Certified Tester
Advanced Level Syllabus
Technical Test Analyst

Version 2019

German Testing Board e.V.

Übersetzung des englischsprachigen Lehrplans des International Software Testing Qualifications Board (ISTQB®), Originaltitel: Certified Tester Advanced Level, Technical Test Analyst Syllabus des ISTQB, Fassung 2019.

.

Änderungshistorie

Version	Datum	Bemerkungen
V2019	05.04.2020	Deutschsprachige Fassung freigegeben

Inhaltsverzeichnis

Dank

Der ISTQB CTAL TTA Syllabus wurde von einem Kernteam der Advanced Level-Arbeitsgruppe des International Software Testing Qualifications Board (ISTQB) erstellt: Graham Bath, Judy McKay, Mike Smith.

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

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

Dani Almog	Andrew Archer	Rex Black
Armin Born	Sudeep Chatterjee	Tibor Csöndes
Wim Decoutere	Klaudia Dussa-Zieger	Melinda Eckrich-Brájer
Peter Foldhazi	David Frei	Karol Frühauf
Jan Giesen	Attila Gyuri	Matthias Hamburg
Tamás Horváth	N. Khimanand	Jan te Kock
Attila Kovács	Claire Lohr	Rik Marselis
Marton Matyas	Judy McKay	Dénes Medzihradszky
Petr Neugebauer	Ingvar Nordström	Pálma Polyák
Meile Posthuma	Stuart Reid	Lloyd Roden
Adam Roman	Jan Sabak	Péter Sótér
Benjamin Timmermans	Stephanie van Dijck	Paul Weymouth

Die englische Originalausgabe wurde von der Hauptversammlung des ISTQB® am 20. Oktober 2019 offiziell freigegeben.

Das German Testing Board (GTB) dankt dem Reviewteam der deutschsprachigen Fassung 2019: Matthias Hamburg, Marc-Florian Wendland, Jan te Kock, Monika Bögge, Dr. Klaudia Dussa-Zieger (Leitung).

0. Einführung in den Lehrplan

0.1 Zweck dieses Dokuments

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

1. Nationalen/regionalen Boards zur Übersetzung in die jeweilige Landessprache(n) und zur Akkreditierung von Trainingsprovidern. 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. Zertifizierungsstellen zur Ableitung von Prüfungsfragen in ihrer Landessprache, die sich an den Lernzielen der jeweiligen Lehrpläne orientieren.
3. Trainingsprovidern zur Erstellung von Kursmaterialien und zur Bestimmung angemessener Lehrmethoden.
4. Prüfungskandidaten zur Vorbereitung auf die Zertifizierungsprüfung (entweder als Teil eines Seminars oder kursunabhängig).
5. Allen Personen weltweit, die im Bereich Software- und Systementwicklung tätig sind, zur Förderung des Berufsbildes des Software- und Systemtesters, 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 und erhalten.

0.2 Der Certified Tester Advanced Level im Softwaretesten

Die Advanced Level Core Qualifizierung besteht aus drei separaten Lehrplänen, die sich auf folgende Rollen beziehen:

- Testmanager
- Test Analyst
- Technical Test Analyst

Die ISTQB Advanced Level Overview 2019 [ISTQB_AL_OVIEW] ist ein separates Übersichtsdokument, das folgende Informationen enthält:

- Geschäftlicher Nutzen für die einzelnen Lehrpläne
- Matrix mit Verfolgbarkeit zwischen geschäftlichem Nutzen und Lernzielen
- Zusammenfassung der einzelnen Lehrpläne
- Beziehungen zwischen den Lehrplänen

0.3 Prüfungsrelevante Lernziele und kognitive Stufen

Die Lernziele unterstützen den jeweiligen geschäftlichen Nutzen und dienen zur Ausarbeitung der Prüfung für die Zertifizierung als CTAL Technical Test Analyst (CTAL-TTA).

Den einzelnen Lernzielen ist jeweils eine kognitive Stufe des Wissens, K2, K3 oder K4, zugeordnet, die jeweils am Anfang des Kapitels aufgeführt und wie folgt klassifiziert ist:

- K2: Verstehen
- K3: Anwenden
- K4: Analysieren

Die Definitionen aller Begriffe, die direkt unter den Kapitelüberschriften als Schlüsselbegriffe aufgeführt sind, sollen wiedergegeben werden können (K1), auch wenn diese in den Lernzielen nicht ausdrücklich erwähnt werden.

0.4 Erwartete Erfahrungen

Einige der Lernziele für den CTAL Technical Test Analysten setzen grundlegende Erfahrungen in den folgenden Bereichen voraus:

- Allgemeine Programmierkonzepte
- Allgemeine Systemarchitekturkonzepte

0.5 Die Prüfung

Die Zertifizierungsprüfung für den CTAL Technical Test Analysten basiert auf diesem Lehrplan. Zur Beantwortung einer Prüfungsfrage kann Wissen aus mehreren Abschnitten dieses Lehrplans erforderlich sein. Alle Abschnitte dieses Lehrplans sind prüfungsrelevant, außer der Einführung und der Anhänge. Im Lehrplan sind bibliografische Referenzen zu Standards, Fachbücher und andere ISTQB®-Lehrpläne enthalten. Prüfungsrelevant sind jedoch nur die im vorliegenden Lehrplan zusammengefassten Inhalte der referenzierten Materialien.

Das Format der Prüfung ist Multiple Choice, bestehend aus 45 Fragen. Zum Bestehen der Prüfung müssen mindestens 65% der Gesamtpunktzahl erzielt werden.

Prüfungen können als Teil eines akkreditierten Trainingsseminars oder unabhängig davon (z.B. bei einer Zertifizierungsstelle oder in einer öffentlichen Prüfung) abgelegt werden. Die Teilnahme an einem akkreditierten Trainingsseminar stellt keine Voraussetzung für das Ablegen der Prüfung dar.

0.6 Voraussetzung für die Prüfung

Voraussetzung für die Prüfung zum CTAL Technical Test Analysten ist das erworbene Zertifikat zum ISTQB® Certified Tester Foundation Level (CTFL®).

0.7 Akkreditierung von Trainingskursen

Nationale ISTQB-Mitgliedsboards können Trainingsprovider akkreditieren, deren Kursmaterial diesem Lehrplan entspricht. Die Trainingsprovider sollten sich von ihrem nationalen Board oder von der Stelle, die die Akkreditierungen durchführt, die entsprechenden Akkreditierungsrichtlinien einholen. Ein akkreditierter Trainingskurs ist als lehrplankonform anerkannt, und es darf im Rahmen des Kurses eine ISTQB-Prüfung beinhalten.

0.8 Detaillierungsgrad des Lehrplans

Der Detaillierungsgrad dieses Lehrplans ermöglicht international einheitliche Kurse und Prüfungen. Um dieses Ziel zu erreichen, besteht der Lehrplan aus den folgenden Elementen:

- Allgemeine Lehrziele, die die Absicht des CTAL Technical Test Analyst-Lehrplans beschreiben.
- Eine Auflistung der Begriffe, an die sich Schulungsteilnehmer erinnern müssen.
- Lernziele der einzelnen Wissensgebiete, die die zu erreichenden kognitiven Lernergebnisse beschreiben.

- Eine Beschreibung der Schlüsselkonzepte, einschließlich Verweisen auf Quellen wie anerkannte Literatur oder Standards

Der Lehrplaninhalt ist keine Beschreibung des gesamten Wissensgebietes, sondern spiegelt den Detaillierungsgrad wider, der in Advanced Level-Trainingsseminaren abzudecken ist. Der Lehrplan konzentriert sich auf Materialien, die für alle Softwareprojekte gelten können, einschließlich agiler Projekte. Der Lehrplan enthält keine spezifischen Lernziele in Bezug auf einen bestimmten Softwareentwicklungslebenszyklus, aber es wird besprochen wie diese Konzepte in agilen Projekten, anderen Ausprägungen von iterativen und inkrementellen Lebenszyklen und in sequenziellen Lebenszyklen angewendet werden.

0.9 Aufbau des Lehrplans

Der Lehrplan besteht aus sechs Kapiteln mit prüfungsrelevanten Inhalten. Die Kapitelüberschrift spezifiziert die Mindestzeit, die für den Unterricht und die Übungen des Kapitels vorgesehen ist; eine weitere Differenzierung der Zeitangabe je Unterkaptel ist nicht vorgesehen. Für akkreditierte Trainingskurse erfordert der Lehrplan eine Unterrichtszeit von mindestens 21 Stunden und 15 Minuten, die sich wie folgt auf die sechs Kapitel verteilt:

- Kapitel 1: Die Aufgaben des Technical Test Analysten beim risikobasierten Testen (30 Minuten)
- Kapitel 2: White-Box-Testverfahren (345 Minuten)
- Kapitel 3: Analytische Testverfahren (210 Minuten)
- Kapitel 4: Qualitätsmerkmale bei technischen Tests (345 Minuten)
- Kapitel 5: Reviews (165 Minuten)
- Kapitel 6: Testwerkzeuge und Testautomatisierung (180 Minuten)

1. Die Aufgaben des Technical Test Analysten beim risikobasierten Testen - 30 min

Schlüsselbegriffe

Produktrisiko, Risikobewertung, Risikoidentifizierung, Risikominderung, risikobasiertes Testen, Risikostufe

Lernziele für die Aufgaben des Technical Test Analysten im risikoorientierten Test

1.2 Risikoorientierte Testaufgaben

TTA-1.2.1 (K2) Die allgemeinen Risikofaktoren zusammenfassen, die der Technical Test Analyst typischerweise berücksichtigen muss

TTA-1.2.2 (K2) Die Aktivitäten des Technical Test Analysten bei einer risikobasierten Testvorgehensweise zusammenfassen

1.1 Einführung

Das Aufsetzen und Management einer risikobasierten Teststrategie liegt in der Verantwortung des Testmanagers. In der Regel wird der Testmanager jedoch die Unterstützung des Technical Test Analysten anfordern, um sicherzustellen, dass die risikobasierte Testvorgehensweise fachgerecht implementiert wird.

Technical Test Analysten arbeiten innerhalb des vom Testmanager für das Projekt festgelegten risikobasierten Testrahmens. Sie bringen ihr Wissen über die technischen Produktrisiken ein, die mit dem Projekt verbunden sind, wie z.B. Risiken in Bezug auf IT-Sicherheit, Systemzuverlässigkeit und Performanz.

1.2 Risikoorientierte Testaufgaben

Aufgrund ihrer besonderen technischen Expertise wirken Technical Test Analysten aktiv an den folgenden risikobasierten Testaufgaben mit:

- Risikoidentifizierung
- Risikobewertung
- Risikominderung

Diese Aufgaben werden im iterativ gesamten Projektlebenszyklus durchgeführt, um neu auftretenden Produktrisiken, sich ändernden Prioritäten und mit der regelmäßigen Bewertung und Kommunikation des Risikostatus zu behandeln.

1.2.1 Risikoidentifizierung

Wenn im Risikoidentifizierungsprozess eine möglichst breite Auswahl an Stakeholdern involviert wird, ist die Wahrscheinlichkeit größer, , dass dabei die größtmögliche Anzahl signifikanter Risiken aufgedeckt wird. Da Technical Test Analysten über spezifische technische Fähigkeiten verfügen sind sie in besonderer Weise dazu geeignet, Experten-Interviews zu führen, Brainstorming mit Kollegen durchzuführen, sowie zur Analyse der aktuellen und vergangenen Erfahrungen, um die Bereiche für mögliche Produktrisiken zu bestimmen. Insbesondere arbeiten Technical Test Analysten eng mit anderen Stakeholdern wie Entwicklern, Architekten, Betriebsingenieuren, Produktverantwortlichen (Product Owner) und Mitarbeitern des technischen Supports und Service-Desk-Technikern zusammen, um die technischen Risiken zu ermitteln, die sich auf das Produkt und das Projekt auswirken. Die Einbeziehung anderer Stakeholder wird normalerweise vom Testmanager gefördert, denn sie stellt sicher, dass alle Ansichten berücksichtigt werden.

Die Risiken, die vom Technical Test Analysten identifiziert werden könnten, basieren typischerweise auf den in Kapitel 4 aufgeführten Qualitätsmerkmalen [gemäß ISO25010]. Zu diesen gehören beispielsweise:

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

1.2.2 Risikobewertung

Während es bei der Risikoidentifizierung darum geht, so viele vorhandene Risiken wie möglich zu identifizieren, befasst sich die Risikobewertung mit der Untersuchung und Kategorisierung dieser identifizierten Risiken. Zu diesem Zweck werden die Eintrittswahrscheinlichkeit und das Schadensausmaß der identifizierten Risiken bestimmt. Die Eintrittswahrscheinlichkeit wird normalerweise als die Wahrscheinlichkeit interpretiert, dass das potenzielle Problem im System unter Test auftreten kann.

Technical Test Analysten tragen dazu bei das potenzielle technische Produktrisiko für jedes einzelne Risiko zu bestimmen und zu verstehen. Im Gegensatz dazu tragen Test Analysten dazu bei, die potenziellen geschäftlichen Auswirkungen eines Problems zu verstehen, falls dieses auftritt.

Projektrisiken können den Gesamterfolg des Projekts beeinflussen. Normalerweise müssen folgende allgemeine Projektrisiken berücksichtigt werden:

- Konflikte zwischen Stakeholdern hinsichtlich der technischen Anforderungen
- Kommunikationsprobleme aufgrund der geographischen Verteilung der Entwicklungsorganisation
- Werkzeuge und Technologie (einschließlich relevanter Fähigkeiten, diese anzuwenden)
- Zeitmangel, knappe Ressourcen und Druck durch das Management
- Fehlen einer frühzeitigen Qualitätssicherung
- Häufige Änderungen der technischen Anforderungen

Produktrisiken können zu einer größeren Anzahl an Fehlerzuständen führen. Normalerweise müssen bei der Risikobewertung die folgenden allgemeinen Faktoren berücksichtigt werden:

- Komplexität der Technologie
- Komplexität der Codestruktur
- Verhältnis von wiederverwendetem zu neuem Code
- Hohe Anzahl von Fehlerzuständen in Zusammenhang mit technischen Qualitätsmerkmalen (Fehlerhistorie)
- Probleme mit technischen Schnittstellen, Integrationsprobleme

Anhand der vorhandenen Informationen über das Risiko schlägt der Technical Test Analyst eine anfängliche Risikostufe gemäß der vom Testmanager vorgegebenen Richtlinien vor. Beispielsweise können Testmanager festlegen, dass die Risiken mit einem Wert von 1 bis 10 kategorisiert werden sollten, wobei die Risikokategorie 1 für das höchste Risiko steht. Der anfängliche Wert kann vom Testmanager geändert werden, nachdem die Ansichten aller Stakeholder berücksichtigt wurden.

1.2.3 Risikominderung

Während des Projekts beeinflussen die Technical Test Analysten, wie das Testen auf die identifizierten Risiken reagiert. Dies beinhaltet im Allgemeinen:

- Reduzierung der Risiken durch die Ausführung der wichtigsten Tests (in Bereichen mit hohem Risiko) und durch die Umsetzung geeigneter Maßnahmen zur Risikominderung und Vorkehrung gegen Eventualitäten, die im Testkonzept festgelegt sind.
- Bewerten von Risiken basierend auf Informationen, die im Projektverlauf gewonnen wurden, sowie nutzen dieser Informationen für die Umsetzung von Maßnahmen zur Risikominderung, die die Eintrittswahrscheinlichkeit reduzieren oder das Schadensausmaß der bekannten Risiken vermeiden.

Der Technical Test Analyst arbeitet oft mit Spezialisten in Bereichen wie IT-Sicherheit und Performanz zusammen, um Risikominderungsmaßnahmen und -lemente der Teststrategie des Unternehmens zu definieren. Weitere Informationen sind in den Specialist-Lehrplänen des ISTQB, wie dem Advanced Level Sicherheitstester-Lehrplan [ISTQB_ALSEC_SYL] und dem Foundation Level Performance Testing-Lehrplan [ISTQB_FLPT_SYL] enthalten.

2. White-Box-Testverfahren - 345 min

Schlüsselbegriffe

Anweisungstest, API-Testen, atomare Bedingung, Entscheidungstest, Kontrollflusstest, Mehrfachbedingungstest, modifizierter Bedingungs-/Entscheidungstest, Pfadtest, verkürzte Auswertung, White-Box-Testverfahren, zyklomatische Komplexität

Lernziele für White-Box-Testverfahren

Anmerkung: Die Lernziele TTA-2.2.1, -2.3.1, -2.4.1, -2.5.1 und -2.6.1 verwenden den Begriff "Spezifikationselement". Damit können Elemente wie Codefragmente, Anforderungen, User-Stories, Anwendungsfälle und funktionale Spezifikationen gemeint sein.

2.2 Anweisungstest

TTA-2.2.1 (K3) Den Anweisungstest anwenden, um Testfälle für ein bestimmtes Spezifikationselement zu erstellen, die eine definierte Überdeckung erzielen

2.3 Entscheidungstest

TTA-2.3.1 (K3) Den Entscheidungstest anwenden, um Testfälle für ein bestimmtes Spezifikationselement zu erstellen, die eine definierte Überdeckung erzielen

2.4 Modifizierter Bedingungs-/Entscheidungstest (MC/DC)

TTA-2.4.1 (K3) Den Modifizierte Bedingungs-/Entscheidungstest anwenden, um Testfälle für ein bestimmtes Spezifikationselement zu erstellen, die eine definierte Überdeckung erzielen

2.5 Mehrfachbedingungstest

TTA-2.5.1 (K3) Den Mehrfachbedingungstest anwenden, um Testfälle für ein bestimmtes Spezifikationselement zu erstellen, die eine definierte Überdeckung erzielen

2.6 Basispfadtest

TTA-2.6.1 (K3) McCabe’s vereinfachte Basispfad-Methode anwenden, um Testfälle für ein bestimmtes Spezifikationselement zu erstellen

2.7 API-Testen

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

2.8 White-Box-Testverfahren auswählen

TTA-2.8.1 (K4) Für eine vorgegebene Projektsituation ein geeignetes White-Box-Testverfahren auswählen

2.1 Einführung

In diesem Kapitel werden hauptsächlich White-Box-Testverfahren beschrieben. Diese Verfahren sind auf Code und andere Strukturen, wie z.B. Flussdiagramme von Geschäftsprozessen, anwendbar.

Jedes der beschriebenen Verfahren leitet Testfälle systematisch aus strukturellen Elementenab und befasst sich gezielt mit besonderen Aspekten der betrachteten Struktur. Die Verfahren liefern Überdeckungskriterien, die gemessen und mit dem für das Projekt oder Unternehmen definierten Ziel verglichen werden müss. Eine vollständige Überdeckung (Überdeckungskriterien sind zu 100% erfüllt) bedeutet nicht, dass die Menge der Tests vollständig ist, sondern nur dass das vorliegende Testverfahren für die untersuchte Struktur keine weiteren nützlichen Tests liefert.

