Certified Tester

Advanced Level Syllabus
Technical Test Analyst

Version 2012

International Software Testing Qualifications Board

Deutschsprachige Ausgabe. Herausgegeben durch
Austrian Testing Board, German Testing Board e.V. &
Swiss Testing Board

Teil-Arbeitsgruppe Advanced Level Technical Test Analyst: Graham Bath (Vorsitzender), Paul Jorgensen, Jamie Mitchell; 2010-2012.

Änderungsübersicht

Version	Datum	Bemerkungen
ISEB v1.1	04. September 2001	ISEB Practitioner Syllabus
ISTQB 1.2E	September 2003	ISTQB® Advanced Level Syllabus von EOQ-SG
V2007	12. Oktober 2007	Certified Tester Advanced Level Syllabus Version 2007 (Lehrplan Aufbaukurs Certified Tester)
D100626	26. Juni 2010	Einarbeitung der in 2009 angenommenen Änderungen, Aufteilung der jeweiligen Kapitel für die einzelnen Module
D101227	27. Dezember 2010	Annahme von Formatänderungen und Korrekturen, die keinen Einfluss auf die Bedeutung der Sätze haben
Entwurf V1	17. September 2011	Erste Version des Lehrplans des neuen separaten TTA-Moduls auf Grundlage des vereinbarten Scoping-Dokuments. Review durch AL Arbeitsgruppe
Entwurf V2	20. November 2011	Review-Version für die nationalen Boards
Alpha 2012	09. März 2012	Einarbeitung aller Reviewbemerkungen der nationalen Boards, die zur Version Oktober 2011 eingingen
Beta 2012	07. April 2012	Beta-Version zur Vorlage bei der Hauptversammlung
Beta 2012	08. Juni 2012	Beta-Freigabe für nationale Boards nach technischer Überarbeitung
Beta 2012	27. Juni 2012	Einarbeitung von EWG- und Glossar-Bemerkungen
RC 2012	15. August 2012	Release Candidate – finale Anmerkungen der nationalen Boards eingearbeitet
RC 2012	02. September 2012	Kommentare von BNLTB und Stuart Reid eingearbeitet und von Paul Jorgensen gegengelesen
GA 2012	19. Oktober 2012	Letzte Überarbeitungen und Bereinigungen für das GA Release
Deutsche Überarbeitung		
Entwurf 0.1	18. Dezember 2012	Erstübersetzung mit Anmerkungen zur Glossarkonsistenz von Elke Bath
Entwurf 0.2	07. Januar 2013	Zeilennummern eingefügt von Martin Klonk
Entwurf 0.3	13. Januar 2013	Einarbeitung der Änderungen der englischen Version GA 2012 / 19OCT2012 und Umsetzung der Anmerkungen zur Glossarkonsistenz von Elke Bath, Index übersetzt
Entwurf 0.4	27. Januar 2013	Ergebnisse 1. Review zusammengetragen (noch keine redaktionelle Überarbeitung)
Entwurf 0.5	31. Januar 2013	Ergebnisse 1. Review eingearbeitet, wo offensichtlich, offene Punkte als Kommentare ausgewiesen und mit Excel-Tabelle der Reviewergebnisse synchronisiert.
Entwurf 0.6	02. Februar 2013	Ergebnisse der Redaktionssitzung (Telco) vom 1.2. eingearbeitet.
Entwurf 0.7	18. Februar 2013	Ergebnisse der 2. Reviewrunde eingearbeitet.
Entwurf 0.8	24. Februar 2013	Ersetzungsliste für bestimmte Wörter aus TM-Syllabus-Review eingearbeitet.
Entwurf 0.9	04. März 2013	Überarbeitung aufgrund Redaktionssitzungen (Telco) vom 25.2. und 4.3.
Final German	19. April 2013	Finalisierung

Inhaltsverzeichnis

Dank

Dieses Dokument wurde von einem Kernteam der Teil-Arbeitsgruppe „Advanced Level Syllabus – Technical Test Analyst" des International Software Testing Qualifications Board erstellt. Dieser Arbeitsgruppe gehörten an: Graham Bath (Vorsitzender), Paul Jorgensen, Jamie Mitchell.

Das Kernteam bedankt sich beim Reviewteam und bei den nationalen Boards für die Vorschläge und Beiträge.

Bei Fertigstellung des Advanced Level Lehrplans hatte die Arbeitsgruppe „Advanced Level Syllabus" die folgenden Mitglieder (in alphabetischer Reihenfolge):

Graham Bath, Rex Black, Maria Clara Choucair, Debra Friedenberg, Bernard Homès (stellvertretender Vorsitzender), Paul Jorgensen, Judy McKay, Jamie Mitchell, Thomas Mueller, Klaus Olsen, Kenji Onishi, Meile Posthuma, Eric Riou du Cosquer, Jan Sabak, Hans Schaefer, Mike Smith (Vorsitzender), Geoff Thompson, Erik van Veenendaal, Tsuyoshi Yumoto.

Folgende Personen haben an Review, Kommentierung und der Abstimmung über diesen Lehrplan mitgearbeitet (in alphabetischer Reihenfolge):

Dani Almog, Graham Bath, Franz Dijkman, Erwin Engelsma, Mats Grindal, Dr. Suhaimi Ibrahim, Skule Johansen, Paul Jorgensen, Kari Kakkonen, Eli Margolin, Rik Marselis, Judy McKay, Jamie Mitchell, Reto Mueller, Thomas Müller, Ingvar Nordstrom, Raluca Popescu, Meile Posthuma, Michael Stahl, Chris van Bael, Erik van Veenendaal, Rahul Verma, Paul Weymouth, Hans Weiberg, Wenqiang Zheng, Shaomin Zhu.

Dieses Dokument wurde in der englischen Fassung von der Hauptversammlung des ISTQB® am 19. Oktober 2012 offiziell freigegeben.

0. Einführung in den Advanced Level Syllabus

0.1 Zweck dieses Dokuments

Dieser Lehrplan bildet die Grundlage für das Softwaretest-Qualifizierungsprogramm der Aufbaustufe (Advanced Level) für den Technical Test Analyst. Das ISTQB® stellt den Lehrplan folgenden Adressaten zur Verfügung:

1. Nationalen/regionalen Boards zur Übersetzung in die jeweilige Landessprache und zur Akkreditierung von Ausbildungsanbietern. Die nationalen Boards können den Lehrplan an die eigenen sprachlichen Anforderungen anpassen sowie die Querverweise ändern und an die bei ihnen vorliegenden Veröffentlichungen angleichen.
2. Prüfungsinstitutionen zur Erarbeitung von Prüfungsfragen in der jeweiligen Landessprache, die sich an den Lernzielen der jeweiligen Lehrpläne orientieren.
3. Ausbildungsanbietern zur Erstellung ihrer Kursunterlagen und zur Bestimmung einer geeigneten Unterrichtsmethodik.
4. Prüfungskandidaten zur Vorbereitung auf die Prüfung (als Teil des Ausbildungslehrgangs oder auch kursunabhängig).
5. Allen Personen, die im Bereich Software- und Systementwicklung tätig sind und die professionelle Kompetenz beim Testen von Software und Systemen verbessern möchten, sowie als Grundlage für Bücher und Fachartikel.

Das ISTQB® kann auch anderen Personenkreisen oder Institutionen die Nutzung dieses Lehrplans für andere Zwecke genehmigen, wenn diese vorab eine entsprechende schriftliche Genehmigung einholen.

0.2 Überblick

Der Advanced Level besteht aus drei separaten Lehrplänen:

- Testmanager
- Test Analyst
- Technical Test Analyst

Im Überblicksdokument zum Advanced Level [ISTQB_AL_OVIEW] sind folgende Informationen enthalten:

- Mehrwert für jeden Lehrplan
- Zusammenfassung der einzelnen Lehrpläne
- Beziehung zwischen den Lehrplänen
- Beschreibung der kognitiven Stufen des Wissens
- Anhänge

0.3 Prüfungsrelevante Lernziele

Die Lernziele unterstützen den Mehrwert und dienen zur Ausarbeitung der Prüfung für den Technical Test Analyst Advanced Level (CTAL-TTA). Allgemein gilt, dass der gesamte Inhalt des vorliegenden Advanced Level Lehrplans entsprechend der Lernziele der kognitiven Stufe K1 geprüft werden kann. Dies bedeutet, dass die Prüfungskandidaten Begriffe oder Konzepte erkennen, sich an sie erinnern und sie wiedergeben können. Die relevanten Lernziele der kognitiven Stufen K2 (Verstehen), K3 (Anwenden) und K4 (Analysieren) werden immer zu Beginn des jeweiligen Kapitels angegeben.

0.4 Erwartungen

Einige Lernziele für den Technical Test Analyst setzen grundlegende Erfahrungen in den folgenden Bereichen voraus:

- Allgemeine Programmierungskonzepte
- Allgemeine Systemarchitekturkonzepte

1. Aufgaben des Technical Test Analysten beim risikoorientierten Test - 30 Minuten

Begriffe

Produktrisiko, Risikoanalyse, Risikobeherrschung, Risikobewertung, Risikoidentifizierung, Risikostufe, risikoorientierter Test

Lernziele für die Aufgaben des Technical Test Analysten beim risikoorientierten Test

1.3 Risikobewertung

TTA-1.3.1 (K2) Sie können die allgemeinen Risikofaktoren zusammenfassen, die der Technical Test Analyst berücksichtigen muss

Übergreifende Lernziele

Das nachfolgende Lernziel bezieht sich auf Inhalte, die in mehreren Abschnitten dieses Kapitels behandelt werden.

TTA-1.x.1 (K2) Sie können die Aktivitäten des Technical Test Analysten in Zusammenhang mit Testplanung und -durchführung beim risikoorientierten Test zusammenfassen

1.1 Einführung

Der Testmanager trägt die Gesamtverantwortung für Festlegung und Management einer risikoorientierten Teststrategie. In der Regel wird der Testmanager die Unterstützung des Technical Test Analysten anfordern, um sicherzustellen, dass die risikoorientierte Vorgehensweise korrekt implementiert wird.

Aufgrund ihrer besonderen technischen Kenntnisse sollten Technical Test Analysten an den folgenden risikoorientierten Testaufgaben aktiv mitwirken:

- Risikoidentifikation
- Risikobewertung
- Risikobeherrschung

Diese Aufgaben erfolgen iterativ während des gesamten Projekts und befassen sich mit auftretenden Produktrisiken, mit der Änderung von Prioritäten und mit der regelmäßigen Bewertung und Kommunikation des Risikostatus.

Technical Test Analysten agieren im Rahmen des vom Testmanager für das Projekt festgelegten risikoorientierten Testens. Sie steuern ihr Wissen über die technischen Risiken des Projekts bei, wie beispielsweise Risiken in Zusammenhang mit der Sicherheit, Zuverlässigkeit und Performanz des Systems.

1.2 Risikoidentifizierung

Je breiter die Basis der Stakeholder im Risikoidentifizierungsprozess ist, desto höher ist die Wahrscheinlichkeit, dass dabei die größtmögliche Menge an wesentlichen Risiken aufgedeckt wird. Da die Technical Test Analysten häufig über spezifische technische Fähigkeiten verfügen, sind sie in besonderer Weise dazu geeignet, Experten-Interviews zu führen, Brainstorming mit Kollegen durchzuführen, sowie aktuelle als auch vergangene Erfahrungen zu analysieren, um die Bereiche für mögliche Produktrisiken zu bestimmen. Technical Test Analysten arbeiten insbesondere mit ihren Fachkollegen des technischen Bereichs (z.B. Entwickler, Systemarchitekten, Betriebsingenieure) eng zusammen, um die technischen Risiken zu bestimmen.

Zu den Risiken, die bei einem Projekt identifiziert werden könnten, gehören beispielsweise:

- Performanzrisiken (z.B. Antwortzeiten werden bei hoher Last nicht erreicht)
- Sicherheitsrisiken (z.B. Offenlegung vertraulicher Daten durch Sicherheitsangriffe)
- Zuverlässigkeitsrisiken (z.B. Anwendung erfüllt nicht die in der Service Level-Vereinbarung spezifizierte Verfügbarkeit)

Spezifische Risikotypen der einzelnen Softwarequalitätsmerkmale werden in den jeweiligen Kapiteln dieses Lehrplans behandelt.

1.3 Risikobewertung

Während es bei der Risikoidentifikation darum geht, möglichst viele vorhandene Risiken zu identifizieren, befasst sich die Risikobewertung mit der Untersuchung der identifizierten Risiken. Dabei werden die einzelnen Risiken kategorisiert und jeweils die Eintrittswahrscheinlichkeit und das Schadensausmaß bestimmt.

Für die Festlegung der Risikostufe werden für jedes einzelne Risiko die Eintrittswahrscheinlichkeit und die Auswirkung eingeschätzt. Die Wahrscheinlichkeit des Auftretens bezeichnet oft die Wahrscheinlichkeit, dass das potenzielle Problem im getesteten System vorhanden ist.

Der Technical Test Analyst ist involviert, wenn es darum geht, die Risikostufe der einzelnen technischen Risiken festzulegen und zu verstehen. Der Test Analyst trägt dazu bei, die potenziellen betriebswirtschaftlichen Auswirkungen des Problems zu verstehen, falls dieses auftritt.

Folgende allgemeine Faktoren müssen bei der Risikobewertung normalerweise berücksichtigt werden:

- Komplexität der Technologie
- Komplexität der Codestruktur
- Konflikte zwischen Stakeholdern hinsichtlich der technischen Anforderungen
- Kommunikationsprobleme aufgrund der räumlichen Verteilung der Entwicklungsorganisation
- Werkzeuge und Technologie
- Zeitdruck, knappe Ressourcen, Druck durch das Management
- Fehlen einer „frühzeitigen“ Qualitätssicherung
- häufige Änderungen bei technischen Anforderungen
- hohe Fehlerraten in Zusammenhang mit technischen Qualitätsmerkmalen
- technische Schnittstellen-/Integrationsproblematik

Anhand der vorhandenen Informationen über das Risiko, muss der Technical Test Analyst die Risikostufen für technische Risiken basierend auf den vom Testmanager vorgegebenen Richtlinien festlegen. Der Testmanager gibt beispielsweise vor, Risiken mit einem Wert zwischen 1 und 10 zu kategorisieren (wobei die Risikokategorie 1 für das größte Risiko steht).

1.4 Risikobeherrschung

Während des Projekts beeinflussen Technical Test Analysten wie das Testen auf die identifizierten Risiken reagiert. Dazu gehört im Allgemeinen:

- Reduzieren der Risiken durch die Ausführung der wichtigsten Tests zurerst und durch die Umsetzung der Maßnahmen zur Risikobeherrschung und Vorkehrung gegen Risiken, die in der Teststrategie und im Testkonzept festgelegt sind.
- Bewerten der Risiken basierend auf zusätzlichen Informationen und Erkenntnissen, die im Projektverlauf gewonnen wurden, sowie die Nutzung dieser Informationen für die Umsetzung von Maßnahmen zur Risikobeherrschung, die die Eintrittswahrscheinlichkeit oder das Schadensausmaß von bereits identifizierten und analysierten Risiken reduzieren.

2. Strukturbasierter Test - 225 Minuten

Begriffe

Anweisungstest, atomare Bedingung, Bedingungs-/Entscheidungstest, Bedingungstest, Kontrollflusstest, Mehrfachbedingungstest, Pfadtest, strukturbasiertes Verfahren, verkürzte Auswertungt

Lernziele für den strukturbasierten Test

2.2 Einfacher Bedingungstest

TTA-2.2.1 (K2) Sie können verstehen, wie eine Bedingungsüberdeckung erzielt wird und weshalb diese Art des Testens weniger gründlich sein kann als die Entscheidungsüberdeckung

2.3 Bedingungs-/Entscheidungstest

TTA-2.3.1 (K3) Sie können den Bedingungs-/Entscheidungstest anwenden, um Testfälle zu erstellen, die einen definierten Überdeckungsgrad erzielen

2.4 Modifizierter Bedingungs-/Entscheidungstest

TTA-2.4.1 (K3) Sie können den modifizierten Bedingungs-/Entscheidungstest[1] anwenden, um Testfälle zu erstellen, die einen definierten Überdeckungsgrad erzielen

2.5 Mehrfachbedingungstest

TTA-2.5.1 (K3) Sie können den Mehrfachbedingungstest anwenden, um Testfälle zu erstellen, die einen definierten Überdeckungsgrad erzielen

2.6 Pfadtest

TTA-2.6.1 (K3) Sie können Pfadtest anwenden, um Testfälle zu erstellen

2.7 API-Test

TTA-2.7.1 (K2) Sie können die Anwendbarkeit des API-Tests und die damit gefundenen Fehlerzustände verstehen

2.8 Strukturbasierte Verfahren auswählen

TTA-2.8.1 (K4) Sie können für eine vorgegebene Projektsituation ein geeignetes strukturbasiertes Testentwurfsverfahren auswählen

[1] (engl. Modified Condition / Decision Coverage, abgekürzt MC/DC)

2.1 Einführung

In diesem Kapitel werden hauptsächlich strukturbasierte Testentwurfsverfahren beschrieben, die auch als White-Box- oder codebasierte Testverfahren bezeichnet werden. Dabei werden Code, Daten, Architektur und/oder Kontroll- und Datenfluss als Grundlage für den Testentwurf verwendet. Jedes der strukturbasierten Verfahren leitet die Testfälle systematisch aus der Struktur ab und befasst sich gezielt mit besonderen Elementen der betrachteten Struktur. Die Verfahren liefern Kriterien für den Überdeckungsgrad; dieser muss gemessen und mit dem für das Projekt oder Unternehmen definierten Ziel verglichen werden. Das Erreichen eines Überdeckungsgrades von 100% heißt nicht, dass die Menge von Tests vollständig ist. Es bedeutet vielmehr, dass die untersuchte Struktur für das vorliegende Testverfahren keine weiteren nützlichen Tests bietet.

Mit Ausnahme des einfachen Bedingungstests sind die im vorliegenden Lehrplan beschriebenen strukturbasierten Testentwurfsverfahren gründlicher als die Testverfahren zur Anweisungs- und Entscheidungsüberdeckung, die im Foundation Level Lehrplan behandelt werden [ISTQB_FL_SYL].

Im vorliegenden Lehrplan werden die folgenden Testentwurfsverfahren behandelt:

- Einfacher Bedingungstest
- Bedingungs-/Entscheidungstest
- Modifizierter Bedingungs-/Entscheidungstest (MC/DC)
- Mehrfachbedingungstest
- Pfadtest
- API-Test

Die ersten vier der aufgelisteten Testentwurfsverfahren basieren auf WAHR/FALSCH-Bedingungen und decken weitgehend die gleichen Arten von Fehlerzuständen auf. Ganz gleich wie komplex eine Entscheidung auch sein mag, kann sie letztlich als WAHR oder FALSCH bewertet werden, und somit wird ein Pfad durch den Code verfolgt und ein anderer nicht. Weil eine komplexe Entscheidung nicht wie erwartet getroffen wird, zeigt sich ein Fehlerzustand dadurch, dass der beabsichtigte Pfad nicht durchlaufen wird,

Generell werden die ersten vier Testentwurfsverfahren zunehmend gründlicher; es müssen immer mehr Testfälle erstellt werden, um die beabsichtigte Überdeckung zu erzielen, und um immer subtilere Exemplare der jeweiligen Fehlerart aufzudecken.

Siehe auch [Bath08], [Beizer90], [Beizer95], [Copeland03] und [Koomen06].

2.2 Einfacher Bedingungstest

Im Vergleich zum Entscheidungstest (oder Zweigtest), bei dem die Entscheidung als Ganzes betrachtet und das Ergebnis in separaten Testfällen als WAHR oder FALSCH bewertet wird, befasst sich der einfache Bedingungstest damit, wie die Entscheidung getroffen wird. Jede Entscheidung besteht aus einer oder aus mehreren atomaren Bedingungen, die jeweils zu einem booleschen Wert ausgewertet werden. Die logische Kombination dieser Werte ist ausschlaggebend für die endgültige Entscheidung. Die Testfälle müssen jede einzelne atomare Teilbedingung in beide Richtungen (WAHR und FALSCH) bewerten, um diesen Überdeckungsgrad zu erzielen.

Anwendbarkeit
Der einfache Bedingungstest ist wegen der unten beschriebenen Schwierigkeiten wahrscheinlich nur in abstrakter Hinsicht interessant. Nichtsdestotrotz muss man dieses Testverfahren erst einmal

verstanden haben, um die darauf aufbauenden Testverfahren zu verstehen, die eine höhere Überdeckung erreichen.

