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

Architektur und Testbarkeit: Eine Checkliste (nicht nur) für Softwarearchitekten – Teil 2

Im ersten Teil dieses Artikels, erschienen in der letzten Ausgabe von OBJEKTspektrum, haben wir motiviert, welchen Nutzen eine Checkliste mit konkreten Testbarkeitsanforderungen und zugehörigen Architekturempfehlungen im Projektalltag hat, und den Begriff „(technische) Testbarkeit" geschärft. Zentrale Elemente dabei waren die ISTQB-Begriffe „Point of Control" bzw. „Point of Observation" sowie die allgemeine Testfall-Definition (Abbildung 1). Aus dieser abgeleitet haben wir den ersten Bereich der Checkliste zum Thema „Startzustand und Vorbedingungen herstellen" entwickelt (Tabelle 1). Analog werden wir nun die Bereiche „Eingaben ins SUT vornehmen", „Ausgaben des SUT prüfen" und „Nachbedingungen prüfen" behandeln.

Eingaben ins SUT vornehmen

1. UI-Objekte erkennen und bedienen

Automatisierte System- und Akzeptanztests benutzen meistens die Benutzungsoberfläche (GUI) als Eingabeschnittstelle. Für eine nachhaltige Automatisierung mit kontrollierbaren Wartungskosten ist es unerlässlich, dass die UI-Entwicklung von Anfang an eine robuste Objekterkennung gewährleistet, da anderenfalls die Anpassungsaufwände in der Testautomatisierung rasch überhand nehmen können.

Die Aufgabe des Architekten in diesem Zusammenhang ist einerseits die Prüfung, ob die einzelnen Entwickleraufgaben (z. B. User-Storys oder Use-Cases) so geschnitten und verteilt werden können, dass die Oberflächenstruktur eine gewisse Beständigkeit hat. Andererseits muss der Architekt überprüfen, ob die eingesetzte Technologie von dem Testframework unterstützt wird und die eventuell zur Laufzeit generierten Element-IDs für die gleichen Elemente auch immer gleich bleiben – sich also insbesondere nicht ändern, wenn das Element in der Oberfläche minimal verschoben wird. Viele Technologien, die Oberflächen-Artefakte generieren (z. B. aus einem Objekt- oder Datenmodell), bieten keine Möglichkeit, solche konstanten IDs zu erzeugen.

Sollte die im Projekt definierte Technologie es erlauben, die IDs für die Oberflächenelemente selbst zu definieren, muss im Entwicklungsteam klar kommuniziert werden, dass diese IDs eine besondere Bedeutung haben und nicht ohne Grund geändert werden dürfen. Maßnahmen zur Einhaltung sind: Klare Kommunikation der Auswirkungen der Änderung an die Entwickler, regelmäßige Reviews, statische Code-QS mit der Überprüfung, ob die IDs wie definiert gesetzt sind.

2. Datenbank befüllen

Manche Vorbedingungen oder auch Eingaben ins SUT finden am besten direkt in Datenbank-Tabellen statt (z. B. Flags zur Simulation, etwa von Gesperrt-Zuständen). Viele Architekten finden es ausreichend, wenn eine Persistenzschicht definiert, qualitätsgesichert und ausreichend dokumentiert wird – die Datenbankschema-Definition wird nur noch generiert. Dabei ist es aber wichtig zu beachten, dass der Generator durch mehrere Parameter gesteuert wird und ein Austausch des Generators Änderungen in der Schemadefinition zur Folge haben kann. Um die nötige Stabilität im Projekt zu gewährleisten und die Dokumentation der Datenbankschema-Definition für alle zugänglich und verständlich zu gestalten, sollten Datenbankschemata in einem separaten, abstrakten Modell definiert werden. Hierfür sind sowohl grafische, als auch XML-basierte Modellierungswerkzeuge, wie z. B. „Liquibase", gut geeignet. Durch eine externe Schemadefinition hat der Tester dann die Möglichkeit, auf eine gültige, verständliche Dokumentation zuzugreifen und eigene Skripte und Werkzeuge daran auszurichten. Funktionalität, die in Datenbanksystemen realisiert wird (z. B. Trigger und Stored Procedures), erschwert den Test (nebenbei wird das Datenbanksystem zum SUT deklariert) und kann in meisten Fällen durch Architektur anders angesiedelt werden, vorzugsweise direkt im SUT.

