Top-FIT

Automatisierte Akzeptanztests mit FIT

Stefan Roock

Akzeptanztests testen ein Softwaresystem aus Anwendersicht auf Akzeptanzkriterien. Das ist ein alter Hut. Dass man einen Großteil dieser Tests – nämlich mindestens den Test jeglicher Funktionalität – automatisieren kann und sollte, ist da schon eine etwas neuere Idee. Und dass mit dem Open-Source-Werkzeug FIT dafür ein einfaches und mächtiges Instrument verfügbar ist, ist noch viel zu wenig bekannt. Idealerweise werden auch die Fachexperten in das Erstellen der Tests involviert, daher ist dieser Artikel so geschrieben, dass er auch für Fachexperten verständlich sein sollte.

Die Bedeutung von Akzeptanztests

Mit Akzeptanztests* wird die Systemfunktionalität aus Sicht der Anwender überprüft. In jedem Softwareprojekt werden Akzeptanztests durchgeführt. In den meisten Fällen setzen sich dazu ausgewählte Fachexperten an das System und testen manuell. Teilweise werden die Tests ad hoc, ohne vorherige Planung durchgeführt. Mindestens bei größeren Systemen ist jedoch eine Planung notwendig, sodass ein Testkonzept erstellt wird.

Das manuelle Testen bedeutet bei den modernen iterativen (agilen) Vorgehensweisen, dass die Fachexperten sehr häufig testen müssen. Da mit jedem Inkrement vorhandene Funktionalität prinzipiell in Mitleidenschaft gezogen werden kann, müsste eigentlich nach jeder Iteration das gesamte System getestet werden. Bei Releasezyklen von 1 bis 3 Monaten führt das schnell zu hohen Testaufwänden und zu Frustration bei den Fachexperten – wer hat schon Lust, alle sechs Wochen dieselbe Systemfunktionalität zu testen?

Daher wäre es am einfachsten, wenn Akzeptanztests möglichst weit automatisiert werden. Die sowieso notwendigen Testkonzepte bieten dafür eine geeignete Grundlage, sodass für die Automatisierung nur geringer Mehraufwand entsteht. Und für die Automatisierung selbst bietet sich FIT an.

Was ist FIT?

FIT steht für *Framework for Integrated Tests* und ist ein frei verfügbares Open-Source-Werkzeug, das den Ansatz verfolgt, technische Testausführung und Testdaten voneinander zu trennen. Die Testdaten werden in Tabellen beschrieben, die Testausführung wird über sogenannte Fixtures programmiert (dazu unten mehr).

Zahl1	Zahl2	Summe	Differenz
3	2	5	1
0	0	0	0

Tabelle 1: Testdaten für einen Taschenrechner

* Neben *Akzeptanztest* sind auch folgende Begriffe geläufig: *Funktionstest* – weil die Funktionalität des Systems getestet wird – sowie *Anwender-/Kundentest* – weil aus Sicht der Anwender/Kunden getestet wird.

Beginnen wir mit einem ganz einfachen Fall. Tabelle 1 testet einen ganz einfachen Taschenrechner, der nur addieren und subtrahieren kann. Die erste Zeile der Tabelle stellt die Spaltenüberschriften dar. Aus ihnen können wir ersehen, was die Eingabedaten (*Zahl1*, *Zahl2*) und was die erwarteten Ausgabedaten (*Summe*, *Differenz*) sind.

Dass die Spalten *Zahl1* und *Zahl2* die Eingabewerte und *Summe* und *Differenz* die Ausgabewerte der Systemfunktion sind, ist in der Tabelle nicht eindeutig erkennbar. Wir als Menschen können die Rolle der Spalten erraten, aber FIT benötigt einen kleinen Hinweis: Die Ausgabewerte enden mit einem Klammernpaar, wie Tabelle 2 zeigt.

