www.architecture4testability.com
www.imbus.de/veranstaltungen/trends-in-testing-2015/

Architektur und Testbarkeit: Eine Checkliste (nicht nur) für Softwarearchitekten – Teil 1

Der Begriff „Testbarkeit" ist in der Praxis oft schwer zu konkretisieren. In dieser Hinsicht ähnelt er der „Qualität" – man kümmert sich häufig erst um sie, wenn sie fehlt. Greift man diese Konkretisierung aber an, stellt man fest, dass die meisten Testbarkeitsanforderungen an die Architektur-Kollegen im Projektteam adressiert sind. Dieser zweiteilige Beitrag sammelt typische Testbarkeitsthemen und erläutert in Form einer Checkliste mit „Dos & Don'ts", wie sie seitens der Architektur jeweils frühzeitig berücksichtigt werden können.

> „Lieber Architekt: Frage nicht, was der Test für Dein Projekt tun kann – frage, was Du mit Deiner Architektur für den Test tun kannst!"
> *(frei nach John F. Kennedy)*

Wenn im Anfangsstadium eines IT-Projekts Architektur-Diskussionen geführt und erste Entscheidungen gefällt werden, ist nach unserer Erfahrung häufig kein Testspezialist daran beteiligt. Das liegt typischerweise daran, dass das Testen im Projektplan als „späte Phase" aufgefasst wird und die geplanten Tester-Rollen zu Projektbeginn noch nicht besetzt sind (Indikator: „Jetzt müssen wir erst einmal kräftig entwickeln, damit es für die Tester überhaupt etwas zu tun gibt!"). Oder aber es liegt daran, dass in den Köpfen noch ein Silo-Denken vorherrscht, das dafür sorgt, dass Tester bei Architektur-Meetings schlicht nicht zu den Eingeladenen gehören.

Die Konsequenzen sind vorgezeichnet: Wenn die Tester mit ihren Konzeptions- und Vorbereitungsarbeiten beginnen, müssen sie feststellen, dass die für ihre Teststrategie benötigte Testbarkeit leider nicht oder nicht ausreichend gegeben ist. Als die grundlegenden Weichenstellungen zur Architektur erfolgten, war einfach kein Anwalt für die Testbarkeitsthemen mit von der Partie.

Wenn man Architektur – frei nach Martin Fowler (vgl. [Fow02]) – als diejenigen Design-Entscheidungen definiert, deren Änderungen die höchsten Kosten verursachen würden, ist klar, was eine solche Situation für Test und *Qualitätssicherung (QS)* bedeutet: Der Zug ist oft schon abgefahren. Die Testbarkeitswünsche lassen sich nicht mehr nachträglich berücksichtigen, da dies zu aufwändig wäre – und dem Test bleibt nur noch übrig darzulegen, welche Risiken durch fehlende Tests oder welche zusätzlichen Testkosten durch Workarounds entstehen. Beides ist sicher nicht optimal.

Eine weitere Rolle könnte dabei spielen, dass der Begriff „Testbarkeit" nicht konkret genug definiert ist. Darin ähnelt er dem Begriff „Qualität", den messbar zu formulieren in jedem Projekt eine Herausforderung darstellt – gerade im Vergleich zu „in time" und „in budget", die im Vergleich zu „in quality" höchst trivial zu messen sind. Beide Begriffe werden intensiv verwendet, aber wenn man nachfragt, was *genau* darunter verstanden wird, erlebt man, dass sie nicht konkreter beschrieben werden können – und wenn, dann höchstens anhand einzelner Beispiele –, aber dass es jedem sofort auffällt, wenn sie fehlen. Da Testbarkeit als nicht-funktionales Qualitätsmerkmal für Softwaresysteme gilt, ist diese Parallele aber letztlich wenig überraschend. Und nicht zuletzt: Beide IT-Domänen haben nur geringe Überschneidungen. Dies manifestiert sich in unterschiedlichen Begriffswelten, Methoden und Artefakten. Welcher Architekt kennt zum Beispiel die Testbegriffe nach ISTQB-Standard (vgl. [IST15])? Welcher Tester kann bei Architektur-Diskussionen über Patterns oder lose Kopplung aktiv mitreden? Im Grunde reden wir hier von zwei Welten, die sich in der Regel nur sehr zaghaft berühren und austauschen – und zu allem Überfluss auch noch unterschiedliche Sprachen sprechen.

Zwischen zwei Welten vermitteln

Die Frage lautet also: Was dagegen tun? Mit diesem Artikel möchten wir den Dialog zwischen den Disziplinen Architektur und Test gewissermaßen vorwegnehmen, indem wir:

1. Als Vorbereitung den Begriff „Testbarkeit" schärfen.
2. Typische Testbarkeitsanforderungen versammeln und beschreiben.
3. Zu jeder Anforderung mögliche Architekturmaßnahmen diskutieren.

Ziel ist es, auf diese Weise die *frühzeitige* Berücksichtigung von Testbarkeitsanforderungen in Architekturentscheidungen zu fördern. Das Ergebnis soll nicht nur Architekten als Checkliste dienen, sondern auch Testmanagern, Testautomatisierern und Testern in agilen Teams helfen, diesen dringend nötigen Dialog mit den Architektur-Kollegen frühestmöglich zu führen. (Übrigens war genau das das Hauptergebnis der Folge 24 des ArchitekTour-Podcasts (vgl. [Hei10]): „Liebe Architekten, liebe Tester – redet mehr miteinander.")

