahl an Tests treffen, um viele kritische Fehlerwirkungen zu finden
- Ziel: bei Abbruch des Tests zu einem beliebigen Zeitpunkt soll das bis dahin beste Ergebnis erreicht werden
- Gleichverteilung der Tests auf Testobjekte nicht sinnvoll: kritische Teile müssen gut getestet werden, bei nicht-kritischen Teilen keine Ressourcen verschwenden
- Kriterien für Priorisierung nicht allgemeingültig, sondern abhängig vom Projekt, Anwendungsbereich und von Kundenwünschen

Testprozess (13)

Kriterien für Priorisierung

- Fehlerschwere: Schaden im Fehlerfall minimieren
- Wahrscheinlichkeit des Fehlereintritts: Fehler in oft benutzten Funktionen treten mit höherer Wahrscheinlichkeit auf als Fehler in selten benutzten Funktionen
- Fehlerrisiko: Schaden im Fehlerfall multipliziert mit Eintrittswahrscheinlichkeit
- Wahrnehmung der Fehlerwirkung durch Endanwender: Fehler in GUI verunsichern so, dass alle Informationen in Frage gestellt werden
- Priorität der Anforderungen: Funktionen haben für Kunden unterschiedliche Bedeutung. Auf bestimmte Funktionalität kann verzichtet werden, auf andere nicht.

Testprozess (14)

Kriterien für Priorisierung (Fortsetzung)

- Qualitätsmerkmale mit hoher Bedeutung für Kunden müssen intensiv getestet werden
- Sicht der Entwicklung: Komponenten intensiv testen, deren Versagen schwer wiegende Auswirkungen haben
- Komplexität der Komponenten: komplizierte Systemteile intensiv testen, da Entwickler hier fehlerhaft programmiert haben. Aber: vermeintlich einfache Programmteile sind auch fehlerhaft, da Entwickler zu sorglos
- Projektrisiko: die Fehlerwirkungen, die eine aufwändige Überarbeitung erfordern/Ressourcen binden, müssen früh erkannt werden

Testprozess (15)

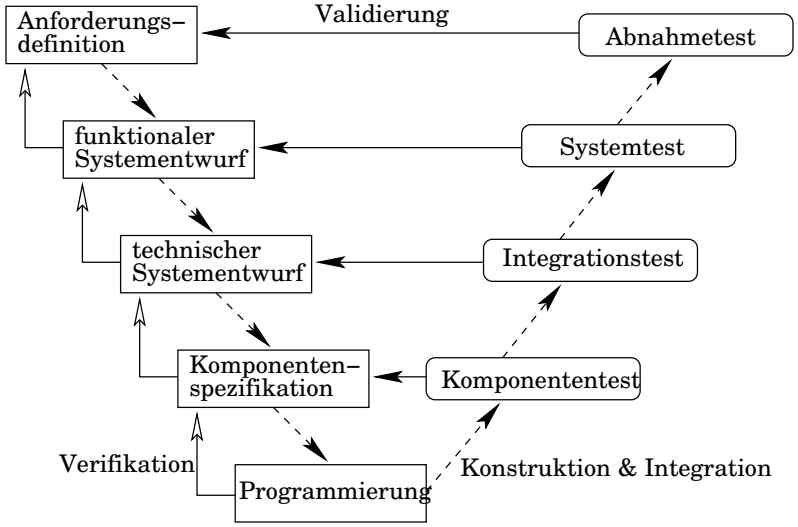
Psychologie des Testens

- Menschen machen Fehler, geben es aber nur ungern zu!
- Entwicklertest: eigene Arbeit kritisch prüfen → notwendiger Abstand zum eigenen Produkt vorhanden?
  - * eigene Arbeit wird zu optimistisch beurteilt → Test zu oberflächlich
  - * grundsätzlicher Designfehler (wg. falsch verstandener Aufgabe) → nicht durch Test erkennbar

  ⇒ paarweise zusammenarbeiten, Kollege testet
- unabhängiger Tester ist unvoreingenommen, mögliche Missverständnisse können gefunden werden, die Einarbeitung in Testobjekte kostet Zeit

Testen im Software-Lebenszyklus

allgemeines V-Modell:



Testen im Software-Lebenszyklus (2)

Anforderungsdefinition Wünsche und Anforderungen des Auftraggebers sammeln, spezifizieren und verabschieden.