Einschränkungen/Schwierigkeiten
Wenn in einer Entscheidung zwei oder mehrere atomare Teilbedingungen enthalten sind, dann ist es möglich, dass durch beim Testentwurf unklug ausgewählte Testdaten die Bedingungsüberdeckung, nicht jedoch die Entscheidungsüberdeckung, erreicht wird. Beispiel: Für die Entscheidung „A und B":

	A	B	A und B
Test 1	FALSCH	WAHR	FALSCH
Test 2	WAHR	FALSCH	FALSCH

Um 100% Bedingungsüberdeckung zu erzielen, werden die beiden Tests (siehe Tabelle) durchgeführt. Obwohl diese beiden Tests 100% Bedingungsüberdeckung erzielen, erzielen sie keine Entscheidungsüberdeckung, da die Entscheidungsausgänge in beiden Fällen als „FALSCH" bewertet sind.

Wenn Entscheidungen aus nur einer einzigen atomaren Teilbedingung bestehen, sind der einfache Bedingungstest und der Entscheidungstest identisch.

2.3 Bedingungs-/Entscheidungstest

Der Bedingungs-/Entscheidungstest spezifiziert, dass sowohl die Bedingungsüberdeckung (siehe oben) als auch die Entscheidungsüberdeckung (siehe Foundation Level Lehrplan [ISTQB_FL_SYL]) erreicht werden muss. Durch klug ausgewählte Testdatenwerte für die atomaren Teilbedingungen lässt sich diese Überdeckung erzielen, ohne dass zusätzlich zu den für die Bedingungsüberdeckung benötigten Testfällen noch weitere benötigt werden.

Im Beispiel unten wird dieselbe Entscheidung „A und B" getestet wie im oben beschriebenen Beispiel. Bedingungs-/Entscheidungsüberdeckung kann mit der gleichen Anzahl von Tests erzielt werden, wenn unterschiedliche Testwerte gewählt werden.

	A	B	A und B
Test 1	WAHR	WAHR	WAHR
Test 2	FALSCH	FALSCH	FALSCH

Dieses Verfahren kann daher den Vorteil einer höheren Effizienz bieten.

Anwendbarkeit
Diese Überdeckung sollte dann in Betracht gezogen werden, wenn der getestete Code zwar wichtig, aber nicht kritisch ist.

Einschränkungen/Schwierigkeiten
Da für dieses Verfahren möglicherweise mehr Testfälle benötigt werden als für die Entscheidungsüberdeckung, kann es problematisch werden, wenn die Zeit knapp ist.

2.4 Modifizierter Bedingungs-/Entscheidungstest

Dieses Verfahren liefert eine höhere Kontrollflussüberdeckung. Wenn N einzelne atomare Teilbedingungen vorliegen, dann lässt sich die modifizierte Bedingungs-/Entscheidungsüberdeckung normalerweise mit N+1 einzelnen Testfällen erzielen. Der modifizierte Bedingungs-/Entscheidungstest

erzielt Bedingungs-/Entscheidungsüberdeckung; allerdings müssen dabei auch folgende Punkte erfüllt sein:

1. Mindestens ein Test, bei dem sich der Entscheidungsausgang ändern würde, wenn die atomare Teilbedingung X WAHR wäre
2. Mindestens ein Test, bei dem sich der Entscheidungsausgang ändern würde, wenn die atomare Teilbedingung X FALSCH wäre
3. Für jede einzelne atomare Teilbedingung gibt es Tests, die Punkt 1 und 2 erfüllen

	A	B	C	(A oder B) und C
Test 1	WAHR	FALSCH	WAHR	WAHR
Test 2	FALSCH	WAHR	WAHR	WAHR
Test 3	FALSCH	FALSCH	WAHR	FALSCH
Test 4	WAHR	FALSCH	FALSCH	FALSCH

Im Beispiel oben ist sowohl die Entscheidungsüberdeckung (der Entscheidungsausgang ist sowohl WAHR als auch FALSCH) als auch die Bedingungsüberdeckung erzielt (A, B und C sind alle sowohl WAHR als auch FALSCH).

In Test 1 ist A WAHR und der Ausgang ist WAHR. Wenn A FALSCH wird (wie in Test 3; andere Werte bleiben unverändert), wird das Ergebnis FALSCH.
In Test 2 ist B WAHR und der Ausgang ist WAHR. Wenn B FALSCH wird (wie in Test 3; andere Werte bleiben unverändert), wird das Ergebnis FALSCH.
In Test 1 ist C WAHR und der Ausgang ist WAHR. Wenn C FALSCH wird (wie in Test 4; andere Werte bleiben unverändert), wird das Ergebnis FALSCH.

Anwendbarkeit
Dieses Verfahren ist in der Softwareentwicklung für die Luft- und Raumfahrtindustrie sowie für andere sicherheitskritische Systeme weit verbreitet. Es sollte für sicherheitskritische Software eingesetzt werden, wo ein Fehler eine Katastrophe auslösen könnte.

Einschränkungen/Schwierigkeiten
Es kann kompliziert sein, die modifizierte Bedingungs-/Entscheidungsüberdeckung zu erzielen, wenn ein spezifisches Element in einem Ausdruck mehrfach vorkommt; in solchen Fällen spricht man von einem „gekoppelten" Element. Je nach Entscheidungsanweisung im Code ist es eventuell nicht möglich, den Wert des gekoppelten Elements so zu variieren, dass dies allein zu einer Änderung des Entscheidungsausgangs führt. Ein möglicher Ansatz für den Umgang mit diesem Problem ist, dass nur atomare Teilbedingungen, die keine solche Kopplungen enthalten, auf Ebene der modifizierten Bedingungs-/Entscheidungsüberdeckung getestet werden müssen. Ein anderer Ansatz ist, dass jede Entscheidung mit gekoppelten Elementen von Fall zu Fall analysiert wird.

Einige Programmiersprachen und/oder Interpreter wurden so entworfen, dass sie bei komplexen Entscheidungen ein verkürztes Auswertungsverhalten aufweisen. Dies bedeutet, dass der ausführende Code nicht den gesamenten Ausdruck auswertet, wenn das Endergebnis der Bewertung durch einen Teil des Ausdrucks bewertet werden kann. Beispiel: Wenn die Entscheidung „A und B" bewertet werden soll, dann gibt es keinen Grund, B zu bewerten, wenn A als FALSCH bewertet wird. Kein Wert von B kann den endgültigen Wert ändern; daher kann Ausführungszeit eingespart werden, wenn B nicht bewertet werden muss. Die verkürzte Auswertung kann die Erzielung der modifizierten Bedingungs-/Entscheidungsüberdeckung beeinträchtigen, da manche der dafür benötigten Tests nicht ausgeführt werden können.

2.5 Mehrfachbedingungstest

In seltenen Fällen kann es erforderlich sein, dass alle möglichen Kombinationen von Wahr/Falsch-Bedingungen einer Entscheidung getestet werden müssen. Dieses erschöpfende Testen wird als Mehrfachbedingungsüberdeckung bezeichnet. Die Anzahl der benötigten Tests ist abhängig von der Anzahl der atomaren Teilbedingungen der Entscheidungsanweisung und lässt sich mit 2^n berechnen, wobei n die Anzahl ungekoppelter atomarer Teilbedingungen ist. Im bereits verwendeten Beispiel werden folgende Tests benötigt, um Mehrfachbedingungsüberdeckung zu erzielen:

	A	B	C	(A oder B) und C
Test 1	WAHR	WAHR	WAHR	WAHR
Test 2	WAHR	WAHR	FALSCH	FALSCH
Test 3	WAHR	FALSCH	WAHR	WAHR
Test 4	WAHR	FALSCH	FALSCH	FALSCH
Test 5	FALSCH	WAHR	WAHR	WAHR
Test 6	FALSCH	WAHR	FALSCH	FALSCH
Test 7	FALSCH	FALSCH	WAHR	FALSCH
Test 8	FALSCH	FALSCH	FALSCH	FALSCH

Falls die Programmiersprache die verkürzte Auswertung verwendet, verringert sich dadurch häufig die Anzahl der ausgeführten Tests, je nach Reihenfolge und Gruppierung der logischen Operationen, die unter den jeweiligen atomaren Teilbedingungen durchgeführt werden.

Anwendbarkeit
Dieses Testverfahren wurde traditionell zum Testen eingebetteter Software verwendet, die zuverlässig ohne Systemabstürze über lange Zeiträume laufen sollte (z.B. Telefon Switches mit einer erwarteten Lebensdauer von 30 Jahren). Dieses Testverfahren wird wahrscheinlich bei den meisten kritischen Anwendungen durch den modifizierten Bedingungs-/Entscheidungstest ersetzt.

Einschränkungen/Schwierigkeiten
Da die Anzahl der Testfälle direkt aus der Wahrheitswertetabelle (auch Wahrheitstafel oder Wahrheitsmatrix genannt) mit allen atomaren Teilbedingungen abgeleitet werden kann, lässt sich der Überdeckungsgrad leicht bestimmen. Allerdings ist angesichts der vielen benötigten Testfälle die modifizierte Bedingungs-/Entscheidungsüberdeckung in den meisten Situationen geeigneter.

2.6 Pfadtest

Beim Pfadtest werden mögliche Pfade im Code identifiziert und dann Testfälle entworfen, die diese Pfade abdecken. Grundsätzlich wäre es nützlich, jeden einzelnen Pfad durch das System zu testen. In jedem nichttrivialen System könnte so jedoch die Anzahl der Testfälle aufgrund der im Code vorhandenen Schleifen übertrieben groß werden.

Wenn man das Problem von unbegrenzten Schleifendurchläufen einmal außen vor lässt, ist die Durchführung von Pfadtests jedoch durchaus realistisch. Bei Anwendung dieses Verfahrens empfiehlt Beizer [Beizer90], Testfälle derart zu erstellen, dass möglichst viele Pfade durch das Softwaremodul von Anfang bis Ende durchlaufen werden. Um diese möglicherweise komplexe Aufgabe zu vereinfachen, empfiehlt er die folgende systematische Vorgehensweise:

1. Als ersten Pfad den einfachsten, funktional sinnvollen Pfad von Anfang bis Ende auswählen.
2. Jeden zusätzlichen Pfad als leichte Variation des vorherigen Pfades auswählen. Für jeden nachfolgenden Test sollte jeweils möglichst nur ein Zweig im Pfad verändert werden. Kurze Pfade sind gegenüber langen Pfaden nach Möglichkeit zu bevorzugen. Funktional sinnvolle Pfade sind gegenüber funktional sinnlosen Pfaden zu bevorzugen.
3. Funktional sinnlose Pfade nur wählen, wenn dies für die Überdeckung erforderlich ist. Beizer weist bei dieser Regel darauf hin, dass solche Pfade irrelevant sein können und hinterfragt werden sollten.
4. Bei der Auswahl der Pfade intuitiv vorgehen (d.h. die Pfade wählen, die am wahrscheinlichsten ausgeführt werden).

Es ist zu beachten, dass bei dieser Strategie manche Pfadsegmente wahrscheinlich mehrfach ausgeführt werden. Der springende Punkt bei dieser Strategie ist, dass jedes Codesegment mindestens einmal ausgeführt wird, möglichst sogar mehrfach.

Anwendbarkeit
Der partielle Pfadtest – so wie oben dargestellt – wird häufig für sicherheitskritische Softwareanwendungen durchgeführt. Er ist eine gute Ergänzung zu den anderen, in diesem Kapitel behandelten Methoden, weil hier Pfade durch die Software betrachtet werden und nicht nur einzelne Entscheidungen.

Einschränkungen/Schwierigkeiten
Obwohl ein Kontrollflussgraph verwendet werden kann, um die Pfade zu bestimmen, wird in der Realität eher ein Werkzeug benötigt, um die Pfade von komplexen Modulen zu berechnen.

Überdeckung
Die Erstellung ausreichend vieler Tests zur Überdeckung aller Pfade (ohne Berücksichtigung von Schleifen) garantiert auch die Erfüllung von Anweisungs- und Zweigüberdeckung. Der partielle Pfadtest liefert gründlicheres Testen als der Zweigtest bei einer nur geringfügig erhöhten Anzahl von Tests. [NIST 96]

2.7 API-Test

Die API (= Application Programming Interface bzw. Programmierschnittstelle) ist der Code, der die Kommunikation zwischen verschiedenen Prozessen, Programmen und/oder Systemen ermöglicht. APIs werden häufig in Client/Server-Systemen verwendet, bei denen ein Prozess anderen Prozessen Funktionalität zur Verfügung stellt.

In gewisser Hinsicht ist der API-Test dem Testen von grafischen Benutzerschnittstellen (GUI) recht ähnlich. Im Mittelpunkt steht die Bewertung von Eingabewerten und zurückgelieferten Daten.

Beim Umgang mit APIs sind Negativtests häufig von entscheidender Bedeutung. Es ist möglich, dass Programmierer, die APIs für den Zugriff auf externe Services außerhalb ihres eigenen Codes verwenden, versuchen, diese Programmierschnittstellen in einer nicht vorgesehenen Art und Weise einzusetzen. Dies bedeutet, dass eine robuste Fehlerbehandlung unbedingt erforderlich ist, um einen fehlerhaften Betrieb zu vermeiden. Möglicherweise ist das kombinatorische Testen vieler verschiedener Schnittstellen notwendig, weil APIs häufig in Verbindung mit anderen APIs verwendet werden, und weil eine einzelne Schnittstelle mehrere Parameter beinhalten kann, deren Werte in vielfacher Weise kombiniert werden können.

APIs sind häufig lose miteinander gekoppelt, was zu dem sehr realen Problem von verlorenen Transaktionen und Timing-Fehlern führen kann. Daher ist gründliches Testen der Wiederherstellungs- und Wiederholungsmechanismen notwendig. Unternehmen, die APIs bereitstellen, müssen

sicherstellen, dass alle Services eine sehr hohe Verfügbarkeit haben; dafür sind häufig gründliche Zuverlässigkeitstests seitens des API-Anbieters sowie Support für die Infrastruktur notwendig.

Anwendbarkeit
Der API-Test gewinnt zunehmend an Bedeutung, da mehr Systeme verteilt arbeiten oder Remote-Bearbeitung einsetzen, um einen Teil der Aufgaben an andere Prozessoren abzugeben. Beispiele dafür sind Betriebssystemaufrufe, service-orientierte Architekturen (SOA), Programmfernaufrufe (engl. Remote Procedure Calls, abgekürzt RPC), Webservices sowie nahezu alle anderen verteilten Anwendungen. Der API-Test ist besonders für das Testen von Multisystemen geeignet.

Einschränkungen/Schwierigkeiten
Um eine API direkt zu testen, benötigt der Technical Test Analyst normalerweise spezialisierte Werkzeuge. Da normalerweise keine grafische Schnittstelle direkt mit der API verbunden ist, könnten Werkzeuge notwendig sein, um eine Anfangsumgebung aufzusetzen, das Daten-Marshalling durchzuführen, die API-Aufrufe auszuführen und die Ergebnisse zu bestimmen.

Überdeckung
Der API-Test beschreibt eine Art des Testens; er bezeichnet keinen bestimmten Überdeckungsgrad. Beim API-Test sollten zumindest sowohl alle Aufrufe der API ausgeführt, als auch alle gültigen sowie alle angemessenen ungültigen Werte genutzt werden.

Fehlerarten
Die Fehlerzustände, die beim API-Test aufgedeckt werden können, sind sehr unterschiedlicher Art. Häufig geht es um Schnittstellenprobleme, sowie um Probleme mit der Datenhandling oder dem Timing und mit dem Verlust oder der Duplizierung von Transaktionen.

2.8 Strukturbasierte Verfahren auswählen

Der Kontext des zu testenden Systems ist ausschlaggebend für den Überdeckungsgrad, der im strukturbasierten Test erzielt werden sollte. Je kritischer ein System ist, desto höher ist der benötigte Überdeckungsgrad. Im Allgemeinen gilt, dass je höher der benötigte Überdeckungsgrad ist, desto mehr Zeit und Ressourcen werden benötigt, um diesen Überdeckungsgrad zu erzielen.

Manchmal lässt sich der benötigte Überdeckungsgrad aus den Standards ableiten, die für das Softwaresystem gelten. Wenn Softwaresysteme beispielsweise für einen Avionikkontext vorgesehen sind, müssen sie möglicherweise dem branchenspezifischen Standard DO-178B[2] (bzw. ED-12B in Europa) entsprechen. Dieser Standard beinhaltet die folgenden fünf Stufen der Fehlerkritikalität:

A. Katastrophal: Das Versagen verursacht das Fehlen einer kritischen Funktionalität, die für einen sicheren Flug oder eine sichere Landung benötigt wird
B. Gefährlich: Das Versagen hat eine große negative Auswirkung auf Sicherheit oder Performanz
C. Bedeutend: Das Versagen ist bedeutend, aber nicht so schwerwiegend wie A oder B
D. Geringfügig: Das Versagen ist wahrnehmbar, hat aber weniger schwerwiegende Auswirkungen als C
E. Keine Auswirkung: Das Versagen hat keine Auswirkung auf die Sicherheit

Wenn das Softwaresystem zur Stufe A gehört, dann muss die modifizierte Bedingungs-/Entscheidungsüberdeckung (MC/DC) erfüllt werden. Für Stufe B ist Entscheidungsüberdeckung vorgeschrieben, modifizierte Bedingungs-/Entscheidungsüberdeckung ist optional. Die Stufe C erfordert mindestens die Erfüllung der Anweisungsüberdeckung.

[2] Seit 2011 gilt bereits die Norm DO-178C

Auch die IEC-61508 ist ein internationaler Standard, bei dem es um die funktionale Sicherheit von programmierbaren, elektronischen und sicherheitsbezogenen Systemen geht. Dieser Standard wurde in vielen verschiedenen Branchen angepasst, z.B. für die Automobilindustrie, für die Eisenbahnbranche, für Produktionsprozesse, für Kernkraftwerke und für den Maschinenbau. Die Kritikalität wird in Sicherheitsanforderungsstufen (engl. Safety Integrity Level oder kurz SIL) definiert (diese sind gestaffelt von 1 = geringste Kritikalität bis 4 = höchste Kritikalität). Für die einzelnen Sicherheitsanforderungsstufen werden folgende Überdeckungen empfohlen:

1. Anweisungs- und Zweigüberdeckung empfohlen
2. Anweisungsüberdeckung sehr empfohlen, Zweigüberdeckung empfohlen
3. Anweisungs- und Zweigüberdeckung sehr empfohlen
4. Modifizierte Bedingungs-/Entscheidungsüberdeckung (MC/DC) sehr empfohlen

Bei modernen Systemen ist es selten, dass alle Verarbeitungsprozesse auf einem einzigen System ausgeführt werden. Der API-Test sollte immer eingesetzt werden, wenn ein Teil der Verarbeitung remote auf einem anderen System erfolgt. Die Kritikalität des Systems sollte den Aufwand bestimmen, der in den API-Test investiert wird.

Bei der Wahl der strukturbasierten Testverfahren sollte sich der Technical Test Analyst vom Kontext des zu testenden Softwaresystems leiten lassen.

3. Analytische Testverfahren - 255 Minuten

Begriffe

Datenflussanalyse, Definition-Verwendungspaare, dynamische Analyse, Kontrollflussanalyse, Umgebungsintegrationstest, paarweiser Integrationstest, Speicherleck, statische Analyse, wilder Zeiger, zyklomatische Komplexität, Datenflussanomalie, Aufrufgraph

Lernziele für analytische Testverfahren

3.2 Statische Analyse

TTA-3.2.1 (K3) Sie können die Kontrollflussanalyse anwenden, um zu ermitteln, ob der Programmcode Anomalien im Kontrollfluss aufweist

TTA-3.2.2 (K3) Sie können die Datenflussanalyse anwenden, um zu ermitteln, ob der Programmcode eine Datenflussanomalie hat

TTA-3.2.3 (K3) Sie können Möglichkeiten vorschlagen, wie die Wartbarkeit/Änderbarkeit von Code durch statische Analyse verbessert werden kann

TTA-3.2.4 (K2) Sie können den Einsatz von Aufrufgraphen für die Bestimmung von Teststrategien für den Integrationstest erklären

3.3 Dynamische Analyse

TTA-3.3.1 (K3) Sie können die Ziele spezifizieren, die durch dynamische Analyse erreicht werden sollen

3.1 Einführung

Es gibt zwei Arten von Analysen: die statische Analyse und die dynamische Analyse.

Die statische Analyse (siehe Abschnitt 3.2) umfasst das analytische Testen der Software ohne Ausführung. Da die Software nicht ausgeführt wird erfolgt die Untersuchung durch ein Werkzeug oder eine Person, um zu bestimmen, ob die Software bei der Ausführung korrekt funktionieren wird. Die statische Betrachtungsweise der Software ermöglicht eine detaillierte Analyse ohne dass die Daten und Vorbedingungen für die Ausführung des Szenarios erstellt werden müssen.

Anmerkung: Die verschiedenen für den Technical Test Analyst relevanten Reviewarten werden in Kapitel 5 behandelt.

