Masterarbeit

am Institut für Informatik der Freien Universität Berlin,
Arbeitsgruppe Software Engineering

Erstellt am Fraunhofer-Institut für Offene Kommunikationssysteme,
Kompetenzzentrum MOTION (Modeling and Testing for System and Service Solutions)

Testobjektspezifische Generierung von JUnit-Testframeworks aus UTP-Modellen

Michael Prüm
michaelpruem@gmail.com
Berlin, 1. Dezember 2013

Gutachterinnen
Prof. Dr.-Ing. Ina Schieferdecker
Prof. Dr. Elfriede Fehr

Betreuer
Dipl.-Inform. (FH), M.Sc. Marc-Florian Wendland
Dipl.-Inform. Max Bureck

Kurzfassung

Diese Arbeit beschreibt die Konzeption und die Implementierung eines Generators, der aus (semi-)formalen Modellen ein testobjektspezifisches Testausführungsframework und den entsprechenden Testcode generiert. Der generierte Testcode kann mit dem weit verbreiteten Testframework JUnit ausgeführt werden. Der Codegenerator ist in das proprietäre Werkzeug Fokus!MBT eingebunden und nutzt zur Generierung dessen Testmodelle. Die logischen Testfälle werden als simplifizierte UML-Sequenzdiagramme beschrieben, angereichert mit dedizierten UTP-Testkonzepten. Aufgrund der Abstrahierung von technischen Details im Testmodell, erfolgt die Kommunikation zwischen dem Testcode und der konkreten SUT über eine Adapterschicht, in der eine logische-technische Abbildung durchgeführt werden muss. Die Schnittstellen zur konkreten SUT können im Adapter erst implementiert werden, wenn der zu testende Code letztendlich vorliegt.

Eigenständigkeitserklärung

Hiermit erkläre ich, dass ich die Masterarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die aus fremden Quellen direkt oder indirekt übernommenen Gedanken als solche kenntlich gemacht habe. Die Masterarbeit habe ich bisher keinem anderen Prüfungsamt in gleicher oder vergleichbarer Form vorgelegt. Sie wurde bisher auch nicht veröffentlicht.

Berlin, 1. Dezember 2013

_____

(Michael Prüm)

Danksagung

An dieser Stelle möchte ich mich bei allen Personen bedanken, die mich bei der Erstellung dieser Arbeit unterstützt haben.

Zunächst bedanke ich mich bei Marc-Florian Wendland und Max Bureck vom Kompetenzzentrum MOTION des Fraunhofer-Instituts für Offene Kommunikationssysteme in Berlin für die umfangreiche, fachliche und persönliche Betreuung. Sie standen jederzeit für Fragen und Diskussionen bereit und haben mich bei der Erstellung dieser Masterarbeit mit zahlreichen Tipps unterstützt.

Darüber hinaus bedanke ich mich bei Prof. Dr.-Ing. Ina Schieferdecker für die Bereitstellung des Themas.

Konventionen

- Begriffe, die für das Verständnis dieser Arbeit notwendig sind, werden am Ende dieser Arbeit im **Glossar** (siehe Seite 77) beschrieben. Neben der deutschen wird immer auch die englische Bezeichnung genannt, da diese oft geläufiger ist oder eher in der Literatur zu finden ist. Bei einigen Begriffen erfolgt die Definition in englischer Sprache, wie in der entsprechenden Literatur angegeben.
- Die Formatierung von Quelltext ist an die Darstellung der *Eclipse* IDE angelehnt.
- In den Quelltexten werden irrelevante Teile aufgrund einer besseren Lesbarkeit mit dem Java-Kommentar //... abgekürzt.
- Java-Klassen bei denen kein vollständiger Pfad angegeben ist, stammen aus dem Paket *java.lang*.
- Als Grundlage für die Erstellung dieser Arbeit wurden eine Reihe von Techniken und Standards verwendet. Im Folgenden sind diese mit der Angabe der verwendeten Version in alphabetischer Reihenfolge aufgelistet:
  - **Acceleo**, Version 3.4.1
  - **Eclipse**, Version 3.7 (Indigo), mit Vorbereitungen für Version 4.3 (Kepler)
  - **Java**, Version 6 Update 45, mit Vorbereitungen für Version 7
  - **JUnit**, Version 4.11
  - **UML**, Version 2.3, mit Vorbereitungen für Version 2.4.1
  - **UTP**, Version 1.2

Inhaltsverzeichnis

1 Einleitung **1**

2 Stand der Technik **4**
2.1 Modellbasierte (Software-) Entwicklung 4
2.1.1 Der Modellbegriff 4
2.1.2 Modellgetriebene Softwareentwicklung 4
2.1.3 Unified Modeling Language (UML) 6
2.2 Testen von Software 9
2.2.1 Testen und Qualität 10
2.2.2 Testen in Softwareentwicklungsmodellen 12
2.2.3 Modellbasiertes Testen (MBT) 15
2.3 UML Testing Profile (UTP) 16
2.3.1 Typbibliothek 17
2.3.2 Testarchitektur 19
2.3.3 Testverhalten 20
2.3.4 Testdaten 22
2.3.5 Testmanagement 24
2.4 Testautomatisierungswerkzeuge 24
2.4.1 Fokus!MBT 27
2.4.2 TTCN-3 29
2.4.3 JUnit 29

3 Problemstellung und Zielsetzung **34**
3.1 Grenzen der Automatisierung 36

4 Einführung in die Fokus!MBT-Modellierungsmethodik **37**
4.1 Testfallmodellierung mit Fokus!MBT 38
4.2 Problemanalyse bzgl. der JUnit-Generierung 39

5 Konzeption **43**
5.1 Testausführungskomponente 43
5.2 Testdaten 45
5.3 Komparatoren 46
5.4 Testarbitrierung 49
5.5 Logisch-technische Abbildung 50
5.6 Logging 51

6 Implementierung **55**
6.1 Acceleo 56
6.2 Testcodegenerierung 57
6.3 Validierung des Codegenerators 58

7 Fallstudie **61**

8 Fazit **74**
8.1 Ausblick 75

1 Einleitung

Softwaresysteme sind heutzutage in immer mehr Bereichen zu finden. Sei es beim Arbeiten im Büro, im Straßenverkehr, beim Arzt, bei der Verwendung von mobilen Geräten oder beim nächsten Geldautomat. „Wie sehr unser gegenwärtiger Alltag durch den Computer wirklich geprägt wird, zeigt eine Studie aus dem Jahr 2002. In jenem Jahr kam jeder US-Bürger täglich mit rund 150 verschiedenen Mikroprozessoren in Kontakt - Tendenz steigend. Der Internet-Kühlschrank und die MP3-Jacke sind zwei aktuelle Beispiele, die diesen Trend mit Leben füllen“ ([Hof08], Seite 2).

Die dazu notwendige Software wird immer komplexer, muss in einer immer kürzen Zeit auf den Markt gebracht werden und dabei noch hohen Qualitätsansprüchen genügen. Um dies zu erreichen sind zum einen strukturierte Entwicklungsprozesse und zum anderen systematische Qualitätssicherungsmaßnahmen, insbesondere das Testen der Software, notwendig.

Nach Utting, Pretschner und Legard (2006) steht die Qualitätssicherung[1] von modernen Softwaresystemen heutzutage noch immer vor großen Herausforderungen: Die Ableitung und die Dokumentation der Testfälle ist häufig nicht nachvollziehbar, dadurch sind die Testziele[2] der Testfälle nicht wiederzuerkennen. Eine unzureichende Dokumentation der Testaktivitäten erschwert die Kommunikation zwischen den beteiligten Akteuren. Die Prozessstrukturen sind unflexibel und oft mangelt es an der Automatisierung bezüglich Testentwurf und Testauswertung (vgl. [UPL06]).

In den vergangenen Jahren wurden zahlreiche Forschungsprojekte durchgeführt, die auf eine systematischere Ausführung des Testentwurfs mittels (semi-)formaler Modelle[3] abzielen. Ein derart aufgesetzter Testentwurfsprozess wird gemeinhin *modellbasiertes Testen* (MBT) genannt. Beim MBT werden alle testrelevanten Informationen in Testmodellen zusammengefasst. Dadurch ist eine Automatisierung von Testaktivitäten, wie die Generierung von Testfällen, leichter möglich. Die Modelle dienen gleichzeitig als Dokumentation und können einfach ausgetauscht und wiederverwendet werden. So wird der Entwicklungsprozess optimiert, die Softwarequalität verbessert und der zeitliche und finanzielle Aufwand reduziert. Das modellbasierte Testen versucht den Grad der Automatisierung zu erhöhen. Dazu zählt die automatische Generierung von Testfällen aus Testmodellen und Testfallspezifikationen[4]. Es existieren bereits eine Reihe von Werkzeugen die aus Testfällen den entsprechenden Testcode einer konkreten Sprache, wie der *Testing and Test Control Notation* (TTCN-3), generieren.

Das Ziel dieser Arbeit besteht aus der Generierung eines testobjektspezifischen JUnit-Testausführungsframeworks (**JUTP**, Akronym aus JUnit und UTP).

[1] „Teil des Qualitätsmanagements, das darauf gerichtet ist, Vertrauen in die Erfüllung der Qualitätsanforderungen zu erzeugen“ ([IG13], Seite 43).

[2] „Ein Grund oder Zweck für den Entwurf und die Ausführung von Tests“ (vgl. [IG13], Seite 61).

[3] Im weiteren Verlauf wird nur noch von formalen Modellen gesprochen, wobei meist die semi-formalen Modelle der UML gemeint sind. Im Glossar auf Seite 77 ist eine Unterscheidung zwischen formalen und semi-formalen Modellen zu finden.

[4] Der Begriff *Testfallspezifikation* ist auf Seite 80 definiert.

JUTP besteht aus einem Codegenerator, der aus formalen Modellen ausführbaren Testcode und ein dazugehöriges Testausführungsframework erzeugt. Die Kommunikation zwischen den Testfällen und dem Testobjekt (SUT, engl.: system under test) findet über eine Adapterschicht statt. Im Gegensatz zu TTCN-3 wird der Adapter abhängig von der modellierten SUT generiert. Durch die Anpassung an das Testobjekt soll die Komplexität des Adapters reduziert und dessen Implementierung vereinfacht werden. Ändert sich die Definition des zu testenden Systems, passt sich der daraus resultierende Generierungscode automatisch den Änderungen an.

Der Testcode soll mit Hilfe von JUnit ausgeführt werden können. JUnit ist der de facto Standard für die Implementierung von technischen Entwicklertests mit Java und ist in allen modernen Entwicklungsumgebungen wie Eclipse und Netbeans integriert (vgl. [Wes05], Seite 21). JUnit fehlen allerdings eine Reihe von wichtigen Funktionen. Es ist nicht möglich die Testausführungsergebnisse individuell zu protokollieren und JUnit besitzt keine Kenntnisse von Konzepten des *UML Testing Profile* (UTP). Deshalb wird zusätzlich zu dem eigentlichen Testcode ein Framework erzeugt, der diese Nachteile beseitigt.

Der Codegenerator wird in das proprietäre Werkzeug *Fokus!MBT* (vgl. [WHS13]) eingebunden. Bereits in einer frühen Softwareentwicklungsphase werden die testrelevanten Informationen modelliert. Fokus!MBT nutzt für die Spezifikation formaler Modelle die Modellierungssprachen UTP, eine testspezifische Erweiterung zur *Unified Modeling Language* (UML). Der Testgenerator erzeugt dann aus diesen Testmodellen Tests in einem hohen Abstraktionsniveau. Dadurch können die Testaktivitäten bereits sehr früh begonnen werden. Stehen dann die konkreten Schnittstellendefinitionen des Testobjekts zur Verfügung, müssen, vor der Testausführung, lediglich diese Schnittstellen implementiert werden.

Die Ausführung der Testfälle findet dabei außerhalb der Fokus!MBT-Umgebung statt. Die Testausführung soll geeignet protokolliert werden, damit eine Rückspielung der Ergebnisse in das Fokus!MBT-Werkzeug möglich ist. Der Import der Testausführungsergebnisse ist dabei nicht Teil dieser Arbeit.

In Kapitel 2 werden die theoretischen Grundlagen und die in dieser Arbeit verwendeten Techniken und Konzepte beschrieben. Dazu gehört eine Definition des Modellbegriffs, einen Darstellung was unter der modellbasierten Softwareentwicklung und dem modellbasierten Testen zu verstehen ist und welche Vor- und Nachteile sich aus diesen ergeben. Der Zusammenhang zwischen dem Testen einer Software und dessen Qualität und die Durchführung des Testens in Softwareentwicklungsmodellen werden erläutert. Darüber hinaus werden eine Reihe von Testautomatisierungswerkzeugen vorgestellt und deren Funktionalität beschrieben, insbesondere Fokus!MBT und JUnit. Außerdem werden die Konzepte der Modellierungssprachen UML und UTP erörtert, die für die Erstellung dieser Arbeit essenziell sind.

Die Problemstellung und Zielsetzung für die Erstellung dieser Arbeit erfolgt in Kapitel 3. Hier wird erläutert, warum sich die Ableitung des Testcodes nicht vollständig automatisieren lässt und worin die Vorteile einer testobjektspezifischen Generierung liegen. Des Weiteren wird beschrieben, weshalb sich JUnit für die Testausführung des generierten Testcodes anbietet und wieso deshalb, neben dem Testcode, zusätzlich auch ein Testausfüh-

rungsframework generiert wird.

Kapitel 4 gibt eine Einführung in die Modellierungsmethodik von Fokus!MBT, speziell wie die Schnittstellen zur SUT beschrieben und wie Testdaten und Testfälle modelliert werden. Das so erstellte Testmodell dient als Grundlage für die Generierung des Testausführungsframeworks und des Testcodes. Darüber hinaus wird erläutert, welche Einschränkungen bei der Modellierung beachtet werden müssen, damit eine Ausführung des Testcodes durch JUnit möglich ist.

Der konzeptionelle Aufbau des testobjektspezifischen JUnit-Testausführungsframeworks und des Testcodes wird in Kapitel 5 beschrieben. Dazu gehören die Testausführungskomponente in der die gesamte Testfalllogik, in Form von JUnit-Testmethoden, integriert ist, Komparatoren für die Analyse des erwarteten SUT-Verhaltens, die Bereitstellung der Testdaten, eine Arbiterimplementierung zur Bestimmung des globalen Verdicts eines Testfalls und eine Reihe von Hilfsfunktionen.

In Kapitel 6 wird die Implementierung des Generators erläutert. Hierzu erfolgt eine kurze Beschreibung über den internen Aufbau von Fokus!MBT und wie der implementierte Generator in dieses integriert wird. Des Weiteren erfolgt eine Erklärung des Codegenerators Acceleo, mit dessen Hilfe die eigentliche Erzeugung des Testcodes und des Frameworks durchgeführt wird, und der Schwierigkeiten bei der Validierung des Generators.

Um die praktische Anwendung von JUTP zu demonstrieren, wird in Kapitel 7 eine reale Fokus!MBT-Komponente mit Hilfe des generierten Frameworks getestet. Dazu werden, anhand eines Beispiels, alle Aktionen durchgeführt, die notwendig sind, um die JUnit-Testfälle erzeugen und ausführen zu können.

In einem letzten Schritt werden in Kapitel 8 die wichtigsten Ergebnisse noch einmal zusammengefasst betrachtet. Es wird ebenso ein Blick auf weitere Forschungsmöglichkeiten gerichtet um JUTP weiter zu optimieren.

2 Stand der Technik

Dieses Kapitel beschreibt den theoretischen Hintergrund, der für die Generierung eines testobjektspezifischen JUnit-Testausführungsframeworks aus UTP-Modellen notwendig ist. Neben der Beschreibung, was unter der modellbasierten Softwareentwicklung zu verstehen ist, wird auch eine Einführung in die UML gegeben. Danach wird erläutert, was das Testen von Software ist, wie das Testen in Softwareentwicklungsmodellen, z.B. dem V- und dem W-Modell, integriert ist, und wie sich mit Hilfe von Modellen Testfälle automatisiert ableiten lassen. Im Abschnitt 2.3 erfolgt eine detaillierte Erläuterung über den UTP-Sprachumfang, dessen Konzepte in dieser Arbeit für die Codegenerierung verwendet werden. Zum Schluss wird in diesem Kapitel ein Überblick über eine Reihe von unterschiedlichen Testautomatisierungswerkzeugen gegeben. In dieser Arbeit wird die Testmodellierungsumgebung Fokus!MBT zur modellbasierten Testfallspezifikation verwendet. Fokus!MBT ist ein proprietäre Entwicklung des Fraunhofer Institut Fokus (Fraunhofer Institut für Offene Kommunikationssysteme), welches konzeptuell auf UTP und technisch auf Eclipse *Papyrus* (vgl. [Ecl13d]) aufsetzt. Darüber hinaus wird das Testframework JUnit erläutert, mit dem der generierte Testcode ausgeführt werden kann.

2.1 Modellbasierte (Software-) Entwicklung

2.1.1 Der Modellbegriff

Nach Herbert Stachowiak ist ein Modell durch drei Hauptmerkmale gekennzeichnet (vgl. [Sta73]):

- **Abbildung**: Ein Modell ist eine Abbildung oder eine Repräsentation eines natürlichen oder künstlichen Originals, wobei dieses selbst wieder ein Modell sein kann.
- **Verkürzung**: Es werden generell nicht alle Merkmale des Originals betrachtet, sondern nur die im jeweiligen Kontext benötigten.
- **Pragmatismus**: Modelle dienen bestimmten Empfängern, sind innerhalb eines bestimmten Zeitintervalls gültig und dienen einem eindeutigem Zweck.

Ein Modell ist somit eine eingeschränkte Abbildung oder eine Vorwegnahme der Wirklichkeit. Es werden nur die gerade benötigten Merkmale des Originals betrachtet. Durch diese Abstraktion von irrelevanten Details lässt sich die Komplexität verbergen und somit beherrschbar machen. Meistens ist es dann erst möglich, sich einen Überblick über das gesamte System zu verschaffen.

2.1.2 Modellgetriebene Softwareentwicklung

In der Softwaretechnik existieren eine Reihe von Paradigmen und Techniken, die bei der Entwicklung mit unterschiedlichen Arten von Modellen verwendet werden.

Beim modellbasierten Engineering *MBE* (engl.: model-based engineering) werden die Modelle in der Regel für den Entwurf und für die Dokumentation eines Softwaresystems erstellt. Die Modelle dienen dabei nicht unbedingt der automatisierten Generierung des Zielsystems, sondern werden meist nur als Grundlage für die manuelle Implementierung

durch Softwareentwickler erstellt. Notwendige Modelländerungen müssen auch immer in der Implementierung übernommen werden, um Inkonsistenzen zwischen Modell und Code zu vermeiden (vgl. [BCW12]).

Damit das Zielsystem, oder auch nur ein Teilsystem, automatisiert generiert werden kann, findet beim modellgetriebenen Engineering *MDE* (engl.: model-driven engineering) eine Formalisierung der Modelle statt. Dabei ist das MDE nur als Obermenge der modellgetriebenen Softwareentwicklung *MDSD* zu sehen, da das modellgetriebene Engineering über die reine Entwicklung hinaus geht (vgl. [BCW12]).

Nach Stahl et al. (2007) ist die modellgetriebene Softwareentwicklung *MDSD* (engl.: model-driven software development) „ein Oberbegriff für Techniken, die aus formalen Modellen automatisiert lauffähige Software erzeugen“ ([SVEH07], Seite 11). Ein Modell ist formal, wenn Syntax und Semantik präzise definiert sind. Für die Erzeugung von lauffähiger Software werden Generatoren benötigt. Ein Generator erzeugt aus einem Modell, Quelltext einer bestimmten Programmiersprache.

Bei der MDSD werden alle Änderungen direkt am Modell vorgenommen. So wird gewährleistet, dass diese immer den aktuellen Stand widerspiegeln. Aus dem Modell kann dann der Quelltext des Systems, oder zumindest ein Teil des Systems, automatisch generiert werden. Für eine automatische Codegenerierung müssen die Modelle und die Generatoren einen hinreichenden Detaillierungsgrad besitzen. Das formale Modell beschreibt die Struktur und das Verhalten des Systems, und der Generator ist für die technische Umsetzung verantwortlich (vgl. [SVEH07]).

Die modellgetriebene Architektur *MDA* (engl.: model-driven architecture) ist ein Softwareentwicklungsansatz der *OMG* (Object Management Group, vgl. [OMG13c]). Zunächst werden berechnungsunabhängige Modelle für eine nicht-technische Darstellung des Systems erzeugt. In einem zweiten Schritt erfolgt eine Transformation in ein plattformunabhängiges Modell der Geschäftslogik. Zum Schluss werden diese in ein plattformspezifisches Modell mittels weiterer Transformationen der eigentlichen Plattform[5] angenähert, das technische Informationen der konkreten Plattform beinhaltet. Zwischen diesen Abstraktionen finden jeweils *Model-to-Text*- und *Model-to-Model*-Transformation statt, was sich aber meistens nicht vollständig automatisieren lässt. Durch die Verwendung der unterschiedlichen Schichten lassen sich geschäftsrelevante und technische Informationen und somit auch die Komplexität des Gesamtsystems auf unterschiedliche Modelle verteilen (vgl. [OMG13d]).

Vorteile

Durch die Abstraktion von konkreten Technologien bzw. Implementierungen sind die modellierten Testfälle unabhängig von der tatsächlichen Implementierung der SUT, wodurch die Austauschbarkeit und die Wiederverwendbarkeit der Modelle vereinfacht werden. Mit Hilfe der modellgetriebenen Softwareentwicklung soll die Softwarequalität. Durch den

[5] Eine Plattform ist eine Menge von Teilsystemen oder Technologien, die eine kohärente Menge von Funktionalität durch Schnittstellen und spezifizierte Verwendungsmuster bereitstellen. Die Schnittstellen können dann ohne genau Kenntnisse von Implementierungsdetails verwendet werden (vgl. [MM03], Seite 13).

hohen Automatisierungsgrad werden Fehlhandlung[6] einer manuellen Implementierung reduziert. Eine qualitativ hochwertige Entwicklung ist aufgrund der hohen Komplexität heutiger Softwaresysteme meist erst durch die Modellunterstützung möglich. Durch eine Abstraktion wird versucht die Gesamtkomplexität beherrschbar zu machen (vgl. [SVEH07]).

