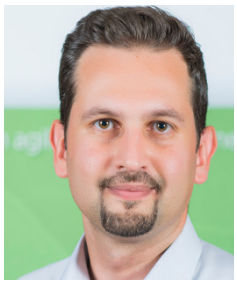
Daniel Knapp
(daniel.knapp@andrena.de)
arbeitet seit 2005 als Softwareentwickler und Coach bei der andrena objects ag. 2010 wurde er zum Leiter des Geschäftsfelds Lösungen berufen. Aktuell leitet er das Geschäftsfeld Finanzdienstleister.



Mustafa Yilmaz
(mustafa.yilmaz@andrena.de)
arbeitet seit 2008 als Softwareentwickler und Coach bei der andrena objects ag. 2015 wechselte er in den Bereich Consulting und berät seine Kunden als Agile Coach und Trainer.

Tools und Techniken in der Testpyramide: Wo eine Größe nicht allen passt

Die Konzepte des agilen Testens – besonders eine umfassende, kontinuierliche Testautomatisierung – sind in den vergangenen Jahren im Markt angelangt. Dabei empfiehlt sich ein Vorgehen anhand der sogenannten Testpyramide. Offen bleibt jedoch die Frage, welches Testvorgehen und Werkzeug sich auf welcher Ebene der Pyramide eignet. Dieser Artikel beschreibt einen Werkzeugkasten, der aus unserer Erfahrung in agilen Projekten entstanden ist. Er enthält verschiedene Lösungsansätze für die gängigsten Testszenerien auf den höheren Ebenen der Testpyramide.

Welches Testvorgehen und welche Werkzeuge empfehlen sich auf welcher Ebene der Testpyramide (vgl. [Fow12])? Auf den unteren Ebenen gelten meistens Unit- und Mock-Frameworks als Mittel der Wahl – darüber herrscht ein breiter Konsens. Ganz anders verhält es sich mit den höheren Ebenen der Pyramide. Dort beantworten die gängigen Praxisrealisierungen diese Frage – wenn überhaupt – mit dem Einsatz ein und desselben Werkzeugs und Vorgehens für unterschiedliche Szenarien, getreu dem Motto: „Wenn man einen Hammer hat, sieht alles wie ein Nagel aus." Dabei ist mit dem *One-Size-Fits-All*-Ansatz kaum zu erreichen, was für Integrationstests selbstverständlich genauso gilt wie für Unittests: Sie müssen wiederhol- und reproduzierbar sein.

Bei integrativen Tests kommt erschwerend hinzu, dass diese in sich komplex genug sind, um für jeden einzelnen Schritt innerhalb des Testablaufs unterschiedliche Ansätze zu verlangen. Dieser Artikel beleuchtet daher vier Teilschritte des Integrationstestens, analysiert die jeweilige Herausforderung und stellt verschiedene Werkzeuge und Ansätze aus dem Java-Umfeld vor, um die Tests kontrollierbar und damit dauerhaft reproduzierbar zu halten.

Wie die beschriebenen Werkzeuge in der Praxis eingesetzt werden können, lässt sich unserer Beispiel-Webapplikation entnehmen, die unter GitHub kostenlos verfügbar ist (vgl. [And15]). Sie enthält ein umfangreiches Readme, das es erlaubt, die verschiedenen Testansätze nachzuvollziehen.

Unsere Beispielapplikation (siehe Abbildung 1) besteht aus mehreren Systemteilen, die über REST-basierte Schnittstellen mitein-

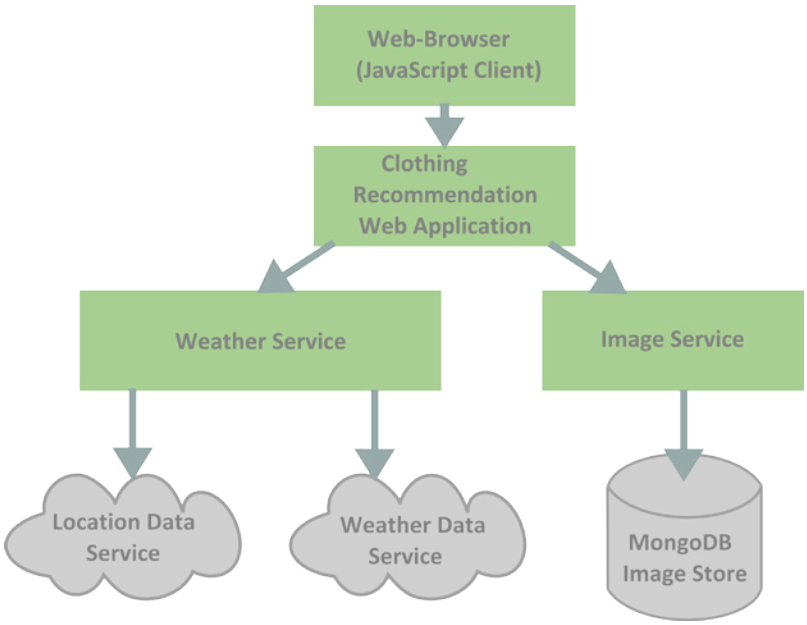


Abb. 1: Schematischer Aufbau der Beispielapplikation.

