Modellbasierter Test mit FitNesse – Ein Erfahrungsbericht aus der Praxis

Dehla Sokenou
GEBIT Solutions, Koenigsallee 75b, D-14193 Berlin
dehla.sokenou@gebit.de

Abstract: Das Papier stellt ein modellgetriebenes Softwareprojekt vor, in dem systematisch mit Hilfe des Testframeworks FitNesse getestet wurde. Dabei werden Vor- und Nachteile des Werkzeugs im praktischen Einsatz beleuchtet.

1 Motivation

In großen Projekten stellt sich meist die Frage nach einem Testwerkzeug, da manuelle Tests in ausreichender Zahl nicht durchgeführt werden können. Einerseits soll dieses Testwerkzeug flexibel genug sein, um es auf die speziellen Bedürfnisse des Projekts anzupassen. Andererseits soll es möglichst einfach in seiner Bedienung sein. Zusätzliche Anforderung in unserem Projektkontext war die Anwendbarkeit einmal entwickelter Testtechniken auch in anderen Projekten, die mit dem gleichen modellgetriebenen Framework für Softwareentwicklung implementiert werden. Das Papier stellt ein modellgetriebenes Projekt vor, in dem systematisch mit Hilfe des FitNesse-Framework [Fit] getestet wurde. Zunächst wird das Projekt vorgestellt, anschließend der Einsatz von FitNesse im Projekt beschrieben. Nach einer Diskussion verwandter Arbeiten wird zum Schluss eine Zusammenfassung unserer Erfahrungen sowie ein Ausblick auf aktuelle und zukünftige Projekte gegeben.

2 Das Projekt

Im Projekt wurde eine Java Enterprise-Anwendung entwickelt, die auf einem JBoss-Applikationsserver läuft und über einen Apache-Webserver Zugriff per Web erlaubt. Die Webseiten werden mit Hilfe von JSF erzeugt. Zusätzlich werden verschiedene mobile Geräte per WLAN eingebunden. Der Projektumfang umfasste bei einer Entwicklungszeit von 2 Jahren ca. 10 Personenjahre (davon ca. 10% für FitNesse-Tests), ca. 2000 Java-Klassen (davon 120 Business-Objekt-Klassen), ca. 220.000 Lines of Code und etwa 100 Datenbanktabellen. Die Anwendung wurde auf der Basis des GEBIT-eigenen TREND-Frameworks [GEB] für modellgetriebene Entwicklung realisiert, dabei wurden knapp 100 Use Cases umgesetzt. Das Projekt wurde –auch aufgrund der FitNesse-Tests– im vorgegebenen Zeitrahmen abgeschlossen und ist inzwischen produktiv im Einsatz.

3 FitNesse im Projekt

Bereits zu Beginn des Projekts wurde nach einem neuen Testwerkzeug gesucht, das sich gut in den modellgetriebenen Softwareentwicklungsprozess einbinden ließ. Dabei erschien FitNesse als geeignet, zumindest einen großen Teil der Anforderungen zu erfüllen.

Warum FitNesse? FitNesse ist durch sein offenes Konzept sehr flexibel. Testfälle werden mit Hilfe eines Wikis als Tabellen definiert. Die Auswertung erfolgt mit Hilfe des dahinterliegenden Fit-Frameworks [MC05], entweder über das Wiki oder mit Hilfe eines Testrunners, der z.B. aus Eclipse heraus gestartet werden kann. Die Testfälle werden mit Hilfe sogenannter Fixtures ausgewertet, kleinen Stücken von Programmcode, die die Verbindung zwischen den zunächst abstrakten Testfällen und dem zu testenden System implementieren. Neben einigen rudimentären Build-in-Fixtures erlaubt FitNesse die Definition eigener Fixtures. Diese Fixtures können entweder produktspezifisch definiert werden oder auch in allgemeiner Form. FitNesse ist nicht beschränkt auf eine Art des Tests, sondern unterstützt durch unterschiedliche Fixture z.B. sowohl Oberflächen- als auch Backend-Tests.

Dies sprach für den Einsatz von FitNesse, da das Framework flexibel erweiterbar genug ist, um eine generische Anbindung zu ermöglichen, die nicht auf das aktuelle Softwareprodukt beschränkt ist. Stattdessen wurden die Fixtures so implementiert, dass sie geeignet sind, alle Produkte zu testen, die modellgetrieben mit TREND implementiert sind. Es wurden etwa 30 generische Fixtures im Projektkontext implementiert, Beispiele sind: *Starten von Use Cases oder Aktivitäten*, *Ausführen von Aktivitätsschritten*, *Prüfen von Werten in einem Objekt oder der Datenbank*. Zusätzlich gibt es projektspezifische Fixtures. Der Großteil der Tests ($>$80%) wurde jedoch mit den generischen Fixtures realisiert.