3. Andere Schnittstellen befüllen

Neben GUI und Datenbank kann ein Testobjekt über weitere spezifische Schnittstellen verfügen, in die zu Testzwecken Eingaben vorgenommen werden müssen.

Die flexible Anbindung von externen Schnittstellen ist immer ein wichtiger Aspekt in der Entwicklung. Einerseits ist es unabdingbar, möglichst nah an der fremden Schnittstellenspezifikation zu bleiben, andererseits erzeugen Änderungen in der Implementierung der externen Schnittstellen, sofern diese überhaupt bereits fertiggestellt sind, unnötige Störungen im Verlauf. Aus Architektursicht ist es empfehlenswert, die Fremdschnittstellen nach Möglichkeit zu kapseln und selbst implementierte oder extern bereitgestellte, aber stabile Mock-Ups zu nutzen. Die Nutzung der Mock-Ups muss transparent für den Entwickler erfolgen und lediglich von der Konfiguration abhängig sein – diese Anforderung legt es nahe, dem Gesamtsystem ein Framework zu Grunde zu legen, das über die Konfiguration die Anbindung/Bereitstellung und Nutzung der Module/Schnittstellen erlaubt (vgl. TU-Abstraktionsschicht in Abbildung 2).

Abb. 1: Genereller Aufbau eines Testfalls.

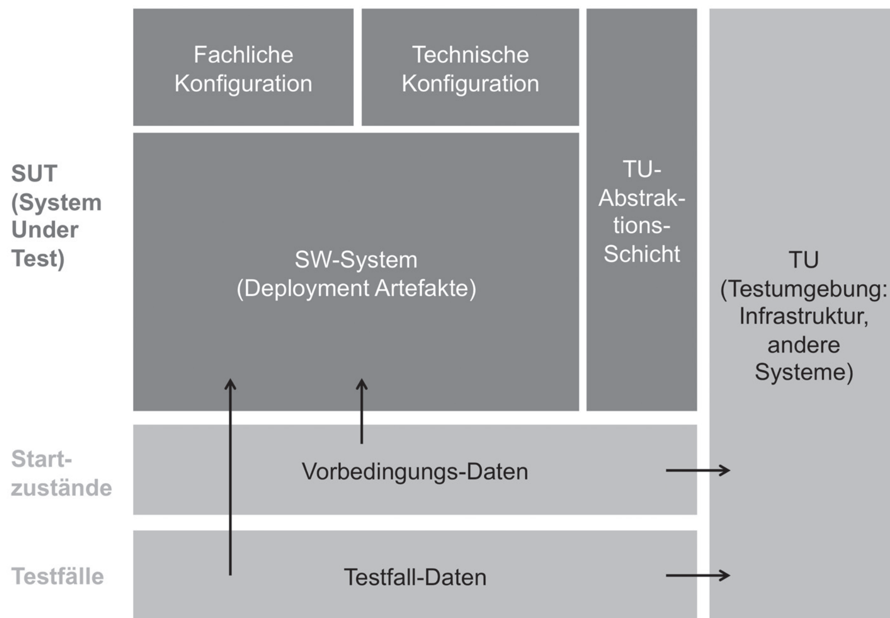

Abb. 2: Testbegriffe aus der Architektur-Perspektive.

Frameworks mit *Dependency-Injection*-Unterstützung und *Inversion of Control* bieten hier hervorragende Unterstützungsmechanismen, insbesondere auch wegen der Automatisierbarkeit der Konfigurationseinstellungen je nach Testumgebung oder Testziel. Dabei muss sichergestellt werden, dass die Entwicklung nicht „ausbricht" und anfängt, fremde Schnittstellen direkt, d.h. unter Umgehung der Abstraktionsschicht, zu nutzen. Dies kann mit relativ wenig Aufwand durch statische Codeanalyse erreicht werden, indem regelmäßig überprüft wird, welche Applikationsmodule welche Pakete importieren bzw. nutzen (siehe Tabelle 2).

