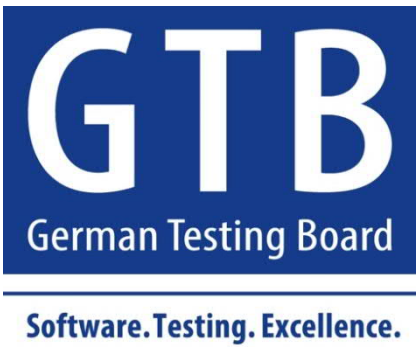
Lehrplan

A4Q
Testing Foundations
for Developers

Version 2021 V1.0 DE



A4Q Alliance for Qualification

Deutschsprachige Ausgabe.
Herausgegeben durch
Alliance for Qualification & German Testing Board e.V.

Zusammenstellung von, um Praktische Kompetenzstufen erweiterten, Lernzielen aus den Lehrplänen des International Software Testing Qualifications Board (ISTQB®), Originaltitel: Certified Tester Foundation Level Syllabus Version 2018 V3.1 sowie Certified Tester Advanced Level Syllabus Technical Test Analyst Version 2019 bzw. den entsprechenden durch GTB, ATB und STB lokalisierten deutschsprachigen Lehrplänen.

Urheberschutzvermerk

Autoren

Autoren und Beteiligte an der englischen Originalausgabe CTFL 2018 V3.1 sowie der Übersetzung in die deutsche Sprache CTFL 2018 V3.1D siehe [ISTQB_FL_SYL].

Autoren und Beteiligte an der englischen Originalausgabe CTAL-TTA 2019 sowie der Übersetzung in die deutsche Sprache CTAL-TTA 2019D siehe [ISTQB_ATTA_SYL].

Autoren an den deutschen & englischen praktischen Kompetenzstufen für A4Q-TF4D sowie die Auswahl der Lernziele aus den o.g. Lehrplänen: Mitglieder der GTB Arbeitsgruppe TF4D: Jürgen Beniermann, Daniel Fröhlich, Thorsten Geiselhart, Matthias Hamburg, Armin Metzger, Andreas Reuys, Erhardt Wunderlich.

Änderungsübersicht dieses Dokuments

Version	Datum	Bemerkung
A4Q-TF4D 2021 V1.0	19.03.2021	Testing Foundations for Developers

Änderungsübersicht der Originalausgaben

Änderungsübersicht der englischen Originalausgabe CTFL 2018 V3.1 sowie der Übersetzung in die deutsche Sprache CTFL 2018 V3.1D siehe [ISTQB_FL_SYL].

Änderungsübersicht der englischen Originalausgabe CTAL-TTA 2019 sowie der Übersetzung in die deutsche Sprache CTAL-TTA 2019D siehe [ISTQB_ATTA_SYL].

Inhaltsverzeichnis

Danksagung

Das Dokument A4Q Testing Foundations for Developers (A4Q-TF4D) wurde formal am 19.03.2021 durch A4Q und GTB veröffentlicht.

Die A4Q und das GTB bedanken sich bei den Mitgliedern der GTB Arbeitsgruppe TF4D: Jürgen Beniermann, Daniel Fröhlich, Thorsten Geiselhart, Matthias Hamburg, Armin Metzger, Andreas Reuys, Erhardt Wunderlich für ihr Engagement und ihre Mitwirkung an der Realisierung dieses Lehrplanes.

Das Team dankt den Reviewern für ihre Reviewbefunde zur Zusammenstellung des A4Q-TF4D:

Arne Becher, Florian Fieber, Dietrich Leimsner, Elke Mai, Carsten Weise

Das Team bedankt sich beim Backoffice für das technische Zusammenstellen der Lehrpläne aus den Originaldokumenten.

Das Team, die A4Q und das GTB bedanken sich bei Andreas Spillner, dem „Vater des Gedankens“ zu diesem Lehrplan mit „Grundlegenden Testfähigkeiten für Entwickler“. Er hatte die Idee, durch eine gezielte Auswahl von Lernzielen aus dem ISTQB Certified Tester Foundation Level und Ergänzungen einen auf die Bedürfnisse von Entwicklern zugeschnittenen Lehrplan zu etablieren. Mit der entsprechenden Schulung können Entwickler die Kenntnisse der für ihre Arbeit erforderlichen Testgrundlagen erwerben bzw. vertiefen, ohne dabei die Befürchtung haben zu müssen, zum Tester umgeschult zu werden. Über das erlangte Wissen wird die Brücke zwischen Entwicklern und Testern geschlagen und die Kommunikation zwischen beiden deutlich verbessert. Aus dem Gedanken heraus, die Vorteile des systematischen Testens für Entwickler zu verdeutlichen, entstand auch sein gemeinsam mit Ulrich Breymann geschriebenes Buch „Lean Testing für C++-Programmierer“ beim dpunkt.verlag, in dem alle grundlegenden Testverfahren für den Entwicklertest beschrieben und ganz konkret an Beispielen bis zur Erstellung der Testfälle dargelegt werden (www.leantesting.de).

0. Einführung in diesen Lehrplan

0.1 Zweck dieses Lehrplans

Dieser Lehrplan A4Q Testing Foundations for Developers (A4Q-TF4D) definiert die grundlegende Testfähigkeiten für einen Entwickler, basierend auf der Basisstufe (Foundation Level) sowie der Aufbaustufe (Advanced Level Technical Test Analyst) des Softwaretestausbildungsprogramms des International Software Testing Qualifications Board (im Folgenden kurz ISTQB® genannt).

Das GTB stellt diesen Lehrplan wie folgt zur Verfügung:

1. Seminaranbietern zur Erstellung von Kursmaterialien und zur Bestimmung angemessener Lehrmethoden.
2. Zertifizierungskandidaten zur Vorbereitung auf die Zertifizierungsprüfung (entweder als Teil eines Seminars oder unabhängig davon).
3. Der internationalen Software- und Systementwicklungs-Community zur Förderung des Berufsbildes des Software- und Systemtesters und als Grundlage für Bücher und Fachartikel.

Das GTB kann anderen Stellen die Nutzung dieses Lehrplans zu anderen Zwecken erlauben, sofern sie zuvor die schriftliche Erlaubnis des GTB erfragen und erhalten.

0.2 Grundlegende Testfähigkeiten für Entwickler (A4Q-TF4D)

Das Ziel dieses Lehrplanes ist, grundlegendende Testfähigkeiten an Entwickler zu vermitteln, nicht Entwickler zu Testern umzuschulen.

Die grundlegenden Testfähigkeiten für Entwickler (Testing Foundations for Developers – TF4D) basieren auf dem „Certified Tester“-Ausbildungsprogramm. Sie sollen die in allen Lebenszyklen notwendigen Testfähigkeiten adressieren, unabhängig davon, ob es eine spezifische Rolle „Tester“ gibt.

Dieser Lehrplan zielt im Speziellen auf Softwareentwickler ab, richtet sich ebenso an Personen in der Rolle Product Owner, Projektmanager, Qualitätsmanager, Softwareentwicklungsmanager, Systemanalytiker (Businessanalysten), IT-Leiter oder Managementberater, welche sich die grundlegenden Testfähigkeiten für Entwickler erwerben möchten.

0.3 Geschäftlicher Nutzen

In diesem Abschnitt wird der geschäftliche Nutzen (Business Outcomes) aufgelistet, der von Kandidaten mit einer Zertifizierung als A4Q-TF4D erwartet werden kann.

Ein A4Q-TF4D kann …

TF4D-BO1 …effiziente und effektive Kommunikation durch Nutzung eines gemeinsamen Vokabulars für den Softwaretest fördern. 1

TF4D-BO2 …die grundlegenden Konzepte des Softwaretestens verstehen. 2

TF4D-BO3 …etablierte Black-Box-Verfahren für den Testentwurf anwenden. 3

TF4D-BO4 …etablierte White-Box-Verfahren für den Testentwurf anwenden. 4

TF4D-BO5 …Tests gemäß vorgegebenen Testentwürfen implementieren.

TF4D-BO6 …implementierte Tests ausführen und Ergebnisse berichten.

TF4D-BO7 …die Qualitätsmerkmale von Code und Architektur durch den Einsatz verschiedener Analysetechniken verbessern. 5

0.4 Prüfbare Lernziele und kognitive Stufen des Wissens

Lernziele unterstützen den geschäftlichen Nutzen und werden verwendet, um die Prüfungen für A4Q-TF4D zu erstellen.

Im Allgemeinen sind alle Inhalte dieses Lehrplans auf K1-Stufe prüfbar, außer der Einführung und der Anhänge. Das bedeutet, vom Prüfling kann gefordert werden, Schlüsselbegriffe oder Konzepte aus jedem Kapitel zu erkennen, sich daran zu erinnern oder wiedergeben zu können. Die Wissensstufen der spezifischen Lernziele werden am Beginn jedes Kapitels genannt und sind wie folgt klassifiziert:

- K1: erinnern
- K2: verstehen
- K3: anwenden

Die Definitionen aller Begriffe, die als Schlüsselbegriffe unterhalb der Kapitelüberschrift aufgelistet sind, sollen im Gedächtnis behalten werden (K1), auch wenn sie nicht ausdrücklich in den Lernzielen erwähnt werden.

Der Lehrplan enthält Lernziele die explizit als *nicht prüfungsrelevant 6* gekennzeichnet sind. Diese Lernziele beinhalten weiterführende Kenntnisse, die erforderlich sind, um die grundlegenden Testfähigkeiten für einen Entwickler abzurunden. Die Kernpunkte dieser Lernziele werden auf K1-Niveau übersichtsmäßig vermittelt.

1 Übernommen von FL-BO1 [ISTQB_FL_OVW]
2 Übernommen von FL-BO2 [ISTQB_FL_OVW]
3 Angepasst aus FL-BO5 [ISTQB_FL_OVW]
4 Angepasst aus TTA3 [ISTQB_AL_OVW]
5 Angepasst aus TTA3 [ISTQB_AL_OVW]
6 Anmerkung: Im Folgenden sind alle Teile des Lehrplans die nicht prüfungsrelevant sind, aber aus verschiedenen Gründen einen sinnvollen Kontext setzen, in blauer, kursiver Schrift hinterlegt.

0.5 Praktische Kompetenzstufen

In den “Grundlegenden Testfähigkeiten für Entwickler” wird das Konzept der praktischen Kompetenzstufen angewendet, welche sich auf praktische Fähigkeiten und Kompetenzen konzentrieren.

Durch praktische Übungen können Kompetenzen erworben werden, wie sie in der folgenden nicht erschöpfenden Liste aufgeführt sind:

- Übungen für Lernziele der Stufe K3, die auf Papier oder mit Textverarbeitungssoftware durchgeführt werden.
- Einrichten und Verwenden von Testumgebungen.
- Testen von Applikationen auf virtuellen und physischen Geräten.
- Verwenden von Werkzeugen zum Testen oder für Aufgaben zur Unterstützung des Testens.

Für die praktischen Ziele gelten die folgenden praktischen Kompetenzstufen:

- H0: Dies kann eine Live-Demo einer Übung oder ein aufgezeichnetes Video beinhalten. Da dies nicht vom Kursteilnehmer selbst durchgeführt wird, ist es streng genommen keine Übung.
- H1: Geführte Übung. Die Kursteilnehmer befolgen eine Abfolge von Schritten, die vom Seminarleiter ausgeführt werden.
- H2: Übung mit Tipps. Die Kursteilnehmer erhalten eine Übung mit relevanten Hinweisen zur Lösung der Übung innerhalb eines vorgegebenen Zeitrahmens.
- H3: Freie Übungen ohne Tipps.

0.6 Die Prüfung

Die Prüfung zu den grundlegenden Testfähigkeiten für Entwickler basiert auf diesem Lehrplan. Antworten auf Prüfungsfragen können die Nutzung von Materialien auf Grundlage von mehr als einem Abschnitt dieses Lehrplans erfordern. Die verwendeten Schlüsselbegriffe gemäß Glossar und alle Abschnitte dieses Lehrplans sind, sofern nicht als *nicht prüfungsrelevant* [7] gekennzeichnet, außer der Einführung und der Anhänge. Standards, Bücher und ISTQB®-Lehrpläne sind als Referenzen genannt, aber deren Inhalt ist nicht über das hinaus prüfungsrelevant, was in diesem Lehrplan selbst aus diesen Standards, Büchern oder ISTQB®-Lehrplänen zusammengefasst ist.

Das Format der Prüfung ist Multiple Choice. Es gibt 40 Fragen. Zum Bestehen der Prüfung müssen mindestens 65% der Fragen (also 26 Fragen) korrekt beantwortet werden. Praktische Kompetenzstufen und Übungen werden nicht abgeprüft.

Prüfungen können als Teil eines akkreditierten Seminars oder unabhängig davon (z. B. in einem Prüfungszentrum oder in einer öffentlichen Prüfung) absolviert werden. Der Abschluss eines akkreditierten Seminars ist keine Voraussetzung für die Teilnahme an der Prüfung.

7 Anmerkung: Im Folgende sind alle Teile des Lehrplans die nicht prüfungsrelevant sind, aber aus verschiedenen Gründen einen sinnvollen Kontext setzen, in blauer, kursiver Schrift hinterlegt.

0.7 Geschlechtsneutrale Formulierungen

Aus Gründen der einfacheren Lesbarkeit wird auf die geschlechtsneutrale Differenzierung, wie beispielsweise Benutzer/innen, verzichtet. Sämtliche Rollenbezeichnungen gelten im Sinne der Gleichbehandlung grundsätzlich für alle Geschlechter.

0.8 Akkreditierung

Das GTB bzw. A4Q akkreditieren Seminaranbieter, deren Kursmaterialien diesem Lehrplan entsprechen. Die Akkreditierungsrichtlinien können beim German Testing Board e.V. bezogen werden. Ein akkreditierter Kurs ist als zu diesem Lehrplan konform anerkannt und darf eine A4Q-TF4D-Prüfung beinhalten.

0.9 Wie ist dieser Lehrplan aufgebaut

Es gibt vier Kapitel mit prüfungsrelevantem Inhalt. Die Hauptüberschrift für jedes Kapitel legt die Zeit fest, die für das Kapitel vorgesehen ist. Für die Unterkapitel ist keine Zeitangabe vorhanden. Für akkreditierte Seminare erfordert dieser Lehrplan mindestens 14,25 Unterrichtsstunden, die sich auf folgende Kapitel aufteilen:

- Kapitel 1: Grundlagen des Testens - 110 Minuten
- Kapitel 2: Testen im Softwareentwicklungslebenszyklus - 70 Minuten
- Kapitel 3: Statischer Test - 225 Minuten
- Kapitel 4: Testverfahren - 450 Minuten

1. Grundlagen des Testens – 110 Minuten

Schlüsselbegriffe

Debugging, Fehlerwirkung, Fehlerzustand, Fehlhandlung, Grundursache, Qualität, Qualitätssicherung, Verfolgbarkeit, Testablauf, Testanalyse, Testbasis, Testbedingung, Testdurchführung, Testdaten, Testen, Testentwurf, Testfall, Testobjekt, Testorakel, Testprozess, Testrealisierung, Testsuite, Testziel, Überdeckung, Validierung, Verifizierung

Lernziele für die Grundlagen des Testens

1.1 Was ist Testen?

FL-1.1.1 (K1) Typische Ziele des Testens identifizieren können

FL-1.1.2 (K2) Testen von Debugging unterscheiden können

1.2 Warum ist Testen notwendig?

FL-1.2.1 (K2) Beispiele dafür geben können, warum Testen notwendig ist

FL-1.2.2 (K2) Die Beziehung zwischen Testen und Qualitätssicherung beschreiben können und Beispiele dafür geben können, wie Testen zu höherer Qualität beiträgt

FL-1.2.3 (K2) Zwischen Fehlhandlung, Fehlerzustand und Fehlerwirkung unterscheiden können

FL-1.2.4 (K2) Zwischen der Grundursache eines Fehlerzustands und seinen Auswirkungen unterscheiden können

1.3 Sieben Grundsätze des Testens

FL-1.3.1 (K2) Die sieben Grundsätze des Softwaretestens erklären können

1.4 Testprozess (nicht prüfungsrelevant)

FL-1.4.1 (K2) Die Auswirkungen des Kontexts auf den Testprozess erklären können

FL-1.4.2 (K2) Die Testaktivitäten und zugehörigen Aufgaben innerhalb des Testprozesses beschreiben können

FL-1.4.3 (K2) Arbeitsergebnisse unterscheiden können, die den Testprozess unterstützen

FL-1.4.4 (K2) Die Bedeutung der Pflege der Verfolgbarkeit zwischen Testbasis und Testarbeitsergebnissen erklären können

1.1 Was ist Testen?

Softwaresysteme sind ein wesentlicher Bestandteil des Lebens: von Fachanwendungen (z. B. im Bankwesen) bis hin zu Verbraucherprodukten (z. B. Autos). Die meisten Menschen haben bereits Erfahrungen mit Software gemacht, die nicht wie erwartet funktioniert hat. Software, die nicht korrekt arbeitet, kann zu vielfältigen Problemen führen, u. a. zu Geld-, Zeit- oder Imageverlust, sogar bis hin zu Verletzungen oder Tod. Softwaretesten ist ein Mittel, die Qualität von Software zu beurteilen und das Risiko einer Fehlerwirkung im Betrieb zu reduzieren.

Es ist eine gängige Fehleinschätzung, dass Testen ausschließlich darin besteht, Tests auszuführen, d.h. im Sinne von: die Software auszuführen und die Ergebnisse zu prüfen. Wie in Abschnitt *1.4 Testprozess* beschrieben, ist Softwaretesten ein Prozess, der viele unterschiedliche Aktivitäten umfasst. Die Testdurchführung einschließlich der Prüfung der Ergebnisse ist nur eine dieser Aktivitäten. Der Testprozess beinhaltet darüber hinaus Aktivitäten wie die Planung, die Analyse, den Entwurf und die Realisierung von Tests, das Berichten über Testfortschritt und -ergebnisse und die Beurteilung der Qualität eines Testobjekts.

Einige Tests beinhalten die Ausführung von Komponenten oder Systemen; derartige Tests werden dynamische Tests genannt. Andere Tests umfassen kein Ausführen von zu testenden Komponenten oder Systemen; derartige Tests werden statische Tests genannt. Testen beinhaltet also auch die Prüfung von Arbeitsergebnissen wie Anforderungen, User-Stories und Quellcode im Rahmen von Reviews.

Eine weitere gängige Fehleinschätzung ist, dass sich Testen ausschließlich auf die Verifizierung von Anforderungen, User-Stories oder anderen Spezifikationen konzentriert. Auch wenn das Testen es erfordert, zu prüfen, ob das System spezifische Anforderungen erfüllt, so umfasst es auch die Validierung, also die Prüfung, ob das System in seiner Einsatzumgebung die Bedürfnisse von Benutzern und anderen Stakeholdern erfüllen wird.

Testaktivitäten werden in unterschiedlichen Lebenszyklen unterschiedlich organisiert und durchgeführt [ISTQB_FL_SYL – Abschnitt 2.1 Softwarelebenszyklus-Modelle].

1.1.1 Typische Ziele des Testens

Für jedes mögliche Projekt können die Ziele des Testens Folgendes beinhalten:

- Arbeitsergebnisse wie Anforderungen, User-Stories, Architekturdesign und Code bewerten, um Fehler zu identifizieren und in Folgearbeitsergebnissen zu vermeiden
- Verifizieren, ob alle spezifischen Anforderungen erfüllt sind
- Prüfen, ob das Testobjekt vollständig ist und validieren, ob das Testobjekt so funktioniert, wie es die Benutzer und andere Stakeholder erwarten
- Vertrauen in das Qualitätsniveau des Testobjekts schaffen
- Fehlerwirkungen und Fehlerzustände aufdecken, wodurch man Risiken aufgrund einer unzureichenden Softwarequalität reduziert
- Stakeholdern ausreichende Informationen zur Verfügung stellen, damit diese fundierte Entscheidungen treffen können, insbesondere bezüglich des Qualitätsniveaus des Testobjekts
- Konform mit vertraglichen, rechtlichen oder regulatorischen Anforderungen oder Standards zu sein und/oder um die Konformität (compliance) des Testobjekts mit diesen Anforderungen oder Standards zu verifizieren

Die Ziele des Testens können abhängig vom Kontext der zu testenden Komponente oder des Systems, das getestet wird, von der Teststufe und dem Softwareentwicklungslebenszyklus-Modell variieren. Diese Unterschiede können zum Beispiel Folgendes beinhalten:

- Während des Komponententests kann es ein Ziel sein, so viele Fehlerwirkungen wie möglich zu finden, damit die zugrunde liegenden Fehlerzustände frühzeitig identifiziert und behoben werden. Ein weiteres Ziel kann es sein, die Codeüberdeckung durch Komponententests zu erhöhen.
- Während des Abnahmetests kann es ein Ziel sein, zu bestätigen, dass das System so funktioniert wie erwartet und die Anforderungen erfüllt wurden. Ein weiteres Ziel dieses Tests kann darin bestehen, Stakeholdern Informationen über das Risiko einer Systemfreigabe zu einem festgelegten Zeitpunkt zu geben.

1.1.2 Testen und Debugging

Testen und Debugging sind unterschiedliche Dinge. Die Durchführung von Tests kann Fehlerwirkungen aufzeigen, die durch Fehlerzustände in der Software hervorgerufen werden. Debugging ist im Gegensatz dazu die Entwicklungsaktivität, die solche Fehlerzustände findet, analysiert und behebt. Die nachfolgenden Fehlernachtests prüfen, ob die Debugging-Maßnahmen die ursprünglichen Fehlerzustände behoben haben. In manchen Fällen sind Tester für die ursprünglichen Tests und die abschließenden Fehlernachtests verantwortlich, während Entwickler das Debugging, die zugehörigen Komponenten- und Komponentenintegrationstests (kontinuierliche Integration) durchführen. Dennoch können in der agilen Softwareentwicklung und in einigen anderen Softwareentwicklungslebenszyklusmodellen Tester auch am Debugging und im Komponententest involviert sein.

Die ISO-Norm (ISO/IEC/IEEE 29119-1) beinhaltet weitere Informationen über Konzepte des Softwaretestens.

1.2 Warum ist Testen notwendig?

Gründliches Testen von Komponenten oder Systemen und ihrer zugehörigen Dokumentation kann dabei helfen, das Risiko von Fehlerwirkungen zu reduzieren, die während des Betriebs auftreten können. Wenn Fehler entdeckt und in der Folge behoben werden, trägt dies zur Qualität der Komponenten oder Systeme bei. Darüber hinaus kann Softwaretesten auch notwendig sein, um vertragliche oder rechtliche Anforderungen oder branchenspezifische Standards zu erfüllen.

1.2.1 Der Beitrag des Testens zum Erfolg

Historisch betrachtet ist es schon immer üblich gewesen, dass Software und Systeme in Betrieb genommen werden, obwohl Fehlerwirkungen infolge von Fehlerzuständen auftreten oder in anderer Weise die Bedürfnisse der Stakeholder nicht erfüllt werden. Allerdings kann der Einsatz geeigneter Testverfahren die Häufigkeit derartiger problematischer Inbetriebnahmen reduzieren, wenn diese Verfahren mit dem entsprechenden Grad an Testkompetenz, in den geeigneten Teststufen und zum richtigen Zeitpunkt im Softwareentwicklungslebenszyklus eingesetzt werden. Beispiele dafür sind u. a.

- Der Einbezug von Testern in Anforderungsreviews oder bei User-Stories-Verfeinerungen (Refinements) kann Fehlerzustände in diesen Arbeitsergebnissen aufdecken. Die Identifizierung und Entfernung von Fehlerzuständen in Anforderungen reduzieren das Risiko der Entwicklung von fehlerhaften oder nicht testbaren Features.
- Die enge Zusammenarbeit von Testern mit Systementwicklern während des Systementwurfs kann das gemeinsame Verständnis für den Entwurf und die Möglichkeiten, es zu testen, verbessern. Dieses bessere Verständnis kann das Risiko grundlegender Entwurfsfehler reduzieren und die Identifikation potenzieller Tests zu einem frühen Zeitpunkt ermöglichen.
- Die enge Zusammenarbeit von Testern und Entwicklern während der Entwicklung des Codes kann das gemeinsame Verständnis der Beteiligten für den Code und wie dieser zu testen ist,

verbessern. Dieses bessere Verständnis kann das Risiko von Fehlerzuständen innerhalb des Codes und in Tests reduzieren.

