von rudolf hauber, markus ziegler,
michael erskine und rolf hilsenbeck

MODELLBASIERTES TESTEN

Der Artikel stellt die Erfahrungen mit der modellbasierten Automatisierung des Testens in Sinne der „Model Driven Architecture" (MDA) in mehreren Großprojekten bei der Firma EADS vor. Es wird gezeigt, wie die verschiedenen Abstraktionsebenen eines Systemmodells für das Testen genutzt werden können. Hierfür werden drei Aspekte des Testens vorgestellt: das Testen der fachlichen Logik und der Modellierungsrichtlinie auf Ebene des fachlichen Modells, das Testen der Modelltransformation vom fachlichen zum Implementierungsmodell sowie das automatische Erstellen von Klassen- und Komponententests aus dem Modell zum Test des Quellcodes.

Firmenhintergrund

Der EADS-Konzern erstellt seit vielen Jahren erfolgreich sehr komplexe Systeme. Sie bestehen aus unterschiedlichsten Hard- und Softwarekomponenten, die in höchst heterogenen Umgebungen zu einem Gesamtsystem integriert werden. Aufgrund der Konzernstruktur wirken häufig verschiedene konzerninterne Unternehmen gemeinsam an einem Projekt mit. Dies muss bei der Vorgehensweise berücksichtigt werden.

Um die hohe Komplexität und die daraus folgende Informationsflut zu meistern, wurden in zahlreichen Projekten neue Methoden und Vorgehensweisen eingeführt, so z. B. die Erstellung großer Systemmodelle mit der UML (vgl. [Hau01]). Diese zunehmend modellbasierte Arbeitsweise stellt die Qualitätssicherung vor die Frage, wie die Konsistenz von Modell und Code sichergestellt werden kann. Immer öfter wird das Systemmodell zum frühzeitigen Nachweis (Verifikation) und Test des Systemverhaltens genutzt. Denn weiterhin gilt der Grundsatz: Je früher ein Fehler erkannt wird, desto billiger ist seine Behebung. Modellbasierte Entwicklung bietet hier ganz neue Ansatzpunkte: Durch Testen bereits in frühen Phasen können zuerst fachliche und dann Designfehler entdeckt, analysiert und behoben werden. Wer diese Möglichkeiten nicht nutzt, verschenkt einen entscheidenden Vorteil der modellbasierten Entwicklung (vgl. [Hau02]).

Ein zusätzlicher Vorteil des modellbasierten Vorgehens ist die Möglichkeit, neben der Systemdokumentation auch einen großen Teil der Testdokumentation zu automatisieren: von der Prüfspezifikation über die Prüfprozedur bis hin zur Protokollierung der Tests.

Verschiedene Testebenen

Große Projekte entwickeln umfangreiche Systemmodelle auf verschiedenen Abstraktionsebenen. Im Sinne der *Model Driven Architecture (MDA)* (vgl. [OMG]) handelt es sich auf der fachlichen Modellebene um ein *Platform Independent Model (PIM)*. Dieses wird über mehrere Verfeinerungsebenen in ein *Platform Specific Model (PSM)* für eine konkrete Implementierungsplattform transformiert. Der Nachweis dieser komplexen Systeme erfolgt dabei auf verschiedenen Stufen. Typische Teststufen sind

- Systemtests,
- Subsystemtests,
- Komponententests und
- Klassentests.

Auf jeder dieser Stufen kann die Information aus dem Systemmodell genutzt werden: zur Testplanung, -spezifikation, -ausführung, -auswertung und -dokumentation. In diesem Artikel werden Techniken für die verschiedenen Stufen erläutert.

Konsistenz des Modells

So trivial es klingt, die Voraussetzung für die Nutzung des Systemmodells für den Test ist die Konsistenz und Korrektheit des Modells. Da ein Modell verschiedene Sichten auf das System und unterschiedliche Abstraktionsebenen umfasst, müssen bereits hier erste modellinterne Maßnahmen ergriffen werden. Beim Modellieren helfen konstruktive Maßnahmen, die Inkonsistenzen gar nicht erst aufkommen lassen – wenn sie konsequent beachtet werden. So sollten zum Beispiel vorhandene Klassen und Operationen in Interaktionsdiagrammen verwendet werden und zusätzlich benötigte Nachrichten gleichzeitig auch als Operationen in die entsprechenden Klassen aufgenommen werden. CASE-Werkzeuge unterstützen diese Arbeit erheblich, z. B. durch Drag&Drop oder durch Auswahllisten.

