AGILE AKZEPTANZTESTS: VISION, PRAXIS UND WERKZEUGE

Die Vision hinter agilen Akzeptanztests ist so einfach wie verlockend: Statt umfangreiche Anforderungsdokumente in mehrdeutiger Prosa zu verfassen und sie den Entwicklern als Pflichtlektüre aufzuzwingen, erfassen wir die fachlichen Anforderungen in Form einfacher und doch realitätsnaher Beispiele. Diese Beispiele werden anschließend direkt, d. h. ohne Übersetzung durch einen Tester, als automatisierte Testfälle vor, während und nach der Programmierung verwendet. In der Praxis trifft diese schöne Idee jedoch auf zahlreiche Schwierigkeiten. Dieser Artikel beschreibt sowohl die Theorie des Ansatzes als auch die Erfahrungen des Autors beim Versuch, der beschriebenen Vision in Projekten möglichst nahe zu kommen.

▸ der autor

Johannes Link
(E-Mail: business@johanneslink.net)
ist unabhängiger Coach für agile Softwareentwicklung, Buchautor und Redaktionsmitglied von OBJEKTspektrum.

Agile Akzeptanztests

„Quality is value to some person", schreibt Gerry Weinberg (vgl. [Wei93]) und bricht mit der Vorstellung, dass Qualität etwas Absolutes ist. Tatsächlich muss Software die Anforderungen der Kunden erfüllen, andernfalls verkommt jegliche Qualitätsbewertung zur reinen Makulatur.

In der Praxis wünschen sich Kunden mindestens dreierlei von einem System:

- Es soll die fachlichen Aufgaben korrekt erfüllen – dies nennt man die funktionalen Anforderungen.
- Es soll nicht-funktionale Eigenschaften aufweisen, etwa im Bereich der Performance oder der Skalierbarkeit.
- Der Nutzen einer Software soll die Kosten für ihre Erstellung und Pflege maßgeblich übersteigen.

Agile Akzeptanztests haben sich aus dem entwickelt, was im *agilen* Vorgehen von *eXtreme Programming (XP)* (vgl. [Bec04]) ursprünglich *Customer Tests* genannt wurde: automatisierte Testfälle, die als Kriterium für die Erfüllung der Kundenerwartungen dienen können. Laufen diese Testfälle fehlerfrei, gilt die Software als *akzeptiert*. Agile Akzeptanztests konzentrieren sich daher überwiegend auf die fachliche Seite[1]). Wenn man – so die Idee – diese fachlichen Vorgaben in Form automatisch überprüfbarer Testfälle festhält, muss man nur noch am Ende eines Entwicklungszyklus die Testfälle starten und erhält einen unbestechlichen Bericht darüber, welche funktionalen Anforderungen erfüllt sind und welche nicht. Möchte man diese Form der Anforderungsspezifikation nicht zusätzlich, sondern an Stelle eines herkömmlichen Anforderungsdokuments benutzen, so muss man dafür sorgen, dass der Kunde bzw. dessen Fachexperten diese Tests schreiben, lesen und im Idealfall sogar verändern und ergänzen können. Bei der Verwendung entwicklungsnaher Testwerkzeuge, wie beispielsweise JUnit (vgl. [JUn]), ist diese Voraussetzung nicht gegeben. Andere Tools müssen her, und so entstanden Frameworks wie:

- Framework for Integrated Test (Fit) (vgl. [Fit])
- FitNesse (vgl. [FitN])
- GreenPepper (vgl. [Gre])
- Twist (vgl. [Tho])
- Concordion (vgl. [Pet09]

[1]) *Es ist durchaus denkbar, auch nicht-funktionale Aspekte in automatisierten Akzeptanztests festzuhalten, dies erfordert jedoch häufig eine andere Abstraktionsstufe als fachliche Tests.*

Das zu erstellende Programm soll in der Lage sein, eine Partie „Nimm 30" gegen einen menschlichen Gegner zu spielen und zu gewinnen. Eine Partie beginnt mit 30 Streichhölzern auf einem Stapel. Die Spieler nehmen abwechselnd bis zu fünf Hölzer vom Stapel. Wer das letzte Hölzchen nimmt, hat gewonnen. Das Programm soll als Programmierschnittstelle (API) einer Java-Bibliothek zur Verfügung stehen.

