Institut für Informatik
Lehrstuhl für Programmierung und Softwaretechnik
LUDWIG–MAXIMILIANS–UNIVERSITÄT MÜNCHEN

Diplomarbeit

Model-Driven Testing

Model-Driven Testing im Kontext von UWE

Oliver Sommer

Informatik Diplom

Aufgabensteller: Prof. Dr. Martin Wirsing
Betreuer: Christian Kroiss
Abgabetermin: 3. Januar 2014

Ich versichere hiermit eidesstattlich, dass ich die vorliegende Arbeit selbstständig angefertigt, alle Zitate als solche kenntlich gemacht sowie alle benutzten Quellen und Hilfsmittel angegeben habe.

München, den 3. Januar 2014

. . . . . .

(Unterschrift des Kandidaten)

Zusammenfassung

Mit der wachsenden Komplexität von Softwaresystemen steigt die Notwendigkeit der frühen Integration von Softwaretests in den Entwicklungsprozess. Sie erlauben das Aufdecken von Fehlern in Systementwürfen und deren Implementierungen bereits in frühen Phasen eines Projektes. So können sie signifikant zur Reduzierung von Entwicklungszeit und daraus resultierenden Kosten beitragen.

Die vorliegende Arbeit stellt einen modellgetriebenen Ansatz zum Blackbox-Testen von Webanwendungen unter Verwendung plattformunabhängiger Modellierung vor. Ausgehend von der Modellarchitektur des UML-Based Web Engineerings (UWE) wird ein ergänzendes Testprofil vorgestellt, welches die Spezifikation von Testmodellen unter Nutzung des UML2.0 Testing Profile (U2TP) erlaubt. Zur Ergänzung der sich daraus ergebenden Möglichkeiten wird die domänenspezifische Sprache UWE Test Specification Language (UTSL) vorgestellt, welche aufbauend auf den von UWE und U2TP zur Verfügung gestellten Konzepten die Spezifikation von Testfällen unter Verwendung des Systemmodells einer UWE Anwendung erlaubt. Zur Modellierung von Testverhalten werden hierbei UML Sequenzdiagramme eingesetzt. Für die Erzeugung eines Testsystems wird eine Modelltransformation in eine Instanz des um Testaspekte erweiterten UWE Metamodells vorgenommen und ein Ansatz vorgestellt, der die Generierung von Testfällen unter Verwendung von JUnit und Selenium ermöglicht. Anhand eines Fallbeispiels wird die Anwendung der vorgestellten Methode demonstriert.

Die Arbeit kommt zu dem Ergebnis, dass die Methode durchführbar ist und die Herangehensweise der Testsystemerstellung für Webanwendungen auf der Grundlage plattformunabhängiger Modellierung interessante Potenziale birgt.

Danksagung

Mein Dank gilt meinem Betreuer Herrn Christian Kroiß, der mit viel Engagement, guten Ideen und unermüdlichem Einsatz diese Diplomarbeit betreut hat.

Dank auch an alle Mitarbeiter des Lehrstuhls Programmierung und Softwaretechnik auch in vergangenen Jahren für die Unterstützung bei Studientätigkeiten und eigenen Projekten.

Mein ganz besonderer Dank jedoch gilt Herrn Prof. Dr. Martin Wirsing, nicht nur für das Bereitstellen dieses interessanten Themas der Diplomarbeit, sondern auch und ganz besonders für die stets freundliche Hilfsbereitschaft und tatkräftige Unterstützung, die er mir nicht nur im Rahmen des Studiums sondern auch bei der Umsetzung eigener Projekte entgegenbrachte.

Inhaltsverzeichnis

Kapitel 1

Einleitung

Der Einsatz von Modellen beim Entwurf komplexer Systeme wird in traditionellen Ingenieurswissenschaften längst als unerlässlich erachtet. Kaum jemand kann sich heute die Entwicklung komplexer Systeme wie beispielsweise von Gebäuden, Brücken oder Automobilen vorstellen, ohne dass zu diesem Zweck eine Reihe von speziellen Systemmodellen angefertigt wird. Modelle unterstützen das Verständnis komplexer Probleme und deren möglicher Lösungen durch das ihnen innewohnende Prinzip der Abstraktion. So ist es naheliegend, dass auch die Herstellung von Softwaresystemen, welche mit zu den komplexesten, von Menschenhand geschaffenen Systemen zählen können, stark von der Nutzung von Modellen und Modellierungstechniken profitiert. Neben den gegebenen Vorteilen der Abstraktion können Modelle in der Softwareentwicklung zusätzlich zum Zweck der Automatisierung mancher Aufgaben eingesetzt werden. Dennoch hat die Nutzung von Modellen in der industriellen Softwareentwicklung lange Zeit keine nennenswerte Verbreitung gefunden und eher eine untergeordnete Rolle gespielt. Seit einiger Zeit jedoch lässt sich ein stetig steigendes Interesse an modellbasierten Techniken verzeichnen, sowohl bei der Erstellung von Systemmodellen als auch bei Verifikationstechniken, welche die Qualitiät von Softwareprodukten bewerten sollen.

Beispielsweise im Bereich des Testens von Software erfreuen sich modellbasierte Ansätze seit einigen Jahren zunehmender Beliebtheit. Als Ergänzung oder manchmal auch als Alternative zur manuellen Erstellung von Tests ist das Erzeugen von Testfällen durch Generierung aus Modellen eine interessante Vorgehensweise, die inbesondere für modellbasierte Entwicklungsprozesse eine frühe Integration von Tests erlaubt und Potenziale zur Kostenersparnis vermuten lässt. Dabei ist die Idee hinter modellbasiertem Testing alles andere als neu. Erste Grundlagen wurden hier unter Verwendung von endlichen Automaten bereits in den 1960er Jahren erarbeitet, eine weite Verbreitung dieser Techniken blieb jedoch zunächst aus. Ein bedeutender Grund hierfür ist sicherlich in der Tatsache zu suchen, dass standardisierte Modellierungstechniken und die zusätzlich benötigten Austauschverfahren und Werkzeuge lange Zeit nur spärlich vorhanden waren oder keine wirkliche Verbreitung in der industriellen Softwareentwicklung fanden. Ohne sie können modellbasierte Testtechniken sehr mühsam und kostenintensiv ausfallen.

Seit Erscheinen der Unified Modeling Language (UML) hat sich in diesem Grundlagenbereich jedoch einiges geändert und Modellierungstechniken werden verstärkt in allen möglichen Bereichen auch der industriellen Softwareentwicklung genutzt. Dazu hat nicht zuletzt die Verbreitung und der hohe Akzeptanzgrad der UML beigetragen. Die nun verfügbaren, standardisierten Modellierungstechniken wurden zunächst hauptsächlich in der Systementwicklung eingesetzt. Die Erweiterungsmechanismen der allgemein

gehaltenen UML erlauben es dabei, die angebotenen Mittel zur Modellierung auf domänenspezifische Bereiche auszudehnen. Durch die rasante Entwicklung im Bereich des World Wide Web ist dabei auch der Ansatz des UML-Based Web Engineering (UWE) entstanden, welcher eine modellgetriebene Methode zur systematischen Entwicklung von Websystemen bietet. Durch eine Erweiterung der UML um Konzepte, die bei der Entwicklung von Webanwendungen benötigt werden, ist mit UWE ein Entwicklungsprozess entstanden, der durch Trennung einzelner Aspekte wie Inhalte, Navigation, Prozesse und Präsentation die Modellierung von webbbasierten Systemen in plattformunabhängiger Weise ermöglicht. Anders als in der Frühzeit der Softwareentwicklung sind die mit UWE erstellten Modelle für Webentwickler auch ohne tiefgehendes Wissen über den UWE Entwicklungsprozess schnell verständlich, vorausgesetzt, entsprechende Kenntnisse der UML sind vorhanden. Durch den Siegeszug der UML in alle erdenklichen Fachbereiche der Softwareentwicklung kann dies jedoch gegenwärtig für die Mehrheit der Softwareingenieure vorausgesetzt werden.

Zusätzlich zur Akzeptanz der UML bei Softwareentwicklern steht zum heutigen Tag eine Fülle von unterstützenden Werkzeugen für die Nutzung der UML zur Verfügung. Die Existenz von Modellierungswerkzeugen, Austauschformaten, Sprachen und Werkzeugen zur Modelltransformation bis hin zu Werkzeugen, welche die Generierung von Programmtext aus Modellen erlauben, wird inzwischen fast als selbstverständlich erachtet und ermöglicht die Realisierung von komplexen, modellbasierten Entwicklungsprozessen auf der Grundlage von Standards. Die Veröffentlichung des modellgetriebenen Systementwicklungsprozesses Model Driven Architecture (MDA) und die enormen Verbesserungen in der UML Version 2.0 haben die Popularität modellbasierter Techniken zusätzlich gefördert und die Diskussion um Entwicklungsmethoden unter Einsatz von Modellen als Projektartefakte neu entfacht. Schon kurz nach Erscheinen der MDA wurde eine Ergänzung jener um Testmodelle diskutiert, die analog zur Vorgehensweise der MDA die systematische Entwicklung von Testmodellen erlaubt[HL03]. Der gewachsene Bedarf an standardisierten Techniken zur Testsystemmodellierung resultierte mit dem UML Testing Profile in der Verabschiedung eines ersten, UML-basierten Standards, welcher umfassende Konzepte zur Spezifikation von Testsystemen erarbeitete und seit 2013 mit der überarbeiteten Version 1.2 in Form des UML2.0 Testing Profile (U2TP) zur Verfügung steht.

Aufgrund der Sprachverwandschaft von UWE und U2TP über die UML entstand die Motivation der vorliegenden Arbeit, die Modellierungsansätze beider Technologien gemeinsam einzusetzen und zu untersuchen, welche Möglichkeiten des modellbasierten Testens von Webanwendungen sich dabei ergeben. Diese Diplomarbeit stellt einen Ansatz zur modellgetriebenen Testsystemerstellung auf der Basis von UWE und U2TP vor. Durch eine vorgestellte Erweiterung des UWE Profils ist es möglich, ein vorhandenes UWE Systemmodell um ein Testmodell zu ergänzen. Innerhalb des Testmodells können mit den Mitteln des U2TP Testumgebungen spezifiziert sowie die von U2TP angebotenen Konzepte zu Testdaten und Testmanagement genutzt werden. Um die von U2TP angebotenen Mittel der Testverhaltensspezifikation um den Bereich Web zu ergänzen, wird die domänenspezifische Sprache UWE Test Specification Language (UTSL) präsentiert, die das Erstellen von Testabläufen für Webanwendungen unter Heranziehung der Oberflächenspezifikation eines UWE Systemmodells ermöglicht. Die sich aus diesem Ansatz ergebenden Möglichkeiten der Testverhaltensmodellierung erlauben die Spezifikation von Testsystemen, die zur Erzeugung von ausführbaren Testfällen genutzt werden können. Ferner werden eine Erweiterung des UWE Metamodells um die verwendeten Testkonzepte vorgestellt und Vorgehensweisen bei der Modelltransfor-

mation und Codegenerierung für Testfälle diskutiert. Dabei zeigt sich, dass sich UWE zusammen mit U2TP und UTSL zur Testspezifikation von Websystemen einsetzen lässt und eine Generierung von Blackbox-Tests auf Basis der plattformunabhängigen UWE-Modellierung möglich ist.

Die Arbeit gliedert sich in insgesamt fünf Teile:

Der Erste Teil (**Kapitel 2**) stellt Hintergrundwissen zu den Themengebieten Softwaretests, Modellgetriebene Softwareentwicklung und UML-Based Web Engineering vor, wobei für den Aspekt des Testens Vertiefungen zu modellbasierten Tests im Allgemeinen sowie dem UML Testing Profile im Speziellen dargeboten werden. Der zweite Teil befasst sich mit grundlegenden Zusammenhängen und Problemstellungen beim Testen von Webanwendungen mit plattformunabhängigen Modellen (**Kapitel 3**). Im dritten Teil werden die domänenspezifische Sprache UWE Test Specification Language und deren Integration mit dem UML Testing Profile vorgestellt sowie Besonderheiten bei der Testverhaltensmodellierung und eine Vorgehensweise zur Testmodellerstellung aufgezeigt (**Kapitel 4**). Dar anschließende Teil (**Kapitel 5**) beschreibt den Prozess der Testsystemerzeugung mit der dafür benötigten Metamodellerweiterung, den Modelltransformationen und Ansätzen zur Generierung von Testfällen. Im fünften und letzten Teil (**Kapitel 6**) wird die Methode anhand eines Beispielsystems für einen Web-Shop demonstriert.

Kapitel 2

Grundlagen

In diesem Kapitel werden Konzepte und Grundlagen vorgestellt, die für das Verständnis der nachfolgenden Kapitel nötig sind.

2.1 Testen und Testautomatisierung

Man kann getrost annehmen, dass jedes Programm oder Softwaresystem ab einer gewissen Größe Fehler enthält. Dies gilt mit an sicher grenzender Wahrscheinlichkeit für jedes System, welches den Umfang eines vermarktbaren Softwareprodukts erreicht hat. Verschiedene Techniken zur Fehlerbeobachtung bieten verschiedene Vor- und Nachteile. Eine besonderheit von Softwaretests ist, dass man damit aufzeigen kann, dass Fehler auftreten werden, möglicherweise zu einem Zeitpunkt bevor ein Produkt ausgeliefert wird.

2.1.1 Grundlagen des Testens

Nach Glenn Meyers ist Testen der Prozess, ein Programm auf systematische Art und Weise auszuführen, um Fehler zu finden[MSB11]. Dies ist nicht nur für vollständige Programme bzw Systeme möglich, sondern auch für einzelne Teile einer Software, sofern diese testbares Verhalten besitzt. Das zu testende System wird beim Testen als *system under test*, kurz *SUT* bezeichnet. Die grundsätzliche Vorgehensweise beim Testen beinhaltet das Senden von Nachrichten an das SUT und die Analyse von resultierendem Verhalten. Die an das SUT gesendeteten Nachrichten werden auch als Teststimuli bezeichnet. Grundsätzliche Voraussetzungen für die erfolgreiche Durchführung von Softwaretests sind nach [ABC82]:

- Das zu testende System bzw Systemteile in ausführbarer Form
- Eine Beschreibung des zu erwartenden Verhaltens
- Beobachtungsmöglichkeiten für das tatsächliche Verhalten
- Spezifikation der berücksichtigten Eingabemöglichkeiten (Definitionsmenge)
- Eine Entscheidungsmöglichkeit darüber, ob beobachtetes und erwartetes Verhalten übereinstimmen.

Der Charakter von Softwaretests lässt sich als destruktiv beschreiben, da hierdurch nicht die Abwesenheit, sondern nur das Vorhandensein von Fehlern gezeigt werden kann[Dij72]. Die Korrektheit eines Systems durch Testen vollständig zu beweisen ist nur dann möglich, wenn die Menge der Ein- und Ausgaben endlich und überschaubar

ist. Dies ist eine Seltenheit, da schon einfachste Programme meistens einen unendlichen Werte- und Definitionsbereich haben. Eine Folge davon ist, dass ein System durch Testen nur stichprobenartig geprüft werden kann. Ein durch Testen gefundener Fehler ist eine Abweichung zwischen Ist-Verhalten (im Testlauf festgestellt) und Soll-Verhalten (in der Spezifikation gefordert) oder ein nicht erfülltes, vom Kunden vorausgesetztes Qualitätskriterium.

2.1.2 Vorgehensweisen

Beim systematischen Testen von Software lassen sich die gebräuchlichen Vorgehensweisen in drei wesentliche Kategorien einteilen[Güd10]. *Manuelles Testen* ist das in der Industrie noch immer am häufigsten eingesetzte Verfahren. Testfälle werden hierbei von einem Domänenexperten erstellt und in der Regel textuell oder tabellarisch dokumentiert. Die Testfälle werden manuell ausgeführt und die Testergbnisse dazu notiert. *Skriptbasierte Testautomatisierung* verfolgt den Ansatz, Testskripte bzw Testprogramme zu erstellen, welche automatisiert ausgeführt werden können. Die dazu verwendeten Testskripte werden entweder manuell oder mittels so genannter *Capture/Replay*-Werkzeuge erstellt. Durch die Möglichkeit der automatisierten Testdurchführung ist der Prozess der Testausführung nicht nur vergleichsweise schneller als bei manuellem Testen, sondern auch mit stets exakt reproduzierbarem Testverhalten beliebig oft wiederholbar. Ein Nachteil bei dieser Vorgehensweise ist die Notwendigkeit, Testfälle anpassen zu müssen, beispielsweise wenn während des Projekts eine Änderung der Spezifikation vorgenommen wird. *Modellbasiertes Testen* nennt man Techniken, welche den Testprozess durch den Einsatz von Modellartefakten systematisieren und automatisieren. Abschnitt 2.2 geht auf diese Testtechnik ausführlicher ein.

Am Ende der Testausführung steht ein Testurteil zur Verfügung (englisch: $Verdict$), welches das Ergebnis des Tests darstellt. Testurteile können z.B. sein:

- **Pass**: es wurde kein Fehlverhalten des SUT festgestellt
- **Fail**: es wurde Fehlverhalten des SUT festgestellt
- **Inconclusive**: ein genaues Testurteil konnte nicht festgelegt werden
- **Error**: während der Testausführung sind Fehler innerhalb der Testumgebung aufgetreten

Üblicherweise lassen sich Testarten nach architektonischen Ebenen oder Projektphasen unterscheiden. Einzelne Beispiele dafür werden in Tabelle 2.1 aufgelistet. Die Begriffe *Unit-Test* und *Modultest* werden in dieser Arbeit als gleichbedeutend behandelt.

Modultest	Labortest einzelner Klassen oder Operationen
Komponententest	Labortest einer Softwarekomponente
Integrationstest	Labortest zur Integration mehrerer Module oder Komponenten
Systemtest	Test des Gesamtsystems (automatisiert oder manuell)
Abnahmetest	Test durch den Kunden
Feldtest	Test während des Einsatzes des Produkts

Tabelle 2.1: *Beispiel für mögliche Unterscheidung von Testebenen und Testarten*

2.1.3 xUnit et al

Unter dem Stichwort xUnit versteht man eine Reihe von Testframeworks, welche Konzepte zur systematischen Testausführung anbieten. Das bekannteste Framework ist JUnit[jun13], welches für die Testautomatisierung in Java-Umgebungen eingesetzt werden kann. Die Bezeichnung xUnit ist aus dem Namen JUnit entstanden, welches das erste Testframework dieser Art mit weiterer Verbreitung war.

Da beim Testen von Software bestimmte Mechanismen verwendet werden, wie z.B. die Fällung eines Testurteils oder Schritte, die zur Vorbereitung eines Tests ausgeführt werden müssen, hat sich diese Art von Werkzeug etabliert, um diesen Zweck in spezialiserter Weise erledigen zu können. Daduch können sich Programmierer stärker auf die Ausformulierung von Tests konzentrieren und die Wahrscheinlichkeit, dass ein Fehler in der Testautomatisierung auftritt, der unbemerkte Seiteneffekte auf das Testurteil hat, ist dadurch sehr gering.

Das hauptsächliche Einsatzgebiet für diese Testframeworks sind Modultests, es können mit diesen jedoch in der Regel auch andere Testarten bis hin zu Systemtests erstellt werden. Frameworks für Modultests und deren Gebrauch sind ausführlich in [MSB11] beschrieben.

2.1.4 Testen von Webanwendungen

Da sich das Testen von Webanwendungen in der Regel als nicht trivial durchführbar erweist, wurden hier eine Reihe von Frameworks geschaffen, die sich auf diesen Zweck spezialisiert haben. Die Ausprägungen verschiedener Web-Testframeworks lassen sich in zwei Kategorien einteilen, von denen eine als *treiberlos* (in Anlehnung an das Wort „Treiber“) bzw direkt kommunizierend bezeichnet werden kann und die andere als *treiberbasiert* bzw indirekt kommunizierend.

Die **treiberbasierten** Ansätze verwenden einen Testtreiber, welcher innerhalb des Tests instanziiert wird und die Stimulation des SUT übernimmt. Dies ist der Regel eine Softwarekomponente mit der Funktionalität eines Webbrowsers. Die Ausführung von Tests erfolgt dann durch Skriptsteuerung des Testtreibers. Diese Gruppe lässt sich weiter in GUI-basierte und sogenannte *headless*-Ansätze unterscheiden. Erstere verwendet Techniken, um ein bestehendes Softwareprodukt aus der Reihe der Webbrowser (beispielsweise Firefox) durch eine geeignete Schnittstelle zu programmieren und so die bereits vorhandene Funktionalität zum Zweck des Testens zu nutzen. Die *headless*-Variante hingegen stellt eine eigene Softwarekomponente zur Verfügung, welche die Funktionalität eines Browsers leisten kann, jedoch keine Benutzeroberfläche besitzt. Letzteres stellt für die Testautomatisierung eine Vereinfachung dar, weil das Testen mit GUI-basierten Ansätzen ohne zusätzlichen Aufwand die Möglichkeiten bei der Testautomatisierung einschränkt.

Bei den **treiberlosen** Ansätzen wird die Philosophie verfolgt, die direkt stattfindende HTTP-Kommunikation zu beschreiben, wozu innerhalb eines Tests HTTP-Anfragen erstellt und an das SUT gesendet werden um die resultierenden Antworten zu analysieren. Das Testframework HttpUnit[htt13] ist ein bekannter Vertreter dieser Variante. Obwohl dies ein berechtigter Ansatz ist, der in einzelnen Fällen eine geeignete Wahl darstellt, lassen sich Benutzerinteraktionen auf der Oberfläche von Webanwendungen auf diese Weise nur schwer realisieren, insbesondere bei der Nutzung von Javascript oder ähnlichen Techniken der Dynamisierung innerhalb der Benutzeroberfläche einer Anwendung.

2.1.4.1 HtmlUnit

HtmlUnit[htm13] ist ein Vertreter der zuvor erwähnten *headless*-Variante für Java-basierte Tests. Hier wird innerhalb eines Tests der Testtreiber als Objekt instanziiert und über eine API programmiert. Hierbei können durch verschiedene Auswahltechniken Elemente des HTML Dokumentenbaums aufgespürt und im Anschluss anaylsiert, manipuliert oder im Fall von Links oder Formularen aktiviert bzw abgesendet werden. HtmlUnit unterstützt auch Javascript

2.1.4.2 Selenium

Selenium[sel13] verfolgt ebenfalls einen treiberbasierten Ansatz, nutzt dabei aber die Vorgehensweise der Steuerung eines bestehenden Webbrowsers. Selenium stellt eine Programmierschnittstelle zur Verfügung, welche die Erstellung von Testfällen ermöglicht, die auf unterschiedlichen Browsern zur Ausführung gebracht werden können. Eine Besonderheit bei Selenium ist, dass neben zahllosen GUI-basierten Browsern auch die Verwendung von HtmlUnit unterstützt wird. Zudem existieren für Selenium auch Treiber von Drittanbietern, wodurch eine Unterstützung für Browsersysteme unterschiedlicher Hersteller verfügbar ist. Die Programmierschnittstelle steht für verschiedene Sprachen zur Verfügung, unter anderem auch Java.

2.2 Modellbasiertes Testen

Modellbasiertes Testen(MBT) ist ein Schlagwort, unter welchem verschiedentste Techniken der Nutzung von Modellartefakten im Testprozess verstanden werden. Zusätzlich dazu wird durch den Begriff auch die Motivation ausgedrückt, Modelle bzw Modellartefakte nicht nur bei der Erstellung von Software im Rahmen modellbasierter Systementwicklung wie z.B. der MDA zu nutzen, sondern auch bei der Generierung, Gütebewertung, Ausführung und Wartung von Testartefakten [Sch07].

Streng betrachtet basiert jede Form des deterministischen, automatisierten Testens auf einem Modell. So könnte man beispielsweise einen JUnit Testfall als Modell bezeichnen, weil die Semantik des Codes in gewisser Weise ein konkretes, deterministisches Modell definiert, welches zur Gütebewertung des zu testenden Systems herangezogen wird. Es sei daher an dieser Stelle erwähnt, dass MBT diese Form von Tests *nicht* mit einschließt. Tatsächlich unter MBT fallen Testverfahren, die explizite Modellartefakte zur Grundlage haben wie beispielsweise auf Basis der UML.

Die Idee des modellbasierten Testings ist nicht neu. Bereits in den 60er Jahren wurden Grundlagen für das modellbasierte Testen mit endlichen Automaten geschaffen. Dennoch hatten sich Techniken aus dem Umfeld des MBT kaum durchgesetzt und kamen wenn dann nur vereinzelt in Verwendung. Ein möglicher Grund dafür mag das jahrzentelange Fehlen von standardisierten Modellierungstechniken sein, die eine wesentliche Grundlage für das Arbeiten mit Modellen darstellen. Diese Situation hat sich durch das Erscheinen der UML verändert und es liegt seit dem eine standardisierte Modellierungssprache vor, die im letzten Jahrzent ihren Weg in nahezu jeden Bereich der Softwareentwicklung gefunden hat. Zusätzlich zum Vorhandensein der UML hat das Propagieren der MDA die Dikussion rund um das Testen auf Basis von Modellen neu entfacht. So kommt es dazu dass in den letzten Jahren MBT immer populärer wurde und sich sowohl Industrie als auch Forschung stärker dem Thema zuwenden.

Als Model-Driven Testing (MDT) werden in dieser Arbeit Testverfahren bezeichnet, welche einen Ansatz verfolgen, der nicht nur auf Basis eines Systemmodells Testfälle erstellt, sondern ein zusätzliches, so genanntes Testmodell für die explizite Definition von Testumgebungen und Testverhalten führt.

2.2.1 Einordnung modellbasierten Testens in Testverfahren

Modellbasiertes Testen kann sowohl in manuellen als auch in automatisierten Testverfahren Verwendung finden. Für gewöhnlich lässt sich in letzteren eine höhere Effizienz erzielen. Desweiteren lassen sich modellbasierte Testtechniken sowohl für statische als auch für dynamische Testzwecke einsetzen.

Statisches Testing Rein statische Testverfahren beschränken sich auf die Analyse von Modellen. Dabei wird üblicherweise die Beschaffenheit des Systemmodells untersucht um dessen strukturelle Eigenschaften und Beziehungen zu prüfen. Grundsätzlich lassen sich dabei drei wesentliche Kriterien untersuchen: Syntax, Semantik und Stil.

Als syntaktische Korrektheit lässt sich die wohlgeformte Modellierung aus Sicht der gegebenen Modellierungsvorschriften verstehen. Es kann also etwa geprüft werden, ob die im speziellen zur Verfügung stehenden Mittel der Modellierung ordnungsgemäß eingesetzt wurden, ob das Modell syntaktisch in sich geschlossen ist und keine syntaktisch notwendigen Elemente fehlen. Die meisten Modellierungswerkzeuge unterstützen diese Art von Tests bereits automatisch. Syntaktische Korrektheit gewährleistet, dass das Modell stets gleich interpretiert wird.

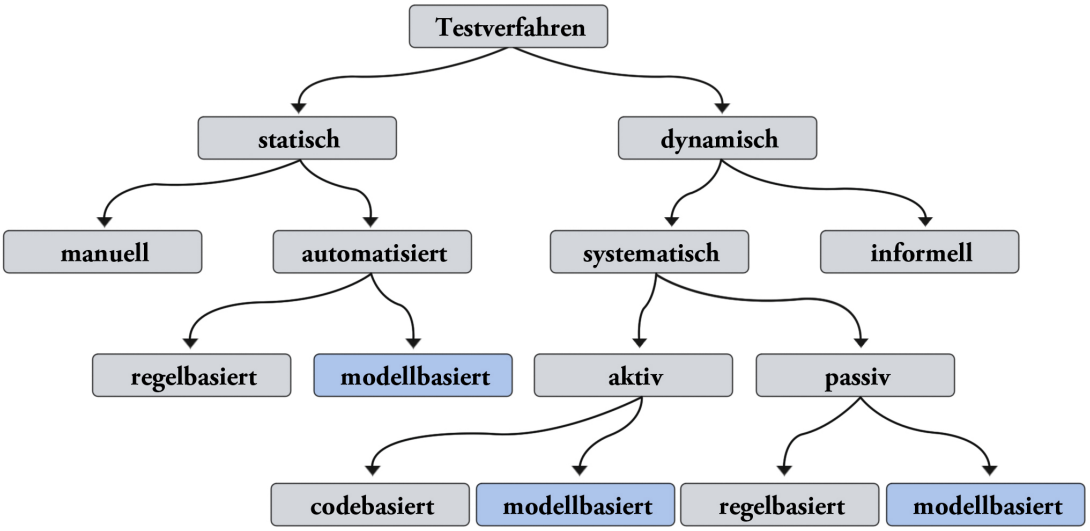


Abbildung 2.1: *Einordnung modellbasierter Testverfahren (nach [Sch07])*

Auf semantischer Ebene lässt sich einerseits, soweit möglich, die semantische Vollständigkeit des Modells untersuchen und andererseits die Konsistenz etwaiger vorhandener Teilmodelle untereinander. Für den Aspekt der semantischen Vollständigkeit lässt sich z.B. dort wo es möglich ist die Vollständigkeit der Umsetzung von Anforderungen an das System untersuchen. Dies könnte z.B. das Vorhandensein von Sicherungsmechanismen, Benutzerrollen oder bestimmten Ansichten betreffen und lässt sich in modellbasierter Weise oftmals nur unter Verwendung einer formalen Anforderungsspezifikation realisieren. Typischerweise werden Systeme unter Verwendung verschiedener Diagrammtypen modelliert, da sich für verschiedene Teilaspekte eine Systems unterschiedliche Notationen eigenen. Hier kommt es oftmals zu Überdeckungen, so dass bestehende Elemente in einem anderen Teil des Modells verwendet oder gar verfeinert werden. Dabei kann es theoretisch zu widersprüchlichen Aussagen kommen, die leicht zu übersehen sind. Die Anaylse der semantischen Konsistenz erlaubt es, solche widersprüchlichen Aussagen zu identifizieren und aufzudecken um semantisch inkonsistene Teile von Modellen zu erkennen.

Das Testen des Modellierungsstils lässt sich prinzipiell nur vornehmen, wenn entsprechende Stilvorgaben existieren. Oft ist dies jedoch der Fall, da sich die Qualität der Modellierung dadurch steigern lässt, z.B. indem durch bestimmte Namenskonventionen Modelle leichter lesbar werden. In diesen Fällen kann es möglich und sinnvoll sein, das Einhalten von Stilvorgaben in Modellen zu prüfen um die dadurch gewünschte Qualitätssteigerung zu gewährleisten.

Dynamisches Testing Häufiger trifft man beim Einsatz modellbasierter Testverfahren auf dynamische Tests. Diese Umfassen alle Testarten, die sich mit der Analyse im Rahmen der Ausführung des Systems befassen. Unterschieden werden hier passive und aktive Herangehensweisen. Beim **passiven** Ansatz werden Ausführungspfade(Traces) aufgezeichnet, während sich das System in Ausführung befindet. Die so gewonnenen Traces werden im Anschluss unter Verwendung der Systemspezifikation ausgewertet. Dadurch können unter anderem Verletzungen von Systeminvarianten und Eigenschaften aufgedeckt werden, die beispielsweise in Varianten der temporalen Logik beschrieben sind.