Nachteile

Die Einführung der MDSD in einem Unternehmen ist nicht trivial. Es müssen Methoden und Abläufe der gesamten Softwareentwicklung angepasst werden. Daraus resultiert auch, dass die Mitarbeiter geschult und evtl. neue Softwarewerkzeuge für eine modellgetriebene Entwicklung angeschafft werden müssen. Fachliche Fehler lassen sich mit der MDSD nicht reduzieren, da sich Fehler im Modell auch auf den genierten Quellcode und somit auf die lauffähige Software auswirken (vgl. [SVEH07]).

2.1.3 Unified Modeling Language (UML)

Die *UML* (Unified Modeling Language) ist eine standardisierte Modellierungssprache zur Spezifikation, Visualisierung und Dokumentation von Systemmodellen[7]. Die UML wurde von der *OMG* (Object Management Group) entwickelt und liegt momentan in der Version 2.4.1 vor (vgl. [OMG13g]). Um die Komplexität einer universellen Sprache zu beherrschen, umfasst die UML nach Born, Holz und Kath (2004) eine Reihe von zentralen Paradigmen (vgl. [BHK04]):

1. Hierarchischer Aufbau der Sprache aus einer Reihe aufeinander aufbauenden oder einander ergänzenden Sprachdefinitionspaketen. Für den Einsatz der Sprache sind Kenntnisse über den Sprachkern und über die, für den jeweiligen Zweck, benötigten Sprachpakete erforderlich.

2. Verwendung von objektorientierten Konzepten, wie Klassifikation, Generalisierung, Strukturierung, Organisation und Datenkapslung als Grundlage der gesamten Sprache.

3. Eindeutige Trennung von Begriffswelt (Menge von Begriffen und Konstruktionsregeln) und Notation (physikalische Darstellung von Modellen als Diagramme), wodurch eine Erweiterung oder Einschränkung der Sprache für einen konkreten Anwendungsfall möglich ist.

4. Mit Hilfe der Metamodellierung lässt sich die UML mit ihren eigenen Sprachdefinitionen selbst erklären. Dadurch ist eine Erweiterbarkeit und Anpassbarkeit der Sprache, aber auch eine maschinelle Speicherung und Verarbeitung von Modellen und Metamodellen möglich.

5. Unterscheidung zwischen Modell (Sammlung von Konzepten aus der UML-Begriffswelt) und Diagramm (Repräsentation verschiedener Sichten auf das Modell), als Konsequenz aus Trennung von Begriffswelt und Notation.

[6] „Die menschliche Handlung, die zu einem falschen Ergebnis führt“ (vgl. [IG13], Seite 21).
[7] Was unter einem Systemmodell zu verstehen ist, wird auf Seite 78 beschrieben.

Die aktuellste UML-Version definiert vierzehn Diagrammtypen, die in drei Kategorien unterteilt sind (siehe Abbildung 1).

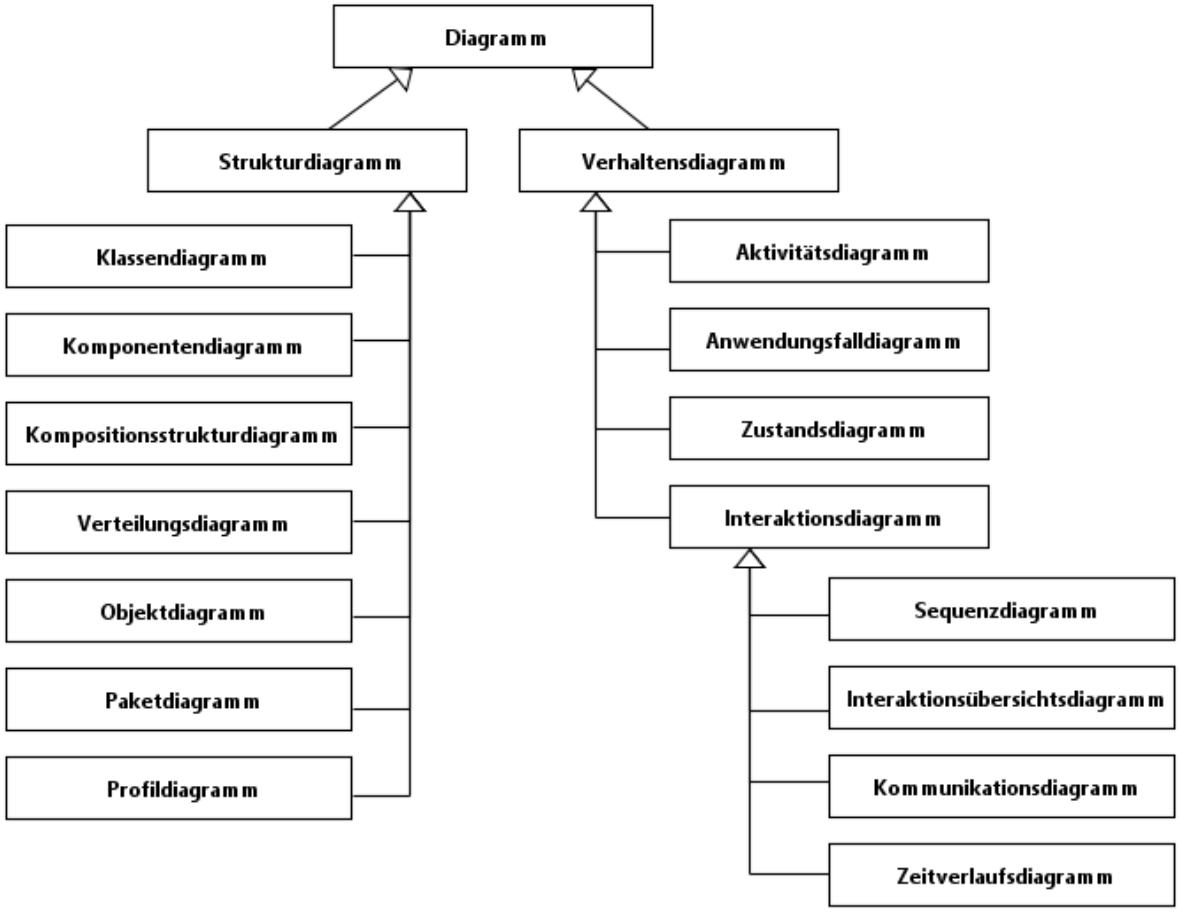

Abbildung 1: Taxonomie von Struktur- und Verhaltensdiagrammen der UML-Version 2.4.1

Mit Strukturdiagrammen werden die statischen Strukturen eines Systems und dessen Teile auf unterschiedlichen Abstraktions- und Implementierungsebenen dargestellt. Die Diagrammelemente repräsentieren die wesentlichen Konzepte einer Anwendung. Strukturdiagramme nutzen keine zeitbezogene Konzepte und beschreiben kein dynamisches Verhalten.

Verhaltensdiagramme zeigen das dynamische Verhalten eines Systems, einschließlich ihrer Methoden und Aktivitäten als eine Folge von Systemveränderungen im Laufe der Zeit.

Mit Hilfe von Interaktionsdiagrammen kann das Verhalten eines Systems durch Interaktionen zwischen Objekten beschrieben werden. Für diese Arbeit sind vor allem Sequenzdiagramme von Bedeutung. Ein Sequenzdiagramm beschreibt eine Interaktion durch eine Abfolge von Nachrichten, die im Verlauf einer Zeitspanne zwischen Objekten ausgetauscht werden.

Eine Interaktion besteht aus einer Menge von Lebenslinien. Eine Lebenslinie repräsentiert einen Teil eines Systems. Zwischen diesen Lebenslinien findet eine Kommunikation in Form von gesendeten Nachrichten statt.

Eine Nachricht wird in einem Sequenzdiagramm als Pfeil dargestellt, wobei die Kommunikationsrichtung durch die Pfeilspitze angegeben wird. Die Pfeillinie muss zwischen Sender und Empfänger entweder horizontal oder abfallend modelliert werden. Dies symbolisiert, dass eine Nachricht sich niemals in der Zeit zurück bewegen kann. Eine Nachricht benötigt entweder keine oder eine positive Anzahl einer Zeiteinheit, um beim Empfänger einzutreffen. Es existieren eine Reihe von unterschiedlichen Nachrichtentypen (siehe Abbildung 2):

- **synchronous call**: Ein *synchronous call* entspricht einem synchronen Operationsaufruf. Die Ausführung des aufrufenden Threads wird solange blockiert, bis das Operationsergebnis vorliegt. Synchrone Aufrufe werden mit einer durchgehenden Linie und einer ausgefüllten Pfeilspitze dargestellt.
- **asynchronous call**: Ein *asynchronous call* entspricht einem asynchronen Operationsaufruf. Der aufrufende Thread wird direkt im Anschluss fortgeführt. Asynchrone Aufrufe werden mit einer durchgehenden Linie und einer geöffneten Pfeilspitze modelliert.
- **asynchronous signal**: Ein *asynchronous signal* ist nicht mit einer Operation, sondern mit einem Signal assoziiert. Ein Signal kann beispielsweise ein WebService-Aufruf sein.
- **create**: Eine *create*-Nachricht wird zu einer Lifeline gesendet, um eine Instanz, des von der Lifeline repräsentierten Typs, zu erzeugen. Die Darstellung erfolgt mit einer gestrichelten Linie und einer geöffneten Pfeilspitze.
- **delete**: Die *delete*-Nachricht wird zum Beenden einer anderen Lifeline gesendet. Ab diesem Zeitpunkt ist mit dieser Lifeline keine Kommunikation mehr möglich.
- **reply**: Eine *reply*-Nachricht ist die Rückantwort zu einem synchronem Operationsaufruf. Dargestellt wird diese durch eine gestrichelte Linie mit einer geöffneten Pfeilspitze.

Mit Hilfe von kombinierten Fragmenten lassen sich alternative Abläufe, Iteration und paralleles Verhalten modellieren. Kombinierte Fragmente bestehen aus einem Interaktionsoperator, einem oder mehreren Interaktionsoperanden und einem optionalen Wächterausdruck. Es stehen folgende Interaktionsoperatoren zur Verfügung:

- **alt**: Definition von alternativen Abläufen. Jede Alternative besitzt einen Wächterausdruck. Kann der Ausdruck zu *wahr* ausgewertet werden, wird die Alternative betreten und ausgeführt. Wenn nicht, wird keiner der Operanden ausgeführt.
- **opt**: Eine Option bezeichnet eine Auswahl eines Verhaltens bei dem entweder der einzige Operand passiert wird oder der Operand wird übersprungen.
- **loop**: Der Wächter kann eine untere (*minint*) und eine obere (*maxint*) Anzahl von Iterationen sowie einen Booleschen-Ausdruck besitzen. Die Schleife wird mindesten *minint*-mal und höchstens *maxint*-mal ausgeführt. Nachdem die Mindestanzahl von Iterationen ausgeführt wurde und der Boolesche-Ausdruck als *falsch* ausgewertet werden kann, wird die Schleife beendet.

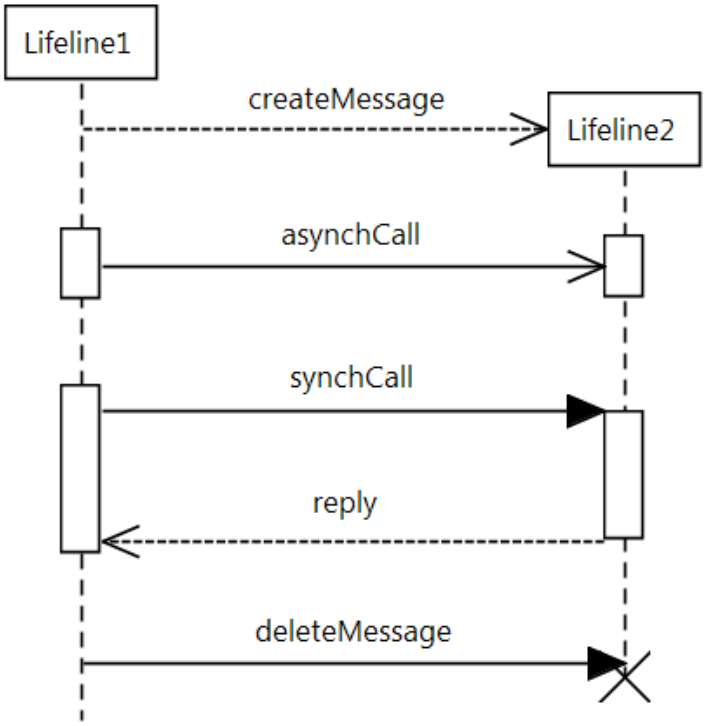

Abbildung 2: Unterschiedliche Nachrichtentypen der UML

- **par**: Mit dem *par*-Operator lässt sich eine parallele Verflechtung beschreiben. Eine parallele Verflechtung definiert alle möglichen Abfolgen in denen die Operanden durchlaufen werden können. Die Reihenfolge innerhalb der Operanden wird dabei aber eingehalten.

Des Weiteren existieren auch noch die Interaktionsoperanden *break*, *seq*, *strict*, *critical*, *ignore*, *consider*, *assert* und *neq* die hier aber nicht weiter beschrieben werden.

Über sogenannte *Gates* können Nachrichten aus anderen Interaktionen empfangen oder an andere Interaktionen gesendet werden. Außerdem können Verhaltensbeschreibungen in andere Interaktion dekomponiert werden. Dadurch kann die Logik auf das Nötigste beschränkt und die Komplexität auf mehrere Interaktionen verteilt werden (vgl. [OMG13g] und [Fak13]).

2.2 Testen von Software

In diesem Abschnitt wird der Zusammenhang zwischen dem Softwaretest und der Qualität einer Software erläutert. Darüber hinaus erfolgt eine Beschreibung, wie das Testen in Vorgehensmodellen durchgeführt wird, wie sich mit Hilfe von Modellen Testfälle automatisiert generieren lassen und welche Vor- und Nachteile dieser Ansatz gegenüber der herkömmlichen Softwareentwicklung besitzt.

2.2.1 Testen und Qualität

Unter *Testen* versteht man den „Prozess, der aus allen Aktivitäten des Lebenszyklus besteht (sowohl statisch als auch dynamisch), die sich mit der Planung, Vorbereitung und Bewertung eines Softwareprodukts und dazugehöriger Arbeitsergebnisse befassen. Ziel des Prozesses ist sicherzustellen, dass diese allen festgelegten Anforderungen[8] genügen, dass sie ihren Zweck erfüllen, und etwaige Fehlerzustände[9] zu finden“ ([IG13], Seite 65).

Das finale Softwareprodukt soll für den Benutzer einen bestimmten Zweck erfüllen. Der Zweck der Software wird in einer Anforderungsspezifikation definiert und dient als Prüfreferenz in der Entwicklung und beim Testen. Mit Hilfe des Testens wird evaluiert, ob das zu testende System den vorher definierten Anforderungen entspricht. Die Software muss die Kundenerwartungen und -bedürfnisse erfüllen. Durch das Testen wird also versucht die Qualität[10] einer Software zu messen.

Nach der Norm *ISO/IEC 25000* (ersetzt *ISO/IEC 9126*) setzt sich die Softwarequalität aus Gebrauchsqualität, innerer und äußerer Qualität zusammen (siehe Abbildung 3).

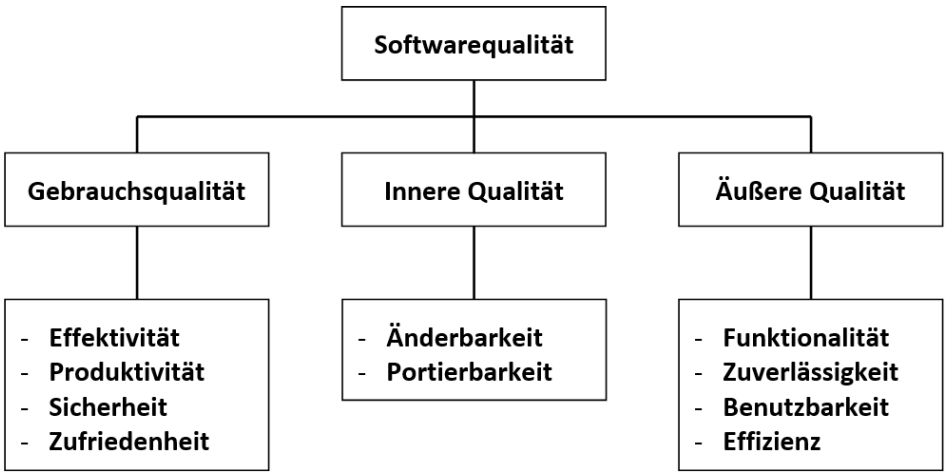

Abbildung 3: Softwarequalität nach *ISO/IEC 25000*

Die Gebrauchsqualität definiert Merkmale, mit denen der Benutzer der Software direkt konfrontiert wird. Dazu gehören Effektivität, Produktivität, Sicherheit und Zufriedenheit. Die Effektivität beschreibt die Genauigkeit und Vollständigkeit bei der Erreichung bestimmter Ziele durch einen Anwender. Der Benutzer der Software muss mittels der Software seine Aufgaben auch unter wirtschaftlichen Gesichtspunkten erfüllen können. Das heißt, dass Kosten- und Zeitaspekte eingehalten werden müssen. Darüber hinaus sollte von der

[8] „Eine vom Benutzer benötigte Eigenschaft oder Fähigkeit, die eine Software erfüllen oder besitzen muss, um einen Vertrag, einen Standard, eine Spezifikation oder ein anderes formales Dokument zu erfüllen“ ([IG13], Seite 46).

[9] Die Begriffe *Fehlerzustand* und *Defekt* werden in dieser Arbeit synonym verwendet (vgl. Seite 77).

[10] „Der Grad, in dem ein System, eine Komponente oder ein Prozess die Kundenerwartungen und -bedürfnisse erfüllt“. „Der Grad, in dem ein Satz inhärenter Merkmale Anforderungen erfüllt“ ([IG13], Seite 43).

Software keine Gesundheitsgefährdung für den Anwender ausgehen. Auch die subjektive Benutzerzufriedenheit sollte gegeben sein.

Die innere Qualität bezieht sich auf die Eigenschaften, die die Weiterentwicklung einer Software betreffen. Dazu zählen Änderbarkeit und Übertragbarkeit. Um Änderungen an einer Software durchführen, müssen die änderungsbedürftigen Teile einfach identifiziert werden können. So lassen sich Fehlerzustände leichter beseitigen oder Verbesserungen einfacher einbauen. Es muss möglich sein, eine Software auf verschiedenen Umgebungen anzupassen und zu installieren. Auch kann es erforderlich sein, dass von einer Software anwendungsspezifische Normen, Vereinbarungen oder gesetzliche Bestimmungen eingehalten werden muss.

Funktionalität, Zuverlässigkeit, Benutzbarkeit und Effizienz gehören zu den äußeren Qualitätsmerkmalen[11]. Die vorhandenen Funktionalitäten müssen mit den festgelegten Anforderungen übereinstimmen. Das bedeutet, dass die Funktionalität die vom Benutzer geforderten Aufgaben angemessen und richtig ausführen, zudem aber auch, dass nicht mehr implementiert wird als gefordert ist. Darüber hinaus muss die Funktionalität das Zusammenspiel mit anderer Software gewährleisten, falls es für die Software relevant ist mit anderen Systemen interoperabel zu sein, sich an anwendungsspezifische Normen, Vereinbarungen oder gesetzliche Bestimmungen halten und nicht autorisierte Zugriffe auf Programme oder Daten verhindern. Zur Zuverlässigkeit gehören Merkmale wie die Reife, Vermeidung von Fehlerwirkungen[12], Fehlertoleranz gegenüber Fehlfunktionen oder einer falschen Bedienung und die Möglichkeit das System bei einem Versagen wieder herzustellen. Zur Benutzbarkeit zählen die Verständlichkeit, Erlernbarkeit und Bedienbarkeit. Ein Anwender sollte ohne großen Aufwand in der Lage sein, die Software für seine Zwecke anzuwenden. Eine Software sollte darüber hinaus auch effizient arbeiten, also kurze Antwort- und Verarbeitungszeiten aufweisen und während der Ausführung so wenige Ressourcen wie möglich beanspruchen.

In Praxis ist es oft nicht möglich, alle Qualitätsmerkmale einer Software vollständig zu erfüllen. Daher muss in der Anforderungsspezifikation präzise, widerspruchsfrei, vollständig und korrekt definiert werden, in welchem Grad die einzelnen Merkmale zu erfüllen sind.

Mit Hilfe des Testens lässt sich die Softwarequalität oftmals erst abschätzen bzw. bestimmen. Fehlerzustände können vor der Fertigstellung erkannt und beseitigt werden. Gefundene Defekte werden zunächst analysiert, dann zur Nachverfolgung dokumentiert, wodurch Fehlhandlungen zukünftig vermieden werden können (vgl. [Sch11], Kapitel 2).

[11] „Fähigkeit oder Eigenschaft, welche die Qualität einer Einheit beeinflusst“. „Ein Satz von Eigenschaften eines Softwareprodukts, anhand dessen seine Qualität beschrieben und beurteilt wird. Ein Softwarequalitätsmerkmal kann über mehrere Stufen in Teilmerkmale verfeinert werden. Qualitätsmerkmale sind Funktionalität, Zuverlässigkeit, Benutzbarkeit, Effizienz, Änderbarkeit und Übertragbarkeit“ ([IG13], Seite 44).

[12] „Abweichung einer Komponente/eines Systems von der erwarteten Lieferung, Leistung oder dem Ergebnis“ ([IG13], Seite 24).

2.2.2 Testen in Softwareentwicklungsmodellen

Mit Hilfe von Vorgehensmodellen lassen sich Softwareprojekte steuern und strukturieren. Dadurch wird die Übersichtlichkeit erhöht, die Komplexität eines Projektes verborgen und somit beherrschbar gemacht. Die Entwicklung erfolgt nach bewährten und standardisierten Verfahren.