Zahl1	Zahl2	Summe()	Differenz()
3	2	5	1
0	0	0	0

Tabelle 2: Testdaten für einen Taschenrechner mit Markierung der Ausgabewerte

Ein größeres System hat eine Vielzahl von Systemfunktionen, sodass wir auch noch angeben müssen, welcher Bereich überhaupt getestet werden soll. Das tun wir in der ersten Tabellenzelle. FIT akzeptiert so beschriebene Tests, wenn sie als Tabellen in HTML-Seiten (z. B. erstellt mit MS-Word) vorliegen. Text außerhalb von Tabellen ignoriert FIT, sodass zwischen den einzelnen Tabellen Erläuterungen zu den Tests stehen können. Häufig werden so die zugehörigen Anforderungen beschrieben. Im Browser sieht unsere Testdefinition für den Taschenrechner also wie in Abbildung 1 gezeigt aus.

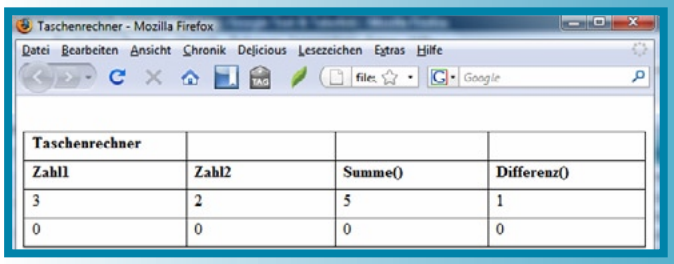


Abb. 1: Browser-Darstellung der Testdefinition

Als Ergebnis liefert FIT HTML-Seiten, die den Tests sehr ähnlich sehen. Lediglich die Ergebniszellen wurden eingefärbt: Die Zelle ist grün, wenn das erwartete Ergebnis mit dem berechneten Ergebnis übereinstimmte. Die Zelle ist rot, wenn das berechnete Ergebnis vom erwarteten Ergebnis abwich. Außerdem wird dann der tatsächlich berechnete Wert mit ausgegeben. So entsteht für unser Beispiel das Ergebnis aus Abbildung 2.

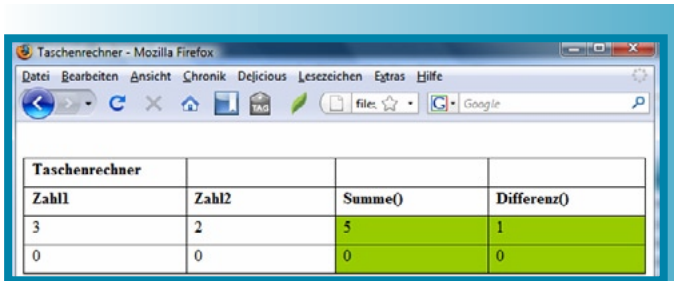
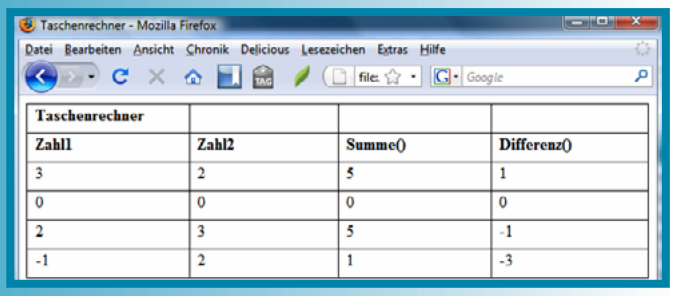


Abb. 2: Farbige Ergebnisanzeige

Tabellen für neue Einsichten

Interessanterweise finden wir sehr schnell Lücken in Tests, wenn die Tests in Tabellenform vorliegen. So ist sofort klar, dass unser Beispieltest keine Aussage darüber macht, wie sich das System bei negativen Zahlen verhält. Das können wir ganz einfach herausfinden, indem wir den Test wie in Abbildung 3 gezeigt erweitern.