Funktionaler Systementwurf Anforderungen auf Funktionen und Dialogabläufe des neuen Systems abbilden.

Technischer Systementwurf Technische Realisierung des Systems entwerfen (Definition der Schnittstellen, Zerlegung in Teilsysteme)

Komponentenspezifikation Für jedes Teilsystem werden Aufgabe, Verhalten, innerer Aufbau und Schnittstelle zu anderen Teilsystemen spezifiziert.

Programmierung Implementierung jedes Bausteins

Testen im Software-Lebenszyklus (3)

Komponententest Erfüllt jeder Baustein für sich die Vorgaben seiner Spezifikation?

Integrationstest Spielen Gruppen von Komponenten so zusammen, wie im technischen Systementwurf vorgesehen?

Systemtest Erfüllt das System als Ganzes die spezifizierten Anforderungen?

Abnahmetest weist das System aus Kundensicht die vertraglich vereinbarten Leistungsmerkmale auf?

Komponententest

- erste Teststufe: unmittelbar nach der Programmierphase erstellte Software-Bausteine werden erstmalig systematisch getestet
- die kleinsten Software-Bausteine sind abhängig von der Programmiersprache: Module, Units, Klassen
- Software-Baustein isoliert von anderen Komponenten des Systems prüfen → komponentenexterne Einflüsse ausschließen, eine Fehlerursache lässt sich eindeutig der Komponente zuordnen
- auch zusammengesetzte Bausteine testen, falls es zusammenhängende Einheit ist

Komponententest (2)

Testumgebung

- auf dieser Teststufe sehr entwicklungsnahe Arbeit
- Tester nutzt die Komponentenschnittstelle
  * versorge die Schnittstelle mit geeigneten Testdaten
  * erfasse die Reaktion der Komponente

  ⇒ Testtreiber notwendig
- **Testtreiber** Programm, dass Schnittstellenaufrufe absetzt und Reaktion des Testobjekts entgegennimmt
  * sinnvoll: lese Testfälle aus Datenbank, protokolliere Testdaten/Resultate mit Datum/Uhrzeit
  * Entwickler-Know-How notwendig

  ⇒ Komponententests werden oft vom Entwickler durchgeführt (→ Entwicklertest)

Komponententest (3)

einfacher Testtreiber

```
bool test_calcPrice() {
  double price;
  bool stat;

  // testcase 01
  price = calcPrice(10000.00, 2000.00, 1000.00, 3, 0);
  stat = stat && (abs(price - 12900.00) < 0.01);
  // testcase 02
  price = calcPrice(25500.00, 3450.00, 6000.00, 6, 0);
  stat = stat && (abs(price - 34050.00) < 0.01);

  ...
  return stat;
}
```


Komponententest (4)

Testziele

- stelle sicher, dass Testobjekt geforderte Funktionalität korrekt und vollständig realisiert
  (Funktionalität = Ein-/Ausgabeverhalten)
- typische Defekte, die aufgedeckt werden: Berechnungsfehler, fehlende/falsch gewählte Programmpfade
- im Zusammenspiel der Komponenten sind falsche Komponentenaufrufe nicht ausgeschlossen
  → **Test auf Robustheit:** verwende Methodenaufrufe, Daten und Sonderfälle, die laut Spezifikation unzulässig sind

Komponententest (5)

Erweiterung des Testtreibers

```
  // testcase 20
  price = calcPrice(-1000.00, 0.00, 0.00, 0, 0);
  stat = stat && (ERR_CODE == INVALID_PRICE);
  ...
  // testcase 30
  price = calcPrice("abc", 0.00, 0.00, 0, 0);
  stat = stat && (ERR_CODE == INVALID_ARGUMENT);
```


Komponententest (6)

Testziele (Fortsetzung)

- **Test der Effizienz** wird nur dort verifiziert, wo es in der Anforderungsdefinition angegeben ist
  * eingebettete Systeme: hardware-seitig nur limitierte Ressourcen
  * Echtzeitsysteme: vorgegebene Zeitschranken müssen garantiert werden
- **Test auf Wartbarkeit** Wieviel Aufwand kostet es den Entwickler, das Programm und dessen Kontext zu verstehen?