Das *V-Modell* ist ein Vorgehensmodell in dem Softwareentwicklung und Qualitätssicherung kombiniert sind (siehe Abbildung 4).

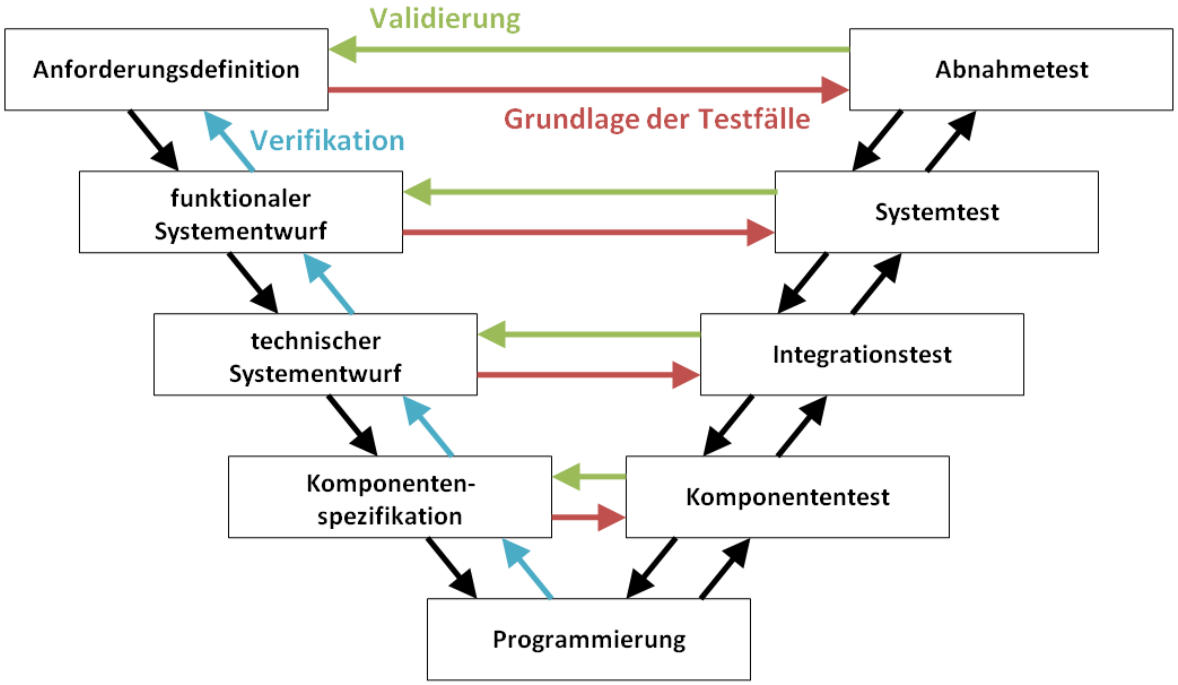

Abbildung 4: Phasen des V-Modells

Der linke Ast des V-Modells besteht aus den Konstruktionsphasen. Zunächst werden die Anforderungen des Auftraggebers ermittelt und ein gemeinsames Verständnis über die Funktionalität des zu erstellenden Softwareproduktes gebildet. Diese Anforderungsspezifikation dient dann sowohl als Grundlage für die nächsten Konstruktionsphasen als auch für den Abnahmetest. Im funktionalen Systementwurf werden aus den Anforderungen die benötigten Funktionen und Abläufe definiert. Der Fokus dieser Stufe liegt bei der Systemarchitektur und dem Systemdesign. Im Anschluss finden die Definition von Systemschnittstellen und die Unterteilung des Systems in möglichst unabhängige Teilsysteme statt. In der Komponentenspezifikation werden Aufgaben, Verhalten und Struktur der Teilsysteme definiert. Dazu gehören auch die Relationen zu anderen Teilsystemen. In der letzten Konstruktionsphase werden die einzelnen Softwarekomponenten implementiert.

Auf der rechten Seite des V-Modells wird jeder Konstruktionsphase eine korrespondierende Teststufe zugeordnet. Die Testfälle der jeweiligen Teststufe basieren auf den Ergebnissen

der entsprechenden Konstruktionsphase. Sobald konsolidierte und abgenommene Arbeitsergebnisse dieser Phase vorliegen kann mit den Testvorbereitungen begonnen werden. So können die Phasen des Testentwurfs und Testrealisierung soweit vorbereitet werden, dass die Testdurchführung später ohne Verzögerung angestoßen werden kann. Die eigentliche Durchführung kann erst angestoßen werden, wenn der zu testende Code letztendlich vorliegt. Parallel zur Anforderungsdefinition werden Abnahmetests vorbereitet, mit deren Hilfe die Anforderungen validiert werden können. Dies ist in Abbildung 4 mit den grünen Pfeilen dargestellt. So lassen sich Fehler bereits auf derselben Abstraktionsstufe finden und korrigieren. Darüber hinaus findet im V-Modell auch eine Verifizierung[13] des Ergebnisses einer Entwicklungsphase gegenüber der Spezifikation der vorherigen Phase statt (vgl. Abbildung 4). So soll überprüft werden, ob die Spezifikation korrekt umgesetzt wurde (vgl. [Sch11], Kapitel 3 und [KK09]).

Zu den Teststufen im V-Modell gehören Komponenten-, Integrations-, System- und Abnahmetests. Mit Hilfe von Komponententests werden die implementierten Softwarebausteine unabhängig voneinander getestet. Komponententests sind in der Regel White-Box-Tests[14], da als Testbasis[15] der zu testende Programmcode verwendet wird. Außerdem wird für die Bestimmung der Soll-Ergebnisse auch die technische Komponentenspezifikation benötigt. Oft werden Platzhalter und Treiber[16] verwendet, um fehlende Softwarekomponenten zu ersetzten und Schnittstellen zu anderen Softwarekomponenten zu simulieren. Ein Platzhalter wird von der zu testenden Softwarekomponente aufgerufen. Ein Treiber dient dazu, diese aufzurufen. Komponententests sollen Defekte aufdecken und die Funktionalität einer Komponente verifizieren.

Der Integrationstest ist die zweite Teststufe im V-Modell. Hier wird das richtige Zusammenspiel der bereits in der ersten Stufe getesteten Komponenten überprüft. Einzelne Komponenten werden schrittweise zu immer größeren Einheiten zusammengesetzt, bis hin zum vollständigen System. Ziel des Integrationstests ist das Finden von Fehlerzuständen in den Schnittstellen der Komponenten, indem versucht wird Fehlerwirkungen zu provoziert.

Der Systemtest überprüft, ob das Gesamtsystem die spezifischen Anforderungen aus Kundensicht erfüllt. Dabei sollte die Testumgebung der späteren Produktivumgebung des Kunden möglichst nahe kommen und realitätsnahe Testdaten verwenden. Das System muss allen funktionalen und nicht-funktionalen Anforderungen genügen.

Mit dem Abnahmetest wird die erstellte Software aus Sicht des Kunden in dessen Einsatzumgebung oder einer ähnlichen Abnahmeumgebung getestet. Wurden gewisse Abnahmekriterien definiert, werden diese mit Hilfe von Abnahmetests überprüft (vgl. [Sch11], Kapitel 3 und [GVV08]).

[13] Im Glossar auf Seite 82 werden die Begriffe *Validierung* und *Verifizierung* voneinander unterschieden.

[14] „Ein Test, der auf der Analyse der internen Struktur einer Komponente oder eines Systems basiert“ (vgl. [IG13], Seite 69).

[15] „Alle Dokumente, aus denen die Anforderungen ersichtlich werden, die an eine Komponente oder ein System gestellt werden, bzw. die Dokumentation, auf der die Herleitung oder Auswahl der Testfälle beruht“ ([IG13], Seite 57).

[16] *Platzhalter* und *Treiber* werden auf den Seiten 78 und 81 beschrieben.

Da im V-Modell die parallele Ausführung der Testphasen zu den entsprechenden Konstruktionsphasen nur unzureichend dargestellt ist, wurde von Andreas Spillner (2002) das W-Modell entwickelt (siehe Abbildung 5). Neben der Verbindung von Testaktivitäten und Testbasis[17] erfolgt eine eindeutige Trennung von Test-, Debug- und Änderungsaktivitäten.

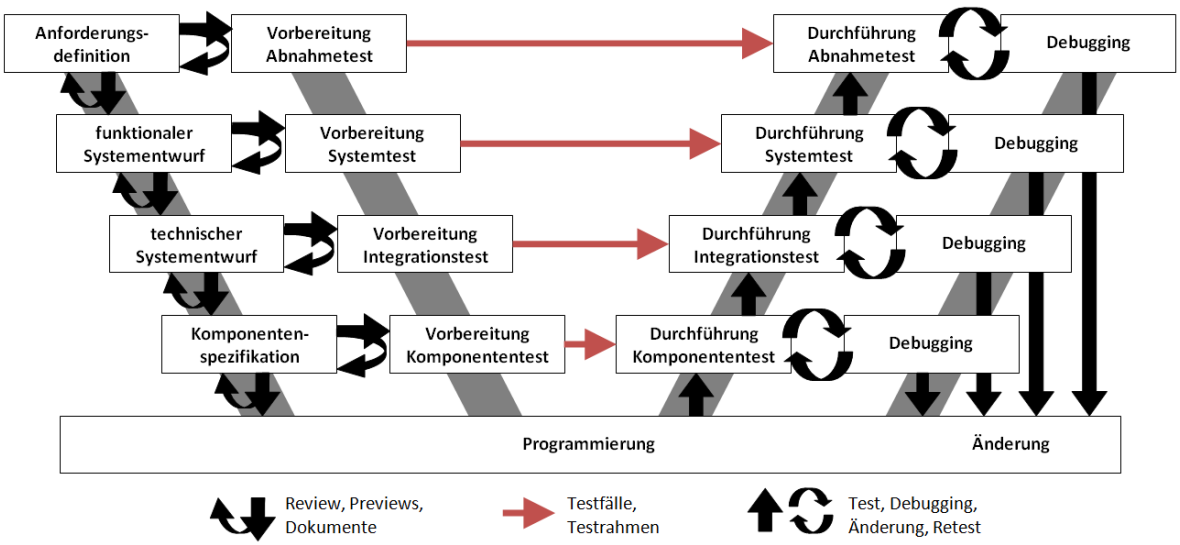

Abbildung 5: Phasen des W-Modells nach Spillner (vgl. [Spi02])

Bereits nach der Durchführung der Risikoanalyse kann die Teststrategie[18] und das Testkonzept[19] festgelegt werden. Im Anschluss an die Anforderungsdefinition werden die Testfälle für den Abnahmetest festgelegt. Parallel zum funktionalen Systementwurf werden die Systemtests geplant. So wird bereits frühzeitig die Systemstruktur auf Testbarkeit analysiert. Danach werden gleichzeitig zum technischen Systementwurf die Integrationstests und zur Komponentenspezifikation die Komponententests vorbereitet. Vor der eigentlichen Programmierung sind bereits alle Testvorbereitungen abgeschlossen. Es müssen nur noch die erstellten Tests ausgeführt werden. Treten bei der Ausführung Fehlerwirkungen auf, müssen die entsprechenden Fehlerzustände mit Hilfe des Debuggings[20] gefunden werden. Nach der Fehlerbeseitigung werden alle Tests wiederholt ausgeführt, um festzustellen, ob die Fehlerzustände behoben, und ob keine neuen Defekte eingebaut wurden (vgl. [Spi02].

[17] „Alle Dokumente, aus denen die Anforderungen ersichtlich werden, die an eine Komponente oder ein System gestellt werden, bzw. die Dokumentation, auf der die Herleitung oder Auswahl der Testfälle beruht“ ([IG13], Seite 57).

[18] „Abstrakte Beschreibung der vorgesehenen Teststufen und der Art und Weise, wie innerhalb dieser Teststufen vorzugehen ist, für eine Organisation oder ein Programm - gültig für ein oder mehrere Projekte“ ([IG13], Seite 64).

[19] Vgl. Seite 80.

[20] „Tätigkeit des Lokalisierens/Identifizierens, Analysierens und Entfernens der Ursachen von Fehlerwirkungen in der Software“ ([IG13], Seite 16).

2.2.3 Modellbasiertes Testen (MBT)

Utting et al. (2006) definieren das MBT in dem Artikel *A taxonomy of model-based testing* als die automatisierte Ableitung von konkreten Testfällen aus abstrakten, formalen Modellen und deren Ausführung: „...we define model-based testing as the automatable derivation of concrete test cases from abstract formal models, and their execution“ ([UPL06], Seite 1). Nach dem Europäischen Institut für Telekommunikationsnormen (ETSI) versteht man unter dem modellbasierten Testen ein Menge von Verfahren zum Generieren von Tests aus Modellen: „**model-based testing**: umbrella of approaches that generate tests from models“ ([ETS11], Seite 6). In der UTP-Spezifikation von 2013 wird die Definition der ETSI noch erweitert: „An umbrella of techniques that use (semi-)formal models as engineering artifacts in order to specify and/or generate test-relevant artifacts, such as test cases, test scripts, reports, etc “ ([OMG13f], Seite 5). Wenn mit Hilfe des MBT aus den Modellen automatisiert Testfälle generiert werden sollen, muss der Grad der Formalisierung dazu ausreichen. Die Modelle dienen nicht ausschließlich dem Zweck der Generierung, sonder auch für die Spezifizierung von testrelevanten Artefakten.

In Abbildung 6 ist der aus drei Hauptschritten bestehende Ablauf des modellbasierten Testens dargestellt. Zunächst wird mit Hilfe der Anforderungen oder vorhandener Spezifikationsdokumentation ein Testmodell[21] erstellt. Die Modellierung kann mit einer formalen Modellierungssprache wie der UML durchgeführt werden. Das Testmodell repräsentiert das erwartete Verhalten der SUT. Die benötigten Testdaten können direkt im Testmodell enthalten sein oder von diesem abstrahiert bereitgestellt werden. Im Modell sollten Assoziationen zwischen Anforderungen und den Modellelementen definiert sein, damit sich die Testergebnisse nach der Testausführung den entsprechenden Anforderungen zuordnen lassen.

Für die Testfallgenerierung müssen bestimmte Auswahlkriterien definiert werden. Dafür können unterschiedliche Überdeckungsarten z.B. dass alle Anforderungen durch mindestens einen Testfall abgedeckt sind oder bekannte Testabdeckungen wie die Zustandsüberdeckung verwendet werden.

Mit Hilfe der Informationen aus dem Testmodell und den Testauswahlkriterien können dann Testfälle automatisiert generieren werden. Die Testfälle müssen eventuell noch in ausführbare Testfälle übertragen werden und können dann manuell oder automatisiert ausgeführt werden. Wenn Modell und Testobjekt sich auf unterschiedlichen Abstraktionsniveaus befinden, kann es notwendig sein, dass die Testfälle vor der Ausführung noch angepasst werden müssen. Diese Anpassung findet in einem sogenannten *Adapter* statt und muss ggf. manuell durch einen Testentwickler erfolgen (vgl. [UPL06] und [Sch12]).

Vorteile

Die erstellten Modelle können neben der genauen Beschreibung der Anforderungen auch als Teil der Dokumentation und für die automatische Generierung von Testfällen verwendet werden. Durch die Modellierung können Eingangsdokumente, wie die funktionale Spezifikation, sehr gut in einer frühen Softwareentwicklungsphase überprüft werden,

[21] Was unter einem *Testmodell* zu verstehen ist, ist auf Seite 80 beschrieben.

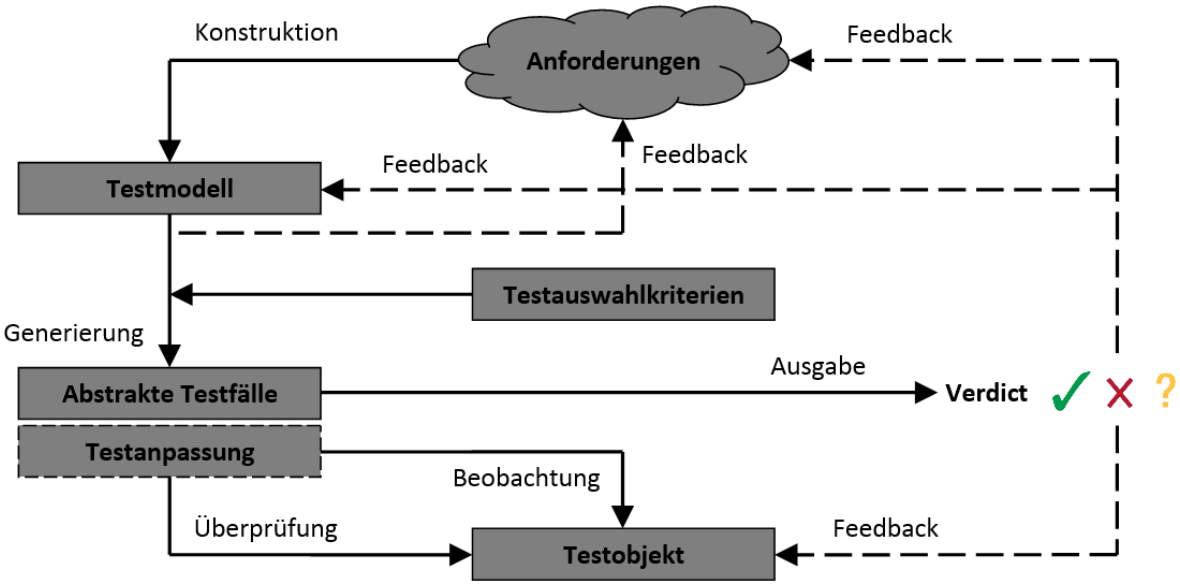

Abbildung 6: Prozesse beim modellbasierten Testen (vgl. [ETS11])

da das Testen eine sehr präzise Beschreibung des erwarteten Systemverhaltens erfordert. Durch den ermöglichten frühen Beginn der Test-Definition wird auch verhindert, dass bei Verzögerungen in der Entwicklung die Zeit für das Entwerfen von Testfällen (welches klassischerweise am Ende einer Entwicklung steht) gekürzt wird. Außerdem können auch Fehler in der Anforderungsspezifikation aufgedeckt werden. Es findet eine bessere Dokumentation der Testfälle statt, was die Kommunikation zwischen Entwicklern und Testern erleichtert. Darüber hinaus kann die Testqualität durch die modellbasierte Qualitätsanalyse verbessert werden. Es existieren bereits eine Reihe von Testwerkzeugen, mit denen sich Testfälle aus einem Modell ableiten lassen (vgl. [UPL06] und [Sch12]).

Nachteile

Da mit Hilfe der Modelle theoretisch eine unendliche Anzahl von Tests generiert werden kann, ist es unabdingbar, eine für den jeweiligen Einsatzzweck geeignete Testauswahlstrategie zu entwickeln. Dieser Vorgang ist nicht trivial und erfordert gut ausgebildete Tester. Die Verwendung des MBT ist in der Praxis oft nur dann sinnvoll, wenn das zu erstellende Softwaresystem eine gewisse Komplexität aufweist oder ein hoher Anspruch an die Softwarequalität vorliegt. Des Weiteren ist die Adapterimplementierung oft mit einem hohen Aufwand verbunden. Da die Architektur und die Schnittstellen der SUT bei der Erstellung des Testmodells oftmals noch nicht bekannt sind, muss die Anpassung im Adapter erfolgen (vgl. [UPL06], [Sch12] und [UL10]).

2.3 UML Testing Profile (UTP)

Der UML fehlen Konzepte, um testrelevante Artefakte zu beschreiben (vgl. [Dai04]). Aus diesem Grund wurde das *UTP* (UML Testing Profile) entwickelt. UTP ist eine von der

OMG standardisierte Modellierungssprache und erweitert die UML mittels des UML-Profilmechanismus um testspezifische Konzepte. Dadurch lassen sich modellbasierte Testspezifikationen[22] entwerfen, visualisieren, spezifizieren, analysieren, konstruieren und dokumentieren. Das UTP schreibt weder eine Methodik noch konkrete Technologien für die Umsetzung vor.

Da das UTP Teil des UML-Ökosystems ist (siehe Abbildung 7), können testbezogene Artefakte mit anderen relevanten Systemartefakten aus unterschiedlichen Profilen des Ökosystems zusammengebracht werden.

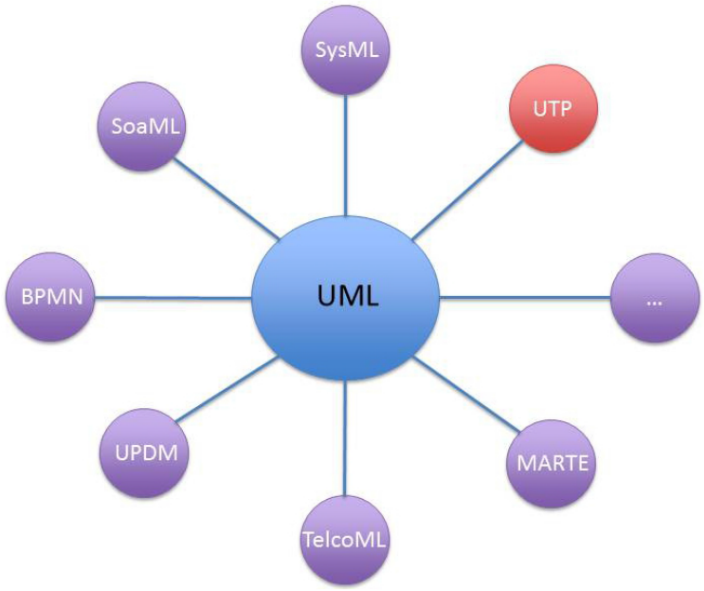

Abbildung 7: Das UML-Ökosystem (vgl. [OMG13f])

Dadurch lassen sich z.B. Anforderungen, Risiken, Anwendungsfälle oder Systemspezifikationen verknüpfen. So ist eine einfachere Zusammenarbeit von Anforderungs-, System- und Testingenieuren möglich (vgl. [OMG13f]). Das UTP besteht aus der Definition einer Bibliothek mit obligatorischen, vordefinierten Typen und den vier logischen Konzeptpaketen: Testarchitektur, Testverhalten, Testdaten und Testmanagement (siehe Tabelle 1).