Bei der dynamischen Analyse (siehe Abschnitt 3.3) muss der Programmcode ausgeführt werden. Durch sie werden diejenigen Programmierfehler aufgedeckt die durch die Ausführung des Codes leichter zu finden sind (z.B. Speicherlecks). Wie auch die statische Analyse kann die dynamische Analyse durch Werkzeuge oder durch eine individuelle Überwachung erfolgen, die während der Ausführung auf bestimmte Indikatoren achtet (wie z.B. rasant wachsenden Speicherbedarf).

3.2 Statische Analyse

Ziel der statischen Analyse ist es, tatsächliche oder potenzielle Fehlerzustände im Programmcode und in der Systemarchitektur zu finden und die Wartbarkeit/Änderbarkeit des Codes zu verbessern. Die statische Analyse wird im Allgemeinen durch Werkzeuge unterstützt.

3.2.1 Kontrollflussanalyse

Die Kontrollflussanalyse ist ein statisches Analyseverfahren bei dem der Kontrollfluss eines Softwareprogramms, entweder mit Hilfe eines Kontrollflussgraphen oder mit Werkzeugunterstützung, analysiert wird. Es gibt eine Reihe von Anomalien im System, die durch dieses Verfahren gefunden werden können. Dazu gehören: schlecht konzipierte Schleifen (z.B. mit mehreren Eingangspunkten), unklare/inkorrekt deklarierte Ziele von Funktionsaufrufen in bestimmten Sprachen (z.B. Scheme), inkorrekte Ablaufsequenzen usw.

Häufig wird die Kontrollflussanalyse verwendet um die zyklomatische Komplexität zu bestimmen. Dabei handelt es sich um eine positive ganze Zahl, die die Anzahl unabhängiger Kontrollflusspfade in einem stark zusammenhängenden Kontrollflussgraphen angibt (Schleifen und Iterationen werden nicht mehr berücksichtigt nachdem sie einmal durchlaufen wurden). Jeder unabhängige Pfad von Anfang bis Ende stellt einen einzigartigen Pfad durch das Softwareprogramm dar und jeder dieser Pfade sollte getestet werden.

Die zyklomatische Zahl ist eine Metrik die allgemein dazu dient, die Gesamtkomplexität eines Softwaremoduls zu beziffern. Nach der Theorie von Thomas McCabe [McCabe 76] gilt, dass je komplexer ein System ist, desto schwieriger ist dessen Wartung und desto mehr Fehlerzustände sind darin enthalten. Im Laufe der Zeit wurde diese Korrelation zwischen Komplexität und der Anzahl der darin enthaltenen Fehlerzustände in vielen Studien festgestellt. Das National Institute of Standards and Technology (NIST, (US-Bundesbehörde)) empfiehlt eine maximale zyklomatische Zahl von 10. Module, für die eine höhere Komplexität ermittelt wurde, sollten in mehrere Module unterteilt werden.

3.2.2 Datenflussanalyse

Die Datenflussanalyse umfasst eine Vielzahl von Techniken, um die Informationen über die Verwendung von Variablen im System zu sammeln. Dabei wird der Lebenszyklus von Variablen

genau geprüft (d.h. wo sie deklariert, definiert, gelesen, bewertet und zerstört werden), da prinzipiell bei jedem dieser Vorgänge Anomalien auftreten können.

Ein verbreitetes Verfahren wird als Define-Use (Definieren-Verwenden) Notation bezeichnet, wobei der Lebenszyklus jeder Variablen in drei bestimmte Aktionen unterteilt wird:

- d: wenn die Variable definiert oder initialisiert wird (d: defined)
- u: wenn die Variable verwendet oder für eine Berechnung oder Entscheidung gelesen wird (u: used)
- k: wenn die Variable gelöscht oder zerstört wird oder nicht mehr erreichbar ist (k: killed)

Diese drei bestimmten Aktionen werden zu Paaren zusammengefasst („Definition-Verwendungspaar"), um die Datenflusspfade darzustellen. Beispiel: Ein „du-Pfad" stellt einen Teil des Programmcodes dar, in dem die Variable definiert und anschließend verwendet wird.

Zu den möglichen Datenflussanomalien gehören die Durchführung der korrekten Aktion der Variablen zum falschen Zeitpunkt oder die Datenverwendung einer Variablen bei einer inkorrekten Aktion. Mögliche Anomalien sind:

- Zuweisung eines ungültigen Wertes zu einer Variablen
- Verwendung einer Variablen ohne dass ihr zuvor ein Wert zugewiesen wurde
- Inkorrekte Pfade aufgrund eines inkorrekten Werts in einer Kontrollfluss-Entscheidung
- Versuch, eine Variable zu verwenden nachdem diese zerstört wurde
- Referenzierung von Variablen wenn diese nicht mehr erreichbar sind
- Deklarieren und Zerstören von Variablen ohne diese zu verwenden
- Variablen erneut definieren bevor diese verwendet wurden
- Nicht-Löschung einer dynamisch zugewiesenen Variablen (Ursache für mögliche Speicherlecks)
- Ändern einer Variablen, das zu unerwarteten Nebenwirkungen führt (z.B. Nachwirkungen und Folgeprobleme nachdem eine globale Variable geändert wurde, ohne alle Verwendungen dieser Variablen zu bedenken)

Die Programmiersprache kann die bei der Datenflussanalyse verwendeten Regeln beeinflussen. Programmiersprachen können dem Programmierer bestimmte Aktionen mit Variablen zulassen, die aber ein Systemverhalten in bestimmten Situationen verursachen können, dass anders ist als vom Programmierer erwartet. Beispiel: Eine Variable ist vielleicht zweimal definiert ohne dass sie tatsächlich verwendet wird, wenn ein bestimmter Pfad verfolgt wird. Solche Verwendungen werden bei der Datenflussanalyse häufig als „verdächtig" bezeichnet. Obwohl diese Verwendung der Variablenzuweisung zulässig ist kann dies später zu Wartungsproblemen des Programmcodes führen.

Der Datenflusstest „verwendet den Kontrollflussgraphen, um die unplausiblen Dinge zu erforschen, die mit Daten passieren können" [Beizer90]. Folglich werden dabei auch andere Fehlerzustände gefunden als beim Kontrollflusstest. Der Technical Test Analyst sollte dieses Verfahren bei der Testplanung berücksichtigen da viele dieser Fehlerzustände zeitweise Ausfälle verursachen, die durch dynamisches Testen schwierig zu finden sind.

Die Datenflussanalyse ist ein statischer Test. Probleme in Zusammenhang mit den Daten im Laufzeitsystem können mit diesem Verfahren übersehen werden. Beispiel: Eine statische Variable enthält einen Zeiger auf ein dynamisch erzeugtes Array, das erst zur Laufzeit überhaupt existiert. Durch die Verwendung von Multiprozessoren und präemptivem Multi-Tasking können Echtzeitsituationen entstehen, die durch Datenflussanalyse oder durch Kontrollflussanalyse nicht aufgedeckt werden können.

3.2.3 Wartbarkeit/Änderbarkeit durch statische Analyse verbessern

Die statische Analyse kann unterschiedlich eingesetzt werden um die Wartbarkeit von Programmcode, Softwarearchitektur und Webseiten zu verbessern.

Die Wartung gestaltet sich generell schwieriger, wenn der Code schlecht geschrieben, unkommentiert und unstrukturiert ist. Es kann für die Entwickler aufwändiger sein, Fehlerzustände im Code zu lokalisieren und zu analysieren. Die Änderung des Codes, die aufgrund einer Fehlerbehebung oder eines neuen Features erforderlich ist, kann dazu führen, dass weitere Fehlerzustände eingeschleust werden.

Die statische Analyse kann mit Werkzeugunterstützung auch die Einhaltung von Programmierkonventionen und -richtlinien im vorhandenen Code verifizieren. Zielsetzung dabei ist, die Wartbarkeit des Codes zu verbessern. Diese Programmierkonventionen und -richtlinien beschreiben die erforderlichen Programmierpraktiken, wie z.B. Namenskonventionen, Kommentierung, Quelltexteinrückung und -modularisierung. Es ist zu beachten, dass statische Analysewerkzeuge im Allgemeinen eher Warnungen als Fehlerzustände anzeigen, selbst wenn die Syntax des Quelltexts korrekt ist.

Ein modularer Aufbau verbessert in der Regel die Wartbarkeit des Codes. Statische Analysewerkzeuge unterstützen die Entwicklung modularen Codes auf unterschiedliche Weise:

- Sie suchen nach Wiederholungen im Code. Diese Codeabschnitte bieten sich an für Strukturverbesserungen (Refactoring) zu Modulen (obwohl sich der Overhead der Modul-Aufrufe auf die Laufzeit auswirken kann, was bei Echtzeitsystemen ein Problem sein könnte).
- Sie erzeugen Metriken, die wertvolle Indikatoren für Codemodularisierung sind, u.a. Metriken für die Kopplung und Kohäsion. Ein System, das eine gute Wartbarkeit haben soll, hat wahrscheinlich weniger gekoppelte Module (Module, die während der Ausführung des Codes aufeinander angewiesen sind) und ein hohes Maß an Kohäsion (eigenständige Module für die Durchführung einer einzigen Aufgabe).
- Sie zeigen bei objektorientiertem Programmcode an, wo abgeleitete Objekte eventuell zu viel oder zu wenig Sichtbarkeit zur übergeordneten „Elternklasse" haben.
- Sie identifizieren Bereiche im Programmcode oder in der Systemarchitektur mit hoher struktureller Komplexität, was generell als ein Indikator für schlechte Wartbarkeit und für potenziell höhere Fehlerdichte gilt. Akzeptable Werte für zyklomatische Komplexität (siehe Abschnitt 3.2.1.) können in Richtlinien spezifiziert werden die sicherstellen, dass der Programmcode im Sinne einer besseren Wartbarkeit und Fehlervorbeugung modular entwickelt wird. Programmcode mit einer hohen zyklomatischen Zahl gilt als Kandidat für eine Modularisierung.

Auch die Wartung einer Webseite kann durch statische Analysewerkzeuge unterstützt werden. Hier geht es darum, zu prüfen, ob die Baumstruktur der Webseite ausgeglichen ist oder ob eine Unausgewogenheit vorliegt, die folgende Konsequenzen haben könnte:

- schwierigere Testaufgaben
- höherer Arbeitsaufwand für die Wartung
- schwierige Navigation für den Nutzer

3.2.4 Aufrufgraphen

Aufrufgraphen sind eine statische Repräsentation der Kommunikationskomplexität. Es handelt sich dabei um gerichtete Graphen, in denen Knoten die Programmeinheiten und Kanten die Kommunikationsbeziehungen zwischen den Einheiten darstellen.

Aufrufgraphen können im Komponententest eingesetzt werden, in dem verschiedene Funktionen oder Methoden einander aufrufen, im Integrations- und Systemtest in denen separate Module sich einander aufrufen, oder im Systemintegrationstest in dem separate Systeme einander aufrufen.

Aufrufgraphen können unterschiedlichen Zwecken dienen:

- Tests entwerfen, die ein bestimmtes Modul oder System aufrufen
- Herausfinden wie viele Stellen in der Software das Modul oder System aufrufen
- Struktur des Gesamtcodes und seiner Architektur bewerten
- Vorgehensweisen für die Reihenfolge der Integration vorschlagen (paarweise und benachbarte Integration; weitere Details siehe unten)

Im Foundation Level Lehrplan [ISTQB_FL_SYL] wurden zwei unterschiedliche Kategorien des Integrationstests (bzw. Integrationsstrategien) behandelt: inkrementelle (Top-Down, Bottom-Up, usw.) und nicht-inkrementelle (Big-Bang). Es wurde festgestellt dass die inkrementellen Methoden vorzuziehen sind da sie den Code in Inkrementen integrieren, was das Eingrenzen von Fehlerzuständen erleichtert, weil die Menge des betroffenen Codes begrenzt ist.

Im vorliegenden Advanced Level Lehrplan werden drei weitere nicht-inkrementelle Methoden eingeführt, die Aufrufgraphen verwenden. Diese sind inkrementellen Methoden vorzuziehen, die wahrscheinlich zusätzliche Builds für das Testen sowie Code zur Unterstützung des Testens, der nicht zum Lieferumfang gehört, benötigen. Diese drei nicht-inkrementellen Methoden sind:

- Paarweiser Integrationstest (nicht zu verwechseln mit dem Black-Box-Testverfahren „paarweises Testen"), der sich auf Komponentenpaare konzentriert, die miteinander zusammenarbeiten, wie aus dem Aufrufgraph für den Integrationstest ersichtlich. Während diese Methode die Anzahl der Software-Versionen nur geringfügig verringert, reduziert sie doch die benötigte Menge von Testrahmen-Code.
- Umgebungsintegrationstests basieren auf allen Knoten, die mit einem bestimmten Knoten verbunden sind. Als Basis für den Test dienen alle Vorgänger- und Nachfolger-Knoten eines bestimmten Knotens im Aufrufgraph.
- McCabe's Entwurfsansatz nutzt die Theorie der zyklomatischen Komplexität wie sie bei Aufrufgraphen für Module angewendet wird. Dabei ist die Erstellung eines Aufrufgraphen erforderlich, der die verschiedenen Möglichkeiten darstellt, wie Module sich gegenseitig aufrufen können. Dazu gehören:
  - Bedingungsloser Modulaufruf: Der Aufruf eines Moduls durch ein anderes findet immer statt
  - Bedingter Modulaufruf: Der Aufruf eines Moduls durch ein anderes findet manchmal statt
  - Sich gegenseitig ausschließender bedingter Modulaufruf: Ein Modul ruft von verschiedenen Modulen eines auf (nur eines!)
  - Iterativer Modulaufruf: Ein Modul ruft ein anderes mindestens einmal auf, kann dieses aber auch mehrmals aufrufen
  - Iterativer bedingter Modulaufruf: Ein Modul kann ein anderes null bis viele Male aufrufen

  Nach Erstellung des Aufrufgraphen lassen sich die Integrationskomplexität berechnen und die Testfälle zur Abdeckung des Graphen erstellen.

Weitere Informationen über Aufrufgraphen und den paarweisen Integrationstest siehe [Jorgensen07].

3.3 Dynamische Analyse

3.3.1 Überblick

Die dynamische Analyse wird eingesetzt, um Fehlerwirkungen aufzudecken, deren Symptome nicht sofort offensichtlich sind. Beispiel: Mögliche Speicherlecks können bei einer statischen Analyse

erkennbar sein (es wird Code gefunden der Speicher allokiert aber nie Speicher freigibt), aber ein Speicherleck ist mit dynamischer Analyse sehr leicht zu erkennen.

Fehlerwirkungen, die nicht ohne weiteres reproduzierbar sind, können erhebliche Konsequenzen für den Testaufwand haben und können die Freigabe der Software oder deren produktiven Einsatz verhindern. Ursachen solcher Fehlerwirkungen können Speicherlecks sein, inkorrekter Einsatz von Zeigern und andere Korruptionen (beispielsweise des System-Stacks) [Kaner02]. Diese Art von Fehlerzuständen kann zu einer allmählichen Verschlechterung der Systemleistung oder sogar zu Systemabstürzen führen; die Teststrategie muss deshalb die mit diesen Fehlern verbundenen Risiken berücksichtigen. Falls nötig, müssen die Risiken durch eine dynamische Analyse reduziert werden (normalerweise mit Hilfe von Testwerkzeugen). Da dies häufig Fehlerzustände sind deren Aufdeckung und Behebung am teuersten ist wird empfohlen, dynamische Analysen frühzeitig im Projekt durchzuführen.

Die dynamische Analyse wird durchgeführt um folgendes zu erreichen:

- Fehlerwirkungen verhindern indem fehlerhafte „wilde“ Zeiger und Speicherlecks aufgedeckt werden.
- Systemausfälle analysieren die nicht leicht reproduzierbar sind.
- Netzwerkverhalten bewerten.
- die Systemleistung verbessern indem Informationen über das Laufzeitverhalten des Systems erfasst werden.

Die dynamische Analyse kann in jeder Teststufe durchgeführt werden. Es sind technische Fähigkeiten und Systemkenntnisse zur Durchführung der folgenden Aufgaben erforderlich:

- Spezifizieren der Testziele für die dynamische Analyse
- den geeigneten Zeitpunkt für Beginn und Ende der Analyse bestimmen
- Analysieren der Testergebnisse

Beim Systemtest können dynamische Analysewerkzeuge selbst dann eingesetzt werden, wenn der Technical Test Analyst nur wenig technische Fähigkeiten hat. Die Werkzeuge erstellen meist umfangreiche Protokolle, die von Personen mit den erforderlichen technischen Kompetenzen analysiert werden können.

3.3.2 Speicherlecks aufdecken

Ein Speicherleck tritt auf wenn der Speicher (RAM), der für ein Programm verfügbar ist, zwar allokiert, aber nicht wieder freigegeben wird wenn er nicht mehr benötigt wird. Der betroffene Speicherbereich gilt als allokiert und ist nicht zur Wiederverwendung verfügbar. Wenn dies häufig vorkommt oder wenn ohnehin wenig Hauptspeicher vorhanden ist, kann der nutzbare Speicherplatz ausgehen. Bisher war die Manipulation des Speichers Aufgabe der Programmierer. Dynamisch allokierter Speicherplatz musste vom Programm wieder im korrekten Umfang freigegeben werden, um ein Speicherleck zu vermeiden. Viele moderne Programmierumgebungen sind mit automatischen oder halbautomatischen Funktionen zur Speicherbereinigung ausgestattet, die dafür sorgen, dass allokierter Speicherplatz ohne Eingriff des Programmierers wieder freigegeben wird. Es kann sehr schwierig sein, Speicherlecks zu identifizieren wenn vorhandener, allokierter Speicherplatz durch automatische Speicherbereinigung freigegeben wird.

Speicherlecks verursachen Probleme die sich erst allmählich entwickeln und oft nicht erkennbar sind, so beispielsweise, wenn die Software erst kürzlich installiert oder das System neu gestartet wurde, was beim Testen oft geschieht. Die negativen Auswirkungen von Speicherlecks werden deshalb oft erst bemerkt wenn das Programm in Produktion gegangen ist.

Ein Symptom eines Speicherlecks ist die kontinuierliche Verlangsamung der Antwortzeiten, die letztendlich zu einem Systemausfall führen kann. Man kann solchen Ausfällen zwar mit einem Neustart (Rebooten) des Systems begegnen, was aber oft unpraktisch oder sogar unmöglich ist.

Bereiche, in denen Speicherlecks vorkommen, lassen sich mit dynamischen Analysewerkzeugen identifizieren, sodass die Speicherlecks korrigiert werden können. Auch mit einem einfachen Speichermonitor lässt sich herausfinden, ob der verfügbare Speicherplatz allmählich verringert wird, auch wenn noch eine Nachanalyse durchgeführt werden müsste, um die genaue Ursache einzugrenzen.

Es müssen noch andere Arten von Engpässen betrachtet werden, beispielsweise bei Dateibezeichnern, Zugriffsgenehmigungen (Semaphore) und Verbindungspools für Ressourcen.

3.3.3 Wilde Zeiger aufdecken

„Wilde“ Zeiger in einem Programm sind Zeiger, die nicht benutzt werden dürfen. Sie entstehen wenn es die Objekte oder die Funktionen, auf die sie zeigen, nicht mehr gibt, oder wenn sie nicht auf den vorgesehenen Speicher verweisen sondern beispielsweise auf einen Speicherbereich außerhalb des zugewiesenen Arrays. Wenn ein Programm fehlerhafte Zeiger enthält, kann dies verschiedene Folgen haben:

- Unter Umständen funktioniert das Programm wie erwartet, wenn der Zeiger auf quasi freie Speicherbereiche zeigt, die vom Programm zurzeit nicht genutzt werden (quasi „frei“ sind), und/oder einen plausiblen Wert enthalten.
- Das Programm kann abstürzen, wenn der Zeiger auf einen Teil des Speichers zeigt und dieser dann verwendet wird, obwohl er für den Betrieb des Systems benötigt wird (z.B. das Betriebssystem).
- Das Programm funktioniert nicht korrekt, weil es auf benötigte Objekte nicht zugreifen kann. Es kann dann zwar weiterhin funktionieren, gibt aber Fehlermeldungen aus.
- Durch fehlerhafte Zeiger können Daten zerstört oder unbrauchbar werden, sodass dann inkorrekte Werte verwendet werden.