Konstruktive Maßnahmen alleine reichen jedoch nicht aus; zusätzlich sind analytische Maßnahmen erforderlich, die die

die autoren

Dr. Rudolf Hauber (E-Mail: RHauber@ka-muc.de) ist bei der Kölsch & Altmann Software & Management Consulting GmbH langjähriger IT-Berater in einer Vielzahl von Großprojekten, in denen er OO -Methoden und -Tools einführt bzw. schult.

Markus Ziegler (E-Mail: Markus.Ziegler@lfk.eads.net) leitet ein integriertes Software-Team bei der EADS/LFK GmbH.

Michael Erskine (E-Mail: Michael.Erskine@lfk.eads.net) ist bei der EADS/LFK GmbH im Bereich Software-Engineering tätig.

Rolf Hilsenbeck (E-Mail: Rolf.Hilsenbeck@dornier.eads.net) ist Projektverantwortlicher für die Softwareentwicklung in einem Großprojekt bei der EADS/Dornier GmbH.

Prüf-ID	Testinhalt	Status
AF01	Gibt es für jeden Use-Case eine Use-Case-Realisierung?	Error
AF02	Gibt es mindestens einen Zustandsautomaten im Use-Case-View?	Error
AF03	Hat jedes Package im Use-Case-View den Stereotyp «use-case package»?	Error
AF04	Gibt es eine Word-Beschreibungsdatei für jeden Use-Case?	Error
AF05	Gibt es ein „Main"-Diagramm in jedem Use-Case-Package?	Error
AF06	Gibt es mindestens ein Sequenzdiagramm pro Use-Case-Realisierung?	Error
AF07	Gibt es einen Use-Case zu jeder Aktion im Zustandsautomaten des Use-Case-View?	Warning
AF08	Gibt es eine Aktion in einem Zustandsautomaten zu jedem Use-Case?	Warning
AF09	Sind alle Diagramme dokumentiert?	Warning
AF010	Sind alle Zustände dokumentiert?	Warning
AF011	Sind alle Use-Cases dokumentiert?	Warning
AF012	Gibt es für jedes Use-Case-Package eine Beschreibung?	Warning
AF013	Sind die Akteure in eigenen «actor package»-Paketen im Use-Case-View definiert?	Error
AF014	Hat jeder Akteur mindestens eine Kommunikationsbeziehung?	Error

Tabelle 1: Prüfung der Modellierung von Anwenderforderungen

Konsistenz des Modells systematisch prüfen. Für große Systeme mit vielen Entwicklern sind die syntaktischen Tests des Modells, die in den CASE-Tools enthalten sind (z. B. „Sind Objekte klassifiziert?"), jedoch häufig nicht hinreichend. Nach jeder Modellierungssitzung sollten mindestens die automatischen Konsistenzprüfungen der CASE-Werkzeuge fehlerfrei absolviert werden. Der fachlich-logische Abgleich ist projektunabhängig nur schwer automatisierbar. Für den jeweiligen Anwendungsbereich lassen sich häufig Regeln aufstellen und bei geeigneter Modellierung skriptgestützt automatisch überprüfen.

Testen der fachlichen Logik

Für das anwendungsbereichsspezifische Testen der fachlichen Logik kann das Systemmodell auf verschiedene Weise genutzt werden. Es sollten die Einhaltung der Modellierungsrichtlinien, die Abläufe der Use-Cases und die Zusammenhänge der Use-Cases geprüft werden.

Überprüfung der Modellierungsrichtlinien

An der Modellierung komplexer Systeme ist meist eine Vielzahl von Systemingenieuren, Designern und Entwicklern beteiligt. Um ein für alle verständliches und firmenweit einheitliches Modell zu erhalten, ist die Aufstellung einer Modellierungsrichtlinie – wie sie auf Codeebene selbstverständlich ist – von entscheidender Bedeutung. Um die Vorteile automatischer Dokumentationsgenerierung und Modelltransformationen nutzen zu können, ist ein einheitlicher Modellaufbau Voraussetzung. Eine Modellierungsrichtlinie legt Namenskonventionen, verbindliche Modellstrukturen und Standarddiagramme fest. Daneben legt sie den anwendungsspezifischen Gebrauch der UML fest, d. h. welche Modellierungskonstrukte wie verwendet und welche UML-Erweiterungen eingesetzt werden.