2.3.1 Typbibliothek

In der Typbibliothek werden eine Reihe von Typen definiert, die obligatorisch für die Implementierung des UTP sind (siehe Abbildung 8).

Zu diesen vordefinierten Typen gehört unter anderem der Aufzählungstyp *Verdict*, der das Ergebnis einer Testfallausführung spezifiziert. Ein Verdict kann einen von fünf möglichen Werten annehmen:

[22] „Ein Dokument, das aus der Testentwurfspezifikation, der Testfallspezifikation und/oder der Testablaufspezifikation besteht“ ([IG13], Seite 64).

Typbibliothek	Testarchitektur	Testverhalten	Testdaten	Testmanagement
Verdict	TestComponent	DetermAlt	DataPartition	TestLog
Arbiter	SUT	FinishAction	DataPool	TestLogApplication
Duration	TestContext	LogAction	DataSelector	TestLogEntry
Timezone		ValidationAction	CodingRule	
Timer		TestCase	LiteralAny	
		TimeOut	LiteralAnyOrNull	
		TimeOutMessage		
		StartTimerAction		
		StopTimerAction		

Tabelle 1: Überblick über die UTP Konzepte

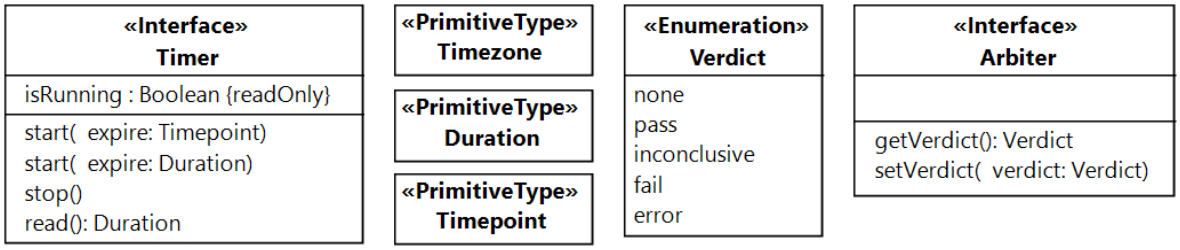

Abbildung 8: Die UTP Typbibliothek

- **none**: Der Testfall wurde noch nicht ausgeführt. Es hat noch keine Kommunikation zwischen einer Testkomponente und der SUT stattgefunden.
- **pass**: Das Testsystem entspricht den Erwartungen. Der Testfall war erfolgreich.
- **inconclusive**: Der Testfall kann weder zu *pass* noch zu *fail* ausgewertet werden.
- **fail**: Das Testsystem entspricht nicht den Erwartungen. Der Testfall war nicht erfolgreich.
- **error**: In der Testumgebung (nicht in der SUT) ist ein Fehler aufgetreten.

Das Ergebnis einer Testfallausführung wird mit Hilfe einer bestimmten Entscheidungsstrategie festgelegt. Dazu dient die Schnittstelle *Arbiter* mit Operationen zum Erfragen und Setzen des aktuellen Verdicts. Ein Testfall oder ein Testkontext verwenden eine Implementierung der Schnittstelle, um das finale Testergebnis zu bestimmen. Der Arbitrationsalgorithmus kann bei Bedarf vom Benutzer individuell definiert werden.

Zur Gruppierung von Testkomponenten kann diesen eine Zeitzone (*Timezone*) zugewiesen werden. Dadurch können Testkomponenten innerhalb derselben Zeitzone Zeitwerte austauschen und vergleichen.

Mit Hilfe des primitiven Typs *Duration* kann ein Zeitintervall definiert werden. Die konkrete Zeiteinheit ist nicht vorgeben und kann vom Benutzer bestimmt werden. Um den genauen Ablaufzeitpunkt eines Timers zu spezifizieren, dient der primitive Typ *Timepoint*.

Auch hier ist das konkrete Format dem Benutzer überlassen. *Timer* sind Hilfsmittel um das Testverhalten zu beobachten und zu steuern. Dadurch können unter anderem Systemblockaden und instabiles Systemverhalten während der Testausführung vermieden werden. Ein Timer kann zu einem bestimmten Zeitpunkt (*Timepoint*) oder nach einer definierten Zeitdauer (*Duration*) ablaufen. In diesem Fall wird automatisch ein *Timeout* an den Timeout-Besitzer (eine Testkomponente oder ein Testkontext) gesendet. Timer können durch ein *StartTimerAction* gestartet, durch ein *StopTimerAction* gestoppt und durch *TimerRunningAction* und *ReadTimerAction* ausgelesen werden (vgl. [OMG13f] und [Dai04]).

2.3.2 Testarchitektur

In der Testarchitektur werden strukturelle Aspekte der Testumgebung[23] und einer dazugehörigen Testkonfiguration beschrieben. Eine Testumgebung umfasst alles was zur Ausführung von Testfällen benötigt wird. Dazu gehören unter anderem Testkomponenten, Datenbanken, Hardware und Stimulatoren. Die Testkonfiguration beschreibt wie Testkomponenten mit dem zu testenden System verbunden sind. UTP definiert die Stereotypen *SUT*, *TestComponent* und *TestContext* (siehe Abbildung 9).

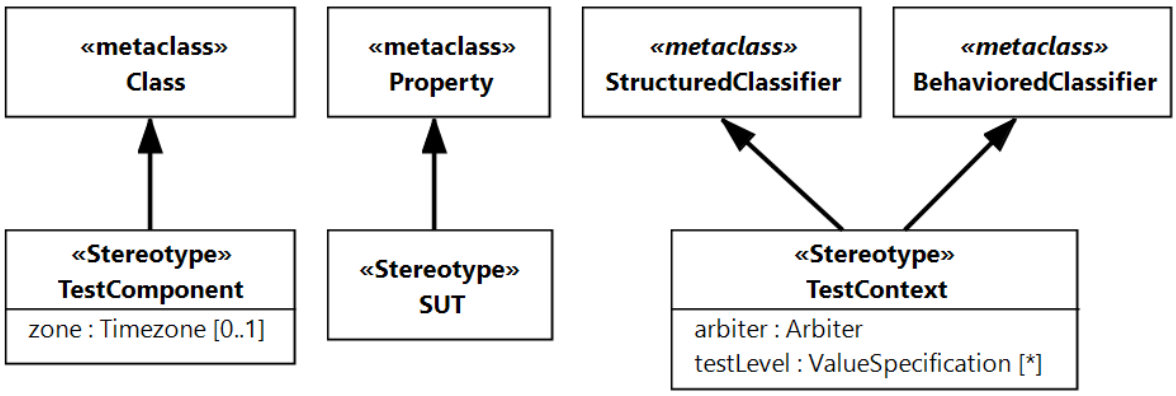

Abbildung 9: Stereotypen der UTP Testarchitektur

Testkomponenten sind Teil der Testumgebung und werden zur Realisierung des Testverhaltens verwendet, indem die SUT mit Testdaten stimuliert wird und die resultierenden Ergebnisse mit den erwarteten verglichen werden.

Mit dem SUT-Stereotyp lassen sich eine Menge von Schnittstellen festlegen, über die ein konkretes Testobjekt während der Testausführung überwacht und beobachtet werden kann. Die SUT ist Teil der Testkonfiguration und beschreibt eine einzelne Komponente, eine Menge von Komponenten oder ein vollständiges System das getestet werden soll. Die SUT wird als Black-Box betrachtet und die Kommunikation läuft ausschließlich über deren öffentliche Schnittstellen. Die SUT stellt die logische Sicht auf die Schnittstellen der tatsächlichen Implementierung dar. Wenn logische und technische Schnittstellen voneinander abweichen,

[23] „Eine Umgebung, die benötigt wird, um Tests auszuführen. Sie umfasst Hardware, Instrumentierung, Simulatoren, Softwarewerkzeuge und andere unterstützende Hilfsmittel“ ([IG13], Seite 59).

erfolgt eine Anpassung im Adapter.

Ein Testkontext (*TestContext*) beschreibt den Kontext für die Spezifikation und die spätere Ausführung von Testfällen. So kann die dazugehörige Testkonfiguration und eine optionale Teststeuerung (Definition der Ausführungsreihenfolge der Testfälle) beschrieben werden. In einem Testkontext sind die Testkomponenten und die Verbindungen zur SUT definiert (siehe Abbildung 10, vgl. [OMG13f] und [Dai04]).

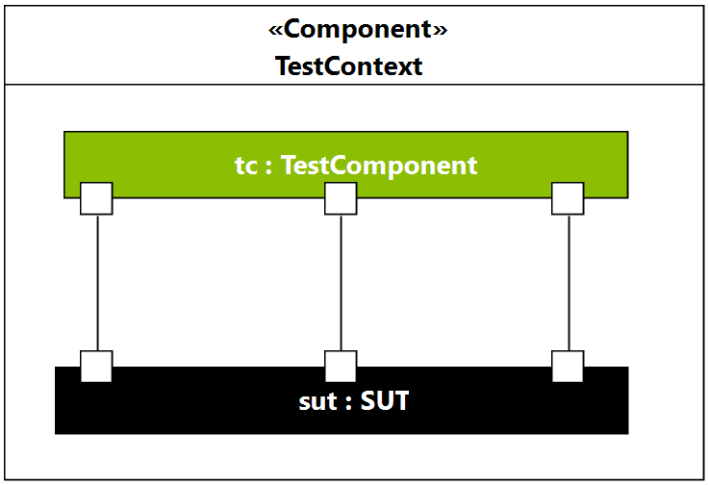

Abbildung 10: UTP TestContext mit TestComponent und SUT

2.3.3 Testverhalten

Das Testverhalten spezifiziert die notwendigen Aktionen und Bewertungen für die Evaluierung der SUT mit Hilfe einer Reihe von Stereotypen (siehe Abbildung 11). Zur Beschreibung des Testverhaltens können alle UML-Verhaltensbeschreibungen einsetzt werden.

Ein Testfall beschreibt, wie eine Menge von Testkomponenten mit einer SUT interagiert und ist die Umsetzung eines Testziels für eine bestimmte Testkonfiguration (vgl. [SDGR03]). Das Testverhalten wird mit Hilfe von Alternativen, Schleifen, Stimulationen der SUT und Beobachtungen von dessen Reaktion spezifiziert. Dazu können alle UML-Verhaltensdiagramme, wie Sequenz-, Interaktions- oder Zustandsdiagramme verwendet werden. Ein Testfall liefert immer ein Verdict als Ergebnis das von einem Arbiter bestimmt wird.

Eine deterministische Alternative ist ein Konzept für Interaktionsmodelle. Es ist ein kombiniertes Fragment mit applizierten *DetermAlt*-Stereotyp. Bei einer deterministischen Alternative werden die Operanden genau in der Reihenfolge ausgewertet, in der sie im Modell auftreten. Wird eine deterministische Alternative in einer Testausführung erreicht, wartet die beteiligte Testkomponente, bis ein Ereignis auftritt. Ein Ereignis kann eine eintreffende Nachricht bei einer Testkomponente oder ein Timeout sein. Erst dann beginnt die Auswertung der deterministischen Alternative in zwei Schritten. Zunächst wird die Wächterbedingung evaluiert. Schlägt diese fehl, wird der gesamte Operand für weitere

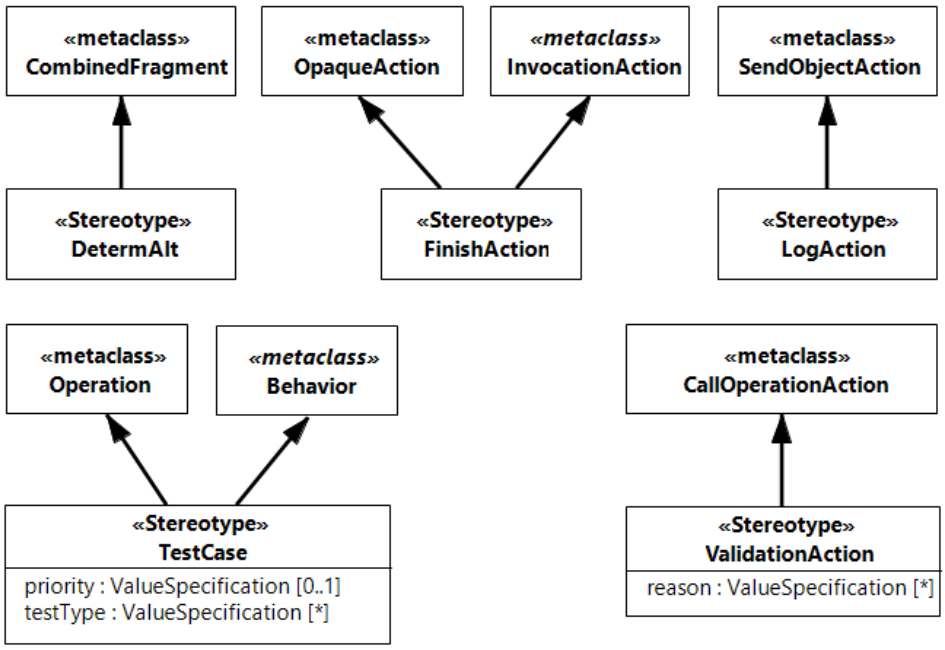

Abbildung 11: Stereotypen des UTP Testverhaltens

Überprüfungen ausgeschlossen und der Operand wird nicht betreten. Trifft die Wächterbedingung zu, folgt ein zusätzlicher Auswertungsschritt, beginnend mit der ersten Nachricht innerhalb des ersten Interaktionsoperanden der Testkomponente. Erwartet man ein Timeout und der entsprechende Timer ist noch nicht abgelaufen, wird der Interaktionsoperand nicht betreten. Erwartet man stattdessen eine Nachricht, wird der Interaktionsoperand nur betreten, wenn die erwartete Nachricht mit der aktuellen übereinstimmt. Bei einer Abweichung des Ergebnisses wird der nächste Interaktionsoperand evaluiert. In Abbildung 12 ist die Verwendung einer deterministischen Alternative in einem Sequenzdiagramm zu sehen.

Mit einer *FinishAction* wird der Testfall für eine bestimmte Testkomponente umgehend beendet. Danach kann auch das lokale Verdict nicht mehr geändert werden. Mit Hilfe einer *LogAction* lassen sich beliebige Entitäten wie einfache Zeichenketten, vollständige Diagramme oder Instanzwerte loggen. *ValidationAction* wird dazu verwendet, um das Verdict in einem Testverhalten zu bestimmen, indem die *setVerdict*-Methode einer Arbiter-Implementierung aufgerufen wird.

In einem Testfall wird normalerweise nur das Standard- oder das erwartete Verhalten der SUT beschrieben. Tritt während der Testausführung ein unerwartetes Verhalten auf, wird ein Standardhandler bereitgestellt. *Defaults* lassen sich individuell für Ereignisse in Interaktionsdiagrammen oder für Zustände in Zustandsdiagrammen, für Testkomponenten einer spezifischen Klasse oder für alle Testkomponenten in einem Testsystem definieren.

UTP erweitert die einfachen UML Zeitkonzepte mit den vordefinierten Typen *Timer*, *Duration* und *Timepoint* (siehe Abschnitt 2.3.1), und Aktionen zum Starten und Stoppen von

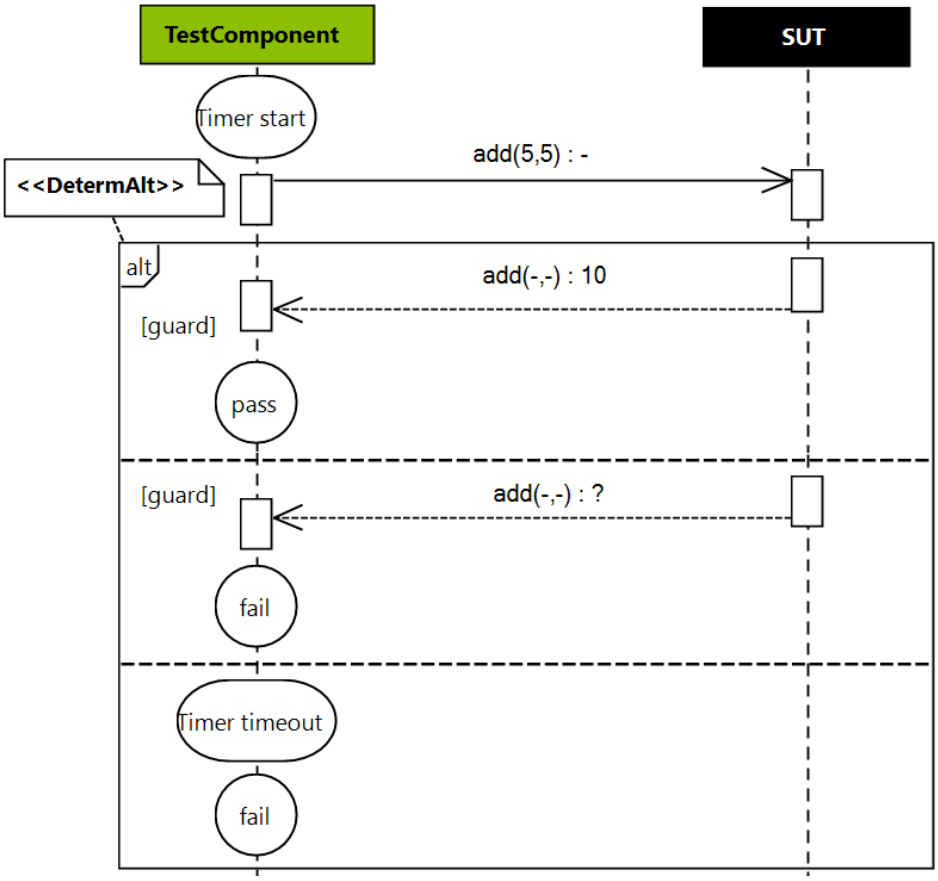

Abbildung 12: Verwendung einer deterministischen Alternative in einem Testfall

Timern. Dadurch lässt sich u.a. in einem Testfall spezifizieren, wie viel Zeit zwischen dem Senden und dem Empfangen einer Nachricht vergehen darf. Diese imperative Form der Zeitkontrolle ergänzt die deklarativen Konzepte des *SimpleTime*-Pakets der UML.

Für das Arbeiten mit verteilten Testumgebungen benötigen die Testkomponenten ihr Timezone-Attribut zur Koordination. Mit den Aktionen *GetTimezoneAction* und *SetTimezoneAction* kann die aktuelle Zeitzone einer Testkomponente zur Laufzeit erfragt und gesetzt werden (vgl. [OMG13f] und [Dai04]).

2.3.4 Testdaten

Das Unterpaket Testdaten definiert Konzepte für Daten, die für die Kommunikation mit der SUT und für die Definition des initialen und finalen Zustands vor bzw. nach der Testfallausführung benötigt werden. Für die Spezifikation von Testdaten stehen die drei Konzepte *DataPool*, *DataPartition* und *DataSelector* zur Verfügung (siehe Abbildung 13).

Ein DataPool spezifiziert einen physikalischen Datencontainer für explizite Werte oder Datenpartitionen für eine (wiederholte) Testausführung. Zur Definition einer Äquiva-

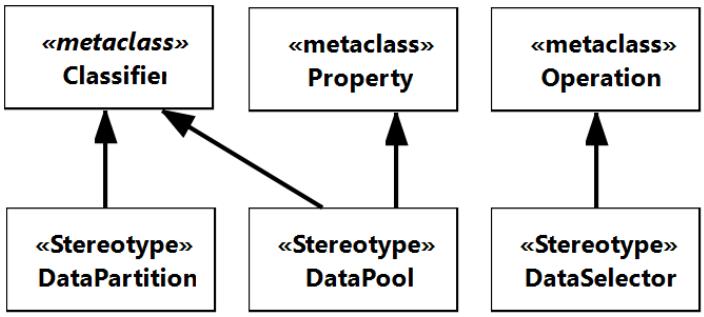

Abbildung 13: Stereotypen *DataPartition*, *DataPool* und *DataSelector*

lenzklasse für einen gegebenen Typ wird eine DataPartition verwendet. So lassen sich vorhandene Datenpartitionen wiederverwenden oder verfeinern. Eine Datenpartition ist mit einem DataPool oder einer anderen Datenpartition verbunden. Ein DataSelector ist ein Zugriff auf konkrete Werte innerhalb eines DataPool. Die Verfeinerung einer Datenpartition wird durch die Erweiterung des entsprechenden Datenselektors vollzogen.

Darüber hinaus definiert UTP noch Konzepte für Kodierungsregeln, für Platzhalter (*Wildcards*) und für die Wiederverwendung von Testdatenwerte (siehe Abbildung 14). Kodierungsregeln geben an wie die Testdatenwerte, die zwischen der Testumgebung und der SUT ausgetauscht werden, zu realisieren sind.

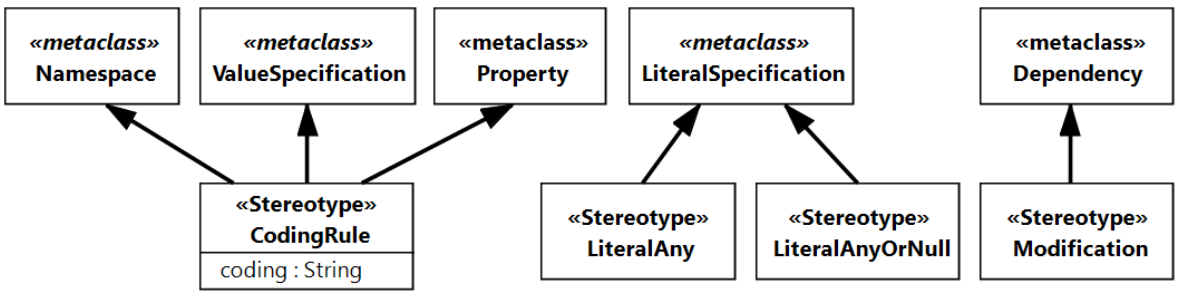

Abbildung 14: *CodingRule*, *Wildcards* und *Modification*

Durch die Verwendung von Platzhaltern ist es möglich, sich nur auf die aktuell relevanten Werte zu konzentrieren und die irrelevanten Daten auszublenden. Dazu wurden die beiden Literale *LiteralAny* und *LiteralAnyOrNull* eingeführt. Der LiteralAny-Platzhalter spezifiziert einen beliebigen Wert aus einer Menge von möglichen Werten und wird mit einem Fragezeichen "?" notiert. Bei einem LiteralAnyOrNull ist zusätzlich auch noch das Fehlen eines Wertes erlaubt. Modelliert wird dieses Literal mit einem Stern "*".

Die UML bietet keine Möglichkeit, um vorhandene Instanzspezifikationen zum Modifizieren wieder zu verwenden. In UTP lassen sich mit dem *Modification*-Konzept bereits existierende Instanzen in eine neue Instanzspezifikation verfeinern oder modifizieren (vgl.

[OMG13f]).

2.3.5 Testmanagement

Unter dem Testmanagement versteht man die „Planung, Aufwandsschätzung, Überwachung und Kontrolle von Testaktivitäten, die üblicherweise durch einen Testmanager erfolgen“ ([IG13], Seite 61). Die Testplanung wird zur Ausarbeitung des Testkonzeptes verwendet. Das UML Testing Profile bietet zur Testplanung das Testziele-Konzept an. Ein Testziel ein „Grund oder Zweck für den Entwurf und die Ausführung von Tests“ ([IG13], Seite 61). Durch eine informelle Spezifikation eines später zu realisierenden Testfalls anhand des Ziels, lassen sich Testaktivitäten und eine Rückverfolgbarkeit zwischen Anforderungen und Testartefakten bereits in einer sehr frühen Projektphase realisieren. Darüber hinaus können Testziele priorisiert werden, um die Testplanung eines Testprojektes oder einer einzelnen Testphase zu optimieren. Testziele beschreiben, wie die SUT auf bestimmte Stimuli reagieren wird und werden durch Testfälle realisiert. Dies gilt insbesondere auch für Ausnahmesituation bzw. nicht-korrektes Verhalten der SUT, welches oftmals in der Anforderungsspezifikation nur unzureichend definiert ist.

Die Ergebnisse einer Testausführung werden in UTP als *Testlog* spezifiziert. In einem Testlog können die Verantwortlichen (die Tester) der Testfallausführung, der Startzeitpunkt der Testausführung, die Laufzeit des Testfalls, das finale Verdict und optional eine Beschreibung über den Grund des finalen Verdicts angegeben werden. Ein Testlog bezieht sich entweder auf einen Testfall oder auf einen Testkontext. UTP erlaubt auch die präzise Darstellung des durchgeführten Testfalls als Verhaltensbeschreibung. In der UTP ist keine genaue Struktur für die Spezifikation eines Testers definiert. Ein Tester-Objekt kann eine einfache textuelle Beschreibung einer Person oder eines Testausführungssystems sein. Mit Hilfe der *TestLogApplication*-Beziehung kann ein konkreter Testlog mit dem im Testlog beschriebenen Testfall oder Testkontext assoziiert werden (vgl. [OMG13f]).

2.4 Testautomatisierungswerkzeuge

Mit Hilfe der Testautomatisierung lassen sich Testplanung[24], Testentwurf, Testausführung und Teststeuerung[25] kontrollieren und die Ergebnisse der Ausführung protokollieren und auswerten. Es existieren eine Vielzahl von Testwerkzeugen, die für unterschiedliche Teststufen, Testphasen, Testarten oder Teststrategien geeignet sind. Einzelne Werkzeuge sind auf genau eine Aufgabe zugeschnitten und andere bieten als Testwerkzeug-Suite eine Vielzahl von Testfunktionen an (vgl. [Sch11].

In dieser Arbeit werden die Testautomatisierungswerkzeuge in drei Arten gruppiert: In Werkzeuge für das Testmanagement, das Testdesign und für die Testausführung. Testmanagement beinhaltet Aktivitäten für die Auswertung, Dokumentation und Administration von Testprozessen[26] sowie Unterstützung beim Anforderungsmanagement.

[24] „Eine Aktivität im Testprozess zur Erstellung und Fortschreibung des Testkonzepts“ ([IG13], Seite 62)

[25] „Als Teststeuerung bezeichnet man die Managementaufgabe zur Entwicklung und Anwendung von Korrekturmaßnahmen, um in einem Testprojekt eine Abweichung vom geplanten Vorgehen zu beherrschen“ ([IG13], Seite 58).

[26] Die Aktivitäten des fundamentalen Testprozesses sind auf Seite 80 beschrieben.

Testdesignwerkzeuge bieten neben der Spezifikation der Testumgebung und Testfälle auch die Möglichkeit der Erstellung oder Generierung der benötigten Testdaten. Bei der Testausführung werden die Testfälle durchlaufen und die Ergebnisse verglichen. Die beispielhaft vorgestellten Testwerkzeuge können sich über mehrere Stufen erstrecken, wobei dann die Zuteilung nach dem überwiegenden Hauptzweck getroffen wurde. Die für diese Arbeit verwendeten Testwerkzeuge Fokus!MBT und JUnit und die Testsprache TTCN-3 werden in den Abschnitten 2.4.1, 2.4.3 und 2.4.2 detailliert beschrieben.

Testmanagementwerkzeuge

Gerade bei großen Softwareprojekten müssen oft hunderte oder tausende Testfälle verwaltet werden. Testmanagementwerkzeuge bieten Funktionen für die Verarbeitung von Systemanforderungen, zur Zeit- und Ressourcenplanung, für die Überprüfung von Abdeckungskriterien sowie Möglichkeiten für die Rückverfolgbarkeit von Tests, Testergebnissen und die Zuordnung zu den entsprechenden Anforderungen (vgl. [Sch11]). Im Folgenden werden eine Reihe proprietärer und freier Testmanagementwerkzeuge vorgestellt, die sich besonders für Testverwaltungsaufgaben eignen:

- **Gemini**: *Gemini* ist ein proprietäres *Issue-* und *Bug-Tracking*-System der Firma *Countersoft*, mit dem sich Defekte erfassen und dokumentieren lassen. Gemini basiert auf der *.NET*-Technologie und bietet eine Visual Studio-, Outlook- und Subversion-Integration. Darüber hinaus lassen sich auch Zeit- und Kostenabschätzungen durchführen (vgl. [Cou13]).
- **HP Quality Center**: Mit der *HP Quality Center*-Software ist neben dem *Bug-Tracking* auch die Verwaltung der Anforderungen, die Erstellung manueller oder automatisierter, funktionaler Tests und das Management der benötigten Testdaten möglich (vgl. [HP13]).
- **IBM Rational Quality Manager**: Der *Rational Quality Manager* ist eine Qualitäts- und Testmanagementsoftware von *IBM*. Diese umfasst eine umfangreiche Synchronisationsmöglichkeit zum Austausch von Informationen, Automatisierungsfunktionen zur Verkürzung der Projektdauer und Steuerungseinrichtungen für Projektkennzahlen und deren Dokumentation. Darüber hinaus lassen sich Testskripte[27] erstellen und Testfälle spezifizieren und ausführen (vgl. [IBM13]).
- **TestLink**: *TestLink* ist ein webbasiertes Testmanagementsystem, das unter der *GNU General Public License* erstellt wird. Es lassen sich Anforderungen definieren, Testkonzepte anlegen, Testfälle erstellen und ausführen und die Ergebnisse der Testausführung protokollieren und analysieren. Außerdem bietet TestLink die Möglichkeit *Bug-Tracking*-Systeme wie *JIRA* oder *Bugzilla* zu integrieren (vgl. [Tea13]).

Testdesignwerkzeuge

Mit Hilfe von Testdesignwerkzeugen lassen sich Testfälle oder Testdaten automatisiert aus der Anforderungsspezifikation oder anderen Quellen generieren. Die Generierung von Testdaten ist oft nicht vollständig möglich und muss gegebenenfalls manuell ergänzt

[27] „Bezeichnet üblicherweise eine Testablaufspezifikation, insbesondere eine automatisierte“ ([IG13], Seite 63).

werden. Oft erlaubt die hohe Abstraktionsstufe eine Generierung von sehr vielen Testfällen, die schwierig „von Hand“ zu erreichen ist. Zudem können automatisiert Kriterien für Testfalldefinition (z.B. für Testdatenauswahl oder Abdeckungskriterien für Kombinationen von Methodenaufrufen - sowohl erlaubte, wie unerlaubte) erfüllt werden. Es kann also kein Kriterium einfach vergessen werden. (vgl. [Sch11]).

- **CTE XL**: *CTE XL* ist ein Editor der Firma *Berner & Mattner* zum Erstellen von Testspezifikationen. Dieser basiert auf dem Klassifikationsbaumverfahren[28], ist domänenunabhängig und lässt sich daher universell einsetzen. Mit Hilfe von Kombinationsregeln und Priorisierungen lassen sich automatisch Äquivalenzklassen und Testfälle generieren. Darüber hinaus bietet CTE XL eine Anbindung an das HP Quality Center und den IBM Rational Quality Manager (vgl. [Sys13]).
- **Conformiq Tool Suite**: Mit der *Conformiq Tool Suite* können funktionale Tests modellbasiert entworfen und generiert werden. Das Soll-Verhalten der SUT wird mit einem Modell beschrieben. Aus diesem lassen sich dann Testpläne, Testfälle und ausführbare Testskripte automatisiert generieren. Die Generierung der Testfälle ist in eine Reihe von Standardformate wie z.B. TTCN-3, C/C++, Java, JUnit, HTML oder Excel möglich. Nach Angaben der Herstellerfirma, der *Verifysoft Technology GmbH*, lässt sich durch die Testfallgenerierung eine fünf- bis zwanzigfache Produktivitätssteigerung und eine höhere Testqualität erzielen (vgl. [Tec13b]).
- **SpecExplorer**: Der *SpecExplorer* von *Microsoft* ist ein in *Visual Studio* integriertes modellbasiert Testwerkzeug. Das Systemverhalten wird durch Regeln in C# und durch die Definition von Modellszenarien als eine Art regulärer Ausdrücke (in der Skriptsprache *Cord*) definiert. Die möglichen Testfälle werden durch die Erkundung des Modells ermittelt und können automatisch generiert werden. Für die Erkundung und für die Testausführung muss eine Adapter implementiert werden, der eine Verbindung zwischen Modell und SUT herstellt (vgl. [SMJU10]).
- **Fokus!MBT**: Siehe Abschnitt 2.4.1.

Testausführungswerkzeuge

Testausführungswerkzeuge dienen zur Bereitstellung der benötigten Testdaten, zur Ausführung der Testfälle, zum Vergleich zischen erwarteten und aktuellem Verhalten und zur Protokollierung des Testvorgangs. Im Anschluss an die Ausführung ist eine Analyse der erreichten Überdeckungsgrade[29] möglich. Durch die Automatisierung kann eine große Anzahl von Testfällen auch wiederholt in kurzer Zeit ausgeführt werden (vgl. [Sch11]).

- **TTworkbench**: Die *TTworkbench* ist eine integrierte Testentwicklungs- und Testausführungsumgebung basierend auf TTCN-3 (siehe Abschnitt 2.4.2). Durch den Entwurf des Testsystems in TTCN-3 kann das zu testende System in einer beliebigen Technologie vorliegen. Die TTworkbench besteht aus mehreren funktionalen Teilen. *TTman*

[28] Vgl. Seite 77.

[29] „Der Grad, ausgedrückt in Prozent, zu dem ein spezifiziertes Überdeckungselement (z.B. Zweig) durch eine Testsuite ausgeführt wurde.“ ([IG13], Seite 14).

bildet die Grundlage zum Verwalten, Ausführen und Analysieren von Testsuiten[30]. Mit dem *CL Editor* lassen sich textbasierte TTCN-3-Testfälle definieren. *TTthree* kompiliert TTCN-3-Module in Ausführbare *JAR*-Dateien. Mit Hilfe des *GFT Editor* lassen sich Testspezifikationen als Sequenzdiagramme modellieren und dokumentieren. Darüber hinaus lassen sich TTCN-3- und Java-basierte Testadapter debuggen um Fehlerzustände aufzuspüren (vgl. [Tec13a]).

- **JUnit**: Siehe Abschnitt 2.4.3.
- **TestNG**: *TestNG* ist ein Framework für Komponenten- und Integrationstests und erweitert die Funktionalität von JUnit. Die aus JUnit bekannten Annotationen können äquivalent verwendet werden. Die *@Test*-Annotation wurde um einige nützliche Attribute erweitert, wie z.B. Gruppierungsmöglichkeiten, eine Methodenbeschreibung, oder eine Priorisierung der Testfälle. TestNG bietet die Möglichkeit, Testsuiten mit Hilfe einer XML-Datei zu definieren. Die XML-Dateien lassen sich dann manuell oder automatisch über *Apache Ant* ausführen. Durch die Angabe einer Gruppierung kann die Ausführungsreihenfolge der Testfälle beeinflusst werden. Es ist möglich, eine harte oder eine weiche Abhängigkeit zu verwenden. Bei der harten Abhängigkeit müssen alle abhängigen Methoden erfolgreich beendet worden sein, bevor die eigene Methode ausgeführt wird. Schlägt die abhängige Methode fehl, wird die eigene Methode nicht aufgerufen, aber im Report vermerkt. Bei einer weichen Abhängigkeit wird die eigene Methode auch dann ausgeführt, wenn die Ausführung einer abhängigen Methode fehlschlägt. Durch sogenannte *Factories* lassen sich Testmethoden mit unterschiedlichen Testdaten aufrufen. Mit TestNG kann die parallele Ausführung simuliert werden. Dazu kann in einem Testfall die Anzahl der unterschiedlichen Threads und die Anzahl der Aufrufe angegeben werden. Die Ergebnisse der Testausführung werden in einer XML- oder in HTML-Datei geschrieben. Die Protokollierungsstruktur kann dabei individuell festgelegt werden (vgl. [Tes13] und [BS08]).
- **Selenium**: *Selenium* ist eine Java-basierte Testumgebung für Webanwendungen. Die Ausführung von Testfällen erfolgt über einen Webbrowser. Dadurch können die Testfälle auf verschiedenen Browsern und auf unterschiedlichen Plattformen ausgeführt werden. Aufgrund der unterschiedlichen JavaScript-Implementierungen in den Browsern müssen entsprechende JavaScript-Code Bibliotheken verwendet werden, die Selenium für alle gängigen Browser bereitstellt. Für die Definition und die Ausführung der Testfälle muss die *Selenese* HTML-Syntax verwendet werden. Selenese besteht aus Befehlen zur Manipulation des Prüflings und Methoden, um Elemente einer Webanwendung zu lokalisieren und den Zustand einer Anwendung zu untersuchen (vgl. [Tho13]).

2.4.1 Fokus!MBT

Fokus!MBT ist „eine flexible, erweiterbare Testwerkzeugkette“ die „Software-Projekte in verschiedenen technologischen Domänen“ unterstützt ([WSH10], Seite 3). Entwickelt wird Fokus!MBT am Kompetenzzentrum MOTION im Fraunhofer-Institut für Offene

[30] „Die Zusammenstellung (Aggregation) mehrerer Testfälle für den Test einer Komponente oder eines Systems, bei der Nachbedingungen des einen Tests als Vorbedingungen des folgenden Tests genutzt werden können“ ([IG13], Seite 64).

Kommunikationssysteme auf Basis der Eclipse-IDE. Grundlage bildet die serviceorientierte *ModelBus*-Infrastruktur (vgl. [Fra13]), mit der es möglich ist, Datenmodelle mit unterschiedlichen Werkzeugen konsistent zu verwenden. Adapter, die als Web-Service implementiert sind, bilden die Schnittstelle zu beliebigen, integrierten Werkzeugen. Mit Fokus!MBT lassen sich Testmodelle automatisch aus Systemmodellen ableiten oder manuell erstellen.

Es erfolgt eine strikte Trennung zwischen System- und Testartefakten, um das Systemmodell nicht mit testspezifischen Informationen zu überladen. Außerdem erlaubt die Trennung zwischen System- und Testmodell die Definition von Testmodellen ohne Vorhandensein eines Systemmodells. Damit kann die Testdefinition noch vor Fertigstellung des Systemmodells beginnen und eventuell parallel durchgeführt werden. Fokus!MBT ermöglicht eine automatische Ableitung von Testfällen, sowohl aus Sequenz- als auch aus Zustandsdiagrammen. Aus den Testfällen lässt sich durch Model-to-Text-Transformatoren ausführbarer Testcode generieren. Zurzeit ist die Generierung der Testfälle in TTCN-3 möglich (vgl. [WSH10]). Die Erweiterung auf JUnit ist Aufgabe dieser Arbeit.

Fokus!MBT basiert auf der Idee mit dem Testen so früh wie möglich zu beginnen. Direkt im Anschluss and die Anforderungsspezifikation werden von den Testingenieuren Testanforderungen[31] aus der Anforderungsspezifikation abgeleitet. Dabei kann eine Systemanforderung durch eine Menge von Testanforderungen verifiziert werden. Die manuell erstellten oder generierten Testfälle besitzen Assoziationen zu bestimmten Testanforderungen. Nach der Ausführung der Testfälle, was außerhalb der Fokus!MBT-Umgebung stattfindet, kann die Protokollierung der Testausführung wieder in das Testmodell importiert werden. Dazu werden Testausführungsergebnisse proprietärer Implementierungen in ein *Fokus!MBT test log* Metamodell überführt. Zurzeit ist der Import von TTCN-3-Testausführungsergebnissen, die von der TTworkbench erzeugt wurden, möglich. Von diesem Log-Modell findet dann wiederum eine Transformation in das Fokus!MBT-Testmodell statt. Durch diese Aufteilung ist das Log-Modell unabhängig von der Struktur des Fokus!MBT-Testmodells. Die Testausführungsergebnisse können den entsprechenden Systemanforderungen dann über die definierten Testanforderungen zugeordnet werden (vgl. [WHS13] und [Ulr12]).

Fokus!MBT setzt sich aus Testkonzepten unterschiedlicher Standards wie UML, UTP (siehe Abschnitt 2.1.3 und 2.3), *SysML* der OMG und der Test & Performance Tools Platform (*TPTP*) von Eclipse (vgl. [Ecl13c]) zusammensetzt. Die Konzepte der Testarchitektur mit dem zentralen Testkontext stammen aus dem UTP (siehe Kapitel 2.3.2). Das Testverhalten wird mit vereinfachten UML-Sequenzdiagrammen beschrieben, die um einige testrelevante Konzepte, wie z.B. die Verwendung von Zeitkonzepten erweitert wurden. Das Testdaten-Paket verwendet die datenbezogenen Konzepte des UTP (vgl. 2.3.4), sowie Wildcards und Kodierungsregeln. Mit Hilfe der Zeitkonzepte lassen sich Timer innerhalb von Interaktionen verwenden und mittels Zeitzonen können Testkomponenten aus verteilten Testumgebungen synchronisiert werden (vgl. [WSH10]).

[31] „Eine Einheit oder ein Ereignis, z.B. eine Funktion, eine Transaktion, ein Feature, ein Qualitätsmerkmal oder ein strukturelles Element einer Komponente oder eines Systems, welche bzw. welches durch einen oder mehrere Testfälle verifiziert werden kann“ ([IG13], Seite 58).

2.4.2 TTCN-3

Die *TTCN-3* (Testing and Test Control Notation) ist eine vom *ETSI* (European Telecommunication Standards Institute) erarbeitete „Testspezifikations- und -implementierungssprache“, „mit deren Hilfe Tests textuell oder grafisch entwickelt und automatisiert zur Ausführung gebracht werden können“ ([Sch08], Seite 1).

Das zu testende System wird als Black-Box betrachtet. Zwischen der Testumgebung und dem Testobjekt findet eine nachrichtenbasierte Kommunikation über definierte Schnittstellen, sogenannte Ports, statt. Ein Testfall sendet einen Stimulus über einen Port an die SUT und empfängt und bewertet die Antworten von dieser. Das zu testende System wird in TTCN-3 mit Hilfe von Modulen beschrieben. Hier erfolgt die Definition der Nachrichtentypen, Ports, Templates, der Testfälle und der Kontrolle über die Testfallausführung. Jede Testkomponente besitzt ihr eigenes lokales Ergebnis. Das globale Testergebnis setzt sich aus den Einzelergebnissen der Testkomponenten zusammen. Ist das Ergebnis von nur einer Testkomponente ein Fehlschlag, so ist auch der gesamte Testfall fehlgeschlagen. Die Testfälle können entweder textuell, grafisch oder tabellarisch definiert werden. Dadurch können TTCN-3-Testfälle sowohl von Entwicklern als auch von Personen ohne Programmierkenntnissen definiert und verstanden werden.

TTCN-3 unterstützt verschiedene Programmiersprachen wie C/C++, Java oder C# und bietet Zugänge für eine Vielzahl von Schnittstellen-Technologien wie ASN.1 und XML. Durch die erweiterte Dokumentationsfunktionalität kann die Lesbarkeit, Wiederverwendbarkeit und Wartbarkeit der Tests erhöht werden. TTCN-3 ist für unterschiedliche Teststufen geeignet, insbesondere für Integrations-, System- und Abnahmetests. Außerdem sind auch nichtfunktionale Tests, wie z.B. Leistungstests, möglich (vgl. [Sch08] und [Zec03]).

2.4.3 JUnit

JUnit ist ein einfaches Unittest-Framework[32] zum Schreiben und Ausführen von wiederholbaren Tests. Es basiert auf der *XUnit*-Architektur, welche 1999 von Kent Beck für die Sprache Smalltalk entwickelt wurde (vgl. [Bec98]). Es ist de facto das Standard Test-Framework für die Programmiersprache Java (vgl. [Wes05]). JUnit liegt mittlerweile in der Version 4.11 vor und ist standardmäßig in den Java Developer Tools von Eclipse als auch in praktisch allen anderen modernen Entwicklungsumgebungen integriert. Es wird vor allem für das Testen von einzelnen Komponenten, in Java Klassen und Methoden, verwendet. Dazu wird das aktuelle Verhalten eines Testfalls mit dem erwarteten verglichen. Mit Hilfe von Annotationen lassen sich beliebige Methoden als Testfall und Klassen als Testklasse definieren.

