Softwaretest für Entwickler

Methoden für Modultest und Integrationstest

Gesamtinhaltsverzeichnis

1

Grundlagen und Begriffsdefinitionen

1 Grundlagen und Begriffsdefinitionen

1.1 Problemstellung

Software hat in den letzten Jahren eine enorme Verbreitung gefunden. Es gibt kaum noch Geräte, Maschinen oder Anlagen, in denen die Steuerung nicht über Software bzw. Softwareanteile realisiert wird. Software trägt somit ganz entscheidend zum Funktionieren der Geräte und Anlagen bei.

Ebenso ist der reibungslose Ablauf eines Betriebs oder einer Organisation weitgehend von der Zuverlässigkeit der Softwaresysteme abhängig, die zur Abwicklung der Geschäftsprozesse oder einzelner Aufgaben eingesetzt werden.

Für viele Organisationen spielt deshalb die Qualität von Softwaresystemen eine immer größere Rolle. Es werden zunehmend Maßnahmen ergriffen, um eine höhere Qualität zu erreichen. Trotz ermutigender Resultate mit verschiedenen Ansätzen zur Qualitätssicherung ist die IT-Branche weit davon entfernt, fehlerfreie Software entwickeln zu können. Die Entwicklung von Softwaresystemen ist nach wie vor ein schwieriges Handwerk und auf unabsehbare Zeit kaum ohne Fehler zu meistern. Die Ursachen für Fehler sind vielfältig und schwer vorhersehbar.

Ein Beispiel aus der Praxis: Überweisungsspanne im Finanzamt

„Ein Software-Fehler hat der Hamburger Finanzbehörde zusätzliche Arbeit für Wochen beschert. 32 000 Überweisungen in einer Gesamthöhe von 65 Millionen Euro können von der SAP-Software der Behörde nicht ihren Absendern zugeordnet werden. (...)

Am 23. September seien erstmals Überweisungen von Bürgern beim Transfer von der HSH-Software auf die Finanzbehörden-Software nicht mehr nach Einzelbuchungen unterteilt aufgelaufen, sondern als Gesamtsumme, so Behörden-Sprecher Burkhard Schlesies. Unmittelbar davor habe die HSH, die aus den vereinten Landesbanken von Hamburg und Schleswig-Holstein hervorgegangen ist, fusionsbedingte Änderungen an ihrer Software vorgenommen. Betroffen seien rund 15 Prozent des Zahlungsverkehrs, der mit einem Software-Modul der Firma SAP erfasst wird. (...)

Die nun aufgelaufenen 32 000 Überweisungen seien anhand der Buchungsbelege zurückzuverfolgen und von der HSH in Excel-Tabellen ausgedruckt worden. Mit dem Abarbeiten dieser Tabellen hätten die Behörden-Mitarbeiter nun bis "sicher weit in den Januar" hinein zu tun. Auch Samstagsarbeit sei nicht ausgeschlossen.“

Quelle:http://www.welt.de/print-welt/article280174/Software_Panne_beim_Finanzamt.html

1.2 Qualität und Fehler in der Software-Entwicklung

Die Qualität eines Software-Produkts hängt nicht nur von der Programmierung ab. Zum Zeitpunkt der Programmierung ist es bereits zu spät und fast aussichtslos, eklatante Fehler oder Mängel aus einem Software-Produkt durch Testen zu beseitigen, Qualität lässt sich nicht in Software hineintesten.

Selbst wenn es noch nicht zu spät sein sollte, Fehler zu beheben, so entstehen doch gewaltige Kosten, wenn Mängel in der Software erst bei der Programmierung oder beim Praxiseinsatz entdeckt und behoben werden.

Fehler oder Mängel sollten also zu einem möglichst frühen Zeitpunkt bei der Software-Entwicklung entdeckt und behoben werden. Dies betrifft selbstverständlich nur solche Fehler, die bereits in der Analyse- oder Spezifikationsphase der Software gemacht wurden.

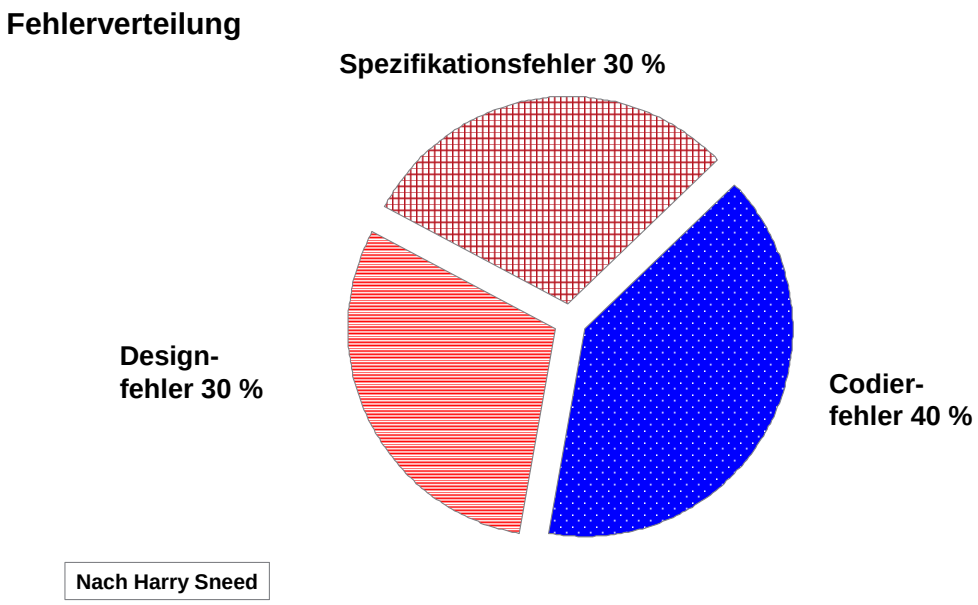


Abbildung 1-1: Fehlerverteilung

Untersuchungen bei großen amerikanischen Konzernen haben jedoch gezeigt, dass der überwiegende Teil der Fehler in der Konzeptionsphase der Software entsteht und nicht bei der Programmierung. Mehr als 60 % der Software-Fehler sind konzeptioneller Natur und weniger als 40 % sind Programmierfehler.

Analysiert man am Ende eines Projektes die Kosten, die durch die Behebung von Fehlern im Projektverlauf entstehen, so stellt man "natürlich" fest: Je später ein Fehler entdeckt wird desto teurer ist dessen Behebung.

Kaum jemand würde jedoch vermuten, dass dieser Unterschied so groß ist, dass die Korrektur eines Fehlers, der erst im Betrieb gefunden wird, 150 Mal so teuer ist, wie das Aufspüren dieses Fehlers bereits in Konzeptphase.

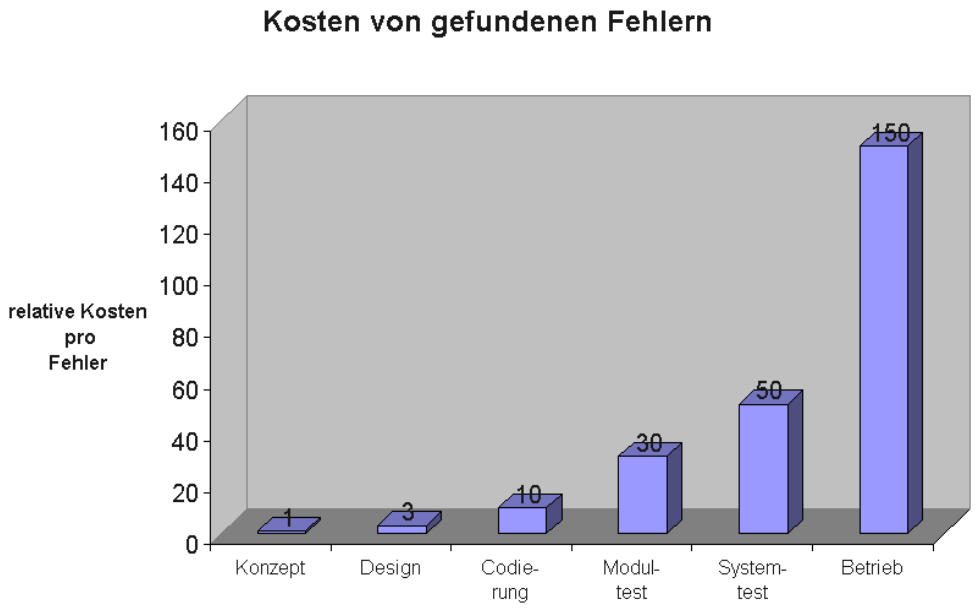


Abbildung 1-2: Kosten von gefundenen Fehlern

Legt man beispielsweise ein Phasenkonzept zugrunde, so stellt sich die Forderung, die Ergebnisse der Konzeptionsphasen (Pflichtenheft, Fachkonzept und DV-Konzept) einer intensiven Qualitätsprüfung zu unterziehen, um hier bereits Mängel entdecken und beheben zu können.

1.3 Kritische Projektfaktoren

Fast jede Software-Entwicklung erfolgt unter einschränkenden wirtschaftlichen und zeitlichen Rahmenbedingungen. Diesen Restriktionen stehen andererseits meist hohe Qualitätsanforderungen gegenüber, deren Erfüllung die Auftraggeber Seite fordert.

Das so entstehende Spannungsverhältnis von beschränkter Zeit, beschränkten Kosten und nicht ausreichend zur Verfügung stehenden qualifizierten Mitarbeitern einerseits und guter Qualität andererseits liegt vielen Software-Projekten zugrunde.

Aus zahlreichen Projekten lassen sich folgende Probleme zu den vier Faktoren „Termin / Zeit“, „Budget / Kosten“, „Projektmitarbeiter“ und „Qualität“ anführen:

Faktor Termin / Zeit:

- Nur 5% aller Projekte werden termingerecht fertig
- Mehr als 60% der Projekte sind 20 und mehr Prozent im Verzug
- Viele Projekte werden wegen Terminverzug ganz aufgegeben.
- Durch Terminverzug können Marktchancen für ein Produkt verschlechtert und dadurch Investitionen unrentabel werden.

Faktor Budget / Kosten

- Die Entwicklungskosten steigen mit zunehmender Komplexität der zu erstellenden Software exponentiell. Der hohe Integrationsgrad moderner Softwaresysteme, die daraus resultierenden komplexen Schnittstellen zwischen den Komponenten und die Forderungen nach ausreichender Benutzerfreundlichkeit und Zuverlässigkeit erhöhen ebenfalls die Entwicklungskosten.
- Die Wartungskosten spielen eine entscheidende Rolle. In der Gesamtschau zeigt sich, dass nicht selten über 60% der gesamten Softwarekosten eines Produktes für Wartung ausgegeben werden.

Faktor Mensch

- Aufgrund der Tatsache, dass die Anwendungen immer komplexer werden, auf der anderen Seite aber immer höhere Anforderungen an die Sicherheit und die Zuverlässigkeit der Systeme gestellt werden, wächst die Verantwortung für jeden Projektmitarbeiter.
- Belastend für alle Projektmitarbeiter können sich des weiteren folgende Faktoren auswirken:
  - Immer kürzer werdende Entwicklungszyklen
  - Ständig wachsender Termindruck
  - Hoher Abstimmungsbedarf zwischen Projektteams
- Die Folge ist, dass bei Ressourcen-Engpässen in der Regel das Projektteam nicht durch zusätzliche Mitarbeiter aufgestockt werden kann (stehen nicht zur Verfügung), sondern dass von den vorhandenen Projektmitarbeitern noch mehr verlangt wird (Überstunden, Wochenendarbeit, nicht Genehmigung von Urlaub in „heißen“ Projektphasen etc).

Faktor Qualität

- Fehler werden oft zu spät gefunden, vielfach erst in der Betriebsphase beim Anwender.
- Die Dokumentation eines Softwareproduktes fehlt, ist unvollständig oder nicht aktuell.
- Wegen Produktmängel werden 50% des Entwicklungsaufwandes oder oft auch mehr für Fehlersuche und Fehlerbehebung aufgewendet.
- Qualität als Entwicklungsziel ist wegen fehlender Qualitätsplanung oft nicht nachweisbar.
- Geplante und notwendige Testaktivitäten werden mangels Zeit, Budget oder Ressourcen nicht oder nicht ausreichender Form durchgeführt.

Fazit:

Häufig treten in auch gut geplanten Projekten nicht erwartete Probleme auf, die dazu führen, dass der zugesagte Ende Termin in Gefahr gerät. Aufgabe des Projektleiters ist es dann, im Rahmen der Projektsteuerung entsprechende Maßnahmen einzuleiten, welche die Gefahr des Terminverzugs minimieren.

Dies bedeutet:

Betrachtung der oben beschriebenen Projektfaktoren und Analyse, an welchem Faktor „gedreht“ werden kann.

Leider zeigt sich hier in der Praxis, dass sehr oft der Faktor „Qualität“ vom Projektleiter herangezogen wird, um die zeitlichen Probleme zu lösen und ein Scheitern des Projektes zu verhindern.

Immer wieder eingesetzte Maßnahmen sind beispielsweise:

- Verringern des zugesagten Leistungsumfangs (bestimmte Funktionen werden zu einem späteren Zeitpunkt realisiert)
- Vernachlässigung oder ggf. kompletter Verzicht auf Dokumentation (diese kann auch nach Projektende noch erstellt werden!?)
- Verkürzung oder Streichung von Testaktivitäten (es wird nicht solange getestet, bis alle Testfälle durchgeführt wurden, sondern es wird solange getestet, bis die Zeit zu Ende ist)

Damit ist zwar unter Umständen das Projekt kurzfristig gerettet, aber:

- Der Kunde erhält nicht das, was er ursprünglich wollte
- Der Kunde ist mit der Qualität der gelieferten Funktionen nicht zufrieden

Die Nachbesserungen und die langfristig zu betrachtenden Wartungskosten sind enorm.

1.4 Einordnung Test in Entwicklungsprozess

1.4.1 Vorgehensmodelle

In der Regel werden von den Unternehmen Software-Projekte nach entsprechenden Vorgehensmodellen (allgemein gültigen oder Unternehmens-spezifischen) abgewickelt. **Die Aktivität „Test“ steht häufig am Ende der Entwicklung**

Dabei spielt es grundsätzlich keine Rolle, nach welcher Art von Vorgehensmodell gearbeitet wird (Wasserfall-Modell, Iterativer und/oder inkrementeller Ansatz etc.).

Nachfolgend als Beispiel **das klassische Phasenmodell:**

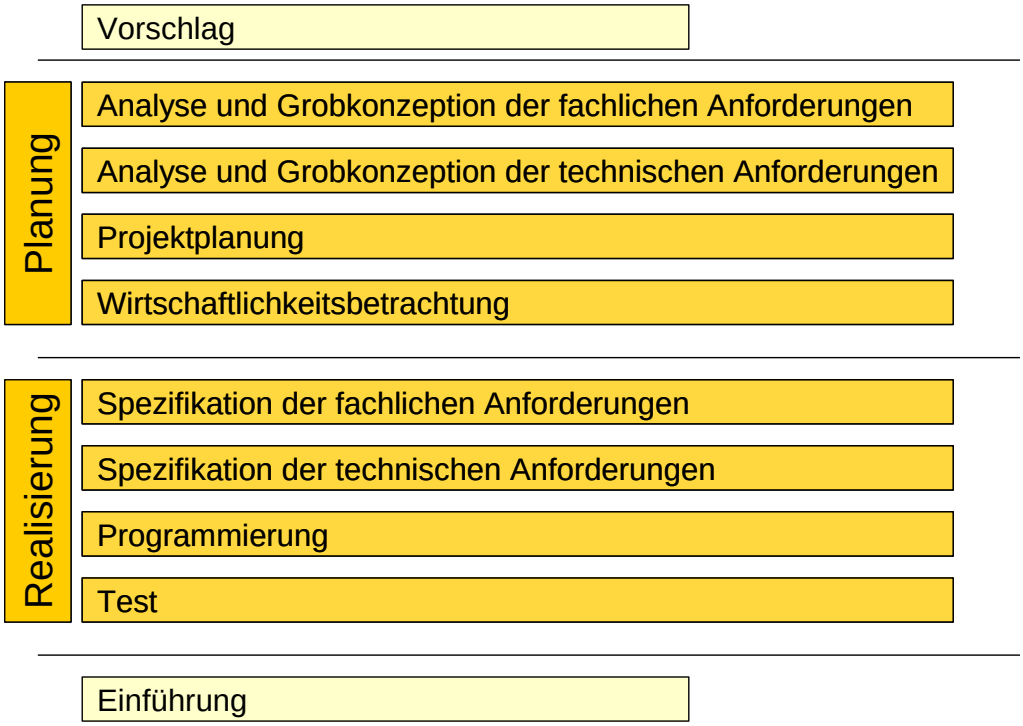


Abbildung 1-3: Allgemeines Phasenmodell

1.4.2 Erläuterung der Phasen

Phase	Inhalt
Vorschlag	**Anlass für Initiierung des Projektes.** Dies kann beispielsweise sein: • Gesetzliche Änderungen • Neue Kundenanforderungen • Migration (z.B. Hardware- oder Plattformwechsel)
Planung	**Analyse und Grobkonzeption der fachlichen Anforderungen** • Ggf. Durchführung einer Ist-Analyse • Sammlung und grobe Beschreibung der fachlichen Anforderungen (Neue Anforderungen oder Änderungen) • Analyse und Bewertung der nicht-funktionalen Anforderungen
	Analyse und Grobkonzeption der technischen Anforderungen • Analyse der Auswirkungen der fachlichen Anforderungen an die Technik (Hardware, Änderung der Architektur, Änderung Datenmodell, Schnittstellen-Änderungen etc.) • Grob-Beschreibung der erforderlichen technischen Änderungen
	Projektplanung Auf Basis der Ergebnisse der Analyse und Grobkonzeption der fachlichen und technischen Anforderungen kann die Projektplanung durchgeführt werden. Folgende Ergebnisse sollten am Ende der Projektplanung vorhanden sein: • Schätzung der Aufwände • Schätzung der entstehenden Kosten • Planung der benötigten Ressourcen • Grober Meilensteinplan (inkl. geschätzter Ende Termin)
	Kosten- / Nutzen-Analyse Wirtschaftlichkeitsbetrachtung

Phase	Inhalt
Realisierung	**Spezifikation der fachlichen Anforderungen** Detaillierte Beschreibung aller fachlichen Anforderungen. Dies kann in textueller Form erfolgen und/oder es werden hierfür spezielle Methoden/Techniken und dazu passende Werkzeuge eingesetzt. Folgende Ergebnisse können beispielsweise Bestandteil der fachlichen Anforderungen sein: • Geschäftsprozess-Modell, Anwendungsfall-Modell, User-Case Beschreibung • Datenmodell inkl. Attributbeschreibung • Beschreibung der Benutzeroberfläche • Beschreibung der Listen/Belege/Statistiken
	Spezifikation der technischen Anforderungen Auf Basis der Festlegungen in der technischen Grobkonzeptionsphase und der fachlichen Detailbeschreibungen müssen die DV-technischen Details beschrieben werden. Dies kann beispielsweise sein: • Detailliertes Architektur-Modell • Diagrammarten aus dem objektorientierten Design • Interaktionsdiagramme • Sequenzdiagramme • Zustandsdiagramme • Physikalisches Datenbankdesign • Schnittstellenbeschreibungen

Phase	Inhalt
Realisierung	**Programmierung** Es sollten unabhängig von der Programmiersprache und der spezifischen Vorgehensweisen immer folgende Aktivitäten durchgeführt werden: • Spezifikation der zu erstellenden Komponente; dies kann sein: – Textuelle Beschreibungen – Pseudocode – Struktur-Diagramme – Entscheidungstabellen • Codierung • Komponenten-Test (auch sehr oft Entwickler- oder Modultest genannt)
	Test Gemeint sind hier alle Teststufen, die nach dem Komponenten-Test durchgeführt werden (dazu mehr in dem nachfolgenden Kapitel)
Einführung	**Übernahme der Anwendung in die Produktion** Dies wird von Projekt zu Projekt unterschiedlich sein. Es kann ausreichend sein, dass Programme von einer Bibliothek in eine andere Bibliothek übernommen werden. In anderen Fällen muss ein Rollout mit umfangreichen Installations- und Schulungsmaßnahmen geplant werden. Nach der tatsächlichen Produktivsetzung folgt noch innerhalb des Projektes die Gewährleistungsphase (Stabilisierung und Optimierung der Anwendung).

1.4.3 Problematik aus Test-Sicht und Lösungsansätze

Problematik beim klassischen Phasenmodell:

- Verzögerungen im Laufe der Realisierung wirken sich direkt auf den Beginn der Testaktivitäten aus.
- Nachdem der Termin „Produktivsetzung“ in der Regel nicht verschoben werden kann, hat dies zur Folge, dass damit automatisch das so genannte „Testfenster“ kleiner wird.
- Die Aufwände und Termine, die im Rahmen der Projektplanung geschätzt wurden, können noch so gut sein – die Zeit steht nicht mehr zur Verfügung.

Ein weiteres Problem ist, dass die Schätzung bez. der Testaktivitäten (Aufwände, Kosten, Ressourcen und Termine) auch bereits in der Phase „Projektplanung“ durchgeführt werden muss. Dies bedeutet, dass in den fachlichen und technischen Grobkonzepten die Testbelange berücksichtigt sein müssen, so dass eine Schätzung möglich ist.

Mögliche Lösungsansätze

- Die Testbelange müssen in einem Projekt von Anfang an berücksichtigt werden
- Dies bedeutet, dass der für den Test Verantwortliche (Testmanager) bereits in den Phasen „Analyse und Grobkonzeption fachlicher und technischer Anforderungen“ mitarbeitet und darauf achtet, dass die Testbelange berücksichtigt werden.
- Nur dann ist es möglich, eine valide Schätzung für den Test in der Phase „Projektplanung“ abzugeben.
- Die Testaktivitäten sollten jeweils zum frühesten möglichen Zeitpunkt beginnen.
- So könnte beispielsweise die Erstellung der Testfälle für den fachlichen Test beginnen, sobald die detaillierte Beschreibung der fachlichen Anforderungen vorliegt. Also beispielsweise parallel zur DV-technischen Beschreibung und Programmierung.
- Die Tests sollten so vorbereitet sein, dass die Aktivitäten im Rahmen der Testdurchführung zeitlich möglichst schnell durchgeführt werden können.

Im Folgenden werden Vorgehensweisen, Methoden und Techniken vorgestellt, die darauf abzielen, dass die oben genannten Lösungsansätze in der Praxis umsetzbar sind.

1.5 Begriffsdefinitionen

1.5.1 Fehler und Mangel

Wann liegt ein nicht anforderungskonformes Verhalten des Systems vor? Eine Situation kann nur dann als fehlerhaft eingestuft werden, wenn vorab festgelegt wurde, wie die erwartete, korrekte, also nicht fehlerhafte Situation aussehen soll.

- Ein **Fehler** ist somit die Nichterfüllung einer festgelegten Anforderung, eine Abweichung zwischen dem Ist-Verhalten und dem Soll-Verhalten (in der Spezifikation oder den Anforderungen festgelegt).
- Ein **Mangel** liegt vor, wenn eine gestellte Anforderung oder eine berechtigte Erwartung nicht angemessen erfüllt wird.

Ein Mangel ist beispielsweise die Beeinträchtigung der Verwendbarkeit bei gleichzeitiger Erfüllung der Funktionalität oder die Nichterfüllung einer angemessenen Erwartung.

1.5.2 Fehlhandlung, Fehlerzustand, Fehlerwirkung

Im Gegensatz zu physikalischen Systemen entstehen Fehler in einem Softwaresystem nicht durch Alterung oder Verschleiß. Jeder Fehler oder Mangel ist seit dem Zeitpunkt der Entwicklung in der Software vorhanden. Er kommt jedoch erst bei der Ausführung der Software zum Tragen.

- Für die Beschreibung dieses Sachverhalts wird der Begriff **Fehlerwirkung** oder **äußerer Fehler** verwendet. Beim Test der Software oder auch erst bei deren Betrieb wird eine Fehlerwirkung für den Tester oder Anwender nach außen sichtbar.
- Zwischen dem Auftreten einer Fehlerwirkung und deren Ursache muss unterschieden werden. Eine Fehlerwirkung hat ihren Ursprung in einem **Fehlerzustand** oder auch **innerer Fehler** bezeichnet.

Ein Problem ist, dass ein Fehlerzustand nicht zu einer Fehlerwirkung führen muss. Eine Fehlerwirkung kann gar nicht, einmal oder immer und somit für alle Benutzer des Systems auftreten. Eine Fehlerwirkung kann weit entfernt vom Fehlerzustand zum Tragen kommen.

- Ursache für das Vorliegen eines Fehlerzustands oder Defekts ist die vorausgegangene **Fehlhandlung** einer Person, beispielsweise die fehlerhafte Programmierung durch den Entwickler.

1.6 Testbegriff

In der Literatur findet man verschiedenste Definitionen des Testbegriffs:

Definition 1:

„Testen ist das Ausführen eines Programms, mit der Absicht Fehler zu finden“.

[Myers, 1982]

Definition 2:

„Testen ist jede einzelne (i. a. stichprobenartige) Ausführung eines Prüfgegenstandes unter spezifizierten Bedingungen zum Zwecke des Überprüfens der (beobachteten) Ergebnisse des Prüfgegenstands im Hinblick auf gewisse gewünschte Eigenschaften.“

[GI-Fachgruppe 2.1.7 TAV (Test, Analyse und Verifikation von Software) der Gesellschaft für Informatik]

Definition 3:

„Testing: The process consisting of all life cycle activities, both static and dynamic, concerned with planning, preparation and evaluation of software products and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit to purpose and to detect defects.”

[STD-GLOSS – engl. Glossar]

Testen bedeutet nachzuweisen, dass die zu testende Anwendung fehlerfrei ist.

Diese Fehlerfreiheit wird man in der Regel nicht erreichen.

Testen bedeutet, Fehler zu finden.

– Hier spielen unter Anderem psychologische Aspekte eine wichtige Rolle und sollten nicht vernachlässigt werden.
– Die Entwicklung von Software wird oft als konstruktive Tätigkeit angesehen.
– Das Testen von Software wird oft als destruktive Tätigkeit angesehen.
  Nicht selten kommt es deshalb in Projekten zu Schwierigkeiten in der Kommunikation zwischen Entwicklung und Test.

Weitere Probleme aus der Definition „Testen“ können sich bei folgenden Fragestellungen ergeben:

- Wann hört man auf zu testen?

  ⇒ wenn man keine Fehler mehr findet?

- Wann war der Test erfolgreich?

  ⇒ wenn man viele Fehler gefunden hat?

  ⇒ Damit war der Test nicht erfolgreich, wenn man beispielsweise nur wenige oder keine Fehler gefunden hat?

Folgende Grundsätze lassen sich für das methodische Testen ableiten:

⇒ für jeden Test werden Testziele benötigt,

⇒ diese Testziele müssen akzeptabel und erreichbar sein,

⇒ für das Testen benötigt man möglichst genaue Vorgaben,

⇒ Tests müssen reproduzierbar sein.

Das Ziel des strukturierten Testens ist keineswegs die Bescheinigung von Fehlerfreiheit. Selbst wenn alle ausgeführten Testfälle keinen Fehler mehr aufweisen, kann nie ausgeschlossen werden, dass nicht zusätzliche weitere Fehlerwirkungen auftreten könnten.

Mit dem strukturierten Vorgehen beim Test soll Folgendes erreicht werden:

- eine möglichst hohe Qualität des zu erstellenden Produkts. Das ist nur in Zusammenhang mit der Fehlerbehebung durch die Entwickler möglich. Testen allein kann das nicht leisten.
- finden von Abweichungen und Fehlern zum frühestmöglichen Zeitpunkt
- Reduzierung des Testaufwands durch strukturierte Vorgehensweise
- Nachvollziehbarkeit, Wiederholbarkeit und ggf. Automatisierbarkeit der Tests
- Erreichung dieser Qualitätsstufe in einem angemessenen Verhältnis zwischen Zeit, Kosten und Ressourcen.

Übersicht der grundsätzlichen Testaktivitäten

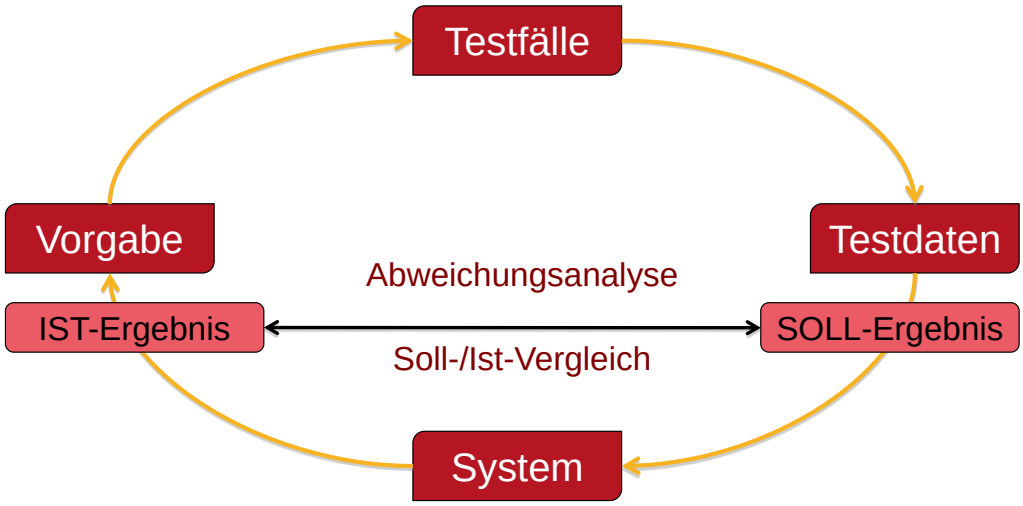


Abbildung 1-4: Übersicht Testaktivitäten

Prozessschritte:

- Aus einer Vorgabe werden Testfälle ermittelt.
- Für die Testfälle werden die hierfür erforderlichen Testdaten definiert.
- Damit kann für jeden Testfall das erwartete Ergebnis (= Soll-Ergebnis) definiert werden.
- Sobald das zu testende System zur Verfügung steht, werden die Testfälle mit den entsprechenden Testdaten durchgeführt. Da die Testfälle auf Basis der Vorgabe erstellt wurden, wird getestet, ob das System den Vorgaben entspricht.
- Bei der Testdurchführung wird ein Ist-Ergebnis erzeugt.
- Es wird ein Soll-/Ist-Vergleich durchgeführt. Entspricht das Ist-Ergebnis dem erwarteten Soll-Ergebnis? Wenn ja, wurde die Vorgabe richtig umgesetzt wurde.
- Wenn bei dem Soll-/Ist-Vergleich eine Abweichung festgestellt wird, muss zunächst analysiert werden, wodurch die Abweichung zu Stande gekommen ist. Hierfür kann es mehrere Gründe geben:
  - Der Testfall wurde falsch aufgebaut.
  - Die Testdaten wurden falsch definiert oder entsprachen zum Zeitpunkt der Testdurchführung nicht mehr dem Stand, der für den Testfall erforderlich gewesen wäre.

– Die Vorgabe war nicht eindeutig. Der Entwickler hat die Vorgabe im Programm anders umgesetzt als der Testfallersteller seinen Testfall aufgebaut hat.
– Oder es wurde tatsächlich eine Fehlerwirkung festgestellt. Der Fehler muss korrigiert und nachgetestet werden.

1.7 Objektorientierte SW-Entwicklung: Testfälle aus Use Cases

In der modernen SW-Entwicklung wird immer häufiger mit objektorientierten Methoden gearbeitet. In objektorientierter Analyse und Design hat sich die Unified Modelling Language (UML) inzwischen als Standard etabliert. Wichtigste Rolle in der ersten Phase der SW-Entwicklung mittels UML spielen die so genannten Use Cases (auch "Geschäftsvorfälle" oder "Anwendungsfälle").

Anwendungsfälle (Use Cases) sind auch Bestandteile Geschäftsprozessmodellierung, jedoch noch näher und konkreter auf die Fragestellung fokussiert: "Was macht der User mit dem System?"

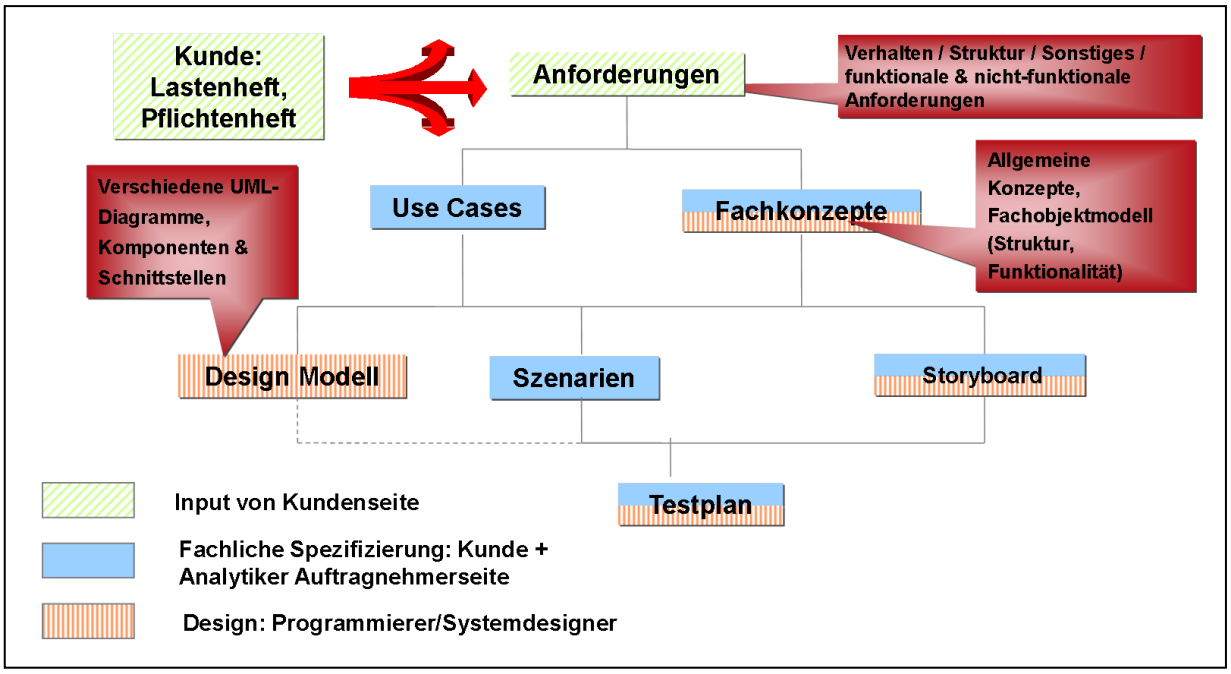


Abbildung 1-5: Testfälle aus Use Cases (1)

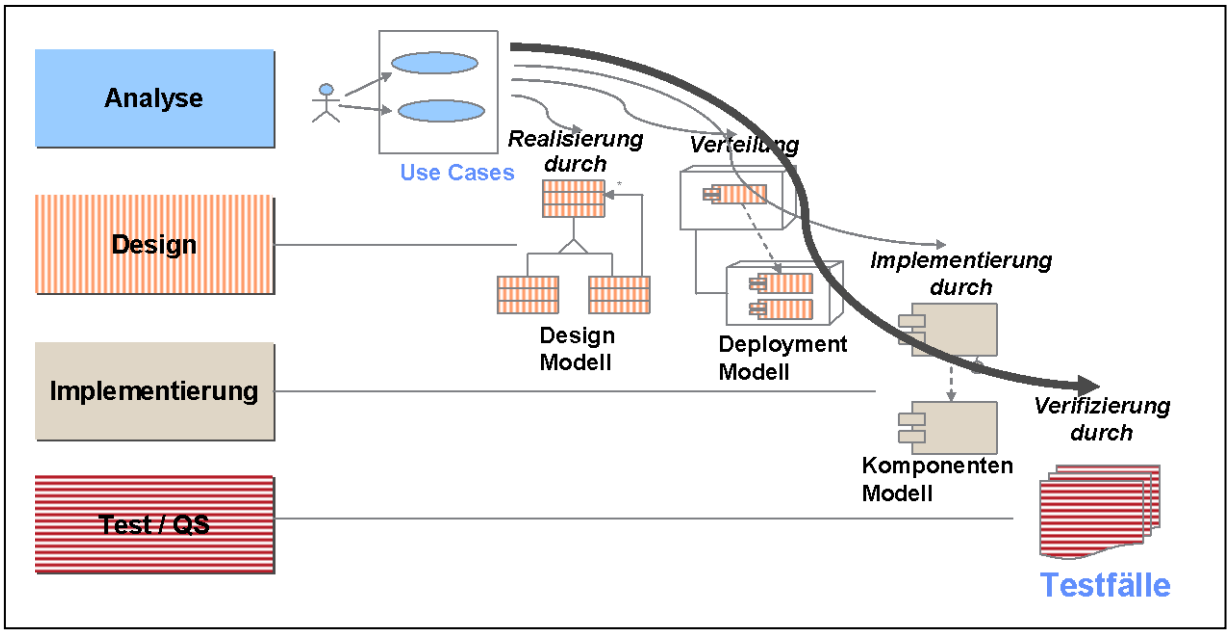


Abbildung 1-6: Testfälle aus Use Cases (2)

Bei der Beschreibung von Use Cases werden fachliche Vorgaben in einer leichten Formalisierung umgesetzt. Ein Use Case kann enthalten:

- Eine verbale Beschreibung, was der User durch diesen Use Case macht (Zusammenfassung, ein oder zwei Sätze).
- Die Voraussetzungen, die am Beginn des Use Cases erfüllt sein müssen (Vorbedingungen bzgl. Daten, Funktionen etc.).
- Der Ablauf des Use Cases (Beschreibung der einzelnen Schritte).
- Das Ergebnis des Use Cases (was ist am Ende im System verändert?).
- Zusätzlich zum normalen Ablauf werden Alternativen (wenn vorhanden) beschrieben:
  - Alternative Behandlungen bei Abweichung vom "Normalfall"
  - Fehler- und Ausnahmesituationen
- Gegebenenfalls werden zusätzliche fachliche Abhängigkeiten beschrieben.
- Schließlich können in Szenarien unterschiedliche Use Cases in konkreten Beispielen kombiniert werden.

Ein großer Vorteil der Use Cases (UC) ist die graphische und übersichtliche Darstellung mit präziser Beschreibung in normierten Formularen (in normaler Umgangssprache). So kann der geforderte Ablauf leicht von den Mitarbeitern der Fachabteilung, aber auch fachfremden Personen verstanden und überprüft werden. Diese UC bilden im Weiteren eine natürliche Grundlage für die Ableitung von Testfällen.

1.8 Qualitätsmerkmale

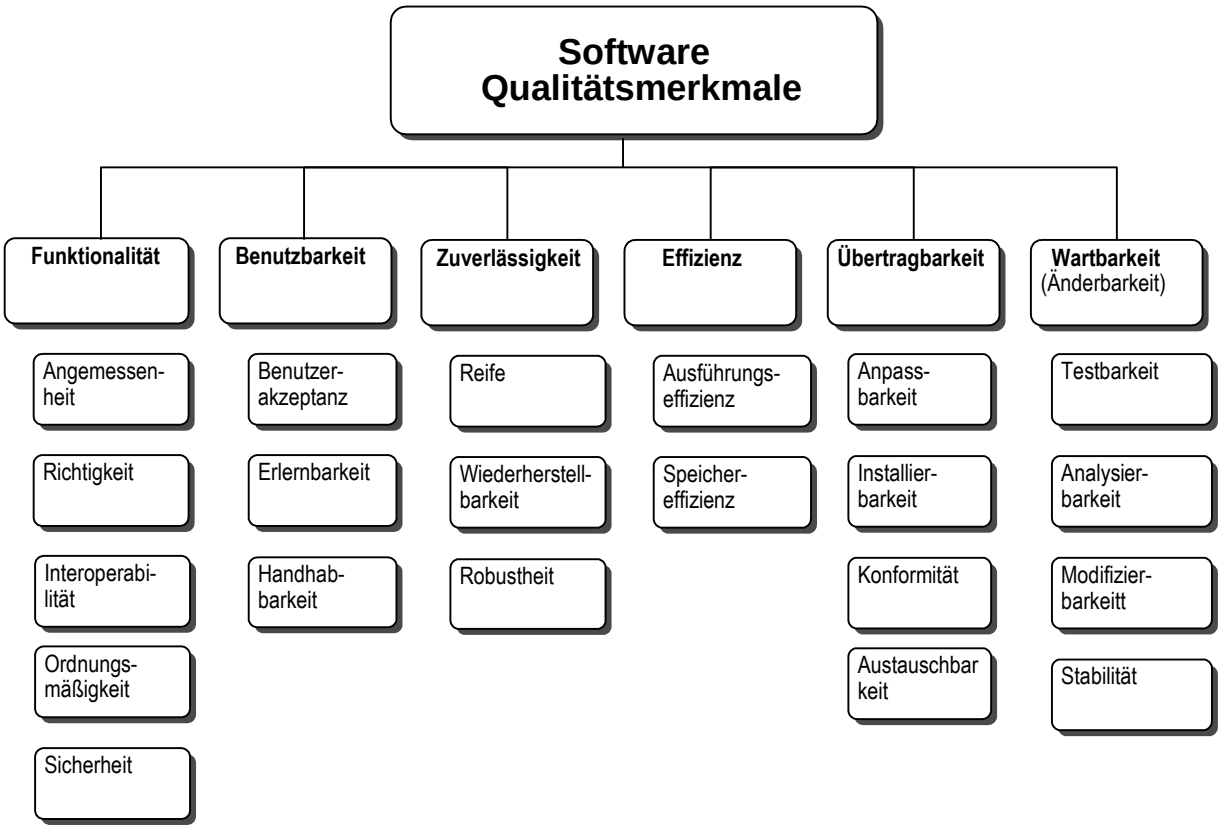


Abbildung 1-7: Software Qualitätsmerkmale

Die Qualitätsmerkmale der Software müssen in der Planungsphase definiert und operationalisiert werden (d. h. Festlegungen anhand von Beispielen treffen, wie die Umsetzung zu erfolgen hat).

