S&S

entwickler magazin

entwickler magazin

www.entwickler-magazin.de

Juli/August 4.2013

Real Application Testing

Google Hacking

Finden, was nicht gefunden werden soll

Professionelle Service

Erholungsfähig?

Der agile Architekt

Scrum für Softwarearchitekten

SERIE **GWT**
MVP und Activities and Places

SERIE **OpenGL**
Landschaftsdarstellung

Warum guter Code und agile Tests ein schönes Paar sind

Drum prüfe, wer sich ewig bindet

What you get is what you see – was passiert jedoch, wenn man beim Testen nicht sieht, was man sehen wollte? Werden Tests und Entwicklung zu sehr getrennt, dann kann „zurück auf Los“ ein sehr weiter Weg sein. Besonders ein großer zeitlicher Abstand zwischen der Codeerstellung und den Tests kann die Entwicklungsgeschwindigkeit deutlich verlangsamen. Je später mangelhafte Codequalität offen gelegt wird, umso aufwändiger wird es, sie zu verbessern. Agiles Testen verfolgt daher den Ansatz, Testen und Entwickeln eng zu verzahnen.

von Maynard Harstick und Daniel Knapp

Automatisierte, einfache Tests von Anfang an reduzieren die Zahl komplexer Tests am Ende. Außerdem fungieren die Tests als Sicherheitsnetz für Entwickler beim Refaktorisieren. Dadurch bleibt der Code flexibel und kann kontinuierlich an die gegebenen Anforderungen angepasst werden. Eine hohe Testabdeckung wird so zum Schlüssel für bessere Softwarequalität.

Es klingt paradox und ist doch nur eine Frage von Ursache und Wirkung: Softwaretests stehen für einen Anfang, unabhängig davon, an welcher Stelle des Produktionsprozesses sie platziert sind. Das gilt zumindest dann, wenn sie nicht hundertprozentig fehlerfrei durchlaufen werden. Ein Test legt die funktionale Qualität der Software offen; verändern kann und soll er sie nicht. Insofern ist er die Instanz, die entscheidet, ob die Marschrichtung hin zur Auslieferung eingeschlagen wird oder zurück an den Start. Aber das ist nur die Wirkung. Die Ursache liegt nicht im Test selbst, es sei denn, er wurde fehlerhaft aufgesetzt. Sie liegt im Code. Und damit verlangt jeder Fehler im Code, an den Anfang der Programmierung oder gar der Konzeptionierung zurückzukehren.

Diese Rückkehr kann aufwändig und teuer werden, und das umso mehr, je größer das Delta zwischen Entwicklung und Eintreffen des Feedbacks wird. Gerade der klassische Ansatz, der Entwicklung und Test personell wie zeitlich komplett trennt, kann hier problematisch werden. Anstatt also das Testen einer eigenen – und autark agierenden – Abteilung zu überlassen, bringt die agile Vorgehensweise die Testexperten und die Entwickler zusammen, idealerweise ins gleiche Team. Denn der rein „externe“ Blick hat einige Nachteile: Es bedeutet Zeit und Aufwand, sich in Produktinkremente hineinzudenken, die jemand anders erstellt hat. Je länger die Zeitspanne zwischen Programmierung und Feedback wird, desto höher ist die Wahrscheinlichkeit, dass auch der Entwickler inzwischen gedanklich in andere Themen involviert ist und sich dann selbst erneut einarbeiten muss. Nicht zuletzt steigt die Gefahr, dass auf den fehlerhaften Code längst andere Inkremente aufbauen, die dann zwangsläufig mitbetroffen sind – ein Dominoeffekt der unerwünschten Art.

Das ist ein Grund, warum agiles Software Engineering auf konstantes und direktes Feedback setzt und dazu Entwicklung und Testen zu engen Freunden oder Kooperationspartnern macht. Der Test wird zu einer „ausführbaren Spezifikation“, die entweder der Entwickler oder der Tester aufsetzt, sobald die Anforderungen vollständig sind. Unit Tests entstehen vor und beim Schreiben von eigentlichem Produktivcode. Man kann sagen, der Produktivcode wird anhand eines Unit Tests entwickelt. Dabei entspricht der Test einer recht feingranula-

ren Spezifikation der jeweiligen Funktonalität. Insofern erfüllt der Code immer die zugrunde liegende Spezifikation eines Unit Tests. Insgesamt klingt das zunächst, als ob gerade Unit Tests viel Aufwand für die Entwickler verursachen. Tatsächlich sind sie jedoch gleich aus zwei Gründen rationell: Erstens, weil für den Produktivcode eben nur so viel entwickelt wird, bis der die Anforderung spezifizierende Test erfüllt ist. Zweitens, weil die Fülle an Unit Tests am Anfang verhindert, während des Lebenszyklus der Software Bugschulden anzuhäufen. Natürlich ist es schön, wenn sich Entwickler rein auf die Funktionalität konzentrieren können. Nur bringt es nicht viel, wenn die Zahl der zu fixenden Bugs synchron mit der Lebensdauer der Software steigt und daher mehr und mehr Zeit ausschließlich für Reparaturarbeiten benötigt wird – bis, im Extremfall, nur noch Schadensbegrenzung möglich ist. Diese Art von Hypothek sollte beim agilen Vorgehen gar nicht erst entstehen.