  Aspekte: Code-Struktur, Kommentare, Verständlichkeit und Aktualität der Dokumente → statischer Test

Komponententest (7)

Teststrategie

- Tester hat in der Regel Zugang zum Source-Code
- Entwickler kann Testfälle unter Ausnutzung des Wissens über Programmstrukturen, Methoden, Variablen entwerfen → White-Box-Test
- mit Debugger können fehlerhafte Werte gesetzt werden
- in der Praxis oft nur Black-Box-Test:
  * oft hunderte elementarer Komponenten → Einstieg in Code nur bei ausgewählten Komponenten praktikabel
  * elementare Programmbausteine zu größeren Einheiten zusammenfassen → Testobjekte zu groß, um Beobachtungen auf Code-Ebene vornehmen zu können

Integrationstest

- zweite Teststufe nach Komponententest: Testobjekte sind bereits getestet und Defekte sind möglichst schon korrigiert
- **Integration** Gruppen der Komponenten werden zu Teilsystemen verbunden
- Zusammenspiel aller Einzelteile miteinander testen → Fehlerzustände in Schnittstellen und im Zusammenspiel zwischen integrierten Komponenten finden
- warum noch Integrationstest, wenn Komponenten bereits getestet sind?
  Beispiel: Komponente berechnet den Preis richtig, aber die aufrufende Komponente übergibt falsche Parameter

Integrationstest (2)

Testobjekte

- Integration bedeutet, Einzelbausteine zu Einheiten zusammenzufassen → nach jedem Schritt Integrationstest
- jedes entstandene Teilsystem ist potentiell Basis für weitere Integration → Testobjekte können mehrfach zusammengesetzt sein
- selten: Software-System auf grüner Wiese entwickeln → vorhandenes System wird geändert, ausgebaut oder mit anderen Systemen gekoppelt
- oft: Systemkomponenten werden am Markt zugekauft (Standardprodukte)
- Alt- und Standardkomponenten werden im Komponententest nicht berücksichtigt → Integrationstest

Integrationstest (3)

Testumgebung

- Testobjekte sind zusammengesetzte Komponenten, die keine anderen Schnittstellen nach außen besitzen, als die Einzelkomponenten

⇒ Testtreiber des Komponententests wiederverwenden

- Schnittstellenaufrufe und Datenverkehr werden über die Testtreiberschnittstelle getestet
  → zusätzliches Diagnoseinstrument: Monitor
- **Monitor** Programm, das Datenbewegungen zwischen Komponenten mitliest und protokolliert (z.B. Netzwerk-Sniffer)

Integrationstest (4)

Testziel Schnittstellenfehler finden: Formate passen nicht, Dateien fehlen oder Entwickler hat System in Komponenten aufgeteilt, die nicht spezifiziert waren

typische Fehler beim Datenaustausch:

- Komponente übermittelt keine oder syntaktisch falsche Daten → empfangende Komponente kann nicht arbeiten oder stürzt ab (funktionaler Fehler, Protokollfehler)
- Komponenten interpretieren die Daten unterschiedlich (widersprüchliche oder fehlinterpretierte Spezifikation)
- Daten werden zwar richtig, aber zum falschen Zeitpunkt (Timing-Probleme) oder in zu kurzen Zeitintervallen übergeben (Durchsatz-, Kapazitäts- oder Lastproblem)

Integrationstest (5)

Kann auf Komponententest verzichtet werden?

- leider in der Praxis oft anzutreffen
- gravierende Nachteile:
  * viele Fehlerwirkungen sind durch funktionale Fehlerzustände einzelner Komponenten verursacht → impliziter Komponententest in ungeeigneter Testumgebung (Zugang zu Einzelkomponente erschwert)
  * kein geeigneter Zugang zu Einzelkomponenten → einige Fehlerwirkungen können nicht provoziert werden
  * Entstehungsort und Ursache einer Fehlerwirkung ist nur schwer einzugrenzen

⇒ schlechtere Fehlerfindungsrate und erhöhter Diagnoseaufwand durch Verzicht auf Komponententest

Integrationstest (6)

Teststrategie In welcher Reihenfolge sollen Komponenten integriert werden, damit notwendige Testarbeiten effizient durchführbar? Effizienz = Testkosten / Testnutzen

Problem in der Praxis: Komponenten werden zu unterschiedlichen Zeiten fertig → **ad-hoc Strategie** Prüfe, ob fertig gestellte Komponente mit anderen, schon geprüften Komponenten integriert werden kann.