Hinweis: Jede Änderung an der Speichernutzung durch das Programm (beispielsweise ein neuer Build nach einer Softwareänderung) kann eine dieser vier Folgen haben. Dies ist vor allem dann kritisch wenn das Programm zunächst wie erwartet funktioniert obwohl es fehlerhafte Zeiger enthält, jedoch nach einer Softwareänderung abstürzt (vielleicht sogar im Produktionseinsatz). Solche Ausfälle sind Symptome eines tieferliegenden Fehlerzustands, siehe auch [Kaner02], Lesson 74. Werkzeuge können fehlerhafte Zeiger identifizieren die ein Programm verwendet, unabhängig von den Folgen der Zeiger für die Programmausführung. Manche Betriebssysteme haben integrierte Funktionen zur Überwachung von Speicherzugriffsverletzungen während der Laufzeit. Beispielsweise meldet dann das Betriebssystem eine Ausnahmesituation, wenn eine Anwendung versucht, auf einen Speicherbereich zuzugreifen, der außerhalb des zulässigen Speicherbereichs der Anwendung ist.

3.3.4 Systemleistung analysieren

Die dynamische Analyse ist nicht nur zum Aufdecken von Fehlern geeignet. Mit einer dynamischen Analyse lässt sich auch die Programmleistung analysieren. Werkzeuge können Leistungsengpässe identifizieren und ein breites Spektrum an Performanzmetriken erzeugen. Damit können die Entwickler das System optimieren. Beispiel: Es können Informationen geliefert werden, dass ein bestimmtes Modul während der Ausführung mehrmals aufgerufen wird. Module, die häufig aufgerufen werden, bieten sich für eine Performanzverbesserung geradezu an.

Die Tester können die Informationen über das dynamische Verhalten der Software mit denen aus den Aufrufgraphen der statischen Analyse (siehe Abschnitt 3.2.4) zusammenführen und so die Module identifizieren, die umfangreich und im Detail getestet werden sollten (z.B. Module, die oft aufgerufen werden und viele Schnittstellen haben).

Die dynamische Analyse der Programmleistung wird häufig während des Systemtests durchgeführt, kann aber auch schon in früheren Testphasen erfolgen, wenn ein einzelnes Teilsystem mit Hilfe eines Testrahmens getestet wird.

4. Qualitätsmerkmale bei technischen Tests - 405 Minuten

Begriffe

Analysierbarkeit, Anpassbarkeit, Austauschbarkeit, betrieblicher Abnahmetest, Effizienz, Koexistenz, Modifizierbarkeit, Installierbarkeit, Nutzungsprofil, Performanztest, Portabilitätstest, Reife, Test der Ressourcennutzung, Robustheit, Sicherheitstest, Stabilität, Testbarkeit, Wartbarkeitstest, Wiederherstellbarkeitstest, Zuverlässigkeitstest, Zuverlässigkeitswachstumsmodell

Lernziele für Qualitätsmerkmale bei technischen Tests

4.2 Allgemeine Planungsaspekte

TTA-4.2.1 (K4) Sie können für ein bestimmtes Projekt und ein bestimmtes zu testendes System die nicht-funktionalen Anforderungen analysieren und dafür die entsprechenden Teile des Testkonzepts erstellen

4.3 Sicherheitstest

TTA-4.3.1 (K3) Sie können die Vorgehensweise für den Sicherheitstest definieren und abstrakte Testfälle dafür entwerfen

4.4 Zuverlässigkeitstest

TTA-4.4.1 (K3) Sie können die Vorgehensweise für den Zuverlässigkeitstest definieren und abstrakte Testfälle für das Zuverlässigkeitsmerkmal und dessen Teilmerkmale nach ISO 9126 entwerfen

4.5 Performanztest

TTA-4.5.1 (K3) Sie können die Vorgehensweise für den Performanztest definieren und abstrakte Nutzungsprofile dafür entwerfen

Übergreifende Lernziele

Die nachfolgenden Lernziele beziehen sich auf Inhalte, die in mehreren Abschnitten dieses Kapitels behandelt werden.

TTA-4.x.1 (K2) Sie können verstehen und erläutern, warum Tests auf Wartbarkeit, Portabilität und Ressourcennutzung Teil einer Teststrategie und/oder Testvorgehensweise sein sollten

TTA-4.x.2 (K3) Sie können für ein bestimmtes Produktrisiko die spezifische nicht-funktionale Testart (bzw. Testarten) definieren, die am besten geeignet ist (sind)

TTA-4.x.3 (K2) Sie können verstehen und erläutern, in welchen Lebenszyklusphasen einer Softwareanwendung nicht-funktionales Testen erfolgen sollte

TTA-4.x.4 (K3) Sie können für ein vorgegebenes Szenario festlegen, welche Fehlerarten Sie durch die Anwendung nicht-funktionaler Testarten aufdecken wollen

4.1 Einführung

Im Allgemeinen sind Technical Test Analysten in erster Linie darauf fokussiert, wie das Produkt funktioniert (und weniger auf die funktionalen Aspekte). Die Tests können in jeder Teststufe stattfinden. Beim Komponententest von Echtzeitsystemen und eingebetteten Systemen sind das Performanz-Benchmarking und das Testen der Ressourcennutzung besonders wichtig. Beim Systemtest und betrieblichen Abnahmetests (Produktionsabnahmetests) ist das Testen von Zuverlässigkeitsmerkmalen (z.B. Wiederherstellbarkeit) angebracht. Tests in dieser Stufe sind auf ein spezifisches System ausgerichtet, d.h. auf eine spezifische Kombination von Hardware und Software. Zu diesem zu testenden System können verschiedene Server, Clients, Datenbanken, Netzwerke und andere Ressourcen gehören. Unabhängig von der Teststufe sollte das Testen entsprechend der festgelegten Risikoprioritäten und verfügbaren Ressourcen erfolgen.

Die Beschreibung der Qualitätsmerkmale orientiert sich am ISO Standard 9126, der als Leitfaden verwendet wurde. Weitere Standards, wie z.B. ISO Standard 25000 [ISO 25000] (der ISO 9126 abgelöst hat), gelten ebenfalls als nützlich. Die Qualitätsmerkmale werden laut ISO 9126 eingeteilt in Produktqualitätsmerkmale (oder -eigenschaften), die jeweils in weitere Teilmerkmale (oder Teileigenschaften) unterteilt werden können. Diese sind in der folgenden Tabelle zusammengefasst, aus der ebenfalls hervorgeht, in welchem Lehrplan (Test Analyst oder Technical Test Analyst) die jeweiligen Merkmale behandelt werden:

Merkmal	**Teilmerkmale**	**Test Analyst**	**Technical Test Analyst**
Funktionalität	Richtigkeit, Angemessenheit, Interoperabilität, Einhaltung von Standards	X	
	Sicherheit		X
Zuverlässigkeit	Softwarereife (Robustheit), Fehlertoleranz, Wiederherstellbarkeit, Einhaltung von Standards		X
Benutzbarkeit	Verständlichkeit, Erlernbarkeit, Operabilität, Attraktivität, Einhaltung von Standards	X	
Effizienz	Performanz (Zeitverhalten), Ressourcen-nutzung, Einhaltung von Standards		X
Wartbarkeit	Analysierbarkeit, Änderbarkeit, Stabilität, Testbarkeit, Einhaltung von Standards		X
Portabilität	Anpassbarkeit, Installierbarkeit, Koexistenz, Austauschbarkeit, Einhaltung von Standards		X

Auch wenn die Verteilung der Zuständigkeiten in unterschiedlichen Organisationen möglicherweise variiert, so ist sie in den vorliegenden Lehrplänen des ISTQB auf diese Weise vorgenommen worden.

Das Teilmerkmal „Einhaltung von Standards“ ist bei jedem der Qualitätsmerkmale aufgeführt. In bestimmten sicherheitskritischen oder streng regulierten Umfeldern müssen einzelne Qualitätsmerkmale bestimmte Standards und Vorschriften erfüllen. Da diese Standards je nach Branche stark variieren, werden sie an dieser Stelle nicht eingehend behandelt. Wenn ein Technical Test Analyst in einem Umfeld arbeitet, in dem Anforderungen für die Einhaltung von Standards gelten, dann muss der Technical Test Analyst diese Anforderungen verstehen und sicherstellen, dass sowohl das Testen als auch die Testdokumentation diese Anforderungen erfüllen.

Für alle in diesem Abschnitt behandelten Qualitätsmerkmale und Teilmerkmale müssen die typischen Risiken erkannt werden, damit eine geeignete Teststrategie ausgearbeitet und dokumentiert werden kann. Für das Testen von Qualitätsmerkmalen müssen das richtige Timing im Lebenszyklus, benötigte

Werkzeuge, Verfügbarkeit von Software und Dokumentation, sowie technisches Fachwissen besondere Beachtung finden. Wenn der Tester die Strategie für jedes einzelne Merkmal und dessen spezifische Testerfordernisse nicht konsequent plant, dann ist es möglich, dass der Zeitplan nicht genug Zeit für Planung, Vorbereitung und Durchführung der Tests vorsieht [Bath08]. Einige Tests, z.B. Performanztests, erfordern umfangreiche Planung, spezielle Ausrüstung, bestimmte Werkzeuge, spezielle Testfähigkeiten, und in den meisten Fällen auch einen erheblichen Zeitaufwand. Das Testen der Qualitätsmerkmale und Teilmerkmale muss in die Gesamttestplanung integriert werden, und es müssen für die Aufgaben ausreichend Ressourcen ausgewiesen werden. Jeder der Testbereiche hat bestimmte Erfordernisse, befasst sich mit bestimmten Problematiken und kann zu unterschiedlichen Zeitpunkten im Softwarelebenszyklus vorkommen. Dies wird in den nachfolgenden Abschnitten behandelt.

Während sich Testmanager mit dem Erstellen und Berichten der zusammengefassten Informationen aus Metriken über die Qualitätsmerkmale und Teilmerkmale befassen, ist der Test Analyst oder der Technical Test Analyst (siehe Tabelle oben) für das Erheben der Informationen zu jeder der Metriken zuständig.

Metriken für die Qualitätsmerkmale, die der Technical Test Analyst in Tests vor Produktionseinführung erhebt, können als Grundlage für die Service Level-Vereinbarungen des Softwaresystems zwischen Lieferant und Stakeholdern (z.B. Kunden, Betreiber) dienen. Diese Tests werden oft auch noch durchgeführt, nachdem die Software in Produktion gegangen ist. Dann ist oft ein separates Team oder eine eigene Bereich dafür zuständig. Dies ist häufig der Fall bei Effizienz- und Zuverlässigkeitstests, die in der Produktionsumgebung zu anderen Testergebnissen führen können als in einer Testumgebung.

4.2 Allgemeine Planungsaspekte

Wenn vergessen wird, nicht-funktionale Tests einzuplanen, kann das den Erfolg einer Anwendung ernsthaft in Frage stellen. Der Technical Test Analyst kann durchaus vom Testmanager damit beauftragt werden, die wichtigsten Risiken für die relevanten Qualitätsmerkmale (siehe Tabelle in Abschnitt 4.1) zu identifizieren und sich um die Planungsaufgaben rund um die angedachten Tests zu kümmern. Die Ergebnisse gehen dann möglicherweise auch in das Mastertestkonzept ein. Auf die folgenden allgemeinen Faktoren ist beim Verrichten dieser Aufgaben zu achten:

- Anforderungen der Stakeholder
- Beschaffung benötigter Werkzeuge und Schulungen
- Anforderungen bzgl. der Testumgebung
- organisatorische Faktoren
- Fragen der Datensicherheit

4.2.1 Anforderungen der Stakeholder

Nicht-funktionale Anforderungen sind oft unbekannt oder kaum spezifiziert. Die Technical Test Analysten müssen deshalb in der Planungsphase die Erwartungshaltungen der Stakeholder bezüglich der technischen Qualitätsmerkmale sondieren und daraufhin bewerten, welche Risiken sie für das Projekt bedeuten.

Ein durchaus üblicher Denkansatz ist, dass die Kunden – so sie mit der derzeitigen Systemversion zufrieden sind – auch mit den neuen Versionen zufrieden sein werden, solange das bisher erreichte Qualitätsniveau eingehalten wird. In diesem Fall kann die bestehende Version des Systems als Referenzsystem dienen. Für einige der nicht-funktionalen Qualitätsmerkmale kann dies eine sehr nützliche Vorgehensweise sein, beispielsweise für die Performanz eines Systems, wenn es den Stakeholdern schwer fällt, die Anforderungen zu spezifizieren.

Beim Ermitteln der nicht-funktionalen Anforderungen sollten verschiedene Perspektiven berücksichtigt werden. Damit keine Anforderungen übersehen werden, müssen unterschiedliche Stakeholder, wie z. B. Kunden, Anwender, Bedien- und Wartungspersonal, befragt werden.

4.2.2 Beschaffung benötigter Werkzeuge und Schulungen

Kommerzielle Werkzeuge oder Simulatoren sind besonders relevant für Performanz- und für bestimmte Sicherheitstests. Technical Test Analysten sollten die Kosten und Vorlaufzeiten für Beschaffung, Training und Einführung der benötigten Werkzeuge einschätzen können. Wenn Spezialwerkzeuge zum Einsatz kommen, sind damit verbundene Lernkurven oder die Kosten für den Einsatz externer Spezialisten ebenfalls zu berücksichtigen.

Einen komplexen Simulator zu entwickeln, kann zu einem eigenen Entwicklungsprojekt werden. Dementsprechend sollte es dann auch geplant werden. Insbesondere müssen das Testen und die Dokumentation für das eigenentwickelte Werkzeug in den Zeitplan und die Ressourcenplanung einfließen. Auch für Aktualisierungen und Nachtests der Simulatoren, die notwendig werden, wenn sich das simulierte Produkt ändert, ist ausreichend Budget und Zeit vorzusehen. Wenn Simulatoren für sicherheitskritische Anwendungen eingesetzt werden sollen, sind auch die entsprechenden Abnahmetests und eine etwaige Zertifizierung des Simulators durch eine unabhängige Instanz in Erwägung zu ziehen.

4.2.3 Anforderungen der Testumgebung

Für viele nicht-funktionale Tests (z.B. Sicherheitstests, Performanztests) wird eine produktionsähnliche Testumgebung benötigt, um realistische Messungen zu ermöglichen. Größe und Komplexität des zu testenden Systems können Planung und Finanzierung der Tests maßgeblich beeinflussen. Da die Kosten für derartige Umgebungen hoch sein können, sollten die folgenden Alternativen in Betracht gezogen werden:

- Verwendung der echten Produktionsumgebung
- Verwendung einer reduzierten Version des Systems. Hierbei muss aber sorgfältig vorgegangen werden, damit die gelieferten Testergebnisse für das echte Produktionssystem ausreichend repräsentativ sind.

Die Durchführung der Tests in der Produktionsumgebung muss sorgfältig geplant werden; sie können wahrscheinlich nur zu bestimmten Zeiten (z.B. Zeiten geringer Nutzung) stattfinden.

4.2.4 Organisatorische Faktoren

Bei nicht-funktionalen Tests ist oft auch das Verhalten mehrerer Komponenten eines Gesamtsystems zu messen (beispielsweise Server, Datenbanken, Netzwerke). Wenn diese Komponenten auf unterschiedliche Standorte und Unternehmen verteilt sind, kann das erheblichen Aufwand für die Planung und Koordinierung der Tests bedeuten. So können bestimmte Softwarekomponenten nur zu bestimmten Uhrzeiten oder nur an bestimmten Tagen im Jahr für den Systemtest zur Verfügung stehen. Unternehmen stellen möglicherweise nur eine begrenzte Anzahl von Tagen zur Verfügung, an denen sie das Testen unterstützen. Es kann zu empfindlichen Störungen der geplanten Tests führen, wenn nicht im Vorfeld geklärt worden ist, dass die Systemkomponenten und das Personal der anderen Unternehmen (d.h. „ausgeliehene“ externe Fachkenntnisse) für die Testzwecke auf Abruf bereitstehen.

4.2.5 Fragen der Datensicherheit

Spezifische Sicherheitsmaßnahmen für ein System sollten bereits in der Testplanungsphase berücksichtigt werden, damit alle vorgesehenen Testaktivitäten tatsächlich durchführbar sind. Werden beispielsweise die Daten verschlüsselt, kann es schwierig sein, Testdaten zu erzeugen und die Testergebnisse zu verifizieren.

Richtlinien und gesetzliche Bestimmungen zum Datenschutz schließen möglicherweise aus, dass die Tester die benötigten Testdaten auf Basis der Produktionsdaten generieren. Es kann eine schwierige Aufgabe sein, die Testdaten zu anonymisieren; dies muss als Teil der Testrealisierung geplant werden.

4.3 Sicherheitstest

4.3.1 Einführung

Sicherheitstests unterscheiden sich von anderen Formen funktionaler Tests in zwei wichtigen Aspekten:

1. Standardverfahren zur Auswahl der Testeingangsdaten können wichtige Sicherheitsaspekte außer Acht lassen.
2. Die Symptome von Sicherheitsfehlern unterscheiden sich grundlegend von Symptomen, die bei anderen funktionalen Tests gefunden werden.

Sicherheitstests untersuchen die Verwundbarkeit eines Systems durch diverse Gefährdungen, indem sie versuchen, die Security Policy eines Systems gezielt außer Kraft zu setzen. Die nachfolgende Liste enthält mögliche Bedrohungen, die beim Sicherheitstest untersucht werden sollten:

- Nicht autorisiertes Kopieren von Anwendungen oder Daten
- Nicht autorisierter Zugriff (d.h. Aktionen ausführen, für die der Nutzer keine Zugriffsberechtigung hat). Das Testen konzentriert sich auf Benutzerrechte, Zugriffsberechtigungen und Privilegien. Diese Informationen sollten in den Spezifikationen des Systems enthalten sein.
- Die Software zeigt weitere, nicht beabsichtigte Seiteneffekte, wenn sie eine vorgesehene Funktion ausführt. Beispiel: Eine Abspielsoftware spielt Audio korrekt ab, schreibt dabei aber Dateien in einen nicht verschlüsselten Zwischenspeicher. Softwarepiraten könnten einen solchen Nebeneffekt ausnutzen.
- Code, der von außen in eine Webseite eingebracht wird, könnte durch Nutzer gezielt eingesetzt werden (Cross-Site-Scripting oder XSS). Dieser Code kann bösartigen Zwecken dienen.
- Überlauf des Eingabebereichs (Speicherüberlauf), der beispielsweise durch Eingabe extrem langer Zeichenketten über ein Eingabefeld der Benutzerschnittstelle ausgelöst werden kann. Nach diesem Überlauf lässt sich möglicherweise bösartiger Code ausführen.
- Denial of Service-Angriff, sodass die Anwendung nicht mehr genutzt werden kann (beispielsweise indem ein Webserver mit Übermengen von Störanfragen überschwemmt wird)
- Vom Nutzer unbemerktes Abhören, Nachahmen und/oder Abändern von (vertraulichen) Daten und anschließende Weiterleitung der Kommunikationen (beispielsweise bei Kreditkarten-Transaktionen) durch Dritte („Man-in-the-middle-Angriff").
- Verschlüsselungscodes knacken, die vertrauliche Daten schützen.
- Logische Fallen (engl. logic bombs, in den USA manchmal „easter eggs" genannt), die in bösartiger Absicht in den Code eingeschleust und nur unter bestimmten Bedingungen aktiviert werden (beispielsweise an einem bestimmten Datum). Werden diese logischen Fallen aktiviert, so lösen sie Schadaktionen aus, wie das Löschen von Dateien oder das Formatieren von Festplatten.

4.3.2 Sicherheitstests planen

Für die Planung von Sicherheitstests sind die folgenden Themen besonders relevant:

- Da Sicherheitsprobleme schon beim Architekturentwurf, Systementwurf und bei der Systemimplementierung verursacht werden können, sind Sicherheitstests in den Modultest-, Integrationstest- und Systemtestphasen einzuplanen. Weil sich die Art der Sicherheits-

bedrohungen ständig ändert, sollten Sicherheitstests auch regelmäßig für die Zeit nach dem Produktivgang der Anwendung vorgesehen werden.

- Die vom Technical Test Analyst vorgeschlagenen Teststrategien sollten Code-Reviews und statische Analyse durch Sicherheitswerkzeuge beinhalten. Damit können Sicherheitsprobleme in der Systemarchitektur, den Entwurfsdokumenten und im Programmcode sehr effektiv aufgedeckt werden, die andererseits beim dynamischen Test leicht übersehen werden.
- Technical Test Analysten werden häufig auch gebeten, bestimmte Sicherheitsangriffe zu entwerfen und auszuführen (siehe unten). Dies erfordert eine sorgfältige Planung und Koordination mit Stakeholdern. Andere Sicherheitstests können in Zusammenarbeit mit Entwicklern oder Test Analysten durchgeführt werden (z.B. Tests der Benutzerrechte, Zugriffsberechtigungen und Privilegien). In der Planung der Sicherheitstests muss auf derartige organisatorische Aspekte sorgfältig eingegangen werden.
- Ein unerlässlicher Aspekt bei der Planung von Sicherheitstests ist das Einholen von Genehmigungen. Für den Technical Test Analyst bedeutet dies konkret, dass er für die Durchführung der geplanten Sicherheitstests vom Testmanager eine ausdrückliche Erlaubnis einholen muss. Alle zusätzlichen, spontan angesetzten Tests könnten für echte Angriffe gehalten werden und für die ausführende Person mit rechtlichen Konsequenzen enden. Wenn dann keine schriftlichen Unterlagen vorgelegt werden können, aus denen der Auftrag und die Genehmigung für die Tests hervorgeht, wird die Ausrede „Wir haben einen Sicherheitstest durchgeführt" wenig überzeugend klingen.
- Es ist zu beachten, dass sich Verbesserungen der Sicherheit eines Systems auf die Systemleistung auswirken können. Nach Sicherheitsverbesserungen am System ist daher zu überlegen, ob Performanztests notwendig sind (siehe Abschnitt 4.5 unten).

