Test von Komponenten

Beitrag für den 18. *TAV-Workshop* der GI-FG 2.1.7 "Test, Analyse und Verifikation von Software"
20. und 21. Juni 2002, Hasso Plattner Institut, Universität Potsdam

Michael Becker, Falk Fraikin[1], Stefan Jungmayr[2], Moritz Schnizler[3], Andreas Schoolmann[4], Mario Winter[2]
TAV-Arbeitskreis TOOP, „Testen objektorientierter Programme“
[1]TU Darmstadt, [2]FernUni Hagen, [3]RWTH Aachen, [4]TUI Infotec Hannover

Zusammenfassung: Nachdem sich objektorientierte Ansätze Mitte der 90er Jahre auf breiter Front durchgesetzt haben ist heutzutage die komponentenbasierte Softwareentwicklung in aller Munde. Trotzdem blieb der Test komponentenbasierter Software — ebenso wie Anfang der 90er Jahre schon der Test objektorientierter Software — lange Zeit unbeachtet.

In diesem Beitrag erläutern wir die Unterschiede zwischen objektorientierter und komponentenbasierter Software und beschreiben am Beispiel der Komponentenarchitektur Enterprise Java Beans (EJB) mögliche Formen des spezifikationsbasierten Tests von Komponenten. Anmerkungen zu entsprechenden Testwerkzeugen runden den Beitrag ab.

1 Komponenten

Eine Komponente ist ein Baustein zur SW-Herstellung, welche ihre durch Kontrakte spezifizierte Funktionalität über Schnittstellen anbietet und nur explizite Abhängigkeiten zu ihrem Kontext besitzt [Szyp98]. Eine Komponente kann unabhängig entwickelt und verteilt werden und durch Dritte mit anderen Komponenten verbunden werden. Komponenten unterscheiden sich von Klassen einer objektorientierten Programmiersprache in mehreren Punkten:

- Komponenten sind meist größer als Klassen. Klassen können zur Implementierung von Komponenten verwendet werden, aber nicht umgekehrt. Klassen werden in einer Programmiersprache entwickelt, Komponenten müssen zu einem Komponentenstandard kompatibel sein (unabh. von der Programmiersprache). Klassen können von anderen Klassen erben, zwischen Komponenten existieren hingegen in der Regel keine Vererbungsbeziehungen.
- Komponenten benötigen zu ihrem Ablauf eine Komponentenumgebung. Diese Umgebung stellt über die Laufzeitumgebung der (objektorientierten) Programmiersprache hinaus weitere Dienste wie transparente Verteilung, Transaktionsmanagement, Persistenz, und Sicherheitskonzepte zur Verfügung.
- Komponenten können einzeln ausgeliefert werden, sind konfigurierbar und sind mit anderen Komponenten kombinierbar. Komponenten werden binär, in ausführbarer Form verbreitet, der Sourcecode (zumindest kommerziell vertriebener Komponenten) ist häufig nicht verfügbar. Sie bestehen aus mehreren, getrennten Artefakten: Implementierung, Schnittstellenbeschreibung(en) und Konfigurationsdateien. Die Dokumentation von Komponenten beinhaltet auch fachliches Wissen aus der Anwendungsdomäne, bei Klassen beschränkt sich die Dokumentation meist auf eine API-Beschreibung.
- Komponenten werden häufig in verteilten Anwendungen eingesetzt. Asynchrone Kommunikation, entfernte Methodenaufrufe (RPCs), Internetprotokolle und andere Technologien spielen daher eine zentrale Rolle.

2 Komponentenarchitekturen

Kommerziell verfügbare Komponentenarchitekturen sind das Corba Component Model (CCM) der Object Management Group [OMG], die in der Java Enterprise Edition (J2EE) enthaltene Enterprise JavaBeans (EJB) Spezifikation von Sun [Sun] und COM+ von Microsoft [MS]. Diese Architekturen unterscheiden sich u.a. nach der Unterstützung von Programmiersprachen und Betriebssystemen, der eingesetzten Kommunikationsmechanismen und der Technologien zur Unterstützung von Persistenz, Transaktionen, Sicherheit und Verzeichnisverwaltung.