- Die Verifizierung und Validierung der Software durch Tester vor der Freigabe kann Fehlerwirkungen aufdecken, die andernfalls übersehen worden wären, und kann den Prozess des Entfernens (d.h. Debugging) von Fehlern, die die Fehlerwirkungen hervorgerufen haben, unterstützen. Dies erhöht die Wahrscheinlichkeit, dass die Software den Bedürfnissen der Stakeholder entspricht und die Anforderungen erfüllt.

Zusätzlich zu diesen Beispielen trägt das Erreichen definierter Testziele (siehe Abschnitt 1.1.1 *Typische Ziele des Testens*) zum allgemeinen Erfolg der Softwareentwicklung und der Wartung bei.

1.2.2 Qualitätssicherung und Testen

Obwohl der Begriff *Qualitätssicherung* (oder kurz QS) oft in Bezug auf das Testen genutzt wird, sind Qualitätssicherung und Testen nicht das Gleiche, allerdings sind sie eng verwandt. Ein allgemeineres Konzept, das Qualitätsmanagement, vereint die beiden Begriffe. Qualitätsmanagement beinhaltet alle Aktivitäten, die eine Organisation im Hinblick auf Qualität leiten und steuern. Neben anderen Aktivitäten beinhaltet Qualitätsmanagement sowohl Qualitätssicherung als auch Qualitätssteuerung. Qualitätssicherung konzentriert sich in der Regel auf die Einhaltung gültiger Prozesse, um Vertrauen dafür zu schaffen, dass die angemessenen Qualitätsgrade erreicht werden. Wenn Prozesse korrekt befolgt werden, weisen die Arbeitsergebnisse, die aus solchen Prozessen hervorgehen, in der Regel eine höhere Qualität auf, was zur Vermeidung von Fehlerzuständen beiträgt. Darüber hinaus ist die Nutzung der Grundursachenanalyse zur Feststellung und Behebung von Fehlerzuständen in Verbindung mit der korrekten Verwendung der Befunde von Retrospektiven (Bewertungssitzungen) zur Verbesserung von Prozessen von Bedeutung für eine effektive Qualitätssicherung.

Qualitätssteuerung beinhaltet verschiedene Aktivitäten, darunter Testaktivitäten, die das Erreichen von angemessenen Qualitätsgraden unterstützen. Testaktivitäten sind Teil des gesamten Softwareentwicklungs- und -wartungsprozesses. Da die Qualitätssicherung sich mit der korrekten Ausführung des gesamten Prozesses beschäftigt, unterstützt Qualitätssicherung korrektes Testen. Wie in den Abschnitten 1.1.1 *Typische Ziele des Testens* und 1.2.1 *Der Beitrag des Testens zum Erfolg* beschrieben, trägt Testen zum Erreichen von Qualität auf verschiedene Arten bei.

1.2.3 Fehlhandlungen, Fehlerzustände und Fehlerwirkungen

Eine Person kann eine Fehlhandlung vornehmen (einen Fehler machen), was zur Entstehung eines Fehlerzustands (auch nur Fehler oder Bug genannt) im Softwarecode oder in einem zugehörigen Arbeitsergebnis führt. Eine Fehlhandlung, die zur Entstehung eines Fehlerzustands in einem Arbeitsergebnis führt, kann eine weitere Fehlhandlung hervorrufen, die wiederum zu der Entstehung eines Fehlerzustands in einem damit zusammenhängenden Arbeitsergebnis führt. Zum Beispiel kann eine Ungenauigkeit bei der Erhebung einer Anforderung zu einer fehlerhaften Beschreibung der Anforderung führen, die dann zu einem Programmierfehler und weiter zu einem Fehlerzustand im Code führt.

Wenn ein Fehlerzustand im Code ausgeführt wird, kann dies eine Fehlerwirkung hervorrufen, aber nicht notwendigerweise unter allen Umständen. Zum Beispiel erfordern einige Fehlerzustände sehr spezifische Eingaben oder Vorgaben, um eine Fehlerwirkung hervorzurufen, was nur selten oder niemals eintritt.

Fehlhandlungen können aus vielfältigen Gründen entstehen, wie z. B.:

- Zeitdruck
- Menschliche Fehlbarkeit
- Unerfahrene oder nicht ausreichend ausgebildete Projektbeteiligte

- Fehlkommunikation zwischen Projektbeteiligten, darunter Fehlkommunikation über Anforderungen und Entwurf
- Komplexität des Codes, des Entwurfs, der Architektur des zugrunde liegenden Problems, das gelöst werden soll, und/oder der genutzten Technologien
- Missverständnisse über systeminterne und systemübergreifende Schnittstellen, insbesondere wenn es eine Vielzahl solcher Schnittstellen gibt
- Neue, unbekannte Technologien

Zusätzlich zu Fehlerwirkungen, die durch Fehlerzustände im Code hervorgerufen werden, können Fehlerwirkungen auch durch Umweltbedingungen ausgelöst werden. Zum Beispiel können Strahlungen, elektromagnetische Felder und Beschmutzung zu Fehlerzuständen in Firmware führen oder die Ausführung von Software durch die Änderung von Hardwarebedingungen beeinflussen.

Nicht alle unerwarteten Testergebnisse sind Fehlerwirkungen. Falsch positive Ergebnisse können aufgrund von Fehlhandlungen in der Durchführung der Tests oder aufgrund von Fehlerzuständen in den Testdaten, der Testumgebung oder anderen Testmitteln oder aus anderen Gründen entstehen. Die umgekehrte Situation kann ebenfalls entstehen, wenn ähnliche Fehlhandlungen oder Fehlerzustände zu „falsch negativen“ Ergebnissen führen. „Falsch negative“ Ergebnisse sind Tests, die Fehlerzustände nicht entdecken, die sie hätten entdecken sollen; „falsch positive“ Ergebnisse sind berichtete Fehlerzustände, die tatsächlich keine Fehlerzustände sind.

1.2.4 Fehlerzustände, Grundursachen und Wirkungen

Die Grundursachen von Fehlerzuständen sind die frühesten Aktionen oder Bedingungen, die zur Entstehung der Fehlerzustände beigetragen haben. Fehlerzustände können analysiert werden, um ihre Grundursachen zu identifizieren und so das Auftreten von ähnlichen Fehlerzuständen in der Zukunft zu verhindern. Durch die Konzentration auf die bedeutendsten Grundursachen kann die Grundursachenanalyse zu Prozessverbesserungen führen, die das Auftreten einer Vielzahl von zukünftigen Fehlerzuständen verhindert.

Angenommen, falsche Zinszahlungen aufgrund einer einzigen Zeile falschen Codes führen zu Kundenbeschwerden. Der fehlerhafte Code wurde aufgrund einer User-Story geschrieben, die wegen des Missverständnisses des Product Owners darüber, wie der Zins zu berechnen sei, unklar formuliert war. Wenn ein Großteil der Fehlerzustände im Bereich der Zinsberechnung auftreten und diese Fehlerzustände ihre Grundursache in ähnlichen Missverständnissen haben, könnte der Product Owner im Bereich der Zinsberechnung geschult werden, um diese Art von Fehlern in der Zukunft zu vermeiden.

In diesem Beispiel zeigen die Kundenreklamationen Auswirkungen. Die falschen Zinszahlungen sind die direkten Fehlerwirkungen. Die falsche Berechnung im Code ist ein Fehlerzustand und dieser resultierte aus dem ursprünglichen Fehlerzustand, der Ungenauigkeit in der User-Story. Die Grundursache des ursprünglichen Fehlerzustands war ein Fehlen an Wissen auf Seiten des Product Owners, das dazu führte, dass er beim Schreiben der User-Story eine Fehlerhandlung beging. Der Prozess der Grundursachenanalyse wird im [ISTQB_ELTM_SYL] und [ISTQB_EITP_SYL] beschrieben.

1.3 Sieben Grundsätze des Testens

In den letzten 50 Jahren wurde eine Reihe von Grundsätzen für das Testen entwickelt, die generelle Leitlinien für alle Tests liefern.

1. Testen zeigt die Anwesenheit von Fehlerzuständen, nicht deren Abwesenheit

Testen kann zeigen, dass Fehlerzustände vorliegen, aber es kann nicht beweisen, dass es keine Fehlerzustände gibt. Testen reduziert die Wahrscheinlichkeit, dass noch unentdeckte Fehlerzustände

in der Software vorhanden sind, aber auch wenn keine Fehlerzustände gefunden werden, ist Testen kein Beweis für Korrektheit.

2. Vollständiges Testen ist nicht möglich

Ein vollständiger Test, bei dem alle möglichen Eingabewerte und deren Kombinationen unter Berücksichtigung aller unterschiedlichen Vorbedingungen ausgeführt werden, ist nicht durchführbar, mit Ausnahme von sehr trivialen Testobjekten. Anstatt zu versuchen, vollständig zu testen, sollten Risikoanalyse, Testverfahren und Prioritäten genutzt werden, um den Testaufwand zu konzentrieren.

3. Frühes Testen spart Zeit und Geld

Um Fehlerzustände früh zu finden, sollten sowohl statische als auch dynamische Testaktivitäten so früh wie möglich im Softwareentwicklungslebenszyklus gestartet werden. Frühes Testen wird oft als *Shift left* bezeichnet. Frühes Testen im Softwareentwicklungslebenszyklus hilft dabei, kostenintensive Änderungen zu reduzieren oder vollständig zu vermeiden (siehe Abschnitt *3.1 Grundlagen des statischen* Tests).

4. Häufung von Fehlerzuständen

Eine kleine Anzahl von Modulen enthält in der Regel die meisten Fehlerzustände, die während des Testens in der Phase vor Inbetriebnahme entdeckt werden, oder ist verantwortlich für die meisten der betrieblichen Fehlerwirkungen. Vorausgesagte Anhäufungen von Fehlerzuständen und die tatsächlich beobachteten Anhäufungen von Fehlerzuständen im Test oder im Betrieb sind ein wichtiger Beitrag zur Risikoanalyse, die genutzt wird, um den Testaufwand zu konzentrieren (wie in Grundsatz 2 erwähnt).

5. Vorsicht vor dem Pestizid-Paradoxon

Wenn die gleichen Tests immer wieder wiederholt werden, finden diese Tests irgendwann keine neuen Fehlerzustände mehr. Um neue Fehlerzustände zu finden, müssen bestehende Tests und Testdaten möglicherweise verändert werden und neue Tests geschrieben werden (Tests sind nicht länger effektiv im Erkennen von Fehlerzuständen, so wie Pestizide nach einer Weile nicht mehr effektiv in der Vernichtung von Insekten sind). In manchen Fällen, wie dem automatisierten Regressionstest, hat das Pestizid-Paradoxon einen vermeintlich positiven Effekt, der in der relativ geringen Anzahl von Regressionsfehlern liegt.

6. Testen ist kontextabhängig

Je nach Einsatzgebiet und Kontext ist das Testen anzupassen. Zum Beispiel wird sicherheitskritische industrielle Steuerungssoftware anders getestet als eine mobile E-Commerce-Applikation. Ein weiteres Beispiel ist das Testen in agilen Projekten, das anders durchgeführt wird als das Testen in einem Projekt mit sequenziellem Softwareentwicklungslebenszyklus [ISTQB_FL_SYL – Abschnitt 2.1 Softwarelebenszyklus-Modelle].

7. Trugschluss: „Keine Fehler“ bedeutet ein brauchbares System

Einige Unternehmen erwarten, dass Tester alle denkbaren Tests durchführen und alle denkbaren Fehlerzustände finden können, aber die Grundsätze 2 und 1 lehren uns, dass dies unmöglich ist. Des Weiteren ist es ein Trugschluss (d.h. ein Irrglaube), zu erwarten, dass *allein* das Finden und Beheben einer großen Anzahl von Fehlerzuständen den Erfolg eines Systems sicherstellen werde. Beispielsweise kann trotz gründlicher Tests aller spezifizierten Anforderungen und Beheben aller gefundenen Fehlerzustände ein System erstellt werden, das schwer zu nutzen ist, dass die Bedürfnisse und Erwartungen der Benutzer nicht erfüllt oder das geringwertigere Qualität hat als vergleichbare Systeme.

Siehe Myers 2011, Kaner 2002, Weinberg 2008 und Beizer 1990 für Beispiele dieser und anderer Grundsätze des Testens.

1.4 *Testprozess (nicht prüfungsrelevant)*

Es gibt nicht den einen universellen Softwaretestprozess, aber es gibt eine Reihe von gebräuchlichen Testaktivitäten. Ohne diese Aktivitäten erreicht das Testen die festgelegten Ziele mit einer weit geringeren Wahrscheinlichkeit. Diese Menge von Testaktivitäten bildet den Testprozess. Der geeignete, spezifische Softwaretestprozess in einer vorgegebenen Situation hängt von vielen Faktoren ab. Welche Testaktivitäten in diesem Testprozess beinhaltet sind, wie diese Aktivitäten eingesetzt werden und wann diese Aktivitäten stattfinden, kann in der Teststrategie eines Unternehmens behandelt werden.

1.4.1 Testprozess im Kontext (nicht prüfungsrelevant)

Kontextabhängige Faktoren, die den Testprozess in einer Organisation beeinflussen, sind u. a.:

- *Verwendetes Softwareentwicklungslebenszyklus-Modell und Projektmethoden*
- *Mögliche Teststufen und Testarten*
- *Produkt- und Projektrisiken*
- *Geschäftsbereich*
- *Betriebliche Beschränkungen, u. a.:*
  - *Budget und Ressourcen*
  - *Fristen*
  - *Komplexität*
  - *Vertragliche und regulatorische Anforderungen*
- *Richtlinien und Praktiken des Unternehmens*
- *Geforderte interne und externe Standards*

Die folgenden Abschnitte beschreiben allgemeine Aspekte von Testprozessen in Unternehmen in Bezug auf Folgendes:

- *Testaktivitäten und Aufgaben*
- *Testarbeitsergebnisse*

Verfolgbarkeit zwischen Testbasis und Testarbeitsergebnissen

Es ist sehr nützlich, wenn die Testbasis (für jede in Betracht gezogene Stufe oder Art des Testens) messbar definierte Überdeckungskriterien hat. Die Überdeckungskriterien können effektiv als Key-Performance-Indicator (KPI) genutzt werden, um die Aktivitäten zu lenken, die das Erreichen der Ziele des Softwaretests aufzeigen (siehe Abschnitt 1.1.1 Typische Ziele des Testens).

Für eine mobile Applikation kann die Testbasis zum Beispiel eine Liste von Anforderungen und eine Liste von zu unterstützenden mobilen Endgeräten enthalten. Jede Anforderung ist ein Element der Testbasis. Jedes zu unterstützende Endgerät ist ebenfalls ein Element der Testbasis. Die Überdeckungskriterien können erfordern, dass wenigstens ein Testfall für jedes Element der Testbasis entwickelt wird. Einmal ausgeführt, geben die Ergebnisse dieser Tests den Stakeholdern an, ob spezifizierte Anforderungen erfüllt sind und ob Fehlerwirkungen in unterstützten Endgeräten beobachtet wurden.

Die ISO-Norm (ISO/IEC/IEEE 29119-2) bietet weitere Informationen zu Testprozessen.

1.4.2 Testaktivitäten und Aufgaben (nicht prüfungsrelevant)

Ein Testprozess besteht aus den folgenden Hauptgruppen von Aktivitäten:

- *Testplanung*
- *Testüberwachung und -steuerung*
- *Testanalyse*
- *Testentwurf*
- *Testrealisierung*
- *Testdurchführung*
- *Testabschluss*

Jede Hauptgruppe von Aktivitäten besteht aus einzelnen Aktivitäten, die im Folgenden in den Unterabschnitten beschrieben werden. Jede einzelne Aktivität besteht ihrerseits aus mehreren Einzelaufgaben, die je nach Projekt oder Release variieren können.

Darüber hinaus werden diese Hauptgruppen von Aktivitäten oft iterativ realisiert, selbst wenn viele davon logisch aufeinander folgend erscheinen. Zum Beispiel beinhaltet agile Entwicklung kleine Iterationen im Softwareentwurf, Build und Test, die kontinuierlich durchgeführt werden und durch stetige Planung unterstützt werden. Also finden auch Testaktivitäten innerhalb dieses Softwareentwicklungsansatzes auf iterativer, kontinuierlicher Basis statt. Selbst bei sequenzieller Softwareentwicklung wird die angestrebte schrittweise logische Abfolge von Aktivitäten der Hauptgruppen sowohl Überlappung, Kombination, Parallelität oder Fortfall von einzelnen Aktivitäten der Hauptgruppen beinhalten, so dass eine Anpassung dieser Hauptaktivitäten innerhalb des System- und Projektkontexts grundsätzlich notwendig ist.

Testplanung

Testplanung beinhaltet Aktivitäten, die die Testziele definieren und den Ansatz für das Erreichen der Testziele innerhalb der Beschränkungen, die sich aus dem Umfeld ergeben, festlegen (z. B. das Spezifizieren geeigneter Testverfahren und Aufgaben sowie die Formulierung eines Testplans zur Einhaltung eines Termins). Testkonzepte können aufgrund von Feedback von Überwachungs- und Steuerungsaktivitäten überarbeitet werden. Testplanung wird in [ISTQB_FL_SYL – Abschnitt 5.2 Testplanung- und schätzung] näher erläutert.

Testüberwachung und -steuerung

Testüberwachung beinhaltet den andauernden Vergleich des tatsächlichen Fortschritts mit dem geplanten Fortschritt unter Nutzung jeglicher Überwachungsmetriken, die im Testkonzept definiert wurden. Teststeuerung beinhaltet das Ergreifen notwendiger Maßnahmen zur Erreichung der Ziele des Testkonzepts (das über den Zeitverlauf hinweg aktualisiert werden kann). Testüberwachung und -steuerung werden durch die Bewertung von Endekriterien unterstützt, die in einigen Softwareentwicklungslebenszyklusmodellen als Definition-of-Done bezeichnet werden (siehe ISTQB®-AT). Zum Beispiel kann die Bewertung von Endekriterien für die Testdurchführung als Teil einer vorgegebenen Teststufe Folgendes beinhalten:

- *Die Prüfung von Testergebnissen und -protokollen gegen spezifizierte Überdeckungskriterien*
- *Die Beurteilung des Grades der Komponenten- oder Systemqualität auf Grundlage der Testergebnisse und -protokolle*
- *Die Entscheidung, ob mehr Tests notwendig sind (z. B. ob Tests, die ursprünglich einen bestimmten Grad an Produktrisikoüberdeckung erreichen sollten und dies nicht erreicht haben, das Erstellen und Durchführen weiterer Tests erforderlich macht)*

Der Testfortschritt anhand des Plans wird den Stakeholdern in Testfortschrittsberichten mitgeteilt. Diese enthalten auch Abweichungen vom Plan sowie Informationen, die jegliche Entscheidung unterstützt, die Tests zu beenden.

Testüberwachung und -steuerung werden in [ISTQB_FL_SYL – Abschnitt 5.3 Testüberwachung und -steuerung] näher erläutert.

Testanalyse

Während der Testanalyse wird die Testbasis analysiert, um testbare Features zu identifizieren und die entsprechenden Testbedingungen zu definieren. Das heißt, die Testanalyse bestimmt mit messbaren Überdeckungskriterien, „was zu testen ist".

Die Testanalyse beinhaltet die folgenden Hauptaktivitäten:

- *Analyse der Testbasis, die für die in Betracht gezogene Teststufe geeignet ist, z. B.:*
  - *Anforderungsspezifikationen, wie Fachanforderungen, funktionale Anforderungen, Systemanforderungen, User-Stories [8], Epics [9], Anwendungsfall oder ähnliche Arbeitsergebnisse, die das gewünschte funktionale und nicht-funktionale Komponenten- oder Systemverhalten beschreiben*
  - *Entwurfs- und Realisierungsinformationen wie System- oder Softwarearchitekturdiagramme oder -dokumente, Entwurfsspezifikationen, Aufrufdiagramme, Modelldiagramme (z. B. UML- oder Entity-Relationship-Diagramme), Schnittstellenspezifikationen oder ähnliche Arbeitsergebnisse, die die Komponenten- oder Systemstruktur spezifizieren*
  - *Die Realisierung der Komponente oder des Systems selbst, darunter der Code, Datenbank-Metadaten und -abfragen sowie Schnittstellen*
  - *Risikoanalyseberichte, die funktionale, nicht-funktionale und strukturelle Aspekte der Komponente oder des Systems in Betracht ziehen*
- *Bewertung der Testbasis und der Testelemente zur Identifikation von unterschiedlichen Fehlerarten, wie z. B.:*
  - *Mehrdeutigkeiten*
  - *Auslassungen*
  - *Inkonsistenzen*
  - *Ungenauigkeiten*
  - *Widersprüche*
  - *Überflüssige Anweisungen*
- *Identifikation von Features und Feature-Sets, die getestet werden müssen.*
- *Definition und Priorisierung der Testbedingungen für jedes Feature auf Grundlage der Analyse der Testbasis und unter Berücksichtigung funktionaler, nicht-funktionaler und struktureller Merkmale, anderer fachlicher oder technischer Faktoren und des Risikograds*
- *Erfassung von bidirektionaler Verfolgbarkeit zwischen jedem Element der Testbasis und den zugehörigen Testbedingungen (siehe Abschnitte 1.4.3 Testarbeitsergebnisse und 1.4.4 Verfolgbarkeit zwischen der Testbasis und Testarbeitsergebnissen)*

Die Anwendung von Black-Box-, White-Box- und erfahrungsbasierten Testverfahren kann im Prozess der Testanalyse nützlich sein (siehe Kapitel 4), um die Wahrscheinlichkeit zu reduzieren, dass wichtige

---

8 Beschreiben die Anforderungen aus Sicht der Fachbereichsvertreter, aber auch der Entwickler und Tester und werden von diesen gemeinsam verfasst [informativer Bestandteil dieses Lehrplans, [ISTQB_FLAT_SYL].

9 Größere Ansammlungen von untereinander verbundenen Features oder eine Sammlung von Unter-Features, die zu einem einzigen, komplexen Feature zusammengeführt werden [informativer Bestandteil dieses Lehrplans, [ISTQB_FLAT_SYL].

Testbedingungen nicht berücksichtigt werden, und um präzisere und genauere Testbedingungen zu definieren.

In manchen Fällen liefert die Testanalyse Testbedingungen, die als Testziele in Test-Chartas genutzt werden können. Test-Chartas sind typische Arbeitsergebnisse in einigen Arten des erfahrungsbasierten Testens (siehe Abschnitt 4.4.2 Exploratives Testen). Wenn diese Testziele zur Testbasis verfolgbar sind, kann die Überdeckung, die während dieser erfahrungsbasierten Tests erzielt wird, gemessen werden.

Die Identifikation von Fehlerzuständen während der Testanalyse ist ein wichtiger potenzieller Nutzen, insbesondere dann, wenn kein anderer Reviewprozess genutzt wird und/oder der Testprozess eng mit dem Reviewprozess verbunden ist. Derartige Testanalyseaktivitäten verifizieren nicht nur, ob die Anforderungen konsistent, korrekt ausgedrückt und vollständig sind, sondern bewerten auch, ob die Anforderungen in angemessener Weise die Bedürfnisse der Kunden, Benutzer und anderer Stakeholder erfassen. Verfahren wie die verhaltensgetriebene Entwicklung („behavior driven development", BDD) und abnahmetestgetriebene Entwicklung („acceptance test driven development", ATDD), die die Generierung von Testbedingungen und Testfällen aus User-Stories und Abnahmekriterien vor dem Coding beinhalten. Diese Verfahren verifizieren, validieren und erkennen beispielsweise auch Fehlerzustände in den User-Stories und in den Abnahmekriterien [ISTQB_FLAT_SYL]

Testentwurf

Während des Testentwurfs werden die Testbedingungen in abstrakte Testfälle, Sets aus abstrakten Testfällen und andere Testmittel überführt. Die Testanalyse beantwortet also die Frage: „Was wird getestet?", während der Testentwurf die Frage beantwortet: „Wie wird getestet?".