Bei der Erarbeitung unserer Ergebnisse waren wir uns bewusst, dass es eine Architekturbewegung namens *Design for Testability (DFT)* gibt. Allerdings mussten wir feststellen, dass die dort vorgeschlagenen Prinzipien sehr wenig verbreitet sind. Außerdem konzentriert sich DFT hauptsächlich auf entwicklernahe Testaktivitäten wie Unit-Tests und weniger auf die Testdurchführung auf höhere Stufen (vgl. [Jun02], [Zil12]).

Vorbereitungen

Im Folgenden verwenden wir sämtliche Testbegriffe wie im ISTQB-Glossar. Wenn wir von der dort formulierten Definition abweichen oder sie verfeinern bzw. konkretisieren, werden wir ausdrücklich darauf hinweisen.

Je nach Entwicklungsprozess und Integrationsstrategie kommen typischerweise verschiedene so genannte *Teststufen* zum Einsatz, wie beispielsweise:

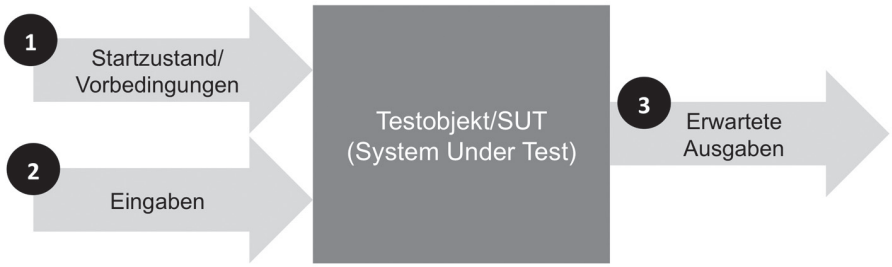

Abb. 1: Genereller Aufbau eines Testfalls.

- Komponententests
- Integrationstests
- Systemtests
- Abnahmetests
- Systemintegrationstests

Diese unterscheiden sich hinsichtlich der Testziele (welche Anforderungen sollen geprüft werden, welche Testabdeckungen werden angestrebt, welche Arten von Fehlern sollen gefunden werden?), der *Testumgebung (TU)*, der Testmethoden, aber vor allem im jeweiligen Testobjekt (Synonym: *System under Test, SUT*). Das kann eine einzelne Klassenoperation sein, ein Verbund von Komponenten mit den zugehörigen Schnittstellen, das fertige Gesamtsystem/Endprodukt oder gar der Verbund aus Endprodukt mit anzubindenden Partner-Systemen.
Was auf jeder Teststufe einheitlich ist, ist der Begriff des *Testfalls*: Dieser verfolgt ein gewisses Testziel und besteht aus mindestens drei Teilen (**siehe Abbildung 1**):

- Angabe des SUT und dessen benötigtem *Startzustand* sowie gegebenenfalls weiterer Vorbedingungen.
- Den *Eingaben* ins SUT.
- Den *erwarteten Ausgaben* (und gegebenenfalls Nachbedingungen, in der Abbildung nicht dargestellt).

Der definierte Startzustand des Testobjekts ist für die Reproduzierbarkeit von Tests entscheidend und sollte – auch für manuell durchzuführende Tests – idealerweise immer automatisiert herstellbar sein.
Der Begriff *Testbarkeit* wird im Allgemeinen definiert als „Fähigkeit eines Software-Produkts für einen Test nach einer Änderung“ (vgl. [IST15]). Dabei unterscheidet man zwischen:

- *Fachlicher Testbarkeit:* Hat der Testdesigner alle Informationen, die er zur Testfall-Erstellung benötigt, in aktueller und korrekter Form?
- *Technischer Testbarkeit:* Sind die designten Testfälle überhaupt ausführbar? Können z. B. die verlangten Vorbedingungen überhaupt technisch realisiert werden?

In unserem Beitrag geht es folglich ausschließlich um die *technische Testbarkeit* – die fachliche Testbarkeit muss im Allgemeinen mit anderen Projektdisziplinen wie Anforderungs-, Risiko- und Fehlermanagement abgestimmt werden.
Für unsere Zwecke ist all dies nicht konkret genug, deshalb ziehen wir zwei weitere ISTQB-Begriffe heran:

- *Points of Control (PoC)* sind Schnittstellen, die es dem Test erlauben, Startzustände herzustellen, Vorbedingungen zu erzeugen (z. B. ein anderes Datum als das heutige) und Eingaben im SUT vorzunehmen.
- *Points of Observation (PoO)* sind Schnittstellen, die es erlauben, die Reaktionen des SUT zu beobachten und gegebenenfalls Nachbedingungen zu prüfen – etwa, dass ein Datensatz nach Testdurchführung wieder zur Bearbeitung freigegeben ist.