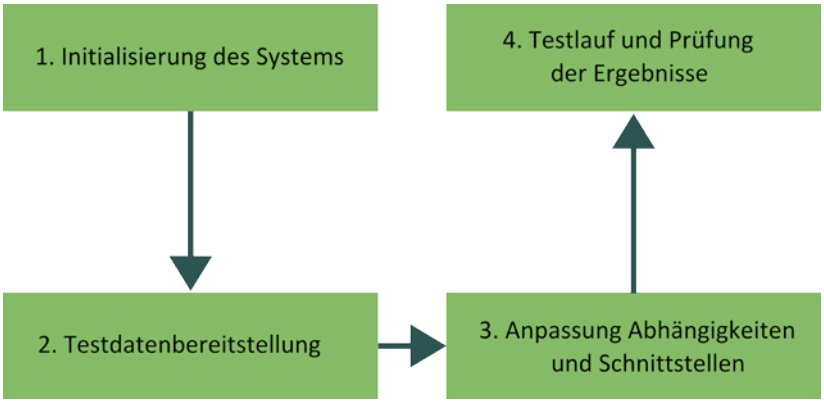


Abb. 2: Vier Teilschritte der Integrationstests.

ander interagieren. Ziel der Anwendung ist es, für die tagesaktuellen Wetterbedingungen einen Kleidungsvorschlag in Form eines Bildes zu präsentieren. Die Anwendung wurde als Webapplikation implementiert und setzt auf „SpringBoot“, „AngularJS“ und „MongoDB“ auf. Als Build-System kommt „Maven“ zum Einsatz. Die Systemteile im Einzelnen sind:

- Ein Image-Service, der aus einer NoSQL-Datenbank Bilder an den Webbrowser liefert.
- Ein Wetterdaten-Service, der auf einen externen Wetterdaten-Dienstleister zugreift und dessen Daten aggregiert.
- Eine Frontend-Anwendung, die die einzelnen Systemteile miteinander verbindet und die Schnittstelle zum Benutzer darstellt.

Wir verweisen in den entsprechenden Abschnitten auf die Beispielapplikation, sofern der vorgestellte Aspekt oder das Werkzeug verwendet wurde und am Beispiel darstellbar ist.

Die vier Teilschritte, die jeder Integrationstest durchläuft, sind (siehe auch Abbildung 2):

1. Das System initialisieren, also in einen testbaren Zustand bringen.
2. Testdaten in geeigneter Form bereitstellen.
3. Bei Bedarf Abhängigkeiten der Systeme untereinander, Schnittstellen und Systemkommunikation ersetzen oder simulieren.
4. Die eigentlichen funktionalen Tests geeignet aufsetzen.

Wie kann nun eine Umsetzung dieser Teilschritte aussehen? Worauf sollte man dabei besonders achten?

Systeme in einen testbaren Zustand bringen

Auf Unittest-Ebene war dieser Schritt einfach, hier lassen sich die Tests leichtgewichtig innerhalb der Entwicklungsumgebung starten, da es primär darum geht, Einzelfunktionen zu testen. Auf der Ebene der Integrationstests dagegen steht das Zusammenspiel von Systemen beziehungsweise deren Komponenten auf dem Prüfstand, was die Initialisierung weitaus komplexer gestaltet.

Eine einfache Lösung scheint es zu sein, feststehende, dauerhaft verfügbare Testumgebungen zu nutzen, deren Komponenten alle bekannt und kontrollierbar sind. Erfahrungsgemäß sind dabei jedoch klassische Nutzungskonflikte nicht ausgeschlossen: Die Testumgebungen werden für verschiedene Arten von Tests – manuelle und automatisierte – verwendet, die sich gegenseitig beeinflussen. In der Folge sind die Ergebnisse weder selbstvalidierend noch verlässlich sowie wiederholbar.

Die bessere Alternative besteht nach unserer Erfahrung darin, die Systeme unter Test leichtgewichtig und so häufig wie nötig selbst beziehungsweise im laufenden Build-Prozess starten zu können. Dafür existieren unterschiedliche Ansätze, die wir im Folgenden vorstellen.

Komponententests mit Frameworks wie Spring oder CDI

Frameworks wie „Spring“ oder „CDI“ bringen entweder eigene (vgl. [Joh15]) Test-Frameworks mit oder es existieren ergänzende Frameworks („CDI Unit“, „Arquillian“, vgl. [Coo11], [Red15]), um Softwarekomponenten eigenständig hochzufahren. Für Komponententests oder begrenzte Integrationstests (z. B. von Datenbankabfragen) eignet sich dieses Vorgehen hervorragend. Komplexe End-to-End-Tests lassen sich dagegen eher nicht abbilden.

End-to-End-Tests: Einzelne Systeminstanzen hochfahren und testen

In der Praxis schreibt ein Entwickler Code und verpackt ihn zum Testen in ein installierbares Artefakt (*Deployable*), dann übergibt er das Artefakt in eine Laufzeitumgebung und führt Tests direkt aus. Die Herausforderung bei der Systeminitialisierung besteht darin, eine unabhängige und reproduzierbare Laufzeitumgebung bereitzustellen, die das Ergebnis der Testläufe nicht negativ beeinflusst.

