Standardisierung der technischen Qualitätssicherung im J2EE-Umfeld der Dresdner Bank

Michael Meurer (Dresdner Bank AG), michael.meurer@dresdner-bank.com
Daniel Simon (SQS AG), daniel.simon@sqs.de

1 Einleitung

Nach Bilanzsumme und Zahl der Kunden zählt die Dresdner Bank zu den führenden europäischen Bankengruppen. Seit 2001 ist die Dresdner Bank als „Kompetenzzentrum Banking“ Teil der Allianz Gruppe. In der Verbindung von Allianz und Dresdner Bank besteht das Potenzial für die Schaffung erheblichen Mehrwerts durch ein größeres Angebot an Finanzprodukten, breiteren Vertriebskanälen sowie mehr Beratungskapazität und Beratungskompetenz. Nicht zuletzt können durch die Kombination von Versicherungsprodukten der Allianz und von Vermögensanlageprodukten der Dresdner Bank die nachgefragten Produkte im Bereich der betrieblichen und privaten Altersvorsorge noch effektiver angeboten werden.

Das breit gefächerte Geschäftsfeld der Bank spiegelt sich auch in der IT-Infrastruktur wider, die unter verschiedenen Aspekten zentrale Dienste zuverlässig und effizient zur Verfügung stellen muss. Dazu zählt der Zugang von Endkunden über Internet zu Diensten wie Depotverwaltung und Online-Banking ebenso wie der Zugriff von Kundenbetreuern und Außendienstmitarbeitern auf aktuelle Daten und Fakten, um maßgeschneiderte Angebote für Privat- und Geschäftskunden zu ermöglichen.

Die strategische Entscheidung der Dresdner Bank, zur Umsetzung dieser Dienste eine J2EE-Infrastruktur zu nutzen, hat verschiedene Konsequenzen auf die Test- und Qualitätssicherungsverfahren, die eingesetzt werden. Mit dem Blick auf Service orientierte Architekturen zur übergreifenden Homogenisierung der Softwarelandschaft hat die Dresdner Bank die Überarbeitung, Anpassung und Standardisierung der bislang verwendeten Test- und Qualitätssicherungsmaßnahmen mit Unterstützung der SQS AG in Angriff genommen. Als wichtige Einflussfaktoren kommen dabei die breiten in der Bank selbst gesammelten Erfahrungen ins Spiel, auf deren Basis die bankinternen Best Practices und Gewichtungen vorgenommen werden. Die bestehenden Testverfahren wurden mit Rücksicht auf die Besonderheiten des J2EE-Umfelds – wo nötig – angepasst und die Einführung von Code Quality Management (CQM) beschlossen. Ein erklärtes Ziel ist die praxisnahe Anwendbarkeit und Werkzeugunterstützung der neuen Konzepte, die sich anhand eines der größten hausinternen Softwareprojekte auf ihre Durchführbarkeit und Tauglichkeit prüfen lassen mussten.

2 Technologisches Umfeld

In der Dresdner Bank ist als strategische Entscheidung festgelegt, dass J2EE-Applikationen Thin-Client Webanwendungen sind.

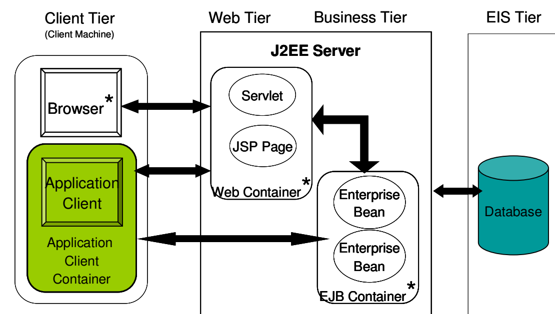

Abbildung 1: Betrachtete Teile (*) der J2EE-Architektur

In Abbildung 1 ist dargestellt, welche Teile der J2EE-Architektur daher im Kontext des Standardisierungsprozesses betrachtet werden. Auf der Client-Seite kommen Thin-Clients in Form von Web-Browsern zum Einsatz; in der Dresdner Bank wird in diesem Zusammenhang eine Erweiterung des Struts-MVC-Frameworks [1] eingesetzt. Die Browser selbst werden als Black Box betrachtet und die zu prüfenden Artefakte der Software-Entwicklung sind hier in erster Linie die zu den verschiedenen Browsern transportierten HTML-Daten, die auf der Serverseite auf unterschiedliche Art zur Verfügung gestellt werden.

