



Praxisbericht zum kollaborativen Testen komplexer Systeme mit verschiedenen Teammitgliedern

Rezeptvorschlag: FitNesse-Shake

Kollaborative Testansätze möchten den verschiedenen Projektbeteiligten eine einfache Zusammenarbeit entlang der fachlichen Anforderungen ermöglichen und zielen auf eine gemeinsame Sichtweise auf Fortschritt und Reifegrad der Softwareentwicklung ab. Diese Ideen sind dem Leserkreis wohl weitgehend bekannt, daher soll der vorliegende Artikel keine Einführung in das Testframework FitNesse werden. Vielmehr möchten wir unsere guten Erfahrungen rund um die konkrete Projekteinführung in einem größeren Softwareprojekt sowie unser Design-„hands-on“ vorstellen und für unser Umfeld kritische Erfolgsfaktoren herausarbeiten.

von Marcel Sauer und Jonas Kilian

Die Herausforderungen waren insbesondere das Zusammenrücken von Entwicklern und Testern im Rahmen der agilen Softwareentwicklung mit Scrum, die kontinuierliche Integration von Softwarebestandteilen sowie das Durchsetzen gegen ein im Haus bereits vorhandenes kommerzielles Testmanagementtool.

Technisch handelt es sich um eine geschäftsregelzentrierte SOA Middleware auf Basis von JEE mit zahlreichen Schnittstellen zu externen und internen Systemen. Umfangreiche Workflows bilden die Grundlage für die größtenteils Backend-lastige Anwendung. Die Entscheidung für ein wikibasiertes Testsystem lag in unserem

Fall sehr nahe: vergleichsweise wenig Oberflächentests, stattdessen viele hintereinander geschaltete Serviceaufrufe, Datenbankzugriffe und Dateisystem-Operationen, die nach einem hohen Abstraktionsgrad und einer leicht verständlichen Oberfläche mit dokumentativem Charakter verlangen, zudem ein Endkunde, der über Umfang und Art der durchgeführten Tests regelmäßig mit aussagekräftigen Testberichten informiert werden möchte.

Dass wir uns für das Open-Source-Produkt FitNesse [1] entschieden haben, liegt in erster Linie daran, dass damit die meisten Erfahrungen vorlagen. Es kommt allerdings in verschiedenen Ausführungen: Wir haben das Slim-Protokoll [2] gewählt, weil wir den Eindruck gewonnen haben, dass hier der langfristige Community-Support am

größten sein wird. Damit konnten wir allerdings nicht auf die doch recht umfangreiche FitLibrary [3] zugreifen, da diese nur mit dem älteren Fit-Protokoll funktioniert, was uns aber weit weniger beeinträchtigt hat, als vermutet.

Unsere Anforderungen an ein Testsystem sind zunächst völlig unabhängig von FitNesse und unterscheiden sich nicht sehr von denen anderer Projekte (Kasten: „Typischer Anspruch an eine in den Entwicklungsprozess integrierte Testautomatisierung“).

Die Akteure

Ein System unter Test (SUT) gliedern wir in folgende grundlegende Bereiche:

- eine deterministische Testumgebung
- eine oder mehrere Applikationen unter Test (AUT)
- eine oder mehrere Attrappen von Drittsystemen (Mocks)

Ob sich die Komponenten auf mehrere Maschinen verteilen oder in einer virtuellen Umgebung existieren, spielt hier eine untergeordnete Rolle, solange ausreichende Nähe zum Produktivsystem gewährleistet ist. Die grundlegenden Komponenten eines jeden Testsystems sind folgende:

- Testpläne meist gegliedert nach fachlichen Domänen und ggf. Risiken
- Test-Repository mit Test-Suiten
- Logische Tests mit Vorbedingungen, Testschritten und Nachbedingungen
- Konkrete Tests als Datenvarianzen eines logischen Tests
- Testablaufbeschreibung und Ergebnisvalidierung (oder auch Testskript)

Als Versionsverwaltung und Test-Repository bietet sich ein SCM-System an, in unserem Fall ist das Subversion. Es empfiehlt sich, die Tests gemeinsam mit der Software unter Test zu versionieren, weil auf diese Art leicht zueinander passende Versionsstände von Software und Tests gepflegt werden können. Liegen die Tests zudem in textueller, menschenlesbarer Form vor, können aus der Entwicklung bekannte Merge- und Diff-Tools [4] sowie beliebige IDEs verwendet werden.

Die restlichen Komponenten bietet FitNesse, zugänglich in Form von Wikiseiten, die einfach auf dem Dateisystem abgelegt sind. Dabei wird nicht zwischen logischen und konkreten Tests unterschieden: Durch die Schachtelung verschiedener Ordner entsteht zwar eine beliebig tiefe Hierarchie, aber es erfolgt keine formale Abhängigkeit der Tests untereinander – jeder Test definiert also sowohl Ablauf als auch Testdaten selbst. Diese

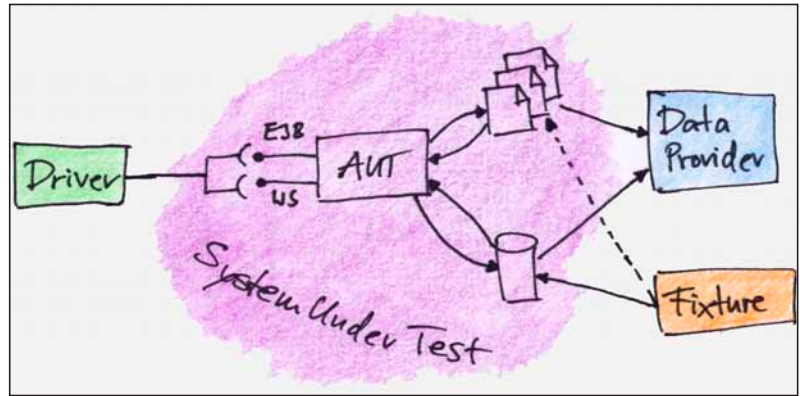


Abb. 1: Interaktion mit dem System unter Test

Typischer Anspruch an eine in den Entwicklungsprozess integrierte Testautomatisierung