Bei den **aktiven** Testverfahren werden konkrete Teststimuli an das SUT gesendet und die daraus resultierenden Reaktionen beobachtet und ausgewertet. Auf diese Weise können Struktur- und Verhaltensaspekte des SUT während der Ausführung analysiert und bewertet werden. Dazu besteht meistens die Möglichkeit, Daten- Struktur- und Verhaltensinformation aus dem Systemmodell zu gewinnen und entsprechende Tests abzuleiten. Ein anderer Ansatz ermöglicht die Definition von Testabläufen, indem einzelne Testfälle durch die explizite Angabe von Verhaltensspezifikationen definiert werden. Dies erfolgt üblicherweise durch die Verwendung entsprechender Verhaltensdiagramme wie z.B. Zustandsautomaten oder Interaktionsdiagrammen wie z.B. Sequenzdiagramme. So lässt sich das Absetzen von Teststimuli an das SUT und die Analyse der Reaktionen explizit formulieren. Auf die verschiedenen Varianten des aktiven modellbasierten Testings wird in Abschnitt 2.2.3 näher eingegangen. Abbildung 2.1 zeigt eine mögliche Einordnung modellbasierter Testverfahren.

2.2.2 Relationen zwischen Test und System

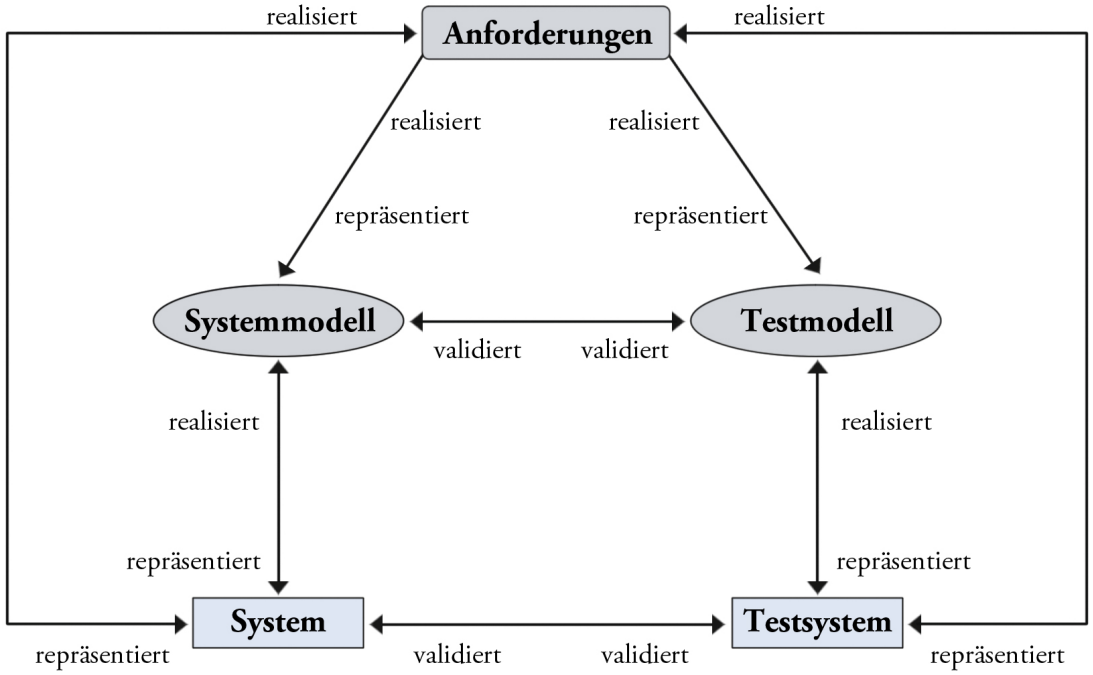


Abbildung 2.2: *Relationen zwischen Testsystem und System und deren Modellen (nach [Sch07])*

In einem ordentlichen Entwicklungsprozess existiert zu jeden System eine Menge von Anforderungen. Die auf dieser Grundlage erstellen Systeme und Testtestsysteme repräsentieren in gewisser Weise diese Anforderungen, da sie jeweils auf eigene Weise eine Realisierung derselben darstellen. Der selbe Zusammenhang gilt, wenn auch auf anderer Abstraktionsebene, für deren Modelle. In dieser Konstellation haben System und Testsystem eine wechselseitige Beziehung untereinander. Während das Testsytem zum Zweck der Validierung oder Verifikation des Systems zur Vefügung gestellt wird, gilt gleiches auch in umgekehrter Richtung: Das System kann theoretisch zur Validierung des Testsystems herangezogen werden. Auch diese Beziehungen übertragen sich auf die jeweiligen Modelle und erlauben daduch eine gegenseitige Validierung beider Modelle. Dies ist insofern interessant, da eine solche Validierung bereits in einer relativ frühen

Phase des Projekts möglich ist. Abbildung 2.2 veranschaulicht die Zusammenhänge zwischen System und Testsystem.

2.2.3 Varianten modellbasierten Testens

Es gibt mehrere Varianten, wie sich modellbasierte Tests auf Basis von System- und Testmodellen gestalten lassen. Diese sind in Abbildung 2.3 dargestellt und lassen sich in die drei Gruppen Systemmodell-getrieben, Testmodell-getrieben und System- und Testmodell-getrieben einordnen.

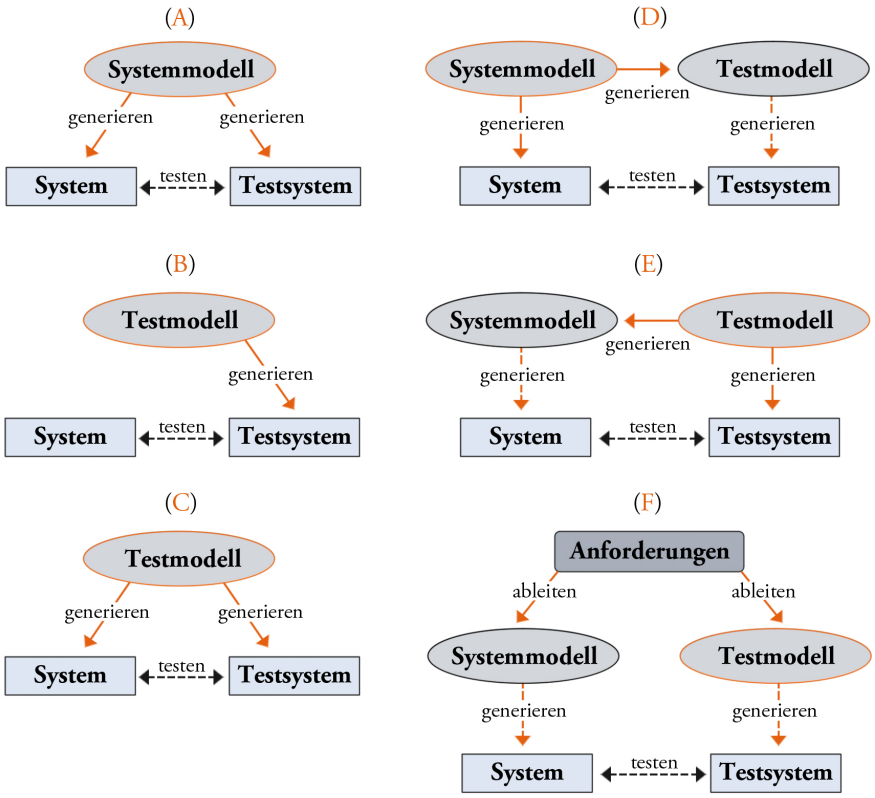


Abbildung 2.3: *Varianten modellbasierten Testings (nach [Sch07])*

Systemmodell-getrieben Eine weit verbreitete Herangehensweise bei modellbasierten Tests ist die alleinige Verwendung eines einzelnen Systemmodells, in Abbildung 2.3 zu sehen als Variante (a). Bei diesem Ansatz werden System und Testsystem jeweils vollständig oder in Teilen aus dem Systemmodell generiert. Für die Erstellung singulärer Systemmodelle lassen sich in der Praxis verschiedenste Modellierungstechniken heranziehen. Oftmals werden diese Modelle mit der UML erstellt, es ist jedoch auch möglich andere Techniken zu verwenden wir z.B. Message Sequence Charts, Petrinetze, temporalte Logiken etc. Die systemmodell-getriebene Variante bietet den Vorteil, dass nur ein einzelnes Modell erstellt werden muss. Bei modellgetriebenen Prozessen ist ein solches solches bereits im Entwicklungsprozess vorhanden. Dadurch ergibt sich nicht die Notwendigkeit eines zusätzlichen Modellierungsaufwandes und zugleich taucht das Problem möglicher semantischer Inkonsistenzen zwischen Test- und Systemmodell gar nicht erst auf. Tatsächlich resultiert daraus bei genauer Betrachtung aber zugleich auch

ein Nachteil. Aufgrund der Tatsache, dass System und Testsystem aus dem gleichen Modell erzeugt werden, ist es auf diese Weise nicht möglich, Fehler im Systemmodell durch Tests aufzudecken. Eine Unabhängigkeit zwischen System und Testsystem ist also nicht gewährleistet und eine mögliche Validierung des Systemmodells mit Rücksicht auf die Anforderungen ist bei diesem Ansatz durch Testing kaum oder nur sehr eingeschränkt möglich. Eine weitere Einschränkung bei dieser Vorgehensweise ist die Einschränkung der daraus resultierenden Testmöglichkeiten auf die in der Systemspezifikation Vorhandenen Merkmale. Berücksichtigt man die Tatsache, dass totale Systemmodellierung im Sinne einer vollständigen Spezifikation oftmals nicht praktikabel ist, weswegen viele Systemmodelle nur eine bestimmte Granularität an Details enthalten und somit streng betrachtet nicht vollständig sind, so ist schnell zu erkennen, dass sich diese Tatsache auch auf das bei dieser Variante erstelle Testsystem auswirkt, welches somit über einen gewissen Detailgrad nicht hinauswachsen kann.

Testmodell-getrieben Eine anderer Strategie für modellbasiertes Testen verfolgt den Ansatz des eigenständigen Testmodells. Analog zur systemmodell-getriebenen Ansatz steht hier das Testmodell im Zentrum.

Das Testsystem wird aus dem Testmodell erzeugt, in Abbildung 2.3 dargestellt als Variante (b). Zur Modellierung des Testsystems können proprietäre aber auch standardisierte Techniken eingesetzt werden. Standardisierte Techniken hierfür sind das UML 2.0 Testing Profile (U2TP) oder OMG sowie die Testing und Test Control Notation(TTCN-3), ein ETSI und ITU-T-Standard. Ein Vorteil bei der Verwendung des U2TP besteht in der Möglichkeit, Elemente aus dem Systemmodell wiederzuverwenden. Ein Vorteil bei der Verwendung von TTCN-3 liegt in der durchgängigen Automatisierung bei der Testausführung, welche für lokale und verteilte Plattformen gewährleistet ist. Dafür kann der Einarbeitungsaufwand für TTCN-3 als höher eingestuft werden. Sowohl U2TP als auch TTCN-3 erlauben eine grafische Darstellung der Tests.

Variante (c) ist dabei eine optionale Ausprägung, die die Generierung des Systemmodells aus dem Testmodell verfolgt. Sie ist ein möglicher Ansatz für die Kombination der Paradigmen „test first“ und „modellbasiert“, jedoch ist hier natürlicherweise auch das Problem der fehlenden Unabhängigkeit zwischen Systemmodell und Testmodell gegeben. Hierzu findet sich ein besserer Ansatz in Variante (e)

System- und Testmodell-getrieben Die konsequenteste Umsetzung des modellbasierten Gedankens findet man in den Varianten (d), (e) und (f). Hier werden Modelle sowohl für die System- als auch für die Testseite eingesetzt. Eine Möglichkeit hierbei ist die Ableitung oder Generierung des einen Modells aus dem anderen, also das Testmodell aus dem Systemmodell zu erzeugen oder umgekehrt. Nach Ableitung bzw Generierung ist immer noch eine Verfeinerung z.B. durch Ergänzungen möglich. Die am häufigsten eingesetze Variante ist (d). Meist werden hier automatenbasierte Techniken eingesetzt[Sch07]. Die Variante (e) ist sicherlich auch eine interessante Vorgehensweise. Sie wurde zwar von einigen Firmen(darunter auch IBM) propagiert, jedoch ist bisher nichts konkretes über eine tiefere Auseinandersetzung damit bekannt [Sch07]. Die letzte hier aufgeführte Möglichkeit besteht darin, System- und Testmodell getrennt voneinander zu erstellen. Auf diese Weise können die unterschiedlichen Sichtweisen der System- und Testentwickler eigene Ansätze in Bezug auf die Anforderungen entwickeln. Dies führt zwar theoretisch zu Redundanzen und lässt Spielraum für mögliche Inkonsistenzen, erlaubt es aber, etwaige semantische Fehler im Testmodell, wie nicht ordnungsgemäß umgesetzte Anforderungen, durch das unabhängig erstellte Testmodell

aufzudecken. Diese letzte Variante wird auch in [PP05] erwähnt.

2.3 Model Driven Architecture

Dieser Abschnitt erläutert Voraussetzungen und Grundlagen der Model Driven Architecture (MDA), Erweiterungen der MDA um testrelevante Konzepte sowie Werkzeuge zur allgemein modellgetriebenen Entwicklung von Software.

2.3.1 Metamodellierung und die Meta Object Facility

Das Konzept der Metamodellierung ist eine essentielle Grundlage für die Definition und maschinelle Verarbeitung von Modellierungssprachen und deren Modellen. Die Meta Object Facility[Gro03] (Kurzschreibweise MOF) spielt dabei die Rolle einer Metametamodellierungssprache. Das Prinzip dahinter lässt sich in Kürze an einem abstrakten Beispiel illustrieren. Man nehme eine Beschreibungssprache MM und definiere damit eine Beschreibungssprache M, welche wiederum verwendet wird, um eine Beschreibungssprache L zu definieren, durch deren Anwendung die Modellierung von softwarerelevanten Konzepten durch Ingenieure durchgeführt wird. L wäre dann eine konkrete Modellierungssprache (z.B. die UML). Ausgehend von der Ebene von L betrachtet stellt nun M das zugehörige Metamodell dar (z.b. das UML Metamodell) und MM das Metametamodell (z.B. MOF).

Durch die Beschreibung von Modellierungssprachen auf Basis einer solchen Metamodellierungshierarchie steht eine einheitliche Definitionsgrundlage zur Verfügung, die es ermöglicht, Modelle, welche auf dem selben Metametamodell basieren, durch ein und das selbe Austauschformat zwischen Werkzeugen verschiedener Hersteller zu bewegen. Zusätzlich ergibt sich auf Basis von Metamodellen eine Grundlage für die Möglichkeit, Informationen zwischen Modellinstanzen unterschiedlicher Metamodelle zu konvertieren, sofern sich diese ein gemeinsames Metametamodell teilen. Letzteres bildet die Grundlage für die später benötigten Modelltransformationen.

Eine übliche Betrachtungsweise der Metamodellierungshierarchie für Sprachen aus dem Umfeld der UML ist die Darstellung durch vier horizontale Schichten, welche als M3 bis M0 bezeichnet werden. Die oberste Schicht M3 symbolisiert die Ebene der Metametamodelle, gefolgt von M2 für Metamodelle und M1 für Benutzermodelle. Die vierte und letzte Schicht M0 symbolisiert Instanzen der realen Welt, im Umfeld der Softwareentwicklung üblicherweise Objektinstanzen von Klassen, welche auf der darüberliegenden Ebene M1 definiert wurden. Eine Darstellung der Metamodellerierungshierarchie mit den Schichten M3 bis M0 ist in Tabelle 2.2 gezeigt.

	Ebene	Repräsentation durch
M3	Metametamodell	MOF
M2	Metamodell	UML Metamodell
M1	Modell	Benutzermodell
M0	System	Instanzen zur Laufzeit

Tabelle 2.2: *Ebenen M3 bis M0 der Metamodellierungshierarchie*

Die Konzepte der Metamodellierung bilden eine wichtige Grundlage für die Model Driven Architecture, welche in Abschnitt 2.3 erläutert wird. Das gebräuchliche Austauschformat, welches beim Austausch von metamodellbasierten Modellartefakten am

häufigsten eingesetzt wird, ist das XML-basierte *XML Metadata Interexchange*, kurz XMI[Bai02].

2.3.2 Unified Modeling Language

Die Unified Modeling Language[RJB04] (Kurzschreibweise: UML) ist eine Modellierungssprache, welche zur Spezifikation von Softwaresystemen entwickelt wurde. Sie beschreibt grundlegende Modellierungskonzepte, die bei der Modellierung von Softwaresystemen nützlich sind und unterstützt durch eine Reihe von Diagrammtypen auch grafische Notationen. Die UML wird in Form eines MOF-basierten Metamodells angeboten und definiert die durch sie notierbaren Elemente in Form von Metaklassen.

Sterotypen Ein besonderes Merkmal der UML sind die vielfältigen Erweiterungsmechanismen, welche es erlauben, die UML für plattform- oder domänenspezifische Zwecke anzupassen. Ein grundlegendes Notationselement ist dabei der Stereotyp, welcher die Erweiterung bestehender UML-Elemente um plattform- oder domänenspezifische Charakteristika erlaubt. Bei der Definition eines Stereotyps wird festgelegt, welche Metaklasse oder Metaklassen selbiger erweitern kann. Dies bedeutet, dass die Anwendung eines Stereotyps nur auf Instanzen der entsprechenden Metaklassen erfolgen kann. Stereotypen werden ähnlich wie UML Schlüsselworte notiert indem sie durch ein Paar französischer Guillemets(«, ») eingeschlossen werden. Zu Stereotypen können auch Eigenschaften definiert werden, welche bei der Anwendung eines Stereotyps mit Werten belegt werden können. Oft finden Stereotypen im Kontext von Codegenerierungsverfahren Anwendung.

Profilerweiterungsmechanismus Ein grundlegender Erweiterungsmechanismus der UML ist die Profilerweiterung. Ein Profil definiert einmalig eine Menge von Erweiterungen und kann anschließend in mehreren UML-Modellen eingebunden werden, was die Nutzung der Erweiterungen innerhalb des Modells ermöglicht. Dabei stellen Profile in der Regel Stereotypen zur Verfügung. Um ein Profil und die darin enthaltenen Stereotypen innerhalb eines Modells zu verwenden, muss das Profil auf das Modell *angewendet* werden. Die daraus resultierende Beziehung von Modell zu Profil wird *Profilanwendung* genannt. Die von der UML zur Verfügung gestellten Diagrammtypen lassen sich in Struktur- und Verhaltensdiagramme einordnen. Auf grundlegende Diagrammtypen der UML wird hier nicht näher eingegangen. Sie werden als bekannt vorausgesetzt und werden beispielsweise in [RQZ07] ausführlich erläutert. Es folgen kurze Erläuterungen zu den in dieser Arbeit verwendeten Modellierungskonstrukten einiger Diagramme.

Sequenzdiagramm Ein Diagrammtyp, der in dieser Arbeit eine wichtige Rolle spielt, ist das Sequenzdiagramm. Sequenzdiagramme gehören zu den UML Interaktionsdiagrammen, welche Verhaltensdiagramme sind. Interaktionen ermöglichen die Feindarstellung von Kommunikationsabläufen zwischen mehreren an einer Interaktion teilnehmenden Modellelementen. Sequenzdiagramme heben die zeitliche Reihenfolge einer Interaktion hervor. Auf die Grundlagen von Sequenzdiagrammen soll an dieser Stelle nicht näher eingegangen werden, weswegen ebenfalls auf [RQZ07] verwiesen wird.

Innerhalb von Sequenzdiagrammen kann auf einer Lebenslinie eine *Invariante* einbeschrieben werden, welche eine Zusicherung durch eine Bedingung ausdrückt, die in einer geeigneten Sprache zu formulieren ist, z.B. mit MVEL. Die Semantik einer Invariante besagt, dass die enthaltene Formel als Zusicherung verstanden wird genau

dann, wenn der darauf folgende, sich auf der selben Lebenslinie befindliche Endpunkt einer Nachricht ausgeführt wird. Die Zustandsinvariante gilt dabei *direkt vor* der Ausführungsspezifikation des folgenden Nachrichtenendpunkts. Die Notation erfolgt durch Darstellung der Formel, eingeschlossen in geschweiften Klammern. Ein Beispiel dafür ist in Kapitel 4, Abschnitt 4.3.2 in Abbildung 4.7(a) zu sehen.

Eine weitere Form von Zusicherungen in Sequenzdiagrammen ist die *Zustandsinvariante*. Sie kann auf der Lebenslinie eines zustandsbehafteten UML-Elements spezifiziert werden. Die Auswertung erfolgt ebenfalls direkt vor dem darauf folgenden Nachrichtenendpunkt, jedoch wird durch die mit einer Zustandsinvariante verbundenen Zusicherung ausgedrückt, dass sich das UML-Element der entsprechenden Lebenslinie zum Zeitpunkt der Gültigkeit in einem bestimmten Zustand befinden muss.

Zustandsautomaten Mit Zustandsautomaten steht eine Möglichkeit zur Verfügung, das Verhalten einzelner Elemente zu beschreiben. Sie basieren auf dem Prinzip der endlichen Automaten und erweitern dieses um zusätzliche Konzepte. Für die vorliegende Arbeit spielen Zustandsautomaten eine untergeordenete Rolle. Es sei hier jedoch auf ein Detail zur Definition von Zuständen hingewiesen. Zustände werden innerhalb von *Regionen* definiert. Um einen Zustand zu definieren, muss eine Zustandsmaschine erstellt werden, die mindestens eine Region enthält. Zustände selbst können nur innerhalb von Regionen definiert werden.

Bedeutung für die modellgetriebene Softwareentwicklung Die UML stellt eine wichtige Grundlage für die Model Driven Architecture und weitere modellgetriebene Entwicklungsansätze dar. Ein hierbei nützliches Merkmal der UML ist die Möglichkeit, Modelle bzw Spezifikationen ohne größeren Aufwand zwischen verschiedenen Entwicklungswerkzeugen austauschen zu können. Dies ist durch die Natur der UML als MOF-basiertes Metamodell begründet. erreicht. Aus diesem Grund können UML-Modelle in XMI-Datenstrukturen abgelegt und von anderen Werkzeugen eingelesen werden. Für die Bearbeitung von UML-Modellen sind Werkzeuge verschiedener Hersteller verfügbar. In dieser Arbeit wird für die Modellierung das Programm MagicDraw[mag13] eingesetzt.

2.3.3 Vorgehensweise der MDA

Die Model Driven Architecture beschreibt einen Entwicklungsansatz, der sich auf die Verwendung von Modellen als zentrale Projektartefakte stützt[PM06]. Zusätzlich dazu wird die Trennung von fachlicher Funktionalität, plattformunabhängigem sowie plattformspezifischem Entwurf motiviert. Für diesen Zweck wird die Verwendung von drei Modellebenen unterschiedlichen Abstraktionsgrades vorgeschlagen. Die korrespondierenden Modellartefakte zur Systemmodellierung dieser drei Ebenen werden als CIM (computation independent model), PIM (platform independent model) und PSM (platform specific model) bezeichnet.

In einem berechnungsunabhängigen Modell (CIM) werden die für ein Projekt relevanten Lösungsansätze auf rein fachlicher Ebene spezifiziert. Sämtliche IT-relevanten Aspekte werden in diesem Modell ausgeblendet. Als Gedankenbeispiel dazu könnte in einem CIM beispielsweise festgehalten werden, dass ein Kunde die Möglichkeit haben soll, ein nach Gewicht vergütetes Produkt durch einen Bezahlvorgang zu erwerben. Auf Ebene der berechnungsunabhängigen Modellierung wird dabei auf die genaueren Umstände, die eine technische Realisierung erfordern, nicht eingegangen. Ein solches

Modell bietet eine solide Grundlage, um einen konzeptionellen Entwurf für ein Softwaresystem zu erstellen.

Auf der plattformunabhängigen Ebene (PIM) werden Konzepte definiert, die eine softwareseitig realisierbare Lösung des Problems beschreiben, ohne jedoch weiter auf eine konkrete technische Plattform für mögliche Implementierungen einzugehen. Diese Modellierungsebene stellt eine Verfeinerung der zuvor beschriebenen berechnungsunabhängigen Modellierung dar. So könnte in unserem Gedankenbeispiel hier vorgesehen werden, dass die Quantität der erworbenen Produkte durch eine geeignete Waage gemessen wird und anschließend ein elektronischer Bezahlvorgang durch ein Kartenzahlungssystem erfolgen soll. Nicht enthalten in diesem Modell wären Details zur genauen Produktwahl der Waage oder welche konkreten Kartenlese- und Kassensysteme eingesetzt werden sollen. In diesem Entwurf bereits enthalten sein können jedoch bestimmte Verfahren, welche beispielsweise von den Zahlungsssystemen unterstüzt werden sollen. Die Modelle dieser Abstraktionsebene beschreiben damit einen Entwurf, der bereits die technische Realisierbarkeit enthält, ohne jedoch auf konkrete Zielplattformen für das zu erstellende System einzugehen.

Auf der plattformspezifischen Ebene (PSM) wird das zuvor in einem PIM definierte System erneut verfeinert, so dass ein Modell entsteht, welches die Realisierung unter einer konkreten Plattform beschreibt. Die daraus resultierende Spezifikation des Systems ist nun so konkret an implementierungsspezifische Details gebunden, dass Strukturen und benötigte Strukturelemente, die das resultierende System aufweisen soll, bereits erkennbar sind. Für unser Gedankenbeispiel könnte dies nun bedeuten, dass die Verwendeten Schnittstellen zwischen Waage und Kasse so exakt beschrieben werden, dass eine zweifelsfreie Verifikation des resultierenden Systems anhand des plattformspezifischen Modells möglich ist.

2.3.4 Ergänzung der MDA durch Testmodelle

Die zuvor beschriebenen Konzepte und Vorgehensweisen der MDA gehen nicht auf testrelevante Aspekte ein. Da diese jedoch für ordentliche Entwicklungsprozesse als unerlässlich erachtet werden können, sind dazu entsprechende Überlegungen getätigt worden. In [BSKH05] und [BCD+06] werden Ansätze vorgestellt, welche den Ideen der MDA folgen und zu deren plattformunabhängigen und plattformspezifischen Systemmodellen als Additum entsprechende Testmodelle sowie neu entstandene Beziehungen zwischen Modellartefakten und entsprechende Vorgehensweisen propagieren.

In Analogie zur Namenskonvention der entsprechend unter der MDA verwendeten Modellartefakte (PIM und PSM) werden die entsprechenden Testmodelle als PIT (*Platform Independet Test Model*) und PST (*Platform Specific Test Model*) bezeichnet. Ergänzt um die Betrachtungen der berechnungsunabhängigen Ebene (CIM) lässt sich diesen zusätzlichen Modellen noch ein berechnungsunabhängiges Testmodell CIT (*Computation Independet Test Model* hinzufügen, wie in [Sch07] beschrieben. Die dadurch ermöglichte, erweiterte Modellumgebung mit den unterschiedlichen Abstraktionsebenen der MDA und sowohl System- als auch Testmodellen ist in Abbildung 2.4 schematisch dargestellt. Als nicht näher definiertes Artefakt sind hier zusätzlich die Anforderungen eingezeichnet, auf deren Basis CIM und CIT erstellt werden sowie die erstellten Softwareartefakte, die am Ende der Modellhierarchie aus den jeweiligen plattformspezifischen Modellen gewonnen werden. Die zusätzlich eingezeichneten Beziehungen zwischen den einzelnen Modellartefakten verdeutlichen die Beziehungen, die zwischen diesen Modellen bestehen bzw bestehen können. Neben der MDA-typischen

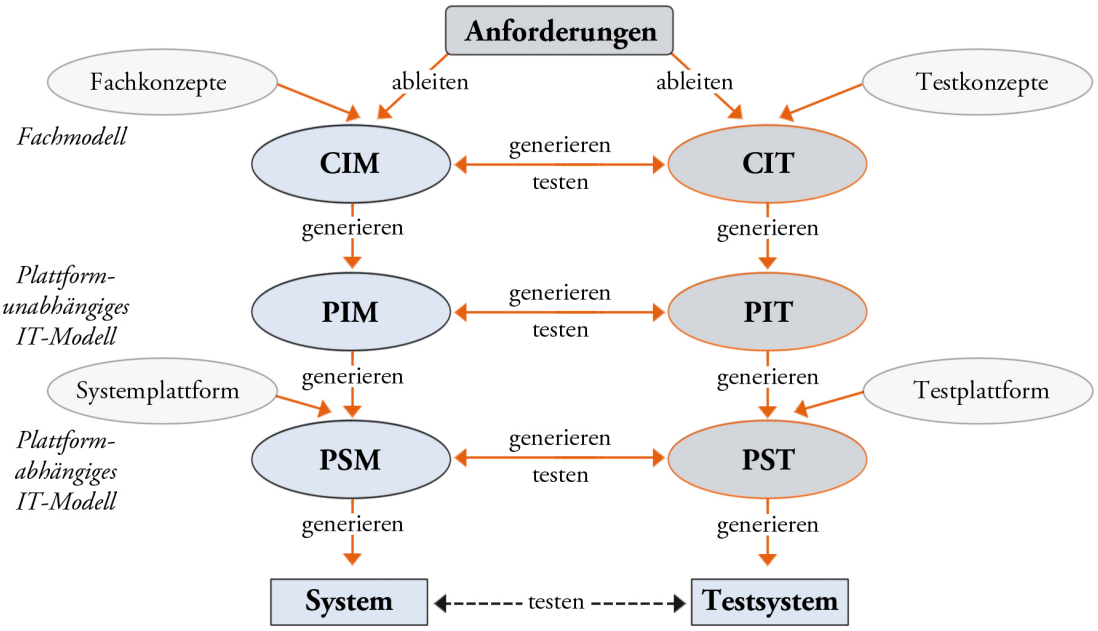


Abbildung 2.4: *MDA mit Erweiterung um Testmodelle (nach [Sch07])*

Vorgehensweise der *top-down*-Generierung von der berechnungsunabhängigen hin zur plattformspezifischen Ebene (dargestellt durch vertikale Pfeile) besteht zusätzlich die Möglichkeit aus den jeweiligen Systemmodellen Testmodelle zu generieren, welche zum Testen ersterer verwendet werden können (horizontale Pfeile).

Wie in Abschnitt 2.2.3 bereits erwähnt, ist theoretisch auch eine Umkehrung dieser Beziehungen möglich, so dass Systemmodelle aus Testmodellen generiert werden und die so erzeugten Systemmodelle zur Verifikation bzw Validierung der entsprechenden Testmodelle herangezogen werden können. An dieser Stelle sei auch erwähnt, dass grundsätzlich jedem Generierungsschritt eine anschließende Verfeinerung des erzeugten Modells folgen kann. Je nach Vorgehensweise bei der Verfeinerung kann dies jedoch Einschränkungen für die Möglichkeiten der Projektautomatisierung zur Folge haben.

2.3.5 Eclipse Modeling Framework

Das Eclipse Modeling Framework[Bud04] (Kurzschreibweise: EMF) ist eine Entwicklungsumgebung, die auf der Entwicklungsplattform eclipse[ecl14] aufbaut und mit der Zielsetzung erstellt wurde, Softwareentwicklung auf Basis von Modellen verschiedener Arten zu ermöglichen bzw zu erleichtern. Die verwendeten Kerntechnologien sind dabei Java, XML und UML.

EMF stellt eine breite Palette von Werkzeugen zur modellgetriebenen Softwareentwicklung zur Verfügung und unterstützt den Austausch von Modellartefakten durch das XMI Format. Mit den Werkzeugen des EMF sind eine Reihe von Prozessen durchführbar, die bei der modellgetriebenen Software verwendet werden. Dazu gehören unter anderem metamodellbasierte *Model-to-Model*-Transformationen, welche unter Nutzung einer Transformationssprache Instanzen eines Metamodells in Instanzen eines anderen Metamodells transformieren können. Die im nachfolgenden Abschnitt erläuterte Transformationssprache ATL ist ein Beispiel dafür und wird von EMF durch eine Reihe von Werkzeugen unterstützt. Ferner sind für EMF Werkzeuge für sogenannte *Model-To-Text*-Transformationen verfügbar, mit welchen die Generierung von Quellcode aus

Modellen durchführbar ist.

2.3.6 Atlas Transformation Language

Die Atlas Transformation Language[JAB+06] (ATL) ist eine Sprache für die Beschreibung von Modelltransformationen, wie sie häufig im Bereich der modellgetriebenen Softwareentwicklung verwendet werden. Das Prinzip dahinter lässt sich beschreiben als die Überführung eines Modells, welches eingelesen wird in ein neues, zu erstellendes Modell. Dabei kann die ATL Modelle verarbeiten, welche Instanzen MOF-basierter Metamodelle sind. Sie unterstützt deklarative und imperative Programmierkonstrukte. Die Definition von Modelltransformationen erfolgt durch die Erstellung einzelner Transformationsregeln, welche auf einem eingehenden und einem ausgehenden Modell operieren. Für die Atlas Transformation Language stehen Werkzeuge in der Eclipse-Umgebung zur Verfügung[ecl13a]. Dazu gehören ein Editor, ein Compiler sowie ein Debugger. Ausführliche Hintergründe zur ATL sind in [JAB+06] zu finden.

2.4 UML Testing Profile

Mit der wachsenden Komplexität heutiger Softwaresysteme steigt auch der Bedarf an systematischen Vorgehensweisen zum Testen selbiger. Wie in Abschnitt 2.3.4 bereits erwähnt wurde, können beispielsweise die Konzepte der MDA auch auf den Prozess des Testens angewendet werden. Trotz ihrer Vielfältigkeit bietet die UML auch in der aktuellen Version 2.4.1 keine tiefergehenden Konzepte zur Definition von Testmodellen. Diese Lücke versucht das UML Testing Profile zu füllen, welches seit 2005 als offizieller Standard der Object Management Group zur Verfügung steht.

2.4.1 Einordnung

Das UML2.0 Testing Profile[UML13] (hier verwendete Kurzschreibweise U2TP) definiert eine Sprache zur systematischen Erstellung von Testspezifikationen, welche im Rahmen von Blackbox-Testverfahren genutzt werden kann[Dai04]. Die dabei verfolgten Konzepte lassen sich in die Konzeptgruppen Testarchitektur, Testverhalten, Testdaten, zeitbezogene Konzepte und Testmanagement einordnen. Auf Basis dieser Konzepte ist eine direkt auf der UML basierende Sprache in Form eines UML-Profils entstanden, die sich zum Entwerfen, Visualisieren, Spezifizieren, Analysieren und Dokumentieren von Modellartefakten eignet, welche im Rahmen der Testsystementwicklung genutzt werden können[UML13].

Das U2TP lässt sich grundsätzlich in Verbindung mit Testansätzen jeglicher Art nutzen. Durch die Ausprägung als UML-Profil lässt sich das U2TP nahtlos in bestehende UML-Umgebungen integrieren und bietet eine ideale Grundlage zur Testsystemspezifikation in Entwicklungsprozessen, welche die UML bereits an anderer Stelle nutzen, z.B. zum Zweck der System- oder Anforderungesspezifikation. In diesen Fällen ist möglich, UTP in bestehende Modelle einzubinden und bereits vorhandene Elemente von System- oder Anforderungsmodellen im Rahmen der Testsystemmodellierung weiter zu nutzen. Dieser Umstand erleichtert das Schließen von Lücken zwischen den Artefakten verschiedener Projektphasen. Eine weitere Eigenschaft, die U2TP aufgrund seiner UML-basierten Natur aufweist, ist die Möglichkeit, Testmodelle schrittweise zu erstellen.

Trotz der Veröffentlichung einer ersten Version im Jahr 2005[OMG05] ist bis heute wenig über konkrete Werkzeugunterstützung zu finden. Als Beispiel für vorhandene

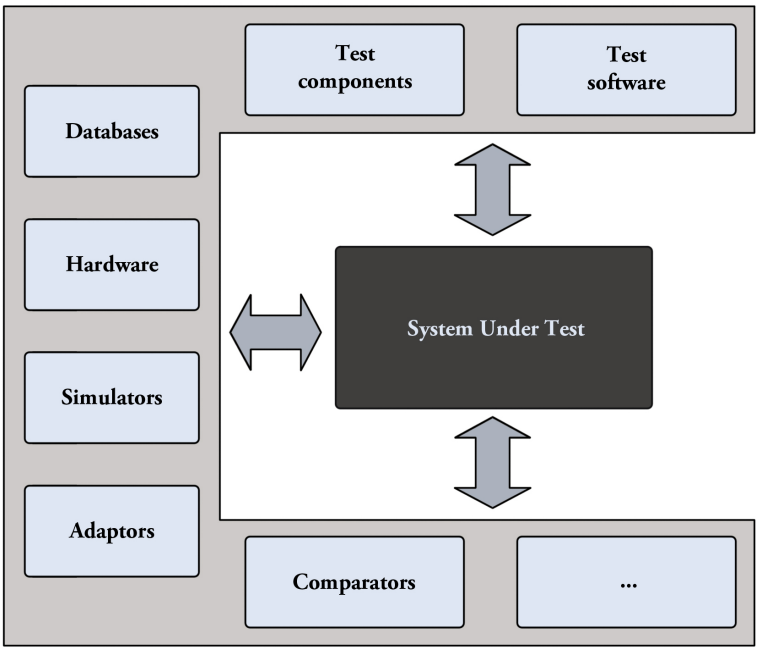


Abbildung 2.5: *U2TP TestUmgebung (nach [UML13])*

Werkzeuguntestützung sei hier nur kurz das Eclipse Projekt *Hyades* erwähnt, welches eine Umgebung zur Qualitätssicherung von Software anbietet und zwischenzeitlich in die *Eclipse Test and Performance Tools Platform*[ecl13b] eingegliedert wurde.

2.4.2 Architektur

Die von U2TP angebotenen Strukturierungsmöglichkeiten fußen auf dem Konzept des so genannten *Testkontextes*. Dieser steht im Zentrum der Testsystemmodellierung und hält Beziehungen zu sämlichen am Test beteiligten Instanzen. Ein Testkontext fungiert dabei als Gruppierungsmechanismus für einzelne Testfälle und enthält zugleich die *Testkonfiguration*, welche die Beziehung einzelner *Testkomponenten* hin zum SUT festlegt. Eine *Testkomponente* kann dabei jede Art von am Test beteiligte Softwarekomponente sein, welche aktiv oder passiv am Testgeschehen teilnimmt und möglicherweise direkt die Stimulation des SUT übernimmt.

Testkomponenten werden innerhalb eines Testmodells erstellt und durch Anwendung des U2TP Stereotyps «TestComponent» als solche typisiert. Ebenso innerhalb des Systemmodells wird ein *Testkontext* als UML Classfier von Typ **Class** erstellt und mit dem Stereotyp «TestContext» versehen. Der Strukturbereich einer mit «TestContext» typisierten Klasse repäsentiert die dem Testkontext zugehörige *Testkonfiguration*. Ihm werden das SUT, Testkomponenten und ggf. weitere, während der Testausführung benötigte Strukturelemente hinzugefügt. Die Teilnehmer einer Testkonfiguration manifestieren sich dadurch als Eigenschaften (UML **Property**) des Testkontextes. So ist es möglich, ein oder mehrere bestehende Elemente eines Systemmodells der Testkonfiguration hinzuzufügen. Auf die daraus resultierende UML **Property** kann nun der Stereotyp «SUT» angewendet werden, welcher das entsprechende Element als SUT des Testkontextes auszeichnet. Abbildung 2.5 zeigt die schematische Darstellung einer U2TP Testumgebung. Eine Testumgebung enthält alle für eine Testsystemspezifikation benötigten Bestandteile wie beispielsweise Datenbanken, Geräte oder zusätzliche Testkompontenten wie Komparatoren oder Factories zur Testdatenerzeugung.

Weitere von U2TP angebotene Strukturelemente sind unter anderem *Arbiter*, *Scheduler* und *DataPool*. Ein *Arbiter* (zu deutsch etwa „Schiedsrichter“) stellt Mittel zur Verfügung, um für einen Testkontext ein Gesamttesturteil zu fällen. *Scheduler* können eingesetzt werden, um die Testausführung feingranular zu steuern und möglicherweise Testkomponenten zu instanziieren, am Test beteiligte Komponenten zu synchronisieren, nebenläufige Testausführungen zu ermöglichen und die Terminierung einer Testfallausführung festzustellen[Dai04]. Das Konzept des Datenpools zur Verwendung von Testdaten in Tests wird in Abschnitt 2.4.4 näher erläutert.

2.4.3 Testverhalten

Unter Testverhalten versteht U2TP die Spezifikation dynamischen Testverhaltens unter Verwendung von Techniken aus der UML wie z.B. Interaktionsdiagrammen, Zustandsmaschinen und Aktivitätsdiagrammen. Grundlegende Aufgabe eines Testverhaltens ist die Stimulation des SUT mit dem Ziel, die darauf folgenden Reaktionen zu beobachten, auszuwerten und bis zum Ende der Testausführung entsprechende Testurteile zu fällen. Für die Erstellung eines einzelnen Testfalls wird dem Testkontext eine Operation hinzugefügt und mit dem Stereotyp «TestCase» versehen. Zur Spezifikation des Verhaltens eines Testfalls wird ein entsprechendes UML Verhaltensdiagramm erstellt und der mit «TestCase» versehenen Operation als UML Verhaltensspezifikation zugeordnet. Dabei kann prinzipiell jede Art von Verhaltensdiagramm aus der UML verwendet werden. In den meisten Fällen eignen sich dazu am besten UML Interaktionen wie z.B. Sequenzdiagramme. Am Test beteiligte Instanzen wie SUT und Testkomponenten werden dabei einem Sequenzdiagramm als Lebenslinien hinzugefügt. Auf dieser Grundlage kann nun durch Modellierung eines Nachrichtenflusses zwischen den Lebenslinien der genaue Ablauf eines Testfalls beschrieben werden, wozu beispielsweise Vorkommen der UML *CallOperation* das Senden von Nachrichten an die öffentlichen Schnittstellen des SUT ausdrücken. Diese Vorgehensweise bietet für das von U2TP verfolgte Konzept des Blackbox-Testens eine Fülle an Möglichkeiten zur Testablaufspezifikation.

Zusätzlich bietet U2TP Konstrukte an, welche das Einfügen testbezogener Aktionen ermöglichen. So erlaubt eine *ValidationAction* an beliebiger Stelle des Tests das Fällen eines Testurteils. Durch eine *LogAction* können Nachrichten einem möglicherweise vorhandenen Testprotokoll hinzugefügt werden. Eine *FinishAction* erlaubt die frühzeitige Terminierung eines Testfalls. Für die Definition von mit Testfällen verbundenen Testzielen stehen eigene Strukturelemente zur Verfügung welche in 2.4.5 näher erläutert werden.

2.4.4 Daten

Die Spezifikation von Testfällen beinhaltet fast immer auch die Spezifikation von Testdaten. Zum diesem Zweck stellt U2TP Konzepte für die Definition von Datenpools, Datenpartitionen und Datenselektoren an. *Datenpools* werden als Klassen in das Testmodell einbeschrieben und mit dem Stereotyp «DataPool» versehen. Ein solcher Datenpool fungiert als Container für explizit definierte Testdaten, welche beispielsweise in Testfällen verwendet werden können. Das für die Testfallerstellung wichtige Konzept von Äquivalenzklassen wird durch *Datenpartitionen* unterstützt. Diese werden als Klassen einbeschrieben und unter Verwendung des Stereotyps «DataPartition» gekennzeichent. Eine Datenpartition repräsentiert dabei direkt eine Äquivalenzklasse. Datenpartitionen müssen entweder mit einem Datenpool assoziiert werden oder mit einer (übergeordneten) Datenpartition. Eine Kaskadierung von Datenpartition bzw Äquivalenzklassen ist

also möglich. Für die Auswahl von Daten aus Datenpartitionen stehen *Datenselektoren* bereit. Ein Datenselektor ist eine mit dem Stereotyp «DataSelector» typisierte Operation, welche entweder einer Datenpartition oder einem Datenpool hinzugefügt werden darf. Ein Beispiel für die Spezifikation eines Datenpools ist in Abbildung 6.6 in Kapitel 6 zu sehen.

Zusätzlich zu den bereits vorgestellten Konzepten zu Testdaten erlaubt U2TP die Definition von *Kodierungsregeln* (*Coding Rules*), welche während der Kommunikation mit dem SUT verwendet werden können, um Daten zu kodieren oder zu dekodieren. Dazu wird der Stereotyp «CodingRule» bereitgestellt, welcher auf UML-Instanzen der Metaklassen **Property**, **ValueSpecification** oder **Namespace** angewendet werden kann. Somit besteht eine mögliche Anwendung darin, innerhalb von Testfällen einzelne Parameter, welche auch literale Wertdefinitionen sein können mit einer solchen Kodierungsregel zu versehen. Eine andere Möglichkeit ist beispielsweise die Anwendung des Stereotyps auf einzelne Felder einer Testdateninstanz. Letztere Variante resultiert darin, dass die Daten bei Verwendung stets auf die spezifizierte Weise kodiert werden.

2.4.5 Testmanagement

Auch für Zwecke des Testmanagements stellt U2TP Mittel zur Verfügung. Zur Unterstützung bei der Testplanung wird die Verwendung von *Testzielsetzungen* angeboten. Die Erzeugung von Testzielsetzungen erfolgt durch die Erstellung von UML-Klassen mit angewendetem Stereotyp «TestObjectiveSpecification». Dieser Stereotyp besitzt Eigenschaften, durch welche die Testziele näher spezifiziert werden können. Zwei zwingend erforderliche Eigenschaften sind *id*, um eindeutige Identifikatoren vergeben zu können und *specification*, welche für eine knappe und präzise Beschreibung der Testzielsetzung vorgesehen ist. Eine weitere, optionale Eigenschaft ist *priority*. Sie erlaubt das Vergeben von Prioritäten, die bei der Ablaufsteuerung der Tests genutzt werden kann, um die Reihenfolge der Testausführung zu optimieren.

Ein besonderes Merkmal von U2TP Testzielsetzungen ist die Möglichkeit, Beziehungen zu Anforderungen herstellen zu können. Dabei präsentiert sich U2TP sehr flexibel bezüglich der Art und Weise, wie Anforderungen im Einzelnen spezifiziert werden dürfen und es kann praktisch jede vorhandene UML-basierte Notation verwendet werden. Dies erlaubt nicht nur freie Wahl bei einer eigenen Vorgehensweise zur Definition von Anforderungen, sondern ermöglicht auch die Nutzung von UML-basierten Standards der Anforderungsspezifikation wie z.B. SysML[OMG10] oder der von UWE angebotenen Methode durch das Anforderungmodell aus [KK12]. Auch eine gleichzeitige Nutzung von Anforderungen aus verschiedenen Modellen mit unterschiedlicher Notation ist damit realisierbar, wie in Abbildung 2.6 dargestellt. Falls jedoch Modelltransformationen geplant sind, welche die Überführung der Modelle in eine Instanz eines vom UML Metamodell verschiedenen Metamodells vornehmen, so ist darauf zu achten, dass die Anforderungen bei diesem Prozess übernommen werden können. Eine einfach Möglichkeit der flexiblen Realisierung hierfür setzt voraus, dass die Instanzen von Anforderungen in Modellen der Metaklasse *NamedElement* zugehörig sind und der Name der Instanz gleich dem Identifikator gesetzt wurde. Auf diese Weise kann auf eine Transformation des Anforderungsmodells verzichtet werden und stattdessen über die «trace»-Beziehung der Name einer Anforderungsdefinition in die transformierte Instanz der Testzielsetzung übernommen werden.

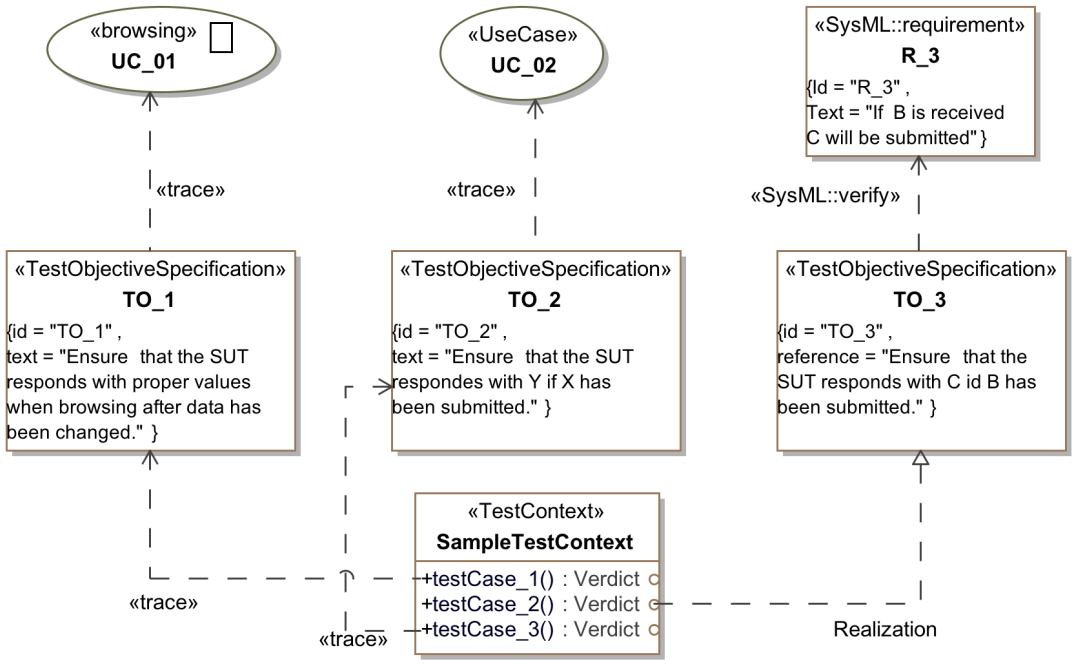


Abbildung 2.6: *Rückverfolgbarkeit von Anforderungen mit unterschiedlichen Notationen der Anforderungsspezifikation.*

2.4.6 Zeitbasierte Tests

Das UML Testing Profile stellt auch Konzepte zur Verfügung, um Abläufe nach zeitlichen Kriterien zu testen. So kann es z.B. in einem Projekt gefordert sein, einen bestimmten Wartezeitraum einzuhalten und diesen im Test zu überprüfen. Ein anderer möglicher Fall wäre eine aus den Anforderungen hervorgehende nicht-funktionale Anforderung, die verlangt, dass die Reaktionszeit des Systems stets unter einer gewissen Schwelle liegen muss (z.B. maximal 2 Sekunden Reaktionszeit auf eine Benutzeranfrage). Für diese Fälle stellt U2TP optionale Konzepte für Zeitpunkte und Zeiträume zur Verfügung. Sie können mit Konstrukten, welche die UML seit der Version 2.0 anbietet, in die Testabläufe einbeschrieben werden und die Testspezifikation so um zeitliche Aspekte ergänzen.

2.4.7 Mappings zu JUnit und TTCN-3

Das UML Testing Profile beschreibt Relationen zu JUnit und der Testspezifikationssprache TTCN-3. Sie erlauben einen Überblick über mögliche Äquivalenzen verwendeter Konzepte und geben aufschlussreiche Hinweise dazu, wie ein Vorgehen bei der Erstellung von Testsystemen auf Basis von U2TP-Testmodellen erfolgen kann. Zur Generierung von Testfällen für den Standard TTCN-3, herausgegeben von ETSI/ITU-T wurden zudem Ansätze in [ZDSD05] erforscht, welche Regeln für eine mögliche Transformation präsentieren. Von einer vollständigen Generierung von Testfällen ist bisher nichts bekannt, jedoch exisitert eine Unterstützung zur *Skeleton*-Generierung von TTCN-3 Tests, welche anschließend manuell komplettiert werden müssen. Die von U2TP präsentierten Tabellen können Aufgrund des Umfangs hier nicht wiedergegeben werden. Der geneigte Leser sei daher hier höflich auf Anhang C der U2TP-Spezifikation[UML13] hingewiesen.

2.4.8 U2TP Metamodell

Bis zur Version 1.1 wurde mit der Spezifikation des U2TP auch ein zugehöriges, MOF-basiertes Metamodell angeboten. In der aktuell vorliegenden Version 1.2 wurde eine Fortsetzung dessen aus nicht näher genannten Gründen ausgesetzt. Dennoch ist in der aktuellen Version ein Hinweis vorhanden mit einigen Informationen, die sich für das Verständnis der Struktur und beim Entwurf eines Metamodells mit U2TP als nützlich erweisen und eine Grundlage für die Erstellung eines Metamodells im weiteren Verlauf dieser Arbeit bildeten. Die Strukturelemente dieses Metamodells sind in Abbildung 2.7 gezeigt.

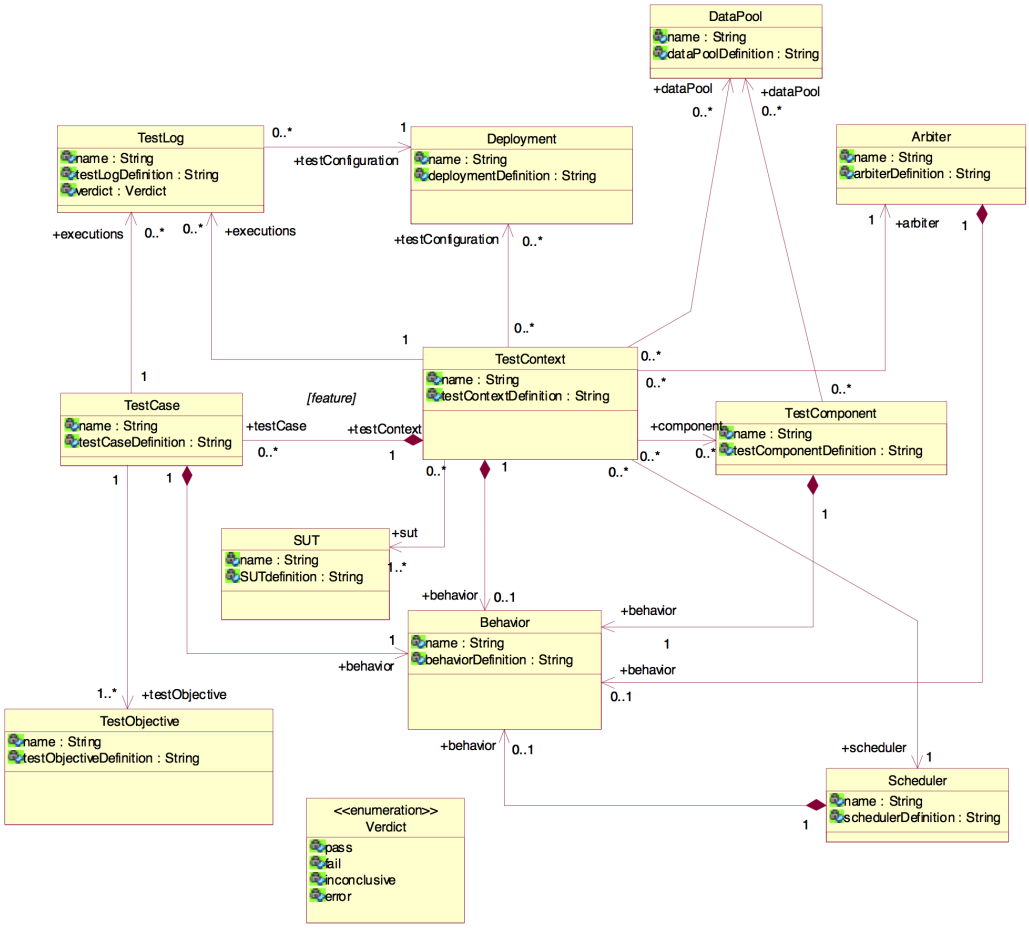


Abbildung 2.7: *U2TP Metamodell (bis Version 1.1) aus: [UML13]*

2.5 UML-Based Web Engineering

2.5.1 Einordnung

UML-Based Web Engineering[Koc01],[uwe13] (Kurzschreibweise UWE) ist ein Softwareentwicklungsansatz zur systematischen Entwicklung von Websystemen. UWE verfolgt dabei einen modellgetriebenen Ansatz, der auf die Erstellung von Systemen der Domäne Web spezialisiert ist und auf plattformunabhängiger Modellierung basiert. Der Ansatz hinter UWE verfolgt die Trennung der Aspekte Inhalte, Navigation, Präsentation, Prozesse und Benutzersitzung. Aufbauend auf diesen Konzepten ist eine Modellarchitektur aus korrespondierenden Teilmodellen entstanden, deren Struktur in Abbildung 2.8 dargestellt ist.

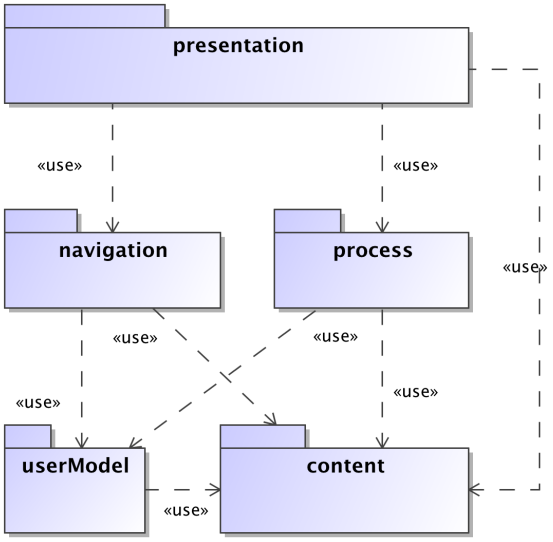


Abbildung 2.8: *UWE Modellarchitektur*

Als Grundlage für den Entwicklungsprozess steht eine UML-Profilerweiterung zur Verfügung, welche zur Spezifikation genutzt werden kann, sowie ein Metamodell und dazugehörige Modelltransformationen[KKK11]. Neben der Werkzeugunterstützung zur Modelltransformation existieren auch Verfahren zur Generierung von Anwendungen [Kro08], [KKK09] und ein MagicDraw-Plugin[BK09] zur Unterstützung bei der Modellierung.

2.5.2 Zusätzliche Nutzung von MVEL in dieser Arbeit

Die Sprache MVFLEX Expression Language[mve13] (MVEL) ist eine Sprache, welche für den Zugriff auf Eigenschaften von Java-Objekten genutzt werden kann. Das von MVEL verfolgte Konzept besteht darin, auf Java-Objekte durch Ausdrücke in der Sprache MVEL zugreifen zu können. Dabei wird lesender und schreibender Zugriff unterstützt. Eine Auswertung von MVEL-Ausdrücken kann erfolgen, indem ein Java-Objekt und ein MVEL-Ausdruck an einen Interpreter übergeben werden. Auf diese Weise können MVEL-Formeln über Java-Objekten ausgewertet werden. MVEL bietet eine Mischung aus dynamischer und statischer Typisierung und unterstützt zusätzliche Konzepte wie beispielsweise List Comprehension.

Kapitel 3

Testen von UWE Webanwendungen mit U2TP

3.1 Testen mit plattformunabhängigen Modellen

Im Fokus der vorliegenden Arbeit liegt das Blackbox-Testen von Websystemen auf Basis von plattformunabhängigen Spezifikationen. Ein Konzept der Modellierung auf plattformunabhängiger Ebene ist es, plattformspezifische Details auszulassen. Ausführbare Unit-Tests, die einzelne Systemartefakte innerhalb der Systemgrenzen in isolierter Weise testen, sind daher auf Basis solcher Modelle zunächst nicht realisierbar. Mögliche Ansatzflächen für Testing beschränken sich in der Regel auf vorhandene Schnittstellen an den Systemgrenzen.

Dennoch bietet dieser Ansatz beim Testen von Websystemen spezifische Stärken. Websysteme besitzen an der Systemgrenze standardisierte Schnittstellen, die sich aus der Warte des Blackbox-Testings unabhängig von der plattformspezifischen Realisierung testen lassen. Durch die domänenspezifische Ausprägung des Ansatzes hinter UWE sind die Konzepte des Web bei der Nutzung von plattformunabhängiger Modellierung nutzbar. Auf dieser Basis ist in Verbindung mit U2TP möglich, eine Methode zu entwickeln, um Testfälle zu spezifizieren, die genug Aussagekraft besitzen um eine Verifikation von Websystemen durch Blackbox-Tests durchzuführen. Eine Ausführung dieser Testfälle ist dabei ohne Bindung an die Zielplattform des konkret zu testenden SUT möglich. Kann die Ausführbarkeit dieser Tests für eine Plattform erreicht werden, so können beliebige Zielplattformen mit einem einzigen Testsystem getestet werden.

Zwar ist es aus der Perspektive der MDA mit Erweiterung um Testmodelle möglich, aus plattformunabhängigen Testmodellen (PIT) plattformspezifische Testmodelle (PST) abzuleiten und die darin enthaltenen, plattformspezifischen Informationen zur Erzeugung von Testsystemen zu verwenden, dies ist jedoch ein anderer Ansatz als der in dieser Arbeit verfolgte.

3.2 UWE Testmodell mit U2TP

Durch die Sprachverwandschaft über die UML bilden UWE und U2TP eine breite Basis für modellgetriebenes Testen. Damit die Konzepte des U2TP im UWE Entwicklungsprozess angewendet werden können, ist die Erweiterung der für UWE zur Verfügung stehenden Modelle um ein Testmodell erforderlich. Dazu wird ein neuer Stereotyp «testingModel» eingeführt, der auf die UML Metaklasse *Model* angewendet wird und die

Definition des Testmodells übernimmt. Die Konzepte des UML 2.0 Testing Profile können nun genutzt werden, um innerhalb des neu gewonnenen Testmodells Testumgebungen zu spezifizieren.