Im vorliegenden Lehrplan werden die folgenden Testverfahren behandelt:

- Anweisungstest
- Entscheidungstest
- Modifizierter Bedingungs-/Entscheidungstest
- Mehrfachbedingungstest
- Basispfadtest
- API-Testen

Anweisungstest und Entscheidungstest werden im Foundation Level-Lehrplan [ISTQB_FL_SYL] eingeführt. Anweisungstests untersuchen die ausführbaren Anweisungen im Code, während Entscheidungstests die Entscheidungen im Code untersuchen und den Code testen, der auf Grundlage des Entscheidungsergebnisses ausgeführt wird.

Der modifizierte Bedingungs-/Entscheidungstest (MC/DC) und der Mehrfachbedingungstest basieren auf Entscheidungsprädikaten; diese beiden Verfahren decken weitgehend die gleichen Arten von Fehlerzuständen auf. Ganz gleich wie komplex ein Entscheidungsprädikat auch sein mag, die Entscheidung wird letztlich zu WAHR oder FALSCH evaluiert, was wiederum bestimmt, welcher Pfad imd Code ausgeführt wird. Ein Fehlerzustand ist festgestellt, wenn der beabsichtigte Pfad nicht genommen wird weil ein Entscheidungsprädikat nicht wie erwartet evaluiert wurde.

Generell nehmen die ersten vier Testverfahren in ihrer Gründlichkeit zu (wobei der Basispfadtest gründlicher ist als Anweisungs- und Entscheidungstests). Gründlichere Testverfahren erfordern, dass mehr Testfälle erstellt werden müssen, um die beabsichtigten Überdeckung zu erzielen, und um subtilere Fehlerzustände aufzudecken.

Siehe auch [Bath14], [Beizer90], [Beizer95], [Copeland03], [McCabe96], [ISO29119] und [Koomen06].

2.2 Anweisungstest

Beim Anweisungstest werden die ausführbaren Anweisungen im Code ausgeführt. Die Überdeckung wird an der Anzahl der im Test ausgeführten Anweisungen dividiert durch die Gesamtzahl aller ausführbaren Anweisungen im gesamten Testobjekt gemessen und üblicherweise als Prozentsatz angegeben.

Anwendbarkeit
Dieser Überdeckungsgrad sollte als das Minimum für jeden zu testenden Programmcode betrachtet werden.

Einschränkungen/Schwierigkeiten

Entscheidungen werden bei diesem Testverfahren nicht berücksichtigt. Selbst bei einer hohen Anweisungsüberdeckung können bestimmte Fehlerzustände in der Logik des Codes unerkannt bleiben.

2.3 Entscheidungstest

Der Entscheidungstest untersucht die Entscheidungen im Code und testen den Code, der auf Grundlage des Entscheidungsergebnisses ausgeführt wird. Dazu folgen die Testfälle den Kontrollflüssen, die von einem Entscheidungspunkt ausgehen. (z.B. „bei einer IF-Anweisung einen für “wahr” und einen für “falsch”, bei einer CASE-Anweisung wären Testfälle für alle möglichen Ergebnisse nötig, auch für das Standardergebnis). Die Überdeckung wird an der Anzahl der im Test ausgeführten Entscheidungsergebnisse dividiert durch die Gesamtzahl aller Entscheidungsergebnisse im gesamten Testobjekt gemessen und üblicherweise als Prozentsatz angegeben.

Im Vergleich zu den nachfolgend beschriebenen modifizierten Bedingungs-/Entscheidungstests und Mehrfachbedingungstests betrachtet der Entscheidungstest nur die gesamte Entscheidung als Ganzes und evaluieren die Entscheidung in separaten Testfällen zu WAHR oder FALSCH.

Anwendbarkeit
Dieser Überdeckungsgrad sollte dann in Betracht gezogen werden, wenn der zu testende Code wichtig oder sogar kritisch ist (siehe Tabelle in Abschnitt 2.8).

Einschränkungen/Schwierigkeiten
Wenn die Zeit knapp bemessen ist, kann die Durchführung von Entscheidungstests problematisch sein, da hier mehr Testfälle erforderlich sind als beim Anweisungstest. Entscheidungstests berücksichtigen nicht im Detail, wie eine Entscheidung mit mehreren Bedingungen getroffen wird, daher können Fehlerzustände bei Kombinationen dieser Bedingungen unerkannt bleiben.

2.4 Modifizierter Bedingungs-/Entscheidungstest

Im Vergleich zum Entscheidungstest, die die gesamte Entscheidung als Ganzes betrachten und die Entscheidungen in separaten Testfällen zu WAHR oder FALSCH evaluieren, befasst sich der modifizierte Bedingungs-/Entscheidungstest damit, wie eine Entscheidung getroffen wird, wenn sie mehrere Bedingungen enthält (ansonsten handelt es sich einfach um einen Entscheidungstest).

Jede Entscheidung besteht aus einer oder aus mehreren atomaren Bedingungen, die jeweils zu einem booleschen Wert ausgewertet werden. Diese werden dann logisch kombiniert das endgültige Entscheidungsergebnis zu bestimmen. Mit diesem Verfahren wird überprüft, ob jede der atomaren Bedingungen unabhängig und korrekt in das Ergebnis der Gesamtentscheidung einfließt.

Dieses Verfahren bietet eine höhere Überdeckung als die Überdeckung beim Anweisungs- und Entscheidungstest, wenn es Entscheidungen gibt, die mehrere Bedingungen enthalten. Wenn N einzelne unabhängige atomare Bedingungen vorliegen, dann lässt sich die modifizierte Bedingungs-/Entscheidungsüberdeckung normalerweise mit N+1 eindeutigen Testfällen erzielen. Der modifizierte Bedingungs-/Entscheidungstest erfordert Paare von Testfällen, die zeigen, dass eine einzelne atomare Bedingung den Entscheidungsausgang unabhängig beeinflussen kann.

Im folgenden Beispiel wird die Anweisung "Wenn (A oder B) und C, dann ..." betrachtet.

	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

In Test 1 ist A WAHR und das Gesamtergebnis ist WAHR. Wenn A FALSCH wird (wie in Test 3; die anderen Werte bleiben unverändert), wird das Ergebnis FALSCH; dies zeigt, dass A den Entscheidungsausgang unabhängig beeinflussen kann.

In Test 2 ist B WAHR und das Gesamtergebnis ist WAHR. Wenn B FALSCH wird (wie in Test 3; die anderen Werte bleiben unverändert), wird das Ergebnis FALSCH; dies zeigt, dass B den Entscheidungsausgang unabhängig beeinflussen kann.

In Test 1 ist C WAHR und das Gesamtergebnis ist WAHR. Wenn C FALSCH wird (wie in Test 4; die anderen Werte bleiben unverändert), wird das Ergebnis FALSCH; dies zeigt, dass C den Entscheidungsausgang unabhängig beeinflussen kann.

Es ist zu beachten, dass es im Gegensatz zu Anweisungs- und Entscheidungstest beim modifizierten Bedingungs-/Entscheidungstest keine "definierte Überdeckung" gibt; sie ist entweder erreicht (d.h. 100% Überdeckung) oder nicht.

Anwendbarkeit
Dieses Verfahren ist in der Softwareentwicklung für die Luft- und Raumfahrtindustrie sowie für andere sicherheitskritische Systeme weit verbreitet. Es wird für sicherheitskritische Software eingesetzt werden, wo eine Fehlerwirkung zu einer Katastrophe führen könnte.

Einschränkungen/Schwierigkeiten
Es kann kompliziert sein, die modifizierte Bedingungs-/Entscheidungsüberdeckung zu erzielen, wenn eine Variable in einer Entscheidung mit mehreren Bedingungen mehrfach vorkommt; in solchen Fällen spricht man von „gekoppelten" Bedingungen. Abhängig von der Entscheidung ist es möglicherweise nicht möglich, den Wert des gekoppelten Bedingung so zu variieren, dass sie 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, durch die modifizierten Bedingungs-/Entscheidungsüberdeckung getestet werden müssen. Ein anderer Ansatz besteht darin, jede Entscheidung mit gekoppelten Bedingungen von Fall zu Fall zu analysieren.

Einige Programmiersprachen und/oder Interpreter sind so ausgelegt, dass sie die Auswertung einer komplexen Entscheidungsanweisung im Code verkürzt durchführen. Das heißt, dass der ausführende Code möglicherweise nicht den gesamten Ausdruck auswertet, wenn das Endergebnis der Bewertung bereits nach der Auswertung nur eines Teils des Ausdrucks bestimmt werden kann. Beispiel: Wenn die Entscheidung "A und B" evaluiert werden soll, dann gibt es keinen Grund B auszuwerten, wenn A bereits zu FALSCH evaluiert wurde. Kein Wert von B kann den Entscheidungsausgang ändern, so dass Ausführungszeit eingespart werden kann, wenn B nicht ausgewertet werden muss. Die verkürzte Auswertung kann die Erzielung der modifizierten Bedingungs-/Entscheidungsüberdeckung beeinträchtigen, da manche erforderlichen Tests möglicherweise nicht ausgeführt werden können.

2.5 Mehrfachbedingungstest

In seltenen Fällen kann es erforderlich sein, dass alle möglichen Kombinationen von atomaren Bedingungen einer Entscheidung getestet werden müssen. Dieses erschöpfende Testen wird als Mehrfachbedingungstest bezeichnet. Die Anzahl der erforderlichen Tests hängt von der Anzahl der atomaren Bedingungen in der Entscheidungsanweisung ab und lässt sich mit 2^N berechnen, wobei N die Anzahl entkoppelter atomarer Bedingungen ist. Dies bedeutet für das bereits verwendete Beispiel, dass die folgenden Tests erforderlich sind, um die 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

Die erzielte Überdeckung wird an der Anzahl der einzelnen im Test ausgeführten Bedingungskombinationen dividiert durch die Gesamtzahl der Bedingungskombinationen im Testobjekt gemessen, normalerweise ausgedrückt als Prozentsatz.

Anwendbarkeit
Dieses Testverfahren wird zum Testen eingebetteter Software verwendet, von der erwartet wird über lange Zeiträume hinweg zuverlässig und ohne Systemabstürze zu laufen (z.B. Telefonvermittlungen mit einer erwarteten Lebensdauer von 30 Jahren).

Einschränkungen/Schwierigkeiten
Da die Anzahl der Testfälle direkt aus einer Wahrheitstabelle mit allen atomaren Teilbedingungen abgeleitet werden kann, lässt sich dieser Überdeckungsgrad leicht bestimmen. Aufgrund der großen Anzahl von erforderlichen Testfällen ist jedoch die modifizierte Bedingungs-/Entscheidungs-überdeckung für die meisten Situationen praktikabler.

Falls die Programmiersprache die verkürzte Auswertung verwendet, verringert sich dadurch häufig die Anzahl der tatsächlich ausgeführten Testfälle, je nach Reihenfolge und Gruppierung der logischen Operationen, die auf den jeweiligen atomaren Teilbedingungen ausgeführt werden.

2.6 Basispfadtest

Beim Pfadtest werden im Allgemeinen Pfade durch den Kontrollfluss des Codes identifiziert und dann Testfälle entworfen, die diese Pfade abdecken. Grundsätzlich wäre es nützlich, jeden einzelnen Pfad durch das System zu testen. Bei nichttrivialen Systemen könnte so jedoch die Anzahl der Testfälle aufgrund etwaiger im Code vorhandener Schleifen übermäßig groß werden. Im Gegensatz dazu ist die Durchführung von Basispfadtests entsprechend der von McCabe entwickelten vereinfachten Basispfad-Methode eine realistische Option [McCabe96].

Das Verfahren wird in den folgenden Schritten angewendet:

1. Ein Kontrollflussdiagramm für ein bestimmtes Spezifikationselement (z.B. Code oder funktionale Entwurfsspezifikation) erstellen. Dabei ist zu beachten, dass dies auch der erste Schritt zur Durchführung einer Kontrollflussanalyse sein kann (siehe Abschnitt 3.2.1).
2. Einen Basispfad durch den Code auswählen (es sollte kein Ausnahmepfad sein). Dieser Basispfad sollte der wichtigste Pfad zum Testen sein (dies könnte anhand des Risikos beurteilt werden).

3. Den zweiten Pfad auswählen, indem das Ergebnis der ersten Entscheidung im Pfad geändert wird, wobei so viele weitere Entscheidungsergebnisse mit denen im Basispfad identisch bleiben sollten, wie möglich.
4. Den dritten Pfad auswählen, indem das Ergebnis der zweiten Entscheidung im Pfad geändert wird. Bei Entscheidungen mit mehreren möglichen Pfaden (z.B. eine Case-Anweisung) sollte jedes Ergebnis der Entscheidung ausgeführt werden, bevor mit der nächsten Entscheidung fortgefahren wird.
5. Weitere Pfade auswählen, indem jedes der Entscheidungsergebnisse des Basispfads geändert wird. Bei neuen Entscheidungen sollte zunächst das wichtigste Ergebnis bevorzugt werden.
6. Wenn alle Entscheidungsergebnisse des Basispfads abgedeckt sind, sollte die gleiche Vorgehensweise auch auf die weiteren Pfade angewandt werden, bis alle Entscheidungsergebnisse des Spezifikationselements ausgeführt worden sind.

Anwendbarkeit
Die vereinfachte Basispfad-Methode - wie oben beschrieben - wird oft bei sicherheitskritischen Softwareanwendungen verwendet. Sie ist eine gute Ergänzung zu den anderen in diesem Kapitel behandelten Methoden, weil sie sich mit den Pfaden durch die Software befasst und nicht nur mit einzelnen Entscheidungen.

Einschränkungen/Schwierigkeiten
Zum Zeitpunkt, als der vorliegende Lehrplan veröffentlicht wurde, war die Werkzeugunterstützung für den Basispfad-Test sehr begrenzt.

Überdeckung
Das oben beschriebene Verfahren sollte die vollständige Überdeckung aller linear unabhängigen Pfade gewährleisten, und die Anzahl der Pfade sollte der zyklomatischen Komplexität des Codes entsprechen. Je nach Komplexität des Codes kann es sinnvoll sein, ein Werkzeug zu verwenden, um zu überprüfen, ob die vollständige Überdeckung der Basispfade erzielt wurde. Die Überdeckung wird an der Anzahl der linear unabhängigen, im Test ausgeführten Pfade dividiert durch die Gesamtzahl der linear unabhängigen Pfade im Testobjekt gemessen, normalerweise ausgedrückt als Prozentsatz. Der Basispfadtest liefert gründlicheres Testen als die Entscheidungsüberdeckung bei einer nur geringfügig erhöhten Anzahl von Tests [NIST96].

2.7 API-Test

Eine API (engl. Application Programming Interface) ist 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.

Der API-Test ist eher eine Testart als ein Testverfahren. In gewisser Hinsicht ist der API-Test dem Testen einer grafischen Benutzungsoberfläche (eng. graphical user interface (GUI)) recht ähnlich. Der Schwerpunkt liegt auf der Auswertung von Eingabewerten und zurückgegebenen Daten.

Beim Umgang mit APIs sind Negativtests häufig von entscheidender Bedeutung. Es ist möglich, dass Programmierer, die APIs verwenden um auf Dienste außerhalb ihres eigenen Codes zuzugreifen, versuchen könnten APIs in einer nicht vorgesehenen Art zu verwenden. Das bedeutet, dass eine robuste Fehlerbehandlung unerlässlich ist, um einen fehlerhaften Betrieb zu vermeiden. Möglicherweise ist kombinatorisches Testen vieler verschiedener Schnittstellen erforderlich, weil APIs oft in Verbindung mit anderen APIs verwendet werden, und weil eine einzige Schnittstelle mehrere Parameter enthalten kann, deren Werte auf viele Arten 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 API-Schnittstellen bereitstellen, müssen sicherstellen, dass alle Services die zugesicherte Verfügbarkeit leisten; daher sind oft gründliche Zuverlässigkeitstests seitens des API-Anbieters sowie Support für die Infrastruktur notwendig.

Anwendbarkeit
Der API-Test gewinnt zunehmend an Bedeutung, insbesondere in Zusammenhang mit dem Testen von Multisystemen, da diese verteilt arbeiten oder Remote-Verarbeitung einsetzen, um einen Teil der Aufgaben an andere Prozessoren auszulagern. Beispiele hierfür sind:

- Betriebssystemaufrufe
- Service-orientierte Architekturen (SOA)
- Fernaufruf von Funktionen (engl. Remote Procedure Calls (RPC))
- Webservices

Die Kapselung von Software in Containern [Burns18] führt zur Aufteilung eines Softwareprogramms in mehrere Container, die über Mechanismen wie die oben aufgeführten miteinander kommunizieren. API-Tests sollten auch diese Schnittstellen gezielt testen.

Einschränkungen/Schwierigkeiten
Um eine API direkt zu testen, benötigt der Technical Test Analyst normalerweise spezialisierte Werkzeuge. Da es normalerweise keine direkte grafische Benutzungsoberfläche zur API gibt, könnten Werkzeuge erforderlich sein, um einen initialen Testrahmen aufzusetzen, das Daten-Marshalling durchzuführen, die API aufzurufen und das Ergebnis zu bestimmen.

Überdeckung
Der API-Test beschreibt eine Art des Testens; er bezeichnet keinen bestimmten Überdeckungsgrad. Beim API-Test sollten zumindest die Aufrufe an die API beinhaltet sein, die sowohl mit gültigen Eingabewerten als auch mit ungültigen Eingaben zur Überprüfung der Ausnahmebehandlung ausgeführt werden. Gründlichere API-Tests können sicherstellen, dass alle aufrufbaren Einheiten mindestens einmal oder alle möglichen Aufrufe mindestens einmal ausgeführt werden.

Fehlerarten
Die Fehlerzustände, die beim API-Test gefunden werden können, sind recht unterschiedlich. Häufig geht es um Schnittstellenprobleme, sowie um Probleme mit dem Datenhandling, dem Timing, dem Verlust oder der Duplizierung von Transaktionen.

2.8 White-Box-Testverfahren auswählen

Der Kontext des Systems unter Test wirkt sich auf das Produktrisiko und die Kritikalitätsstufen aus (siehe unten). Diese Faktoren beeinflussen die erforderliche Überdeckungsmetrik (und damit das zu verwendende White-Box-Testverfahren) und die zu erzielende Überdeckung entscheidend. Im Allgemeinen gilt: Je kritischer das System und je höher das Produktrisiko, desto höher der benötigte Überdeckungsgrad, und desto mehr Zeit und Ressourcen werden benötigt, um diese Überdeckung zu erzielen.