Stub: noch nicht implementierte Komponente für Testdurchführung durch Platzhalter ersetzen/simulieren

je früher mit Integrationstest begonnen wird, umso mehr Zeit muss in Erstellung von Stubs investiert werden

Integrationstest (7)

optimale Strategie hängt von Randbedingungen ab und ist in jedem Projekt individuell zu analysieren

- Systemarchitektur bestimmt, aus welchen Komponenten das System besteht und wie diese voneinander abhängig sind
- Projektplan legt fest, zu welchen Zeitpunkten einzelne Teile entwickelt werden und testbereit sein sollten
- Testkonzept bestimmt, wie intensiv einzelne Aspekte des Systems zu testen sind

Integrationstest (8)

Top-down Integration Beginne mit der Komponente, die selbst nicht aufgerufen wird (außer vom Betriebssystem) → untergeordnete Komponenten durch Stubs ersetzen

sukzessive Komponenten niedrigerer Systemschichten integrieren → höhere Schichten dienen als Testtreiber

- **Vorteil:** einfache Testtreiber, da übergeordnete Komponenten bereits getestet
- **Nachteil:** viele Stubs für untergeordnete, noch nicht integrierte Komponenten

Integrationstest (9)

Bottom-up Integration Beginne mit elementaren Komponenten des Systems, die keine weiteren Komponenten aufrufen (außer vom Betriebssystem)

Teilsysteme werden sukzessive aus getesteten Komponenten zusammengesetzt

- **Vorteil:** keine Stubs
- **Nachteil:** übergeordnete Komponenten müssen durch Testtreiber simuliert werden

Integrationstest (10)

Ad-hoc Integration Bausteine in Reihenfolge der Fertigstellung integrieren

- **Vorteil:** Zeitgewinn, da jeder Baustein frühestmöglich in seine Umgebung integriert wird
- **Nachteil:** Stubs und Testtreiber notwendig

Systemtest

- dritte Teststufe: prüfe, ob spezifizierte Anforderungen vom Produkt erfüllt werden
- warum noch notwendig nach Integrationstest?
  * bisher nur Test gegen technische Spezifikation, jetzt Test aus Perspektive des Kunden (Vorgriff auf Abnahmetest)
  * einige Funktionen/Eigenschaften resultieren aus Zusammenspiel *aller* Systemkomponenten
- im Fallbeispiel: Fahrzeug wird konfiguriert (`DreamCar`), Finanzierung und Versicherung werden kalkuliert (`EasyFinance`, `NoRisk`), Bestellung wird übermittelt (`JustInTime`) und Verträge werden archiviert (`ContractBase`)
  → alle Komponenten müssen korrekt zusammenspielen

Systemtest (2)

Testumgebung

- muss späterer Produktivumgebung sehr nahe kommen
- statt Testtreiber und Stubs sollen tatsächlich verwendete Hard- und Software-Produkte installiert sein
- **Nie in Produktivumgebung des Kunden testen!**
  * nach Murphies Gesetz: Es werden Fehlerwirkungen auftreten! → Gefahr, dass Produktivumgebung beeinträchtigt wird (mehrere Autos können nicht verkauft werden, Image-Verlust)
  * Tester haben keine Kontrolle über Systemparameter und Konfiguration → Tests sind nicht reproduzierbar

Systemtest (3)

Testziel
Prüfe, ob System die gestellten Anforderungen erfüllt.