Mit diesen beiden Begriffen können wir konkrete Testbarkeitsanforderungen kategorisieren und beschreiben, indem wir typische PoCs und PoOs versammeln und aus Architektursicht diskutieren. Folglich definieren wir in diesem Artikel „technische Testbarkeit“ als „Realisierungsgrad aller PoCs und PoOs, die zur Erreichung der jeweiligen Testziele benötigt werden“.
Da PoC und PoO jeweils Schnittstellen sind, ist klar, warum die Ansprechpartner für die Tester bei den Architekten zu finden sind. Aufgabe der Architekten sollte es – neben vielen anderen Aufgaben – unseres Erachtens also sein, die Testbarkeitsanforderungen frühzeitig einzuholen und in den Architekturentscheidungen angemessen zu berücksichtigen. Übrigens gelten die meisten Testbarkeitsanforderungen für alle Teststufen, wenn auch unterschiedlich hinsichtlich Bedeutung und Realisierungsaufwand.
Betrachten wir die bisher versammelten Begriffe aus der Architekturperspektive, ergeben sich erste nützliche Abgrenzungen, die in **Abbildung 2** dargestellt sind. Auf

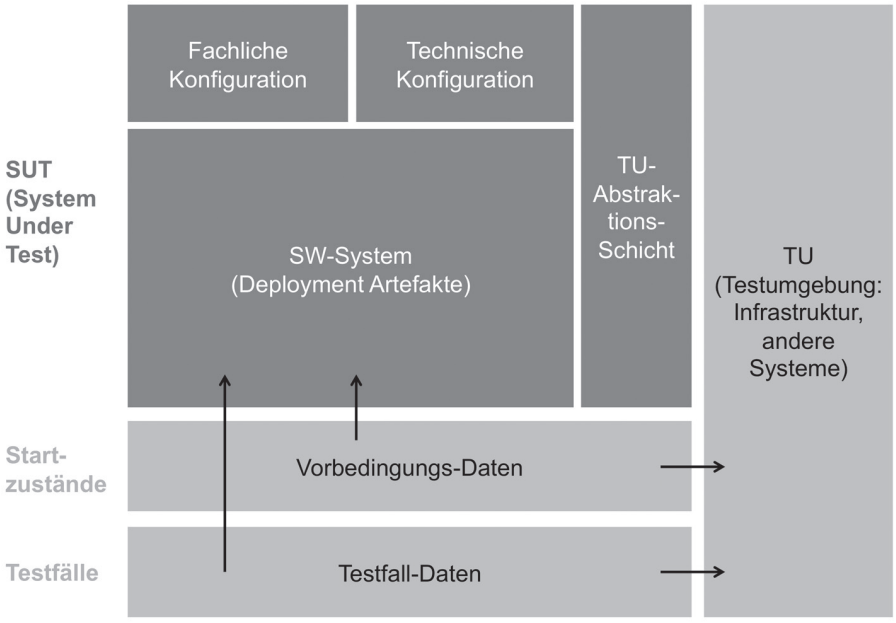

Abb. 2: Testbegriffe aus der Architektur-Perspektive.

der linken Seite sehen wir drei horizontale Schichten – von oben nach unten:

- *Das SUT* als lauffähiges (aber leeres) Testobjekt – bestehend aus Deployment-Artefakten, einer zugehörigen fachlichen Konfiguration (enthält Basisdaten, wie z. B. ein PLZ-Verzeichnis, das zur Ausführung herangezogen und benötigt wird), einer technischen Konfiguration (mit Konfigurationsdaten, wie z. B. Pfadangaben) und – bei Bedarf – einer Abstraktionsschicht zur Testumgebung, die sich um Themen wie Schnittstellen zu Umsystemen, Datum/Uhrzeit und Testkennungen/Berechtigungen kümmert.
- *Die Vorbedingungsdaten*, die vor einer Testdurchführung ins SUT eingespielt werden, um die benötigten Startzustände zu realisieren.
- *Die Testfalldaten*, also die Eingaben und erwarteten Ausgaben, die jeder Testfall selbst mitbringt.

Die Testumgebung auf der rechten Seite des Bildes enthält neben der Infrastruktur auch Softwarekomponenten, wie z. B. (externe) Partnersysteme.
Nebenbei haben wir damit den Begriff *Testdaten* weiter ausdifferenziert in

- Konfigurationsdaten
- Vorbedingungsdaten
- Testfalldaten

Mögliche Quellen für Vorbedingungsdaten sind der vom Projekt gepflegte Bestand an Vorbedingungsdaten (testzielgetrieben und von mehreren Testfällen wiederverwendbar) oder einzelne importierte Datenobjekte, z. B. aus dem Produktivdatenbestand zur Nachstellung von Fehlersituationen.
Die Festlegung der TU-Abstraktionsschicht ist in vielen Architekturen zumindest teilweise etabliert, z. B. die Abstraktion von der Datenbank oder vom Dateisystem. Zur Erreichung eines besseren Testbarkeitsgrads des SUT erscheint es uns aber sinnvoll, weitere Aspekte in dieser Abstraktionsschicht zu berücksichtigen. Beispiele hierfür sind:

- Der Zugriff auf die Partnersysteme durch die Einführung einer flexiblen Architektur, die den Austausch realer Schnittstellen durch Mock-up-Implementierungen besonders gut unterstützt.
- Die Einführung spezieller Einheiten für den Zugriff auf die Systemfunktionalität wie Datums- und Uhrzeitabfrage, Rechte- und Rollenverzeichnis usw., um diese Parameter für Testzwecke verstellen zu können.

Anhand dieses Schaubildes (**siehe Abbildung 2**) können wir nun verschiedene typische Testbarkeitsanforderungen beschreiben und diskutieren. Die im Folgenden vorgestellten Testbarkeitsanforderungen sind meistens unabhängig sowohl von den für die Entwicklung eingesetzten Technologien oder Frameworks, als auch von der Art des Systems (wie z. B. Embedded, Web- oder Desktop-Applikation). Im Gegensatz dazu sind die Strategien für die Architektur zur Erfüllung dieser Anforderungen im großen Maße abhängig von der gewählten Grobarchitektur und von der Art des Systems. Im Folgenden illustrieren wir die Vorschläge zur Umsetzung der Architekturempfehlungen jeweils beispielhaft basierend auf JEE-Architekturen.