JUnit kennt drei unterschiedliche Zustände für das Ergebnis einer Testausführung. Bei einer erwarteten Abweichung wird eine Ausnahme vom Typ *AssertionError* geworfen (*Failure*). Alle übrigen Ausnahmen entsprechen einer unerwarteten Abweichung von Ist- und Sollverhalten und resultieren in einem Fehler (*Error*). Tritt keine Ausnahme auf, wird

[32] „Ein Werkzeug, das eine Umgebung für einen Komponententest bereitstellt. In dieser Umgebung wird die Komponente isoliert oder mit geeigneten Treibern und Platzhaltern getestet. Darüber hinaus wird dem Entwickler zusätzliche Unterstützung (z.B. Debugging) zur Verfügung gestellt“ ([IG13], Seite 67).

die Testausführung als erfolgreich interpretiert.

Mit der Java-Annotation *@Test* kann eine öffentliche, rückgabefreie Methode ohne Parameter als Testfall deklariert werden. Zum Ausführen erzeugt JUnit eine neue Instanz der Klasse und ruft die annotierte Methode auf. Alle Ausnahmen, die im Testfall auftreten, werden als Failure interpretiert. Tritt keine Ausnahme auf, wird angenommen, dass der Testfall erfolgreich war. Die *@Test*-Annotation besitzt einen optionalen Parameter *expected*, mit dem angegeben werden kann, dass eine bestimmte Ausnahme innerhalb des Testfalls erwartet wird. Der Testfall schlägt fehl, wenn keine Ausnahme auftritt oder der Typ der Ausnahme von der Deklaration abweicht. Die erwartete Ausnahme muss eine Klasse sein, die von *Throwable* abgeleitet ist (siehe Listing 1). Außerdem kann mit dem zweiten, optionalen *timeout*-Parameter eine Zeitspanne in Millisekunden angegeben werden. Falls die Ausführung länger dauert als definiert, schlägt der Testfall fehl (siehe Listing 1).

```
@Test (expected = ClassNotFoundException.class, timeout = 1000)
public void testCase() {
    // ...
}
```

Listing 1: Testfall mit erwarteter Ausnahme und *timeout*-Parameter

Mit den Annotationen *@BeforeClass*, *@Before*, *@After* und *@AfterClass* können Methoden definiert werden, die vor bzw. nach einem Testfall ausgeführt werden. Dadurch lässt sich gemeinsames Verhalten auslagern. Die eigentlichen Testfälle werden dadurch schlanker und beinhalteten nur noch das eigentliche Testverhalten. Dies kann z.B. dafür verwendet werden, um eine Verbindung zu einer Datenbank auf- und abzubauen oder Objekte zu erzeugen, die von einer Menge von Testfällen benötigt werden. *BeforeClass*- und *AfterClass*-Methoden müssen statisch sein und werden genau einmal vor bzw. nach allen Testfällen ausgeführt. Eine mit *@Before* oder *@After* annotierte Methode wird jeweils einmal vor bzw. nach jedem Testfall aufgerufen (siehe Listing 2). Zu beachten ist, dass die Aufrufreihenfolge von Methoden mit diesen Annotationen nicht deterministisch ist.

Die Klasse *org.junit.Assert* enthält eine Menge von statischen Methoden für alle primitiven Typen, Objekte und Felder, die zum Vergleich von Soll- und Ist-Werten verwendet werden können. Bei der Verwendung von Gleitkommazahlen dient ein weiterer Parameter als Toleranz. Außerdem lässt sich bei jeder Methode eine optionale Nachricht für den Fehlerfall angeben (siehe Abbildung 15). Bei Ungleichheit von aktuellen und erwarteten Werten wird eine Ausnahme vom Typ *AssertionError* geworfen.

Mit Hilfe der *@Suite*-Annotation lässt sich eine beliebige Anzahl von Testklassen gemeinsam ausführen. Die Testklassen werden dann in der angegeben Reihenfolge durchlaufen. Die Ausführungsreihenfolge der einzelnen Testmethoden in einer Testklasse ist nicht spezifiziert. Ist allerdings eine bestimmte Reihenfolge der Testmethoden in einer Testklasse erwünscht, kann die Klasse mit *FixMethodOrder(MethodSorters.NAME_ASCENDING)* annotiert werden. Die Testmethoden werden dann lexikografisch nach ihren Methodennamen ausgeführt. Mit *@RunWith* werden die Testfälle mit dem angegebenen *Runner* anstelle des

```
@BeforeClass
public static void setUpClass() {
}

@Before
public void setUp() {
}

@After
public void tearDown() {
}

@AfterClass
public static void tearDownClass() {
}
```

Listing 2: JUnit *BeforeClass*-, *Before*-, *After*- und *AfterClass*-Methoden

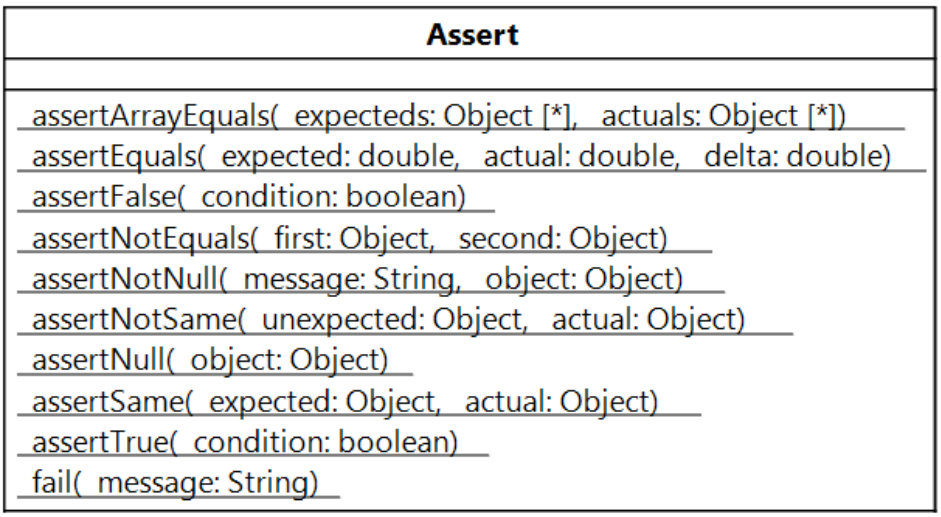

Abbildung 15: *Assert*-Klasse mit einer Auswahl von Methoden

standardmäßigen *JUnit-BlockJUnit4ClassRunner* ausgeführt (siehe Listing 3).

```
@RunWith(Suite.class)
@Suite.SuiteClasses({TestClass1.class, TestClass2.class})

public class TestSuite1 {
}
```

Listing 3: *TestSuite*-Klasse mit zwei Testklassen

Mit Hilfe von Regeln lässt sich das Verhalten einer Testmethode erweitern oder neu definieren. In JUnit gibt es eine Reihe von Regeln, die verwendet oder erweitert werden können. Außerdem ist es möglich seine eigenen Regeln zu definieren. Die *Timeout*-Regel dient dazu, eine Time-out-Zeit in Millisekunden zu definieren, die für alle Testmethoden einer Testklasse gilt (siehe Listing 4).

```
@Rule
public Timeout globalTimeout = new Timeout(1000);
```

Listing 4: *Timeout*-Regel

Der *ExpectedException*-Regel kann außer dem Typ der erwarteten Ausnahme auch deren genaue Nachricht angegeben werden (siehe Listing 5).

```
@Rule
public ExpectedException thrown = ExpectedException.none();

@Test
public void shouldTestExceptionMessage()
        throws IndexOutOfBoundsException {
    List<Object> list = new ArrayList<Object>();

    thrown.expect(IndexOutOfBoundsException.class);
    thrown.expectMessage("Index: 0, Size: 0");
    list.get(0);
}
```

Listing 5: *ExpectedException*-Regel

Der Name des aktuellen Testfalls lässt sich mit der *TestName*-Regel innerhalb der Testmethode verwenden (siehe Listing 6). Der Name ist auch schon in der *@Before*-Methode, welche vor dem Testfall ausgeführt wird, verfügbar.

```
@Rule
public TestName name = new TestName();

@Test
public void testA() {
    assertEquals("testA", name.getMethodName());
}
```

Listing 6: *TestName*-Regel

Mit der *TemporaryFolder*- und der *ExternalResource*-Regel lassen sich innerhalb eines Testfalls temporäre Dateien, Ordner und externe Ressourcen (z.B. ein Server, ein Socket oder eine

Datenbankverbindung) erzeugen, die, unabhängig vom Ergebnis der Testausführung, wieder gelöscht werden. Mit dem *Parameterized*-Runner lassen sich die Testfälle einer Testklasse parametrisieren. Bei der Ausführung wird für jede mögliche Kombination aus Testmethode und Testdatenelement eine Instanz erstellt und ausgeführt. Um die individuellen Testfälle zu identifizieren, kann mit Hilfe der *@Parameters*-Annotation als Parameter ein Name für die Testfälle angeben werden. Mit *index* wird der Index des aktuellen Parameters und mit *0*, *1*, ... der Wert des ersten, zweiten, ... Parameters spezifiziert (vgl. [BGSC13]).

3 Problemstellung und Zielsetzung

Das Ziel dieser Arbeit besteht darin einen Codegenerator zu implementieren, der aus UTP-Modellen ein testobjektspezifisches JUnit-Testausführungsframework und den entsprechenden Testcode erzeugt.

Der Testcodegenerator wird in das proprietäre Werkzeug Fokus!MBT eingebunden. Mit Hilfe von Fokus!MBT wird ein Testmodell erstellt, das alle Informationen enthält, die für die Testcodegenerierung notwendig sind. Dazu zählen die Schnittstellenbeschreibungen der SUT, die logischen Testfälle und die für die Testausführung notwendigen Testdaten.

Die logischen Testfälle werden als simplifizierte UML-Sequenzdiagramme beschrieben, die mit dedizierten UTP-Testkonzepten angereichert sind. Für die Generierung des Testcodes können sowohl generierte als auch manuell abgeleitete Testfälle verwendet werden. Der Testcodegenerator erzeugt dann aus den logischen Testfällen konkreten Testcode in einem hohen Abstraktionsniveau. Durch diese Abstraktion können die Phasen des Testentwurfs und der Testrealisierung soweit vorbereitet werden, dass die Testdurchführung später ohne Verzögerung angestoßen werden kann. Wenn der zu testende Code letztendlich vorliegt, müssen für die Testausführung lediglich die Schnittstellen zum Testobjekt implementiert werden. Dieser Ansatz des „frühen Testens“ deckt sich mit den Gedanken des V- bzw. des W-Modells (siehe Abschnitt 2.2.2), in dem die Testvorbereitungen beginnen, sobald die konsolidierten und abgenommen Arbeitsergebnisse einer Konstruktionsphase vorliegen.

Ein Testmodell trennt logische Informationen von den technischen eines Systemmodells und beschreibt wie ein System zu testen ist. Technische Aspekte werden vernachlässigt und das von außen beobachtbare Verhalten wird betont. Aufgrund dieser Trennung unterscheiden sich die im Testmodell definierten Schnittstellen des Testobjekts von den technischen der konkreten SUT. Die Kommunikation zwischen den Testfällen und der SUT erfolgt deshalb über eine Adapterschicht. Die Informationen der logischen Testfälle werden im Adapter konkretisiert und an das Testobjekt weitergeleitet. Die Resultate der SUT sind zu detailliert um mit den erwarteten Modellergebnissen verglichen zu werden und müssen daher für die Überprüfung wieder abstrahiert werden. Der Adapter kann erst implementiert werden, wenn der zu testende Code letztendlich vorliegt. Die logisch-technische Abbildung kann dabei nicht automatisiert werden und muss manuell von einem Testentwickler durchgeführt werden.

In TTCN-3 haben alle Adaptermethoden die gleiche Struktur und besitzen eine Menge von Fallunterscheidungen, um auf die standardisierten Parameter geeignet reagieren zu können (siehe Listing 7). Im Gegensatz zu TTCN-3 wird der JUTP-Adapter testobjektspezifisch generiert. Für jede SUT-Schnittstelle wird im Adapter eine eigene, an die Schnittstellenbeschreibung angepasste, Methode bereitgestellt. Dadurch wird eine aussagekräftige Adapterschicht erzielt, wodurch die Adapterkomplexität reduziert und dessen Implementierung vereinfacht wird.

„JUnit gilt als das am weitesten verbreitete Unit-Test-Framework für die Programmiersprache Java“ ([SBB12], Seite 146). Aufgrund der daraus resultierenden hohen Nutzerakzeptanz ist der generierte Testcode von JUnit ausführbar. JUnit fehlen allerdings einige wichtige,

```
public TriStatus triCall(TriComponentId componentId,
    TriPortId portId, TriAddress sutAddress,
    TriSignatureId signatureId, TriParameterList parameterList) {

  if (signatureId.getSignatureName().endsWith("addEntry")) {
    return addEntry(componentId, tsiPortId, sutAddress,
                    signatureId, parameterList);
  }

  if (signatureId.getSignatureName().endsWith("clear")) {
    return clear(componentId, tsiPortId, sutAddress,
                 signatureId, parameterList);
  }

  throw new RuntimeException("TA could not handle call to "
    + signatureId.getSignatureName());
}
```

Listing 7: Adaptermethode in TTCN-3

testrelevante Konzepte. Für die Rückspielung der Testausführungsergebnisse ist eine geeignete Protokollierung notwendig. Ein JUnit-Ergebnis setzt sich nur aus der Durchführungszeit einer Testmethode, dem Testergebnis (Pass, Error oder Failure) und einer optionalen Fehlermeldung zusammen. Es sind keine Informationen über das erwartete und das aktuelle Ergebnis, den genauen Start- und Endzeitpunkt einer Testmethode oder die verwendeten Testdaten verfügbar. JUnit besitzt keine Kenntnisse von einigen nützlichen UTP-Konzepten wie z.B. einem Arbiter mit dem das Testergebnis individuell interpretiert werden kann. Der Vergleich von Ist- und Soll-Verhalten wird in JUnit mit Hilfe der *assert*-Methoden (siehe Abbildung 15) durchgeführt. Mit diesen Methoden werden zwei Werte vom selben Typ auf Gleichheit überprüft. Bei der Spezifikation von Testdaten können aber neben konkreten Werten auch Intervalle, reguläre Ausdrücke oder Wildcards verwendet werden, die sich mit JUnit nicht vergleichen lassen.

Um diese Nachteile von JUnit zu kompensieren, wird zusätzlich zu dem eigentlichen Testcode ein Testausführungsframework erzeugt. Zu diesem gehören die Testausführungskomponente, in der die gesamte Testfalllogik in Form von JUnit-Testmethoden integriert ist, Komparatoren für die Analyse des erwarteten SUT-Verhaltens, die Bereitstellung der Testdaten, eine Arbiterimplementierung zur Bestimmung des Testfall-Verdicts und eine Reihe von Hilfsfunktionen.

Durch das Testframework können die Testfälle theoretisch auch unabhängig von JUnit ausgeführt werden. Dazu können die Testmethoden manuell aufgerufen werden. Ist- und Soll-Verhalten wird mir Hilfe der Komparatoren überprüft und der Arbiter ist für die Bestimmung des entsprechenden Verdicts zuständig. Die Ausführungsergebnisse werden protokolliert und lassen sich in Fokus!MBT zurückspielen und auswerten. Dazu muss lediglich der JUnit-Code, der sich ausschließlich in der Testausführungskomponente befindet,

entfernt werden.

In JUTP, wie auch in TTCN-3 und UTP, besteht ein Testfall aus einer Menge von Nachrichten mit denen das Verhalten der SUT überprüft werden soll. Entspricht der Wert einer Nachricht nicht dem erwarteten, wird das lokale Verdict auf *fail* gesetzt und die nächste Nachricht wird ausgeführt und ausgewertet solange bis alle Nachrichten im Testfall durchlaufen wurden. Das globale Ergebnis des Testfalls wird dann mit Hilfe eines Arbitrierungsalgorithmus bestimmt. Dieser Algorithmus kann z.B. bestimmen, dass das globale Verdict auf *fail* gesetzt wird, sobald ein einzelnes lokales Verdict den Wert *fail* zugewiesen bekommt. Auch dann werden trotzdem noch alle weiteren Nachrichten innerhalb des Testfalls durchlaufen.

Mit dem Testausführungsframework kann die Ausführung eines Testfalls abgebrochen werden sobald sich Ist- und Soll-Werte unterscheiden. Ist dies der Fall, wird der aktuelle Testfall beendet und der nächste Testfall wird abgearbeitet. Bei einer großen Anzahl von Testfällen mit vielen Nachrichten kann so die Testausführungszeit deutlich verringert werden, da nicht immer alle Testfälle vollständig durchlaufen werden müssen.

3.1 Grenzen der Automatisierung

Innerhalb eines generierten JUnit-Testfalls erfolgt eine Stimulation der SUT um das erwartete mit dem aktuellen Verhalten zu vergleichen. Die Stimulation erfolgt in Form eines Methodenaufrufs mit den dazu benötigten Testdaten. Zum Zeitpunkt der Erstellung des Testmodells, das als Grundlage für die Generierung dient, sind die konkreten Schnittstellen des Testobjekts in der Regel (noch) nicht bekannt. Dies liegt vor allem daran, dass mit dem Fokus!MBT-Werkzeug das modellbasierte Testen bereits in einer sehr frühen Softwareentwicklungsphase beginnt.

Aus diesem Grund kann aus dem JUnit-Testcode die SUT nicht direkt angesprochen werden, da deren spätere Schnittstellendefinitionen noch nicht spezifiziert wurden. Die Kommunikation mit dem Testobjekt erfolgt daher über eine Adapterschicht.

Der Adapter fungiert als Übersetzer zwischen der „Sprache“ der Testausführungsumgebung und der „Sprache“ der SUT. Innerhalb der Testfälle werden die logischen Testdaten zunächst über einen Methodenaufruf an den Adapter weitergeleitet. Innerhalb des Adapters kann es notwendig sein, die Testdaten in die von der SUT erwartete Form zu transformieren. Nach dieser Transformation erfolgt die eigentliche Stimulation des Testobjekts mit den angepassten Daten. Die SUT reagiert auf die Stimulation und liefert gegebenenfalls als Ergebnis einen Wert an den Adapter zurück. Im Adapter muss wiederum eine Dekodierung der SUT-Daten in das Typsystem der Testausführungsumgebung stattfinden. Danach sendet der Adapter die abstrakten Objekte, in der geforderten Form, zurück an einen Testfall. Im Testfall ist nun der Vergleich zwischen dem erwarteten und dem von der SUT erhaltenen Wert möglich.

Dabei werden im Adapter lediglich Methodenrümpfe generiert. Die eigentliche Anpassung muss manuell von einem Testentwickler implementiert werden, sobald die konkreten Schnittstellendefinitionen des zu testenden Systems vorliegen.

4 Einführung in die Fokus!MBT-Modellierungsmethodik

JUTP soll in Fokus!MBT (siehe Abschnitt 2.4.1) eingebunden werden. Die Vorgehensweise von der Definition der Testanforderungen bis zur Rückspielung und Analyse der Testausführungsergebnisse ist in Abbildung 16 zu sehen. Zunächst werden in Fokus!MBT eine Menge von Testanforderungen definiert, die sich auf genau eine Anforderung aus der Anforderungsspezifikation beziehen.

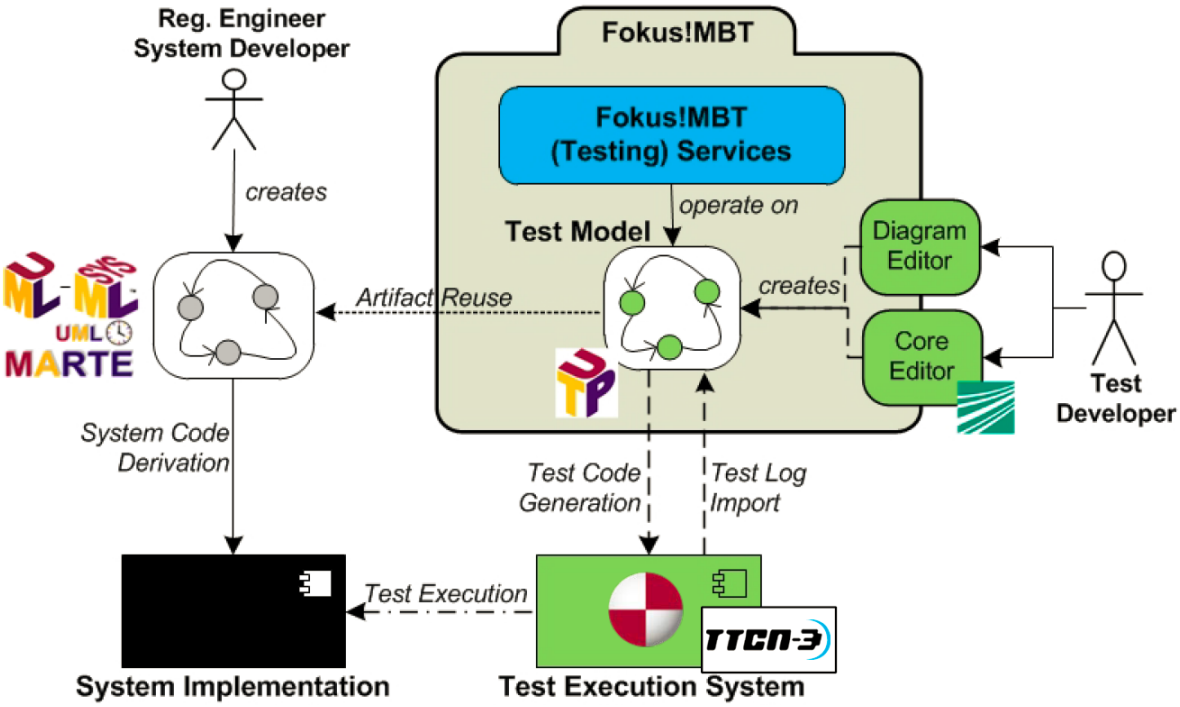

Abbildung 16: Fokus!MBT Konzept nach Wendland, Hoffmann und Schieferdecker (vgl. [WHS13])

Alle testrelevanten Informationen werden in einem Testmodell spezifiziert. Die dazu nötigen Informationen stammen aus dem Systemmodell und aus der Anforderungsspezifikation. Anhand des Testmodells werden die entsprechenden Testfälle generiert oder modelliert. Für die Kommunikation mit dem Testobjekt müssen die dazu benötigten Schnittstellen definiert werden. Für die Testfallgenerierung ist die Spezifizierung von Testbasismodellen[33] notwendig. Ein Testbasismodell wird als UML-Zustandsdiagramm modelliert. Wie Testfälle und die dazu notwendigen Testdaten in Fokus!MBT modelliert werden, ist in Abschnitt 4.1 beschrieben.