Kasten 1: *Das Nimmspiel – Spezifikation der Anforderungen.*

Ein Beispiel mit Fit

Die einfache und gerade deshalb bestechende Idee hinter Fit und seiner Wiki-Variante FitNesse ist der Gedanke eines ausführbaren Anforderungsdokuments. Dieses Dokument kann beliebigen Text enthalten, wird jedoch an den entscheidenden Stellen um Tabellen ergänzt, die die automatisch ausführbaren Testbeschreibungen enthalten. Während die Urform von Fit nicht mehr aktiv weiterentwickelt wird, erfreut sich FitNesse einer regen Benutzergemeinde und erlaubt das Ansteuern von Applikationen in Java, .NET, C++, Delphi, Python, Ruby, Smalltalk und Perl.

Betrachten wir das Prinzip an einem Beispiel: ein Programm, das in der Lage sein soll, gegen einen menschlichen Gegner eine vereinfachte Variante des Nimmspiels zu spielen. Die Prosaspezifikation der Anforderungen ist in Kasten 1 wiedergegeben.

DAS NIMM-30 SPIEL

Ein neues Spiel beginnt mit 30 Hölzern.

Nimm 30		
Starte neues Spiel		
Check	Anzahl Hoelzer	30

Abb. 1: *Das erste Testszenario mit FitNesse.*

Die Anforderungen scheinen klar und wir beginnen mit dem ersten Testszenario (siehe Abb. 1). In Fit bezeichnet die erste Zeile typischerweise den Namen einer Anbindungsklasse, Fixture genannt. Eine solche Fixture-Klasse enthält Methoden, die

Das Nimm-30 Spiel

Ein neues Spiel beginnt mit 30 Hölzern.

Nimm 30		
Starte neues Spiel		
Check	Anzahl Hoelzer	30 *expected* 0 *actual*

Abb. 2: Fehlschlagendes Testszenario.

dann durch das Framework beim Ablauf der Tests aufgerufen werden. In unserem Fall sieht diese Klasse in etwa so aus[2]):

```
public class Nimm30 {
  public void starteNeuesSpiel() {...}
  public int anzahlHoelzer() {...}
}
```

Die Methoden der Fixture-Klasse ergeben sich aus dem Text der jeweiligen Tabellenzeile und werden meist von Hand erstellt. So wird aus „Starte neues Spiel" die Signatur public void starteNeuesSpiel() und „Check | Anzahl Hoelzer | 30" ruft beim Testlauf anzahlHoelzer() auf und vergleicht das tatsächliche Ergebnis mit dem erwarteten Wert „30".

Im Beispiel sorgen die Methoden-Implementierungen dafür, dass die entsprechen Objekte der Nimm-30-Bibliothek erzeugt und die entsprechenden Aufrufe an diesen Objekten durchgeführt werden. Fixture-Klassen sind daher nichts anderes als einfache Fassaden, die unsere Testsprache in die eigentliche Applikationssprache übersetzen. In vielen Fällen ist es sogar möglich, auf spezielle Fassaden ganz zu verzichten und direkt auf die Fachklassen zuzugreifen. Das verringert den Aufwand für die Automatisierung unserer Testszenarien erheblich.

Nimm 30		
Starte neues Spiel		
Check	Anzahl Hoelzer	30

Abb. 3: Erfolgreiches Szenario.

Die Ausführung der Testfälle übernimmt das Framework. Das „Reporting" der Testläufe sieht den ursprünglichen Testszenarien sehr ähnlich, lediglich die Information über Erfolg und Misserfolg eines Aufrufs der Fixture-Klasse wird hinzugefügt (siehe Abb. 2). In dem dort gezeigten Fall schlägt der Test fehl, da die Anbindung – oder die Implementierung der Fachlogik – noch nicht den Anforderungen entspricht. Im Erfolgsfall erscheinen die erwarteten Werte in Grün (siehe Abb. 3).

Noch ist außer der Initialisierung nichts passiert, doch sobald man echte Spielszenarien in eine automatisierbare Form bringt, merkt man, dass gewisse Dinge in unserer Prosaspezifikation (Kasten 1) nicht eindeutig festgelegt waren. Zwei „Details" fallen recht schnell auf:

- Es wird nicht gesagt, welcher der beiden Spieler beginnt.
- Das Nehmen von null Hölzern ist nicht ausdrücklich verboten.

In unserem ausführbaren Beispiel müssen wir uns jedoch festlegen, damit wir ein deterministisch wiederholbares Testszenario erhalten (siehe Abb. 4).