Startzustand und Vorbedingungen herstellen können

Tests – sowohl neue Tests, Regressionstests, als auch Fehlernachtests – müssen reproduzierbar gestaltet werden. Dazu gehört die zuverlässige, möglichst automatisierte Wiederherstellung des Startzustands, da anderenfalls gleiche Eingaben ins SUT zu unterschiedlichen Reaktionen bzw. Testergebnissen führen können.

1. Startzustand im SUT erzeugen
Auf welchen Wegen aber kann der Test die benötigten Systemzustände provozieren und als Teststartpunkt nutzen? Diejenigen Systemzustände, die einem Normalablauf entsprechen, können meistens leicht erreicht werden. Andere wiederum benötigen spezielle Eingriffe, wie z. B.: Ein Protokollservice fällt aus und ist nicht verfügbar; Ergebnis: Systembedienung ist nicht möglich, Datenbank nicht erreichbar; zu testen: für den Benutzer erscheint ein leicht verständlicher Hinweis, ohne technische Details offenzulegen.
Wenn diese Zustände nicht nur vom SUT, sondern auch von der Umgebung abhängig sind, können sie meistens nur durch die Manipulation der Systemkonfiguration erreicht werden, was die Automatisierung und Wiederholbarkeit zusätzlich erschwert. Die Reproduzierbarkeit von Softwaretests (sowohl für Regressions- als auch für Fehlernachtests) hängt wesentlich von der Fähigkeit der Testumgebung ab, das Testobjekt in einen definierten Ausgangszustand zu versetzen. Dieser Prozess besteht in der Regel aus zwei wesentlichen Schritten:

- Versetzen des Systems in den „ausführbaren" Initialzustand (Normalfall)
- Versetzen des Systems in einen für den Test gewünschten Sonderzustand

Die Herstellung des *Initialzustands* umfasst alle Schichten des Testobjekts, inklusive der zum Betrieb notwendigen Infrastruktur (z. B. Konfiguration von Applikationsservern) und der Datenhaltung. Zudem muss sichergestellt werden, dass keine Seiteneffekte aus vorherigen/parallelen Testläufen oder auf zukünftige Testläufe möglich sind („Gedächtnis").
Da sich Umgebungskonfigurationen gemeinsam mit dem Testobjekt weiterentwickeln, ist eine solche reproduzierbare Kombination aus Softwaresystem (Deployment-Artefakt) und seiner Umgebung nur möglich, wenn möglichst viele der Umgebungs-Artefakte zusammen mit dem Testobjekt versioniert und automatisiert ausführbar/bereitstellbar sind („Infrastructure as Code", z. B. Konfigurationsskripte, Setup-Skripte, Einspielen benötigter Datensätze, Kopiervorlagen der Zielumgebung).
Eine besondere Herausforderung im Test ist die versionierte Verwaltung der Datenhaltung, also der Datenbank-Schemata und der jeweils zugehörigen (Test-)Datenbestände. Bei jedem Deployment einer Version des Testobjekts in die Testumgebung ist das Datenschema zusammen mit den hinterlegten Inhalten auf die gleiche Version zu migrieren. Erst diese gemeinsame Versionierung von Testobjekt, Infrastruktur und Datenhaltung ermöglicht eine so umfassende Automatisierung des Deployments, dass verschiedene Versionen eines Testobjekts reproduzierbar in einer einzigen Testumgebung getestet werden können. Anders ausgedrückt: Für die Reproduzierbarkeit müssen sowohl Produkt- als auch Test-Artefakte im benötigten Umfang in ein professionelles Konfigurationsmanagement überführt werden, um sie im jeweils benötigten Stand idealerweise auf Knopfdruck bereitstellen zu können.
Der für den Test gewünschte *Sonderzustand* ist ein Systemzustand, der jede mögliche Konstellation der Teilsystem-Zustände darstellen kann. Für den Test ist es dabei essenziell zu wissen, welche definierten Zustände das System grundsätzlich durchlaufen kann und welche funktionalen und nicht-funktionalen Anforderungen für die-

se Zustände gelten. Damit hier zwischen Architektur und Test Transparenz geschaffen werden kann, muss ein Zustandsmodell des Systems erstellt und gepflegt werden. Zustandsmodelle sind natürlich auch für die Architekturherleitung und -dokumentation ein wichtiges Artefakt und erleichtern die Kommunikation zwischen Stakeholdern wesentlich. Bei Bedarf wird der Test dieses definierte Zustandsmodell um feinere Zustände anreichern und diese als Startzustand für Tests nutzen wollen. Beispiele hierfür sind:

- Fehlen wesentlicher Systemkomponenten oder Umsysteme, wie z. B. die Datenbank.
- Datenvalidierungsdienste oder LDAP-Verzeichnis.
- System überlastet bzw. Abarbeitung der Aufträge im vorgesehenen Zeitraum nicht möglich usw.

Der zweite Schritt, um Testbarkeit zu realisieren, sind die Definition und Einführung von Mechanismen, mit denen das System in diese definierten Zustände versetzt werden kann. Die vollständige Automatisierung des Bereitstellungsprozesses hilft, Fehler bei der Bereitstellung der Testumgebung zu vermeiden und den Prozess mit geringem Aufwand auszuführen. Die Aufgabe des Architekten ist die Schaffung der Voraussetzungen für die Wiederholbarkeit des Prozesses ohne Nebenwirkungen in allen vorgesehenen Umgebungen. Was nicht verschwiegen werden soll: In vielen Fällen ist das nur durch die Manipulation in der TU-Abstraktionsschicht möglich (**siehe Abbildung 2**).
Langfristig kann dieses Ziel unter anderem durch folgende Maßnahmen gewährleistet werden:

- Nachvollziehbare Dokumentation und Kommunikation der Richtlinien.
- Code-Reviews mit dem Ziel, Abweichungen von den Richtlinien zu erkennen.
- Automatisierte Codeanalyse mit dem Ziel, Umgehungslösungen im Code aufzudecken.
- Vollautomatisierung des Bereitstellungsprozesses.

2. Einzelnen Datensatz einspielen
Die gezielte Versorgung von Testumgebungen mit Testdaten (vor allem Vorbedingungsdaten) erfordert es, einzelne Datensätze (fachliche Datenobjekte, z. B. „Kunde 4711“) in das SUT einspielen zu können. Dies wird in der Praxis beispielsweise dann nötig, wenn Fehlersituationen aus den Produktivumgebungen zur Fehleranalyse nachgestellt werden müssen, die sich mit vorhandenen Testdaten nicht reproduzieren lassen.
Das Datenschema eines zu entwickelnden Systems kann schnell groß und komplex werden. Die Identifikation und Extraktion eines vollständigen Datensatzes, der sich auf mehrere Tabellen verteilt (z. B. Kunde mit Adresse und Kontoverbindung, Bestellposition und Bestellung mit Kunde, Versand- und Rechnungsadressen und Kontoverbindung) ist damit auch eine entsprechende Herausforderung. Damit auch große Datenschema-Definitionen keine Probleme bereiten – unter anderem bezüglich der Identifikation der zusammengehörenden Daten –, müssen einige Regeln beachtet werden:

- Die Verwaltung des Schemas und der Datensätze ist einfacher, wenn die Datenschema-Definition dem fachlichen Domänenmodell folgt und gut dokumentiert ist.
- Es muss nachvollziehbar sein, welche Datensätze durch das System manipuliert werden und welche für den Startzustand bereitgestellt werden müssen.
- Der Einsatz von technisch motivierten Tabellen, wenn in der Datenbank nur Key-Value-Paare gehalten werden und die Zusammengehörigkeit der Datensätze über Logik im Code berechnet wird, erschwert die Navigation und die Nutzung vieler verbreiteter Werkzeuge.
- Das Datenbankschema muss in sich geschlossen sein – alle Beziehungen müssen im Schema bereits erkennbar sein und nicht erst über den Programmcode hergestellt werden.

Beim Reproduzieren eines Fehlers ist es oft notwendig, die Aktivitäten, die zu einem Fehler geführt haben, am gleichen Datensatz zu wiederholen, weil z. B. dieser Datensatz spezielle Werte beinhaltet, die zu dem Fehler führen. Wenn diese Vorgehensweise automatisiert wird, werden alternative Vorgehensweisen, z. B. das Kopieren von Produktiv-Datenbanken für die Reproduktion der Fehler im Live-System, obsolet – jetzt können per Knopfdruck nur die Datensätze aus der Produktionsdatenbank kopiert werden, die für die Fehlerreproduktion tatsächlich notwendig sind. (Abgesehen davon haben Echtdaten ohnehin nur in gut begründeten Ausnahmefällen etwas im Test verloren.)
Zum Einspielen des kopierten Datensatzes können je nach Format und technischen Möglichkeiten z. B. CSV oder SQL genutzt werden. Frameworks wie „DBUnit“ oder „Liquibase“ bieten aber viel mehr Komfort und Robustheit gegen eventuelle Fehler in der Handhabung. Spezielle Werkzeuge, wie z. B. „Jailer“, ermöglichen die Identifikation und den Export zusammenhängender Datensätze. Mit einer solchen Werkzeugunterstützung ist mit der Komplexität des Datenbankschemas sicherlich etwas einfacher umzugehen, aber die Herausforderungen lassen sich nicht vollkommen beseitigen. Bei größeren Datenbankschema-Definitionen ist es deshalb unumgänglich, das Datenbankschema zu partitionieren. Eine für die optimale Nachvollziehbarkeit praxiserprobte und für fast alle Recherche- und Analyseaktivitäten im Projekt geeignete Partitionierung erfolgt entlang der fachlichen Spezifikation des SUT (**siehe Abbildung 2**).
Neben regelmäßigen Architektur-Reviews sollte man für diesen Themenkreis Maßnahmen vorsehen wie die folgenden:

- Regelmäßiges Review der Datenstruktur auf Einhaltung der Architekturvorgaben
- Vier-Augen-Prinzip/Code-Reviews bei Änderungen der Datenstruktur
- Werkzeuggestützte QS des Schemas

3. Datensätze duplizieren/klonen
Für schnelles Feedback an das Entwicklungsteam müssen Tests parallel durchgeführt werden können, ohne sich gegenseitig zu behindern. Diese Anforderung trifft insbesondere für automatisierte Tests zu, die unbeobachtet (z. B. über Nacht) durchgeführt werden sollen. Dazu ist es häufig nötig, denselben Datensatz in mehrfachen Ausprägungen vorliegen zu haben, getrennt z. B. durch Nummernkreise, und diese zwischen den parallelen Testläufen geeignet aufzuteilen. Diese zu erzeugen und dabei die referenzielle Integrität zu erhalten, ist eine typische Testanforderung. Mechanismen zur Duplikation zu haben, ist aber auch nützlich für die Erzeugung von Massendaten für Last- und Performancetests.
Die Bereitstellung der Testfall- und Vorbedingungs-Daten in der richtigen Konstellation und in ausreichender Menge kann zeit- und ressourcenaufwändig werden. In vielen Fällen ist es ausreichend, einen vollständigen Datensatz zu haben und diesen für ähnliche Testfälle zu duplizieren. Reine Dupli-

kate reichen in den meisten Fällen wegen der Einschränkungen für die fachlichen und technischen Primärschlüssel aber nicht aus. Wenn aber diese Einschränkungen bekannt sind und die Schlüsselausprägungen in definierte Bereiche aufgeteilt werden können (z. B. von 1 bis 999 für Testfall A und von 1.000 bis 1.999 für Testfall B), können die Datensätze in vielen Fällen durch einfache Manipulation der Schlüsselwerte kopiert werden. Damit diese Manipulation aber möglich ist, müssen die Fremdschlüssel-Beziehungen nachvollziehbar und möglichst einfach sein. Außerdem stellt eine feste Semantik in den Primärschlüsseln sehr häufig ein komplexes Problem dar.

Eine klare und sowohl für Entwickler als auch für Tester verständliche Dokumentation der Datenbankschema-Definition kann dabei helfen, dass die wichtigen Felder, die die Beziehungen zu anderen Tabellen herstellen, erkannt und geeignet adressiert werden. Außerdem können vorhandene Werkzeuge zum Testdatenmanagement (vgl. [Tes]) auf diesen wohldefinierten Artefakten aufsetzen und das Projektteam unterstützen.

In vielen Projekten wird die Datenbankschema-Dokumentation zugunsten der Dokumentation des Domänenobjektmodells aufgegeben. Dabei wird vergessen, dass das Ableiten der Datenbankschema-Definition aus dem Domänenobjektmodell auf vielen implizite Annahmen basiert, die meistens nicht dokumentiert, aber auf jeden Fall für einen Tester nur sehr schwer nachvollziehbar sind.

Um die Fähigkeit der Duplikation von Datensätzen langfristig sicherzustellen, sind deshalb Mindestanforderungen für die Dokumentation einzuführen und die beschriebenen Anforderungen an das Datenbankschema mit regelmäßigen Schema-, Code- und Dokumentations-Reviews zu überprüfen.

4. Datum und Uhrzeit kontrollieren/ simulieren

Manche Testergebnisse (Reaktionen des SUT) sind abhängig von der Systemzeit. Typische Beispiele sind Berechnungen zum Jahreswechsel oder solche mit einer umfangreichen fachlichen Vorgeschichte. Viele Tests lassen sich im Aufwand signifikant reduzieren, wenn man – je nach aktuell gegebenem Systemdatum – nicht für jede Durchführung einen geeigneten Datensatz neu finden oder gar neu anlegen muss, sondern dies als Tester nur einmal tut und bei Testwiederholung das Systemdatum kontrolliert auf den gewünschten Wert setzt.

Die Manipulation der Systemzeit kann in vernetzten Umgebungen aber sehr schwierig bis unmöglich werden – vor allem muss das Systemverhalten analysiert werden, wenn die Systemzeit der zu testenden Applikation und der angebundenen Fremdsysteme nicht übereinstimmt.

Für die Manipulation der Systemzeit im SUT ist es am einfachsten sicherzustellen, dass der Zugriff auf die Systemzeit zentralisiert nur an einer Stelle erfolgt. Durch spezielle Erweiterungen des SUT ist es dann möglich, beim Systemstart oder auch zur Laufzeit einen Parameter mit Zeitversatz oder einen neuen Wert mitzugeben. In Applikationen, die z. B. auf JEE-Technologien basieren, ist es am einfachsten, eine Factory-Klasse (nennen wir diese Klasse `DateProvider`) zu definieren, die für die Erstellung und Manipulation aller `Date`- und `Calendar`-Instanzen zuständig ist.

Die Vorgabe zur Nutzung einer solchen `DateProvider`-Klasse (oder eines vergleichbaren Konstrukts) muss gegenüber den Entwicklern klar kommuniziert werden. Außerdem ist es empfehlenswert, eine statische Codeanalyse einzuführen und eine Regel zu definieren, die eine direkte Initialisierung der `Date`- oder `Calendar`-Klassen entdeckt und entsprechend meldet.