Aufgrund der Ähnlichkeit von EJB und CCM[1] und der technischen Diversität der „gewachsenen“ COM/COM+ Architektur konzentrieren wir uns im Weiteren auf EJB.

3 Rollen von Komponenten-Entwicklern

Bei der Entwicklung von EJB-basierten Systemen werden folgende Rollen unterschieden:

- Der J2EE Product Provider stellt eine zu der J2EE Spezifikation konforme Umgebung (z.B. den Container als Laufzeitumgebung der Komponenten) zur Verfügung.
- Die Tool Provider stellen Werkzeuge für die von den restlichen Rollen wahrgenommenen Aufgaben zur Verfügung (development, assembly, packaging).
- Application Component Provider erstellen (unter anderem) Enterprise JavaBeans für J2EE Anwendungen. Als Ergebnis der EJB-Entwicklung wird eine EJB Java Archive (JAR) Datei erstellt, die sowohl den ausführbaren Code der EJB selbst als auch einen Deployment Deskriptor enthält, welcher die Funktionalität und weitere wichtige Charakteristiken der Komponente beschreibt.
- Der J2EE Client Developer entwickelt den Client. Als Ergebnis wird eine JAR Datei geliefert, die sowohl den

1. CCM ist eine Obermenge von EJB, für die zur Zeit jedoch noch keine Implementierung verfügbar ist.

Client als auch einen entsprechenden Deployment Deskriptor enthält.

- Der Application Assembler baut aus den Komponenten eine J2EE Anwendung und liefert als Ergebnis eine Enterprise Archive (EAR) Datei, die sowohl die Komponenten als auch einen Deployment Deskriptor enthält.
- Application Deployer (und der Administrator) stellen die auf EJBs basierende Anwendung auf dem J2EE Server zur Verfügung, passen ggf. die Deployment Deskriptoren an (Transaktionsverhalten, DB-Mapping, Sicherheitsmechanismen) und stellen Ressourcen wie Datenbankverbindungen zur Verfügung.

Diesen Rollen werden unterschiedliche Testarten zugeordnet. Der Application Component Provider testet die Funktionalität der Komponente im Entwicklertest wie gewohnt, teilweise außerhalb des Containers, teilweise in einem (Test-) Container. Zusätzlich testet er, ob die Komponente den durch den Komponentenstandard spezifizierten (Standard-)Lebenszyklus erfüllt. Durch Installationstests in unterschiedlichen Containern wird auch der Deployment Deskriptor geprüft. Der Application Assembler überprüft mit Blackbox Tests alle von ihm eingesetzten EJBs und testet ebenfalls den Deployment Deskriptor (wie oben). Der Application Deployer testet die Korrektheit des Containers (möglichst anwendungsunabhängig). Hierzu gehören die Prüfung des Transaktionsverhaltens, der Sicherheits- und der Persistenzmechanismen.

4 Testtechniken und Testendekriterien

Welches Testverfahren für eine Komponente verwendet werden kann hängt in erster Linie davon ab, wer den Test durchführt. Typischerweise muss zwischen dem Entwickler einer Komponente, der die Implementierung kennt, und ihrem Anwender, der in der Regel nur eine Spezifikation der Komponente in Händen hält, unterschieden werden. So kann der Entwickler im Prinzip dieselben Testverfahren einsetzen, die er bereits bisher verwendet hat, um den von ihm entwickelten Code zu prüfen.