Die bisherigen Teilmodelle der UWE Modellhierarchie befassen sich mit der Spezifikation des Systemmodells einer Anwendung. Sie sind Systemmodelle und werden im Verlauf der Arbeit auch als *UWE Systemmodell* bezeichnet, womit die Summe der systemspezifischen Teilmodelle gemeint ist. Neben dem nun konzeptionell klar trennbaren UWE System- und Testmodell kann theoretisch auch ein Anforderungsmodell vorliegen. In diesem Fall besteht die Möglichkeit, die von U2TP angebotenen Mechanismen zur Rückverfolgbarkeit von Anforderungen einzusetzen. Im Verlauf der Arbeit wird zusätzlich zum Vorhandensein eines UWE System- und Testmodells von der Existenz eines Anforderungsmodells ausgegangen. Um die zur Verfügung stehenden Mittel bei der Testsystemmodellierung voll auszuschöpfen, wird Zugriff auf die Inhalte der Anforderungs- und Systemmodelle aus dem Testmodell heraus benötigt. Eine mögliche Strukturierung von System-, Test- und Anforderungsmodell mit den für die Testmodellierung relevanten Beziehungen ist in Abbildung 3.1 dargestellt.

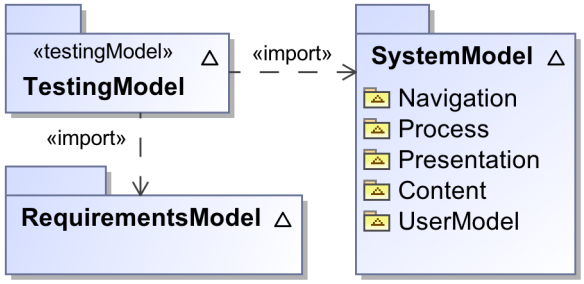


Abbildung 3.1: *Strukturierung der Teilmodelle für System, Testsystem und Anforderungen*

Innerhalb des Testmodells können nun die Konzepte des U2TP in vollem Umfang genutzt werden, um Testumgebungen zu definieren, Testfälle unter Einbeziehung des UWE Systemmodells zu erstellen und die Testfälle über U2TP Testzielsetzungen mit den Anforderungen in Beziehung zu setzen. Es sei an dieser Stelle noch erwähnt, dass das Vorhandensein eines Anforderungsmodells keine Voraussetzung für die Testmodellierung darstellt. Lediglich die in Abschnitt 2.4.5 erwähnten, zusätzlichen Möglichkeiten des Testmanagements zur Rückverfolgbarkeit von Anforderungen in Tests setzen das Vorhandensein einer entsprechenden Anforderungsspezifikation voraus.

3.3 Testing unter Nutzung des UWE Systemmodells

Ziel beim Blackbox-Testen auf Basis einer UWE Spezifikation ist in dieser Arbeit die Verifikation eines Websystems anhand der in der Spezifikation enthaltenen Teilmodelle bzw der darin enthaltenen Elemente. Die bei Blackbox-Tests analysierbaren Antworten eines Websystems sind HTML-Dokumente. Sie sind Bestandteil der Nachrichten, die mit dem SUT ausgetauscht werden und bilden die Grundlage für unser Vorhaben.

Dadurch ist es zunächst möglich, die in den empfangenen HTML-Dokumenten enthaltene Oberfläche unter Heranziehung der Oberflächenspezifikation eines UWE Präsentationsmodells zu verifizieren. Eine wichtige Voraussetzung dafür ist die Möglichkeit, Vorkommen von HTML-Elementen eindeutig den korrespondierenden Instanzen

der Präsentationsspezifikation zuzuordnen. Da diese Voraussetzung mit dem verwendeten Modellierungsansatz nicht immer aus sich heraus gegeben ist, wurde hierzu eine Lösung erdacht, die in dem auf diesen folgenden Abschnitt 3.4 beschrieben wird. Dies vorausgesetzt kann nicht nur das Vorhandensein von Elementen der Präsentationsspezifikation überprüft werden, sondern auch die Existenz erwarteter Information darin, z.B. über Textvergleiche. Zusätzlich dazu ist es möglich, HTML-Eingabefelder gezielt auszulesen oder mit Texteingaben zu befüllen und Formulare abzusenden oder Links zu aktivieren.

Dieser Umstand eröffnet Möglichkeiten zur Navigation innerhalb der vom SUT ausgelieferten Ansichten anhand des Navigationsmodells und ermöglicht die Prüfung der Zuordnung von Ansichten des Präsentationsmodells zu den spezifizierten Navigationsklassen des Navigationsmodells. Dadurch ist es möglich, die Kongruenz zwischen Navigations- und Präsentationsmodell sicherzustellen. Letzteres ist in eingeschränktem Maße auch durch einen rein systemmodell-getriebenen Ansatz durchführbar[Som13]. Die Möglichkeiten, die sich aufgrund des system- und testmodell-getriebenen Ansatzes in dieser Arbeit eröffnen, gehen jedoch darüber hinaus. Durch das Befüllen und Absenden von Eingabeformularen ergeben sich durch Einsatz eines Testmodells zusätzliche Möglichkeiten um einzelne Ausführungspfade von UWE Prozessen in gezielter Weise zu durchlaufen und etwaige Antworten mit erwartetem Verhalten zu vergleichen. Testbare Ausführungspfade von UWE Prozessen sind in diesem Fall solche, die in Abhängigkeit von Benutzereingaben ein bestimmtes beobachtbares Verhalten zeigen. Dies stellt eine brauchbare Grundlage für die Verifikation des Prozessmodells dar und erlaubt auch eine mindestens partielle Verifikation des Inhaltsmodells und des Benutzermodells.

Eine alternative Quelle für Verifikationsmöglichkeiten beim Blackbox-Testing von Websystemen ergibt sich bei Betrachtung der Kommunikation auf Ebene des Protokolls HTTP. So ist es grundsätzlich auch möglich, nur durch HTTP-Anfragen das Vorhandensein von bestimmten Inhalten zu verifizieren, Formulare abzusenden oder vorgeschriebene Reaktionszeiten zu überprüfen. Auch eine Kombination der Testebenen HTML und HTTP kann zur Erweiterung der Verifikationsmöglichkeiten sinnvoll sein. Die in dieser Arbeit vorgestellte Methode beschränkt sich jedoch auf den treiberbasierten Ansatz (siehe 2.1.4) und lässt HTTP-spezifische Vorgehensweisen soweit möglich ausser Acht.

3.4 Identifikation von Obeflächenelementen

Zum Zweck der Handhabung von Benutzeroberflächen stellt U2TP keine konkreten Mittel zur Verfügung. Auch die in[Bak09] gebotenen Überlegungen für das Testen von Benutzerschnittstellen sind allgemein gehalten und nicht ausreichend, um hinreichende Konzepte für die Handhabung von Weboberflächen zu liefern, so dass eine spätere Ausführbarkeit von generierten Testfällen ermöglicht würde.

Durch die strukturelle Semantik der mit UWE spezifizierten Oberflächen ist es bereits möglich, Aussagen über eine mögliche Konformanz von spezifizierten und beim Test vorgefundenen Ansichten zu treffen. Ein möglicher Ansatz zur Verifikation der Oberflächenstruktur, der durch eine heuristische Vorgehensweise untersucht, ob die vorgefundene HTML-Struktur eine gültige Implementierung einer konkreten Ansicht der Oberflächenspezifikation sein könnte, wurde in [Som13] vorgestellt. Ein Problem, welches der darin gebotene Ansatz nicht vollständig lösen konnte, war die zweifelsfreie Zuordnung von spezifizierten Oberflächenelementen zu den korrespondierenden Vorkommen in den Antworten des SUT. Dies liegt unter anderem darin begründet, dass

die in Frage kommenden, identitätsstiftenden Merkmale für HTML-Oberflächen wie z.B. HTML-Attribute oder CSS-Klassen dem Bereich der implementierungsspezifischen Details zuzuordnen sind und auf der Ebene der plattformunabhängigen Modellierung zunächst keine Relevanz haben.

Die modellierbare Eigenschaft *ID* des UWE Metamodell-Elements *UIElement* ist zwar zum Zweck der Identifizierung einzelner Oberflächenelemente konzipiert, da jedoch im UWE Entwicklungsprozess keine speziellen Vorgaben zur Implementierungsweise gemacht werden, gibt es keine eindeutige Aussage darüber, ob und wenn ja wie diese Eigenschaft im Zuge einer Implementierung umzusetzen ist. Sie stellt daher keine verlässliche Lösung für unser Vorhaben dar. Auch die Heranziehung der modellierbaren Eigenschaft *styleClass* führt zu keinem befriedigenden Ergebnis. Sie stellt zwar eine Eigenschaft dar, für die eine klare Vorgehensweise bei der Implementierung vorgesehen ist, dient jedoch ausdrücklich dem Zweck der optischen Ausgestaltung der Oberfläche[KKK11]. Bei der damit verbundenen Vorgehensweise wird in der Regel auf die Wiederverwendbarkeit von Stilklassen Wert gelegt. Daher kann damit gerechnet werden, dass *eine* Stilklasse auf *mehrere* Oberflächenelemente angewendet wird. Hier wäre eine tiefgreifendere Umdeutung des Verwendungszwecks notwendig, welche die ursprünglich intendierte Funktion ad absurdum führen würde. Es bleibt also bei dem Fazit, dass ohne zusätzliche Implementierungsvorschriften eine eindeutige Identifikation von Oberflächenelementen mit den zur Verfügung stehenden Mitteln nicht verlässlich realisierbar ist.

Die in dieser Arbeit verwendete Lösung des Problems der eindeutigen Identifikation von Oberflächenelementen erfordert wenige manuelle Schritte, die vom Tester vorzunehmen sind. Die Vorgehensweise dazu lässt sich in drei Schritten beschreiben:

1. Erstelle eine Liste aller Oberflächenelemente der UWE Präsentationsspezifikation
2. Finde für jedes Element ein Unterscheidungsmerkmal oder füge dem SUT eines hinzu
3. Trage die Unterscheidungsmerkmale in die Liste ein

Schritt (1) lässt sich durch eine triviale Model-to-Text-Transformation automatisieren. So ist es ohne weiteres möglich, eine Vorlage für eine maschinenlesbare Konfigurationsdatei zu erzeugen, welche mit den anschließenden Ergebnissen komplettiert und zu Beginn der Testausführung vom Testsystem eingelesen wird. Schritt (2) lässt sich realisieren durch Nutzung von CSS-Klassen in der Implementierung, welche eindeutige Unterscheidbarkeit gewährleisten. Schritt (3): ist klar.

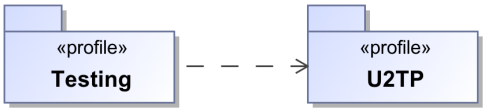
Für jedes Element der Oberflächenspezifikation werden so Merkmale festgelegt, welche eine eindeutige Identifizierung erlauben. Es wird dabei auch davon ausgegangen, dass es durchführbar ist, CSS-Klassen in die Ansichtem des Systems einzufügen, um zusätzliche Unterscheidungsmöglichkeiten herzustellen, falls dies in Schritt (2) erforderlich sein sollte.

3.5 UWE Testing Profile

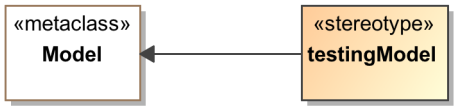
Das UWE Testing Profile ist ein UML-Profil, welches als Ergänzung zum UWE Profil Version 1.7 konzipiert ist. Es stellt die in dieser Arbeit propagierten Testkonzepte für den UWE Entwicklungsprozess zur Verfügung und erweitert die Möglichkeiten des UML2.0 Testing Profile. Dabei enthält es auch Profilelemente, welche eine direkte Spezialisierung von Elementen des U2TP vornehmen. Daher ergibt sich eine natürliche Abhängigkeit auf das U2TP wie in Abbildung 3.2(a) dargestellt ist. Aufgrund dieser Abhängigkeit wurde die Bereitstellung als sparates Profil erdacht, so dass eine direkte Einbettung in das UWE Profil, obwohl ohne weiteres möglich, nicht erforderlich ist. Im Folgenden werden die Profilelemente des UWE Testing Profile behandelt. Für eine Referenz des UML2.0 Testing Profile sei an dieser Stelle auf [UML13] verwiesen.

3.5.1 UWE Testmodell

Grundlage für die Testmodellierung mit dem UWE Testing Profile ist das Testmodell. Hierfür wird der Stereotyp «testingModel» eingeführt, welcher die UML Metaklasse *Model* erweitert, wie dargestellt in Abbildung 3.2(b).



(a) Abhängigkeit des UWE Testprofils auf das UML 2.0 Testing Profile



(b) Stereotyp testingModel als Erweiterung der UML Metaklasse Model

Abbildung 3.2: *Linke Abbildung: Abhängigkeit des UWE Testing Profile von U2TP Rechte Abbildung: Stereotyp testingModel des UWE Testing Profile*

Als Voraussetzung für die Erstellung eines UWE Testmodells sind für das im Entwicklungsprozess dafür vorgesehene Modell zunächst die Schritte für die Nutzung eines UML-Profils erforderlich. Um nach erfolgter Profilanwendung ein UWE Testmodell zu erstellen, kann im Modell ein neues UML-Element vom Typ *Model* angelegt und mit dem Stereotyp «testingModel» des UWE Testing Profile versehen werden. Damit ist die Erstellung des Testmodells abgeschlossen. Durch die oben beschriebene Abhängigkeit stehen damit auch alle Konzepte des U2TP zur Verfügung. Da sich dieses neue Modell durch seinen Testcharakter deutlich von den bestehenden UWE Systemmodellen absetzt, können diese nun als Systemmodell zusammengefasst werden. Systemmodell, Testmodell und - falls vorhanden - Anforderunsmodell können so als nebeneinander koexistierend in das Modellartefakt einbeschrieben werden. Hier sei nochmals auf Abbildung 3.1 hingewiesen, welche den Zusammenhang zwischen involvierten Modellen und Teilmodellen veranschaulicht.

3.5.2 Schnittstelle UTSL

Das UWE Testing Profile stellt eine Schnittstelle namens UTSL zur Verfügung, welche die Sprachelemente der in Abschnitt 4.1 vorgestellten domänenspezifischen Sprache *UWE Test Specification Language* in Form einer wohldefinierten Signatur zur Verfügung stellt. In der Signatur steht für jeden Befehl der Sprache eine gleichnamige Operation zur Verfügung. Auf Grundlage dieser Schnittstelle erlauben die Mittel der UML

die Modellierung von Abläufen der von UTSL zur Verfügung gestellten Testaktionen durch Nutzung von Sequenzdiagrammen. Für den Zweck der Testmodellierung bietet die nachfolgend erläuterte U2TP Testkomponente diese Schnittstelle an.

3.5.3 TestDriver

Ebenfalls vom UWE Testing Profile zur Verfügung gestellt wird ein Testtreiber, der *TestDriver*. Dabei handelt es sich um eine Softwarekomponente, welche über die Funktionalität eines skriptgesteuerten Web-Browsers verfügt und durch Abfolgen von UTSL-Befehlen Testfälle ausführen kann. Dadurch lassen sich Abfolgen von spezifizierten Benutzerinteraktionen ausführen sowie Auswertungen vornehmen, welche zu Testurteilen führen. *TestDriver* bietet die Schnittstelle UTSL an, um die programmierbarkeit von Webtests zu ermöglichen.

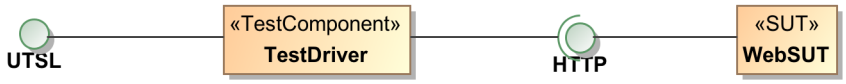


Abbildung 3.3: *TestDriver als U2TP Testkomponente*

3.5.4 StateMapping

Der Stereotyp «StateMapping» erweitert die UML Metaklasse *State* und besitzt die Eigenschaft *navigationNode*. Durch das Hilfskonstrukt des StateMapping ist es möglich, in Sequenzdiagrammen auf der Lebenslinie des Testtreibers Zustandsinvarianten zu modellieren, welche die Zusicherung auf eine UWE *NavigationClass* vornehmen. Das Vorkommen einer solchen Zusicherung besagt während der Ausführung des Tests, dass die vorliegende Oberflächenansicht des SUT als die Ansicht der mittels *navigationNode* spezifizierten NavigationClass verifiziert werden kann.

Die genaue Vorgehensweise bei der Testmodellierung wird in Abschnitt 4.3.2 beschrieben. Abbildung 3.4 zeigt ein Beispiel für einen so stereotypisierten Zustand mit Zuordnung zu einer NavigationClass durch die Eigenschaft *navigationNode*.

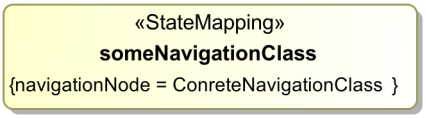


Abbildung 3.4: *Beispiel für eine Anwendung des Stereotyps StateMapping*

Kapitel 4

Domänenspezifische Erweiterung für U2TP

Die in dieser Arbeit vorgestellte Methode verwendet einen treiberbasierten Ansatz zur Spezifikation und Ausführung von Webtests. Der Tester bekommt dadurch Zugriff auf das Dokumentenmodell und kann Benutzerinteraktionen nachstellen sowie Testurteile auf Basis von Auswertungen des Dokumentenmodells fällen. Die Ebene der HTTP-Kommunikation bleibt dabei weitgehend verborgen. Die von U2TP gebotenen Mittel der Testverhaltensspezifikation sind allgemein gehalten und bieten keine geeigneten Konzepte für das Testing von Webanwendungen. Daher wird im Folgenden die domänenspezifische Sprache UTSL vorgestellt, welche zur Erweiterung des Funktionsumfangs von U2TP dient und die Beschreibung von Testverhalten im Web auf plattformunabhägige Weise ermöglicht. Sie soll die Sichtweise des Testdesigners und des Systemdesigners zusammenführen und dazu dienen, Testaktionen im Web beschreiben zu können, Zustände des Dokumentenmodells unter Verwendung des UWE Systemmodells und MVEL-Ausdrücken auszuwerten und für die Testautomatisierung relevante Aktionen durchzuführen.

4.1 DSL für Webtests: UTSL

UTSL weist Ähnlichkeiten mit den Aktionen bestehender Frameworks zum Testen von Webanwendungen auf. Ein Unterschied liegt jedoch in der Art und Weise, wie Elemente innerhalb eines HTML-Dokuments lokalisiert werden. So kommen bei Selenium oder HtmlUnit typischerweise Selektoren zum Einsatz, welche die Lokalisierung von Elementen auf HTML-syntaktischer Basis vornehmen[sel13], [htm13]. Hier erfolgt dies über den in Abschnitt 3.4 erwähnten Mechanismus der Adressierung über Elemente des Systemmodells unter der zusätzlichen Verwendung von MVEL-Ausdrücken.

4.1.1 Webaktionen

Dieser Abschnitt erläutert die Sprachelemente von UTSL, welche zur Formulierung von Webaktionen dienen. Innerhalb eines Dokumentenmodells stehen die typischen Elemente des Web zur Verfügung und stellen Ansatzpunkte für Testdesigner zur Verfügung. Als mögliche Aktionen innerhalb eines automatisierten Testfalls können z.B. Formulare befüllt und abgesendet werden oder HTML-Links aktiviert und verfolgt werden.

Die elementarste Aktion, die den im Web intendierten Benutzeraktionen nachempfunden ist, ist **click**. Sie wird mit einer MVEL-Referenz auf ein Element des System-

modells aufgerufen und respräsentiert einen Mausklick auf das entsprechende Element. Wird nach einer beliebigen Aktion das Absenden einer HTTP-Anfrage mit verbundenem Nachladen des Dokumentenmodells oder Teilen davon erwartet, so muss dies durch einen anschließenden, expliziten Wartebefehl an der entsprechenden Stelle im Ablauf ausgedrückt werden. Ein solcher Befehl, der das Warten auf die Antwort des SUT bedeutet, kann an verschiedenen Stellen notwendig sein, um den erwarteten Neuaufbau des Dokumentenmodells nach einem Seitenübergang abzuwarten, da sonst die ordnungsgemäße Fortführung des Testfalls aufgrund noch nicht vorhandener Elemente eventuell verhindert wird. UTSL bietet dafür Befehle in unterschiedlicher Ausprägung an. Wird das Nachladen des gesamten Dokumentenmodells aufgrund des Ladens einer neuen Seite erwartet, so kann dies mit dem Steuerbefehl **waitForPageToLoad** ausgedrückt werden. Für Fälle, in denen nicht das Nachladen des gesamten Dokumentenmodells, sondern nur eines Teils davon erwartet wird (beispielsweise mittels AJAX), steht der Befehl **waitForElementPresent** zur Verfügung, der ebenfalls mit einer MVEL-Referenz zu einer Instanz des UWE Präsentationsmodells aufgerufen wird und wartet, bis das entsprechende Element im Dokumentenmodell identifizierbar ist. Für den häufigen Fall, dass nach einem Benutzerklick ein Seitenübergang mit verbundenem Nachladen des gesamten Dokumentenmodells erwartet wird, steht die Kombination beider Vorgänge in Form des Befehls **clickAndWait** zur Verfügung.

Mit dem Befehl **sendKeys** können Zeichenketten an HTML-Elemente gesendet werden. Auf diese Weise können z.B. Texteingaben in Eingabefeldern vorgenommen werden. Dazu erwartet **sendKeys** zwei Argumente: eine MVEL-Referenz auf das Eingabeelement und eine MVEL-Referenz auf ein Instanzattribut des Datenpools, welches den einzugebenden Text darstellt. Umgekehrt kann mit **getValue** der Inhalt eines Textfeldes ausgelesen und zurückgegeben werden. Mittels **select** bzw **deselect** ist es möglich, in Auswahlfeldern, welche im HTML-Code durch die HTML-Elemente `<select>` und `<option>` repräsentiert werden, einzelne Optionen auszuwählen bzw abzuwählen. Als Argument erwarten **select** bzw **deselect** eine MVEL-Referenz auf die jeweilige Option. Für Auswahlfelder kann ebenso **getValue** verwendet werden, um den literalen Wert des Auswahlfeldes auszulesen. Dabei ist zu beachten, das zurückgelieferte Werte bei **getValue** immer Strings sind, also auch Zahlenwerte werden als Zeichenketten ausgelesen. Die letzte Aktion aus diesem Teil der Sprachbefehle ist **epsilon**. Sie steht für eine Aktion, mit der keine Ausführung verbunden ist. Eine solche Aktion kann bei der Modellierung beispielsweise nützlich sein, um die Auswertung einer Zustandsinvariante zu ermöglichen, welche sich am Ende eines Ablaufs befindet. Da die UML für die Semantik einer Zustandsinvariante vorschreibt, dass diese nur im Zusammenhang mit der Ausführung eines darauf folgenden Nachrichtenvorkommens ausgewertet wird, wird eine sich am Ende des Ablaufs befindliche Zustandsinvariante nicht ausgewertet. Dieser Umstand kann so durch Einfügen einer folgenden **epsilon**-Aktion kompensiert werden, ohne die eigentliche Semantik der Testverhaltensspezifikation zu beeinflussen. Tabelle 4.1 zeigt eine Übersicht der in diesem Abschnitt vorgestellten Sprachelemente. Dabei bezeichnet *Typ* den Typ des Rückgabewertes.

4.1.2 Zusicherungen und Auswertungen

Die nächste Gruppe von UTSL-Befehlen analysiert Elemente einer Seite auf Basis von ja/nein-Entscheidungen, wobei Instanzen des Systemmodells, welche in die Entscheidungen mit einbezogen werden, über MVEL-Audrücke adressiert werden. Jede Aktion dieser Gruppe stützt sich dabei auf eines von sechs zur Verfügung gestellten Prüfkrite-

Typ	Befehl	Parameter
void	click	uiElement
void	clickAndWait	uiElement
void	waitForPageToLoad	-
void	waitForElementPresent	uiElement
void	sendKeys	uiElement, text
void	select	uiOptionElement
String	getValue	uiElement
void	epsilon	-

Tabelle 4.1: *Sprachelemente von UTSL: Webaktionen*

rien. Ein Prüfkriterium kann beispielsweise die Existenz eines Oberflächenelements in der vom SUT ausgelieferten Ansicht sein. Tabelle 4.2 listet die zur Verfügung gestellten Prüfkriterien mit kurzen Erläuterungen auf.

Zu jedem Prüfkriterium stehen verschiedene Ausprägungen zur Verfügung, die als Zusicherungen und Auswertungen unterschieden werden. Zusicherungen brechen die Ausführung des Tests ab, wenn das Ergebnis der Auswertung des Prüfkriteriums negativ ausfällt. Ihre Semantik ist vergleichbar mit der von Zusicherungen (*Assertion*) in xUnit-artigen Testframeworks. Auswertungen hingegen liefern das Ergebnis der Überprüfung als boolschen Wert zurück, was eine mögliche Weiterverwendung innerhalb des Testfalls ermöglicht. Letzteres ist insofern nützlich, als es die Semantik von UML-Interaktionen erlaubt, Rückgabewerte von Methodenaufrufen in lokalen Variablen vorzuhalten. Dies erlaubt innerhalb von Tests die Weiterverarbeitung von Auswertungsergebnissen, beispielsweise für die Fällung von Testurteilen, welche von mehr als einem Prüfkriterium abhängig sind.

Zusätzlich zu den Ausprägungen der Testkriterien in Form von Zusicherungen und Auswertungen liegt für jedes Testkriterium auch eine Form für die Verneinung desselben vor. Auf diese Weise kann beispielsweise nicht nur die Existenz eines Elements verifiziert werden, sondern auch dessen Abwesenheit. Dies ist vor allem für die Zusicherungen notwendig, da eine Verneinung des Ergebnisses im Anschluss an die Auswertung hier nicht durchführbar ist. So ergeben sich durch die Kombination der sechs Prüfkriterien, zwei Auswertungsarten und der Verneinungsmöglichkeit pro Prüfkriterium insgesamt 24 verschiedene einsetzbare Ausprägungen von UTSL-Befehlen, auf deren ausführliche Auflistung hier verzichtet wird. Das Code-Listing in Abbildung 4.1 zeigt jedoch einige Beispiele für mögliche Ausprägungen.

```
1       assertElementPresent(uiElement);
2       assertNotElementPresent(uiElement);
3       c1 = verifyElementTextPresent(uiElement, text);
4       c2 = verifyNotElementTextPresent(uiElement, text);
5       assertNavigationClass(navigationClass);
6       c3 = verifyElementPresent(uiElement);
```

Abbildung 4.1: *Sprachelemente von UTSL: Beispiele für Zusicherungen und Auswertungen*

Kriterium	*Parameter*	*Erläuterung*
NavigationNode	node	die aktuelle Ansicht ist der NavigationClass *node* zugeordnet
PresentationPage	mvelPath	durch *mvelPath* adressierte PresentationPage ist die vorliegende Ansicht
TextPresent	text	gesuchter Text befindet sich an beliebiger Stelle auf der Seite
ElementPresent	mvelPath	durch MVEL-Ausdruck adressiertes Element befindet sich an beliebiger Stelle auf der Seite
ElementText	mvelPath, text	gesuchter Text ist exakt der Textinhalt des adressierten Elements
ElementTextPresent	mvelPath, text	gesuchter Text befindet sich an beliebiger Stelle innerhalb des adressierten Elements

Tabelle 4.2: *Testkriterien für Zusicherungen und Auswertungen*

4.1.3 Testaktionen

Die dritte und letzte Gruppe von UTSL-Befehlen stellt Mittel zur Ablaufsteuerung und Testsystematisierung bereit. Der Befehl **setVerdict** ist eine Aktion zur Fällung des Testurteils. Er stellt eine direkte Repräsentation der U2TP *ValidationAction* (siehe 2.4.3) dar und kann innerhalb eines Testfalls beliebig oft aufgerufen werden. Als Argument erwartet er das zu Fällende Testurteil, wie es von U2TP vorgeschrieben wird. Ebenso auf den Konzepten des U2TP beruhend sind die Befehle **log** und **finish**. Ersterer wird dazu verwendet, Einträge in ein während der Testausführung möglicherweise erstelltes Testprotokoll einzufügen. Die zu protokollierende Nachricht wird als Argument übergeben. Der Befehl **finish** empfängt keine Argumente und wird eingesetzt, um die Ausführung eines Testfalls vorzeitig zu beenden. Dies kann beispielsweise der Optimierung der Testausführung in umfangreicheren Testumgebungen dienen. Der letzte Befehl ist **resetBackend**. Dieser ist für zukünftige Fälle vorgesehen, falls während eines Tests der Inhalt der Datenbank des SUT auf den Anfangszustand zurückgesetzt werden soll. Tabelle 4.3 listet die Befehle dieses Abschnitts auf.

Typ	Befehl	Parameter
void	setVerdict	verdict
void	log	text
void	finish	
void	resetBackend	

Tabelle 4.3: *Sprachelemente von UTSL: Testaktionen*

4.1.4 Verwendung von MVEL-Referenzen in Argumenten

Zur Adressierung von Elementen des Systemmodells oder des Datenpools werden MVEL-Ausdrücke verwendet. Abbildung 4.2 zeigt das einfache Beispiel einer Zusicherung, welche die Existenz eines Eingabefelds aus dem Systemmodell überprüft:
Der MVEL-Term `SearchPage.queryInput` wird zur Laufzeit des Tests unter Nutzung des Systemmodells und der Zuordnungen für Oberflächenelemente ausgewertet. Dabei wird innerhalb des Systemmodells das Element `SearchPage` gesucht, welches in

```
1       assertElementPresent('SearchPage.queryInput')
```

Abbildung 4.2: *Verwendung von MVEL-Ausdrücken in Argumenten, hier am Beispiel einer Zusicherung.*

diesem Beispiel ein UWE Präsentations-Element vom Typ PresentationPage mit einem zugehörigen Texteingabefeld `queryInput`. Der Ausdruck `SearchPage.queryInput` bezeichnet dabei die Eigenschaft `queryInput` des Elements `SearchPage`, also konkret das Eingabefeld. Durch die erfolgten Zuordnungen zwischen Präsentationsmodell und Oberflächenelementen des SUT (siehe Abschnitt 3.4) kann so überprüft werden, ob das gesuchte Element in der Antwort des SUT auffindbar ist. Anschließend kann zusätzlich der Typ des Elementes überprüft werden, also beispielsweise ob es sich um ein Eingabefeld oder einen Link handelt.

MVEL-Ausdrücke zur Adressierung von Präsentationselementen können syntaktisch als Pfade innerhalb der als Baum betrachteten Struktur einer Seitenspezifikation verstanden werden. Dabei können sie theoretisch absolut oder relativ aufgebaut werden. Absolut aufgebaute Ausdrücke beginnen mit einer **PresentationPage** und enthalten, durch Punkte getrennt, jedes Element des Pfades von der PresentationPage hin zum final adressierten Element. Relative Ausdrücke beginnen mit der Instanz von **PresentationGroup** und werden von dort an analog aufgebaut.