Viele dieser Vorgaben können automatisiert überprüft werden. Dies können Skripte übernehmen, die die üblichen offe-

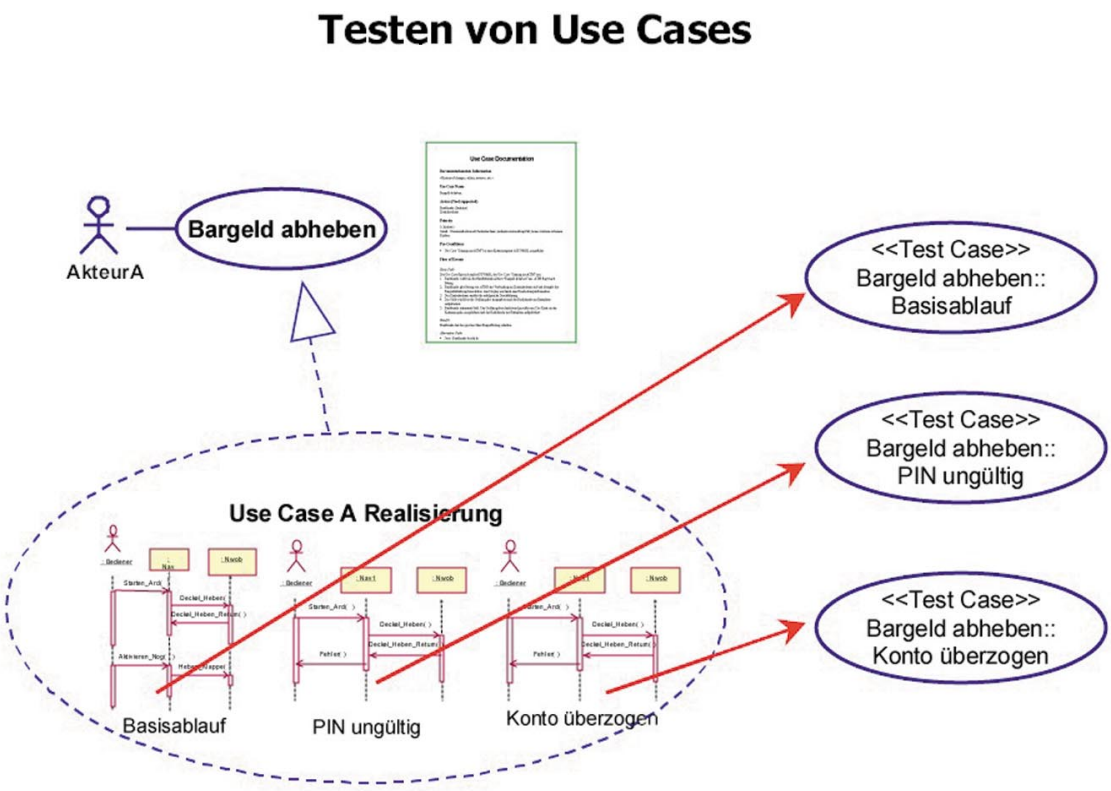

Abb. 1: Nachweis von Use-Cases, basierend auf Szenarien

nen Programmierschnittstellen von Modellierungswerkzeugen nutzen. Dadurch werden Design-Reviews drastisch entlastet, da man sich auf rein fachliche Aspekte konzentrieren kann.

Tabelle 1 enthält eine Liste von Prüfungen eines Modells; diese beruhen auf einer Modellierungsrichtlinie, die festlegt, wie Anwenderforderungen modelliert werden sollen. Neben diesen formalen Tests kann das PIM für semantische Tests genutzt werden, um funktionale Aspekte möglichst früh nachzuweisen.

Testen von Use-Cases

Die funktionalen Aspekte werden in der Regel in einem Use-Case-Modell dokumentiert. Damit stellt sich die Frage: Wie können Use-Cases getestet werden und in welchen Systemzuständen dürfen sie ausgeführt werden? Use-Cases beschreiben dabei eine Funktionalität in allgemeiner Form als Folge von Interaktionen zwischen externen Einheiten (Akteuren) und dem System. Das System wird als Blackbox betrachtet. Dabei kann ein Use-Case mehrere mögliche konkrete Abläufe (Szenarien) umfassen, die für einen Blackbox-Test genutzt werden. Die konkreten Szenarien liegen im Modell als Sequenzdiagramme für Basis- und Alternativabläufe der Use-Cases vor. Dies ist eine Formalisierung der textuellen Use-Case-Beschreibung, die die Realisierung des Use-Cases durch Systemelemente (Subsysteme, Komponenten oder Klassen) zeigt. Das System muss dazu in einem Architekturschritt vorher in Subsysteme, Komponenten oder Klassen zerlegt sein. Die Sequenzdiagramme stellen die zu verifizierende Spezifikation im Sinne des Testens dar (vgl. Abb. 1). Diese Zerlegung kann sowohl für System- als auch für Subsystem-Use-Cases erfolgen.