Klassische Server-Systeme

Arbeitet man unter Einsatz von klassischen Applikationsservern und konventionellen Deployment-Modellen, so bieten beispielsweise Arquillian oder das Maven-Cargo-Plug-in (vgl. [Cod15]) die Option, aus den Build- und Testläufen heraus verschiedene Applikationsserver zu managen und für den Testlauf aktiv zu schalten. Im Rahmen eines Maven-Build-Prozesses lassen sich diese vor und nach der Testausführung in den Build-Ablauf zum Zwecke des Server-Lebenszyklus-Managements einklinken. Im Optimalfall befindet sich ein aktuelles Paket des hauseigenen und somit individuell konfigurierten Applikationsservers in einem Artefakt-Repository. Dann kann es im Rahmen des Build-Prozesses ohne Zutun des Entwicklers oder eines Administrators frisch aufgesetzt werden.

Das Maven-Dependency-Plug-in (vgl. [Apa15]) bietet sämtliche Mechanismen zur Unterstützung dieses Vorgehens. So werden Nebeneffekte aus vorangegangenen Testläufen ausgeschlossen. Die Ablaufumgebung für Tests ist mit Ausnahme des Betriebssystems und beispielsweise Datenspeichern sowohl auf dem Entwicklerrechner als auch auf einem *Continuous-Integration*-Server (*CI*-Server) vergleichbar.

Frameworks wie Spring Boot oder Play

Frameworks wie „Spring Boot“ oder „Play“ (vgl. [Piv15], [Pla]) drehen den klassischen Ansatz um, bei dem mehrere Applikationen in einen Applikationsserver gepackt werden. Stattdessen enthalten sie einen eingebetteten Server innerhalb der Applikation. Somit ist es relativ leicht, die Applikation inklusive der Laufzeitumgebung für Testläufe in einem jungfräulichen Zustand zu initialisieren. Ein simpler *java-jar*-Befehl ist zum Starten ausreichend. Generell gelten aber die gleichen Einschränkungen wie im vorigen Abschnitt: Externe Ressourcen unterscheiden sich möglicherweise von Testumgebung

zu Testumgebung. Ein Praxisbeispiel hierzu und zu vielen anderen Szenarien in diesem Artikel befindet sich in unserer Beispielapplikation [And15].

Containerized-Applikationen
Setzt man dagegen Container-Technologien wie „Docker“ (vgl. [Doc-b]) ein, um eigene Applikationen oder Applikationsserver zu verpacken, so resultieren daraus installierbare Images, die unabhängig vom eigenen Betriebssystem die gleichen Bedingungen für die Tests schaffen können.

Grundsätzlich lassen sich Container-Technologien auch für Legacy-Systeme einsetzen. Die Testumgebung kann unter Zuhilfenahme eines Docker-Maven-Plug-ins (vgl. [Huß15]) gestartet und frisch initialisiert werden.

Generell kann ein Gesamtsystem durchaus aus mehreren Container-Images bestehen. Um solche Systeme oder Systemverbünde mittels diverser Container-Technologien zu „bootstrappen“, also in einer geordneten Reihenfolge hochzufahren und bereitzustellen, bietet sich der Einsatz von „Docker Compose“ (vgl. [Doc-a]) an. Docker Compose ist ein Tool, um Multi-Container-Applikationen in Docker zu definieren und zu betreiben.

Alle bisher vorgestellten Ansätze dienten dem ersten Schritt, das System zu initialisieren, um später die Tests dagegen laufen zu lassen. Der folgende Schritt besteht in der Datenmanipulation zum Zweck der Testdaten-Bereitstellung.

Testdaten-Bereitstellung

Um die Bedeutung dieses Schrittes zu illustrieren, machen wir einen kleinen Exkurs darüber, was es bedeutet, etwas zu testen. Grundsätzlich vermutet der Entwickler, eine bestimmte Konstellation könne eintreten. Dementsprechend spezifiziert er diese Vermutung und legt darüber hinaus fest, wie das Ergebnis des Tests aussehen soll. Der testende Entwickler weiß zu diesem Zeitpunkt natürlich auch schon, welchen Teil des Systems er überprüft. Daher existiert bereits eine implizite Annahme darüber, welche Daten der Test benötigt, um in der angegebenen Konstellation positiv zu verlaufen.

Automatisierte, funktionale Tests sind dazu da, gewisse Verhaltensweisen zu verifizieren. Je nachdem, wie die Eingangsdaten im System variiert werden, können unterschiedliche Ergebnisse erzielt werden. Das ist trivial, aber nur auf den ersten Blick. Die Herausforderung besteht darin, zu vermeiden, dass sich die – oft parallel verlaufenden – Testläufe der einzelnen Entwickler gegenseitig beeinflussen. Diese Gefahr besteht grundsätzlich, wenn alle Tester gleichzeitig auf eine zentrale Datenspeicherinstanz zugreifen. Es gilt, die gegenseitige Einflussnahme zu verhindern.