Unterscheide zwei Klassen:

- **funktionale Anforderungen** spezifizieren das Verhalten, welches das System erbringen muss (Angemessenheit, Richtigkeit, Interoperabilität, Ordnungsmäßigkeit, Sicherheit)
- **nicht-funktionale Anforderungen** beschreiben Attribute des funktionalen Verhaltens (wie gut soll die Funktion erbracht werden) → beeinflusst, wie zufrieden der Kunde/Anwender ist (Merkmale: Zuverlässigkeit, Benutzbarkeit, Effizienz, Änderbarkeit, Übertragbarkeit)

Systemtest (4)

Anforderungsbasiertes Testen

- die Anforderungen des Auftraggebers werden in der Projektphase *Anforderungsdefinition* gesammelt und im Anforderungsdokument niedergeschrieben (oft als Lasten- oder Pflichtenheft bezeichnet)
- das Anforderungsdokument wird durch Review verifiziert und freigegeben (wird Bestandteil der Verträge zwischen Kunde und Software-Hersteller)
- zu jeder Anforderung mindestens einen Testfall ableiten und in Systemtestspezifikation dokumentieren
- Systemtestspezifikation ist durch Review zu verifizieren
- in der Regel mehr als ein Testfall pro Anforderung

Systemtest (5)

Anforderungen im Fallbeispiel

A 100 Der Anwender kann ein Fahrzeugmodell aus dem jeweils aktuellen Modellprogramm zur Konfiguration auswählen.

A 101 Für ein ausgewähltes Modell wird die lieferbare Zusatzausstattung angezeigt. Aus dieser kann die individuelle Wunschausstattung zusammengestellt werden.

A 102 Zu der jeweils ausgewählten Konfiguration wird laufend der Gesamtpreis gemäß aktueller Preisliste berechnet und angezeigt.

Systemtest (6)

Anforderungsbasiertes Testen im Fallbeispiel

T 102.1 Ein Modell wird ausgewählt; dessen Grundpreis wird gemäß Verkaufshandbuch angezeigt.

T 102.2 Es wird eine Zusatzausstattung ausgewählt, der Fahrzeugpreis erhöht sich um den Preis des Zubehörs.

T 102.3 Es wird eine Zusatzausstattung abgewählt, der Fahrzeugpreis sinkt entsprechend.

T 102.4 Es werden drei Zusatzausstattungen ausgewählt; die Rabattierung erfolgt gemäß Spezifikation.

Sind alle definierten Testfälle fehlerfrei gelaufen, wird das System als validiert betrachtet!

Systemtest (7)

Geschäftsprozessbasiertes Testen

- gut geeignet, wenn das System einen Geschäftsprozess des Kunden automatisieren oder unterstützen soll
- Geschäftsprozessanalyse (Teil der Anforderungsanalyse) zeigt, welche Prozesse relevant sind, wie oft und in welchem Kontext sie vorkommen, welche Personen bzw. Fremdsysteme beteiligt sind usw.
- Testszenarien bilden typische Geschäftsvorfälle ab
- Priorität richtet sich nach Häufigkeit und Relevanz der entsprechenden Geschäftsprozesse

Systemtest (8)

Geschäftsprozess im Fallbeispiel

- Kunde wählt aus verfügbaren Modellen einen Typ aus
- er informiert sich zu diesem Typ über Ausstattungen und Preise und entscheidet sich für Wunschfahrzeug
- Verkäufer schlägt Finanzierungsalternativen vor
- Kunde entscheidet sich und schließt Vertrag ab

Systemtest (9)

Anwendungsfallbasiertes Testen

- wie geht der Anwender mit dem System um, welche Aktionen führt er typischerweise aus?
- verschiedene Anwendergruppen haben eigene Benutzerprofile (z.B. Käufer Kleinwagen, Käufer Mittelklasse, Nicht-Käufer)
- typische Aktionsmuster (Anwendungsfälle) in typischer Häufigkeit identifizieren und Testfälle ableiten
- Priorität anhand der Häufigkeit festlegen
- im Fallbeispiel: Anwender will Autos auswählen/kaufen

  wichtig für Akzeptanz: leicht bedienbar, schnelle Antwortzeit, übersichtliche Ausgabe

Systemtest (10)

Test nicht-funktionaler Anforderungen

- **Lasttest:** Messen des Systemverhaltens in Abhängigkeit steigender Systemlast (parallel arbeitende Anwender, Anzahl Transaktionen)
- **Performanztest:** die Verarbeitungsgeschwindigkeit und Antwortzeit für bestimmte Anwendungsfälle messen, oft in Abhängigkeit steigender Last
- **Volumen-/Massentest:** Systemverhalten in Abhängigkeit zur Datenmenge beobachten (z.B. Verarbeitung sehr großer Dateien)
- **Stresstest:** beobachte Systemverhalten bei Überlast
- **Test der (Daten-)Sicherheit** gegen unberechtigten Systemzugang oder Datenzugriff