Ausgaben des SUT prüfen

1. UI-Objekte erkennen und auslesen

Hier gelten dieselben Architektur-Maßnahmen wie bei „UI-Objekte erkennen und bedienen".

2. Datenbank auslesen

Entwickler sind zuweilen uneinig darüber, ob in Projekten mit Datenbank-Abstraktionsschicht (z.B. JPA in JEE-Applikationen) die Unit-Tests die Datenbankzugriffe mittesten sollten oder nicht. Grundsätzlich gilt, dass die Abstraktionsschicht eine bereits vielfach getestete Komponente darstellt und ihre Funktionalität nicht mehr getestet werden muss. Das bedeutet, dass es ausreicht, beim Persistieren von Daten via Abstraktionsschicht wieder über die Abstraktionsschicht zu überprüfen, ob diese Daten korrekt persistiert wurden. Die alternative Vorgehensweise – Überprüfung der erfolgten Persistierung der Daten direkt in der Datenbank (ohne die Abstraktionsschicht) – wird nicht empfohlen, da dies die Unit-Tests abhängig von der Infrastruktur macht und ihre Portierung mit zusätzlichem Aufwand verbunden ist.

In speziellen Fällen ist es aber unabdingbar, Daten direkt in der Datenbank zu überprüfen – sei es das Testen der Konfiguration der Caching-Mechanismen oder der Konfiguration der Abstraktionsschicht selbst: In jedem Projekt müssen entsprechende Tests definiert sein. Damit die enge Bindung an die Datenbank nicht zum Nachteil wird, wird empfohlen, spezielle Werkzeuge einzusetzen, die den Zugriff zwar ohne die Abstraktionsschicht realisieren, aber eigene Mechanismen anbieten, um unabhängig von den Datenbank-Treibern den Zugriff zu realisieren und die Daten auszulesen.

Zuletzt sollte noch darauf hingewiesen werden, dass die Datenbanktabellen-Struktur und die Klassenstruktur der Persistenzschicht durchaus Unterschiede aufweisen

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".

Testbarkeits-Anforderung	an	Mögliche Architekturmaßnahmen zur Realisierung	Einhaltung	Zu vermeiden
UI-Objekte erkennen und bedienen	SUT	Technologieauswahl Verwendung fachlich eindeutiger und stabiler Identifier Abstimmung mit Testautomatisierern	Code-Analyse, Code-Review	Einsatz von Technologien mit generierten UIS / generierten (dynamischen) IDS
Datenbank befüllen	SUT	Abstrakte und verständliche DB-Schema-Definition Automatisierung der DB-Initialisierung Laufende Pflege der Initialisierungsdaten (RECHTS?)	Architektur-Review Regelmäßige Analyse manueller Tätigkeiten	Ableiten des DB-Schemas aus der Klassenstruktur Manuelle Eingriffe in der DB-Bereitstellung
Andere Schnitt-stellen befüllen	SUT	Realisierung der TU-Abstraktionsschicht	Code-Analysen Architektur- und Code-Reviews	Direkte Zugriffe auf Fremd-systeme/Schnittstellen

Tabelle 2: Architekturmaßnahmen im Kontext „Eingaben ins SUT vornehmen“.

können. So kann die Tabellenstruktur sich ändern, obwohl die Klassenstruktur im engeren Sinne identisch bleibt (z. B. nur durch die Änderung der Konfiguration oder der Annotationen). Solche Sonderfälle sind bei der Gestaltung der Abfragen zu berücksichtigen. Einen guten Funktionsumfang für die Testunterstützung mit Datenbankzugriff bieten Frameworks wie „DBUnit“ im Java-Umfeld und „ndbunit“ in der .NET-Welt.

3. Log- und Protokoll-Dateien auslesen

Manche Testergebnisse sind aus Logfiles auszulesen, da das System hier die zu prüfende Information ablegt.