Betrachtungsgegenstand auf der J2EE-Serverseite sind insbesondere die Artefakte, die innerhalb eines Web- oder eines EJB-Containers zum Einsatz kommen und in Java oder Java-ähnlichen Artefakten verfasst werden. Dazu zählen Servlets, Java Server Pages (JSPs) sowie die unterschiedlichen Arten der (Enterprise) JavaBeans (EJBs).

Die Backend-Schicht der J2EE-Infrastruktur war in diesem Projekt nicht Gegenstand der Betrachtungen, da hier keine signifikanten Unterschiede zu bereits etablierten Prüf- und Qualitätssicherungsverfahren ausgemacht werden konnten.

Um die Integrationsqualität der entwickelten Artefakte zu gewährleisten, ist frühes Testen und Prüfen notwendig. Im Rahmen dieses Projektes wurden daher u.a. QS-Maßnahmen in Form von Entwicklertests (oder Low-Level-Tests) und statischen Analysen im Sinne des CQM eingeplant und umgesetzt.

3 Statische Analysen und Unit-Tests

Sowohl statische Analysen als auch Komponententests bieten den Vorteil, dass die Prüfung bzw. die Testausführung bereits frühzeitig nach der Erstellung der Testobjekte erfolgen kann. Im Falle der statischen Analysen und damit des gesamten CQM kann sogar projektübergreifend die Güte der entwickelten Software-Artefakte beurteilt werden, da hierfür „nur" ein compilierfähiges System notwendig ist. Die spezifische Infrastruktur lässt sich vielfach sehr schnell im Bereich statischer Analysen aufbauen – die Regelmengen lassen sich anpassen, um spezielle Konstrukte und Muster ins Auge zu fassen.

Anders ist die Situation beim dynamischen Testen: Im J2EE-Umfeld stellt die Integration einzelner Komponenten nur dann einen sinnvollen Schritt dar, wenn die entsprechenden Komponenten bereits vorher einzeln erfolgreich geprüft wurden. Abgesehen jedoch von einfachen Java-Klassen, stellt sich sehr bald das Problem, dass Klassen, die gegen eine komplexe Infrastruktur programmiert werden, eben diese Infrastruktur auch für Unit-Tests und einfache technische Integrationstests benötigen. Im J2EE-Umfeld muss folglich zum Testen von Klassen, die mit EJB-Technologie arbeiten, ein EJB-Container zur Verfügung stehen. Die Simulation eines solchen Containers zu erstellen steht aufgrund des hohen Aufwands außer Frage. Die Lösung bietet sich hier in Form von Erweiterungen des JUnit-Frameworks [2] an. Grundsätzlich können dazu das JUnitEE-Framework [3] oder das deutlich komplexere Cactus-Framework [4] eingesetzt werden. Letzteres subsumiert mit seinen Möglichkeiten das leichtgewichtigere JUnitEE-Framework, ist aber deutlich komplexer und daher auch schwerer zu überblicken. Beide Frameworks arbeiten mit den jeweiligen Containern zusammen und ermöglichen die Testausführung im Kontext des Applikationservers.

Als Alternativvorschlag ist hier der Ansatz der „Box-Metapher" [5] zu erwähnen, der mittels Verlagerung sämtlicher fachlicher Logik aus EJBs heraus in gewöhnliche Java-Klassen darauf abzielt, die Tests der fachlichen Logik im gewöhnlichen Unit-Test-Umfeld durchzuführen und die EJBs quasi zu reinen Delegatoren und Fassaden zu degradieren. Dieser Vorschlag fällt allerdings eine Design-Entscheidung, die in aktuellen Projekten nicht praktikabel erscheint.