5. Testbenutzer mit benötigten Rechten bereitstellen

Manche Testergebnisse sind abhängig von den Berechtigungen des angemeldeten Benutzers. Zu Testzwecken sind typischer-

Testbarkeits-Anforderung	an	Mögliche Architekturmaßnahmen zur Realisierung	Einhaltung	Zu vermeiden
Startzustand im SUT erzeugen	SUT	Zustandsmodell definieren Automatisierte Mechanismen zur Zustandserzeugung	Zustandsmodell pflegen VM für Umgebungs-konfigurationen	Manuelle Bereitstellung von Infrastuktur oder Testdaten
Einzelnen Datensatz einspielen	SUT	Schema partitionieren (keine zirkulären Beziehungen) VM-/KM-Werkzeuge auf Schemata anwenden (z. B. Liquibase) Werkzeuggestützte Extraktion zusammenhängender Datensätze (z. B. Jailer)	Datenstrukturen reviewen Vier-Augen-Prinzip bei Schema-Änderungen Werkzeuggestütze Analyse (z. B. auf zirkuläre Beziehungen)	Fremdschlüssel-Beziehungen, die nur im Code abgebildet werden Technisch motivierte Tabellen, die zu komplexen Referenzen führen
Datensätze duplizieren/ klonen	SUT	Dokumentiertes, in sich konsistentes und abgeschlossenes DB-Schema Einfache, fachlich motivierte Fremdschlüsselbeziehungen	Code- und Schema-Reviews Mindeststandards für Schema-Dokumentation	Komplexe Fremdschlüssel-Beziehungen Berechnete Fremdschlüssel
Datum und Uhrzeit simulieren	TU	Realisierung der TU-Abstraktionsschicht	Statische Code-Analysen (z. B. SonarQube) Architektur- und Code-Reviews	nur Verwendung von Echtzeit Verwendung der DB-Zeit
Benutzer und Rechte konfigurieren	TU	Realisierung der TU-Abstraktionsschicht (z. B. für Mock-Fähigkeit) Steuerung der Rechte über Konfiguration	Code-Analysen Architektur- und Code-Reviews	Ermitteln der Rechte im Code
Fremd-komponenten konfigurieren	TU	Realisierung der TU-Abstraktionsschicht (z. B. für Mock-Fähigkeit) Schichtarchitektur mit Zugriffsregeln Einhaltung der Architektur-Empfehlungen auch in Fremdkomponenten	statische Code-Analysen (z. B. SonarQube) Architektur- und Code-Reviews	Direkte Zugriffe auf Fremd-systeme/Plattform-Funktionalität Fremdsysteme, die sich nicht an die Architektur-Empfehlungen halten

Tabelle 1: Architekturmaßnahmen im Kontext „Startzustand und Vorbedingungen herstellen“.

weise verschiedene Testkennungen in der Testumgebung nötig, die unterschiedliche Rechtekonstellationen abbilden. Diese müssen zudem in ausreichender Zahl zur Verfügung stehen (etwas für die parallele Durchführung von Tests).

Unternehmenssoftware verfügt oft über ein klar definiertes Rechte-/Rollenkonzept für Anwender. Damit kann z. B. der Zugriff auf sensible Daten eingeschränkt werden, entweder indem Abfrageergebnisse entsprechend gefiltert werden oder indem das User-Interface sich abhängig von Rolle und Rechten anpasst. Dadurch erweitert sich aber der Umfang der durchzuführenden Tests.

Die für die Umsetzung eines Rechte-/Rollenkonzepts notwendigen Benutzerdaten werden üblicherweise in einem externen System vorgehalten (Verzeichnisdienst als LDAP oder ActiveDirectory), was in **Abbildung 2** in der Testumgebung einzuordnen ist. In den verschiedenen Teststufen muss die Infrastruktur entsprechend eingerichtet werden. Da die Einbindung und Änderungen an einem Verzeichnisdienst mit einem nicht unerheblichen Aufwand verbunden sind, ist es eine mögliche Vereinfachung, den Verzeichnisdienst nur auf Testumgebungen zu höheren Teststufen einzubinden und auf den vorherigen Teststufen auf einen Mock-Ansatz („Fake-User-Ansatz“) zurückzugreifen. Entscheidend ist dabei die Frage, ob die jeweiligen Testziele durch ein solches Mocking gegebenenfalls gefährdet sind. Die Nutzung einer lokalen Quelle von Benutzerdaten, z. B. einer XML-Datei, unterstützt flexibel das Testen der Anwendung und blendet Seiteneffekte der Benutzerdaten-Infrastruktur, wie z. B. eine zeitverzögerte Aktivierung neuer Benutzerkonten, weitgehend aus.

Die Verwendung echter Benutzerkonten im Test ist aus Sicherheitsgründen zu vermeiden, da auch auf produktionsnahen Testumgebungen der Zugriff auf Produktionsdaten verhindert werden muss. (Außerdem sind auch für automatisierte Tests Konten nötig und Teammitglieder sollten ihre eigenen hierfür nicht bereitstellen müssen.) Abhilfe schafft die Verwendung von Domänen, sodass für jede Testumgebung eigene Benutzerkonten zur Verfügung stehen, die auf der Produktionsdomäne über keinerlei Zugriffsrechte verfügen.

In der Implementierung ist darauf zu achten, dass die Rechteverwaltung möglichst nicht im Code, sondern über eine Konfiguration erfolgt. Besonders kritisch ist die Berechnung der Rechte im Ablauf der Programmausführung zu sehen, zumal wenn diese Funktionalität nicht in separate, leicht austauschbare Module ausgelagert werden kann.

Die Anbindung der Verzeichnisdienste muss soweit eingeschränkt sein, dass nur auf den dafür vorgesehenen Umgebungen ein Zugriff erfolgen kann. Zudem müssen – auch aus datenschutzrechtlichen Gründen – für die Administration der Benutzerkonten Qualitätsstandards aufgestellt und regelmäßig überprüft werden. Damit ist auf den entwicklungsnahen Umgebungen wie vorgesehen nur der Fake-User-Ansatz möglich, während auf den höherwertigen Testumgebungen jeweils die zuständige Benutzer-Domäne eingebunden wird.