Im Projekt war das Testen zunächst ein Randthema. Sehr schnell wurde jedoch erkannt, dass die Qualität einer Software mit der vorliegenden Komplexität nur durch entwicklungsbegleitende automatisierte Tests nachhaltig zu sichern ist. Deshalb wurde nach etwa einem halben Jahr ein Testspezialist in das Projekt geholt, der die wenigen bereits vorhandenen Fixtures in umfassender Weise erweiterte und ergänzte und systematisch die Funktionen der Software auf Basis der Modelle und der Spezifikation durch Tests überprüfte. Die Erstellung erfolgte zwar manuell, aber auf Basis der Modelle.

Es wurden zwei Arten von Testsuiten erstellt. Der “rote Faden” erreicht eine 100% Überdeckung von Use Cases, Aktivitäten und Aktivitätsschritten, aber als Black-Box-Test systembedingt keine 100% Überdeckung der Implementierung. Er umfasst 17450 Prüfungen in 350 Tests, die mit einer Laufzeit von 630 Sekunden durchgeführt werden. Eine weitere Testsuite überprüft spezielle Funktionen, z.B. bisher nicht erreichten oder kritischen Code. Diese umfasst 15000 Prüfungen in 220 Tests mit einer Laufzeit von 940 Sekunden. Im Laufe des Projekts waren zwangsläufig Änderungen an den Testfällen erforderlich. Diese waren bedingt durch Änderungen an den Modellen und Änderungen der anderen implementierten Teile. Die Testfälle wurden innerhalb der Projektlaufzeit etwa 500 Mal angepasst, wobei eine Anpassung meist nur einen oder wenige Testfälle betraf. Bei der Menge der Testfälle ist eine Anpassung in dieser Größenordnung aus unserer Erfahrung normal und wird z.B. bei Einsatz von Werkzeugen zum reinen Oberflächentest noch übertroffen.

Nach Einführung der systematischen Tests durch einen speziell dafür eingesetzten Exper-

ten verbesserte sich die Qualität der sich in der Entwicklung befindlichen Software rapide. Durch die gute Abdeckung der Funktionen durch Tests wurden neue Fehler schnell gefunden. Besonders gut geeignet waren sie für den Regressionstest, d.h. sie verhinderten das Einbringen neuer Fehler in bestehenden Softwareteilen, die bereits eine gute Qualität aufwiesen. Ziel war stets der fehlerfreie Testdurchlauf. Liefen die FitNesse-Testfälle ohne Fehler durch, so bestand so viel Vertrauen in die Stabilität der Software, dass dem Kunden auch außerhalb der Releases ein Prototyp zur Verfügung gestellt werden konnte. Von den ca. 800 Fehlern im Issuetracking-System sind ca. 500 Fehler durch die FitNesse-Tests initial aufgedeckt worden. Die anderen Fehler wurden z.T. durch Entwicklertests, z.T. vom Kunden in den Prototypen und Releases gefunden.

4 Verwandte Arbeiten

Abbot, ein klassisches Capture-Replay-Werkzeug, wird ebenfalls in einigen unserer Projekte eingesetzt. Abbot wurde in den 4 Jahren seines Einsatzes in unseren Projekten kontinuierlich um TREND- und Projektspezifika erweitert. Im Gegensatz zu FitNesse ist Abbot für den Tester zunächst einfacher zu bedienen. FitNesse bietet jedoch ein abstrakteres, technisch einfacher aufgebautes Framework für die Implementierung eigener Fixtures. Die Testfälle sind mit FitNesse abstrakter definiert, da nicht einzelne Klicks auf der Benutzeroberfläche protokolliert werden, sondern eine Anweisung im Testfall erst durch die dahinterliegende Fixture ausgewertet wird.

Fallstudien zu TTCN-3 (vgl. [BSSG04, DGNP04]) wurden meist in der Domäne der Telekommunikationssysteme durchgeführt. Die Ergebnisse dieser Fallstudien lassen sich in der Regel nicht auf die Domäne der betrieblichen Anwendungssysteme übertragen, die nicht den reaktiven Character von Telekommunikationssystemen haben. Zudem benötigen modellbasierte Testverfahren, die auf TTCN-3 basieren, meist spezielle Testmodelle, modelliert z.B. mit dem UML Testing Profile [UTP04]. Unsere Tests basieren ausschließlich auf den vorhandenen Designmodellen. Auch für den Leirios Test Designer [BBC+06], einem Werkzeug für den modellbasierten Test, ist die Definition spezieller Testmodelle nötig, ebenfalls in UML, doch ohne das UML Testing Profile zu nutzen.