Verständlich und benutzerfreundlich

- Fokussierung auf Regressionsfähigkeit und hohen fachlichen Abstraktionsgrad
- Tests gegen verschiedene Umgebungen laufen zu lassen, erfordert nicht mehr als einen Klick
- Geschwindigkeit und Spaß beim Entwickeln und Testen durch minimale Roundtrips
- Tests können in beliebigen Suites zusammengestellt werden (über Stichworte bzw. Tags)
- Tester können mit geringer Einarbeitung neben den Tests auch selbst fachliche Bausteine pflegen
- Schreiben von Code ist keine Voraussetzung für einen Tester, die Hürde sollte dennoch sehr gering sein
- Entwickler können mit geringer Einarbeitung Tests und technische Testbausteine erstellen
- Anpassen der Tests aufgrund neuer oder geänderter Features dauert nie länger als das Anpassen der Software selbst
- Statistiken und Berichte werden vollautomatisch erzeugt

Robust und nachhaltig

- Der letzte gute Softwarestand ist jederzeit einsehbar und auslieferungsfähig
- Für gefundene Fehler wird vor der Behebung ein Test als „Beweis“ erstellt
- Das Regressionspaket kann jederzeit erweitert und verändert werden
- Sowohl fachliche als auch technische Bausteine können wiederverwendet werden
- Label und Branches für die Software erfassen die dazu passenden Tests
- Häufigkeit und Dauer der Ausführung wird pro Test und Testbaustein gemessen und dient als Ansatz für Performanceoptimierungen
- Kosteneinsparung durch geringe oder keine Lizenzgebühren
- Schnelle Unterstützung von anderen Nutzern und Toolherstellern erhalten

Offen und kollaborativ

- Tests dienen als ausführbare Dokumentation einer User-Story
- Unterstützen und fördern von „Cross Functional Teams“ bestehend aus Businessanalysten, Testern und Entwicklern
- Einbettung in die agile Softwareentwicklung: Eine User-Story gilt nicht als abgeschlossen, bevor die zugehörigen Tests erfolgreich durchgelaufen sind
- Jeder im Projekt kann Tests gegen den aktuellen Softwarestand ausführen
- Tests können User-Stories oder Fehlern im Issue-Tracking-System flexibel zugeordnet werden
- Für jedes neue Feature wird zuerst ein „Happy-Test“ vom Entwickler angelegt, der vor „Übergabe“ an den Tester laufen muss
- Möglichst aktive Community für die Tools und deren Weiterentwicklung vorhanden



Abb. 2: Packages unterscheiden zwischen Baustein und Technologieadapter

Wikiseiten werden von einem integrierten Webserver an den Browser ausgeliefert, FitNesse bietet hierfür eine Restful API. Hinter dem internen Webserver läuft der so genannte SlimServer als Kindprozess; beide können sowohl lokal als auch zentral laufen, ersterer natürlich nur dort, wo der Rechner Zugriff auf das Test-Repository besitzt. Die Tests werden im Browser ausgeführt, FitNesse interpretiert die Wikisyntax und ruft zugehörigen Java-Code im Hintergrund auf. Hier kommt jetzt unser Design eines Testframeworks ins Spiel, das zudem möglichst unabhängig von FitNesse sein soll, um die Bausteine auch unabhängig zu benutzen, doch dazu später mehr.

Wiederverwendbare, toolunabhängige Testbausteine

Wir unterscheiden zwischen folgenden Arten von Komponenten, die wir der Einfachheit halber „Testbausteine“ nennen (Abb. 1):

- Aufruf der AUT, um ein gewünschtes Verhalten zu stimulieren (*Driver*)
- Abfragen von Zuständen der AUT, um diese zu validieren (*DataProvider*)
- Modifikation der AUT, um bestimmte Test(vor)bedingungen zu erzeugen (*Fixture*)

Der Begriff „Test-Fixture“ dürfte bekannt sein. Verwirrend ist jedoch, dass die FitNesse-Doku jegliche ausführ-

bare Tabelle einer Wikiseite als „Fixture“ bezeichnet, ganz egal, was der dahinter liegende Java-Code tut. Um hier sauber zu unterscheiden, nennen wir diese Komponenten FitNesse Fixture oder FitNesse Action Table. Das Konzept DataProvider haben wir uns namentlich von TestNG geborgt. Die entsprechenden Interfaces sind folgende:

Driver.java:

```
public interface Driver {
    public Result invoke();
}
```

Fixture.java:

```
public interface Fixture {
    public void execute();
}
```

DataProvider.java:

```
public interface DataProvider<T> {
    public <T> load();
}
```

Die Unterscheidung in diese drei Typen hat sich als gut erwiesen, denn wir erreichen dadurch, dass ein Testbaustein immer nur genau eine Sache macht (Single Responsibility Principle). Bevor wir in das Design tiefer einsteigen, möchten wir kurz den typischen Aufbau eines Testablaufs im FitNesse-Wiki demonstrieren.