Ihre Ausprägung (Merkmale) sind meist vom Kontext des zu erstellenden Systems abhängig, häufig z. B. auch von der Unternehmenskultur.

Allgemeine SW-Qualitätsmerkmale nach DIN 66272 bzw. ISO/IEC 9126 sind:

- Funktionalität

  Vollständigkeit der Funktionen der Anwendung in Bezug auf die funktionalen Anforderungen und zwar mit:

  – Angemessenheit der Realisierung
  – Richtigkeit
  – Interoperabilität (Verträglichkeit der Funktionen)
  – Ordnungsmäßigkeit (z. B. nach HGB für Buchhaltungen)
  – Sicherheit (Zugriffsschutz, Benutzerprofile)

Wichtig für den Betrieb und Unterhalt einer Software sind auch die so genannten „Nicht-Funktionalen Anforderungen“ (mit den im Schaubild jeweils genannten Unterpunkten):

- Benutzbarkeit

  Die Benutzerfreundlichkeit ist die Eignung eines Produktes, den zu seiner Nutzung benötigten Aufwand für die vorgesehenen Benutzer gering zu halten und das Beurteilen der Handhabung durch die Benutzer positiv zu beeinflussen.

- Zuverlässigkeit

  Die Zuverlässigkeit beschreibt die Fähigkeit einer Software, ein bestimmtes Leistungsniveau (ununterbrochener Betrieb) aufrechtzuerhalten.

- Effizienz

  Die Effizienz beschreibt das Zeit- und Ressourcenverhalten einer Software.

- Übertragbarkeit

  Ist die Eignung eine Software in andere Umgebungen (Hardware, Organisation, Sprachen) übertragbar zu sein.

- Wartbarkeit (Änderbarkeit)

  Beschreibt den erforderlichen Aufwand um eine Software zu ändern. Dazu gehört die Analysierbarkeit, Stabilität und Testbarkeit.

Zusammenfassung:

Es gibt sechs definierte QS-Merkmale für SW. Es ist wichtig, bereits bei der Beauftragung der Systemerstellung die wichtigsten gewünschten QS-Merkmale zu definieren und anhand von Beispielen zu verdeutlichen.

1.9 Testen und Qualitätssicherung

Unter Software-Qualitätssicherung (SQS) verstehen wir eine Menge von Regeln, Methoden, Maßnahmen und Werkzeugen, die sicherstellen sollen, dass ein Software-Produkt und der dahin führende Entwicklungsprozess vorgegebene Anforderungen, Normen und Standards erfüllen.

Nach diesem Verständnis besteht das vorrangige Ziel der SQS vor allem darin, Fehler bei der Definition und Konstruktion von Software zu vermeiden. Die SQS stellt dazu eine Reihe von sog. **konstruktiven QS-Maßnahmen** zur Verfügung.

Konstruktive QS-Maßnahmen sind z. B.

- Anwendung der Strukturierten Programmierung,
- Einsatz der Strukturierten Analyse,
- Top-Down-Entwurf und Modularisierung,
- Verwendung höherer Programmiersprachen statt ASSEMBLER,
- Einsatz von Werkzeugen (Entwicklung, Dokumentation, Test etc.),
- Standardisierung durch wiederverwendbare Software-Bausteine,
- Installation eines Konfigurationsmanagements (Änderung-, Versions- und Archivierungsverfahren).

Durch konstruktive QS-Maßnahmen soll Qualität von Anfang an konstruiert und nicht erst später in das Software-Produkt 'hineingetestet' werden.

'Testen' deckt praktisch den analytischen Teil der SQS ab. Das heißt, beim Testen werden Software-Produkte (Entwicklungs- und Zwischenprodukte, Endprodukte, Dokumente etc.) analysiert, um Fehler aufzudecken. Wir setzen deshalb die sog. analytischen QS-Maßnahmen mit Testen gleich.

Ähnlich wie bei der Definition des Begriffs 'Fehler' spielen auch bei der Definition von 'Qualität' die Anforderungen und Vorgaben eine entscheidende Rolle.

Unter Qualität verstehen wir die Beschaffenheit eines Produkts bzgl. seiner Eignung festgelegte und vorausgesetzte Anforderungen zu erfüllen.

Genauso wie Testen den gesamten Software-Entwicklungsprozess betrifft, muss auch die SQS durchgängig vom realen Problem bis zur Problemlösung durch ein Software-Produkt wirken.

Frühzeitig im Software-Entwicklungsprozess entdeckte Fehler sind nicht nur wirtschaftlicher zu beheben, als später entdeckte Fehler, sondern

frühzeitige Fehlerentdeckung vermindert generell die Anzahl der im Endprodukt verbleibenden Fehler, trägt also entscheidend zur Qualitätsverbesserung bei.

Die wichtigste und wirkungsvollste Methode zur frühzeitigen Fehlererkennung bilden die sog. REVIEW-Verfahren. Andere Begriffe hierfür sind QS-Sitzungen oder auch Walkthroughs.

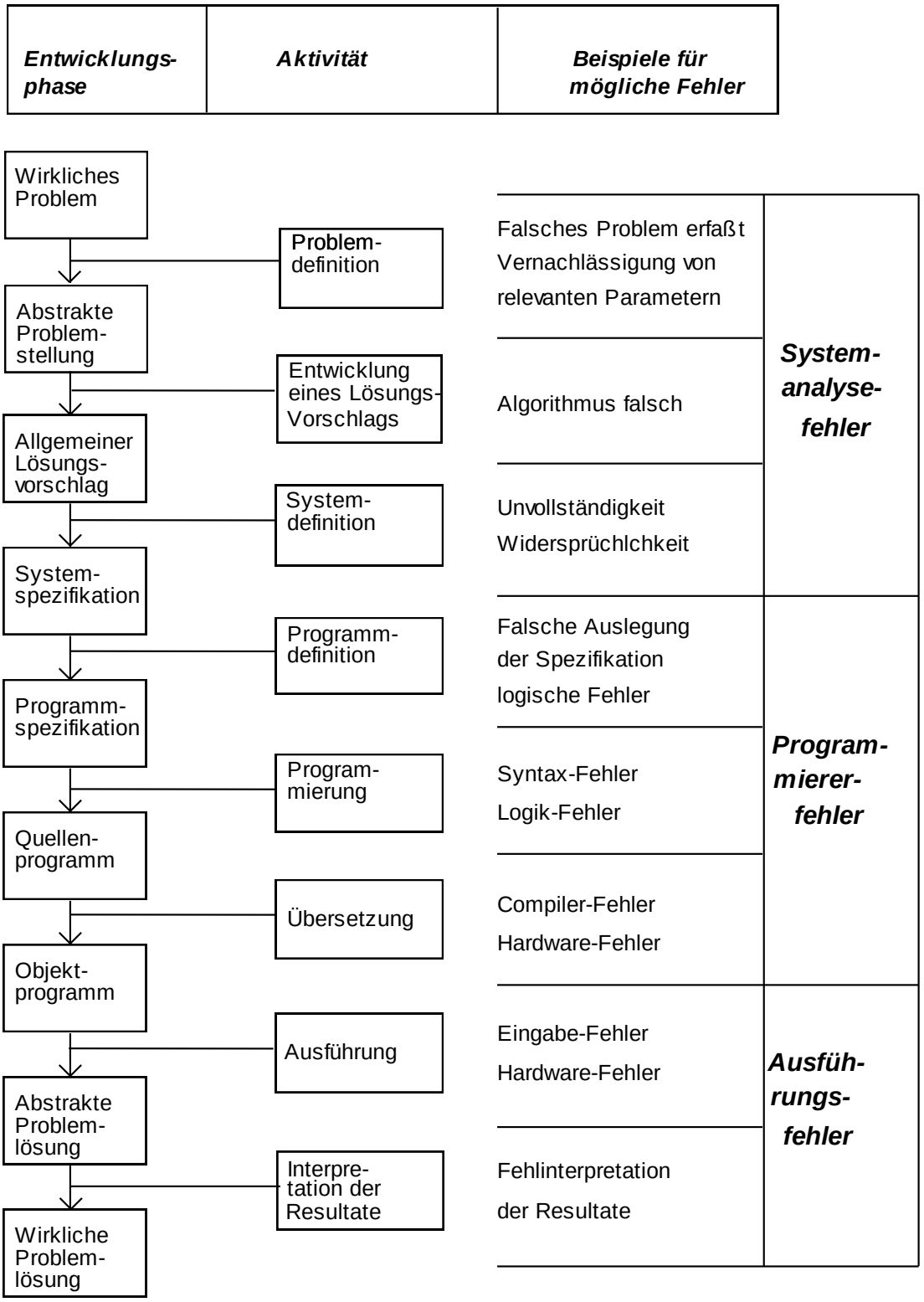


Abbildung 1-8: Stufen in der Software-Entwicklungskette und mögliche Fehler

1.9.1 REVIEW-Verfahren

Was ein REVIEW nicht ist...

- eine der üblichen Projektsitzungen
- die Garantie, dass das Produkt zu 100 % frei von Fehlern ist
- die Kontrolle von Projektmitarbeitern durch Vorgesetzte
- eine Diskussion von Lösungen zu einem Problem
- die dynamische Analyse von Programmen, um z. B. Laufzeitfehler zu entdecken
- eine unendliche Geschichte

Was ein REVIEW sein sollte...

ein strukturiertes Gruppengespräch zum formalen und inhaltlichen Test eines Projekt(zwischen)ergebnisses (Dokument, Programm) durch mehrere sachkundige Personen in verschiedenen Rollen: Moderator, Autor, Inspektor, Auftraggeber, (Entscheider).

Es sollte in jeder Organisation individuell festgelegt werden, ob Entscheider und Vorgesetzte am REVIEW teilnehmen dürfen. Wenn es ausschließlich um eine objektive, gemeinsame Überprüfung von Arbeitsergebnissen geht, ist dies nicht sinnvoll. Wenn im REVIEW jedoch auch Entscheidungen über den weiteren Projektverlauf (Abbruch, Änderung der Aufgabenstellung etc.) getroffen werden, ist die Teilnahme von Entscheidern und damit Vorgesetzten aber unerlässlich.

Merkmale des REVIEW-Verfahrens

Ziel Überprüfung der Produktqualität bzgl. expliziter Anforderungen und impliziter Standards/Normen in formaler und inhaltlicher Hinsicht

Zeitpunkt Empfehlenswert nach Fertigstellung wichtiger Zwischenergebnisse oder nach Erreichen von Projektmeilensteinen

Beteiligte Projektleiter, Autor, Moderator, DV-Koordinator, Fachleute des Auftraggebers (Fachabteilung) als Inspektoren, evtl. Vertreter des Lenkungsausschusses; ineffektiv, bei mehr als 6 bis 8 Personen

Inhalt Prüfung und Aufdeckung von Fehlern, Mängeln oder Schwachstellen; keine Diskussion von Lösungen, da dies Aufgabe des Projekts ist

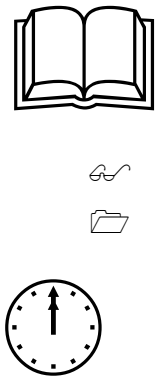
Ablauf

1. Eröffnung durch den Moderator
2. Präsentation des Prüfprodukts durch Projektleiter und Autor
3. Diskussion der Mängel
4. Protokollierung der Mängel
5. Beschluss über weitere Vorgehensweise
6. Moderator ist für Einhaltung der Spielregeln verantwortlich

Vorteile

1. **Frühzeitige** Überprüfung von Zwischenprodukten
2. **Gemeinsame** Überprüfung durch die Beteiligten
3. Referenteneffekt beim Projektleiter/Autor, d. h. durch das Präsentieren der eigenen Arbeitsergebnisse werden einige Fehler erst bewusst
4. Ermittlung von Kennzahlen für künftige Projekte

Wichtig Der Erfolg eines REVIEWS hängt entscheidend davon ab, ob die Beteiligten sich die Zeit genommen haben, das zu überprüfende Produkt vor der Sitzung intensiv zu studieren. Das Prüfobjekt ist deshalb den Beteiligten mindestens zwei Wochen vorher zur Verfügung zu stellen.



- Formelles oder informelles Gruppengespräch, zum Nachweis der Erfüllung der Produktqualität
- Zeitliche Dauer: 2 bis 3 Stunden je Sitzung
- Keine Diskussion von Lösungen
- Festhalten der Review-Ergebnisse in einem Review-Protokoll
- Fortsetzung und / oder Wiederholung ist möglich
- Ermittlung von Kennzahlen

Abbildung 1-9: REVIEW-Verfahren im Überblick

1.9.2 Qualitätseigenschaften von Software

Gegenstand von REVIEW-Verfahren und des Testens (Testziele) sind auch sog. Qualitätsmerkmale (Q-Merkmale) von Software. Im Allgemeinen erwarten wir als Anwender eine **benutzerfreundliche** Anwendung, was immer dies auch bedeuten mag. Als Entwickler ist uns die Benutzerfreundlichkeit relativ egal, dafür erwarten wir ein **wartungsfreundliches** Anwendungssystem. Wichtige Qualitätsmerkmale wie z. B.

- Portabilität
- Wartbarkeit
- Wiederverwendbarkeit
- Benutzerfreundlichkeit
- Robustheit
- Sicherheit

müssen im Zuge der Projektdefinition explizit definiert und operationalisiert werden.

Nach dem Motto

'Alles, was man quantifizieren muss, kann auf eine Art gemessen werden, die auf jeden Fall besser ist, als gar nichts zu messen.'

können für solche Q-Merkmale messbare Attribute aufgestellt werden, die konkret überprüfbar sind. Andernfalls werden sich später Auftraggeber und Entwickler vortrefflich über Vorhanden- oder Nichtvorhandensein von Q-Merkmalen streiten.

Q-Merkmale lassen sich nach verschiedenen Kriterien strukturieren. Beispiele hierfür finden sich auf den folgenden Abbildungen.

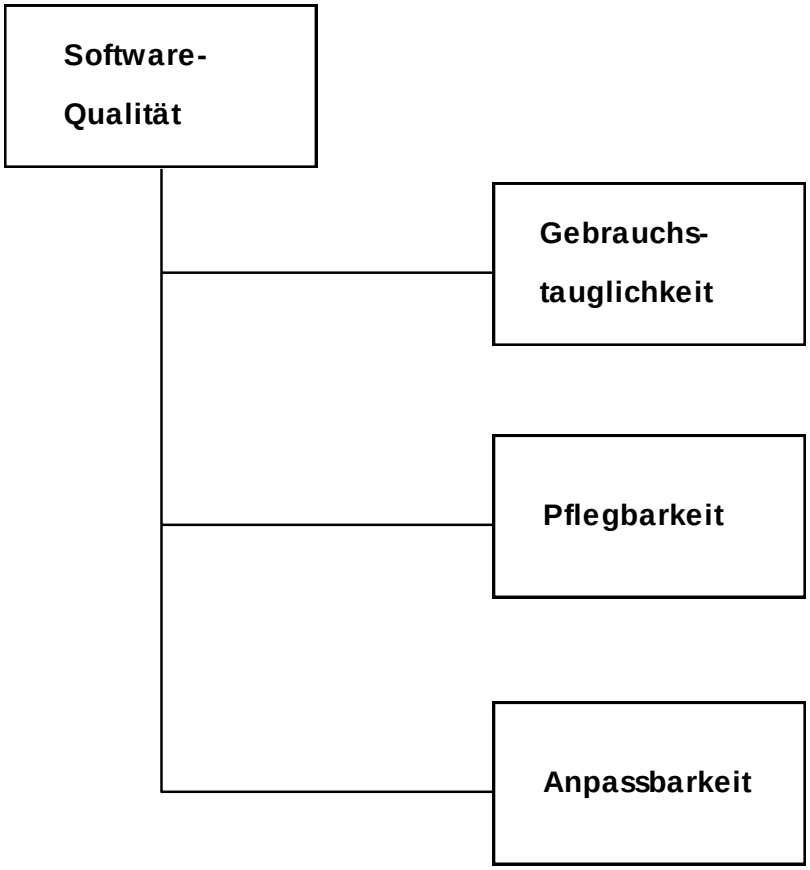


Abbildung 1-10: Grobstrukturierung von Q-Merkmalen

Zum Schluss dieses Abschnittes sehen Sie einige Beispiele dafür, wie sich Q-Merkmale operationalisieren lassen.

Benutzerfreundlichkeit

... Eignung eines Produktes, den zu seiner Nutzung benötigten Aufwand für die **vorgesehenen** Benutzer gering zu halten und das Beurteilen der Handhabung durch die Benutzer positiv zu beeinflussen.

Der **Aufwand** für den Benutzer ist z. B.

- Zeit zum Erlernen der Benutzung des Produkts,
- Zeit und Anzahl der Schritte, um Fehlersituationen zu erkennen und zu meistern,
- Zeit der Vorbereitung bis zur Nutzung (Rüstzeit),
- Antwortzeit des Systems nach Benutzereingaben.

Der Grad der **Handhabbarkeit** wird u. a. bestimmt durch

- Anzahl, Komplexität und Zuverlässigkeit der Bedienungsschritte relativ zur auszuführenden Funktion,
- Bedienerführung (Menüs, Masken, Dialog),
- Übersichtlichkeit der Informationsdarstellung (Kopf-/Fußzeilen, Fenster, Icons, Bildschirm-Layout),
- Unterstützung in Fehlersituationen.

Aufgrund dieses Beispiels könnten Sie in Ihren Projekten jeweils eine Checkliste erstellen, um das Merkmal Benutzerfreundlichkeit bewerten zu können, falls es gefordert ist.

1.9.3 Wartungsfreundlichkeit

... Eignung eines Produktes, den Aufwand für das Erkennen von Fehlerursachen und ihre Korrektur, sowie für die Durchführung von Änderungen mit dem **vorgesehenen Personal** innerhalb eines **gegebenen Zeitraums** bei **Verfügbarkeit entsprechender Werkzeuge** und Ersatzteile gering zu halten.

Beachten Sie die getroffenen Voraussetzungen!

Kennzeichnend für die Wartungsfreundlichkeit sind also der Zeitbedarf und der Aufwand pro Wartungsvorgang. Dieser erstreckt sich vom Zeitpunkt des Eintritts eines Fehlers bis zum Zeitpunkt, wo das System wieder funktioniert. Insofern setzt sich die Wartungsfreundlichkeit aus der Verfügbarkeit des Systems, der Qualifikation des Wartungspersonals und dem Vorhandensein von Werkzeugen und Ersatzteilen zusammen.

Hierbei kann die Verfügbarkeit formelmäßig erfasst werden:

$$V = \frac{MTTF}{MTTF + MTTR}$$

mit MTTF (Mean Time To Fail), also mittlere Zeit bis zu einem Fehler,

MTTR (Mean Time To Repair), also mittlere Zeit für die Fehlerbehebung

Zusammenfassung:

Testen kann als analytischer Teil der Software-Qualitätssicherung gesehen werden. Dabei stellt die SQS Methoden zur Verfügung (REVIEW-Verfahren, Q-Merkmale, Qualitätsmaße,...), die das Testen von Software-Produkten insbesondere in frühen Entwicklungsphasen erlauben.

1.10 Kriterien für das Beenden von Tests

Wenn Sie in EDV-Abteilungen fragen, wann Tests oder Testabschnitte beendet werden, werden Sie meistens die Antwort erhalten, "wenn die dafür vorgesehene Zeit abgelaufen ist."

Die Zeit muss aus praktischen Erwägungen natürlich ein Testende-Kriterium sein. Sie darf aber auf keinen Fall das einzige oder das wichtigste Kriterium sein. Wichtigere Testende-Kriterien als die Zeit sind beispielsweise

- Qualität der Testfälle,
- Systematik der Vorgehensweise beim Testen,
- eingesetzte Testmethoden,
- vorgegebene Testziele und vor allem die
- Anzahl der entdeckten Fehler insgesamt und pro Zeiteinheit.

Idealerweise kann sich der Projektleiter oder Testmanager aufgrund zahlreicher Faktoren schon vor Beginn des Testprozesses Gedanken über die zu erwartenden Fehler machen und versuchen diese abzuschätzen. Dann hätten die am Test Beteiligten ganz konkrete Vorgaben bzgl. der Anzahl der zu entdeckenden Fehler. Dies wäre ganz im Sinne unserer Definition des Begriffs 'Testen'.

Notwendig für solch eine Fehlerabschätzung sind über viele Jahre aus zahlreichen Projekten zusammengetragene Erfahrungswerte und Kennzahlen. Diese sind aber leider meistens in der Praxis nicht verfügbar, da die Testdokumentation häufig zu kurz kommt.

Nehmen wir einmal an, wir verfügen über ausreichende Kennzahlen und können vor dem Testen eine Fehlerabschätzung vornehmen.

Beispiel (Testendekriterien)

Es ist ein Programm mit 10.000 Befehlen zu testen.

Aufgrund der Erfahrungen aus früheren Projekten nehmen wir an, dass nach allen Reviews noch 5 Fehler pro 100 Anweisungen übrigbleiben. Das heißt, wir erwarten eine Gesamtzahl von etwa 500 Fehlern in diesem Programm.

Unser Test Ziel: Entdeckung von 98 % der Codier- und Logikfehler

Entdeckung von 95 % der Entwurfsfehler

Aufgrund unserer Erfahrungen aus früheren Projekten nehmen wir weiter an, dass 40 % der Gesamtfehler Codier- und Logikfehler (d.h. 200) und 60 % Entwurfsfehler (d.h. 300) sind.

Testaufgabe: Finde mindestens 196 Codier- und Logikfehler und 285 Entwurfsfehler bis der Testabbruch erfolgen kann.

Im nächsten Schritt haben wir abzuschätzen, zu welchem Zeitpunkt im Testprozess wir wieviel Prozent der Fehler entdecken können. Zum Beispiel:

	Codier- und Logikfehler	Entwurfsfehler
Modultest	65 %	0 %
Funktionstest	30 %	60 %
Systemtest	3 %	35 %
Gesamt	98 %	95 %

1.10.1 Testendekriterien

Wir geben also folgende Kriterien für das Beenden des Tests vor:

1. Der Modultest wird beendet, wenn 130 Fehler entdeckt und korrigiert wurden (65 % der 200 Codier- und Logikfehler).
2. Der Funktionstest wird beendet, wenn 240 Fehler (30 % von 200 plus 60 % von 300) entdeckt und korrigiert wurden oder wenn die vorgesehene Zeit (z. B. 4 Monate) abgelaufen ist, je nachdem was **später** eintritt (u. U. gibt es ja noch mehr Fehler als wir geschätzt haben).
3. Der Systemtest wird beendet, wenn 111 Fehler (3 % von 200 plus 35 % von 300) entdeckt und korrigiert wurden oder wenn die vorgesehene Zeit (z.B. 3 Monate) abgelaufen ist.

Nun kann das 'Problem' auftreten, dass wir nicht genügend Fehler finden. Dies wird in der Praxis kaum vorkommen. Wenn wir dennoch einmal den Eindruck haben, es sind nicht so viele Fehler vorhanden, wie wir geschätzt haben, können wir beispielsweise den Testprozess (Methoden, Testfälle etc.) von einem Außenstehenden objektiv überprüfen lassen. Ist an unserer Vorgehensweise und den eingesetzten Verfahren nichts zu bemängeln, können wir dann das Testen mit gutem Gewissen abbrechen.

Weiter ist die statistische Verfolgung und Auswertung des Testens unbedingt erforderlich. Wenn pro Zeiteinheit permanent noch Fehler gefunden werden, sollte auf keinen Fall das Testen beendet werden. Wenn jedoch pro Zeiteinheit zusehends weniger Fehler entdeckt werden, kann das Testen je nach Kritikalität der Anwendung beendet werden.

1.11 Zusammenfassung:

Das Testen von Software ist ein wichtiger Bestandteil der Entwicklung und Wartung.

Das Testen ist nicht mehr nur eine Phase, die nach der Implementierung kommt, sondern ist ebenso ernst zu nehmen wie die anderen Aktivitäten der Entwicklung von Systemen.

Testen ist zu einer eigenständigen Aufgabe und Tätigkeit geworden. Es ist ein Prozess, bestehend aus Planung und Vorbereitung einerseits und Messen und Prüfen andererseits.

Testen dient dazu, die Charakteristika eines Systems festzustellen und die Unterschiede zwischen dem Ist- und dem Sollverhalten aufzuzeigen. Gute Qualität kann als Erfüllung der Anforderungen gesehen werden. Somit ist das Ergebnis des Testens, die vorhandene Qualität aufzuzeigen und Verbesserungen zu ermöglichen.

2

Teststufen im Softwarelebenszyklus

2 Teststufen im Softwarelebenszyklus

2.1 Entwicklungs- und Teststufen

Die Komplexität des Gesamttestumfangs erfordert die methodische Aufteilung des Testens in mehrere Teststufen. Große Systeme bestehen aus Subsystemen, diese wiederum aus Moduln. Sie sind häufig in ein komplexes, vernetztes Umfeld integriert.

Im Laufe des Entwicklungsprozesses eines Systems werden deren einzelne Komponenten immer weiter zusammengebaut, das System als Ganzes damit immer weiter integriert. Entsprechend richtet sich der Fokus des Testens in der frühen Phase auf die einzelnen Funktionalitäten und Module, im weiteren Verlauf verschiebt sich der Fokus auf die Schnittstellen und das Zusammenwirken der Subsysteme.

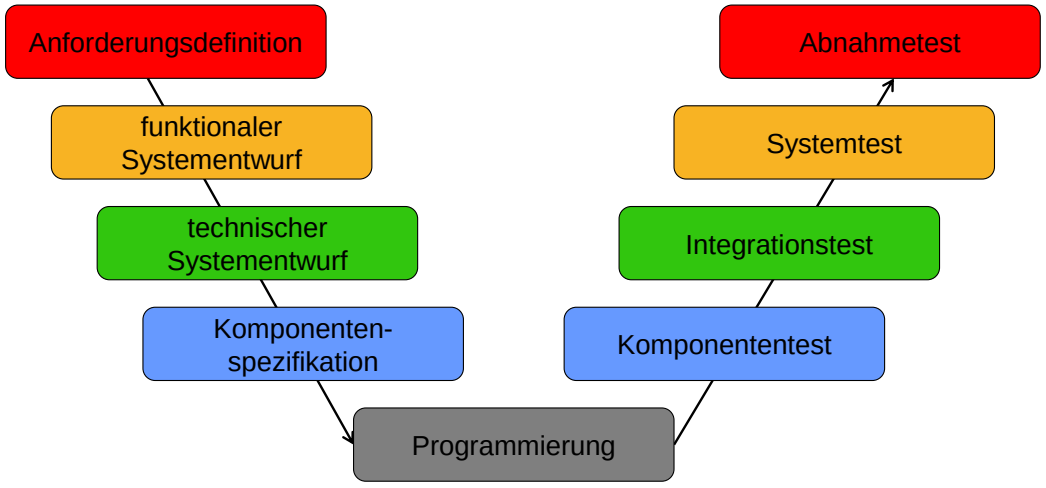


Abbildung 2-1: Gegenüberstellung der Entwicklungs- und Teststufen

Grundsätze, die sich daraus ableiten:

- Jede Entwicklungsphase hat eine ihr entsprechende Teststufe
- Die Testfälle für die entsprechende Teststufe werden aus den jeweiligen Ergebnissen der zugehörigen Entwicklungsphase abgeleitet
- Damit wird im Test immer „gegen“ die entsprechende Vorgabe getestet.
- Bei Fehlen der Vorgabe stellt sich die Frage, woraus dann die Testfälle abgeleitet werden. Beispielsweise: gegen was testet der Entwickler, wenn die Komponentenspezifikation fehlt?

2.2 Erläuterung der Teststufen

2.2.1 Teststufe: Entwicklertest

2.2.1.1 Schreibtischtest

Diesen Test sollte jeder Autor machen, der ein Objekt (Dokument, Pseudocode, Entscheidungstabelle, Modul etc.) erstellt hat. Er erfordert im Vergleich zu allen anderen Tests den geringsten Aufwand. Die gröbsten Fehler können so sehr schnell entdeckt werden.

Selbstverständlich ist ein Schreibtischtest nur bei sehr kleinen, überschaubaren Testobjekten möglich.

2.2.1.2 Komponententest

In der ersten Teststufe werden die in der unmittelbar vorangegangenen Programmierphase erstellten Software-Bausteine erstmalig einem systematischen Test unterzogen.

Abhängig davon, welche Programmiersprache die Entwickler einsetzen, werden diese kleinsten Softwareeinheiten unterschiedlich bezeichnet:

- Modul
- Unit
- Klasse
- Methode.

Von der verwendeten Programmiersprache abstrahiert, wird allgemein von Komponente gesprochen.

Im Komponententest wird jeweils ein einzelner Softwarebaustein geprüft und zwar isoliert von anderen Softwarebausteinen des Systems. Damit werden komponentenexterne Einflüsse ausgeschlossen.

Es kann im Komponententest grundsätzlich unterschieden werden zwischen

- **statischen Prüfungen**

und

- **dynamischen Tests.**

Ziele der **statischen Prüfungen** sind:

- Kompilierbarkeit
- Änderbarkeit
- Überprüfbarkeit
- Wartbarkeit
- Metriken
- Programmierrichtlinien.

Ziele der **dynamischen Tests** sind:

- Funktionalität
- Lauffähigkeit
- Umsetzung der Anforderungen,
- Schnittstellenkonformität,
- Laufzeit / Performance,
- Speicherverwaltung
- Robustheit
- Korrektheit der Algorithmen.

Grundsätzlich sollte der Komponententest zunächst als so genannter **White-Box-Test** durchgeführt werden.

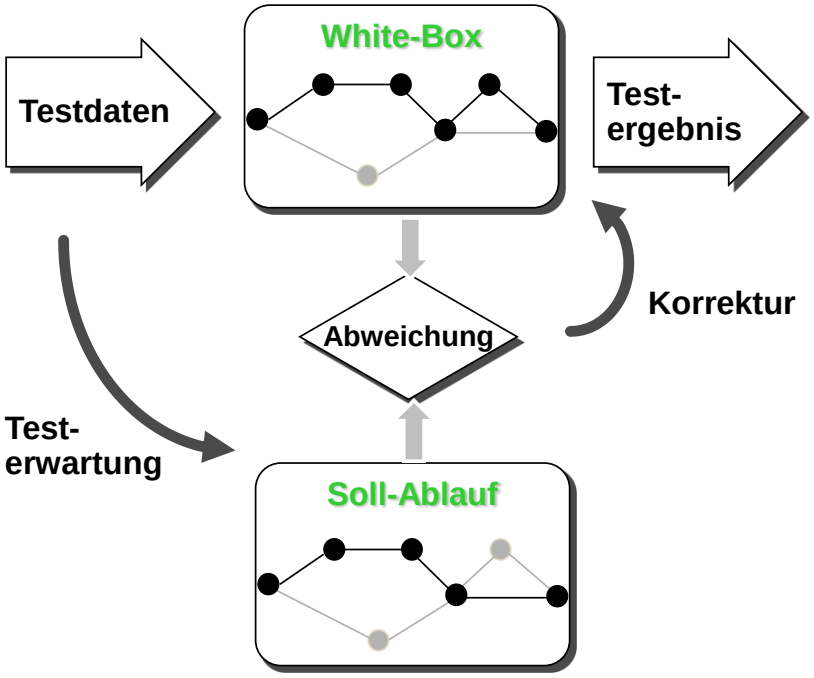


Abbildung 2-2: White-Box-Test

Beim White-Box-Test wird der Grad der Testabdeckung unter Kenntnis der internen Struktur des Testobjekts betrachtet. Es wird der Anteil des zu testenden Source-Codes gemessen, der durch die Testfälle ausgeführt wird.

White-Box-Verfahren benutzen zum Herleiten von Testfällen die Programmstruktur. Mit dieser Strategie wird **nicht** geprüft, ob das Testobjekt überhaupt das vorgegebene Problem löst, also die gestellten fachlichen Anforderungen erfüllt. Dies hat zur Folge, dass der Entwickler im Komponententest auch nach dem Black-Box-Verfahren testen sollte → siehe Systemtest.

White-Box-Verfahren werden auch als strukturelle Testverfahren bezeichnet, da sie die Struktur des Testobjekts berücksichtigen.

2.2.1.3 Teststufe: Modultest

Module bezeichnen die kleinsten Programmkomponenten wie

- Funktionen
- Unterprogramme
- Prozeduren
- Makros.

Module sind im Umfang und in der Zahl ihrer Schnittstellen zur Umgebung gut überschaubar. Daher kann der Test detaillierter erfolgen und es sind Fehler in einem Modul leichter zu entdecken und zu beheben.

Im Modultest werden die Operationen und deren Zusammenwirken isoliert von der Systemumgebung getestet. Je nachdem, welche anderen Module bereits erstellt sind, muss evtl. die Aktivierung des Moduls durch einen Treiber oder die Importe des Moduls durch Platzhalter simuliert werden.

Typische Testaspekte, die für die Aufstellung von Testfällen relevant sind, umfassen:

- Funktionen des Moduls
- Modulstruktur
- Ausnahmebedingungen, Sonderfälle etc.
- Performance.

Typische Fehlerklassen, die beim Modultest festgestellt werden, sind:

- fehlende Pfade
- Berechnungsfehler
- Datendeklarationsfehler
- Datenreferenzfehler.

Voraussetzungen für den Modultest sind das Vorliegen

- der aktuellen Modulspezifikation

und

- des Modul-Feinentwurfs einschließlich des Programmlistings.

2.2.2 Teststufe: Integrationstest

Im **Integrationstest** wird geprüft, ob Gruppen von Komponenten wie im technischen Systementwurf vorgesehen zusammenspielen

Testziel des Integrationstests ist es, Schnittstellenfehler aufzudecken.

Typische Softwarefehler, die durch diesen Test aufgedeckt werden, sind:

- Falsche Schnittstellenformate
- Fehler im Datenaustausch zwischen den Komponenten
- Veränderte Komponenten oder Systeme an den Schnittstellen.

Beim Integrationstest werden sowohl interne Schnittstellen zwischen den Komponenten als auch die Schnittstellen zu externen Softwaresystemen geprüft.

Grundsätzlich wird zwischen zwei Arten von Integrationsstrategien unterschieden:

- **Non-inkrementelle** oder Big-Bang-Integration

  Alle Komponenten werden auf einmal zusammengefügt und getestet

- **Inkrementelle Integration**

  Die verschiedenen Komponenten werden nacheinander zusammengefügt. Die Reihenfolge hängt vom gewählten Verfahren ab:

  - **Top-down-Integration**

    Die Integration und der Test beginnen mit dem Softwarebaustein, der weitere Bausteine aufruft, aber selbst nicht aufgerufen wird.

  - **Bottom-up-Integration**

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

  - **Funktional orientierte Integration**

    Die verschiedenen Softwarebausteine werden nach ihrer funktionalen Zusammengehörigkeit integriert und getestet. Besonderes Gewicht wird dabei auf die Kernfunktionen der Anwendung gelegt.

  - **Ad-hoc-Ansatz**

    Die verschiedenen Softwarebausteine werden in der zufälligen Reihenfolge ihrer Fertigstellung integriert und getestet.

2.2.3 Teststufe: Systemtest

Im Systemtest wird geprüft, ob das System als Ganzes die spezifizierten Anforderungen erfüllt. Es sollte nur das zu testende System betrachtet werden und nicht die angrenzenden Systeme, die Daten entgegennehmen und/oder Daten anliefern.

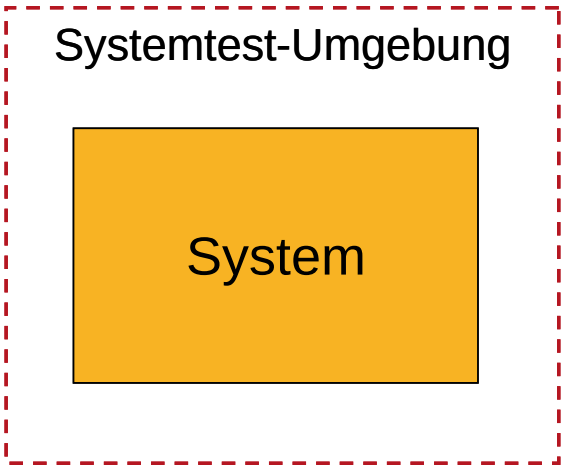


Abbildung 2-3: Teststufe Systemtest

Testziele im Systemtest:

- Test des integrierten Systems um zu prüfen, ob und wie gut die spezifizierten Anforderungen vom Produkt erfüllt werden
- Aufdeckung falscher, unvollständiger und widersprüchlich umgesetzter Anforderungen
- Identifizierung vergessener und undokumentierter Anforderungen
- Betrachtung des Systems aus Kundensicht
- Die Einbettung in die Umgebung ist gewährleistet
- Das Systemdesign ist tragfähig.

Der Systemtest wird als so genannter **Black-Box-Test** durchgeführt.

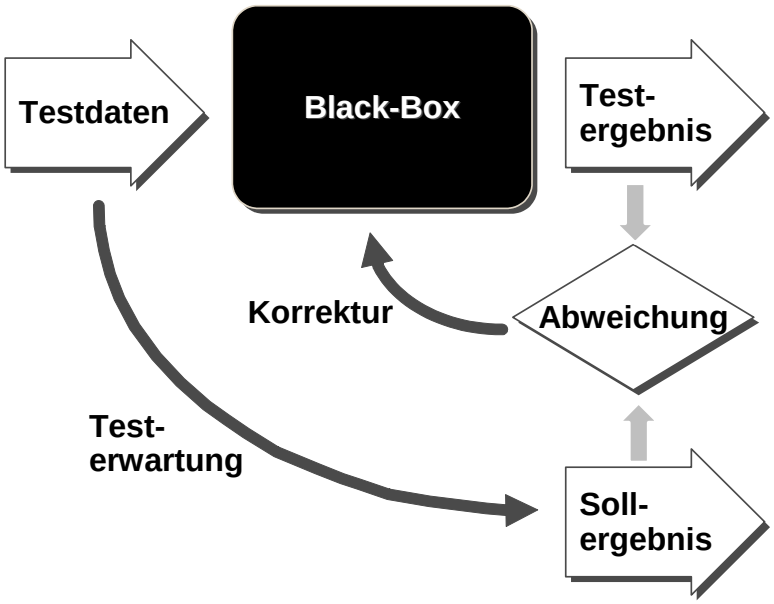


Abbildung 2-4: Black-Box-Test

Bei dieser Strategie interessiert nicht das interne Verhalten oder die Struktur des Testobjekts.

Das Testobjekt ist für den Tester eine Black-Box, die Daten aufnimmt und wieder ausgibt. Was das Testobjekt intern mit diesen Daten macht, wie sie verarbeitet werden und welchen Weg sie innerhalb des Testobjektes durchlaufen, hat für den Tester keine Bedeutung.

Der Black-Box-Test stützt sich allein auf die Spezifikationsunterlagen, die zum Testobjekt gehören. Testziele bestehen nicht darin, gewisse Pfade im Testobjekt abzudecken, sondern darin, bestimmte Aufgaben oder Funktionen gemäß Spezifikation auszuführen. Ein vollständiger Test umfasst jede Kombination der Eingabewerte (zulässige und unzulässige) und ist daher praktisch kaum möglich.

Wesentliche Voraussetzung für die Anwendung von Black-Box-Methoden ist eine genaue Spezifikation des Testobjekts.

2.2.4 Teststufe: Abnahmetest

Im Abnahmetest (auch UAT – user acceptance test – genannt) wird geprüft, ob das System aus Kundensicht die vertraglich vereinbarten Leistungsmerkmale aufweist.

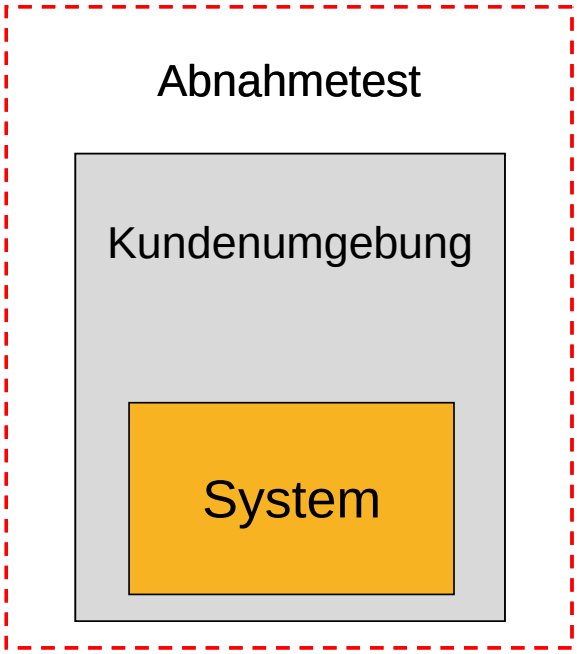


Abbildung 2-5: Teststufe Abnahmetest

Der Abnahmetest findet grundsätzlich in Abnahmeumgebung des Kunden statt. Die Sicht und das Urteil des Kunden stehen im Vordergrund.

Testziele im Abnahmetest:

- Entspricht das System den Erwartungen der unterschiedlichen Anwendergruppen?
- Wird das System von allen Anwendergruppen akzeptiert?
  - “Vertrauensbildende Maßnahme”
- Einsatzfähigkeit
- Vollständigkeit.

2.3 Validierung und Verifikation

2.3.1 Begriff: Validierung

Bei Anwendung eines Vorgehensmodells wird das Softwaresystem zunehmend detaillierter beschrieben. Da Fehler am einfachsten auf derselben Abstraktionsstufe gefunden werden, auf der sie entstanden sind, wird jeder Entwicklungsstufe eine korrespondierende Teststufe zugeordnet.

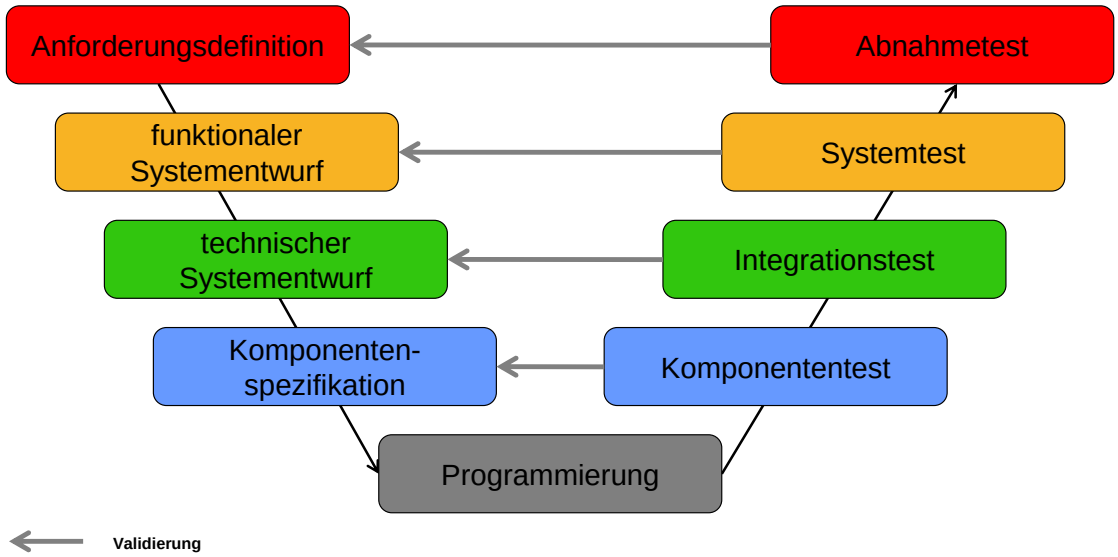


Abbildung 2-6: Validierung

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

Dieses Prüfen der Entwicklungsergebnisse gegen die ursprünglichen Anforderungen wird Validierung genannt (validieren = bekräftigen, für gültig erklären)

Beim Validieren bewertet der Tester, ob ein Produkt seine festgelegte Aufgabe tatsächlich löst und deshalb für seinen Einsatzzweck tauglich ist.

2.3.2 Begriff: Verifikation

Verifikation 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.

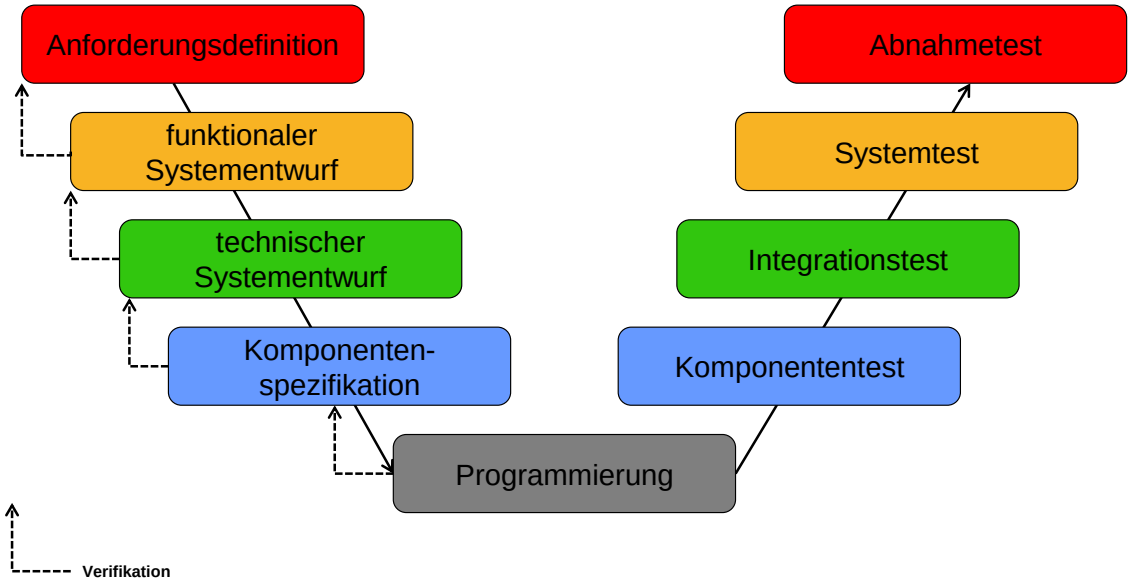


Abbildung 2-7: Verifikation

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

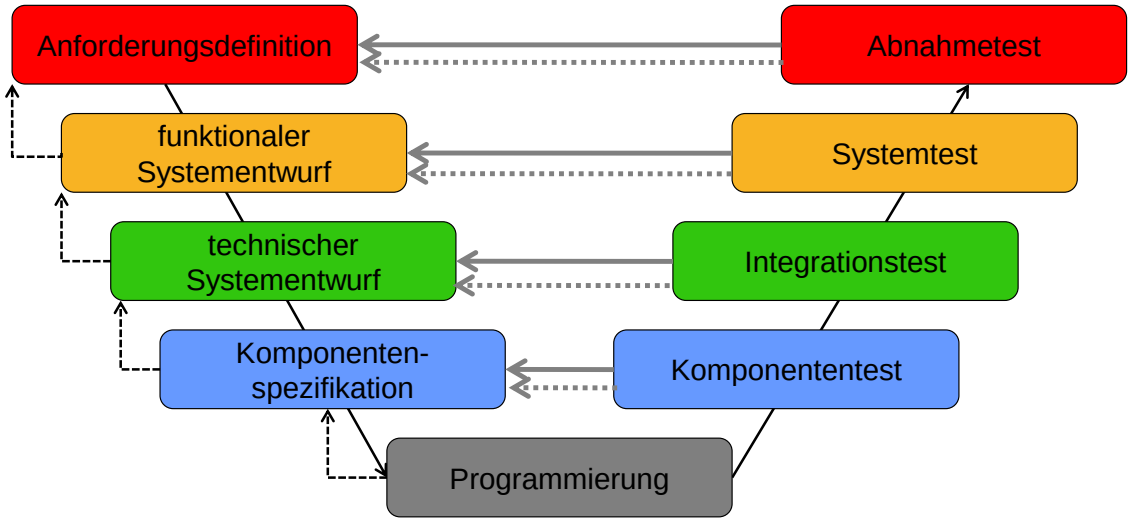


Abbildung 2-8: Verifikation und Validierung

2.4 Übersicht: Fundamentaler Testprozess

Um in einem Softwareprojekt Tests strukturiert durchführen zu können, muss die Aufgabe „Testen“ in kleinere Arbeitsabschnitte gegliedert werden:

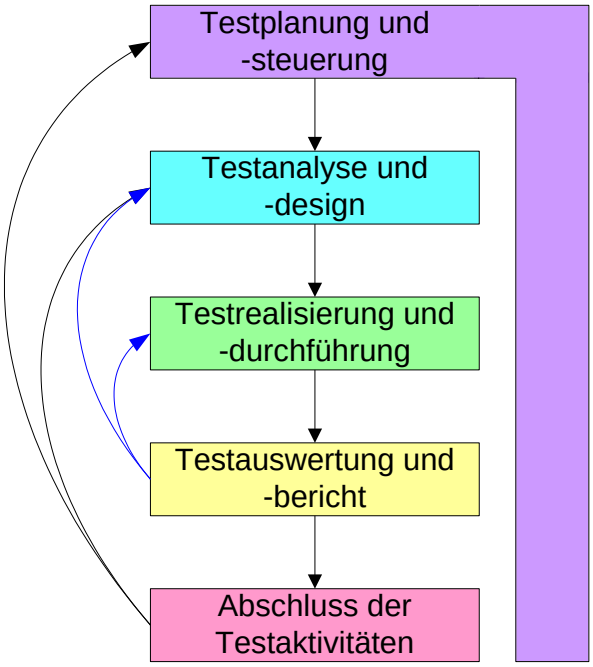


Abbildung 2-9: Fundamentaler Testprozess

Obgleich die Aufgaben in sequentieller Reihenfolge im Testprozess angegeben sind, können sie sich überschneiden und teilweise auch gleichzeitig durchgeführt werden.

Diese Teilaufgaben bilden den **so genannten fundamentalen Testprozess** und werden im Folgenden näher erläutert.

2.5 Erläuterung der Testphasen

2.5.1 Testplanung und -steuerung

Mit der Planung des Tests wird am Anfang des Software-Entwicklungsprojektes begonnen.

Aufgaben und Zielsetzung des Tests müssen festgelegt werden.

Für den gesamten Testprozess müssen folgende Projektfaktoren geplant werden:

- Aufwand
- Termine / Meilensteine
- benötigte Ressourcen
- Sonstige Hilfsmittel
- Kosten.

Eine weitere Kernaufgabe der Planung ist die Bestimmung der Teststrategie. Da ein vollständiger Test nicht möglich ist, müssen Prioritäten anhand einer Risikoeinschätzung gesetzt werden.

Kritische Systemteile müssen eine erhöhte Aufmerksamkeit im Test erfahren, also intensiver getestet werden.

Ziel der Teststrategie ist die optimale Verteilung der Tests auf die „richtigen“ Stellen des Softwaresystems.

Die Ergebnisse der Planung sind im **Testkonzept** (Testplan) festzuhalten.

2.5.2 Testanalyse und -design

Hauptaufgaben:

- Analyse, ob die Vorgaben detailliert genug sind, um daraus Testfälle abzuleiten
- Spezifikation der Testfälle unter Anwendung der im Testkonzept festgelegten Vorgehensweise (= logische Testfälle)
- Definition von Testdatenbeständen, die vorhanden sein müssen, damit der Testfall durchgeführt werden kann
- Festlegung der benötigten Testinfrastruktur und deren Einrichtung
- Ggf. Vorbereitung von Testrahmen

Vorbereitung eines Smoke-Tests; wird zu Beginn der Phase „Testdurchführung“ durchgeführt, um zu prüfen, ob Testobjekt und Testinfrastruktur eine minimale Reife (Testbarkeit) besitzen.

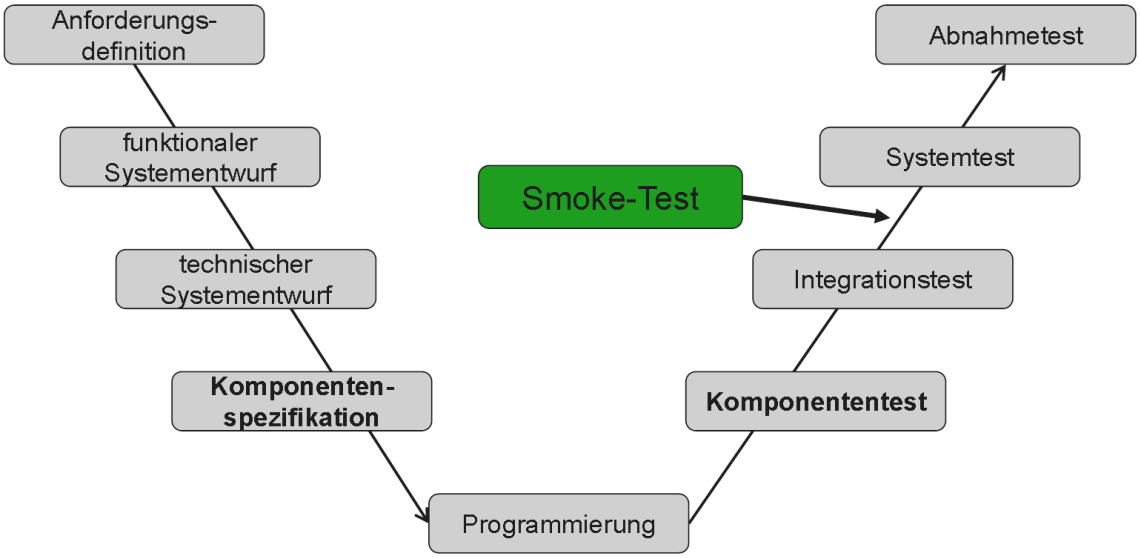


Abbildung 2-10: Smoketest

2.5.3 Testrealisierung und -durchführung

Hauptaufgaben:

- Erstellung und ggf. Priorisierung konkreter Testfälle (mit Daten, Szenarien).
- Einrichtung von Testrahmen
- Ggf. Vorbereitung einer Testautomatisierung
- Ausführung der geplanten Tests
- Soll-/Ist-Vergleich der Ergebnisse und ggf. Durchführung der Abweichungsanalyse
- Protokollierung der Tests bzw. der Fehler
- Nachtest der Testfälle, die fehlerhaft waren (nach Korrektur durch die Entwicklung
- Durchführung Regressionstests (bei Bedarf)

2.5.4 Testauswertung und -bericht

Hauptaufgaben:

- Bewertung der Gesamtheit der festgestellten Abweichungen
- Aufbereitung der Ergebnisse für die Entscheidungsträger
- Ermittlung, ob das festgelegte Kriterium zur Beendigung der Tests erfüllt ist
- Ggf. sind die Kriterien anzupassen und weitere Tests vorzusehen
- Ggf. sind weitere Testendekriterien heranzuziehen

2.5.5 Abschluss der Testaktivitäten

Hauptaufgaben:

- Prüfen, inwiefern alle geplanten Arbeitsergebnisse erstellt und geliefert wurden
- Abschluss von Vorfalls Berichten (Fehlermeldungen etc.)
- Ggf. Anstoßen von Änderungsanforderungen für weiter bestehende Fehler
- Dokumentation des Abschlusses der Testaktivitäten
- Abschluss, Dokumentation und Archivierung der Testmittel
- Ggf. Übergabe der Testmittel an die Wartungsorganisation
- Ermittlung der Lessons Learned (was ist beim nächsten Mal zu beachten, um Fehler von vorne herein zu vermeiden)

3

Testmethoden und -verfahren zur Erstellung der Testfälle

3 Testmethoden und -verfahren zur Erstellung der Testfälle

3.1 Definition logischer Testfall

Der „Logische Testfall“ ist ein Testfall ohne Angaben von konkreten Ein- und Ausgabewerten; es werden meist nur Wertebereiche angegeben. Die Anzahl der logischen Testfälle ist in der Regel überschaubar.

Der logische Testfall wird auch als abstrakter Testfall bezeichnet.

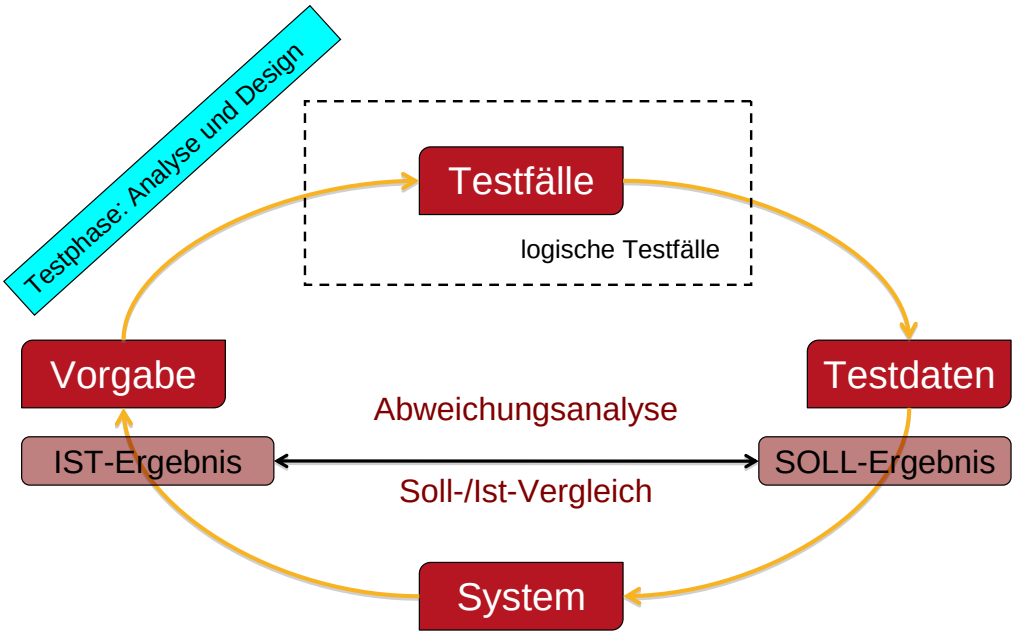


Abbildung 3-1: Übersicht Testaktivitäten – Erstellung logische Testfälle

Im Rahmen der Testphase „Analyse und Design“ werden aus den Vorgaben die logischen Testfälle abgeleitet. Dies gilt grundsätzlich für alle Entwicklungs- und dazugehörenden Teststufen.

Voraussetzung ist, dass es für die jeweilige Teststufe auch entsprechende Entwicklungsergebnisse gibt.

Beispiel:

Sollen im Systemtest die Funktionen eines Dialogsystems auf Richtigkeit geprüft werden (z. B. das Anlegen eines neuen Versicherungsvertrags für einen bestehenden Versicherungsnehmer), so muss hierfür eine entsprechende Beschreibung (beispielsweise Use-Case-Beschreibung) in der Phase „Funktionaler Systementwurf“ erstellt worden sein.

Die Qualität der Testfälle hängt unmittelbar von der Qualität der jeweiligen Vorgabe ab.

Dies bedeutet umgekehrt:

Ist die Vorgabe mangelhaft oder fehlt sie im Extremfall komplett, fehlt damit auch die Basis für die Testfallermittlung.

Konsequenzen daraus:

- Für die Testfallerstellung ist auf alle Fälle Fachwissen erforderlich
- Ggf. müssen Testfälle aus dem zu testenden System abgeleitet werden
- Man verzichtet komplett auf die Testfallerstellung und geht sofort in die Testdurchführung, sobald das System zur Verfügung gestellt wird (→ exploratives Testen; Erläuterung dazu später).

Hauptproblem:

Das Ziel, Testaufwände in frühere Testphasen (z. B. Analyse und Design) zu verlagern, wird damit nicht erreicht. Die wesentlichen Aufwände entstehen in der Testdurchführung.

3.2 Ziele der Testfallerstellung

Der Entwurf der Testfälle ist eine sehr wesentliche Tätigkeit, da von ihr die Qualität des Tests und damit die des gesamten lauffähigen Systems abhängt.

Dabei werden einerseits die Testfälle so optimiert, dass ihre Zahl und die dadurch hervorgerufene Zahl der Testläufe möglichst klein bleibt, andererseits ein möglichst hoher Deckungsgrad erzielt wird.

Grundsätzlich sollte man sich vor oder spätestens während der Erstellung der logischen Testfälle Gedanken machen, welche Testabdeckung erreicht werden soll.

Folgende Möglichkeiten gibt es:

- Funktionale Sicht
  - Jede Funktion soll mindestens einmal durchlaufen werden
  - Jeder Bedingungsausgang muss mindestens einmal durchlaufen werden
  - Jede mögliche Bedingungskombination muss getestet werden
- Datenmässige Sicht
  - Jeder Dateninhalt muss auf Gültigkeit geprüft werden.

Dazu folgendes Beispiel:

Es soll der Prozess „Brief drucken“ getestet werden. Der Prozess besteht aus 5 Funktionen.

Ziele: Beispiel „Brief drucken“

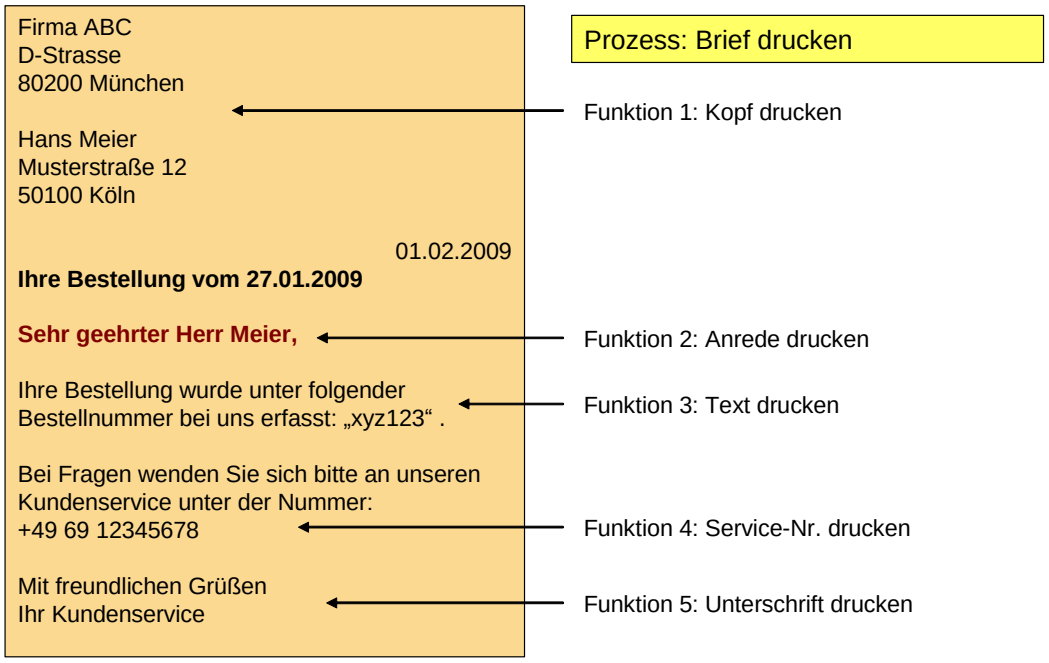


Abbildung 3-2: Beispiel „Brief drucken“ (1)

Die Funktion „Anrede drucken“ sieht 3 Varianten vor (abhängig von einem bei den Personendaten gespeicherten Kennzeichen):

- Sehr geehrte Frau
- Sehr geehrter Herr
- Sehr geehrte Damen und Herren

In der Funktion „Service-Nr. drucken“ können, abhängig von der erfassten Bestellung, 2 unterschiedliche Nummern ausgedruckt werden.

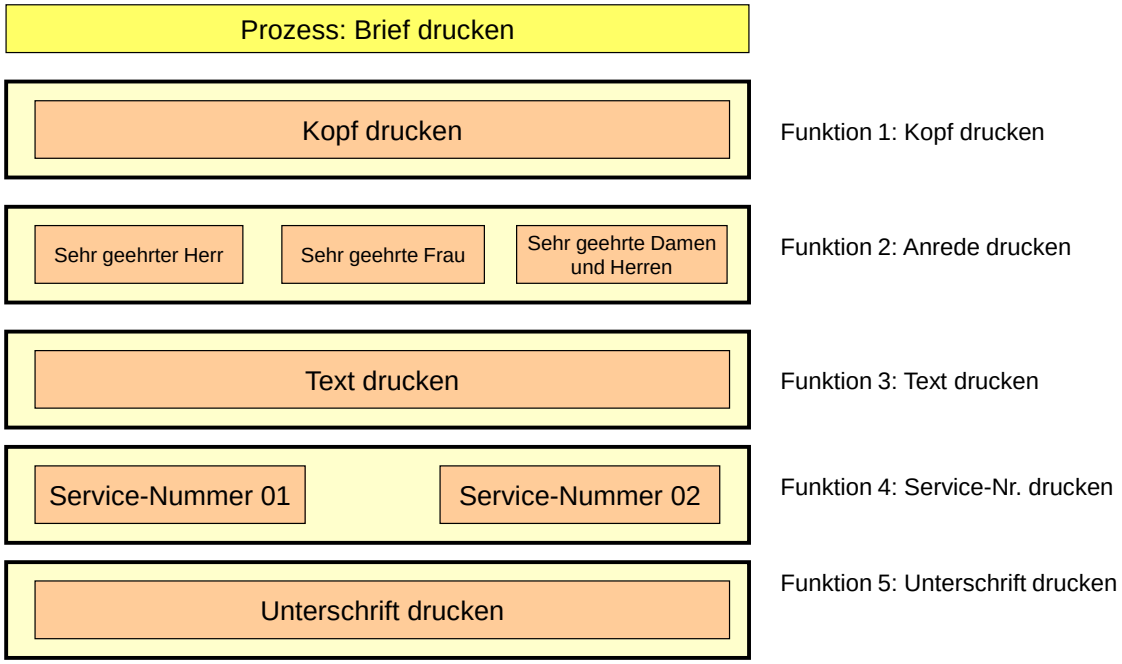


Abbildung 3-3: Beispiel „Brief drucken“ (2)

Abhängig von der Testabdeckung, die erreicht werden soll, wird eine unterschiedliche Anzahl an Testfälle benötigt.

Jede Funktion soll mindestens einmal durchlaufen werden

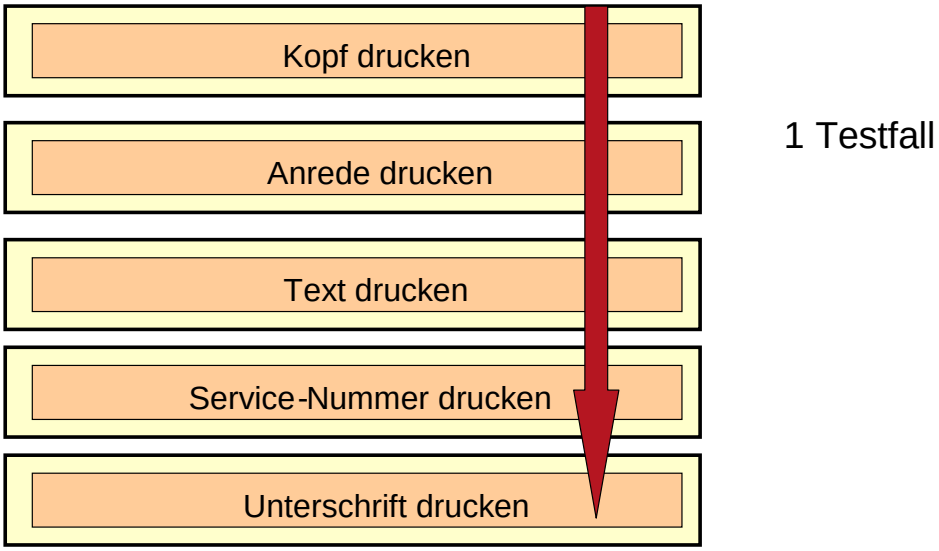


Abbildung 3-4: Beispiel „Brief drucken“ (3)

Um jede Funktion des Prozesses „Brief drucken“ einmal zu durchlaufen, genügt 1 Testfall. Die Möglichkeiten in den Funktionen „Anrede drucken“ und „Service-Nr. drucken“ spielen dabei keine Rolle. Es ist ausreichend, den Test mit einer beliebigen Person und einer beliebigen Bestellung durchzuführen.

Jeder Bedingungsausgang muss mindestens einmal durchlaufen werden

Wie viele Testfälle werden benötigt, um jeden **Bedingungsausgang** einmal zu durchlaufen?

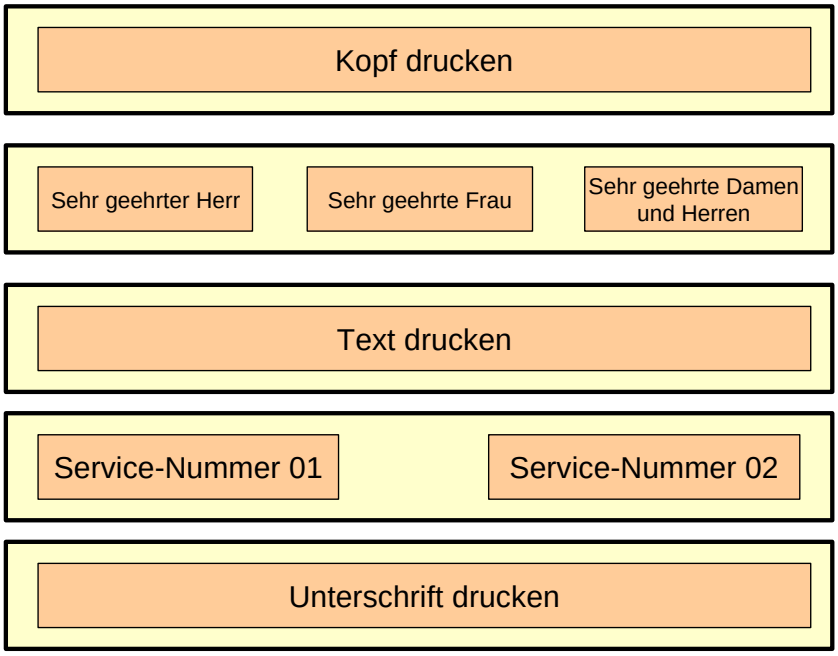


Abbildung 3-5: Beispiel „Brief drucken“ (4)

Hier muss analysiert werden, ob es in einer Funktion zu einer Abfrage (Bedingung) mehrere Möglichkeiten (Ausgänge) gibt.

Funktion „Anrede drucken“:

Wenn Person = weiblich, dann Ausdruck = Sehr geehrte Frau

Wenn Person = männlich, dann Ausdruck = Sehr geehrter Herr

Wenn Person = juristische Person, dann Ausdruck = Sehr geehrte Damen und Herren

Funktion „Service-Nr. drucken“:

Wenn Bestellung <= 10 Artikel, dann Ausdruck Service-Nummer 1

Wenn Bestellung > 10 Artikel, dann Ausdruck Service-Nummer 2

Es muss für jeden möglichen Ausgang innerhalb der Funktionen ein Testfall definiert werden.

Dies bedeutet:

1. Testfall: Person ist weiblich und bestellt 5 Artikel
2. Testfall: Person ist männlich und bestellt 18 Artikel
3. Testfall: juristische Person; Anzahl der bestellten Artikel ist egal, da bereits beide Ausgänge getestet wurden; also beispielsweise Bestellung von 2 Artikel

Jede mögliche Bedingungskombination muss getestet werden

Wie viele Testfälle werden benötigt, um jede **Bedingungskombination** zu testen?

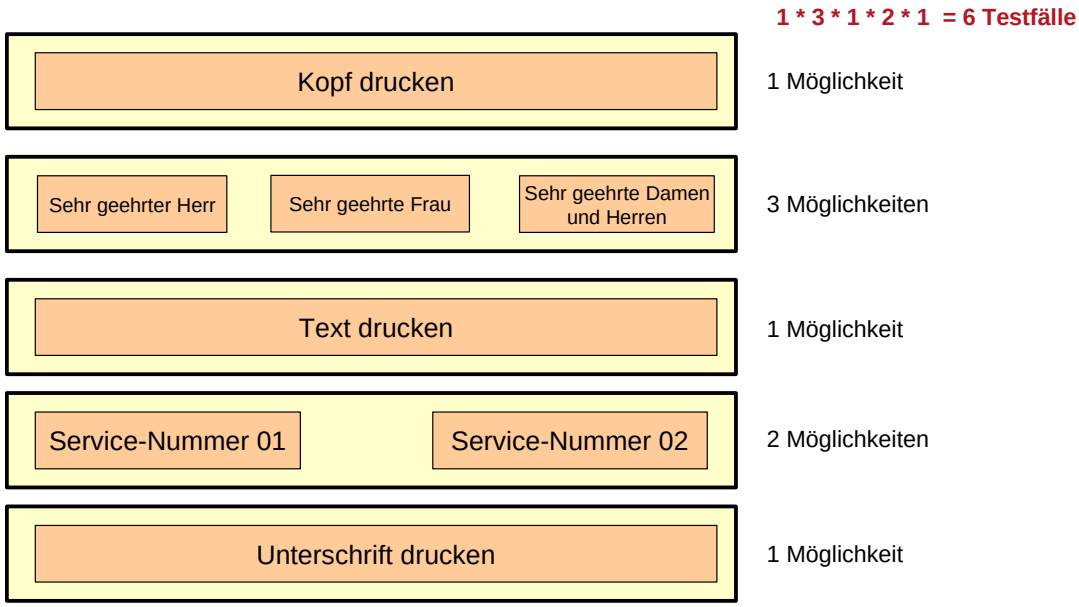


Abbildung 3-6: Beispiel „Brief drucken“ (5)

Hier muss analysiert werden, ob es fachliche Abhängigkeiten zwischen dem Personenkennzeichen (→ Ausdruck der entsprechenden Anrede) und der Anzahl der bestellten Artikel (→ Ausdruck der entsprechenden Service-Nr.) gibt.

Wäre dies der Fall, müssen alle möglichen Kombinationen getestet werden.

Dies bedeutet:

1. Testfall: Person ist weiblich und bestellt 5 Artikel
2. Testfall: Person ist weiblich und bestellt 12 Artikel
3. Testfall: Person ist männlich und bestellt 9 Artikel
4. Testfall: Person ist männlich und bestellt 99 Artikel
5. Testfall: juristische Person; bestellt 10 Artikel
6. Testfall: juristische Person; bestellt 11 Artikel

Jeder Dateninhalt muss auf Gültigkeit geprüft werden

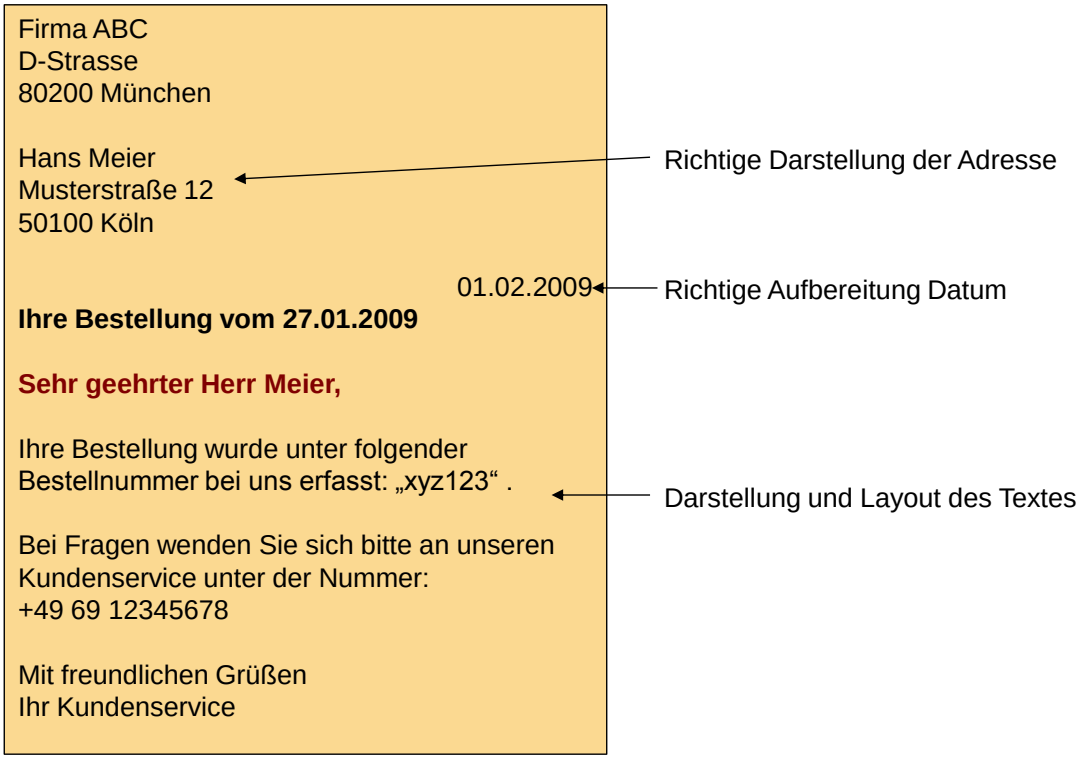


Abbildung 3-7: Beispiel „Brief drucken“ (6)

Es wird geprüft, ob die Daten korrekt ausgedruckt werden

Beispiele:

– Stimmt das Datumsformat?
– Wird die Bestellung richtig ausgedruckt?
– Wird die Adresse des Kunden richtig ausgedruckt?
– Etc.

3.3 Möglichkeiten der Testfallermittlung

3.3.1 Übersicht

Es gibt grundsätzlich 3 Möglichkeiten, um Testfälle zu ermitteln:

- Testfallermittlung durch Anwendung von Methoden (Testfallentwurfsverfahren)
- Intuitive Testfallermittlung durch Anwendung von Fachwissen und Erfahrung
- Explorative Testfallermittlung.

3.3.2 Methodische Testfallermittlung

Das Ziel des Testens ist der Nachweis der Erfüllung der festgelegten Anforderungen und die Aufdeckung von eventuellen Abweichungen und Fehlerwirkungen. Dabei sollen mit möglichst wenig Aufwand möglichst viele Anforderungen überprüft bzw. Fehlerwirkungen nachgewiesen werden. Um diesen Ziel nahe zu kommen, muss ein systematisches Vorgehen bei der Erstellung der Testfälle gewählt werden.

Zur systematischen Erstellung der Testfälle gibt es die zwei bereits beschriebenen Testfallentwurfsverfahren:

- White-Box-Verfahren
- Black-Box-Verfahren.

Die White-Box-Verfahren werden in erster Linie im Komponententest angewendet.

Im Rahmen dieses Seminars werden ausschließlich Methoden, die dem Black-Box-Verfahren zugeordnet werden, behandelt.

Dies sind:

⇒ Test mit Entscheidungstabellentechnik (nach DIN 66241)

⇒ Äquivalenzklassenverfahren

⇒ Grenzwertanalyse auf der Basis von Äquivalenzklassen.

3.3.3 Intuitive Testfallermittlung

Die Testfälle werden ohne Anwendung von Methodik ausschließlich auf Basis des Fachwissens und der Erfahrung des Testfallerstellers ermittelt.

Grundlage für diese Vorgehensweise ist die intuitive Fähigkeit und die Erfahrung, Testfälle nach erwarteten Fehlerzuständen und -wirkungen auszuwählen.

Die Testfälle basieren auf der Erfahrung, wo Fehler in der Vergangenheit aufgetreten sind, oder der Vermutung des Testers, wo Fehler in Zukunft wahrscheinlich auftreten werden.

3.3.4 Explorative Testfallermittlung

Sind die Dokumente, die als Grundlage für die Erstellung der Testfälle heranzuziehen sind, von schlechter Qualität oder nicht vorhanden, kann das so genannte „explorative Testen“ helfen.

Beim explorativen Testen werden die Testaktivitäten nahezu parallel durchgeführt. Die möglichen Elemente des Testobjekts, die einzelnen Aufgaben und Funktionen werden „erforscht“. Dann wird entschieden, welche Teile getestet werden sollen. Dabei werden wenige Testfälle zur Ausführung gebracht und das Ergebnis wird analysiert. Mit der Ausführung wird das „unbekannte“ Verhalten des Testobjekts weiter geklärt. Auffälligkeiten und weitere Informationen dienen zur Erstellung der nächsten Testfälle. Schritt für Schritt wird so das Wissen über das zu testende System angesammelt.

3.4 Test mit Entscheidungstabellen nach DIN 66241

3.4.1 Definition

Entscheidungstabellen sind Hilfsmittel zur Beschreibung komplexer Entscheidungssituationen. Die Entscheidungstabelle baut auf der Erkenntnis auf, dass Abläufe dann komplex und unübersichtlich werden, wenn die auszuführenden Aktionen nicht mehr sequentiell aufeinander folgen, sondern als Voraussetzung für ihre Ausführung zunächst geprüft werden muss, ob bestimmte Bedingungen erfüllt oder nicht erfüllt sind.

Eine Entscheidungssituation kann am besten durch die Angabe aller relevanten Entscheidungsregeln beschrieben werden. Die Entscheidungsregeln beinhalten Bedingungen, die in ihren verschiedenen „Erfüllt-“ und „Nicht erfüllt-Kombinationen“ anzeigen, wann bestimmte Aktionen ausgeführt werden müssen.

Vorteile der Entscheidungstabelle im Allgemeinen:

- Eindeutige und damit auch effektivere Verständigung
- Übermittlung komplexer Sachverhalte in komprimierter, aber trotzdem verständlicher Form
- Einfache und verständliche Dokumentation.

3.4.2 Ziele der Entscheidungstabellentechnik

Bei Anwendung der Entscheidungstabellentechnik zur methodischen Ermittlung von Testfällen sollten folgende Ziele erreicht werden:

- Ableitung von Testfällen aus fachlichen Vorgaben
- Die Testfälle können direkt aus der Entscheidungstabelle abgelesen werden
- Aufdeckung von Kombinationen, die von anderen Vorgehensweisen übersehen wurden.

Damit:

Bei Erstellung der logischen Testfälle wird nochmals automatisch eine (formale) Qualitätssicherung der Vorgaben durchgeführt.

Die Qualität der Testfälle wird durch die Möglichkeit, die Vollständigkeit nachzuweisen, wesentlich erhöht.