MVEL-Audrücke in Argumenten können auch für Elemente des Testmodells verwendet werden, z.B. für den Zugriff auf Datenpools. Abbildung 4.3 zeigt ein einfaches Beispiel dazu. Der Ausdruck `dataPool.SampleProduct.productName` adressiert

```
1    type('SearchPage.queryInput','dataPool.SampleProduct.productName')
```

Abbildung 4.3: *Verwendung von MVEL-Ausdrücken in Argumenten, hier am Beispiel einer Texteingabe aus einem Datenpool.*

hierbei einen Datenpool, der mit der Bezeichnung *dataPool* spezifiziert wurde, entnimmt die darin enthaltene Objektinstanz *SampleProduct* und liefert den Wert des Instanzattributs *productName* zurück. Die Semantik des Ausdrucks in Abbildung 4.3 lässt sich also als die Eingabe des im Instanzattribut *productName* enthaltenen Werts in das über *SearchPage.queryInput* adressierte Eingabefeld beschreiben. Die Spezifikation von literalen Strings als Argumente ist theoretisch möglich, momentan jedoch nicht vorgesehen.

4.1.5 PresentationAlternative und IteratedPresentationGroup

Die Handhabung von Instanzen einer UWE *IteratedPresentationGroup* bedarf keiner zusätzlichen Sprachelemente und kann unter Nutzung von MVEL erfolgen. Dazu ist in Abbildung 4.4 eine Beispielkonstellation für eine Seitenansicht gezeigt.

Zu sehen ist die Seitendefinition durch eine *presentationPage* mit einer in *presentationAlternatives* enthaltenen *iteratedPresentationGroup*, welche zwei Textfelder enthält. In der nachfolgenden Abbildung 4.5 sind dazu drei MVEL-Ausdrücke abgebildet, die verschiedene Möglichkeiten zur Adressierung von Elementen aufzeigen. Der Ausdruck in Zeile 1 referenziert die *IteratedPresentationGroup*. In Zeile 2 wird durch zusätzliche Angabe des Index `0` die erste darin enthaltene Zeile adressiert. Der letzte Ausdruck referenziert das Textfeld *Field_1* in der ersten Zeile der *IteratedPresentationGroup*.

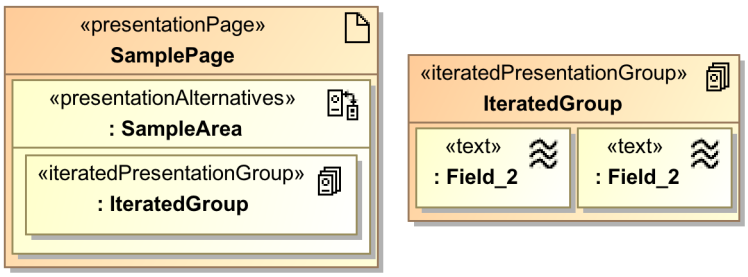


Abbildung 4.4: *Schaubeispiel für die Einbettung einer IteratedPresentionGroup*

```
1   verifyElementTextPresent(SamplePage.SampleArea.IteratedGroup);
2   verifyElementTextPresent(SamplePage.SampleArea.IteratedGroup[0]);
3   verifyElementTextPresent(SamplePage.SampleArea.IteratedGroup[0].Field_1);
```

Abbildung 4.5: *Verwendung von MVEL-Ausdrücken zur Adressierung von Elementen innerhalb einer IteratedPresentationGroup*

4.2 Integration mit U2TP

Das UML 2.0 Testing Profile motiviert die Spezialisierung und Erweiterung für domänenspezifische Einsatzbereiche[UML13]. Eine solche Erweiterung ist in unserem Fall zum Zweck der Einbettung von UTSL angebracht. Während der Ausführung der Tests sollen die mit UTSL spezifizierten Testfälle von einer zusätzlichen Komponente, unserem Testtreiber, ausgeführt werden. Für die Integration von zusätzlichen Softwarekomponenten, die aktiv oder passiv am Testgeschehen teilnehmen, bietet U2TP das Konzept der Testkomponente an. Dabei wird in der U2TP Spezifikation auch explizit die Möglichkeit erwähnt, dass die Stimulation des SUT von einer solchen Testkomponente vorgenommen wird[UML13, S. 21]. Aus diesem Grund gestaltet sich die Integration von UTSL relativ einfach.

Über das UWE Testing Profile wird der Testtreiber als U2TP Testkomponente *TestDriver* zur Verfügung gestellt. *TestDriver* ist durch Anwendung des U2TP Stereotyps «TestComponent» als U2TP Testkomponente entsprechend typisiert. Da in unserem Fall für die Definition des SUT innerhalb eines Testkontextes nicht direkt Elemente des Systemmodells verwendet werden können[1], wird ein dafür vorgesehenes SUT als *WebSUT* ebenfalls über das Profil zur Verfügung gestellt. Dieses SUT bietet die HTTP-Schnittstelle an, welche von *TestDriver* zum Zweck der Teststimulation konsumiert wird. Auf diese Weise lässt sich der Sachverhalt, dass Testtreiber und SUT via HTTP kommunizieren, auf U2TP-konforme Weise durch Schnittstellen modellieren, wie in Abbildung 3.3 in Kapitel 3 zu sehen ist. *TestDriver* bietet die Schnittstelle *UTSL* an, dessen Signatur die Sprachelemente von UTSL enthält. Dies ermöglicht in UML Interaktionen die Ansprache des Testtreibers über die parametrisierbaren Methoden der Schnittstelle *UTSL*.

[1] Als Kandidat für ein SUT käme in unserem Fall ein UWE Systemmodell in Frage. Instanzen der UML-Metaklasse *Model* eignen sich jedoch nicht dafür, dem Strukturbereich einer Instanz der UML-Metaklasse *Class* als Eigenschaft hinzugefügt zu werden.

4.3 Testverhaltensspezifikation mit U2TP und UTSL

Der Einsatz von Sequenzdiagrammen scheint für den Zweck der Testfallspezifikation die meisten Möglichkeiten zu bieten. Für die Durchsetzung der U2TP-Konzepte ist wichtig, dass die U2TP Testaktionen *ValidationAction*, *LogAction* und *FinishAction* verfügbar und in den Testabläufen spezifizierbar sind. Sie können kompakt als parametrisierte Interaktionsfragmente realisiert werden. Alternativ könnten die U2TP-Elemente *Arbiter*, *Log* und *Scheduler* auch als Instanzen mit eigenen Lebenslinien repräsentiert werden. Dadurch könnten *ValidationAction* am Arbiter, *LogAction* am Log und *FinishAction* am Scheduler über Operationen verfügbar gemacht werden.

4.3.1 Testfallmodellierung mit Sequenzdiagrammen

Die am Test beteiligten Teilnehmer werden als Lebenslinien in das Sequenzdiagramm einbeschrieben. In unserem Fall ist es überflüssig, das SUT als eigene Lebenslinie zu instanziieren, da der modellierte Nachrichtenfluss innerhalb des Testfalls zwischen dem ausführenden Testkontext und dem Testtreiber stattfindet. Je nach Anforderung an den Testfall können zusätzliche Testkomponenten eingebunden werden, vorausgesetzt sie sind an den Testkontext angebunden. UTSL-Aktionen können durch Nachrichten an den Testtreiber spezifiziert werden. Fragmente können referenziert werden, um Teilabläufe von Testfällen zu modularisieren und an anderer Stelle wiederzuverwenden. Für Fallunterscheidungen können alternative oder optionale Fragmente herangezogen werden. Das Sequenzdiagramm *sampleTest* in Abbildung 4.6 zeigt beispielhaft die Grundzüge der Testfallmodellierung mit UTSL. Zusätzlich zu Testkontext und Testtreiber nimmt in diesem Beispiel eine fiktive Komponente *Validator* am Testgeschehen teil.

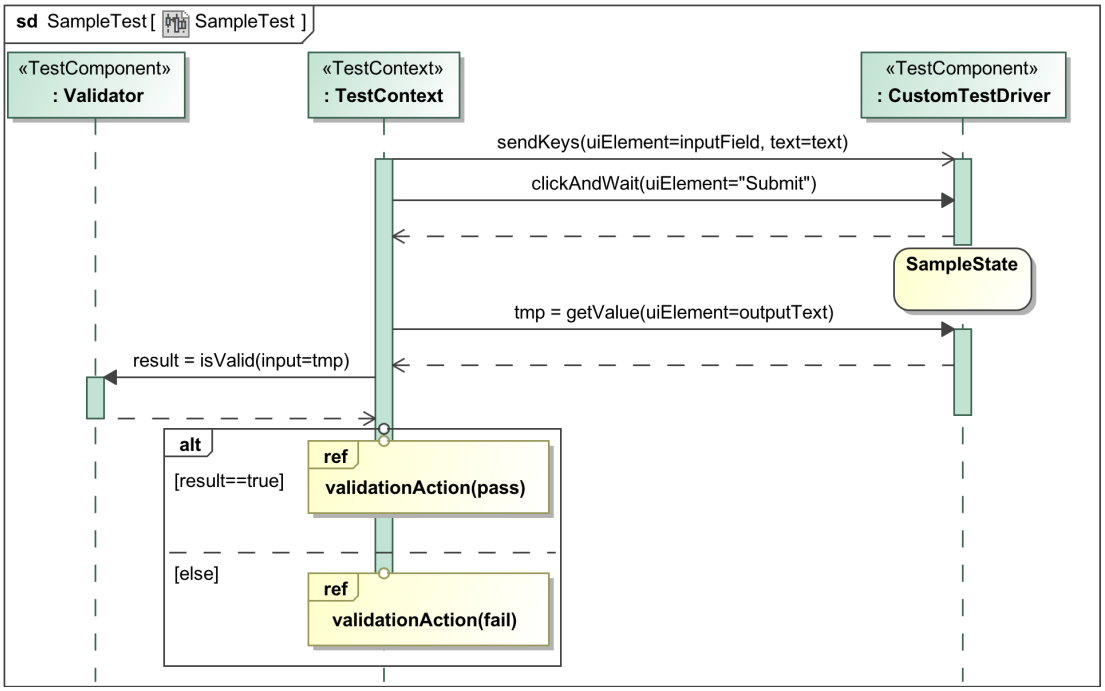


Abbildung 4.6: *Grundzüge der Testfallmodellierung mit UWE*

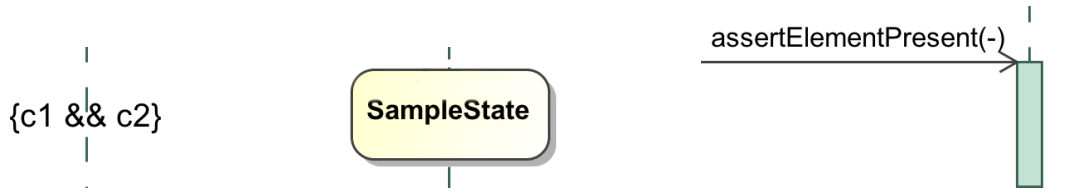
Die erste Nachricht spezifiziert das Vorkommen einer Tastatureingabe. Sie erfolgt asynchron und bedarf keiner Rückmeldung. Anschließend wird ein Formularbutton angeklickt und auf die Antwort des SUT, also auf das Nachladen gewartet. Durch eine

Zustandsinvariante (ausführlich erläutert im folgenden Abschnitt 4.3.2) wird sichergestellt, dass die vom SUT gelieferte Ansicht der in der Präsentationsspezifikation für den Navigationsknoten *SampleState* spezifizierten Ansicht entspricht. Nun wird aus dem durch `SampleState.message` spezifizierten Oberflächenelement der Textinhalt ausgelesen und in der Variable *tmp* abgelegt. Die Methode *isValid()* der zusätzlichen Testkomponente *Validator* wird nun von TestContext aufgerufen und *tmp* als Parameter übergeben. Das Ergebnis wird in *result* abgelegt. Zuletzt wird das Testurteil in Abhängigkeit von *result* gefällt.

4.3.2 Zusicherungen

Auf Grundlage der von der UML zur Verfügung gestellten Zusicherungen auf Lebenslinien in Sequenzdiagrammen und der durch UTSL und das UWE Testing Profile zusätzlichen Möglichkeiten ergeben sich drei Varianten der Modellierung von Zusicherungen in Testfällen. Durch die von der UML angebotenen Zusicherungen auf Lebenslinien lassen sich mit unserem Ansatz Zusicherungen unter Verwendung von MVEL-Formeln spezifizieren wie dargestellt in Abbildung 4.7(a). Diese sind auf der Lebenslinie des Testkontextes zu erstellen. Zur Formulierung der Invariante können zuvor erstellte Variablen genutzt werden, die beispielsweise durch eine vorhergenede UTSL-Auswertungen erzeugt wurden. Mittels Zustandsinvarianten können Zusicherungen zu Navigationszuständen vorgenommen werden. Damit Zustandsinvarianten bei der Testmodellierung genutzt werden können, ist es notwendig, einen für die zu testende Anwendung spezialisierten Testtreiber zu erstellen, welcher Pseudozustände zur anschließenden Nutzung als Zustandsinvarianten zur Verfügung stellt. Dazu muss im Testmodell eine neue Klasse erstellt werden, welche mit «TestComponent» als Testkomponente typisiert wird und eine Generalisierungsbeziehung hin zu dem *TestDriver* des UWE Testing Profile erhält. Anschließend kann innerhalb des spezialisierten Testtreibers eine UML Zustandsmaschine erstellt werden, um in einer darin enthaltenen *Region* Zustände zu erzeugen, welche mit dem Stereotyp «StateMapping» versehen werden. Über die dem Stereotyp zugehörige Eigenschaft *navigationClass* wird der Zustand der gewünschten Instanz einer *NavigationClass* des Navigationsmodells verknüpft. Anschließend kann eine Zustandsinvariante auf Lebenslinien des zuvor spezialisierten Testtreibers modelliert werden. Ein Beispiel für eine Zustandsinvariante ist in Abbildung 4.7(b) gezeigt.

Die von UTSL angebotenen Befehle für Zusicherungen ermöglichen zusätzlich die Erstellung von Zusicherungen durch UTSL-Aufrufe unter Einbeziehung des UWE Systemmodells. Ein Beispiel hierfür zeigt Abbildung 4.7(c).



(a) Zusicherung durch Invariate unter Verwendung von MVEL

(b) Zusicherung durch Zustandsinvariante

(c) Variante durch UTSL-Aufruf am Testtreiber

Abbildung 4.7: *Beispiele für Zusicherungsmöglichkeiten in Sequenzdiagrammen*

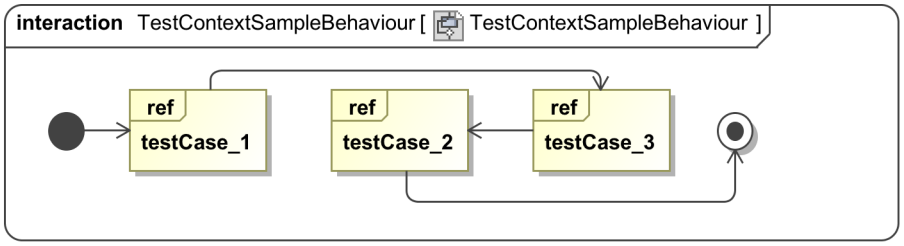


Abbildung 4.8: *Optionale Ablaufdefinition für Testsfälle*

4.3.3 Ablaufsteuerung von Testfällen

Die Ausführung von Testfällen eines Testkontextes wird hier als Testablaufsteuerung bzw Ablaufsteuerung bezeichnet. Sie kann durch eine Verhaltensdefinition des Testkontextes explizit definiert werden. Dazu können UML Interaktionsübersichtsdiagramme eingesetzt werden. Abbildung 4.8 zeigt ein Beispiel dafür. Andere Möglichkeiten, die Ablaufsteuerung explizit vorzunehmen, bestehen theoretisch darin, die Testfälle von außerhalb des TestContext aufzurufen[UML13] oder einen spezialisierten Scheduler zu erstellen. Wird keine explizite Ablaufsteuerung vorgenommen, so kann als Standardverhalten die Reihenfolge der Testfalldeklarationen im Testkontext herangezogen werden.

4.4 Vorgehen bei der Testmodellerstellung mit UTSL

Dieser Abschnitt beschreibt die Schritte, welche bei der Erstellung eines Testmodells durchgeführt werden. Dafür wird das Vorliegen eines UWE Systemmodells vorausgesetzt, sowie das Vorhandensein von im Vorfeld erarbeiteten Testzielsetzungen. Als optional wird das Vorliegen eines zusätzlichen Anforderungsmodells angenommen.

4.4.1 Definition von Testzielsetzungen

Für jede Testzielsetzung wird eine neue Klasse in das Testmodell einbeschreiben und der Stereotyp «TestObjectiveSpecification» des U2TP auf sie angewendet. Die zur Anwendung des Stereotyps zwingend erforderlichen Eigenschaften *id* und *specification* werden ebenfalls spezifiziert. Unter *id* wird der Identifikator der Testzielsetzung eingetragen und durch *specification* die korrespondierende Beschreibung notiert. Die optionale Eigenschaft *priority* kann spezifiziert werden, falls eine Optimierung der Testablaufsteuerung über Prioritäten gewünscht ist. Ist ein Anforderungsmodell vorhanden, so können über die UML «trace»-Beziehung die entsprechenden Zusammenhänge zu einzelnen Anforderungen hergestellt werden. In Abbildung 4.9 ist ein exemplarisches Beispiel einer Testzielsetzung mit Bezug auf eine Anforderung abgebildet.

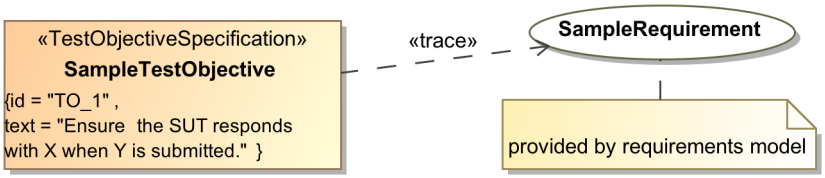


Abbildung 4.9: *Beispielzuordnung von Testzielsetzung und Anforderung*

4.4.2 Testkontext- und Testfallerstellung

Die Erstellung eines Testkontextes wird durch Erzeugung einer Klasse mit angewendetem Stereotyp «TestContext» begonnen. Zur Komplettierung eines minimalen Testkontexts werden die Instanzen von *WebSUT* und *TestDriver* aus dem UWE Testing Profile dem Strukturbereich des Testkontextes hinzugefügt, wie anhand des Beispiels in Abbildung 4.10 zu sehen ist. Um die spätere Nutzung von systemmodellspezifischen Zustandsinvarianten zu ermöglichen kann alternativ eine spezialisierte Form von *TestDriver* erstellt und um Zustände ergänzt werden, welche unter Verwendung des Stereotyps «StateMapping» in Beziehung zu Knoten des Navigationsmodells gesetzt werden können. In diesem Fall ist die Zuordnung dieses spezialisierten Testtreibers zum SUT unter Verwendung der Schnittstelle *HTTP* als zusätzlicher Schritt erforderlich.

Einzelne Testfälle werden erstellt, indem dem Testkontext neue Operationen hinzugefügt und der Stereotyp «TestCase» auf selbige angewendet wird. Die Signaturen der Operationen sind so zu erstellen, dass der Rückgabewert als Typ *Verdict* spezifiziert ist. Hier empfiehlt sich die Einhaltung der für Testfälle üblichen Namenskonventionen, den Namen des Testfalls auf das Suffix „Test“ enden zu lassen. Die Zuordnung von Testfällen zu Testzielsetzungen kann nun durch Einbeschreiben einer UML Beziehung vom Typ «trace» oder durche eine UML *Realization* vorgenommen werden. Das Beispiel in Abbildung 4.10 veranschaulicht dies unter Verwendung der letzteren Variante.

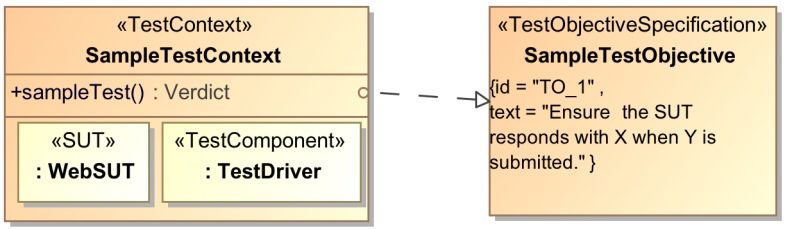


Abbildung 4.10: *Beispiel zur Modellierung von Testkontext und Testfall mit Bezug auf eine Testzielsetzung.*

4.4.3 Datenpools

Die Spezifikation eines Datenpools erfolgt durch Erstellung einer UML-Klasse mit anschließender Anwendung des Stereotyps «DataPool». Zusätzlich können Datenpartitionen und Datenselektoren erstellt werden. Die genaue Vorgehensweise hierfür wird ausführlich am Fallbeispiel in Kapitel 6 demonstriert.

4.4.4 Testverhalten spezifizieren

Wie bereits erwähnt wird in der vorliegenden Arbeit zu Testverhaltensmodellierung nur auf den Einsatz von Sequenzdiagrammen eingegangen. Die hierfür einsetzbaren Mittel wurden in Abschnitt 4.3.1 bereits erläutert und werden hier nicht erneut aufgeführt. Daher beschränkt sich dieser Abschnitt auf die Erläuterung von wichtigen Zusammenhängen, von denen ein Gelingen der Testfallmodellierung abhängig ist.

Bei der Modellierung den Nachrichtenflusses ist darauf zu achten, dass die Wahl zwischen synchronen und asynchronen Nachrichten entsprechend der Intention des modellierten Methodenaufrufs ausfällt. Soll als Ergebnis einer modellierten Testaktion ein Rückgabewert übermittelt werden, welcher einer lokalen Variable zur weiteren Verarbeitung zugewiesen wird, so ist stets eine synchrone Nachricht mit zugeordneter Anwortnachricht zu modellieren. Auch Testaktionen, welche eine möglicherweise verzögerte Antwort des SUT mit sich ziehen, wie beispielsweise das Laden einer Seite nach Aktivierung eines Links, werden durch einen synchronen Aufruf modelliert, dessen rückläufige Antwortnachricht symbolisiert, dass die Anwort des SUT vom Testtreiber empfangen wurde. Testaktionen, die keine beobachtbare Antwort des SUT zur Folge haben, wie beispielsweise die Eingabe eines Textes, können hingegen als asynchrone Nachrichten modelliert werden. Ferner ist darauf zu achten, dass ein Testfall bei der Ausführung nur dann messbare Testergebnisse produziert, wenn Testurteile gefällt werden.

Kapitel 5

Ausführung von Tests

In diesem Kapitel werden Vorgehensweisen erläutert, die für die Erzeugung von Testfällen auf Grundlage einer Testsystemspezifikation verwendet werden können.

5.1 Metamodell für Testmodelle

Während der Ausführung der Tests wird Zugriff auf das Testmodell sowie das Systemodell des zu testenden Systems benötigt. Um die Arbeit mit dem Systemmodell zu vereinfachen, stellt UWE ein eigenes Metamodell zur Verfügung. Die während der Modellierungsphase erstellte Spezifikation, welche zum Zeitpunkt der Modellierung als eine Instanz des UML Metamodells vorliegt, wird dabei durch Modelltransformationen in ein neues Modellartefakt überführt, welches eine Instanz des UWE Metamodells ist.

Wir wollen an dieser Stelle eine Modelltransformation auch für das Testmodell durchführen. Dieser Schritt bietet den Vorteil, dass zusätzliche Details, die in UML-Diagrammen enthalten sind und für unser Vorhaben nicht relevant sind, ausgelassen werden können. Dazu wurde ein Metamodell mit der Zielsetzung entworfen, Testumgebungen mit Testfällen abbilden zu können und in das UWE Metamodell zu integrieren. Im Folgenden werden die darin enthaltene Strukturelemente, welche hauptsätzlich eine Abbildung der von U2TP gebotenen Konzepte sind, sowie die Feinstruktur des für die Repräsentation der Testverhaltensspezifikation verantwortlichen Teils des Metamodells erläutert.

5.1.1 Strukturelemente für U2TP

Dieser Teil des Metamodells baut auf den Konzepten des U2TP Metamodells auf und orientiert sich an der Grundstruktur des bis zur Version 1.1 von U2TP angebotenen Metamodells (siehe 2.4.8). Dies erlaubt den ausführlichen Gebrauch der Spezifikationsmöglichkeiten des U2TP und ermöglicht eine Abbildung von U2TP-basierten Konzepten. Zentrales Element zur Strukturierung nach U2TP ist das Element *TestContext*. Hier befinden sich die Zuordnungen zu den bei der Ausführung wichtigen Elementen wie *SUT*, *Scheduler*, *Arbiter*, *Datapool*, *TestLog*, *TestObjective*, *TestComponent* und *TestCase*. Sie alle definieren den Testkontext näher und müssen über diesen bei der Ausführung erreichbar sein, sofern sie genutzt werden. *SUT* ist dabei ein notwendiges Pflichtelement und muss zwingend vorhanden sein. *Arbiter* und *Scheduler* müssen nur bei zusätzlichen Anforderungen spezifiziert werden, da für einfache Fälle zur Verfügung zu stellende Standardimplementierungen ausreichen. Eine Ansicht des Metamodellschemas für die zuvor genannten Aspekte ist in Abbildung 5.1 dargestellt.

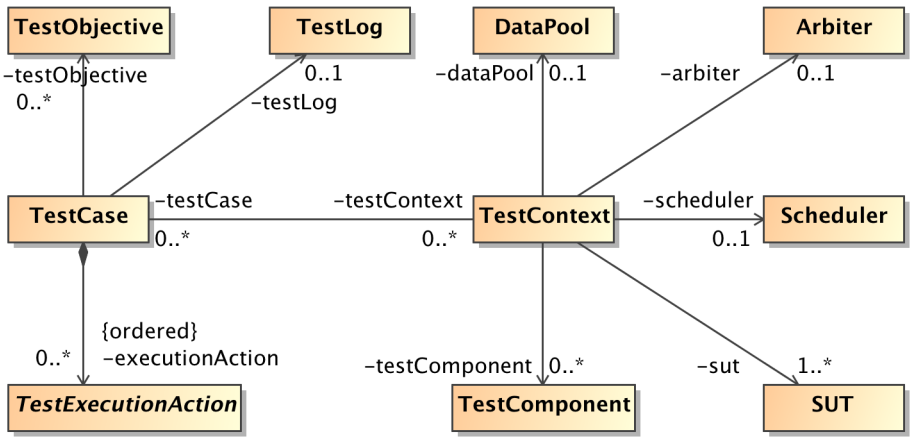


Abbildung 5.1: *Metamodell: Strukturelemente*

5.1.2 Verhaltenselemente für UTSL

Die einzelnen Verhaltenselemente von Testverhaltensspezifikationen werden im Metamodell durch eine geordnete Folge von Testaktionen abgebildet, welche als Instanzen von *TestExecutionAction* repräsentiert werden. Da nicht alle der in Sequenzdiagrammen verwendbaren Verhaltenskonstrukte hier auf die gleiche Weise abgebildet werden können, ist *TestExecutionAction* als abstrakte Klasse definiert und wird im Einzelnen durch weitere Elemente spezialisiert, welche die Repräsentation der entsprechenden Verhaltenselemente in geeigneter Weise vornehmen.

Abbildung 5.2 zeigt eine Detailansicht dieser Elemente des Metamodells. *Execution* übernimmt die Repräsentation der UTSL-Webaktionen aus Kapitel 4 (Abschnitt 4.1.1). *Assertion* sind die Zusicherungen von UTSL aus Abschnitt 4.1.2, *Verification* die ebenfalls dort beschriebenen Auswertungen. *Verdict* steht für eine U2TP *VerificationAction*, *Finish* für die U2TP *FinishAction* und *Log* für eine U2TP *LogAction*, welche in Abschnitt 2.4.3 eingeführt wurden. Das letzte Element *Case* wird für die Realisierung von Fallunterscheidungen genutzt.

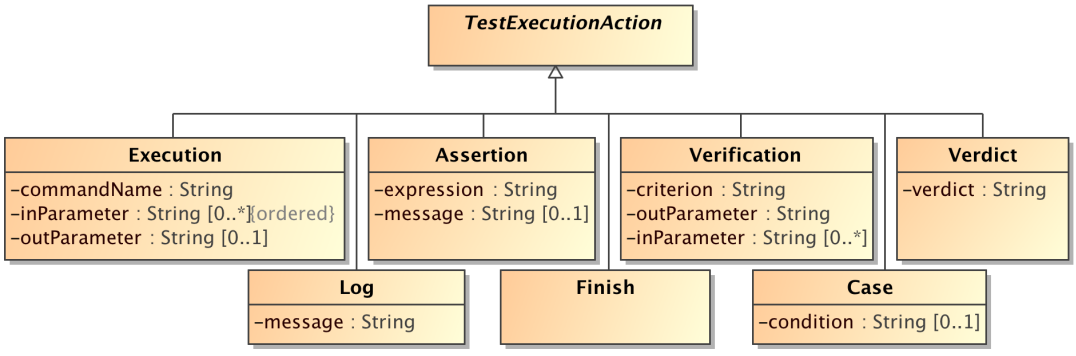


Abbildung 5.2: *Metamodell: Verhaltenselemente*

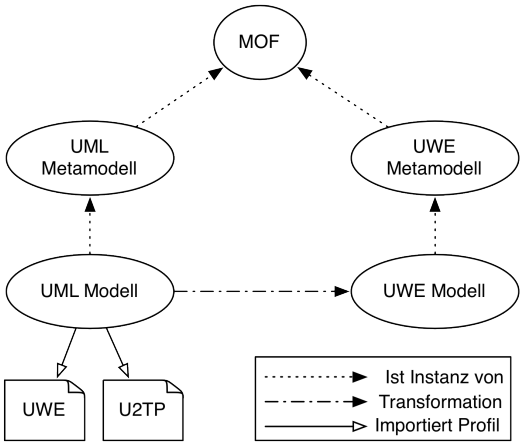


Abbildung 5.3: *Schematische Ansicht der Metamodell-basierten Transformation.*

5.2 Metamodellbasierte Transformation

Zu den Aufgaben der Modelltransformationen in eine Instanz des UWE Metamodells gehört die Erstellung des Testmodells, die Überführung der testarchitekturspezifischen Elemente und des auszuführenden Testverhaltens für jeden Testfall. Die Transformation wird werkzeugseitig in die bestehende Tranformationsumgebung der UWE Werkzeugunterstützung integriert und nutzt die Atlas Transformation Language (ATL). Abbildung 5.3 zeigt eine schematische Darstellung dieser metamodell-basierten Transformation mit den Quell- und Zielmodellen, in der die verwendeten Profile *UWE* und *U2TP*, die involvierten Metamodelle *UML* (Quelle) und *UWE* (Ziel) und das ihnen zugrundeliegende Metamodell *MOF* mit den dabei involvierten Beziehungen gezeigt ist.