Dazu gehören auch die kontrollflussbasierten Whitebox Testverfahren, die sich in Theorie und Praxis bewährt haben. Als objektives Testendekriterium dient hierbei der erreichte Grad der Anweisungs-, Zweig- oder Pfadüberdeckung. Diese Verfahren haben zudem den großen Vorteil, dass sie heute von vielen, zum Teil kommerziellen Testwerkzeugen unterstützt werden, die den Testprozess in vielen Bereichen unterstützen und beispielsweise den erzielten Überdeckungsgrad messen.

Im Gegensatz dazu hat der Anwender meist keinen Zugriff auf den Quellcode einer Komponente. Er weiß nicht, wie diese im Detail implementiert ist, und kann so nur Blackbox Testverfahren verwenden. Diese setzen wiederum eine möglichst vollständige Spezifikation der funktionalen und nicht-funktionalen Anforderungen an die Komponente voraus. Da eine Komponente nur dann sinnvoll eingesetzt werden kann, wenn sie vollständig dokumentiert wurde, wird aber praktisch jede Komponente standardmäßig mit einer solchen Spezifikation geliefert.

Aus Entwicklersicht stellen die spezifikationsbasierten Blackbox Testfälle den Abnahmetest dar, den die Komponente passieren muss, bevor der Entwickler sie ausliefern darf. Der Anwender kann mit Hilfe dieser Blackbox Testfälle, die im Idealfall z.B. als ausführbarer Code oder in Form von eingebauten Selbsttests mit der Komponente ausgeliefert werden, erneut prüfen, ob die Komponente auch in der neuen Einsatzumgebung korrekt funktioniert. Außerdem kann der Anwender eine eigene Spezifikation für die Komponente erstellen, die genau angibt, was er von der Komponente im Kontext seiner Anwendung erwartet. Mit Hilfe dieser Spezifikation kann er bereits im Vorfeld prüfen, ob eine ihm unbekannte Komponente alle gestellten Anforderungen erfüllt und korrekt funktioniert. Das heißt, noch bevor er sie tatsächlich in die Anwendung integriert.

Das letztendlich verwendete Blackbox Testverfahren wird durch die Art der Spezifikation bestimmt. Wie die Testfälle erzeugt, und welche Testendekriterien definiert werden, hängt direkt von der Art der Spezifikation ab. Wir unterscheiden folgende vier grundlegenden Arten von Spezifikationen (Tab. 1) und erläutern im Folgenden entsprechende Testtechniken.

Spezifikation/ Diagrammtyp	Charakter	Ausprägung	Überdeckungskriterien
Verträge (Contracts)	Formale Sprache	OCL	Bedingungsüberdeckung
	Formale Theorie	Object-Z, VDM	Alle Klauseln
	Freitext	API	“Alle Funktionen“?
Zustands-diagramme			Zustandüberdeckung, Übergangsüberdeckung, n-Path Überdeckung
Interaktionsdiagramme	Nachrichtenbasiert	SDL MSC, UML-Sequenzdiagramm	Nachrichtsüberdeckung, Knoten/ Zweigüberdeckung, Pfadüberdeckung
	Strukturbasiert	Kollaborationsdiagramm	Nachrichtsüberdeckung
Anwendungsfälle	Freitext	Jacobson	Hauptablauf, alle Ausnahmeabläufe
	Ablaufbasiert	Aktivitätsdiagramm	Knoten/Zweigüberdeckung, Pfadüberdeckung
	Nachrichtenbasiert	Sequenzdiagramme	Knoten/Zweigüberdeckung, Pfadüberdeckung

Tabelle 1: Spezifikationstechniken für Komponenten

4.1 Schnittstellentest gegen Verträge (contracts)

Die Schnittstellen einer Komponente werden häufig mit Verträgen spezifiziert, wobei für jede Operation geeignete Vor- und Nachbedingungen angegeben werden. Im Sinne des Design-by-Contract legt die Vorbedingung fest, welche Voraussetzungen ein Klient erfüllen muss, damit er die Operation überhaupt aufrufen darf, während die Komponente umgekehrt mit der Nachbedingung ein bestimmtes Resultat für den Aufruf garantiert.