Der Testentwurf beinhaltet die folgenden Hauptaktivitäten:

- *Entwurf und Priorisierung von Testfällen und Sets an Testfällen*
- *Identifizierung von notwendigen Testdaten zur Unterstützung der Testbedingungen und Testfälle*
- *Entwurf der Testumgebung und Identifizierung benötigter Infrastruktur und Werkzeuge*
- *Erfassung der bidirektionalen Verfolgbarkeit zwischen der Testbasis, den Testbedingungen und den Testfällen (siehe Abschnitt 1.4.4 Verfolgbarkeit zwischen der Testbasis und Testarbeitsergebnissen)*

Die Überführung von Testbedingungen in Testfälle und Sets an Testfällen während des Testentwurfs beinhaltet oft den Einsatz von Testverfahren (siehe Kapitel 4).

Wie bei der Testanalyse kann auch der Testentwurf zur Identifizierung ähnlicher Fehlerarten in der Testbasis führen. Wie bei der Testanalyse ist die Identifizierung von Fehlerzuständen während des Testentwurfs ein wichtiger potenzieller Nutzen.

Testrealisierung

Während der Testrealisierung werden die Testmittel, die für die Testdurchführung notwendig sind, erstellt und/oder vervollständigt, unter anderem werden die Testfälle in Testabläufen in eine Reihenfolge gebracht. Der Testentwurf beantwortet also die Frage: „Wie wird getestet?", während die Testrealisierung die Frage beantwortet: „Ist jetzt alles für die Durchführung der Tests bereit?".

Die Testrealisierung beinhaltet die folgenden Hauptaktivitäten:

- *Entwicklung und Priorisierung von Testabläufen und möglicherweise die Erstellung automatisierter Testskripte*
- *Erstellen von Testsuiten aus den Testabläufen und automatisierten Testskripten (falls es solche gibt)*

- *Anordnen der Testsuiten innerhalb eines Testausführungsplans in einer Art und Weise, die zu einer effizienten Testdurchführung führt, siehe [ISTQB_FL_SYL – Abschnitt 5.2.4 Testausführungsplanung].*
- *Aufbau der Testumgebung (unter anderem möglicherweise Testrahmen, Service-Virtualisierung, Simulatoren und andere Infrastrukturelemente) und Verifizierung, dass alles, was benötigt wird, korrekt aufgesetzt ist*
- *Vorbereitung von Testdaten und Sicherstellung, dass sie ordnungsgemäß in die Testumgebung geladen sind*
- *Verifizierung und Aktualisierung der bidirektionalen Verfolgbarkeit zwischen der Testbasis, den Testbedingungen, den Testfällen, den Testabläufen und den Testsuiten (siehe Abschnitt 1.4.4 Verfolgbarkeit zwischen der Testbasis und Testarbeitsergebnissen)*

Testentwurf und Testrealisierungsaufgaben werden oft kombiniert.

In explorativen Tests und anderen Arten von erfahrungsbasierten Tests können Testentwurf und Realisierung als Teil der Testdurchführung auftreten und dokumentiert werden. Exploratives Testen kann auf Test-Chartas basieren (erstellt als Teil der Testanalyse) und explorative Tests werden unverzüglich durchgeführt, sobald sie entworfen und realisiert sind (siehe Abschnitt 4.4.2 Exploratives Testen).

Testdurchführung

Während der Testdurchführung laufen Testsuiten in Übereinstimmung mit dem Testausführungsplan ab.

Die Testdurchführung beinhaltet die folgenden Hauptaktivitäten:

- *Aufzeichnung der IDs und Versionen des Testelements/der Testelemente oder des Testobjekts, des Testwerkzeugs/der Testwerkzeuge und Testmittel*
- *Durchführung der Tests entweder manuell oder durch Nutzung von Testausführungswerkzeugen*
- *Vergleich der Istergebnisse mit den erwarteten Ergebnissen*
- *Analyse der Anomalien zur Feststellung ihrer wahrscheinlichen Ursachen (z. B. können Fehlerwirkungen aufgrund von Fehlerzuständen im Code entstehen, aber falsch positive Ergebnisse können ebenfalls auftreten; siehe Abschnitt 1.2.3 Fehlhandlungen, Fehlerzustände und Fehlerwirkungen)*
- *Bericht über Fehlerzustände auf Grundlage der beobachteten Fehlerwirkungen siehe [ISTQB_FL_SYL – Abschnitt 5.6 Fehlermanagement]*
- *Aufzeichnung der Ergebnisse der Testdurchführung (z. B. bestanden, fehlgeschlagen, blockiert)*
- *Wiederholung von Testaktivitäten entweder als Ergebnis einer Aktion aufgrund einer Anomalie oder als Teil der geplanten Tests (z. B. Durchführung eines korrigierten Tests, Fehlernachtests und/oder Regressionstests)*
- *Verifizierung und Aktualisierung der bidirektionalen Verfolgbarkeit zwischen der Testbasis, den Testbedingungen, den Testfällen, den Testabläufen und den Testergebnissen*

Testabschluss

Die Aktivitäten am Abschluss des Tests sammeln Daten aus beendeten Testaktivitäten, um Erfahrungen, Testmittel und andere relevante Informationen zu konsolidieren. Testabschlussaktivitäten finden an Projektmeilensteinen statt, z. B. wenn ein Softwaresystem freigegeben wird, ein Testprojekt abgeschlossen (oder abgebrochen) wird, eine agile Projektiteration beendet ist, eine Teststufe abgeschlossen ist oder ein Wartungsrelease abgeschlossen worden ist.

Der Testabschluss beinhaltet die folgenden Hauptaktivitäten:

- *Prüfen, ob alle Fehlerberichte geschlossen sind, Eintragen von Change Requests oder Product-Backlog-Elementen für Fehlerzustände, die am Ende der Testdurchführung weiterhin nicht gelöst sind*
- *Erstellen eines Testabschlussberichts, der den Stakeholdern kommuniziert wird*
- *Finalisieren und Archivieren der Testumgebung, der Testdaten, der Testinfrastruktur und anderer Testmittel für spätere Wiederverwendung*
- *Übergabe der Testmittel an die Wartungsteams, andere Projektteams und/oder Stakeholder, die von deren Nutzung profitieren könnten*
- *Analyse der gewonnenen Erkenntnisse aus den abgeschlossenen Testaktivitäten, um notwendige Änderungen für zukünftige Iterationen, Produktreleases und Projekte zu bestimmen*
- *Nutzung der gesammelten Informationen zur Verbesserung der Testprozessreife*

1.4.3 Testarbeitsergebnisse (nicht prüfungsrelevant)

Im Verlauf des Testprozesses werden Testarbeitsergebnisse erstellt. So wie es wesentliche Unterschiede in der Art und Weise gibt, wie Unternehmen die Testprozesse implementieren, gibt es auch wesentliche Unterschiede in den Arten der Arbeitsergebnisse, die während dieses Prozesses erstellt werden, in der Art und Weise, wie diese Arbeitsergebnisse aufgebaut und verwaltet werden, und welche Namen für diese Arbeitsergebnisse verwendet werden. Dieser Lehrplan richtet sich nach dem zuvor beschriebenen Testprozess und den Arbeitsergebnissen, die in diesem Lehrplan und im ISTQB®-Glossar genannt sind. Die ISO-Norm (ISO/IEC/IEEE 29119-3) kann auch als Richtlinie für Testarbeitsergebnisse genutzt werden.

Viele der in diesem Abschnitt beschriebenen Testarbeitsergebnisse können durch die Nutzung von Testmanagementwerkzeugen und Fehlermanagementwerkzeugen erfasst und verwaltet werden siehe [ISTQB_FL_SYL – Abschnitt 6 Werkzeugunterstützung für das Testen].

Arbeitsergebnisse der Testplanung

Arbeitsergebnisse der Testplanung beinhalten üblicherweise einen oder mehrere Testkonzepte. Das Testkonzept enthält Informationen über die Testbasis, auf die sich die anderen Testarbeitsergebnisse via Verfolgbarkeitsinformationen beziehen werden (siehe unten und Abschnitt 1.4.4 Verfolgbarkeit zwischen der Testbasis und Testarbeitsergebnissen), sowie Endekriterien (oder Definition-of-Done), die während der Testüberwachung und -steuerung genutzt werden. Testkonzepte werden in [ISTQB_FL_SYL – Abschnitt 5.2 Testplanung und -schätzung] beschrieben.

Arbeitsergebnisse der Testüberwachung und -steuerung

Arbeitsergebnisse der Testüberwachung und -steuerung beinhalten üblicherweise verschiedene Arten von Testberichten, unter anderem kontinuierliche und/oder regelmäßig erstellte Testfortschrittsberichte und Testabschlussberichte, die an verschiedenen Abschlussmeilensteinen erstellt werden. Alle Testberichte sollten zielgruppenrelevante Details über den Testfortschritt zum Zeitpunkt des Berichts enthalten sowie die Testdurchführungsergebnisse zusammenfassen, sobald diese verfügbar sind.

Arbeitsergebnisse der Testüberwachung und -steuerung sollten auch Projektmanagementfragen ansprechen wie den abgeschlossenen Aufgaben, die Ressourcenzuordnung und -nutzung und den Aufwand.

Testüberwachung und -steuerung und die Arbeitsergebnisse, die während dieser Aktivitäten erstellt werden, werden in [ISTQB_FL_SYL – Abschnitt 5.3 Testüberwachung und -steuerung] näher erläutert.

Arbeitsergebnisse der Testanalyse

Arbeitsergebnisse der Testanalyse beinhalten definierte und priorisierte Testbedingungen, von denen jede idealerweise bidirektional verfolgbar zu den spezifischen Elementen der Testbasis ist, die sie

abdeckt. Für exploratives Testen kann die Testanalyse das Erstellen von Test-Chartas beinhalten. Die Testanalyse kann auch die Entdeckung und den Bericht von Fehlerzuständen in der Testbasis zur Folge haben.

Arbeitsergebnisse des Testentwurfs

Ergebnisse des Testentwurfs sind Testfälle und Sets von Testfällen, um die Testbedingungen abzudecken, die in der Testanalyse definiert wurden. Es ist oft hilfreich, abstrakte Testfälle ohne konkrete Werte für Eingabedaten und erwartete Ergebnisse zu entwerfen. Solche abstrakten Testfälle sind über mehrere Testzyklen mit unterschiedlichen konkreten Daten wiederverwendbar und dokumentieren dennoch in adäquater Weise den Umfang des Testfalls. Idealerweise ist jeder Testfall bidirektional verfolgbar zu den Testbedingungen, die er abdeckt.

Der Testentwurf resultiert auch

- *im Entwurf und/oder der Identifizierung der notwendigen Testdaten*
- *dem Entwurf der Testumgebung*
- *der Identifizierung der Infrastruktur und Werkzeuge.*

Allerdings schwankt der Umfang, in dem diese Ergebnisse dokumentiert werden, stark.

Arbeitsergebnisse der Testrealisierung

Arbeitsergebnisse der Testrealisierung beinhalten:

- *Testabläufe und die Aneinanderreihung dieser Testabläufe*
- *Testsuiten*
- *Einen Testausführungsplan*

Sobald die Testrealisierung abgeschlossen ist, kann idealerweise das Erreichen von Überdeckungskriterien, die im Testkonzept bestimmt wurden, über die bidirektionale Verfolgbarkeit zwischen Testabläufen und spezifischen Elementen der Testbasis durch die Testfälle und Testbedingungen gezeigt werden.

In manchen Fällen beinhaltet die Testrealisierung das Erstellen von Arbeitsergebnissen, die Werkzeuge nutzen oder von ihnen genutzt werden, wie Service-Virtualisierung und automatisierte Testskripte.

Die Testrealisierung kann auch in der Erstellung und Verifizierung von Testdaten und der Testumgebung resultieren. Die Vollständigkeit der Dokumentation der Verifizierungsergebnisse der Daten und/oder der Umgebung kann stark schwanken.

Die Testdaten dienen dazu, den Eingabewerten und erwarteten Ergebnissen von Testfällen konkrete Werte zuzuweisen. Solche konkreten Werte und explizite Anweisungen, wie diese konkreten Werte genutzt werden sollen, verwandeln abstrakte Testfälle in durchführbare konkrete Testfälle. Die gleichen abstrakten Testfälle können bei Durchführung in anderen Releases des Testobjekts andere Testdaten nutzen. Die konkreten erwarteten Ergebnisse, die mit den konkreten Testdaten verbunden sind, werden durch die Nutzung eines Testorakels identifiziert.

Im explorativen Testen können einige Arbeitsergebnisse des Testentwurfs und der Testrealisierung während der Testdurchführung erstellt werden. Der Umfang, mit dem explorative Tests (und ihre Verfolgbarkeit zu spezifischen Elementen der Testbasis) dokumentiert werden, kann allerdings stark schwanken.

Testbedingungen, die in der Testanalyse definiert sind, können in der Testrealisierung weiter verfeinert werden.

Arbeitsergebnisse der Testdurchführung

Arbeitsergebnisse der Testdurchführung beinhalten:

- *Dokumentation des Status individueller Testfälle oder Testabläufe (z. B. ausführbar, bestanden, fehlgeschlagen, blockiert, geplant ausgelassen usw.)*

- *Fehlerbericht, siehe [ISTQB_FL_SYL – Abschnitt 5.6 Fehlermanagement]*
- *Dokumentation darüber, welches Testelement(e), Testwerkzeug(e) und Testmittel für die Tests benutzt wurden*

Sobald die Testdurchführung vollständig ist, kann idealerweise der Status jedes Elements der Testbasis bestimmt und via bidirektionaler Verfolgbarkeit zu den zugehörigen Testabläufen berichtet werden. Zum Beispiel können wir sagen, welche Anforderungen alle geplanten Tests bestanden haben, welche Anforderungen aufgrund fehlgeschlagener Tests nicht erfolgreich verifiziert sind und/oder Fehlerzustände aufweisen und welche Anforderungen noch auf die Durchführung geplanter Tests warten. Dies erlaubt die Verifizierung, dass Überdeckungskriterien eingehalten wurden, und ermöglicht den Bericht von Testergebnissen in einer Art und Weise, dass sie für die Stakeholder verständlich sind.

Arbeitsergebnisse des Testabschlusses

Testabschlussarbeitsergebnisse beinhalten Testabschlussberichte, offene Punkte zur Verbesserung in nachfolgenden Projekten oder Iterationen, Change Requests oder Product-Backlog-Elemente und die finalisierten Testmittel.

1.4.4 Verfolgbarkeit zwischen der Testbasis und Testarbeitsergebnissen (nicht prüfungsrelevant)

Wie im letzten Abschnitt erwähnt, können Testarbeitsergebnisse und die Namen dieser Arbeitsergebnisse stark variieren. Unabhängig von diesen Variationen ist es wichtig für die Implementierung einer effektiven Testüberwachung und -steuerung, die Verfolgbarkeit durch den gesamten Testprozess zwischen allen Elementen der Testbasis und den verschiedenen Testarbeitsergebnissen, die diesem Element wie zuvor beschrieben zugeordnet sind, zu schaffen und zu erhalten. Über die Bewertung der Testüberdeckung hinaus unterstützt gute Verfolgbarkeit folgende Punkte:

- *Die Auswirkungsanalyse von Änderungen*
- *Testen nachvollziehbarer zu machen*
- *IT-Governance-Kriterien zu erfüllen*
- *Die Verständlichkeit von Testfortschrittsberichten und Testabschlussberichten zu verbessern, um den Status der Elemente der Testbasis einzubeziehen (z. B. Anforderungen, die die Tests bestanden haben, Anforderungen, die in Tests fehlgeschlagen sind, und Anforderungen, für die Tests noch ausstehen)*
- *Bericht über die technischen Aspekte des Testens an die Stakeholder in einer Art und Weise, die sie verstehen können*
- *Bereitstellung von Informationen zur Beurteilung von Produktqualität, Prozessfähigkeit und Projektfortschritt gegenüber Geschäftszielen*

Einige Testmanagementwerkzeuge liefern Modelle für Testarbeitsergebnisse, die einen Teil oder alle Testarbeitsergebnisse, die in diesem Abschnitt erwähnt sind, abdecken. Einige Unternehmen bauen ihre eigenen Managementsysteme, um die Arbeitsergebnisse zu verwalten und die Informationsverfolgbarkeit zu liefern, die sie benötigen.

2. Testen im Softwareentwicklungslebenszyklus – 70 Minuten

Schlüsselbegriffe

Änderungsbezogenes Testen, Auswirkungsanalyse, Fehlernachtest, funktionaler Test, Integrationstest, Komponentenintegrationstest, Komponententest, nicht-funktionaler Test, Regressionstest, Testart, Testbasis, Testfall, Testobjekt, Teststufe, Testumgebung, Testziel, Wartungstest, White-Box-Test

Lernziele für das Testen im Softwareentwicklungslebenszyklus

2.1 Teststufen (nicht prüfungsrelevant)

FL-2.2.1 (K2) Die unterschiedlichen Teststufen unter den Aspekten der Testziele, Testbasis, Testobjekte, typischen Fehlerzustände und Fehlerwirkungen sowie der Testvorgehensweise und Verantwortlichkeiten vergleichen können

2.2 Testarten

FL-2.3.1 (K2) Funktionale, nicht-funktionale und White-Box-Tests vergleichen können

FL-2.3.2 (K2) Den Zweck von Fehlernachtests und Regressionstests vergleichen können

2.3 Wartungstest

FL-2.4.1 (K2) Auslöser für Wartungstests zusammenfassen können

FL-2.4.2 (K2) Den Einsatz der Auswirkungsanalyse im Wartungstest beschreiben können

2.1 *Teststufen (nicht prüfungsrelevant)*

Teststufen sind Gruppen von Testaktivitäten, die gemeinsam organisiert und verwaltet werden. Jede Teststufe ist eine Instanz des Testprozesses, der aus Aktivitäten besteht, die in Abschnitt 1.4 Testprozess beschrieben wurden. Diese Testaktivitäten beziehen sich auf die Software einer festgelegten Entwicklungsstufe von einzelnen Einheiten oder Komponenten bis hin zu vollständigen Systemen oder, falls zutreffend, von Systemen von Systemen. Die Teststufen stehen mit anderen Aktivitäten innerhalb des Softwareentwicklungslebenszyklus in Verbindung. In diesem Lehrplan werden die folgenden Teststufen verwendet:

- *Komponententest*
- *Integrationstest*
- *Systemtest*
- *Abnahmetest*

Teststufen sind durch die folgenden Eigenschaften gekennzeichnet:

- *Spezifische Ziele*
- *Testbasis, um mit Bezug darauf Testfälle abzuleiten*
- *Testobjekt (d.h., was wird getestet)*
- *Typische Fehlerzustände und Fehlerwirkungen*
- *Spezifische Ansätze und Verantwortlichkeiten*

Für jede Teststufe ist eine passende Testumgebung erforderlich. Für den Abnahmetest ist beispielsweise eine produktionsähnliche Umgebung ideal, während die Entwickler im Komponententest üblicherweise ihre eigene Entwicklungsumgebung nutzen.

2.1.1 Komponententest (nicht prüfungsrelevant)

Ziele des Komponententests

Der Komponententest (auch Unit- oder Modultest genannt) konzentriert sich auf Komponenten, die einzeln testbar sind. Die Ziele des Komponententests beinhalten:

- *Risikoreduktion*
- *Verifizierung, ob die funktionalen und nicht-funktionalen Verhaltensweisen der Komponente dem Entwurf und der Spezifikation entsprechen*
- *Schaffen von Vertrauen in die Qualität der Komponente*
- *Finden von Fehlerzuständen in der Komponente*
- *Verhindern, dass Fehlerzustände an höhere Teststufen weitergegeben werden*

In manchen Fällen, insbesondere in inkrementellen und iterativen Entwicklungsmodellen (z. B. agilen Entwicklungsmodellen), in denen Codeänderungen stetig erfolgen, spielen automatisierte Komponentenregressionstests eine Schlüsselrolle. Diese sollen Vertrauen schaffen, dass Änderungen bestehende Komponenten nicht beschädigt haben.

Der Komponententest wird häufig isoliert vom Rest des Systems vorgenommen. In Abhängigkeit vom Softwareentwicklungslebenszyklus-Modell und vom System erfordert dies eventuell Mock-Objekte, Service-Virtualisierung, Rahmen, Platzhalter und Treiber. Der Komponententest kann die Funktionalität (z. B. die Korrektheit von Berechnungen), nicht-funktionale Eigenschaften (z. B. Suche nach Speicherengpässen) und strukturelle Merkmale (z. B. Entscheidungstests) abdecken.

Testbasis

Beispiele für Arbeitsergebnisse, die als Testbasis für Komponententests genutzt werden können, sind u. a.:

- *Feinentwurf*
- *Code*
- *Datenmodelle*
- *Komponentenspezifikationen*

Testobjekte

Gängige Testobjekte für Komponententests sind u. a.:

- *Komponenten, Units oder Module*
- *Code und Datenstrukturen*
- *Klassen*
- *Datenbankmodule*

Gängige Fehlerzustände und Fehlerwirkungen

Beispiele für gängige Fehlerzustände und Fehlerwirkungen für Komponententests sind u. a.:

- *Nicht korrekte Funktionalität (z. B. nicht wie in den Entwurfsspezifikationen beschrieben)*
- *Datenflussprobleme*
- *Nicht korrekter Code und nicht korrekte Logik*

Fehlerzustände werden in der Regel behoben, sobald sie gefunden werden, oftmals ohne formales Fehlermanagement. Wenn Entwickler allerdings Fehlerzustände berichten, liefert dies wichtige Informationen für die Grundursachenanalyse und die Prozessverbesserung.

Spezifische Ansätze und Verantwortlichkeiten

Komponententests werden üblicherweise von dem Entwickler durchgeführt, der den Code geschrieben hat. Zumindest aber erfordert dies den Zugang zum Code, der getestet werden soll. Entwickler können zwischen der Entwicklung einer Komponente und dem Finden und Beheben von Fehlerzuständen wechseln. Entwickler schreiben oft Tests und führen diese aus, nachdem sie den Code für eine Komponente entwickelt haben. Insbesondere in der agilen Entwicklung kann es jedoch auch sein, dass das Schreiben von automatisierten Komponententestfällen dem Schreiben von Anwendungscode vorangeht.

Nehmen wir zum Beispiel die testgetriebene Entwicklung (test driven development, TDD). Die testgetriebene Entwicklung ist hoch iterativ und basiert auf Zyklen zur Entwicklung automatisierter Testfälle, dann erfolgt die Entwicklung und Integration kleiner Teile von Code, gefolgt von der Durchführung von Komponententests, der Korrektur möglicher Probleme und der Restrukturierung (Refactoring) von Code. Dieser Prozess setzt sich fort, bis die Komponente vollständig erstellt und alle Komponententests erfolgreich abgeschlossen sind. Die testgetriebene Entwicklung ist ein Beispiel für den Ansatz „test first“. Obwohl die testgetriebene Entwicklung ihren Ursprung im eXtreme Programming (XP) hat, hat sie sich auch in andere Formen der agilen und in sequenzielle Lebenszyklen verbreitet (siehe ISTQB-AT).