6. Fremdkomponenten in der Testumgebung konfigurieren

Alle genannten Testbarkeitsanforderungen gelten häufig auch für externe Systeme – so etwa das gezielte Einspielen von Daten oder das Setzen der Systemzeit.

Je nach Testziel haben Testfälle einen unterschiedlichen Scope: Für einige Tests reicht es, wenn eine einzelne Operation ohne darunterliegenden Code bzw. andere Systeme getestet werden. Für andere ist es essenziell, Zugriff auf die Partnersysteme oder Infrastruktur-Funktionalität zu haben.

Der Test des SUT zusammen mit Infrastruktur und Partnersystemen kann zu Herausforderungen führen, insbesondere wenn diese Infrastruktur und Partnersysteme nicht die Testbarkeitsanforderungen, die wir in diesem Artikel vorstellen, erfüllen. So wird es z. B. nicht ohne Weiteres möglich sein, die Systemzeit in einer Fremddatenbank zu manipulieren oder Datensätze in ein Fremdsystem zu klonen und zu importieren.

Der in **Abbildung 2** dargestellte Aufbau ist auch aus folgendem Grund empfehlenswert: Sollte eines der Partnersysteme oder die Infrastruktur die für einen Test notwendigen Anforderungen nicht erfüllen, kann dies mit vergleichsweise wenig Aufwand durch eine Eigenimplementierung (Mock-up) erreicht werden. Außerdem ist es möglich, in der Abstraktionsschicht Erweiterungen einzubauen, die sowohl das Testen als auch das Monitoring des Systemverhaltens erleichtert.

Die Einhaltung der konzeptionellen Architekturentscheidungen, wie z. B. das Schichtenmodell der Applikation und zulässige Zugriffspfade durch die Schichten, muss aber fortlaufend überprüft werden. Zur Aufdeckung von ungewollten Aufrufen/Zugriffen eignen sich sowohl regelmäßige Code-Reviews als auch spezielle Regeln für die statische Codeanalyse. Die Definition solcher Regeln ist zuweilen eine spezielle Herausforderung, da es nicht immer ein-

Literatur & Links

[Fow02] M. Fowler, Patterns of Enterprise Application Architecture, Addison-Wesley, 2002

[Hei10] Heise Developer, Podcast: SoftwareArchitekTOUR – Podcast für den professionellen Softwarearchitekten, Folge 24: „Testing & Softwarearchitektur“, 2010, siehe: `http://www.heise.de/developer/artikel/Episode-24-Testing-Softwarearchitektur-1080236.html`

[IST15] International Software Testing Qualifications Board (ISTQB), Glossary, 2015, siehe: `http://www.istqb.org/downloads/glossary.html`

[Jun02] S. Jungmayr, Design for Testability, 2002, siehe: `http://www.jungmayr.de/Publikationen/CONQUEST02_jungmayr.pdf`

[Tes] Testtool Review, Toolübersicht zu Testdatenmanagement, siehe: `https://www.testtoolreview.de/de/toolkategorien/item/68-testdatenmanagement`

[Zil12] F. Zilberfeld, InfoQ, Design for Testability – The True Story, 2012, siehe: `http://www.infoq.com/articles/Testability`

deutig erkennbar ist, in welcher Architekturschicht ein Aufruf stattfindet. In solchen Situationen sind feste Namensschemata für die Code-Artefakte eine große Hilfe: Zum Beispiel kann man Klassen für die Datenzugriffschicht an dem Suffix „DAO“ für „Data Access Object“ im Namen erkennen oder an der Zugehörigkeit zu einem speziell dafür vorgesehenen Paket. Wenn diese Regeln definiert und kommuniziert sind, können die unerwünschten Zugriffe, wie z. B. aus einem Datentransfer-Objekt zu einem Frontend-Controller, schnell erkannt werden.

Ausblick auf Teil 2 diesen Artikels

Damit ist der Themenblock „Startzustand und Vorbedingungen herstellen“ abgeschlossen. In Teil 2 des Artikels werden wir die begonnene *Checkliste* (**siehe Tabelle 1**) vervollständigen und darin die ausstehenden drei Themen „Eingaben ins SUT vornehmen“, „Ausgaben des SUT prüfen“ und „Nachbedingungen prüfen“ analog behandeln. Abschließend werden wir das Erreichte bewerten und einen Vorschlag zur Weiterentwicklung der Checkliste durch die Architekten-Community unterbreiten. ||

Die Autoren

|| Dr. Christian Brandes
(christian.brandes@imbus.de)
arbeitet als leitender Berater und Trainer für die imbus AG. Aktuelle Arbeitsthemen sind Testprozess-Analysen, agiles Testen sowie Schnittstellen des Tests zu anderen IT-Disziplinen.

|| Dr. Shota Okujava
(shota.okujava@isento.de)
ist geschäftsführender Gesellschafter der Isento GmbH sowie als Architekt und Berater für Java-Enterprise-Technologien in Großprojekten tätig. Seine Spezialthemen sind SOA, modellgetriebene Softwareentwicklungsprozesse sowie Integrationstechnologien.

|| Dr. Jürgen Baier
(juergen.baier@modellar.de)
ist geschäftsführender Gesellschafter der Modellar GmbH und als agiler Coach, Architekt und Berater im JEE-Umfeld tätig. Neben agilen Entwicklungsframeworks beschäftigen ihn vorrangig CI/CD und die Synchronisation verteilter Datenbestände.