Das Ziel des Schnittstellentests ist die Prüfung aller öffentlichen Operationen der *Komponente Unter Test* (KUT) [Winter01]. Voraussetzung ist, dass die Schnittstelle der KUT mit Vor- und Nachbedingungen für jede Operation sowie mit einer Invariante spezifiziert ist.

Zunächst prüft man, ob die öffentlichen Operationen der KUT ihre Spezifikation erfüllen sowie das Zusammenspiel mehrerer Operationen der KUT (*Schnittstellen-Verträglichkeitstest*). Ein Testfall im Schnittstellen-Verträglichkeitstest prüft also, ob eine Operation tatsächlich das zugesicherte Resultat produziert, wenn die Vorbedingung beim Aufruf eingehalten wurde. Zusätzlich versucht man, die für die Operationen möglichen Ausnahmen (exceptions) zu erzeugen und zu prüfen (*Schnittstellen-Robustheitstest*). Solche Verträge können entweder formal, beispielsweise mit OCL, oder auch natürlichsprachlich mit geeigneten Kommentaren formuliert werden. Der Test ist abgeschlossen, wenn alle Operationen und Bedingungen sowie ggf. alle Ausnahmen überdeckt wurden.

4.2 Zustandsautomaten (state machine)

Alternativ kann das Verhalten einer Komponente genauso wie bei einem Objekt durch einen Zustandsautomaten spezifiziert und zum Beispiel mit einem UML Zustandsdiagramm dargestellt werden. Für diese Art der Spezifikation spricht, dass es eine umfassende Theorie gibt, um Testfälle aus Zustandsautomaten abzuleiten. Allerdings lässt sich in der Praxis ein Zustandsautomat nur für relativ einfache Komponenten angeben, da ansonsten die Anzahl der Zustände und der resultierenden Testfälle zu groß wird.

4.3 Interaktionsdiagramme (interaction diagrams)

Interaktionsdiagramme haben den Vorteil, dass mit ihnen nicht nur das Verhalten einer einzelnen Komponente, sondern auch das Zusammenspiel mehrerer Komponenten spezifiziert werden kann. Aus einem Interaktionsdiagramm können, ähnlich wie bei den kontrollflussbasierten Verfahren, Testfälle ermittelt werden, die alle Knoten oder Zweige des Diagramms überdecken. Auch hier wird der Test beendet, sobald ein gewisser Grad der Überdeckung ereicht wurde [Fraikin02].

4.4 Anwendungsfälle (use cases)

Mit Anwendungsfällen kann spezifiziert werden, wie mehrere Komponenten bei der Benutzung eines komponentenbasierten Anwendungssystems zusammenspielen. Jeder *Anwendungsfall* (*use case*) formuliert eine in sich abgeschlossene Teilfunktionalität des Anwendungssystems, die für mindestens einen Akteur ein bestimmtes Ergebnis erbringt. Die Funktionalität wird dabei aus der Sicht der Akteure beschrieben und damit das Anwendungssystem als “Black-Box“ betrachtet. Jeder Anwendungsfall formuliert die möglichen Abläufe und Interaktionen, die zwischen Akteuren und dem Anwendungssystem im Rahmen der Teilfunktionalität stattfinden. Testfälle können dann den erwarteten Ablauf sowie die erlaubten Abweichungen von diesem Standardablauf gezielt prüfen. Anwendungsfälle sind in der Regel informeller Natur, da sie bereits während der Anforderungsanalyse entstehen. Dies erweist sich als hinderlich, wenn konkrete Testfälle zu entwickeln sind. Sie können hierzu durch Aktivitäts- oder Sequenzdiagramme ergänzt werden.

5 Vorgehensweise

Der Test bei der Verwendung der EJB-Architektur zerfällt in den Test des Containers und den Test der eigentlichen EJB.