Man erkennt den Versuch, die technischen Details aus den Tabellen fernzuhalten und in den Anbindungscode zu verbannen. Zu diesen Details gehören die Namen der Klassen und Methoden, aber auch notwendiger Initialisierungs- und Aufräumcode, der für die eigentliche Fachlichkeit keine Rolle spielt. Wir streben an, in den Testszenarien ausschließlich die Sprache des Kunden zu sprechen.

Um in Projekten diesem Ziel nahe zu kommen, benötigt man viel Geduld. Nicht nur beim Erstellen und Diskutieren der Beispielszenarien mit den Fachexperten, auch beim Erstellen des Anbindungscodes sind fehlende Informationen und komplexe Transformationen in der Fixture-Klasse oft ein Zeichen dafür, dass die Abstraktionen im Quelltext noch nicht denen der Kundenanforderungen entsprechen. Je dichter diese beiden Sprachen beieinander liegen, desto einfacher fällt die Automatisierung der Testszenarien. In diesem Sinne treiben fachlich orientierte Beispiele unseren Systementwurf in Richtung domänengetriebenes Design (*Domain Driven Design(DDD)* (vgl. [Eva04]).

Organisation der Testszenarien

Alle genannten Werkzeuge bieten die Möglichkeit, Testszenarien in Suiten zusammenzufassen und mehrfach aufgerufene Spezifikationsteile auszulagern. Dass eine Unterteilung und nachvollziehbare Organisation der Testfälle sinnvoll ist, ergibt sich aus dem Wunsch nach ständiger Erweiterung und Anpassung erfolgreicher Software. Die Veränderung der Software erfordert auch die Veränderung der ausführbaren Spezifikation – dazu müssen sich alle betroffenen Teile dieser Spezifikation mit vertretbarem Aufwand ermitteln lassen.

Hier hilft nur ein gehöriges Maß an Ordnung: Je größer die Anzahl der Beispielszenarien ist, desto strenger und disziplinierter muss auf Ordnung geachtet werden. Doch nach welchen Kriterien ist eine Organisation von Testszenarien sinnvoll? In der Praxis finden sich mehrere:

- Strukturierung nach funktionalen Bereichen (z. B. Datenerfassung, Berechnungen, Reporting)
- Strukturierung nach Kunden

Check	Am Zug ist	der Mensch

Mensch nimmt	5	Hoelzer
Check	Am Zug ist	der Computer
Check	Computer nimmt	1
Check	Anzahl Hoelzer	24

Mensch nimmt	2	Hoelzer
Check	Computer nimmt	4

Reject	Mensch nimmt	0	Hoelzer

Abb. 4: Das ergänzte Szenario.

[2]) *„In etwa" deshalb, weil wir uns an dieser Stelle nicht um die konkreten Implementierungsdetails kümmern, sondern nur das Prinzip klar machen wollen.*

- Strukturierung entlang der zeitlichen Entwicklung, beispielsweise nach Iteration und den darin umgesetzten User-Storys

Alle drei Ansätze weisen inhärente Nachteile auf. Eine Strukturierung nach funktionalen Bereichen hört sich gut an, bis man merkt, dass dies in einem rein hierarchischen Gliederungsschema nicht eindeutig machbar ist und viel Restrukturierungsbedarf erfordert. Die Strukturierung nach Kunden führt zwangsläufig zu Duplikation, wenn man sich in einem Produkt bewegt, und diese Duplikation führt zu Mehraufwänden bei der Veränderung von Features.

Die Gliederung von Testszenarien entlang der zeitlichen Achse und den umgesetzten User-Storys passt wunderbar mit der inkrementellen Entwicklung agiler Methoden zusammen, bereitet jedoch bei funktionalen Umbauten das Problem, alle betroffenen Szenarien bestimmen zu müssen. Die ideale Lösung bestünde in unterschiedlichen Sichten auf die gleiche Basis an Suiten – manche Werkzeuge ermöglichen dies durch frei wählbare *Tags*. Aber auch hier ist eine hohe Disziplin und Konsistenz beim Erstellen und der Pflege der Szenarien notwendig. Häufig gelingt das nur, wenn sich eine Person für die Organisation und Reorganisation aller Akzeptanztests besonders verantwortlich fühlt.