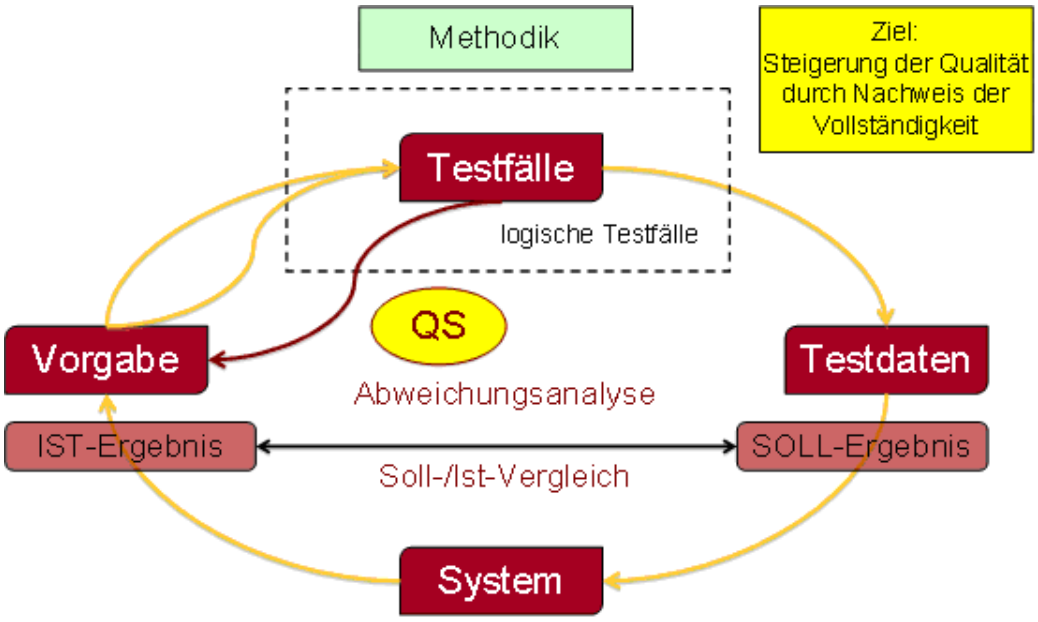


Abbildung 3-8: Übersicht Testaktivitäten – Ziel Einsatz Methodik

3.4.3 Grundsätzlicher Aufbau

Bedingung	Bedingungsanzeiger							
Wetter ist schön								
Tierpark gehen								
Kino gehen								
Aktionen	Aktionsanzeiger							

Regeln	Diese legen fest, unter welchen Voraussetzungen (Bedingungen) bestimmte Maßnahmen (Aktionen bzw. Aktionsfolgen) durchzuführen sind. Die Gesamtheit der Anzeiger (Bedingungsanzeiger und Aktionsanzeiger) einer Spalte bildet eine Regel.
Bedingung	Diese wird in Form einer Frage gestellt und ergibt zusammen mit dem Bedingungs-Anzeiger die Voraussetzung.
Bedingungsanzeiger	Die Gesamtheit der Bedingungsanzeiger pro Spalte kennzeichnet die Voraussetzung für eine Regel. Eine Regel trifft zu, wenn alle angegebenen Voraussetzungen eintreten. Bedingungsanzeiger können folgende Zeichen enthalten: J Bedingung trifft zu N Bedingung trifft nicht zu - Bedingung ist irrelevant.

Aktion	Diese wird in Form einer Anweisung definiert und beschreibt zusammen mit dem Aktionsanzeiger die durchzuführende Maßnahme.
Aktionsanzeiger	Die Gesamtheit der Aktionsanzeiger kennzeichnet die bei Eintritt dieser Voraussetzung zu ergreifenden Maßnahmen. Aktionsanzeiger kann sein: X Aktion muss ausgeführt werden.

3.4.4 Vollständige Entscheidungstabelle

Eine Entscheidungstabelle ist vollständig, wenn alle eingebbaren Bedingungskombinationen durch die angegebenen Regeln der ET (Entscheidungstabelle) abgedeckt sind.

Beispiel für vollständige Entscheidungstabelle:

Bedingung	Bedingungsanzeiger							
Schwiegermutter kommt	J	J	J	J	N	N	N	N
Verein spielt zu hause	J	J	N	N	J	J	N	N
Wetter ist schön	J	N	J	N	J	N	J	N
Tierpark gehen			x					
Kino gehen		x		x				
Stadion gehen	x	x						
Aktionen	Aktionsanzeiger							

Prüfsumme

Anzahl Regeln =

$2^{\text{Anzahl der Bed.}} = 8$

Berechnung der Anzahl der Regeln:

Anzahl Regeln = $2^{\text{Anzahl der Bedingungen}}$

Alle Bedingungen sind mit „UND“ verknüpft

Alle Regeln sind mit „ODER“ verknüpft.

Entwurf einer vollständigen ET

Folgende Schritte müssen beim Erstellen einer vollständigen Entscheidungstabelle durchgeführt werden:

- Erstellen der Bedingungen
- Erstellen der Aktionen
- Erstellung der Bedingungsanzeiger in folgenden Schritten:
  - Errechnen der maximalen Anzahl der Regeln
    - (nach der Formel: Anzahl Regeln = $2^{\text{Anzahl der Bedingungen}}$)
  - Sämtliche Bedingungsanzeiger werden von oben nach unten Bedingungsweise erstellt.
  - Die Bedingungsanzeiger der obersten Bedingung werden folgendermaßen erstellt:

    Anzahl der Regeln / 2 - 1. Hälfte „J“, 2. Hälfte „N“

    Beispiel:

    Bei 4 Bedingungen ergeben sich 16 Regeln; in der obersten Bedingungen stehen deshalb hintereinander 8 „J“ und dann 8 „N“.
  - In der jeweils darunter liegenden Bedingung wird die Anzahl der „J“ und der „N“ wiederum halbiert und von links mit „J“ beginnend die Bedingungsanzeiger aufgefüllt.
  - Dies führt dazu, dass in der untersten Bedingung immer 1 „J“ und 1 „N“ im Wechsel stehen.
- Setzen der jeweiligen Aktionsanzeiger in allen Regeln.

3.5 Konsolidierte Entscheidungstabelle und Konsolidierung

3.5.1 Definition

Der Entwurf einer vollständigen Entscheidungstabelle ist nicht möglich, wenn die Entscheidungstabelle 2 oder mehr Bedingungen beinhaltet, die:

- sich gegenseitig ausschließen,
- mit ODER verknüpft sind oder
- die Bedingung mehr als 2 Zustände annehmen kann (mehr als J und N).

Beispiel:

Aufgrund des Alters einer Person wird die Anzahl der Urlaubstage bestimmt. Wird versucht, diesen Sachverhalt in einer vollständigen Entscheidungstabelle darzustellen, ergibt sich folgender Aufbau:

Konsolidierte Entscheidungstabelle:

	R1	R2
Alter > 60	J	J
Alter > 50	J	J
Alter > 40	J	J
Alter > 30	J	N
Urlaub = 35 Tage	x	
Urlaub = 32 Tage		
Urlaub = 30 Tage		
Urlaub = 28 Tage		
Urlaub = 25 Tage		

Bereits die Regel 2 ergibt eine fachlich falsche Aussage: „Die Person ist älter als 60 und 50 und 40 aber nicht älter als 30 Jahre“. Grund: die Bedingungen sind in diesem Fall mit ODER verknüpft.

Um die Tabelle richtig zu stellen, wird in der ersten Regel in der obersten Bedingung ein „J“ gesetzt. Alle Bedingungen darunter, die mit ODER verknüpft sind, werden nicht mehr geprüft (Kennzeichnung durch eine Irrelevanzanzeiger). Alle weiteren Regeln werden entsprechend beschrieben. Dies führt zu unten stehender Entscheidungstabelle:

Alter > 60	**J**	**N**	**N**	**N**	**N**
Alter > 50	**-**	**J**	**N**	**N**	**N**
Alter > 40	**-**	**-**	**J**	**N**	**N**
Alter > 30	**-**	**-**	**-**	**J**	**N**
Urlaub = 35 Tage	**x**				
Urlaub = 32 Tage		**x**			
Urlaub = 30 Tage			**x**		
Urlaub = 28 Tage				**x**	
Urlaub = 25 Tage					**x**

3.5.2 Grundsätze für eine konsolidierte Entscheidungstabelle

Auch eine konsolidierte Entscheidungstabelle muss auf Vollständigkeit überprüft werden. Dabei wird die Anzahl der theoretisch möglichen Regeln der Anzahl der tatsächlich definierten Regeln gegenübergestellt.

Die Prüfung der Vollständigkeit erfolgt in mehreren Schritten:

- Berechnung der theoretisch möglichen Anzahl der Regeln:

  Anzahl Regeln = $2^{\text{Anzahl der Bedingungen}}$

- Bei der konsolidierten Entscheidungstabelle wird jede Regel für sich betrachtet. Es wird **pro Regel** die Anzahl der tatsächlich definierten Regeln errechnet:
  - Anzahl der tatsächlich definierten Regeln = $2^{\text{Anzahl der Irrelevanzanzeiger}}$
  - Anschließend werden die Werte der einzelnen Regeln zusammenaddiert.
  - Diese Summe muss mit der theoretisch möglichen Anzahl übereinstimmen.

Überprüfung aus Vollständigkeit am Beispiel „Alter“

$2^4 = \mathbf{16}$ **=** $2^3 + 2^2 + 2^1 + 2^0 + 2^0 = \mathbf{16}$

Alter > 60	J	N	N	N	N
Alter > 50	-	J	N	N	N
Alter > 40	-	-	J	N	N
Alter > 30	-	-	-	J	N
Urlaub = 35 Tage	x				
Urlaub = 32 Tage		x			
Urlaub = 30 Tage			x		
Urlaub = 28 Tage				x	
Urlaub = 25 Tage					x

Anzahl der theoretisch möglichen Regeln: $2^4 = 16$

Pro Regel:
Anzahl Regeln = $2^{\text{Anzahl der Irrelevanzanzeiger}}$

Addition der Werte der einzelnen Regeln

3.5.3 Entwurf einer konsolidierten Entscheidungstabelle

Folgende Schritte müssen beim Erstellen einer konsolidierten Entscheidungstabelle durchgeführt werden:

- Erstellen der Bedingungen
- Erstellen der Aktionen
- Erstellung der Bedingungsanzeiger in folgenden Schritten:
  - Die Bedingungsanzeiger werden Regelweise von links nach rechts erstellt.
  - In den ersten Regeln dürfen als Bedingungsanzeiger nur „J“ oder Irrelevanzanzeiger auftreten.
  - Bei jeder weiteren Regel gilt folgende Vorgehensweise:
    - Das niederwertigste „J“ der Vorgängerregel wird zu einem „N“.
    - Alle Bedingungsanzeiger darüber werden kopiert.
    - In den Bedingungsanzeigern darunter kann wiederum nur ein „J“ oder ein Irrelevanzanzeiger stehen.
  - Dieses Vorgehen führt zwangsläufig dazu, dass in der letzten Regel als Bedingungsanzeiger nur noch „N“ oder Irrelevanzanzeiger vorkommen können.
- Zuordnung der entsprechenden Aktionen zu den Regeln.
- Überprüfung der Entscheidungstabelle auf Vollständigkeit.

Beispiel: konsolidierte Entscheidungstabelle:

Bedingung 1	**J**	**J**	**J**	**J**	**N**
Bedingung 2	**J**	**N**	**N**	**N**	**-**
Bedingung 3	**-**	**J**	**J**	**N**	**-**
Bedingung 4	**-**	**J**	**N**	**-**	**-**
Aktion 1	**x**			**x**	
Aktion 2		**x**		**x**	
Aktion 3		**x**	**x**		
Aktion 4			**x**		
Aktion 5			**x**		**x**

3.5.4 Konsolidierung

Eine vollständige Entscheidungstabelle kann konsolidiert werden, wenn folgende Voraussetzungen gegeben sind:

- In der Entscheidungstabelle gibt es mindestens 2 Regeln, welche die gleiche(n) Aktion(en) ausführen.
- Die beiden Regeln unterscheiden sich genau in einem Bedingungsanzeiger (1 J und 1 N), alle anderen Bedingungsanzeiger sind gleich.

Beispiel: Vollständige Entscheidungstabelle

Bedingung 1	J	J	N	N
Bedingung 2	J	N	J	N
Aktion 1	x	x		
Aktion 2		x	x	x

Regel 3 und Regel 4 erfüllen die Voraussetzung für die Konsolidierung. Es wird folgendermaßen konsolidiert:

- Die beiden Regeln werden zu einer Regel zusammengefasst.
- Die Bedingungsanzeiger, die gleich sind, werden jeweils übernommen.
- Die unterschiedlichen Bedingungsanzeiger (J und N) werden durch einen Irrelevanzanzeiger ersetzt (-).
- Die Aktionen werden übernommen.

Bedingung 1	J	J	N	N
Bedingung 2	J	N	J	N
Aktion 1	x	x		
Aktion 2		x	x	x

Bedingung 1	J	J	**N**	
Bedingung 2	J	N	**-**	
Aktion 1	x	x		
Aktion 2		x		

3.6 Prozessorientierte Entscheidungstabellen

3.6.1 Definition

Die prozessorientierten Entscheidungstabellen unterscheiden sich von den üblichen Entscheidungstabellen (nach DIN 66241) im Aufbau.

Während die genormte Entscheidungstabelle die Abhängigkeit der Durchführung von Aktionen von den unterschiedlichen Bedingungskombinationen aufzeigt, wird in einer prozessorientierten Entscheidungstabelle der gesamte Prozess (also sowohl die Bedingung als auch die dazugehörende Aktion) im Bedingungsteil beschrieben.

Im Aktionsteil einer prozessorientierten Entscheidungstabelle wird die Richtigkeit des Ablaufes bezüglich der Vorgabe mit „OK“ gekennzeichnet.

Fehlerhafte Systemvarianten im Sinne der Vorgabe werden mit „NOK“ ausgewiesen.

Um große Prozesse definieren zu können, werden diese in einzelne Entscheidungstabellen strukturiert, deren Steuerung über die Aktion „WEITER ET“ erfolgen kann.

Bei prozessorientierten Entscheidungstabellen, die für die Definition von Testfällen eingesetzt werden, wird den Aktionen eine zusätzliche Zeile „Testfall“ hinzugefügt.

3.6.2 Aufbau einer prozessorientierten ET:

Bedingung : WENN …	Bedingungsanzeiger (J,N,-)		
Aktion: DANN …	Regeln (J,N,-)		
	Aktionsanzeiger (X)		
OK			
NOK			
Testfall			
WEITER ET			

Der Vorteil der prozessorientierten Entscheidungstabelle liegt in erster Linie in der Möglichkeit, einen Prozess als solches (von oben nach unten) zu beschreiben, sodass auch ein Unkundiger diesen Prozess nachvollziehen kann.

Bei der Definition der Regeln werden während der Entwicklung alle in den Bedingungen begründeten Systemvarianten / -zustände analysiert.

Beispiel eines Prozesses:

Nach Abbruch der Client-Server Verbindung erscheint die Frage, ob offline weitergearbeitet werden soll. In Abhängigkeit zur Antwort läuft das Programm entweder weiter oder es stoppt.

Aufbau einer ET nach DIN:

Verbindung bricht ab	J	J	N
Antwort ist „Weiter“	J	N	-
Frage erscheint	X		
System läuft weiter		X	X
System stoppt			

Hier kann nicht sofort erkannt werden, dass zwischen dem Abbruch und der Antwort die Frage erscheint.

Aufbau einer prozessorientierten ET:

	1	2	3	4	5	6	7	8	9	10
Verbindung bricht ab	J	J	J	J	J	J	J	N	N	N
Frage erscheint	J	J	J	J	J	J	N	J	N	N
Antwort ist „Weiter“	J	J	J	N	N	N	-	-	-	-
System läuft weiter	J	N	N	J	N	N	-	-	J	N
System stoppt	-	J	N	-	J	N	-	-	-	-
OK	X				X				X	
NOK		X	X	X		X	X	X		X
Testfall	X				X				X	

In dieser Entscheidungstabelle ist der Prozess eindeutig in seinem Ablauf definiert. Darüber hinaus sind Systemvarianten beschrieben, deren Auftreten als Fehler zu bewerten ist.

Die Anzahl der Systemvarianten entspricht genau der Anzahl der Regeln der Entscheidungstabelle nach DIN.

Wird diese Entscheidungstabelle beim Testen eingesetzt, so wird im Fehlerfalle nicht mehr der Versuch unternommen, den Fehler zu beschreiben, sondern es wird nur die jeweilige Regelnummer protokolliert.

Als Beispiel stehen hier die Regeln 3 und 6.

In beiden Fällen läuft das System weder weiter, noch stoppt es – eine Fehlermeldung im Testprotokoll wäre z. B. „der Computer ist abge-

stürzt“. Kürzer und prägnanter ist hier aber die Angabe der Regelnummer, die diesen Systemzustand beschreibt.

3.6.3 Vorgehensweise beim Aufbau einer prozessorientierten Entscheidungstabelle

Der Prozess wird wie bei der Definition einer Entscheidungstabelle nach DIN in seine Bestandteile – Bedingungen und Aktionen – zerlegt.

Diese werden von oben beginnend meist abwechselnd Zeile für Zeile in den Bedingungsteil eingetragen.

Der Aktionsteil enthält die beschriebenen Einträge OK, NOK, Testfall, WEITER ET.

Nun wird, abermals von oben beginnend, die erste Regel mit J gefüllt. Beendet wird dieses Befüllen mit dem Zutreffen (J) jener Aktion, die diese Prozessvariante (Regel) beendet.

Die weiteren Zeilen der Regel werden mit Irrelevanzanzeigern (-) aufgefüllt.

Die so erzeugte Regel kann kopiert werden. Danach wird das niederwertigste Ja – also das am weitest untenstehende J – durch ein Nein (N) ersetzt. Der Prozess muss ab dieser Stelle weiteranalysiert werden.

Alle anderen Regeln ergeben sich automatisch.

3.6.4 Vorteile

Die Entscheidungstabelle kann weder Redundanzen noch Widersprüche enthalten. Sie ist auch immer vollständig.

Es werden auch Systemvarianten beschrieben, die u. U. noch nie analysiert wurden.

3.6.5 Definition und Bearbeitung von Testfallschablonen

Die Testfallschablonen für Prozesse werden aus den prozessorientierten Entscheidungstabellen abgeleitet.

Ziel ist es dabei, die gültigen Systemvarianten eines Prozesses durchgängig von der ersten Bedingung bis zur letzten Aktion darzustellen.

Dazu werden nur die Regeln mit dem Aktionsanzeiger X bei „OK“ bzw. „Testfall“ in eine eigene Tabelle übernommen.

Jede der so entwickelten Testfallschablonen zeigt anhand der Regeln genau die Anzahl und die Stellung der Bedingungen der durchzuführenden Tests.

Einige Systemvarianten werden die volle Länge der Testfallschablone umfassen, während andere nicht bis in die unterste Zeile reichen, da eine Aktion innerhalb dieser Testfallschablone (z. B. Ausgabe eines Fehlertextes) zum Beenden dieser Variante führt. Diese Regeln werden mit einem „E“ (Ende) in der untersten Zeile gekennzeichnet.

Um auch größere Prozesse durchgängig testen zu können, werden die einzelnen Testfallschablonen unter Berücksichtigung des Eintrages WEITER ET xx untereinander gesetzt.

Die Bearbeitung der zusammenhängenden Testfallschablonen erfolgt von unten nach oben.

In jeder Testfallschablone müssen die mit „E“ (Ende) gekennzeichneten Regeln nach rechts verschoben werden.

Im nächsten Schritt muss sichergestellt werden, dass jede Regel einer Testfallschablone (ab der zweiten) von der darüber liegenden Testfallschablone mit einer durchgängigen Regel erreicht wird, die also nicht zum Ende führt.

Die auf „E“ (Ende) laufenden Regeln werden zu diesem Zweck genau soweit nach rechts geschoben, dass sie außerhalb der darunterliegenden Testfallschablone positioniert sind.

Der dadurch entstehende freie Raum wird durch Kopieren von durchgängigen Regeln dieser Testfallschablone aufgefüllt.

Jeder der beschriebenen Schritte muss für jede einzelne Testfallschablone erfolgen. Die Bearbeitung ist dann beendet, wenn alle Regeln der letzten Testfallschablone durchgängig von der ersten ab erreicht werden können.

Beispiel für ein Endergebnis:

ET1	J J J	J J J	Weitere Varianten und kopierte Varianten								N N -
ET2	J J	J J	Weitere Varianten und kopierte Varianten								
											E
ET3	J J J J J	J J J J J	Weitere Varianten und kopierte Varianten								
									E	E	
ET4	J J J	J J J	Weitere Varianten und kopierte Varianten								
								E			
ET5	J J J - - -	J J J - - -	Weitere Varianten und kopierte Varianten								
						E	E				
ET6	J J -	J J J	Weitere Varianten								
	E	E	E	E	E						

Ein typisches Anwendungsbeispiel ist der Test eines einzelnen Windows:

ET 1	Initialisierung, Focus, enabled/disabled usw.
ET 2 bis ET 5	Verarbeitungslogik, Abhängigkeitsprüfungen usw.
ET 6	Befehlstasten usw.

3.7 Strukturierung der Testfälle nach Testzielen

Ziele, die damit erreicht werden sollen

- Strukturierung der Testfälle
- Es soll mit einem oder mehreren Testfällen ein bestimmter, zielgerichteter Sachverhalt für ein Testobjekt getestet werden
- Nochmalige Qualitätssicherung der Vorgaben, da andere Sichtweise
- Testziele können beispielsweise sein:
  - Test des Aufbaus und der Inhalte einer Maske (Attribute, Wertebereiche etc.)
  - Test von Plausibilitätsprüfungen oder Geschäftsregeln
  - Test von Funktionen.

Beispiel:

Test „Durchführung einer Online-Überweisung“

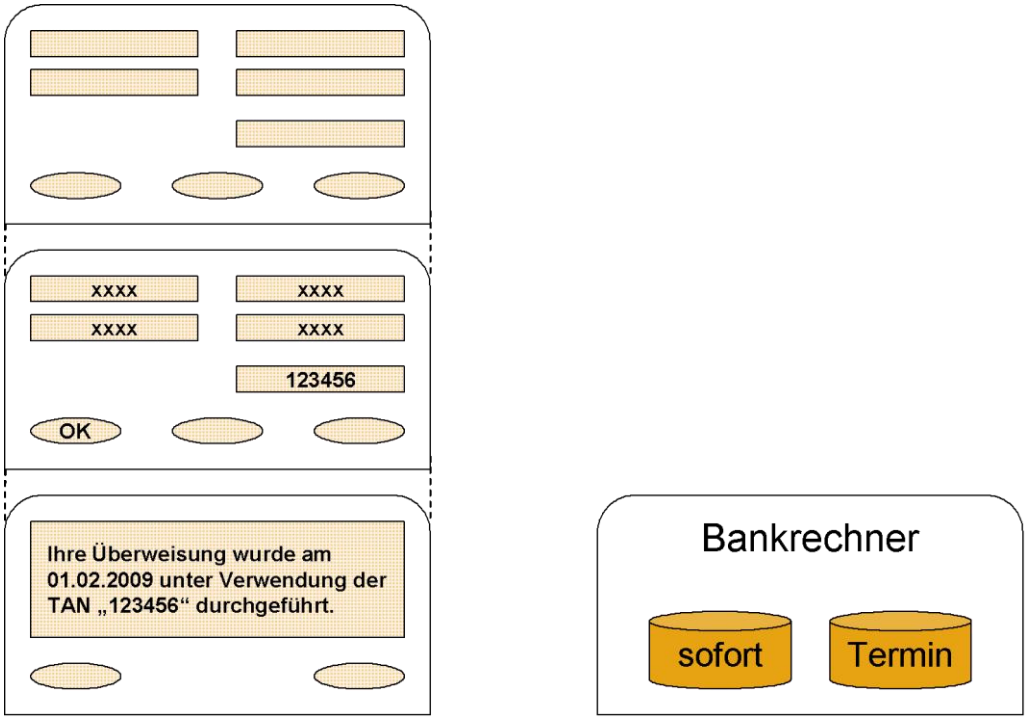


Abbildung 3-9: Testziele (Beispiel „Online-Überweisung“ – 1)

Ablauf:

Teststart: Anzeige der Maske „Überweisung erfassen“.

Es werden auf dieser Maske die erforderlichen Attribute erfasst

Nach Betätigen des Buttons „Überweisung ausführen“ werden die eingegebenen Daten im System geprüft. Bei fehlerhaften Daten wird auf der Maske „Überweisung erfassen“ ein Fehlertext ausgegeben.

Sind alle Daten richtig, wird die Maske „Überweisung ausführen“ angezeigt.

Auf dieser kann die TAN eingegeben werden. Nach Betätigung des Buttons „OK“ erscheint eine Bestätigungsmaske. Vom System werden die Daten an einen Bankrechner übermittelt. Abhängig davon, ob es sich um eine Sofort- oder Terminüberweisung handelt, erfolgt die entsprechende Weiterbearbeitung im Bankrechner.

Pro Maske können die Testfälle nach folgenden Testzielen getrennt erstellt werden:

- Inhalt
- Logik
- Standard
- Steuerung.

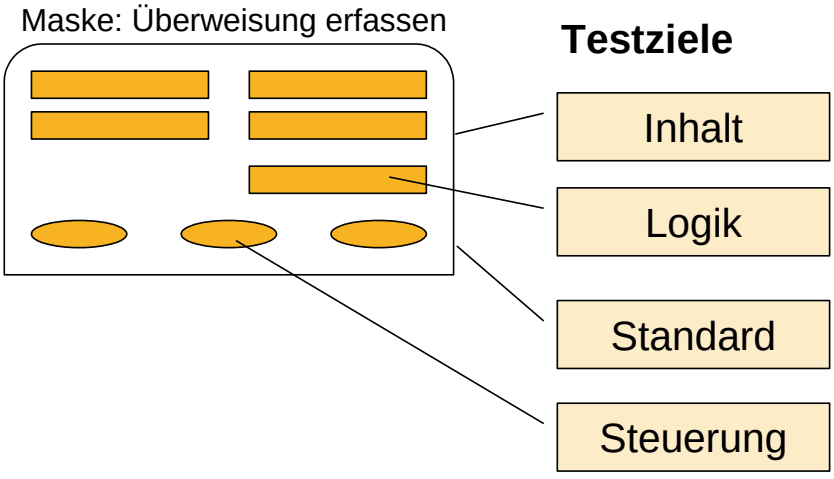


Abbildung 3-10: Testziele (Beispiel „Online-Überweisung“ – 2)

Zum Testziel **Inhalt** gehören beispielsweise folgende Tests:

- Grundsätzlicher Inhalt und Aufbau der Maske
  - Layout
  - Vollständigkeit der Felder auf der Maske
  - Feldart (Combobox, Ausgabefeld etc.)
  - Label (Feldbezeichnung)
  - Vorbelegung
  - Feldaufbereitung (1000er Punkt).
- Inhalte von Tabellen (Combobox, Grid, Listbox, Kontextmenü, Tree - View)
- Pflichtfelder
- Formatprüfungen
  - Formate (Numerisch, Alphanumerisch, Alphabetisch, Datumsfeld)
  - Zulässige Zeichen
  - Feldlängen
  - Eingabe der definierten Anzahl an Zeichen möglich
  - Eingabe weiterer Zeichen nicht möglich.

Zum Testziel **Logik** gehören beispielsweise folgende Tests:

- Werden bei Aktivierung von Radiobuttons oder Checkboxen die richtigen Zusatzfelder angezeigt
- Werden beim Füllen von Feldern automatisch die gewünschten anderen Felder mitgefüllt
- Erscheint das zu den getätigten Eingaben korrekte Ergebnis
- Erscheinen bei Fehleingaben die fachlich richtigen Fehlermeldungen.

Zum Testziel **Standard** gehören beispielsweise folgende Tests:

- Farben
- Bezeichnung und Anordnung der Buttons
- Immer nur ein Fenster aktiv (kein Wechsel zwischen zwei Fenstern möglich)
- Zeichengröße
- Tabellengröße und Zeilenzahl.

Zum Testziel **Steuerung** gehören beispielsweise folgende Tests:

- Ausgangspunkt: immer eine bestimmte Maske
- Von dieser Maske aus werden alle Möglichkeiten der Verzweigung auf andere Masken getestet.
- Der Test ist beendet, wenn die Folgemaske erscheint (noch nicht beim Standarddialog z. B. Frage nach Sichern)
- Die weitere Verarbeitung auf der Folgemaske ist nicht mehr Bestandteil dieses Tests.
- Die Verzweigung auf eine Folgemaske kann durch verschiedene Aktionen ausgelöst werden:
  - Klick auf einen Button
  - Auswahl Pulldown - Menü
  - Auswahl Kontext-Menü
  - Doppelklick auf eine Tabellenzeile innerhalb eines Grids.

Abschließend werden Testfälle für das Testziel **Ablauf** erstellt:

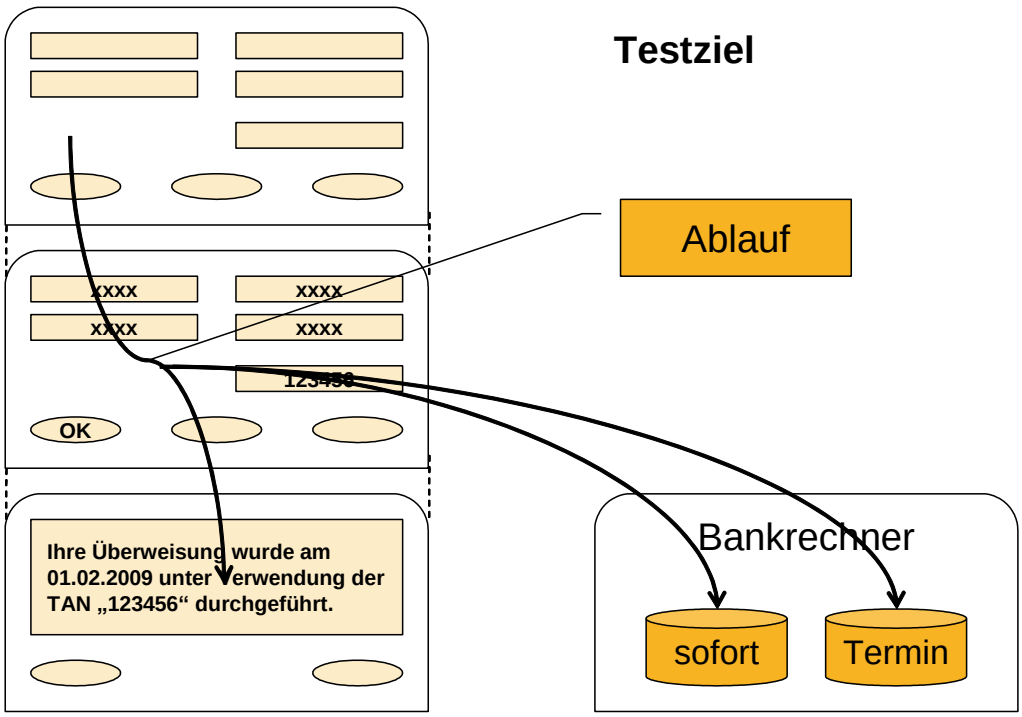


Abbildung 3-11: Testziele (Beispiel „Online-Überweisung“ – 3)

3.8 Theoretischer Ansatz

Zunächst definieren wir einige Begriffe, die rein intuitiv wohlbekannt sind. Wir haben sie in den vorausgegangenen Kapiteln in diesem allgemeinverständlichen Sinne auch bereits benutzt.

Testfall

Unter einem *Testfall* verstehen wir eine Menge von Testeingabedaten, die die gleiche Art von Wirkung beim Testobjekt hervorrufen.

Testmenge

Dies ist die Untermenge von allen Eingabedaten eines Testobjekts, die beim Testen verwendet wird.

Hinweis:

Testmengen enthalten nicht nur Eingangsdaten für ein Testobjekt, sondern auch Ausgabedaten.

Wenn z. B. bei einem Gehaltsabrechnungsprogramm das zu errechnende Monatsgehalt zwischen DM 0,- und DM 15.000,- liegen darf, so sind Testfälle aufzustellen, die zu einem Gehalt von DM 0,- und zu einem Gehalt von DM 15.000,- führen.

Des weiteren müssen wir beim Testen den **Definitionsbereich** des Testobjekts, die Menge der **zulässigen** Eingabedaten und die Menge der **unzulässigen** Eingabedaten betrachten (vgl. Abb. 3.1).

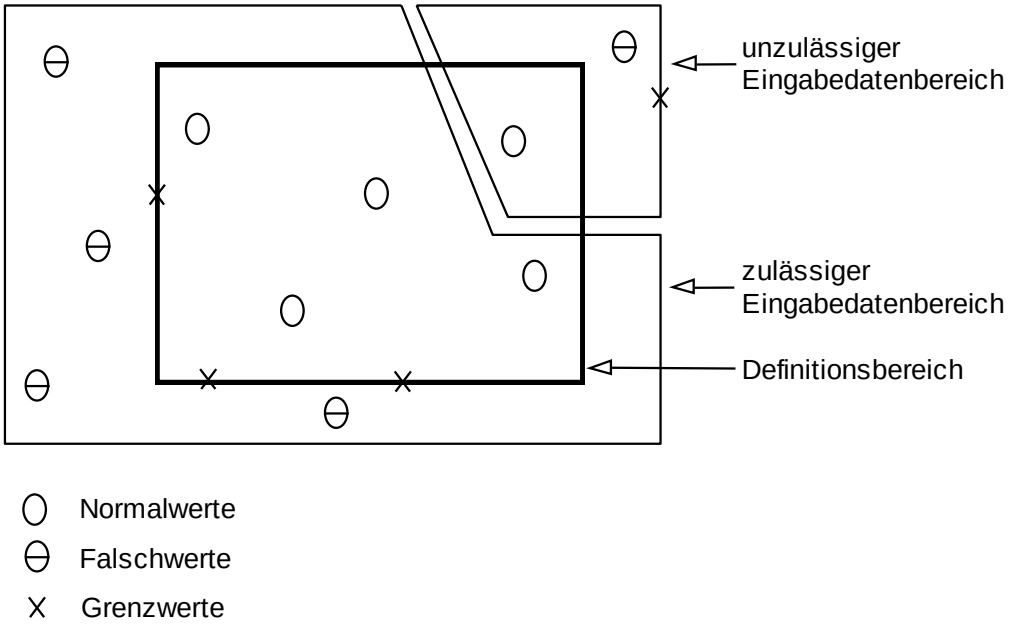


Abbildung 3-12: Wertebereiche von Daten

Stellen wir uns vor, wir haben ein überschaubares Programm zu testen. Wir kennen den Programmcode nicht im einzelnen, sondern nur die Programmvorgabe. Theoretisch könnten wir nun Testfälle für sämtliche zulässigen Eingabedaten des Programmes entwerfen. Damit würden wir überprüfen, ob das Programm das tut, was es tun soll. Da wir aber auch überprüfen müssen, ob das Programm nicht etwas tut, was es nicht tun soll, müssten wir auch Testfälle für sämtliche unzulässige Eingabedaten entwerfen.

So kommen wir auf eine unendliche Anzahl von Testfällen. Noch extremer wird die Situation bei Programmen mit 'Gedächtnis', wie z. B. Betriebssysteme, Datenbanksysteme, Flugreservierungssysteme u. ä. Hier ist eine Transaktion in der Regel von den vorhergehenden Transaktionen abhängig. Man müsste also nicht nur die zulässigen und unzulässigen Transaktionen durchprobieren, sondern auch alle möglichen Sequenzen von Transaktionen.

Fazit

Der sog. erschöpfende Datentest ist nicht möglich.

Wenn wir ein Programm von außen betrachtet also nicht vollständig testen können, könnten wir versuchen, uns die innere Struktur des Programms, die Programmlogik, die Schleifen und Verzweigungen, die Modul- und Prozeduraufrufe, beim Testen genauer vorzunehmen.

Reicht es dabei aus, so viele Testfälle zu entwerfen, dass jede Anweisung im Programm mindestens einmal ausgeführt wird? Selbstver-

ständlich reicht dies nicht aus. Dies sieht man leicht, wenn man sich Schleifen oder Entscheidungen ansieht.

Um ein Programm mit dieser Strategie vollständig auszutesten, müsste man schon so viele Testfälle entwerfen, dass jeder Pfad im Programm mindestens einmal durchlaufen wird (Abb. 3.2). Dies macht man sich am besten klar, wenn man das Programm als Graphen darstellt. Die Knoten bedeuten dabei die Anweisungen und die Kanten stellen den Ablauf dar. Astronomisch hoch wird die Anzahl der Pfade immer dann, wenn Wiederholungsschleifen im Ablaufgraphen vorkommen.

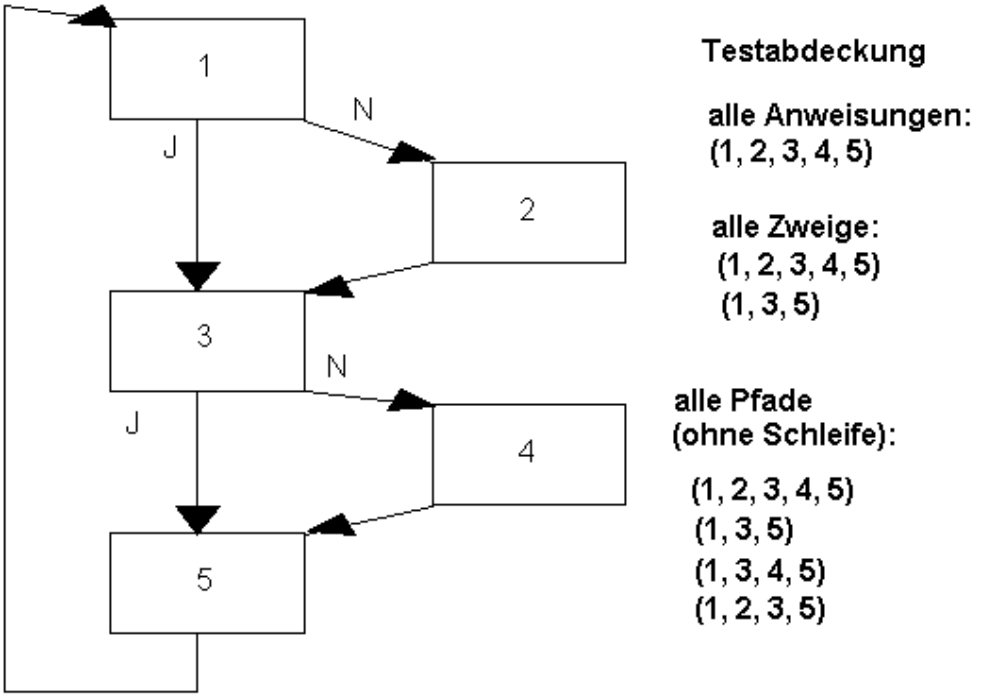


Abbildung 3-13: Ablaufgraph eines einfachen Programms

Ohne Betrachtung der Schleife gibt es im obigen Graphen 4 verschiedene Programmpfade. Bei **n** Schleifendurchläufen ergäbe sich eine Anzahl von

$$4^n + 4^{n-1} + \ldots\ldots + 4^0$$

Pfaden.

Erschöpfendes Pfadtesten ist also ebenso wie erschöpfendes Datentesten unmöglich.

3.9 Ablauforientiertes Testen (White-Box)

Beispiel-Programm:

```
PROCEDURE ZaehleZchn (    VAR VokAnz : CARDINAL;
                          VAR Geszahl : CARDINAL);

VAR Zchn : CHAR;

BEGIN

(1)  Lies (Zchn);

(2)  WHILE ((Zchn ¤ ´A´) AND (Zchn ¦ ´Z´) AND (Geszahl < MAX))
     DO

(3)       Geszahl := Geszahl + 1;

(4)       IF ((Zchn = ´A´) OR (Zchn = ´E´) OR (Zchn = ´I´) OR
                (Zchn = ´O´) OR (Zchn = ´U´)) THEN

(5)            VokAnz := VokAnz + 1;

          END; (*IF*)

(6)       Lies (Zchn);

     END; (*WHILE*)

END ZaehleZchn;
```

Diese Prozedur zählt die Großbuchstaben und Vokale innerhalb einer Zeichenkette. Sie liest und zählt solange Großbuchstaben, bis ein anderes Zeichen erkannt wird oder die Gesamtzahl einen Wert ´MAX´ überschritten hat. Unter den Großbuchstaben werden die Vokale extra gezählt.

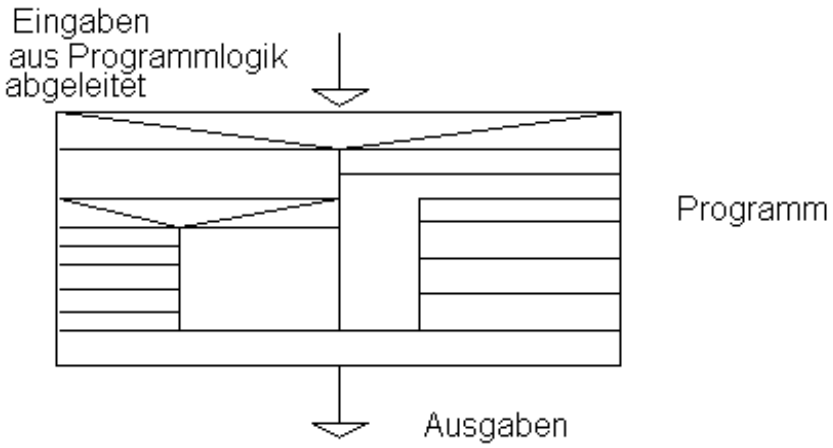
Wenn wir mit Geszahl = 0 und VokAnz = 0 starten ergibt sich beispielsweise aus dem Testfall

(0, 0, ´X´, ´C´, ´A´, ´F´, ´E´, ´H´, ´a´)

das Ergebnis Geszahl = 6 und VokAnz = 2.

Wir greifen in diesem Kapitel noch häufiger auf dieses Beispiel zurück.