2.1.2 Integrationstest (nicht prüfungsrelevant)

Ziele von Integrationstests

Integrationstests konzentrieren sich auf die Interaktion zwischen Komponenten oder Systemen. Ziele der Integrationstests sind u. a.:

- *Risikoreduktion*
- *Verifizierung, ob die funktionalen und nicht-funktionalen Verhaltensweisen der Schnittstellen dem Entwurf und der Spezifikation entsprechen*
- *Vertrauen schaffen in die Qualität der Schnittstellen*
- *Fehlerzustände finden (die in den Schnittstellen selbst oder innerhalb der Komponenten oder des Systems liegen können)*
- *Verhindern, dass Fehlerzustände an höhere Teststufen weitergegeben werden*

Wie beim Komponententest liefern automatisierte Integrationsregressionstests in manchen Fällen das Vertrauen, dass Änderungen bestehende Schnittstellen, Komponenten oder Systeme nicht beschädigt haben.

In diesem Lehrplan wird der Test von zwei Integrationsstufen beschrieben, der auf Testobjekte unterschiedlicher Größe ausgeübt werden kann:

- *Komponentenintegrationstests konzentrieren sich auf die Interaktionen und die Schnittstellen zwischen integrierten Komponenten. Komponentenintegrationstests werden nach Komponententests durchgeführt und sind generell automatisiert. In der iterativen und inkrementellen Entwicklung sind Integrationstests in der Regel Teil des kontinuierlichen Integrationsprozesses (continuous integration).*
- *Systemintegrationstests konzentrieren sich auf die Interaktionen und Schnittstellen zwischen Systemen, Paketen und Microservices. Systemintegrationstests können auch Interaktionen und Schnittstellen abdecken, die von externen Organisationen bereitgestellt werden (z. B. Webservices). In diesem Fall hat die entwickelnde Organisation keinen Einfluss auf die externen Schnittstellen. Dies kann zu verschiedenen Herausforderungen im Testen führen (z. B. Sicherstellen, dass testblockierende Fehlerzustände im Code der externen Organisation behoben sind, Bereitstellen von Testumgebungen usw.). Der Systemintegrationstest kann nach den Systemtests oder parallel zu andauernden Systemtestaktivitäten stattfinden (sowohl in der sequenziellen Entwicklung als auch in der iterativen und inkrementellen Entwicklung).*

Testbasis

Beispiele für Arbeitsergebnisse, die als Testbasis für Integrationstests genutzt werden können, sind u. a.:

- *Software- und Systementwurf*
- *Sequenzdiagramme*
- *Spezifikationen von Schnittstellen und Kommunikationsprotokollen*
- *Anwendungsfälle*
- *Architektur auf Komponenten- oder Systemebene*
- *Workflows*
- *Externe Schnittstellendefinitionen*

Testobjekte

Gängige Testobjekte für Integrationstests sind u. a.:

- *Subsysteme*
- *Datenbanken*
- *Infrastruktur*
- *Schnittstellen*
- *APIs*
- *Microservices*

Typische Fehlerzustände und Fehlerwirkungen

Beispiele für typische Fehlerzustände und Fehlerwirkungen für Komponentenintegrationstests sind u. a.:

- *Falsche Daten, fehlende Daten oder falsche Datenverschlüsselung*
- *Falsche Reihenfolge oder fehlerhafte zeitliche Abfolge von Schnittstellenaufrufen*
- *Falsch angepasste Schnittstellen*
- *Fehlerwirkungen in der Kommunikation zwischen Komponenten*
- *Nicht behandelte oder nicht ordnungsgemäß behandelte Fehlerwirkungen in der Kommunikation zwischen den Komponenten*
- *Nicht korrekte Annahmen über die Bedeutung, Einheiten oder Grenzen der Daten, die zwischen den Komponenten hin- und hergereicht werden*

Beispiele typischer Fehlerzustände und Fehlerwirkungen für Systemintegrationstests sind u. a.:

- *Inkonsistente Nachrichtenstrukturen zwischen den Systemen*
- *Falsche Daten, fehlende Daten oder falsche Datenverschlüsselung*
- *Falsch angepasste Schnittstellen*
- *Fehlerwirkungen in der Kommunikation zwischen den Systemen*
- *Nicht behandelte oder nicht ordnungsgemäß behandelte Fehlerwirkungen in der Kommunikation zwischen den Systemen*
- *Falsche Annahmen über die Bedeutung, Einheiten oder Grenzen der Daten, die zwischen den Systemen hin- und hergereicht werden*
- *Fehlende Konformität mit erforderlichen Richtlinien zur IT-Sicherheit*

Spezifische Ansätze und Verantwortlichkeiten

Komponentenintegrationstests und Systemintegrationstests sollten sich auf die Integration selbst konzentrieren. Zum Beispiel, wenn das Modul A mit dem Modul B integriert wird, sollten sich die Tests auf die Kommunikation zwischen diesen Modulen konzentrieren und nicht auf die Funktionalität der einzelnen Module, da diese in den Komponententests abgedeckt sein sollte. Wenn System X mit System Y integriert wird, sollten sich die Tests auf die Kommunikation zwischen den Systemen konzentrieren, nicht auf die Funktionalität der einzelnen Systeme, da diese in den Systemtests abgedeckt sein sollte. Funktionale, nicht-funktionale und strukturelle Testarten sind geeignet.

Komponentenintegrationstests liegen häufig in der Verantwortung der Entwickler. Systemintegrationstests liegen im Allgemeinen in der Verantwortung der Tester. Idealerweise sollten Tester, die Systemintegrationstests durchführen, die Systemarchitektur verstehen und die Integrationsplanung beeinflusst haben.

Wenn Integrationstests und die Integrationsstrategie geplant werden, bevor Komponenten oder Systeme entwickelt werden, können diese Komponenten oder Systeme in der Reihenfolge entwickelt werden, die für das effizienteste Testen erforderlich ist. Systematische Integrationsstrategien können auf der Systemarchitektur (z. B. Top-down und Bottom-up), auf funktionellen Aufgaben, auf der Reihenfolge der Transaktionsverarbeitung oder auf anderen Aspekten des Systems oder der Komponenten basieren. Um die Isolation von Fehlerzuständen zu vereinfachen und Fehlerzustände früh zu erkennen, sollte die Integration normalerweise inkrementell erfolgen (d.h. nur eine kleine Anzahl von zusätzlichen Komponenten oder Systemen zur gleichen Zeit) statt in einer Art „Big Bang" (d.h. Integration aller Komponenten oder Systeme in einem einzigen Schritt). Eine Risikoanalyse der komplexesten Schnittstellen kann dabei helfen, die Integrationstests zielgerichtet einzusetzen.

Je größer der Umfang der Integration, desto schwieriger wird es, die Fehlerzustände in einer spezifischen Komponente oder einem spezifischen System zu isolieren. Dies führt zu einem höheren Risiko und einem größeren Zeitaufwand für die Fehlerbehebung. Das ist ein Grund dafür, dass kontinuierliche Integration, bei der Software auf Komponentenbasis integriert wird (d.h. funktionale Integration), zur gängigen Vorgehensweise geworden ist. Eine derartige kontinuierliche Integration (continuous integration) beinhaltet häufig automatisierte Regressionstests, idealerweise auf mehreren Teststufen.

2.1.3 Systemtest (nicht prüfungsrelevant)

Ziele des Systemtests

Systemtests konzentrieren sich auf das Verhalten und die Fähigkeiten des Systems oder Produkts. Dies geschieht oft unter Berücksichtigung der End-to-End-Aufgaben, die das System leisten kann, und der nicht-funktionalen Verhaltensweisen, die bei der Verarbeitung dieser Aufgaben zu Tage treten. Ziele des Systemtests sind u. a.:

- Risikoreduktion
- Verifizierung, ob die funktionalen und nicht-funktionalen Verhaltensweisen des Systems dem Entwurf und den Spezifikationen entsprechen
- Validierung, dass das System vollständig ist und wie erwartet funktionieren wird
- Vertrauen in die Qualität des Systems als Ganzes schaffen
- Finden von Fehlerzuständen
- Verhindern, dass Fehlerzustände an höhere Teststufen oder in die Produktion weitergegeben werden

Für einige Systeme kann auch die Verifizierung der Datenqualität ein Ziel sein. Wie beim Komponententest und beim Integrationstest bieten in einigen Fällen automatisierte Systemregressionstests die Gewissheit, dass Änderungen nicht die bestehenden Features oder End-to-End-Fähigkeiten beeinträchtigt haben. Systemtests liefern häufig Informationen, die von Stakeholdern für Freigabeentscheidungen genutzt werden. Systemtests können auch zur Erfüllung rechtlicher oder regulatorischer Anforderungen oder Standards notwendig sein.

Die Testumgebung sollte idealerweise der finalen Ziel- oder der Produktivumgebung entsprechen.

Testbasis

Beispiele für Arbeitsergebnisse, die als Testbasis für Systemtests genutzt werden können, sind u. a.:

- *System- und Softwareanforderungsspezifikationen (funktional und nicht-funktional)*
- *Risikoanalyseberichte*
- *Anwendungsfälle*
- *Epics und User-Stories*

- *Modelle des Systemverhaltens*
- *Zustandsdiagramme*
- *System- und Benutzeranleitungen*

Testobjekte

Gängige Testobjekte für Systemtests sind u. a.:

- *Anwendungen*
- *Hardware/Softwaresysteme*
- *Betriebssysteme*
- *Systeme unter Test (SUT)*
- *Systemkonfiguration und Konfigurationsdaten*

Typische Fehlerzustände und Fehlerwirkungen

Beispiele für typische Fehlerzustände und Fehlerwirkungen für Systemtests sind u. a.:

- *Falsche Berechnungen*
- *Falsche oder unerwartete funktionale oder nicht-funktionale Systemverhaltensweisen*
- *Falsche Kontroll- und/oder Datenflüsse innerhalb des Systems*
- *Versagen bei der korrekten oder vollständigen Ausführung von funktionalen End-to-End-Aufgaben*
- *Versagen des Systems bei der ordnungsgemäßen Arbeit in der/den Systemumgebung/en*
- *Das System funktioniert nicht wie in den System- oder Benutzeranleitungen beschrieben*

Spezifische Ansätze und Verantwortlichkeiten

Systemtests sollten sich auf das allgemeine End-to-End-Verhalten des Systems als Ganzes konzentrieren, sowohl in funktionaler als auch in nicht-funktionaler Hinsicht. Systemtests sollten die Verfahren nutzen, die am besten für die zu testenden Aspekte des Systems geeignet sind (siehe Kapitel 4). Beispielsweise kann eine Entscheidungstabelle erstellt werden, um zu verifizieren, ob ein funktionales Verhalten den Beschreibungen in den Geschäftsregeln entspricht.

Systemtests werden in der Regel von unabhängigen Testern durchgeführt, die sich stark auf Spezifikationen stützen. Fehlerzustände in Spezifikationen (z. B. fehlende User-Stories, falsch benannte Fachanforderungen usw.) können zu einem Verständnisproblem oder Unstimmigkeiten über das erwartete Systemverhalten führen. Derartige Situationen können zu „falsch positiven“ und „falsch negativen“ Ergebnissen führen. Dies verschwendet Zeit und reduziert entsprechend die Effektivität der Fehleridentifikation. Das frühe Einbeziehen von Testern in User-Story-Verfeinerungen (Refinements) oder statische Testaktivitäten, wie Reviews, helfen dabei, das Auftreten solcher Situationen zu reduzieren.

2.1.4 Abnahmetest (nicht prüfungsrelevant)

Ziele des Abnahmetests

Der Abnahmetest konzentriert sich wie der Systemtest typischerweise auf das Verhalten und die Fähigkeiten eines gesamten Systems oder Produkts. Ziele des Abnahmetests sind u. a.:

- *Vertrauen in die Qualität des Systems als Ganzen aufbauen*
- *Validieren, ob das System vollständig ist und wie erwartet funktionieren wird*

- *Verifizieren, ob funktionale und nicht-funktionale Verhaltensweisen des Systems den Spezifikationen entsprechen*

Der Abnahmetest kann Informationen bereitstellen, mit denen sich die Bereitschaft des Systems für den Einsatz und die Nutzung durch den Kunden (Endanwender) beurteilen lässt. Im Abnahmetest können Fehlerzustände gefunden werden, aber das Finden von Fehlerzuständen ist häufig nicht das Ziel und das Finden einer großen Anzahl von Fehlerzuständen im Abnahmetest wird in manchen Fällen als großes Projektrisiko angesehen. Der Abnahmetest kann auch notwendig sein, um rechtliche oder regulatorische Anforderungen oder Standards zu erfüllen.

Häufige Ausprägungen von Abnahmetests sind u. a.:

- *Benutzerabnahmetest*
- *Betrieblicher Abnahmetest*
- *Vertraglicher und regulatorischer Abnahmetest*
- *Alpha- und Beta-Test*

Jeder dieser Ausprägungen wird in den folgenden vier Unterabschnitten genauer beschrieben.

Benutzerabnahmetest (User Acceptance Testing (UAT))

Der Abnahmetest des Systems ist üblicherweise darauf konzentriert, die Bereitschaft des Systems zur Nutzung durch Benutzer in einer realen oder simulierten Betriebsumgebung zu validieren. Das Hauptziel ist es, Vertrauen darin zu schaffen, dass die Benutzer das System so nutzen können, dass es ihre Bedürfnisse und Anforderungen erfüllt und ihre Geschäftsprozesse mit einem Minimum an Schwierigkeiten, Kosten und Risiken ausführt.

Betrieblicher Abnahmetest (Operational Acceptance Testing (OAT))

Der Abnahmetests des Systems durch Mitarbeiter des Betriebs oder Systemadministratoren wird üblicherweise in einer (simulierten) Produktivumgebung durchgeführt. Der Fokus des Tests liegt dabei auf betrieblichen Aspekten, u. a.:

- *Testen von Backups und Wiederherstellungen*
- *Installieren, Deinstallieren und Aktualisieren*
- *Notfallwiederherstellung (Disaster-Recovery)*
- *Benutzerverwaltung*
- *Wartungsaufgaben*
- *Datenlade- und Migrationsaufgaben*
- *Prüfen von IT-Sicherheitsschwachstellen*
- *Performanztest*

Das Hauptziel von betrieblichen Abnahmetests ist es, Vertrauen darin aufzubauen, dass die Betreiber oder Systemadministratoren das System in der betrieblichen Umgebung ordnungsgemäß für die Benutzer funktionsfähig halten können, selbst unter außergewöhnlichen oder schwierigen Bedingungen.

Vertraglicher oder regulatorischer Abnahmetest

Der vertragliche Abnahmetest wird aufgrund von vertraglichen Abnahmekriterien für die Herstellung von kundenspezifisch entwickelter Software durchgeführt. Abnahmekriterien sollten zu dem Zeitpunkt definiert werden, an dem sich die Vertragsparteien auf den Vertrag einigen. Vertragliche Abnahmetests werden häufig durch Benutzer oder unabhängige Tester durchgeführt.

Regulatorische Abnahmetests werden gegen Regularien durchgeführt, die eingehalten werden müssen, beispielsweise staatliche, gesetzliche oder Vorschriften zur funktionalen Sicherheit. Regulatorische

Abnahmetests werden häufig von Benutzern oder unabhängigen Testern durchgeführt. Manchmal werden die Ergebnisse von Aufsichtsbehörden bestätigt oder auditiert.

Das Hauptziel von vertraglichen oder regulatorischen Abnahmetests ist es, Vertrauen darin aufzubauen, dass die vertragliche oder regulatorische Konformität gegeben ist.

Alpha- und Beta-Tests

Alpha- und Beta-Tests werden üblicherweise von Entwicklern von kommerzieller Standardsoftware genutzt, die Rückmeldungen von ihren potenziellen oder bestehenden Benutzern, Kunden und/oder Betreibern erhalten wollen, bevor die Software auf den Markt kommt. Alpha-Tests werden auf Seiten des entwickelnden Unternehmens vorgenommen, nicht vom Entwicklungsteam, sondern von potenziellen oder bestehenden Kunden und/oder Betreibern oder von einem unabhängigen Testteam. Beta-Tests werden von potenziellen oder bestehenden Kunden und/oder Betreibern an ihren eigenen Standorten durchgeführt. Beta-Tests können nach Alpha-Tests erfolgen oder auch ohne, dass ein vorausgehender Alpha-Test stattfand.

Ein Ziel des Alpha- und Beta-Testens ist es, Vertrauen bei potenziellen oder bestehenden Kunden und/oder Betreibern darüber zu schaffen, dass sie das System unter normalen alltäglichen Umständen in der Produktivumgebung(en) nutzen können, um ihre Ziele bequem und mit geringsten Kosten und Risiken zu erreichen. Ein weiteres Ziel kann das Finden von Fehlerzuständen sein, die sich auf die Bedingungen und Umgebungen beziehen, in denen das System genutzt werden wird, insbesondere dann, wenn diese Bedingungen und Umgebungen durch das Entwicklungsteam nur schwer nachzubilden sind.

Testbasis

Beispiele für Arbeitsergebnisse, die als Testbasis für jegliche Form von Abnahmetests verwendet werden können, sind u. a.:

- *Geschäftsprozesse*
- *Benutzer- oder Fachanforderungen*
- *Vorschriften, rechtliche Verträge und Standards*
- *Anwendungsfälle und/oder User Stories*
- *Systemanforderungen*
- *System- oder Benutzerdokumentation*
- *Installationsverfahren*
- *Risikoanalyseberichte*

Zusätzlich können eines oder mehrere der folgenden Arbeitsergebnisse als Testbasis für die Ableitung von Testfällen für betriebliche Abnahmetests genutzt werden:

- *Sicherungs- und Wiederherstellungsverfahren*
- *Disaster-Recovery-Verfahren*
- *Nicht-funktionale Anforderungen*
- *Betriebsdokumentation*
- *Bereitstellungs- und Installationsanweisungen*
- *Performanzziele*
- *Datenbankpakete*
- *Standards oder Vorschriften bzgl. IT-Sicherheit*

Typische Testobjekte

Typische Testobjekte für jegliche Form von Abnahmetests sind u. a.:

- *System unter Test (SUT)*
- *Systemkonfigurationen und Konfigurationsdaten*
- *Geschäftsprozesse des vollintegrierten Systems*
- *Wiederherstellungssysteme und Hot Sites (für Tests zur Business-Continuity (Betriebskontinuität) und Notfallwiederherstellung)*
- *Betriebs- und Wartungsprozesse*
- *Formulare*
- *Berichte*
- *Bestehende und konvertierte Produktionsdaten*

Typische Fehlerzustände und Fehlerwirkungen

Beispiele für typische Fehlerzustände für jegliche Form von Abnahmetests sind u. a.:

- *Systemworkflows erfüllen nicht die Fach- oder Benutzeranforderungen*
- *Geschäftsregeln wurden nicht korrekt umgesetzt*
- *Das System erfüllt nicht die vertraglichen oder regulatorischen Anforderungen*
- *Nicht-funktionale Fehlerwirkungen wie IT-Sicherheitsschwachstellen, nicht angemessene Performanz unter hoher Last oder nicht ordnungsgemäßer Betrieb auf einer unterstützten Plattform*

Spezifische Ansätze und Verantwortlichkeiten

Der Abnahmetest liegt häufig in der Verantwortung der Kunden, Fachanwender, Product Owner oder Betreiber eines Systems. Andere Stakeholder können ebenfalls mit einbezogen werden.

Der Abnahmetest wird oft als die letzte Stufe in einem sequenziellen Entwicklungslebenszyklus verstanden, er kann aber auch zu anderen Zeitpunkten stattfinden, z. B.:

- *Der Abnahmetests eines kommerziellen Standardsoftwareprodukts kann zum Zeitpunkt der Installation oder Integration stattfinden*
- *Der Abnahmetest einer neuen funktionalen Verbesserung kann vor dem Systemtest stattfinden*

In der iterativen Entwicklung können Projektteams verschiedene Formen der Abnahmetests während und am Ende einer Iteration vornehmen, wie die, die sich auf die Verifizierung eines neuen Features gegenüber seinen Abnahmekriterien konzentrieren, oder die, die sich darauf konzentrieren, zu validieren, dass ein neues Feature die Bedürfnisse der Benutzer erfüllt. Darüber hinaus können Alpha- und Beta-Tests entweder am Ende jeder Iteration, nach Abschluss jeder Iteration oder nach einer Serie von Iterationen stattfinden. Benutzerabnahmetests, betriebliche Abnahmetests, regulatorische Abnahmetests und vertragliche Abnahmetests können ebenfalls entweder zum Abschluss jeder Iteration, nach Abschluss jeder Iteration oder nach einer Serie von Iterationen stattfinden.

2.2 Testarten

Eine Testart ist eine Gruppe von Testaktivitäten, die darauf abzielt, spezifische Merkmale eines Softwaresystems oder eines Teils eines Systems auf der Grundlage spezifischer Testziele zu testen. Solche Ziele können u. a. sein:

- Bewertung funktionaler Qualitätsmerkmale wie Vollständigkeit, Korrektheit und Angemessenheit
- Bewertung nicht-funktionaler Qualitätsmerkmale wie Zuverlässigkeit, Performanz, IT-Sicherheit, Kompatibilität und Gebrauchstauglichkeit (Usability)
- Bewertung, ob die Struktur oder Architektur der Komponente oder des Systems korrekt und vollständig ist und den Spezifikationen entspricht
- Bewertung der Auswirkungen von Änderungen wie die Bestätigung, dass Fehlerzustände behoben wurden (Fehlernachtests), und die Suche nach nicht gewünschten Änderungen im Verhalten, die sich aus Software- oder Umgebungsänderungen ergeben (Regressionstests)

2.2.1 Funktionale Tests

Funktionales Testen eines Systems beinhaltet Tests, die Funktionen bewerten, die das System ausführen soll. Funktionale Anforderungen können in Arbeitsergebnissen wie fachliche Anforderungsspezifikation, Epics, User-Stories, Anwendungsfällen oder funktionalen Spezifikationen beschrieben sein. Sie können allerdings auch undokumentiert sein. Die Funktionen sind das, „was" das System tun sollte.

Funktionale Tests sollten in allen Teststufen durchgeführt werden (z. B. können Komponententests auf einer Komponentenspezifikation basieren), obwohl der Fokus auf jeder Stufe ein anderer ist, siehe [ISTQB_FL_SYL Abschnitt 2.1 Softwareentwicklungslebenszyklus-Modelle].

Funktionales Testen betrifft das Verhalten der Software. So können Black-Box-Verfahren genutzt werden, um Testbedingungen und Testfälle für die Funktionalität der Komponente oder des Systems abzuleiten (siehe Abschnitt *4.2 Black-Box-Testverfahren*).

Die Gründlichkeit von funktionalen Tests kann durch die funktionale Überdeckung gemessen werden. Die funktionale Überdeckung ist der Grad, in dem eine gewisse Funktionalität durch Testen ausgeführt wurde, und wird als Prozentsatz der Arten ausgedrückt, die abgedeckt sind. Durch die Nutzung der Verfolgbarkeit zwischen Tests und funktionalen Anforderungen kann beispielsweise der Prozentsatz dieser Anforderungen, die durch die Tests angesprochen werden, berechnet werden. So können potenzielle Überdeckungslücken identifiziert werden.

Funktionaler Testentwurf und die Durchführung funktionaler Tests können spezielle Fähigkeiten oder Wissen erfordern, wie die Kenntnis des bestimmten Fachproblems, das die Software löst (z. B. geologische Modellierungssoftware für die Öl- und Gasindustrie).

2.2.2 Nicht-funktionale Tests

Nicht-funktionale Tests eines Systems bewerten Merkmale von Systemen und Software wie Gebrauchstauglichkeit, Performanz oder IT-Sicherheit. In der ISO-Norm ISO/IEC 25010 findet sich eine Klassifizierung von Softwarequalitätsmerkmalen. Nicht-funktionale Tests sind die Tests, die zeigen, "wie gut" das System sich verhält.

Im Gegensatz zu gängigen Fehleinschätzungen können und sollten nicht-funktionale Tests in den meisten Fällen in allen Teststufen und das so früh wie möglich durchgeführt werden. Das späte Entdecken von nicht-funktionalen Fehlerzuständen kann für den Erfolg eines Projekts extrem gefährlich sein.