4.3.3 Spezifikation von Sicherheitstests

Bestimmte Sicherheitstests lassen sich je nach Ursprung des Sicherheitsrisikos unterscheiden [Whittaker04]:

- Die Benutzerschnittstelle betreffend - Nicht autorisierter Zugriff und bösartige Eingaben
- Das Dateisystem betreffend – Zugriff auf vertrauliche Daten in Dateien oder Repositories
- Das Betriebssystem betreffend – Unverschlüsseltes Ablegen von sicherheitskritischen Informationen, wie z. B. Passwörtern. Wird das System durch bösartige Eingaben zum Absturz gebracht, können die Informationen zugänglich werden
- Externe Software betreffend – Interaktionen zwischen externen Komponenten, die das System nutzt. Sie können auf Netzwerkebene auftreten (wenn beispielsweise inkorrekte Datenpakete oder Meldungen übertragen werden) oder auf Ebene der Softwarekomponenten (wenn beispielsweise eine Softwarekomponente ausfällt, die vom System benötigt wird).

Mögliche Verfahren [Whittaker04] beim Entwickeln der Sicherheitstests sind:

- Nützliche Informationen zur Spezifikation von Tests zusammentragen, wie z.B. die Namen der Mitarbeiter, physikalische Adressen, Details der internen Netzwerke, IP-Nummern, Typ der verwendeten Software/Hardware und die Betriebssystem-Version.
- Schwachstellen des Systems mit gängigen Werkzeugen scannen. Diese dienen nicht dazu, in das System direkt einzudringen, sondern zunächst einmal dazu, jene Schwachstellen zu identifizieren, die Sicherheitvorkehrungen durchbrechen oder später mal durchbrechen könnten. Spezifische Schwachstellen lassen sich auch mit Hilfe von Checklisten identifizieren, z.B. mit Checklisten des National Institute of Standards and Technology (NIST, US-Bundesbehörde) [Web-2].
- Sicherheitsangriffe entwickeln, d.h. mit den gesammelten Informationen werden Testschritte geplant, die die Security Policy eines bestimmten Systems durchbrechen sollen. Für die Angriffspläne müssen mehrere Eingaben über verschiedene Schnittstellen (beispielsweise Benutzerschnittstelle, Dateisystem) spezifiziert werden, um so die schwersten Sicherheitsfehler aufzudecken. Die verschiedenen bei [Whittaker04] beschriebenen

Sicherheitsangriffe sind eine wertvolle Quelle für Verfahren, die speziell für Sicherheitstests entwickelt wurden.

Sicherheitsprobleme können auch durch Reviews (siehe Kapitel 5) und/oder mit Hilfe von statischen Analysatoren (siehe Abschnitt 3.2) aufgedeckt werden. Statische Analysewerkzeuge beinhalten eine umfangreiche Menge von Regeln, die ganz speziell die Sicherheitsbedrohungen betreffen, und anhand derer der Programmcode geprüft wird. So können die Werkzeuge beispielsweise Probleme mit dem Pufferüberlauf aufdecken, weil die Puffergröße nicht geprüft wird, bevor Daten zugewiesen werden.

Statische Analysatoren können für Web-Code verwendet werden, um zu prüfen, ob Sicherheitsverletzungen möglich sind, wie z.B. Code-Injektion, Sicherheit von Cookies, Cross-Site-Scripting, Manipulation von Ressourcen und SQL-Injektion.

4.4 Zuverlässigkeitstest

Der ISO Standard 9126 definiert für das Qualitätsmerkmal der Zuverlässigkeit eines Produktes folgende Teilmerkmale:

- Reife
- Fehlertoleranz
- Wiederherstellbarkeit

4.4.1 Softwarereife messen

Ein Ziel beim Testen der Zuverlässigkeit ist es, ein statistisches Maß der Softwarereife über eine Zeitspanne zu überwachen und mit der gewünschten Zuverlässigkeit zu vergleichen, welche in einem Service Level Agreement spezifiziert sein kann. Solch ein Maß kann die mittlere Betriebsdauer zwischen Ausfällen[3], die mittlere Reparaturzeit nach einem Ausfall eines Systems[4] oder eine andere Messung der Fehlerdichte (beispielsweise die wöchentliche Anzahl der Fehlerwirkungen mit einem bestimmten Schweregrad) sein. Diese Metriken können auch als Endekriterien (beispielsweise für die Produktionsfreigabe) dienen.

4.4.2 Fehlertoleranz testen

Zusätzlich zum funktionalen Testen, welches die Fehlertoleranz der Software gegenüber unerwarteten Eingaben (sog. Negativtests) evaluiert, sind weitere Tests notwendig, welche die Toleranz des Systems gegenüber Fehlern bewerten, die außerhalb der zu testenden Anwendung auftreten. Solche Fehlerzustände werden normalerweise vom Betriebssystem gemeldet (beispielsweise: Festplatte voll, Prozess oder Dienst nicht verfügbar, Datei nicht gefunden, Speicher nicht verfügbar). Tests der Fehlertoleranz können auf der Systemebene durch spezifische Werkzeuge unterstützt werden.
Anmerkung: In Zusammenhang mit der Fehlertoleranz (engl. „fault tolerance" oder „error tolerance") wird auch häufig der Begriff „Robustheit" verwendet (Details siehe [ISTQB_GLOSSARY]).

4.4.3 Wiederherstellbarkeitstest

Weitere Formen von Zuverlässigkeitstests evaluieren die Fähigkeit eines Softwaresystems, wie es sich nach einem Hardware- oder Softwareausfall mit einem definierten Vorgehen wiederherstellen lässt, sodass anschließend der normale Betrieb fortgesetzt werden kann. Zu Wiederherstellbarkeitstests gehören Tests bzgl. Failover (Ausfallsicherheit), Backup und Restore.

[3] Engl. Mean Time Between Failures (MTBF)
[4] Engl. Mean Time to Repair (MTTR)

Ausfallsicherheitstests werden durchgeführt, wenn die Konsequenzen eines Softwareversagens so gravierend sind, dass spezielle Hardware- und/oder Softwaremaßnahmen beim System implementiert wurden, um den weiteren Betrieb selbst nach einem Versagen sicherzustellen. Ausfallsicherheitstests können sinnvoll sein, wenn beispielsweise das Risiko finanzieller Verluste als extrem hoch einzustufen ist, oder wenn kritische Sicherheitsanforderungen vorliegen. Wenn die Ursache solcher Systemausfälle eine Katastrophe sein kann, bezeichnet man diese Art von Wiederherstellbarkeitstests auch als „Desaster Recovery“ Test (Test auf Wiederherstellbarkeit nach einem Totalausfall).

Typische präventive Maßnahmen für Hardwareversagen können eine Lastverteilung auf mehrere Prozessoren, Serverclustern, Prozessoren oder Festplatten inkludieren, sodass bei einem Ausfall sofort die eine Komponente von der anderen übernehmen kann (redundante Systeme). Eine typische software-basierte Maßnahme kann die Implementierung von mehr als einer unabhängigen Instanz des Softwaresystems in so genannten dissimilar redundanten Systemen sein (beispielsweise beim Steuerungssystem eines Flugzeugs). Redundante Systeme sind normalerweise eine Kombination aus Software- und Hardwaremaßnahmen und werden je nach Anzahl der unabhängigen Instanzen als zweifach/dreifach/vierfach redundante Systeme bezeichnet. Redundante Lösungen lassen sich dadurch erreichen, dass zwei (oder mehrere) unabhängige Entwicklungsteams dieselben Softwareanforderungen zur Umsetzung erhalten, mit dem Ziel, dass dieselben Services mit unterschiedlichen Lösungsansätzen geliefert werden müssen. Dies schützt die mehrfach redundanten Systeme, weil fehlerhafte Eingaben sehr unwahrscheinlich dieselben Ergebnisse hervorrufen. Diese Maßnahmen zur Verbesserung der Wiederherstellbarkeit eines Systems können auch dessen Zuverlässigkeit direkt beeinflussen und sollten auch bei der Durchführung der Zuverlässigkeitstests berücksichtigt werden.

Ausfallsicherheitstests sind dafür gemacht, Systeme durch Herbeiführung von Fehlerwirkungen in einer kontrollierten Umgebung oder Simulationen der Fehlerwirkung zu testen. Nach einer Fehlerwirkung wird der Ausfallsicherheitsmechanismus getestet, um sicherzustellen, dass der Vorfall weder Datenverlust noch Datenkorruption bewirkt hat, und dass die Service-Level-Vereinbarungen eingehalten wurden (beispielsweise Funktionsverfügbarkeit, Antwortzeiten). Weitere Informationen zu Ausfallsicherheitstests, siehe [Web-1].

Backup- und Wiederherstellbarkeitstests untersuchen Maßnahmen und Verfahren, welche die möglichen Fehlerwirkungen minimieren sollen. Solche Tests bewerten die (meist in Handbüchern dokumentierten) Verfahrensweisen sowohl für verschiedene Backups als auch für Daten-wiederherstellungsprozeduren, falls Datenverluste oder korrupte Daten auftreten. Die Testfälle werden so entworfen, dass kritische Pfade in den Verfahren abgedeckt sind. Als Trockenübung für diese Szenarien lassen sich technische Reviews durchführen, welche die Handbücher gegen die tatsächlichen Verfahren validieren. Beim betrieblichen Abnahmetest werden diese Szenarien in einer echten Produktionsumgebung oder in einer produktionsähnlichen Umgebung eingesetzt, um ihre tatsächliche Anwendung zu validieren.

Mögliche Metriken, die bei Backup- und Wiederherstellungstests gemessen werden können, sind:

- benötigte Zeit für die verschiedenen Backup-Arten (beispielsweise vollständiges, inkrementelles Backup)
- benötigte Zeit für die Wiederherstellung der Daten
- Definierte Ebenen von garantierten Daten-Backups (beispielsweise Wiederherstellung aller Daten, welche nicht älter als 24 Stunden sind, oder Wiederherstellung spezifischer Transaktionsdaten, welche nicht älter als eine Stunde sind).

4.4.4 Zuverlässigkeitstests planen

Im Allgemeinen sind für die Planung von Zuverlässigkeitstests folgende Aspekte besonders relevant:

- Die Zuverlässigkeit kann auch noch überwacht werden, nachdem die Software in Produktion gegangen ist. Das Unternehmen und die für den Betrieb der Software verantwortlichen Personen müssen befragt werden, wenn die Zuverlässigkeitsanforderungen für die Testplanung erfasst werden.
- Der Technical Test Analyst kann ein Zuverlässigkeitswachstumsmodell (Reliability Growth Model) auswählen, welches die zu erwartenden Zuverlässigkeitswerte über eine Zeitspanne zeigt. Ein solches Modell kann nützliche Informationen für den Testmanager liefern, da die erwarteten mit den erreichten Zuverlässigkeitswerten verglichen werden können.
- Zuverlässigkeitstests sollten in einer produktionsähnlichen Umgebung durchgeführt werden. Die verwendete Umgebung sollte so stabil wie möglich bleiben, damit Zuverlässigkeitstrends über eine Zeitspanne beobachtet werden können.
- Da für die Zuverlässigkeitstests häufig das Gesamtsystem notwendig ist, erfolgen diese Tests meist als Teil des Systemtests. Es ist jedoch auch möglich, Zuverlässigkeitstests für einzelne Komponenten oder für integrierte Komponenten durchzuführen. Auch Reviews der detaillierten Systemarchitektur, des Systemdesigns und des Codes können verwendet werden, um das Risiko von Zuverlässigkeitsproblemen innerhalb des implementierten Systems zu reduzieren.
- Damit die Testergebnisse statistisch bedeutend sind, benötigen Zuverlässigkeitstests lange Durchlaufzeiten. Dies kann Schwierigkeiten bei der Planung mit anderen Tests hervorrufen.

4.4.5 Spezifikation von Zuverlässigkeitstests

Zuverlässigkeitstests können durch ein wiederholbares, vordefiniertes Set an Testfällen erstellt und durchgeführt werden. Diese Tests können zufällig aus einem Pool gewählt werden, oder es werden Testfälle[5] aus einem statistischen Modell mit Hilfe von zufälligen oder pseudo-zufälligen Methoden generiert. Zuverlässigkeitstests können auch auf Anwendungsmustern (auch Nutzungsprofile genannt) basieren (siehe Abschnitt 4.5.4).

Bestimmte Zuverlässigkeitstests können vorsehen, dass besonders speicherintensive Vorgänge wiederholt ausgeführt werden müssen, um so gezielt mögliche Speicherlecks zu entdecken.

4.5 Performanztest

4.5.1 Einführung

Im Standard ISO 9126 ist die Performanz (Zeit- & Ressourcenverhalten) als ein Teilmerkmal des Qualitätsmerkmals der Effizienz eines Produktes definiert. Performanztests untersuchen die Fähigkeit einer Komponente oder eines Systems, auf Eingaben des Nutzers oder eines Systems innerhalb einer definierten Zeit, sowie unter spezifizierten Bedingungen zu reagieren.

Performanzmessungen können je nach Testziel variieren. So kann die Performanzmessung einer einzelnen Softwarekomponente die CPU-Zyklen messen; bei Client-basierten Systemen lässt sich dagegen die Zeit messen, welche benötigt wird, um auf eine bestimmte Nutzeranfrage zu reagieren. Bei Systemen, deren Architektur aus mehreren Komponenten besteht (beispielsweise Clients, Server, Datenbanken), wird die Performanz für die Transaktionen zwischen den einzelnen Komponenten gemessen, um Performanzengpässe zu identifizieren.

[5] hier: konkrete Testfälle (Testdaten)

4.5.2 Arten von Performanztests

4.5.2.1 Lasttests

Lasttests messen die Fähigkeit eines Systems, ansteigende Grade erwarteter, realistischer Systemlasten zu bewältigen, welche eine Anzahl paralleler Nutzer oder Prozesse als Transaktionsanfragen generieren. Die durchschnittlichen Antwortzeiten der Nutzer werden in typischen, unterschiedlichen Nutzungsszenarien (Nutzungsprofilen) gemessen und analysiert. Siehe auch [Splaine01].

4.5.2.2 Stresstests

Stresstests untersuchen die Fähigkeit eines Systems oder einer Softwarekomponente, Spitzenlasten an oder über den spezifizierten Kapazitätsgrenzen oder mit reduzierten Ressourcen (z.B. verfügbare Rechnerkapazität und verfügbare Bandbreite) zu bewältigen. Mit steigender Überbelastung sollte die Systemleistung allmählich, vorhersehbar und ohne Ausfall abnehmen. Vor allem sollte die funktionale Integrität des Systems unter Spitzenlast getestet werden, um mögliche Fehlerzustände bei der funktionalen Verarbeitung aufzudecken oder Dateninkonsistenzen festzustellen.

Ein mögliches Ziel von Stresstests ist die Identifizierung der Grenze, bei der das System tatsächlich ausfällt, und somit die Identifizierung des schwächsten Gliedes in der Kette. Bei Stresstests ist es erlaubt, dem System rechtzeitig zusätzliche Kapazitäten hinzuzufügen (beispielsweise Speicher, CPU-Kapazität, Datenbankspeicher).

4.5.2.3 Skalierbarkeitstests

Skalierbarkeitstests untersuchen die Fähigkeit eines Systems, zukünftige Effizienzanforderungen zu erfüllen, welche über den Gegenwärtigen liegen können. Ziel dieser Tests ist es zu beurteilen, ob das System wachsen kann (beispielsweise mit mehr Nutzern oder größeren Mengen von gespeicherten Daten), ohne zu scheitern oder die gegenwärtig spezifizierten Performanzanforderungen zu überschreiten. Sind die Grenzen der Skalierbarkeit bekannt, lassen sich Schwellenwerte definieren und in der Produktion überwachen, sodass bei bevorstehenden Problemen eine Warnung erfolgen kann. Außerdem kann die Produktionsumgebung durch entsprechende Hardware angepasst werden.

4.5.3 Performanztest planen

Neben den allgemeinen Planungsaspekten, welche in Abschnitt 4.2 beschrieben sind, können auch folgende Aspekte die Planung von Performanztests beeinflussen:

- Je nach verwendeter Testumgebung und zu testender Software (siehe Abschnitt 4.2.3) kann es für Performanztests erforderlich sein, dass das gesamte System implementiert ist, damit überhaupt effektiv getestet werden kann. In diesem Fall wird der Performanztest meist in der Systemteststufe eingeplant. Andere Performanztests, welche in der Komponententeststufe effektiv durchgeführt werden können, sollten auch auf dieser Stufe eingeplant werden.
- Im Allgemeinen ist es wünschenswert, dass erste Performanztests so früh wie möglich durchgeführt werden, selbst wenn noch keine produktionsähnliche Umgebung zur Verfügung steht. Diese frühen Tests können Performanzprobleme aufdecken (z.B. Leistungsengpässe) und das Projektrisiko reduzieren, indem sie zeitaufwändige Fehlerbehebungen in den späteren Softwareentwicklungsphasen oder nach Produktivgang vermeiden.
- Code-Reviews können Performanzprobleme identifizieren, insbesondere wenn sie auf Datenbankinteraktionen, Interaktionen von Komponenten und Fehlerbehandlung fokussiert sind. Es werden insbesondere Probleme mit „Wait/Retry-Logik“ und mit ineffizienten Abfragen gefunden. Diese Code-Reviews sollten frühzeitig im Softwarelebenszyklus eingeplant werden.
- Die Hardware sowie die Software und Netzwerkbandbreiten, welche für den Performanztest notwendig sind, sollten geplant und budgetiert werden. Der Bedarf orientiert sich in erster Linie an der zu erzeugenden Last, welche zum Beispiel die Zahl der simulierten virtuellen Nutzer und dem voraussichtlichen Volumen ihres Netzwerkverkehrs basiert. Wird dies bei der Planung nicht berücksichtigt, sind die Performanzmessungen nicht repräsentativ. Will man

beispielsweise die Anforderungen an die Skalierbarkeit einer stark frequentierten Internetseite verifizieren, kann dafür eine simulierte Nutzung durch Hunderttausende von virtuellen Nutzern nötig sein.

- Die Erzeugung der für den Performanztest benötigten Last kann sich erheblich auf die Kosten für die Beschaffung von Hardware und Werkzeugen auswirken. Diese müssen bei der Planung der Performanztests einkalkuliert werden, damit angemessene Mittel bereitgestellt werden.
- Die Kosten für die Erzeugung der für den Performanztest benötigten Last lassen sich minimieren, wenn die Testinfrastruktur gemietet wird. Dies kann beispielsweise bedeuten, dass für Performanzwerkzeuge Zusatzlizenzen gemietet werden, oder dass für die benötigte Hardware die Dienste von Fremddienstleistern (z.B. Cloud-Dienste) in Anspruch genommen werden. Werden diese Dienste genutzt, dann ist möglicherweise die für den Performanztest verfügbare Zeit begrenzt und muss umso sorgfältiger geplant werden.
- In der Planungsphase sollte auch sorgfältig geprüft werden, dass das geplante Performanzwerkzeug mit den Kommunikationsprotokollen kompatibel ist, die das zu testende System verwendet.
- Performanzbezogene Fehlerzustände haben oft erhebliche Auswirkungen auf die zu testende Software. Wenn die Anforderungen an die Performanz des Systems sehr wichtig sind, ist es meist sinnvoll, die Performanz der kritischen Komponenten zu testen (mittels Treibern und Platzhaltern), und nicht bis zu den Systemtests zu warten.

4.5.4 Spezifikation von Performanztests

Die Testspezifikationen der verschiedenen Performanztests (z.B. Last- und Stresstests) basieren auf definierten Nutzungsprofilen mit verschiedenen Formen des Nutzungsverhaltens bei der Interaktion mit der Anwendung. Für eine Anwendung kann es mehrere Nutzungsprofile geben.

Die Anzahl der Nutzer pro Nutzungsprofil lässt sich mit Monitorwerkzeugen bestimmen (wenn die tatsächliche oder eine vergleichbare Anwendung bereits zur Verfügung steht), oder durch eine Voraussage. Voraussagen können auf Algorithmen basieren oder vom Fachbereich stammen; sie sind besonders wichtig für die Spezifizierung der Nutzungsprofile bei Skalierbarkeitstests.