5 Zusammenfassung und Ausblick

FitNesse überzeugt durch seine offene Architektur. Zusätzlich zu den Fixtures haben wir das Wiki (Syntax und neue Funktionen wie Anlegen neuer Testsuiten) erweitert. Die generische Entwicklung von Fixture ist relativ einfach möglich, so dass FitNesse leicht an andere Frameworks angebunden werden kann oder für den Test der verschiedensten Anwendungen geeignet ist. Gleichzeitig muss man auf die Nachteile von FitNesse hinweisen. FitNesse ist durch die Definition von Testfällen im Wiki weniger intuitiv bedienbar als viele andere Testwerkzeuge. Zum Testen muss immer zunächst der FitNesse-Server gestartet werden.Es gibt wenig Modularisierungs- und Strukturierungsmöglichkeiten. Testfälle

können zwar von anderen Testsuiten referenziert werden, die Testfälle sind jedoch streng hierarchisch strukturiert und ihre Reihenfolge nur durch ihre Benennung festgelegt. Integrationstests lassen sich nur schwierig realisieren, wenn das Zusammenspiel mehrerer Prozesse getestet werden soll. Der Test von Oberflächen ist relativ mühselig, da alle Schritte manuell in Wiki-Tabellen beschrieben werden müssen, es fehlt hier noch an geeigneter Werkzeugunterstützung. Eines der größten Mankos für die Einarbeitung in FitNesse ist die mangelnde Dokumentation des offenen Quellcodes, da dies ein Hindernis bei der (für den praktikablen Einsatz notwendigen) Erweiterung des Testframeworks darstellt.

Nach den guten Erfahrungen in dem beschriebenen Pilotprojekt für den FitNesse-Einsatz wird FitNesse auch in aktuellen Projekten eingesetzt. Wir haben dabei gute Erfahrungen gemacht, wenn mindestens ein Entwickler mit Testerfahrung in einem Projekt ausschließlich für die FitNesse-Tests zuständig ist und Testen nicht nur "nebenbei" passiert. Neben der Erstellung der Testfälle an sich ist dieser auch für die Erweiterung der vorhandenen Fixtures verantwortlich. Aktuell werden wenn möglich FitNesse-Tests direkt von Beginn eines Projekts eingesetzt, was ein hohes Qualitätsniveau bereits vom Projektstart an gewährleistet, ohne den Gesamtaufwand signifikant zu erhöhen. Ergänzend werden mit JUnit [BG00] implementierte Entwicklertests definiert. Zukünftig soll auch die Erstellung von Testfällen unterstützt werden durch ein zusätzliches TREND-Werkzeug zur Aufzeichnung von FitNesse-Testfällen und die automatische Generierung von FitNesse-Testfällen aus den Modellen, wobei bereits bestehende Techniken verwendet werden [Sok06].

Literatur

[BBC+06] E. Bernard, F. Bouquet, A. Charbonnier, B. Legeard, F. Peureux, M. Utting und E. Torreborre. Model-Based Testing from UML Models. In *MOTES-Workshop, Informatik 2006*, number P-94 in LNI, Seiten 223–230, Dresden, Germany, 2006.

[BG00] K. Beck und E. Gamma. Test Infected: Programmers Love Writing Tests. In Dwight Deugo, Hrsg., *More Java Gems*, Seiten 357–376. Cambridge University Press, 2000.

[BSSG04] M. Born, I. Schieferdecker, P. Santos und H.-G. Gross. Model-Driven Development and Testing - A Case Study. In *1st European Workshop on Model Driven Architecture with Emphasis on Industrial Application*, Enschede, Netherlands, 2004.

[DGNP04] Z. R. Dai, J. Grabowski, H. Neukirchen und H. Pals. From Design to Test with UML - Applied to a Roaming Algorithm for Bluetooth Devices. In *TestCom'2004*, number 2978 in LNCS, Oxford, UK, 2004. Springer-Verlag.

[Fit] FitNesse Test Framework. http://fitnesse.org/.

[GEB] GEBIT TREND Framework for Java. http://www.gebit.de/trend.

[MC05] R. Mugridge und W. Cunningham. *Fit for Developing Software - Framework for Integrated Tests*. Prentice Hall, 2005.

[Sok06] D. Sokenou. *UML-basierter Klassen- und Integrationstest objektorientierter Programme*. Dissertation, Technische Universität Berlin, Germany, 2006.

[UTP04] *UML 2.0 Testing Profile Specification, V. 1.0*. OMG, *http://www.omg.org*, 2004.