Um Testbedingungen und Testfälle für nicht-funktionale Tests abzuleiten, können Black-Box-Verfahren (siehe Abschnitt *4.2 Black-Box-Testverfahren*) genutzt werden. Beispielsweise kann die Grenzwertanalyse genutzt werden, um die Stressbedingungen für Performanztests festzulegen.

Die Gründlichkeit von nicht-funktionalen Tests kann durch die nicht-funktionale Überdeckung gemessen werden. Nicht-funktionale Überdeckung ist der Grad, zu dem einige Arten von nicht-funktionalen Elementen durch Tests ausgeführt wurden, und wird als Prozentsatz der Arten der Elemente ausgedrückt, die abgedeckt sind. Beispielsweise lässt sich durch die Nutzung der Verfolgbarkeit zwischen Tests und unterstützten Endgeräten bei mobilen Applikationen der Prozentsatz der Endgeräte, die durch die Kompatibilitätstests betrachtet werden, berechnen, wodurch sich möglicherweise Überdeckungslücken aufdecken lassen.

Nicht-funktionaler Testentwurf und die Durchführung nicht-funktionaler Tests können spezielle Fähigkeiten oder Kenntnisse erfordern, wie die Kenntnis der zugrunde liegenden Schwächen eines Entwurfs oder einer Technologie (z. B. IT-Sicherheitsschwachstellen, die mit bestimmten Programmiersprachen verbunden sind) oder einer bestimmten Benutzergruppe (z. B. die Personas10 von Verwaltungssystemen für Gesundheitseinrichtungen).

In den ISTQB®-ATA, ISTQB®-ATTA und ISTQB®-SEC sowie in anderen ISTQB®-Spezialistenmodulen finden sich weitere Details bezüglich des Testens von nicht-funktionalen Qualitätsmerkmalen.

2.2.3 White-Box-Tests

White-Box-Tests leiten Tests auf Basis der internen Struktur oder der Implementierung eines Systems ab. Die interne Struktur kann Code, Architektur, Workflows und/oder Datenflüsse innerhalb des Systems enthalten (siehe Abschnitt *4.3 White-Box-Testverfahren*).

Die Gründlichkeit von White-Box-Tests kann durch strukturelle Überdeckung gemessen werden. Strukturelle Überdeckung ist der Grad, zu dem einige Arten von strukturellen Elementen durch Tests ausgeführt wurden, und wird ausgedrückt durch den Prozentsatz der Elementart, die übergedeckt wurde.

Auf Stufe der Komponententests basiert die Codeüberdeckung auf dem Prozentsatz des Komponentencodes, der getestet wurde. Die Überdeckung kann in Bezug auf verschiedene Aspekte des Codes (Überdeckungselemente) gemessen werden, wie den Prozentsatz von ausführbaren Anweisungen oder den Prozentsatz von Entscheidungsergebnissen, die in der Komponente getestet wurden. Diese Überdeckungsarten werden zusammengefasst als Codeüberdeckung bezeichnet. Auf der Stufe der Komponentenintegrationstests kann White-Box-Testen auf der Architektur eines Systems basieren, wie den Schnittstellen zwischen den Komponenten. Die strukturelle Überdeckung kann hinsichtlich des Prozentsatzes der Schnittstellen gemessen werden, die durch die Tests ausgeführt wurden.

White-Box-Testentwurf und die Durchführung von White-Box-Tests können spezielle Fähigkeiten und Kenntnisse erfordern, wie Kenntnisse über die Art und Weise, wie der Code aufgebaut ist, wie Daten gespeichert werden (z. B. um mögliche Datenbankanfragen zu bewerten) und wie die Überdeckungswerkzeuge genutzt und ihre Ergebnisse korrekt interpretiert werden.

2.2.4 Änderungsbezogenes Testen

Wenn Änderungen an einem System vorgenommen werden, entweder um einen Fehlerzustand zu korrigieren oder aufgrund einer neuen oder geänderten Funktionalität, sollten Tests durchgeführt werden, um zu bestätigen, dass die Änderungen den Fehlerzustand korrigiert oder die Funktionalität korrekt umgesetzt haben und keine unvorhergesehenen nachteiligen Folgen hervorgerufen haben.

10 Anwenderprofil (Scrum)

- Fehlernachtests: Nachdem ein Fehlerzustand korrigiert wurde, muss die Software mit allen Testfällen getestet werden, die aufgrund des Fehlerzustands fehlgeschlagen sind. Diese sollten mit der neuen Softwareversion erneut ausgeführt werden. Die Software kann auch mit neuen Tests getestet werden, um Änderungen abzudecken, die nötig waren, um einen Fehlerzustand zu beheben. Zumindest müssen die Schritte, die die Fehlerwirkung, die durch den Fehlerzustand entstanden ist, reproduzieren konnten, in der neuen Softwareversion erneut durchgeführt werden. Der Zweck eines Fehlernachtests ist es, zu bestätigen, dass der ursprüngliche Fehlerzustand erfolgreich behoben wurde.
- Regressionstests: Es ist möglich, dass eine Änderung, die in einem Teil des Codes gemacht wurde (egal, ob aufgrund einer Fehlerbehebung oder aufgrund einer anderen Art von Änderung) versehentlich das Verhalten anderer Teile des Codes beeinflusst. Sei es innerhalb der gleichen Komponente, in anderen Komponenten des gleichen Systems oder sogar in anderen Systemen. Änderungen können Änderungen in der Umgebung sein, wie z. B. eine neue Version eines Betriebssystems oder eines Datenbankmanagementsystems. Solche nicht gewünschten Nebeneffekte nennt man Regressionen. Regressionstests beinhalten die Durchführung von Tests, um solche unbeabsichtigten Nebeneffekte zu finden.

Fehlernachtests und Regressionstests werden in allen Teststufen durchgeführt.

Insbesondere in iterativen und inkrementellen Entwicklungslebenszyklen (z. B. agile) resultieren neue Features, Änderungen zu bestehenden Features und Code-Restrukturierung (Refactoring) in häufigen Änderungen am Code, was in gleicher Weise änderungsbezogenes Testen erfordert. Aufgrund der sich kontinuierlich entwickelnden Beschaffenheit des Systems sind Fehlernachtests und Regressionstests sehr wichtig. Dies gilt insbesondere für Systeme des Internets der Dinge, bei denen individuelle Objekte (z. B. Endgeräte) häufig aktualisiert oder ersetzt werden.

Regressionstestsuiten laufen viele Male und entwickeln sich in der Regel nur langsam. So sind Regressionstests ein starker Kandidat für Testautomatisierung. Die Automatisierung dieser Tests sollte früh im Projekt beginnen [siehe ISTQB_FL_SYL – Abschnitt 6 Werkzeugunterstützung für das Testen].

2.3 Wartungstest

Einmal in Produktionsumgebungen verteilt, benötigen Software und Systeme Wartung. Änderungen verschiedener Art in ausgelieferter Software und Systemen sind fast unvermeidbar, entweder um Fehlerzustände zu beheben, die in der betrieblichen Nutzung aufgetreten sind, um neue Funktionalitäten hinzuzufügen oder um bereits gelieferte Funktionalitäten zu löschen oder zu ändern. Wartung wird auch benötigt, um nicht-funktionale Qualitätsmerkmale der Komponente oder des Systems über seine Lebenszeit hinweg zu erhalten oder zu verbessern, insbesondere Performanz, Kompatibilität, Zuverlässigkeit, IT-Sicherheit und Übertragbarkeit.

Wenn Änderungen als Teil der Wartung vorgenommen werden, sollten Wartungstests durchgeführt werden, sowohl um den Erfolg zu bewerten, mit dem die Änderungen vorgenommen wurden, als auch um mögliche Seiteneffekte (z. B. Regressionen) in Teilen des Systems zu prüfen, die unverändert bleiben (was üblicherweise der Großteil des Systems ist). Wartung kann geplante Releases und ungeplante Releases (sog. Hot Fixes) beinhalten.

Ein Wartungsrelease kann, abhängig vom Umfang, Wartungstests in mehreren Teststufen erfordern und verschiedene Testarten nutzen. Der Umfang von Wartungstests hängt ab von:

- Der Risikohöhe der Änderung (z. B. der Grad, zu dem der geänderte Bereich der Software mit anderen Komponenten oder Systemen kommuniziert)
- Der Größe des bestehenden Systems
- Der Größe der Änderung

2.3.1 Auslöser für Wartung

Es gibt verschiedene Gründe, warum Softwarewartung und somit auch Wartungstests stattfinden, sowohl für geplante als auch für ungeplante Änderungen.

Wir können die Auslöser für Wartung wie folgt klassifizieren:

- Modifikation, wie geplante Verbesserungen (z. B. releasegesteuert), korrigierende Änderungen und Notfalländerungen, Änderungen der betrieblichen Umgebung (wie geplante Betriebssystem- oder Datenbankaktualisierungen), Aktualisierungen von kommerzieller Standardsoftware und Patches für Fehlerzustände und Schwachstellen
- Migration, wie von einer Plattform zu einer anderen, was betriebliche Tests der neuen Umgebung und auch der veränderten Software erfordert oder Tests der Datenkonvertierung, wenn Daten von einer anderen Anwendung in das System migriert werden, das gewartet wird
- Außerbetriebnahme, z. B. wenn eine Anwendung das Ende ihres Lebenszyklus erreicht. Wenn eine Anwendung oder ein System außer Betrieb genommen wird, kann dies Tests der Datenmigration oder, wenn lange Datenspeicherungsfristen erforderlich sind Tests der Datenarchivierung erfordern
- Das Testen von Wiederherstellungsverfahren nach der Archivierung bei langen Aufbewahrungsfristen kann ebenso notwendig sein
- Regressionstests können erforderlich sein, um sicherzustellen, dass jede Funktionalität, die in Betrieb bleibt, weiterhin funktioniert.

Für Systeme des Internets der Dinge können Wartungstests durch die Einführung von vollständig neuen oder modifizierten Dingen in das System ausgelöst werden, wie Hardwaregeräte und Softwaredienste. Die Wartungstests für derartige Systeme konzentrieren sich insbesondere auf Integrationstests auf verschiedenen Ebenen (z. B. Netzwerkebene, Anwendungsebene) und auf IT-Sicherheitsaspekte, insbesondere solche, die sich auf persönliche Daten beziehen.

2.3.2 Auswirkungsanalyse für Wartung

Die Auswirkungsanalyse bewertet die Veränderungen, die für ein Wartungsrelease gemacht wurden, um die gewünschten Folgen sowie die erwarteten als auch möglichen Nebeneffekte einer Veränderung zu identifizieren und um die Bereiche des Systems zu identifizieren, die von der Veränderung betroffen sind. Die Auswirkungsanalyse kann auch dabei helfen, die Wirkung einer Veränderung auf bestehende Tests zu identifizieren. Die Nebeneffekte und betroffenen Bereiche im System müssen für Regressionen getestet werden, möglicherweise nach der Aktualisierung aller bestehenden Tests, die von der Veränderung betroffen sind.

Die Auswirkungsanalyse kann vor der Veränderung durchgeführt werden, um bei der Entscheidung zu helfen, ob die Veränderung vorgenommen werden sollte. Dies erfolgt auf der Grundlage der potenziellen Folgen für andere Bereiche des Systems.

Die Auswirkungsanalyse kann schwierig sein, wenn:

- Spezifikationen (z. B. Fachanforderungen, User-Stories, Architektur) veraltet sind oder fehlen
- Testfälle nicht dokumentiert oder veraltet sind
- Bidirektionale Verfolgbarkeit zwischen Tests und der Testbasis nicht aufrechterhalten wurde
- Werkzeugunterstützung schwach ist oder nicht existiert
- Die beteiligten Personen keine Fach- oder Systemkenntnis haben
- Während der Entwicklung nur ungenügende Aufmerksamkeit auf die Wartbarkeit der Software gelegt wurde

3. Statischer Test – 225 Minuten

Schlüsselbegriffe

Ad-hoc-Review, checklistenbasiertes Review, Datenflussanalyse, Definition-Verwendungspaar, dynamischer Test, Kontrollflussanalyse, paarweiser Integrationstest, perspektivisches Lesen, Review, rollenbasiertes Review, statische Analyse, statischer Test, szenariobasiertes Review, Umgebungsintegrationstest, zyklomatische Komplexität

Lernziele für den statischen Test

3.1 Grundlagen des statischen Tests (nicht prüfungsrelevant)

FL-3.1.1 (K1) Arten von Softwarearbeitsergebnissen erkennen können, die durch die verschiedenen statischen Testverfahren geprüft werden können

FL-3.1.2 (K2) Beispiele nennen können, um den Wert des statischen Tests zu beschreiben

FL-3.1.3 (K2) Den Unterschied zwischen statischen und dynamischen Verfahren unter Berücksichtigung der Ziele, der zu identifizierenden Fehlerzustände und der Rolle dieser Verfahren innerhalb des Softwarelebenszyklus erklären können

3.2 Reviewverfahren anwenden

FL-3.2.4 (K3) Ein Reviewverfahren auf ein Arbeitsergebnis anwenden können, um Fehlerzustände zu finden

HO-3.2.4 (H2) Ein Stück Code anhand einer Checkliste reviewen. Die Befunde dokumentieren.

Leitfaden für die praktische Kompetenzstufe:

Eine typische Checkliste für ein Codereview mit verschiedenen Anomalien bereitstellen.

Ein Stück Code bereitstellen, das verschiedene Anomalien aus der Checkliste enthält. Den Teilnehmern ist eine Vorlage für eine Befundliste zur Verfügung zu stellen, um dort die Befunde zu dokumentieren. Die Befundliste wird mit den Teilnehmern durchgesprochen.

3.3 Statische Analyse

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

HO-3.2.1 (H1) Auf ein Stück Code ein statisches Analysewerkzeug anwenden, um typische Kontrollflussanomalien zu finden. Den Bericht des Werkzeugs verstehen und wie die Anomalien die Qualitätsmerkmale des Produkts beeinträchtigen

Leitfaden für die praktische Kompetenzstufe:

Ein oder mehrere Stücke Code bereitstellen, die syntaktisch korrekt sind, und verschiedene Arten von Kontrollflussanomalien enthalten, die im Syllabus erwähnt werden.

Die Teilnehmer bei der Ausführung des statischen Analysewerkzeugs und beim Anzeigen des Berichts über die Anomalien anleiten. Die Teilnehmer sollen die gefundenen Fehlerzustände besprechen und die betroffenen Qualitätsmerkmale benennen (funktionale Korrektheit, Wartbarkeit, IT-Sicherheit etc.).

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

HO-3.2.2 (H1) Für ein Stück Code den Bericht eines statischen Analysewerkzeug betreffend Datenflussanomalien verstehen und wie die Anomalien funktionale Korrektheit und Wartbarkeit beeinträchtigen

Leitfaden für die praktische Kompetenzstufe:

Ein Stück Code bereitstellen, das syntaktisch korrekt ist, welches bei einigen Variablen die wesentlichen Arten von Datenflussanomalien, welche im Syllabus erwähnt werden, enthält.

Führen Sie das statische Analysewerkzeug aus, erklären Sie den Teilnehmern die gefundenen Datenflussanomalien, und erörtern Sie mit ihnen die Auswirkungen auf die funktionale Korrektheit bzw. auf die Wartbarkeit.

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

HO-3.2.3 (H2) Für ein Stück Code, das gegen gegebene Programmierkonventionen und -richtlinien verstößt, die Wartbarkeitsmängel beheben, welche vom statischen Codeanalysewerkzeug berichtet wurden. Anschließend durch Nachtest bestätigen, dass sie korrigiert sind und verifizieren, dass keine neuen Mängel eingeführt wurden

Leitfaden für die praktische Kompetenzstufe:

Ein Dokument mit Programmierkonventionen und -richtlinien den üblichen Arten von Anforderungen aus dem Syllabus zur Verfügung stellen.

Ein Stück Code zur Verfügung stellen, das syntaktisch korrekt ist, und Verstöße gegen die Programmkonventionen und -richtlinien enthält.

Ein statisches Analysewerkzeug ausführen, und den Teilnehmern den Bericht über die Verstöße zur Verfügung stellen. Die Teilnehmer sollen die Wartbarkeitsfehler auf Basis der Programmierkonventionen und -richtlinien und den Hinweisen des statischen Analysewerkzeugs korrigieren. Sie sollen die statische Analyse nachtesten und bestätigen, dass die Fehlerzustände korrigiert und keine neuen hinzugefügt worden sind.

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

3.1 *Grundlagen des statischen Tests (nicht prüfungsrelevant)*

Im Gegensatz zu dynamischen Tests, die die Ausführung der Software, die getestet wird, erfordern, verlässt sich statischer Test auf die manuelle Prüfung der Arbeitsergebnisse (d.h. Reviews) oder auf die werkzeuggestützte Bewertung des Codes oder anderer Arbeitsergebnisse (d.h. statische Analyse). Beide Arten des statischen Tests beurteilen den Code oder andere Arbeitsergebnisse, die getestet werden, ohne den Code oder das Arbeitsergebnis, das getestet wird, tatsächlich auszuführen.

Die statische Analyse ist wichtig für sicherheitskritische Computersysteme (z. B. Luftfahrt, medizinische oder Kernkraftsoftware), aber statische Analysen sind auch in anderen Bereichen wichtig und üblich geworden. Zum Beispiel ist die statische Analyse ein wichtiger Teil der IT-Sicherheitstests. Die statische Analyse ist auch häufig eingebettet in Werkzeuge für automatisierte Software Build & Verteilung, z. B. in der agilen Entwicklung, in der kontinuierlichen Auslieferung (continuous delivery) und in der kontinuierlichen Bereitstellung in die Zielumgebung (continuous deployment.

3.1.1 Arbeitsergebnisse, die durch statische Tests geprüft werden können (nicht prüfungsrelevant)

Fast jedes Arbeitsergebnis kann durch statische Tests geprüft werden (Reviews und/oder statische Analysen), z. B.:

- *Spezifikationen, u. a. Fachanforderungen, funktionale Anforderungen, und IT-Sicherheitsanforderungen*
- *Epics, User-Stories und Abnahmekriterien*
- *Architektur und Entwurfsspezifikationen*
- *Code*
- *Testmittel einschließlich Testkonzepten, Testfällen, Testablauf und automatisierten Testskripten*
- *Benutzeranleitungen*
- *Webseiten*
- *Verträge, Projektpläne, Zeitpläne und Budgetplanung*
- *Aufsetzen der Konfiguration u. Infrastruktur*
- *Modelle wie Aktivitätsdiagramme, die für modellbasiertes Testen genutzt werden können (siehe ISTQB®-MBT und Kramer 2016)*

Reviews können auf jedes Arbeitsergebnis angewendet werden, von dem die Teilnehmer wissen, wie es zu lesen und zu verstehen ist. Statische Analysen können effizient auf jedes Arbeitsergebnis mit einer formalen Struktur angewendet werden (üblicherweise Code oder Modelle), für die es ein geeignetes statisches Analysewerkzeug gibt. Statische Analysen können sogar mit Werkzeugen durchgeführt werden, die Arbeitsergebnisse bewerten, die wie Anforderungen in natürlicher Sprache geschrieben sind (z. B. Prüfung auf Rechtschreibung, Grammatik und Lesbarkeit).

3.1.2 Vorteile des statischen Tests (nicht prüfungsrelevant)

Statische Testverfahren haben eine Reihe von Vorteilen. Wenn sie früh im Softwareentwicklungslebenszyklus eingesetzt werden, können statische Tests das frühe Erkennen von Fehlerzuständen ermöglichen, noch bevor dynamische Tests durchgeführt werden (z. B. in Anforderungs- oder Entwurfsspezifikationsreviews, in Backlog-Refinements usw.). Fehlerzustände, die früh gefunden werden, lassen sich häufig kostengünstiger entfernen als Fehlerzustände, die später im

Lebenszyklus gefunden werden, insbesondere im Vergleich zu Fehlerzuständen, die erst gefunden werden, nachdem die Software auf die Zielumgebung gebracht wurde und aktiv in Nutzung ist. Die Nutzung statischer Testverfahren zum Auffinden von Fehlerzuständen und die folgende prompte Behebung dieser Fehlerzustände ist für das Unternehmen fast immer kostengünstiger als der Einsatz dynamischer Tests zum Auffinden von Fehlerzuständen im Testobjekt und deren Behebung. Dies gilt insbesondere unter Berücksichtigung der zusätzlichen Kosten, die mit der Aktualisierung anderer Arbeitsergebnisse sowie der Durchführung von Fehlernachtests und Regressionstests einhergehen.

Zusätzliche Vorteile statischer Tests können u. a. sein:

- *Effizienteres Erkennen und Korrigieren von Fehlerzuständen schon vor der Durchführung dynamischer Tests*
- *Identifizieren von Fehlerzuständen, die in dynamischen Tests nicht leicht zu finden sind*
- *Verhindern von Fehlerzuständen im Entwurf oder der Kodierung durch das Aufdecken von Inkonsistenzen, Mehrdeutigkeiten, Widersprüchen, Auslassungen, Ungenauigkeiten und Redundanzen in Anforderungen*
- *Erhöhen der Entwicklungsproduktivität (z. B. durch verbesserte Entwürfe, mehr wartungsfähigen Code)*
- *Reduzieren von Entwicklungskosten und -zeit*
- *Reduzieren von Testkosten und -zeit*
- *Reduzieren der Gesamtkosten der Qualität über die Lebenszeit der Software hinweg, aufgrund von weniger Fehlerwirkungen zu einem späteren Zeitpunkt im Lebenszyklus oder nach Auslieferung in die Produktion*
- *Verbesserte Kommunikation zwischen Teammitgliedern durch die Teilnahme an Reviews*

3.1.3 Unterschiede zwischen statischen und dynamischen Tests (nicht prüfungsrelevant)

Statischer und dynamischer Test können die gleichen Ziele verfolgen (siehe Abschnitt 1.1.1 Typische Ziele des Testens), wie die Beurteilung der Qualität von Arbeitsergebnissen und das Identifizieren von Fehlerzuständen so früh wie möglich. Statischer und dynamischer Test ergänzen sich gegenseitig, da sie unterschiedliche Fehlerarten finden.

Ein Hauptunterschied ist, dass statische Tests Fehlerzustände in Arbeitsergebnissen direkt finden, anstatt erst die Fehlerwirkungen zu identifizieren, die durch die Fehlerzustände entstehen, wenn die Software läuft. Ein Fehlerzustand kann in einem Arbeitsergebnis über einen langen Zeitraum hinweg bestehen, ohne eine Fehlerwirkung zu verursachen. Der Pfad, auf dem der Fehlerzustand liegt, wird vielleicht selten genutzt oder ist schwer zu erreichen. Es ist dann nicht einfach, einen dynamischen Test zu konstruieren und durchzuführen, der den Fehlerzustand aufdeckt. Statische Tests können einen solchen Fehlerzustand unter Umständen mit wesentlich geringerem Aufwand finden.

Ein weiterer Unterschied ist, dass statische Tests genutzt werden können, um die Konsistenz und interne Qualität der Arbeitsergebnisse zu verbessern, während dynamische Tests in der Regel auf extern sichtbares Verhalten konzentriert sind.

Im Vergleich zu dynamischen Tests sind typische Fehlerzustände, die einfacher und kostengünstiger durch statische Tests zu finden und zu beheben sind, u. a. folgende:

- *Anforderungsfehler (z. B. Inkonsistenzen, Mehrdeutigkeiten, Widersprüche, Auslassungen, Ungenauigkeiten und Redundanzen)*
- *Entwurfsfehler (z. B. ineffiziente Algorithmen oder Datenbankstrukturen, hohe Koppelung, geringe Kohäsion)*

- *Programmierfehler (z. B. Variablen mit nicht definierten Werten, Variablen, die deklariert, aber nie genutzt werden, unerreichbarer Code, doppelter Code)*
- *Abweichungen von Standards (z. B. fehlende Einhaltung von Programmierrichtlinien)*
- *Falsche Schnittstellenspezifikationen (z. B. unterschiedliche Maßeinheiten im aufrufenden und im aufgerufenen System)*
- *Schwachstellen in der IT-Sicherheit (z. B. Neigung zu Pufferüberlauf)*
- *Lücken oder Ungenauigkeiten in der Verfolgbarkeit oder die Überdeckung der Testbasis (z. B. fehlende Tests für ein Abnahmekriterium)*

Darüber hinaus können die meisten Wartbarkeitsfehler nur durch statische Tests gefunden werden (z. B. ungeeignete Modularisierung, schlechte Wiederverwendbarkeit von Komponenten, schwer analysierbarer Code, der ohne das Einfügen neuer Fehlerzustände schwer zu modifizieren ist).