Um zu kontrollieren, welche Daten und Datenschemata in ein System eingespeist werden, bieten sich – im Falle des Einsatzes relationaler Datenbanken – Werkzeuge wie das „Maven-SQL-Plug-in“ (vgl. [Moj15]), „Flyway“ (vgl. [Fly15]) oder „Liquibase“ (vgl. [Liq15]) an. Mit diesen kann der Entwickler Schemata und Daten hinterlegen, die das gewählte Werkzeug inkrementell in das frisch aufgesetzte System einspeist, ehe der Test ausgeführt wird. So kann im Rahmen des Testlaufs auch die Änderung des Datenschemas überprüft werden.

Ähnliche Werkzeuge existieren auch im Umfeld von nicht-relationalen Datenbanken, beispielsweise auch für dokumentenorientierte Datenbanken wie „MongoDB“ (vgl. [Mon15-a], [Mon15-c]).

Die Frage nach der gewählten Datenspeicher-Instanz ist damit nach wie vor offen. Ihre Antwort hängt von der Art des Testlaufs ab. Bei einem Integrations- oder End-to-End-Testlauf auf dem Entwicklungsrechner des Entwicklers empfiehlt es sich, eine zentrale Datenspeicher-Instanz durch eine *in memory*- oder *embedded*-Variante wie „HSQLDB“ (vgl. [HSQ15]) oder „H2“ (vgl. [H2]) zu ersetzen. Dabei wird die Datenbank-Instanz pro Testlauf neu initialisiert und nach dem Testlauf verworfen. Mehrere parallele Tester beeinflussen sich damit nicht gegenseitig.

Alternativ lassen sich Speicher gleich in ein Container-Image packen, sofern das praktikabel erscheint. So können nahezu sämtliche Datenspeicher als *self-contained*-Images aus zentralen Online-Diensten wie „DockerHub“ bezogen werden. Container-Images bieten Entwicklern nicht nur den Vorteil, dass die Installation entfällt. Sie erlauben es auch, die Persistenz schon vorab mit sinnvollen Testdaten anzureichern, auf die Tests dann zugreifen können. Auch das macht den Testlauf reproduzierbar.

Aus einem zentralen CI-Build-Lauf dagegen würde man eine zentrale Datenspeicher-Instanz ansteuern. Relationale Datenbanken sind nur ein Beispiel, andere externe Ressourcen wie Messaging-Systeme oder NoSQL-Speicher folgen analogen Regeln. So existieren für populäre NoSQL-Speicher wie „Neo4J“ (vgl. [Neo]), „Redis“ (vgl. [Sty15]) oder „Mongo-DB“ (vgl. [Mon15-b]) *embedded* oder *containerized* Varianten, Messaging-Systeme wie „ActiveMQ“ (vgl. [Apa11]) können ebenfalls eingebettet betrieben werden.

Manchmal ist es jedoch nicht möglich, Systeme mit komplexen Datenbeständen mit vertretbarem Aufwand aus dem Testlauf heraus zu bestücken. Gründe dafür können sein, dass das Datenschema hierfür zu komplex oder dass die Testdaten-Menge zu groß ist und es daher zu lange dauert, die Datenbank zu initialisieren.

In diesem Fall bleibt zu überlegen, ob es sinnvoller erscheint, für stündliche oder nächtliche Testläufe exemplarische, bereits gefüllte „Abzüge“ der Datenbank zur Verfügung zu stellen, die wenigstens eine gewisse Wiederholbarkeit sichern und nach dem Testlauf verworfen werden.

Umgang mit Abhängigkeiten und Schnittstellen

Abhängigkeiten: Isolation von Kollaborateuren für funktionale Tests
Kommuniziert das zu testende System im Rahmen von Tests mit externen Anwendungen oder Systemen, auf die der testwillige Entwickler gar keinen Einfluss hat, dann sind die Ansätze aus dem Abschnitt „Systeme in einen testbaren Zustand bringen“ nicht mehr hinreichend. Die transitiv abhängigen Systeme oder Anwendungen müssten vor dem Testlauf grundsätzlich ebenfalls initialisiert und für den Test verfügbar gemacht werden. Das gleiche gilt, wenn geschlossene Systeme vorhanden sind, die keine automatisierten Eingriffe erlauben, oder wenn Ergebnisse externer Systemanfragen über die Zeit stark variieren. In manchen Fällen ist es auch schlichtweg zu teuer oder fehlerträchtig, die externen Systeme anzusprechen.

In dieser Konstellation besteht der Lösungsansatz darin, die zu testende Umgebung geeignet zu modifizieren, sodass sie von der produktiven Umgebung an definierten Stellen abweicht. Bei der Modifikation ersetzt der Entwickler bestimmte Teile des Gesamtsystems an den identifizierten Systemgrenzen durch Attrappen, selbst geschriebene Komponenten, die im Test anstelle der echten Kollaborateure definierbare und stabile Ergebnisse liefern. Voraussetzung dafür ist allerdings, schon bei der Entwicklung des später zu testenden Systems zu berücksichtigen, dass man nachher auf diese Weise testen möchte.