Systemtest (11)

Test nicht-funktionaler Anforderungen (Fortsetzung)

- **Test der Stabilität/Zuverlässigkeit** im Dauerbetrieb (z.B. Ausfälle pro Betriebsstunde bei gegebenem Benutzerprofil)
- **Test auf Robustheit** gegenüber Fehlbedienung, Hardware-Ausfall usw. sowie Prüfung der Fehlerbehandlung und des Wiederanlaufverhaltens
- **Test auf Kompatibilität/Datenkonversion:** Prüfung der Verträglichkeit mit vorhandenen Systemen. Import und Export von Datenbeständen usw.
- **Test unterschiedlicher Konfigurationen** des Systems, z.B. unterschiedliche Betriebssystemversionen, Landessprache, Hardware-Plattform usw.

Systemtest (12)

Test nicht-funktionaler Anforderungen (Fortsetzung)

- **Test auf Benutzerfreundlichkeit:** prüfen auf Erlernbarkeit und Angemessenheit der Bedienung; Verständlichkeit der Systemausgaben bezogen auf die Anwendergruppe
- **Prüfen der Dokumentation** auf Übereinstimmung mit dem Systemverhalten (z.B. Bedienungsanleitung, GUI)
- **Prüfen auf Änderbarkeit/Wartbarkeit:** Verständlichkeit und Aktualität der Entwicklerdokumente, modulare Systemstruktur usw.

nicht-funktionale Anforderungen sind oft zu schwammig formuliert: „Das System soll schnell reagieren“ oder „leicht bedienbar sein“ → nicht testbar!

Systemtest (13)

Test nicht-funktionaler Anforderungen (Fortsetzung)

- Systemtester sollten am Review des Anforderungsdokuments teilnehmen und darauf achten, dass jede nicht-funktionale Anforderung testbar formuliert wird!
- auch *selbstverständliche*, nicht spezifizierte Anforderungen müssen validiert werden
  z.B. Aussehen und Bedienmechanismus (Look&Feel) muss Stil und Konventionen der Betriebssystemumgebung entsprechen
- auf vorhandene funktionale Tests zurückgreifen und die nicht-funktionale Größe messen (z.B. möglich bei Last- oder Performanztest): gemessener Wert unterhalb eines Grenzwerts → Test bestanden

Systemtest (14)

Problem in der Praxis: unklare Kundenanforderungen

- existieren keine Anforderungen → jedes Systemverhalten ist zulässig bzw. nicht bewertbar
- Anforderungen nur in Köpfen der Anwender/Kunden → Tester muss gewünschtes Sollverhalten nachträglich zusammentragen
- in der Regel: zu ein und derselben Sache existieren verschiedene Ansichten und Vorstellungen → Klärungs- und Entscheidungsprozesse viel zu spät, zeit- und kostenintensiv, Fertigstellung verzögert sich
- Entwicklern fehlen klare Ziele → konstruiertes System erfüllt implizite Kundenanforderungen nicht, Systemtest stellt Scheitern des Projekts offiziell fest

Abnahmetest

- **bisher:** Testarbeiten in Verantwortung des Herstellers, bevor Software an Kunden übergeben wird
- **jetzt:** abschließender Test aus Sicht des Kunden:
  * Kunde kann Test direkt nachvollziehen
  * Kunde ist direkt beteiligt/verantwortlich
  * Urteil des Kunden steht im Vordergrund

Abnahmetest (2)

Test auf vertragliche Akzeptanz

- Testkriterien: im Entwicklungsvertrag festgeschriebene Abnahmekriterien (klar und eindeutig formulieren)
- in der Praxis: im Systemtest Abnahmekriterien prüfen und Testfälle erstellen, beim Abnahmetest nur die relevanten Testfälle wiederholen
- der Kunde muss die Akzeptanztestfälle selber entwerfen oder sorgfältigem Review unterziehen (denn Software-Hersteller kann Akzeptanzbedingungen falsch verstanden haben)
- Test findet in Systemumgebung (nicht in der Produktivumgebung!) des Kunden statt → aufgrund unterschiedlicher Testumgebungen kann Testfall fehlschlagen