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

A. Katastrophal: eine Fehlerwirkung kann den Ausfall einer kritischen Funktionalität verursachen, die für einen sicheren Flug oder eine sichere Landung benötigt wird

B. Gefährlich: eine Fehlerwirkung kann eine große schädliche Auswirkung auf die funktionale Sicherheit oder Performanz haben
C. Schwer: eine Fehlerwirkung kann bedeutend, aber nicht so schwerwiegend wie A oder B sein
D. Leicht: eine Fehlerwirkung ist wahrnehmbar, hat aber weniger schwerwiegende Auswirkungen als C
E. Keine Auswirkung: die Fehlerwirkung kann hat keine Auswirkung auf die Sicherheit

Wenn das Softwaresystem zur Kritikalitätsstufe A gehört, dann muss die modifizierte Bedingungs-/Entscheidungsüberdeckung erzielt werden. Für Systeme der Kritikalitätsstufe B ist 100% Entscheidungsüberdeckung vorgeschrieben, modifizierte Bedingungs-/Entscheidungsüberdeckung ist optional. Die Kritikalitätsstufe C erfordert mindestens die Erfüllung der Anweisungsüberdeckung.

IEC 61508 [IEC61508] ist ein weitere internationaler Standard für die funktionale Sicherheit programmierbarer, elektronischer, sicherheitsbezogener Systeme. Dieser Standard wurde in vielen verschiedenen Branchen angepasst, u.a. in der Automobilindustrie, in der Eisenbahnbranche, für Produktionsprozesse, für Kernkraftwerke und für den Maschinenbau. Die Kritikalität wird hier anhand von Sicherheitsintegritätsstufen) (engl. Safety Integrity Level (SIL)) definiert. Diese sind gestaffelt von SIL1 (= geringste Kritikalität) bis SIL4 (= höchste Kritikalität). Für die einzelnen Sicherheitsintegritätsstufen werden im Standard Empfehlungen für die Testüberdeckung gegeben, die in der folgenden Tabelle aufgeführt sind (die genauen Definitionen für die einzelnen Stufen und für die Bedeutung von "empfohlen" und "sehr empfohlen" sind in der Norm enthalten).

SIL	100% Anweisungs-überdeckung	100% Zweigüberdeckung-(Entscheidungsüberdeckung)	100% modifizierte Bedingungs-/Entscheidungsüberdeckung
1	Empfohlen	Empfohlen	Empfohlen
2	**Sehr empfohlen**	Empfohlen	Empfohlen
3	**Sehr empfohlen**	**Sehr empfohlen**	Empfohlen
4	**Sehr empfohlen**	**Sehr empfohlen**	**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 entfernt auf einem anderen System erfolgt. Die Kritikalität des Systems sollte den Aufwand bestimmen, der in den API-Test investiert wird.

3. Analytische Testverfahren - 210 min

Schlüsselbegriffe

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

Lernziele für analytische Testverfahren

3.2 Statische Analyse

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

TTA-3.2.2 (K2) Erklären, wie die Datenflussanalyse verwendet wird, um zu ermitteln, ob der Programmcode Datenflussanomalien aufweist

TTA-3.2.3 (K3) Möglichkeiten vorschlagen, wie die Wartbarkeit von Programmcode durch statische Analyse verbessert werden kann

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

3.3 Dynamische Analyse

TTA-3.3.1 (K3) Die dynamische Analyse anwenden, um ein bestimmtes Ziel zu erreichen

3.1 Einführung

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

Statische Analyse (siehe Abschnitt 3.2) umfasst das analytische Testen der Software sie dabei auszuführen. Die Prüfung wird von einem Werkzeug oder einer Person durchgeführt um herauszufinden, ob die Software korrekt funktionieren wird, wenn sie ausgeführt wird. Die statische Sicht auf die 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 Analysten relevanten Reviewarten werden in Kapitel 5 behandelt.

Die dynamische Analyse (siehe Abschnitt 3.3) erfordert die tatsächliche Ausführung des Codes und zielt auf die Aufdeckung von Fehlerzuständen ab, die einfacher zu finden sind, wenn die Software ausgeführt wird (z.B. Speicherlecks). Wie auch die statische Analyse wird 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 entdecken, sowie die Wartbarkeit 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 analysiert wird, entweder mit Hilfe eines Kontrollflussgraphen oder eines Werkzeuges. Es gibt eine Reihe von Anomalien in Systemen, die mit Hilfe dieses Verfahrens 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.

Die Kontrollflussanalyse kann zur Bestimmung der zyklomatischen Komplexität verwendet werden. Der für die zyklomatische Komplexität ermittelte Wert ist eine positive ganze Zahl, die die Anzahl der linear unabhängigen Kontrollflusspfade in einem stark zusammenhängenden Kontrollflussgraphen angibt. Schleifen und Wiederholungen werden nicht mehr berücksichtigt, nachdem sie einmal durchlaufen wurden. Jeder linear unabhängige Pfad von Anfang bis Ende stellt einen einzelnen Pfad durch das Softwaremodul dar, der getestet werden sollte.

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 nach Möglichkeit in mehrere Module aufgeteilt werden.

3.2.2 Datenflussanalyse

Die Datenflussanalyse umfasst eine Vielzahl von Verfahren, um Informationen über die Verwendung von Variablen in einem System zu sammeln. Dabei wird der Lebenszyklus von Variablen untersucht (d.h. wo sie deklariert, definiert, gelesen, ausgewertet und zerstört werden), da Anomalien bei jedem dieser Operationen auftreten können, als auch wenn die richtige Reihenfolge der Operationen nicht eingehalten wird.

Ein gebräuchliches Verfahren wird als Define-Use (Definieren-Verwenden)-Notation bezeichnet, bei dem der Lebenszyklus jeder Variablen in drei bestimmte atomare Aktionen aufgeteilt wird:

- d (= defined): Die Variable wird deklariert, initialisiert oder definiert
- u (= used): Die Variable wird verwendet oder für eine Berechnung oder Entscheidung gelesen.
- k (= killed): Die Variable wird gelöscht oder zerstört oder ist nicht mehr erreichbar.

Eine gängige Alternative zur Notation d-u-k ist: d (= define/definieren) - r (=reference or read/verweisen oder lesen) - u (= undefined/undefiniert).

Diese drei atomaren Aktionen werden zu Paaren (sog. "Definition-Verwendungs-Paare") zusammengefasst, um den Datenfluss darzustellen. Beispiel: Ein "du-Pfad" stellt ein Fragment des Programmcodes dar, in dem eine Datenvariable definiert und anschließend verwendet wird.

Zu den möglichen Datenflussanomalien gehören die Durchführung der korrekten Aktion der Variablen zum falschen Zeitpunkt oder die Anwendung einer inkorrekten Aktion auf den Daten, die in der Variable gespeichert sind.

Mögliche Anomalien sind:

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

Die verwendete Programmiersprache kann die bei der Datenflussanalyse verwendeten Regeln leiten. Programmiersprachen können es dem Prgogrammierer erlauben bestimmte Operationen auf den Variablen auszuführen, die unter Umständen dazu führen, dass sich das System anders verhält als vom Programmierer erwartet. Beispiel: Eine Variable kann zweimal definiert sein, ohne dass sie tatsächlich verwendet wird, wenn ein bestimmter Pfad verfolgt wird. Bei der Datenflussanalyse werden solche Verwendungen häufig als "verdächtig" bezeichnet. Auch wenn diese Verwendung der Variablen-zuweisung zulässig ist, kann dies später zu Problemen mit der Wartbarkeit des Programmcodes führen.

Der Datenflusstest "verwendet den Kontrollflussgraphen, um die nicht plausiblen Dinge zu erforschen, die mit Daten passieren können" [Beizer90] und findet daher andere Fehlerzustände als die Kontroll-flussanalyse. Der Technical Test Analyst sollte dieses Verfahren bei der Testplanung berücksichtigen, da viele dieser Fehlerzustände sporadische Ausfälle verursachen, die durch dynamisches Testen schwierig zu finden sind.

Die Datenflussanalyse ist ein statisches Verfahren. 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 durch statische Analyse verbessern

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

Grundsätzlich gilt, dass schlecht geschriebener, unkommentierter und unstrukturierter Code schwieriger zu warten ist. Es kann für die Entwickler aufwändiger sein, Fehlerzustände im Code zu lokalisieren und zu analysieren. Auch Änderungen des Codes zur Fehlerbehebung oder zum Hinzufügen eines neuen Features können dazu führen, dass weitere Fehlerzustände eingeführt werden.

Die statische Analyse kann mit Werkzeugunterstützung auch die Einhaltung von Programmierkonventionen und -richtlinien im vorhandenen Programmcode 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 dann, wenn die Syntax des Quellcodes korrekt ist.

Statische Analysewerkzeuge können auch die Prüfung von Code unterstützen, der für die Implementierung von Websites verwendet wird, insbesondere um eine mögliche Gefährdung durch IT-Sicherheitsschwachstellen wie Code-Einschleusung (code injection), Cookie-Sicherheit, webseitenübergreifendes Skripten (Cross-Site-Scripting (XSS)), Ressourcenmanipulation und SQL-Einschleusung (SQL injection) zu prüfen. Weitere Einzelheiten sind in Abschnitt 4.3 und im Advanced Level Sicherheitstester-Lehrplan [ISTQB_ ALSEC_SYL] enthalten.

Ein modularer Aufbau verbessert in der Regel die Wartbarkeit des Codes. Statische Analysewerkzeuge unterstützen die Entwicklung von modularem Code auf folgende Weise:

- Sie suchen nach Wiederholungen im Code. Diese Codeabschnitte bieten sich an für ein Refactoring zu Modulen (wenngleich sich die Modulaufrufe auf die Laufzeit auswirken können, was bei Echtzeitsystemen ein Problem sein könnte).
- Sie erzeugen Metriken, die wertvolle Indikatoren für die Codemodularisierung sind. Dazu gehören u.a. Metriken für die Kopplung und Kohäsion. Ein System, das eine gute Wartbarkeit aufweisen soll, hat eher weniger gekoppelte Module (d.h. Module, die während der Ausführung des Codes aufeinander angewiesen sind) und ein hohes Maß an Kohäsion (der Grad zu dem Module in sich geschlossen und nur für die Durchführung einer einzigen Aufgabe ausgerichtet sind).
- Bei objektorientiertem Code zeigen sie an wo abgeleitete Klassen eventuell zu viel oder zu wenig Sichtbarkeit zu übergeordneten (Eltern-)Klassen haben.
- Sie identifizieren Bereiche im Programmcode oder in der Systemarchitektur mit hoher struktureller Komplexität.

Auch die Wartung von Websites kann durch statische Analysewerkzeuge unterstützt werden. Hier geht es darum, zu prüfen, ob die Baumstruktur der Website ausgewogen ist oder ob ein Ungleichgewicht vorliegt, das die folgenden Konsequenzen haben könnte:

- Schwierigere Testaufgaben
- Höherer Arbeitsaufwand für die Wartung
- Schwierige Navigation für den Nutzer

3.2.4 Aufrufgraphen

Aufrufgraphen sind eine statische Darstellung der Kommunikationskomplexität. Es handelt sich dabei um gerichtete Graphen, in denen Knoten die Programmmodule und Kanten die Kommunikationsbeziehungen zwischen diesen Modulen darstellen.

Aufrufgraphen können im Komponententest verwendet werden, wo verschiedene Funktionen oder Methoden einander aufrufen, im Integrations- und Systemtest, wo separate Module einander aufrufen, oder im Systemintegrationstest, wo separate Systeme einander aufrufen.

Aufrufgraphen können für folgende Zwecke verwendet werden:

- Tests entwerfen, die ein bestimmtes Modul oder System aufrufen
- Herausfinden, wie viele Stellen in der Software das Modul oder System aufrufen
- Die Struktur des Codes und der Systemarchitektur evaluieren
- Vorschläge für die Reihenfolge der Integration bereitstellen (z.B. paarweise und Umgebungs-Integration wie nachfolgend beschrieben)

Im Foundation Level-Lehrplan [ISTQB_FL_SYL] wurden zwei unterschiedliche Arten von Integrationstests (bzw. Integrationsstrategien) behandelt: inkrementelle (Top-Down, Bottom-Up, usw.) und nicht-inkrementelle (Big-Bang). Es wurde festgestellt, dass inkrementelle Vorgehensweisen vorzuziehen sind, da die Eingrenzung von Fehlerzuständen erleichtert wird, weil der Code in Inkrementen integriert wird was wiederum die Menge des betroffenen Codes begrenzt.

Im vorliegenden Advanced Level-Lehrplan werden drei weitere nicht-inkrementelle Vorgehensweisen eingeführt, die Aufrufgraphen verwenden. Diese können inkrementellen Vorgehensweisen vorgezogen werden, da inkrementelle Vorgehensweisen wahrscheinlich zusätzliche Builds bis zum Abschluss des Testens benötigen. Zudem erfordern die inkrementellen Vorgehensweisen, dass Code zur Unterstützung des Testens geschrieben werden muss, der nicht ausgeliefert wird. Die neuen drei nicht-inkrementellen Vorgehensweisen sind:

- Der Paarweiser Integrationstest (nicht zu verwechseln mit dem Black-Box-Testverfahren „Paarweises Testen") konzentriert sich auf Komponentenpaare, die miteinander zusammenarbeiten, wie aus dem Aufrufgraph für den Integrationstest ersichtlich. Während diese Vorgehensweise die Anzahl der Builds nur geringfügig verringert, reduziert sie doch die benötigte Menge von Testrahmen-Code.
- Der Umgebungsintegrationstests basiert 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, wendet diese jedoch auf Aufrufgraphen für Module an. Zu diesem Zweck ist die Erstellung eines Aufrufgraphen erforderlich, der die verschiedenen Möglichkeiten aufzeigt, 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 ein einziges auf
    - 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.

Für weitere Informationen zur Verwendung von Aufrufgraphen und Umgebungsintegrationstests, siehe [Jorgensen07].

3.3 Dynamische Analyse

3.3.1 Überblick

Die dynamische Analyse wird eingesetzt, um Fehlerwirkungen zu erkennen, die nicht unmittelbar an den externen Schnittstellen des Testobjekts erkennbar sind. Beispiel: Mögliche Speicherlecks können durch statische Analyse erkannt werden (es wird Code gefunden, der Speicher zuweist, aber nie Speicher freigibt), mit der dynamischen Analyse sind sie jedoch sehr leicht erkennbar.

Fehlerwirkungen, die nicht ohne weiteres reproduzierbar sind, können erhebliche Konsequenzen für den Testaufwand, die Freigabe der Software oder deren produktiven Einsatz haben. Die Ursache solcher Fehlerwirkungen können Speicher- oder Ressourcenlecks, ein inkorrekter Einsatz von Zeigern und andere Korruptionen (z.B. des Systemstacks) sein [Kaner02]. Diese Art von Fehlerwirkungen können zu einer graduellen Verschlechterung der Systemleistung oder sogar zu Systemabstürzen führen. Daher muss die Teststrategie die mit solchen Fehlern verbundenen Risiken berücksichtigen und gegebenenfalls zur Risikominderung eine dynamische Analyse durchzuführen (normalerweise mit Hilfe von Testwerkzeugen). Da dies häufig Fehlerwirkungen sind, deren Aufdeckung und Behebung am teuersten ist, wird empfohlen mit der dynamischen Analyse frühzeitig im Projekt zu beginnen.

Die dynamische Analyse kann angewendet werden, um Folgendes zu erreichen:

- Fehlerwirkungen verhindern, indem Speicherlecks (siehe Abschnitt 3.3.2) und wilde Zeiger (siehe Abschnitt 3.3.3) aufgedeckt werden
- Systemausfälle analysieren, die nicht leicht reproduzierbar sind
- Netzwerkverhalten bewerten
- Verbesserung der Systemleistung, indem Informationen über das Laufzeitverhalten des Systems verwendet werden um fundierte Änderungen zu ermöglichen

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
- Bestimmen des geeigneten Zeitpunkts für Beginn und Ende der Analyse
- 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 die einem Programm zur Verfügung stehenden Speicherbereiche (RAM) von diesem Programm belegt, aber anschließend nicht wieder freigegeben werden, wenn sie nicht mehr benötigt werden. Dieser Speicherbereich bleibt belegt und steht nicht zur Wiederverwendung zur Verfügung. Wenn dies häufig vorkommt oder wenn wenig Speicher vorhanden ist, kann der nutzbare Speicherplatz ausgehen. Traditionell lag die Manipulation der Speicherbereiche in der Verantwortung der Programmierer. Dynamisch belegter 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 belegter Speicherplatz ohne direktes

Eingreifen des Programmierers wieder freigegeben wird. Es kann sehr schwierig sein, Speicherlecks zu identifizieren, wenn vorhandener, belegter Speicherplatz durch automatische Speicherbereinigung freigegeben wird.

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

Das primäre Symptom eines Speicherlecks ist eine kontinuierliche Verschlechterung der Antwortzeiten des Systems, die letztlich zu einem Systemausfall führen kann. Solchen Ausfällen kann zwar durch einen Neustart (Rebooten) des Systems vorgebeugt werden, was allerdings nicht immer praktikabel oder sogar unmöglich ist.

Viele dynamische Analysewerkzeuge identifizieren Bereiche im Code, in denen Speicherlecks vorkommen, so dass diese korrigiert werden können. Mit einem einfachen Speichermonitor lässt sich ebenfalls herausfinden, ob der verfügbare Speicherplatz im Laufe der Zeit weniger wird, auch wenn hier noch eine Folgeanalyse erforderlich wäre, um die genaue Ursache zu ermitteln.

Es sollten auch weitere Arten von Engpässen betrachtet werden, beispielsweise bei , der Dateizugriffsverwaltung, Semaphoren und Verbindungspools für Ressourcen.

3.3.3 Wilde Zeiger aufdecken

"Wilde" Zeiger in einem Programm sind Zeiger, die nicht mehr korrekt sind und nicht mehr 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 Speicherbereich verweisen, sondern auf einen Speicherbereich außerhalb des zugewiesenen Arrays. Wenn ein Programm wilde Zeiger verwendet, kann dies verschiedene Folgen haben:

- Das Programm kann seine erwartete Leistung erbringen. Dies kann passieren, wenn der Zeiger auf einen Speicherbereich zeigt, der quasi frei ist und derzeit vom Programm nicht benutzt wird, und/oder der einen plausiblen Wert enthält.
- Das Programm kann abstürzen. Dies kann passieren, wenn der Zeiger auf einen Systembereich zeigt, der dann falsch verwendet wird, aber eigentlich für den Betrieb der Software benötigt wird (z.B. das Betriebssystem).
- Das Programm funktioniert nicht korrekt, weil es auf benötigte Objekte nicht zugreifen kann. Unter diesen Bedingungen kann das Programm zwar weiterhin funktionieren, gibt aber Fehlermeldungen aus.
- Daten in den Speicherbereichen können durch den wilden Zeiger zerstört oder unbrauchbar werden, sodass dann inkorrekte Werte verwendet werden (was auch ein IT-Sicherheitsrisiko darstellen kann).

Jede Änderung an der Speichernutzung durch das Programm (z.B. ein neuer Build nach einer Softwareänderung) kann eine dieser vier Folgen haben. Dies ist vor allem dann kritisch, wenn das System zunächst wie erwartet funktioniert, obwohl es fehlerhafte Zeiger enthält, aber nach einer Softwareänderung unerwartet abstürzt (vielleicht sogar im Produktivbetrieb). Solche Ausfälle sind häufig Symptome eines zugrunde liegenden Fehlerzustands (siehe auch [Kaner02], "Lektion 74"). Werkzeuge können fehlerhafte Zeiger identifizieren, die vom Programm verwendet werden, unabhängig von deren Folgen für die Programmausführung. Manche Betriebssysteme haben integrierte Funktionen zur Überwachung von Speicherzugriffsverletzungen während der Laufzeit. So kann das Betriebssystem beispielsweise eine Ausnahme melden, wenn eine Anwendung versucht, auf einen Speicherbereich zuzugreifen, die außerhalb des zulässigen Speicherbereichs dieser Anwendung ist.

3.3.4 Performanz des Systems analysieren

Die dynamische Analyse ist nicht nur zur Erkennung der bisher genannten Fehlerarten geeignet. Bei der dynamischen Analyse der Systemperformanz helfen Werkzeuge, Performanzengpässe zu identifizieren und eine ganze Reihe von Performanzmetriken zu erzeugen, mit denen die Entwickler die Systemperformanz optimieren können. Beispiel: Es können Informationen darüber geliefert werden, wie oft ein Modul während der Ausführung 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 Systemperformanz wird häufig während des Systemtests durchgeführt, kann aber auch schon in früheren Testphasen erfolgen, wenn ein einzelnes Teilsystems mit Hilfe eines Testrahmens getestet wird. Weitere Informationen sind im Foundation Level Performance Testing-Lehrplan [ISTQB_FLPT_SYL] enthalten.

4. Qualitätsmerkmale bei technischen Tests - 345 min

Schlüsselbegriffe

Analysierbarkeit, Anpassbarkeit, Austauschbarkeit, betrieblicher Abnahmetest, Authentizität, Fehlertoleranz, Installierbarkeit, Integrität, IT-Sicherheit, Kapazität, Koexistenz, Kompatibilität, Modifizierbarkeit, Modularität, Nichtabstreitbarkeit, Nutzungsprofil, Performanz, Qualitätsmerkmal, Reife, Ressourcennutzung, Testbarkeit, Übertragbarkeit, Verfügbarkeit, Vertraulichkeit, Wartbarkeit, Wiederherstellbarkeit, Wiederverwendbarkeit, Zeitverhalten, Zurechenbarkeit, Zuverlässigkeit, Zuverlässigkeitswachstumsmodell

Lernziele für Qualitätsmerkmale bei technischen Tests

4.2 Allgemeine Planungsaspekte

TTA-4.2.1 (K4) Für ein bestimmtes Szenario die nicht-funktionalen Anforderungen analysieren und dafür die entsprechenden Inhalte des Testkonzepts erstellen

TTA-4.2.2 (K3) Für ein bestimmtes Produktrisiko die spezifische nicht-funktionale Testart (bzw. Testarten) definieren, die am besten geeignet ist (sind)

TTA-4.2.3 (K2) Verstehen und erläutern, in welchen Phasen des Softwareentwicklungslebenszyklus nicht-funktionales Testen typischerweise erfolgen sollte

TTA-4.2.4 (K3) Für ein vorgegebenes Szenario festlegen, welche Fehlerarten Sie durch die Anwendung nicht-funktionaler Testarten erwarten aufzudecken

4.3 Sicherheitstest

TTA-4.3.1 (K2) Die Gründe für die Einbeziehung des IT-Sicherheitstests in eine Testvorgehensweise erläutern

TTA-4.3.2 (K2) Die wichtigsten Aspekte erläutern, die bei der Planung und Spezifizierung von IT-Sicherheitstests zu berücksichtigen sind

4.4 Zuverlässigkeitstest

TTA-4.4.1 (K2) Die Gründe für die Einbeziehung von Zuverlässigkeitstests in eine Testvorgehensweise erläutern

TTA-4.4.2 (K2) Die wichtigsten Aspekte erläutern, die bei der Planung und Spezifizierung von Zuverlässigkeitstests zu berücksichtigen sind

4.5 Performanztest

TTA-4.5.1 (K2) Die Gründe für die Einbeziehung von Performanztests in eine Testvorgehensweise erläutern

TTA-4.5.2 (K2) Die wichtigsten Aspekte erläutern, die bei der Planung und Spezifizierung von Performanztests zu berücksichtigen sind

4.6 Wartbarkeitstest

TTA-4.6.1 (K2) Die Gründe für die Einbeziehung von Wartbarkeitstests in eine Testvorgehensweise erläutern

4.7 Übertragbarkeitstest

TTA-4.7.1 (K2) Die Gründe für die Einbeziehung von Übertragbarkeitstests in eine Testvorgehensweise erläutern

4.8 Kompatibilitätstest

TTA-4.8.1 (K2) Die Gründe für die Einbeziehung von Kompatibilitätstests in eine Testvorgehensweise erläutern

4.1 Einführung

Im Allgemeinen fokussieren Technical Test Analysten "wie" das Produkt funktioniert (und weniger auf die funktionalen Aspekte, die besagen, "was" das Produkt kann). Diese Tests können auf jeder Teststufe stattfinden. Beispiel: Beim Komponententest von Echtzeitsystemen und eingebetteten Systemen sind das Performanz-Benchmarking und das Testen der Ressourcennutzung besonders wichtig. Beim Systemtest und den betrieblichen Abnahmetests ist das Testen von Zuverlässigkeitsmerkmalen (z.B. Wiederherstellbarkeit) angebracht. Die Tests dieser Teststufe sind auf ein spezifisches System ausgerichtet, d.h. auf eine spezifische Kombination von Hardware und Software. Zu diesem System unter Test können verschiedene Server, Clients, Datenbanken, Netzwerke und andere Ressourcen gehören. Unabhängig von der Teststufe sollte das Testen entsprechend der Risikoprioritäten und den verfügbaren Ressourcen durchgeführt werden.

Es ist zu beachten, dass sowohl dynamische als auch statische Tests (siehe Kapitel 3) angewandt werden können, um die in diesem Kapitel beschriebenen nicht-funktionalen Qualitätsmerkmale zu testen.

Die Beschreibung der Produktqualitätsmerkmale orientiert sich am ISO Standard 25010 [ISO25010], der als Richtschnur für die Beschreibung der Merkmale und ihrer Untermerkmale dient. Diese sind in der nachstehenden Tabelle aufgeführt, aus der ebenfalls hervorgeht, welche Merkmale/Untermerkmale im Test Analyst-Lehrplan und welche im Technical Test Analyst-Lehrplan behandelt werden:

Qualitätsmerkmal	Untermerkmale	Test Analyst	Technical Test Analyst
Funktionale Angemessenheit	Funktionale Korrektheit, funktionale Angemessenheit, funktionale Vollständigkeit	X	
Zuverlässigkeit	Softwarereife, Fehlertoleranz, Wiederherstellbarkeit, Verfügbarkeit		X
Gebrauchstauglichkeit (Usability)	Erkennbare Angemessenheit, Erlernbarkeit, Operabilität, Ästhetik der Benutzungsschnittstelle, Benutzerfehlerschutz, Barrierefreiheit	X	
Performanz	Zeitverhalten, Ressourcennutzung, Kapazität		X
Wartbarkeit	Analysierbarkeit, Modifizierbarkeit, Testbarkeit, Modularität, Wiederverwendbarkeit		X
Übertragbarkeit	Anpassbarkeit, Installierbarkeit, Austauschbarkeit	X	X
IT-Sicherheit	Vertraulichkeit, Datenintegrität, Nichtabstreitbarkeit, Zurechenbarkeit, Authentizität		X
Kompatibilität	Koexistenz		X
	Interoperabilität	X	

Anmerkung: In Anhang A befindet sich eine Tabelle, die die in ISO 9126 verwendeten Begriffe (wie in Lehrplan Version 2012 verwendet) mit den Begriffen der neueren ISO 25010 (wie im vorliegenden Lehrplan verwendet) vergleicht.

Für alle in diesem Abschnitt behandelten Qualitätsmerkmale und -Untermerkmale müssen die typischen Risiken erkannt werden, damit eine geeignete Testvorgehensweise erarbeitet und dokumentiert werden kann. Beim Testen von Qualitätsmerkmalen müssen der Zeitplan im

Softwareentwicklungslebenszyklus, benötigte Werkzeuge, die geforderten Standards, die Verfügbarkeit von Software und Dokumentation sowie technisches Fachwissen besondere Beachtung finden. Ohne eine Strategie zur Behandlung jedes einzelnen Merkmals und dessen spezifischen Testbedarfs wird der Tester möglicherweise nicht genügend Zeit für Planung, Vorbereitung und Durchführung der entsprechenden Tests im Zeitplan bekommen [Bath14].

Einige dieser Tests, z.B. die Performanztests, können eine umfangreiche Planung, spezielle Ausrüstung, bestimmte Werkzeuge, spezielle Testfähigkeiten und in den meisten Fällen auch einen erheblichen Zeitaufwand erfordern. Das Testen der Qualitätsmerkmale und -Untermerkmale muss in die Gesamttestplanung einfließen, und es müssen für die Aufgaben angemessene Ressourcen ausgewiesen werden. Jede dieser Testarten hat bestimmte Erfordernisse, befasst sich mit bestimmten Problematiken und kann in unterschiedlichen Phasen des Softwareentwicklungslebenszyklus stattfinden. Dies wird in den folgenden Abschnitten beschrieben.

Während sich Testmanager mit dem Zusammenstellen und Berichten der zusammengefassten Informationen aus Metriken über die Qualitätsmerkmale und -Untermerkmale befassen, sind die Test Analysten oder die Technical Test Analysten für das Erheben der Informationen zu den jeweiligen Metriken zuständig (Zuständigkeiten siehe Tabelle oben).

Metriken von Qualitätsmerkmalen, die der Technical Test Analyst in Tests vor dem Produktivbetrieb erhebt, können als Grundlage für Service Level-Vereinbarungen zwischen Lieferanten und Stakeholdern (z.B. Kunden, Betreiber) dienen. In manchen Fällen können die Tests auch noch durchgeführt werden, nachdem die Software produktiv gegangen ist. Dann ist oft ein separates Team oder ein separater Unternehmensbereich dafür zuständig. Dies ist häufig bei Performanz- und Zuverlässigkeitstests der Fall, die in der Produktivumgebung zu anderen Testergebnissen führen können als in einer Testumgebung.

4.2 Allgemeine Planungsaspekte

Wenn versäumt wird, nicht-funktionale Tests einzuplanen kann das den Erfolg einer Anwendung ernsthaft in Frage stellen. Der Technical Test Analyst kann 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 in Zusammenhang mit den vorgesehenen Tests zu kümmern. Die Ergebnisse gehen dann möglicherweise auch in das Mastertestkonzept ein.

Bei der Durchführung dieser Aufgaben sind die folgenden allgemeinen Faktoren zu berücksichtigen:

- Anforderungen der Stakeholder
- Beschaffung benötigter Werkzeuge und Schulungen
- Anforderung an die Testumgebung
- Organisatorische Faktoren
- Fragen der Datensicherheit
- Risiken und typische Fehlerzustände

4.2.1 Anforderungen der Stakeholder

Nicht-funktionale Anforderungen sind oft schlecht spezifiziert oder gar nicht vorhanden. Daher müssen Technical Test Analysten in der Planungsphase die Erwartungshaltungen der betroffenen Stakeholder bezüglich der technischen Qualitätsmerkmale sondieren und bewerten, welche Risiken mit diesen verbunden sind.

Ein üblicher Ansatz ist, davon auszugehen, dass Kunden, die mit der bestehenden Systemversion zufrieden sind, auch mit den neuen Versionen zufrieden sein werden, solange das bisher erreichte Qualitätsniveau erhalten bleibt. In diesem Fall kann die bestehende Version des Systems als

Referenzsystem dienen. Für einige der nicht-funktionalen Qualitätsmerkmale kann dies eine sehr nützlicher Ansatz sein, beispielsweise für die Performanz des Systems, wenn es Stakeholdern schwerfällt, die Anforderungen zu spezifizieren.

Es ist ratsam, beim Ermitteln der nicht-funktionalen Anforderungen verschiedene Perspektiven zu berücksichtigen. Hierzu müssen unterschiedliche Stakeholder, wie z.B. Kunden, Product Owner, Anwender, Betriebs- und Wartungspersonal, befragt werden. Wenn wichtige Stakeholder ausgelassen werden, dann werden wahrscheinlich einige Anforderungen nicht berücksichtigt. Weitere Informationen zur Erfassung von Anforderungen sind im Lehrplan für Advanced Level Testmanager [ISTQB_ALTM_SYL] enthalten.

In agilen Projekten können die nicht-funktionalen Anforderungen als User-Stories vorliegen, oder sie können in den Anwendungsfällen als nicht-funktionale Einschränkungen zur spezifizierten Funktionalität hinzugefügt sein.

4.2.2 Beschaffung benötigter Werkzeuge und Schulungen

Kommerzielle Werkzeuge oder Simulatoren sind besonders relevant für den Performanztest und für bestimmte IT-Sicherheitstests. Technical Test Analysten sollten die Kosten und Vorlaufzeiten für Beschaffung, Training und Einführung der benötigten Werkzeuge abschätzen. Wenn Spezialwerkzeuge zum Einsatz kommen sind die damit verbundenen Lernkurven und/oder die Kosten für den Einsatz externer Spezialisten ebenfalls in der Planung zu berücksichtigen.

Die Eigenentwicklung von komplexen Werkzeugen oder Simulatoren kann durchaus zu einem eigenständigen Entwicklungsprojekt werden und sollte auch dementsprechend geplant werden. Insbesondere müssen das Testen und die Dokumentation des eigenentwickelten Werkzeugs in die Zeit- und Ressourcenplanung einfließen. Auch für Aktualisierungen und Nachtests der Simulatoren, die notwendig werden, wenn sich das simulierte System ändert, ist ausreichend Budget und Zeit vorzusehen. Wenn Simulatoren für IT-sicherheitskritische Anwendungen eingesetzt werden sollen, sind auch die entsprechenden Abnahmetests und eine etwaige Zertifizierung des Simulators durch eine unabhängige Instanz zu berücksichtigen.

4.2.3 Anforderungen an die Testumgebung

Für viele nicht-funktionalen Tests (z.B. IT-Sicherheitstests, Performanztests) wird eine produktivähnliche Testumgebung benötigt, um realistische Messungen zu ermöglichen. Größe und Komplexität des Systems unter Test können die 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 tatsächlichen Produktivumgebung
- Verwendung einer reduzierten Version des Systems. Hierbei muss aber sorgfältig vorgegangen werden, damit die erhaltenen Testergebnisse für das tatsächliche Produktivsystem ausreichend repräsentativ sind
- Verwendung von Cloud-basierten Ressourcen als Alternative zur direkten Beschaffung der Ressourcen
- Verwendung virtualisierter Umgebungen

Der Zeitpunkt für die Durchführung der Tests muss sorgfältig geplant werden, da dies wahrscheinlich, nur zu bestimmten Zeiten (z.B. Zeiten geringer Nutzung) stattfinden kann.

4.2.4 Organisatorische Faktoren

Bei nicht-funktionalen Tests ist oft auch das Verhalten mehrerer Komponenten eines Gesamtsystems zu messen (z.B. Server, Datenbanken, Netzwerke). Wenn diese Komponenten auf unterschiedliche Standorte und Unternehmen verteilt sind, kann dies erheblichen Aufwand für die Planung und Koordination der Tests bedeuten. So könnten bestimmte Softwarekomponenten nur zu bestimmten Uhrzeiten oder nur an bestimmten Tagen des Jahres für Systemtests zur Verfügung stehen. Unternehmen, die das Testen unterstützen, stehen möglicherweise nur für eine begrenzte Anzahl von Tagen zur Verfügung. Es kann zu empfindlichen Störungen der geplanten Tests führen, wenn nicht im Vorfeld geklärt worden ist, dass die Systemkomponenten und das Personal anderer Unternehmen (d.h. „ausgeliehene“ externe Expertise) für die Testzwecke auf Abruf bereitstehen.

4.2.5 Fragen der Datensicherheit

Spezifische IT-Sicherheitsmaßnahmen für ein System sollten bereits in der Testplanungsphase berücksichtigt werden, damit alle vorgesehenen Testaktivitäten tatsächlich durchführbar sind. Wenn beispielsweise die Daten verschlüsselt werden, kann die Erstellung von Testdaten und die Verifizierung der Ergebnisse schwierig sein.

Richtlinien und gesetzliche Bestimmungen zum Datenschutz schließen möglicherweise aus, dass die erforderlichen Testdaten auf der Grundlage von Produktivdaten (z.B. persönliche Daten, Kreditkartendaten) generiert werden. Die Anonymisierung von Testdaten ist eine schwierige Aufgabe und muss als Teil der Testrealisierung eingeplant werden.

4.2.6 Risiken und typische Fehlerzustände

Die Identifizierung und das Management von Risiken ist eine grundlegende Überlegung bei der Testplanung (siehe Kapitel 1). Der Technical Test Analyst identifiziert Produktrisiken indem er sein Wissen über die typischen Fehlerarten nutzt, die für ein bestimmtes Qualitätsmerkmal zu erwarten sind. Dies ermöglicht die Auswahl der Testarten, die zur Minderung dieser Risiken geeignet sind. Diese spezifischen Aspekte werden in den nachfolgenden Abschnitten dieses Kapitels behandelt, die die einzelnen Qualitätsmerkmale beschreiben.

4.3 IT-Sicherheitstest

4.3.1 Gründe für die Berücksichtigung von IT-Sicherheitstests

IT-Sicherheitstests untersuchen die Verwundbarkeit eines Systems durch diverse Gefährdungen, bei denen versucht wird, die IT-Sicherheitsrichtlinie eines Systems gezielt außer Kraft zu setzen. Die nachfolgende Liste enthält mögliche Gefährdungen, die beim IT-Sicherheitstest untersucht werden sollten:

- Nicht autorisiertes Kopieren von Anwendungen oder Daten
- Nicht autorisierter Zugriff (z.B. Aktionen ausführen, für die der Beutzer keine Berechtigung besitzt). Diese Tests sind auf Benutzerrechte, Zugriffsberechtigungen und Privilegien fokussiert. Diese Informationen sollten in den Spezifikationen des Systems enthalten sein.
- Software, die weitere, nicht beabsichtigte Nebenwirkungen aufweist, wenn sie ihre beabsichtigte Funktion ausführt. Beispiel: Ein Media Player spielt Audio korrekt ab, schreibt dabei aber Dateien in einen nicht verschlüsselten Zwischenspeicher. Softwarepiraten könnten eine solche Nebenwirkung ausnutzen.
- Bösartiger Code wird von außen in eine Webseite eingeschleust und durch nachfolgende Benutzer ausgeführt (webseitenübergreifendes Skripten, cross-site scripting (XSS)).
- Pufferüberlauf (Speicherüberlauf), der durch die Eingabe von extrem langen Zeichenketten in ein Eingabefeld der Benutzungsschnittstelle ausgelöst werden kann, die vom Code nicht korrekt

verarbeitet werden können. Über Pufferüberlaufgefährdung besteht die Möglichkeit, bösartigen Code auszuführen.
- Dienstblockade (Denial-of-Service, DoS), die bewirkt, dass Benutzer nicht mit einer Anwendung interagieren können (z.B. durch Überlastung eines Webservers mit Störanfragen).
- Das Abfangen, Nachahmen und/oder Verändern und die anschließende Weiterleitung von Kommunikationen (z.B. Daten von Kreditkartentransaktionen) durch einen Dritten, ohne dass der Benutzer das Vorhandensein der dritten Partei bemerkt (Manin-the-middle-Angriff).
- Knacken der Verschlüsselungscodes, die zum Schutz sensibler Daten verwendet werden
- Logische Fallen (engl. logic bombs, auch “easter eggs” genannt), die in bösartiger Absicht in den Code eingeschleust werden und die nur unter bestimmten Bedingungen aktiviert werden (z.B. an einem bestimmten Datum). Wenn diese logischen Fallen aktiviert werden, lösen sie Schadaktionen aus, wie das Löschen von Dateien oder das Formatieren von Festplatten.

4.3.2 IT-Sicherheitstest planen

Für die Planung von IT-Sicherheitstests sind die folgenden Themen besonders relevant:
- Da IT-Sicherheitsprobleme schon beim Architekturentwurf, Systementwurf und der Systemimplementierung verursacht werden können, sind IT-Sicherheitstests bei den Komponententests, Integrationstests und Systemtests einzuplanen. Da sich IT-Sicherheitsbedrohungen ständig ändern können, sollten IT-Sicherheitstests auch regelmäßig für die Zeit nach dem Produktivgang des Systems eingeplant werden. Dies gilt insbesondere für dynamische offene Architekturen wie das Internet of Things, bei denen die Produktivphase durch viele Aktualisierungen der verwendeten Software- und Hardwareelemente gekennzeichnet ist.
- Die vom Technical Test Analysten vorgeschlagenen Testvorgehensweisen sollten Code-Reviews, Reviews der Architektur und des Entwurfs, sowie die statische Analyse des Codes mit Hilfe von IT-Sicherheitstestwerkzeugen beinhalten. Diese Maßnahmen können sehr effektiv sein, um IT-Sicherheitsprobleme aufzudecken, die beim dynamischen Test leicht übersehen werden.
- Technical Test Analysten werden häufig auch gebeten, bestimmte IT-Sicherheitsangriffe zu entwerfen und auszuführen (siehe unten). Dies erfordert eine sorgfältige Planung und Koordination mit Stakeholdern (einschließlich der Spezialisten für IT-Sicherheitstests). Andere IT-Sicherheitstests können in Zusammenarbeit mit Entwicklern oder Test Analysten durchgeführt werden (z.B. Tests der Benutzerrechte, Zugriffsberechtigungen und Privilegien).
- Ein wesentlicher Aspekt bei der Planung von IT-Sicherheitstests ist das Einholen von Genehmigungen. Für den Technical Test Analysten bedeutet dies konkret, dass er vom Testmanager die ausdrückliche Erlaubnis zur Durchführung der geplanten IT-Sicherheitstests einholen muss. Alle zusätzlichen, spontan angesetzten Tests könnten für echte Angriffe gehalten werden, was für die ausführende Person rechtliche Konsequenzen haben könnte. Wenn es keine schriftlichen Unterlagen gibt, aus denen die Beauftragung und Genehmigung der Tests hervorgehen, wird die Ausrede "Wir haben einen IT-Sicherheitstest durchgeführt" wenig überzeugend klingen.
- Die gesamte Planung von IT-Sicherheitstests sollte mit dem IT-Sicherheitsbeauftragten des Unternehmens koordiniert werden, falls diese Rolle im Unternehmen vorhanden ist.
- Es ist zu beachten, dass Verbesserungen der IT-Sicherheit eines Systems sich auf dessen Performanz oder Zuverlässigkeit auswirken können. Nach der Durchführung von Verbesserungen der IT-Sicherheit ist es ratsam, die Durchführung von Performanz- oder Zuverlässigkeitstests in Erwägung zu ziehen (siehe Abschnitte 4.4 und 4.5).

Für die Planung von IT-Sicherheitstests können ggf. spezifische Standards zur Anwendung kommen, wie z.B. [ISA/IEC 62443-3-2], die für industrielle Automatisierungs- und Steuerungssysteme gilt.

Der Advanced Level Sicherheitstester-Lehrplan [ISTQB_ALSEC_SYL] enthält weitere Informationen über die wichtigsten Elemente des IT-Sicherheitstestkonzepts.

4.3.3 Spezifikation von IT-Sicherheitstests

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

- Die Benutzungsschnittstelle betreffend – Nicht autorisierter Zugriff und böswillige Eingaben
- Das Dateisystem betreffend – Zugriff auf vertrauliche Daten in Dateien oder Repositorien
- Das Betriebssystem betreffend – Speicherung sensibler Informationen wie Passwörter in unverschlüsselter Form. Wird das System durch böswillige Eingaben zum Absturz gebracht, können die Informationen zugänglich werden
- Externe Software betreffend – Interaktionen zwischen externen Komponenten, die das System nutzt. Diese können auf Netzwerkebene auftreten (wenn beispielsweise inkorrekte Datenpakete oder Meldungen übertragen werden) oder auf der Ebene der Softwarekomponenten (wenn beispielsweise eine Softwarekomponente ausfällt, die vom System benötigt wird).

Die Untermerkmale der IT-Sicherheit laut ISO 25010[ISO25010] liefern auch eine Grundlage, auf der IT-Sicherheitstests spezifiziert werden können. Diese konzentrieren sich auf die folgenden Aspekte der IT-Sicherheit:

- Vertraulichkeit – der Grad, zu dem eine Komponente oder ein System sicherstellt, dass Daten nur für diejenigen zugänglich sind, die über eine Zugangsberechtigung verfügen.
- Integrität – der Grad, zu dem eine Komponente oder ein System nur autorisierten Zugriff und Änderung einer Komponente, eines Systems oder von Daten zulässt.
- Nichtabstreitbarkeit – der Grad, zu dem Aktionen oder Ereignisse nachweislich stattgefunden haben, so dass die Aktionen oder Ereignisse später nicht abgestritten werden können.
- Zurechenbarkeit – der Grad, zu dem Aktionen eines Akteurs eindeutig zu diesem verfolgt werden können.
- Authentiztät – der Grad, z zu dem die Identität eines Subjekts oder einer Ressource als die behauptete nachgewiesen werden kann.

Die folgende Vorgehensweise [Whittaker04] kann zur Entwicklung von IT-Sicherheitstests verwendet werden:

- Nützliche Informationen zur Spezifikation von Tests zusammentragen, wie z.B. Namen von Mitarbeitern, physikalische Adressen, Details der internen Netzwerke, IP-Nummern, Typ/Version der verwendeten Software / Hardware und die Betriebssystemversion
- Das System mit gängigen Werkzeugen auf Verwundbarkeiten scannen. Diese Werkzeuge dienen nicht direkt dazu, das System/die Systeme zu kompromittieren, sondern um Verwundbarkeiten zu identifizieren, die eine Verletzung der IT-Sicherheitsrichtlinie darstellen oder zu einer solchen führen können. Spezifische Verwundbarkeiten lassen sich auch mit Hilfe von Informationen und Checklisten identifizieren, z.B. mit Checklisten des National Institute of Standards and Technology (NIST, US-Bundesbehörde) [Web-1] und der Open Web Application Security Project™ (OWASP) [Web-4].
- IT-Sicherheitsangriffe/Angriffspläne entwickeln, d.h. mit den gesammelten Informationen werden Testschritte geplant, die die IT-Sicherheitsrichtlinie eines bestimmten Systems durchbrechen sollen. Für die geplanten Angriffe müssen mehrere Eingaben über verschiedene Schnittstellen (z.B. Benutzungsschnittstelle, Dateisystem) spezifiziert werden, um die schwerwiegendsten IT-Sicherheitsfehler aufzudecken. Die verschiedenen in [Whittaker04] beschriebenen "Angriffe" sind eine wertvolle Quelle für Verfahren, die speziell für IT-Sicherheitstests entwickelt wurden.

Anmerkung: Angriffspläne können für Penetrationstests entwickelt werden (siehe [ISTQB_ALSEC_SYL]).

IT-Sicherheitsprobleme können auch durch Reviews (siehe Kapitel 5) und/oder mit Hilfe statischer Analysewerkzeuge (siehe Abschnitt 3.2) aufgedeckt werden. Statische Analysewerkzeuge beinhalten

eine umfangreiche Menge von Regeln, die ganz speziell IT-Sicherheitsbedrohungen betreffen, und anhand derer der Programmcode geprüft wird. Beispiel: Probleme mit dem Pufferüberlauf, die verursacht werden wenn die die Puffergröße vor der Zuweisung von Daten nicht geprüft wird, können von den Werkzeugen aufgedeckt werden.

Abschnitt 3.2 (Statische Analyse) und der Advanced Level Sicherheitstester-Lehrplan [ISTQB_ALSEC_SYL] enthalten weitere Informationen über den IT-Sicherheitstest.

4.4 Zuverlässigkeitstest

4.4.1 Einführung

Der Standard ISO 25010 klassifiziert die folgenden Untermerkmale für das Produktqualitätsmerkmal Zuverlässigkeit:

- Reife
- Fehlertoleranz
- Wiederherstellbarkeit
- Verfügbarkeit

4.4.2 Softwarereife messen

Ein Ziel beim Testen der Zuverlässigkeit ist es, ein statistisches Maß der Softwarereife über eine Zeitspanne zu überwachen und dieses mit der angestrebten Zuverlässigkeit zu vergleichen, die in einer Service Level-Vereinbarung spezifiziert sein kann. Gemessen werden kann die mittlere Betriebsdauer zwischen Ausfällen (engl. Mean Time Between Failures (MTBF)), die mittlere Reparaturzeit nach einem Ausfall eines Systems (engl. Mean Time To Repair (MTTR)) oder eine andere Messung der Fehlerdichte (z.B. Anzahl der Fehlerwirkungen/Ausfälle mit einem bestimmten Schweregrad pro Woche). Diese Metriken können auch als Endekriterien (beispielsweise für die Produktivfreigabe) dienen.

Anmerkung: Reife im Kontext der Zuverlässigkeit ist nicht zu verwechseln mit der Reife des gesamten Softwaretestprozesses, wie im ISTQB-Lehrplan Advanced Level Testmanager [ISTQB_ALTM_SYL] behandelt.

4.4.3 Fehlertoleranz testen

Zusätzlich zum funktionalen Testen, bei dem die Toleranz der Software im Hinblick auf den Umgang mit ungültigen Eingabewerten (sog. Negativtests) evaluiert wird, sind weitere Tests erforderlich, die die Toleranz eines Systems gegenüber Fehlerzuständen und den damit verbundenen Fehlerwirkungen bewerten, die außerhalb der Anwendung unter Test auftreten. Solche Fehlerzustände werden normalerweise vom Betriebssystem oder anderen Systemen bzw. Komponenten gemeldet (z.B. Festplatte voll, Prozess oder Dienst nicht verfügbar, Datei nicht gefunden, Speicher nicht verfügbar). Tests der Fehlertoleranz können auf Systemebene durch spezielle Werkzeuge wie z.B. Fehlereinfügungswerkzeuge unterstützt werden.

Anmerkung: Die Begriffe "Robustheit" und "Toleranz gegen Fehleingaben" werden auch häufig verwendet, wenn es um Fehlertoleranz geht (weitere Einzelheiten, siehe [ISTQB_GLOSSAR]).

4.4.4 Wiederherstellbarkeitstest

Weitere Formen von Zuverlässigkeitstests evaluieren die Fähigkeit des Softwaresystems, wie sich dieses nach Hardware- oder Softwareausfällen auf eine vorher festgelegte Weise wiederherstellen lässt, so dass anschließend der normale Betrieb fortgesetzt werden kann. Die

Wiederherstellbarkeitstests umfassen Tests bezüglich Ausfallsicherheit, Backup und Wiederherstellbarkeit.

Ausfallsicherheitstests werden durchgeführt, wenn die Folgen eines Softwareversagens so gravierend sind, dass spezifische Hardware- und/oder Softwaremaßnahmen beim System implementiert wurden, um den Systembetrieb selbst nach einem Versagen sicherzustellen. Ausfallsicherheitstests können sinnvoll sein, wenn beispielsweise das Risiko finanzieller Verluste extrem hoch ist oder wenn kritische Sicherheitsanforderungen vorliegen. Wenn die Ursache solcher Systemausfälle ein katastrophales Ereignis sein kann, bezeichnet man diese Art von Wiederherstellbarkeitstests auch als „disaster recovery"-Test (Test auf Wiederherstellbarkeit nach einem Totalausfall).

Typische präventive Maßnahmen für Hardwareversagen können eine Lastverteilung auf mehrere Prozessoren, Servercluster, Prozessoren oder Festplatten sein, so dass bei einem Ausfall sofort eine Komponente von einer anderen übernehmen kann (redundante Systeme). Eine typische softwarebasierte Maßnahme könnte die Implementierung von mehr als einer unabhängigen Instanz eines Softwaresystems in so genannten dissimilär redundanten Systemen sein (beispielsweise beim Steuerungssystem eines Flugzeugs). Redundante Systeme sind typischerweise eine Kombination aus Software- und Hardwaremaßnahmen und werden je nach Anzahl der unabhängigen Instanzen als zweifach/dreifach/vierfach redundante Systeme bezeichnet. Dissimilär redundante Lösungen lassen sich dadurch erreichen, dass zwei (oder mehrere) unabhängige Entwicklungsteams dieselben Softwareanforderungen zur Umsetzung erhalten, mit dem Ziel dieselben Services mit unterschiedlichen Lösungsansätzen zu liefern. Dies schützt die mehrfach redundanten Systeme, weil fehlerhafte Eingaben sehr unwahrscheinlich dieselben Ergebnisse haben werden. 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 zu testen, indem sie Fehlerwirkungen simulieren oder tatsächlich Ausfälle in einer kontrollierten Umgebung verursachen. Nach einer Fehlerwirkung wird der Ausfallsicherheitsmechanismus getestet, um sicherzustellen, dass der Vorfall weder zu Datenverlust noch Datenkorruption geführt hat, und dass die Service Level-Vereinbarungen eingehalten wurden (z.B. Funktionsverfügbarkeit, Antwortzeiten).

Backup- und Wiederherstellungstests fokussieren die Maßnahmen und Verfahren, welche zur Minimierung der Auswirkungen möglicher Fehlerwirkungen vorgesehen sind. Solche Tests bewerten die (normalerweise in einem Handbuch dokumentierten) Verfahrensanweisungen zur Durchführung verschiedener Backups und zur Wiederherstellung der Daten im Falle von Datenverlust oder Datenkorruption. 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 die Szenarien in einer Produktivumgebung oder in einer produktivsähnlichen Umgebung getestet, um ihre tatsächliche Anwendung zu validieren.

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

- Zeitaufwand für die verschiedenen Backup-Arten (z.B. vollständig, inkrementell)
- Zeitaufwand für die Wiederherstellung der Daten
- Definierte Ebenen von garantierten Daten-Backups (z.B. Wiederherstellung aller Daten, die nicht älter als 24 Stunden sind, oder Wiederherstellung bestimmter Transaktionsdaten, die nicht älter als eine Stunde sind)

4.4.5 Verfügbarkeitstest

Jedes System, das Schnittstellen zu anderen Systemen und/oder Prozessen hat (z.B. zum Empfang von Eingaben), ist auf die Verfügbarkeit dieser Schnittstellen angewiesen, um die allgemeine Funktionsfähigkeit zu gewährleisten.

Der Verfügbarkeitstest dient vor allem den folgenden Zwecken:

- Feststellung, ob die erforderlichen Systemkomponenten und Prozesse verfügbar sind (auf Abruf oder kontinuierlich) und auf Anfragen erwartungsgemäß reagieren
- Bereitstellung von Metriken, aus denen die Gesamtverfügbarkeit des Systems ermittelt werden kann (oft als prozentualer Anteil der Zeit in einer Service Level-Vereinbarung spezifiziert)
- Prüfung, ob ein Gesamtsystem betriebsbereit ist (z.B. als eines der Kriterien für den betrieblichen Abnahmetest).

Verfügbarkeitstests werden sowohl vor als auch nach der Aufnahme des produktiven Betriebs durchgeführt und sind in den folgenden Situationen besonders relevant:

- Wenn Systeme aus mehreren Systemen bestehen (d.h. bei Multisystemen). Die Tests konzentrieren sich hierbei auf die Verfügbarkeit aller Einzelsysteme.
- Wenn Systeme oder Dienstleistungen extern bezogen werden (z.B. von einem Drittlieferanten). Die Tests konzentrieren sich hierbei auf die Messung der prozentualen Verfügbarkeit, um sicherzustellen, dass die Service Level-Vereinbarungen eingehalten werden.

Die Verfügbarkeit kann mit speziellen Monitoren (bzw. Überwachungswerkzeugen) oder durch die Durchführung spezifischer Tests gemessen werden. Solche Tests sind meist automatisiert und können parallel zum normalen Betrieb stattfinden, sofern sie den normalen Betrieb nicht beeinträchtigen (z.B. durch Verringerung der Performanz).

4.4.6 Zuverlässigkeitstests planen

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

- Die Zuverlässigkeit kann auch noch überwacht werden, nachdem die Software produktiv gegangen ist. Die 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 dem Testmanager nützliche Informationen liefern, da die erwarteten mit den erreichten Zuverlässigkeitswerten verglichen werden können.
- Zuverlässigkeitstests sollten in einer produktivähnlichen Umgebung durchgeführt werden. Die verwendete Umgebung sollte so stabil wie möglich bleiben, damit Zuverlässigkeitstrends über eine Zeitspanne hinweg beobachtet werden können.
- Da für Zuverlässigkeitstests oft das Gesamtsystem notwendig ist werden diese Tests meist als Teil des Systemtests durchgeführt. 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 die Planung und Koordination mit anderen Tests erschweren.

4.4.7 Spezifikation von Zuverlässigkeitstests

Zuverlässigkeitstests können durch ein wiederholbares, vordefiniertes Set an Testfällen erstellt und durchgeführt werden. Dabei kann es sich um Tests handeln, die nach dem Zufallsprinzip aus einem Pool gewählt werden, oder es werden Testfälle aus einem statistischen Modell mit Hilfe von zufälligen oder pseudo-zufälligen Verfahren generiert. Zuverlässigkeitstests können auch auf Nutzungsmustern (auch Nutzungsprofile genannt) basieren (siehe Abschnitt 4.5.3).

Wenn Zuverlässigkeitstests geplant sind, die automatisiert parallel zum normalen Betrieb laufen (z.B. um die Verfügbarkeit zu testen), werden sie im Allgemeinen so einfach wie möglich spezifiziert, um mögliche negative Auswirkungen auf die Performanz des Systems zu vermeiden.

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

4.5 Performanztest

4.5.1 Arten von Performanztests

4.5.1.1 Lasttests

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

4.5.1.2 Stresstests

Stresstests untersuchen die Fähigkeit eines Systems oder einer Softwarekomponente, Spitzenlasten an oder über den spezifizierten Kapazitätsgrenzen oder mit reduzierten Ressourcen (z.B. verfügbare Bandbreite) zu bewältigen. Mit steigender Überbelastung sollte die Systemleistung langsam, vorhersehbar und ohne Ausfall abnehmen. Insbesondere 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, an der das System tatsächlich ausfällt, so dass das schwächsten Glied in der Kette bestimmt werden kann. Bei Stresstests ist es erlaubt, dem System rechtzeitig zusätzliche Kapazitäten hinzuzufügen (z.B. Speicher, CPU-Kapazität, Datenbankspeicher).

4.5.1.3 Skalierbarkeitstests

Skalierbarkeitstests untersuchen die Fähigkeit eines Systems, zukünftige Effizienzanforderungen zu erfüllen, die über den derzeit geltenden Anforderungen liegen. Das Ziel der Tests ist es zu beurteilen, ob das System wachsen kann (z.B. mit mehr Benutzern, größeren Mengen gespeicherter Daten), ohne einen Punkt zu erreichen, an dem die derzeit spezifizierten Performanzanforderungen nicht erfüllt werden oder das System ausfällt. Sind die Grenzen der Skalierbarkeit bekannt, lassen sich Schwellenwerte definieren und im Produktivbetrieb überwachen, so dass bei bevorstehenden Problemen eine Warnung erfolgen kann. Darüber hinaus kann die Produktivumgebung durch entsprechende Hardware angepasst werden, um die erwarteten Erfordernisse zu erfüllen.

4.5.2 Performanztest planen

Zusätzlich zu den allgemeinen Planungsaspekten, die in Abschnitt 4.2 beschrieben sind, können die folgenden Faktoren die Planung von Performanztests beeinflussen:

- Je nach verwendeter Testumgebung und der zu testenden Software (siehe Abschnitt 4.2.3) kann es für Performanztests erforderlich sein, dass das gesamte System implementiert ist bevor effektiv getestet werden kann. In diesem Fall wird der Performanztest meist im Systemtest eingeplant. Andere Performanztests, die bereits im Komponententest 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 produktivähnliche Umgebung zur Verfügung steht. Diese frühen Tests können Performanzprobleme (z.B. Performanzengpässe) aufdecken und das Projektrisiko reduzieren, indem sie zeitaufwändige Korrekturen in der Softwareentwicklungsphase oder nach Produktivgang vermeiden.
- Code-Reviews können Performanzprobleme identifizieren, insbesondere, wenn sie auf Datenbankinteraktionen, Komponenteninteraktionen und Fehlerbehandlung fokussiert sind, insbesondere im Zusammenhang mit „Wait/Retry-Logik“ und mit ineffizienten Abfragen . Diese Code-Reviews sollten frühzeitig im Softwareentwicklungslebenszyklus eingeplant werden.
- Die Hardware, Software und Netzwerkbandbreiten, die für den Performanztest benötigt werden, sollten eingeplant und budgetiert werden. Der Bedarf orientiert sich in erster Linie an der zu erzeugenden Last, die beispielsweise auf der Anzahl der zu simulierenden virtuellen Benutzer und dem voraussichtlich von ihnen erzeugten Netzwerkverkehr basiert. Wird dies bei der Planung nicht berücksichtigt, sind die Performanzmessungen nicht repräsentativ. Beispiel: Zur Verifizierung der Anforderungen an die Skalierbarkeit einer stark frequentierten Internetseite kann eine simulierte Nutzung durch Hunderttausende von virtuellen Benutzern nötig sein.
- Die Erzeugung der für den Performanztest erforderlichen Last kann sich erheblich auf die Kosten für die Beschaffung von Hardware und Werkzeugen auswirken. Diese müssen bei der Planung der Performanztests einkalkuliert werden, um sicherzustellen, dass ausreichende 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 zusätzliche Lizenzen gemietet werden, oder dass für die benötigte Hardware die Dienste von Fremddienstleistern (z.B. Cloud-Dienste) in Anspruch genommen werden. Werden solche Dienste genutzt, dann ist die für den Performanztest verfügbare Zeit möglicherweise begrenzt und muss umso sorgfältiger geplant werden.
- In der Planungsphase sollte darauf geachtet werden, dass das geplante Performanzwerkzeug mit den Kommunikationsprotokollen kompatibel ist, die das System unter Test verwendet.
- Performanzbezogene Fehlerzustände haben oft erhebliche Auswirkungen auf das System unter Test. Wenn Anforderungen an die Performanz des Systems sehr wichtig sind, ist es meist sinnvoll die Performanz kritischer Komponenten (über Treiber und Platzhalter) schon früh im Lebenszyklus zu testen und nicht bis zu den Systemtests zu warten.

Weitere Informationen zur Planung von Performanztests sind im Foundation Level Performance Testing-Lehrplan [ISTQB_FLPT_SYL] enthalten.

4.5.3 Spezifikation von Performanztests

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

Die Anzahl der Benutzer pro Nutzungsprofil lässt sich durch Testmonitore bestimmen (wenn die tatsächliche oder eine vergleichbare Anwendung bereits zur Verfügung steht), oder auch durch Voraussagen. Voraussagen können auf Algorithmen basieren oder vom Fachbereich bereitgestellt werden. Diese sind besonders wichtig für die Spezifizierung der Nutzungsprofile bei Skalierbarkeitstests.

Nutzungsprofile sind Grundlage für die Art und Anzahl der Testfälle des geplanten Performanztests. Häufig werden Testwerkzeuge eingesetzt, welche die Menge an simulierten virtuellen Benutzern für das zu testende Nutzungsprofil generieren (siehe Abschnitt 6.2.2).

Weitere Informationen zum Testentwurf von Performanztests sind im Foundation Level Performance Testing-Lehrplan [ISTQB_FLPT_SYL] enthalten.

4.5.4 QualitätsUntermerkmale von Performanz

Der Standard ISO 25010 klassifiziert die folgenden Untermerkmale für das Produktqualitätsmerkmal Performanz:

- Zeitverhalten
- Ressourcennutzung
- Kapazität

4.5.4.1 Zeitverhalten

Das Zeitverhalten ist auf die Fähigkeit einer Softwarekomponente oder eines Systems fokussiert, auf Eingaben des Benutzers oder eines Systems innerhalb einer definierten Zeit und unter spezifizierten Bedingungen zu reagieren. Die Messungen des Zeitverhaltens können je nach Testziel variieren. Für einzelne Softwarekomponenten kann das Zeitverhalten anhand der CPU-Zyklen gemessen werden. Bei Client-basierten Systemen lässt sich dagegen die Zeit messen, welche benötigt wird, um auf eine bestimmte Benutzeranfrage zu reagieren. Bei Systemen, deren Architekturen aus mehreren Komponenten bestehen (z.B. Clients, Server, Datenbanken), wird das Zeitverhalten der Transaktionen zwischen den einzelnen Komponenten gemessen, um Performanzengpässe zu identifizieren.

4.5.4.2 Ressourcennutzung

Tests der Ressourcennutzung bewerten die Verwendung der Systemressourcen (z.B. Speichernutzung, Festplattenkapazitäten, Netzwerkbandbreite, Verbindungen) anhand definierter Referenzwerte. Die Verwendung dieser Ressourcen wird sowohl bei normaler Systemauslastung gemessen als auch in Stresssituationen (z.B. bei hohen Transaktionsraten oder großen Datenvolumen), um zu bestimmen, ob eine ungewöhnliche Zunahme vorliegt.

Bei eingebetteten Echtzeitsystemen spielt beispielsweise die Speichernutzung (engl. memory footprint) eine wichtige Rolle für die Performanztests. Wenn der “memory footprint” das zulässige Maß überschreitet, dann hat das System möglicherweise zu wenig Speicher, um die erforderlichen Aufgaben im spezifizierten Zeitraum auszuführen. In der Folge kann das System langsamer werden oder sogar abstürzen.

Auch die dynamische Analyse kann die Ressourcennutzung untersuchen (siehe Abschnitt 3.3.4) und Performanzengpässe identifizieren.

4.5.4.3 Kapazität

Die Kapazität eines Systems (einschließlich Software und Hardware) bestimmt die maximale Grenze, bis zu der ein bestimmter Parameter vom System verarbeitet werden kann. Die Anforderungen bezüglich der Kapazität werden normalerweise von technischen und betrieblichen Stakeholdern spezifiziert und können unterschiedliche Parameter betreffen, beispielsweise die maximale Anzahl von Benutzern, die eine Anwendung zu einem bestimmten Zeitpunkt nutzen können, die maximale Datenmenge, die pro Sekunde übertragen werden kann (d.h. die Bandbreite) oder die maximale Anzahl von Transaktionen, die pro Sekunde verarbeitet werden können.

Die Vorgehensweise für das Testen der Kapazitätsgrenzen ist im Allgemeinen der in den Abschnitten 4.5.2 und 4.5.3 beschriebenen Vorgehensweise für das Testen der Performanz ähnlich. Nutzungsprofile für Kapazitätstests konzentrieren sich auf die Erzeugung der Last zum Testen der jeweiligen Grenze. Beispielsweise kann Last generiert werden, die das System der maximalen Datenübertragungsmenge aussetzt. Auch Vorgehensweisen für Stresstests und Skalierbarkeitstests können zum Testen der

Kapazität eingesetzt werden, um das Systemverhalten über die spezifizierten Kapazitätsgrenzen hinaus zu testen (siehe Abschnitte 4.5.1.2 bzw. 4.5.1.3).

4.6 Wartbarkeitstest

Ein wesentlich größerer Teil des Softwarelebenszyklus entfällt auf die Phase, in der die Software gewartet wird, und nicht auf die Phase in der sie entwickelt wird. Bei Wartungstests werden die Auswirkungen von Änderungen an einem System im Betrieb oder ihrer Umgebung getestet. Um sicherzustellen, dass die Wartung eines Systems so effizient wie möglich durchgeführt werden kann, wird im Rahmen der Wartbarkeitstests untersucht, 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 die Softwarewartung

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

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

4.6.1 Statische und dynamische Wartbarkeitstests

Geeignete Verfahren für statische Wartbarkeitstests sind statische Analysen und Reviews, wie in den Abschnitten 3.2 und 5.2 beschrieben. Mit den Wartbarkeitstests sollte begonnen werden, sobald die Entwurfsdokumente zur Verfügung stehen, und sie sollten während der gesamten Phase der Code-Implementierung fortgesetzt werden. Da die Wartbarkeit in den Code und in die zugehörige Dokumentation der einzelnen Komponenten eingebaut ist, kann die Wartbarkeit schon früh im Softwareentwicklungslebenszyklus bewertet werden; es ist unnötig, damit zu warten, bis das System fertig ist und schon läuft.

Dynamische Wartbarkeitstests untersuchen in erster Linie die dokumentierten Verfahren, die für die Wartung einer Anwendung entwickelt wurden (z.B. für die Durchführung von Software-Upgrades). Als Testfälle dienen ausgewählte 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, wenn mehrere Abteilungen/Unternehmen an den Supportverfahren beteiligt sind. Diese Tests können Teil der betrieblichen Abnahmetests sein.

4.6.2 Untermerkmale der Wartbarkeit

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) und nach der Testbarkeit (d.h. der Aufwand für das Testen von Änderungen). Zu den Faktoren, die sowohl die Analysierbarkeit als auch die Testbarkeit beeinflussen, gehören die Verwendung guter Programmierpraktiken (z.B. Kommentierung, Namensgebung für Variablen, Einrückung) und die Verfügbarkeit technischer Dokumentation (z.B. Spezifikationen für den Systementwurf, Schnittstellenspezifikationen).

Weitere relevante Untermerkmale des Qualitätsmerkmal Wartbarkeit [ISO25010] sind:

- Modifizierbarkeit
- Modularität
- Wiederverwendbarkeit

4.7 Übertragbarkeitstest

4.7.1 Einführung

Übertragbarkeitstests untersuchen im Allgemeinen den Grad, zu dem eine Softwarekomponente oder ein System in eine vorgesehene Umgebung übertragen werden kann, entweder bei der Erstinstallation oder aus einer bestehenden Umgebung.

Untermerkmale des Qualitätsmerkmals Übertragbarkeit [ISO25010] sind:

- Installierbarkeit
- Anpassbarkeit
- Austauschbarkeit

Übertragbarkeitstests können schon mit einzelnen Komponenten beginnen (z.B. Austauschbarkeit einer bestimmten Komponente wie beispielsweise Wechsel von einem Datenbankmanagementsystem auf ein anderes) und werden erweitert, wenn mehr Code zur Verfügung steht. Die Installierbarkeit ist möglicherweise erst dann testbar sein, wenn alle Komponenten des Produkts funktionieren.

Da die Übertragbarkeit 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 wirksames Mittel sein, um potenzielle Übertragbarkeitsanforderungen und -probleme zu identifizieren (z.B. die Abhängigkeit von einem bestimmten Betriebssystem).

4.7.2 Installierbarkeitstests

Installierbarkeitstests testen die Installation von Software in einer Zielumgebung samt den dokumentierten Installationsanleitungen . Diese Tests können auch Software adressieren, mit der ein Betriebssystem auf einem Prozessor installiert wird, oder einen Installationswizard, mit dem ein Produkt auf einem Client-PC installiert wird.

Typische Ziele von Installierbarkeitstests sind:

- Validieren, dass die Software erfolgreich installiert werden kann, indem die Anweisungen in einem Installationshandbuch (einschließlich der Ausführung von Installationsskripten) befolgt werden oder ein Installationswizard verwendet wird. 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 (wenn z.B. bestimmte DLLs können nicht geladen werden), ohne das System in einem undefinierten Zustand zu belassen (beispielsweise mit nur teilweise installierter Software oder inkorrekten Systemkonfigurationen).
- Testen, ob sich eine teilweise Installation/Deinstallation der Software abschließen lässt.
- Testen, ob ein Installationswizard eine ungültige Hardware-Plattform oder BetriebssystemkKonfigurationen erfolgreich identifizieren kann.

- Messen, ob der Installationsvorgang im spezifizierten Zeitrahmen oder mit weniger als der spezifizierten Anzahl von Schritten abgeschlossen werden kann.
- Validieren, dass sich eine frühere Version der Software installieren („Downgrade“ der aktuellen Software) oder dass sich die Software deinstallieren lässt

Nach dem Installierbarkeitstest 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 Installierbarkeitstests werden in der Regel Gebrauchstauglichkeitstests durchgeführt (beispielsweise zur Validierung, dass die Anwender bei der Installation verständliche Anweisungen und Feedback/Fehlermeldungen erhalten).

4.7.3 Anpassbarkeitstests

Anpassbarkeitstests überprüfen, ob eine bestimmte Anwendung in allen vorgesehenen Zielumgebungen (Hardware, Software, Middleware, Betriebssystem usw.) korrekt funktioniert. Ein anpassungsfähiges System ist ein offenes System, das sein Verhalten an Änderungen in der Umgebung oder an eigenen Systemteilen anpassen kann. Für die Spezifikation von 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 vordefinierten Verfahrens auf verschiedene spezifizierte Umgebungen portieren lässt. Beim Testen kann dieses Verfahren bewertet werden.

Anpassbarkeitstests können zusammen mit Installierbarkeitstests durchgeführt werden. Normalerweise finden anschließend funktionale Tests statt, um Fehlerzustände aufzudecken, die möglicherweise beim Anpassen der Software an eine andere Umgebung eingeschleust wurden.

4.7.4 Austauschbarkeitstests

Austauschbarkeitstests testen die Austauschbarkeit von Softwarekomponenten eines Systems durch andere Komponenten. 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.

4.8 Kompatibilitätstest

4.8.1 Einführung

Kompatibilitätstest untersuchen die folgenden Aspekte [ISO25010]:

- Koexistenz
- Interoperabilität -– Dies ist im ISTQB Advanced Level Test Analyst-Lehrplan [ISTQB_ALTA_SYL] beschrieben.

4.8.2 Koexistenztests

Computersysteme, die nicht miteinander interagieren, sind dann kompatibel, wenn sie in derselben Umgebung (beispielsweise auf derselben Hardware) ausgeführt werden können, ohne sich gegenseitig in ihrem Verhalten zu beeinflussen (beispielsweise durch Ressourcenkonflikte). Koexistenztests sollten durchgeführt werden, wenn neue oder aktualisierter Software in Umgebungen installiert werden, in denen bereits andere Anwendungen installiert sind.

Typische Ziele von Koexistenztests sind:

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

Koexistenzprobleme sollten analysiert werden, wenn die vorgesehene Zielumgebung geplant wird, die Tests werden jedoch normalerweise erst nach erfolgreichem Abschluss der Systemtests durchgeführt.

5. Reviews - 165 min

Schlüsselbegriffe
Anti-Pattern

Lernziele für Reviews

5.1 Aufgaben von Technical Test Analysten bei Reviews

TTA 5.1.1 (K2) Erklären, warum die Vorbereitung des Reviews für Technical Test Analysten wichtig ist

5.2 Checklisten in Reviews verwenden

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

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

5.1 Aufgaben von Technical Test Analysten bei Reviews

Technical Test Analysten müssen aktiv an den technischen Reviews teilnehmen und ihre individuelle Sichtweise einbringen. Alle Reviewteilnehmer sollten ein formales Reviewtraining erhalten haben, damit sie ihre jeweiligen Rollen besser verstehen. Zudem sollten sie vom Nutzen gut durchgeführter technischer Reviews überzeugt sein. Dazu gehört auch eine konstruktive Zusammenarbeit mit den Autoren bei der Beschreibung und Diskussion von Reviewbefunden. 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 einbringen, die von den Entwicklern möglicherweise übersehen werden. Darüber hinaus spielen Technical Test Analysten eine wichtige Rolle bei der Definition, Anwendung und Pflege von Review-Checklisten und bei der Bereitstellung von Informationen über den Schweregrad von Fehlerzuständen.

Unabhängig von der Art des Reviews muss der Technical Test Analyst ausreichend Zeit zur Vorbereitung bekommen. Die Vorbereitungszeit 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, sowie zu bestimmen, was im Arbeitsergebnis fehlt. Ohne eine angemessene Vorbereitungszeit könnte das Review auf eine redaktionelle Überarbeitung des Dokuments reduziert werden, anstatt ein echtes Review zu sein. Zu einem guten Review gehört es, die Inhalte zu verstehen, zu bestimmen was fehlt, und zu verifizieren, ob das beschriebene Produkt mit anderen Produkten, die entweder bereits entwickelt wurden oder sich in der Entwicklung befinden, konsistent ist. Beispiel: Beim Review eines Stufentestkonzepts für den Integrationstest muss der Technical Test Analyst auch die Objekte berücksichtigen, die integriert werden sollen. Wann werden sie für die Integration bereit sein? Gibt es Abhängigkeiten, die dokumentiert werden müssen? Sind Daten für den Test der Integrationspunkte verfügbar? Ein Review konzentriert sich nicht allein auf das Arbeitsergebnis, das geprüft wird, sondern es muss auch die Interaktion des Reviewgegenstands mit anderen Arbeitsergebnissen des Systems berücksichtigt werden.