Beispielsweise könnte man in einer Spring-basierten Applikation den Zugriff auf externe Systeme in Komponenten kap-

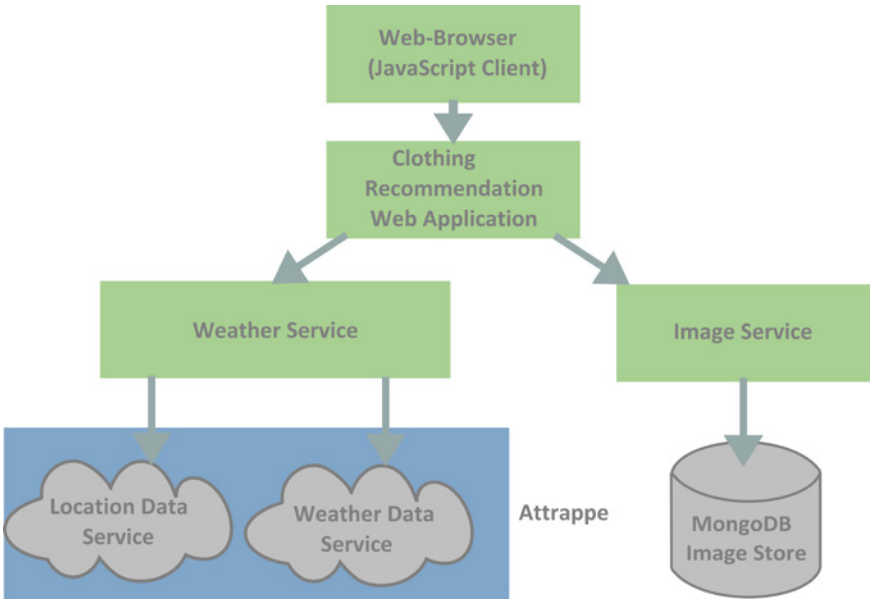


Abb. 3: Attrappen-Einsatz.

seln, die zur Laufzeit mit verschiedenen *Bean-Profilen* (vgl. [Joh15]) aktiviert werden. Im CDI-Umfeld empfiehlt sich die Implementierung zum Beispiel verschiedener *Alternatives* (vgl. [Ora13]).

Attrappen

Attrappen (siehe Abbildung 3) spielen eine besondere Rolle, wenn es darum geht, Systemabhängigkeiten zu testen. Wir hatten bereits beschrieben, dass wir mit „Attrappen" selbst geschriebene Komponenten meinen. Offen ist jetzt die Frage, was wir in diese Attrappen implementieren.

Grundsätzlich können wir verschiedene Arten von Attrappen in das System stellen. Dazu schlagen wir zwei Ansätze vor:

1. *Der programmatische Ansatz:* Die implementierten Attrappen liefern fixe, definierte Daten zurück, damit das System auf eine bestimmte Weise reagiert. Konkret bedeutet das, Code zu programmieren. Die Attrappen selbst werden hierbei in der eigenen Komponente verankert.
2. *Der Capture&Replay-Ansatz:* Hier werden Antworten von Kollaborateuren im Echtbetrieb aufgezeichnet und die aufgezeichnete Antwort wird im Testlauf abgespielt. Dabei steuern wir der Softwarekomponente eine definierte Antwort im Format des Kommunikationsprotokolls zu. Die Applikation bearbeitet diese Antwort im Rahmen des Tests mit der echten Schnittstellenlogik. Unterschiedliche Datensätze werden dabei im Kommunikationsformat aufgenommen und innerhalb des Tests immer wieder abgespielt. Daher werden Transformations- und Kommunikationslogik auch mit getestet.

Ein auf diese Art und Weise modifiziertes (Sub-)System erlaubt stabile funktionale, integrative Tests, deren Ergebnisse nicht von externen Kollaborateuren beeinflusst werden.

Selbstverständlich sollten die genannten Tests um wenige exemplarische End-to-End-Tests ergänzt werden, die den Aufruf der Echtsysteme enthalten, sofern möglich. Hier wird man allerdings den Test auf wenige zu testende Aspekte begrenzen (z. B. „Aufruf erfolgreich").

Testen des Kommunikationsverhaltens

Diesbezüglich sollte man zwei grundlegende Szenarien unterscheiden: den Test des Kommunikationsverhalten eines Systems einerseits als Anbieter einer externen Schnittstelle und andererseits als Konsument einer externen Schnittstelle. Sofern man mehrere Applikationsbestandteile selbst verantwortet, von denen einer eine Remote-Schnittstelle anbietet und ein anderer diese konsumiert, ergibt sich ein drittes Szenario. Damit bieten sich Möglichkeiten, die Bestandteile entkoppelt zu testen.

Integrative Tests (System als Provider)

Der erste Ansatz besteht darin, das laufende System im Test integrativ, das heißt im späteren Kommunikationsformat, anzusteuern. Natürlich ist es sinnvoll, die Applikation vorab durch Anwenden der Ansätze aus dem Abschnitt „Systeme in einen testbaren Zustand bringen" zu initialisieren (siehe oben). Im Anschluss können Remoting-Schnittstellen im Falle von REST-basierter Kommunikation beispielsweise mittels „Rest-assured" (vgl. [Jay]) getestet werden. Dieses Werkzeug bietet eine Fluent-Interface-DSL an, um das erwartete Verhalten der angebotenen Schnittstelle zu spezifizieren. Im Falle von SOAP-basierter Kommunikation testet man sinnvollerweise über Code-generierte Implementierungen von Client-Bibliotheken, die mit den üblichen Testtreibern wie JUnit und zugehörigen Assertion-Bibliotheken angesteuert und verifiziert werden können.

Integrative Tests (System als Consumer)

Sofern das eigene System über Schnittstellen Inhalte aus Fremdsystemen konsumiert, sollte die Behandlung der externen Antworten typischerweise ebenfalls im Rahmen von Testläufen überprüft werden. Hierzu hatten wir im Abschnitt „Abhängigkeiten: Isolation von Kollaborateuren für funktionale Tests" über Attrappen diverse Lösungsmöglichkeiten vorgestellt. Möchte man jedoch – in Ergänzung zum funktionalen Verhalten – Erkenntnisse über ein korrektes Kommunikationsverhalten des eigenen Systems gewinnen, ist es erforderlich, über eine Remote-Strecke zu kommunizieren.

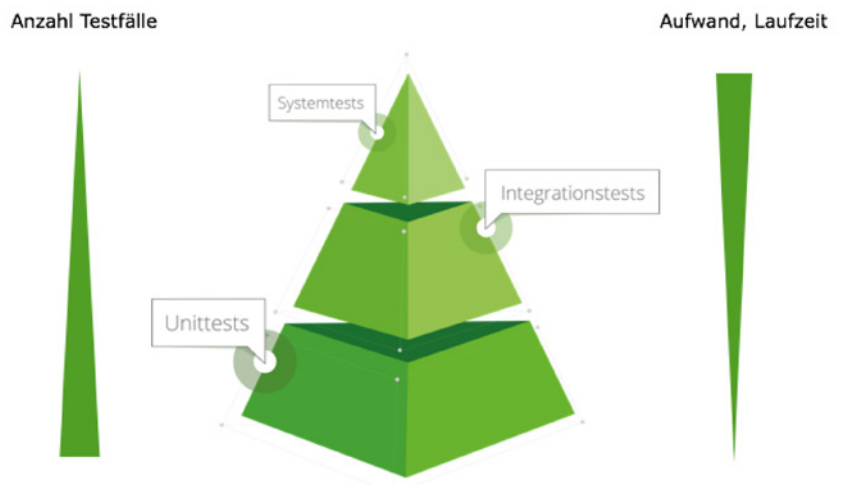
Der Test selbst wird auch im Rahmen dieses Szenarios nicht zwingend gegen einen echten Service ausgeführt, sondern beispielsweise von einer Middleware-Komponente mit Routing-Fähigkeiten auf eine extern platzierte Attrappe geroutet. Dieser Ansatz erlaubt es somit ebenfalls, stabile Testergebnisse zu erhalten.

Ergänzen sollte man das Testverfahren um exemplarische Smoke-Tests. Sie überprüfen periodisch die Annahmen über Fremdsysteme, die man im Rahmen der Implementierung der Attrappen trifft. Damit kann man beispielweise unerwarteten Schnittstellenänderungen begegnen.

Verifikation mit Hilfe von Kontrakten

Hinsichtlich des dritten Szenarios existiert die Möglichkeit, reine Kommunikations- und Interaktionsaspekte leichtgewichtiger zu testen, ohne einen kompletten Softwareverbund hochzufahren. So eignet sich beispielsweise das Werkzeug „Pact" (vgl. [DiU]) im Microservice-Umfeld, um das Interaktionsverhalten von zwei miteinander kommunizierenden Rest-Services zu beschreiben und sowohl den Server als auch den Client weitestgehend unabhängig voneinander zu testen (siehe Abbildung 4). Dabei

Auf der Ebene der Unittests finden sich sehr viele Tests, die sehr schnell und mit geringer Komplexität ausgeführt werden können. Je integrativer getestet wird, desto komplexer wird das Test-Setup und desto stärker steigt die Testausführungszeit. Deshalb sollte man sich auf den höheren Ebenen der Testpyramide – die selbstverständlich auf den niederen Schichten aufbauen – auf möglichst wenige aussagefähige Tests beschränken



Kasten 1: Die Testpyramide.

wird im Rahmen eines Pact-Tests zunächst eine Spezifikationsdatei erstellt, welche für zu erwartende Anfragen die entsprechenden Antworten spezifiziert (die Datei enthält die Struktur und exemplarische Daten von Beispielaufrufen und -antworten).

Im Rahmen des Client-Testlaufs instanziiert Pact einen HTTP-Server und spielt die aufgezeichneten Antworten ab. Deren korrekte Verarbeitung kann damit im Testlauf verifiziert werden. Im Falle des Servertests erzeugt Pact aus der Spezifikation einen Client, der die aufgezeichneten Anfragen gegen den Server abschickt, um im Anschluss die Antworten zu überprüfen.