Eine durchdachte Protokollierung kann nicht nur bei der Fehlersuche im System helfen, sondern somit auch zur Vereinfachung der Testausführung beitragen. In speziellen Situationen können Log-Einträge als Nachweis für die erfolgreiche Ausführung einer Aktivität mit externen Schnittstellen ausreichen, wenn z. B. die detaillierte Analyse der Zustände in externen Schnittstellen zu aufwändig ist.

Bei der Definition der Richtlinien für Protokollierung ist darauf zu achten, dass sich eine nachvollziehbare Gewichtung der Log-Ausgaben für die Log-Stufen ergibt. Auf den Produktivsystemen wird meistens Log-Stufe „Info“ oder „Warn“ aktiviert – d. h. im Entwicklungsteam muss kommuniziert sein, welche Aktionen mit dieser Log-Stufe zu protokollieren sind. Ein einheitliches Logging-Framework über Komponenten hinweg vereinfacht die Handhabung und die Verwaltung und die Konfiguration für das Logging kann so außerdem zentralisiert werden.

In vielen Projekten mit Bezug auf persönliche Daten werden besondere Vorschriften bezüglich Protokolldaten definiert. Beispielsweise muss es grundsätzlich vermieden werden, personenbezogene Daten zu protokollieren. Das erleichtert auch den Zugriff auf die Log-Dateien, da keine vorgelagerten Verwaltungsprozesse wie Freigaben notwendig sind.

Aktuelle Softwaresysteme werden meist auf mehrere Knoten (z. B. im Cluster) verteilt. Es ist besonders wichtig darauf zu achten, dass entweder jeder Knoten für sich so weit abgeschlossen ist, dass die Fehlersuche innerhalb des Knotens stattfinden kann, oder das Logging-Framework so zu konfigurieren, dass die Protokolleinträge auf unterschiedlichen Systemen einem Ereignis zugeordnet werden können. Hierfür benötigt man spezielle Werkzeuge zur Zusammenführung mehrerer Protokolldateien und zur Analyse. Damit die Zusammenführung möglich wird, müssen diverse Protokolleinträge eines Ereignisses einander zugeordnet werden können. Das kann auf einfache Weise durch die Ausgabe der ID des Initialereignisses, durch Thread-ID oder durch Framework-interne Kontext-ID erfolgen. Einige Frameworks sind darüber hinaus imstande, Protokolldateien auf jedem Knoten im Cluster automatisiert in eine zentrale Referenzliste einzutragen, sodass kein Knoten versehentlich vergessen wird.

Zuletzt sei darauf hingewiesen, dass die Konfiguration des Formats der Protokollausgaben nicht weniger wichtig ist. Einerseits müssen die Protokollausgaben möglichst alle notwendigen Details liefern, damit die Fehleranalyse möglich ist. Andererseits ist aber darauf zu achten, dass die Protokolldateien nicht allzu groß werden – die Bereithaltung der Protokolldaten für mehrere Tage kann für einige Großsysteme bereits zu einer Herausforderung werden. Und auch der Test benötigt Informationen, wie er die zu prüfende Information (z. B. automatisiert) findet.

4. Andere Schnittstellen auslesen

Hier gelten dieselben Architekturmaßnahmen wie bei „Andere Schnittstellen befüllen“ (siehe Tabelle 3).

Endzustand und Nachbedingungen prüfen

1. Zustand des SUT auslesen

Analog zur kontrollierten Erzeugung eines Startzustands zu Testbeginn muss der Test erkennen, in welchem Zustand sich das SUT nach Abschluss des Tests befindet, um prüfen zu können, dass ein Soll-Zustand wie verlangt tatsächlich erreicht wurde. Aber