Jeder Ablauf, d. h. jedes Sequenzdiagramm, wird durch einen Testfall (*Test-Case*) nachgewiesen, der genau die externen Schritte des Sequenzdiagramms abspielt und die erwarteten Ergebnisse mit den beobachteten Resultaten vergleicht.

Die Use-Case-Schritte liegen in Form von Nachrichten in den Sequenzdiagrammen vor. Ein *Tracing* der Nachrichten auf die Beschreibungsschritte in den textuellen Use-Case-Dokumenten kann bei der Szenarienmodellierung aufgebaut werden (z. B. Nachrichteneigenschaft *{UseCase Schritt= nn}*).

Die in das System fließenden Nachrichten werden im Test-Case-Design zu Testschritten. Die aus dem System gehenden Nachrichten werden zu Verifikationspunkten und dienen der Prüfung der Zwischenergebnisse.

Diese Vorgehensweise erlaubt eine automatische Generierung der Test-Cases für ein Testmanagement-Tool. Die entstehenden Test-Cases werden in einer XML-Datei abgelegt, die in das Testmanagement-Werkzeug importiert wird. Das *Tracing* der Test-Cases auf die Use-Cases wird dabei automatisch aufgebaut. Auch eine Vielzahl weitergehender individueller Testinformationen zu einem Testfall wird beim Export aus dem Modellierungswerkzeug definiert (Vorbedingung, Nachbedingung, Nachweisart, Iteration, Konfiguration etc.). Vor- und Nachbedingungen werden aus den hinterlegten Use-Case-Beschreibungen, die als Word-Datei vorliegen, automatisch extrahiert.

Die Testschritte sind implizit durch die externen Nachrichten definiert; d. h. jede Nachricht von oder zu einem Akteur eines Sequenzdiagramms stellt einen Testschritt bzw. Verifikationspunkt des Test-Cases dar. Nicht berücksichtigt werden muss die interne Kommunikation zwischen den Subsystemen. Ein Skript analysiert die Sequenzdiagramme, extrahiert die Testschritte und exportiert die Informationen in eine XML-Datei. Hierzu ein Beispiel: Das Sequenzdiagramm zu dem Szenario „Bargeld abheben" zeigt Abbildung 2.

Im Beispiel gibt es die eingehenden externen Nachrichten eingegebenKarte(), eingegebenPIN(), ausgewaehltBargeldabhebung(), eingegebenBetrag(), commitedTransaction(), entnommenKarte() und entnommenBargeld() sowie die ausgehenden Nachrichten der Bildschimausgaben und tryTransaction().

Aus den Nachrichten ergibt sich das Test-Case-Design in Abbildung 3. Die Use-Case-Schritte und die Testschritte müssen sich *nicht 1:1* entsprechen.

Die Testdaten werden durch Nachrichteneigenschaften (z. B. *{test=(betrag= 400)}*) bereits im Sequenzdiagramm festgelegt. Dabei muss es sich entweder um fundamentale Datentypen handeln oder die Daten müssen über eine Testdaten-ID eindeutig in einem Testdaten-Pool identifizierbar sein.

Der größte Teil der gesamte Testplanung und -spezifikation wird somit automatisiert aus den Use-Case-Beschreibungen und den Use-Case-Realisierungen abgeleitet. Zur Validierung des Gesamtsystems sollte diese Vorgehensweise jedoch nicht verwendet werden.

Testen von Zustandsautomaten

Zustandsautomaten und Aktivitätsdiagramme sind für Tests besonders geeignet, da sie es erlauben, das dynamische Verhalten vollständig zu beschreiben. Die Tests prüfen das korrekte Verhalten bei allen Zustandsübergängen und können den gesamten Zustandsraum überdecken; auch Verklemmungen können von ihnen gefunden werden.