Beispielsweise deswegen nicht, weil der Code praktisch ständig refaktorisiert wird. Für dieses Refaktorisieren bilden die Tests eine Art Sicherheitsnetz, indem sie die Funktionalität einer Software dokumentieren. Mit diesem Schutz können die Entwickler den Code „aufräumen“, verschlanken und an ihm feilen, ohne Funktionalitäten zu gefährden.

Test nicht gleich Test

Unabhängig von der Vielzahl an Bezeichnungen, Testtypen und Nomenklaturen lassen sich Testfälle in der so genannten *Testpyramide* (**Abb. 1**) nach ihrer Abstraktionsebene gliedern. Anders gesagt unterscheiden sich die Testfälle der einzelnen Ebenen hinsichtlich ihrer Komplexität und Zeitdauer.

Demnach empfiehlt es sich, die einzelnen Testfälle in der Praxis unterschiedlich einzusetzen. Wie, das zeigt ein Beispiel aus dem Unternehmen arvato infoscore GmbH, ein Dienstleister für ein integriertes und wertorientiertes Management von Kundenbeziehungen und Zahlungsflüssen. Als solcher bearbeitet das Unternehmen bis zu 50 000 Bonitätsanfragen pro Stunde. Die durchschnittliche Antwortzeit beträgt für drei Viertel aller Bonitätsanfragen weniger als eine Sekunde, bei insgesamt ca. 40 Millionen gespeicherten Merkmalen.

Die zugrunde liegende Systemlandschaft ist seit fünfzehn Jahren gewachsen und entsprechend komplex. Einzelne Systemkomponenten hatte die arvato infoscore GmbH selbst entwickelt, andere wurden von Dienstleistern erstellt und später übernommen. Derzeit werden nun die Einzelsysteme in einer plattformbasierten Lösung konsolidiert. Dazu erteilte die arvato infoscore GmbH dem Karlsruher Beratungs- und Entwicklungshaus andrena objects den Auftrag, die Systeme zu analysieren und Empfehlungen abzuleiten. Eine dieser Empfehlungen lautete, die Testabdeckung weiter zu erhöhen, besonders bei den älteren Systembestandteilen. Denn diese hohe Testabdeckung bildet die Voraussetzung für umfangreiche Systemumstrukturierungen, beispielsweise, um eine technische Schichtung einzuführen.

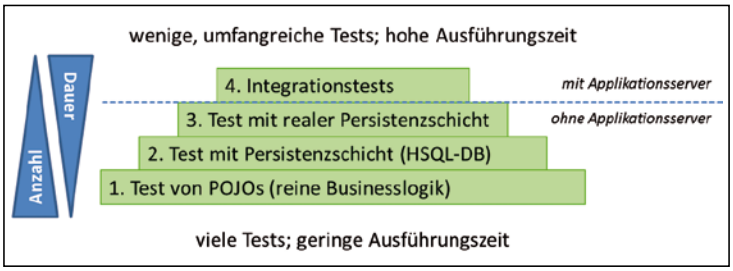

Abb. 1: Testpyramide

Die angestrebte Erhöhung der Testabdeckung ist in diesem Fall ein Synonym für „Ausbau der automatisierten Tests“. Dazu entwickelte andrena objects ein Testkonzept, das primär für ein Teilprojekt pilotiert wurde. Seine beiden wesentlichen Eigenschaften bestehen darin, erstens die Testfälle nach ihrer Abstraktionsebene zu klassifizieren – sprich, die oben erwähnte Testpyramide einzuführen. Zweitens unterstützen entsprechende Hilfsklassen jede Testebene. Dazu nehmen sie nötige Konfigurationen vor und stellen oft benötigte Funktionen bereit. Das ermöglicht den Entwicklern, sich auf das Schreiben der Tests zu konzentrieren.

Innerhalb der Testpyramide bilden codenahe, reine Unit Tests die unterste und breiteste Ebene. Sie testen ausschließlich die Businesslogik, prüfen einzelne Klassen, sind permanent ausführbar und nehmen nur Sekunden an Laufzeit in Anspruch. Gemeinsam bilden sie das primäre Fangnetz, innerhalb dessen die Software restrukturiert und weiterentwickelt werden kann. Rein theoretisch ließe sich die gesamte Funktionalität auf dieser Ebene prüfen.