Abnahmetest (3)

Test auf Benutzerakzeptanz

- zu empfehlen, wenn Kunde und Anwender des Systems verschiedene Personengruppen sind → Akzeptanz jeder Gruppe sicherstellen
- obwohl System funktional in Ordnung ist, kann Systemeinführung an mangelnder Akzeptanz scheitern (z.B. zu umständlich zu bedienen) → frühzeitig Prototypen vorstellen
- Umfang des Akzeptanztests risikoabhängig:
  * Individual-Software: umfassender Test
  * Standard-Software: Paket installieren und repräsentative Anwendungsszenarien testen
- Schnittstellen zu anderen Systemen testen

Abnahmetest (4)

Feldtest

- der Software-Hersteller kann den Systemtest nicht für alle möglichen Produktivumgebungen durchführen (oft bei Standard-Software)
- Einflüsse aus nicht vollständig bekannten/spezifizierten Umgebungen erkennen und ggf. beheben
- **Test durch repräsentative Kunden:** (Beta-Test)
  * Hersteller liefert stabile Vorabversionen an ausgewählten Kundenkreis, dessen Produktivumgebungen die verschiedenen möglichen Umgebungen gut abdecken
  * Kunde führt Testszenarien vom Hersteller durch oder setzt vorläufiges Produkt probehalber ein und liefert Fehlermeldungen an Hersteller zurück

Test neuer Produktversionen

- Software-Entwicklung ist nach bestandenem Abnahmetest und Auslieferung nicht beendet: Software ist oft Jahre im Einsatz und wird vielfach korrigiert, geändert und erweitert → neue Produktversion
- **Software-Wartung:** Software altert nicht, aber Produkt wird an geänderte Einsatzbedingungen angepasst, Defekte werden beseitigt
- **geplante Weiterentwicklung:** aus Kosten- oder Zeitgründen zurückgestellte Funktionalität umsetzen, Erweiterungen im Zuge geplanter Marktausdehnung

⇒ **Regressionstest:** erneuter Test eines bereits getesteten Programms nach Wartung/Weiterentwicklung

Test neuer Produktversionen (2)

Software-Wartung im Fallbeispiel:

- einige Händler betreiben System mit veraltetem Betriebssystemstand → Ausfälle beim Host-Zugriff
- Auswahl der Zusatzausstattung zu umständlich, wenn das System für Preisvergleiche genutzt wird → Ausstattungskombinationen zwischenspeichern und nach Änderung wieder zurückholen
- seltene Versicherungstarife können nicht berechnet werden, da Kalkulationsvorschriften nicht implementiert
- bei Fahrzeugbestellung nach 15 Minuten kein Bestellvorgang vom Werkrechner bestätigt, System trennt Verbindung, um unbenutzte Verbindungen zu vermeiden → Händler muss Vorgang wiederholen, Kunde ist verärgert

Test neuer Produktversionen (3)

typische Wartungsanlässe:

- System wird unter neuen/nicht geplanten Einsatzbedingungen betrieben
- neue Kundenwünsche
- Funktionen für seltene Sonderfälle wurden vergessen
- sporadische Ausfälle (oft durch externe Einflüsse)

Trotz Wartung darf nicht auf Komponenten-, Integrations- und Systemtests verzichtet werden, frei nach dem Motto: „Da wir ständig neue Versionen rausbringen, ist es nicht so schlimm, wenn wir Fehler übersehen.“

Test neuer Produktversionen (4)

geplante Weiterentwicklung im Fallbeispiel:

- Host im Rechenzentrum erhält neue Kommunikationssoftware → entsprechendes Modul in `JustInTime` muss angepasst werden
- verschiedene Systemerweiterungen, die zum ersten Liefertermin nicht fertig gestellt werden konnten
- Ausdehnung auf europäisches Händlernetz → landesspezifische Anpassungen, Bedientexte und Handbücher übersetzen

Iterative Software-Entwicklung ist heute der Regelfall!
→ Regressionstest

Test neuer Produktversionen (5)

Regressionstest

- bereits getestetes Programm nach Modifikation prüfen, ob durch Änderungen neue Defekte eingebaut bzw. bisher maskierte Fehlerzustände freigelegt wurden
- Umfang der Regressionstests:
  1. Fehlernachtest
  2. Test geänderter Funktionalität
  3. Test neuer Funktionalität
  4. vollständiger Regressionstest
- Änderungen können unerwartete Seiteneffekte haben → Fehlernachtest (1) und Tests am Ort der Modifikation (2, 3) reichen nicht aus!