Neben den für die Transformation verwendeten ATL-Transformationsregeln werden auch Hilfsmethoden zur Verfügung gestellt, die die Handhabung der testspezifischen Elemente erleichtern. Zu letzterem gehören vor allem Routinen zur Lokalisierung der stereotypisierten Elemente innerhalb des Modells. Diese Hilfsmethoden stellen eine wichtige Grundlage für die eigentlichen ATL-Transformationen dar, indem sie das Identifizieren der spezifizierten Elemente erleichtern. Die eigentlichen Transformationsregeln sind für die Überführung der Elemente der UML-Metamodellinstanz in die zu erstellende UWE-Metamodellinstanz verantwortlich. Dabei gestaltet sich die Transformation des Testmodells als relativ trivial, wie an der beispielhaften Transformationsregel im Code Listing in Abbildung 5.4 zu sehen ist.

```
1 rule TestingModel {
2     from s : UML2!"uml::Model" ( s.isTestingModel())
3     to t : UWE!"uwe::testing::TestingModel"  (
4       name <- s.name,
5       testContext  <- s.packagedElement->select(e | e.isTestContext()),
6       dataPool  <- s.packagedElement->select(e | e.isDataPool()),
7       testCase  <- s.packagedElement->select(e | e.isTestCase()),
8       log  <- s.packagedElement->select(e | e.isTestLog()),
9       objective  <- s.packagedElement->select(e | e.isTestObjective()),
10      sut  <- s.packagedElement->select(e | e.isSut()).first(),
11      arbiter  <- s.packagedElement->select(e | e.isArbiter()),
12    )
13 }
```

Abbildung 5.4: *ATL Transformation zur Erstellung des Testmodells.*

Neben der Transformation der U2TP-Standardelemente, welche ebenfalls relativ trivial ist, werden die wesentlichen Einstiegspunkte der Transformation durch Vorkommen von «TestContext» bestimmt. Es folgt die Zuordnung von Mitgliedern des jeweiligen Testkontextes. Während der Transformation eines Testkontextes wird die Reihenfolge der Ausführung der Testfälle bestimmt. Liegt für ein Vorkommen von «TestContext» eine UML *BehaviourSpecification* vor, so wird diese als UML Interaktion interpretiert, welche durch ein Interaktionsübersichtsdiagramm (*Interaction Overview*) spezifiziert wurde. In diesem Fall wurde die Reihenfolge der Testfallausführung explizit vorgenommen, wie in Abbildung 4.8 in Kapitel 4 bereits gezeigt wurde. Liegt kein eigenes Verhalten vor, so werden die Tests in der Reihenfolge der Methodendeklarationen im UML Testmodell interpretiert.

Einstiegspunkt für die Transformation der Testfälle sind die mit «TestCase» versehenen Instanzen innerhalb des Testmodells. In unserem Fall sind dies Methoden des U2TP Testkontextes. Für jeden Testfall wird durch Auswertung des entsprechenden Sequenzdiagramms eine Reihenfolge der darin modellierten Nachrichten und weiteren Vorkommen von testrelevanten Aktionen bestimmt, so dass sich nach der Transformation eine lineare Abfolge einzelner Testausführungsaktionen ergibt. Resultierende Testausführungsaktionen können dabei die in 5.1.2 beschriebenen Elemente des Metamodells sein:

- Ausführungsaktion (UTSL Webaktion),
- Zusicherung (durch Zustandsinvariante oder explizit)
- Auswertung
- Fällen des Testurteils (U2TP **VerificationAction**),
- Protokolleintrag (U2TP **LogAction**)
- Beendigung der Testausführung (U2TP **FinishAction**).
- Fallunterscheidung (zur Realisierung von bedingten Fragmenten)

Aufgabe der entprechenden Transformationsregel für Testfälle ist es, die im Sequenzdiagramm modellierten Nachrichten, Zustandsinvarianten und kombinierten Fragmente in entsprechende Instanzen von **TestExecutionAction** zu überführen und dabei sicherzustellen, dass die resultierenden Testausführungsaktionen die gleiche Abfolge wie in der ursprünglichen UML-Interaktion behalten. Da die Reihenfolge der Nachrichten im Sequenzdiagramm nur über Vorkommen von UML *BehaviourExecutionSpecification* verlässlich bestimmbar ist, sind dafür zusätzliche Schritte nötig. Die hierfür gefundene Vorgehensweise nutzt die Tatsache, dass sämtliche in einem Sequenzdiagramm vorkommenden Elemente als geordnete Folge von Instanzen der UML Metaklasse *Fragment* ausgelesen werden können. Durch Iteration über die enthaltenen Fragmente kann der Typ jedes Fragments genauer bestimmt und falls nötig auf geeignete Weise aufgelöst werden. Abbildung 5.5 zeigt beispielhaft eine ATL-Transformation für Testfälle, welche diese Vorgehensweise unter Berücksichtigung von nur Nachrichten und Zustandsinvarianten demonstriert. Zu sehen ist in Zeile 13 die Deklaration der Iteration über die in der Sequenz enthaltenen Fragmente. Die zu Beginn der Iteration als leer initialisierte Sequenz *res* wird bei jedem Durchlauf durch ein Fragment ergänzt, sofern dieses übernommen und auf eine Testausführungsaktion abgebildet werden soll. Für jedes vorgefundene Fragment wird dazu durch Fallunterscheidung entschieden, ob eine Übernahme in die resultierende Sequenz *res* erfolgt und wenn ja, inwieweit zusätzliche Verarbeitungsschritte notwendig sind.

In Zeile 14 wird geprüft, ob eine UML **BehaviourExecutionSpecification** vorliegt, welche für uns das Vorkommen einer Nachricht als nächstes Interaktionsfragment kennzeichnet. Ist dies der Fall, so wird in Zeile 17 die korrespondierende Nachricht über die Hilfsmethode **findMessageForBehaviourExectionSpecification** aufgespürt und in Zeile 18 den zu übernehmenden Testausführungsaktionen hinzugefügt. Zustandsinvarianten werden in Zeile 20 geprüft und ohne weitere Konversion in Zeile 21 übernommen. Nach Durchlaufen aller Fragmente wird *res* zurückgegeben und in Zeile 11 auf die Assoziation *executionAction* des Testfalls abgebildet, welche der finale Ablageort für die noch zu transformierenden Testausführungsaktionen ist.

```
1  rule TestCase {
2     from
3        s: UML2!"uml::Operation" (
4           if thisModule.inElements -> includes(s) then
5              s -> oclIsTypeOf(UML2!"uml::Operation")
6              and not s.getMethods().first().oclIsUndefined()
7           else false endif and s.isTestCase()
8        )
9     to
10       t : UWE!"uwe::testing::TestCase" (
11          executionAction <- (
12             s.getMethods().first()
13             .getFragments()->iterate(e; res: Sequence(OclAny) = Sequence{} |
14                if(e.oclIsTypeOf(UML2!"uml::BehaviorExecutionSpecification"))
15                then
16                   let subMessage : Sequence(UML2!"uml::Message") =
17                      e->findMessageForBehaviorExecutionSpecification() in
18                         res->union(subMessage)
19                else
20                   if e.oclIsTypeOf(UML2!"uml::StateInvariant") then
21                   res->append(e)
22                   else res endif
23                endif
24             )
25          )
26       )
27 }
```

Abbildung 5.5: *ATL Transformationsregel zur Erstellung von Testfällen.*

Die Transformation der einzelnen Testausführungsaktionen in die jeweiligen Subtypen erfolgt durch separate Transformationsregeln. Für Teststimuli werden der Name der auszuführenden Testaktion und deren mögliche ein- und ausgehenden Parameter in geeigneter Weise transformiert. Für Zusicherungen wird die angeknüpfte Bedingung übernommen sowie eine optionale Nachricht, die im Falle einer Verletzung in das Testprotokoll übernommen wird.

Die Transformation von kombinierten Fragmenten des Typs *ref* erfolgt durch Einsetzen der im referenzierten Fragment enthaltenen Elemente, welche wiederum Nachrichten, Zustandsinvarianten oder Fragmente sein können. Dazu wird ein rekursiver Ansatz benötigt für den Fall dass Fragmente weitere Fragmente enthalten. Bedingte UML-Fragmente des Typs *opt* werden ähnlich wie *ref*-Fragmente aufgelöst und zusätzlich durch das speziell für diesen Zweck geschaffene Fallunterscheidungskonstrukt mit *IF* und *ENDIF* eingefasst. Ähnlich kann mit *alt*-Fragmenten verfahren werden. Die in den Tests verwendeten MVEL-Ausdrücke werden erst zur Laufzeit ausgewertet.

5.3 Testsystemgenerierung

Aus dem in einer UWE Metamodellinstanz vorhandenen Testfällen kann nun Quelltext für Testfälle generiert werden. Es handelt sich hierbei also um eine *Model-to-Text*-Transformation. Benötigt werden eine konkrete Zielplattform, die Einbettung in eine Umgebung zur Testautomatisierung sowie ein Framework für Webtests, welches für eine Ausführung von UTSL-Aktionen geeignet ist.

Für die Wahl der Zielplattform gibt es grundsätzlich keine besonderen Einschränkungen. Für den hier vorgestellten Ansatz wurde eine Entscheidung zugunsten der Plattform Java getroffen, da durch sie eine breite Werkzeugunterstützung angeboten ist. Dazu gehören nicht nur Frameworks zur Testautomatisierung, wie beispielsweise JUnit, sondern auch Selenium, welches geeignet ist, um UTSL-Aktionen umzusetzen und in unseren Tests zum Einsatz kommen soll. Nach der Festlegung des Webtestframeworks ist die Wahl einer Technologie für den Aspekt der Testautomatisierung erforderlich. Auf Basis von Java bietet sich eine Einbettung in JUnit an. Junit bietet Testautomatisierung an, ist mit Selenium nutzbar und erlaubt die Einbettung in Umgebungen für fortlaufende Integrationstests.

Während der Ausführung der Testfälle werden in unserem Fall zusätzliche Artefakte benötigt. Dazu gehört zunächst das für den Test benötigte Systemmodell, welches zur Laufzeit als Instanz des UWE Metamodells vorliegt und durch die Werkzeugunterstützung für Java des Eclipse Modeling Frameworks in den Testfall eingebunden werden kann. Ferner werden Testparameter benötigt, welche die vom Tester zur Verfügung gestellten Informationen zum SUT enthalten. Dies sind die zur Ansprache des SUT benötigte HTTP-URL sowie die Mappings von UWE Präsentationselementen zu den Repräsentanten der von SUT ausgelieferten HTML-Ansichten, wie in 3.4 erläutert. Zuletzt ist das Vorhandensein einer MVEL-Umgebung notwendig, die Auswertung von MVEL-Formeln vornehmen zu können. Um die genannten Artefakte zur Verfügung zu stellen, wurde eine Basis-Klasse `UWETestBase` erstellt. Sie hält die genannten Artefakte als Klassenmitglieder und initialisiert diese im Konstruktor, wie das Code Listing in Abbildung 5.6 zeigt.

Die Erzeugung von Quelltext ist mit den Werkzeugen der Eclipse Modeling Platform realisierbar. Das transformierte Modell wird dazu als Instanz des UWE-Metamodells eingelesen und für jeden Testkontext eine eigene Klasse erzeugt. Diese erhält für jeden innerhalb des Testkontexts vorkommenden Testfall eine gleichnamige Methode, welche mit der JUnit-Annotation **@Test** versehen wird. Ein Beispiel für einen möglichen generierten Quelltext wird in Kapitel 6 gezeigt.

```
1  package de.lmu.ifi.pst.uwe.mdt.selenium;
2
3  import junit.framework.TestCase;
4  import de.lmu.ifi.pst.uwe.mbt.tcg.fsm.FSM;
5  import de.lmu.ifi.pst.uwe.mbt.tcg.nav.GraphUtil;
6  import de.lmu.ifi.pst.uwe.mbt.tcg.uwe.UWEReader;
7  import de.lmu.ifi.pst.uwe.mdt.selenium.utils.MVELContext;
8  import de.lmu.ifi.pst.uwe.uwe.Model;
9
10 public class UWETestBase extends TestCase {
11    protected UWETestParameter testParameter = null;
12    protected Model model = null;
13    protected MVELContext mvelContext = null;
14
15    public UWETestBase(String name) throws Exception {
16       super(name);
17       try {
18          testParameter = UWETestParameter.mkSimpleShopConfig();
19
20          model = UWEReader.getMainModel("model.uwe.xmi");
21
22          mvelContext = new MVELContext(model);
23       } catch (Exception e) {
24          System.out.println("error setting up test: " + e.getMessage());
25          throw e;
26       }
27    }
28 }
```

Abbildung 5.6: *Basisklasse UWETest*

Kapitel 6

Fallbeispiel: ProductShop

Als Beispiel für eine Anwendung dient ein vereinfachter Online-Shop, in welchem jedes Produkt nur einfach erworben werden kann. Dies könnte z.B. eine Videothek sein oder ein Radiosender, der seinen Zuhörern gegen Entgeld Kopien von eigenproduzierten Hörspielen für den privaten Gebrauch zur Verfügung stellt. Für die Erstellung der Spezifikationen in UML wird das Modellierungswerkzeug MagicDraw eingesetzt.

6.1 Kurzbeschreibung der Anforderungen

Produktnamen können in eine Suchmaske eingegeben werden, als Ergebnis der Suche wird eine Liste von Produkten, deren Titel oder Beschreibung auf den Suchbegriff zutreffen, ausgegeben. Zu jedem Produkt in der Trefferliste gibt es eine eigene Detailansicht, welche über einen Link innerhalb des jeweiligen Suchergebnisses erreichbar ist. Auf den Detailseiten werden ausführliche Informationen zum Produkt angezeigt und eine Möglichkeit geboten, das Produkt dem Warenkorb hinzuzufügen bzw aus dem Warenkorb zu entfernen, falls es zuvor hinzugefügt wurde. Der Inhalt des Warenkorbs kann stets über eine eigene Ansicht eingesehen und dort bestellt werden. Der Benutzer wird nach Betätigung einer Bestellung aufgefordert, diese zu bestätigen. Nach Bestätigung erzeugt das System einen Auftrag mit den Produkten aus dem Warenkorb und einer Gesamtsumme. In der darauf folgenden Ansicht werden die Inhalte der Bestellung zusammen mit der Gesamtsumme angezeigt. Der Bereich des Online-Shops ist geschützt, Zugriff wird den Kunden durch Anmeldung mit einer persönlichen Benutzerkennung ermöglicht.

Die Anforderungen stehen in Form eines Anforderungsmodells zur Verfügung, welches durch das vereinfachte Anwendungsfall-Diagramm in Abbildung 6.1 repräsentiert wird. Die dargestellte Systemgrenze symbolisiert den geschützten Bereich. Eine Auflistung der Anwendungsfälle mit einer kurzen Beschreibung befindet sich in Tabelle 6.1. Die drei farblich hervorgehobenen Anwendungsfälle *search products*, *add products to cart* und *checkout cart* werden nachfolgend durch Testzielsetzungen zum Zweck des Testens ausgewählt. Eine ausführlichere Beschreibung der Anforderungen befindet sich in Anhang A.

Name des Use Case	Beschreibung des Use Case
search Products	Über ein Suchfeld mit einer Texteingabe können Produkte gesucht werden. Das Ergenis der Suche wird als Trefferliste ausgegeben. Jeder Eintrag in der Liste besitzt einen Link zu einer Detailseite des Produkts.
view product details	Eine Detailansicht zeigt zu jedem Produkt ausführliche Informationen. über einen Link kann das Produkt dem Warenkorb hinzugefügt bzw. entfernt werden.
add product to cart	Fügt das gezeigte Produkt dem Warenkorb hinzu.
remove product from cart	Entfernt das gezeigte Produkt aus dem Warenkorb.
checkout cart	Leitet den Bestellvorgang ein. Die Inhalte des Warenkorbs werden aufgelistet und eine Afforderung zur Bestätigung des Vorgangs angezeigt. Ein Link liegt zur Bestätigung der Bestellung, ein zweiter Link bricht den Bestellvorgang ab.
view order details	Anzeige der Bestellungsdetails mit Gesamtsumme.

Tabelle 6.1: *Ausgewählte Usecases für das Beispiel „SimpleShop“*

6.2 Systemspezifikation

Entsprechend dieser Anforderungen ist mit der UWE Profilerweiterung ein Systemmodell erstellt worden, welches die Andorderungen in einfacher Weise umsetzt und nachfolgend beschrieben ist. Aufgrund des Umfangs der Abbildungen wird hier nur eine Auswahl gezeigt. Die übrigen Abbildunge befinden sich im Anhang A.

Zur Übersicht der navigierbaren Ansichten ist in Abbildung 6.2 das Navigationsmodell dargestellt. Zu sehen sind Navigationsklassen *Login* und *Logout*, welche den Lebenszyklus der Benutzersitzung handhaben und die Sicherung des geschützten Bereichs übernehmen. Im Gegensatz zu Logout ist Login ist ein Prozess mit Benutzerinteraktion, welcher im Anhang in Abbildung A.8 abgebildet ist. *Search*, *SearchResults* und *ProductDetail* definieren drei Ansichten für die Suchmaske, die Trefferliste und die Detailansicht der Produkte. *AddProduct* und *RemoveProduct* sind Prozesse ohne eigene Ansicht. Sie werden in der Modellierung innerhalb ihrer Prozesse durch Instanzen des Prozessknotentyps *SystemAction* realisiert, welche das Hinzufügen von Produkten in bzw. Entfernen von Produkten aus dem Warenkorb übernehmen und keine weiteren Ansichten besitzen. *ShoppingCart* steht für eine Navigationsklasse, die durch einen Prozess repräsentiert ist. Dieser Prozess ist in Abbildung A.9 gezeigt. Die einzelnen Ansichten des Prozesses, welche Benutzerinteraktionen ermöglichen, sind im Prozess durch Instanzen des UWE Prozesselements *UserAction* modelliert. Der Prozess beginnt mit der Ansicht *ShowCart*. Hier kann der Benutzer den Warenkorb einsehen und optional die Bestellung tätigen, worauf die Ansicht in *ConfirmationPrompt* gezeigt wird, die zu einer zusätzlichen Kaufbestätigung auffordert. Ein alternativer Link erlaubt den Abbruch des Bestellvorgans und leitet den Benutzer auf die Ansicht *OrderCancelled* weiter, welche den Grund des Abbruchs entsprechend als Benutzerwunsch anzeigt. Bestätigt der Benutzer die Bestellung hingegen, so wird eine Bestellung mit den Inhalten des Warenkorbs erzeugt. Nach erfolgreichem Abschluss der Bestellung wird der Inhalt des Warenkorbs geleert und dem Benutzer die Ansicht *OrderConfirmed* mit der Zusammenfassung des Auftrags angezeigt. Falls ein Systemfehler bei der Durchführung der Bestellung aufgetreten ist, wird die Ansicht *OrderCancelled* angezeigt, welche als

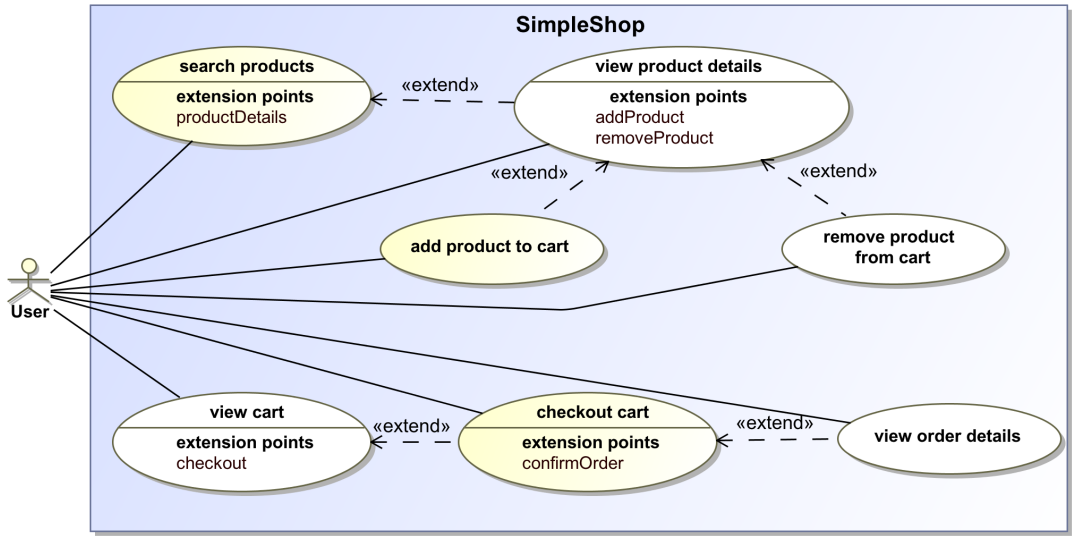


Abbildung 6.1: *Anforderungen für SimpleShop, symbolisiert als zu testende Anwendungsfälle*

Grund des Abbruchs einen Systemfehler berichtet.

MainMenu ist ein durch «Menu» typisierter Navigationsknoten, der ein navigierbares Menu zur Verfügung stellt, welches dem Beutzer in *Search*, *SearchResults* und *ProductDetails* zur Verfügung steht und von dort über Links den Zugriff auf *Search*, *ShoppingCart* und *Logout* ermöglicht.

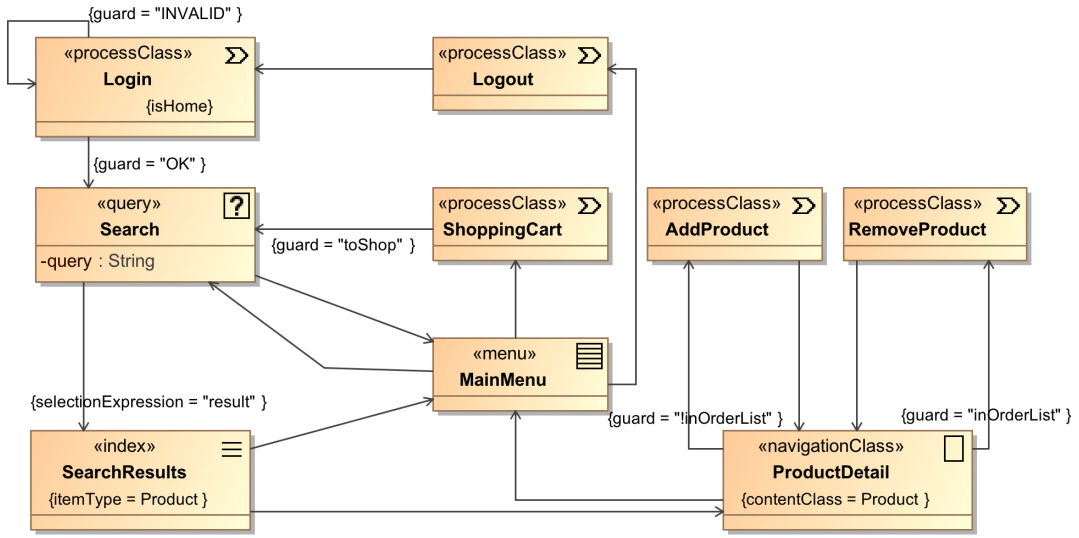


Abbildung 6.2: *SimpleShop Navigation*

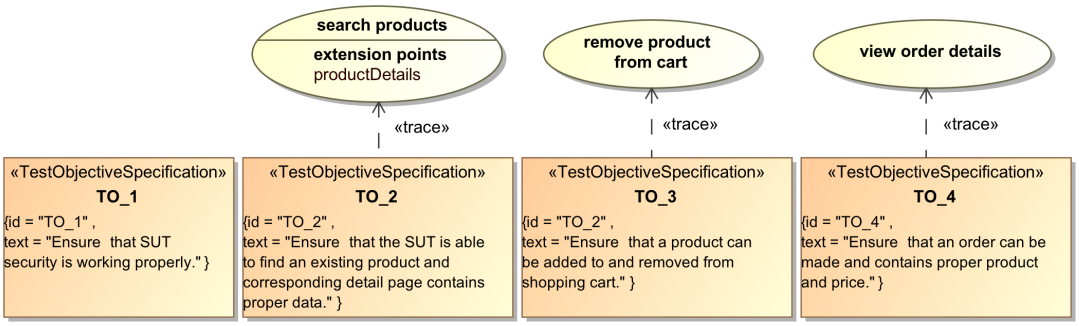


Abbildung 6.3: *Testzielsetzungen mit Beziehungen zu Requirements*

6.3 Testsystemspezifikation

Aufgabe des Testverantwortlichen ist es in diesem Beispiel, sicherzustellen, dass eine Auswahl der Anforderungen entsprechend der Spezifikation umgesetzt wurde und die erwartete Funktionalität leistet. Dazu wurden bei der Testplanung Testzielsetzungen erarbeitet, welche in Tabelle 6.2 aufgelistet sind. Sie sollen messbare Anhaltspunkte geben, welche Aspekte der Funktionalität getestet werden, um die Funktionaltät des SUT gemäß den Anforderungen sicherzustellen. Jede Testzielsetzung wird mit einem Identifikator und einer kurzen Beschreibung festgehalten.

ID	Beschreibung (englisch)
TO_1	Ensure that SUT security is working properly.
TO_2	Ensure that the SUT is able to find an existing product and corresponding detail page contains proper data.
TO_3	Ensure that a product can be added to and removed from shopping cart.
TO_4	Ensure that an order can be made and contains proper product and price.

Tabelle 6.2: *Testzieldefinitionen für „SimpleShop“*

Nach Vorliegen der Testzielsetzungen, wird nun ein Modell für die Spezifikation des Testsystems erstellt. Mit MagicDraw ist dazu eine UML Instanz vom Typ *Model* angelegt und mit dem Stereotyp «testingModel» aus dem UWE Testprofil versehen worden. Damit ist die Erstellung des Testmodells abgeschlossen und es können Inhalte einbeschrieben werden.

Begonnen wird mit der Spezifikation der Testzielsetzungen aus Tabelle 6.2. Diese werden jeweils als neue Klassen in das Testmodell einbeschrieben und unter Verwendung des U2TP Stereotyps «TestObjectiveSpecification» typisiert. Die jeweiligen Identifikatoren der Testzielsetzungen und die dazu gehörigen Beschreibungen werden durch die dem Stereotyp zugehörigen Eigenschaften näher spezifiziert. Um eine Rückverfolgbarkeit der Anforderungen in den Testergebnissen zu ermöglichen, werden die mit U2TP spezifizierten Testzielsetzungen auf geeignete Weise mit den Anforderungen aus Abschnitt 6.1 in Beziehung gesetzt. Dazu wird die ebenfalls von der UML zur Verfügung gestellte „trace“-Beziehung von den Testzielsetzungen hin zu den jeweiligen Anwendungsfällen einbeschrieben. Abbildung 6.3 zeigt das Ergebnis dieser Festlegung.

Damit sind Testzielsetzungen und deren Beziehungen zu den Anforderungen defi-

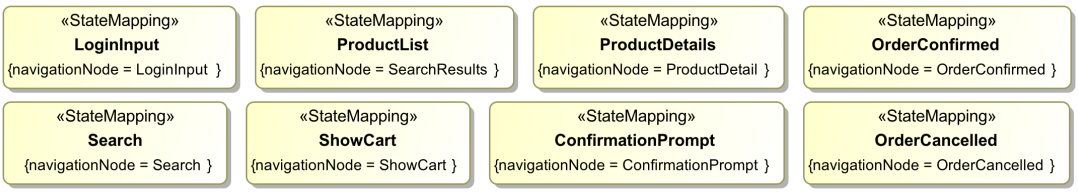


Abbildung 6.4: *StateMapping für Zustandsinvarianten*

niert. Im nächsten Schritt wird die Testumgebung definiert. Ergebnis dieses Schritts wird ein U2TP-Testkontext sein, welcher Testfälle enthält, das SUT, den Testtreiber und einen Datenpool für Testdaten. Für Testkontext wird die Klasse *ShopTestContext* in das Modell einbeschrieben und mit «TestContext» typisiert. Ihr werden für jeden Testfall eine Operation mit Rückgabewert von Typ `Verdict` einbeschrieben und jeweils mit «TestCase» stereotypisiert. Für die einzelnen Testfälle wurden die Namen *searchProductTest*, *addToCartTest*, *viewOrderDetailsTest* sowie *securitTest* gewählt.

Um innerhalb der Testfälle Zustandsinvarianten verwenden zu können, soll ein angepasster Testtreiber erstellt werden. Dazu wird eine neue Klasse *ShopTestClient* erstellt und mit dem Stereotyp «TestComponent» versehen. Nun wird dem Diagramm die Instanz *TestDriver* des UWE Testing Profile hinzugefügt und eine Spezialisierung von *TestDriver* zu *ShopTestclient* einbeschrieben. Damit für *ShopTestClient* Zustände angelegt werden können, wird für diesen ein UML Zustandsautomat erstellt, welcher der dem *ShopTestClient* als UML *OwnedBehaviour* zugewiesen wird. Anschließend wird für jede NavigationClass des Navigationsmodells ein Zustand erstellt, mit dem Stereotyp «StateMapping» typisiert sowie der Eigenschaft *navigationNode* des Stereotyps die entsprechende NavigationClass zugewiesen. Das Ergebnis dieses Schritts ist in Abbildung 6.4 dargestellt. Nun werden *ShopClient* und die dem UWE Testing Profile entnommene Instanz *WebSUT* dem Strukturbereich des Testkontextes hinzugefügt, wie in Abbildung 6.5 abgebildet.

Zur Erstellung des Datenpools wird eine neue Klasse erstellt und mit «DataPool» stereotypisiert. Wie in Abbildung 6.6 zu sehen werden zunächst Klassen für die Definition von Klassen der Testobjekte angelegt und dazugehörige Objektinstanzen mit konkreten Werten als Attribute erstellt. Zusätzlich werden Datenselektoren unter Verwendung des StereotypsDataPartition angelegt, welche in diesem Fall jeweils eine mit «DataSelector» typisierte Operation als Datenselektor erhalten. Wie in der Abbildung dargestellt müssen die Datenpartitionen von der Klasse erben, welche die Klassendefinition für Objektinstanzen repräsentiert. Die Zuordnung der Partitionen zum Datenpool erfolgt durch gewöhnliche UML Assoziationen, wobei die Kardinalitäten mit einem Wert spezifiziert werden, der die Menge der in Partition enthaltenen Objektinstanzen wiederspiegelt. Der Name der Datenpartition wird in den Sequenzdiagrammen der Testverhaltensspezifikation für den Zugriff auf die Datenpartition verwendet. Deswegen wird hier der etwas ungewöhnliche Name *data* in Kleinschreibweise verwendet. Damit der fertig erstellte Datenpool in der Testumgebung verwendet werden kann, wird er dem Struktbereich des Testkontextes hinzugefügt. Damit ist die Erstellung des Testkontextes abgeschlossen.