3.1 Low-Level-Tests

Eine der Herausforderungen im J2EE-Umfeld ist die Ausführung von Komponenten- und Clustertests. Die Herausforderung ergibt sich insbesondere aus der großen Dominanz der Infrastrukturvorgaben durch die Verwendung von Applikations- und Webservern, in denen die einzelnen Klassen und Komponenten schließlich verwendet werden sollen. Auf die angebotene Infrastruktur greifen diese Klassen naturgemäß zurück – und sind damit außerhalb dieser Infrastruktur nur noch eingeschränkt testbar. Aus Gründen des Aufwands verbietet sich die Implementierung eines J2EE-Simulators und erfordert es, den Entwicklern zur Durchführung von Low-Level-Tests eine geeignete Testumgebung zur Verfügung zu stellen.

Diese Testumgebung besteht einerseits aus den entsprechenden Applikationsservern, die mit den Testobjekten umgehen können (im Falle der Dresdner Bank ist die Verwendung des WSAD [6] auch auf der Entwicklungsplattform vorhanden und einsatzbereit). Andererseits muss für unterschiedliche Testartefakte für den Unit-Test Standard-Funktionalität in Form von Test-Frameworks bereitgestellt werden.

Für die einzelnen Artefakte sind dies:

- einfache Java-Klassen: JUnit
- EJB-Klassen: Cactus
- Servlets (im Zusammenspiel mit struts): Cactus und StrutsTestCase [7]
- JSPs: Cactus

Alternativ lässt sich bei EJB-Klassen auch das Testframework JUnitEE einsetzen, das deutlich leichtgewichtiger und einfacher handhabbar als Cactus ist. Allerdings sind dann die Tests im Applikaitonsserver auf einfache Klassen und EJBs eingeschränkt. Cactus bietet über eine Integration des JUnitEE-Ansatzes hinaus weitaus mehr Möglichkeiten bei der Umsetzung der Tests.

3.2 Code Quality Management

Für den regelmäßigen Einsatz statischer Analysen im Rahmen des Code Quality Managements wurden zunächst innerhalb der Bank in Zusammenarbeit mit den Software-Architekten der Projekte Programmierrichtlinien entwickelt. Diese Programmierrichtlinien sind für die Entwickler verbindlich und müssen der Implementierung beachtet werden. Sie sind aufgeteilt in

- bankweite Richtlinien: Die Flexibilität beim Einsatz von Projektmitarbeitern innerhalb der Bank wird erleichtert, da die Unabhängigkeit von spezifischen Skills gefördert und ein Corporate Knowledge über Grundsätze der Entwicklung aufgebaut wird.
- projektspezifische Architektur: Die Anwendungsarchitektur wird in den Richtlinien für individuelle Projekte festgelegt und die Implementierung wird gegen diese Architektur geprüft.
- projektspezifische Richtlinien: Um den Projekten die Möglichkeit zur Anpassung und Ergänzung der Richtlinien zu geben, sind an dieser Stelle weitere Regeln möglich.

Die Regeln folgen in ihrem Aufbau dem CQM-Vorgehen [8] unter Verwendung bidirektionaler Qualitätsmodelle, die im Rahmen des BMBF Forschungsprojekts QBench [9] entwickelt und bereits in der Praxis erprobt wurden. Sie wurden zunächst Werkzeug unabhängig formuliert, ohne jedoch die automatische Prüfbarkeit aus den Augen zu verlieren. Nach Abstimmung der Regeln mit den Architekten wurde eine zentrale, zweistufige Prüfinfrastruktur errichtet. Einfache, lokal prüfbare Regeln, werden innerhalb der Entwicklungsumgebung mittels des Codecheker-Tools Checkstyle [10] geprüft und geben den Programmierern sofortiges Feedback.

Darüber hinaus gehende Regeln, insbesondere die Prüfung der Einhaltung der Architektur führt der Architekt in regelmäßigen Abständen unabhängig von den Entwicklern aus. Die Regeln, die die Entwickler prüfen müssen, werden hier noch mal geprüft, um sicherzustellen, dass die individuellen Prüfungen auch durchgeführt wurden und um die Resultate für die Gesamtprüfung weiterzuverwenden – die Konsistenz der Prüfungen ist daher besonders wichtig. Die zentrale Prüfung wurde mittels des Werkzeugs Sotograph [11] realisiert.