3.2 Reviewverfahren anwenden

Reviews variieren von informellen bis zu formalen Reviews. Informelle Reviews sind dadurch gekennzeichnet, dass sie keinem definierten Prozess folgen und kein formal dokumentiertes Ergebnis haben. Formale Reviews sind durch die Teilnahme von Teams, dokumentierten Ergebnissen des Reviews und dokumentierten Vorgehensweisen für die Durchführung des Reviews gekennzeichnet. Die Formalität eines Reviewprozesses hängt von Faktoren wie dem Softwareentwicklungslebenszyklus-Modell, der Reife des Entwicklungsprozesses, der Komplexität der zu prüfenden Arbeitsergebnisse, rechtlichen oder regulatorischen Anforderungen und/oder der Notwendigkeit für einen Prüfnachweis ab.

Der Fokus eines Reviews hängt von den vereinbarten Zielen des Reviews ab (z. B. Finden von Fehlerzuständen, Verständnisgewinnung, Training von Teilnehmern wie Testern und neuen Teammitgliedern oder Diskussion und Entscheidung durch Konsens).

Die ISO-Norm (ISO/IEC 20246) enthält tiefergehende Beschreibungen des Reviewprozesses für Arbeitsergebnisse sowie Rollen und Reviewverfahren.

3.2.1 Die Anwendung von Reviewverfahren

Es gibt eine Reihe von Reviewverfahren, die während des individuellen Reviews (d.h. der individuellen Vorbereitung) angewendet werden können, um Fehlerzustände zu erkennen. Diese Verfahren können für alle oben beschriebenen Reviewarten verwendet werden. Die Effektivität der Verfahren kann in Abhängigkeit von der genutzten Reviewart variieren. Beispiele für unterschiedliche individuelle Reviewverfahren für verschiedene Reviewarten werden unten aufgelistet.

Ad hoc

In einem Ad-hoc-Review wird Gutachtern nur wenig oder gar keine Anleitung an die Hand gegeben, wie die Aufgabe ausgeführt werden soll. Gutachter lesen häufig das Arbeitsergebnis sequenziell und identifizieren und dokumentieren Befunde, sobald sie auf sie stoßen. Ad-hoc-Reviews sind ein verbreitet genutztes Verfahren, das wenig Vorbereitung erfordert. Dieses Verfahren hängt stark von den Fähigkeiten der Gutachter ab und kann zu vielen doppelt berichteten Befunden führen.

Checklistenbasiert

Ein checklistenbasiertes Review ist ein systematisches Verfahren, bei dem die Gutachter Befunde auf Basis von Checklisten erkennen, die bei Reviewbeginn verteilt werden (z. B. durch den Reviewmoderator). Eine Reviewcheckliste besteht aus einem Set an Fragen, die auf potenziellen Fehlerzuständen basieren und sich aus Erfahrungen ableiten lassen. Checklisten sollten spezifisch auf die Art des Arbeitsergebnisses, das Gegenstand des Reviews ist, zugeschnitten sein und regelmäßig aktualisiert werden, um Befundarten, die in früheren Reviews übersehen wurden, abzudecken. Der

Hauptvorteil des checklistenbasierten Verfahrens ist die systematische Überdeckung typischer Fehlerarten. Es sollte darauf geachtet werden, nicht nur einfach der Checkliste im individuellen Review zu folgen, sondern auch ein Auge auf Fehlerzustände außerhalb der Checkliste zu haben.

Szenarien und Dry Runs (Probeläufe)

In einem szenariobasierten Review erhalten Gutachter strukturierte Richtlinien, wie sie ein Arbeitsergebnis durchlesen sollen. Ein szenariobasiertes Review unterstützt Gutachter dabei, Probeläufe mit dem Arbeitsergebnis auf Basis der erwarteten Nutzung des Arbeitsergebnisses durchzuführen (falls das Arbeitsergebnis in einem angemessenen Format, wie in Form von Anwendungsfällen, dokumentiert ist). Diese Szenarien geben Gutachtern bessere Richtlinien darüber an die Hand, wie spezifische Fehlerarten identifiziert werden können, als einfache Checklisteneinträge. Wie bei checklistenbasierten Reviews sollten Gutachter sich nicht ausschließlich auf die dokumentierten Szenarien beschränken, um andere Fehlerarten (z. B. fehlende Leistungsmerkmale) nicht zu übersehen.

Perspektivisch

Ähnlich wie im rollenbasierten Review (s.u.) nehmen beim perspektivischen Lesen Gutachter im individuellen Review unterschiedliche Standpunkte von unterschiedlichen Stakeholdern ein. Typische Standpunkte von Stakeholdern sind u. a. Endanwender, Marketing, Designer, Tester oder Betrieb. Die Nutzung unterschiedlicher Stakeholder-Standpunkte führt zu mehr Tiefe im individuellen Review mit weniger Doppelungen von Befunden unter den Gutachtern.

Darüber hinaus verlangt perspektivisches Lesen von den Gutachtern auch, dass sie versuchen, das Arbeitsergebnis, das Gegenstand des Reviews ist, zu nutzen, um das Produkt zu generieren, das sie daraus ableiten wollen. Zum Beispiel würde ein Tester versuchen, einen Entwurf für Abnahmetests zu generieren, wenn er perspektivisches Lesen bei einer Anforderungsspezifikation durchführt, um zu sehen, ob alle notwendigen Informationen enthalten sind. Außerdem wird erwartet, dass im perspektivischen Lesen Checklisten genutzt werden.

Empirische Studien haben gezeigt, dass perspektivisches Lesen das effektivste allgemeine Verfahren für das Review von Anforderungen und technischen Arbeitsergebnissen ist. Ein Schlüsselfaktor für den Erfolg ist der korrekte Einbezug und die angemessene Gewichtung der Sichtweisen der unterschiedlichen Stakeholder basierend auf den Risiken. Siehe Shull 2000 für Details über perspektivisches Lesen und Sauer 2000 für die Effektivität unterschiedlicher Reviewverfahren.

Rollenbasiert

Ein rollenbasiertes Review ist ein Verfahren, in dem die Gutachter das Arbeitsergebnis aus der Perspektive von individuellen Stakeholderrollen bewerten. Typische Rollen beinhalten spezifische Arten von Endanwendern (erfahren, unerfahren, Senioren, Kinder usw.) und spezifische Rollen innerhalb einer Organisation (Benutzeradministrator, Systemadministrator, Performanztester usw.). Es werden die gleichen Prinzipien wie beim perspektivischen Lesen angewendet, da die Rollen ähnlich sind.

3.3 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.

3.3.1 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.

3.3.2 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 Programmierer 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 Variablenzuweisung 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 Kontrollflussanalyse. 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.

3.3.3 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 aufwendiger 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

3.3.4 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].

4. Testverfahren – 450 Minuten

Schlüsselbegriffe

Anweisungstest, Anweisungsüberdeckung, Anwendungsfallbasierter Test, Äquivalenzklassenbildung, atomare Bedingung, Black-Box-Testverfahren, Entscheidungstabellentest, Entscheidungstest, Entscheidungsüberdeckung, erfahrungsbasierte Testverfahren, Grenzwertanalyse, Mehrfachbedingungstest, modifizierter Bedingungs-/Entscheidungstest, Testverfahren, Überdeckung, verkürzte Auswertung, White-Box-Testverfahren, Zustandsübergangstest

Lernziele für Testverfahren

4.1 Kategorien von Testverfahren

FL-4.1.1 (K2) Die Eigenschaften, Gemeinsamkeiten und Unterschiede zwischen Black-Box-Testverfahren, White-Box-Testverfahren und erfahrungsbasierten Testverfahren erklären können

4.2 Black-Box-Testverfahren

FL-4.2.1 (K3) Die Äquivalenzklassenbildung anwenden können, um Testfälle aus vorgegebenen Anforderungen abzuleiten

HO-4.2.1 (H2) Für ein vorhandenes Spezifikationselement ist, unter Anwendung der Äquivalenzklassenbildung, eine Testsuite zu spezifizieren und zu implementieren. Die Testsuite ist mit der zugehörigen Software auszuführen.

Leitfaden für die praktische Kompetenzstufe:

Ein Spezifikationselement und die zugehörige Software als Testobjekt zur Verfügung stellen. Das Testobjekt soll Fehlerzustände enthalten, die mit Hilfe von Äquivalenzklassenbildung aufgedeckt werden können.

Die Teilnehmer sollen die Testfälle spezifizieren, implementieren, ausführen, und sicherstellen, dass alle Äquivalenzklassen abgedeckt sind. Falls nicht, sollen sie Testfälle hinzufügen bis das Ziel erreicht ist und alle Fehlerzustände entdeckt wurden.

Optional: Die Fehlerzustände korrigieren und die Testfälle erneut durchführen, um nachzuweisen, dass die Fehlerzustände korrigiert und keine neuen hinzugefügt worden sind.

Das Beispiel soll zu Testfällen für mindestens 2 gültige und mindestens 1 ungültige Äquivalenzklasse führen.

FL-4.2.2 (K3) Die Grenzwertanalyse anwenden können, um Testfälle aus vorgegebenen Anforderungen abzuleiten

HO-4.2.2 (H2) Für ein vorhandenes Spezifikationselement ist, unter Anwendung der Grenzwertanalyse, eine Testsuite zu spezifizieren und zu implementieren. Die Testsuite ist mit der entsprechenden Software auszuführen.

Leitfaden für die praktische Kompetenzstufe:

Ein Spezifikationselement und die zugehörige Software als Testobjekt zur Verfügung stellen. Das Testobjekt soll Fehlerzustände enthalten, die mit Hilfe von Grenzwertanalyse entdeckt werden können.

Die Teilnehmer sollen die Testfälle spezifizieren, implementieren, ausführen, und sicherstellen, dass alle Grenzwerte abgedeckt sind. Falls nicht, sollen sie Testfälle hinzufügen bis das Ziel erreicht ist und alle Fehlerzustände entdeckt wurden.

Optional: Die Fehlerzustände korrigieren und die Testfälle erneut durchführen, um nachzuweisen, dass die Fehlerzustände korrigiert und keine neuen hinzugefügt worden sind.

Das Beispiel soll zu Testfällen für mindestens 4 Grenzwerte (2 Grenzen) führen.

FL-4.2.3 (K3) Entscheidungstabellentests anwenden können, um Testfälle aus vorgegebenen Anforderungen abzuleiten

HO-4.2.3 (H2) Für ein vorhandenes Spezifikationselement ist, unter Anwendung von Entscheidungstabellentests, eine Testsuite zu spezifizieren und zu implementieren. Die Testsuite ist mit der entsprechenden Software auszuführen.

Leitfaden für die praktische Kompetenzstufe:

Ein Spezifikationselement und die zugehörige Software als Testobjekt zur Verfügung stellen. Das Testobjekt soll Fehlerzustände enthalten, die mit Hilfe von Entscheidungstabellentests entdeckt werden können.

Die Teilnehmer sollen die Testfälle spezifizieren, implementieren, ausführen, und sicherstellen, dass alle Tabelleneinträge abgedeckt sind. Falls nicht, sollen sie Testfälle hinzufügen bis das Ziel erreicht ist und alle Fehlerzustände entdeckt wurden.

Optional: Die Fehlerzustände korrigieren und die Testfälle erneut durchführen, um nachzuweisen, dass die Fehlerzustände korrigiert und keine neuen hinzugefügt worden sind.

Die Tabelle im Beispiel soll mindestens 3 Bedingungen enthalten.

FL-4.2.4 (K3) Zustandsübergangstests anwenden können, um Testfälle aus vorgegebenen Anforderungen abzuleiten

HO-4.2.4 (H2) Für ein vorhandenes Spezifikationselement ist, unter Anwendung von Zustandsübergangstests, eine Testsuite zu spezifizieren und zu implementieren. Die Testsuite ist mit der entsprechenden Software auszuführen.

Leitfaden für die praktische Kompetenzstufe:

Ein Stück Code und die zugehörige Softwarekomponentenspezifikation zur Verfügung stellen. Einige ausführbare Anweisungen sollen Fehlerzustände enthalten, die mit Hilfe von Zustandsübergangstests entdeckt werden können (z. B. Zustandsautomat).

Die Teilnehmer sollen die Testfälle spezifizieren, implementieren, ausführen, und sicherstellen, dass alle Zustandsübergänge abgedeckt sind. Falls nicht, sollen sie Testfälle hinzufügen bis das Ziel erreicht ist und alle Fehlerzustände entdeckt wurden.

Das Beispiel soll zu mindestens 5 Testfällen führen.

FL-4.2.5 (K2) Erklären können, wie man Testfälle aus einem Anwendungsfall ableitet

4.3 White-Box-Testverfahren

4.3.1 Anweisungstest

FL-4.3.1 (K2) Anweisungsüberdeckung erklären können

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

HO-2.2.1 (H2) Für ein gegebenes Spezifikationselement und ein entsprechendes Stück Code eine Testsuite mit dem Ziel einer 100%igen Anweisungsüberdeckung entwerfen und implementieren, und nach der Ausführung verifizieren, dass das Testziel erreicht wurde.

Leitfaden für die praktische Kompetenzstufe:

Ein Stück Code und die entsprechende Softwarekomponentenspezifikation zur Verfügung stellen. Einige ausführbare Anweisungen sollen Fehlerzustände enthalten, welche durch Anweisungsüberdeckung gefunden werden können.

Die Teilnehmer sollten die Testfälle spezifizieren, implementieren und ausführen, und sicherstellen, dass 100%ige Anweisungsüberdeckung erreicht ist. Falls nicht, sollten sie weitere Testfälle hinzufügen, bis das Testziel erreicht ist und alle Fehlerzustände entdeckt wurden.

Das Beispiel soll zu mindestens 3 Testfällen führen.

4.3.2. Entscheidungstest

FL-4.3.2 (K2) Entscheidungsüberdeckung erklären können

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

HO-2.3.1 (H2) Für ein gegebenes Spezifikationselement und ein entsprechendes Stück Code eine Testsuite mit dem Ziel einer 100%igen Entscheidungsüberdeckung entwerfen und implementieren, und nach der Ausführung verifizieren, dass das Testziel erreicht wurde.

Leitfaden für die praktische Kompetenzstufe:

Ein Stück Code und die entsprechende Softwarekomponentenspezifikation zur Verfügung stellen. Einige Entscheidungen oder Ausführungspfade sollen Fehlerzustände enthalten, welche durch Entscheidungsüberdeckung, aber nicht unbedingt durch Anweisungsüberdeckung gefunden werden können.

Die Teilnehmer sollen die Testfälle spezifizieren, implementieren und ausführen, und verifizieren, dass 100%ige Entscheidungsüberdeckung erreicht ist. Falls nicht, sollten sie weitere Testfälle hinzufügen, bis das Testziel erreicht ist und alle Fehlerzustände entdeckt wurden.

Das Beispiel soll zu mindestens 3 Testfällen führen. Es soll den Vorteil der Entscheidungsüberdeckung gegenüber der Anweisungsüberdeckung zeigen.

4.3.3 Der Beitrag von Anweisungs- und Entscheidungstests

FL-4.3.3 (K2) Die Bedeutung von Anweisungs- und Entscheidungsüberdeckung erklären können

4.3.4 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

HO-2.4.1 (H2) Für ein gegebenes Spezifikationselement und ein entsprechendes Stück Code, das eine Entscheidung mit mehreren atomaren Bedingungen enthält, eine Testsuite mit dem Ziel einer 100%igen modifizierten Bedingungs-/Entscheidungsüberdeckung entwerfen, implementieren und ausführen.

Leitfaden für die praktische Kompetenzstufe:

Ein Stück Code, der eine Entscheidung mit mehreren, voneinander unabhängigen, atomaren Bedingungen enthält, und die zugehörige Softwareentwicklungsspezifikation zur Verfügung stellen. Die Entscheidung im Code soll einen Fehlerzustand enthalten, der durch modifizierte Bedingungs-/Entscheidungsüberdeckung gefunden werden kann.

Die Teilnehmer sollen die Testfälle spezifizieren, implementieren und ausführen, und verifizieren, dass 100%ige modifizierte Bedingungs-/ Entscheidungsüberdeckung erreicht ist.

Das Beispiel soll zu Testfällen für mindestens 3 atomare Bedingungen führen. Es soll die Vorteile gegenüber dem Entscheidungstest zeigen.

4.4 Erfahrungsbasierte Testverfahren (nicht prüfungsrelevant)

FL-4.4.1 (K2) Die intuitive Testfallermittlung erklären können

FL-4.4.2 (K2) Exploratives Testen erklären können

FL-4.4.3 (K2) Checklistenbasiertes Testen erklären können

4.1 Kategorien von Testverfahren

Zweck von Testverfahren, einschließlich der hier beschriebenen, ist es, eine Hilfe beim Bestimmen von Testbedingungen, Testfällen und Testdaten zu sein.

Die Wahl, welches Testverfahren zu nutzen ist, hängt von einer Reihe von Faktoren ab, darunter u. a.:

- Die Komplexität der Komponente oder des Systems
- Regulatorische Standards
- Kundenanforderungen oder vertragliche Anforderungen
- Risikostufe und Risikoarten
- Verfügbare Dokumentation
- Kenntnisse und Fähigkeiten des Testers
- Verfügbare Werkzeuge
- Zeit und Budget
- Softwareentwicklungslebenszyklus-Modell
- Die Arten von Fehlerzuständen, die in der Komponente oder dem System erwartet werden

Einige Verfahren sind in bestimmten Situationen und auf bestimmten Teststufen geeigneter, andere passen auf alle Teststufen. Wenn Testfälle erstellt werden, nutzen Tester in der Regel eine Kombination aus verschiedenen Testverfahren, um mit gegebenem Testaufwand die besten Ergebnisse zu erzielen.

Die Nutzung von Testverfahren in der Testanalyse, im Testentwurf und in der Testrealisierung kann von sehr informell (wenig bis gar keine Dokumentation) bis hin zu sehr formal reichen. Der passende Grad an Formalität hängt vom Kontext des Testens ab, u. a. von der Reife des Test- und Entwicklungsprozesses, zeitlichen Beschränkungen, Informationssicherheits- oder regulatorischen Anforderungen, den Kenntnissen und Fähigkeiten der involvierten Personen und dem eingesetzten Softwareentwicklungslebenszyklus-Modell.

4.1.1 Kategorien von Testverfahren und ihre Eigenschaften

In diesem Lehrplan werden Testverfahren als Black-Box-, White-Box- oder erfahrungsbasierte Testverfahren klassifiziert.

Black-Box-Testverfahren (auch spezifikationsbasierte Verfahren genannt) basieren auf einer Analyse der zugehörigen Testbasis (z. B. formale Anforderungsdokumente, Spezifikationen, Anwendungsfälle, User-Stories oder Geschäftsprozesse). Diese Verfahren können sowohl auf funktionale als auch auf nicht-funktionale Tests angewendet werden. Black-Box-Testverfahren konzentrieren sich auf die Eingaben und Ausgaben des Testobjekts, ohne seine interne Struktur zu berücksichtigen.

White-Box-Testverfahren (auch strukturelle oder strukturbasierte Verfahren genannt) basieren auf einer Analyse der Architektur, dem Feinentwurf, der internen Struktur oder dem Code des Testobjekts. Anders als Black-Box-Testverfahren konzentrieren sich White-Box-Testverfahren auf die Struktur und die Abläufe innerhalb des Testobjekts.

Erfahrungsbasierte Testverfahren nutzen die Erfahrung von Entwicklern, Testern und Benutzern, um Tests zu entwerfen, umzusetzen und auszuführen. Diese Verfahren werden oft mit Black-Box- und White-Box-Verfahren kombiniert.

Gängige Merkmale von Black-Box-Testverfahren sind u. a.:

- Testbedingungen, Testfälle und Testdaten werden aus einer Testbasis abgeleitet, die Softwareanforderungen, Spezifikationen, Anwendungsfälle und User-Stories beinhalten kann.

- Testfälle können genutzt werden, um Lücken zwischen den Anforderungen und der Realisierung der Anforderungen sowie Abweichungen von den Anforderungen zu erkennen.
- Die Überdeckung wird anhand der getesteten Elemente in der Testbasis gemessen und aufgrund der Verfahren, die auf die Testbasis angewendet wird.

Gängige Merkmale von White-Box-Testverfahren sind u. a.:

- Testbedingungen, Testfälle und Testdaten werden aus einer Testbasis abgeleitet, die Code, Softwarearchitektur, Feinentwurf oder andere Arten an Informationen zur Struktur der Software enthalten kann.
- Die Überdeckung wird auf Basis der getesteten Elemente innerhalb einer ausgewählten Struktur gemessen (z. B. dem Code oder den Schnittstellen) und das Verfahren auf die Testbasis angewendet.

Gängige Merkmale von erfahrungsbasierten Testverfahren sind u. a.:

- Testbedingungen, Testfälle und Testdaten werden aus einer Testbasis abgeleitet, die schlicht aus den Kenntnissen und Erfahrungen der Tester, Entwickler, Benutzer und anderer Stakeholder bestehen kann.
- Diese Kenntnisse und Erfahrungen beinhalten die erwartete Nutzung der Software, ihrer Umgebung, mögliche Fehlerzustände und die Verteilung dieser Fehlerzustände.

Der internationale Standard ISO/IEC/IEEE 29119-4 enthält Beschreibungen von Testverfahren und ihren zugehörigen Überdeckungsmaßen (siehe Craig 2002 und Copeland 2004 für weitere Informationen über Verfahren).

4.2 Black-Box-Testverfahren

4.2.1 Äquivalenzklassenbildung

Äquivalenzklassenbildung teilt Daten so in Äquivalenzklassen auf (auch als Partitionen bezeichnet), dass alle Elemente einer vorgegebenen Äquivalenzklasse erwartungsgemäß in derselben Art und Weise verarbeitet werden (siehe Kaner 2013 und Jorgensen 2014). Es gibt Äquivalenzklassen für gültige und ungültige Werte.

- Gültige Werte sind Werte, die von der Komponente oder dem System akzeptiert werden sollten. Eine Äquivalenzklasse, die gültige Werte enthält, wird “gültige Äquivalenzklasse” genannt.
- Ungültige Werte sind Werte, die von der Komponente oder dem System abgelehnt werden sollten. Eine Äquivalenzklasse, die ungültige Werte enthält, wird “ungültige Äquivalenzklasse” genannt.
- Äquivalenzklassen können für jedes Datenelement in Bezug auf das Testobjekt gebildet werden, u. a. für Eingaben, Ausgaben, interne Werte, zeitbezogene Werte (z. B. vor oder nach einem Ereignis) und für Schnittstellenparameter (z. B. integrierte Komponenten, die während Integrationstests getestet werden).
- Jede Klasse kann (wenn nötig) in Unterklassen unterteilt werden.
- Jeder Wert muss eindeutig einer einzigen Äquivalenzklasse zugeordnet werden können.
- Wenn ungültige Äquivalenzklassen in Testfällen verwendet werden, sollten sie individuell getestet werden, d.h. nicht in Kombination mit anderen ungültigen Äquivalenzklassen, um sicherzustellen, dass Fehlerwirkungen nicht maskiert bleiben. Fehlerwirkungen können maskiert bleiben, wenn mehrere Fehlerwirkungen zur gleichen Zeit auftreten, aber nur eine sichtbar ist, so dass die anderen Fehlerwirkungen unentdeckt bleiben.