3.9.1 Testabdeckungsgrad



Beim sog. White-Box Testen betrachten wir die interne Programmstruktur. Das übergeordnete Testziel besteht darin, mit möglichst wenigen Testfällen alle Anweisungen, Zweige oder sogar Pfade im Testobjekt abzudecken. Der Prozentsatz der abgedeckten Programmelemente heißt auch **Testdeckungsgrad**. Je nach Kritikalität des Testobjekts sind bei der Testplanung die angestrebten Testdeckungsgrade vorzugeben. Eine solche Vorgabe könnte lauten:

Modultest	Anweisungsabdeckung =	98 %
	Zweigabdeckung =	85 %

Hinweis:

Mit dieser Strategie wird nicht überprüft, ob das Testobjekt überhaupt das vorgegebene Problem löst, also die gestellten fachlichen und sachlichen Anforderungen erfüllt. Insbesondere fehlende Pfade, aber auch Mängel in der Spezifikation, können durch diese Vorgehensweise nicht aufgedeckt werden.

Testfälle werden bei dieser Strategie also mit dem Ziel entworfen, ganz bestimmte Kontrollstrukturen im Testobjekt zu durchlaufen. Beispielsweise braucht man einen Testfall, der bewirkt, dass in einer bestimmten Entscheidung genau der Nein-Zweig ausgeführt wird (vgl. Abb. 3.3). Dabei möchte man selbstverständlich mit einer möglichst geringen Anzahl von Testfällen auskommen.

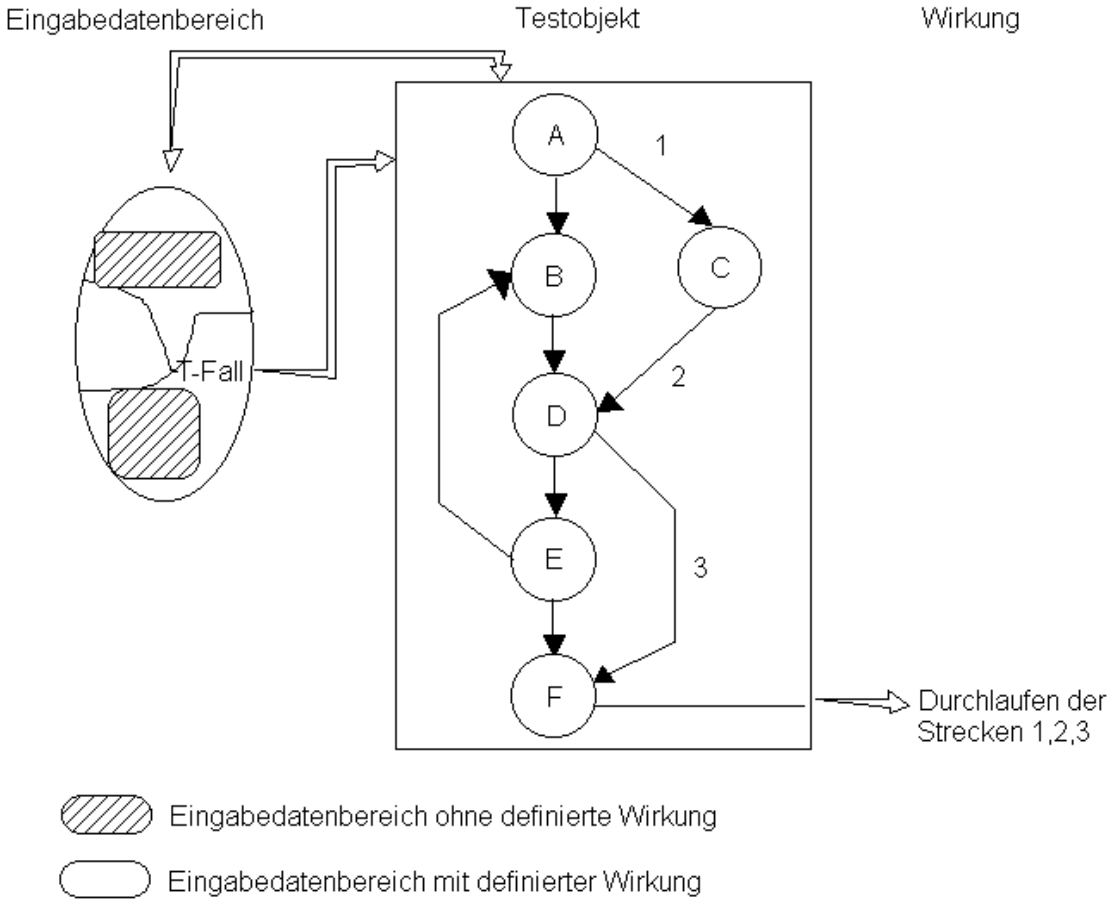


Abbildung 3-14: Testobjektorientierte Testfallbestimmung

Zur Bestimmung dieser minimalen Anzahl von Testfällen sind Entscheidungstabellen sehr nützlich (vgl. Kap 3.1.4). Aber auch das in Kapitel 1 erläuterte Komplexitätsmass v(G) von Mc Cabe führt zu einer optimalen Anzahl von Testfällen zur vollständigen Zweigüberdeckung.

Das Erreichen einer hohen Testabdeckung erfordert u. U. sehr viele Testfälle. Eine Zweigabdeckung von 50 % kann z. B. mit nur 2 Testfällen erreicht werden. Für 70 % Zweigabdeckung braucht man dann vielleicht schon 5 Testfälle und für 90 % Zweigabdeckung gar 15 Testfälle.

Für den Modultest unterscheiden wir die sog. **Ck**-Abdeckungsmasse. Dabei bedeutet:

C0 = Anweisungsabdeckung

Verhältnis der Anzahl der durchlaufenen Anweisungen zur Gesamtzahl der Anweisungen eines Testobjekts.

C0 = 100 % bedeutet, dass alle Anweisungen mind. einmal mit einem Testfall ausgeführt wurden.

C1 = Zweigabdeckung

Verhältnis der Anzahl der durchlaufenen Zweige zur Gesamtzahl der Zweige eines Testobjekts.

C1 = 100 % bedeutet, dass alle Entscheidungen mind. einmal mit 'Ja' und einmal mit 'Nein' ausgeführt wurden.

C2 = Abdeckung aller Bedingungskombinationen

Verhältnis der Anzahl der durch Testfälle abgedeckten Bedingungskombinationen zur Gesamtzahl der möglichen Bedingungskombinationen.

C2 = 100 % bedeutet, dass innerhalb eines Testobjekts alle möglichen Kombinationen von Bedingungen durch Testfälle abgedeckt wurden.

C3 = Pfadabdeckung

Verhältnis der Anzahl der durch Testfälle abgedeckten Pfade zur Gesamtzahl der Pfade eines Testobjekts.

C3 = 100 % bedeutet, dass alle möglichen Pfade durch Testfälle abgedeckt wurden. Falls das Testobjekt keine Wiederholungsschleife hat, ist C2 = C3.

In der Praxis begnügt man sich in der Regel mit C0 = 95 % und C1 = 85 %, da sonst der Testaufwand zu hoch wird.

3.9.2 Bedingungsüberdeckung

Beim C1 - und beim C2 - Test betrachten wir den Wert der gesamten Bedingung. Nun kann eine komplexe Bedingung aber aus vielen atomaren Elementarbedingungen zusammengesetzt sein.

Beispiel (vgl. Prozedur ZaehleZchn):

a) ((Zchn ¤ ´A´) AND (Zchn ¤ ´Z´) AND (Geszahl < MAX)) enthält drei atomare Bedingungen

b) ((Zchn = ´A´) OR (Zchn = ´E´) OR (Zchn = ´I´) OR (Zchn = ´O´) OR (Zchn = ´U´)) enthält fünf atomare Bedingungen

Der C1 - und C2 - Test wird solchen komplexen Bedingungen nicht gerecht.

Einfacher Bedingungsüberdeckungstest

Hier müssen wir Testfälle aufstellen, so dass jede atomare Bedingung mind. einmal den Wert *wahr* und einmal den Wert *falsch* annimmt. Wenn in einem Programm nur die im obigen Beispiel genannten Bedingungen vorkommen, brauchen wir folgende Testfälle:

	1						2	3
Geszahl	0	1	2	3	4	5	0	MAX
Zchn	´A´	´E´	´I´	´O´	´U´	´1´	´a´	´D´
Zchn ¤ ´A´	W	W	W	W	W	F	W	W
Zchn ✪ ´Z´	W	W	W	W	W	W	F	W
Geszahl < MAX	W	W	W	W	W	W	W	F
Zchn = ´A´	W	F	F	F	F	-	-	-
Zchn = ´E´	F	W	F	F	F	-	-	-
Zchn = ´I´	F	F	W	F	F	-	-	-
Zchn = ´O´	F	F	F	W	F	-	-	-
Zchn = ´U´	F	F	F	F	W	-	-	-

Laut dieser Matrix hätten wir mit den 3 Testfällen

- ✶ (0, ´A´, ´E´, ´I´, ´O´, ´U´, ´1´)
- ✸ (0, ´a´)
- ✹ (MAX, ´D´)

den einfachen Bedingungsüberdeckungstest realisiert.

Dieser Test beinhaltet aber **keine** vollständige Zweigüberdeckung, d. h. den C1-Test.

Mehrfacher Bedingungsüberdeckungstest

Hier reicht es bei einer komplexen Bedingung nicht aus, jede atomare Bedingung mind. einmal *wahr* und einmal *falsch* werden zu lassen. Hier müssen wir versuchen, sämtliche Kombinationen der Wahrheitswerte der einzelnen atomaren Bedingungen darzustellen. In der Praxis sind viele Bedingungskombinationen aber gar nicht darstellbar.

Betrachten wir dazu das aus 5 atomaren Elemetarbedingungen zusammgesetzte Beispiel b) von oben. Es ergibt sich eine Wahrheitswertetabelle mit 2**5 = 32 logisch möglichen Variationen, von denen aber nur 6 herstellbar sind.

Zchn	´A´	´E´	´I´	´O´	´U´	sonstige	?	?	?
Zchn = ´A´	W	F	F	F	F	F	W	...	W
Zchn = ´E´	F	W	F	F	F	F	W	...	W
Zchn = ´I´	F	F	W	F	F	F	F	...	W
Zchn = ´O´	F	F	F	W	F	F	F	...	W
Zchn = ´U´	F	F	F	F	W	F	F	...	W
Bed. b)	W	W	W	W	W	F	W	...	W
		6 Variationen darstellbar					26 Variationen nicht darstellbar		

Dieser Mehrfach-Bedingungsüberdeckungstest schließt die Zweigüberdeckung ein. Er ist jedoch aufwendig zu realisieren und enthält zahlreiche in der Praxis nicht darstellbare Fälle.

Minimale Mehrfach-Bedingungsüberdeckung

Hier sieht unsere Wahrheitswertetabelle folgendermaßen aus:

	1							2	3
Geszahl	0	1	2	3	4	5	6	0	MAX
Zchn	´A´	´E´	´I´	´O´	´U´	´B´	´1´	´a´	´D´
Zchn ¤ ´A´	W	W	W	W	W	W	F	W	W
Zchn ⭘ ´Z´	W	W	W	W	W	W	W	F	W
Geszahl < MAX	W	W	W	W	W	W	W	W	F
Bed. a)	W	W	W	W	W	W	F	F	F
Zchn = ´A´	W	F	F	F	F	F	-	-	-
Zchn = ´E´	F	W	F	F	F	F	-	-	-
Zchn = ´I´	F	F	W	F	F	F	-	-	-
Zchn = ´O´	F	F	F	W	F	F	-	-	-
Zchn = ´U´	F	F	F	F	W	F	-	-	-
Bed. b)	W	W	W	W	W	F	-	-	-

Bei diesem Test muss nicht nur jede atomare Elementarbedingung einer komplexen Bedingung mind. einmal der Wert wahr und einmal den Wert falsch annehmen, sondern zusätzlich fordern wir dies auch für die übergeordneten zusammengesetzten Teilbedingungen. Wie wir der obigen Tabelle entnehmen können, brauchen wir das Testdatum Zchn = ´B´ zusätzlich, damit Bedingung b) den Wert ´F´ annimmt.

Dieser minimale Mehrfach-Bedingungsüberdeckungstest enthält den Zweigüberdeckungstest. Er ist aber schärfer als dieser.

3.9.3 Boundary interior-Pfadtest

Dieses Testverfahren liegt zwischen Zweigüberdeckung (C1) und Pfadüberdeckung (C3). Der vollständige Pfadabdeckung ist aufgrund der durch Schleifen verursachten hohen Anzahl von Pfaden in der Praxis nicht durchzuführen. Der *Boundary interior-Pfadtest* gibt uns eine Methode für den intensiven Test von Schleifen an.

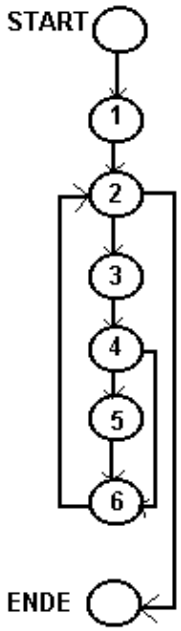
Für jede Schleife des zu testenden Programms gibt es zwei Klassen von Pfaden:

Eine Klasse enthält alle Pfade, die die Schleife zwar betreten, sie jedoch nicht wiederholen. Diese werden als Grenztests (*boundary tests*) bezeichnet.

Die andere Klasse umfasst alle Pfade, die mindestens eine Schleifenwiederholung enthalten. Diese Pfade dienen zum Test des Schleifeninneren (*interior tests*).

Von den zum Grenztest gehörenden Pfaden werden jene ausgeführt, die unterschiedliche Pfade innerhalb des Schleifenkörpers verfolgen. Die Testfälle für den Test des Schleifeninneren verfolgen entsprechend die unterschiedlichen Pfade während der ersten beiden Ausführungen des Schleifenkörpers.

Wir nehmen wieder das Beispiel *ZaehleZchn* mit dem zugehörigen Kontrollflussgraphen zur Demonstration des Boundary interior-Pfadtests.




```
BEGIN
(1)   Lies (Zchn);
(2)   WHILE ((Zchn ≥ ´A´) AND (Zchn ≤ ´Z´) AND (Geszahl < MAX))
      DO
(3)         Geszahl := Geszahl + 1;
(4)         IF ((Zchn = ´A´) OR (Zchn = ´E´) OR (Zchn = ´I´) OR
                (Zchn = ´O´) OR (Zchn = ´U´)) THEN
(5)               VokAnz := VokAnz + 1;
            END; (*IF*)
(6)         Lies (Zchn);
      END; (*WHILE*)
END ZaehleZchn;
```

1. Testfall für Pfad außerhalb der Scghleife

Aufruf mit Geszahl = MAX

Pfad: (START, 1, 2, ENDE)

2. Testfälle für den Grenztest

a) Aufruf mit Geszahl = 0, Zchn = ´A´, ´1´

Pfad: (START, 1, 2, 3, 4, 5, 6, 2, ENDE)

b) Aufruf mit Geszahl = 0, Zchn = ´B´, ´1´

Pfad: (START, 1, 2, 3, 4, 6, 2, ENDE)

Beide Testfälle betreten die Schleife, ohne sie zu wiederholen. Sie decken die im Schleifeninneren möglichen zwei Pfade ab.

3. Testfälle für den Schleifenkörper

a) Aufruf mit Geszahl = 0, Zchn = ´E´, ´I´, ´B´, ´1´

Pfad: (START, 1, 2, 3, 4, 5, 6, 2, 3, 4, 5, 6, 2, 3, 4, 6, 2, ENDE)

b) Aufruf mit Geszahl = 0, Zchn = ´E´, ´B´, ´1´

Pfad: (START, 1, 2, 3, 4, 5, 6, 2, 3, 4, 6, 2, ENDE)

c) Aufruf mit Geszahl = 0, Zchn = ´B´, ´E´, ´1´

Pfad: (START, 1, 2, 3, 4, 6, 2, 3, 4, 5, 6, 2, ENDE)

d) Aufruf mit Geszahl = 0, Zchn = ´B´, ´B´, ´1´

Pfad: (START, 1, 2, 3, 4, 6, 2, 3, 4, 6, 2, ENDE)

Diese vier Testfälle bewirken mindestens eine Schleifenwiederholung und verfolgen die vier möglichen Pfade während der zwei Ausführungen des Schleifenkörpers.

3.9.4 Komplexitätsmasse

Auch mit Hilfe von Komplexitätskenngrößen kann eine optimale Anzahl von Testfällen für den White-Box Test gefunden werden. Dies ist ein fundamentaler Gesichtspunkt beim Testen aus Sicht der Wirtschaftlichkeit.

Grundlage ist die Idee, dass ein Testobjekt umso schwieriger zu testen ist, je mehr zu testende Pfade es besitzt. In folgenden Schritten gelangt man nun zu Testfällen für den White-Box Test:

1. Erstelle den Ablaufgraphen des Moduls und berechne

   v(G) = e - n + 2p, wobei

   e, die Anzahl der Kanten im Ablaufgraphen,

   n, die Anzahl der Knoten im Ablaufgraphen und

   p, die Anzahl der unabhängigen Teilgraphen im Ablaufgraphen bezeichnen.
2. Suche v(G) unabhängige Pfade im Testobjekt.
3. Erstelle Testfälle, so dass diese v(G) Pfade durchlaufen werden.
4. Ermittle für diese v(G) Testfälle die Soll-Ergebnisse anhand der Spezifikation.
5. Führe die Testläufe aus.

Manuell funktioniert dies natürlich nur bei sehr kleinen Modulen. Bei komplexeren Testobjekten braucht man in der Praxis Werkzeuge, sog. Testabdeckungsanalysatoren, um festzustellen, welche Pfade in einem Testobjekt durchlaufen werden.

3.10 Datenorientiertes Testen (Black-Box)

Bei dieser Strategie - auch Funktionstest genannt - interessiert uns **nicht** das interne Verhalten oder die interne Struktur des Testobjekts. Das Testobjekt ist für uns eine Black-Box, die Daten aufnimmt und wieder ausgibt. Was das Testobjekt intern mit diesen Daten macht, wie sie verarbeitet werden und welchen Weg sie innerhalb des Testobjekts durchlaufen, hat für uns keine Bedeutung (vgl. Abb. 3.4).

Wir stützen uns allein auf die Spezifikationsunterlagen, die zum Testobjekt gehören. Aus der Anforderungsanalyse, dem Fach- und DV-Konzept werden Testfälle abgeleitet. Testziele bestehen nicht darin, gewisse Pfade im Testobjekt abzudecken, sondern darin, bestimmte Aufgaben oder Funktionen gemäß Spezifikation auszuführen. Dabei haben die Wertebereiche der Daten und ihre Ränder eine besondere Bedeutung.

Es gibt zahlreiche Black-Box Testmethoden. Wir wollen auf die folgenden näher eingehen:

a) Aufgabenorientierte Testfallbestimmung
b) Funktionsorientierte Testfallbestimmung
c) Äquivalenzklassenbildung
d) Grenzwertanalyse
e) Ursache-Wirkung-Analyse
f) Statistische Testdatenauswahl
g) Erfahrungsorientierte (intuitive) Testdatenauswahl.

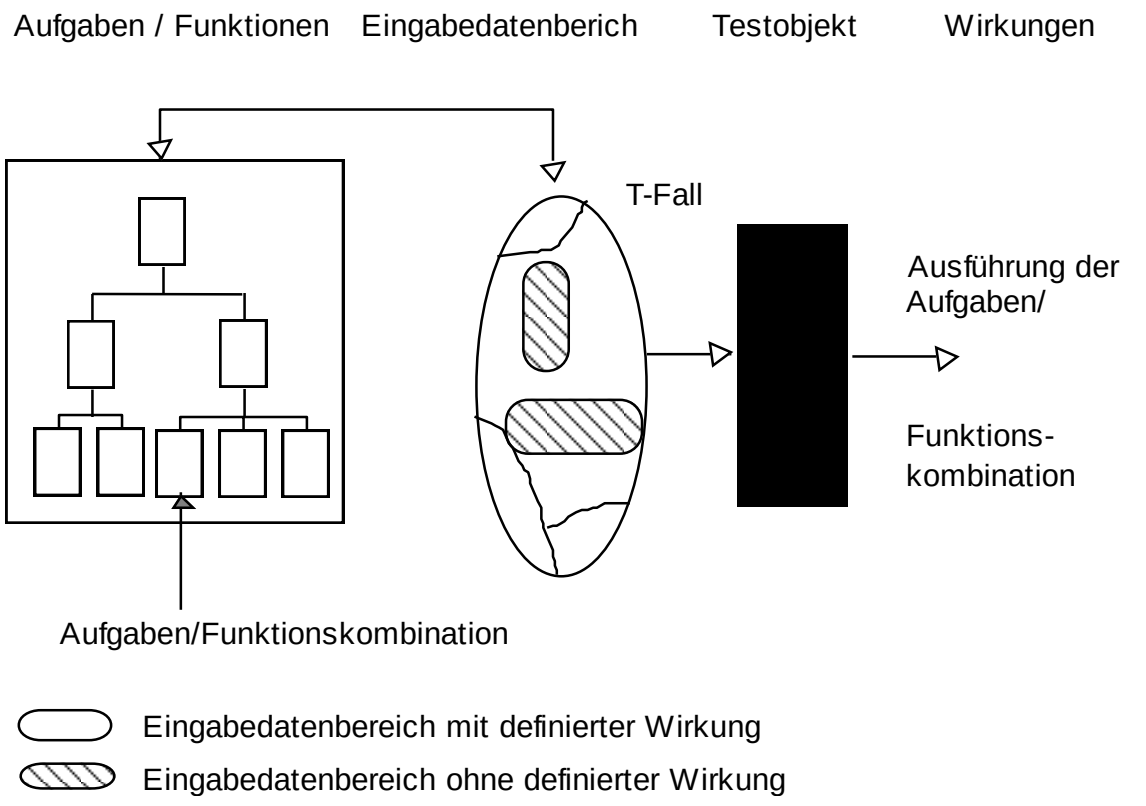


Abbildung 3-15: Aufgaben- / Funktionsorientierte Testfallbestimmung

3.10.1 Aufgabenorientierte Testfallbestimmung

Testfälle werden auf der Grundlage der Problembeschreibung und der Anforderungsanalyse festgelegt. Gehen wir davon aus, dass die durch das Testobjekt zu lösende Aufgabe hierarchisch in fachliche Teilaufgaben zerlegt ist (Baumdiagramm), so bestimmen wir Testfälle, dass diese Teilaufgaben gelöst werden.

Bei Bestimmung der Testfälle müssen wir zuvor prüfen, welche Definitionsbereiche für die von der konkreten Aufgabe zu verarbeitenden Informationen zulässig sind und diese festlegen.

Die Testfälle sind auf das spezifizierte Normalverhalten des Testobjekts ausgerichtet. Ausnahmen und Grenzfälle interessieren hierbei nicht.

Beispiel:

Testobjekt: Modul 'Kundenstammverwaltung'

Teilaufgaben Testfälle

	1	2	3	4	5	6
Kundenstamm initialisieren	X					
Kundenstamm aktualisieren		X	X		X	
Kundenstamm selektiv ausgeben			X	X	X	X
Kundenstamm löschen						
		X		X		

Black-Box Testfälle werden häufig in einer sog. **Testfallmatrix** dargestellt. Man sieht in diesem Beispiel, dass die Testfälle Nr. 4, 5 und 6 überflüssig sind und deshalb gelöscht werden können. Testen soll ja schließlich auch wirtschaftlich sein.

3.10.2 Funktionsorientierte Testfallbestimmung

Im Gegensatz zur aufgabenorientierten Testfallbestimmung, bei der vor allem fachliche, problembezogene Gesichtspunkte relevant sind, sind bei der funktionsorientierten Testfallbestimmung der DV-Entwurf, Schnittstellen und Programmspezifikationen inkl. Datenstrukturen die relevanten Aspekte. Testfälle sollen die in diesen Vorgaben beschriebenen fachlichen und DV-technischen Funktionen zur Ausführung bringen.

Auch hier wird davon ausgegangen, dass die Funktionen in einem Baumdiagramm hierarchisch aufgegliedert vorliegen (vgl. Abb. 3.5).

Bei einem systematischen Vorgehen wird man zunächst von den Verarbeitungsfunktionen ausgehen. Über Eingabe- und Ausgabefunktionen sowie der Interaktion der Funktionen wird man dann bis auf die Ebene der Programmfunktionen hinuntergehen.

Auch bei der funktionsorientierten Testfallbestimmung konzentriert man sich auf die sog. Normalfälle.

Hinweis

Sowohl beim aufgabenorientierten, als auch beim funktionsorientierten Testen kann man keine Fehler entdecken, die auf fehlerhafte oder unvollständige Anforderungs- und Problembeschreibungen zurückgehen.

Außerdem können Fehler, die auf im Testobjekt zusätzlich enthaltene Aufgaben und Funktionen zurückgehen, nicht entdeckt werden.

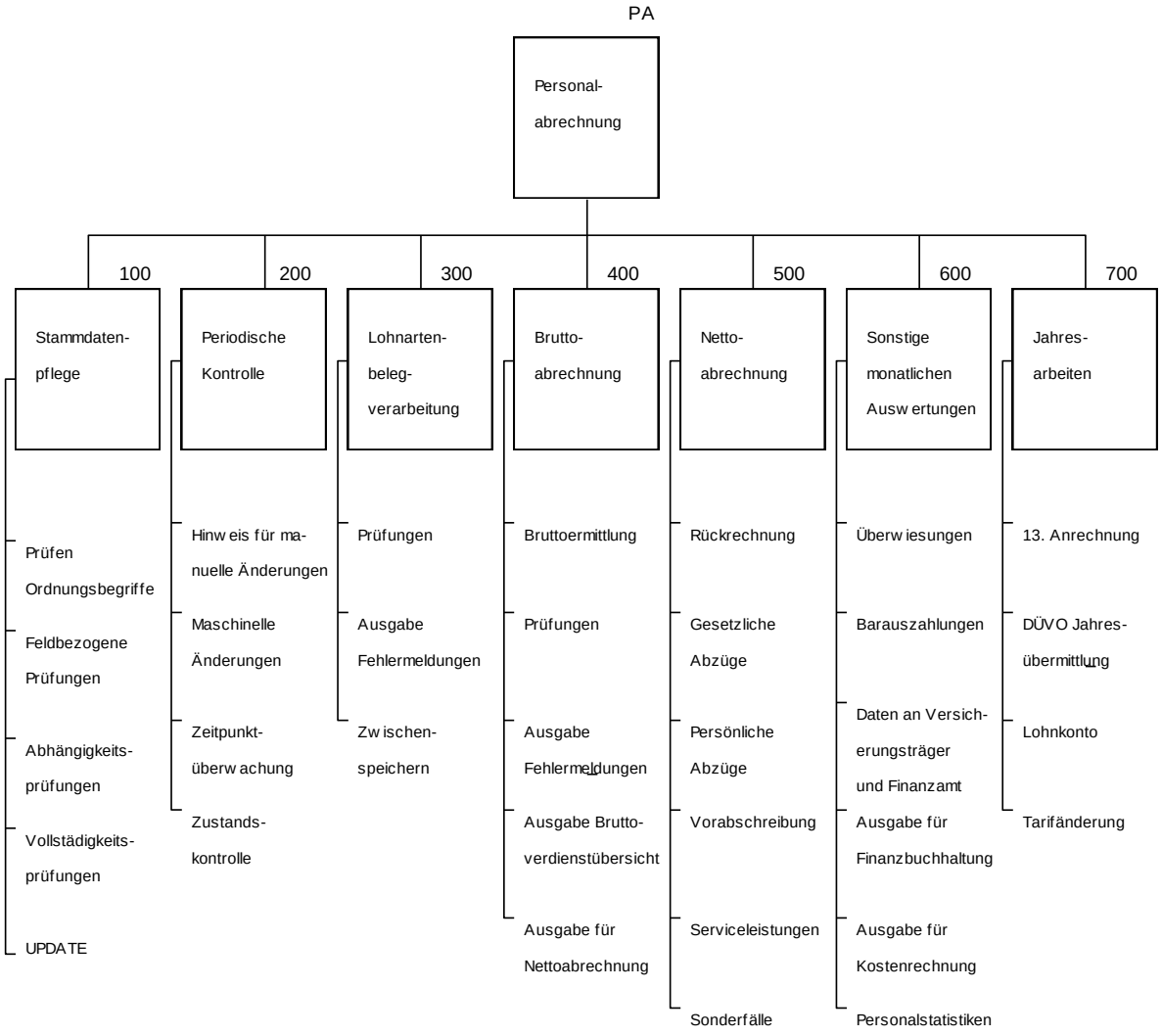


Abbildung 3-16: Funktionsorientierte hierarchische Gliederung der Aufgabe 'Personalabrechnung'

3.10.3 Äquivalenzklassenbildung

In ersten Abschnitt dieses Kapitels haben wir gesehen, dass der vollständige Eingabetest aufgrund unendlich vieler Testfälle unmöglich ist. Wir müssen also die Zahl der Testfälle dahingehend verringern, dass wir mit einem Testfall u. U. gleich eine ganze Klasse von Fehlern aufdecken (z. B. falsche Verarbeitung der Werte einer Eingabegröße vom Typ Zeichenkette).

Unter einer Äquivalenzklasse verstehen wir eine Menge von Werten gleicher Größe. Für alle Werte einer Äquivalenzklasse gilt, dass bei der Testausführung mit einem Repräsentanten dieser Klasse die gleiche Wirkung (entdeckte Fehlerart und -anzahl) auftritt, als wenn mit einem beliebigen anderen Wert dieser Klasse getestet wird.

Liegen die Werte im spezifizierten Definitionsbereich der Ein- oder Ausgaben, so handelt es sich um eine **gültige** Äquivalenzklasse; ansonsten um eine ungültige Äquivalenzklasse.

Beispiel: Tagesdatum

gültige Äquivalenzklassen	ungültige Äquivalenzklassen
1 <= T <= 31	T < 1 T > 31
1 <= M <= 12	M < 1, M > 12
J >= 1994	J < 1994

wobei T den Tag, M den Monat und J das Jahr bezeichnen mögen.

Zum Entwurf von Testfällen wird folgendermaßen verfahren:

- Einteilung der Ein- / Ausgabebereiche in Äquivalenzklassen
- Bestimmung eines Repräsentanten und Definition des Testfalls
- Prüfung der Äquivalenzklassenzerlegung auf Vollständigkeit und Wirksamkeit.

Eingabebedingung	gültige Äquivalenz-klassen	Ungültige Äquivalenz-klassen
Anzahl der Feld-deskriptoren	einer, >eins	keine
Größe des Feldnamens	1 - 6	0, >6
Feldname	mit Buchstaben mit Ziffern	sonstige Zeichen
Feldname beginnt mit Buchstaben	ja	nein
Anzahl Dimensionen	1 - 7	0, >7
Obere Grenze ist	Konstante INTEGER-Variable	Feldelementname sonstiges
INTEGER-Variablenname	hat Buchstaben hat Ziffern	hat sonst irgend etwas
INTEGER-Variable be-ginnt mit Buchstaben	ja	nein
Konstante	-65534 - 65534	<-65534, >65534
untere Grenze zu oberer Grenze	grösser als gleich	kleiner als
untere Grenze ist	eine Konstante INTEGER-Variable	Feldelementnamen irgend etwas sonst

Abbildung 3-17: Darstellung von Äquivalenzklassen

3.10.4 Grenzwertanalyse

Aus der Bildung der gültigen und ungültigen Äquivalenzklassen ergeben sich automatisch Grenzwerte als die Ränder der Äquivalenzklassen. Jeder Rand einer Äquivalenzklasse (sowohl des Eingaberaumes wie des Ausgaberaumes) muss in einem Testfall auftreten. Auch alle Ausgabe-Äquivalenzklassen müssen durch entsprechende repräsentative Eingabewerte berücksichtigt werden.

Der Hintergrund dieser Methode besteht ganz einfach in der Erfahrung, dass bei Ausnahme- und Grenzwerten die Wahrscheinlichkeit eines Fehlers sehr viel grösser ist, als bei den sog. Normalwerten.

Aus dem vorhergehenden Beispiel zur Äquivalenzklassenbildung ergeben sich folgende Grenzwerte und Testfälle:

Grenzwerte	**Testfälle**
T = 1, T = 31	T = 1, T = 0 T = 31, T = 32
M = 1, M = 12	M = 1, M = 0 M = 12, M = 13
J = 1994	J = 1994, J = 1995

Wir können zur Grenzwertanalyse folgende Richtlinien aufstellen:

1. Definiere Testfälle für die Grenzen und für **ungültige** Eingabewerte direkt neben den Grenzen (z. B. gültig -1.0 bis +1.0; dann Testfälle für -1.0, +1.0, -1.001 und +1.001, falls der Definitionsbereich entsprechend aussieht)
2. Ist ein Wertebereich vorgegeben, so definiere Testfälle für den oberen und unteren Grenzwert und für Werte direkt neben den Grenzwerten außerhalb des Bereichs (z. B. Datei kann 1 - 255 Sätze enthalten, dann Testfälle für 0, 1, 255, 256 Sätze)
3. Richtlinien 1 und 2 gelten auch für Ausgabebedingungen (z. B. monatliche Sozialabgabe liegt zwischen DM 0.00 und DM 2.200, dann Testfälle für Abzug von DM 0.00 und DM 2.200 und für negativen Abzug und Abzug grösser DM 2.200)
4. Wenn Ein-/Ausgabebereich eine geordnete Menge ist (sequentielle Datei, Tabelle), so prüfe erstes, letztes und kein Element.

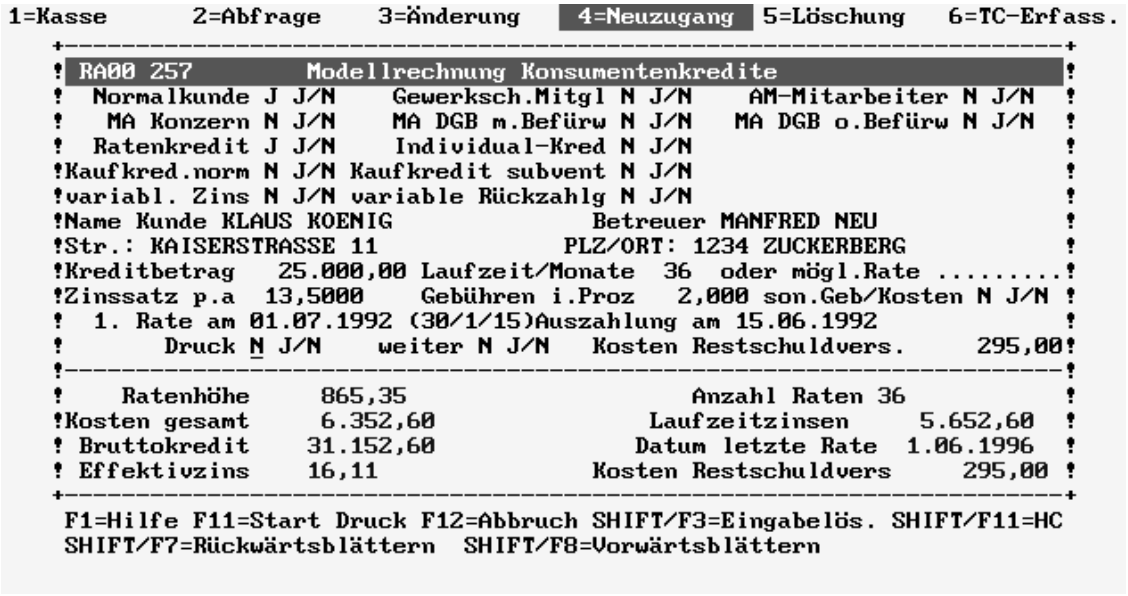
3.10.5 Eingabetest

Bei dieser Testmethode geht es darum,

① den Definitionsbereich von Eingabedaten,
② die richtige Syntax der Eingabedaten,
③ die Fehlerhinweise des Programms,
④ Handling der Masken (Tab-, Return-, Esc-Taste),
⑤ interne Plausibilitätsprüfungen der Anwendung,
⑥ die Help-Funktion und weitere PF-Tasten sowie
⑦ die Vorbelegungen in einer Maske

auf Herz und Nieren zu testen.

Dieser Test zielt auf die Qualitätsmerkmale ´Robustheit´ und ´Benutzerfreundlichkeit´ ab. Er betrifft vor allem Eingabemasken in Anwendungssystemen.



Was wäre in dieser Beispielmaske zu überprüfen?

Auch Berechnungen eines Programms wie im obigen Beispiel sollten ruhig einmal stichprobenartig mit dem Taschenrechner überprüft werden.

Nach dem heutigen Stand der Technik sind folgende Mindestanforderungen bzgl. Robustheit und Benutzer Freundlichkeit an ein Dialogsystem zu stellen:

⇒ keine Benutzereingabe darf ein Anwendungssystem zum Absturz bringen

⇒ bei formal ungültigen Eingaben sind vom Programm aussagekräftige Fehlerhinweise zu liefern, die dem Anwender weiterhelfen

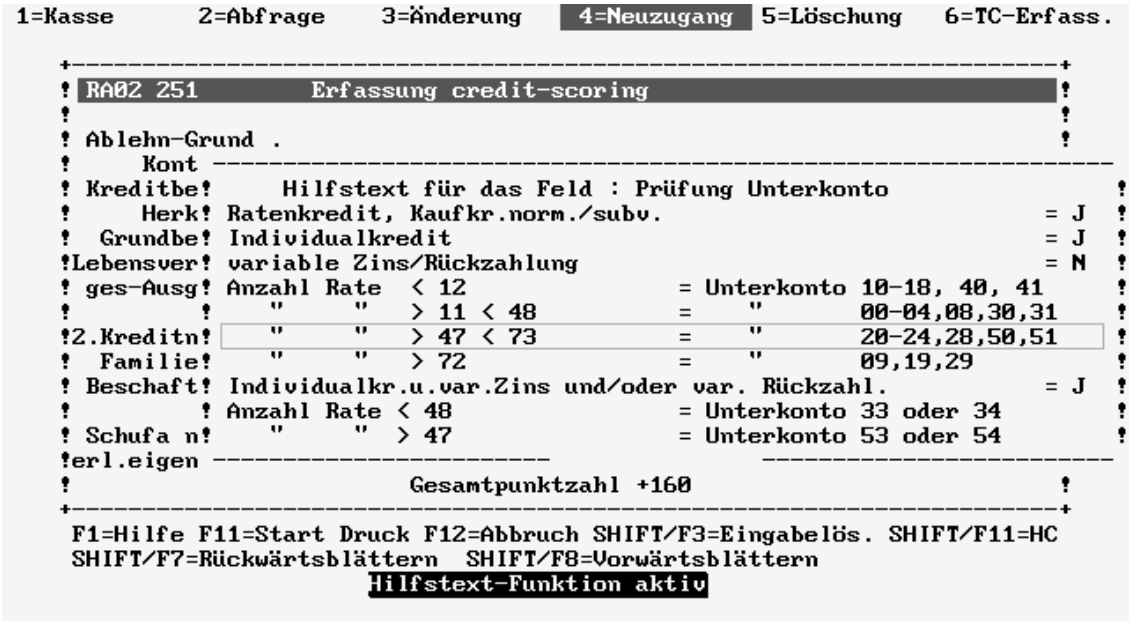
⇒ zu allen Feldern einer Maske sollte möglichst eine Help-Funktion oder Prompt-Funktion (Schlüsselfelder) zur Verfügung stehen

⇒ durch Plausibilitätsprüfungen sind vom Programm widersprüchliche oder inkonsistente Eingaben mit entsprechenden Hinweisen abzufangen (z. B. Privat-KZ und KK-Betriebsnummer widersprüchlich)

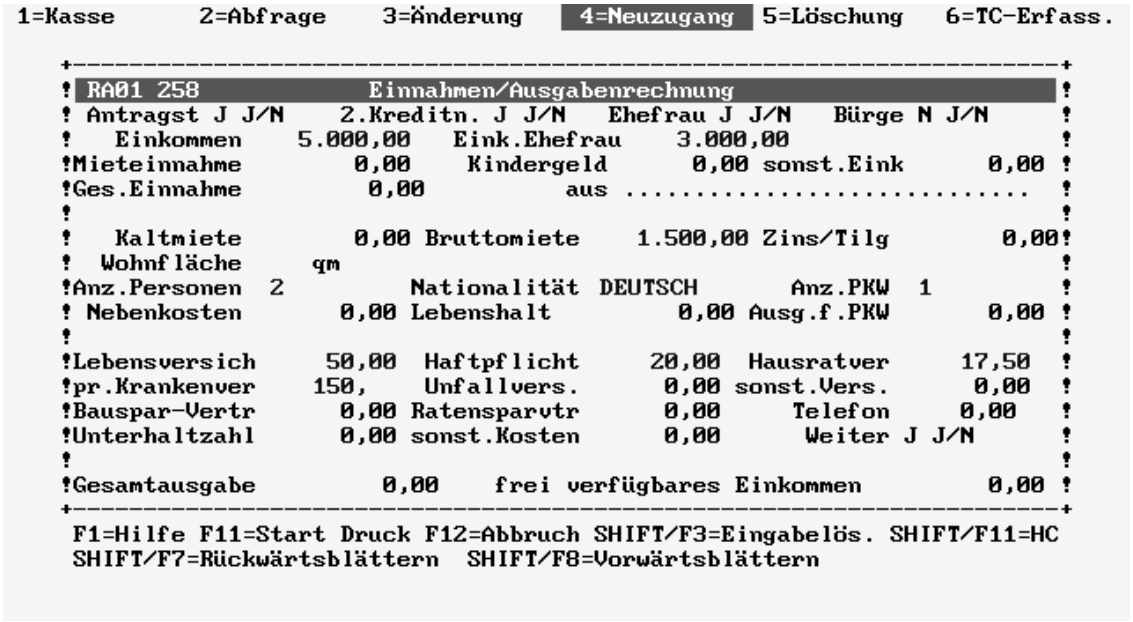
⇒ bis zur Freigabe einer Maske müssen alle Benutzereingaben mit entsprechenden Editierfunktionen wieder korrigiert werden können

⇒ Maskenbezeichnungen und insbesondere Feldbezeichnungen müssen in einem Anwendungssystem konsistent sein

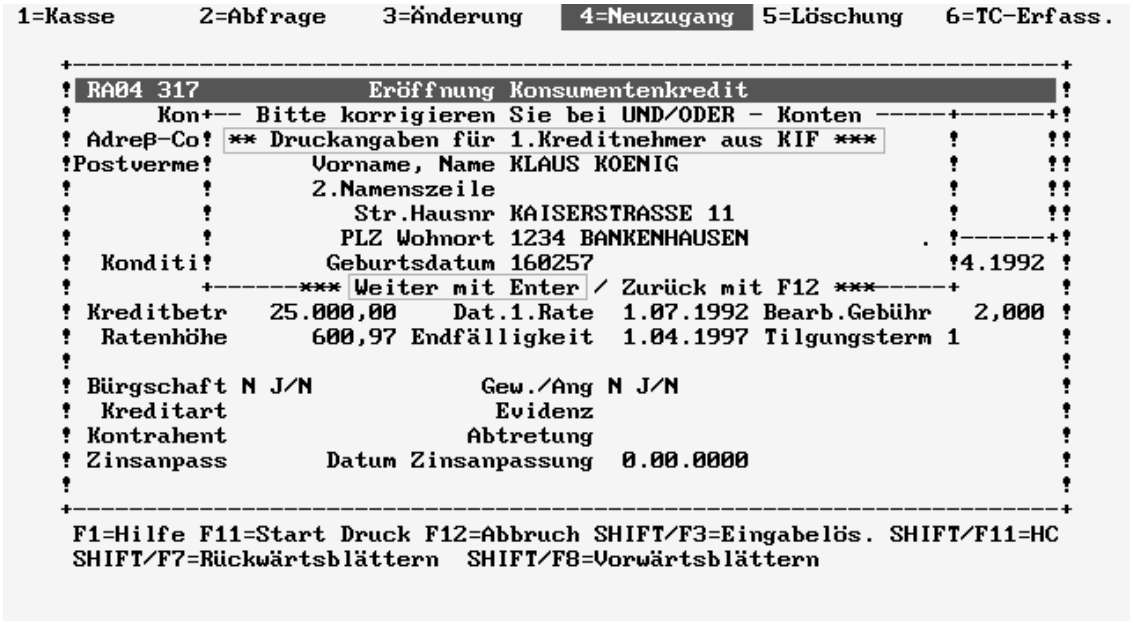
⇒ Masken müssen richtig initialisiert (vorbelegt) sein



Funktioniert die Help-Funktion wie vorgesehen?