5.2 Checklisten in Reviews verwenden

Checklisten werden bei Reviews verwendet, damit die Teilnehmer angehalten sind, bestimmte Punkte im Laufe des Reviews zu verifizieren. Checklisten können 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 zielgerichtet mit bestimmten Qualitätsmerkmalen oder Themenbereichen befassen. Zum 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 der IT-Sicherheit oder Performanz des Systems befassen.

Die nützlichsten Checklisten sind die, die sukzessive von einem Unternehmen bzw. einer organisatorischen Einheit entwickelt wurden, da diese Checklisten folgendes wiedergeben:

- Art des Produkts
- Lokales Entwicklungsumfeld
  - Personal
  - Werkzeuge
  - Prioritäten
- Historie früherer Erfolge und gefundener Fehlerzustände
- Besondere Themen (z.B. Performanz, IT-Sicherheit)

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

Manche Unternehmen erweitern das gängige Konzept von Software-Checklisten um sogenannte „Anti-Patterns", die sich auf häufige Fehlhandlungen, schlechte Verfahren und ineffektive Praktiken beziehen. Der Begriff ist abgeleitet vom populären Konzept der "Entwurfsmuster" (engl. Design patterns), bei denen es sich um wiederverwendbare Lösungen für häufig wiederkehrende Probleme handelt, die sich in der Praxis als effektiv erwiesen haben [Gamma94]. Ein Anti-Pattern ist demzufolge eine häufig vollführte Fehlhandlung, die oft als „nützliche schnelle Lösung" gedacht war.

Es ist zu wichtig zu verstehen, dass wenn eine Anforderung nicht testbar ist, ein Fehlerzustand in dieser Anforderung vorliegt. Nicht testbar bedeutet, dass die Anforderung so spezifiziert ist, dass der Technical Test Analyst nicht bestimmen kann, wie sie zu testen ist. Beispiel: Eine Anforderung, die besagt „Die Software soll schnell sein", ist nicht testbar. Wie soll ein Technical Test Analyst bestimmen, ob die 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). Dabei handelt es sich zudem um eine übergreifende Anforderung, die bei einer komplexen Anwendung ohne Probelme viele einzelne Testfälle hervorbringen könnte. Verfolgbarkeit von dieser Anforderung bis zu den Testfällen ist ebenfalls entscheidend, denn wenn sich die Anforderung ändern sollte, müssen alle Testfälle überprüft und bei Bedarf aktualisiert werden.

5.2.1 Architekturreviews

Die Softwarearchitektur betrifft die grundlegende Organisationsstruktur eines Systems und seiner Komponenten, der Beziehungen der Komponenten untereinander, zur Systemumgebung, und den Grundsätzen, die für Systementwurf und -entwicklung gelten. [ISO42010], [Bass03].

Die für ein Review der Architektur verwendeten Checklisten[1] 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 Instanziierung (lazy instantiation)
- Nebenläufigkeit von Transaktionen
- Prozesstrennung zwischen OLTP (Online Transactional Processing) und OLAP (Online
- Analytical Processing)
- Replikation von Daten

5.2.2 Code-Reviews

Checklisten für Code-Reviews sind zwangsläufig sehr detailliert. Genau wie Checklisten für Architekturreviews, sind sie dann am nützlichsten, wenn sie spezifisch auf eine Programmiersprache, ein Projekt und ein Unternehmen zugeschnitten sind. Die Einbeziehung von Anti-Patterns auf Code-Ebene ist hilfreich, insbesondere für weniger erfahrene Softwareentwickler.

Die für Code-Reviews verwendeten Checklisten[1] können die folgenden sechs Bereiche abdecken:

1. Struktur

[1] Die Prüfungsfrage liefert zur Beantwortung einen Teil der Checkliste.

- Implementiert der Code vollständig und korrekt den Entwurf?
- 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 in einer einzigen Prozedur zusammengefasst werden könnten?
- Ist die Speichernutzung effizient?
- Wird Symbolik benutzt anstatt „magische Nummern"-Konstanten (engl. Magic numbers) 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 verständlich und angemessen dokumentiert und in einem leicht zu wartenden Kommentierungstil geschrieben?
   - Passen alle Kommentare zum Code?
   - Entspricht die Dokumentation den geltenden Standards?

3. Variablen
   - Sind alle Variablen richtig definiert und liegt einesinnvolle, konsistente und eindeutige Namensgebung vor?
   - Gibt es redundante oder ungenutzte Variablen?

4. Arithmetische Operationen
   - Wird im Code vermieden, dass Gleitkommazahlen auf Gleichheit geprüft werden?
   - Verhindert der Code Rundungsfehler systematisch?
   - Vermeidet der Code Additionen und Subtraktionen mit sehr unterschiedlich großen Zahlen?
   - Werden Teiler (Divisoren) auf 0 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?
   - Ist für jede Case-Anweisung ein Standardwert vorgegeben?
   - Sind die Abbruchbedingungen von Schleifen offensichtlich und ausnahmslos erreichbar?
   - Sind die Index-Variablen oder Teilskripte unmittelbar vor der Schleife richtig initialisiert?
   - Können 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?
   - Werden importierte Daten und Eingabeparameter auf Gültigkeit und Vollständigkeit getestet?
   - Sind alle Ausgangsvariablen zugewiesen?

- Wird in jeder Anweisung das richtige Datenelement verarbeitet?
- Wird jeder zugewiesene Speicher freigegeben?
- Werden bei Zugriffen auf externe Geräte Fehlerbehandlungen und Zeitüberschreitungen (Timeouts) 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?

6. Testwerkzeuge und Testautomatisierung - 180 min

Schlüsselbegriffe

datengetriebenes Testen, Debugging, Emulator, Fehlereinpflanzung, Hyperlink, Mitschnitt, Performanz, schlüsselwortgetriebener Test, Simulator, Testdurchführung, Testmanagement

Lernziele für Testwerkzeuge und Testautomatisierung

6.1 Ein Testautomatisierungsprojekt definieren

TTA-6.1.1 (K2) Die Aktivitäten des Technical Test Analysten in Zusammenhang mit dem Aufsetzen eines Testautomatisierungsprojekts zusammenfassen

TTA-6.1.2 (K2) Die Unterschiede zwischen datengetriebener und schlüsselwortgetriebener Testautomatisierung zusammenfassen

TTA-6.1.3 (K2) Die technischen Probleme zusammenfassen, die häufig dafür verantwortlich sind, dass Testautomatisierungsprojekte nicht die geplante Investitionsrendite (engl. return on investment (ROI)) erzielen

TTA-6.1.4 (K3) Die Schlüsselwörter erstellen, die auf einem vorgegebenen Geschäftsprozess basieren

6.2 Spezifische Testwerkzeuge

TTA-6.2.1 (K2) Den Zweck von Werkzeugen zur Fehlereinpflanzung und zum Fehlereinfügen zusammenfassen

TTA-6.2.2 (K2) Die wichtigsten Eigenschaften von Performanztestwerkzeugen sowie die Themen rund um deren Implementierung zusammenfassen

TTA-6.2.3 (K2) Die allgemeinen Verwendungszwecke von Werkzeugen für das webbasierte Testen erläutern

TTA-6.2.4 (K2) Erläutern, wie Werkzeuge das Konzept des modellbasierten Testens unterstützen

TTA-6.2.5 (K2) Darlegen, wie Werkzeuge eingesetzt werden können, um den Komponententest und den Build-Prozess zu unterstützen

TTA-6.2.6 (K2) Darlegen, wie Werkzeuge eingesetzt werden können, um das Testen mobiler Applikationen zu unterstützen

6.1 Ein Testautomatisierungsprojekt definieren

Um kosteneffektiv zu sein, müssen Testwerkzeuge (und insbesondere solche, die die Testausführung unterstützen) sorgfältig konzipiert und entworfen werden. Die Implementierung einer Testautomatisierungsstrategie ohne eine solide Testautomatisierungsarchitektur führt meist zu Werkzeuglandschaften, deren Wartung kostspielig ist, die ihren Zweck nicht ausreichend erfüllen und die die angestrebte Investitionsrendite nicht erzielen.

Ein Testautomatisierungsprojekt sollte als ein Softwareentwicklungsprojekt verstanden werden. Das bedeutet, dass Architekturdokumente, detaillierte Entwurfsdokumente, Reviews des Entwurfs und des Codes, Komponenten- und Komponentenintegrationstests, sowie ein abschließender Systemtest erforderlich sind. Beim Testen kann es zu unnötigen Verzögerungen oder Komplikationen kommen, wenn der verwendete Testautomatisierungscode instabil oder ungenau ist.

In Zusammenhang mit der Testautomatisierung führen Technical Test Analysten unter anderem folgende Aufgaben durch:

- Bestimmen, wer für die Testausführung verantwortlich sein wird (eventuell in Abstimmung mit einem Testmanager)
- Das für das Unternehmen am besten geeignete Werkzeug, den Zeitrahmen, die Kompetenzen innerhalb des Teams, die Wartungsanforderungen usw. auswählen (dies könnte bedeuten, dass entschieden wird ein eigenes Werkzeug zu entwickeln anstatt eines zu beschaffen)
- Die Anforderungen für die Schnittstellen zwischen dem Automatisierungswerkzeug und anderen Werkzeugen (z.B. Testmanagementwerkzeug, Fehlermanagementwerkzeug, Werkzeuge für die kontinuierliche Integration) definieren
- Adapter entwickeln, die für die Schnittstelle zwischen dem Testausführungswerkzeug und der Software unter Test erforderlich sein können
- Den Ansatz für die Automatisierung auswählen, d.h. einen schlüsselwortgetriebenen oder datengetriebenen Ansatz (siehe Abschnitt 6.1.1)
- In Zusammenarbeit mit dem Testmanager die Kosten für Implementierung (einschließlich Schulung) schätzen. Bei agilen Projekten erfolgt dies normalerweise in Projekt-/Sprintplanungs-sitzungen mit dem gesamten Team.
- Die Zeitplanung für das Automatisierungsprojekt erstellen und Zeit für die Wartung einplanen
- Die Test Analysten und Businessanalysten darin schulen, wie die Daten für die Automatisierung zu verwenden und zu liefern sind
- Bestimmen, wann und wie die automatisierten Tests ausgeführt werden
- Bestimmen, wie die Testergebnisse der automatisierten Tests mit denen der manuellen Tests kombiniert werden

In Projekten mit Schwerpunkt auf der Testautomatisierung kann auch ein Testautomatisierungs-entwickler (TAE) mit vielen dieser Aufgaben beauftragt werden (für weitere Informationen siehe ISTQB-Lehrplan Advanced Level Testautomatisierungsentwickler [ISTQB_ALTAE_SYL]). Je nach den Erfordernissen des Projekts und vorhandenen Präferenzen können bestimmte organisatorische Aufgaben auch von einem Testmanager übernommen werden. In agilen Projekten ist die Zuordnung dieser Aufgaben zu Rollen meist flexibler und weniger formal.

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.

6.1.1 Den Automatisierungsansatz auswählen

In diesem Abschnitt werden die folgenden Faktoren behandelt, die den Testautomatisierungansatz beeinflussen:

- Automatisieren über die grafische Benutzungsoberfläche
- Verwendung des datengetriebenen Testens
- Verwendung des schlüsselwortgetriebenen Testens
- Umgang mit Fehlerwirkungen und Ausfällen
- Berücksichtigen des Systemzustands

Der ISTQB-Lehrplan Advanced Level Testautomatisierungsentwickler [ISTQB_ALTAE_SYL] enthält weitere Informationen über die Auswahl der Automatisierungsansätze.

6.1.1.1 Automatisieren über die grafische Benutzungsoberfläche

Testautomatisierung ist nicht auf das Testen über die grafische Benutzungsoberfläche beschränkt. Es gibt Werkzeuge, die das automatisierte Testen auf API-Ebene, durch eine Kommandozeilenschnittstelle (engl. command-line interface, CLI) und über sonstige Schnittstellen der Software unter Test unterstützen. Eine der ersten Entscheidungen, die der Technical Test Analyst treffen muss, ist die Bestimmung der am besten geeigneten Schnittstelle, die für die Testautomatisierung verwendet werden soll. Allgemeine Testausführungswerkzeuge erfordern die Entwicklung von Adaptern für diese Schnittstellen; dieser Aufwand ist bei der Planung zu berücksichtigen.

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

Mitschnittwerkzeuge können als geeignete Ausgangsbasis für die Entwicklung von Automatisierungsskripten verwendet werden. 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).

6.1.1.2 Verwendung eines datengetriebenen Automatisierungsansatzes

Je nach Art der zu testenden Software 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 wäre nicht effizient für jeden dieser zu testenden Werte ein dediziertes automatisiertes Testskript zu entwickeln und zu warten. Dieses Problem wird üblicherweise gelöst, indem die Daten aus den Skripten in einen externen Speicher, z.B. eine Tabellenkalkulation oder eine Datenbank, verschoben werden. 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 erwarteten Ergebniswerte liefern (z.B. ein Wert, der in einem Textfeld angezeigt wird, oder eine Fehlermeldung). Dies wird als datengetriebenes Testen bezeichnet.

Bei diesem Ansatz werden zusätzlich zu den Testskripten, die die gelieferten Daten verarbeiten, ein Testrahmen und eine Infrastruktur benötigt, um die Ausführung des Skripts oder der Menge von Skripten zu unterstützen. Die eigentlichen Daten, die in der Tabellenkalkulation oder Datenbank gespeichert

sind, werden von Test Analysten erstellt, die sich mit der Geschäftslogik der Software auskennen. In agilen Projekten kann auch ein Vertreter des Fachbereichs (z.B. der Product Owner) an der Definition der Daten beteiligt sein, insbesondere für Abnahmetests. Diese Arbeitsteilung ermöglicht es den für die Entwicklung von Testskripten zuständigen Personen (z.B. dem Technical Test Analysten), sich auf die Implementierung intelligenter Automatisierungsskripte zu konzentrieren, während der Test Analyst Autor 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.

6.1.1.3 Verwendung eines schlüsselwortgetriebenen Ansatzes

Die sogenannte schlüsselwortgetriebene oder aktionswortgetriebene Ansatz 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, wird eine abstrakte Metasprache erstellt, die die auszuführenden Aktionen beschreibt, sie aber nicht direkt ausführt. Jede Aussage dieser Sprache beschreibt einen Geschäftsprozess (oder einen Teil davon), der getestet werden sollte. Beispiel: Schlüsselwörter für Geschäftsprozesse könnten sein: „Anmelden“, „Benutzer_anlegen“ oder „Benutzer_löschen“. Ein Schlüsselwort beschreibt eine abstrakte Aktion, die in der Anwendungsdomäne ausgeführt werden wird. Konkrete Aktionen bezeichnen die Interaktion mit der Softwareschnittstelle selbst, wie z.B.: „Schaltfläche_betätigen“, „Aus_Liste_auswählen“ oder „Baum_durchlaufen“. Diese können zum Testen jener Funktionen der grafischen Benutzungsoberfläche ebenfalls definiert und verwendet werden, die nicht genau zu den vorhandenen Schlüsselwörtern des Geschäftsprozesses passen.

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

Diese klare Aufteilung der Kenntnisse über Geschäftslogik und der eigentlichen Programmierung zur Implementierung der Testautomatisierungsskripte sorgt für die effektivste Nutzung der Testressourcen. Der Technical Test Analyst kann in seiner Rolle als Testautomatisierer seine Programmierfähigkeiten effektiv einbringen, ohne dass er dazu spezifische Fachkenntnisse verschiedenster Geschäftsbereiche benötigt.

Durch die Trennung des Codes von den sich veränderbaren Daten wird die Testautomatisierung von den Änderungen isoliert. Dies wiederum verbessert die Wartbarkeit des Codes insgesamt und steigert die Investitionsrendite der Testautomatisierung.

6.1.1.4 Umgang mit Fehlerwirkungen und Ausfällen

Beim Entwerfen der Testautomatisierung ist es wichtig, Fehlerwirkungen und Ausfälle der Software unter Test zu antizipieren und zu behandeln. Der Testautomatisierer muss festlegen, was die Automatisierungssoftware tun sollte, wenn eine Fehlerwirkung auftritt. Sollte die Fehlerwirkung aufgezeichnet werden und die Tests fortgesetzt werden? Sollten die Tests abgebrochen werden? Kann die Fehlerwirkung durch eine bestimmte Aktion (z.B. Betätigen einer Schaltfläche in einem Dialogfenster) oder vielleicht durch eine Verzögerung im Test behandelt werden? Nicht behandelte

Fehlerwirkungen in der Software unter Test können die Ergebnisse der nachfolgenden Tests unbrauchbar machen und Probleme mit dem Test verursachen, der zum Zeitpunkt des Auftretens der Fehlerwirkung ausgeführt wurde.

6.1.1.5 Berücksichtigen des Systemzustands

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

6.1.2 Geschäftsprozesse für die Automatisierung modellieren

Um einen schlüsselwortgetriebene Ansatz für die Testautomatisierung zu implementieren, müssen die zu testenden Geschäftsprozesse in der abstrakten schlüsselwortbasierten Sprache modelliert bzw. spezifiziert werden. Es ist wichtig, dass diese so gestaltet ist, dass die Benutzer (z.B. die Test Analysten im Projekt, bzw. die Product Owner als Vertreter des Fachbereichs in agilen Projekten) intuitiv damit arbeiten können.

Schlüsselwörter werden in der Regel verwendet, um abstrakte Geschäftsinteraktionen mit einem System unter Test abzubilden. Beispiel: Das Schlüsselwort „Auftrag_stornieren" umfasst die Überprüfung, ob der Auftrag existiert, die Verifizierung der Zugriffsrechte der Person, die die Stornierung veranlasst, die Anzeige des Auftrags, der storniert werden soll, und das Anfordern der Stornierungsbestätigung. Der Test Analyst verwendet Sequenzen von Schlüsselwörtern (z.B. „Anmelden", „Auftrag_auswählen", „Auftrag_stornieren") sowie die relevanten Testdaten, 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
Benutzer_anlegen	Benutzer1	Pass1	Benutzer hinzugefügt
Benutzer_anlegen	@Rec34	@Rec35	Benutzer hinzugefügt
Passwort_zurücksetzen	Benutzer1	Willkommen	Passwort zurückgesetzt
Benutzer_löschen	Benutzer1		Ungültiger Benutzername/Passwort
AddBenutzer_anlagen	Benutzer3	Pass3	Benutzer hinzugefügt
Benutzer_löschen	Benutzer2		Benutzer nicht gefunden

Das Automatisierungsskript, sucht in dieser Tabelle nach den Eingabewerten für das Automatisierungsskript. Beispiel: In der Zeile mit dem Schlüsselwort "Benutzer_löschen " wird nur der Benutzername benötigt. Um einen neuen Benutzer hinzuzufügen, werden sowohl der Benutzername als auch das Passwort benötigt. Es kann auch von einem Datenspeicher auf Eingabewerte verwiesen werden; dies ist in der zweiten Zeile beim Schlüsselwort "Benutzer_anlegen" der Fall, in der ein Verweis auf die Daten angegeben ist anstatt der Daten selbst. Dies erhöht die Flexibilität beim Zugriff auf Daten, die sich während der Testausführung ändern können. So können datengetriebene mit schlüsselwort-getriebenen Ansätze kombiniert werden.