Nutzungsprofile sind Grundlage für die Art und Anzahl der Testfälle des geplanten Performanztests. In der Regel werden Testwerkzeuge eingesetzt, welche die Menge an simulierten bzw. „virtuellen“ Nutzern für das zu testende Nutzungsprofil generieren (siehe Abschnitt 6.3.2).

4.6 Ressourcennutzung

Im Standard ISO 9126 ist die Ressourcennutzung als ein Teilmerkmal des Qualitätsmerkmals der Effizienz enthalten. Tests der Ressourcennutzung bewerten die Verwendung von Systemressourcen (beispielsweise Speichernutzung, Festplattenkapazitäten, Bandbreiten im Netz, Verbindungen) anhand definierter Referenzdaten. Die Verwendung dieser Ressourcen wird sowohl bei normaler Systemauslastung gemessen als auch in Stresssituationen (z.B. bei hohen Transaktionsraten oder großen Datenmengen), um so zu bestimmen, ob eine ungewöhnliche Zunahme vorliegt.

So spielt beispielsweise bei eingebetteten Echtzeitsystemen die Speichernutzung (memory footprint) eine wichtige Rolle bei Performanztests. Wenn der „memory footprint“ das zulässige Maß überschreitet, dann hat das System möglicherweise zu wenig Speicher, um die erforderlichen Aufgaben im spezifizierten Zeitrahmen durchzuführen. Dadurch kann das System langsamer werden oder sogar abstürzen.

Außerdem kann auch die dynamische Analyse die Ressourcennutzung untersuchen (siehe Abschnitt 3.3.4) und Leistungsengpässe identifizieren.

4.7 Wartbarkeitstest

Ein wesentlich größerer Teil des Lebenszyklus einer Software entfällt auf die Phase, in der die Software gewartet wird. Nur ein vergleichsweise kleiner Teil entfällt auf die Phase der Softwareentwicklung. Bei Wartungstests werden die Änderungen an einem System im Betrieb oder die Auswirkungen einer geänderten Umgebung auf ein System im Betrieb getestet. Um sicherzustellen, dass die Wartung eines Systems möglichst effizient durchgeführt werden kann, untersuchen Wartbarkeitstests wie einfach der Programmcode analysiert, geändert und getestet werden kann.

Zu den typischen Wartbarkeitszielen der Stakeholder (z.B. Systemeigentümer oder -betreiber) gehören:

- Minimierung der Besitz- oder Betriebskosten der Software
- Minimierung der Ausfallzeiten für Softwarewartung

Die Wartbarkeitstests sollten in einer Teststrategie und/oder in der Testvorgehensweise enthalten sein, wenn einer oder mehrere der folgenden Faktoren zutreffen:

- Änderungen der Software sind wahrscheinlich, nachdem diese in Produktion gegangen ist (z.B. Fehlerbehebungen oder geplante Aktualisierungen).
- Nach Ansicht der Stakeholder überwiegt der Nutzen, der sich aus dem Erreichen der oben erwähnten Ziele für den Softwarelebenszyklus ergibt, die Kosten für die Durchführung der Wartbarkeitstests und die Einbringung der erforderlichen Änderungen.
- Das Risiko einer schlechten Wartbarkeit der Software (z.B. lange Reaktionszeiten nach Fehlern, die von Anwendern und/oder Kunden berichtet werden) rechtfertigt die Durchführung von Wartbarkeitstests.

Geeignete Verfahren für Wartbarkeitstests sind statische Analysen und Reviews, wie in Abschnitt 3.2 und 5.2 beschrieben. Mit den Wartbarkeitstests sollte begonnen werden, sobald die Entwurfsdokumente zur Verfügung stehen, und während des gesamten Zeitraums der Implementierung fortdauern. Da die Wartbarkeit in den Code und in die zugehörige Dokumentation der einzelnen Codekomponenten eingebaut ist, sollte die Wartbarkeit bereits frühzeitig im Lebenszyklus bewertet werden; es ist unnötig, damit zu warten, bis das System fertig ist und schon läuft.

Dynamische Wartbarkeitstests untersuchen vorrangig die dokumentierten Verfahren, die für die Wartung einer Anwendung entwickelt wurden (beispielsweise für ein Update der Software). Als Testfälle dienen Wartungsszenarien, um sicherzustellen, dass die geforderten Service Levels mit den dokumentierten Verfahren erreicht werden können. Diese Art des Testens ist besonders bei komplexen Infrastrukturen wichtig, bei denen mehrere Abteilungen/Unternehmen an den Supportverfahren beteiligt sind. Die Tests können Teil der betrieblichen Abnahmetests sein. [Web-1]

4.7.1 Analysierbarkeit, Modifizierbarkeit, Stabilität und Testbarkeit

Die Wartbarkeit eines Systems lässt sich auf unterschiedliche Arten messen: nach der Analysierbarkeit (d.h. dem Aufwand für die Diagnose von Problemen, die im System identifiziert wurden), nach der Modifizierbarkeit (d.h. dem Aufwand für Programmänderungen), und nach der Testbarkeit (d.h. der Aufwand für das Testen von Änderungen). Die Stabilität hängt in erster Linie von der Reaktion des Systems auf Änderungen ab. Systeme mit niedriger Stabilität zeigen nach jeder Änderung eine große Anzahl an Folgeproblemen. [ISO9126] [Web-1].

Der benötigte Aufwand für die Wartungsaufgaben hängt von mehreren Faktoren ab, wie z.B. von der verwendeten Softwaredesign-Methodik (beispielsweise objektorientiert) und von den verwendeten Programmierstandards.

Es ist zu beachten, dass der Begriff „Stabilität" in diesem Kontext nicht mit den Begriffen „Robustheit" und „Fehlertoleranz" verwechselt werden darf, die in Abschnitt 4.4.2 behandelt werden.

4.8 Portabilitätstest

Portabilitätstests untersuchen, wie einfach es ist, eine Software in ihre vorgesehene Umgebung zu übertragen, entweder bei der Erstinstallation oder aus einer bestehenden Umgebung. Bei Portabilitätstests werden Installierbarkeit, Koexistenz/Kompatibilität, Anpassbarkeit und Austauschbarkeit getestet.

Portabilitätstests können schon mit einzelnen Komponenten beginnen (z.B. Austauschbarkeit einer bestimmten Komponente wie beispielsweise Wechseln eines Datenbankmanagementsystems auf ein anderes) und werden erweitert, wenn mehr Code verfügbar wird. Die Installierbarkeit kann möglicherweise erst testbar sein, wenn alle Komponenten des Produkts funktionieren. Da die Portabilität beim Systementwurf berücksichtigt und in das Produkt eingebaut werden muss, ist dieses Qualitätsmerkmal schon in den frühen Phasen des Systementwurfs und der Systemarchitektur wichtig. Reviews des Entwurfs und der Architektur können ein sehr produktives Mittel sein, um mögliche Portabilitätsanforderungen und -probleme zu identifizieren (z.B. Abhängigkeit von einem bestimmten Betriebssystem).

4.8.1 Installationstest

Installationstests testen die Installation von Software und die dokumentierten Installationsanleitungen für die Installation der Software in einer Zielumgebung. Das kann Software mit einschließen, mit der ein Betriebssystem auf einem Prozessor installiert wird, oder ein Installationsprogramm (Wizard), mit dem ein Produkt auf einem Client-PC installiert wird.

Typische Ziele von Installationstests sind:

- Validieren, dass die Software erfolgreich installiert werden kann, indem man die Anweisungen des Installationshandbuchs befolgt (einschließlich der Ausführung von Installationsskripten) oder einen Installations-Wizard nutzt. Dabei sind auch die unterschiedlichen Optionen für mögliche Hardware-/Software-Konfigurationen sowie für verschiedene Installationsstufen (Erstinstallation oder Update) zu testen.
- Testen, ob die Installationssoftware richtig mit Fehlerwirkungen umgeht, die bei der Installation auftreten (beispielsweise wenn bestimmte DLLs nicht geladen werden), ohne das System in einem undefinierten Zustand zu lassen (beispielsweise mit nur teilweise installierter Software oder inkorrekten Systemkonfigurationen).
- Testen, ob sich eine nur teilweise Installation/Deinstallation der Software abschließen lässt.
- Testen, ob ein Installations-Wizard eine ungültige Hardware-Plattform oder Betriebssystem-Konfigurationen erfolgreich identifizieren kann.
- Messen, ob der Installationsvorgang im spezifizierten Zeitrahmen oder mit einer geringeren als der spezifizierten Anzahl von Schritten abgeschlossen werden kann.
- Validieren, dass sich eine frühere Version installieren (die aktuelle Software „downgraden") oder die Software deinstallieren lässt.

Nach dem Installationstest wird normalerweise die Funktionalität getestet, um Fehlerzustände aufzudecken, die durch die Installation eingeschleust worden sein können (beispielsweise inkorrekte Konfigurationen, nicht mehr verfügbare Funktionen). Parallel zu den Installationstests laufen in der Regel Benutzbarkeitstest (beispielsweise zur Validierung, dass die Anwender bei der Installation verständliche Anweisungen und Feedback/Fehlermeldungen erhalten).

4.8.2 Koexistenz-/Kompatibilitätstest

Computersysteme, die nicht miteinander interagieren, sind dann kompatibel, wenn sie in derselben Umgebung (beispielsweise auf derselben Hardware) ausgeführt werden können, ohne sich

gegenseitig zu beeinflussen (beispielsweise durch Ressourcenkonflikte). Kompatibilitätstests sollten dann durchgeführt werden, wenn eine neue Software oder eine Aktualisierung in einer neuen Umgebung (beispielsweise auf einem Server) installiert wird, in der bereits Anwendungen installiert sind.

Kompatibilitätsprobleme können entstehen, wenn eine Anwendung als die einzige in einer Umgebung getestet wurde (in der Inkompatibilitäten nicht erkennbar sind) und wenn diese Anwendung dann in einer anderen (beispielsweise der Produktionsumgebung) installiert wird, in der bereits andere Anwendungen installiert sind.

Typische Ziele von Kompatibilitätstests sind:

- Bewertung möglicher negativer Auswirkungen auf die Funktionalität, wenn Anwendungen in derselben Umgebung geladen werden (beispielsweise Konflikte in der Ressourcennutzung, wenn mehrere Anwendungen auf demselben Server laufen)
- Bewertung der Auswirkungen auf jede Anwendung, die sich aus Modifikationen oder dem Installieren einer aktuelleren Betriebssystemversion ergeben

Kompatibilitätsprobleme sollten analysiert werden, wenn die vorgesehene Zielumgebung geplant wird; die Kompatibilitätstests werden jedoch normalerweise erst nach dem erfolgreichen Abschluss der System- und Anwenderabnahmetests durchgeführt.

4.8.3 Anpassbarkeitstests

Anpassbarkeitstests sollen zeigen, ob eine bestimmte Anwendung in allen geplanten Zielumgebungen korrekt funktioniert (Hardware, Software, Middleware, Betriebssystem usw.). Ein anpassungsfähiges System ist daher ein offenes System, das sein Verhalten an Änderungen in der Umgebung oder in eigenen Teilsystemen anpassen kann. Für die Spezifikation der Anpassbarkeitstests müssen Kombinationen der beabsichtigten Zielumgebungen identifiziert, konfiguriert und dem Testteam zur Verfügung gestellt werden. Diese Umgebungen werden dann mit ausgewählten funktionalen Testfällen getestet, bei denen die verschiedenen Komponenten der Anwendung in der Testumgebung geprüft werden.

Anpassbarkeit kann sich darauf beziehen, dass sich die Software durch Ausführung eines definierten Verfahrens auf verschiedene spezifizierte Umgebungen portieren lässt. Beim Testen kann dieses Verfahren bewertet werden.

Anpassbarkeitstests können zusammen mit Installationstests durchgeführt werden. Normalerweise finden anschließend funktionale Tests statt, um Fehlerzustände aufzudecken, die durch das Anpassen der Software an die andere Umgebung entstanden sein können.

4.8.4 Austauschbarkeitstest

Austauschbarkeitstests sind darauf fokussiert, die Austauschbarkeit von Softwarekomponenten eines Systems gegen andere Komponenten zu testen. Das ist besonders relevant bei Systemen, die kommerzielle Standardsoftware für bestimmte Systemkomponenten verwenden.

Austauschbarkeitstests können parallel zu den funktionalen Integrationstests durchgeführt werden, wenn alternative Komponenten für die Integration in das Gesamtsystem verfügbar sind. Die Austauschbarkeit lässt sich in technischen Reviews oder Inspektionen der Systemarchitektur oder des Systementwurfs bewerten, bei denen Wert auf eine klare Definition der Schnittstellen zu möglichen Austauschkomponenten gelegt wird.

5. Reviews - 165 Minuten

Begriffe

Anti-Pattern

Lernziele für Reviews

5.1 Einführung

TTA 5.1.1 (K2) Sie können erklären, warum die Vorbereitung des Reviews für den Technical Test Analyst wichtig ist

5.2 Checklisten in Reviews verwenden

TTA 5.2.1 (K4) Sie können einen Architekturentwurf analysieren und Probleme anhand einer im Lehrplan enthaltenen Checkliste identifizieren

TTA 5.2.2 (K4) Sie können ein Stück Programmcode oder Pseudo-Code analysieren und Probleme anhand einer im Lehrplan enthaltenen Checkliste identifizieren

5.1 Einführung

Technical Test Analysten müssen aktiv am Review-Prozess teilnehmen und ihre individuelle Perspektive einbringen. Sie sollten ein formales Review-Training erhalten haben, damit sie ihre jeweiligen Rollen bei technischen Review-Prozessen besser verstehen. Alle Teilnehmerinnen und Teilnehmer müssen vom Nutzen gut durchgeführter technischer Reviews überzeugt sein. Wenn sie ordnungsgemäß durchgeführt werden, leisten Reviews nicht nur den größten einzelnen, sondern auch den kosteneffektivsten Beitrag zur gelieferten Qualität. Für eine vollständige Beschreibung technischer Reviews, einschließlich zahlreicher Review-Checklisten, siehe [Wiegers02]. Technical Test Analysten nehmen normalerweise an technischen Reviews und Inspektionen teil, bei denen sie Gesichtspunkte des Systemverhaltens beisteuern, die von Entwicklern oft übersehen werden. Außerdem spielen Technical Test Analysten eine wichtige Rolle bei Definition, Anwendung und Pflege von Review-Checklisten und bei der Bereitstellung von Informationen über den Schweregrad von Fehlerzuständen.

Unabhängig von der Art des Reviews, das durchgeführt wird, muss der Technical Test Analyst genug Zeit für die Vorbereitung bekommen. Diese Zeit wird benötigt, um das Arbeitsergebnis zu prüfen, um die Dokumente, auf die verwiesen wird, zu prüfen und zu verifizieren, dass das Arbeitsergebnis mit diesen konsistent ist, und um zu bestimmen, was im Arbeitsergebnis fehlt. Ohne angemessene Vorbereitungszeit könnte das Review auf die Überarbeitung des Dokuments beschränkt sein, anstatt ein echtes Review zu sein. Zu einem guten Review gehört es, die Inhalte zu verstehen, zu bestimmen, was fehlt, und zu verifizieren, dass das beschriebene Produkt mit anderen, bereits entwickelten Produkten konsistent ist (bzw. mit anderen Produkten, die derzeit entwickelt werden). Beispiel: Beim Review eines Stufentestkonzepts für den Integrationstest muss der Technical Test Analyst auch die Objekte berücksichtigen, die integriert werden sollen. Sind sie bereit für die Integration? Gibt es Abhängigkeiten, die dokumentiert werden müssen? Sind Daten verfügbar, um die Integrationsstellen zu testen? Ein Review konzentriert sich nicht allein auf das Arbeitsergebnis, das geprüft wird, sondern es muss auch die Interaktion des Review-Gegenstandes mit anderen Objekten des Systems berücksichtigt werden.

Es passiert leicht, dass sich der Autor eines im Review geprüften Arbeitsergebnisses kritisiert fühlt. Der Technical Test Analyst sollte bei Reviewbemerkungen bedenken, dass die Zusammenarbeit mit dem Autor letztlich dazu dient, das bestmögliche Arbeitsergebnis zu erzielen. Mit einer solchen Grundhaltung werden die Kommentare konstruktiv formuliert sein, und sie werden sich am Review-Gegenstand orientieren und nicht am Autor. Beispiel: Wenn eine Aussage zweideutig ist, dann ist es besser zu sagen „Ich verstehe nicht, was ich testen soll, um zu verifizieren, ob diese Anforderung korrekt implementiert wurde. Können Sie mir helfen, das besser zu verstehen?“ anstatt „Die Anforderung ist zweideutig, die wird keiner verstehen können“.

Aufgabe des Technical Test Analysten beim Review ist es, sicherzustellen, dass die im Arbeitsergebnis gelieferten Informationen ausreichend sind, um das Testen zu unterstützen. Wenn Informationen nicht vorhanden, nicht klar und eindeutig sind, oder wenn sie nicht detailliert genug sind, dann ist dies möglicherweise ein Fehler, der vom Autor korrigiert werden muss. Eine positive statt eine kritische Grundhaltung trägt dazu bei, dass Kommentare besser entgegengenommen und die Sitzungen produktiver werden.

5.2 Checklisten in Reviews verwenden

Checklisten werden bei Reviews verwendet, damit die Teilnehmer angehalten werden, bestimmte Punkte im Laufe des Reviews zu prüfen. Sie können auch dazu beitragen, Reviews zu entpersonalisieren, z.B. mit der Aussage, „Dies ist dieselbe Checkliste, die für alle Reviews verwendet wird, nicht nur für das vorliegende Arbeitsergebnis.“ Checklisten können allgemein gehalten sein und

für alle Reviews verwendet werden, oder sie können sich speziell mit bestimmten Qualitätsmerkmalen oder Themenbereichen befassen. Beispiel: Mit einer allgemeinen Checkliste könnte verifiziert werden, ob die Begriffe „soll" und „sollte" richtig verwendet sind, oder ob die Formatierung und ähnliche Elemente korrekt bzw. konform sind. Eine spezifische Checkliste könnte sich mit Sicherheit oder Performanz des Systems befassen.

Die nützlichsten Checklisten sind die, die allmählich über einen Zeitraum von einem einzelnen Unternehmen entwickelt wurden, da diese Checklisten folgendes wiedergeben:

- Art des Produkts
- Örtliches Entwicklungsumfeld
  - Personal
  - Werkzeuge
  - Prioritäten
- Historie früherer Erfolge und Fehlerzustände
- Spezifische Themen (z.B. Performanz, Sicherheit)

Checklisten sollten an das Unternehmen und möglicherweise sogar an das Projekt angepasst werden. Die in diesem Kapitel beschriebenen Checklisten sind nur als Beispiele gedacht.

Manche Unternehmen erweitern das gängige Konzept von Software-Checklisten um sogenannte „Anti-Patterns" (Antimuster), die sich auf häufige Fehler, schlechte Verfahren und ineffektive Praktiken beziehen. Der Begriff ist abgeleitet vom Konzept des „Design-Pattern" (Entwurfsmuster), bei denen es sich um wiederverwendbare Vorlagen zur Problemlösung handelt, die sich in praktischen Situationen als effektiv bewährt haben [Gamma94]. Ein Anti-Pattern ist demzufolge ein häufig anzutreffender Fehler, der oft als „nützliche Abkürzung" gedacht war.

Es ist zu beachten, dass wenn eine Anforderung nicht testbar ist, ein Fehlerzustand in der Anforderung vorliegt. Nicht testbar bedeutet, dass die Anforderung so spezifiziert ist, dass der Technical Test Analyst nicht bestimmen kann, wie sie getestet werden soll. Beispiel: Die Anforderung „Die Software soll schnell sein" ist nicht testbar. Wie soll der Technical Test Analyst bestimmen, ob eine Software schnell ist? Wenn die Anforderung stattdessen festlegt „Die Software muss eine maximale Antwortzeit von drei Sekunden bei bestimmten Lastbedingungen haben", dann ist die Testbarkeit dieser Anforderung erheblich besser (vorausgesetzt, dass die „bestimmten Lastbedingungen" näher spezifiziert sind, z.B. Anzahl gleichzeitiger Nutzer, Aktivitäten der Nutzer). Es handelt sich außerdem um eine übergreifende Anforderung, die bei einer komplexen Anwendung viele einzelne Testfälle hervorbringen könnte. Auch ist die Verfolgbarkeit von dieser Anforderung zu den Testfällen kritisch; wenn sich die Anforderung ändert, müssten alle Testfälle geprüft und bei Bedarf aktualisiert werden.

5.2.1 Architekturreviews

Softwarearchitektur betrifft die fundamentale Organisationsstruktur eines Systems und seiner Komponenten, die Beziehung der Komponenten untereinander, zur Systemumgebung, und den Grundsätzen, die für Systementwurf und -entwicklung gelten [ANSI/IEEE Std 1471-2000], [Bass03].