Specification by Example

Gojko Adzic beschreibt in seinem Buch „Bridging the Communication Gap" (vgl. [Adz09]) den oben skizzierten Ansatz, den er „Specification by Example" nennt. Praxisnahe Beispiele sind ein ideales Vehikel, um aus potenziell mehrdeutigen, fehlerhaften und unvollständigen Anforderungsdokumenten eindeutige Geschäftsregeln zu extrahieren. Dabei ist die Diskussion der Beispielszenarien mit dem Kunden mindestens so wichtig wie die Möglichkeit, diese Beispiele zu automatisieren und als Regressionstests zu verwenden.

Im Umkehrschluss bedeutet das, dass automatisierte Akzeptanztests, die von den Entwicklern allein erstellt werden, nur einen Teil des denkbaren Nutzens entfalten und im besten Fall die Erwartungen der Entwickler an das System überprüfen. Gerade dies beobachte ich immer wieder: Werkzeuge wie Fitnesse werden verwendet, um Integrations- und Systemtests auf einer technischen Ebene durchzuführen, nicht um die Erwartungen der Kunden zu belegen. Im besten Fall hat man damit eine zusätzliche Technologie und Indirektion bei seinen technisch orientierten Regressionstests erreicht. Im schlechtesten Fall hat man sich Testfälle mit viel Redundanz geschaffen, die zudem kaum durch automatisierte Refaktorisierungen wieder in Form gebracht werden können. Wäre man für diese Integrationstests bei JUnit & Co geblieben, hätte man als Entwickler zumindest die integrierte Entwicklungsumgebung auf seiner Seite.

Wie passen Akzeptanztests in den Prozess?

In der Theorie kann man sowohl die Werkzeuge als auch die Idee der beispielhaften Testszenarien in einem klassischen, nachgelagerten Testansatz verwenden. Die von mir empfohlene Alternative erweitert jedoch die Idee der testgetriebenen Entwicklung (vgl. [Bec03]) auch auf Akzeptanztests:

- Beispielszenarien werden *vor Beginn* der Feature-Implementierung erstellt, z.B. in dafür vorgesehenen Spezifikationsworkshops, an denen sowohl die Fachexperten als auch die Tester und das Entwicklungsteam teilnehmen.
- Die Anbindung der textuellen Szenarien an das System – in Fit sind das die Fixture-Klassen – ist der erste Schritt der Feature-Implementierung. Typischerweise nimmt der Aufwand hierfür im Laufe des Projekts immer weiter ab, wenn sich die Testsprache ihrer Vollständigkeit nähert.
- Nun werden die einem Feature zugeordneten Testszenarien Schritt für Schritt erfüllt, indem Code ergänzt, erweitert und verändert wird. Häufig entdeckt man dabei fachliche Lücken in den Szenarien, die dazu führen, dass die Beispiele ergänzt oder angepasst werden müssen – aber nur in Absprache mit den Fachexperten bzw. dem Tester!
- Ein Feature gilt dann als umgesetzt, wenn alle zugehörigen Beispielszenarien laufen und zudem der Implementierungscode alle sonstigen Qualitätsansprüche erfüllt – beispielsweise Unit-Test-Abdeckung, Redundanzfreiheit und Programmierrichtlinien – und wenn sowohl Entwickler als auch Experten keine fachlichen Lücken mehr sehen.

Dieses Vorgehen setzt voraus, dass das eingesetzte Testwerkzeug eine Vorab-Spezifikation der Testszenarien erlaubt. Diese Voraussetzung ist bei Capture&Replay-Werkzeugen nicht gegeben, da diese ein lauffähiges System zum Aufzeichnen der Testfälle voraussetzen.

Probleme aus der Praxis

Schon seit Jahren bringe ich die Ideen der automatisierten Akzeptanztests in meine Beratungsprojekte ein. Einige der dabei immer wieder auftretenden Probleme und Fallstricke möchte ich hier schildern.

Selten werden automatisierte Akzeptanztests von Beginn an eingesetzt. Im Nachhinein findet man jedoch oft eine Systemstruktur, die sich nicht gut für das Anbinden durch einfache Fixture-Klassen eignet. Typisch sind die enge Verzahnung der Logik mit der Benutzungsoberfläche und die mangelnde Abkapselung von Fremdsystemen. Vor der offensichtlichen Lösung, nämlich einer Refaktorisierung, die das System testbar macht, schrecken viele Teams jedoch zurück und betreiben ihre Akzeptanztests schließlich über die grafische Benutzungsoberfläche (GUI). Dies führt fast immer zu Problemen mit Stabilität, langer Laufzeit und hohen Wartungsaufwänden. Meine persönliche Faustformal lautet, dass nur ein kleiner Teil aller automatisierten Tests das echte GUI benutzen sollte: nur solche Tests, die die korrekte Verbindung von tatsächlicher Präsentation und Präsentationslogik als Testziel haben. Alles andere verringert die Agilität, statt sie zu erhöhen.