Ansprechen von Java-Code aus einer Wiki Page

Damit ein Baustein seine Arbeit verrichten kann, muss er mit Testdaten „gefüttert“ werden. FitNesse bietet hier verschiedene Formen von Tabellen an, deren Spalten in der Regel auf Setter einer dahinter liegenden Java Bean gemapped werden:

```
! MyBean |
| Name | create? |
| Sauer | OK |

public class MyBean {
    private String name;
    public void setName(String name) {
        this.name = name;
    }

    public String invoke() {
        ..
        return "OK";
    }
}
```

Für das eigentliche Ausführen wird eine beliebige Methode der Java Bean aufgerufen (im Wiki mit einem Fragezeichen hinter dem Spaltennamen versehen), dessen

Listing 1

```
session.lookup: collection of type [class sample.bestellsystem.Kunde]
session.lookup: collection of type [class sample.bestellsystem.Artikel]
...
[in] BestellungViaWebserviceDriver.invoke
... low level Infos
[out] BestellungViaWebserviceDriver.invoke, return value:
    GenericResult[result=sample.bestellsystem.BestellStatus[...]]
Storing return value in session: id =
    BestellungViaWebserviceDriver.class
```

Listing 2

Class	Hits	Total (ms)	Max (ms)	Avg (ms)
BestellungProKundeDataProvider	20	12396	1346	619

Ergebnis als String validiert werden kann. Das Abfragen von Daten wiederum erfolgt in FitNesse mit einer so genannten Query Table, die Java Bean dazu muss eine gleichnamige Methode bereitstellen:

```

|Query:MyBean |
| Name |Vorname |
| Sauer | Marcel |

public class MyBean {
    public List<Map<String, String>> query() {
        return ..
    }
}

```

Wir empfehlen eine direkte Abhängigkeit zwischen fachlicher Darstellung einer Wikiseite und dahinter liegenden Testbausteinen zu vermeiden, zumal es hier keine 1:1-Beziehung geben muss. Stattdessen bauen wir die Testausführung nach fachlich sinnvollen Kriterien auf, indem wir hinter jede FitNesse Action Table einen Technologieadapter schalten, der im FitNesse typischen Java-Bean-Style aufgebaut ist und intern an einen oder mehrere Testbausteine delegiert (Abb. 2).

Ähnliche Technologieadapter benutzen wir für Aufrufe von der Kommandozeile oder das Load-Testing Tool „Grinder“. So sind wir frei im Refactoring von Testbausteinen, ohne uns um mögliche Seiteneffekte auf äußere Schnittstellen kümmern zu müssen, und können Testbausteine toolunabhängig portieren und wiederverwenden.

Die Eingabedaten für einen Technologieadapter müssen natürlich nicht derart explizit angegeben werden, sie können auch aus dem Ergebnis eines vorherigen Testschritts stammen. Da man bei Erstellung der Testdaten flexibel bleiben möchte, ganz egal, welche Bausteine diese später nutzen, empfiehlt sich das Speichern aller Testdaten in einer Testsession. Der Zugriff darauf erfolgt aus der Adapterschicht, sodass ein Baustein immer autark benutzt werden kann und keine impliziten Annahmen über Daten in der Testsession trifft (Separation of Concerns).

Die Bausteine aus Sicht eines Testers

Die typische Kombination von FitNesse „Action Tables“ besteht aus dem EVA-Prinzip:

1. Dateneingabe mittels FitNesse „Action Table“ (wir nennen sie „Data Table“)
2. Systemstimulation mittels Test-Driver
3. Ergebnisauswertung mit FitNesse „Query Table“ (wir nennen sie „Driver Result Table“)

Natürlich müssen diese Komponenten nicht immer starr hintereinander ausgeführt werden. Denkbar sind etwa mehrere Systemaufrufe vor der Auswertung oder Void-Aufrufe, die kein Ergebnis liefern. Da dieses Dreigespann jedoch einen hohen Wiedererkennungswert beim

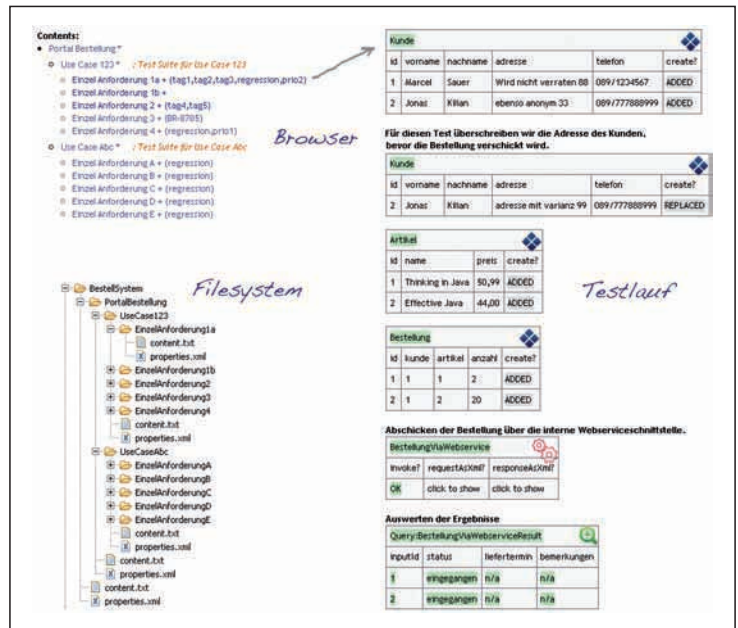


Abb. 3: Wikiseiten im Dateisystem und im Browser nach erfolgtem Testlauf

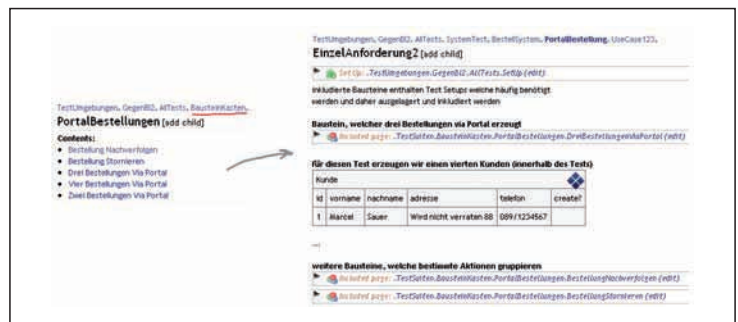


Abb. 4: Ausgelagerte fachliche Bausteine werden inkludiert und mit Datenvarianzen versehen

Benutzer hat (Abb. 3), vergeben wir für die Typen eigene Icons im FitNesse GUI.