Sowohl für die modellierten als auch für die generierten Testfälle müssen bestimmte Mo-

[33]Testbasismodelle sind konzeptuelle Modelle die eine Testbasis beschreiben. Unter einer Testbasis versteht man alle „Dokumente, aus denen die Anforderungen ersichtlich werden, die an eine Komponente oder ein System gestellt werden, bzw. die Dokumentation, auf der die Herleitung oder Auswahl der Testfälle beruht“ ([IG13], Seite 57).

dellierungsrichtlinien beachtet werden, damit die Transformation in JUnit-Testfälle möglich ist. Die Einschränkungen und die Gründe für diese Notwendigkeit werden in Abschnitt 4.2 erläutert.

4.1 Testfallmodellierung mit Fokus!MBT

Bevor man mit Fokus!MBT Testfälle spezifizieren kann, wird ein Testmodell erstellt indem die Schnittstellen des Testobjekts modelliert werden. Zu jeder Anforderung können eine Menge von Testanforderungen definiert werden, welche als Bedingungen zur Verifikation von Anforderungen bzw. einzelnen Merkmalen von Anforderungen dienen. Jede Testanforderung besitzt eine eindeutige ID, einen Namen und eine textuelle Beschreibung, in der das zu testende Verhalten dargestellt wird. Eine Testanforderung ist in Fokus!MBT durch eine *requirements verification dependency* (Konzept der *SysML* (Systems Modeling Language), vgl. [OMG13e]) mit einer Anforderung assoziiert.

Zu jeder Teststufe[34] können eine Menge von Testobjekten mit dem benötigten Detaillierungsgrad erstellt werden. Die Schnittstellen eines Testobjektes werden mit Hilfe von UML-Interfaces definiert. Hier werden Operationen und die dazu benötigten Datentypen definiert, mit denen die Kommunikation mit der SUT erfolgt. Zu einem Testobjekt lassen sich Ports erstellen, die einer Menge von SUT-Interfaces zugeordnet sind.

Aus diesen Informationen können automatisiert Testkonfigurationen und Testarchitekturen in Form von UML-Kompositionsstrukturdiagrammen und UML-Klassendiagrammen generiert werden (vgl. [Kar12]). Eine Testkonfiguration beschreibt wie Testkomponenten mit dem zu testenden System über bestimmte Ports verbunden sind.

Die Modellinformationen, und die noch zu erstellenden Testdaten, dienen als Grundlage für die Erstellung von Testfällen. Testfälle beziehen sich dabei immer auf einen bestimmten Testkontext und verfügen über die entsprechenden Informationen. Ein Testfall wird als UML-Sequenzdiagramm mit erweiterten UTP-Konzepten modelliert. SUT und Testkomponente werden als Lebenslinie repräsentiert, über die die Kommunikation mit Hilfe von Nachrichten stattfindet. Es können asynchrone Aufrufe oder Signale oder synchrone Aufrufe verwendet werden. Zu einer synchronen Nachricht gehört eine bestimmte Rückantwort. Über einen Modellierungsassistenten kann der Port und ein bestimmtes Verhalten, in Form einer Operation eines SUT-Interfaces, einer konkreten Nachricht zugeordnet werden.

Besitzt eine Operation Parameter oder erwartet einen Wert als Rückantwort, müssen diese als Testdaten modelliert werden. Dazu verwendet Fokus!MBT das UTP-Testdatenkonzept (siehe Abschnitt 2.3.4). Es werden Datenpartitionen erstellt, die von einem bestimmten Datentyp abgeleitet sind. Für die Definition eines konkreten Datenwertes wird eine UML-Instanzspezifikation für eine Datenpartition erstellt. Neben einfachen Literalen für Zahlen, booleschen Werten oder Zeichenketten können als Parameter auch die aus dem UTP bekannten Wildcards, reguläre Ausdrücke, Intervalle und Instanzwerte verwendet werden.

[34] „Eine Teststufe ist eine Gruppe von Testaktivitäten, die gemeinsam ausgeführt und verwaltet werden. Teststufen sind mit Zuständigkeiten in einem Projekt verknüpft“ ([IG13], Seite 61).

Als Rückantwortwert sind nur konkrete Instanzspezifikationen oder der UTP-Platzhalter *LiteralAny* erlaubt, da dieser als erwarteter Wert beim Vergleich mit den Testausführungsergebnissen verwendet wird.

Neben der Verwendung von Nachrichten können für die Beschreibung des Testverhaltens auch die aus der UML bekannten kombinierten Fragmente verwendet werden. Die UML-Alternative lässt sich in eine deterministische UTP-Alternative umwandeln. Ebenso stehen auch die UTP-Zeitkonzepte zur Verfügung. Es ist möglich Timer, mit konkreten Zeitwerten zu erstellen, zu starten und zu stoppen und Time-outs zu definieren. Darüber hinaus kann das aktuelle *Verdict* an einer beliebigen Position auf einer Lebenslinie modelliert werden.

4.2 Problemanalyse bzgl. der JUnit-Generierung

Der generierte Testcode sollen mit Hilfe des Test-Frameworks JUnit (siehe Abschnitt 2.4.3), ausgeführt werden können. Der Funktionsumfang von JUnit ist sehr beschränkt. Die Grundfunktionalität besteht ausschließlich aus dem Vergleich eines erwarteten und eines aktuell empfangenen Wertes.

JUnit ist in der Programmiersprache Java geschrieben und ist auch zum Testen von Java-Programmen gedacht. Da Java turing-vollständig ist, lassen sich demnach alle Funktionen berechnen, die auch von einer universellen Turingmaschine berechenbar sind (vgl. [Fis13], Seite 15). Das bedeutet, dass man mit der Programmiersprache Java alle berechenbaren Funktionen implementieren kann. JUnit wird in dieser Arbeit aber nur als eingeschränkter Konzeptraum betrachtet. Um einen lauffähigen Testfall zu erhalten ist es natürlich notwendig, dass sich der JUnit Code in einer Java-Klasse befindet und dass auch sonst die Java-Syntax eingehalten wird. Bei der Generierung des JUnit-Testframeworks wird also darauf geachtet, so weit wie möglich nur Funktionalität des fundamentalen *java.lang*-Paketes zu verwenden. So müssen weniger Objekte über Imports eingebunden werden. Dadurch soll die Komplexität der Testfälle gering gehalten werden, was die Lesbarkeit und die Anpassbarkeit verbessert. Darüber hinaus ist es Notwendig, dass die Testfälle in möglichst vielen Ausführungsumgebungen ohne Anpassung lauffähig sind. So ist die Testausführung an keine spezifischen Werkzeuge gebunden und möglichst universell einsetzbar. Für die Ausführung der Testfälle ist natürlich die JUnit-Funktionalität obligatorisch. Das bedeutet, dass die Ausführungsumgebung neben dem *java.lang*-Paket auch die Funktionen von JUnit (*org.junit*) bereit stellen muss.

In Java kann mit dem Interface *java.lang.Runnable* und der Klasse *java.lang.Thread* paralleles Verhalten implementiert werden. Dazu wird eine neue Klasse erstellt, die von *Runnable* abgeleitet ist und deren *run*-Methode überschreibt, in der das parallele Verhalten definiert wird. Im Anschluss wird ein Objekt der Klasse *Thread* erzeugt, das im Konstruktor eine Instanz der vorhin erstellten *Runnable*-Klasse übergeben bekommt. Mit der *start*-Methode der Thread-Klasse wird der Code der *run*-Methode in einem neuen Thread nebenläufig ausgeführt (vgl. [Ull11], Kapitel 14). Somit ist es in Java durchaus möglich Aktionen parallel auszuführen. Komplikationen treten aber auf, wenn in einem Testfall eine Testkomponente oder eine SUT auf das Eintreffen einer asynchronen Nachricht wartet.

In Abbildung 17 wird von einer Testkomponente eine synchrone Nachricht an die SUT

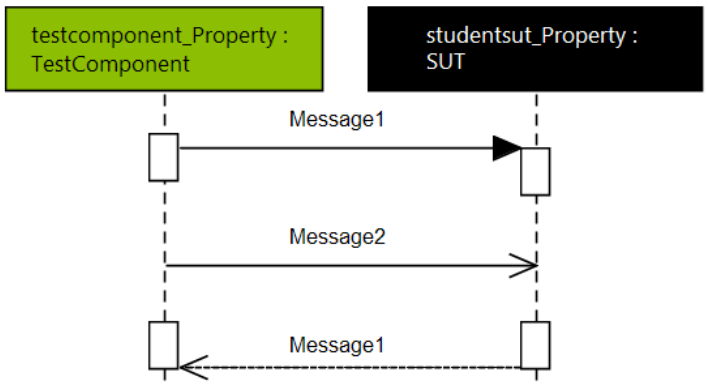

Abbildung 17: Testfall mit asynchroner Nachricht

gesendet. Bevor die Rückantwort dieser Nachricht eintrifft wird eine asynchrone Nachricht an die SUT gesendet. Das Senden der Nachrichten lässt sich relativ einfach, wie oben beschrieben, in Java umsetzten. Das Problem liegt aber beim Empfangen, bzw. beim Warten auf eine Nachricht. Dazu müsste eine Möglichkeit implementiert werden, um während der Ausführung der gesamten Testfalllogik asynchrone Nachrichten zu empfangen, bzw. darauf zu warten, ob das erwartete asynchrone Ergebnis schon vorliegt. Darüber hinaus besitzt die *run*-Methode des *Runnable*-Interface keinen Rückgabewert. Java bietet mit dem Paket *java.util.concurrent* eine Reihe von Möglichkeiten die beschriebenen Probleme zu lösen, was aber zu einer enormen Verkomplizierung des generierten Testfall-Codes führen würde (vgl. [Ull11], Kapitel 14). Aus diesem Grund wird bei der Testfallspezifikation auf Modellierungskonzepte verzichtet, für die eine parallele Ausführung notwendig wäre.

Eine weitere Einschränkung betrifft die Anzahl der Testkomponenten die in einem Testfall verwendet werden können. Abbildung 18 zeigt einen Testfall in dem zwei Testkomponenten mit einer SUT kommunizieren. In dem Beispiel werden nur synchrone Nachrichten verwendet. Nach dem Senden der ersten Nachricht durch die erste Testkomponente und noch bevor die Antwort dieser Nachricht empfangen wird, findet zwischen der zweiten Testkomponente und der SUT ebenfalls eine synchrone Kommunikation statt. Um dieses Testverhalten mit JUnit bzw. mit Java zu beschreiben, müsste jede Testkomponente durch einen eignen Thread beschrieben werden. Die Kommunikation wird in Java durch einfache Methodenaufrufe realisiert. Ohne die Verwendung von Parallelität ist ein Methodenaufruf in Java *atomar*. Das bedeutet, dass es nicht möglich ist, innerhalb eines Methodenaufrufs einen anderen Methodenaufruf durchzuführen. Deshalb darf bei der Testfallspezifikation pro Testfall nur eine einzige Testkomponente verwendet werden.

Weitere Einschränkungen betreffen das UTP-Zeitverhalten. UTP bietet die Möglichkeit Timer zu definieren, die zu einem bestimmten Zeitpunkt oder nach einer bestimmten Zeitspanne ablaufen. Für die Umsetzung dieses Zeitverhaltens bietet Java die Klassen *Timezone*, *Timer* und *TimerTask* aus dem *java.util*-Paket (vgl. [Ull11], Kapitel 14). *Timer* startet dabei aber wiederum einen eigenen Thread der das Zeitverhalten verwaltet. Deshalb wird die Funktionalität bei der Testfallmodellierung auch um das UTP-Zeitverhalten reduziert. Außerdem ist auch die Verwendung des *par*-Operators des kombinierten Fragments unter-

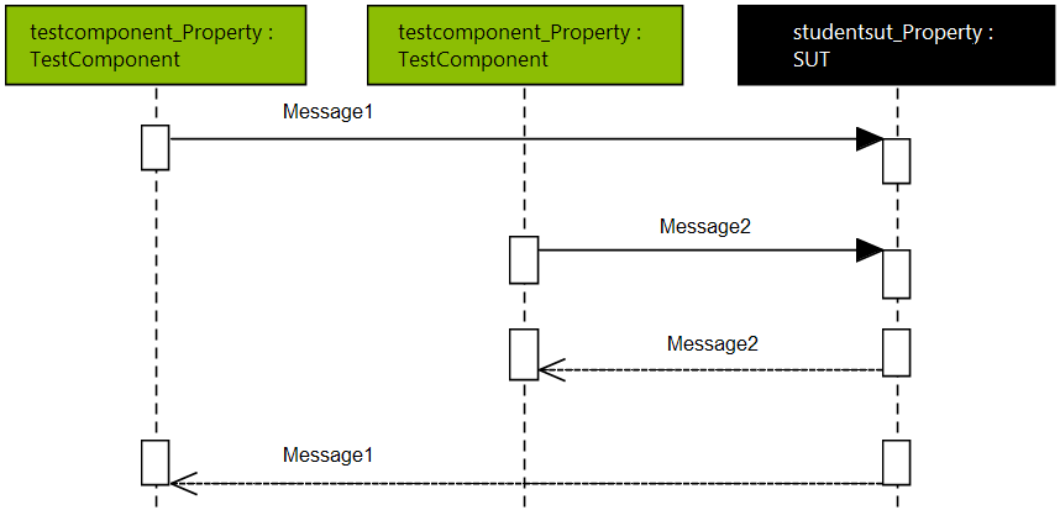

Abbildung 18: Testfall, in dem zwei Testkomponenten mit einer SUT kommunizieren

sagt. Wie der Name des Operators bereits nahelegt, können mehrere Nachrichten parallel empfangen oder gesendet werden.

In der UML kann eine Operation eine Menge von Parametern besitzen. Ein Parameter kann vom Typ *in*, *inout*, *out* oder *return* sein. Die Parameter einer Operation sind in Java immer vom Typ *in*. Deshalb werden *in*-, *inout*- und *out*-Parameter als „normale“ Java-Parameter angesehen. Der UML-*return*-Parameter ist äquivalent zu einem *return*-Parameter in Java.

UTP kennt neben den Verdicts *none*, *pass*, *fail* und *error* auch *inconclusive*. In JUnit kann ein ergebnisloser Zustand nicht auftreten. Um die Konsistenz zur UTP zu wahren, enthält der generierte Aufzählungstyp Verdict auch alle UTP-Aufzählungsliterale. Ergebnislose Zustände werden deshalb automatisch als *fail* angesehen.

In der UML existieren die Modellelemente *Interface*, *DataType* und *Class*, die alle von der abstrakten Metaklasse *Classifier* abgeleitet sind. Diese Objekte können eine Menge von Eigenschaften und Operationen besitzen und werden bei der Testfallgenerierung als Java-Klassen interpretiert. Dabei muss beachtet werden, dass bei der Modellierung alle Interfaces, Datentypen und Klassen eindeutige Namen besitzen. Bei gleicher Namensgebung wird der erste, vorkommende *Classifier* verwendet. Ein weiteres Element mit gleichen Namen wird dann nicht mehr berücksichtigt.

Die Kommunikation zwischen einer Testkomponente und dem Testobjekt erfolgt über Nachrichten. Die Nachrichten werden in einem Sequenzdiagramm entsprechend dem Zeitverlauf von oben nach unten modelliert. Dabei bietet die UML keine eindeutige Möglichkeit die einzelnen Nachrichten voneinander zu unterscheiden. Aus diesem Grund ist es notwendig, dass alle Nachrichten innerhalb eines Testfalls einen eindeutigen Namen besitzen. Nur über diesen Namen sind die verwendeten Nachrichten unterscheidbar.

Ebenso muss der Name des Testfalls innerhalb eines Testkontextes eindeutig sein. Die Testfälle eines Testkontextes werden in eine einzelne Testklasse generiert. Der Testfallname entspricht später dem JUnit-Testmethodenname. Da JUnit-Testfälle parameterlose Methoden sein müssen, darf der Methodenname pro Klasse nur einmal vergeben werden.

5 Konzeption

In diesem Kapitel wird der konzeptionelle Aufbau von JUTP beschrieben. In Abbildung 26, am Ende dieses Kapitels, ist die vollständige JUTP-Architektur zu sehen. Die einzelnen Teile werden in den folgenden Abschnitten gesondert beschrieben. Die Testausführungskomponente beinhaltet alle Testfälle eines Testkontextes (siehe Abschnitt 5.1). Die vollständige Testfalllogik ist ausschließlich in dieser Komponente abgebildet.

Für die Stimulation der SUT werden Testdaten benötigt. Abschnitt 5.2 beschreibt wie den entsprechenden Testfällen die benötigten Testdaten bereitgestellt werden. Komparatoren vergleichen die Rückgabewerte des Testobjektes mit den erwarteten Werten, um eine Aussage über das Testergebnis treffen zu können. Das Testergebnis wird mit Hilfe einer Arbiterimplementierung bestimmt. Außerdem ist es möglich, die Testausführung bei der ersten Abweichung von Soll- und Ist-Ergebnis zu beenden.

Die Kommunikation zwischen der Testausführungskomponente und der SUT erfolgt über eine Adapterschicht. Der Adapter transformiert das logische Typsystem in das technische der SUT. Die Antworten des Testobjekts müssen wiederum in das logische Typsystem dekodiert werden. Dazu dient eine Adapterklasse, die in Abschnitt 5.5 spezifiziert wird.

Um die Testausführungsergebnisse mit Fokus!MBT analysieren zu können, müssen die Testergebnisse protokolliert werden. Abschnitt 5.6 beschreibt zwei unterschiedliche Logging-Möglichkeiten die in die Testausführungskomponente integriert werden können.

Mit Hilfe des Testausführungsframeworks ist es theoretisch möglich die Testfälle auch unabhängig von JUnit auszuführen. Der Vergleich des erwarteten SUT-Verhaltens findet in den Komparatoren statt. Das Ergebnis-Verdict eines Testfalls wird mit einem Arbiter ermittelt und alle Testausführungsergebnisse werden Protokolliert. Am Ende eines Testfalls wird das globale Verdict an JUnit übergeben, dass dann nur noch dafür zuständig ist das Resultat anzuzeigen.

5.1 Testausführungskomponente

In Fokus!MBT lässt sich für jede Teststufe ein Modell erstellen welches das zu testende System in dem benötigen Detaillierungsgrad beschreibt. Jedes dieser Testmodelle besteht aus einer Menge von Testobjekten, Testkomponenten und Testkontexten und deren Schnittstellen- und Verhaltensbeschreibungen.

Für jeden Testkontext wird ein unabhängiges Paket generiert, das eine beliebige Anzahl von Testfällen, einen Adapter und eine Testausführungskomponente beinhaltet. Diese Testausführungskomponente (TestClass, siehe Abbildung 19), bildet die Grundlage für die Ausführung eines Testfalls und besteht aus einer einzelnen Java-Testklasse. In der Testklasse befinden sich alle Testfälle eines Testkontextes. Zunächst werden alle zur Testausführung benötigten Objekte angelegt. Dazu gehört ein Arbiter- (siehe Abschnitt 5.4), ein Adapter- (siehe Abschnitt 5.5) und ein Logging-Objekt (siehe Abschnitt 5.6).