In eine ähnliche Wartungsfalle gerät man, wenn das Erstellen von Testszenarien überwiegend durch *Copy&Paste* bereits existierender Szenarien geschieht. Dabei entsteht häufig eine so große Menge an Redundanz, dass jede spätere Änderung der Software zu immensen Anpassungsaufwänden bei den Akzeptanztests führt. Dadurch wird die Entwicklung am Ende langsamer und nicht – wie erhofft – schneller. Die richtige Gegenmaßnahme besteht – ähnlich wie bei Programmcode – in der ständigen Eliminierung von Duplikation. Meist erfolgt dies durch die Erweiterung der Testsprache, in FitNesse z. B. durch eine neue Fixture-Klasse, die die Gemeinsamkeiten bestehender Szenarien in der Implementierung verbirgt und nur die Unterschiede außen sichtbar macht. Seltener sollte man die Modularisierungsmechanismen des jeweiligen Test-Frameworks benutzen, in FitNesse sind das include-Anweisungen. Solche Mechanismen machen fast immer aus einem für Laien lesbaren Text ein Computerprogramm und zerstören so die zu Grunde liegende Idee

eines ausführbaren Spezifikationsdokuments.

Ein weiterer typischer Fallstrick ist die Verwendung von Akzeptanztests an Stelle von Unit-Tests. Für Teams mit wenig Unit-Testing-Know-how scheint dies oft der ideale Ausweg aus der Zwickmühle von mangelnder Qualität und großem Zeitdruck zu sein, da „das Testen von außen ja ein anderer erledigen kann." Doch weder die Testabdeckung, noch die Robustheit, noch das Design des Codes lassen sich durch externes funktionales Testen in ähnlich starker Weise positiv beeinflussen wie durch fokussierte Entwicklertests. Wer einmal versucht hat, mit funktionalen Akzeptanztests eine Abdeckung jenseits der 80 % zu erreichen, weiß, wovon ich spreche. Einem testgetriebenen Team gelingt dies auf der Codeebene meist spielend.

Die Einbindung der Fachexperten wird durch die meisten Werkzeuge nur mäßig unterstützt. So lassen sich beispielsweise in FitNesse Testszenarien im Web einpflegen, aber sowohl die Wiki-Sprache als auch die Notwendigkeit, sich explizit mit Versionierung und technischen Details auseinanderzusetzen, erschwert die direkte Pflege der Szenarien durch „Nicht-Techies". Dieses Problem kann man deutlich abmildern, wenn die Erstellung von Testfällen gemeinsam vom Fachexperten mit einem Entwickler oder einem technisch versierten Tester geschieht. Insbesondere bei geeigneten Workflows für das Zusammenspiel von Tests und getestetem Code haben alle Werkzeuge noch Aufholbedarf.

Behaviour-Driven Development

Die oben genannten Werkzeuge sind nicht die einzigen, die den Fokus auf das gewünschte Verhalten eines Systems, anstatt seiner Implementierung legen. Als konsequente Fortführung der testgetriebenen Entwicklung kann man den Ansatz des *Behaviour-Driven Developments (BDD)* betrachten (vgl. [BDD]). Die darauf aufbauenden Werkzeuge[3]) ermöglichen ebenfalls die Formulierung von Testfällen in einer sehr fachnahen Sprache. Ein aktuelles Werkzeug ist beispielsweise „Cucumber" (vgl. [Cuc]), das via JRuby bzw. IronRuby die verhaltensgetriebene Entwicklung von beliebigen JVM- und .NET-Applikationen erlaubt.

Der Unterschied zwischen BDD- und Akzeptanztest-Werkzeugen liegt mehr in der technischen Ausführung als in der zu Grunde liegenden Philosophie: Häufig handelt es sich um interne domänenspezifische Sprachen (DSLs) einer dynamisch typisierten Skriptsprache, mit all den Vor- und Nachteilen, die diese mit sich bringen. Hinzu kommt bei BDD-Werkzeugen meist ein festgelegtes Schema zur Strukturierung von Szenarien: *Given-When-Then*. Ein Teil des obigen Szenarios könnte man in Cucumber folgendermaßen formulieren:

```
Scenario: Computer plays winning strategy
  Given there are 30 matches
  And it's the human's turn
  When the human takes 5 matches
  Then the computer takes 1 match
```

Das Beispiel wurde ins Englische übersetzt, um die Standard-Struktur eines typischen BDD-Szenarios deutlich zu machen. Solche Spezifikationen lesen sich im Englischen spürbar flüssiger als im Deutschen, auch wenn die Schlüsselwörter in zahlreichen natürlichen Sprachen zur Verfügung stehen.