Hinzu kommen eine beliebige Anzahl von Test-Fixtures zur Herstellung eines Test-Setups, etwa um die Datenbank mit Initialdaten zu befüllen oder Eingabedateien für eine Batch-Verarbeitung zu erzeugen; sowie beliebig viele DataProvider, die Systemzustände wie Datenbanken und Dateisystem auslesen. Auch diese Komponenten bekommen zur Wiedererkennung ein Icon.

Die allgemein gültigen Ergebnis-Strings kapseln wir in Utility-Methoden, sodass jeder FitNesse-Adapter immer die gleichen Rückgabewerte liefert, nicht etwa mal „O.K.“ und ein anderes Mal „Success“.

Zusätzlich wird bei einer Data Table vermerkt, ob ein Wert zur Testsession hinzugefügt oder überschrieben wurde. Durch das Überschreiben von vordefinierten Testdaten ermöglichen wir ein Customizing von allgemeinen Testdaten in einem spezifischen Testfall und schaffen so die Basis für einen hohen Abstraktionsgrad von Testabläufen durch Wiederverwendung fachlicher Bausteine. FiNesse unterstützt uns hier insbesondere durch das Inkludieren von vordefinierten Wiki-Snipp-

Abb. 5: Testbausteine werden dekoriert und schließlich von einem Technologieadapter aufgerufen

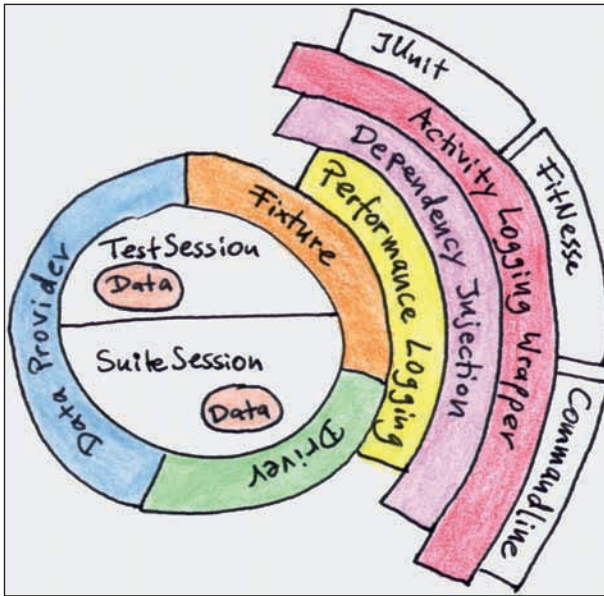


Abb. 6: Anpassen der FitNesse-Einstiegseite an die Projektbedürfnisse

Dev1	Dev2	Dev3	Lokale Umgebung (profile)
TestSuiten	TestSuiten	TestSuiten	TestSuiten
RegressionTests	RegressionTests	RegressionTests	RegressionTests
Ergebnisse	Ergebnisse	Ergebnisse	Ergebnisse
EntwicklerTests	EntwicklerTests	EntwicklerTests	EntwicklerTests
SprintDemos	SprintDemos	SprintDemos	SprintDemos
Run Smoke Tests	Run Smoke Tests	Run Smoke Tests	Run Smoke Tests

WEITERFÜHRENDE INFORMATIONEN:

- FixtureGalleries
- TextBausteine
- HowTo
- Server Logfiles anzeigen

lets, die wir in einem fachlich motivierten Bausteinkasten unterbringen (Abb. 4)

Damit ein Technologieadapter auf die Ergebnisse eines vorherigen Testbausteins zugreifen kann, werden auch alle Ergebnisse eines DataProviders oder Drivers automatisch zur Testsession hinzugefügt und können von dort auch wieder abgefragt werden. Als Schlüssel dient meist der Klassennamen des datenerzeugenden Bausteins, was für unsere Fälle bislang immer ausreichend genau war.

Für Negativtests enthält eine Driver Result Table häufig eine fachliche Exception mit entsprechender Fehlermeldung. Für diesen Fall kommen wir mit einem generischen Adapter aus, der das Ergebnis eines ausgeführten Drivers aus der Testsession ausliest und die Assertion der Exception (Message, Typ etc.) im GUI ermöglicht.

Interessant wird es, wenn die Eingabe- oder Rückgabedaten komplexer werden, was ihren hierarchischen Aufbau betrifft: Da man mit FitNesse-Bordmitteln auf zweidimensionale Tabellen beschränkt ist, müssen die Daten entsprechend „relationalisiert“ werden (Abb. 3). Wir haben uns bewusst gegen die Auslagerung der Testdaten in mächtigere Datenstrukturen entschieden, die dann etwa parallel zum Test abgelegt werden, um den Lesefluss nicht zu gefährden und die Formulierung der Tests mit einheitlichen Prinzipien zu versehen. In unserem

Umfeld haben wir damit gute Erfahrungen gemacht, auch wenn die Eingabedaten eine Bildschirmseite weit überschreiten können, so werden sie jedoch meist in fachliche Bausteine ausgelagert und so zentral gepflegt.