Um mit diesem Verfahren eine 100%ige Überdeckung zu erzielen, müssen die Testfälle alle festgelegten Klassen (auch ungültige Klassen) abdecken, indem sie mindestens einen Wert aus jeder Klasse nutzen. Die Überdeckung wird gemessen durch die Anzahl der Äquivalenzklassen, die durch mindestens einen Wert getestet wurden, dividiert durch die Gesamtzahl der identifizierten Äquivalenzklassen, üblicherweise in Prozent ausgedrückt. Äquivalenzklassenbildung kann auf allen Teststufen angewendet werden.

4.2.2 Grenzwertanalyse

Die Grenzwertanalyse ist eine Erweiterung der Äquivalenzklassenbildung, aber sie kann nur dann genutzt werden, wenn die Klasse geordnet ist und aus numerischen oder sequenziellen Daten besteht. Die Minimum- und Maximum-Werte (oder erste und letzte Werte) einer Klasse sind ihre Grenzwerte (siehe Beizer 1990).

Nehmen wir zum Beispiel an, ein Eingabefeld akzeptiert einen einstelligen Integer-Wert als Eingabe und nutzt eine Zifferntaste zur Einschränkung der Eingabe, damit nur Integer-Werte als Eingabe möglich sind. Der gültige Bereich reicht von 1 bis einschließlich 5. Es gibt also drei Äquivalenzklassen: ungültig (zu niedrig), gültig, ungültig (zu hoch). Für die gültige Äquivalenzklasse sind die Grenzwerte 1 und 5. Für die ungültige (zu hohe) Klasse ist der Grenzwert 6. Für die ungültige (zu niedrige) Klasse gibt es nur einen Grenzwert, 0, da dies eine Klasse mit nur einem Wert ist.

Im obigen Beispiel identifizieren wir zwei Grenzwerte pro Grenze. Die Grenze zwischen ungültig (zu niedrig) und gültig ergibt die Testwerte 0 und 1. Die Grenze zwischen gültig und ungültig (zu hoch) ergibt die Testwerte 5 und 6. Einige Varianten dieses Verfahrens identifizieren drei Grenzwerte pro Grenze: die Werte vor, auf und gerade über der Grenze. Im vorigen Beispiel würde die Nutzung von drei Grenzwerten zu den niedrigen Testwerten 0, 1 und 2 und zu den höheren Testwerten 4, 5 und 6 führen (siehe Jorgensen 2014).

Das Verhalten an den Grenzen der Äquivalenzklasse ist wahrscheinlich eher nicht so korrekt wie das Verhalten innerhalb der Klassen. Es ist wichtig, sich zu vergegenwärtigen, dass sowohl die spezifizierten als auch die implementierten Grenzen über oder unterhalb ihrer eigentlich vorzusehenden Position verschoben, ganz ausgelassen oder mit unerwünschten zusätzlichen Grenzen ergänzt sein können. Grenzwertanalysen und -tests decken fast alle diese Fehlerzustände auf, indem sie die Software provozieren, ein anderes Verhalten zu zeigen, als es für die Klasse zu erwarten wäre, zu der die Grenzwerte gehören.

Grenzwertanalysen können auf allen Teststufen angewandt werden. Dieses Verfahren wird üblicherweise dazu genutzt, Anforderungen zu testen, die eine Reihe von Zahlen (einschließlich Datum und Zeit) enthalten. Grenzwertüberdeckung für eine Klasse wird gemessen als die Anzahl der getesteten Grenzwerte dividiert durch die Gesamtzahl der identifizierten Grenzwerte – meist ausgedrückt als Prozentzahl.

4.2.3 Entscheidungstabellentests

Entscheidungstabellen sind sinnvoll, um komplexe Regeln in Geschäftsprozessen zu erfassen, die ein System umzusetzen hat. Bei der Erstellung der Entscheidungstabellen identifiziert der Tester Bedingungen (zumeist Eingaben) und die daraus folgenden Aktionen (zumeist Ausgaben) des Systems. Diese bilden die Zeilen einer Tabelle, üblicherweise dann auch mit den Bedingungen oben und den Aktionen unten. Jede Spalte bezieht sich auf eine Entscheidungsregel, bestehend aus einer spezifischen Kombination von Bedingungen und den damit verbundenen Aktionen. Die Werte der Bedingungen und Aktionen werden üblicherweise als boolesche Werte dargestellt (wahr oder falsch) oder als eigenständige Werte (z. B. rot, grün, blau), können aber auch Zahlen oder Zahlenreihen sein. Derlei unterschiedliche Arten von Bedingungen und Aktionen können auch zusammen in der gleichen Tabelle auftreten.

Die gängige Notation in Entscheidungstabellen sieht wie folgt aus:

Für Bedingungen:

- J bedeutet, die Bedingung ist richtig (kann auch als Ja oder 1 dargestellt werden).
- N bedeutet, die Bedingung ist falsch (kann auch als Nein oder 0 bezeichnet werden).
- – bedeutet, der Wert der Bedingung hat für die Aktionen keine Bedeutung (kann auch als N/A bezeichnet werden).

Für Aktionen:

- X bedeutet, die Aktion sollte stattfinden (kann auch als Ja oder J oder 1 bezeichnet werden).
- Leer bedeutet, die Aktion sollte nicht stattfinden (kann auch als – oder Nein oder N oder 0 bezeichnet werden).

Eine vollständige Entscheidungstabelle hat genau so viele Spalten (Testfälle), dass jede Kombination von Bedingungen abgedeckt ist. Durch das Löschen von Spalten mit Bedingungskombinationen, die das Ergebnis nicht beeinflussen, kann sich die Anzahl der Testfälle erheblich verringern. Zum Beispiel durch das Entfernen von unmöglichen Kombinationen von Bedingungen. Für weitere Informationen dazu, wie Entscheidungstabellen optimiert werden können, siehe ISTQB®-ATA).

Der übliche Mindestüberdeckungsstandard für Entscheidungstabellentests besteht darin, pro Entscheidungsregel der Tabelle mindestens einen Testfall zu haben. Das beinhaltet üblicherweise das Abdecken aller Kombinationen von Bedingungen. Die Überdeckung wird gemessen als die Anzahl der Entscheidungsregeln, die durch wenigstens einen Testfall getestet wurden, dividiert durch die Gesamtzahl der Entscheidungsregeln – üblicherweise als Prozentzahl dargestellt.

Die Stärke von Entscheidungstabellentests besteht darin, dass sie dabei helfen, alle wichtigen Kombinationen von Bedingungen zu identifizieren, die ansonsten leicht übersehen werden könnten. Sie helfen auch, Lücken in den Anforderungen zu finden. Sie können in jeder Teststufe auf alle Situationen angewendet werden, in denen das Verhalten der Software von einer Kombination von Bedingungen abhängt.

4.2.4 Zustandsübergangstest

Komponenten oder Systeme können unterschiedlich auf ein Ereignis reagieren, abhängig vom gegenwärtigen Zustand oder von Abläufen in der Vergangenheit (z. B. den Vorgängen, die seit Inbetriebnahme des Systems stattgefunden haben). Die Abläufe in der Vergangenheit können durch das Modell von Zuständen abgebildet werden. Ein Zustandsübergangsdiagramm zeigt zum einen alle möglichen Softwarezustände selbst und dann auch, wie die Software in diesen Zustand eintritt, austritt und zwischen den Zuständen wechselt. Ein Übergang wird durch ein Ereignis hervorgerufen (z. B. Benutzereingabe eines Werts in ein Feld). Das Ereignis führt zu einem Übergang. Das gleiche Ereignis kann zu zwei oder mehreren Übergängen aus dem gleichen Zustand herausführen. Die Zustandsänderung kann häufig Reaktionen der Software zur Folge haben (z. B. Ausgabe einer Berechnung oder einer Fehlermeldung).

Eine Zustandsübergangstabelle zeigt alle gültigen Übergänge und auch alle ungültigen, aber möglichen Übergänge zwischen Zuständen auf. Außerdem enthält sie alle Ereignisse und zu erwartende Reaktionen der Software für gültige Übergänge. Zustandsübergangsdiagramme zeigen üblicherweise nur die gültigen Übergänge und schließen ungültigen Übergänge aus.

Es werden Tests entworfen, um eine übliche Sequenz von Zuständen abzubilden, um alle Zustände herbeizuführen, um jeden Übergang auszuführen, um spezifische Sequenzen von Übergängen auszuführen oder schließlich, um ungültige Übergänge zu testen.

Zustandsübergangstest werden für menügeführte Anwendungen genutzt und häufig in der Softwarebranche für eingebettete Systeme (embedded systems) eingesetzt. Das Verfahren ist auch gut für die Modellerstellung eines Geschäftsszenarios mit spezifischen Zuständen geeignet oder für das

Testen der Bildschirmnavigation. Das Konzept eines Zustands ist abstrakt – es kann sich dabei um einige Zeilen Code oder gar um einen gesamten Geschäftsprozess handeln.

Die Überdeckung wird üblicherweise gemessen als die Anzahl der ermittelten Zustände oder Übergänge, die getestet wurden, dividiert durch die Gesamtzahl der ermittelten Zustände oder Übergänge im Testobjekt überhaupt – üblicherweise als Prozentzahl dargestellt. Für weitere Informationen zu Überdeckungskriterien für Zustandsübergangstest siehe ISTQB®-ATA.

4.2.5 Anwendungsfallbasierter Test

Anwendungsfälle (Use Cases) sind eine spezielle Form, um Interaktionen mit Softwareelementen darzustellen. Sie enthalten Anforderungen an die Softwarefunktionalität. Anwendungsfälle haben Akteure (menschliche Benutzer, externe Hardware oder andere Komponenten oder Systeme) und Objekte (eine Komponente oder das System, auf das der Anwendungsfall angewendet wird).

Jeder Anwendungsfall definiert ein bestimmtes Verhalten, das ein Objekt in Zusammenarbeit mit einem oder mehreren Akteuren ausführen kann (UML 2.5.1 2017). Ein Anwendungsfall kann durch Interaktionen und Aktivitäten sowie durch Vorbedingungen, Nachbedingungen und falls angemessen auch in natürlicher Sprache beschrieben werden. Interaktionen zwischen den Akteuren und dem Objekt können zu Veränderungen des Objektzustands führen. Interaktionen können grafisch durch Workflows, Aktivitätsdiagramme oder Geschäftsprozessmodelle dargestellt werden.

Ein Anwendungsfall besteht aus mehreren möglichen Varianten seines grundlegenden Verhaltens, was u. a. Sonder- und Fehlerbehandlungen einschließt (Antwort- und Wiederherstellungsmechanismen des Systems nach Programmier-, Anwendungs- und Kommunikationsfehler, die z. B. zu Fehlermeldungen führen). Tests werden entworfen, um das definierte Verhalten nachzuweisen (grundlegendes, außergewöhnliches oder alternatives Verhalten und die Fehlerbehandlungsroutinen). Die Überdeckung kann durch den Anteil der getesteten Anwendungsfallvarianten bezogen auf die Gesamtzahl der Anwendungsfallvarianten gemessen werden – und wird üblicherweise als Prozentsatz dargestellt.

Für weitere Informationen zu den Überdeckungskriterien für anwendungsfallbasierte Tests siehe ISTQB®-ATA.

4.3 White-Box-Testverfahren

White-Box-Tests basieren auf der internen Struktur des Testobjekts. White-Box-Testverfahren können in allen Teststufen genutzt werden, aber die beiden codebasierten Verfahren, die in diesem Abschnitt beschrieben sind, werden am häufigsten in der Komponententeststufe genutzt. Es gibt weiterführende Verfahren, die zur genaueren Überdeckung in einigen sicherheitskritischen, einsatzkritischen oder hochintegrierten Umgebungen eingesetzt werden, aber diese werden hier nicht weiter behandelt. Für weitere Informationen zu diesen Verfahren siehe ISTQB®-TTA.

4.3.1 Anweisungstests und -überdeckung

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.

4.3.2 Entscheidungstest und -überdeckung

Der Entscheidungstest untersucht die Entscheidungen im Code und testet 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 zum nachfolgend beschriebenen modifizierten Bedingungs-/Entscheidungstest betrachtet der Entscheidungstest nur die gesamte Entscheidung als Ganzes und evaluiert die Entscheidung in zwei 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.

4.3.3 Der Beitrag von Anweisungs- und Entscheidungstests

100% Anweisungsüberdeckung bedeutet, dass alle potenziell ausführbaren Anweisungen im Code auch mindestens einmal ausgeführt wurden. Es stellt aber nicht sicher, dass die komplette Entscheidungslogik getestet wurde. Von den zwei in diesem Lehrplan beschriebenen White-Box-Verfahren kann der Anweisungstest weniger Überdeckung als der Entscheidungstest erzielen.

Wenn 100% Entscheidungsüberdeckung erzielt wird, führt dies alle Entscheidungsergebnisse aus, inklusive eines Tests des „wahr“-Ergebnisses und auch des „falsch“-Ergebnisses, selbst wenn es keine ausdrückliche Anweisung für den „falsch“-Fall gibt (z. B. für den Fall, dass eine IF-Anweisung keine ELSE-Anweisung im Code enthält). Anweisungsüberdeckung hilft Fehlerzustände im Code zu finden, der durch andere Tests nicht ausgeführt wurden. Entscheidungsüberdeckung hilft Fehlerzustände im Code zu finden, für die andere Tests nicht alle "wahr"/"falsch"-Fälle ausgeführt haben.

Das Erzielen von 100% Entscheidungsüberdeckung garantiert 100% Anweisungsüberdeckung (aber nicht umgekehrt).

4.3.4 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.

4.4 Erfahrungsbasierte Testverfahren (nicht prüfungsrelevant)

Bei der Anwendung von erfahrungsbasierten Testverfahren werden die Testfälle aus den Kenntnissen und der Intuition des Testers heraus entwickelt sowie aus seiner Erfahrung mit ähnlichen Anwendungen und Technologien. Diese Verfahren können Testfälle identifizieren, die durch andere systematischere Verfahren nicht so leicht zu identifizieren wären. Abhängig vom Ansatz und der Erfahrung des Testers können diese Verfahren stark unterschiedliche Überdeckungen und Effizienz erreichen. Die Überdeckung ist bei diesen Verfahren schwer zu beurteilen und möglicherweise nicht messbar.

Gängige erfahrungsbasierte Testverfahren werden in den nächsten Abschnitten beschrieben.

4.4.1 Intuitive Testfallermittlung (nicht prüfungsrelevant)

Intuitive Testfallermittlung ist ein Verfahren, das das Auftreten von Fehlhandlungen, Fehlerzuständen und Fehlerwirkungen aufgrund des Wissens des Testers vermutet, u. a.:

- *Wie hat die Anwendung früher funktioniert?*
- *Welche Arten von Fehlhandlungen werden üblicherweise gemacht?*
- *Fehlerwirkungen, die in anderen Anwendungen aufgetreten sind*

Ein methodischer Ansatz für das Verfahren der intuitiven Testfallermittlung ist das Erstellen einer Liste möglicher Fehlhandlungen, Fehlerzustände und Fehlerwirkungen, zu der anschließend Tests entworfen werden, die die erwarteten Fehlerwirkungen und die Fehlerzustände offenlegen. Diese Fehlhandlungs-, Fehlerzustands- und Fehlerwirkungslisten können aufgrund von Erfahrungen oder Informationen über Fehlerzustands- und Fehlerwirkungen oder auch aufgrund allgemeiner Kenntnis darüber, warum Software fehlschlägt, erstellt werden.

4.4.2 Exploratives Testen (nicht prüfungsrelevant)

In explorativen Tests werden informelle (nicht vordefinierte) Tests während der Testdurchführung dynamisch entworfen, ausgeführt, aufgezeichnet und ausgewertet. Die Testergebnisse werden genutzt, um mehr über die Komponente oder das System zu erfahren und um Tests für die Bereiche zu vertiefen, die mehr Tests erfordern.

Exploratives Testen wird manchmal mit Hilfe von sitzungsbasiertem Testen (session-based testing) durchgeführt, um die Ausführung zu strukturieren. Beim sitzungsbasierten Testen werden explorative Tests innerhalb eines definierten Zeitfensters durchgeführt und der Tester nutzt eine Test-Charta mit Testzielen, die ihn beim Testen anleiten. Der Tester kann Sitzungsblätter (Session-Sheets) verwenden, um die ausgeführten Schritte und die Ergebnisse zu dokumentieren.

Exploratives Testen ist dort am nützlichsten, wo es wenig oder ungenügende Spezifikationen oder einen besonderen Zeitdruck für das Testen gibt. Exploratives Testen ist ebenso nützlich, um andere formalere Testverfahren zu ergänzen.

Exploratives Testen steht im starken Zusammenhang mit reaktiven Testverfahren (siehe [ISTQB_FL_SYL – Abschnitt 5.2.2 Teststrategie und Testvorgehensweise]). Exploratives Testen kann die Nutzung anderer Black-Box-, White-Box- oder erfahrungsbasierter Verfahren beim Testen mit einbeziehen.

4.4.3 Checklistenbasiertes Testen (nicht prüfungsrelevant)

Beim checklistenbasierten Testen entwerfen, realisieren und führen Tester Tests mit dem Ziel durch, alle Testbedingungen aus einer Checkliste abzudecken. Als Teil der Analyse erstellen Tester eine neue Checkliste oder erweitern die bestehende Checkliste. Tester können aber auch eine bestehende Checkliste ohne Änderungen verwenden. Solche Checklisten können auf Grundlage von Erfahrungen,

Wissen über das Wesentliche in der Nutzung oder einem guten Verständnis darüber, warum und wie Software fehlschlägt, erstellt werden.

Checklisten können erstellt werden, um verschiedene Testarten zu unterstützen, einschließlich funktionaler und nicht-funktionaler Tests. Beim Fehlen detaillierter Testfälle können checklistenbasierte Tests eine gute Hilfestellung geben und ein bestimmtes Maß an Konsistenz liefern. Da es sich hier um Listen auf einer eher allgemeineren Ebene handelt, ist es wahrscheinlich, dass eine gewisse Variabilität beim eigentlichen Testen auftritt. Diese Variabilität führt einerseits zu einer potenziell höheren Überdeckung, aber andererseits zu einer geringeren Wiederholbarkeit.

5. Literaturhinweise

5.1 Normen/Standards

ISO/IEC/IEEE 29119-1 (2013) Software and systems engineering - Software testing - Part 1: Concepts and definitions

ISO/IEC/IEEE 29119-2 (2013) Software and systems engineering - Software testing - Part 2: Test processes

ISO/IEC/IEEE 29119-3 (2013) Software and systems engineering - Software testing - Part 3: Test documentation

ISO/IEC/IEEE 29119-4 (2015) Software and systems engineering - Software testing - Part 4: Test techniques

ISO/IEC 25010, (2011) Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) System and software quality models

ISO/IEC 20246: (2017) Software and systems engineering — Work product reviews

UML 2.5, Unified Modeling Language Reference Manual, http://www.omg.org/spec/UML/2.5.1/, 2017

[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. [ISTQB_FL_SYL - 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

5.2 ISTQB®-Dokumente

[ISTQB_GLOSSARY] Glossary of Terms used in Software Testing, Version 3.2, 2019

[ISTQB_FL_SYL] Foundation Level Syllabus, Version 2018 V3.1

[ISTQB_FL_OVW] Foundation Level Overview 2018

[ISTQB_FLAT_SYL] Foundation Level Agile Tester Syllabus, Version 2014

[ISTQB_FLPT_SYL] Foundation Level Performance Testing Syllabus, Version 2018

[ISTQB_FLMBT_SYL] Foundation Level Model-Based Testing Syllabus, Version 2015

[ISTQB_FLMAT_SYL] Foundation Level Mobile Application Testing Syllabus, 2019

[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_AL_OVW] Advanced Level TA & TTA Overview 2019

[ISTQB_ALTA_SYL] Advanced Level Test Analyst Syllabus, Version 2019

[ISTQB_ATTA_SYL] Advanced Level Technical Test Analyst Syllabus, Version 2019

[ISTQB_ALTM_SYL] Advanced Level Test Manager Syllabus, Version 2012

[ISTQB_ELTM_SYL] Expert Level Test Management Syllabus, Version 2011

[ISTQB_EITP_SYL] Expert Level Improving the Test Process Syllabus, Version 2011

5.3 Bücher und Artikel

Beizer, B. (1990): Software Testing Techniques (2e), Van Nostrand Reinhold, Boston MA

Black, R. (2009): Managing the Testing Process (3e), John Wiley & Sons, New York NY

Black, R. (2017): Agile Testing Foundations, BCS Learning & Development Ltd, Swindon UK

Buwalda, H. et al. (2001): Integrated Test Design and Automation, Addison Wesley, Reading MA

Copeland, L. (2004): A Practitioner's Guide to Software Test Design, Artech House, Norwood MA

Craig, R.; Jaskiel, S. (2002): Systematic Software Testing, Artech House, Norwood MA

Crispin, L.; Gregory, J. (2008): Agile Testing, Pearson Education, Boston MA

Fewster, M.; Graham, D. (1999): Software Test Automation, Addison Wesley, Harlow UK

Gilb, T.; Graham, D. (1993): Software Inspection, Addison Wesley, Reading MA

Graham, D.; Fewster, M. (2012): Experiences of Test Automation, Pearson Education, Boston MA

Gregory, J.; Crispin, L. (2015): More Agile Testing, Pearson Education, Boston MA

Jorgensen, P. (2014): Software Testing, A Craftsman's Approach (4e), CRC Press, Boca Raton FL

Kaner, C.; Bach, J.; Pettichord, B. (2002): Lessons Learned in Software Testing, John Wiley & Sons, New York NY

Kaner, C.; Padmanabhan, S.; Hoffman, D. (2013): The Domain Testing Workbook, Context-Driven Press, New York NY

Kramer, A.; Legeard, B. (2016): Model-Based Testing Essentials: Guide to the ISTQB® Certified Model- Based Tester: Foundation Level, John Wiley & Sons, New York NY

Myers, G. (2011): The Art of Software Testing (3e), John Wiley & Sons, New York NY

Sauer, C. (2000): “The Effectiveness of Software Development Technical Reviews: A Behaviorally Motivated Program of Research,” IEEE Transactions on Software Engineering, Volume 26, Issue 1, pp 1-14

Shull, F.; Rus, I.; Basili, V. (2000): “How Perspective-Based Reading can Improve Requirement Inspections.” IEEE Computer, Volume 33, Issue 7, pp 73-79

van Veenendaal, E. (ed.) (2004): The Testing Practitioner (Chapters 8 - 10), UTN Publishers, The Netherlands

Weinberg, G. (2008): Perfect Software and Other Illusions about Testing, Dorset House, New York NY

Wiegers, K. (2002): Peer Reviews in Software, Pearson Education, Boston MA

[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

5.4 Deutschsprachige Bücher und Artikel (in diesem Lehrplan nicht direkt referenziert)

Bath, G.; McKay, J.; Gronau, V. (Übersetzung) (2015): Praxiswissen Softwaretest – Test Analyst und Technical Test Analyst Aus- und Weiterbildung zum Certified Tester – Advanced Level nach ISTQB®-Standard, 3., überarbeitete Auflage, dpunkt.verlag, Heidelberg

Bucsics, Th.; Baumgartner, M.; Seidl, R.; Gwihs, St. (2015): Basiswissen Testautomatisierung; Konzepte, Methoden und Techniken, 2., aktualisierte und überarbeitete Auflage, dpunkt.verlag, Heidelberg

Hendrickson, E. (2014): Explore It! Wie Softwareentwickler und Tester mit explorativem Testen Risiken reduzieren und Fehler aufdecken (Aus dem Amerikanischen übersetzt von Meike Mertsch), dpunkt.verlag, Heidelberg

Liggesmeyer, P. (2009): Software-Qualität, Spektrum-Verlag, Heidelberg, Berlin

Linz, T. (2016): Testen in Scrum-Projekten Leitfaden für Softwarequalität in der agilen Welt. Aus- und Weiterbildung zum ISTQB® Certified Agile Tester – Foundation Extension, 2., aktualisierte und überarbeitete Auflage, dpunkt.verlag, Heidelberg