Sind diese Eingaben plausibel?



Kann man hier wirklich mit <F12> zurück?

3.10.6 Statistische und intuitive Testdatenauswahl

Der Sinn dieser Methode ist umstritten. Denn jeder Test muss ja nachvollziehbar und wiederholbar sein. Dies ist beim statistischen Testen nicht immer gewährleistet.

Statistische Tests geben Wahrscheinlichkeitsaussagen wieder, z. B. über erwartete

- Restfehlerraten,
- Zuverlässigkeit und
- Risiko

des Testobjekts.

Das Ziel solcher Tests ist u. a.

- Die Wahrscheinlichkeit eines Software-Ausfalls soll unter eine gewisse Grenze mit einem Vertrauensniveau von P % liegen.
- Die Zahl der Fehler, die noch vorliegen (Restfehlerrate), wird unterhalb eines bestimmten Wertes erwartet.
- Die wegen eines Software-Fehlers erwarteten Kosten sollen unterhalb eines gegebenen Wertes mit einem Vertrauensniveau von P % liegen.

Man sieht an diesen Beispielen schon, dass solche Tests für sicherheitsorientierte, äußerst kritische Software relevant sind.

Auch bei Realzeit-Systemen sind diese Tests notwendig als Ergänzung zu bereits erfolgten systematischen Tests. Man rechnet bei Realzeit-Systemen damit, dass nur 4 bis 10 Prozent der gesamten Software 50 bis 90 Prozent der gesamten Laufzeit des Systems beanspruchen. Stresstests sollten sich daher auf diesen begrenzten, aber dominierenden Teil konzentrieren.

Test aufgrund von Erfahrungen

Dies ist ein auf die Wahrnehmungsfähigkeit des Menschen bzw. auf heuristisches Vorgehen ausgerichteter Ansatz. Wir haben es daher nicht mit einer Methode im strengen Sinne zu tun, aber mit einem in der Praxis sehr wirkungsvollen Verfahren.

Es gibt Personen, meist sehr erfahrene Programmierer oder Systemanalytiker, die durch bloße Intuition Fehler aufspüren können.

Meist trifft dies auf Fehlerkategorien zu, die immer wieder auftreten (falsche Schleifeninitialisierung, falscher Gruppenwechsel bei Verarbeitung mehrerer Dateien, Rechnen mit Werten unterschiedlichen Typs etc.).

Dieser Ansatz eignet sich sehr gut dazu, systematisch mit den vorgestellten Methoden erzeugte Testfälle, qualitativ noch zu verbessern.

Zusammenfassend möchten wir eine pragmatische Strategie zu den Testmethoden vorschlagen, die insbesondere für den Modultest gilt:

1. Überprüfen Sie die Spezifikation mit der Ursache-Wirkung-Analyse und Entscheidungstabellen.
2. Stellen Sie die gültigen und ungültigen Äquivalenzklassen auf.
3. Führen Sie die Grenzwertanalyse durch.
4. Ergänzen Sie Ihre Testmenge durch intuitive Testfälle.
5. Vervollständigen Sie Ihre Testmenge durch Testfälle für den White-Box Test.

3.11 Erstellung der konkreten Testfälle

3.11.1 Definition „Konkreter Testfall“

Die in der Testphase „Analyse und Design“ erstellten logischen Testfälle müssen in der Testphase „Realisierung und Durchführung“ ergänzt werden um:

- Tatsächliche, konkrete Werte für die Testdaten
- Ggf. Detailbeschreibung der durchzuführenden Prüfungen
- Zusammenstellung der Testfälle zu Testsequenzen und Testszenarien mit Beschreibung der Vorbedingungen und Nachbedingungen.

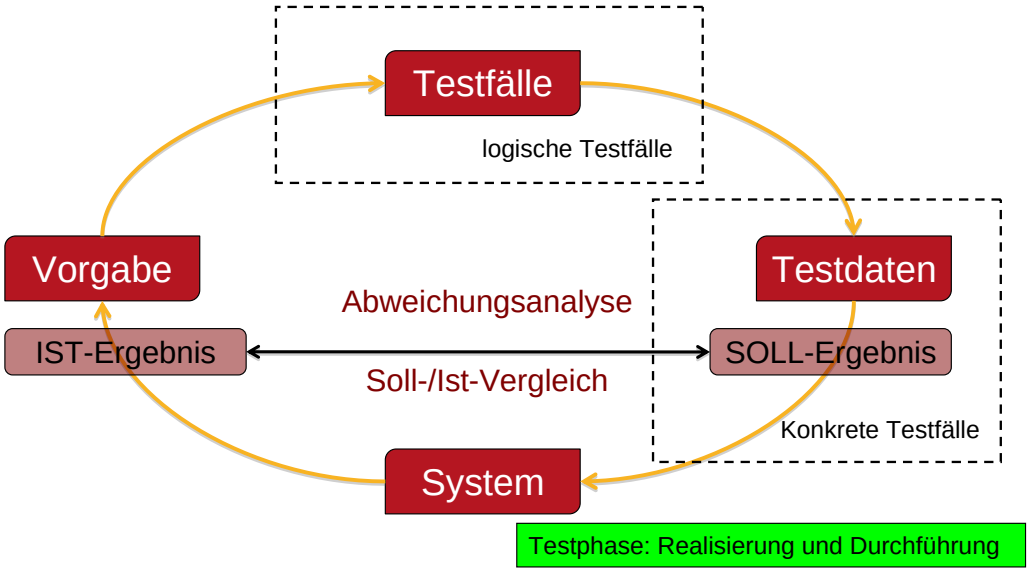


Abbildung 3-18: Übersicht Testaktivitäten – Erstellung konkreter Testfälle

3.11.2 Testdaten

3.11.2.1 Unterscheidung: Primär- und Sekundärtestdaten

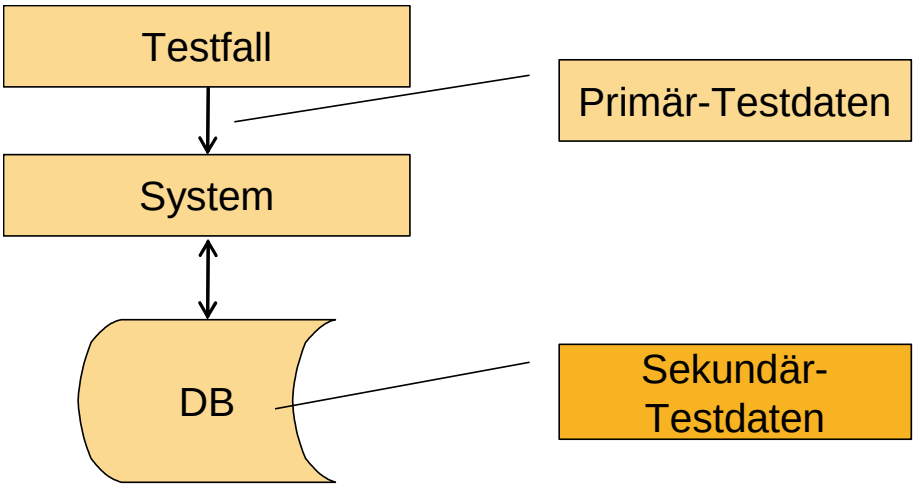


Abbildung 3-19: Unterscheidung Primär- und Sekundärtestdaten

Primärtestdaten:

Dies sind die Daten, die bei der Durchführung eines Testfalls eingegeben werden. Z. B.:

- Eingabe der Daten durch den Tester über eine Maske
- Einlesen einer Schnittstellen-Datei

Sekundärtestdaten:

Das sind die Daten, die als Voraussetzung für den Testfall in der entsprechenden Datenbasis (Datei, Datenbank etc.) vorhanden sein müssen.

Grundsätzlich gilt:

Sowohl die Primär- als auch die Sekundärtestdaten müssen

→ pro Testfall

→ auf Basis der Spezifikation

→ vor Ausführung der Tests

definiert werden.

3.11.2.2 Arten von Testdaten

Abstrakte Testdaten:

Die Daten entsprechen nicht der Realität. Sie werden ausschließlich für die Optimierung der Testdurchführung benötigt (→ Soll-/Ist-Vergleich)

Synthetische Testdaten:

Synthetische Testdaten werden auf Basis der definierten Testfälle mit dem Ziel entwickelt, alle im System vorhandenen Möglichkeiten von Bedingungskombinationen zu testen und so einen hohen Testabdeckungsgrad zu erreichen.

Echtdaten:

Echtdaten beziehen sich auf echte Geschäftsvorfälle. Sie sollten so aufgebaut sein, dass jeder Testfall auf ein bestimmtes Ziel ausgerichtet ist. Um dies zu erreichen muss die Anzahl der Testfälle überschaubar bleiben.

Sind in der Regel ein Abzug der Produktionsdaten (komplett oder Auswahl).

Problematik: Anonymisierung

Massendaten:

Massendaten als Testdaten beziehen sich auf echte Geschäftsprozesse mit dem Ziel die Gesamtzusammenhänge der unterschiedlichen Systeme nahezu unter Echtzeitbedingungen zu testen.

Massendaten werden insbesondere für Last- und Performance-Tests eingesetzt.

3.11.2.3 Problematik: Erstellung der Sekundärtestdaten

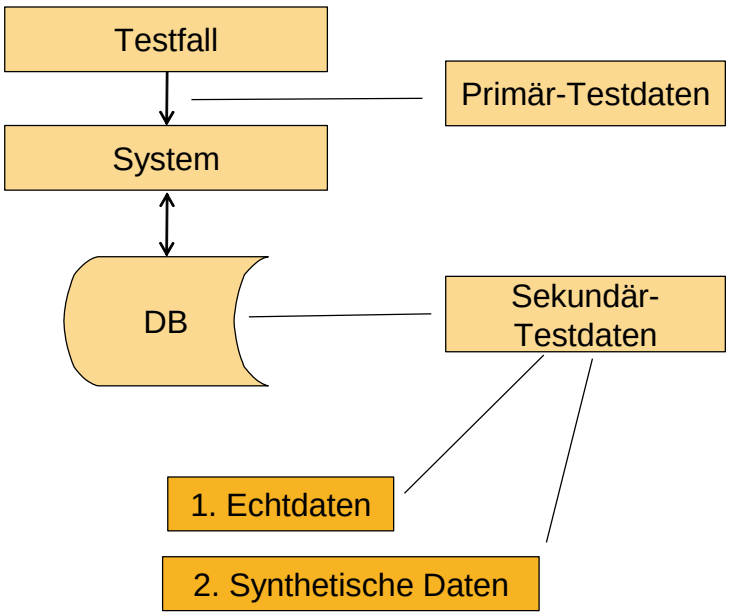


Abbildung 3-20: Erstellung der Sekundärtestdaten

In der Praxis ist die Definition und die Bereitstellung der Sekundärtestdaten häufig sehr problematisch und meist mit einem sehr hohen Aufwand verbunden.

Grundsätzlich werden die Sekundärtestdaten entweder synthetisch erstellt oder es wird mit Echtdaten gearbeitet.

Nachfolgend einige Probleme, die sich bei der Verwendung von **synthetischen Daten** ergeben können:

- Sehr hoher Erstellungsaufwand
- Korrektheit der Daten
- Alterung
- Abbildung der echten Geschäftsvorfälle.

Nachfolgend einige Probleme, die sich bei der Verwendung von **Echtdaten** ergeben können:

- Anpassungsaufwand
- Ggf. muss Testfall angepasst werden
- Datenmenge
- Selektion der benötigten Daten
- Wiederholbarkeit.

3.11.3 Äquivalenzklassenbildung und Grenzwertanalyse

3.11.3.1 Problemstellung

Bei dem Testziel

„Jeder Dateninhalt soll auf Gültigkeit geprüft werden“

kann die Anzahl der möglichen Konstellationen sehr schnell sehr groß werden.

Beispiel:

Auf einer Maske befindet sich das Eingabefeld „Familienname“.

Es ist folgendermaßen definiert:

- Länge: 30 Stellen
- Format: Alphanumerisch
- Zulässige Zeichen:
  - a – z
  - A – Z
  - 0 – 9
  - Sonderzeichen: „+“, „-“, „&“
- Alle anderen Sonderzeichen sind nicht zulässig.

Bei Eingabe eines nicht zulässigen Zeichens soll eine Fehlermeldung erscheinen.

Um hier alle möglichen Kombinationen auszutesten, würde die Anzahl der Testfälle nahe unendlich gehen.

In der Praxis wird man 3 Testfälle entwickeln:

- Eingabe nur zulässige Zeichen
- Eingabe nur unzulässige Zeichen
- Eingabe Kombination zulässige und unzulässige Zeichen.

Das Ziel der Äquivalenzklassenbildung und Grenzwertanalyse ist es, bei einer sehr hohen Anzahl an Kombinationsmöglichkeiten mit wenigen Testfällen dennoch eine möglichst hohe Qualität zu erreichen.

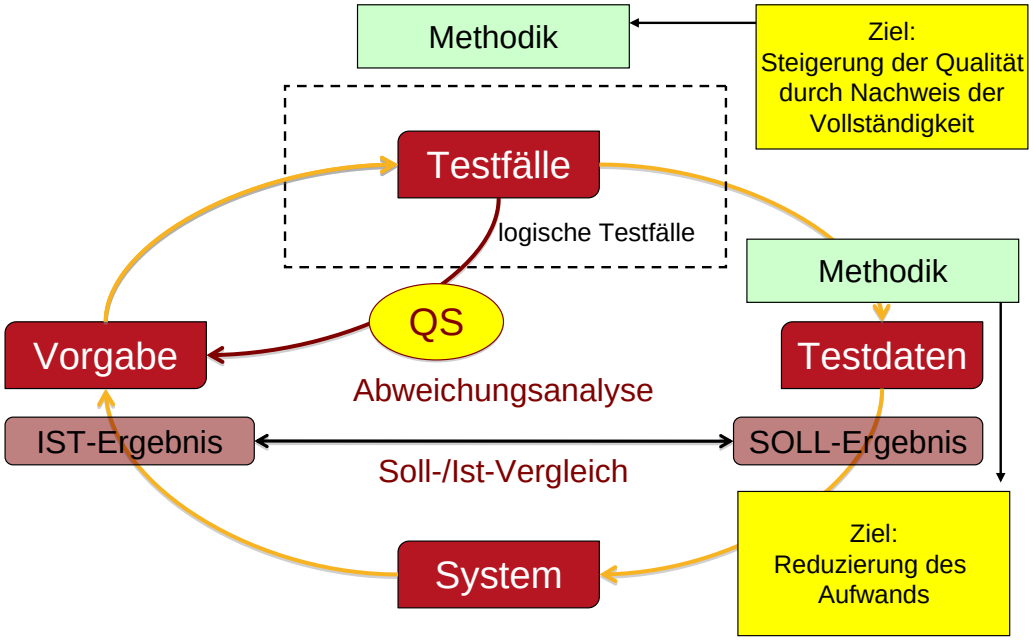


Abbildung 3-21: Übersicht Testaktivitäten – Ziel Einsatz Methodik

Beispiel: Abfrage Alter

- Alle Personen, deren Alter größer 20 und kleiner oder gleich 30 ist, erhalten beim Einkauf 20 % Rabatt, alle anderen Personen erhalten 10 %.
- Mit welchen Daten (Alter einer Person) sollte dieser Sachverhalt getestet werden?

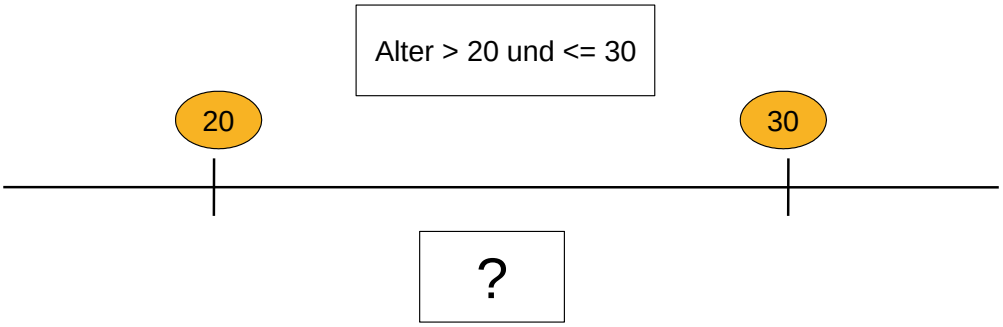


Abbildung 3-22: Beispiel Äquivalenzklassenbildung (1)

3.11.3.2 Äquivalenzklassenbildung

Die Menge der möglichen Eingabewerte für ein Eingabedatum wird in Äquivalenzklassen unterteilt. Zu einer Äquivalenzklasse gehören alle Eingabedaten, bei denen der Tester davon ausgeht, dass sich das Testobjekt bei Eingabe eines beliebigen Datums aus der Äquivalenzklasse gleich verhält. Der Test eines Repräsentanten einer Äquivalenzklasse wird als ausreichend angesehen, da davon ausgegangen wird, dass das Testobjekt für alle anderen Eingabewerte derselben Äquivalenzklasse keine andere Reaktion zeigt.

Neben den Äquivalenzklassen, die gültige Eingaben umfassen, sind auch solche für ungültige Eingaben zu berücksichtigen.

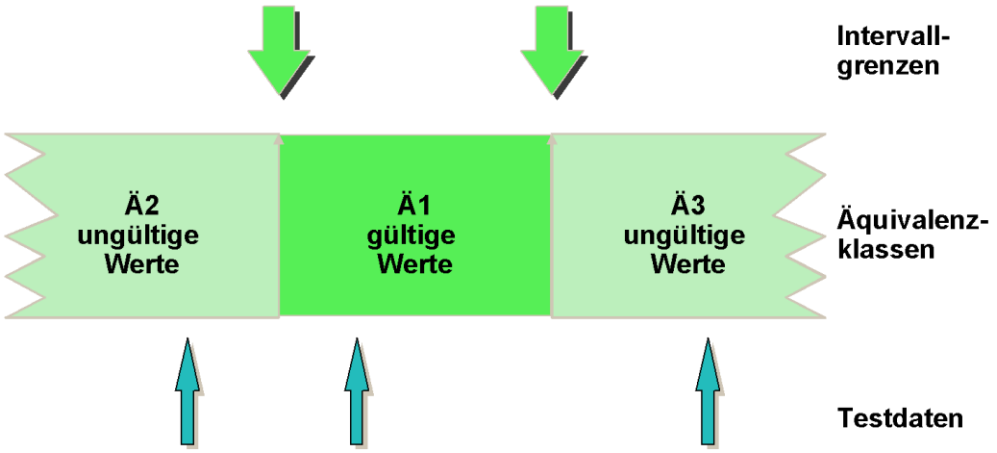


Abbildung 3-23: Äquivalenzklassenbildung

Vorgehensweise:

Um die Testfälle systematisch abzuleiten, wird für jede zu testende Eingabevariable der Definitionsbereich ermittelt. Der Definitionsbereich ist die Äquivalenzklasse aller zulässigen Eingabewerte. Diese Werte muss das Testobjekt gemäß der Spezifikation verarbeiten. Die Werte außerhalb des Definitionsbereichs werden als Äquivalenzklasse mit unzulässigen Werten betrachtet. Auch für diese Werte ist zu prüfen, wie sich das Testobjekt verhält.

Dies ergibt folgende Testdaten für das Beispiel „Alter“:

- Alle Personen, deren Alter größer 20 und kleiner oder gleich 30 ist, erhalten beim Einkauf 20 % Rabatt, alle anderen Personen erhalten 10 %.
- Mit welchen Daten (Alter einer Person) sollte dieser Sachverhalt getestet werden?

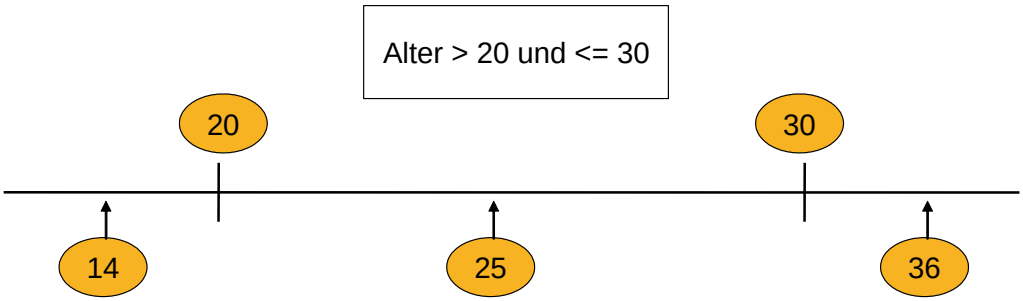


Abbildung 3-24: Beispiel Äquivalenzklassenbildung (2)

3.11.3.3 Grenzwertanalyse

Da an den Grenzen zulässiger Datenbereiche erfahrungsgemäß häufig Fehler auftreten, sollte als **Ergänzung zur Äquivalenzklassenanalyse** für jeden zu testenden Parameter eine **Grenzwertanalyse** durchgeführt werden.

Bei der **Grenzwertanalyse** werden die Ränder der Äquivalenzklassen einer Überprüfung unterzogen. An jedem Rand werden der exakte Grenzwert und die beiden benachbarten Werte innerhalb und außerhalb der Äquivalenzklasse getestet.

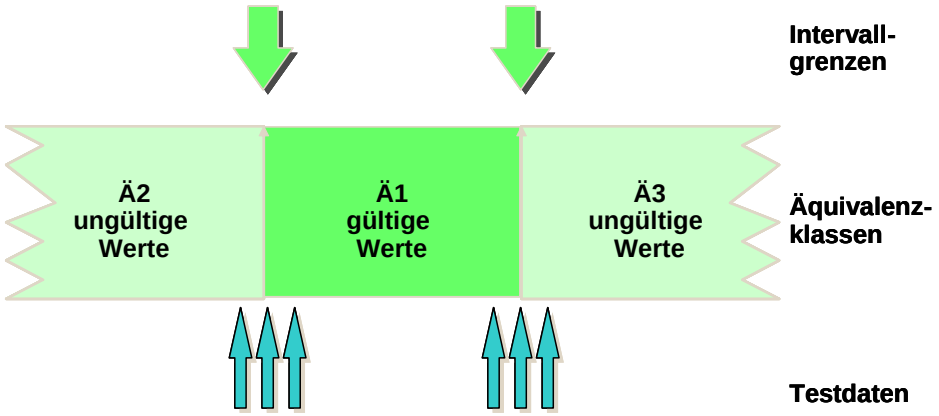


Abbildung 3-25: Grenzwertanalyse

Damit ergeben sich für das Beispiel „Alter“ folgende Testfälle:

- Alle Personen, deren Alter größer 20 und kleiner oder gleich 30 ist, erhalten beim Einkauf 20 % Rabatt, alle anderen Personen erhalten 10 %.
- Mit welchen Daten (Alter einer Person) sollte dieser Sachverhalt getestet werden?

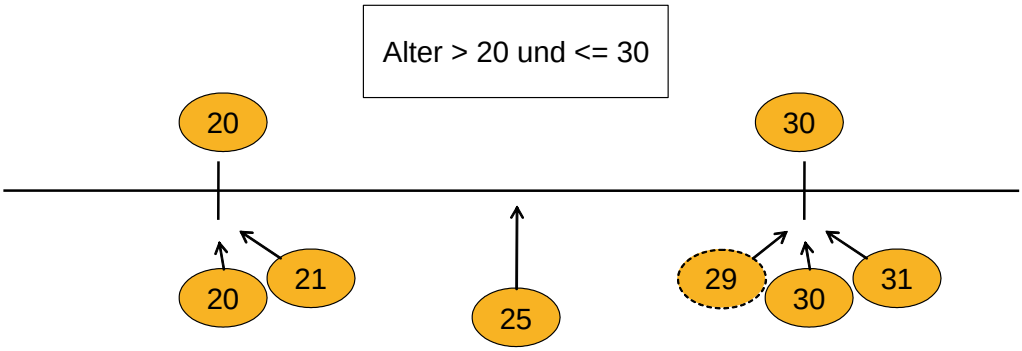


Abbildung 3-26: Beispiel Äquivalenzklassenbildung und Grenzwertanalyse

Methodik: Visualisierung

Ziele der Visualisierung:

- Testdaten werden so erstellt, dass die Korrektheit des Testfalls sehr schnell zu erkennen ist → Optimierung Soll-/Ist-Vergleich
- Kein meist langwieriges Nachrechnen oder Kontrollieren des Ist-Ergebnisses erforderlich
- Ggf. kann sogar aufgrund des Testergebnisses auf die Fehlerursache geschlossen werden
- Verwendung von abstrakten Testdaten hilfreich → „sprechende Testdaten

Beispiel:

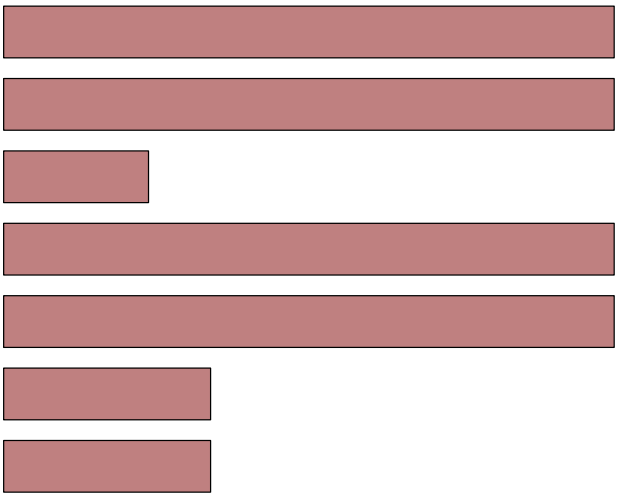
Bei der Eingabe von Daten auf einer Maske soll folgendes getestet werden:

- Kann pro Eingabefeld die richtige, im Konzept definierte Stellenanzahl eingegeben werden?
- Werden die Daten in der richtigen Länge abgespeichert?
- Werden die Daten wieder in der richtigen Länge auf der Maske angezeigt?
- Wird der Feldinhalt im richtigen Feld angezeigt?

Beispiel Namenskonvention:

Mit welchen Werten werden die einzelnen Felder gefüllt?

- NAME
- Vorname
- Postleitzahl
- Ort
- Straße
- Beitrag
- Summe



Durch die Eingabe der unten stehenden Daten (abstrakte Testdaten) ist es möglich, auf „den ersten Blick“ Abweichungen zu erkennen.

NAME	A.....Name.....A
Vorname	B.....Vorname.....B
Postleitzahl	10001
Ort	C.....Ort.....C
Straße	D.....Straße.....D
Beitrag	20000,02
Summe	30000,03

3.11.4 Testsequenzen und Testszenarien

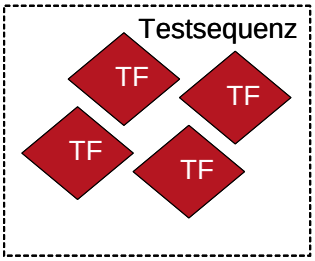
3.11.4.1 Definition

Nachdem die konkreten Testfälle beschrieben sind, sollte vor der endgültigen Durchführung der Testfälle noch der genaue Ablauf festgelegt werden.

Hierzu werden Testfälle zu Testsequenzen zusammengefasst und ggf. wiederum mehrere Testsequenzen zu Testszenarien.

Testseqzenz

- Zusammenlegung von mehreren Testfällen
- Nachbedingung des einen Tests ist eine mögliche Vorbedingung des Folgetests
- Ziel ist die effektivere Durchführung und die Schaffung einer übersichtliche Struktur



Testszenario

- Legt die ausführbare Reihenfolge der Aktionen (d. h. den Ablauf der Testfälle) fest
- Können auch in einem Testskript (automatisiertes Testszenario) festgehalten werden

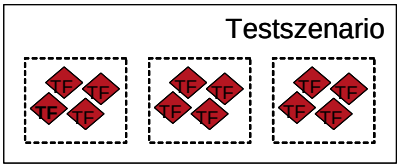


Abbildung 3-27: Testsequenzen und Testszenarien

3.11.4.2 Wiederholbarkeit und Automatisierbarkeit

Bei der Zusammenfassung von Testfällen zu Testsequenzen steht die Frage im Mittelpunkt:

Sollen die Testfälle/Testszenarien wiederholbar sein?

Ein Testfall ist erst dann wiederholbar, wenn er beliebig oft, zu beliebigen Zeitpunkt ausgeführt werden kann und immer das **selbe Ergebnis** bringen soll.

Dies bedeutet im Umkehrschluss:

Ändert sich das Ergebnis des Testfalls (z. B. Ergebnis einer Berechnung), weil sich das Datum verändert hat (Ausführung an verschiedenen Tagen), so kann der Testfall zunächst nicht als wiederholbar bezeichnet werden.

Es müsste wieder (unter Umständen kein geringer) Aufwand investiert werden, um nach dem Soll-/Ist-Vergleich im Rahmen der Abweichungsanalyse zu prüfen, ob das Ist-Ergebnis vom Soll-Ergebnis abweicht, weil

- sich die Voraussetzungen geändert haben (z. B. anderes Datum) und damit zwangsläufig ein anderes Ist-Ergebnis erscheinen muss. Dieses muss wiederum geprüft werden (nachgerechnet), ob es richtig ist.

oder

- ob tatsächlich ein Fehler vorliegt.

Die Folge ist, dass wieder ein nicht unerheblicher Aufwand in der Testdurchführung geleistet werden muss.

Eine Grundvoraussetzung für die **Automatisierbarkeit** von Testfällen/Testsequenzen ist die Wiederholbarkeit. Insbesondere der Vorteil des „maschinellen“ Soll-/Ist-Vergleichs geht verloren, wenn immer eine „manuelle“ Prüfung bei Abweichungen erfolgen muss.

Um das Ziel „Wiederholbarkeit“ zu erreichen, müssen äußere Einflüsse, die den Ablauf oder das Ergebnis des Testfalls verändert können, ausgeschaltet werden.

Dies könnte beispielsweise sein:

- Berechnung erfolgt nicht mit dem Systemdatum, sondern mit einem fest in der Datenbank hinterlegten „synthetischen“ Systemdatum
- Daten, die von außen über eine Schnittstelle in das zu testende System gelangen, kommen nicht von dem Fremdsystem, sondern werden an der Schnittstelle bereitgestellt (simuliert).

Sehr oft scheitert die Wiederholbarkeit an den Sekundärtestdaten:

- Datenveränderungen können nicht rückgängig gemacht werden
- Ein Neuladen der Datenbank mit dem Ausgangsdatenbestand ist nicht oder nur mit sehr viel Aufwand / Zeit verbunden.

Ein weiterer Gesichtspunkt beim Aufbau wiederholbarer und insbesondere automatisierbarer Testsequenzen ist die Abfolge der Testsequenzen. Hier gibt es grundsätzlich 2 Möglichkeiten:

- Die Testsequenzen bauen aufeinander auf. Dies bedeutet: Die Ergebnisse und Nachbedingungen einer Testsequenz sind die Voraussetzung und Eingangsbedingung für die nächste Testsequenz.
- Alle Testsequenzen können unabhängig von einander gestartet werden. Jede Testsequenz startet von einem definierten Ausgangspunkt mit festgelegten Ausgangszuständen und muss demnach am Ende den Ausgangszustand wieder herstellen bzw. zum Ausgangspunkt zurückkehren.

Diese Überlegungen gelten nicht nur für die Testsequenzen, sondern sollten auch bei der Testfallbeschreibung beachtet werden.

Die nachfolgende Testsequenzbeschreibung ist so aufgebaut, dass die einzelnen Testsequenzen unabhängig von einander durchgeführt werden können.

3.11.4.3 Testszenarienbeschreibung

Unter dem Gesichtspunkt der Wiederholbarkeit sollte die Testszenarienbeschreibung folgende Inhalte haben:

- Verwaltungsinformationen; z. B.:
  - Eindeutiger Name
  - ggf. Ziel der Testsequenz
  - Erfasser / Bearbeiter
  - Datum der Erfassung, Durchführung
  - ggf. Verweis auf Spezifikation
  - Besonderheiten.
- Ausgangszustand; z. B.:
  - angemeldeter Benutzer
  - Berechtigungen
  - Ausgangsdatenbestand
- Navigation zum Testfall.

Nachdem jede Testsequenz von einem gemeinsamen Ausgangspunkt startet, muss beschrieben werden, wie der Zustand erreicht werden kann, an dem die Testfälle starten.

Dies könnte in tabellarischer Form erfolgen:

Aktion	**Ergebnis**
Auswahl in Menüleiste „Suchen“	Suchmaske erscheint
Eingabe Suchbegriff und Klick auf Button Suchen	Trefferliste erscheint
1. Namen selektieren	Bearbeitungsmaske erscheint Ausgangspunkt ist erreicht

- Beschreibung der Testfälle
  - Ablauf des Testfalls (inkl. der Daten)
  - Ergebnis des Testfalls
  - durchzuführende Prüfungen
  - ggf. durchzuführende Maßnahmen, damit der nächste Testfall innerhalb der Testsequenz gestartet werden kann.
- Ausgangszustand wieder herstellen

  Damit die nächste Testsequenz starten kann, muss wieder zum Ausgangspunkt zurückgekehrt werden und der Ausgangszustand wieder hergestellt werden.

4

Testen objektorientierter Programme

4 Testen objektorientierter Programme

4.1 Beispiel

Imperative Kommandofolge

Beginn:

lösche den Bildschirm;
setze den Cursor auf Position (10, 2);
gib den Text aus: ´Bitte Zahlen (z1, z2) eingeben!´
warte auf eine Eingabe vom Benutzer;
Übernehme zwei Zahlen (z1, z2);
transformiere (z1, z2) in die duale Codierung (z1`, z2`);
addiere (z1`, z2`) und weise die Summe der Variablen E` zu;
setze den Cursor auf Position (10, 4);
transformiere die duale Variable E` in die ASCII-Codierung E;
stelle das Ergebnis ´E´ dar;

Ende;

Objektorientierten Ablauffolge

Beginn:

Bildschirm, lösche deinen Inhalt;
Cursor, setze dich auf Position (10, 2);
Bildschirm, gib den Text aus: ´Benutzer gib bitte zwei Zahlen ein!´;
Tastatur, übernehme eine ASCII-codierte Zahl in die Variable (z1);
Tastatur, übernehme eine ASCII-codierte Zahl in die Variable (z2);
Transformierer, setze (z1) in die duale Zahl (z1`) um;
Transformierer, setze (z2) in die duale Zahl (z2`) um;
Addierer, bilde die Summe aus den dualen Summanden (z1`, z2`);
Addierer, weise die Summe der dualen Variablen (E`) zu;
Transformierer, setze (E`) in eine ASCII-Zahl (E) um;
Cursor, setze dich auf Position (10, 4);
Bildschirm, stelle die ASCII-Zahl (E) dar;

Ende;

4.2 Objektorientierung

Das Eingangsbeispiel macht deutlich, dass wir es bei objektorientierten Programmen nicht mit Befehlen an den Computer zu tun haben, die dieser ausführen soll. An die Stelle dieser Befehle sind sog. *Botschaften* getreten, die an ein bestimmtes *Objekt* (Bildschirm, Cursor, Addierer...) gerichtet sind. Dadurch wir das Objekt veranlasst, eine bestimmte Operation auszuführen. Auf die benötigten Operanden hat die Operation entweder impliziten Zugriff oder sie werden ihr als Bestandteil der Botschaft übergeben.

Die Operanden in den Botschaften können aktuelle Werte, variable Datenobjekte oder auch Objekte sein.

Zustands-größen:	Position (x, y) Bewegungsrichtung	$0 \le x \le 127$; $0 \le y \le 39$ links, rechts, oben, unten
Attribute:	Symbol Farbe	P (Pfeil), H (Hand), K (Karo) ... schwarz, weiß, rot, grün, blau ...
Operationen:	setze dich auf Position (x, y); aktualisiere das Symbol auf den Wert (S); aktualisiere die Farbe auf den Wert (F); aktualisiere die Bewegungsrichtung auf den Wert (R); bewege dich in der vorgegebenen Bewegungsrichtung um (s) Stellen; liefere die aktuelle Position (x, y); liefere den Wert für das aktuelle Symbol (S); liefere die aktuelle Farbe (F); liefere die aktuelle Bewegungsrichtung (R); ...	

Objekt ´Cursor´ mit Attributen, Zustandsgrößen und zugeordneten Operationen

Objekte mit den gleichen Eigenschaften, die sich lediglich durch die temporären Werte ihrer Attribute und Zustandsgrößen unterscheiden, werden zu sog. *Klassen* zusammengefasst.

Eine *Klasse* ist also eine abstrakte Repräsentation für eine Menge von konkreten Objekten mit dem gleichen Satz von Datenobjekten und zugeordneten Operationen. Ein Objekt kann demnach als *Instanz* einer bestimmten Klasse aufgefasst werden. Als *Instanzmerkmale* bezeichnet man die Datenobjekte und Operationen, die in einer Klasse spezifiziert werden und ein Objekt der Klasse charakterisieren.

Solche Klassen existieren nicht isoliert voneinander, sondern stehen in unterschiedlichen Beziehungen. Diese drücken sich beispielsweise in gleichen Attributen oder vergleichbaren Operationen aus. Klassen mit

solchen Beziehungen können aus bereits definierten Klassen durch sog. *Vererbung* abgeleitet werden.

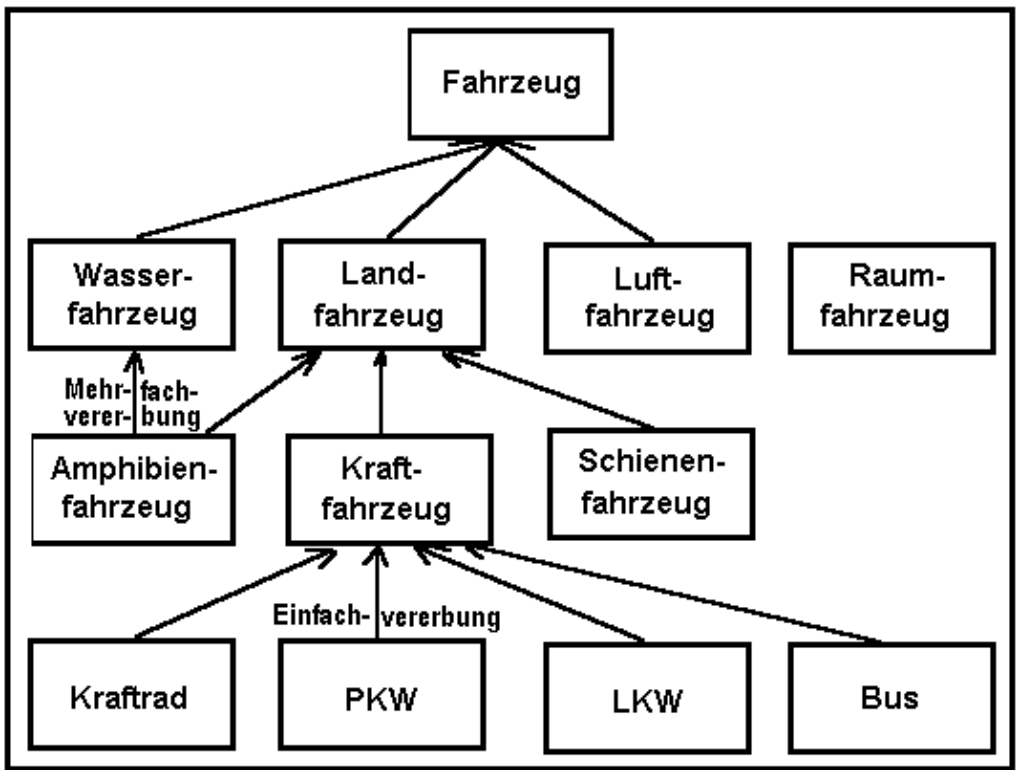


Abbildung 4-1 Objekte, die verschiedene Arten von Fahrzeugen repräsentieren

Objekte bieten sich als die idealen konstruktiven Bausteine zur Gestaltung der Ablaufstruktur eines Software-Systems an.