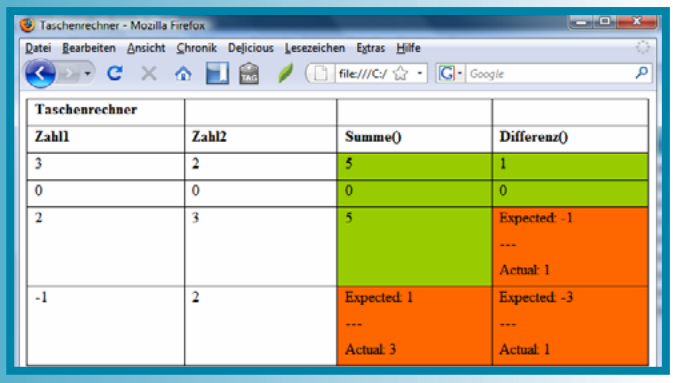


Taschenrechner			
Zahl1	**Zahl2**	**Summe()**	**Differenz()**
3	2	5	1
0	0	0	0
2	3	5	-1
-1	2	1	-3

Abb. 3: Erweiterung

Führen wir den erweiterten Test aus, bekommen wir vielleicht das Ergebnis (*Expected* zeigt den erwarteten Ergebniswert, *Actual* den tatsächlichen Ergebniswert) aus Abbildung 4.

Aus den angezeigten Fehlern beim Test kann man nicht nur ablesen, dass der Taschenrechner nicht korrekt mit negativen Zahlen umgeht. Man kann außerdem erahnen, wo das Problem liegt: Der Taschenrechner interpretiert auch negative Zahlen immer als positive Zahlen und liefert auch immer nur positive Zahlen.



Taschenrechner			
Zahl1	**Zahl2**	**Summe()**	**Differenz()**
3	2	5	1
0	0	0	0
2	3	5	Expected: -1 --- Actual: 1
-1	2	Expected: 1 --- Actual: 3	Expected: -3 --- Actual: 1

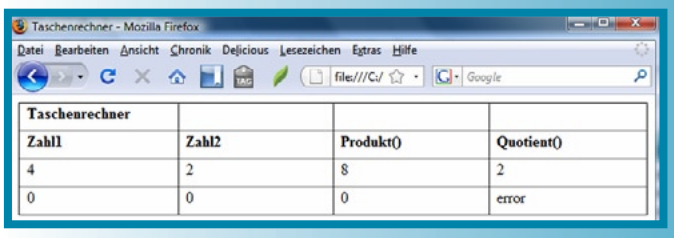
Abb. 4: Ergebnisse der Erweiterung

Umgang mit Fehlern

Wenn wir jetzt zusätzlich Multiplikation und Division testen wollen, werden wir mit Fehlerfällen konfrontiert. Division durch 0 ist nicht erlaubt. Für diesen Fall schreiben wir einfach *error* als erwarteten Wert in die Tabelle (s. Abb. 5). Tritt beim Ausführen der Funktion ein Fehler** auf, stimmen erwarteter und tatsächlicher Wert überein und der Test ist erfolgreich. Tritt kein Fehler auf, schlägt der Test fehl.

Damit unser Beispieltest auch tatsächlich automatisch ausgeführt werden kann, muss noch die Verbindung zum System hergestellt werden. Dies erfolgt über die sogenannten *Fixtures*, die programmiert werden müssen. Wie das funktioniert, beschreibt der nächste Abschnitt.

** Technischer Hinweis: Fehler bedeutet z. B. in Java das Werfen einer Exception.



Taschenrechner			
Zahl1	**Zahl2**	**Produkt()**	**Quotient()**
4	2	8	2
0	0	0	error

Abb. 5: Fehlerbehandlung

Erstellung der Test-Spezifikationen

FIT selbst ist mit Tabellen in HTML-Dateien zufrieden – wie die da reingekommen sind, ist FIT egal. Glücklicherweise können die meisten Werkzeuge heute nach HTML exportieren, sodass jeder sein Lieblingswerkzeug verwenden kann: von MS-Word und Excel über Open-Office bis hin zu Google-Docs ist alles möglich.

Die Test-Spezifikationen sollten mit in der Versionsverwaltung abgelegt werden, damit ein gegebenenfalls restaurierter Systemstand gleich die zugehörigen Tests dabei hat.