Checklisten, die für ein Review der Architektur verwendet werden, könnten beispielsweise verifizieren, dass die folgenden Aspekte korrekt implementiert sind (zitiert aus [Web-3]):

- „Verbindungspooling" – den für die Ausführung benötigten Zeitaufwand, der mit dem Aufbau von Datenbankverbindungen zusammenhängt, reduzieren und einen gemeinsamen Pool von Verbindungen schaffen
- Lastverteilung – gleichmäßige Verteilung der Last zwischen einer Menge von Ressourcen
- Verteilte Verarbeitung
- Caching – eine lokale Kopie der Daten verwenden, um die Zugriffszeiten zu reduzieren
- Verzögerte Instantiierung (lazy instantiation)

- Parallelität von Transaktionen
- Prozesstrennung zwischen OLTP (Online Transactional Processing) und OLAP (Online Analytical Processing)
- Replikation von Daten

Weitere Details, die jedoch nicht prüfungsrelevant sind, finden Sie in [Web-4]. Dort wird auf ein Veröffentlichung verwiesen, das 117 Checklisten aus 24 untersuchten Quellen enthält. Es werden unterschiedliche Kategorien von Checklisten-Punkten behandelt und Beispiele für gute Checklisten-Punkte gegeben und auch für solche, die vermieden werden sollten.

5.2.2 Code-Reviews

Checklisten für Code-Reviews sind zwangsläufig sehr detailliert, und genau wie bei den Checklisten für Architektur-Reviews sind sie dann am nützlichsten, wenn sie spezifisch auf eine Programmiersprache, ein Projekt und ein Unternehmen zugeschnitten sind. Die Einbeziehung von Anti-Pattern auf Codeebene ist hilfreich, besonders für weniger erfahrene Softwareentwickler.

Checklisten für Code-Reviews können die folgenden sechs Bereiche abdecken (basiert auf [Web-5]):

1. Struktur
   - Ist das Design im Code vollständig und korrekt implementiert?
   - Entspricht der Code den einschlägigen Programmierkonventionen?
   - Ist der Code gut strukturiert, konsistent im Stil und einheitlich formatiert?
   - Gibt es Prozeduren, die nicht aufgerufen oder nicht benötigt werden, oder gibt es unerreichbaren Code?
   - Gibt es im Code Überbleibsel vom Testen (Platzhalter, Testroutinen)?
   - Kann Code durch Aufrufe externer wiederverwendbarer Komponenten oder Bibliotheksfunktionen ersetzt werden?
   - Gibt es Codeblöcke die sich wiederholen, und die in einer einzigen Prozedur zusammengefasst werden könnten?
   - Ist die Speichernutzung effizient?
   - Wird Symbolik benutzt anstatt „magische" Konstanten oder String-Konstanten?
   - Gibt es Module, die übermäßig komplex sind, und daher umstrukturiert oder in mehrere Module aufgeteilt werden sollten?

2. Dokumentation
   - Ist der Code eindeutig, ausreichend dokumentiert und in einem leicht zu wartenden Stil kommentiert?
   - Passen alle Kommentare zum Code?
   - Entspricht die Dokumentation den geltenden Standards?

3. Variablen
   - Sind alle Variablen richtig mit sinnvollen, konsistenten und eindeutigen Namen definiert?
   - Gibt es redundante oder ungenutzte Variablen?

4. Arithmetische Operationen
   - Wird im Code vermieden, dass Gleitkommazahlen auf Gleichheit geprüft werden?
   - Verhindert der Code systematisch Rundungsfehler?
   - Vermeidet der Code Additionen und Subtraktionen mit sehr unterschiedlich großen Zahlen?
   - Werden Teiler (Divisoren) auf Null oder auf Rauschen getestet?

5. Schleifen und Zweige

- Sind alle Schleifen, Verzweigungen und Logikkonstrukte vollständig, korrekt und richtig verschachtelt?
- Werden in IF-ELSEIF Ketten die häufigsten Fälle zuerst getestet?
- Werden alle Fälle in einem IF-ELSEIF oder CASE Block behandelt, einschließlich der ELSE- oder DEFAULT-Klauseln?
- Gibt es für jede Case-Anweisung einen Standardwert?
- Sind die Abbruchbedingungen von Schleifen offensichtlich und immer erreichbar?
- Sind die Index-Variablen oder Teilskripte unmittelbar vor der Schleife richtig initialisiert?
- Können irgendwelche Anweisungen, die sich innerhalb der Schleife befinden, auch außerhalb der Schleife platziert werden?
- Wird vermieden, dass der Code in der Schleife die Index-Variable manipuliert, oder diese nach Beendigung der Schleife verwendet?

6. Defensive Programmierung

- Werden Indizes, Zeiger und Teilskripte mit Bezug auf Arrays, Datensätze oder Dateigrenzen getestet?
- Wird getestet, ob importierte Daten und Eingabeparameter gültig und vollständig sind?
- Sind alle Ausgangsvariablen zugewiesen?
- Wird in jeder Anweisung das richtige Datenelement verarbeitet?
- Wird jeder zugewiesene Speicher freigegeben?
- Wird Code zur Behandlung von Zeitüberläufen oder Fehlerbedingungen für Zugriffe auf externe Geräte verwendet?
- Wird vor einem Zugriff auf Dateien geprüft, ob diese existieren?
- Sind alle Dateien und Geräte in einem korrekten Zustand, wenn das Programm beendet wird?

Für weitere Beispiele von Checklisten, die bei Code-Reviews in unterschiedlichen Teststufen verwendet werden, siehe [Web-6].

6. Testwerkzeuge und Automatisierung - 195 Minuten

Begriffe

datengetriebenes Testen, Debugging-Werkzeug, Fehlereinpflanzungswerkzeug, Hyperlink-Testwerkzeug, Mitschnittwerkzeug, schlüsselwortgetriebener Test, Performanztestwerkzeug, statischer Analysator, Testausführungswerkzeug, Testmanagementwerkzeug

Lernziele für Testwerkzeuge und Automatisierung

6.1 Integration und Informationsaustausch zwischen Werkzeugen

TTA-6.1.1 (K2) Sie können die technischen Aspekte erläutern, die zu berücksichtigen sind, wenn mehrere Werkzeuge zusammen verwendet werden

6.2 Ein Testautomatisierungsprojekt definieren

TTA-6.2.1 (K2) Sie können die Aktivitäten des Technical Test Analysten in Zusammenhang mit dem Aufsetzen eines Testautomatisierungsprojekts zusammenfassen

TTA-6.2.2 (K2) Sie können die Unterschiede zwischen datengetriebener und schlüsselwort-getriebener Testautomatisierung zusammenfassen

TTA-6.2.3 (K2) Sie können die technischen Probleme zusammenfassen, die häufig dafür verantwortlich sind, wenn Testautomatisierungsprojekte nicht die geplante Rentabilität erzielen

TTA-6.2.4 (K3) Sie können eine Schlüsselwort-Tabelle erstellen, die auf einem vorgegebenen Geschäftsprozess basiert

6.3 Spezifische Testwerkzeuge

TTA-6.3.1 (K2) Sie können den Zweck von Werkzeugen zur Fehlereinpflanzung und zum Fehlereinfügen zusammenfassen

TTA-6.3.2 (K2) Sie können die wichtigsten Eigenschaften von Performanztestwerkzeugen und Monitoren sowie die Themen rund um deren Implementierung zusammenfassen

TTA-6.3.3 (K2) Sie können die allgemeinen Verwendungszwecke von Werkzeugen für das webbasierte Testen erläutern

TTA-6.3.4 (K2) Sie können erläutern, wie Werkzeuge das Konzept des modellbasierten Testens unterstützen

TTA-6.3.5 (K2) Sie können darlegen, wie Werkzeuge eingesetzt werden können, um den Komponententest und den Build-Prozess zu unterstützen

6.1 Integration und Informationsaustausch zwischen Werkzeugen

Während die Auswahl und Integration von Werkzeugen zu den Aufgaben des Testmanagers gehört, kann der Technical Test Analyst die Aufgabe erhalten, die Integration eines oder mehrerer Werkzeuge zu prüfen und sicherzustellen, dass die Daten aus unterschiedlichen Testbereichen (wie z.B. aus der statischen Analyse, der automatisierten Testdurchführung und dem Konfigurationsmanagement) richtig verfolgt werden. Je nach den Programmierfähigkeiten des Technical Test Analysten kann dieser auch bei der Erstellung des Codes mitwirken, der Werkzeuge miteinander verbindet, die sich nicht von selbst miteinander integrieren.

Ein ideales Toolset verhindert, dass Informationen über die einzelnen Werkzeuge hinweg dupliziert werden. Es ist aufwändiger und zudem fehleranfälliger, wenn Testdurchführungsskripte sowohl in der Testmanagement-Datenbank als auch im Konfigurationsmanagementsystem gespeichert werden. Besser wäre, wenn das Testmanagementsystem eine Komponente für das Konfigurations-management hätte, oder wenn diese mit dem im Unternehmen bereits verwendeten Konfigurationsmanagementwerkzeug integriert werden könnte. Gut miteinander integrierte Fehlerverfolgungs- und Testmanagementwerkzeuge ermöglichen es dem Tester, während der Testfalldurchführung einen Fehlerbericht zu erstellen, ohne dafür das Testmanagementwerkzeug zu verlassen. Gut integrierte statische Analysatoren sollten entdeckte Abweichungen und Warnungen direkt an das Fehlermanagementsystem berichten können (dies sollte allerdings konfigurierbar sein, da möglicherweise sehr viele Warnungen generiert werden).

Die Beschaffung von Testwerkzeugen vom selben Anbieter bedeutet nicht zwangsläufig, dass die Werkzeuge in einem angemessenen Umfang miteinander arbeiten. Wenn über Methoden zur Integration von Werkzeugen nachgedacht wird, dann sind datenzentrierte Ansätze vorzuziehen. Die Daten müssen ohne manuelle Intervention, zeitnah, mit garantierter Richtigkeit (einschließlich Fehlerbehandlung) ausgetauscht werden können. Es ist zwar hilfreich, eine konsistente Bedienung zu bieten, bei der Werkzeugintegration sollten jedoch Erfassen, Speichern, Schutz und Präsentation von Daten im Vordergrund stehen.

Unternehmen sollten die Kosten einer Automatisierung des Informationsaustausches bewerten und mit dem Risiko vergleichen, dass Informationen verloren gehen, oder dass die Synchronisation der Daten leidet, weil manuelle Eingriffe notwendig sind. Da die Integration von Werkzeugen teuer oder schwierig sein kann, sollte dieser Aspekt in der Werkzeugstrategie vorrangig berücksichtigt werden.

Einige integrierte Entwicklungsumgebungen (engl. Integrated Development Environment oder IDE) vereinfachen die Integration von Werkzeugen, die in derselben Umgebung arbeiten können. Die Werkzeuge, die sich das Rahmenwerk teilen, sehen dann einheitlich aus und lassen sich einheitlich benutzen. Allerdings garantiert eine ähnliche Benutzerschnittstelle noch nicht, dass der Informationsaustausch zwischen den Komponenten problemlos läuft. Eine komplette Integration muss möglicherweise zuerst noch programmiert werden.

6.2 Ein Testautomatisierungsprojekt definieren

Um kosteneffektiv zu sein, müssen Entwurf und Architektur von Testwerkzeugen und insbesondere von Testautomatisierungswerkzeugen mit großer Sorgfalt erstellt werden. Wenn eine Testautomatisierungsstrategie ohne solide Systemarchitektur implementiert wird, führt das in der Folge meist zu Toolsets mit Werkzeugen, die teuer zu warten sind, die ihren Zweck nicht ausreichend erfüllen, und die die angestrebte Rentabilität nicht erzielen.

Ein Testautomatisierungsprojekt sollte als ein Softwareentwicklungsprojekt verstanden werden. Dafür sind Architekturdokumente, detaillierte Entwurfsdokumente, Reviews des Entwurfs und des Codes,

Komponenten- und Komponentenintegrationstests, sowie ein abschließende Systemtest erforderlich. Beim Testen kann es zu unnötigen Verzögerungen oder Komplikationen kommen, wenn der verwendete Testautomatisierungscode instabil oder ungenau ist. In Zusammenhang mit der Testautomatisierung führt der Technical Test Analyst mehrere Aktivitäten durch. Dazu gehören:

- Bestimmen, wer für die Testausführung verantwortlich sein wird
- Das am besten geeignete Werkzeug für das Unternehmen, den Zeitplan, die Fähigkeiten im Team, die Wartungsanforderungen, usw. auswählen (dies kann bedeuten, ein eigenes Werkzeug selber zu entwickeln anstatt eines zu beschaffen)
- Die Anforderungen für die Schnittstellen zwischen dem Automatisierungswerkzeug und anderen Werkzeugen (z.B. Testmanagementwerkzeug, Fehlermanagementwerkzeug) definieren
- Die Vorgehensweise für die Automatisierung auswählen, d.h. schlüsselwortgetriebene oder datengetriebene Vorgehensweise (siehe Abschnitt 6.2.1)
- In Zusammenarbeit mit dem Testmanager die Kosten für Implementierung (einschließlich Schulung) schätzen
- Die Zeitplanung für das Automatisierungsprojekt erstellen und Zeit für die Wartung einzuplanen
- Die Test Analysten und Business Analysten darin schulen, wie die Daten für die Automatisierung zu verwenden und zu liefern sind
- Bestimmen, wie die automatisierten Tests ausgeführt werden
- Bestimmen, wie die Testergebnisse der automatisierten Tests mit denen der manuellen Tests kombiniert werden

Diese Aktivitäten und die getroffenen Entscheidungen beeinflussen die Skalierbarkeit und die Wartbarkeit der Automatisierungslösung. Es muss ausreichend Zeit investiert werden, um die Optionen zu prüfen, verfügbare Werkzeuge und Technologien zu untersuchen und die Zukunftspläne des Unternehmens zu verstehen. Manche dieser Aktivitäten erfordern genauere Untersuchungen als andere, insbesondere im Entscheidungsprozess. Diese Aktivitäten werden in den nachfolgenden Abschnitten ausführlicher behandelt.

6.2.1 Die Vorgehensweise für die Automatisierung auswählen

Testautomatisierung beschränkt sich nicht auf Eingaben über die Benutzerschnittstelle. Es gibt Werkzeuge, die das automatisierte Testen auf API-Ebene durch eine Command Line Interface (CLI) und sonstige Schnittstellen der zu testenden Software unterstützen. Eine der ersten Entscheidungen, die der Technical Test Analyst treffen muss ist, welche die effektivste Schnittstelle ist, die für die Testautomatisierung verwendet werden soll.

Eine Schwierigkeit beim Testen über die grafische Benutzerschnittstelle ist, dass diese sich im Zuge der Softwareentwicklung öfters ändert. Je nach Design des Testautomatisierungscodes kann dies einen erheblichen Wartungsaufwand zur Folge haben. Beispiel: Wird die Mitschnittfunktion eines Testautomatisierungswerkzeugs verwendet, kann dies zur Folge haben, dass die automatisierten Testfälle (oft als Testskripte bezeichnet) nicht mehr korrekt ausgeführt werden können, wenn sich die Benutzerschnittstelle ändert. Das liegt daran, dass das aufgezeichnete Skript Interaktionen mit den grafischen Objekten erfasst, während der Tester die Software manuell ausführt. Ändern sich die Objekte, auf die zugegriffen wird, müssen die aufgezeichneten Skripte ebenfalls aktualisiert und an die Änderungen angepasst werden.

Mitschnittwerkzeuge können als geeignete Ausgangsbasis für die Entwicklung von Automatisierungsskripten dienen. Der Tester zeichnet eine Testsitzung auf, und das aufgezeichnete Skript wird dann modifiziert, um die Wartbarkeit zu verbessern (indem beispielsweise Abschnitte im aufgezeichneten Skript durch wiederverwendbare Funktionen ersetzt werden).

Je nach Art der Software, die getestet wird, können die für die einzelnen Tests verwendeten Daten unterschiedlich, die ausgeführten Testschritte aber fast identisch sein (z.B. wenn beim Testen der Fehlerbehandlung eines Eingabefelds mehrere ungültige Werte eingegeben und die jeweils ausgegebenen Fehlermeldungen überprüft werden). Es ist nicht effizient, für jeden dieser Werte ein automatisiertes Testskript zu entwickeln und zu warten. Üblicherweise wird dieses Problem gelöst indem die Daten aus den Skripts herausgenommen und extern gespeichert werden, z.B. in einem Tabellenblatt oder einer Datenbank. Es werden Funktionen erstellt, die auf die spezifischen Daten für jede Ausführung des Testskripts zugreifen; so kann mit einem einzigen Skript eine Menge von Testdaten abgearbeitet werden, die die Eingabewerte und die erwarteten Ergebniswerte liefern (z.B. ein Wert, der in einem Textfeld angezeigt wird, oder eine Fehlermeldung). Dies wird als datengetriebene Vorgehensweise bezeichnet. Dabei wird ein Testskript entwickelt das die gelieferten Daten verarbeitet, und außerdem ein Testrahmen und die Infrastruktur die benötigt werden, um die Ausführung des Skripts oder der Menge von Skripten zu unterstützen. Die eigentlichen Daten, die im Tabellenblatt oder der Datenbank gespeichert sind, werden von Test Analysten erstellt, die sich mit der Geschäftslogik der Software auskennen. Diese Arbeitsteilung ermöglicht es denjenigen, die für die Entwicklung der Testskripte zuständig sind (z.B. dem Technical Test Analysten), sich auf die Implementierung intelligenter Automatisierungsskripte zu konzentrieren, während der Test Analyst Eigentümer des eigentlichen Tests bleibt. In den meisten Fällen wird der Test Analyst für die Ausführung der Testskripte verantwortlich sein, nachdem die Automatisierung implementiert und getestet wurde.

Die sogenannte schlüsselwortgetriebene oder aktionswortgetriebene Vorgehensweise geht einen Schritt weiter und trennt auch die Aktion, die mit den gelieferten Daten durchgeführt werden soll, vom Testskript [Buwalda01]. Um die durchzuführenden Aktionen vom Testskript zu trennen, erstellen Experten des Geschäftsbereichs (z.B. Test Analysten) eine abstrakte Metasprache, die eher die Aktionen beschreibt, als direkt ausführbar zu sein. Jede Anweisung dieser Metasprache beschreibt einen Geschäftsprozess (oder Teile davon), der getestet werden kann. Beispiel: Schlüsselworte von Geschäftsprozessen könnten sein: „Anmelden“, „Benutzer_anlegen“ oder „Benutzer_löschen“. Ein Schlüsselwort beschreibt eine abstrakte Aktion, die in der Anwendungsdomäne durchgeführt werden kann. Konkretere Aktionen bezeichnen die Interaktion mit der Softwareschnittstelle selbst, wie z.B. „Schaltfläche_betätigen“, „Aus_Liste_auswählen“ oder „Baum_ durchlaufen“. Diese können zum Testen der GUI-Funktionen verwendet werden, welche nicht genau zu den vorhandenen Schlüsselwörtern des Geschäftsprozesses passen.

Sobald die Schlüsselwörter und Daten für die Testskripte vorliegen, setzt der Automatisierungsspezialist (z.B. Technical Test Analyst) die geschäftsprozessbezogenen Schlüsselwörter und untergeordneten Aktionen in ausführbaren Testautomatisierungscode um. Die Schlüsselwörter und Aktionen können, zusammen mit den für den Test vorgesehenen Daten, in Tabellenkalkulationsprogrammen gespeichert, oder direkt in spezielle Werkzeuge eingegeben werden, die die schlüsselwortgetriebene Testautomatisierung unterstützen. Der Testautomatisierungsrahmen implementiert die Schlüsselwörter als eine Menge von einer oder mehreren ausführbaren Funktionen oder Skripten. Die Werkzeuge lesen die Testfälle mit den Schlüsselwörtern und rufen die entsprechenden Testfunktionen oder Testskripte auf, die diese implementieren. Die ausführbaren Skripte sind hochgradig modular implementiert, damit sie bestimmten Schlüsselwörtern leicht zugeordnet werden können. Für die Implementierung der modularen Testskripte sind Programmierkenntnisse erforderlich.

Diese klare Aufteilung der Kenntnisse über Geschäftslogik und der tatsächlichen Programmierung zur Implementierung der Testskripte sorgt für die effektivste Nutzung der Testressourcen. Der Technical Test Analyst kann in seiner Rolle als Testautomatisierungsspezialist seine Programmierfähigkeiten effektiv einsetzen, ohne dass spezifische Kenntnisse vieler Geschäftsbereiche erforderlich sind.

Durch die Trennung des Codes von den sich verändernden Daten isoliert man die Testautomatisierung von den Änderungen. Dies verbessert die Wartbarkeit des Codes insgesamt und steigert die Rentabilität für die Testautomatisierung.