Rössler, Peter; Schlich, Maud; Kneuper, Ralf (2013): Reviews in der System- und Softwareentwicklung: Grundlagen, Praxis, kontinuierliche Verbesserung, 1. Auflage, dpunkt.verlag, Heidelberg

Sneed, Harry M.; Baumgartner, Manfred; Seidl, Richard (2011): Der Systemtest – Von den Anforderungen zum Qualitätsnachweis, 3., aktualisierte und erweiterte Auflage, Carl Hanser Verlag, München

Spillner, A.; Linz, T. (2019): Basiswissen Softwaretest:, Aus- und Weiterbildung zum Certified Tester - Foundation Level nach ISTQB®-Standard, 6., überarbeitete u. aktualisierte Auflage, dpunkt.verlag, Heidelberg

Spillner, A.; Breymann, U. (2016): Lean Testing für C++-Programmierer – angemessen statt aufwendig testen, dpunkt.verlag, Heidelberg

Winter, Mario; Ekssir-Monfared, Mohsen; Sneed, Harry M.; Seidl, Richard; Borner, Lars (2012): Der Integrationstest – Von Entwurf und Architektur zur Komponenten- und Systemintegration, Carl Hanser Verlag, München

Winter, M.; Roßner, Th.; Brandes, Ch.; Götz, H. (2016): Basiswissen modellbasierter Test, Aus- und Weiterbildung zum ISTQB® Foundation Level – Certified Model-Based Tester, 2., vollständig überarbeitete und aktualisierte Auflage, dpunkt.verlag, Heidelberg

5.5 Andere Quellen (in diesem Lehrplan nicht direkt referenziert)

Black, R.; van Veenendaal, E.; Graham, D. (2019): Foundations of Software Testing: ISTQB® Certification (4e), Cengage Learning, London UK

Hetzel, W. (1993): Complete Guide to Software Testing (2e), QED Information Sciences, Wellesley MA

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-4] https://www.owasp.org/index.php/Category:OWASP_Top_Ten_Project

Kapitel 4: [Web-1] [Web-4]

6. Anhang

6.1 Übersicht Lernziele

		TF4D-BO1	TF4D-BO2	TF4D-BO3	TF4D-BO4	TF4D-BO5	TF4D-BO6	TF4D-BO7	Zeit TF4D	Zeit TF4D& Opt	Vorgaben für Trainer/Anbieter/Akkreditierung/Prüfungen Leitfaden für Praktische Kompetenzstufen
									800	855	
Kapitel 1	Grundlagen des Testens									110	
Keywords CTFL:	Debugging; Fehlerwirkung; Fehlerzustand; Fehlhandlung; Grundursache; Qualität; Qualitätssicherung; Verfolgbarkeit; Testablauf; Testanalyse; Testbasis; Testbedingung; Testdurchführung; Testdaten; Testen; Testentwurf; Testfall; Testobjekt; Testorakel; Testprozess; Testrealisierung; Testsuite; Testziel; Überdeckung; Validierung; Verifizierung;										
Keywords TTA:											
1.1 Was ist Testen ?											übernehmen
FL-1.1.1	(K1) Typische Ziele des Testens identifizieren können		x						5		übernehmen
FL-1.1.2	(K2) Testen von Debugging unterscheiden können	x							15		übernehmen
1.2 Warum ist Testen notwendig ?											
FL-1.2.1	(K2) Beispiele dafür geben können, warum Testen notwendig ist	x							15		übernehmen
FL-1.2.2	(K2) Die Beziehung zwischen Testen und Qualitätssicherung beschreiben können und Beispiele dafür geben können, wie Testen zu höherer Qualität beiträgt		x						15		übernehmen
FL-1.2.3	(K2) Zwischen Fehlhandlung, Fehlerzustand und Fehlerwirkung unterscheiden können	x					x		15		übernehmen
FL-1.2.4	(K2) Zwischen der Grundursache eines Fehlerzustands und seinen Auswirkungen unterscheiden können		x				x		15		übernehmen
1.3 Sieben Grundsätze des Testens											
FL-1.3.1	(K2) Die sieben Grundsätze des Softwaretestens erklären können		x						15		übernehmen Beispiele passend zu den unteren Teststufen wählen (ggf. neue Beispiele erstellen)
1.4 Testprozess										15	Das Kapitel 1.4 in 15 - 20 Minuten übersichtsmäßig abhandeln. Keine Prüfungsfragen.
FL-1.4.1	(K2) Die Auswirkungen des Kontexts auf den Testprozess erklären können										Kurze Einführung: Faktoren eingrenzen auf die relevanten Teststufen
FL-1.4.2	(K2) Die Testaktivitäten und zugehörigen Aufgaben innerhalb des Testprozesses beschreiben können										Der Fokus liegt auf den Aktivitäten Analyse, Entwurf, Realisierung und Durchführung, die iterativ durchgeführt werden. Die anderen Aktivitäten werden innerhalb der Entwicklungsaktivitäten durchgeführt.
FL-1.4.3	(K2) Die Arbeitsergebnisse unterscheiden können, die den Testprozess unterstützen										Beschränken auf die Artefakte, die nicht Teil der Entwicklungsartefakte sind und daher auf anderem Wege erstellt werden
FL-1.4.4	(K2) Die Bedeutung der Pflege der Verfolgbarkeit zwischen Testbasis und Testarbeitsergebnissen erklären können										Kann kurz erläutert werden als Erweiterung der Traceability zwischen Testbasis und Entwicklungsartefakten.
Kapitel 2	Testen im Softwareentwicklungslebenszyklus									70	
Keywords CTFL:	änderungsbezogenes Testen; Auswirkungsanalyse; Fehlernachtest; funktionaler Test; Integrationstest; Komponentenintegrationstest; Komponententest; nicht-funktionaler Test; Regressionstest; Testart; Testbasis; Testfall; Testobjekt; Teststufe; Testumgebung; Testziel; Wartungstest; White-Box-Test;										
Keywords TTA:											

2.1 Teststufen										10	Kürzen auf 10 Minuten um eine Übersicht zu vermitteln Keine Prüfungsfragen
FL-2.2.1	(K2) Die unterschiedlichen Teststufen unter den Aspekten der Testziele, Testbasis Testobjekte, typischen Fehlerzustände und Fehlerwirkungen sowie der Testvorgehensweise und Verantwortlichkeiten vergleichen können										Z.B. Tabelle -> 1 Folie/Flipchart 1. Achse Teststufen 2. Achse typische (Ziele, Testbasis, Testobjekte, Fehler, Verantwortlichkeiten)
2.2 Testarten											
FL-2.3.1	(K2) Funktionale, nicht-funktionale und White-Box-Tests vergleichen können	x		x	x				15		übernehmen
FL-2.3.3	(K2) Den Zweck von Fehlernachtests und Regressionstests vergleichen können	x					x		15		übernehmen
2.3 Wartungstest											
FL-2.4.1	(K2) Auslöser für Wartungstests zusammenfassen können		x						15		übernehmen
FL-2.4.2	(K2) Den Einsatz der Auswirkungsanalyse im Wartungstest beschreiben können		x						15		übernehmen
Kapitel 3	Statischer Test									225	
Keywords CTFL:	Ad-hoc-Review; checklistenbasiertes Review; dynamischer Test; perspektivisches Lesen; Review; rollenbasiertes Review; statische Analyse; statischer Test; szenariobasiertes Review;										
Keywords TTA:	Datenflussanalyse; Definition-Verwendungs-Paar; Kontrollflussanalyse; paarweiser Integrationstest; statische Analyse; Umgebungsintegrationstest; zyklomatische Komplexität;										
3.1 Grundlagen des statischen Tests										15	Kernpunkte auf 15 Minuten reduziert. Keine Prüfungsfragen
FL-3.1.1	(K1) Arten von Softwarearbeitsergebnissen erkennen können, die durch die verschiedenen statischen Testverfahren geprüft werden können										Die Beispiele reduzieren. (Code, Architekturen und Modelle)
FL-3.1.2	(K2) Beispiele nennen können, um den Wert des statischen Tests zu beschreiben										Kapitel 3.1.2 ohne den Abschnitt zusätzliche Vorteile
FL-3.1.3	(K2) Den Unterschied zwischen statischen und dynamischen Verfahren unter Berücksichtigung der Ziele, der zu identifizierenden Fehlerzustände und der Rolle dieser Verfahren innerhalb des Softwarelebenszyklus erklären können										Ersetzen durch TTA Kapitel 3.1
3.2 Reviewverfahren anwenden											
FL-3.2.4	(K3) Ein Reviewverfahren auf ein Arbeitsergebnis anwenden können, um Fehlerzustände zu finden					H2		x	60		übernehmen
HO-3.2.4	(H2) Ein Stück Code anhand einer Checkliste reviewen. Die Befunde dokumentieren.							x			Eine typische Checkliste für ein Codereview mit verschiedenen Anomalien bereit stellen. Ein Stück Code bereit stellen, das verschiedene Anomalien aus der Checkliste enthält. Den Teilnehmern ist eine Vorlage für eine Befundliste zur Verfügung zu stellen um dort die Befunde zu dokumentieren. Die Befundliste wird mit den Teilnehmern durchgesprochen.
3.3 Statische Analyse											

TTA-3.2.1	(K3) Die Kontrollflussanalyse anwenden, um zu ermitteln, ob der Programmcode Anomalien im Kontrollfluss aufweist					H1		x	60		übernehmen
HO-3.2.1	(H1) Auf ein Stück Code ein statisches Analysewerkzeug anwenden, um typische Kontrollflussanomalien zu finden. Den Bericht des Werkzeugs verstehen und wie die Anomalien die Qualitätsmerkmale des Produkts beeinträchtigen.										Ein oder mehrere Stücke Code bereit stellen, die syntaktisch korrekt sind, und verschiedene Arten von Kontrollflussanomalien enthalten, die im Syllabus erwähnt werden. Die Teilnehmer bei der Ausführung des statischen Analysewerkzeugs und beim Anzeigen des Berichts über die Anomalien anleiten. Die Teilnehmer sollen die gefundenen Fehlerzustände besprechen und die betroffenen Qualitätsmerkmale benennen (funktionale Korrektheit, Wartbarkeit, IT-Sicherheit etc.).
TTA-3.2.2	(K2) Erklären, wie die Datenflussanalyse verwendet wird, um zu ermitteln, ob der Programmcode Datenflussanomalien aufweist					H0		x	15		übernehmen; Acthung: LO wird sich in CTAL-TTA v4.0 von K2 auf K3 ändern!
HO-3.2.2	(H1) Für ein Stück Code den Bericht eines statischen Analysewerkzeug betreffend Datenflussanomalien verstehen und wie die Anomalien funktionale Korrektheit und Wartbarkeit beeinträchtigen.										Ein Stück Code bereit stellen, das syntaktisch korrekt ist, welches bei einigen Variablen die wesentlichen Arten von Datenflussanomalien, welche im Syllabus erwähnt werden, enthält. Führen Sie das statische Analysewerkzeug aus, erklären Sie den Teilnehmern die gefundenen Datenflussanomalien, und erörtern Sie mit ihnen die Auswirkungen auf die funktionale Korrektheit bzw. auf die Wartbarkeit.
TTA-3.2.3	(K3) Möglichkeiten vorschlagen, wie die Wartbarkeit von Programmcode durch statische Analyse verbessert werden kann					H2		x	60		übernehmen
HO-3.2.3	(H2) Für ein Stück Code, das gegen gegebene Programmierkonventionen und -richtlinien verstößt, die Wartbarkeitsmängel beheben, welche vom statischen Codeanalysewerkzeug berichtet wurden. Anschließend durch Nachtest bestätigen dass sie korrigiert sind, und verifizieren dass keine neuen Mängel eingeführt wurden.										Ein Dokument mit Programmierkonventionen und -richtlinien den üblichen Arten von Anforderungen aus dem Syllabus zur Verfügung stellen. Ein Stück Code zur Verfügung stellen, das syntaktisch korrekt ist, und Verstöße gegen die Programmkonventionen und -richtlinien enthält. Ein statisches Analysewerkzeug ausführen, und den Teilnehmern den Bericht über die Verstöße zur Verfügung stellen. Die Teilnehmer sollen die Wartbarkeitsfehler auf Basis der Programmierkonventionen und -richtlinien und den Hinweisen des statischen Analysewerkzeugs korrigieren. Sie sollen die statische Analyse nachtesten und bestätigen, dass die Fehlerzustände korrigiert und keine neuen hinzugefügt worden sind.
TTA-3.2.4	(K2) Den Einsatz von Aufrufgraphen für die Bestimmung von Teststrategien für den Integrationstest erklären							x	15		übernehmen; Acthung: LO wird sich in CTAL-TTA v4.0 ändern!
Kapitel 4	Testverfahren									450	
Keywords CTFL:	Anweisungsüberdeckung; Anwendungsfallbasierter Test; Äquivalenzklassenbildung; Black-Box-Testverfahren; Entscheidungstabellentest; Entscheidungsüberdeckung; erfahrungsbasiertes Testverfahren; Grenzwertanalyse; Testverfahren; Überdeckung; White-Box-Testverfahren; Zustandsübergangstest;										
Keywords TTA:	Anweisungstest; atomare Bedingung; Entscheidungstest; Mehrfachbedingungstest; modifizierter Bedingungs-/Entscheidungstest; verkürzte Auswertung; White-Box-Testverfahren;										
4.1 Kategorien von Testverfahren											
FL-4.1.1	(K2) Die Eigenschaften, Gemeinsamkeiten und Unterschiede zwischen Black-Box-Testverfahren, White-Box-Testverfahren und erfahrungsbasierten Testverfahren erklären können	x		x	x				15		übernehmen
4.2 Black-Box-Testverfahren											
FL-4.2.1	(K3) Die Äquivalenzklassenbildung anwenden können, um Testfälle aus vorgegebenen Anforderungen abzuleiten	x		x		H2			60		übernehmen

HO-4.2.1	(H2) Für ein vorhandenes Spezifikationselement ist, unter Anwendung der Äquivalenzklassenbildung, eine Testsuite zu spezifizieren und zu implementieren. Die Testsuite ist mit der zugehörigen Software auszuführen.										Ein Spezifikationselement und die zugehörige Software als Testobjekt zur Verfügung stellen. Das Testobjekt soll Fehlerzustände enthalten, die mit Hilfe von Äquivalenzklassenbildung aufgedeckt werden können. Die Teilnehmer sollen die Testfälle spezifizieren, implementieren, ausführen, und sicherstellen, dass alle Äquivalenzklassen abgedeckt sind. Falls nicht, sollen sie Testfälle hinzufügen bis das Ziel erreicht ist und alle Fehlerzustände entdeckt wurden. Optional: Die Fehlerzustände korrigieren und die Testfälle erneut durchführen, um nachzuweisen, dass die Fehlerzustände korrigiert und keine neuen hinzugefügt worden sind. Das Beispiel soll zu Testfällen für mindestens 2 gültige und mindestens 1 ungültige Äquivalenzklasse führen.
FL-4.2.2	(K3) Die Grenzwertanalyse anwenden können, um Testfälle aus vorgegebenen Anforderungen abzuleiten	x		x		H2			60		übernehmen
HO-4.2.2	(H2) Für ein vorhandenes Spezifikationselement ist, unter Anwendung der Grenzwertanalyse, eine Testsuite zu spezifizieren und zu implementieren. Die Testsuite ist mit der entsprechenden Software auszuführen.										Ein Spezifikationselement und die zugehörige Software als Testobjekt zur Verfügung stellen. Das Testobjekt soll Fehlerzustände enthalten, die mit Hilfe von Grenzwertanalyse entdeckt werden können. Die Teilnehmer sollen die Testfälle spezifizieren, implementieren, ausführen, und sicherstellen, dass alle Grenzwerte abgedeckt sind. Falls nicht, sollen sie Testfälle hinzufügen bis das Ziel erreicht ist und alle Fehlerzustände entdeckt wurden. Optional: Die Fehlerzustände korrigieren und die Testfälle erneut durchführen, um nachzuweisen, dass die Fehlerzustände korrigiert und keine neuen hinzugefügt worden sind. Das Beispiel soll zu Testfällen für mindestens 4 Grenzwerte (2 Grenzen) führen.
FL-4.2.3	(K3) Entscheidungstabellentests anwenden können, um Testfälle aus vorgegebenen Anforderungen abzuleiten	x		x		H2			60		übernehmen
HO-4.2.3	(H2) Für ein vorhandenes Spezifikationselement ist, unter Anwendung von Entscheidungstabellentests, eine Testsuite zu spezifizieren und zu implementieren. Die Testsuite ist mit der entsprechenden Software auszuführen.										Ein Spezifikationselement und die zugehörige Software als Testobjekt zur Verfügung stellen. Das Testobjekt soll Fehlerzustände enthalten, die mit Hilfe von Entscheidungstabellentests entdeckt werden können. Die Teilnehmer sollen die Testfälle spezifizieren, implementieren, ausführen, und sicherstellen, dass alle Tabelleneinträge abgedeckt sind. Falls nicht, sollen sie Testfälle hinzufügen bis das Ziel erreicht ist und alle Fehlerzustände entdeckt wurden. Optional: Die Fehlerzustände korrigieren und die Testfälle erneut durchführen, um nachzuweisen, dass die Fehlerzustände korrigiert und keine neuen hinzugefügt worden sind. Die Tabelle im Beispiel soll mindestens 3 Bedingungen enthalten.
FL-4.2.4	(K3) Zustandsübergangstest anwenden können, um Testfälle aus vorgegebenen Anforderungen abzuleiten	x		x		H2			60		übernehmen

HO-4.2.4	(H2) Für ein vorhandenes Spezifikationselement ist, unter Anwendung von Zustandsübergangstests, eine Testsuite zu spezifizieren und zu implementieren. Die Testsuite ist mit der entsprechenden Software auszuführen.										Ein Stück Code und die zugehörige Softwarekomponentenspezifikation zur Verfügung stellen. Einige ausführbare Anweisungen sollen Fehlerzustände enthalten, die mit Hilfe von Zustandsübergangstests entdeckt werden können (z.B. Zustandsautomat). Die Teilnehmer sollen die Testfälle spezifizieren, implementieren, ausführen, und sicherstellen, dass alle Zustandsübergänge abgedeckt sind. Falls nicht, sollen sie Testfälle hinzufügen bis das Ziel erreicht ist und alle Fehlerzustände entdeckt wurden. Das Beispiel soll zu mindestens 5 Testfällen führen.
FL-4.2.5	(K2) Erklären können, wie man Testfälle aus einem Anwendungsfall ableitet	x		x					15		übernehmen
4.3 White-Box-Testverfahren											
4.3.1 Anweisungstest											
FL-4.3.1	(K2) Anweisungsüberdeckung erklären können	x			x				15		LOs miteinander kombinieren, aufgrund von Redundanz ist der Text des FL LOs nicht im Lehrplan enthalten.
TTA-2.2.1	(K3) Den Anweisungstest anwenden, um Testfälle für ein bestimmtes Spezifikationselement zu erstellen, die eine definierte Überdeckung erzielen	x			x	H2			30		
HO-2.2.1	(H2) Für ein gegebenes Spezifikationselement und ein entsprechendes Stück Code eine Testsuite mit dem Ziel einer 100%igen Anweisungsüberdeckung entwerfen und implementieren, und nach der Ausführung verifizieren, dass das Testziel erreicht wurde.										Ein Stück Code und die entsprechende Softwarekomponentenspezifikation zur Verfügung stellen. Einige ausführbare Anweisungen sollen Fehlerzustände enthalten, welche durch Anweisungsüberdeckung gefunden werden können. Die Teilnehmer sollten die Testfälle spezifizieren, implementieren und ausführen, und sicherstellen, dass 100%ige Anweisungsüberdeckung erreicht ist. Falls nicht, sollten sie weitere Testfälle hinzufügen bis das Testziel erreicht ist und alle Fehlerzustände entdeckt wurden. Das Beispiel soll zu mindestens 3 Testfällen führen.
4.3.2 Entscheidungstest											
FL-4.3.2	(K2) Entscheidungsüberdeckung erklären können	x			x				15		LOs miteinander kombinieren, aufgrund von Redundanz ist der Text des FL LOs nicht im Lehrplan enthalten.
TTA-2.3.1	(K3) Den Entscheidungstest anwenden, um Testfälle für ein bestimmtes Spezifikationselement zu erstellen, die eine definierte Überdeckung erzielen	x			x	H2			30		
HO-2.3.1	(H2) Für ein gegebenes Spezifikationselement und ein entsprechendes Stück Code eine Testsuite mit dem Ziel einer 100%igen Entscheidungsüberdeckung entwerfen und implementieren, und nach der Ausführung verifizieren, dass das Testziel erreicht wurde.										Ein Stück Code und die entsprechende Softwarekomponentenspezifikation zur Verfügung stellen. Einige Entscheidungen oder Ausführungspfade sollen Fehlerzustände enthalten, welche durch Entscheidungsüberdeckung, aber nicht unbedingt durch Anweisungsüberdeckung gefunden werden können. Die Teilnehmer sollen die Testfälle spezifizieren, implementieren und ausführen, und verifizieren dass 100%ige Entscheidungsüberdeckung erreicht ist. Falls nicht, sollten sie weitere Testfälle hinzufügen bis das Testziel erreicht ist und alle Fehlerzustände entdeckt wurden. Das Beispiel soll zu mindestens 3 Testfällen führen. Es soll den Vorteil der Entscheidungsüberdeckung gegenüber der Anweisungsüberdeckung zeigen.

4.3.3 Der Beitrag von Anweisungs- und Entscheidungstests											
FL-4.3.3	(K2) Die Bedeutung von Anweisungs- und Entscheidungsüberdeckung erklären können	x			x				15		übernehmen
4.3.4 Modifizierte Bedingungs-/Entscheidungsabdeckung (MC/DC)											
TTA-2.4.1	(K3) Den Modifizierten Bedingungs-/Entscheidungstest anwenden, um Testfälle für ein bestimmtes Spezifikationselement zu erstellen, die eine definierte Überdeckung erzielen	x			x	H2			60		übernehmen
HO-2.4.1	(H2) Für ein gegebenes Spezifikationselement und ein entsprechendes Stück Code, das eine Entscheidung mit mehreren atomaren Bedingungen enthält, eine Testsuite mit dem Ziel einer 100%igen modifizierten Bedingungs-/Entscheidungsüberdeckung entwerfen, implementieren und ausführen.										Ein Stück Code, der eine Entscheidung mit mehreren, voneinander unabhängigen, atomaren Bedingungen enthält, und die zugehörige Softwareentwicklungsspezifikation zur Verfügung stellen. Die Entscheidung im Code soll einen Fehlerzustand enthalten, der durch modifizierte Bedingungs-/Entscheidungsüberdeckung gefunden werden kann. Die Teilnehmer sollen die Testfälle spezifizieren, implementieren und ausführen, und verifizieren, dass 100%ige modifizierte Bedingungs-/ Entscheidungsüberdeckung erreicht ist. Das Beispiel soll zu Testfällen für mindestens 3 atomare Bedingungen führen. Es soll die Vorteile gegenüber dem Entscheidungstest zeigen.
4.4 Erfahrungsbasierte Testverfahren										15	Kernpunkte auf 15 Minuten reduziert. Keine Prüfungsfragen
FL-4.4.1	(K2) Die intuitive Testfallermittlung erklären können										Gemeinsame Eigenschaften der erfahrungsbasierten Testtechniken sind: Testbedingungen, Testfälle und Testdaten, die aus einer Testgrundlage gewonnen wurden, die die Kenntnisse und Erfahrungen der Tester, Entwickler, Anwender und anderer Interessengruppen berücksichtigen. Diese Kenntnisse und Erfahrungen enthalten auch die erwartete Anwendung der Software, ihre Umgebung, voraussichtliche Mängel und die Verteilung dieser Mängel.
FL-4.4.2	(K2) Exploratives Testen erklären können										siehe Anmerkungen zu FL-4.4.1
FL-4.4.3	(K2) Checklistenbasiertes Testen erklären können										siehe Anmerkungen zu FL-4.4.1