Zu den Faktoren, die berücksichtigt werden sollten, gehören die folgenden:

- Je feingranularer die 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 konkreten Aktionen ("Schaltfläche_betätigen", "Aus_Liste_auswählen" usw.) spezifizieren, können die schlüsselwortgetriebenen Tests besser mit unterschiedlichen Situationen umgehen. Da diese Aktionen jedoch direkt mit der grafischen Benutzungsoberfläche 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 vorkommen, 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.

Die schlüsselwortgetriebene Testautomatisierung kann die Wartungskosten einer Testautomatisierung erheblich reduzieren, aber sie ist kostspieliger und schwieriger in der Entwicklung. Außerdem ist für den korrekte Entwurf ein größerer Zeitaufwand nötig, um die Testautomatisierung so umzusetzen, dass damit die angestrebte Investitionsrendite erzielt werden kann.

Der ISTQB-Lehrplan Advanced Level Testautomatisierungsentwickler [ISTQB_ALTAE_SYL] enthält weitere Informationen über die Modellierung von Geschäftsprozessen für die Automatisierung.
.

6.2 Spezifische Testwerkzeuge

Dieser Abschnitt enthält eine Übersicht über die Werkzeuge, die von Technical Test Analysten zusätzlich zu den im ISTQB Foundation Level-Lehrplan [ISTQB_FL_SYL] beschriebenen Werkzeugen mituntner eingesetzt werden.

Anmerkung: Detaillierte Informationen über Testwerkzeuge sind in den folgenden ISTQB-Lehrplänen enthalten:

- Mobile Application Testing [ISTQB_FLMAT_SYL]
- Performaztest [ISTQB_FLPT_SYL]
- Model-Based Testing [ISTQB_FLMBT_SYL]
- Testautomatisierungsentwickler [ISTQB_ALTAE_SYL]

6.2.1 Fehlereinpflanzungs- und Fehlereinfügungswerkzeuge

Fehlereinpflanzungswerkzeuge modifizieren den zu testenden Code (möglicherweise unter Verwendung vordefinierter Algorithmen), um die Überdeckung zu prüfen, die durch bestimmte Tests erreicht wird. Bei systematischer Anwendung kann so die Qualität der Tests (d.h. ihre Fähigkeit, die hinzugefügten Fehler zu erkennen) bewertet und gegebenenfalls verbessert werden.

Fehlereinfügungswerkzeuge fügen absichtlich falsche Eingaben in das Testobjekt ein, um sicherzustellen, dass die Software mit dem Fehler umgehen kann. Die Eingaben werden eingefügt, um die normalen Ausführungsmechanismen des Codes zu stören und eine breitere Testüberdeckung zu ermöglichen (z.B. um mehr ungültige Testbedingungen und Fehlerbehandlungsmechanismen abzudecken). Im Kontext von Fehlertoleranztests erzeugen oder simulieren Fehlereinfügungswerkzeuge Fehlerwirkungen, die in den Umgebungskomponenten der Software unter Test auftreten könnten.

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.

6.2.2 Performanztestwerkzeuge

Performanztestwerkzeuge haben die folgenden Hauptfunktionen:

- Lastgenerierung
- Messung, Überwachung, Visualisierung und Analyse des Antwortverhaltens des Systems unter bestimmter Last
- Einblicke in den Umgang von System- und Netzwerkkomponenten mit den Ressourcen geben

Zur Lastgenerierung wird ein vordefiniertes Nutzungsprofil (siehe Abschnitt 4.5.3) als Skript implementiert. Das Skript kann zunächst für einen einzelnen Benutzer erfasst (möglicherweise mit einem Mitschnittwerkzeug) und dann mittels des Lasttestwerkzeugs für das spezifizierte Nutzungsprofil implementiert werden. Die Implementierung muss die Variationen der Daten pro Transaktion (oder Menge von Transaktionen) berücksichtigen.

Performanztestwerkzeuge generieren Last, indem sie eine große Zahl von Benutzern ("virtuelle" Benutzer) simulieren, die gemäß ihrer festgelegten Nutzungsprofile Aufgaben ausführen und eine bestimmte Menge an Eingabedaten erzeugen. Im Gegensatz 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 Benutzerinteraktion über eine grafische Benutzungsoberfläche. Dadurch werden in der Regel weniger separate Testsitzungen benötigt. Einige Lastgenerierungswerkzeuge können die Anwendung auch über Benutzungsschnittstelle steuern, um so die Antwortzeiten des Systems unter Last genauer zu messen.

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

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

Wichtige Faktoren beim Implementieren von Performanztestwerkzeugen sind:

- Die für die Lastgenerierung benötigte Hardware und Netzwerkbandbreiten
- Kompatibilität des Werkzeugs mit dem Kommunikationsprotokoll des Systems unter Test
- Ausreichende Flexibilität des Werkzeugs für die einfache Implementierung unterschiedlicher Nutzungsprofile
- Bereitstellung benötigter Überwachungs-, Analyse- und Berichtsfunktionen

Performanztestwerkzeuge werden in der Regel erworben und nicht selbst entwickelt, da ihre Entwicklung aufwändig ist. Es kann jedoch sinnvoll sein, ein spezifisches Performanztestwerkzeug selbst zu entwickeln, wenn technische Einschränkungen den Einsatz eines verfügbaren Produkts nicht zulassen, oder wenn Nutzungsprofil und die bereitzustellenden Funktionen relativ einfach sind.

Weitere Informationen über den Performanztest sind im Foundation Level Performanztest-Lehrplan [ISTQB_FLPT_SYL] enthalten.

6.2.3 Werkzeuge für den webbasierten Test

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

- Hyperlink-Testwerkzeuge werden verwendet, um Websites zu scannen und zu prüfen, dass keine Hyperlinks ungültig sind
- 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 Benutzer eine Serververbindung herstellen
- Einfache Testausführungswerkzeuge, die mit unterschiedlichen Browsern funktionieren
- Werkzeuge zum Scannen des Servers, um nicht verlinkte Dateien zu identifizieren
- HTML-Syntaxprüfprogramme
- Cascading Style Sheet (CSS)-Prüfprogramme
- Werkzeuge zur Prüfung von Standardverletzungen (z.B. Abschnitt 508 des US-amerikanischen Barrierefreiheitsstandards, bzw. M/376 in Europa)
- Werkzeuge, die eine Reihe von IT-Sicherheitsproblemen identifizieren

Gute Quellen für „Open Source“-Werkzeuge für den webbasierten Test sind:

- Das World Wide Web Consortium (W3C) [Web-3]: Diese Organisation legt Standards für das Internet fest und stellt eine Vielzahl von Werkzeugen zur Verfügung, um Fehlerzustände zu identifizieren, die diese Standards verletzen.
- Die Web Hypertext Application Technology Working Group (WHATWG) [Web-5]: Diese Organisation legt HTML-Standards fest und verfügt über ein Werkzeug, das die HTML-Validierung durchführt [Web-6].

Einige Werkzeuge, 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 liefern (z.B. HTTP-Fehler 404). Dies sind nützliche Informationen für Entwickler, Webmaster und Tester.

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

6.2.4 Werkzeugunterstützung für modellbasiertes Testen

Modellbasiertes Testen (MBT) ist ein Testverfahren, bei dem ein formales Modell (wie z.B. endliche oder erweiterte Zustandsmaschinen) eingesetzt wird, um das erwartete Verhalten eines softwaregesteuerten Systems während der Ausführung zu beschreiben. Kommerzielle MBT-Werkzeuge (siehe [Utting07]) liefern oft eine Funktion, der es dem Benutzer ermöglicht, das Modell "auszuführen" um interessante Ausführungssequenzen zu speichern und diese als Testfälle zu verwenden. Auch andere ausführbare Modelle, wie beispielsweise Petri-Netze und Zustandsautomaten, unterstützen MBT.

MBT-Modelle (und -werkzeuge) können eingesetzt werden, um große Mengen unterschiedlicher Ausführungssequenzen zu generieren. Ebenfalls kann mithilfe von MBT-Werkzeugen die potentiell große Menge (Stichwort: Testfallexplosion) möglicher generierte Pfade reduziert werden Das Testen mit diesen Werkzeugen kann eine andere Sichtweise auf die zu testende Software bieten. Dies kann dazu führen, dass Fehlerzustände gefunden werden, die im funktionalen Test vielleicht übersehen wurden.

Weitere Informationen über Testwerkzeuge für modellbasiertes Testen sind im Foundation Level Model-Based Testing-Lehrplan [ISTQB_FLPT_SYL] enthalten.

6.2.5 Komponententest- und Build-Werkzeuge

Auch wenn Komponententest- und Build-Automatisierungswerkzeuge im Grunde Entwicklungswerkzeuge sind, werden sie vielfach von Technical Test Analysten verwendet und gewartet. Dies trifft insbesondere im Kontext eines agilen Entwicklungsmodells zu.

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

Debugging-Werkzeuge erleichtern den manuellen Komponententest auf der Code-Ebene. Sie ermöglichen es Entwicklern und Technical Test Analysten, Variablenwerte während der Ausführung zu ändern und den Code Zeile für Zeile auszuführen. Debugging-Werkzeuge werden von Entwicklern auch eingesetzt, um den Fehlerzustand im Code zu lokalisieren, wenn das Testteam eine Fehlerwirkung berichtet hat.

Build-Automatisierungswerkzeuge ermöglichen es nach jeder Änderung einer Softwarekomponente automatisch einen neuen Build-Lauf auszulösen. Nach Abschluss des Builds führen andere Werkzeuge die Komponententests automatisch aus. Dieser Grad der Automatisierung, bei der der Build-Prozess eingeschlossen ist, ist normalerweise Bestandteil kontinuierlicher Integrationsumgebungen.

Wenn sie korrekt aufgesetzt und konfiguriert werden, können diese Arten von Werkzeugen einen sehr positiven Effekt auf die Qualität der Builds haben, die zum Testen freigegeben 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.

6.2.6 Werkzeugunterstützung für das Testen mobiler Applikationen

Emulatoren und Simulatoren sind Werkzeuge, die häufig eingesetzt werden, um das Testen von mobilen Anwendungen zu unterstützen.

6.2.6.1 Simulatoren

Ein Simulator einer mobilen Plattform modelliert die Laufzeitumgebung des mobilen Endgerätes. Auf einem Simulator getestete Applikationen werden zu einer eigenen (dedizierten) Version kompiliert, die im Simulator, jedoch nicht auf einem realen Gerät funktioniert. Beim Testen werden Simulatoren manchmal auch als Ersatz für echte Geräte verwendet. Die auf einem Simulator getestete Anwendung unterscheidet sich jedoch von der App, die auf dem echten mobilen Endgerät verwendet wird.

6.2.6.2 Emulatoren

Ein Emulator modelliert die Hardware und verwendet dieselbe Laufzeitumgebung wie die physische Hardware. Applikationen, die kompiliert wurden, um auf einem Emulator bereitgestellt und getestet zu werden, könnten auch auf dem echten mobilen Endgerät verwendet werden.

Emulatoren werden häufig eingesetzt, um die Kosten von Testumgebungen zu reduzieren, indem sie echte Geräte ersetzen. Ein Emulator kann ein Gerät allerdings nicht vollständig ersetzen, da sich der Emulator möglicherweise anders verhält als das Mobilgerät, das er versucht nachzuahmen. Darüber hinaus werden einige Funktionen wie (Multi-)Touchfunktionen, Beschleunigungsmesser usw.

möglicherweise nicht unterstützt. Dies liegt zum Teil an den Einschränkungen der Plattform, auf der der Emulator ausgeführt wird.

6.2.6.3 Gemeinsame Aspekte

Simulatoren und Emulatoren sind in der frühen Entwicklungsphase sehr nützlich, da sie normalerweise in die Entwicklungsumgebungen integriert sind und eine frühzeitige Bereitstellung, Testen und Überwachung von Applikationen ermöglichen

Um den Emulator oder Simulator zu verwenden, muss dieser gestartet werden, die erforderliche App muss auf ihnen installiert werden und wird dann so getestet, als ob sie sich auf dem echten Gerät befände. Jede Entwicklungsumgebung für mobile Betriebssysteme wird normalerweise mit einem eigenen Emulator und Simulator geliefert. Es stehen Emulatoren und Simulatoren von Drittanbietern zur Verfügung.

In der Regel lassen sich verschiedene Nutzungsparameter auf den Emulatoren und Simulatoren einstellen. Diese Einstellungen können die Netzwerkemulation mit unterschiedlichen Geschwindigkeiten, Signalstärken und Paketverlusten, das Ändern der Ausrichtung, das Generieren von Unterbrechungen und die GPS-Standortdaten umfassen. Für einige dieser Einstellungen, beispielsweise für GPS-Standortdaten oder Signalstärken, kann dies sehr nützlich sein, da diese mit echten Geräten schwierig oder kostspielig zu replizieren sind.

Weitere Informationen sind im Foundation Level Mobile Application Testing-Lehrplan [ISTQB_FLMAT_SYL] enthalten.

7. Referenzen

7.1 Standards

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

- [RTCA DO-178C/ED-12C]: Software Considerations in Airborne Systems and Equipment Certification, RTCA/EUROCAE ED12C. 2013.
  Kapitel 2

- [ISO9126] ISO/IEC 9126-1:2001, Software Engineering – Software Product Quality
  Kapitel 4

- [ISO25010] ISO/IEC 25010 (2014) Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) System and software quality models
  Kapitel 2 und 4

- [ISO29119] ISO/IEC/IEEE 29119-4 International Standard for Software and Systems Engineering - Software Testing Part 4: Test techniques. 2015
  Kapitel 2

- [ISO42010] ISO/IEC/IEEE 42010:2011
  Systems and software engineering - Architecture description
  Kapitel 5

- [IEC61508] IEC 61508-5 (2010) Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems, Part 5: Examples of methods for the determination of safety integrity levels
  Kapitel 2

7.2 Dokumente des ISTQB

- [ISTQB_AL_OVIEW] Advanced Level Overview, Version 2019
- [ISTQB_ALSEC_SYL] Advanced Level Security Testing Syllabus, Version 2016
- [ISTQB_ALTAE_SYL] Advanced Level Test Automation Engineer Syllabus, Version 2017
- [ISTQB_FL_SYL] Foundation Level Syllabus, Version 2018
- [ISTQB_FLPT_SYL] Foundation Level Performance Testing Syllabus, Version 2018
- [ISTQB_FLMBT_SYL] Foundation Level Model-Based Testing Syllabus, Version 2015
- [ISTQB_ALTA_SYL] Advanced Level Test Analyst Syllabus, Version 2019
- [ISTQB_ALTM_SYL] Advanced Level Test Manager Syllabus, Version 2012
- [ISTQB_FLMAT_SYL] Foundation Level Mobile Application Testing Syllabus, 2019
- [ISTQB_GLOSSARY] Glossary of Terms used in Software Testing, Version 3.2, 2019

7.3 Fachliteratur

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

[Bath14] Graham Bath, Judy McKay, “The Software Test Engineer’s Handbook (2nd edition)”, Rocky Nook, 2014, 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
[Burns18] Brendan Burns, “Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services”, O’Reilly, 2018, ISBN 13: 978-1491983645
[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] 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
[McCabe96] Arthur H. Watson and Thomas J. McCabe. "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric" (PDF), 1996, NIST Special Publication 500-235.
[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
[Utting07] 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. Diese Referenzen wurden zum Zeitpunkt der Veröffentlichung dieses Advanced Level Lehrplans überprüft. Das ISTQB übernimmt keine Verantwortung dafür, wenn diese Referenzen nicht mehr verfügbar sind.

[Web-1] http://www.nist.gov NIST National Institute of Standards and Technology,
[Web-2] http://www.codeproject.com/KB/architecture/SWArchitectureReview.aspx
[Web-3] http://www.W3C.org
[Web-4] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project
[Web-5] https://whatwg.org
[Web-6] https://whatwg.org/validator/

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

8. Anhang A: Übersicht über die Qualitätsmerkmale

Die folgende Tabelle vergleicht die in ISO 9126 verwendeten Begriffe (wie im Technical Test Analyst-Lehrplan Version 2012 verwendet) mit den Begriffen der neueren [ISO25010] (wie in der vorliegenden Lehrplan-Version 2019 verwendet). Es werden nur die Qualitätsmerkmale aufgeführt, die für den Technical Test Analysten relevant sind.

ISO/IEC 25010	ISO/IEC 9126-1	Bemerkungen
Performanz	**Effizienz**	
Zeitverhalten	Zeitverhalten	
Ressourcennutzung	Ressourcennutzung	
Kapazität		Neues Untermerkmal
Kompatibilität		Neues Merkmal
Koexistenz	Koexistenz	Verschoben von Übertragbarkeit
Interoperabilität		Verschoben von Funktionalität (Test Analyst)
Zuverlässigkeit	**Zuverlässigkeit**	
Softwarereife	Reife	
Verfügbarkeit		Neues Untermerkmal
Fehlertoleranz	Fehlertoleranz	
Wiederherstellbarkeit	Wiederherstellbarkeit	
IT-Sicherheit	**Sicherheit**	War bisher ein Untermerkmal der Funktionalität und hatte keine Untermerkmale
Vertraulichkeit		Neues Untermerkmal
Datenintegrität		Neues Untermerkmal
Nichtabstreitbarkeit		Neues Untermerkmal
Zurechenbarkeit		Neues Untermerkmal
Authentizität		Neues Untermerkmal
Wartbarkeit	**Wartbarkeit**	
Modularität		Neues Untermerkmal
Wiederverwendbarkeit		Neues Untermerkmal
Analysierbarkeit	Analysierbarkeit	
Modifizierbarkeit	Stabilität	Kombiniert Änderbarkeit und Stabilität
	Änderbarkeit	
Testbarkeit	Testbarkeit	
Übertragbarkeit	**Übertragbarkeit**	
Anpassbarkeit	Anpassbarkeit	
Installierbarkeit	Installierbarkeit	
	Koexistenz	Verschoben zu Kompatibilität
Austauschbarkeit	Austauschbarkeit	
	Compliance	Gestrichen in 25010

9. Index