3.3 Dashboard

Um die Ergebnisse der Entwicklertests und des CQMs der Projektleitung und dem Management zeitnah zur Verfügung zu stellen, ist ein Dashboard vorgesehen, das die Daten aus den unterschiedlichen QS-Aktivitäten aggregiert und aufbereitet. Die Verwaltung der Daten basiert dabei auf einem Repository, das über punktuelle Datensichten hinaus auch Trendbetrachtungen ermöglicht.

4 Erfahrungen/Ergebnisse

Die Umsetzung der entwickelten Konzepte am Beispiel eines großen Projekts der Bank hat gezeigt, dass die Konzeption tragfähig ist und die vorgeschlagene Vorgehensweise in die Alltagspraxis übertragen werden kann.

Die Konzeption der Unit-Tests mit Hilfe unterschiedlicher Test-Frameworks sah vor, für einfache Klassen und Klassencluster JUnit und für J2EE-Artefakte wie EJBs, Servlets oder JSPs entsprechende Teile des Cactus-Frameworks zu nutzen. Die technische Integration in den bestehenden Projektkontext war zum Teil aufwändig. Die Verflechtung der bislang eingesetzten Testframeworks zu lösen und durch das Cactus-Framework zu ersetzen, setzte ein erhebliches Infrastrukturwissen voraus. Schlussendlich ließ sich die geplante Cactus-Integration jedoch durchführen und kann derart bewerkstelligt werden, dass dem Anwender des Testframeworks in Zukunft die Komplexität der Installation völlig verborgen bleibt. Die Erstellung einzelner Komponententests im Rahmen des Proof of Concept war nur mit Hilfe der Entwickler möglich, da hierzu in erheblichem Maße Anwendungswissen und ein Mindestmaß an fachlichem Verständnis zur Sinnhaftigkeit der Tests beiträgt. In Summe kann festgehalten werden, dass die Integration des Cactus-Frameworks in die Entwicklerumgebung erfolgreich verlaufen ist. Die vorhandenen Entwicklertests auf der Basis von JUnitEE lassen sich mit geringem Umfang portieren oder können einfach weiter verwendet werden.

Im Gegensatz zu den dynamischen Tests hatten die statischen Analysewerkzeuge auf der Installationseite keine nennenswerten Schwierigkeiten. Das Checkstyle-Plugin für WSAD arbeitete klaglos mit der Entwicklerumgebung zusammen. Die für Architekturprüfungen vorgesehene Infrastruktur auf der Basis des Sotographen hat ohne Probleme auch große Mengen Code automatisiert vermessen und integriert dabei auch die dem Entwickler durch Checkstyle an die Hand gegebenen Prüfungen.

Im Gegensatz zu den Unit-Tests, für die ein Grundverständnis der Fachlichkeit der Anwendung vonnöten ist, braucht ein Prüfer bei der statischen Analyse nur technisches Wissen bei der Erstellung der Anwendungsarchitekturen in den Werkzeugformaten. Gerade diese Architektur muss unserer Erfahrung nach entsprechend präzise formuliert als Soll-Architektur vorliegen, gegen die eine Implementierung auch geprüft werden kann.

5 Referenzen

[1] Apache Struts, http://struts.apache.org/
[2] JUnit, http://www.junit.org/
[3] JUnitEE, http://www.junitee.org/
[4] Apache Cactus, http://jakarta.apache.org/cactus/
[5] Peters, V., Simple Design and Unit Testing with Enterprise JavaBeansTM: The Box Metaphor, XP2001
[6] WSAD, Websphere Studio Application Developer, IBM, Version 5.1
[7] StrutsTestCase, http://strutstestcase.sf.net/
[8] Simon, F., Seng, O., Mohaupt, Th., Code Quality Management mit dem Codes-Sünden-Index, dpunkt-Verlag, Mai 2006
[9] Forschungsprojekt QBench, http://www.qbench.de/
[10] Checkstyle, http://checkstyle.sf.net/
[11] Sotograph, Software Tomographie GmbH