Bei hinlänglichen komplexen Eingabedaten empfehlen wir, diese in XML-Dateien auszulagern, die dann mit einem XML-Editor idealerweise direkt im Browser bearbeitet werden. Hierbei ist darauf zu achten, dass der fachliche Abstraktionsgrad nicht verloren geht und diese Testdaten auch von Nichttechnikern gepflegt werden können. Die Editoren Xopus [5] und XXE [6] bieten hier recht interessante Ansätze, die wir in anderen Projekten erfolgreich nutzen konnten. Sollten hingegen die Ausgabedaten komplexer werden, empfiehlt sich eine fachlich adäquate Vereinfachung für die Driver Result Table und das Wegkapseln nötiger Validierungen oder Auswertungen in dem dahinter liegenden Technologieadapter. Seitenweise Ergebnistabellen, die sich gegenseitig referenzieren und dann die Basis für Testvalidierungen darstellen sollen, gilt es unbedingt zu vermeiden.

Wir fügen jeder FitNesse Action Table für einen Driver generisch (via Superklasse) Eingabe- und Ausgabedaten im XML-Format hinzu, die sich per Mausklick aufklappen lassen (*requestAsXml/responseAsXml*, Abb. 3). Diese Streams werden mit X-Stream [7] auf den Java-Objekten der Bausteine selbst erzeugt und helfen dem Tester, direkt in der Wikiseite fehlerhafte Daten oder auch fehlerhafte Technologieadapter selber zu erkennen. Sie dienen lediglich der schnellen Information, auf ihnen werden keine Validierungen durchgeführt, weil dies natürlich das Konzept einer fachlichen Abstraktion im Wiki ad absurdum führen würde, wenn hinterher doch wieder vergleichsweise Low-Level-XML-Ströme verglichen werden.

Dekorieren von Testbausteinen mit übergreifenden Funktionalitäten

Die Erstellungen von Interfaces für unsere Testbausteine und die Instanziierung mittels Factory Methods bietet uns eine einfache Möglichkeit, zusätzliche Funktionalitäten mittels Decorator Pattern über alle Bausteine anzubieten, etwa ein einheitliches Logging der Testdurchführung (Listing 1).

Ergebnis ist ein guter Einblick in den Testablauf, der ein Erkennen etwaiger Fehler ermöglicht. Durch die feine Granularität der Bausteine kommen wir in den allermeisten Fällen ohne explizite Logging-Aufrufe in den Bausteinen selbst aus und erreichen so homogene und aussagekräftige Debug-Ausgaben. Ebenfalls über einen Decorator bringen wir zentrale Messpunkte über Häufigkeit und Dauer der Ausführung einzelner Bausteine an (Listing 2).

Dadurch können wir bei komplexeren Testsuiten Engpässe sofort feststellen und Zeitfresser schneller eliminieren. Zusätzlich ermöglichen wir einen einfachen Mechanismus zur Dependency Injection mittels Auto-wiring by Type (Listing 3) – wobei wir uns gegen eine Full-featured-Lösung wie Google Guice [8] oder das

Spring Framework [9] entschieden haben, um unsere Testbausteine schlank zu halten und gegen übermäßigen AOP-Missbrauch und ähnliches vorzugreifen – aber das ist Geschmackssache. Mittels Dependency Injection bringen wir neben den Testbausteinen noch allgemeine verwendbare Komponenten (Beans) ins Spiel, die wir nur sehr sparsam einsetzen, hauptsächlich um umgebungsabhängige Konfigurationswerte auszulesen. Alles andere fügt sich sehr schön in einen der drei Typen von Testbausteinen ein, wenn man es konsequent anwendet. Das gesamte Zusammenspiel unserer Designprinzipien ist in **Abbildung 5** zu sehen.

Ein Projekt in Bewegung

Wir stellen fest, dass ein Testprojekt in einem Umfeld dieser Größe stärker als andere Module im Quellcode in Bewegung ist, weil jedes Team dort immer durch muss. Es empfiehlt sich daher, auf einen äußerst schnellen Build-Prozess zu achten und darauf, dass 99 % der Änderungen an Wikiseiten, Technologieadaptern oder Bausteinen sofort ohne Roundtrips ausführbar sind. Zu diesem Zweck werden sowohl das Ausgabeverzeichnis der IDE (z. B. *target/classes*) als auch die Tests selbst (z. B. *src/main/fitnesse*) und die Umgebungsconfiguration (z. B. *src/main/environments*) direkt auf den Klassenpfad des SlimServers gelegt.

Die Anzahl der Tests ist binnen weniger Monate stark gestiegen. Während der laufenden Entwicklung sind ca. zwei Drittel im Schnitt im Regressionspaket enthalten, werden also regelmäßig ausgeführt und bewertet. Die zugehörigen Lines of Code ohne Kommentare belaufen sich laut Sonar [10] auf ein knappes Fünftel des gesamten Projektumfangs.

FitNesse aufhübschen

Um den Anforderungen aller Projektbeteiligten gerecht zu werden, mussten wir FitNesse stellenweise um zusätzliche GUI-Funktionalitäten erweitern. Am einfachsten geht das mit einer Kombination aus JavaScript-Code und CSS, die in die Seiten eingebettet werden und sie nach dem Rendering im Browser unmittelbar manipulieren. Als JavaScript Library haben wir uns für jQuery [11] entschieden und setzen damit im Wesentlichen folgende Ergänzungen um:

- Hinzufügen der Knöpfe zum Umschalten der Umgebungen: hierbei wird einfach der Browser-URL manipuliert, in dem die Umgebung als FitNesse-Softlink [12] enthalten ist
- Hinzufügen eines Knopfes, um die Bearbeitungshistorie eines Tests anzusehen (*svn log ...*)
- Hinzufügen der Icons zu den verschiedenen Typen von FitNesse Action Tables: Wir erkennen die verschiedenen Typen am Namen des Technologieadapters, der in der Tabelle steht und rendern das Icon einfach als CSS *background-image* hinein
- Auf- und Zuklappen der Driver-Ein- und -Ausgaben als XML-Streams

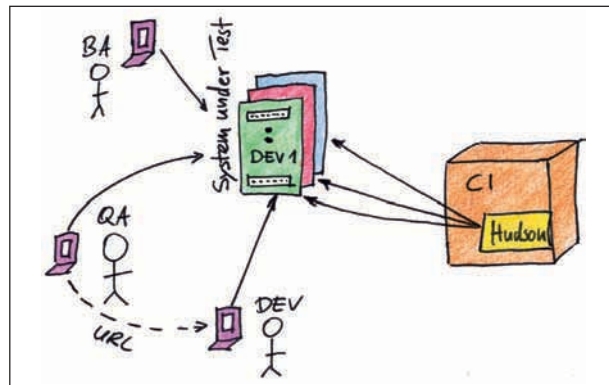


Abb. 7: FitNesse-Instanzen laufen sowohl auf dem Build-Server als auch bei den Projektbeteiligten lokal

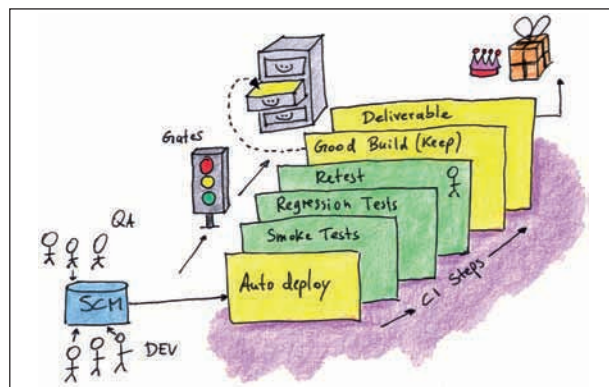


Abb. 8: Kontinuierliche Bewertung der Software und Archivierung des Last Known Good Build

FitNesse auf normalem Weg erweitern

Bereits bei FitNesse vorgesehene Erweiterungsmöglichkeiten sind zum einen eigene Widgets, d. h. spezielle Anweisungen in den Wikiseiten, die dahinter liegenden Code aufrufen. Wir nutzen sie u. a. für Folgendes:

- die Erstellung von Variablen mit Zufallszahlen, etwa für Kundennummern
- die Erstellung von Variablen bestimmter Datumsangaben (z. B. Heute vor zwei Wochen)
- das Rendern aller zu einem Tests zugehörigen Tags in die Wikiseite selbst

Zum anderen können auch so genannte *Responder* geschrieben werden (Hooks), die via Rest-API zu erreichen sind (z. B. <http://localhost:8888/MyTest?responder=myResponder>). Wir nutzen diese in erster Linie, um die Testhierarchie als XML-Stream zur Weiterverarbeitung bereitzustellen.

FitNesse darüber hinaus erweitern

Sind Änderungen gewünscht, die auf obigem Weg nicht erfüllt werden können, muss man selber Hand an FitNesse anlegen. Das ist erstmal kein Problem, schließlich ist es ein Open-Source-Tool, dessen Common Public License dies auch nicht grundsätzlich verbietet. Der Quellcode selbst ist über die vorgesehenen Hooks hinaus leider nicht sehr erweiterungsfreundlich aufgebaut, eine Mischung aus Inner Classes, Statics und Singletons machen dem geeigneten Entwickler dabei das Leben schwer. Immer wieder sind

Abb. 9: Test-ergebnisse werden vom Hudson grafisch dargestellt

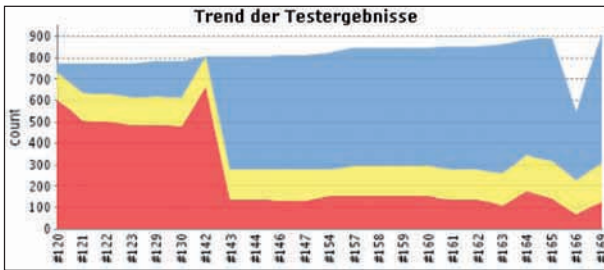
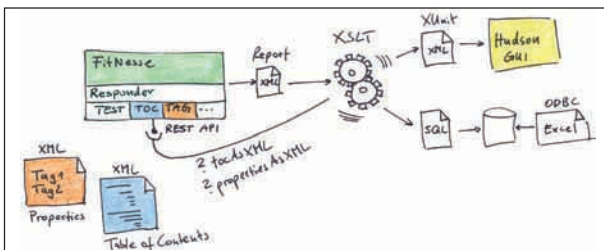


Abb. 10: Aufbereitung der Testergebnisse für Hudson Report und Excel-Dokumente



wir darauf gestoßen, Ergänzungen nur durch Voranstellen eigener Klassen gleichen Namens auf dem Klassenpfad (Classpath Overrides) zu erzwingen. Unsere Ergebnisse kann man unter *FitnessTools* [13] „bestaunen“. Die zwei Musskriterien für die Einführung von FitNesse in unserem Umfeld, die wir auf diese Weise umgesetzt haben, sind zum einen die Validierung von Ergebnissen einer FitNesse Query Table mittels regulärer Ausdrücke und zum anderen der Zugriff auf Zelleninhalte vorangegangener FitNesse Action Tables direkt aus dem Wiki heraus. Letzteres geschieht mit einer eigenen Syntax, da FitNesse dies nicht bietet: `$TabellenName1_2_SpaltenName`, wobei die erste Zahl den Index der FitNesse Action Table in dem aktuellen Test darstellt und die zweite Zahl die Ergebnisreihe in der so gewählten Tabelle. Letzteres funktioniert nur dann, wenn die Reihenfolge der Ergebnisse immer gleich ist, da wir über einen Index zugreifen. Das kann man in FitNesse leicht erzwingen, indem man statt einer Query Table eine so genannte Ordered Query Table benutzt. Nachteil derartiger Erweiterungen ist natürlich potenziell erhöhter Aufwand beim Upgrade auf eine neuere FitNesse-Version, da derartige Low-Level-APIs sicher weniger stabil sind als die offiziell vorgesehenen Extension Points.