Beim Entwerfen der Testautomatisierung müssen Fehlerwirkungen und Ausfälle der Software antizipiert und behandelt werden. Der Automatisierungsspezialist muss festlegen, was die Software tun sollte, wenn eine Fehlerwirkung auftritt. Sollte die Fehlerwirkung aufgezeichnet und die Tests fortgesetzt werden? Sollten die Tests abgebrochen werden? Kann die Fehlerwirkung durch eine bestimmte Aktion behandelt werden (z.B. durch die Betätigung einer Schaltfläche in einer Dialogbox) oder vielleicht durch eine Verzögerung im Test? Nicht behandelte Fehlerwirkungen in der Software können die Ergebnisse der nachfolgenden Tests unbrauchbar machen und Probleme mit dem Test verursachen, der zum Zeitpunkt des Ereignisses ausgeführt wurde.

Außerdem muss der Zustand des Systems zu Beginn und am Ende der Tests berücksichtigt werden. Es kann erforderlich sein, dass das System nach Abschluss der Testausführung automatisch wieder in einen definierten Zustand zurückgesetzt wird. Dadurch wird ermöglicht, dass eine automatisierte Testsuite wiederholt ausgeführt werden kann, ohne dass das System manuell zurückgesetzt werden muss. Dazu muss die Testautomatisierung beispielsweise Daten, die erstellt wurden, wieder löschen, oder den Status von Datensätzen in einer Datenbank ändern. Der Automatisierungsrahmen sollte sicherstellen, dass am Ende der Tests ein korrekter Abschluss erfolgt ist (z.B. Abmelden, nachdem die Tests abgeschlossen sind).

6.2.2 Geschäftsprozesse für die Automatisierung modellieren

Um eine schlüsselwortgetriebene Vorgehensweise für die Testautomatisierung zu implementieren, müssen die zu testenden Geschäftsprozesse in der abstrakten schlüsselwortbasierten Sprache modelliert werden. Es ist wichtig, dass diese so gestaltet ist, dass die Benutzer (wahrscheinlich die Test Analysten im Projekt) intuitiv damit arbeiten können.

Im Allgemeinen dienen die Schlüsselwörter dazu, abstrakte Geschäftsinteraktionen mit einem System abzubilden. Beispiel: „Auftrag_stornieren" involviert die Überprüfung, ob der Auftrag existiert, Verifizierung der Zugriffsrechte der Person, die die Stornierung veranlasst, Anzeige des Auftrags, der storniert werden soll, und das Anfordern der Stornierungsbestätigung. Der Test Analyst verwendet Folgen von Schlüsselwörtern (z.B. „Anmelden", „Auftrag_auswählen", „Auftrag_stornieren") sowie die relevanten Daten, um die Testfälle zu spezifizieren. Das nachfolgende Beispiel zeigt eine einfache schlüsselwortgetriebene Eingabetabelle, die verwendet werden kann, um die Fähigkeit der Software im Umgang mit Benutzerkonten (Hinzufügen, Zurücksetzen, Löschen von Benutzerkonten) zu testen:

Schlüsselwort	**Benutzer**	**Passwort**	**Ergebnis/angezeigte Meldung**
Add_User	User1	Pass1	Benutzer hinzugefügt
Add_User	@Rec34	@Rec35	Benutzer hinzugefügt
Reset_Password	User1	Welcome	Passwort zurückgesetzt
Delete_User	User1		Nutzer wurde gelöscht
Add_User	User3	Pass3	Benutzer hinzugefügt
Delete_User	User2		Benutzer nicht gefunden

Das Automatisierungsskript sucht in dieser Tabelle nach den Eingabewerten für das Automatisierungsskript. Beispiel: In der Zeile „Delete_User" wird nur der Benutzername benötigt. Um einen neuen Benutzer hinzuzufügen, werden sowohl Benutzername als auch Passwort benötigt. Es kann auch von einem Datenspeicher auf Eingabewerte verwiesen werden; dies ist in der zweiten Zeile bei "Add_User" der Fall, in der ein Verweis auf die Daten angegeben ist anstatt der eigentlichen Daten. Dies erhöht die Flexibilität beim Zugriff auf die Daten, die sich während der Testausführung ändern können. So können datengetriebene Vorgehensweisen mit schlüsselwortgetriebenen Ansätzen kombiniert werden.

Die folgenden Punkte sollten berücksichtigt werden:

- Je höher die Granularität der Schlüsselwörter, desto spezifischer sind die Szenarien, die abgedeckt werden können. Allerdings kann durch die abstrakte Sprache die Wartung komplexer werden.
- Wenn Test Analysten auch die konkrete Aktionen ("Schaltfläche_betätigen", "Aus_Liste_auswählen" usw.) spezifizieren, können die schlüsselwortgetriebenen Tests besser mit verschiedenen Situationen umgehen. Da diese Aktionen direkt mit der Benutzerschnittstelle (GUI) verbunden sind, kann für die Tests auch mehr Wartungsaufwand erforderlich werden, wenn es zu Änderungen kommt.
- Die Verwendung von Sammelbegriffen als Schlüsselwörter kann die Entwicklung einfacher, die Wartung aber komplizierter machen. Es kann beispielsweise sechs verschiedene Schlüsselwörter geben, die alle einen Datensatz erstellen. Sollte evtl. ein Schlüsselwort erstellt werden, das alle sechs Schlüsselwörter der Reihe nach aufruft, um die Aktion zu vereinfachen?
- Ganz gleich wieviel Analyse für die Schlüsselwortsprache aufgewendet wird, es wird immer wieder dazu kommen, dass neue oder andere Schlüsselwörter benötigt werden. Ein Schlüsselwort hat zwei unterschiedliche Aspekte: die Geschäftslogik, die darin zum Ausdruck kommt, und die Automatisierungsfunktionalität, die es ausführt. Es muss daher ein Prozess gefunden werden, der beide Aspekte berücksichtigt.

Eine schlüsselwortgetriebene Testautomatisierung kann die Wartungskosten der Testautomatisierung deutlich verringern, ist aber kostspieliger und schwieriger in der Entwicklung. Außerdem ist für das korrekten Design ein höherer Zeitaufwand nötig, um die Testautomatisierung so umzusetzen, dass damit die angestrebte Rentabilität erzielt werden kann.

6.3 Spezifische Testwerkzeuge

Dieser Abschnitt enthält Informationen über Werkzeuge, die meist von Technical Test Analysten verwendet werden, und die über die Werkzeuge hinausgehen, die im Advanced Level Test Analyst Lehrplan [ISTQB_ALTA_SYL] und im Foundation Level Lehrplan [ISTQB_FL_SYL] behandelt werden.

6.3.1 Werkzeuge zur Fehlereinpflanzung und zum Einfügen von Fehlern

Fehlereinpflanzungswerkzeuge werden vor allem auf Code-Ebene verwendet, um systematisch einzelne oder bestimmte Arten von Fehlerzuständen zu generieren. Diese Werkzeuge fügen absichtlich Fehlerzustände in das Testobjekt ein, um so eine Bewertung der Qualität der Testsuiten zu ermöglichen (z. B. deren Eigenschaft, Fehler zu finden).

Werkzeuge zum Fehlereinfügen werden eingesetzt, um die Fehlerbehandlungsmechanismen des Testobjekts bei anormalen Betriebsbedingungen zu testen. Werkzeuge zum Fehlereinfügen generieren absichtlich falsche Eingabewerte, um sicherzustellen, dass die Software damit umgehen kann.

Werkzeuge zur Fehlereinpflanzung und zum Fehlereinfügen werden vor allem von Technical Test Analysten verwendet; sie können jedoch auch von Entwicklern eingesetzt werden, um neu entwickelten Code zu testen.

6.3.2 Performanztestwerkzeuge

Performanztestwerkzeuge haben zwei Hauptfunktionen:

- Lastgenerierung
- Messung und Analyse des Systemverhaltens unter bestimmter Last

Zur Lastgenerierung wird ein definiertes Nutzungsprofil (siehe Abschnitt 4.5.4) als Skript implementiert. Das Skript kann zunächst für einen einzelnen Nutzer erfasst (beispielsweise mit einem Mitschnittwerkzeug), und anschließend mittels des Lasttestwerkzeugs für das festgelegte Nutzungsprofil implementiert werden. Die Implementierung muss die Datenschwankungen pro Transaktion (oder Menge von Transaktionen) berücksichtigen.

Performanztestwerkzeuge generieren Last, indem sie eine große Zahl von Anwendern (virtuelle Anwender) mit deren spezifischen Nutzungsprofilen simulieren, um eine bestimmte Menge von Eingabedaten zu generieren. Im Unterschied zu einzelnen automatisierten Skripten zur Testausführung erzeugen viele Performanztestskripte die Interaktionen mit dem System auf der Ebene des Kommunikationsprotokolls, und nicht durch die Simulation von Interaktionen über eine graphische Benutzerschnittstelle. Dadurch werden in der Regel weniger separate Testsitzungen benötigt. Einige Lastgenerierungswerkzeuge können die Anwendung auch über die Benutzerschnittstelle steuern, um so die Antwortzeiten des Systems bei der generierten Last genauer zu messen.

Performanztestwerkzeuge liefern eine Vielzahl von Messungen für die Analyse während oder nach Durchführung des Tests. Typische Metriken und Berichte dieser Testwerkzeuge sind:

- Anzahl simulierter Anwender im Testverlauf
- Anzahl und Art der von den simulierten Anwendern erzeugten Transaktionen sowie Eingangsrate der Transaktionen
- Antwortzeiten für bestimmte, von den Anwendern angeforderte Transaktionen
- Berichte und Graphen, die Systemlast und Antwortzeiten gegenüberstellen
- Berichte über Ressourcennutzung (z.B. Auslastung im Zeitverlauf mit Mindest- und Maximalwerten)

Wichtige Faktoren beim Implementieren von Performanztestwerkzeugen sind:

- Die für die Lastgenerierung benötigte Hardware und Netzwerkbandbreiten
- Kompatibilität des Werkzeugs mit dem Kommunikationsprotokoll des zu testenden Systems
- ausreichende Flexibilität des Werkzeugs für die einfache Implementierung unterschiedlicher Nutzungsprofile
- benötigte Überwachungs-, Analyse- und Berichtsfunktionen

Performanztestwerkzeuge werden in der Regel gekauft und nicht selbst entwickelt, weil deren Entwicklung sehr aufwändig ist. Es kann aber sinnvoll sein, ein bestimmtes Performanztestwerkzeug selbst zu entwickeln, wenn technische Einschränkungen den Einsatz eines verfügbaren Produkts nicht zulassen, oder wenn Nutzungsprofil und benötigte Funktionen relativ einfach sind.

6.3.3 Werkzeuge für den webbasierten Test

Es gibt eine Vielzahl „Open Source“-Werkzeuge und kommerzieller Spezialwerkzeuge für den Test von Webseiten. Die nachfolgende Liste enthält die Verwendungszwecke einiger gebräuchlicher webbasierter Testwerkzeuge:

- Mit Hyperlink-Werkzeugen werden Webseiten darauf gescannt und kontrolliert, ob keine Hyperlinks ungültig sind oder fehlen
- HTML- und XML-Prüfwerkzeuge überprüfen, ob die HTML- und XML-Standards in den von der Webseite erstellten Seiten eingehalten werden
- Lastsimulatoren testen das Serververhalten, wenn viele Anwender eine Serververbindung herstellen
- Einfache Testausführungswerkzeuge, die mit unterschiedlichen Browsern funktionieren
- Werkzeuge zum Scannen des Servers, um nicht verlinkte Dateien zu identifizieren
- HTML-Prüfprogramme
- Cascading Style Sheet (CSS)-Prüfprogramme
- Werkzeuge zur Prüfung von Standardverletzungen (z.B. Abschnitt 508 der US-amerikanischen Zugänglichkeitsvorschriften, bzw. M/376 in Europa)

- Werkzeuge, die eine Reihe von Sicherheitsproblemen identifizieren

Eine gute Quelle für „Open Source“-Werkzeuge für den webbasierten Test ist in [Web-7] enthalten. Die Organisation hinter dieser Webseite setzt Standards für das Internet und liefert eine Vielzahl von Werkzeugen, um Fehlhandlungen zu identifizieren, die diese Standards verletzen.

Einige spezifische Testwerkzeuge, die mit *web crawler* ausgerüstet sind, können auch Informationen über die Größe der Seiten, über die Dauer des Herunterladens sowie über das Vorhandensein/Fehlen von Seiten (HTTP error 404) liefern. Dies sind nützliche Informationen für Entwickler, Webmaster und Tester.

Test Analysten und Technical Test Analysten verwenden diese Werkzeuge vorwiegend beim Systemtest.

6.3.4 Werkzeugunterstützung für modellbasiertes Testen

Modellbasiertes Testen (MBT) ist ein Testverfahren, bei dem ein formales Modell (wie z.B. ein endliche Zustandsmaschine) eingesetzt wird, um das erwartete Verhalten eines software-gesteuerten Systems während der Ausführung zu beschreiben. Kommerzielle MBT-Werkzeuge (siehe [Utting 07]) liefern oft eine Funktion, die es dem Benutzer ermöglicht, das Modell „auszuführen“. Interessante Ausführungssequenzen können gespeichert und als Testfälle verwendet werden. Auch andere ausführbare Modelle, wie beispielsweise Petrinetze und State Charts, unterstützen MBT. MBT-Modelle (und -werkzeuge) können eingesetzt werden, um große Mengen von unterschiedlichen Ausführungssequenzen zu generieren.

Mit MBT-Werkzeugen lässt sich die große Anzahl möglicher Pfade, die in einem Modell generiert werden können, reduzieren.

Das Testen mit diesen Werkzeugen kann eine andere Sichtweise auf die zu testende Software liefern. Dies kann dazu führen, dass Fehlerzustände gefunden werden, die im funktionalen Test vielleicht übersehen wurden.

6.3.5 Komponententest- und Build-Werkzeuge

Während Komponententest- und Build-Automatisierungswerkzeuge in vielen Fällen von Entwicklern verwendet werden, werden sie besonders im Kontext eines agilen Entwicklungsmodells von Technical Test Analysten eingesetzt und gewartet.

Komponententestwerkzeuge sind häufig auf die Programmiersprache zugeschnitten, die zur Kodierung des Softwaremoduls verwendet wird. Wenn beispielsweise Java als Programmiersprache verwendet wurde, dann könnte JUnit für die automatisierten Modultests eingesetzt werden. Für viele andere Programmiersprachen gibt es eigene spezifische Testwerkzeuge; diese werden unter dem Oberbegriff „xUnit-Test-Framework“ zusammengefasst. Diese Testrahmen generieren Testobjekte für jede erzeugte Klasse, was die Aufgaben der Programmierer beim Automatisieren von Komponententests wesentlich vereinfacht.

Debugging-Werkzeuge erleichtern den manuellen Komponententest auf tiefster Ebene; sie ermöglichen es den Entwicklern und den Technical Test Analysten, Programmvariablen während der Ausführung zu ändern und die Softwareprogramme Zeile für Zeile auszuführen. Debugging-Werkzeuge werden von Entwicklern auch eingesetzt, um Probleme im Code einzugrenzen und zu identifizieren, wenn das Testteam eine Fehlerwirkung berichtet hat.

Build-Automatisierungswerkzeuge erlauben es nach jeder Änderung einer Softwarekomponente, automatisch einen neuen Buildprozess auszulösen. Nachdem dieser Buildprozess abschlossen ist,

führen andere Werkzeuge automatisch die Komponententests durch. Dieser Grad an Automatisierung ist normalerweise Bestandteil kontinuierlicher Integrationsumgebungen.

Wenn sie korrekt implementiert werden, können diese Arten von Werkzeugen einen sehr positiven Effekt auf die Qualität der Builds haben, die den Testern übergeben werden. Falls durch eine Änderung eines Programmierers Regressionsfehler in den Build eingefügt werden, führt das meist dazu, dass einige der automatisierten Tests nicht mehr erfolgreich ausgeführt werden. Dies ermöglicht eine sofortige Untersuchung der Ursache der Fehlerwirkungen, bevor der Build für die Testumgebung freigegeben wird.

7. Referenzen

7.1 Standards

In diesem Lehrplan werden die folgenden Standards in den jeweils angegebenen Kapiteln erwähnt.

- ANSI/IEEE Std 1471-2000, Recommended Practice for Architectural Description of Software-Intensive Systems
  Kapitel 5
- IEC-61508
  Kapitel 2
- [ISO 25000] ISO/IEC 25000:2005, Software Engineering - Software Product Quality Requirements and Evaluation (SQuaRE)
  Kapitel 4
- [ISO9126] ISO/IEC 9126-1:2001, Software Engineering – Software Product Quality
  Kapitel 4
- [RTCA DO-178B/ED-12B]: Software Considerations in Airborne Systems and Equipment Certification, RTCA/EUROCAE ED12B.1992.
  Kapitel 2

7.2 Dokumente des ISTQB

[ISTQB_AL_OVIEW]	ISTQB Advanced Level Überblick, Version 2012
[ISTQB_ALTA_SYL]	ISTQB Advanced Level Test Analyst Syllabus, Version 2012
[ISTQB_FL_SYL]	ISTQB Foundation Level Syllabus, Version 2012
[ISTQB_GLOSSARY]	ISTQB/GTB Standardglossar der Testbegriffe, Version 2.2, 2012

7.3 Literatur

[Bass03] Len Bass, Paul Clements, Rick Kazman „Software Architecture in Practice (2nd edition)”, Addison-Wesley 2003] ISBN 0-321-15495-9

[Bath08] Graham Bath, Judy McKay, „The Software Test Engineer’s Handbook”, Rocky Nook, 2008, ISBN 978-1-933952-24-6

[Beizer90] Boris Beizer, "Software Testing Techniques Second Edition", International Thomson Computer Press, 1990, ISBN 1-8503-2880-3

[Beizer95] Boris Beizer, „Black-box Testing”, John Wiley & Sons, 1995, ISBN 0-471-12094-4

[Buwalda01]: Hans Buwalda, „Integrated Test Design and Automation” Addison-Wesley Longman, 2001, ISBN 0-201-73725-6

[Copeland03]: Lee Copeland, „A Practitioner's Guide to Software Test Design”, Artech House, 2003, ISBN 1-58053-791-X

[Gamma94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, „Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1994, ISBN 0-201-63361-2

[Jorgensen07]: Paul C. Jorgensen, „Software Testing, a Craftsman’s Approach third edition”, CRC press, 2007, ISBN-13:978-0-8493-7475-3

[Kaner02]: Cem Kaner, James Bach, Bret Pettichord; „Lessons Learned in Software Testing”; Wiley, 2002, ISBN: 0-471-08112-4

[Koomen06]: Tim Koomen, Leo van der Aalst, Bart Broekman, Michael Vroon, "TMap Next for result-driven testing"; UTN Publishers, 2006, ISBN: 90-72194-79-9

[McCabe76] Thomas J. McCabe, "A Complexity Measure", IEEE Transactions on Software Engineering, Vol. SE-2, No. 4, December 1976. PP 308-320
[NIST96] Arthur H. Watson and Thomas J. McCabe, "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric", NIST Special Publication 500-235, Prepared under NIST Contract 43NANB517266, September 1996.
[Splaine01]: Steven Splaine, Stefan P. Jaskiel, „The Web-Testing Handbook”, STQE Publishing, 2001, ISBN 0-970-43630-0
[Utting 07] Mark Utting, Bruno Legeard, „Practical Model-Based Testing: A Tools Approach”, Morgan-Kaufmann, 2007, ISBN: 978-0-12-372501-1
[Whittaker04]: James Whittaker and Herbert Thompson, „How to Break Software Security”, Pearson / Addison-Wesley, 2004, ISBN 0-321-19433-0
[Wiegers02] Karl Wiegers, "Peer Reviews in Software: A Practical Guide", Addison-Wesley, 2002, ISBN 0-201-73485-0

7.4 Sonstige Referenzen

Die folgenden Referenzen verweisen auf Informationen im Internet. Obwohl diese Referenzen zum Zeitpunkt der Veröffentlichung dieses Advanced Level Lehrplans geprüft wurden, übernimmt das ISTQB keine Verantwortung, wenn die referenzierten Websites nicht mehr verfügbar sind.

[Web-1] www.testingstandards.co.uk
[Web-2] http://www.nist.gov NIST National Institute of Standards and Technology
[Web-3] http://www.codeproject.com/KB/architecture/SWArchitectureReview.aspx
[Web-4] http://portal.acm.org/citation.cfm?id=308798
[Web-5] http://www.processimpact.com/pr_goodies.shtml
[Web-6] http://www.ifsq.org
[Web-7] http://www.W3C.org

Kapitel 4: [Web-1], [Web-2]
Kapitel 5: [Web-3], [Web-4], [Web-5], [Web-6]
Kapitel 6: [Web-7]

8. Index