Sowohl der Client als auch der Server können somit im Rahmen des Continuous-Builds unabhängig voneinander gegen die Datei verifiziert werden. Ein vollständig integrativer Test ist infolgedessen erst einmal nicht notwendig, er kann möglichst spät in der Build-Pipeline folgen. Das entlastet die CI-Pipeline.

Dieser prinzipielle Ansatz lässt sich grundsätzlich in weiteren Umfeldern einsetzen, in denen über ein konkretes Austauschformat kommuniziert wird. Erfüllen Konsument und Anbieter die Restriktionen des Austauschformats, also den zugrunde liegenden Kontrakt für exemplarische Daten, kann man davon ausgehen, dass die einzelnen Teilsysteme prinzipiell fehlerfrei miteinander kommunizieren können.

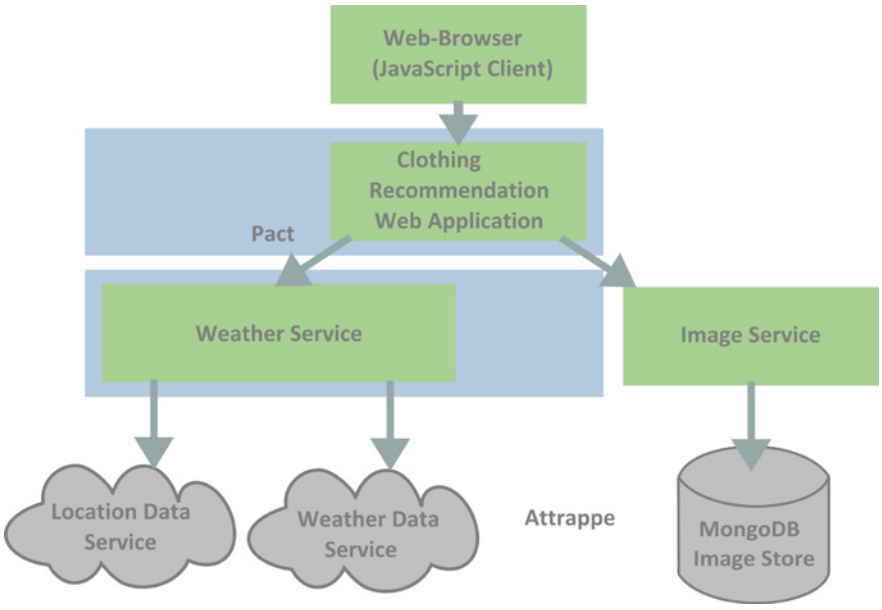


Abb. 4: Pact-Test.

Funktionale Tests

Ein und dasselbe Werkzeug wird oft für verschiedenste Zwecke verwendet. Im Java-Umfeld ist es meistens JUnit, das in der Praxis für Integrationstests auf verschiedenen Ebenen eingesetzt wird. Man kann Oberflächentests in JUnit „einbetten“ und von JUnit aus ansteuern (z. B. mittels „Selenium WebDriver“, vgl. [Sel]) oder die unterschiedlichsten Client-Bibliotheken einhängen, beispielsweise um Webservice-Schnittstellen oder Mail-Server anzusprechen.

Das ist nach unserer Erfahrung sehr praktisch, solange das Werkzeug erstens ausschließlich von den Entwicklern selbst eingesetzt wird und zweitens die Testergebnisse im Grunde auch nur für die Entwickler interessant sind.

Anders sieht es aus, wenn Fachabteilungen die Testdaten und -ergebnisse spezifizieren und die Inhalte der Tests kennen und verändern möchten. Bei Java-Code (JUnit), der in einer Entwicklungsumgebung abläuft, weiß die Fachabteilung weder, was der Test prüft, noch, wie das Ergebnis ausfällt.

Transparenz für die Fachbereiche schaffen Werkzeuge wie Fit(-Nesse) (vgl. [Fit15]) oder Cucumber (vgl. [Cuc15]). Denn sie ermöglichen es, Tests (und somit beispielhafte Anforderungen) in einer domänenspezifischen Notation zu formulieren und menschenlesbar darzustellen. Der Vorteil ist offensichtlich: Die Endanwender können die Testdaten beziehungsweise deren Spezifikationen menschenlesbar mitbestimmen. Im Idealfall können sie diese Aufgabe sogar selbst übernehmen, allerdings sollten die Entwickler ihnen dabei helfen, einen klaren Testfokus zu formulieren und die konkreten Aspekte des Tests transparent zu machen. Die Entwicklerseite muss also dafür sorgen, dass die Fachbereichstester in der Lage sind, gewisse Teilaspekte mit dieser domänenspezifischen Sprache zu formulieren.

Ein weiterer Vorteil der domänenspezifischen Notation liegt in der Entkopplung der relevanten Testdaten und des unterliegenden Kommunikationsformats. Die Abstraktion des Tests von der zugrunde liegenden Technologie ermöglicht einen klaren Testfokus.