Beispiel einer Klassenspezifikation als Modell für Kreisobjekte:

Klasse: KREIS

Komponentenobjekte:

MittelPunkt: PUNKT;

Primäre Attribute:

Radius: cm;

RandBreite: mm := 0.25;

RandFarbe: Farbe := SCHWARZ;

FüllMuster: Muster := GLAS;

MusterFarbe: Farbe := HINTERGRUND;

MaxGeschw: cmSek;

Abhängige Attribute:

Umfang: cm; Fläche: cm^2;

Zustandsgrößen:

Sichtbar: BOOLEAN;

VergrFaktor: Faktor := 1.0;

VerschiebeDistanz: cm := 0.0;

VerschiebeRichtung: Grad;

Operationen:

erzeuge_Kreis (MittelPunkt: PUNKT; Radius: cm);

lösche_Kreis; zeige_Kreis;

ist_sichtbar return BOOLEAN;

setze_RandDaten (RandBreite: cm; RandFarbe: Farbe);

setze_FüllDaten (FüllMuster: Muster; MusterFarbe: Farbe);

setze_MaxGeschw (Gschw: cmSek);

setze_VergrFaktor (VergrFaktor: Faktor);

setze_VerschiebeDaten (Distanz: cm; Richtung: Grad);

markiere_MittelPunkt (Markierung: MarkSymbol);

vergrößere_prompt;

verschiebe_prompt;

vergrößere_kontinuierlich (Geschw: cmSek);

verschiebe_kontinuierlich (Geschw: cmSek);

ende KLASSE KREIS;

Die Ablaufstruktur entsteht nicht wie bisher als eine hierarchische Aufruffolge, die ausgehend von einem Hauptprogramm über höherrangige Steuerprozeduren nach unten auf die auszuführenden Operationen wirkt. Sie wird stattdessen aus Bausteinen mit wohldefinierten und eindeutigen Schnittstellen zusammengefügt. Jeder Baustein deckt jeweils einen speziellen, abgeschlossenen Bereich des realen Anwendungsbereichs ab. Er repräsentiert eine Einheit, die wie ein natürliches Objekt im Zusammenspiel mit anderen Objekten agiert und reagiert.

4.3 Zusicherungen

Operationen müssen sowohl *syntaktisch* (formal, grammatikalisch) als auch *semantisch* (inhaltlich) richtig *spezifiziert* (definiert) werden.

Beispiele für eine syntaktische Spezifikation:

procedure get_String (EString: out:STRING; Laenge: out NATURAL);

Die Prozedur ´get_String´ liefert zwei Ausgangsgrößen, nämlich die Zeichenkette, die zuletzt eingegeben wurde, und die Länge der Zeichenkette.

procedure setze_Stein (Stein: FIGUR; X, Y: POSITION; Paarung: in out BRETT := PBeispiel);

Die Prozedur ´setze-Stein´ belegt das Feldelement (X, Y) mit einem Stein, falls dies nach den Spielregeln möglich ist. Sie wird auf eine aktuelle Spielpaarung angewendet, mit der die Prozedur versorgt wird. Da diese im allgemeinen durch die Prozedur verändert wird, muss für den Parameter ´Paarung´ der Modus **in out** gewählt werden.

Durch die syntaktische Spezifikation der Operation, insbesondere durch eine geeignete, den Zweck der Operation erklärende Wahl der Namen für die Operation, die Parameter und deren Typ, wird bereits ein deutlicher Bezug zur Semantik der Operation hergestellt. Die korrekte Interpretation der Namen im Hinblick auf das beabsichtigte Verhalten der Operation ist natürlich nicht gewährleistet.

Eine wichtige Methode der semantischen Spezifikation von Operationen sind die sog. *Zusicherungen*.

Zusicherungen (*assertions*) sind logische Bedingungen, die sich primär auf die Zustandsgrößen eines Objektes beziehen, die von den auf das Objekt anwendbaren Operationen gemeinsam genutzt werden. Dabei werden folgende Gruppen von Zusicherungen unterschieden:

① Vorbedingungen

Einschränkungen von Eingangsparametern (Botschaften) und Zuständen eines Objekts vor Ausführen einer Operation

② Nachbedingungen

Einschränkungen von Ausgangsparametern und Zuständen eines Objekts nach Ausführung einer Operation

③ Invarianten

Bedingungen, die sowohl vor als auch nach Ablauf der Operation erfüllt sein müssen. In der Regel sind hier variable Zustandsgrößen aufgeführt, deren jeweiliger Wert nach dem Ablauf der Operation der gleiche sein muss wie vor ihrem Ablauf.

Eine andere Möglichkeit der semantischen Spezifikation von Operationen sind beispielsweise auch die altbekannten **Entscheidungstabellen**.

Zusicherungen werden in Form von

⇨ logischen Ausdrücken mit Vergleichsoperatoren (**=, /=, <, <=, >, >=**)

⇨ logischen Operatoren **and, or, xor**

⇨ Enthalten sein Operatoren **in, not in**

⇨ Universal- oder Existential-Quantifier **for_all, exists**

gemacht.

Auf der folgenden Seite sehen wir dazu ein Beispiel. Es geht dort um die Vereinbarung von Typen und gemeinsam genutzten Datenobjekten. Bei ´Char´ und ´Zahlen´ handelt es sich im Beispiel um Zustandsgrößen, auf welche die Funktionen ´find_Ziffer´ und ´find_Index´ angewendet werden.

Beispiel:

```
subtype ZIFFER is CHARACTER range ´0´.. ´9´;
type LISTE is array (1 .. 100) of STRING (1 .. 10);
Char: CHARACTER := ´0´; Zahlen: LISTE;
Spezifikation der zugehörigen Operationen:
function find_Ziffer (Zahl: in STRING) return BOOLEAN asserts
precond
    (Char in ZIFFER) and (for_all i in 1 .. Zahl´LENGTH: Zahl(i) in ZIFFER);
postcond
    find_Ziffer (Zahl) = exists i in 1 .. Zahl´LENGTH: Zahl(i) = Char;
invariants
    Char´NEW = Char´OLD; Zahlen´NEW = Zahlen´OLD; Zahl´NEW =
    Zahl´OLD;
end find_Ziffer;
function find_Index (Zahl: in STRING) return NATURAL asserts
precond
    for_all i in Zahl´RANGE: Zahl(i) in ZIFFER;
postcond
    if exists i in Zahlen´RANGE: Zahlen(i) = Zahl
    then Zahlen (find_Index (Zahl)) = Zahl;
    else find_Index (Zahl) = 0;
    end if;
invariants
    Char´NEW = Char´OLD; Zahlen´NEW = Zahlen´OLD; Zahl´NEW =
    Zahl´OLD;
end find_Index;
... weitere Operationen
```

Die Vorbedingung in ´find_Ziffer´ verlangt, dass ´Char´ ein Ziffernzeichen enthält, und dass die übermittelte Zeichenkette ´Zahl´ ausschließlich aus Ziffernzeichen besteht. Die Nachbedingung in ´find_Ziffer´ bedeutet, dass der Ergebniswert dieser Funktion dem Wert TRUE entsprechen muss, wenn ´Zahl´ ein Ziffernzeichen enthält, andernfalls dem Wert FALSE. Durch die invarianten Bedingungen wird ausgedrückt, dass die Zustandsgrößen ´Char´ und ´Liste´ und die Eingangsgröße ´Zahl´ durch keine der beiden Funktionen verändert werden dürfen.

4.4 Unterschiede Funktionsorientierung - Objektorientierung

Konventionell strukturierte Software-Systeme haben folgenden hierarchischen Aufbau:

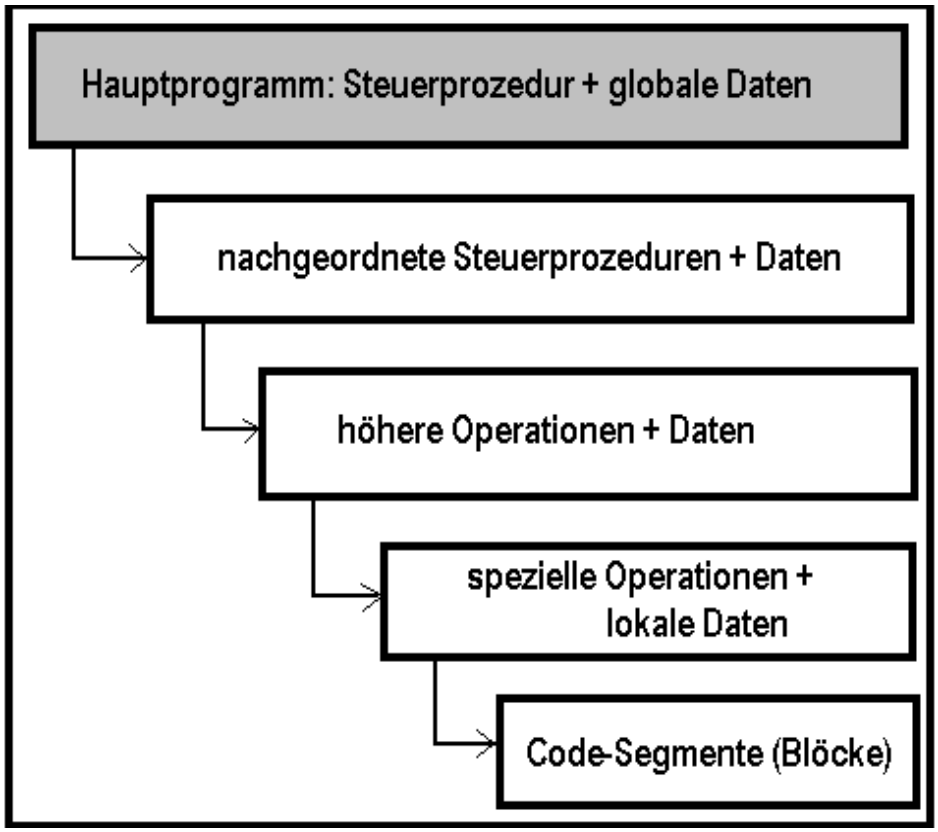


Abbildung 4-2

Bei mehrfacher Nutzung von untergeordneten Operationen kann es zu Konflikten zwischen den vereinbarten Datentypen kommen.

Insgesamt besitzen konventionelle, funktionsorientierte Software-Systeme folgende Merkmale:

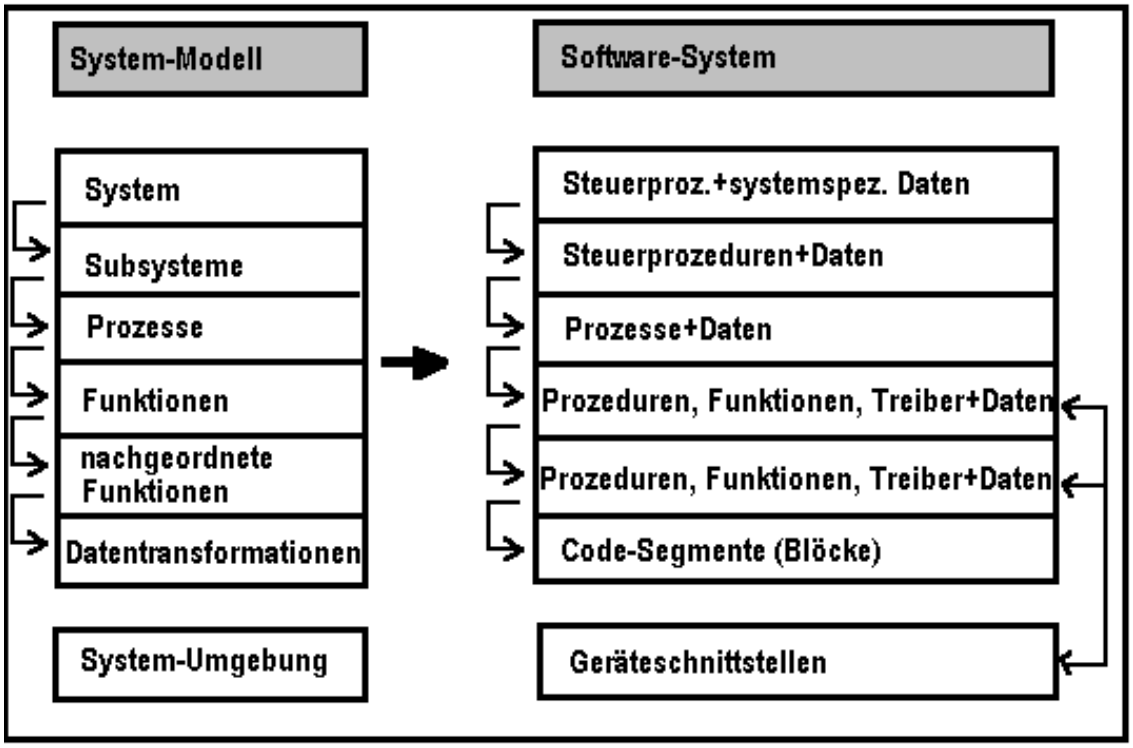
① Hierarchische Strukturen, die auf einen funktionalen Ablauf zugeschnitten sind.

② Getrennte Betrachtung von Programm- und Datenraum führt zu Schnittstellenproblemen.

③ Primär wird die Ablaufstruktur betrachtet und danach eine ´modulare´ Aufbaustruktur darauf zugeschnitten. Module greifen auf gemeinsame Datenstrukturen zu.

④ Diese Vorgehensweise führt zu massiven Problemen bei Änderungen oder Wiederverwendung.

Diese Charakteristika haben ihren Ursprung bereits in dem aus der Analysephase entstandenen Systemmodell, aus dem das Software-Modell direkt abgeleitet wird.



Software-Entwicklung: Systemmodell --> Software-System

Abbildung 4-3

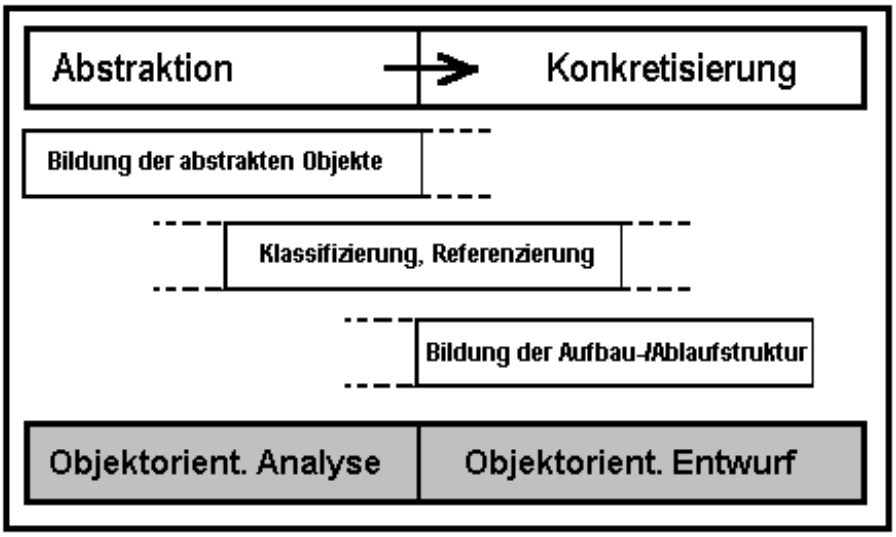
Die objektorientierte Vorgehensweise weist dagegen folgende Merkmale auf:

① Strukturen, die durch ein hohes Maß an Abstraktion gekennzeichnet sind, verbunden mit der Dezentralisierung und Lokalisierung der Daten und der darauf anwendbaren Operationen.

② Es entstehen elementare Bausteine mit einer unbekannten inneren Struktur und einer definierten Schnittstelle, über welche die Nutzung der verfügbaren Operationen ausschließlich möglich ist.

③ Auf Basis der elementaren Bausteine lassen sich höhere Aufbaustrukturen erstellen. Aus diesen können zunächst Prototypen der Ablaufstruktur des Software-Systems abgeleitet werden. Aus diesen werden schließlich verbesserte Bausteine, die mit allen Systemanforderungen übereinstimmen.

Objektorientierung ist weniger eine Frage der Codierung als vielmehr eine Frage von Analyse und Entwurf. Hier gibt es erhebliche Unterschiede zur konventionellen Vorgehensweise.



Objektorientierter Ansatz: Analyse und Entwurf

Abbildung 4-4

4.5 Testansätze für OOP

4.5.1 Testproblematik bei OOP

Wie wir gesehen haben, basieren die konventionellen Testmethoden und Integrationsstrategien auf hierarchisch aufgebauten, funktionsorientierten Programmen, Modulen und Systemen.

Insbesondere beim White-Box Testen gehen wir davon aus, dass unsere zu testenden Module einen kontrollflussorientierten, strukturierten Aufbau haben, in dem sich eindeutig die abzudeckenden Zweige und Pfade identifizieren lassen. Auch bei den Integrationsstrategien war unser Ausgangspunkt ein modular aufgebautes, hierarchisch strukturiertes Modul-System mit einer klaren Aufrufhierarchie.

Die Black-Box Testmethoden sind prinzipiell verwendbar. Wir müssen aber beachten, dass die Vorgaben gegen die wir testen, etwas anders aussehen als wir es von herkömmlichen Fach- und DV-Konzepten gewohnt sind. Auch Entscheidungstabellen sind hilfreich.

Probleme beim Test objektorientierter Programme:

- ⇨ Es gibt mehr Abhängigkeiten gegenüber Funktionsorientierung, die sich äußert durch
  - starke Modularisierung,
  - Vererbung und
  - implizit gemeinsam genutzten Daten.
- ⇨ Der Argumentbereich von Operationen besteht nicht nur aus Eingangsparametern (Botschaften), sondern auch aus dem Objektzustand.
- ⇨ Es fehlt ein klar definiertes Benutzungsprofil, da sehr stark auf Wiederverwendbarkeit geachtet wird.

Dies führt zu folgenden Schwierigkeiten:

- ⇨ Test ist an Funktionen orientiert, Programme sind objektorientiert
- ⇨ Eingrenzung des Funktionsbereichs der Operationen einer Klasse
- ⇨ Integration von Spezifikation und Test
- ⇨ Zustandsinitialisierung
- ⇨ Objektzustandsprotokollierung
- ⇨ Selektive Instrumentierung
- ⇨ Testüberdeckungsermittlung nach Nutzungsprofil

4.5.2 Generelle Teststrategie

Prinzipiell sollte ein Test bestehen aus dem

- ⇒ Instanztest
- ⇒ Kontexttest
- ⇒ Vollständigkeitstest
- ⇒ Zustandstest.

Instanztest

Hierbei geht es um die Auswahl repräsentativer Instanzen einer Klasse mit ihren konkreten Merkmalen, also insbesondere den Instanz-Datenobjekten und Instanz-Operationen.

Kontexttest

Hierbei geht es um die Simulation des Empfangs aller möglichen Botschaften an ein Objekt. Es sollte versucht werden, alle praxisrelevanten Ausnahmebedingungen auszulösen und alle dynamischen Bedingungen aller praxisrelevanten Objekte zu erzeugen.

Vollständigkeitstest

Versuch, alle Operationen und Anweisungen der Objekte mit Testfällen zu überdecken. Dies umfasst auch die Veränderung aller Objekte und deren Attribute.

Zustandstest

Hierbei geht es um die Erzeugung aller praxisrelevanten Zustände und Zustandsübergänge aller praxisrelevanten Objekte.

Aufgrund des hohen Abstraktionsniveaus bedingt durch das Ziel einer hohen Wiederverwendbarkeit der Objekte und Klassen, muss man sich beim Test auf die konkreten, praxisrelevanten Objekte, Operationen und Bedingungen beschränken.

Besonderes Gewicht muss beim Test auf die spezifizierten Klassen und ihren Beziehungen gelegt werden. In diesem Zusammenhang spricht man auch vom sog. *Klassentest*.

4.5.3 Die Methode Klassentest

Der Klassentest besteht aus dem sog. *Unit-Test* und dem *Integrationstest*.

❶ **Unit-Test**

Voraussetzung für den Test einer Entwurfseinheit ist, dass diejenigen Entwurfseinheiten (Basismodule), die sie für ihren Ablauf benötigt, bereits getestet sind. Im Hinblick auf eine Aufrufhierarchie muss der Test also inkrementell von unten nach oben durchgeführt werden.

Konkret umfasst der Test

- jede Ausprägung eines Objekts
- jede Operation isoliert durch entsprechende Botschaften
- alle Objektzustände
- die Folgen abhängiger Operationen einer Klasse mit ihren Interaktionen

Für diesen Zweck müssen nach Bedarf Testprozeduren entwickelt werden, die geeignete Testdaten generieren und die Ergebnisse in auswertbarer Form darstellen.

❷ **Integrationstest (Ablaufverfolgung)**

Es sollen die Interaktionen einer Kette von Klassen ausgeführt werden (Kettentest). Der Integrationstest umfasst folgende beiden Ziele:

- Prüfung der Auswirkungen bei Verwendung von geerbten Attributen und Operationen (Kette durch Vererbungshierarchie)
- Prüfung der Zustandsfolgen durch Botschaften zwischen Instanzen (Kette durch unabhängige Objekte)

Besonders kritisch ist dieser Test bei der Überprüfung von parallelen Abläufen, da hierbei das zeitliche Verhalten berücksichtigt werden muss.

Auch der OO-Test muss bereits bei der Spezifikation der Klassen ansetzen. Semantisches Mittel hierzu ist die Zusicherungstechnik.

Zum Abschluss noch ein kurzer Hinweis zu den Anforderungen an eine Testumgebung zum OO-Test.

4.5.4 Anforderungen an eine Testumgebung

Anforderungen an eine OO-Testumgebung

Testtreiber
Simulation der vererbten Werte.
Aktivierung der Operationen.

Botschaftengenerator
Erzeugung der Eingangsparameter.

Zustandsinitialisator
Versetzung des Objekts in den Vorzustand.

Zustandsauswerter
Festlegung des Nachzustandes des Objekts.

Botschaftenauswerter
Vergleich der Ausgangsnachricht mit der vorgegebenen Nachbedingung.

Testmonitor
Protokollierung der Ausführungsfolgen von Operationen innerhalb einer Klasse und in Klassenketten, sowie der Zustandsfolgen

Bei Auswahl einer OO-Entwicklungs- und Programmierumgebung sollte darauf geachtet werden, dass derartige Testwerkzeuge integriert sind.

4.6 Objektorientiertes Vorgehensmodell

Bei der herkömmlichen, funktionsorientierten Software-Entwicklung wird im Prinzip sequentiell von der Anforderungsanalyse, über Fach- und DV-Konzept, bis zur Integration und Realisierung vorgegangen. Das Ergebnis einer Phase wird verifiziert und geht als Input in die nächste Phase ein:

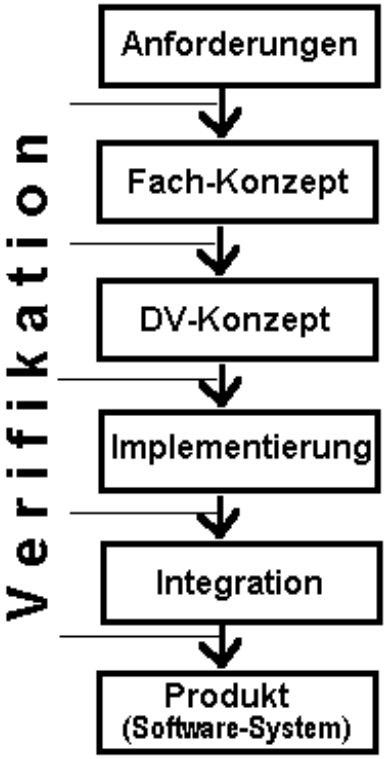


Abbildung 4-5

Mit der konkreten Testplanung muss spätestens im Fachkonzept begonnen werden.

Bei der objektorientierten Software-Entwicklung geht man dagegen nicht sequentiell, sondern viel stärker schleifenförmig (iterativ) vor. Die Phasen Anforderungsanalyse, Entwurf und Implementierung werden wiederholt durchlaufen und münden jeweils in einem Prototyp. Dieser Prototyp wird evaluiert (bewertet) und bildet die Basis für einen erneuten Schleifendurchlauf, in dem der vorangehende Prototyp korrigiert und optimiert wird.

Objektorientiertes Vorgehensmodell:

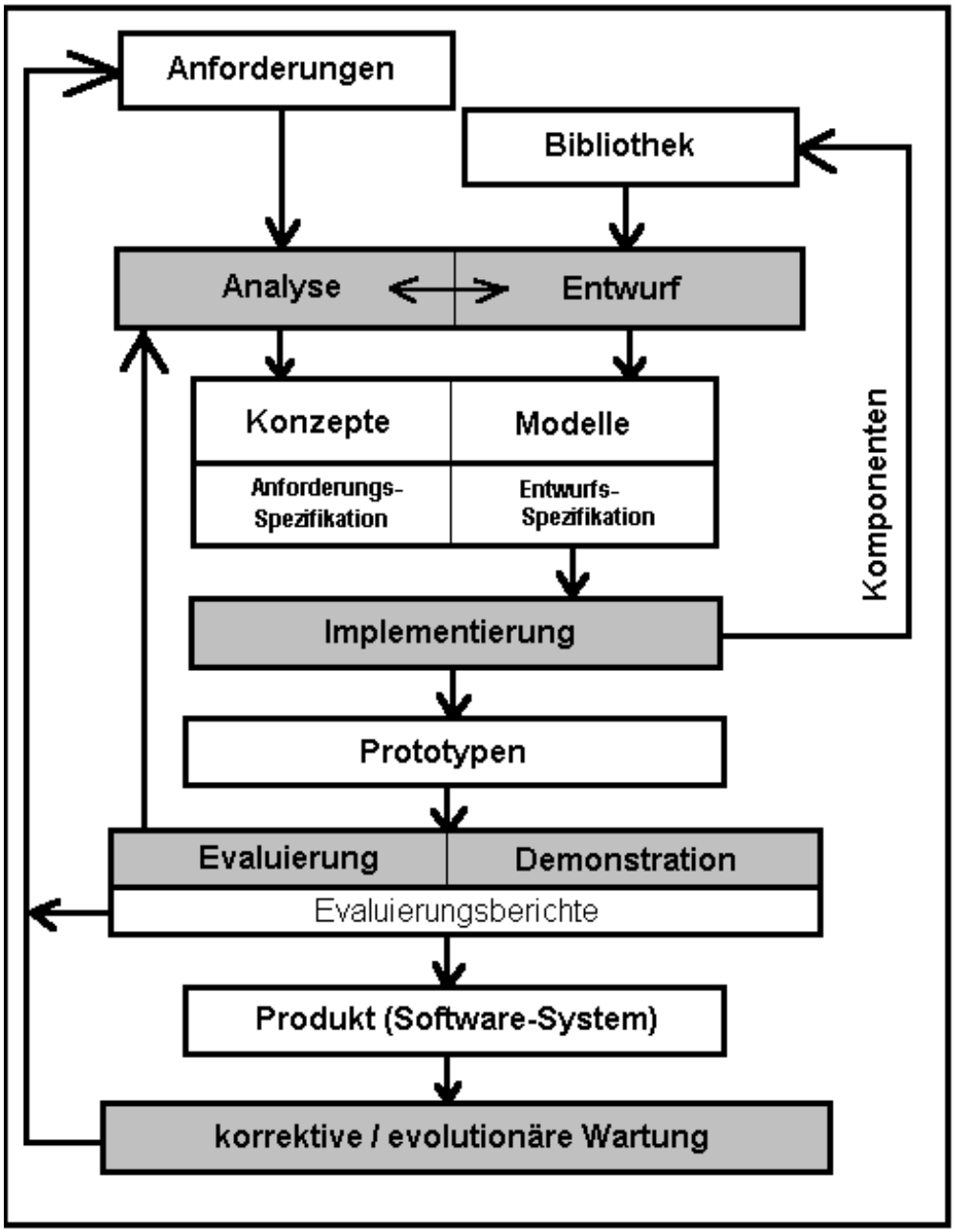


Abbildung 4-6

In Bezug auf das Testen stellt sich hier das Problem, dass bei jedem Schleifendurchlauf der entstandene Prototyp erneut getestet werden müsste (vgl. Regressionstest). Es handelt sich schließlich jeweils um ein neues Software-Produkt.

5

Testdurchführung und Fehlermanagement

5 Testdurchführung und Fehlermanagement

5.1 Abgrenzung der Teststufen

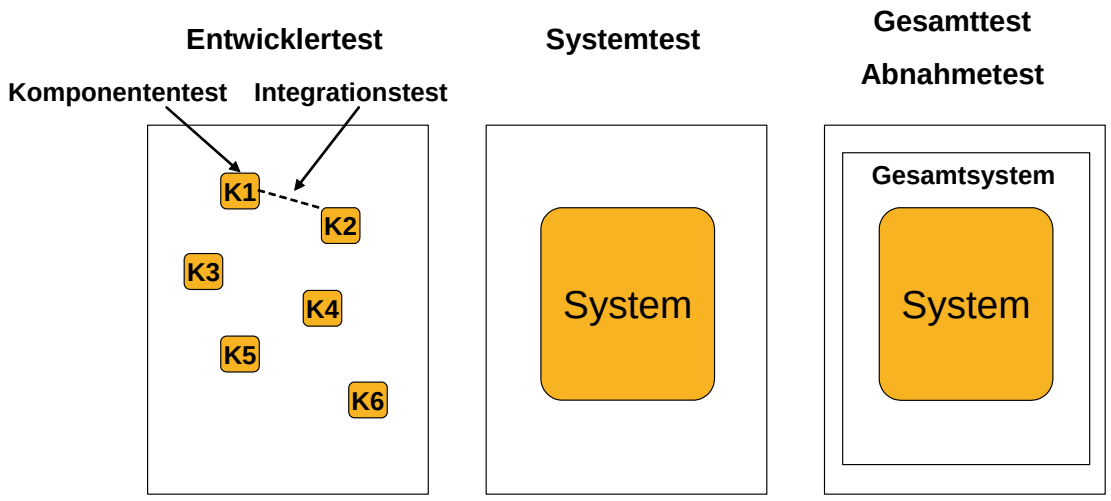


Abbildung 5-1: Abgrenzung der Teststufen

5.1.1 Entwicklertest

Der Entwicklertest unterteilt sich in den Komponenten- und Integrationstest.

Der **Komponententest** testet die einzelnen Komponenten, die in der Anwendungsentwicklung erstellt werden. Der Komponententest verfolgt das Ziel der Sicherstellung der Lauffähigkeit der Programme und des Nachweises der Korrektheit gegen die Spezifikation.

Schwerpunkt des Komponententests ist es zu testen, ob die in der DV-Spezifikation beschriebenen Anforderungen richtig umgesetzt worden sind. Die Anforderungen umfassen sowohl technische und fachliche Korrektheit als auch formale Prüfungen (z. B. Plausibilitätsprüfungen), ggf. Datenbankschnittstellen und gültige Programmierrichtlinien.

Wichtig: Dabei handelt es sich um einen White-Box-Test, der in der Regel vom Entwickler der Komponente eigenverantwortlich in seiner Entwicklungsumgebung durchgeführt wird.

Typische Testendekriterien im Entwicklertest leiten sich aus den White-Box-Testverfahren ab:

⇒ Anweisungsabdeckung (z. B. 100 % gefordert)

⇒ Zweigabdeckung

⇒ Bedingungsabdeckung.

Im **Integrationstest** werden die Schnittstellen zwischen den Komponenten aus technischer Sicht hinsichtlich Stabilität und Vollständigkeit getestet.

5.1.2 Systemtest

Der **Systemtest** ist in der Regel eine Teststufe, die sicherstellt, dass das entwickelte System gemäß den fachlichen Vorgaben vollständig und korrekt umgesetzt wurde.

Der Systemtest soll Abweichungen im Zusammenspiel zwischen integrierten Komponenten oder Systemen aufdecken.

Der Systemtest soll den Nachweis des korrekten Zusammenwirkens aller fachlichen Einzelfunktionen innerhalb des betrachteten Systems erbringen. Fokus ist die Gültigkeit von Funktionsabläufen. Es sollen Fehler im Zusammenwirken der Einzelfunktionen gefunden werden.

5.1.3 Gesamttest

Der **Gesamttest** testet das gesamte Portfolio übergreifend. Im Fokus des Tests stehen so genannte End-to-End Geschäftsprozesse, d. h. eine vollständige und korrekte Verarbeitung zwischen allen beteiligten Systemen.

Beim Test der Geschäftsprozesse sollen Abweichungen im Zusammenhang mit der gewünschten Leistungserbringung des Gesamtsystems aufgedeckt werden.

5.1.4 Testfälle pro Teststufe

Um die oben beschriebenen Ziele in den jeweiligen Teststufen zu erreichen, müssen entsprechende Testfälle definiert werden.

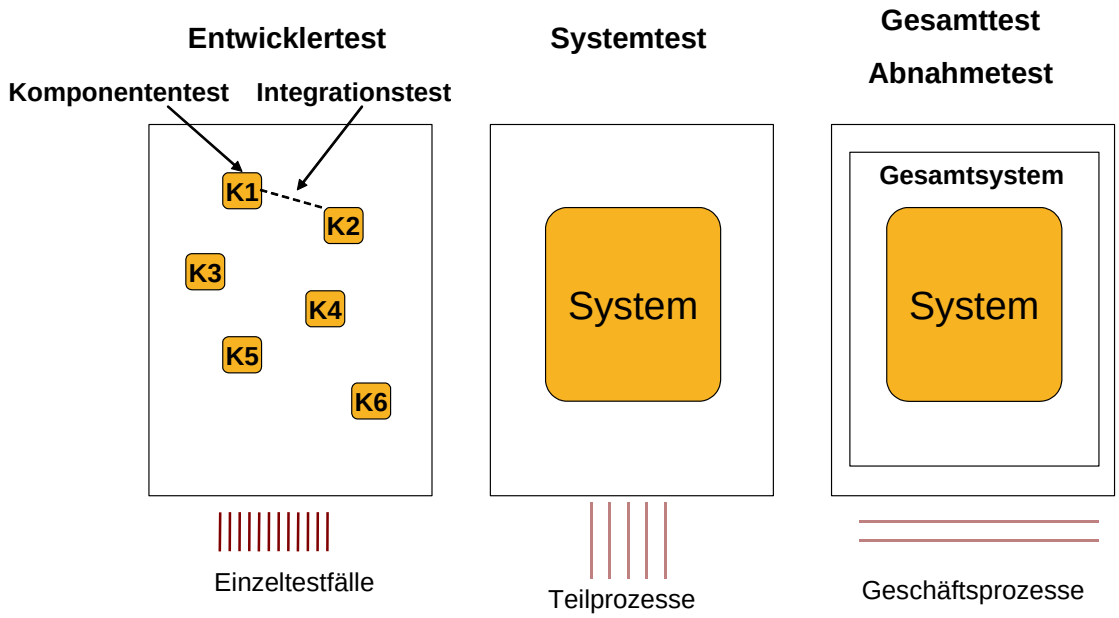


Abbildung 5-2: Testfälle pro Teststufe

Im Entwicklertest wird es eine Vielzahl von spezifischen Testfällen geben, die eine ganz bestimmte Problematik testen (Einzelfunktionen).

Das Ziel im Systemtest ist der Nachweis des Zusammenwirkens der Einzelfunktionen und damit liegt der Fokus bez. der Testfälle auf den Teilprozessen.

Im Gesamttest werden letztendlich Testfälle für die Geschäftsprozesse benötigt.

5.2 Last- und Performance-Test

5.2.1 Ziele und Testendekriterien

Der Performancetest dient dazu, die Leistungsfähigkeit bzw. Effizienz der Anwendung oder eines Moduls zu bewerten. Effizienz bezeichnet hier das Verhältnis zwischen dem Leistungsniveau der Software und dem Umfang der eingesetzten Betriebsmittel unter festgelegten Bedingungen. Im Performance-Test wird gegen technische Vorgaben in Form von Leistungsbeschreibungen im IT-Konzept und / oder Fachkonzept und dem Nutzungsprofil getestet.

Prinzipiell kann der Performancetest in allen Entwicklungsphasen eine Rolle spielen. Je nach Anforderungen an das Qualitätsniveau und der geforderten Testtiefe (Detaillierungsgrad der Messungen) kann der Performance-Test mit verschiedenen Verfahren durchgeführt werden. Die verwendeten Verfahren unterscheiden sich neben der erreichbaren Testtiefe vor allem durch den mit ihnen verbundenen Aufwand.

Prinzipiell gliedert sich ein umfassender Performancetest in drei Schritte:

a) Performance-Messungen

b) Lasttest

c) Stresstest.

Mit den Performance-Messungen wird die lastabhängige Charakteristik zeitkritischer Anwendungen gemessen. Die Ergebnisse der Performance Messungen geben Auskunft über das Skalierungsverhalten der Anwendung. Bei den Performance-Messungen werden die Testobjekte separat ausgeführt und ausgemessen.

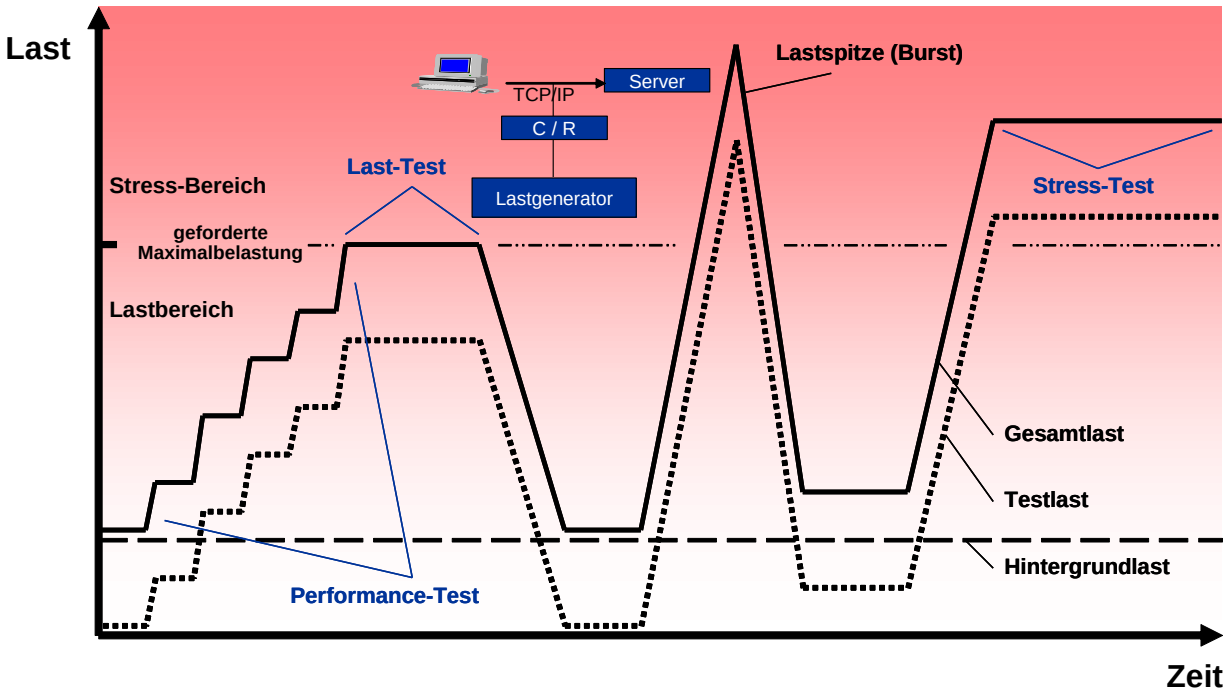


Abbildung 5-3: Abgrenzung Performance- / Last- und Stresstest

Der Lasttest testet das System in seinem Gesamtverhalten unter hoher Belastung. Mit einem Lasttest, der das reale Verhalten von Anwendern simuliert (Nutzungsprofil), kann das Verhalten der Anwendung unter dauerhafter (normaler) Last gemessen werden. Bei Lasttests werden die Geschäftsprozesse entsprechend der Nutzungsprofile zu einem oder mehreren Szenarien zusammengefasst.

Der Stresstest testet das System unter maximaler Belastung. Der Stresstest zeigt die Grenzen der Anwendung. Bei Stresstests werden wie bei dem Lasttest die Geschäftsprozesse entsprechend der Nutzungsprofile zu einem oder mehreren Szenarien zusammengefasst.

Durch die Analyse der Ergebnisse werden die Leistungsgrenzwerte der Anwendung oder einzelner Module ermittelt. Das Ergebnis hilft dabei, Flaschenhälse in der Anwendung und der dazugehörenden Infrastruktur zu lokalisieren.

Der Performancetest ist dann abgeschlossen, wenn die definierten Anforderungen durch den Test sichergestellt wurden, bzw. Der Auftraggeber sich einverstanden erklärt, definierte Abweichungen zuzulassen.

5.2.2 Voraussetzungen für die Testausführung

⇒ Die Anwendung muss fachlich weitestgehend korrekt und stabil sein

⇒ Eine produktionsnahe bzw. identische Testumgebung muss vorliegen

⇒ Die Testscripts und Testdaten müssen zu diesem Zeitpunkt komplett sein.