5.1 Container-Test

Anhand eines applikations- (oder unternehmens-)spezifischen operationalen Profils ist jeder neu zu unterstützende Container sowie jede neue Version eines bereits unterstützten Containers im Applikationsserver zu installieren und zu prüfen. Hierzu gehören neben den Prüfungen der erforderlichen Funktionalitäten auch entsprechende Performanz-, Last- und Sicherheitstests.

5.2 Bean-Test

Der Test einer implementierten EJB gliedert sich in den Test der EJB-Container-Schnittstelle, den Test der EJB-Applikations-Schnittstelle sowie den Deployment-Test.

- EJB-Container-Test: Im EJB-Container-Test ist zunächst mit wiederverwendbaren, generischen Tests der Standard-Lebenszyklus der Bean je nach Bean-Kategorie zu prüfen. Hierzu können die in der EJB-Spezifikation angegebenen Zustandsdiagramme verwendet werden.
- EJB-Applikation-Test: Eigentlich können EJBs nur im Kontext eines EJB-Containers sinnvoll getestet werden, da dieser den nötigen Rahmen und die in Abschnitt 1 aufgezählten Dienste zur Verfügung stellt. Der Unit-Test einer EJB-Komponente in einem Container des Application Servers ist jedoch nicht praktikabel, da das Deployment der Komponente viel Zeit in Anspruch nimmt. Auch das Testen von EJBs in einem Container Test-Stub ist zum Scheitern verurteilt, da der Aufwand für die Entwicklung einer geeigneten Testumgebung (Attrappe) der Entwicklung eines echten Containers sehr nahe käme [Link02]. Man kann aber versuchen, die Enterprise Java Beans nur als Fassade für die eigentliche Fachlogik zu verwenden und so zumindest die Fachlogik außerhalb eines Containers einem Unittest zu unterziehen. Dieses Vorgehen wird auch als Box-Metapher beschrieben. Hier sieht man die Enterprise Java Beans nur als zusätzliche Service-Ebene an, die über der Ebene mit der Fachlogik angesiedelt ist. Die Fachlogik sollte dabei unabhängig von der EJB-Ebene sein. Dieses Vorgehen ist relativ leicht anwendbar für sog. Stateless Session Beans, deren Verhalten

unabhängig von ihrer Vorgeschichte ist; bei den anderen Bean Arten sind die Container-Abhängigkeiten oft zu groß.
Wird die Bean im Container getestet, so sind entsprechende Treiber-Klassen (Servlets/JSP/HTML) zu erstellen. Für den IBM WebSphere Applikationsserver bzw. den entsprechenden Container kann ein manuell zu bedienender generischer Treiber verwendet werden.

- EJB-Deployment-Test: Im EJB-Deployment-Test ist für alle unterstützten Applikationsserver bzw. Container das reibungslose Funktionieren des Deployment-Skriptes und insbesondere der entsprechenden Deployment-Deskriptoren (s. Abschnitt 3) zu prüfen.

6 Werkzeuge

Es stellt sich die Frage, ob es spezifische Anforderungen an Testwerkzeuge gibt, die zum dynamischen Testen von EJBs verwendet werden sollen. Ein Bereich, der im vorigen Abschnitt direkt ins Auge fällt, ist das Deployment der EJBs. Da das Deployment von EJBs zum einen ein komplexer und fehleranfälliger Prozess ist und zum anderen üblicherweise bei jedem Testlauf erneut durchgeführt werden muss (falls sich beteiligter Code geändert hat), ist es wünschenswert, dass ein Testwerkzeug diesen Vorgang automatisiert. Sehr bewährt hat sich hier u.a. die Verwendung von Ant [HL02][Ant].