Das Mapping der Sätze (there are 30 matches) auf Programmcode geschieht anhand regulärer Ausdrücke als Teil der so genannten step definitions. Im Beispiel sähe das etwa so aus:

```
Given /there are (.*) matches/ do |n|
  assert game.count_matches == n
end
```

Man beachte, dass sowohl die Spezifikation der Testschritte als auch die Szenarien selbst ausführbaren Ruby-Code darstellen. Daher werden – im Gegensatz zu Fit – kein zusätzlicher Parser und keine zusätzliche Ausführungsumgebung benötigt. Dies ist gleichzeitig eine Stärke und eine Schwäche: eine Stärke, weil es den Gesamtaufbau der Umgebung vereinfacht, und eine Schwäche, weil es den konzeptionellen Unterschied zwischen Entwicklertests und fachlich-orientierten Akzeptanztests verwischt.

Die Entscheidung zwischen BDD und Fit-artigen Akzeptanztests ist nicht einfach. Auf beiden Seiten entstehen zur Zeit neue Werkzeuge und die bestehenden entwickeln sich weiter. BDD tritt oft mit dem Anspruch an, testgetriebene Entwicklung auf Unit-Test-Ebene vollständig ersetzen zu können, während sich reine Akzeptanztestwerkzeuge als notwendige Ergänzung zu „TDD classic" verkaufen. In der Theorie spricht wenig gegen eine glückliche Heirat der Ansätze – außer der Tatsache, dass die Schnittmenge der praktizierenden Vertreter beider Ansätze nur klein ist.

[3]) *Eine Aufzählung findet sich unter http://behaviour-driven.org/Implementations.*

Fazit

Die Vision einer automatisch verifizierbaren Anforderungsspezifikation ist nicht nur verlockend, sondern wird auch von einer Reihe praxistauglicher Werkzeuge unterstützt. Mit diesen Werkzeugen kann man den Ansatz der testgetriebenen Entwicklung von Unit-Tests auf fachliche Akzeptanztests erweitern. Die Umsetzung der Vision ist dennoch schwierig, vor allem in einem „klassischen" Umfeld, in dem die Kommunikation zwischen Kunden, Fachexperten und Entwicklungsteam mehr als notwendiges Übel denn als große Chance begriffen wird.

Am meisten Erfolg im Bereich der automatisierten Akzeptanztests versprechen Teams, die neben qualitätsbewussten und kommunikationsfreudigen Entwicklern auch eine enge Beziehung zu den Fachexperten haben. Sinnvolle und lohnende Akzeptanztests lassen sich nicht nebenbei entwickeln und pflegen, sondern müssen im Zentrum der Anforderungserhebung und Dokumentation stehen. ■

Literatur & Links

[Adz09] G. Adzic, Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing, Neuri Limited, 2009

[BDD] Behaviour-Driven Development (BDD), siehe: behaviour-driven.org/

[Bec03] K. Beck, Test-Driven Development: By Example, Addison-Wesley, 2003

[Bec04] K. Beck, C. Andres, Extreme Programming Explained: Embrace Change, Addison-Wesley, 2nd Ed, 2004

[Cuc] Cucumber, siehe: www.cukes.info/

[Eva04] E. Evans, Domain Driven Design: Tackling Complexity in the Heart of Software, Addison-Wesley, 2004

[Fit] Framework for Integrated Test (Fit), siehe: http://fit.c2.com

[FitN] FitNesse, siehe: www.fitnesse.org/

[Gre] GreenPepper, siehe: www.greenpeppersoftware.com

[JUn] JUnit.org, siehe: www.junit.org/

[Mug05] R. Mugridge, W. Cunningham, Fit for Developing Software: Framework for Integrated Tests, Prentice Hall, 2005

[Pet09] D. Peterson, Concordion, 2009, siehe: www.concordion.org/

[Tho] ThoughtWorks, twist, 2009, siehe: www.studios.thoughtworks.com/twist-agile-test-automation

[Wei93] G.M. Weinberg, Quality Software Management: First-Order Measurement, Dorset House, 1993