5.2.3 Testvorgehensbeschreibung

Gemessen werden als Testobjekte entweder performancerelevante Transaktionen, also Online- oder Batchfunktionen, die in der Anwendung durchgeführt werden, oder einzelne Hardware-Komponenten. Ein Geschäftsprozess ist eine Kette von Transaktionen. Es handelt sich hierbei um einen typischen Ablauf, wie ihn ein Anwender durchführt. Die performancerelevanten Geschäftsprozesse werden im Rahmen einer Profilanalyse (Profiling) durch Nutzungsprofile der Anwendung ermittelt. Hierbei wird durch Gespräche mit den Fachabteilungen und eventuell Statistiken aus der Produktion analysiert, welche Geschäftsprozesse am häufigsten ausgeführt werden und somit besonders performancerelevant sind. Die ermittelten Geschäftsprozesse werden mit Lasttesttools aufgezeichnet und können in einem Retest durch simulierte Nutzer mehrfach wieder abgespielt werden.

Durch die Kombination mehrerer Geschäftsprozesse können dann Lasttestszenarien entwickelt werden. Es werden somit charakteristische Transaktionen einzeln und in deren Zusammenspiel getestet, um sowohl die Effizienz der einzelnen als auch die Abhängigkeiten zwischen mehreren Transaktionen zu erfassen.

Die Sollergebnisse für den Performancetest ergeben sich durch die technischen Vorgaben in Form von Leistungsbeschreibungen im IT-Konzept.

Wichtig: Für den Performancetest ist die Organisation der Testdaten wichtig. Im Gegensatz zu fachlichen Tests werden Massendaten benötigt (bspw. Logins für alle virtuellen User), die nur unter Umständen aus Produktionsdaten gewonnen werden können. Die Massendaten dürfen nicht zu ähnlich sein (um ein realistisches Caching abbilden zu können), aber auch nicht zu sehr differieren (um mit vertretbarem Aufwand parametrisieren zu können).

Nach der Testdurchführung werden die Messungen zu einem Testbericht zusammengefasst; der neben den Messergebnissen auch deren Interpretation (Leistungsbewertung) und Empfehlungen zum Tuning der Anwendung enthält. So lassen sich in Form von Messwerten fundierte statistische Angaben treffen, wie z. B. in Bezug auf die durchschnittliche Antwortzeit, den Durchsatz, die maximale Prozessanzahl oder den benötigten Speicherplatz. Auf Basis dieser Daten wird die Leistungsfähigkeit der Anwendung analysiert und bewertet.

Durch Retests können umgesetzte Tuning-Maßnahmen verifiziert werden.

5.2.4 Anforderungen an die Testumgebung

Die Testumgebung muss möglichst produktionsnah oder produktionsidentisch sein.

Dies betrifft insbesondere:

- Das bestehende Datenvolumen in den Systemen
- Die Leistungsmerkmale und die Ausgestaltung der Hardware
- Die Leistungszuweisung zu logischen Partitionen
- Einer eventuell zusätzlich zu berücksichtigenden „Hintergrundlast“ durch andere Anwendungen.

5.3 Abnahmetest

Teststufen im V-Modell

Teststufen im V-Modell

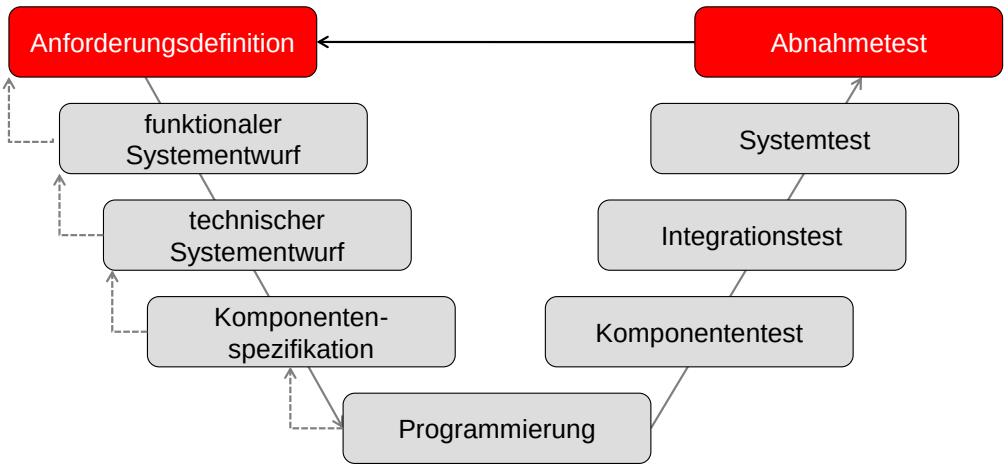


Abbildung 5-4: Übersicht Teststufen – Abnahmetest

5.3.1 Ziele und Testendekriterien

Ziel des Abnahmetests ist, den Auftraggeber in seiner Entscheidung zur Abnahme des Gesamtsystems zu unterstützen. Im Abnahmetest wird final ein Subset aller Testarten geprüft, welche aus Sicht des Auftraggebers geeignet sind, die Abnahmeunterschrift leisten zu können (kritische Funktionen, kritische Prozesse, kritische GeVos).

Das Testende ist erreicht wenn

⇒ alle Testaktivitäten laut Vorgabe ausgeführt wurden

⇒ alle definierten Testfälle im Test ausgeführt wurden

⇒ alle definierten Testfälle ohne Abweichungen einer bestimmten Klasse zu den festgelegten Ergebnissen führen. Alle beteiligten Systeme müssen im Test berücksichtigt werden.

5.3.2 Voraussetzungen

Voraussetzung für die Durchführung des Abnahmetests ist, dass alle Systemtests freigegeben wurden und alle Systeme in der Abnahmetestumgebung installiert und ausführbar sind.

5.3.3 Testvorgehen

Der Abnahmetest gestaltet sich als letzter Test, der zeitlich erst dann beginnt, wenn alle Testfälle im Komponententest und im Systemtest die definierten Testendekriterien erreicht haben. Die definierten Abnahmetestfälle werden im Vorfeld auf die vorherigen Teststufen verteilt und getestet. Somit wird gewährleistet, dass bereits alle abnahmerelevanten Testfälle vor dem Abnahmetest mindestens einmal getestet wurden.

5.3.4 Testvorbereitung

Testobjekte im Abnahmetest sind alle bereits in vorherigen Teststufen und Testarten getesteten Testobjekte. Darüber hinaus kann der Auftraggeber weitere Testobjekte definieren, die aus seiner Sicht die Funktionsfähigkeit des Gesamtsystems prüfen und seine Abnahmeentscheidung unterstützen. Auswahlkriterium für die Testfälle werden insbesondere aus der Sicht des Auftraggebers kritische Testfälle sein sowie Testfälle, die in vorangegangenen Teststufen zu Fehlern geführt haben.

5.4 Fehlermanagement

5.4.1 Definition aus Sicht des Fehlermanagements

Ein Fehler oder eine Abweichung ist

- jedes wichtige, ungeplante Ereignis,
  - das z. B. während des Testens auftritt
    - und daraufhin untersucht oder korrigiert werden muss.

5.4.2 Der Fehlermanagementprozess

Fehlermeldungen sollten in einer zentralen Fehlerdatenbank erfasst und gespeichert werden auf die unter anderem Entwickler, Tester, Testmanager und Projektmanager Zugriff haben.

Der Fehlermanagementprozess beginnt nach jeder Testdurchführung, spätestens jedoch nach jedem Testzyklus --> Analyse des Testprotokolls:

Wurde der Fehler verursacht durch

- einen falschen Testfall, Testautomatisierung?
- oder durch den Tester?

Wenn der Fehler durch den Tester verursacht wurde:

- Fehlermeldung schreiben
- Wenn das Problem bereits bekannt und erfasst wurde sollte die entsprechende Fehlermeldung ergänzt werden
- Die Analyse des Problems ist Aufgabe des Entwicklers

Jede Fehlermeldung wird je nach vermuteter Ursache einem Verantwortlichen (z. B. Entwickler) zugewiesen.

Testmanager und Projektmanager können sich jederzeit über den Status einzelner Fehlermeldungen sowie des gesamten Projektes informieren.

Eine aktuelle Informationshaltung in der Fehlerdatenbank ermöglicht es Probleme sehr schnell zu erkennen und rechtzeitig geeignete Gegenmaßnahmen zu definieren.

5.4.3 Fehlerstatusmodell

Beispiel eines möglichen Fehlerstatusmodells:

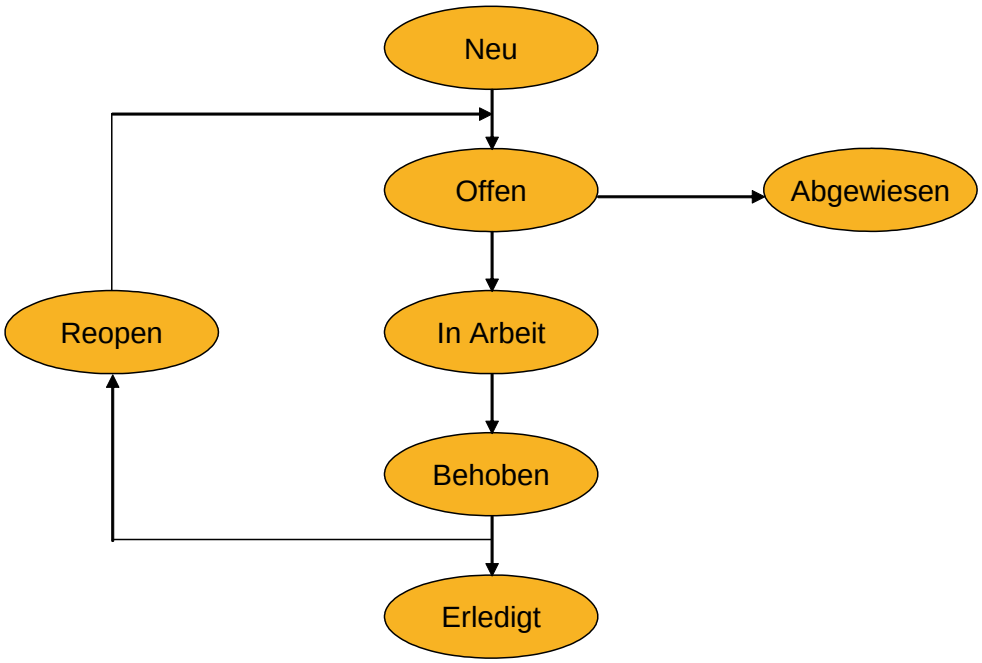


Abbildung 5-5: Fehlerstatusmodell

Beschreibung des Status:

Status	Gesetz durch	Bedeutung
Neu	Tester	Neue Meldung wird erfasst.
Offen	Testmanager	Neue und nicht erfolgreich getestete Meldungen werden gesichtet und auf Vollständigkeit geprüft.
Abgewiesen	Testmanager	Meldung als unberechtigt zurückgewiesen.
In Arbeit	Entwickler	Fehler befindet sich beim Entwickler zur Behebung.
Behoben	Entwickler	Entwickler hat den Fehler behoben; der Nachtest kann erfolgen
Erledigt	Tester	Die Fehlerbehebung war erfolgreich.
Reopen	Tester	Die Fehlerbehebung war nicht erfolgreich.

5.4.4 Inhalte einer Fehlermeldung

Fehlermeldungen sind das **zentrale** Kommunikationsmedium zwischen Entwickler und Tester!

Daher sollten Fehlermeldungen

- neutral und diplomatisch,
- knapp (so knapp wie möglich, so ausführlich wie nötig)
- und verständlich

formuliert sein!

Es bietet sich an Fehlermeldungen nach Identifikation, Klassifikation und der Problembeschreibung zu unterteilen.

5.4.4.1 Identifikation

Attribut	Bedeutung
Nummer	Laufende, eindeutige Meldungsnummer
Testobjekt	Bezeichnung des Testobjekts
Version	Identifikation der genauen Version des Testobjekts
Plattform	Identifikation der Plattform. bzw. der Testumgebung in der der Fehler auftritt.
Entdecker	Identifikation des Tester, der das Problem meldet
Entwickler	Identifikation des für das Testobjekt verantwortlichen Entwicklers
Erfassung	Datum, ggf. Uhrzeit an dem das Problem beobachtet wurde

5.4.4.2 Klassifikation

Status	Gesetz durch	Bedeutung
Neu	Tester	Neue Meldung wird erfasst.
Offen	Testmanager	Neue und nicht erfolgreich getestete Meldungen werden gesichtet und auf Vollständigkeit geprüft.
Abgewiesen	Testmanager	Meldung als unberechtigt zurückgewiesen.
In Arbeit	Entwickler	Fehler befindet sich beim Entwickler zur Behebung.
Behoben	Entwickler	Entwickler hat den Fehler behoben; der Nachtest kann erfolgen
Erledigt	Tester	Die Fehlerbehebung war erfolgreich.
Reopen	Tester	Die Fehlerbehebung war nicht erfolgreich.

5.4.4.3 Problembeschreibung

Status	Gesetz durch	Bedeutung
Neu	Tester	Neue Meldung wird erfasst.
Offen	Testmanager	Neue und nicht erfolgreich getestete Meldungen werden gesichtet und auf Vollständigkeit geprüft.
Abgewiesen	Testmanager	Meldung als unberechtigt zurückgewiesen.
In Arbeit	Entwickler	Fehler befindet sich beim Entwickler zur Behebung.
Behoben	Entwickler	Entwickler hat den Fehler behoben; der Nachtest kann erfolgen
Erledigt	Tester	Die Fehlerbehebung war erfolgreich.
Reopen	Tester	Die Fehlerbehebung war nicht erfolgreich.

5.5 Fehlermanagement versus Anforderungsmanagement

- Fehlermeldungen beschreiben nicht immer Mängel am Produkt.
- Oft beschreibt eine „Fehlermeldung“ gewünschte Erweiterungen der Produktspezifikation.
- Solche Wünsche sind eigentlich neue Anforderungen, die nach der Freigabe der Produktanforderungen eingeführt werden.
- Oft ist die Grenze zwischen „gültigen“ Fehlermeldungen und „ungerechtfertigten“ Wünschen fließend.
- Normalerweise entscheidet das so genannte **Change Control Board** über die Annahme oder Ablehnung von Änderungsanforderungen.
- Das **Change Control Board** kann eine Priorisierung der Bearbeitung von Fehlern der gleichen Fehlerklasse festlegen.

6

Testplanung und -dokumentation

6 Testplanung und -dokumentation

6.1 Grundsätzliche Dokumentation

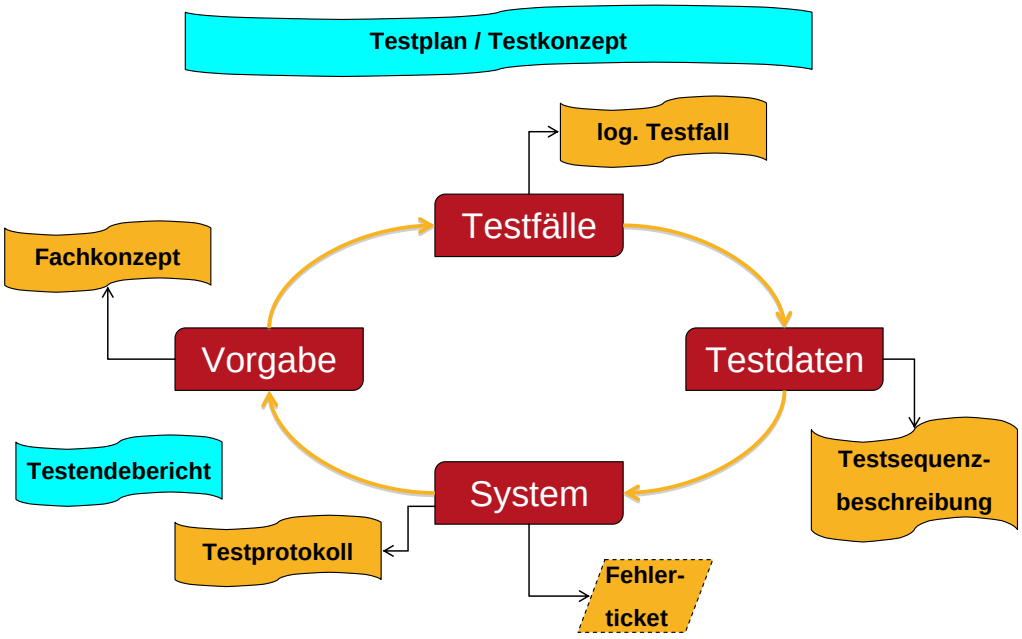


Abbildung 6-1: Übersicht Testdokumentation

Im Verlaufe des Entwicklungsprozesses sollten folgende Dokumente erstellt werden:

- Basis für die Testfallermittlung sind die „Vorgaben“. Dies können beispielsweise sein:
  - Fachkonzept als Ergebnis des funktionalen Systementwurfs
  - DV-Design als Ergebnis des technischen Systementwurfs
- Beschreibung der logischen Testfälle
- Beschreibung der konkreten Testfälle, der Testsequenzen und ggf. der Testszenarien
- Protokollierung der durchgeführten Testfälle
- Bei Auftreten eines Fehlers Erfassung eines Fehlertickets

Aus Testmanagement-Sicht entstehen als grundlegende Dokumente:

- Testplan/Testkonzept und Testendebericht.

6.2 Testkonzept (Testplan)

Die Planung und die Vorbereitung für den Test müssen in einem Projekt so früh wie möglich beginnen. Einfluss darauf können beispielsweise haben:

- Risiken
- Ziele
- Randbedingungen
- Kritikalität des Produktes.

Die Planungsarbeiten des Testmanagers umfassen u. a.

- Festlegung der Teststrategie mit Auswahl der angemessenen Testmethoden
- Entscheidung über Art und Umfang der Testumgebung und der Testautomatisierung
- Festlegung des Zusammenspiels der verschiedenen Teststufen und Integration der Testaktivitäten mit anderen Projektaktivitäten
- Entscheidung, wie Testergebnisse ausgewertet und evaluiert werden
- Festlegung von Metriken zur Messung und Überwachung des Testverlaufs und der Produktqualität sowie Definition der Kriterien für das Testende
- Festlegen des Umfangs der Testdokumente, u. a. durch das Bereitstellen von Templates
- Erstellung der Testplanung mit Entscheidung, wer, was, wann und in welchem Umfang testet
- Schätzung des Testaufwandes und der Testkosten; Aktualisierung von Schätzungen und Plänen im Testverlauf.

Die Ergebnisse dieser Überlegungen beschreibt und begründet der Testmanager im **Testkonzept**.

Gliederung des Testkonzeptes nach IEEE 829:

1) Testkonzeptbezeichnung
2) Einführung
3) Testobjekte
4) Zu testende Leistungsmerkmale
5) Leistungsmerkmale, die nicht getestet werden
6) Teststrategie
7) Abnahme- und Testendekriterien
8) Kriterien für Testabbruch und Testfortsetzung
9) Testdokumentation
10) Testaufgaben
11) Testinfrastruktur
12) Verantwortlichkeiten / Zuständigkeiten
13) Personal, Einarbeitung, Ausbildung
14) Zeitplan / Arbeitsplan
15) Planungsrisiken
16) Genehmigung / Freigabe.

6.3 Beispielgliederung:Testhandbuch

1. Einleitung
- 1.1 Vorwort
- 1.2 Aufbau des Handbuchs
- 1.3 Testorganisation im Überblick
  - 1.3.1 Testobjekte
  - 1.3.2 Testaufgaben
  - 1.3.3 Teststufen
  - 1.3.4 Bearbeiter
  - 1.3.5 Testaktivitäten für Testdurchführung
  - 1.3.6 Test-Tools und Testumgebung
  - 1.3.7 Aufwands- und Aktivitätenverteilung

2. Testmanagement
- 2.1 Voraussetzungen
- 2.2 Testaufgaben
- 2.3 Hauptaktivitäten
- 2.4 Ablauf
- 2.5 Bearbeiter
- 2.6 Ergebnisse

3. Teststufen
- 3.1 Modultest
  - 3.1.1 Voraussetzungen
  - 3.1.2 Testobjekte
  - 3.1.3 Testaufgaben
  - 3.1.4 Hauptaktivitäten
  - 3.1.5 Ablauf
  - 3.1.6 Bearbeiter
  - 3.1.7 Ergebnisse
  - 3.1.8 Testabbruchkriterien
- 3.2 Integrationstest
  - 3.2.1 Voraussetzungen

7. Begriffe

8. Beispieldokumente

9. Formulare

6.4 Testendebericht

Der Testendebericht wird am Ende jeder Teststufe erstellt.

Folgende Inhalte sollte ein Testendebericht mindestens haben:

- Durchgeführte Testaktivitäten; Besonderheiten
- Durchgeführte Testfälle (Anzahl)
- Nicht durchgeführte Testfälle → Begründung
- Risiken aufgrund der nicht durchgeführten Testfälle
- Gesamtstatus der Fehler; insbesondere Betrachtung der noch offenen Fehler
- Risiken aufgrund der noch offenen Fehler
- Empfehlung bez. weiterem Vorgehen.

7

Testwerkzeuge

7 Testwerkzeuge

7.1 Überblick Testwerkzeuge

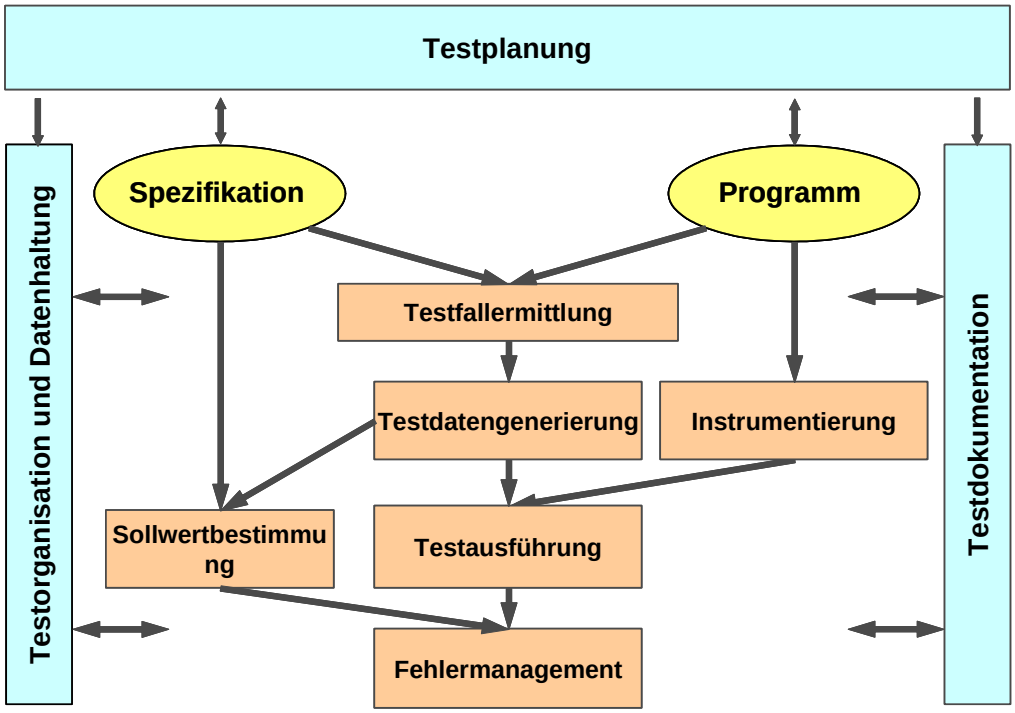


Abbildung 7-1: Übersicht Testwerkzeuge [nach K. Grimm, TU Berlin 1994]

Testwerkzeuge werden unterschieden nach Unterstützung für:

- das Management des Testens: Test-, Anforderungs-, Abweichungs- und Konfigurationsmanagement
- den statischen Test: Reviews, statische Analyse, Modellierung
- die Testspezifikation: Testentwurf, Testdatengeneratoren und -Editoren
- die Testdurchführung und -Protokollierung: automatisierte Testausführung
- Performancemessungen und Testmonitore
- Fehlermanagement.

7.2 Arten von Testwerkzeugen

7.2.1 Maschinelle Testhilfen

Testen ist ein äußerst kreativer Prozess. Diese Kreativität liegt vor allem darin, gute Testfälle aufzustellen, die mit hoher Wahrscheinlichkeit Fehler aufdecken. Diese intellektuell herausfordernde Aufgabe kann selbstverständlich nicht durch automatische Testwerkzeuge wahrgenommen werden. Auf der anderen Seite können Werkzeuge, die an den richtigen Stellen im Testprozess eingesetzt werden, u. U. den Testaufwand erheblich reduzieren. Einige sinnvolle Tests sind ohne den Einsatz bestimmter Werkzeuge gar nicht möglich.

Betrachten wir rückblickend die bisher behandelten Testmethoden und Teststrategien, so wünschen wir uns maschinelle Testhilfen vor allem in folgenden Bereichen:

- Testdatenhandling
- Testablaufanalyse
- Modul- und Integrationstest
- Regressionstest
- Testüberwachung
- graphische Darstellung
- Debugging.

Ein nicht zu unterschätzender Vorteil bei der Verwendung von Testwerkzeugen liegt auch darin, dass man diverse Statistiken und Protokolle zu Dokumentationszwecken gleich mitgeliefert bekommt.

7.2.2 Testdatenhandling

Solche Werkzeuge unterstützen bei der Erstellung von Testdaten. Diese können aus bestehenden Beständen extrahiert oder neu generiert werden. Auch die Generierung von Zufalls-Testdaten für statistische Tests ist möglich. Neben der Erstellung von Testdaten übernehmen solche Werkzeuge auch die Verwaltung und Dokumentation der Testdatenbestände einschließlich von Verwendungsnachweisen.

Zur Ergebniskontrolle ist ein automatischer Abgleich der Testvorhersagen mit den Test Ergebnissen möglich. Abweichungen oder unvergleichbare Datenelemente werden dabei protokolliert.

7.2.3 Testablaufanalyse

Beim White-Box-Testen braucht man Informationen über die im Testobjekt jeweils ausgeführten Anweisungen, Zweige oder Pfade. Nur so lässt sich feststellen, ob der angestrebte Testdeckungsgrad wie z. B. C0, C1 oder C2 erreicht wurde. Manuell kann dies bei komplexen Test-

objekten kaum festgestellt werden. Vor der Verwendung solcher Werkzeuge muss das Testobjekt in der Regel instrumentiert werden.

Auch zur Analyse des Datenflusses und der Datenabdeckung existieren Werkzeuge.

Beim Einsatz solcher Testhilfen werden diverse Protokolle generiert. In den Abbildungen 7.2 bis 7.5 sieht man beispielsweise einen Datenflussbericht, ein Testabdeckungsprotokoll, einen Datenabdeckungsbericht und einen Testpfadbericht.

7.2.4 Modul- und Integrationstest

Beim Modultest und beim Integrationstest muss die Umgebung eines Moduls durch STUBS oder DRIVER simuliert werden. Das Testobjekt (Modul) wird in eine standardisierte Umgebung eingebettet, die die Schnittstellen zur Außenwelt simuliert. Das Prinzip eines solchen Testbetts oder auch Testrahmens ist in Abbildung 7.6 zu sehen.

PROGRAMM:	X------X			Datum:	99.99.99
TESTFALL	STUBNAME	STUBCALL	ERGEBNIS-DATUM	AUSGABE-WERT	FEHLER
1	MODULK	1	N	999	
			0	1	*
	DATEIA	1	SATZA		
			FELDA1	X... X	
			FELDA2	9999	
			FELDA3	99	
	DATEIA	2	SATZA		
			FELDA1	X... X	
			FELDA2	9999	
			FELDA3	99	*

Abbildung 7-2: Datenflussbericht

MODUL::	X------X			Datum:	99.99.99
	ABLAUFZWEIG	STMT NR	DURCHLÄUFE	GEFEHLT	
	1	1	2		
	2	5	1		
	3	10	1		
	4	15	2		
	5	20	0	*	
	6	25	2		
	7	30	1		
	8	35	1		
	9	40	5		
	10	45	2		
Getroffene Ablaufzweige = 9 Nicht getroffene Ablaufzweige = 1 Testabdeckungsgrad = 90 %					

Abbildung 7-3: Testabdeckungsprotokoll

MODUL-NAME	X------X			Datum: SEITE:		99.99.99 999
Daten-Name	Satz-Name	Basis-Name	Prädikat	Eingabe	Ausgabe	nicht benutz
Feld1		MODNAME		*		
Feld3	STRUC	STRUCPTR	*			
Feld5	STRUC	STRUCPTR	*			
Feld8	REC1	FILE1	*			*
Anzahl der Datenelemente = 4 Anzahl der benutzten Datenelemente = 3 Datenabdeckung = 75 %						

Abbildung 7-4: Datenabdeckungsbericht

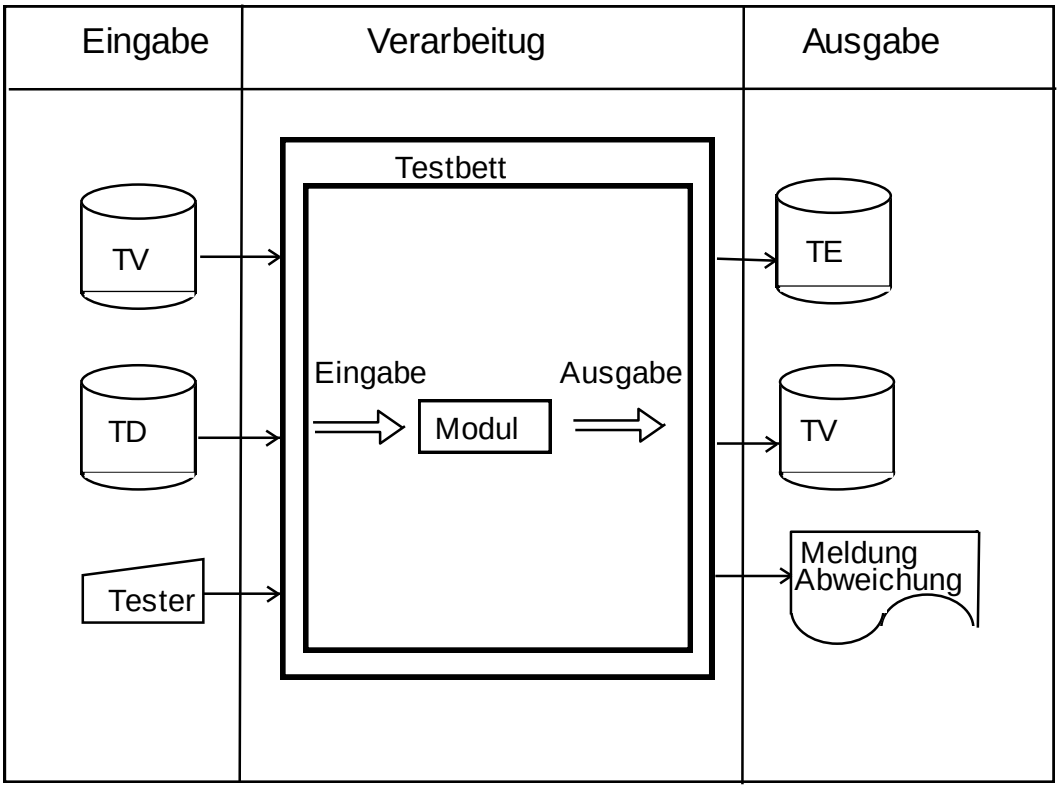
PROGRAMM : X------X Datum: 99.99.99

99

TESTFALL:

	MODUL	ABLAUFZWEIG	STMT NR	DURCHLÄUFE
	HAUPT	2	10	1
		4	18	1
		7	24	5
		11	30	5
		15	40	1
		16	44	1
	MODULA	1	1	1
		3	91	1
		9	16	1
	MODULA1	1	1	1
		4	8	1
		5	10	100

Abbildung 7-5: Testpfadbericht



TD Testdaten
TE Testergebnisse
TV Testvorhersage

Abbildung 7-6: Testbett – prinzipieller Aufbau

Ablauf

1. Der Tester baut Testdaten/ Testvorhersagen auf, die in eigenen Bibliotheken gespeichert werden.
2. Beim Prüfvorgang werden die Testdaten über das Testbett an den zu prüfenden Modul übergeben; die Ergebnisse werden mit den Testvorhersagen verglichen.
3. Je nach Wunsch, wird ein Protokoll ausgegeben
   - mit gesamter Ein-/ Ausgabe
   - mit Abweichungen der geplanten Ausgabe von der tatsächlichen Ausgabe

Regressionstest

Wenn in einem Testlauf Fehler entdeckt wurden und behoben sind, können durch die Fehlerbehebung neue Fehler entstanden sein. Deshalb ist ein Wiederholungstest mit denselben Testdaten und in derselben Testumgebung wie im Testlauf vor der Fehlerbehebung notwendig. Einige Test-Tools besitzen Komponenten, die diesen Regressionstest unterstützen.

Testüberwachung

Zur Testüberwachung werden Werkzeuge angeboten, die Ablaufprotokollierung, Ablaufstatistik und eine Ergebnisprüfung während des Ablaufs durchführen.

Die Ablaufprotokollierung stellt Daten über den Programmablauf zur Ergebnisprüfung nach dem Ablauf und zur Fehlerlokalisierung zur Verfügung. Die Ablaufstatistik stellt diese Daten in statistischer Form durch absolute bzw. relative Häufigkeiten der einzelnen Programmobjekte und durch Datenbereiche dar. Als Programmobjekte finden hierbei Anweisungsmarken, Anweisungstypen, Programmstrukturen und Datenelemente Verwendung.

Graphische Darstellung

Zur Bestimmung einer optimalen Anzahl von Testfällen oder zur Beurteilung der Komplexität von Testobjekten ist eine graphische Darstellung des Ablaufgraphes des Testobjekts sinnvoll. Es existieren Testhilfen, die die Programmstruktur graphisch darstellen und gleichzeitig anzeigen, welche Teile des Ablaufgraphen im Testlauf abgedeckt werden.

Debugging

Eine große Zahl der am Markt verfügbaren Testwerkzeuge unterstützen vor allem bei der Fehlerlokalisierung. Ein Programm, das normaler-

weise kompiliert wird, wird mit einem Testwerkzeug Anweisung für Anweisung interpretiert. Dabei kann man sich ständig die Variableninhalte anzeigen lassen und ändern. Bei manchen Debuggern lässt sich sogar der Programmablauf interaktiv ändern. Durch den Dialog mit dem Testsystem kann man das Verhalten des Testobjekts besser verstehen. Weitere Werkzeuge aus diesem Bereich dienen der Dump-Analyse.

7.3 Unterstützung Testmanagement

Das Tool sollte das Management des gesamten Testprozesses unterstützen; hierzu gehören: Testvorbereitung, Testdurchführung, Datenhaltung, Testdokumentation, Analyse und Auswertung der Ergebnisse.

Die in der Testvorbereitung erstellten Ergebnisse (Testfälle und dazu gehörende Testdaten) werden zentral verwaltet. An die Testfälle werden die für den automatisierten Ablauf notwendigen Testskripte angehängt. Über Testszenarien ist es möglich, beliebige Testabläufe zusammen zu stellen. Die Testergebnisse werden ebenfalls vom Testmanagement-Tool verwaltet. Damit muss es möglich sein, eine lückenlose Dokumentation des Testprozesses im Tool aufzubauen (Fachkonzept → Testfälle und Testdaten → Testszenarien → Testskripte → Testergebnisse).

Darüber hinaus sollte das Tool die im Rahmen des Testmanagements anfallenden Planungs-, Kontroll- und Steuerungsaktivitäten unterstützen und die jeweiligen Ergebnisse komfortabel verwalten.

Anforderungen:

- Erfassung, Verwaltung und Planung aller Testaktivitäten
- Abbildung der Ressourcen- und Zeitplanung
- Unterstützung der Teststeuerung und des Berichtswesens
- Erfassung, Verwaltung und Planung der Testfälle und Testsequenzen
- Verknüpfung der Testfälle mit den Anforderungen.

7.4 Unterstützung Testfallermittlung

Das Tool unterstützt die strukturierte, methodische Testfallermittlung; als Basis für die Testfallermittlung werden die vorhandenen fachlichen Spezifikationen herangezogen. Die Testfälle werden auf Redundanzfreiheit und Vollständigkeit überprüft. Die für den Testfall notwendigen Testdaten werden beim Testfall dokumentiert. Gleiches gilt für das erwartete Sollergebnis.

Anforderungen:

- Strukturierung der Fachlichkeit in Testobjekte
- Unterstützung von Regeln zur effizienten Testfallbildung
- Testfalleditor mit Basis-Textverarbeitungsmerkmalen
- Status für Testfälle und Testobjekte
- Unterstützung von Methoden zur Testfallerstellung
- Testfallverwaltung und Dokumentation.

7.5 Unterstützung Testdatengenerierung

Das Tool muss die Möglichkeit bieten, für beliebige Tests die jeweiligen Datenbanken mit den entsprechend benötigten Testdaten zu füllen, die Datenbanken wieder zu entladen und den Ausgangszustand wieder herzustellen. Des Weiteren sollten Veränderungen des Datenbestandes in der Datenbank selbst und nicht beispielsweise über die Oberfläche der Anwendung überprüft werden können.

Anforderungen Testdatenerstellung:

- Importfunktionalität für projektspezifische Datenstrukturen
- Generelle Wertzuweisungen (Default-Belegungen)
- Bedingte Wertzuweisungen
- Sicherstellung der Datenkonsistenz durch Darstellung von Relationen (z. B. Fremdschlüsselbeziehungen)
- Funktionen zur Datumsverarbeitung
- Basis-Textverarbeitungsmerkmale
- Effiziente Erzeugung von Massendaten
- Generierungsmöglichkeiten aus Testfällen
- Import-Möglichkeit existierender Testdaten.

Anforderungen Testdatenbestandsverwaltung:

- Beliebige Daten durchsuchen, editieren, kopieren, übertragen und umformatieren
- Komfortable Generierung von Testdaten (synthetische Testdaten, Massendaten etc.)
- Update-Fähigkeit bei Datenmodell-Änderungen
- Extrahieren von Testdaten aus vorhandenen Datenbanken oder Dateien
- Komfortables Laden und Entladen der Datenbank zu beliebigen Zeitpunkten und beliebig oft
- Überprüfung der Datenbank-Inhalte aus fachlicher Sicht (es darf hierzu kein besonderes technisches Wissen bez. Datenbank-Aufbau und -Funktionalität erforderlich sein)
- Unterstützung plattformübergreifend und für unterschiedlichste Datenhaltungssysteme
- Konvertierung von Daten (z. B. aus einer Host-basierten DB2-Datenbank in eine relationale Datenbank)
- Vergleiche von Datenbeständen vor und nach dem Testlauf (auf Einfügen, Ändern und Löschen).

7.6 Unterstützung Testausführung (Capture/Replay-Tool)

Ein Capture/Replay-Tool testet in erster Linie grafische Oberflächen (Masken, Eigenschaften und Verhalten der Attribute auf den Masken, Maskenfolgen und Funktionen, die über die Maske aufgerufen werden und deren richtige Ausführung wiederum über die Maske überprüft werden kann).

Der Einsatz eines Capture/Replay-Tools sollte erst erfolgen, wenn die zu testende Anwendung qualitativ einen technisch und fachlich weitest gehend stabilen Stand hat. Der Nutzen eines Capture/Replay-Tools liegt vor allem im Regressionstest. Damit ist es möglich, einmal aufgezeichnete Testabläufe (Testskripte) bei jeder neuen Version der Anwendung wieder ablaufen zu lassen. Das Capture/Replay-Tool zeigt dann alle Veränderungen, die sich durch die neue Version ergeben haben. Es zeigt damit insbesondere die Stellen in der Anwendung, an denen sich durch die neue Version hätte nichts verändern dürfen, tatsächlich aber Abweichungen vorhanden sind.

Arbeitsweise:

- Arbeiten auf der Bedienoberfläche der zu testenden Software.
- Zeichnen während der Testsitzung alle manuell ausgelösten Bedienschritte und die auf dem GUI ausgelösten Operationen auf und speichern die zur Wiedererkennung der Objekte notwendigen Eigenschaften.
- Bedienschritte werden in einem Testskript auf Ausführungsanweisungen der meist proprietären Skriptsprache abgebildet.
- Die so entstehenden Testskripte können z. B. je Testfall abgespeichert werden.
- Während der Aufzeichnung oder bei der Nachbearbeitung können Soll-/Ist-Vergleiche in das Testskript aufgenommen werden.
- Die Testskripte können zu jedem Zeitpunkt wieder abgespielt werden.
- Beim Wiederabspielen des Testskripts werden die Ist-Werte mit den Soll-Werten verglichen. Unterscheiden sich die Werte, schlägt der Test fehl und im Fehlerprotokoll erfolgt ein Eintrag.

Vorwiegende Einsatzbereiche:

- Automatisierung von Regressionstests
- Automatisierung funktionaler Tests
- Last- und Performancetests (Aufzeichnung auf der Protokollebene).

7.7 Unterstützung Fehlermanagement

Anforderungen:

- Das Tool unterstützt die komfortable Bearbeitung eines Fehlers über den gesamten Lebenszyklus (Erfassung, Bearbeitung, Nachtest, Abschluss).
- Es muss möglich sein, im Tool ein Rollen- und Berechtigungskonzept abzubilden (wer darf welche Tätigkeiten durchführen bzw. den Status eines Fehlers verändern).
- Zu einem Fehler müssen entsprechende Attribute erfassbar sein (z.B. Status, Priorität, Beschreibung, Erfasser, Bearbeiter etc.).
- Das Tool sollte komfortable Auswertungsmöglichkeiten (Recherchen, Reports etc). zur Verfügung stellen.