Weitere spezifische Anforderungen an Testwerkzeuge hängen von der Frage ab, auf welchem Niveau die Tests durchgeführt werden sollen. Geht es um das Testen der Implementierungslogik, d.h. um das Testen von domänenspezifischer Funktionalität in der Teststufe Unittest, so bieten sich herkömmliche Vorgehensweisen an, da es sich bei EJBs schließlich um 'normale' Javaklassen handelt, und ein Deployment für diesen Testzweck nicht notwendigerweise durchgeführt werden muss. In der Praxis verbreitete Hilfsmittel sind hier z.B. JUnit [JUnit] und Mock Objects [Mock], eine detaillierte Übersicht gibt [Link02].

Naheliegenderweise muss ebenfalls beim funktionalen Black-Box-Test, bzw. Systemtest einer verteilten Anwendung keine besondere Vorkehrung für eventuell verwendete EJBs getroffen werden, da Tests auf dieser Ebene per Definition keine Annahmen über das Innenleben eines Systems treffen sollten. In diesem Fall reduziert sich die durch EJBs verursachte Komplexität daher rein auf das Thema (Re-)Deployment, da das zu testende System sich zumindest auf aktuellem Stand in einem ausführbaren Zustand befinden muss.

Spezielle Anforderungen an ein Testwerkzeug ergeben sich jedoch dann, wenn man Interaktionen von EJBs mit dem zugehörigen EJB Container testen möchte, da sich diese Interaktionen nicht ohne Weiteres mit herkömmlichen Mitteln beobachten lassen. Eine Möglichkeit bestünde im Implementieren eines Testcontainers, der die Interaktionen mit den EJBs prüfen könnte. Dies hat jedoch zum einen den Nachteil des sehr hohen Aufwands, der bei der Implementierung eines — wenn auch rudimentären — Containers anfällt (vgl. Abschnitt 5.2). Zum anderen könnte man durch solche Tests keine Aussage darüber treffen, ob die zu testenden EJBs auch tatsächlich im für den späteren Betrieb gedachten Container fehlerfrei arbeiten würden.

Ein sehr vielversprechender Ansatz zur Lösung dieses Problems wird von Cactus [Cactus] verfolgt. Bei Cactus handelt es sich um ein Open Source Rahmenwerk, das auf JUnit beruht und dieses erweitert. Um eine Beobachtung der Interaktion zwischen EJBs und Container zu ermöglichen, werden sogenannte Wrapper-Objekte für eine Reihe der nach dem J2EE-Standard relevanten Objekte (z.B. Request, Response, Session) zur Verfügung gestellt. Diese können bei Bedarf vom Tester mit zusätzlichen Informationen versehen, manipuliert und ausgelesen werden. Dies erfordert zwar auf der einen Seite einiges an Detailwissen über die Arbeitsweise eines Containers, erleichtert auf der anderen Seite jedoch das Herstellen von testrelevanten Zuständen sowie das Testen der Interaktionen ganz erheblich.

7 Literatur und Verweise

[Fraikin02] Falk Fraikin: SeDiTeC - Testen auf der Basis von Sequenzdiagrammen. Softwaretechnik-Trends, Band 21, Heft 21, 2002, S. 8

[HL02] Richard Hightower, Nicholas Lesiecki: Java Tools for Extreme Programming. John Wiley & Sons, 2002

[Link02] Johannes Link, Unit Tests mit Java, dpunkt-Verlag, 2002

[Szyp98] Clemens Szyperski: Component Software - Beyond Object-Oriented Programming. Addison-Wesley / ACM Press, 1998. ISBN 0-201-17888-5.

[Winter01] Mario Winter: Testfallermittlung aus Komponentenschnittstellen. Beitrag zum Imbus QS-Tag 01, Nürnberg, 2001

[Ant] http://jakarta.apache.org/ant/

[Cactus] http://jakarta.apache.org/cactus/

[JUnit] http://www.junit.org/

[Mock] http://mockobjects.sourceforge.net/

[MS] http://www.microsoft.com/com

[OMG] http://www.omg.org

[Sun] http://java.sun.com/products/ejb