FitNesse als Kollaborationsplattform nutzen

FitNesse als Kollaborationsplattform nutzen

Um alle relevanten Informationen schnell im Zugriff zu haben, kann man die FitNesse-Seiten mit beliebigem HTML-Quellcode versehen (Abb. 6).

Für neue Features wird gemeinsam vom Entwickler und Tester ein neues ausführbares Testtemplate angelegt, das die Funktionsweise des Tests in englischer Sprache beschreibt. Mit fortschreitender Entwicklung werden diese Sätze vom Entwickler um neue FitNesse Action Tables erweitert, die dann tat-

sächlich mit der Applikation interagieren. Diese „Entwicklertests“ beweisen das Funktionieren der Gutfälle (Happy-Cases) einer User-Story stichprobenartig und helfen bei der testgetriebenen Entwicklung. Der Tester entscheidet schließlich über Art und Umfang und überführt sie zeitnah in Regressionstests, wobei viele weitere Testfälle hinzugefügt und wenn nötig fachliche Testbausteine erweitert werden. Wenn alle Tests durchlaufen, gilt eine User-Story als abgeschlossen.

Alle Testbausteine werden in der Fixture Gallery dokumentiert und mit Beispielen versehen. Die Gallery muss immer aktuell sein und gilt als Abnahmekriterium für einen neu entwickelten Testbaustein durch einen Tester. Eigene FitNesse Widgets und allgemeine Infos zur Funktionsweise von FitNesse werden zentral in einem „How to“ (Abb. 6) gepflegt, das ebenfalls nur einen Klick entfernt ist.

Jedes Projektmitglied hat lokal eine eigene FitNesse-Instanz laufen und kann von dort nicht nur gegen das zentrale Testsystem arbeiten, sondern auch gegen die Entwicklungsrechner seiner Kollegen testen (Abb. 7). So können einzelne Tests bzw. deren Zwischenergebnisse leicht per Link verteilt oder aus einem Fehlerbericht verlinkt werden.

Da unsere Tests den globalen Zustand der Applikation im Test verändern, sind sie nicht parallel ausführbar. Um Fehler diesbezüglich von vornherein zu vermeiden, werden parallele Zugriffe auf ein System unter Test durch Halten eines Lock-Files während der Testausführung unterbunden, FitNesse bietet hier keine fertige Lösung.

Der kontinuierliche Integrationsprozess

Neben der Ausführung von Regressionstests nutzen wir FitNesse für das Überprüfen der Basisfunktionalität nach einem automatischen Deployment mittels so genannter Smoke-Tests. Auch wenn die Regressionstests während der Projektdurchführung nie 100 % erfolgreich sind, muss dies bei den Smoke-Tests stets gewährleistet sein. Die Tests werden aufgrund von Software- oder Teständerungen automatisch angestoßen und der Zusammenhang zwischen Fehlerursache und -wirkung darf nicht verloren gehen, das heißt eine Übersicht aller Änderungen seit dem letzten Testlauf soll leicht einsehbar sein (Abb. 8).

FitNesse bietet hier die Möglichkeit, die Übersicht von außen via HTTP oder Kommandozeile bzw. Apache Ant [14] aufzurufen sowie die Testergebnisse zu archivieren und später anzuzeigen. Alles andere muss man selbst hinzufügen. Das macht aber nichts, denn ein klassischer Build-Server ist hierfür viel besser geeignet. Zusätzlich bringt dieser eine Benutzerverwaltung mit, kann gute Softwarestände mit einem Label oder Tag versehen, Changelogs anzeigen und vieles mehr. Wir haben uns für Hudson [15] entschieden.

Für jeden Testlauf werden die Ergebnisse zentral archiviert, es können neben den Logfiles auch mittels HTML-Link der zugehörige Testbericht in FitNesse angezeigt werden. Dafür ist es nötig, die FitNesse-Instanz nach einem Testlauf nicht zu beenden, sondern auf dem

Listing 3

```
public class
    CustomerDataProvider
        implements
    DataProvider<Customer> {
    @Inject
    private
        RemoteFileManager
            fileManager;
    @Inject
    private MyConfig config;

    @Override
    public Customer load() {
        ...
    }
}
```

Build-Server laufen zu lassen, um die Ergebnisse zu präsentieren. Trickreicher ist aber das Konvertieren der Testergebnisse, etwa ins JUnit-Format, was von Hudson ausgelesen und als Grafik dargestellt werden kann (Abb. 9). FitNesse legt die Ergebnisse intern als XML-Datei ab, die wir nach dem Testdurchlauf mittels XSL ins Zielformat transformieren. Auf dieselbe Art erfolgt auch die Archivierung der Testergebnisse in einer Datenbank, um diverse Excel-Berichte zu erstellen. Die FitNesse-Ergebnisdatei enthält nur grundlegende Informationen, weitere Infos muss man sich über das Restful API abholen, was mit XSL und diversen Transformationsschritten leicht möglich ist und ein weiterer Grund die FitNesse-Instanz nach dem Testlauf nicht zu beenden (Abb. 10).