Für Entwickler: FIT-Tests ans System anbinden

Wir haben bisher gesehen, wie man FIT-Tests schreiben kann. Damit ist aber noch nicht die Frage beantwortet, wie sie an das zu testende System angebunden werden. Woher weiß FIT, welche Methoden an welchen Klassen aufzurufen sind?

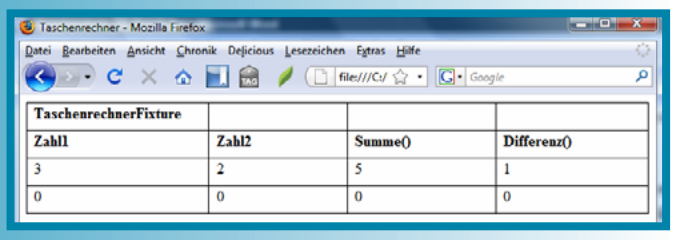
Für die Anbindung der Tests an das System werden sogenannte *Fixtures* verwendet. Das sind spezielle Klassen, die von passenden Fixture-Oberklassen abgeleitet werden. Die Fixture-Klasse, die wir für das oben gezeigte Beispiel mit Addition und Subtraktion programmieren müssen, leiten wir von **`ColumnFixture`** ab:

```
public class TaschenrechnerFixture extends ColumnFixture {
  public int Zahl1, Zahl2;
  private Taschenrechner taschenrechner;
  public int Summe() {
    return taschenrechner.addiere(Zahl1, Zahl2);
  }
  public int Differenz() {
    return taschenrechner.addiere(Zahl1, Zahl2);
  }
}
```

Für jeden Eingabewert (hier **`Zahl1`** und **`Zahl2`**) muss es ein gleichnamiges öffentliches Feld (**`public`**) geben. Für jeden Ausgabewert muss eine gleichnamige öffentliche Methode ohne Parameter existieren (hier **`Summe`** und **`Differenz`**). Hier entsteht meistens ein Konflikt mit den Code-Richtlinien der gewählten Programmiersprache. In Java sind z. B. groß geschriebene Feld- oder Methodennamen unüblich. Es empfiehlt sich hier, den Konflikt zugunsten des Tests zu lösen und von den Code-Richtlinien abzuweichen. So bleiben die Tests gut lesbar und da kein Programmcode auf Fixtures zugreift, wird durch die Abweichung von den Code-Richtlinien auch kein anderer Code korrumpiert.

Damit der Test mit unserer Fixture-Klasse arbeitet, müssen wir den Test anpassen: in der ersten Zelle der Tabelle (s. Abb. 6) muss der Klassenname der Fixture stehen, einschließlich Package-Namen.

Der Ablauf beim Ausführen des Tests sieht folgendermaßen aus:



TaschenrechnerFixture			
Zahl1	Zahl2	Summe()	Differenz()
3	2	5	1
0	0	0	0

Abb. 6: Taschenrechner mit Fixture

- Man startet den *FileRunner* aus FIT und gibt ihm den Namen der HTML-Testdatei mit.
- Der *FileRunner* liest die HTML-Datei ein und geht für jede Tabelle folgende Schritte durch:
  - Erzeuge ein Objekt der Fixture-Klasse, die in der ersten Tabellenzelle angegeben ist.
  - Übertrage je Zeile alle Eingabewerte in die gleichnamigen öffentlichen Felder des Fixture-Objektes.
  - Rufe je Zeile für alle Ausgabewerte die gleichnamigen öffentlichen, parameterlosen Methoden des Fixture-Objektes auf.
  - Vergleiche die Rückgabewerte der aufgerufenen Methoden mit den erwarteten Werten im Test. Markiere die Zelle Grün, wenn beide Werte übereinstimmen. Markiere die Zelle Rot und schreibe erwarteten und tatsächlichen Wert in die Zelle, wenn die beiden Werte nicht übereinstimmen.