Testbarkeits-Anforderung	an	Mögliche Architekturmaßnahmen zur Realisierung	Einhaltung	Zu vermeiden
UI-Objekte erkennen und auslesen	SUT	Technologieauswahl Verwendung fachlich eindeutiger und stabiler Identifier Abstimmung mit Testautomatisierern	Code-Analyse, Code-Review	Einsatz von Technologien mit generierten UIS / generierten (dynamischen) IDS
Datenbank auslesen	SUT	Abstrakte und verständliche DB-Schema-Definition Werkzeugunterstützung	Architektur-Review Regelmäßige Analyse manueller Tätigkeiten	Ableiten des DB-Schemas aus der Klassenstruktur Manuelle Eingriffe in der DB-Bereitstellung
Log- /Protokoll-Dateien auslesen	SUT	Richtlinien für Log-Level und Log-Ausgaben Einheitliches Logging-Framework und -Format (z. B. log4j) Dokumentation der Log-Level und -Ziele	Code-Analysen Architektur- und Code-Reviews	Einsatz unterschiedlicher Frameworks/Formate Fehlende Richtlinien Log-Ausgaben verstreut in
Andere Schnittstellen auslesen	SUT	Realisierung der TU-Abstraktionsschicht	Code-Analysen Architektur- und Code-Reviews	Direkte Zugriffe auf Fremd-systeme/Schnittstellen

Tabelle 3: Architekturmaßnahmen im Kontext „Ausgaben des SUT prüfen“.

Testbarkeits-Anforderung	an	Mögliche Architekturmaßnahmen zur Realisierung	Einhaltung	Zu vermeiden
Zustand des SUT auslesen	SUT	Zustandsmodell definieren Automatisierte Mechanismen zur Zustandsabfrage	Zustandsmodell pflegen	-
Fremd-komponenten auslesen	TU	Realisierung der TU-Abstraktionsschicht	Code-Analysen Architektur- und Code-Reviews	Direkte Zugriffe auf Fremdsysteme / Plattformfunktionalität Fremdsysteme, die sich nicht an die Architekturempfehlungen halten

Tabelle 4: Architekturmaßnahmen im Kontext „Nachbedingungen prüfen“.

nicht nur für den Test, sondern auch für die Wartung des Systems ist es notwendig, diese Zustände eindeutig identifizieren zu können. Als Voraussetzung zur Erfüllung dieser Anforderung müssen die Systemzustände dokumentiert und nachvollziehbar sein. Außerdem ist es essenziell, geeignete PoOs pro Zustand bereitzustellen, damit der Tester den Zustandsübergang verifizieren kann. Eine Alternative wäre die explizite Protokollierung des erreichten Systemzustands.

2. Fremdkomponenten in der Testumgebung auslesen

Zuweilen kann es aus Testsicht erforderlich sein, Endzustand und Nachbedingungen auch von externen Komponenten auslesen und prüfen zu können – wobei das eher die Ausnahme ist, da es nicht Teil des Testauftrags sein sollte, die Korrektheit angebundener Partnersysteme mitzutesten. Ansonsten gelten dieselben Architektur-Maßnahmen wie bei „Fremdkomponenten befüllen“ (siehe Tabelle 4).

Zusammenfassung und Ausblick

Wie wir gesehen haben, sind die diskutierten Maßnahmen für Architekten allesamt wohlbekannt und keine Geheimwissenschaft. Nur wurde nach unserem Kenntnisstand vorher noch nie die Brücke zu konkreten Testbarkeitsnotwendigkeiten geschlagen und dabei der Versuch einer umfassenden Sammlung unternommen. Hierin sehen wir den größten Nutzen unserer bisherigen Arbeit.
Folglich betrachten wir die *Checkliste* (siehe Tabellen 1 bis 4) in der aktuellen Form nur als Startschuss – idealerweise wird sie künftig von Architekten genutzt und ergänzt, verbessert und erweitert. Deshalb haben wir sie auf einer eigenen Seite publiziert und Kanäle für Erweiterungswünsche eingerichtet (vgl. [Ise]). Der weiteren Entwicklung sowie konstruktivem Feedback jeder Art sehen wir mit Spannung entgegen. ||

Literatur & Links

[Bra15] C. Brandes, S. Okujava, J. Baier, Architektur und Testbarkeit: Eine Checkliste (nicht nur) für Softwarearchitekten – Teil 1, in: OBJEKTspektrum 1/2016

[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`

[Ise] Isento GmbH, Architektur und Testbarkeit: Eine Checkliste (nicht nur) für Software-Architekten, siehe: `http://www.architecture4testability.com`

[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`

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.