Voraussetzung dafür ist, dass das Systemverhalten oder entsprechende Aspekte vollständig modelliert sind. Beispielsweise kann ein Zustandsautomat alle Systemzustände erfassen. Innerhalb der Automaten-Zustände stehen dann die Aktionen für die Use-Cases, die im entsprechenden Systemzustand ausgeführt werden können (vgl. Abb. 4). An Zu-

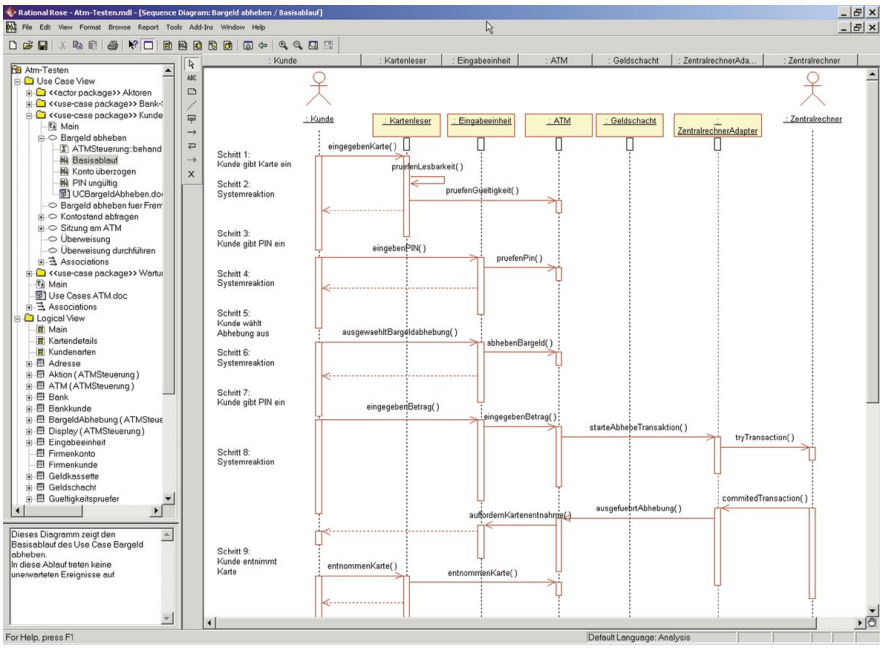

Abb. 2: Sequenzdiagramm Akteurnachrichten

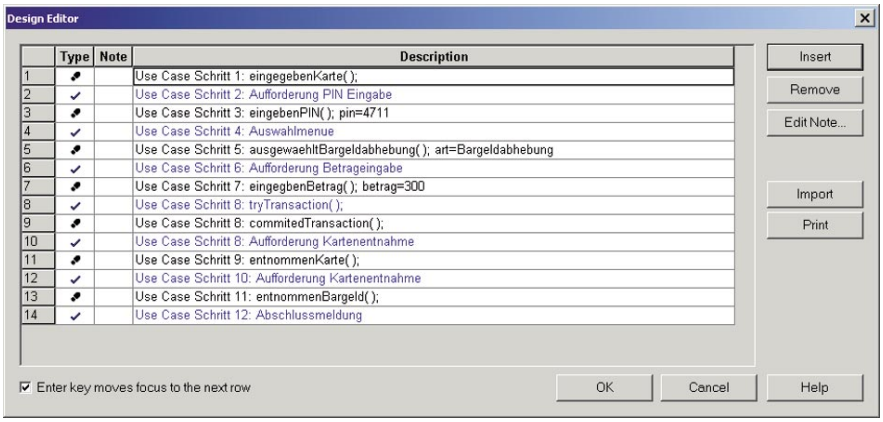

Abb. 3: Test-Case-Design

standsübergängen können als auslösende Ereignisse auch spezielle Schritte aus den Use-Cases im Format *Use-Case Name [Schritt als Bedingung]* angegeben werden. Damit wird der fachliche Zusammenhang von Use-Cases und Sicherheitsaspekten erfasst.

Je nach Mächtigkeit des eingesetzten CASE-Tools wird diese Information für das Testen genutzt:

- Im einfachsten Fall wird statisch die Vollständigkeit der Zuordnung Use-Case zu Aktionen geprüft: Wurde ein Use-Case vergessen? Gibt es zu jeder Aktion einen Use-Case?
- Die nächste Stufe ist der dynamische Test. Dabei werden Sequenzdiagramme der Use-Case-Realisierungen gegen die Use-Case-Zustandsautomaten dynamisch getestet. Diese Möglichkeit bietet z. B. „Rational Rose RealTime".
- Die höchste Teststufe ist die Simulation des Systemverhaltens. Mit dieser Testform wird die Vollständigkeit des Systems geprüft: Welche Use-Cases müssen mindestens ausgeführt werden, um alle Systemzustände zu durchlaufen? Oder: Welche Use-Cases müssen ausgeführt werden, um zu einem bestimmten Zustand zu kommen?

Testen der Modelltransformationen PIM-PSM

Während der Verfeinerung im Design unterliegt das fachliche Modell (PIM) Transformationen, um zu einer Implementierungsvorgabe für eine konkrete Plattform zu werden. Einige dieser Transformationen können automatisch erfolgen: vom Generieren einfacher Coderahmen und Datenbanktabellen über Datenzugriffsschichten, Framework-Anbindungen (EJB, CORBA) bis hin zur Systemimplementierung (z. B. „Rose Real Time", „Tau", „Rhapsody").

Das Erstellen dieser Transformatoren erfordert besondere Sorgfalt, da man sich anschließend auf ihre Korrektheit verlässt. Sie sollten zuerst an einfachen Verifikationsmodellen während der Entwicklung getestet werden und danach am konkreten Systemmodell. Funktioniert der Transformator erst einmal, wird er implizit durch erfolgreiche Komponenten- und Klassentests stets neu getestet.

Dabei hat sich ein iterativer Ansatz für die Entwicklung der Transformatoren bewährt. Ein Demonstrator dient zur Erforschung der Fähigkeiten und Probleme. Die zweite Ausbaustufe justiert das Generierungskonzept fein und liefert lauffähigen Code für ein Probemodell. Die dritte Ausbaustufe wird auf das konkrete Modell angewendet.

Durch das iterative Vorgehen entstehen schnelle Rückmelde-Zyklen, die helfen, das Risiko allzu ambitionierter Transformatoren für das Projekt zu reduzieren.

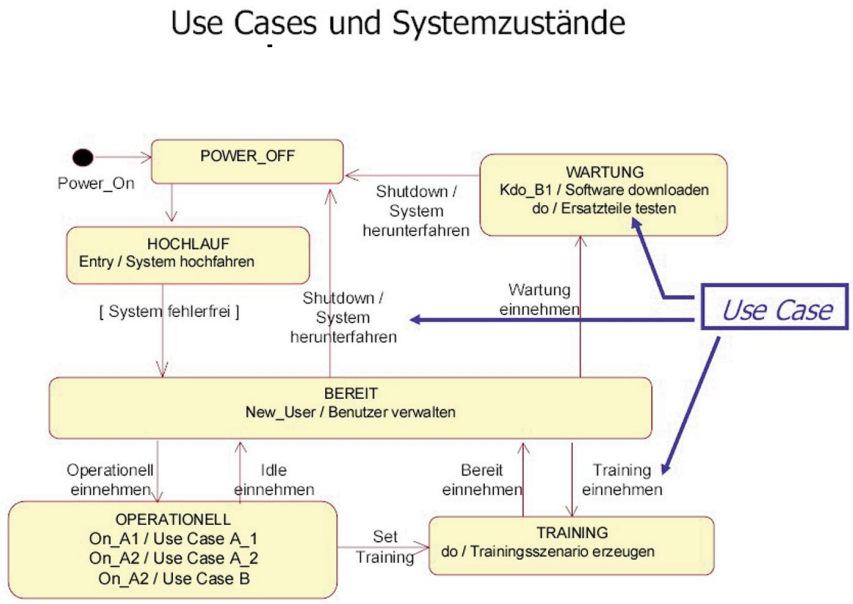

Abb. 4: Systemzustände und Use-Cases

Testen des Quellcodes

Das PSM enthält Informationen, aus denen automatisch Tests generiert werden können. Diese Tests prüfen, ob der Quellcode genau das im PSM spezifizierte Verhalten umsetzt.

Komponenten- und Klassentests

Für den Komponenten- und Klassentest werden aus dem PSM funktionale Tests generiert. Basierend auf den Modellinformationen erzeugt ein Skript den Testcode. Dabei füllt das Skript eine Testcode-Schablone, die auf den bewährten Testframeworks „JUnit" (vgl. [Sou-b]) oder „CppUnit" (vgl. [Sou-a]) beruht. Das Skript extrahiert die relevanten Informationen aus dem Modell und schreibt sie in die Schablone (**vgl. Abb. 5**).

Der Test selbst orientiert sich an den Schnittstellen. Für jede Komponente und Klasse wird eine Testfallklasse generiert, deren Aufgabe der Test der Schnittstellenoperationen ist. Vor- und Nachbedingungen des Tests werden aus dem Modell ermittelt und als Kommentar im Testcode abgelegt.

Durch eine Trennung der verschiedenen Aspekte (PSM, Testcodegenerierung, klassentyp-spezifische Testcode-Schablonen) ist die Testfallgenerierung sehr flexibel. Die Generierung ist unabhängig von Modelländerungen: Ändert sich der Modellinhalt, muss lediglich der Testcode neu generiert werden. Auch Änderungen am Testcode werden sehr einfach, da nur die Schablone angepasst werden muss. Die Schablone nutzt intern die Klassen aus dem Testframework. Die Entwickler erhalten dadurch die Basis für einheitlichen und robusten Testcode.

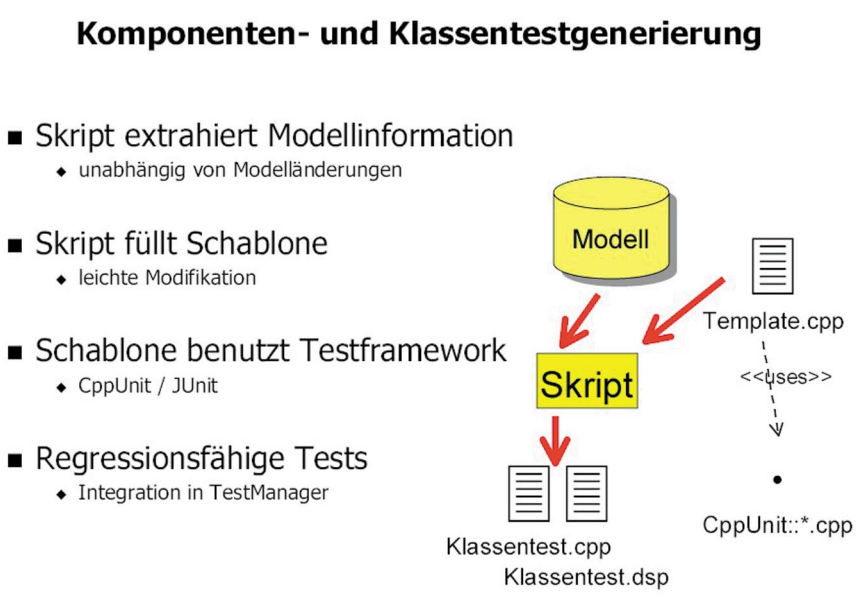

Abb. 5: Automatische Generierung von Komponenten- und Klassentests

Benötigte Test-*Stubs* können über Abhängigkeitsbeziehungen automatisch ermittelt und über *Mock*-Objekte (vgl. [Moc]) integriert werden. Dadurch entstehen intelligente *Stubs*, um Fehlerverhalten simulieren zu können. Ferner können aus Interaktionsdiagrammen Zusatzinformationen extrahiert werden, die in den Testcode-Rahmen geschrieben werden (z.B. Überprüfung spezieller Nachrichten).

Diese modellgestützte Vorgehensweise liefert regressionsfähige und jederzeit aktualisierbare Tests. Die Testfallklassen werden dafür in ein Testmanagement-Tool eingebunden. Bei Codeänderungen werden die Tests durch das Werkzeug neu übersetzt und ausgeführt.

Testdatenauswahl

Die Testdaten werden so ausgewählt, dass das Sequenzdiagramm vollständig durchlaufen wird. In einfachen Fällen genügt es, skalare Werte zu wählen. Diese werden als Nachrichten-Eigenschaft des entsprechenden Interaktionsdiagramms hinterlegt. Für komplexe Testdaten ist dies nicht möglich. Sie können jedoch in einem Testdaten-Pool hinterlegt und über ihre eindeutige Testobjekt-ID identifiziert werden.

Als Testdaten-Pool für die Testdatenbank empfiehlt sich eine Lösung auf der Grundlage von XML, dem De-facto-Standard zur Beschreibung von Daten. Die komplexen Testdaten sind Instanzen von komplexen Klassen, die im Modell definiert sind. Für die komplexen Klassen können aus dem Modell direkt Datentyp-Definitionen oder XML-Schemata generiert werden. Über XSL-*Stylesheets* können dann auf einfache Weise komfortable Oberflächen erzeugt werden, die die Eingabe von konkreten Testdaten erlauben

Der Zugriff auf den XML-Testdaten-Pool kann mittels Open-Source-Parsern (siehe z. B. [Apa]) generisch ausgelegt werden. Über der Zugriffsschicht liegt dann eine aus dem Modell generierte Mapping-Schicht, die die Daten aus dem Modell auf die Daten der Zugriffsschicht abbildet. (**siehe Abb. 6**). Demnächst kann hier das „Java Architecture for XML Binding" (vgl. [Sun]) genutzt werden.

Komplexe Ergebnisobjekte sollten ebenfalls als XML-Dateien abgelegt werden. Die Datenauswertung erfolgt automatisiert oder über XSL-*Stylesheets* mit visuellen Oberflächen.

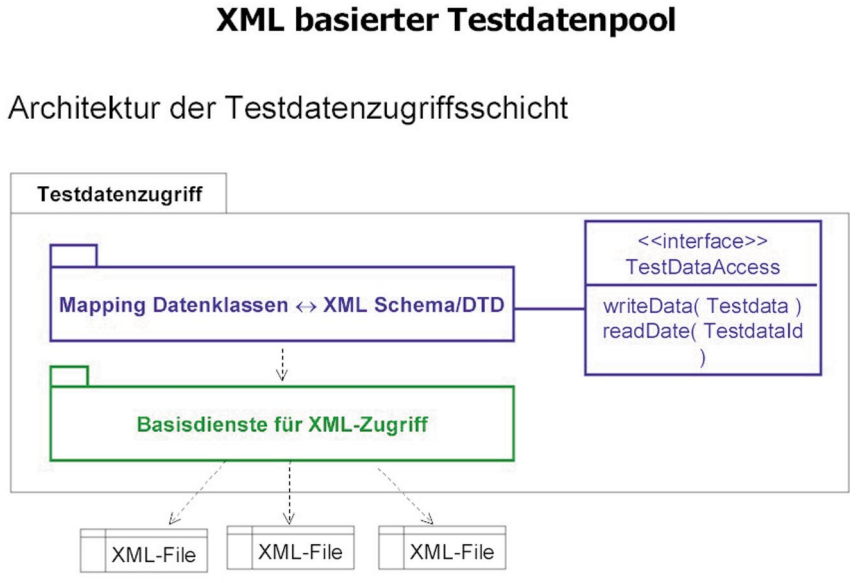

Abb. 6: Testdaten-Pool

Zusammenfassung

Modellbasierte Entwicklung ist inzwischen Stand der Technik. Die Nutzung der Modelle zum frühzeitigen Nachweis und Test steckt jedoch erst in den Kinderschuhen. Hier muss sich noch einiges tun, um die Qualität der Systeme vom ersten Moment an zu erhöhen. Der Artikel zeigt nur einige Möglichkeiten dazu auf. Entscheidend ist, dass Testen als integraler Bestandteil des Entwicklungsprozesses – von der ersten Anforderungssimulation bis zum automatisierten Implementierungstest – gelebt wird. Dabei kann ein modellbasiertes Vorgehen einen entscheidenden Beitrag liefern. ■

Literatur & Links

[Apa] The Apache Software Foundation, Apache XML Project, siehe: xml.apache.org

[Hau01] R. Hauber, M. Reinhold, T. Rittel, D. Wagner, Große Systeme in den Griff bekommen, in: OBJEKTspektrum 5/01

[Hau02] R. Hauber, T, Rittel, D. Wagner, Projektmanagement mit Requirements-Engineering und UML im Griff, in: OBJEKTspektrum 3/02

[Moc] Mock-Frameworks, OFFIS, EasyMock: www.easymock.org; Mock Maker: www.mockmaker.org; Mock Objects: www.mockobjects.com

[OMG] Object Management Group, OMG Model Driven Architecture (siehe: www.omg.org/mda)

[Sou-a] SourceForge.net, C++ Testframework CppUnit: cppunit.sourceforge.net

[Sou-b] SourceForge.net, Java Testframework JUnit, siehe: www.junit.org

[Sun] Sun, Java Architecture for XML Binding (siehe: java.sun.com/xml/jaxb)