Für Entwickler: Fachlogik oder Oberfläche testen

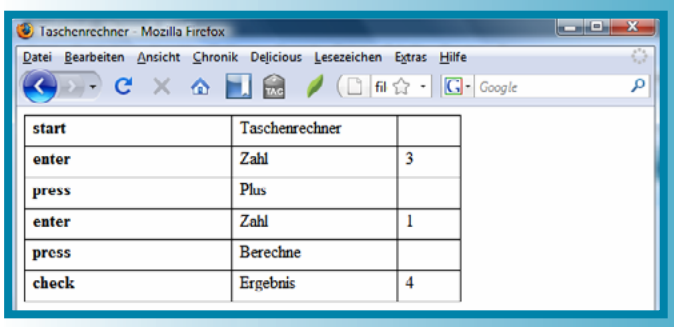
Im Taschenrechner-Beispiel testen wir die Fachlogik des Systems über ein entsprechendes API (Klasse `Taschenrechner`). Die Darstellung in der angegebenen Tabellenform legt diese Art des Testens auch nahe.

FIT unterstützt noch eine andere Art der Tests, sogenannte Action-Tests (über Action-Fixtures). Diese orientieren sich an der typischen Handhabung eines Systems und kennen Aktionen wie das Starten eines Systemteils, das Eingeben von Werten, das Auslösen von Aktionen sowie das Auslesen von Werten. Abbildung 7 zeigt einen typischen Taschenrechner.

Abb. 7: Typischer Taschenrechner

Für diesen Taschenrechner könnte ein Test mit Action-Fixture wie in Abbildung 8 aussehen. Dieser Test orientiert sich an der typischen Handhabung eines Taschenrechners. Ob der Test tatsächlich über die Benutzungsoberfläche des Taschenrechners testet oder auch das Taschenrechner-API verwendet, entscheiden wir bei der Programmierung der Fixture.

Wenn wir den Test tatsächlich über die Benutzungsoberfläche durchführen wollen, benötigen wir zusätzliche Werkzeuge zur Steuerung der Anwendung über die Oberfläche. Für Webanwendungen kommt z. B. Selenium RC in Frage und Java-Swing-Oberflächen kann man mit Jemmy kontrollieren. Abbildung 9 zeigt beide Varianten des FIT-Testens im Überblick.



start	Taschenrechner	
enter	Zahl	3
press	Plus	
enter	Zahl	1
press	Berechne	
check	Ergebnis	4

Abb. 8: Action-Test

Wann man Fachlogik testet und wann man lieber über die Benutzungsoberfläche geht, ist eine altbekannte Streitfrage. Zunächst ist zu beachten, dass die Frage, ob Fachlogik oder Oberfläche, der Ansatzpunkt für Tests ist, unabhängig von der Frage ist, ob man Column- oder Action-Fixtures verwendet. Man kann Column-Fixtures auch auf die Oberfläche zugreifen lassen, genauso wie man Action-Fixtures gegen das API des Systems programmieren kann. Generell haben Column-Fixtures gegenüber Action-Fixtures den Vorteil, dass sie beim Testen von Berechnungsfunktionen zu viel kürzeren Tests führen. Wenn man allerdings Abläufe testen muss, führen Action-Fixtures zu kürzeren Tests.

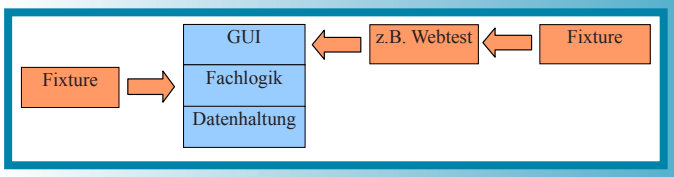


Abb. 9: Varianten des FIT-Testens im Überblick

Ähnlich ambivalent verhält es sich bei der Frage, ob man Fachlogik oder Oberfläche testen sollte. Das Testen der Fachlogik ist in der Regel viel einfacher als das Fernsteuern der Anwendung über die Benutzungsoberfläche. Außerdem laufen die Fachlogik-Tests in der Regel viel schneller durch. Dafür führen die Tests über die Benutzungsoberfläche zu einer höheren Testabdeckung – die Benutzungsoberfläche selbst bleibt bei Fachlogik-Tests ungetestet. Diesen Vorteil kann man wahlweise auch als Nachteil ansehen. Mitunter haben wir Schwierigkeiten, eine bestimmte Funktion über die Fachlogik zu testen. Das bedeutet fast immer, dass die Logik in der Benutzungsoberfläche liegt und da gehört sie nicht hin. Das Testen über die Fachlogik deckt also immer wieder Entwurfsschwächen auf und fordert uns auf, entsprechende Refactorings durchzuführen.

Für Entwickler: FIT einführen

Für das Warmwerden mit FIT empfiehlt sich in der Regel das Testen von Fachlogik mit Column-Fixtures. Hier ist der Aufwand am geringsten und trotzdem kann ein Großteil des Systems getestet werden. Schrittweise nimmt man dann Action-Fixtures hinzu, um die Abläufe zu testen. Und dann sollte das Testen über die Benutzungsoberfläche mit denselben Tests möglich sein. Nur die Fixture-Implementierung sollte ausgetauscht werden müssen, sodass man eine Fixture-Architektur wie in Abbildung 10 erhält.

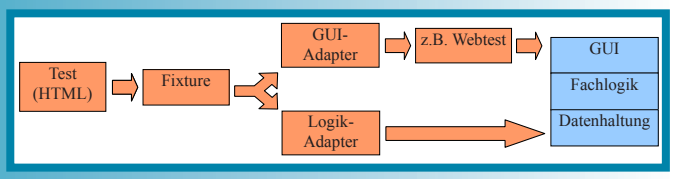


Abb. 10: Fixture-Architektur

Neben diesem technischen Aspekt verdient die Zusammenarbeit mit den Fachexperten eine nähere Betrachtung. FIT ist prädestiniert dazu, dass Nicht-Entwickler die Tests formulieren. Bevor man allerdings beginnt, Fachexperten in die Test-Formulierung einzubeziehen, sollten die Entwickler FIT technologisch im Griff haben. Das bedeutet, dass sie einige FIT-Tests selbst erfolgreich erstellt und mit Fixtures ans System angebunden haben sollten. Sie sollten sich Skripte für die Ausführung der Tests erstellt und die Testausführung in die Build-Umgebung integriert haben. Wenn die technischen Aspekte reibungslos funktionieren, ist es Zeit, Fachexperten hinzu zu ziehen. Schließlich sollten diese am besten wissen, was die Akzeptanzkriterien für die Systemfunktionen sind. Die Zusammenarbeit kann dabei flexibel je nach technischer Kompetenz der Fachexperten gestaltet werden. Im Idealfall schreiben die Fachexperten die Tests vollständig selbst und die Entwickler setzen nur noch die konkreten Namen der Fixture-Klassen an den Anfang jeder Tabelle. Mindestens für die Anfangsphase ist es jedoch häufig realistischer, dass die Entwickler die Tests in ihrer Grundstruktur erstellen und die Fachexperten die Testdaten liefern oder ergänzen.

Test-First-Vorgehen

Mit FIT kann man auch nach dem Test-First-Ansatz vorgehen. Man erstellt zuerst den FIT-Test für eine Funktion und implementiert diese erst danach. Damit wird der FIT-Test zu einer ausführbaren Anforderungsspezifikation. So bekommen wir bereits bei der Formulierung der Anforderungen hilfreiches Feedback: Häufig stellen wir nämlich beim Schreiben der FIT-Tests fest, dass es ganz viele ungeklärte Fragen in der Anforderung gibt. Test-First-Testen mit FIT führt damit zu besseren Anforderungen. Gleichzeitig haben die Entwickler ein gutes Maß für die Frage, wann die Funktion fertig implementiert ist: wenn der FIT-Test durchläuft.

FIT gegen den Rest der Welt

Heute ist eine Vielzahl von kommerziellen und freien Werkzeugen für Akzeptanztests verfügbar. Die meisten dieser Werkzeuge erlauben das elegante Aufzeichnen von Tests, einfach indem die Anwendung durchlaufen wird. So verlockend dieses Vorgehen auf den ersten Blick erscheint: es ist auf Dauer von Nachteil. Zum einen sind die aufgezeichneten Tests fragil gegenüber Änderungen an der Benutzungsoberfläche und müssen dann immer wieder neu aufgezeichnet werden. Zum anderen führt die Aufzeichnung in großen Systemen zu unzähligen Redundanzen in den Tests. Die höchst unangenehme Folge ist, dass kleine Änderungen am System irrsinnig teuer werden, weil unzählige Tests angepasst werden müssen. Und nicht zuletzt ist mit aufgezeichneten Tests natürlich kein Test-First-Vorgehen möglich.

Immerhin haben die meisten Tools den Fluchtweg schon eingebaut, auf dem man diesen Problemen entgehen kann. Man kann die aufgezeichneten Tests in der Regel als Quelltext generieren lassen und dann nachbearbeiten. So können Redundanzen entfernt und die Tests robuster gegen Änderungen an der Benutzungsoberfläche gestaltet werden. Leider schließt man damit die meisten Fachexperten wieder von der Gestaltung der Akzeptanztests aus.

Zusammenfassung

Das Beispiel hat nur einen kleinen Ausschnitt der Möglichkeiten von FIT gezeigt. Über weitere Fixture-Klassen eröffnen sich vielfältige weitere Möglichkeiten, Tests zu formulieren. Die gezeigten Varianten mit Column- und Action-Fixtures sind aber bereits sehr mächtig. Das bedeutet insbesondere, dass man FIT-Tests mit sehr wenig Aufwand in sein Projekt einführen kann.

Auch für bereits laufende Projekte eignet sich FIT sehr gut. Es ist beispielsweise sehr viel einfacher, ein ungetestetes System mit FIT-Tests auszustatten als mit Unit-Tests. Mit FIT-Tests kann man mit wenig Aufwand eine sehr hohe Testabdeckung erreichen. So kann man z. B. dieses FIT-Test-Sicherheitsnetz benutzen, um notwendige Refactorings an seinem System durchzuführen. Die fehlenden Unit-Tests können dann schrittweise ergänzt werden.

Und sobald die Entwickler FIT im Griff haben, sollten sie den Fachexperten die Möglichkeit geben, ihre Anforderungen in FIT-Tests zu beschreiben.

Es gibt zwei prominente Erweiterungen von FIT. Die erste ist FitLibrary, die viele nützliche weitere Fixtures mitbringt. Die zweite Erweiterung ist FitNesse, das ein Wiki integriert, sodass die Tests direkt aus dem Browser heraus editiert und ausgeführt werden können. Das englischsprachige Buch [MuCu05] über FIT behandelt diese beiden FIT-Aufsätze mit. Außerdem ist dieses Buch auch für Kunden und Anwender als Einführung geeignet, da der erste Teil technikfrei ist. Eine entwicklerorientierte Einführung in die testgetriebene Entwicklung bietet [West05]. Dieses Buch legt zwar den Schwerpunkt auf Unit-Tests mit JUnit, am Ende jedoch findet sich zusätzlich eine Einführung in FIT.

Literatur und Links

[FIT] Web-Site zu FIT mit Beispielen und Download-Möglichkeiten, `http://fit.c2.com`

[FitNesse] Web-Site zu FitNesse mit Beispielen und Download-Möglichkeiten, `http://www.fitnesse.org`

[MuCu05] R. Mugridge, W. Cunningham, Fit for Developing Software, Prentice Hall, 2005

[West05] F. Westphal, Testgetriebene Entwicklung mit JUnit & FIT, dpunkt Verlag, 2005

Dipl.-Inform. Stefan Roock ist Senior IT-Berater bei der akquinet it-agile GmbH in Hamburg. Er verfügt über mehrjährige Erfahrung aus agilen Softwareprojekten (Scrum, eXtreme Programming, Feature Driven Development) als Coach, Trainer, Scrum-Master/Facilitator und Entwickler. Darüber hinaus hat er zahlreiche Artikel und Tagungsbeiträge über agile Softwareentwicklung verfasst und ist Autor der Bücher „Software entwickeln mit eXtreme Programming“ und „Refactorings in großen Softwareprojekten“.
E-Mail: stefan.roock@it-agile.de.