Fazit und Ausblick

Viele Softwareprojekte erfordern immer kürzere Releasezyklen und damit auch häufig wiederkehrende Tests. Für die Bewertung von Fortschritt und Reifegrad bieten sich fachliche Akzeptanztests an, auch wenn diese wohl in den wenigsten Projekten vom Fachbereich selbst geschrieben oder ergänzt werden. Dennoch stellen sie unserer Erfahrung nach eine gute Basis für die projektweite Kommunikation dar und helfen dabei, sich nicht „selbst zu belügen“.

Dem gegenüber stehen Aufwand und Kosten für das initiale Aufsetzen und Pflegen einer Testumgebung. Wie in der Softwareentwicklung auch, kann man diese zwar nicht eliminieren, aber überschaubar halten: gutes Design, wiederverwendbare fachliche und technische Bausteine (z. B. für Last- und Performancetests) sowie einfache und gut dokumentierte Handhabung sind einige wichtige Voraussetzungen dafür. Ideal sind auch minimale Abhängigkeiten zur Applikation im Test, damit die Tests auf interne Veränderungen der Applikation im Zuge der Weiterentwicklung weniger anfällig reagieren.

Um die Aufwände in Form von Zeit- und Kostenersparnissen durch früh gefundene Fehler und weniger kommunikative Missverständnisse wieder hereinzubekommen, ist unserer Erfahrung nach eine breite Akzeptanz bei allen Projektbeteiligten unabdingbar. Tester sollten keinesfalls mit „ihrem Tool“ alleine gelassen werden, was bei schwergewichtigen Tools und starker Kapselung tendenziell leichter passieren kann. Der Open-Source-Gedanke und schnelle Erfolgserlebnisse durch gemeinsame Nutzung und kurze Roundtrips sind dabei sehr hilfreich.

Insbesondere für Applikationen ohne Benutzeroberfläche bietet der Wikiansatz von FitNesse einen brauchbaren „GUI-Ersatz“, um eine verständliche Beschreibung der Vorgänge im System zu erhalten. Der Community-Support ist recht gut. Bezüglich Wartbarkeit und Pflege sind Textdateien mit Wikisyntax zwar nicht optimal, wenn aber von vornherein eine sinnvolle fachliche Abstraktion gewählt wird, sind hier auch nicht allzu viele Änderungen nötig, die wir dann mit klassischem Suchen und Ersetzen durchführen.

Die eigentlichen Testbausteine von FitNesse zu trennen, lohnt sich in jedem Fall, allein schon deshalb, weil

man bei Refactoring-Arbeiten an Testbausteinen keine unerwarteten Seiteneffekte auf die Wikiseiten zu erwarten hat. Tut man dies, hält man sich auch eine spätere Adaption anderer Tools für wikibasierte Akzeptanztests offen, z. B. Green Pepper [16] und Concordion [17]. Ersteres ist kommerziell und setzt auf Atlassian Confluence und Jira [18] auf, während Concordion ebenfalls lizenzkostenfrei unter APL 2.0 daherkommt. Neben den QA-getriebenen Anforderungen sollte unbedingt auf Entwicklerakzeptanz und einfache Einbindung in den Continuous-Integration-Prozess geachtet werden.

Wir glauben, dass sich diejenigen fachlichen Akzeptanztests, die als integraler Bestandteil kontinuierlicher Integration erschaffen werden, nicht nur in agilen Projektumfeldern etablieren werden, sondern zukünftig mehr und mehr bei größeren Festpreisprojekten oder Mischformen zum Einsatz kommen. Diese werden dann nicht ausschließlich von QA-Mitarbeitern erstellt und erst zum Abgabetermin geliefert, sondern kollaborativ erschaffen und auch kontinuierlich an den Kunden kommuniziert. Eine ausführbare Spezifikation als Beweis der gelieferten Funktionalität hilft beiden Seiten leichter Vertrauen zu gewinnen und Umfang und Qualität der geleisteten Arbeit gegenseitig zu schätzen.



Marcel Sauer ist Java-Entwickler im Bereich JEE, Testautomatisierung und agile Entwicklung. Er hat FitNesse bereits in mehreren Projekten erfolgreich eingesetzt und ist davon überzeugt, dass kollaboratives Testen nicht nur den Team-Spirit fördert, sondern auch entscheidend für den Erfolg von Softwareprojekten ist.



Jonas Kilian ist Senior Consultant bei der Flavia IT-Management GmbH. Er hat sich auf kontinuierliche Integration in größeren Entwicklungsumfeldern spezialisiert und ist Committer für das Open-Source-Projekt Molybdenum, ein Firefox-Plug-in zum browserbasierten Testen.

Links & Literatur

- [1] <http://fitnesse.org>
- [2] <http://fitnesse.org/FitNesse.UserGuide.SIIM.SlimProtocol>
- [3] <http://sourceforge.net/projects/fitlibrary/>
- [4] z. B. <http://kdiff3.sourceforge.net/>
- [5] <http://xopus.com/>
- [6] <http://www.xmlmind.com/xmleditor/>
- [7] <http://xstream.codehaus.org/>
- [8] <http://code.google.com/p/google-guice/>
- [9] <http://www.springsource.org/>
- [10] <http://www.sonarsource.org>
- [11] <http://jquery.com/>
- [12] <http://fitnesse.org/FitNesse.UserGuide.SymbolicLinks>
- [13] <http://sourceforge.net/projects/fitnessetools/>
- [14] <http://ant.apache.org>
- [15] <http://HUDSON-CI.ORG>
- [16] <http://www.greenpeppersoftware.com>
- [17] <http://www.concordion.org>
- [18] <http://www.atlassian.com/software/confluence>