Nachdem nun Definitionen der benötigten Testfälle vorliegen können diese in Beziehung zu den Testzielsetzungen gesetzt werden. In diesem Beispiel wird hierfür jedem Testfall eine Zielsetzung zugeordnet. Dazu wird ein Testfall mit der UML-Beziehung *Realization* der entsprechenden Testzielsetzung zugewiesen. Das Ergebnis dieser Zuwei-

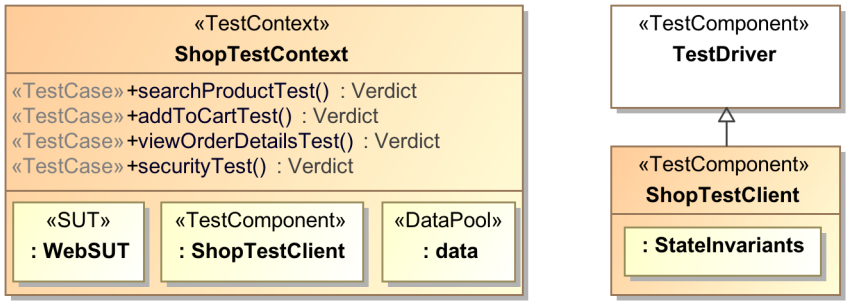


Abbildung 6.5: *Spezifikation der Testumgebung für SimpleShop*

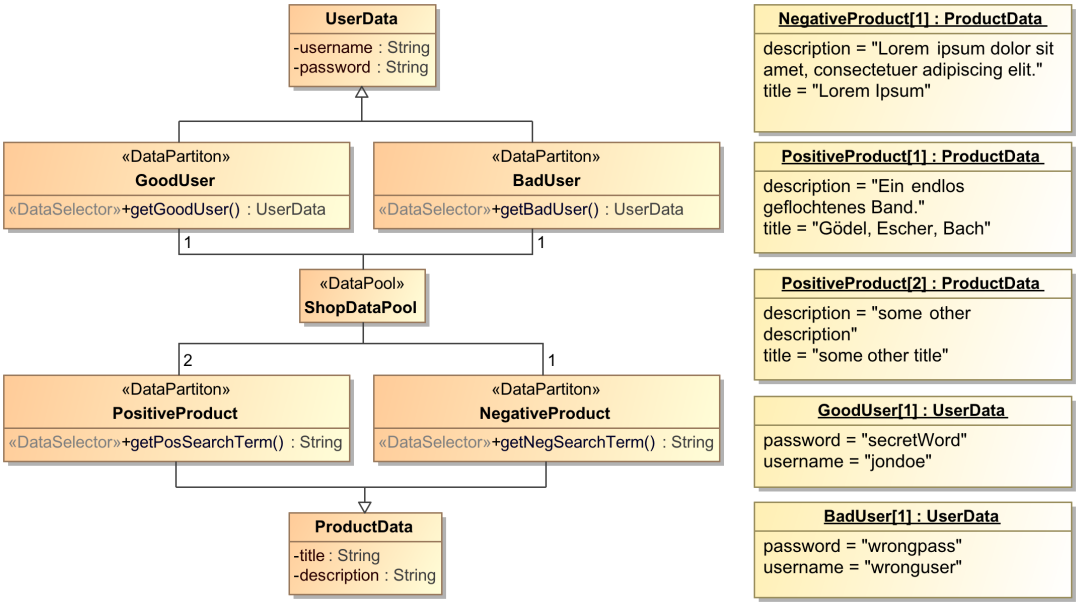


Abbildung 6.6: *Datenpool mit Testdaten als Instanzen und Datenpartitionen*

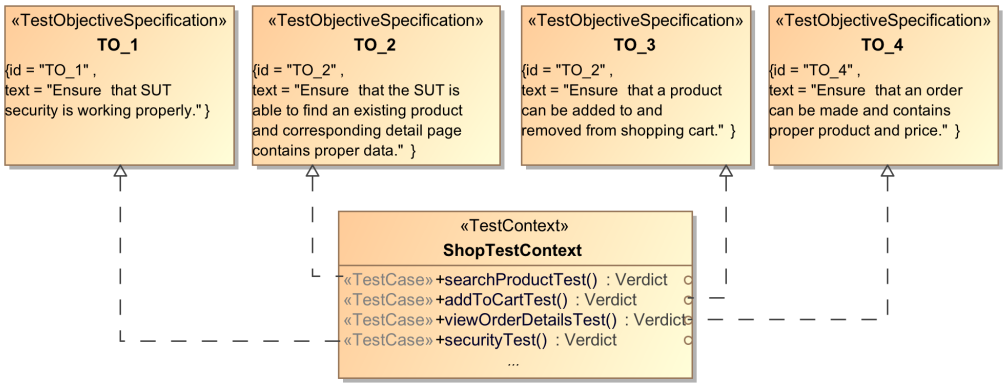


Abbildung 6.7: *Testfälle mit Beziehungen zu Testzielsetzungen*

sung ist in Abbildung 6.7 dargestellt.

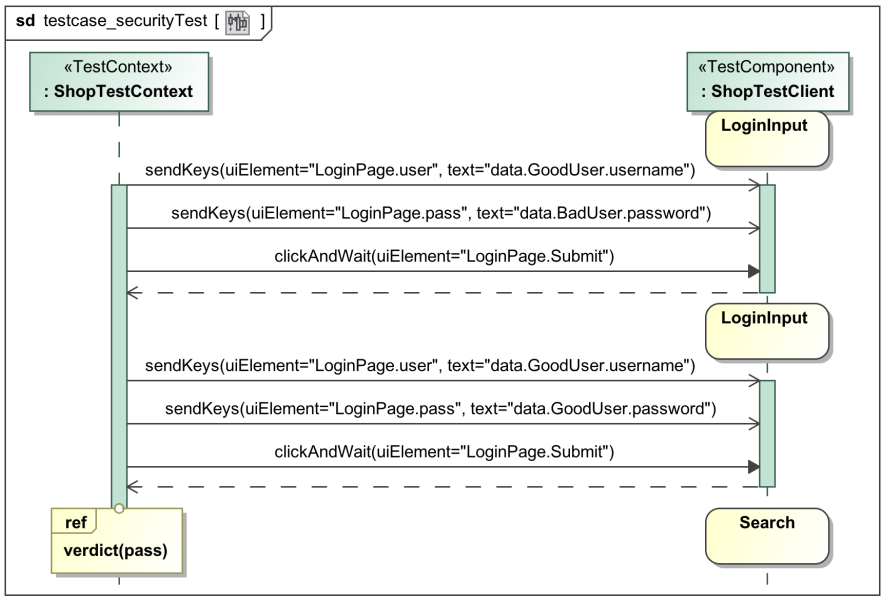


Abbildung 6.8: *Modellierung des Testfalls* securityTest

Im vorerst letzten Schritt der Testsystemmodellieung erfolgt nun die Verhaltensspezifikation der Tests. Im vorliegenden Beispiel bestehen natürliche Abhängigkeiten zwischen den Testzielsetzungen, so dass die Erfüllung der Testzielsetzungen TO_2 bis TO_4 jeweils von der davor liegenden abhängig ist. Das bedeutet in diesem Fall z.B. konkret: Gelingt es nicht, über die Suchmaske das gewünschte Produkt zu finden, so können die darauf folgenden Ziele ebenfalls nicht erfüllt werden. Auf die Auswirkungen dieser Abhängigkeiten auf die Rückverfolgbarkeit der Anforderungen wird im Beispiel aus Gründen der Übersichtlichkeit nicht näher eingegangen. Für die Modellierung der Testfälle ist es durch diesen Umstand jedoch teilweise möglich und auch sinnvoll, einzelne wiederverwendbare Fragmente zu erstellen. Durch die Verwendung dieser Fragmente in den Testfällen wird die Testmodellierung nich nur übersichtlicher, es können auch mögliche Redundanzen, die Ursachen für Fehler bei der Testmodellerstellung sein können, vermieden werden. Für den ersten Testfall *securityTest* sollen im Testverhalten zwei Schritte durchgeführt werden:

1. Anmeldung mit falscher Benutzerkennung (Erwartung: gelingt nicht)
2. Anmeldung mit gültiger Benutzerkennung (Erwartung: gelingt)

Gelingt eine Anmeldung, so erscheint die Suchmaske, sonst erscheint erneut die Anmeldeaufforderung. Auf diese Weise kann erkannt werden, ob der Anmeldevorgang erfolgreich war. Die als Sequenzdiagramm modellierte Spezifikation des Testfalls ist in Abbildung 6.8 dargestellt. Eine zu Beginn gesetzte Zustandsinvariante stellt sicher, dass das vom SUT empfangene Dokument die erwartete Ansicht enthält, hier die Anmeldeseite LoginInput. Für den ersten Anmeldevorgang wird ein gültiger Benutzername über die Datenpartition *data* ausgewählt und in das entsprechende Eingabefeld der Anmeldeseite eingegeben. Es wird anschließend jedoch nicht das dazugehörige Passwort verwendet, sondern das passwort eines ungültigen Benutzers. Nach Absenden des Formulars durch Drücken des entsprechenden Buttons in der Ansicht und dem damit verbundenen Warten auf die Antwort des SUT wird der Testfall fortgesetzt. Zunächst stellt eine erneute Zustandsinvariante sicher, dass die Anmeldeseite nicht verlassen wurde, also der vorhergehende Anmeldeversuch erfolglos war. Im Anschluss wird die Anmeldung

mit gültigen Benutzerdaten wiederholt und über die gesetzte Zustandsinvariante der Erfolg des Anmeldevorgangs sichergestellt. Am Ende des Testfalls wird das Testurteil auf *pass* gesetzt. Damit ist die Modellierung dieses Testfalls vollständig.

In den nun folgenden Testfällen wird innerhalb des geschützten Bereichs der Anwendung getestet. Da für jeden Testfall das Öffnen der Verbindung und eine Anmeldung am System erforderlich ist, wird ein Fragment erstellt, welches diesen Ablauf enthält und innerhalb von Testfällen wiederverwendet werden kann. Das erstellte Fragment ist in Abbildung 6.9 gezeigt. Um sicherzustellen, dass der Anmeldevorgang erfolgreich war, wird dies am Ende mit dem Befehl *assertPage* sichergestellt. Das wiederverwendbare Fragment für die Systemanmeldung ist damit komplettiert.

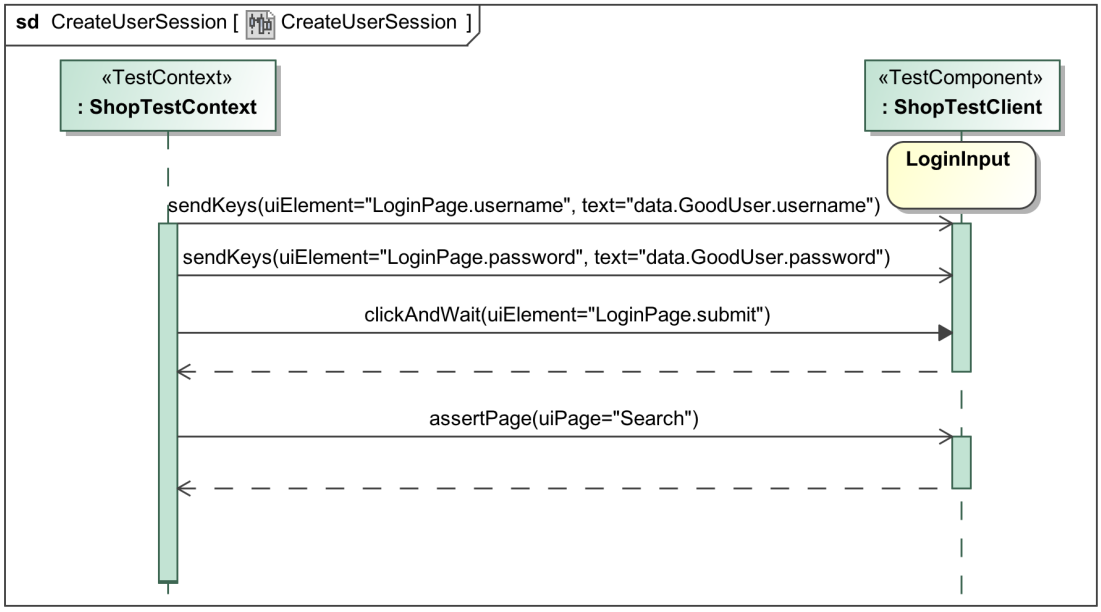


Abbildung 6.9: *Wiederverwendbares Fragment für die Anmeldung am System mit einer gültigen Benutzerkennung*

Nun wird der zweite Testfall beschrieben, *searchProductTest*, dargestellt in Abbildung A.10. Durchgeführt wird eine Verifikation der Suchfunktion und der Produktpräsentation. Hier wird für den Anmeldevorgang am System zu Beginn des Testfalls das Fragment aus Abbildung 6.9 verwendet. Nach erfolgter Anmeldung wird der Titel eines Vorhandenen Produkts, das zuvor aus dem Datenpool über die entsprechende Partition ausgewählt wurde, in die Eingabemaske der Suchfunktion eingegeben. Die Suche wird ausgelöst über Aktivierung des dafür vorgesehenen Knopfs im Präsentationsmodell. Da mit diesem Vorgang ein Neuaufbau der Seite zu erwarten ist, wird der synchrone Aufruf gewählt, welcher auf die Antwort des Testtreibers nach Neuaufbau des Dokumentenmodells wartet. Nach Systemspezifikation ist damit ein Übergang im Navigationsgraph erfolgt. Es sollte daher vom SUT die mit dem Navigationsknoten *ProductList* verbundene Ansicht präsentiert worden sein, welche in diesem Fall eine IteratedPresentationGroup enthalten soll, nämlich die Trefferliste der Suche. Da der weitere Verlauf des Testfalls vom Erfolg dieses Übergangs abhängig ist, wird dies durch eine Zustandsinvariante entsprechend sichergestellt.

Für die Ergebnisliste wird über den MVEL-Ausdruck `ProductList[0]` zunächst das Vorhandensein der erwarteten Ergebniszeile geprüft. Anschließend wird getestet, ob in dieser Zeile der Inhalt des Feldes *title* zu finden ist. Hier wurde der exakte Textver-

gleich gewählt, es dürfen also keine zusätzlichen Zeichen vorkommen. Anschließend wird das Vorkommen des Links zur Produktdetailansicht sichergestellt, der Link aktiviert und auf die Anwort des SUT gewartet.

Die Spezifikation der beiden letzten Testfälle *addToCartTest* und *viewOrderDetails-Test* erfolgt nach den gleichen Prinzipien der Testmodellierung. Für *addToCartTest* wird ein parametrisiertes, Kombiniertes Fragment *cfrag_searchProduct* eingesetzt, welches den Teilablauf von *searchProductTest* bis zur Ansicht ProductDetails enthält. Die darin enthaltene Ablaufspezifikation enthält keine Abläufe, die nicht schon gezeigt worden wären. Die Spezifikation von *addToCartTest* ist im Anhang in Abbildung A.11 zu sehen. Ähnlich verhält es sich mit dem Testfall *viewOrderDetailsTest*. Bei letzterem wurde ein Teilablauf zum Hinzufügen von Produkten in den Warenkorb modularisiert und durch das parametrisierte Fragment *cfrag_placeProductInCart* eingebunden. Die Spezifikation von *viewOrderDetailsTest* ist in Abbildung A.12, ebenfalls im Anhang befindlich, abgebildet.

Typ	TestExecutionAction	Parameter(in)
Assert	assertNavigationNode	node=LoginInput
Exec	sendKeys	uiElement=LoginPage.user, text=data.GoodUser.username
Exec	sendKeys	uiElement=LoginPage.pass, text=data.BadUser.password
Exec	click	uiElement=LoginPage.Submit
Exec	waitForPageToLoad	-
Assert	assertNavigationNode	node=LoginInput
Exec	sendKeys	uiElement=LoginPage.user, text=data.GoodUser.username
Exec	sendKeys	uiElement=LoginPage.pass, text=data.GoodUser.password
Exec	click	uiElement=LoginPage.submit
Exec	waitForPageToLoad	-
Assert	assertPage	uiElement=Search
Verdict	setVerdict	PASS

Tabelle 6.3: *Darstellung der Instanzen von TestExecutionAction nach Transformation, hier zu sehen der Testfall securityTest*

6.4 Transformation und Generierung

Nach erfolgter Spezifikation mit MagicDraw liegt eine Modelldatei vor, welche die Spezifikationen für Anforderungen, System und Testsystem enthält. Für die weitere Verarbeitung ist der Inhalt der Modelle in ein XMI-Dokument mit Namen `simpleshop.xmi` exportiert worden. Die Ausführungsumgebung der UWE Werkzeuge für Transformationen in Eclipse wurde so konfiguriert, dass `simpleshop.xmi` als Eingabemodell für die ATL-Transformationen eingerichtet ist und das UWE Metamodell der Transformation zur Verfügung steht. Nach erfolgter Transformation steht als Ergebnisartefakt die Datei `simpleshop.uwe.xmi` zur Verfügung. Sie ist eine Instanz des UWE Metamodells und enthält das transformierte System- und das Testmodell.

Das darin vorliegende Testmodell enthält die Instanzen der nun transformierten Elemente der Testsystemspezifikation. Innerhalb des Testmodells befindet sich der Testkontext und die weiteren zuvor in diesem Kapitel spezifizierten, strukturellen Elemente des Testmodells, welche dem Schema in Abschnitt 5.1 entsprechend erzeugt wurden. Das spezifierte Testverhalten einzelner Testfälle wurde, wie in Abschnitt 5.2 erläutert, durch die Erzeugung einzelner Instanzen von *TestExecutionAction* abgebildet. Sie liegen nun innerhalb der Instanzen von *TestContext* als geordnete Folge der spezialisierten Ausprägungen von *TestExecutionAction* (siehe 5.1.2) vor. Tabelle 6.3 zeigt das Ergebnis dieser Transformation für den Testfall securityTest. Zum Vergleich sei hier auf das ursprüngliche Sequenzdiagramm in Abbildung 6.8 hingewiesen. Ein beispielhafter Ausschnitt für ein mögliches Ergebnis eines generierten Java-Quelltextes ist im Code Listing in Abbildung 6.10 gezeigt. Dargestellt ist die Anfangssequenz des Testfalls *securityTest*.

Element der UWE Spezifikation	CSS-Selektor
LoginPage	body.loginpage
LoginPage.user	input.loginuser
LoginPage.pass	input.loginpass
LoginPage.Submit	input.loginsubmit

Tabelle 6.4: *Zuordnungen von UWE Oberflächenelementen zu Elementen der Implementierung durch CSS-Selektoren*

6.5 Testvorbereitung

Die Implementierung wurde unter Verwendung der Sprache PHP erstellt. Damit eine Ausführung der Tests möglich ist, sind die dazu benötigten Abbildungen von Elementen der Oberflächenspezifikation zu den korrespondierenden Elementen in den HTML-Ansichten erstellt worden. Das Ergebnis dieses Vorgangs wird hier anhand der HTML-Seite für die Benutzeranmeldung näher erläutert.

Die Spezifikation der dafür vorgesehenden Ansicht *LoginInput* ist in Abbildung 6.11 dargestellt. Sie schreibt das Vorhandensein zweier Eingabefelder *user* und *pass* zur Eingabe des Benutzernamens und des Passwortes vor, sowie einen Button *Submit*, um die Übermittlung der Daten auszulösen. In Abbildung 6.12 ist die Ausgabe der HTML-basierten Ansicht, welche bei Aufruf der korrespondierenden Seite der Implementierung vom Webserver ausgeliefert wird. Zu sehen sind eine HTML-Seitendefinition mit einigen typischen, für uns nicht weiter relevanten Merkmalen wie z.B. die Dokumentendefiniton (Zeilen 1 und 2) mit entsprechendem Kopfbereich (Zeilen 3 bis 7). Die für uns relevanten Inhalten sind im Rumpf der Seite zu suchen. Die Rumpfdeklaration (HTML *body* in Zeile 8) besitzt die CSS-Klassendefinition `loginform`. Sie kann verwendet werden, um die Identifikation der Seitendefinition *LoginPage* (Abbildung 6.11) durchzuführen. Im Rumpf enthalten sind anderem ein Formular (Zeile 10), bestehend aus zwei Eingabefeldern für Benutzername (Zeile 12) und Passwort (Zeile 14), sowie einem *Submit*-Button (Zeile 16) zum Absenden des Formulars. Auch hier finden sich ausreichende Klassendefinitionen für das Formular (*loginform*) und die Formularelemente (*loginuser*, *loginpass* und *loginsubmit*), welche zur Lokalisierung dieser Elemente genutzt werden kann.

Während einer Besprechung zwischen dem Testverantwortlichen und dem Implementierer werden die einzelnen Teile der Präsentationsspezifikation durchgegangen und mit den HTML-Ausgaben des erstellten Systems verglichen. Dazu wurde das fertiggestellte System auf einen Webserver aufgespielt und eingerichtet, so dass es über eine URL erreichbar ist. Der Implementierer führt dem Tester das System vor und erläutert dabei, welche HTML-Elemente zu den einzelnen Instanzen in der Spezifikation zugehörig sind. Der Tester notiert sich diese Zuordnungen und trägt sie in die Konfigurationsdatei mit den Testparametern ein.

```
1 package de.lmu.ifi.pst.uwe.mdt.selenium.testcases;
2 /* import statements removed for code listing example */
3
4 public class SecurityTest extends UWETestBase {
5
6    public SecurityTest(String name) throws Exception {
7       super(name);
8    }
9
10   @Test
11    public void testSomething() throws Exception {
12       WebDriver driver = new HtmlUnitDriver(true);
13
14       driver.get(testParameter.getStartUrl());
15
16       /* assertNavigationNode */
17       UWETestUtil.assertNavigationNode("LoginInput", model, driver);
18
19       /* sendKeys */
20     {   String arg0 = "LoginPage.user";
21         String arg1 = "data.GoodUser.username";
22         String literalText = mvelContext.resolveMvelExpression(arg1);
23         String cssSelector = testParameter.resolveUiMapping(arg0);
24         WebElement element = driver.findElement(
25               By.cssSelector(cssSelector));
26         element.sendKeys(literalText);   }
27
28       /* sendKeys */
29     {   String arg0 = "LoginPage.pass";
30         String arg1 = "data.BadUser.password";
31         String literalText = mvelContext.resolveMvelExpression(arg1);
32         String cssSelector = testParameter.resolveUiMapping(arg0);
33         WebElement element = driver.findElement(
34               By.cssSelector(cssSelector));
35         element.sendKeys(literalText);   }
36
37
38       /* click */
39     {   String arg0 = "LoginPage.Submit";
40         String cssSelector = testParameter.resolveUiMapping(arg0);
41         WebElement element = driver.findElement(
42               By.cssSelector(cssSelector));
43         element.click(); }
44
45       /* wait */
46    {    WebElement tmpElement = (new WebDriverWait(driver, 2))
47                .until(ExpectedConditions.presenceOfElementLocated(
48                    By.tagName("body")));  }
49
50
51       /* assertNavigationNode */
52       UWETestUtil.assertNavigationNode("LoginInput", model, driver);
53
54      /* remaining actions of securityTest removed for code example */
55       driver.quit();
56    }
57 }
```

Abbildung 6.10: *Beispiel für generierbaren Java Code*

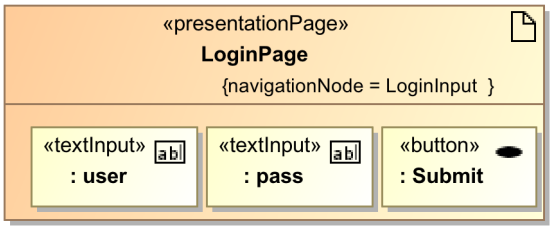


Abbildung 6.11: *Präsentationsspezifikation für Systemanmeldung*

```
1  <!DOCTYPE html>
2  <html>
3  <head>
4      <meta charset="UTF-8">
5      <title>SimpleShop Login</title>
6      <link rel="stylesheet" type="text/css" href="styles.css"/>
7  </head>
8  <body class="loginpage">
9     <h3>Login</h3>
10    <form action="login.php" method="post" class="loginform">
11       username:
12        <input type="text" name="username" value="" class="loginuser">
13        <br>
14       password:
15        <input type="password" name="password" value="" class="loginpass">
16        <input type="submit" name="submit" value="login" class="loginsubmit">
17    </form>
18 </body>
19 </html>
```

Abbildung 6.12: *Ausschnitt aus HTML-Code des Anmeldeformulars*

Kapitel 7

Zusammenfassung und Ausblick

In der vorliegenden Arbeit wurde ein Ansatz vorgestellt, welcher die Generierung von Testfällen zum Zweck des Blackbox-Testens von Websystemen auf Basis plattformunabhängiger Modelle ermöglicht. Die dabei verwendete Vorgehensweise lässt sich als system- und testmodellgetriebene Variante in das Umfeld des modellgetriebenen Testens einordnen. Ausgehend von der UML als Modellierungsgrundlage werden zu diesem Zweck die Methoden des UML-Based Web Engineering (UWE) zur Systemmodellierung verwendet und das UML2.0 Testing Profile (U2TP) zur Testsystemmodellierung.

In diesem Rahmen wurde das UWE Testing Profile entwickelt, welches die Erstellung einer Testsystemspezifikation zu einer mit UWE entworfenen Anwendung erlaubt, sowie die domänenspezifische Sprache UWE Test Specification Language (UTSL), welche die Spezifikation von Webtests unter Verwendung eines UWE Systemmodells zulässt. Weiterhin wurden eine Ergänzung des UWE Metamodells vorgenommen sowie Modelltransformationen erstellt, welche die Überführung der Testsystemspezifikation in eine UWE Metamodellinstanz ermöglichen. Für die anschließende Codegenerierung auf Basis des metamodellbasierten Testmodells wurde ein Ansatz unter Verwendung von JUnit und Selenium vorgeschlagen, welcher die Ausführung von UTSL-Aktionen unter Verwendung eines UWE Systemmodells zur Laufzeit leisten kann. Zudem wurde die Integration von UTSL als U2TP Testkomponente erläutert sowie die Vorgehensweise bei der Testmodellerstellung unter Verwendung selbiger. Anhand des Fallbeispiels SimpleShop wurde eine Durchführung der Methode ausgehend von einem Anforderungs- und Systemmodell bis hin zu ausführbarem Testcode demonstriert.

Wie dabei aus der vorliegenden Arbeit hervorgeht, ist das Testen von Webanwendungen mit der beschriebenen Methode durchführbar. Die Kombination aus UWE, U2TP und UTSL ermöglicht die Spezifikation von Testmodellen, sodass diese nach erfolgten Transformationen zur Ausführung gebracht werden können. Dies wird unter anderem durch den Umstand ermöglicht, dass aus dem Testmodell heraus Elemente des Systemmodells verwendet werden können, um eine Beurteilung der Reaktionen des SUT zur Laufzeit durchzuführen. Eine weitere Besonderheit bei diesem Ansatz ist die durchgehend plattformunabhängige Herangehensweise, welche die von UWE zur Verfügung gestellten Mittel durch den Ansatz der domänenspezifischen Sprache um die Eigenschaft der Testbarkeit ergänzen. Daher scheint die Kombination von UWE, U2TP und UTSL geeignet zu sein, Web-Testsysteme aus plattformunabhängigen Modellen zu erzeugen.

Die Möglichkeit der frühzeitigen Verifikation von Systemen ist mit dieser Methode gegeben. Durch die optionale, anforderungsorientierte Erstellung von Testzielsetzungen und damit verbundenen Testfällen ist auch eine zusätzliche Möglichkeit zur Validierung

des Systemodells insofern vorhanden, als während der Spezifikation des Testverhaltens mögliche Schwächen des Systementwurfs aufgedeckt werden können. Durch den operationellen Charakter von UTSL lassen sich auch Vorgänge testen, welche sich nicht ohne weiteres aus einem Systemmodell ableiten lassen. Durch die Nutzbarmachung der Potenziale des U2TP ist dabei die Möglichkeit gegeben, mit relativ einfachen Mitteln durchaus anspruchsvolle Testszenarien umzusetzen. Dabei besteht zudem die Möglichkeit, Testmodelle schrittweise zu erstellen.

Die genannten Punkte könnten ein Anreiz sein, die vorgestellte Methode auch in Projekten einzusetzen, die keine Verwendung des UWE Entwicklungsprozesses vorsehen. Begünstigt wird diese Vorstellung durch den Umstand, dass die Technik zur Erstellung eines UWE Systemmodells für Webentwickler mit UML-Kenntnissen verhältnismäßig leicht erlernbar ist.

Die Ergebnisse dieser Arbeit geben auch Anhaltspunkte für weitere Tätigkeiten. Obwohl die vorgestellte Methode auch Unterstützung für Oberflächen mit dynamischen HTML-Realisierungen wie beispielsweise AJAX bietet, ist eine Erweiterung des Umfangs für z.B. *Drag-and-Drop*-basierte Oberflächenelemente realisierbar. Ferner ist zu vermuten, dass durch eine Zuordnung von Testdaten zu den Eingabeparametern von Prozessen sich letztere einfacher testen lassen. Nicht zuletzt wäre eine Verbesserung der Werkzeugunterstützung denkbar, etwa durch ein MagicDraw-Plugin zur Modellierung von Testfällen.

Anhang A

SimpleShop

In diesem Anhang werden Teile der Spezifikation des Beispielsystems SimpleShop abgebildet, welche in Kapitel 6 aus Platzgründen nicht aufgenommen wurden.

A.1 Systemmodell

Abbildung A.1 zeigt das Inhaltsmodell sowie das Benutzermodell.
In den Abbildungen A.2 bis A.7 sind die übrigen Teile der Präsentationsspezifikation zu sehen.
Abbildung A.8 zeigt die Spezifikation des Prozesses *Login*, in Abbildung A.9 ist die Spezifikation des Prozesses *ShoppingCat* dargestellt.

A.2 Testmodell

In den Abbildungen A.10 bis A.12 sind die Spezifikationen der Testfälle *searchProductTest*, *addToCartTest* sowie *checkoutCartTest* abgebildet. Das Code-Listing in Abbildung A.1 zeigt die Darstellung des Testfalls *searchProductTest* in Metamodelldarstellung.

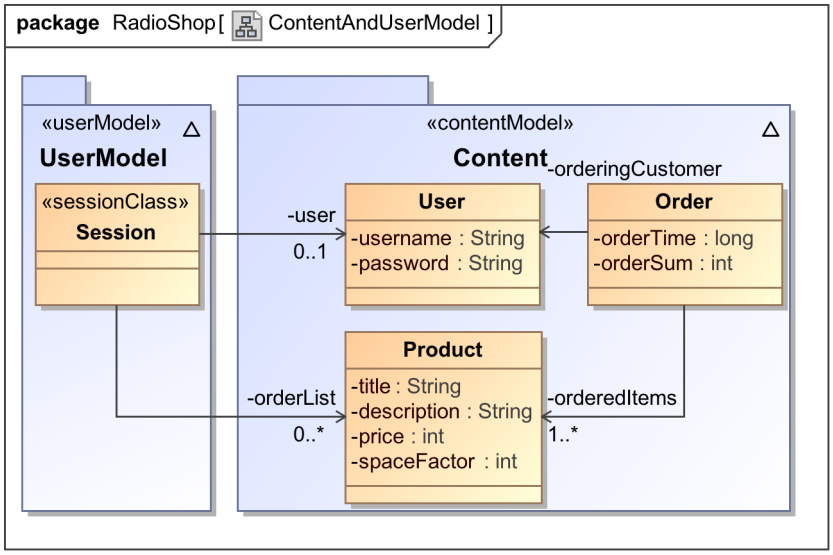


Abbildung A.1: *SimpleShop Inhaltsmodell*

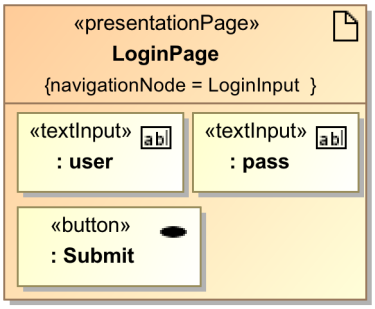


Abbildung A.2: *SimpleShop Präsentation: LoginPage*

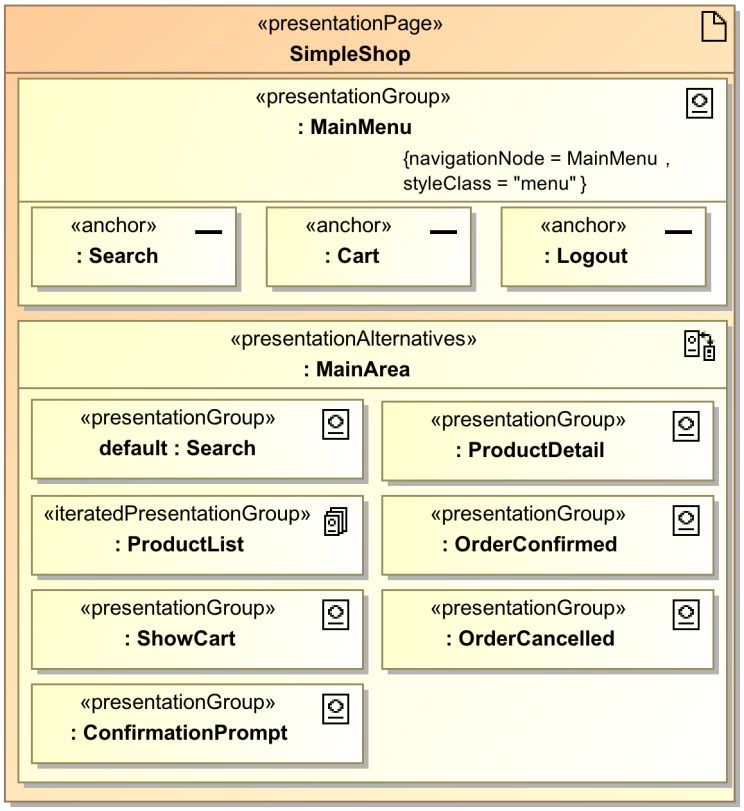


Abbildung A.3: *SimpleShop Präsentation: SimpleShop*

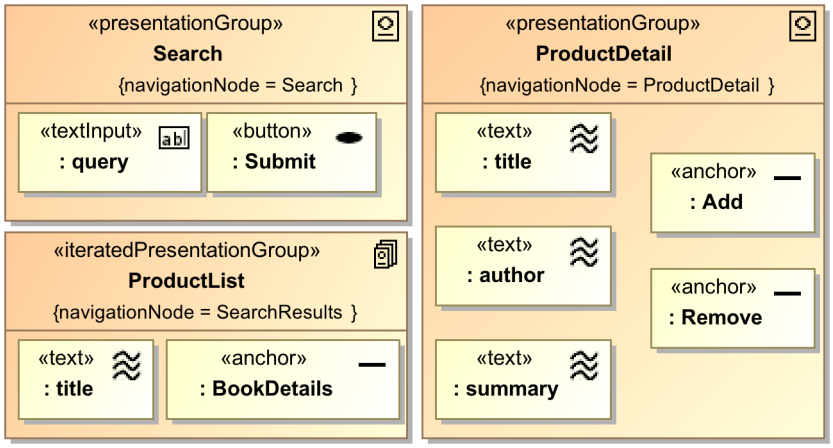


Abbildung A.4: *SimpleShop Präsentation: Search, SearchResult, ProductDetail*

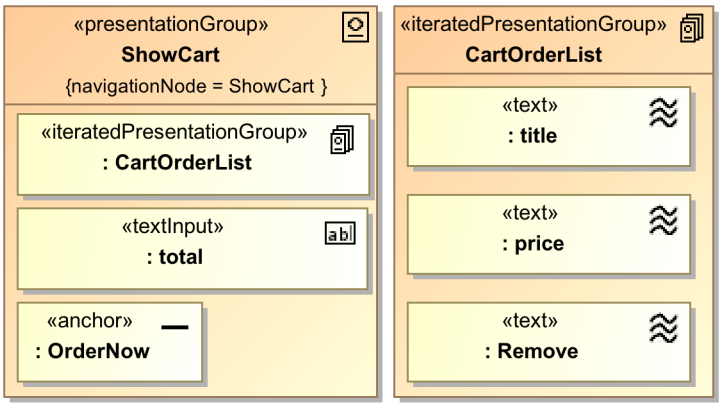


Abbildung A.5: *SimpleShop Präsentation: ShowCart, CartOrderList*

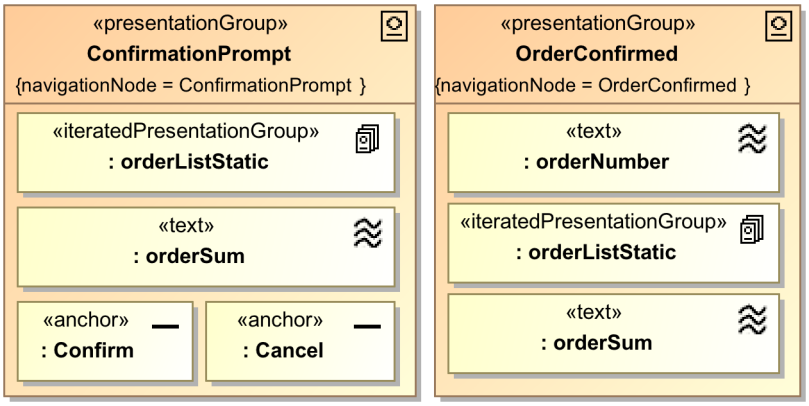


Abbildung A.6: *SimpleShop Präsentation: ConfirmationPrompt, OrderConfirmed*

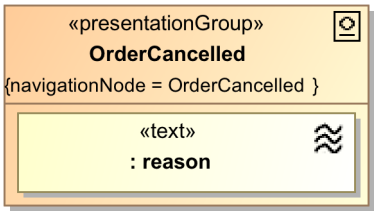


Abbildung A.7: *SimpleShop Präsentation: OrderCancelled*

Typ	TestExecutionAction	Parameter(in)	Parm. (out)
Assert	assertNavigationNode	node=LoginInput	
Exec	type	uiElement=LoginPage.user, text=data.GoodUser.username	
Exec	type	uiElement=LoginPage.pass, text=data.GoodUser.password	
Exec	click	uiElement=LoginPage.Submit	
Exec	waitForPageToLoad	-	
Assert	assertNavigationNode	node=LoginInput	
Exec	dataBySelector	dataSelector=getPosProduct	a
Exec	sendKeys	uiElement=Search.query, text=a.title	
Exec	click	uiElement=Search.Submit	
Exec	waitForPageToLoad	-	
Assert	assertNavigationNode	node=ProductList	
Assert	assertElementPresent	uiElement=ProductList[0]	
Assert	assertElementText	uiElem=ProductList[0].title, text=a.title	
Assert	assertElementPresent	uiElem=ProductList[0] .productDetails	
Exec	click	uiElem=ProductList[0] .productDetails	
Exec	waitForPageToLoad	-	
Assert	assertNavigationNode	node=ProductDetails	
Verify	verifyTextPresent	text=a.title	c1
Verify	verifyElemTextPresent	uiElement=ProductDetails .description, text=a.description	c2
Verify	verifyElementPresent	uiElement=ProductDetails.Add	c3
Case	if	c1 && c2 && c3	
Verdict	setVerdict	PASS	
Case	endif	-	
Case	if	!(c1 && c2 && c3)	
Verdict	setVerdict	FAIL	
Case	endif	-	

Tabelle A.1: *Transformierter Testfall searchProductTest*

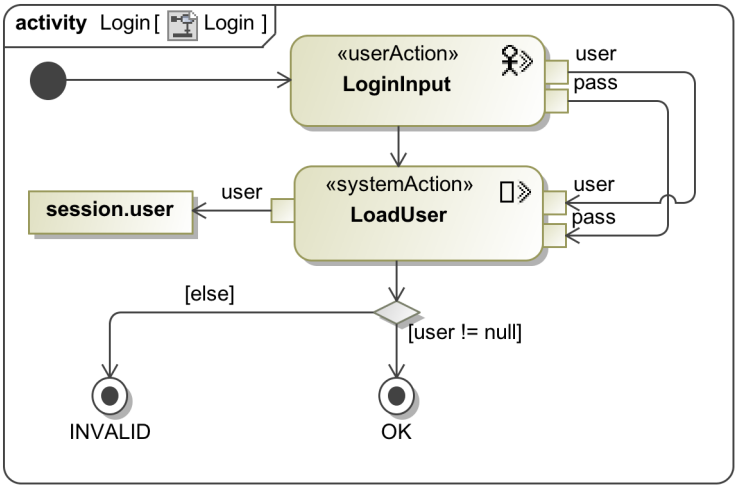


Abbildung A.8: *SimpleShop Prozess: Login*

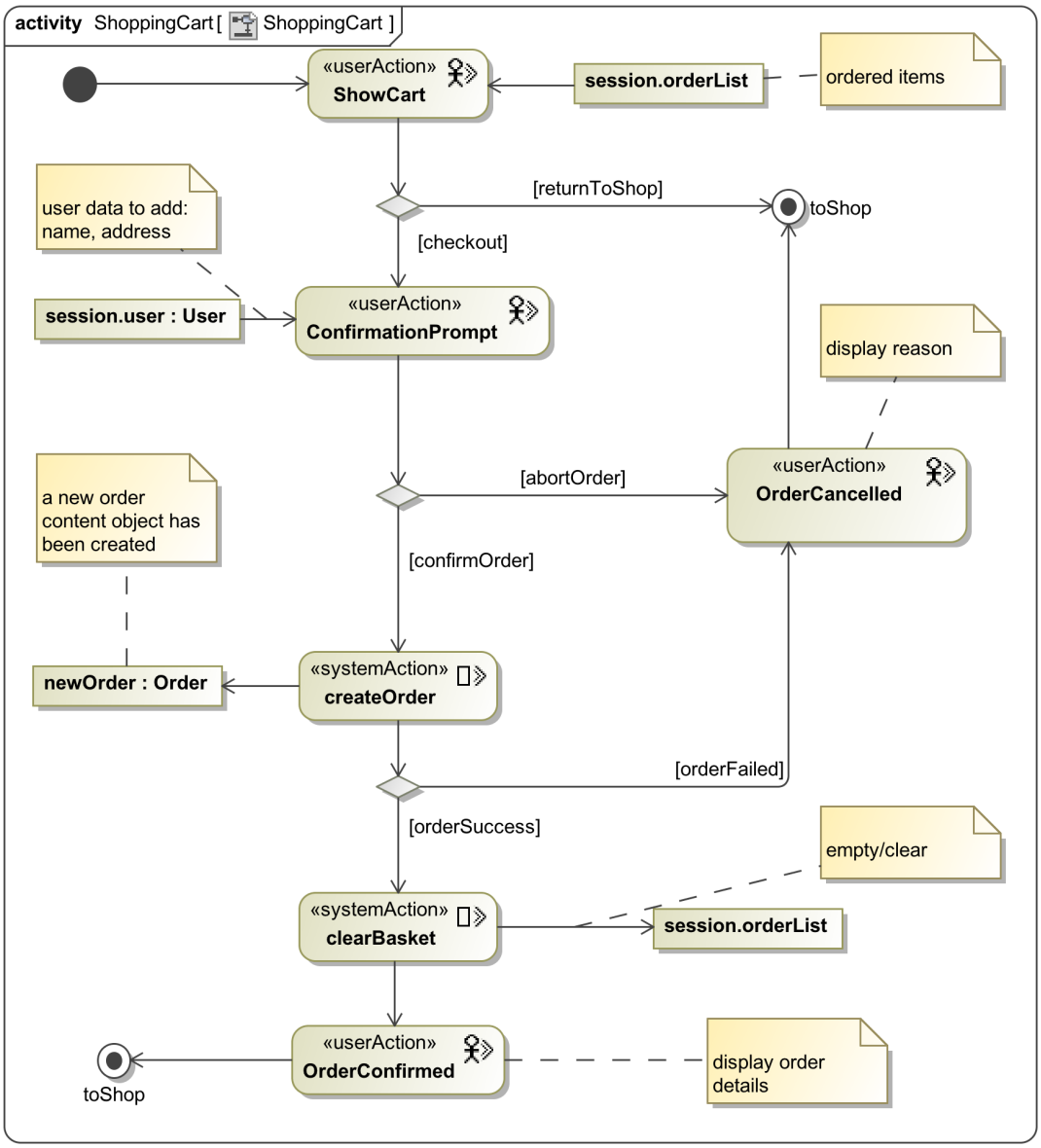


Abbildung A.9: *SimpleShop Prozess: Checkout*

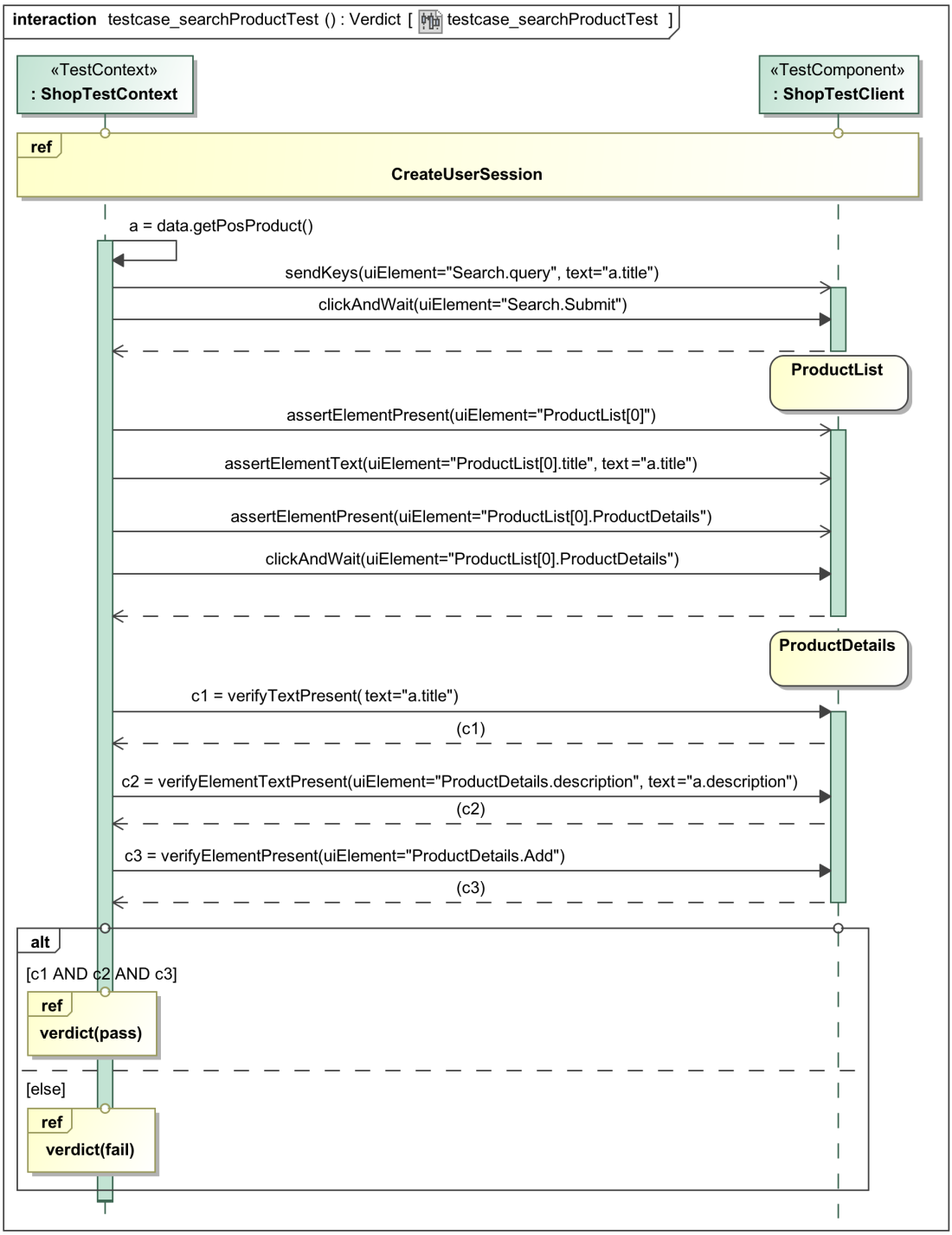


Abbildung A.10: *Verhaltesspezifikation für Testfall SearchProductTest*

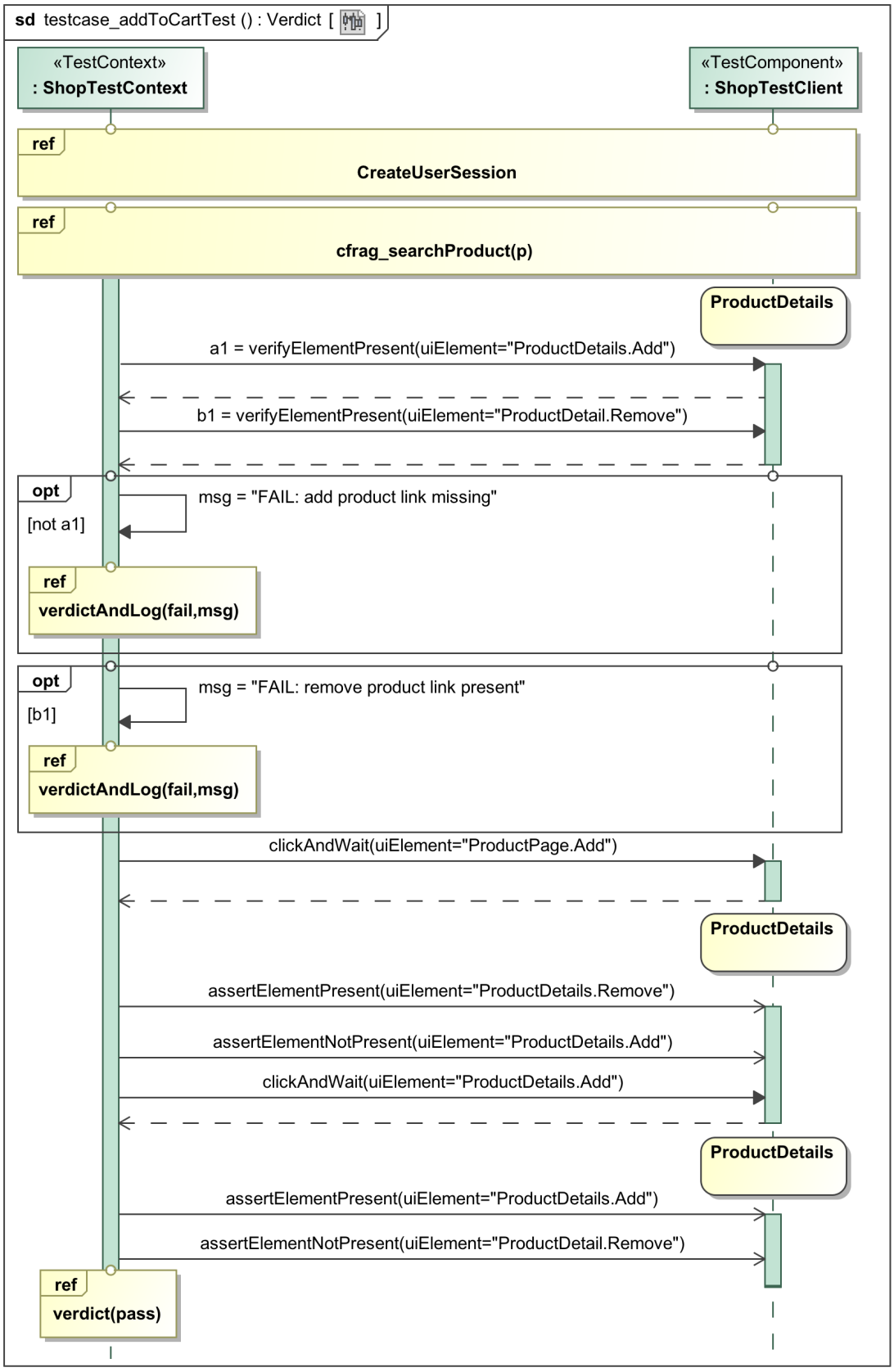


Abbildung A.11: *Verhaltesspezifikation für Testfall addToCartTest*

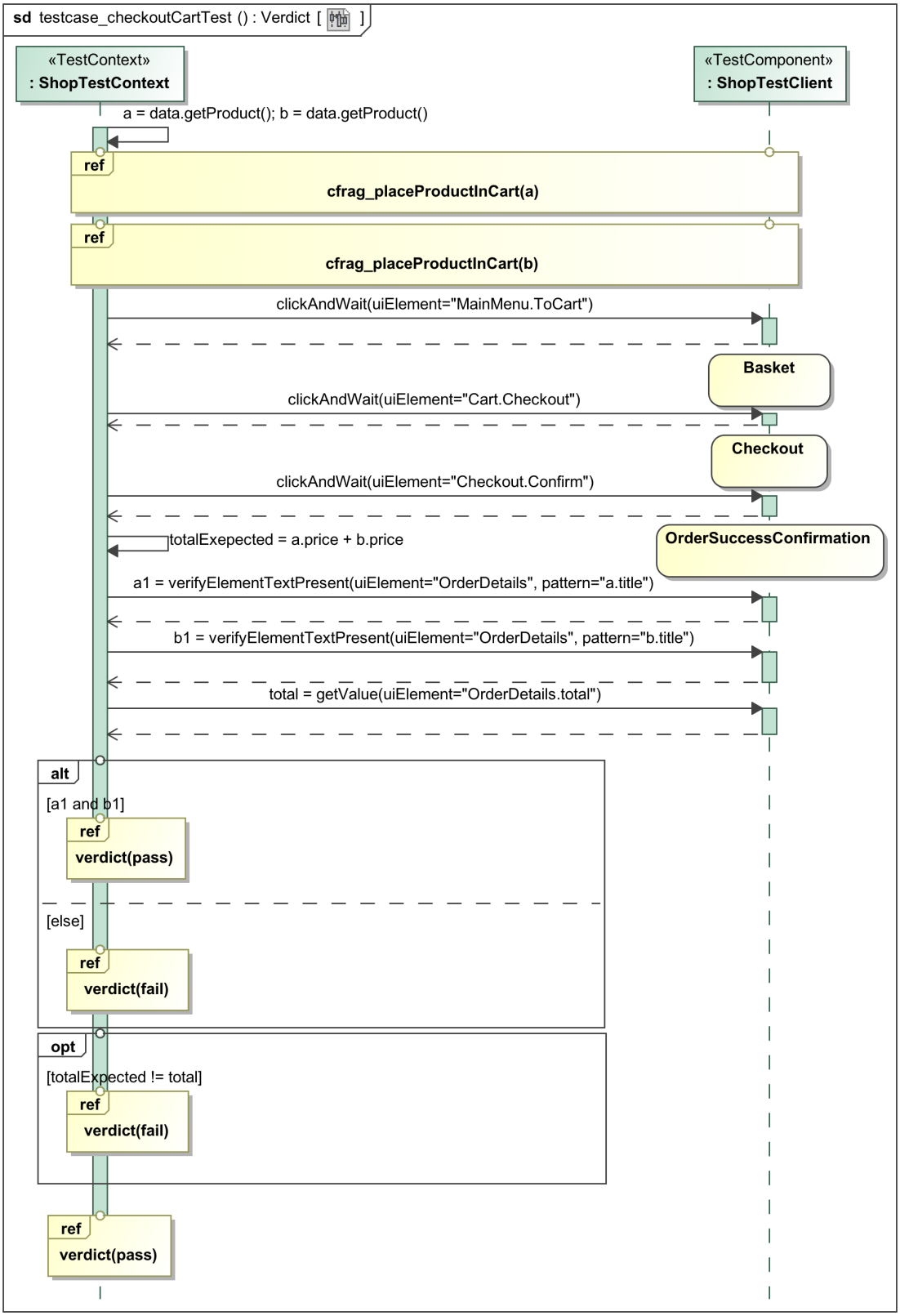


Abbildung A.12: *Verhaltesspezifikation für Testfall checkoutCartTest*

Abbildungsverzeichnis

Tabellenverzeichnis

Inhalt der beigelegten CD

Die beigelegte CD enthält folgenden Inhalt:

- diese Diplomarbeit in PDF Format,
- UWE Metamodell mit Testerweiterung als MagicDraw Quelldatei,
- UWE Testing Profile als MagicDraw Datei,
- Spezifikation des Beispielsystems *SimpleShop*,
- Implementierung des Beispielsystems,
- Kopie sämtlicher verwendeter Eclipse-Projekte (enthält Transformationen).

Literaturverzeichnis

[ABC82] Adrion, W. R. ; Branstad, Martha A. ; Cherniavsky, John C.: Validation, Verification, and Testing of Computer Software. In: *ACM Comput. Surv.* 14 (1982), Nr. 2, S. 159–192

[Bai02] Baisley, Donald E.: *Method in a computing system for comparing XMI-based XML documents for identical contents.* Dezember 31 2002

[Bak09] Baker, Paul: *Model-Driven Testing: Using the UML Testing Profile.* Berlin, Heidelberg : Springer-Verlag, 2009

[BCD+06] Busch, M ; Chaparadza, R ; Dai, ZR ; Hoffmann, A ; Lacmene, L ; Ngwangwen, T ; Ndem, GC ; Ogawa, H ; Serbanescu, D ; Schieferdecker, I u. a.: Model transformers for test generation from system models. In: *Proceedings of Conquest 2006, 10th International Conference on Quality Engineering in Software Technology*, 2006

[BK09] Busch, Marianne ; Koch, Nora: MagicUWE–A CASE Tool Plugin for Modeling Web Applications. In: *Web Engineering.* Springer, 2009, S. 505–508

[BSKH05] Born, Marc ; Schieferdecker, Ina ; Kath, Olaf ; Hirai, Chiaki: Combining system development and system test in a model-centric approach. In: *Rapid Integration of Software Engineering Techniques.* Springer, 2005, S. 132–143

[Bud04] Budinsky, Frank: *Eclipse modeling framework: a developer's guide.* Addison-Wesley Professional, 2004

[Dai04] Dai, Zhen R.: Model-Driven Testing with UML 2.0 / Computing Laboratory, University of Kent. 2004. – Forschungsbericht

[Dij72] Dijkstra, Edsger W.: The humble programmer. In: *Communications of the ACM* 15 (1972), Nr. 10, S. 859–866

[ecl13a] *Eclipse homepage - Atlas Transformation Language.* `http://www.eclipse.org/atl/`. Version: Dezember 2013

[ecl13b] *Website: Eclipse Test & Performance Tools Platform Project.* `http://www.eclipse.org/tptp/`. Version: Dezember 2013

[ecl14] *The Eclipse Project.* `http://www.eclipse.org/`. Version: Januar 2014

[Gro03] Group, Object M.: *Meta Object Facility (MOF) 2.0 Core Specification.* 2003. – Version 2

[Güd10] Güdali, Baris: Starthilfe für modellbasiertes Testen: Entscheidungsunterstützung für Projekt- und Testmanager. In: *SIGS DATACOM* (2010), März

[HL03] Heckel, Reiko ; Lohmann, Marc: Towards model-driven testing. In: *Electronic Notes in Theoretical Computer Science* 82 (2003), Nr. 6, S. 33–43

[htm13] *Website: HtmlUnit.* `http://htmlunit.sourceforge.net/`. Version: Dezember 2013

[htt13] *Website: HttpUnit.* `http://httpunit.sourceforge.net/`. Version: Dezember 2013

[JAB+06] Jouault, Frédéric ; Allilaire, Freddy ; Bézivin, Jean ; Kurtev, Ivan ; Valduriez, Patrick: ATL: a QVT-like transformation language. In: *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications* ACM, 2006, S. 719–720

[jun13] *Website: JUnit.* `http://junit.org/`. Version: Dezember 2013

[KK12] Koch, Nora ; Kozuruba, Sergej: Requirements models as first class entities in model-driven web engineering. In: *Current Trends in Web Engineering.* Springer, 2012, S. 158–169

[KKK09] Kroiss, Christian ; Koch, Nora ; Knapp, Alexander: Uwe4jsf: A model-driven generation approach for web applications. In: *Web Engineering.* Springer, 2009, S. 493–496

[KKK11] Kroiss, Christian ; Koch, Nora ; Kozuruba, Sergej: UWE Metamodel and Profile - User Guide and Reference / Ludwig-Maximilians-Universität München. 2011. – Forschungsbericht

[Koc01] Koch, Nora: *Software Engineering for Adaptive Hypermedia Systems: Reference Model, Modeling Techniques and Development Process.* Uni-Druck Verlag, 2001. – PhD. Thesis, Ludwig-Maximilians-Universität München

[Kro08] Kroiss, Christian: *Modellbasierte Generierung von Web-Anwendungen mit UWE.* 2008

[mag13] *Magicdraw UML, No Magic Inc.* `http://www.magicdraw.com/`. Version: Dezember 2013

[MSB11] Myers, Glenford J. ; Sandler, Corey ; Badgett, Tom: *The art of software testing.* John Wiley & Sons, 2011

[mve13] *MVFLEX Expression Language.* `http://mvel.codehaus.org/`. Version: Dezember 2013

[OMG05] OMG, UML: testing profile Version 1.0. In: *OM Group, Editor* (2005)

[OMG10] *OMG Systems Modeling Language (OMG SysML) 1.2 Specification.* 2010. – Version 1.2

[PM06] Petrasch, Roland ; Meimberg, Oliver: *Model Driven Architecture: eine praxisorientierte Einführung in die MDA.* dpunkt Verlag, 2006

[PP05] Pretschner, Alexander ; Philipps, Jan: 10 Methodological Issues in Model-Based Testing. In: *Model-based testing of reactive systems*. Springer, 2005, S. 281–291

[RJB04] Rumbaugh, James ; Jacobson, Ivar ; Booch, Grady: *Unified Modeling Language Reference Manual, The (2nd Edition)*. Pearson Higher Education, 2004

[RQZ07] Rupp, Chris ; Queins, Stefan ; Zengler, Barbara: *UML 2 glasklar: Praxiswissen für die UML-Modellierung*. 3. München : Hanser, 2007

[Sch07] Schieferdecker, Ina: Modellbasiertes Testen. In: *OBJEKTspektrum* (2007)

[sel13] *Website: Selenium*. `http://www.seleniumhq.org/`. Version: Dezember 2013

[Som13] Sommer, Oliver: *UWE MBT - Modellbasiertes Testen von Webanwendungen*. 2013

[UML13] *UML Testing Profile (UTP) 1.2 Specification*. 2013. – Version 1.2

[uwe13] *Website: UML-Based Web Engineering*. `http://uwe.pst.ifi.lmu.de/`. Version: Februar 2013

[ZDSD05] Zander, Justyna ; Dai, ZhenRu ; Schieferdecker, Ina ; Din, George: From U2TP Models to Executable Tests with TTCN-3 - An Approach to Model Driven Testing. In: *Testing of Communicating Systems* Bd. 3502. Springer Berlin Heidelberg, 2005, S. 289–303