Test neuer Produktversionen (6)

Regressionstest (Fortsetzung)

- vollständiger Regressionsttest ist notwendig, aber in der Praxis zu zeit- und kostenintensiv
- Abwägen zwischen niedrigen Kosten und hohem Risiko → Auswahlstrategien:
  * wiederhole die Tests, denen hohes Risiko zugeordnet wurde
  * bei funktionalen Tests Verzicht auf Sonderfälle
  * Einschränkung auf bestimmte Konfigurationen (z.B. englischsprachige Produktversion auf bestimmter Betriebssystemumgebung)
  * einschränken auf bestimmte Teilsysteme

Glossar

Testbasis Alle Dokumente, aus denen Anforderungen an das Testobjekt ersichtlich werden

Testrahmen Sammlung aller Programm (u.a. Testtreiber und Platzhalter), die notwendig sind, um Testfälle auszuführen, auszuwerten und Testprotokolle aufzuzeichnen

Testmethoden

Übersicht:

- verifizieren: Verifikation, symbolische Ausführung
- analysieren: Metriken, Analyse von Anomalien
- statischer Test: Inspektion, Review, Walkthrough
- dynamischer Test:
  * funktional (black-box): entspricht das Programm der Spezifikation?
  * strukturell (white-box): kontrollfluss- oder datenfluss-orientiert
  * disversifizieren: vergleiche Programmvarianten

Testmethoden (2)

Ziel: Korrektheit des Programms sicherstellen

- entspricht das Programm der Spezifikation?
- entspricht das Programm den Anforderungen der User?

verifizierende Verfahren:

- **Verifikation:** teste mit Hilfe mathematischer Mittel die Konsistenz zwischen Spezifikation und Implementierung einer Systemkomponente
  $\Rightarrow$ präzise, vollständige Spezifikation erforderlich
- **symbolische Ausführung:** teste den Quelltext mit allgemeinen symbolischen Eingabewerten durch einen Interpreter

Testmethoden (3)

analysierende Verfahren:

- Eigenschaften von Systemkomponenten vermessen und darstellen $\Rightarrow$ You can't control what you can't measure!
- Metriken stellen Eigenschaften (wie strukturelle Komplexität, Programmlänge, Anzahl Kommentare und Vererbungstiefe) quantitativ dar und erlauben daher einen Quervergleich mit bisher ermittelten Maßzahlen
- Analyse von Anomalien (z.B. Datenflussanomalie: eine Variable wird verwendet, obwohl noch keine Zuweisung erfolgte)

Testmethoden (4)

dynamische Testverfahren:

- übersetztes und ausführbares Programm wird mit konkreten Eingabewerten ausgeführt
- Programm wird in möglichst realer Umgebung getestet

$\Rightarrow$ Stichprobentests können nicht die Korrektheit des Programms beweisen

statische Testverfahren:

- Systemkomponenten werden nicht zur Laufzeit sondern anhand des Quellcodes direkt (auf dem Papier) getestet
- Arten: Inspektion, Review und Walkthrough

Testmethoden (5)

disversifizierender Test:

- vergleiche die Testergebnisse verschiedener Programmversionen (Regressionstest)
- Mutationen-, Pertubationen- und Back-To-Back-Test

funktionaler Test (black-box):

- erstelle Testfälle anhand der Spezifikation der Systemkomponenten
- Arten: Äquivalenzklassenbildung, Grenzwertanalyse, Ursache-Wirkungsanalyse

Testmethoden (6)

struktureller Test (white-box):

- leite Vollständigkeit einer Menge von Testfällen anhand des Kontroll- oder Datenflusses ab
- Kontrollflussorientiert: Anweisungs-, Zweig-, Pfad- und Bedingungsüberdeckungstest
- Datenflussorientiert: Defs/Uses-Verfahren, required K-Tupel-Test und Datenkontextüberdeckungsverfahren