Um das näher zu erklären: Beispielsweise sieht ein Testablauf grundsätzlich Anfragen in einem bestimmten Format vor, das aus den Teilmengen von A bis C besteht. In einem spezifischen Fall ist jedoch nur Abschnitt C relevant, um ein bestimmtes

Testergebnis zu erreichen. Dann sollte es möglich sein, im Rahmen des Tests eben nur Abschnitt C zu manipulieren, während für die anderen Abschnitte sinnvolle Daten per Default vorgegeben werden. Andernfalls steigt der verbundene Wartungsaufwand für Test-Suiten schnell in eine zeitlich wie ökonomisch unerwünschte Höhe.

Genauso wie mit den Testdaten verhält es sich mit der Verifikation des erwarteten Testergebnisses: Ein klarer Testfokus lässt sich nur erzeugen, indem ausschließlich die für den Test relevanten Aspekte ausgewertet werden.

Fazit

Zusammengefasst geht es darum, bedarfsgerecht die richtigen Werkzeuge und Ansätze zu wählen, damit die richtigen Tests zum richtigen Zeitpunkt ablaufen können und das System sich selbst evaluieren kann. Wichtig ist dabei, dass alle vier beschriebenen Schritte automatisiert und unabhängig voneinander laufen können. Denn nur dann ist gesichert, dass die Tests so häufig und schnell wie möglich Feedback an die Entwickler liefern, um gezielt Fehler beheben zu können.

Das spart gegenüber der manuellen Ausführung und Auswertung von Tests eine Menge Zeit und Aufwand. Dafür lohnt es sich auch, nicht ein Werkzeug oder Vorgehen für alles vorzusehen, sondern, das, was am besten passt. ■

Links

[And15] Andrena objects ag, Testing-Tools-Demo, 2015, siehe: https://github.com/andrena/testing-tools-demo
[Apa11] The Apache Software Foundation, ActiveMQ, 2011, siehe: http://activemq.apache.org/how-do-i-embed-a-broker-inside-a-connection.html
[Apa15] The Apache Software Foundation, Apache Maven Dependency Plugin, 2015, siehe: https://maven.apache.org/plugins/maven-dependency-plugin/
[Cod15] Codehaus, Cargo, 2015, siehe: https://codehaus-cargo.github.io/cargo/Home.html
[Coo11] B. Cooke, Jglue, 2011, siehe: http://jglue.org/cdi-unit
[Cuc15] Cucumber Ltd., 2015, siehe: https://cucumber.io
[DiU] DiUS Computing Pty Ltd., pact-jvm, siehe: https://github.com/DiUS/pact-jvm
[Doc-a] Docker, Overview over Docker Compose, siehe: https://docs.docker.com/compose/
[Doc-b] Docker, Build, Ship, Run, siehe: https://www.docker.com
[Fit15] FitNesse, 2015, siehe: http://www.fitnesse.org
[Fly15] Flyway, 2015, siehe: http://flywaydb.org
[Fow12] M. Fowler, TestPyramid, 2012, siehe: http://martinfowler.com/bliki/TestPyramid.html
[H2] H2 Database Engine, siehe: http://www.h2database.com/html/main.html
[HSQ15] The HSQL Development Group, HyperSQL, 2015, siehe: http://hsqldb.org
[Huß15] R. Huß, Maven plugin for running and creating Docker images, 2015, siehe: https://github.com/rhuss/docker-maven-plugin
[Jay] Jayway, rest-assured, siehe: https://github.com/jayway/rest-assured
[Joh15] R. Johnson et al., Spring Framework Reference Documentation, 2015, siehe: http://docs.spring.io/spring-framework/docs/current/spring-framework-reference/htmlsingle/#beans-definition-profiles
[Liq15] Liquibase, Source Control for your Database, 2015, siehe: http://www.liquibase.org
[Moj15] MojoHaus, SQL Maven Plugin, 2015, siehe: http://www.mojohaus.org/sql-maven-plugin/
[Mon15-a] Mongobee, MongoDB data migration tool for Java, 2015, siehe: https://github.com/mongobee/mongobee
[Mon15-b] Mongeez, MongoDB Easy Change Management, 2015, siehe: https://github.com/secondmarket/mongeez
[Mon15-c] Mongo, 2015, siehe: https://hub.docker.com/_/mongo/
[Neo] Neotechnology, The Neo4j Manual v2.3.1, siehe: http://neo4j.com/docs/stable/tutorials-java-embedded.html
[Ora13] Oracle, The Java EE 6 Tutorial, 2013, siehe: http://docs.oracle.com/javaee/6/tutorial/doc/gjsdf.html
[Piv15] Pivotal Software, Spring, 2015, siehe: http://projects.spring.io/spring-boot/
[Pla] Play, siehe: https://playframework.com
[Red15] Red Hat, Inc., Arquillian, 2015, siehe: http://arquillian.org
[Sel] SeleniumHQ, Selenium WebDriver, siehe: http://www.seleniumhq.org/projects/webdriver/
[Sty15] K. Styrc et al., embedded-redis, 2015, siehe: https://github.com/kstyrc/embedded-redis