Die Praxis hingegen ist häufig komplexer, weil Fremdsysteme angebunden werden, im speziellen Fall eine Datenbank. Deshalb stehen auf der zweiten Ebene Integrationstests, in der lokalen Entwicklungsumgebung allerdings nicht gegen die reale Persistenzschicht, sondern gegen eine „HSQL-in-Memory“-Datenbank. Diese virtuelle Datenbank ist auch für mehrere gleichzeitig testende Entwickler schnell ansprechbar und sichert die lokale Unabhängigkeit der Tests und damit die Konsistenz des Zustands. Gleichbleibende Ausgangslagen bedeuten wiederholbare Tests, und die sind ihrerseits eine der Grundbedingungen für agiles Testen. Alle Änderungen der Datenbank nach dem Test zurückzunehmen – was hier der Fall ist –, verhindert die unerwünschten Interferenzen zwischen den einzelnen Tests.

Trotzdem sind natürlich auch Tests gegen die reale Datenbank sinnvoll. Sie laufen in der dritten Ebene und gewährleisten, dass die Persistenzschicht richtig konfiguriert wurde.

Diese drei Arten von Testfällen bewegen sich innerhalb der Java Virtual Machine, im Idealfall werden sie alle innerhalb eines definierten Zeitplans automatisiert ausgeführt, z. B. auf einem Build-Server (wie Hudson oder Jenkins).

Die oberste Schicht an Integrationstests berücksichtigt, dass sich manche Konfigurationen erst nach dem Hochfahren des Servers überprüfen lassen, wenn ein

System auf einem Applikationsserver läuft. Tests der Ebene 4 starten zuerst den Server und schicken dann von außen Anfragen an das zu testende System.

Diese Tests werden sowohl manuell als auch automatisiert durchgeführt. Der Testmanager innerhalb des Scrum-Teams leitet diese Tests, die einen hohen Automatisierungsgrad haben. Die Ergebnisse dieser Tests und die enge Zusammenarbeit der Entwickler mit den Testmanagern innerhalb des Sprints helfen hierbei, dann schnell auf eventuelle Fehler zu reagieren.

Unabhängig davon, wer testet: Generell lautet die Faustregel, dass die Anzahl der Tests abnehmen sollte, je höher die Testebene ist. Denn synchron mit der Ebene steigt die Komplexität, nicht nur im Schreiben der Tests, sondern auch in der Ausführung. Wird etwa die Businesslogik in allen denkbaren Varianten auf der untersten, also Unit-Test-Ebene durchgespielt, dann reichen auf der Ebene der Integrationstests exemplarisch ein positives und ein negatives Beispiel. Da die Tests der untersten Ebene die schnellsten und einfachsten sind, sinkt der Gesamtaufwand signifikant. Der abgestufte Einsatz der verschiedenen Testfälle macht es den Entwicklern leichter, ihr Testmanagement in diesem Sinne effizienter zu gestalten.

Fazit

Tests gezielt weitreichend zu automatisieren und, je nach Abstraktionsgrad, unterschiedlich einzusetzen, erhöht die Testabdeckung und damit mittelbar die Codequalität. Damit sinken die ansonsten sehr häufigen, späteren Kosten zur Fehlerbehebung deutlich. Außerdem ist die hohe Testabdeckung elementare Voraussetzung für das Refaktorisieren, das seinerseits zur Basis des agilen Programmierens gehört. Und für agiles Programmieren hat sich das Beispielunternehmen arvato infoscore GmbH klar entschieden. Dazu werden bereits XP-Techniken wie Pair Programming und Test-First eingesetzt. Erste Eindrücke der betroffenen Entwickler und Fachbereiche sind sehr positiv. Im Verbund mit den Testfällen auf verschiedenen Ebenen wurde also die Basis gelegt, die Softwarequalität noch weiter zu erhöhen – sei es aus Anwender- oder aus Entwicklersicht.

Womit dann auch geklärt wäre, warum agiles Testen und guter Code ein wirklich schönes Paar sind. Agiles Testen unterstützt Entwickler dabei, guten Code zu entwickeln. Guter Code ist das Ziel jedes Entwicklers. Und wirtschaftlich sinnvoll.

Maynard Harstick ist Testmanager bei arvato infoscore im Bereich IT-Risk Management. Er ist seit über fünfzehn Jahren in der Softwarequalitätssicherung sowohl im klassischen Entwicklungsbereich als auch im agilen Umfeld tätig.

Daniel Knapp studierte Wirtschaftsingenieurwesen am KIT und arbeitet seit 2005 als Softwareentwickler bei der andrena objects ag. 2010 wurde er zum Leiter des Geschäftsfelds Lösungen berufen. Sein Hauptinteresse gilt dem Agile Software Engineering.

Notizen:

