mehr zum thema:
www.sensler.de/MDA
www.michael-kunz.de/MDA

TESTAUTOMATISIERUNG MIT MODELLGETRIEBENER TESTSKRIPT-ENTWICKLUNG

Die Vorteile modellgetriebener Entwicklung lassen sich auch für die Zwecke der Testautomatisierung nutzen. Grundlagen hierzu sowie eine Konzeption und die praktische Umsetzung – zugeschnitten auf ein einzelnes Projekt – werden im ersten Teil dieses Artikels vorgestellt. Der zweite Teil stellt für einen projektübergreifenden Einsatz Weiterentwicklungen und Verallgemeinerungen auf Basis von Kaskadierungen generativer Architekturen dar.[1])

Notwendigkeit von Testautomatisierung

Die Komplexität von Softwaresystemen nimmt zu und fachliche und technische Veränderungen in der Wartungs- und Weiterentwicklungsphase gewinnen an Bedeutung. Softwarefehler können dabei zu großen Schäden führen. Um die Qualität von Software aus Kunden- und Herstellersicht zu sichern, kommen verschiedene Prinzipien, Konzepte, Methoden und Werkzeuge während des Entwicklungsprozesses und der Wartung zum Einsatz. Die Systeme können häufig nicht mehr mit vertretbarem Aufwand manuell getestet werden. Das gilt insbesondere für durchzuführende Regressionstests.

Automatisierte Modultests sind im Vergleich zu automatischen Tests grafischer Oberflächen in der Regel mit „angemessenem" Aufwand z. B. durch das JUnit-Framework (vgl. [Gam01]) zu realisieren.

Für Tests grafischer Benutzungsoberflächen gibt es im Wesentlichen zwei verschiedene Ansätze, um eine werkzeuggestützte Automatisierung zu realisieren. Der erste Ansatz ist die Verwendung von so genannten *Capture-/Replay*-Werkzeugen, die einen Mechanismus für das Aufzeichnen und Wiedergeben der Benutzeraktivitäten besitzen. Eines der bekanntesten Werkzeuge für diesen Ansatz ist „Apache JMeter" (vgl. [Apa]). Der zweite Ansatz nutzt die Möglichkeit, Anwendungen durch Programmierung der Benutzeraktivität zu testen. Bei diesem Ansatz werden spezifische Programmierschnittstellen verwendet, um programmatisch eine Simulation der Benutzeraktivitäten umzusetzen. Ein Vertreter dieses Ansatzes ist beispielsweise das OpenSource-Projekt „HTTPUnit" (vgl. [Gol03]).

Capture-/Replay-Testverfahren von Benutzungsoberflächen

Capture-/Replay (CR) steht für das Aufzeichnen und Wiedergeben von Anwendungsverhalten und Interaktionen meist auf der Ebene der Oberfläche einer Anwendung. Dies wird üblicherweise durch die Zwischenschaltung eines Proxys ermöglicht.

Dabei setzt der Tester die zuvor erstellten Testfallspezifikationen durch Nutzung der Anwendung manuell um (siehe Abb. 1). Das aufgezeichnete Testskript muss nachbearbeitet werden, um eine Parametrisierung (z. B. für *Session-Handling*-Testdaten) sowie eine Überprüfung des erwarteten Anwendungsverhaltens sicherzustellen (*Checkpoints* bzw. Zusicherungen). Das Ergebnis der Schritte Testfallspezifikation, *Capture* und Nachbearbeitung ist ein parametrisiertes Testskript in der Sprache des gewählten Testwerkzeugs für einen spezifischen Testfall und eine spezifische Anwendungsversion. Das *Replay* des Testskripts kann danach wiederholbar mit dem jeweiligen Testwerkzeug vorgenommen werden, bis sich die Fachlichkeit oder Technik der Anwendung verändern. Dieses Vorgehen ist für jeden zu automatisierenden Testfall durchzuführen.

Die Wartungsfalle

Wird z. B. bei einer Software für Lebensversicherungen auf einer Maske zur Personendatenerfassung ein Feld „Enkel" im Zuge einer Weiterentwicklung der Anwendung hinzugefügt, so hat diese Änderung zur Folge, dass jedes Testskript, das diese Maske anspricht, angepasst werden muss. In vielen Fällen ist es nicht sinnvoll, die Änderungen in den aufgezeichneten Testskripten vorzunehmen, sodass eine erneute Aufzeichnung notwendig wird. Ursache hierfür ist, dass viele CR-Tools nur unzureichende Modularisierungen der Testskripte zulassen. So bietet beispielsweise JMeter keinen Mechanismus, um Subprozesse zu definieren und aufzurufen, sodass diese redundant in dem jeweiligen Hauptprozess auskodiert werden müssen. Grundlegende technische Änderungen der

[1]) *Der Artikel basiert auf der Diplomarbeit von Carsten Sensler, die auf der OOP im Rahmen des Diplomarbeitswettbewerbs von JavaSPEKTRUM („Java Award für Innovation 2006") den ersten Platz belegte; siehe:* www.sigs-datacom.de/sd/publications/ js/innovation_award_2006.htm

die autoren

Carsten Sensler (E-Mail: Carsten.Sensler@bluecarat.de) ist Berater bei der blueCarat AG mit dem Schwerpunkt Qualitätssicherung von Web-Anwendungen sowie Konzeption und Umsetzung modellgetriebener Architekturen.

Michael Kunz (E-Mail: Michael.Kunz@bluecarat.de) ist bei der blueCarat AG verantwortlich für strategische Technologiefragen, Konzeption und Umsetzung (modellgetriebener) Architekturen sowie die Gestaltung von Entwicklungsprozessen.

Peter Schnell (E-Mail: Peter.Schnell@aldautomotive.de) ist Projekt- und Abteilungsleiter bei der ALD Autoleasing D GmbH. Er hat mehrjährige Erfahrung mit MDA und war für die erfolgreiche Durchführung von zwei MDA-Großprojekten verantwortlich.

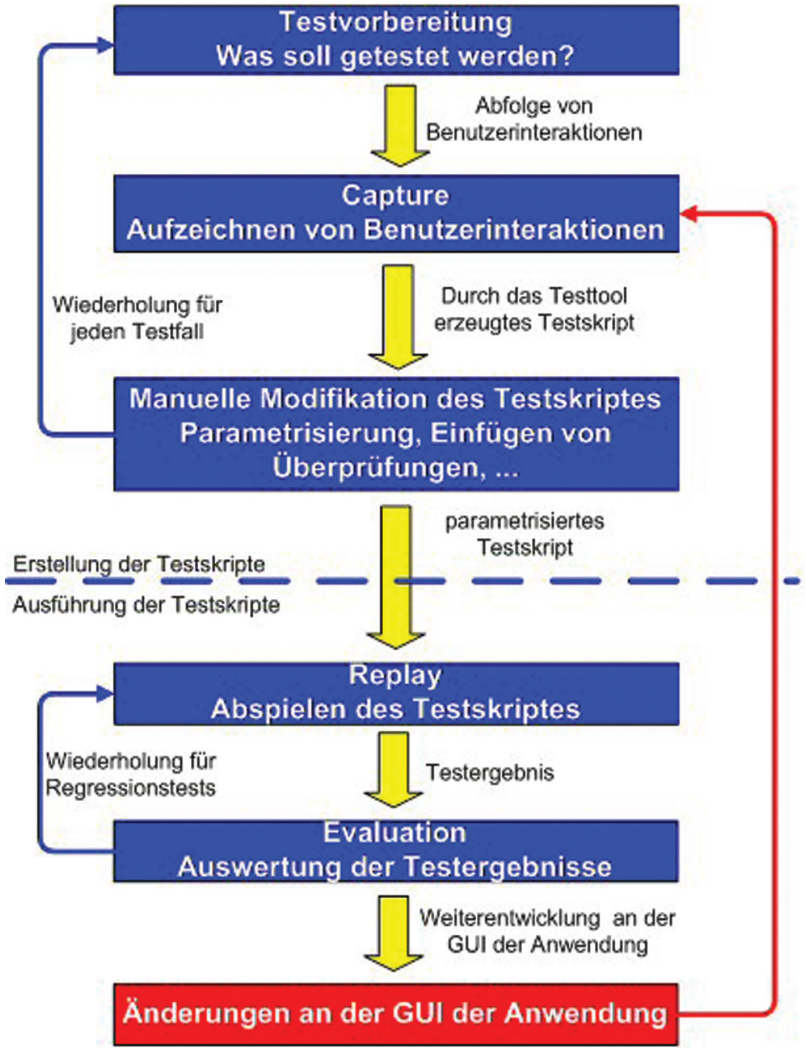

Abb. 1: Konventionelles CR-basiertes Testverfahren

Anwendung – z. B. eine Umstellung des Login-Mechanismus bei einer Web-Anwendung – führen zu einem flächendeckenden Änderungsbedarf.

Dieses Problem der Wartungsfalle kann mit einem modellgetriebenen Vorgehen gemildert werden (siehe Abb. 2). Die fachliche Beschreibung von Testfällen in einer testwerkzeugunabhängigen domänenspezifischen Sprache und die daraus im Wege einer Modell-zu-Code-Transformation generierte technische Umsetzung in Testskripten entkoppelt die fachliche Beschreibung der Testfälle von ihrer technischen Umsetzung in Form von Testskripten.

Für ein großes deutsches Versicherungsunternehmen wurde eine konzernweite Postkorb-Anwendung auf Web-Basis entwickelt. Für diese Anwendung wurde im Rahmen einer Diplomarbeit (vgl. [Sen05]) eine modellgetriebene Testskript-Entwicklung konzipiert und umgesetzt. Zum Einsatz kamen hierbei das b+m Generator-Framework 2.0 (vgl. [b+m06]) und in einer Weiterentwicklung „openArchitectureWare 3.0" (vgl. [Ecl]). Als Zielplattform der Testskripte diente JMeter.

Vorgehensmodell für eine modellgetriebene Testskript-Entwicklung

Beim Einsatz modellgetriebener Testskriptgenerierung müssen MDA-zentrierte Vorgehensmodelle wie z. B. der „b+m Generative Development Process" (vgl. [b+m03]) im Architekturstrang um zusätzliche Aktivitäten erweitert werden. Die hierbei relevanten Aktivitäten sind in Abbildung 3 dargestellt: Die erste Phase „Analyse der GUI des Testlings" dient dazu, die Anwendung zu verstehen und auf Aspekte zu untersuchen, die prinzipiell ein CR-basiertes Testen ausschließen. Ein Beispiel für ein Ausschlusskriterium sind dynamisch zur Laufzeit erzeugte veränderliche IDs von grafischen Komponenten statt fester, eindeutiger Namen. In derartigen Fällen kann weder ein konventionelles noch ein modellgetriebenes CR-basiertes Testen durchgeführt werden, da sich aus Sicht der Testskripte die Anwendung bei jedem Durchlauf ändert.

Bei der Erstellung einer Referenzimplementierung werden mit dem gewählten CR-Testwerkzeug alle Masken und Transitionen zwischen diesen aufgezeichnet.

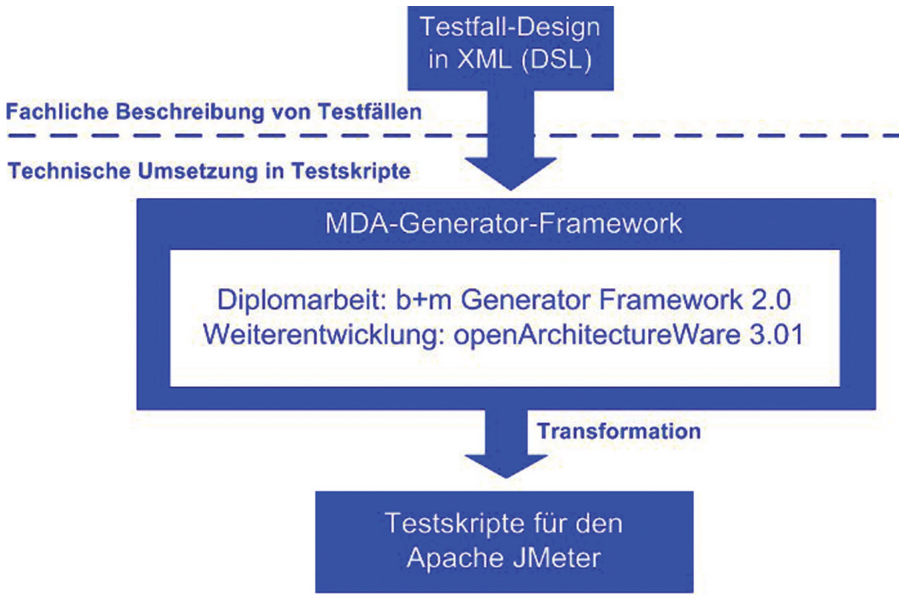

Abb. 2: Entkopplung der fachlichen Beschreibung von der technischen Umsetzung bei modellgetriebener Testskript-Entwicklung

Abb. 3: Vorgehensmodell für modellgetriebene Testskript-Entwicklung

In einer dritten Phase werden die Ergebnisse der Referenzimplementierung analysiert, die konstanten bzw. variablen Anteile faktorisiert und im Anschluss daran wird eine domänenspezifische Sprache (*Domain Specific Language, DSL*) erstellt. Die konstanten Anteile werden in der Phase 5 (Umsetzung der Transformation) in die Templates kopiert, die variablen Anteile werden durch die Java-Implementierung des Metamodells zu Objektattributen parametrisiert und zur Transformation verwendet. Als Transformationssprache dient die Template-Sprache Xpand (weitere Details dazu unter [Ecl]). Unterstützend wird in der Regel ein ANT-Skript zur automatischen Generierung erstellt.

Danach werden die relevanten Testfälle in dieser DSL formuliert (Testfalldesign) und es werden daraus die im jeweiligen Testwerkzeug (hier JMeter) ausführbaren Testskripte generiert. Einen exemplarischer Auszug aus einem derartigen Testfalldesign zeigt Listing 1. Nach erfolgter Generierung müssen zur Validierung der generativen Architektur die erzeugten Testskripte einmalig auf Fehler geprüft werden.

Das modellgetriebene Vorgehen (siehe Abb. 4) bietet nach der einmaligen Umsetzung den Vorteil, dass bei Änderungen an der Benutzungsoberfläche der zu testenden Anwendung meistens „nur" die Transformationen zentral geändert werden müssen. Dieser Aspekt verringert den Wartungs- und Änderungsaufwand erheblich.

In Abbildung 5 sind die Änderungsaufwände eines konventionellen und eines modellgetriebenen Vorgehens einander qualitativ gegenüber gestellt.

```
<!DOCTYPE Test SYSTEM "Transformation-TestfallDesignsprache-Model-Code.dtd">
<Test testName="ExampleTest" numberOfThreads="1" numberOfRepetitions="2" rampUpTime="5">
  <ResultCollector type="ViewResults"/>
  <TestAction name="Open application" actionKey="startApplication">
    <Assertion assertionTyp="text" assertionPattern="Username" condition="contains"/>
    <Assertion assertionTyp="text" assertionPattern="Password" condition="contains"/>
  </TestAction>
  ...
</Test>
```

Listing 1: Exemplarischer Auszug aus einem Testfalldesign in einer DSL

Kaskadierungen modellgetriebener Testskript-Entwicklung

Nachdem im ersten Teil des Artikels die Grundlagen modellgetriebener Testautomatisierung und eine „einfache" generative Architektur für ein einzelnes Projekt vorgestellt wurden, beleuchtet der zweite Teil nun Verallgemeinerungen, die sinnvoll sind, wenn eine modellgetriebene Testautomatisierung über mehrere unterschiedliche Projekte hinweg zum Einsatz kommen soll. Zentrales Instrument hierbei ist eine Kaskadierung generativer Architekturen, die eine Modularisierung der Heterogenität und damit eine Erhöhung von Wiederverwendung ermöglicht.

Notwendigkeit und Einflussfaktoren

Die oben dargestellte, auf ein einzelnes Projekt zugeschnittene generative Architektur ist an spezifische Rahmenbedingungen gebunden. Die verwendete Designsprache ist in einem zweiten Projekt ohne Anpassungen nur unter folgenden Voraussetzungen einsetzbar:

1. Es werden ausschließlich Tests grafischer Benutzungsoberflächen betrachtet.
2. Die zu testende Anwendung verwendet die Basistechnologie HTML.

Die erstellten Transformationen sind zusätzlich an folgende Rahmenbedingungen gebunden:

3. Die betrachtete Anwendung ist in Bezug auf relevante technische Eigenschaften (z. B. *Session-Handling*) gleich gestaltet.
4. Es wird das Testwerkzeug JMeter verwendet.
5. Die betrachtete Anwendung hat die gleichen fachlichen Abläufe bzw. die gleiche Funktionalität.

Im allgemeinen Fall sind aber auch andere Tests, wie beispielsweise funktionsorientierte Tests von Web-Services, andere Basistech-

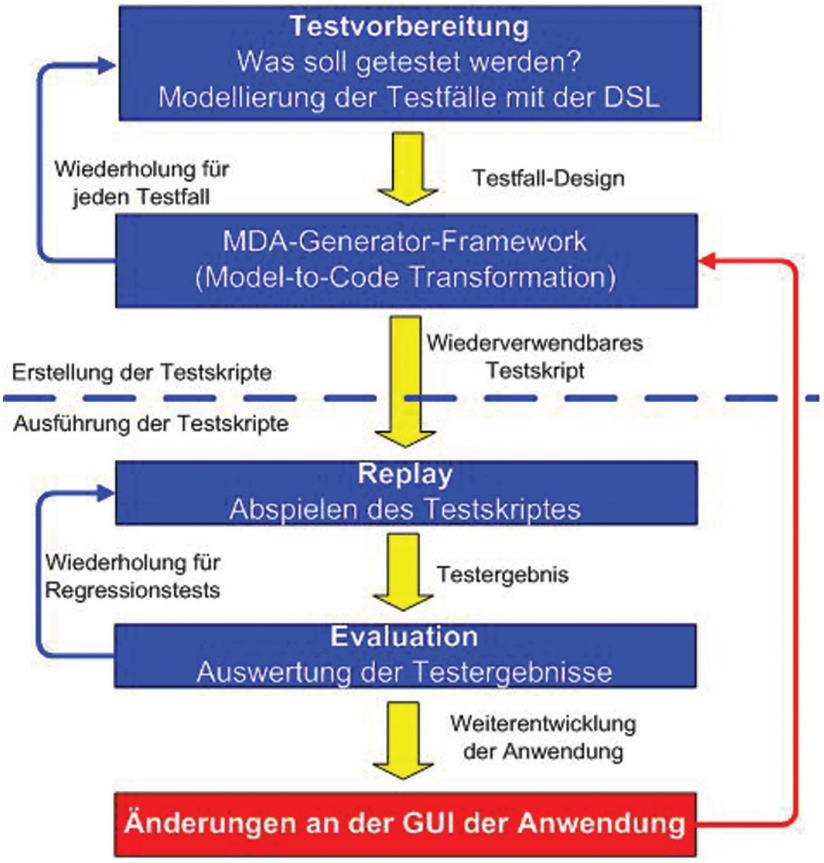

Abb. 4: Modellgetriebenes Vorgehensmodell nach konzipierter generativer Architektur

nologien wie eine *Rich Client Platform (RCP)*, andere Anwendungen mit anderen technischen Besonderheiten oder anderer Fachlichkeit und andere Testwerkzeuge, relevant.

Sobald sich an den Rahmenbedingungen in mindestens einem der fünf vorgenannten Bereiche Veränderungen ergeben, müssen bei der im ersten Teil des Artikels vorgestellten „einfachen" Vorgehensweise Transformationen und gegebenenfalls die Designsprache angepasst werden. Wird innerhalb eines Unternehmens oder über mehrere Kunden hinweg ein Einsatz über mehrere Projekte hinweg angestrebt, so sind daher je nach Aufgabenstellung Abstraktionen in Form kaskadierter generativer Architekturen sinnvoll. Derartige Kaskadierungen bauen auf einer ersten generativen Architektur eine zweite, abstraktere generative Architektur auf. So kann z. B. oberhalb einer architekturzentrierten generativen Architektur eine domänenzentrierte generative Architektur erstellt werden, die fachliche Beschreibungen konsumiert und Ergebnisse in der Designsprache der architekturzentrierten generativen Architektur liefert. Auf diese Weise werden in den jeweiligen Ebenen unterschiedliche Aspekte separiert und wiederverwendbar gemacht (vgl. [Sta05], S. 295). Im Folgenden wird eine Reihe von möglichen Abstraktionen für den Testbereich vorgestellt. Wird aus Vereinfachungsgründen bei einer konkreten Problemstellung auf eine oder mehrere dieser Ebenen verzichtet, so sind die Testspezifikationen auf der jeweils nachgelagerten Ebene in der Designsprache dieser Ebene manuell zu erstellen, statt sie zu generieren.

Durch den Einsatz der hier vorgestellten Abstraktionen lassen sich wesentliche Ziele

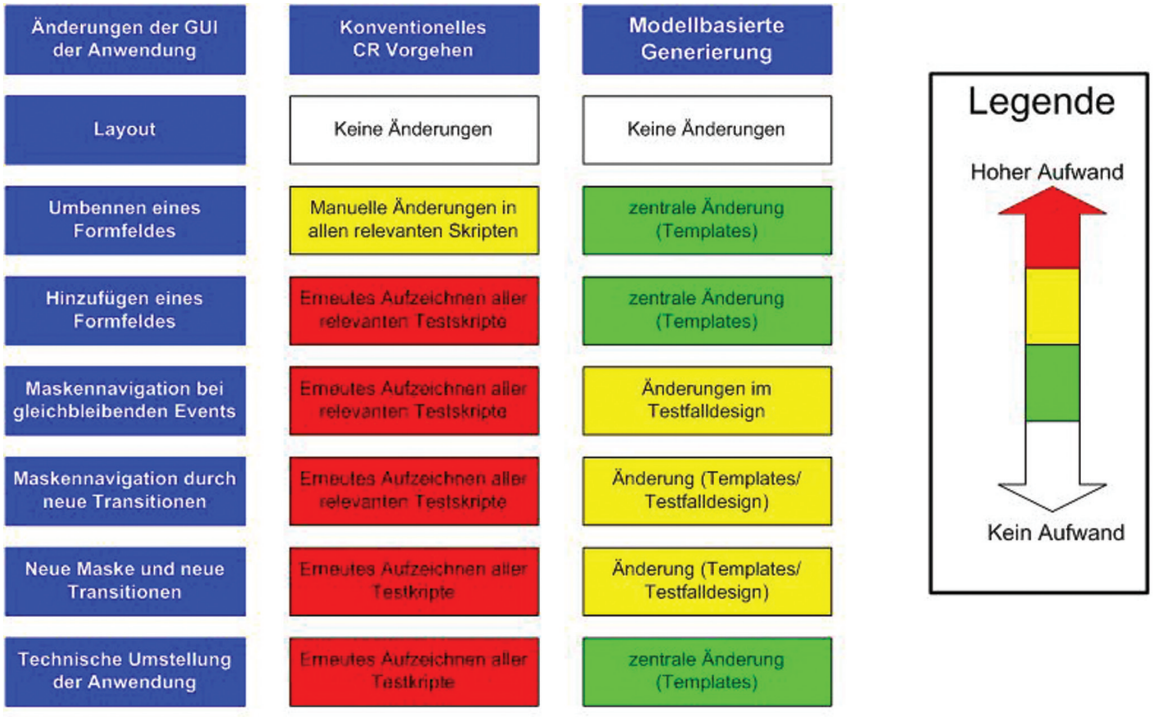

Abb. 5: Aufwände konventioneller und modellgetriebener Testskript-Entwicklung am Beispiel von JMeter

eines Produktlinien-Engineerings (Automatisierung und Wiederverwendung) für den Bereich der modellgetriebenen Testskript-Entwicklung erreichen. Eine umfassende Interpretation einer derartigen Vorgehensweise aus Sicht eines Produktlinien-Engineerings ist allerdings nicht Gegenstand dieses Artikels.

Abstraktionen und Kaskadierungen

Abbildung 6 gibt einen Überblick über potenziell sinnvolle Abstraktionen und Kaskadierungen. Wird ein kaskadierter domänenzentrierter Ansatz verfolgt (fachliche Domäne über der Domäne „Architektur", vgl. [Sta05], S. 295ff.), so werden Analysemodelle erstellt und im Wege einer Modell-zu-Modell-Transformation in Designmodelle überführt.

Für die Zwecke einer architekturzentrierten modellgetriebenen Entwicklung von automatisierten Tests (AC-MDTD) müssen dann zusätzlich fachliche Testfallspezifikationen erstellt werden. Diese enthalten:

- fachliche Modelle, die den konkreten Durchlauf durch den Testling (ablaufrelevante Daten und Ereignisse) sowie die erwarteten eintretenden Abläufe (z.B. angesteuerte Masken) beschreiben, soweit die Testart dynamische Aspekte hat, und
- fachliche Beschreibungen der Testdaten und der zu erwartenden Ergebnisse.

Bis hierhin sind die Beschreibungen frei von jeder Bindung an eine konkrete Testart, Basistechnologie des Testlings, Testwerkzeug oder Technik des Testlings.

In einer zweiten Modell-zu-Modell-Transformation können auf Basis der Designmodelle sowie der ablaufrelevanten Testfallspezifikationen technische Testfallspezifikationen erzeugt werden. Diese sind nun gebunden an eine spezifische Testart und Basistechnologie des Testlings. Sie sind aber noch frei von jeder Bindung an ein spezifisches Testwerkzeug und die Technik eines Testlings und sind dementsprechend in einer testwerkzeug- und technikunabhängigen DSL formuliert.

In einer dritten Modell-zu-Code-Transformation können auf Basis der so erzeugten oder gegebenenfalls auch manuell erstellten technischen Testfallspezifikation Testskripte erzeugt werden. Diese sind zusätzlich gebunden an die Sprache eines spezifischen Testwerkzeugs und die techni-

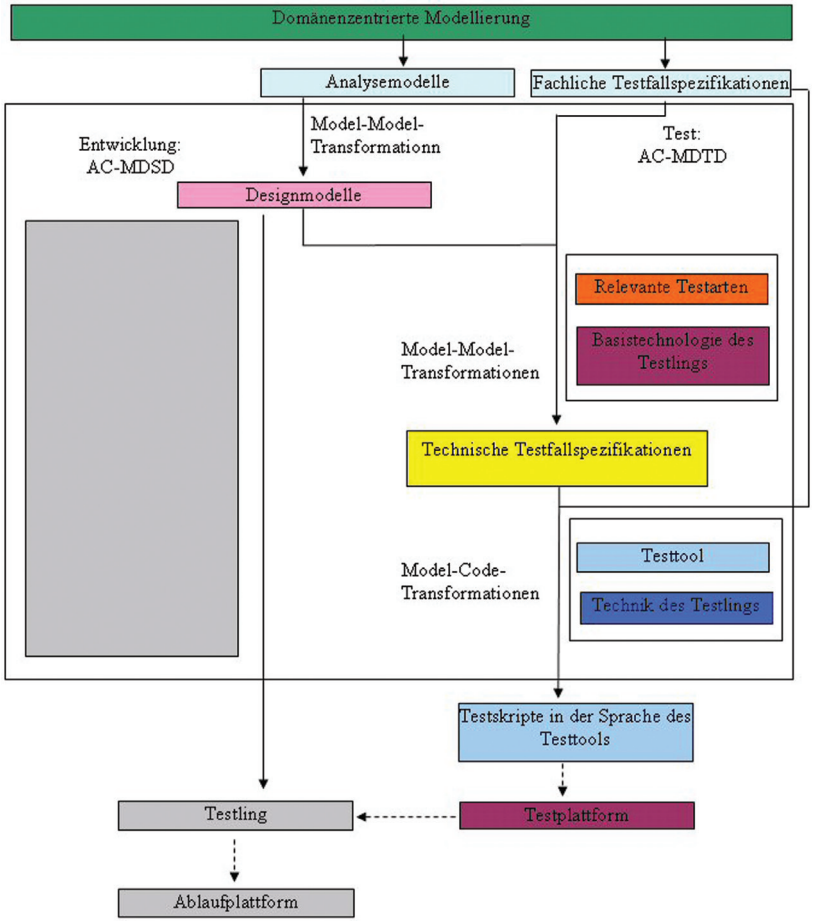

Abb. 6: Überblick über Abstraktionen und Kaskadierungen

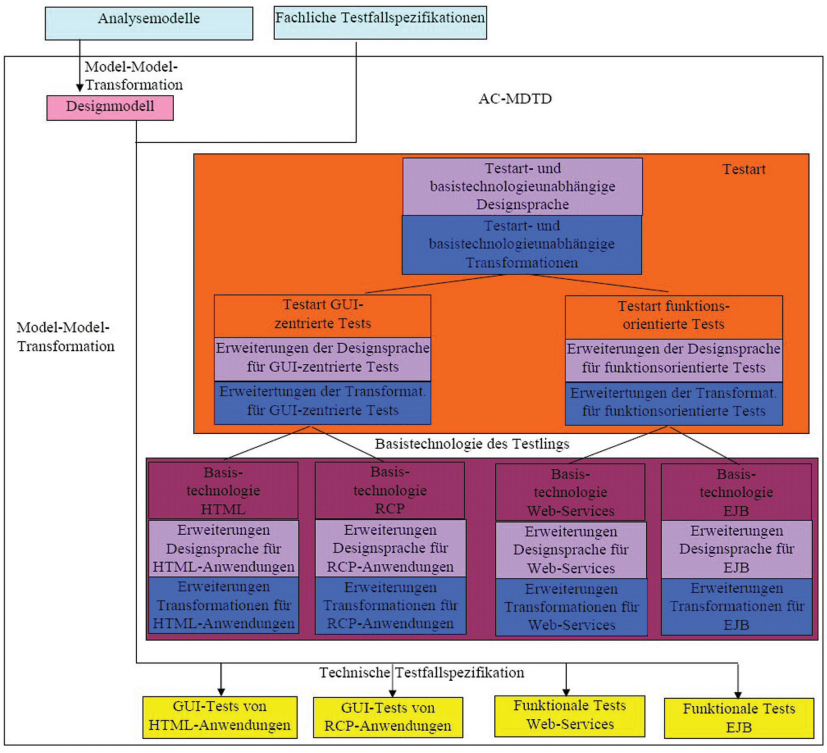

Abb. 7: Abstraktion von den relevanten Testarten und der Basistechnologie des Testlings

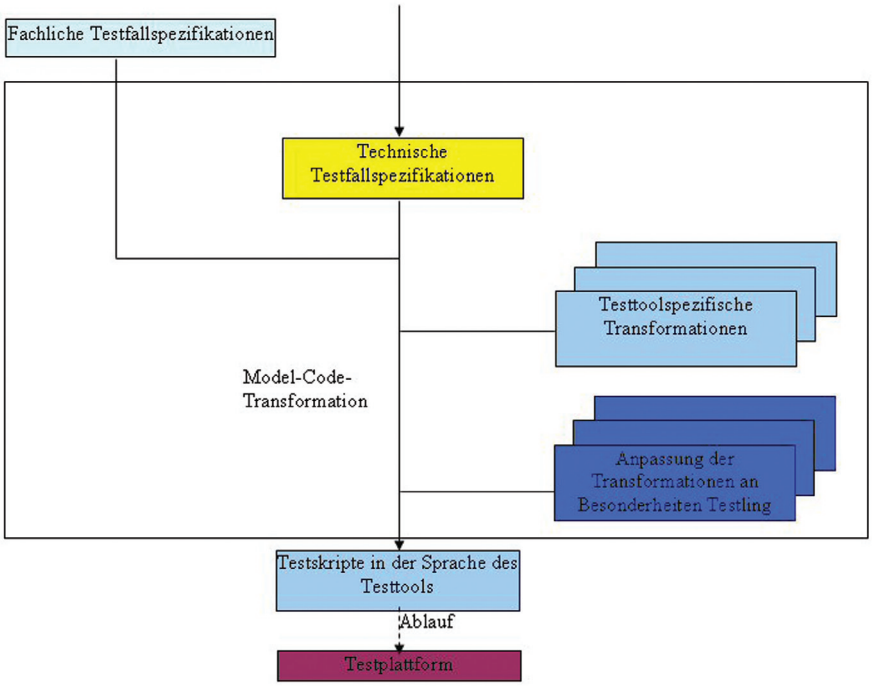

Abb. 8: Transformation der technischen Testfallspezifikationen in Testskripte

schen Besonderheiten der Nutzung der zu Grunde liegenden Basistechnologie durch den jeweiligen Testling.

Im weiteren Verlauf werden die letztgenannten beiden Transformationen etwas näher beleuchtet, an einem exemplarischen Anwendungsbeispiel in einigen wesentlichen Zügen erläutert sowie Einflussfaktoren auf die Entscheidung aufgeführt, welche Abstraktionen und Kaskadierungen bei einer konkreten Problemstellung sinnvoll sein könnten.

Abstraktion von Testarten und Basistechnologie

Die Designsprachen für die technischen Testfallspezifikationen und die damit verbundenen Transformationen unterscheiden sich (teilweise) je nach Testart und Basistechnologie des Testlings. So sind z. B. bei Tests von grafischen Benutzungsoberflächen neben erwarteten Ergebnissen auch Abläufe zu testen, bei funktionsorientierten Tests hingegen nur Ergebnisse. Die Basistechnologie HTML kennt nur seitenweise Interaktionen, RCP hingegen auch feldweise Interaktionen. Eine exemplarische, für weitere Testarten und Technologien gegebenenfalls entsprechend zu erweiternde Taxonomie für die beiden Ebenen Testart und Basistechnologie des Testlings zeigt Abbildung 7.

Abstraktion von Testwerkzeugen und technischen Besonderheiten

Innerhalb einer Gattung von Testarten und Basistechnologien ist zwar eine einheitliche Designsprache definierbar, es müssen dann aber in einem weiteren Modell-zu-Code-Transformationsschritt Testskripte erzeugt werden, die in der Sprache des jeweiligen Testwerkzeugs formuliert sind, die mit den Testdaten und den erwarteten Ergebnissen parametrisiert sind und die die technischen Besonderheiten des jeweiligen Testlings berücksichtigen (siehe Abb. 8). Hierzu ist je Testwerkzeug ein Satz von Transformationen notwendig, der je Gattung von technischen Testfallspezifikationen die dort zu Grunde gelegte DSL auf die konkrete Skriptsprache abbildet.

Die zu Anpassungen an diesen Transformationen führenden technischen Besonderheiten des jeweiligen Testlings sind bei web-basierten Anwendungen beispielsweise:

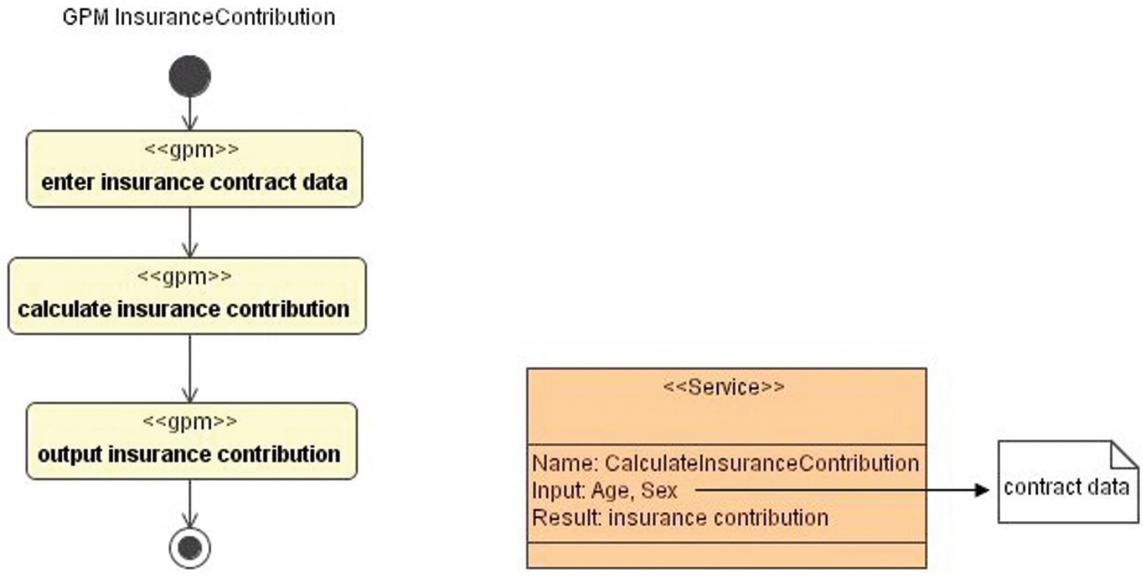

Abb. 9: Analysemodelle

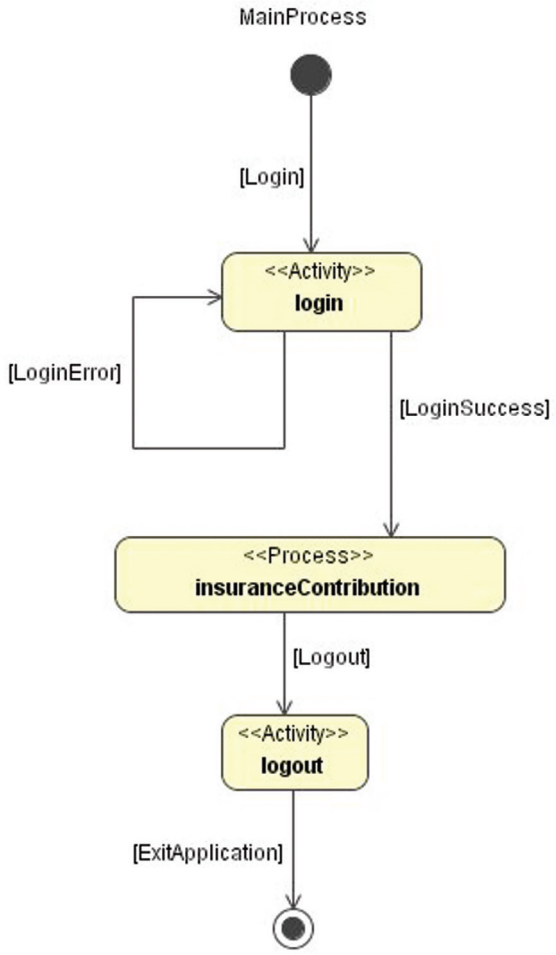

Abb. 10a: Designmodell (Hauptprozess)

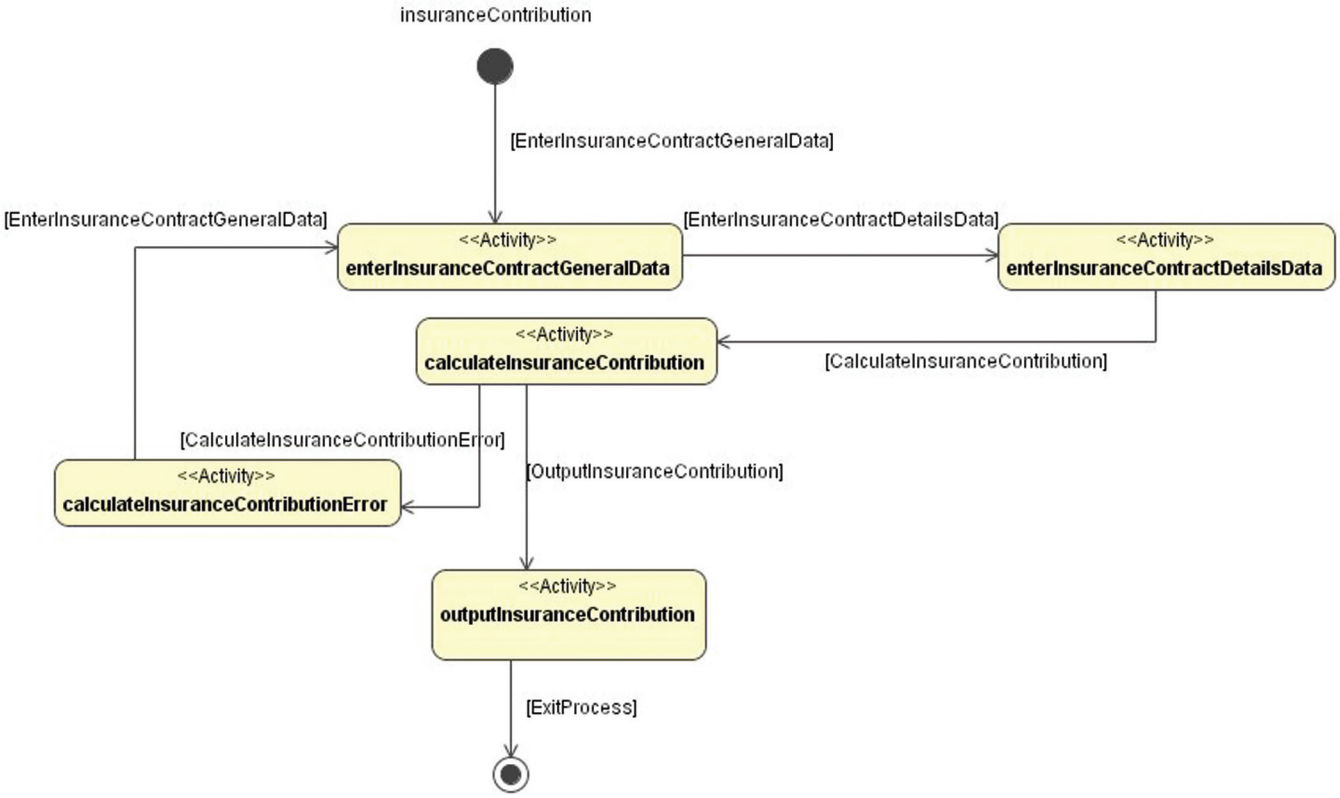

Abb. 10b: Designmodell (Subprozess)

- *Session-Handling*
- Login-Mechanismen
- Prozessschrittsteuerung
- Spezielle Elemente der grafischen Benutzungsoberfläche (z. B. Drop-Down-Menü oder Checkboxen)

Anwendungsbeispiel

In einer Anwendung sind auf der Ebene Geschäftsprozessmodellierung (GPM) die in Abbildung 9 dargestellten Analysemodelle für einen einfachen Anwendungsablauf und den Aufruf eines Services formuliert.

Der Prozessschritt calculate insurance contribution ruft den Service CalculateInsuranceContribution auf und gibt das Ergebnis einer Beitragsberechnung im Schritt output insurance contribution aus.

Es wird ein kaskadierter domänenzentrierter Ansatz genutzt, bei dem aus den oben aufgeführten Analysemodellen Designmodelle generiert werden (siehe Abb. 10a und 10b).

Die Anwendung soll für gelegentliche Anwender auf Basis von HTML und für häufige Anwender auf Basis von RCP realisiert werden. Es sind automatisierte Benutzungsoberflächentests der beiden resultierenden Anwendungen und der genutzten Services gewünscht.

Die Beschreibung der fachlichen Testfallspezifikationen für den Durchlauf durch

```
Event; Expected Navigation Result; Test-Data
EnterInsuranceContractGeneralData; Check-for-enterInsuranceContractGeneralData; generalData
EnterInsuranceContractDetailsData; Check-for-enterInsuranceContractDetailData; contributionData
CalculateInsuranceContribution; Check-for-calculateInsuranceContribution;
OutputInsuranceContribution; Check-for-outputInsuranceContribution; contributionData
```

Listing 2: *Testfallspezifikation für einen Durchlauf durch den Subprozess „insuranceContribution"*

```
Assertion-Name; Assertion-Pattern; Condition
Check-for-enterInsuranceContractGeneralData;"Geben Sie die Vertragsgrunddaten ein"; "contains"
Check-for-enterInsuranceContractGeneralData;"Fehler"; "contains not"
Check-for-enterInsuranceContractDetailsData;"Geben Sie die beitragsrelevanten Detaildaten ein"; "contains"
```

Listing 3: *Spezifikation von Assertions für die Prüfung der Maskennavigation bei Listing 2*

```
age; sex; contribution
23; m; 134,78
67; f; 543,89
```

Listing 4: *Testdaten für die Aktivitäten „enterInsuranceContractDetailsData" und "outputInsuranceContribution" in der Datei „contributionData.csv"*

```
<!DOCTYPE Test SYSTEM "Transformation-TestfallDesignsprache-Model-Code.dtd">
<Test testName="insuranceContribution" numberOfThreads="1" numberOfRepetitions="2" rampUpTime="5">
  <ResultCollector type="ViewResults"/>
  <ResultCollector type="Statistics"/>
...
  <TestAction name="enterInsuranceContractGeneralData" actionKey="EnterInsuranceContractGeneralData"
              testData="generalData.csv">
    <Assertion assertionName="Check-for-enterInsuranceContractGeneralData" assertionTyp="text" assertionPattern="Geben
              Sie die Vertragsgrunddaten ein" condition="contains"/>
    <Assertion assertionName="Check-for-enterInsuranceContractGeneralData" assertionTyp="text" assertionPattern="Fehler"
              condition="contains not"/>
  </TestAction>
...
</Test>
```

Listing 5: *Fragment einer technischen Testfallspezifikation für eine HTML-basierte Anwendung*

```
<!DOCTYPE Test SYSTEM "Transformation-TestfallDesignsprache-Model-Code.dtd">
<Test testName="insuranceContribution" numberOfThreads="1" numberOfRepetitions="2" rampUpTime="5">
  <ResultCollector type="ViewResults"/>
  <ResultCollector type="Statistics"/>
...
  <TestAction name="enterInsuranceContractGeneralData" actionKey="EnterInsuranceContractGeneralData " testData="
              generalData.csv">
    <Assertion assertionName="Check-for-enterInsuranceContractGeneralData" assertionTyp="text" assertionPattern="Geben
              Sie die Vertragsgrunddaten ein" condition="contains"/>

    <Assertion assertionName="Check-for-enterInsuranceContractGeneralData" assertionTyp="text" assertionPattern="Fehler"
              condition="contains not"/>
    <SubTestAction name="fillName" actionKey="FillName">
      <Assertion assertionTyp="text" assertionPattern="Fehler" condition="contains not"/>
    </SubTestAction>
    <SubTestAction name="fillSurname" actionKey="FillSurname">
      <Assertion assertionTyp="text" assertionPattern="Fehler" condition="contains not"/>
    </SubTestAction>
  </TestAction>
...
</Test>
```

Listing 6: *Fragment einer technischen Testfallspezifikation für eine RCP-basierte Anwendung*

den Testling (Abfolge der fachlichen Ereignisse, die für den jeweiligen Testfall den tatsächlichen Durchlauf durch die im Design spezifizierten Aktivitätsdiagramme festlegen), der Prüfung der korrekten Navigation durch die Anwendung sowie der Testdaten und der erwarteten Ergebnisse gestaltet sich für den Subprozess insuranceContribution in einem konkreten Testfall wie in den **Listings 2, 3 und 4** gezeigt.

Die in **Listing 4** aufgeführten Testdaten lassen sich gleichzeitig für den funktionalen Test des Services CalculateInsurance Contribution verwenden. Dieses wird hier nicht weiter beleuchtet.

Aus den fachlichen Testfallspezifikationen der **Listings 2 bis 4** lassen sich im Zusammenspiel mit den Designmodellen für benutzungsoberflächen-zentrierte Tests je nach Basistechnologie die nachfolgend aufgeführten technischen Testfallspezifikationen generieren. **Listing 5** zeigt hierbei Fragmente aus einer technischen Testfallspezifikation für eine HTML-basierte Anwendung mit seitenweiser Interaktion und **Listing 6** zeigt Fragmente für eine RCP-basierte Anwendung mit feldweiser Interaktion.

TestAction stellt hierbei einen Testschritt, SubTestAction eine feldweise Interaktion mit dem Anwender, actionKey das fachliche Ereignis und Assertion die formulierte Ergebnisprüfung dar. TestActions, SubTestActions und actionKeys werden aus den Designmodellen im Zusammenspiel mit den fachlichen Testfallspezifikationen abgeleitet, die Assertions ausschließlich aus den fachlichen Testfallspezifikationen.

In einer weiteren Modell-zu-Code-Transformation lassen sich aus derartigen technischen Testfallspezifikationen Testskripte für ein für HTML geeignetes Testwerkzeug wie z. B. JMeter generieren (**siehe Abb. 11**). Da JMeter keine Prozeduren unterstützt, wird dabei die Verschachtelung von Haupt- und Subprozessen expandiert.

Eine RCP-basierte Anwendung kennt feldweise Interaktionen. Hierzu sind die Designmodelle zu verfeinern (hier nicht dargestellt) und die DSL muss durch ein zusätzliches Konstrukt SubTestActions erweitert werden (**Listing 6**).

Für RCP-basierte Anwendungen lassen sich in ähnlicher Weise wie für JMeter gezeigt Testskripte für ein geeignetes Testwerkzeugs (z. B. „Abbot") generieren. Im vorgestellten Beispiel sind die Design-

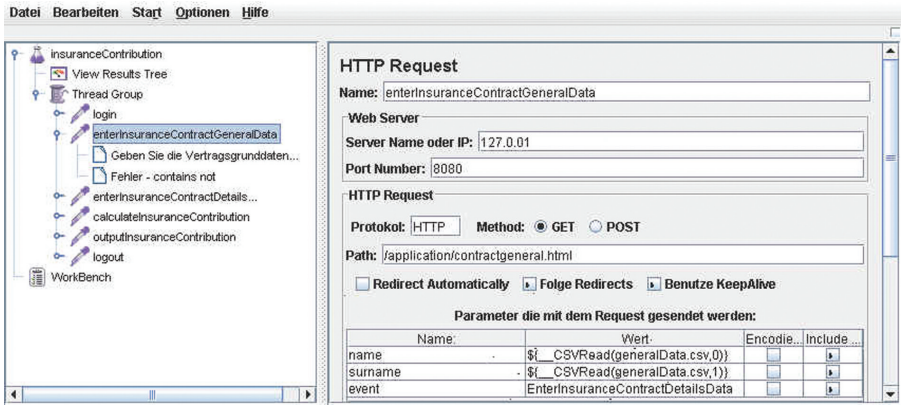

Abb. 11: JMeter-Screenshot für den Testfall

sprachen unter anderem bezüglich der Konstrukte TestAction und Assertion identisch. Diese sind daher transformationsseitig nur einmal umzusetzen. Für eine Validierung der technischen Testfallspezifikationen lässt sich eine entsprechende Dokumenttypdefinition (DTD) generieren.

Sinnvolles Maß von Abstraktionen und Kaskadierungen

Jede Kaskadierung erhöht die Komplexität sowie den Konzeptions- und Umsetzungsaufwand der generativen Architektur und sollte daher nur dann angestrebt werden, wenn Größe und Zahl oder der Qualitätsanspruch der Projekte, für die die modellgetriebene Testskript-Entwicklung erfolgt, dieses angemessen erscheinen lassen. Die Anforderungen sind umso höher, je umfangreicher gemessen an Zahl der Anwendungen und Basistechnologien die Testlinge sind und je höher das angestrebte Maß an Wiederverwendung fachlicher Modelle und technischer Artefakte (Designsprachen und Transformationen) sind. Die folgenden Fragen geben Hinweise darauf, welches Maß im jeweiligen Fall tendenziell sinnvoll sein könnte:

- Ist eine anwendungsübergreifende Verwendbarkeit wesentlicher Inhalte der Transformationen gefordert?
- Ist eine testwerkzeugübergreifende Verwendbarkeit wesentlicher Inhalte der Transformationen gefordert?
- Ist eine Verwendung wesentlicher Inhalte der Transformationen und der Designsprache über mehrere Basistechnologien hinweg gefordert?
- Ist eine Verwendung wesentlicher Inhalte der Transformationen und der Designsprache über mehrere Testarten hinweg gefordert?
- Ist eine übergreifende Verwendung einer domänenzentrierten Modellierung über Anwendungsentwicklung und Testentwicklung hinweg gefordert?
- Ist eine Verwendung fachlicher Testfallspezifikationen über Teststufen, Basistechnologien und Anwendungen hinweg gefordert?

AC-MDSD	Als **Architecture Centric Model Driven Software Development** wird eine Softwareentwicklung bezeichnet, wenn sie implementierungsrelevante Modelle und Architekturen betont und in einer gewissen Weise einsetzt (vgl. [Sta05], S. 11ff.).
AC-MDTD	**Architecture Centric Model Driven Test Development** bezeichnet in Anlehnung an AC-MDSD eine modellgetriebene Entwicklung von automatisierten Tests.
DSL	Eine **Domain Specific Language** beschreibt Schlüsselaspekte einer Domäne formal. Sie wird beschrieben durch ein Metamodell einschließlich statischer Semantik, eine korrespondierende konkrete Syntax und eine (dynamische) Semantik und kann gegebenenfalls durch Editoren unterstützt werden. Domänen können technisch (z. B. Domäne Architektur) oder fachlich sein (vgl. [Sta05], S. 68f.).
Generative Architektur	Unter einer generativen Architektur wird die Kombination der Beschreibungen von DSL, Transformationen und Zielplattform verstanden.

Glossar

Zusammenfassung und Ausblick

Die modellgetriebene Testskript-Entwicklung verbessert die praktische Umsetzung automatisierter Tests hinsichtlich Effizienz und Qualität deutlich, wenn das Umfeld stimmt. Das trifft unabhängig davon zu, ob die Anwendung selbst modellgetrieben entwickelt wird oder nicht. Allerdings sind für die Effizienz modellgetriebener Testskript-Entwicklung die gleichen Effekte zu beobachten, die auch für die Wirtschaftlichkeitsbeurteilung einer Einführung modellgetriebener Softwareentwicklung gelten (vgl. [Sta05], S. 315ff.).

Die modellgetriebene Testskript-Entwicklung lässt sich mittels kaskadierter generativer Architekturen flexibel und mit hohem Wiederverwendungsgrad an die im jeweiligen Fall gültigen Anforderungen an eine testart-, basistechnologie-, testwerkzeug- und anwendungsübergreifende Verwendung anpassen. Dies betrifft sowohl die generativen Architekturen selbst als auch fachliche Testfallspezifikationen sowie ursprünglich für die Anwendungsentwicklung erstellte Domänenmodelle mit testrelevanten Inhalten.

Ein praktikabler Umfang und sinnvolle Ausgestaltungen der vorgestellten Kaskadierungen sowie die Anwendung auf unterschiedliche Einsatzkontexte werden aktuell in laufenden Projekten praktisch erprobt. ■

Literatur & Links

[Apa] Apache Software Foundation, Apache JMeter, siehe: jakarta.apache.org/jmeter/index.html

[b+m03] b+m Informatik AG, b+m Generative Development Process, 2003, siehe: www.architectureware.de/download/b+m_Generative_Development_Process.pdf

[b+m06] b+m Informatik AG, Das Generator FrameWork, 2006, siehe: www.architectureware.de/produkte/generator_framework.htm

[Ecl] Eclipse Foundation, openArchitectureWare, siehe: www.eclipse.org/gmt/oaw

[Gam01] E. Gamma, K. Beck, JUnit, siehe: www.junit.org, 2001

[Gol03] R. Gold, HttpUnit, 2003, siehe: httpunit.sourceforge.net

[Sen05] C. Sensler, Diplomarbeit „Generierung von Testskripten für automatische Regressionstests mit Werkzeugen und Methoden der generativen Softwareentwicklung", 2005

[Sta05] T. Stahl, M. Völter: Modellgetriebene Softwareentwicklung, dpunkt.verlag, 2005