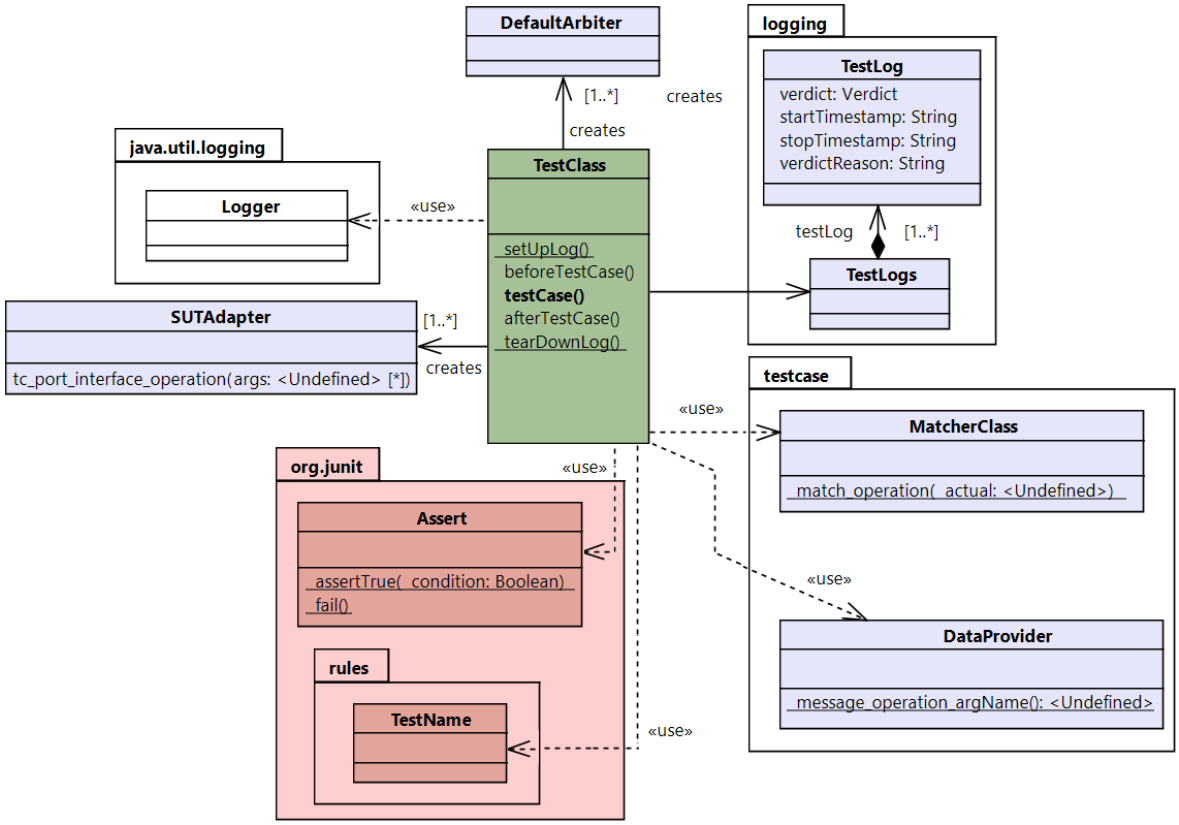

Abbildung 19: Assoziierte Komponenten der Testklasse

Die Testausführungskomponente verwendet die JUnit-Annotationen *@Rule*, *@BeforeClass*, *@Before*, *@After*, *@AfterClass* und *@Test* (vgl. Abschnitt 2.4.3). Mit der *TestName*-Regel ist der Zugriff auf den Namen des aktuellen Testfalls möglich. In der mit *@BeforeClass* annotierten *setUpLog*-Methode wird der Logger erzeugt. Dieser Logger wird während der gesamten Lebensdauer der Testklasse verwendet.

Vor der Ausführung eines Testfalls wird stets die *beforeTestCase*-Methode aufgerufen. Hier wird für jeden Testfall ein eigener Arbiter erzeugt, in dem das Resultat des Vergleichs zwischen Soll- und Ist-Verhalten als Verdict festgelegt wird. Danach wird für den Testfall ein Adapter erzeugt und dessen *beforeTestCase*-Methode mit dem Namen des aktuellen Testfalls aufgerufen. Im Anschluss wird der Start des Testfalls mit Hilfe des Loggers protokolliert.

Zu einer Testklasse gehören eine Menge von Testfällen. Ein Testfall entspricht einer JUnit-Testmethode. Jede Testmethode besitzt einen *Timeout*-Parameter der bestimmt wie lange die Ausführung des Testfalls dauern darf. Wird diese Zeitspanne überschritten, schlägt der Testfall fehl. Dadurch werden Verklemmungen vermieden und alle Testfälle können in endlicher Zeit durchlaufen werden.

Innerhalb des Testfalls findet die Überprüfung des SUT-Verhaltens statt. Eine Testkom-

ponente kommuniziert über bestimmte Ports mit dem Testobjekt. Die Ports werden in Fokus!MBT mit Hilfe von Schnittstellen und entsprechenden Operationen beschrieben. In Java findet die Kommunikation nicht über Ports sondern durch einfache Methodenaufrufe statt. Damit sich die Konzepte von Testkomponente, Port und Schnittstellendefinition auch in den generierten JUnit-Testfällen wiederfinden, werden diese im Methodenname berücksichtigt. Nachrichten an die SUT werden zunächst an den Adapter gesendet. Der entsprechende Methodenname beinhaltet den Namen der sendenden Testkomponente, gefolgt vom Portnamen, über den die Nachricht gesendet wird, dem Namen der Schnittstelle, in dem die Operation definiert ist, und dem Namen der Operation. So wird die Konsistenz zur Modellierung bewahrt.

Eine Nachricht kann eine Menge von Parametern benötigen, die als Testdaten von der Klasse *DataProvider* bereitgestellt werden. Ist in der Nachricht ein Rückgabeparameter definiert, wird das von der SUT erhaltene Ergebnis mit Hilfe eines Komparators analysiert. Das Komparatorergebnis wird dann dem Arbiter übergeben, der dazu ein Verdict-Resultat ermittelt.

Im Anschluss an die letzte Testfallnachricht wird das vom Arbiter ermittelte, globale Testfallverdict an JUnit übergeben. JUnit bestimmt anhand dieses Wertes das Ergebnis des aktuellen Testfalls.

Wurde ein Testfall durchlaufen oder die definierte Zeitspanne überschritten, wird die *afterTestCase*-Methode aufgerufen. Hier werden das Ende und das Resultat der Testausführung protokolliert. Darüber hinaus wird die *afterTestCase*-Methode mit dem aktuellen Testfallnamen des Adapters aufgerufen, bevor dessen Ressourcen wieder frei gegeben werden. Im Adapter ist dann noch die Ausführung von finalisierenden Aktionen möglich.

Nach der Ausführung aller Testfälle innerhalb der Testausführungskomponente werden die protokollierten Testausführungsergebnisse in einer XML-Datei gespeichert. Mit Hilfe dieser XML-Datei ist der Import der Testergebnisse mit Fokus!MBT möglich. Die Testausführungsergebnisse können dann mit den entsprechenden Testanforderungen verglichen und ausgewertet werden.

5.2 Testdaten

Um das Verhalten einer SUT zu analysieren sind in der Regel Testdaten notwendig. Diese Daten werden als Parameter einer Nachricht an ein Testobjekt übermittelt. Das zu testende System reagiert auf die Nachricht mit der Rückgabe eines Ergebnisses. Die Testdaten werden mit dem Fokus!MBT-Werkzeug modelliert und sind Teil des Testmodells. Für die Ausführung mit JUnit wird für jeden Testfall eine eigene *DatenProvider*-Klasse generiert, in der die Testdaten erzeugt und bereitgestellt werden (siehe Abbildung 20).

Dabei steht für jeden Nachrichtparameter eine eigene statische Methode zur Verfügung, die das benötigte Objekt liefert. Der Methodenname setzt sich aus dem Namen der Nachricht und dem Parameternamen zusammen. Besteht das benötigte Objekt aus einem primitiven UML-Datentyp (Boolean, Integer, Real oder String), wird dieser direkt zurückgegeben.

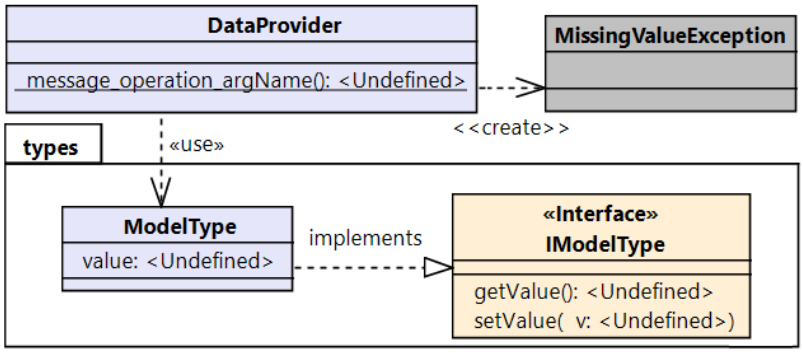

Abbildung 20: Bereitstellung der Testdaten

Komplexe Datentypen oder Enumerationen müssen zunächst im Testmodell modelliert werden. Für jeden Datentyp wird in der Testausführungsumgebung eine Schnittstelle und eine Implementierung generiert, welche Zugriffsmethoden besitzen, damit ein Objekt von diesem Datentyp erstellt werden kann. Die benötigten Testdaten können dann mit Hilfe dieser erzeugt und an den entsprechenden Testfall geliefert werden.

Wurde für einen Parameter kein Wert modelliert, wird stattdessen von der Methode eine *MissingValueException* geworfen. Diese Ausnahme wird im Testfall abgefangen, der aktuelle Testfall bekommt das Verdict *error* zugewiesen und dieses Ergebnis wird der JUnit-Ausführungsumgebung mitgeteilt.

5.3 Komparatoren

Um das Verhalten eines Testobjekts zu überprüfen, muss dieses stimuliert und dessen Reaktion analysiert werden. Dazu sendet man in einem Testfall Nachrichten an die SUT. Das resultierende Ergebnis wird dann mit dem erwarteten verglichen. In der JUnit-Testausführungsumgebung geschieht dies mit Hilfe von Methodenaufrufen.

Eine Methode kann eine Menge von Parametern als Testdaten an das zu testende System übermitteln und als Ergebnis einen Wert von diesem erwarten. Besitz die Methode keinen Ergebniswert (*void*-Methode), wird das Verhalten der SUT als richtig interpretiert, wenn der Methodenaufruf erfolgreich durchlaufen wurde und die Ausführung weiter fortgeführt werden kann. Wird von dem Testobjekt ein Wert als Antwort gefordert, muss dieser mit einem erwarteten Wert verglichen werden. Zu diesem Zweck werden für jeden Testfall eigene Komparatoren in Form von *Matcher*-Klassen generiert (siehe Abbildung 21).

In jedem Komparator steht für jede Nachricht eine statische Methode für den Vergleich zur Verfügung. Als Parameter wird ein Objekt übergeben das als Ergebnis vom Testobjekt geliefert wurde. Innerhalb der Methode wird das Ergebnisobjekt analysiert. Entspricht das Objekt den Erwartungen, liefert die boolesche Methode den Wert *true*, ansonsten *false*. Tritt bei der Überprüfung ein Fehler auf, wird eine *MatchException* geworfen. Die Ausnahme wird im Testfall aufgefangen, das Verdict des Arbiters wird auf *error* gesetzt und

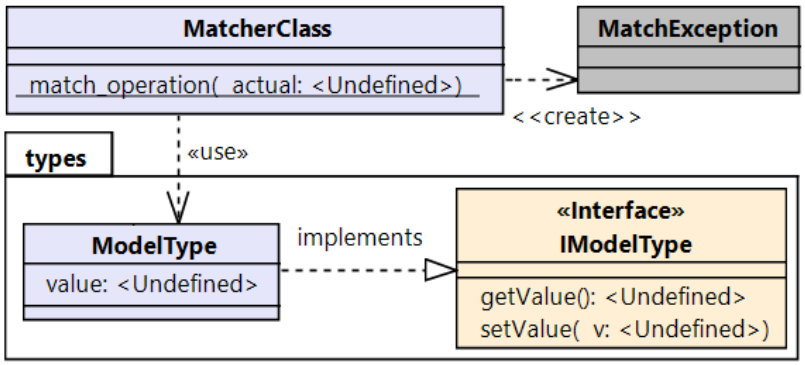

Abbildung 21: Komparator für den Vergleich von Ist- und Sollergebnis

im Anschluss beendet JUnit die Ausführung des aktuellen Testfalls mit einem Fehler.

Das erwartete Ergebnis wird in Fokus!MBT in Form einer Wertspezifikation (ValueSpecification) modelliert (siehe Abbildung 22). Die roten Wertspezifikationen stammen aus der UML, die blauen aus dem UTP und die beiden grünen wurden für die Verwendung im Fokus!MBT-Werkzeug neu eingeführt.

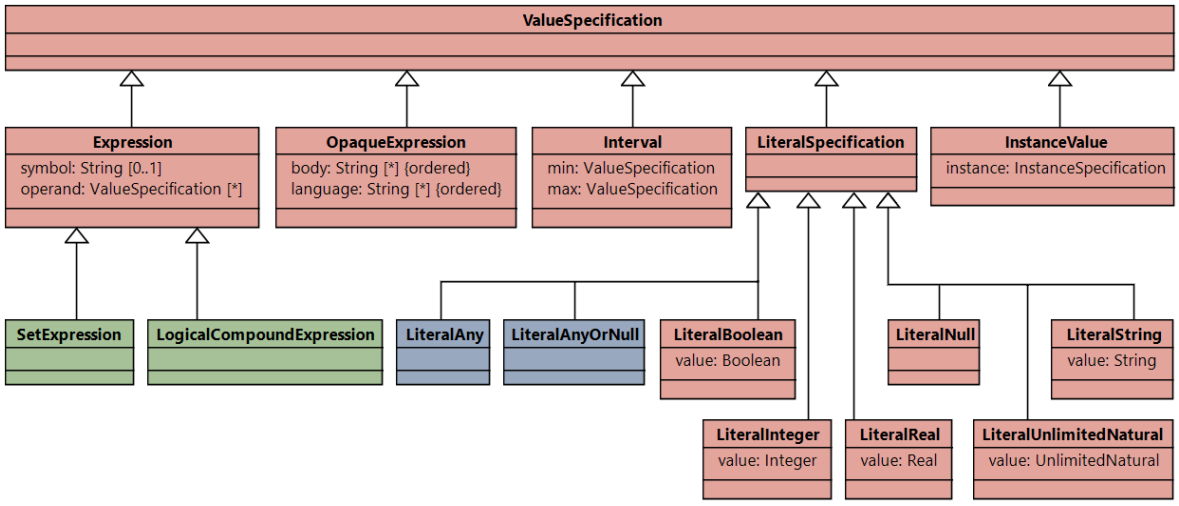

Abbildung 22: Mögliche Ergebniswerte die mit Fokus!MBT modelliert werden können

Für die Modellierung konkreter Werte werden Literale verwendet. Für Werte vom Typ Boolean, Integer, Real, UnlimitedNatural und String besitzen die entsprechenden Literale ein Attribut in dem der konkrete Wert definiert werden kann. Hier kann der Komparator den vom Testobjekt gelieferten Wert mit dem erwarteten vergleichen und das Resultat zurückliefern. Bei einem LiteralNull wird das SUT-Ergebnis auf *null* überprüft.

LiteralAny und *LiteralAnyOrNull* stammen aus dem UTP und definieren, dass ein be-

liebiger Wert bzw. ein beliebiger oder gar kein Wert erlaubt ist. Ein *LiteralAny* wird in einer *Matcher*-Methode auf ungleich *null* überprüft. Bei einem *LiteralAnyOrNull* ist der Vergleich nicht notwendig, da hier alle möglichen Werte als Ergebnis erlaubt sind. Damit das Vorhandensein des Literals auch in einem Komparator zu erkennen ist, wird untersucht ob das SUT-Resultat gleich oder ungleich *null* ist.

Mit Hilfe von Intervallen lassen sich ganze Bereiche mit gültigen Werten definieren. Intervalle besitzen als Attribute einen minimalen und einen maximalen Wert. Dieser Wert kann in der UML wiederum ein beliebiger Wert vom Typ ValueSpecification sein. Um bei der Generierung einen lauffähigen Code zu erzeugen, sind hier einige Einschränkungen nötig. Der minimale und der maximale Wert müssen vom gleichen Typ sein und als *min*- und *max*-Wert sind nur Literale vom Typ *Integer*, *UnlimitedNatural* und *Real* erlaubt. Wird für eine Zeichenkette ein *min*- und ein *max*-Wert vom Typ Integer angegeben, muss sich der String aus mindestens *min* und aus maximal *max* Zeichen zusammensetzten. Ein typischer Vergleich unter Verwendung eines Intervalls mit Integer-Werten ist in Listing 8 zu sehen.

```
if (actual >= 1234 && actual <= 5678) {
    return true;
} else {
    return false;
}
```

Listing 8: Vergleich im Komparator bei einem Intervall

Eine *OpaqueExpression* beinhaltet einen sprachspezifischen Text der zur Beschreibung einer Menge von Werten dient. In dem Attribut *body* kann eine beliebige Anzahl von Zeichenketten definiert werden. In dem dazugehörigen *language*-Attribut ist die Sprache für den entsprechenden Ausdruck spezifiziert. Die Sprache muss eindeutig sein und die Größe des *body*- und des *language*-Attributs müssen übereinstimmen. Bei der Generierung wird nur ein Text berücksichtigt der für die Sprache *Java* definiert wurde. Der Text wird dabei als regulärer Ausdruck angesehen, der zur Laufzeit zu *true* oder *false* ausgewertet werden kann. Ist dies nicht der Fall, ergibt sich ein Syntaxfehler und der generierte Code ist nicht ausführbar. Für den Vergleich im Komparator wird die Methode *matches* der Klasse *String* verwendet (vgl. [Ull11], Kapitel 4). Die Methode liefert *true*, wenn eine Zeichenkette mit einem beliebigen, als Parameter übergebenen, regulären Ausdruck übereinstimmt.

Die beiden von *Expression* abgeleiteten Wertspezifikationen *SetExpression* und *LogicalCompoundExpression* wurden neu in Fokus!MBT eingeführt. Die *SetExpression* spezifiziert eine Menge von einzelnen Literalen oder Intervallen. Die einzelnen Werte müssen dabei vom gleichen Typ sein, damit eine sinnvolle Generierung möglich ist. Das von der SUT gelieferte Ergebnis wird dann mit allen Werten der *SetExpression* verglichen. Die Werte der Menge sind dabei *ODER*-verknüpft. Es muss mindestens ein Wert übereinstimmen damit der Vergleich erfolgreich ist. Bei der *LogicalCompoundExpression* kann das Symbol-Attribut den Wert *AND*, *OR* oder *NOT* besitzen und die Operanden müssen vom Typ *SetExpression* sein. Die einzelnen Mengen lassen sich so *UND*- oder *ODER*-verknüpfen. Außerdem kann die erlaubte Ergebnismenge mit *NOT* negiert werden. Das bedeutet, dass alle beliebigen Werte

erlaubt sind, außer den Werte der *SetExpression*.

Ein Instanzwert spezifiziert mit dem Instanz-Attribut den Wert, der durch eine Instanzspezifikation modelliert ist. Eine Instanzspezifikation ist ein Modellelement das eine Instanz repräsentiert und besitzt als Attribut eine Menge von Slots und eine optionale Wertspezifikation. Repräsentiert die Instanz einen primitiven Typ sind die Slots leer und der eigentliche Wert ist mit der Wertspezifikation beschrieben. Besteht ein Objekt aus mehreren Attributen werden diese in den Slots hinterlegt. Dabei kann ein Attribut wiederum ein Instanzwert sein der aus mehreren Attributen besteht. Der Komparator durchläuft die Instanzen und baut dabei einen beliebig komplexen Vergleich zusammen (siehe Listing 9).

```
if (student.getLastName().equals("Lennon")
    student.getFirstName() == null &&
    student.getDegreeProgram() == DegreeProgram.Mathematics &&
    student.getId() >= 12 && actual.getId() <= 34 &&
    student.getCourse().getLecturer().getName().equals("Dylan") &&
    student.getCourse().getLectureRoom().equals(3010) &&
    student.getThesis().getGrade().equals("2.3") &&
    student.getThesis().getTitle().equals("to be, or not to be")) {
  return true;
} else {
  return false;
}
```

Listing 9: Komparator beim Vergleich von komplexen Datentypen

5.4 Testarbitrierung

Für die Arbitrierung der Testergebnisse wird die Schnittstelle *IArbiter* generiert (siehe Abbildung 23). Die Schnittstelle bietet die aus dem UTP bekannten Methoden zum Setzen und Erfragen des Verdicts. Darüber hinaus werden für die Verwendung mit JUnit die Methoden *getVerdictForAssert* und *setVerdictAfterMatch* benötigt.

Die *getVerdictForAssert*-Methode transformiert ein Verdict-Literal in einen booleschen Wert. Ein *pass* wird als *true* interpretiert und *fail* und *inconclusive* als *false*. Dies ist notwendig um das Verdict-Ergebnis an die JUnit Methode *AssertTrue* zu übergeben, die als Parameter einen booleschen Wert erwartet. Besitzt das Verdict des Arbiters den Wert *none* oder *error*, wird eine *ArbiterException* geworfen die im Testfall abgefangen wird. JUnit beendet dann die Ausführung der aktuellen Testmethode mit einem Fehler (Error).

Mit der Methode *setVerdictAfterMatch* wird das lokale Verdict entsprechend dem Ergebnis eines Komparators, das nur *wahr* oder *falsch* sein kann, auf *pass* oder *fail* gesetzt. Wurde von einem Komparator der Wert *false* ermittelt, kann mit Hilfe einer *ArbiterAbortTestCaseException* die Ausführung des aktuellen Testfalls beendet werden. Das führt dazu, dass im Testfall das Verdict des Arbiters von JUnit ausgewertet wird. So kann die aktuelle Testfallausführung direkt beenden werden, sobald Soll- und Ist-Wert einer Nachricht nicht übereinstimmen.

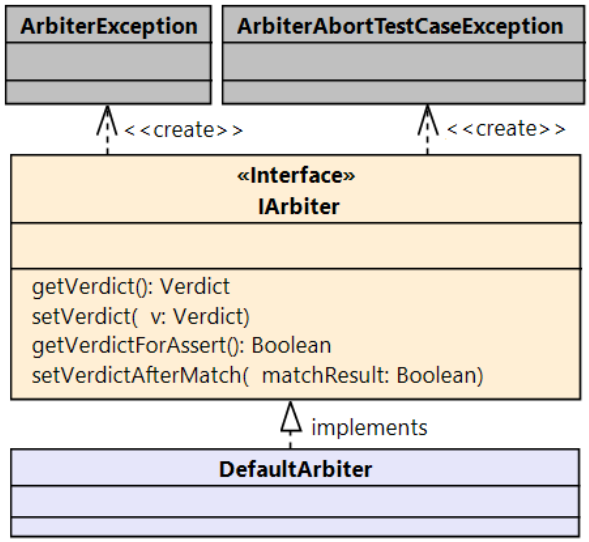

Abbildung 23: Arbiter-Schnittstelle und dessen Standardimplementierung

Mit dem *DefaultArbiter* steht eine Implementierung der Arbiter-Schnittstelle bereit, die bei der Ausführung standardmäßig von den Testfällen verwendet wird. Durch eine eigene Ableitung des Arbiter-Interfaces kann das Verhalten des Arbiters individuell implementiert werden.

5.5 Logisch-technische Abbildung

Die Kommunikation zwischen den Testfällen und dem Testobjekt erfolgt über eine Adapterschicht. Der Adapter hat die Aufgabe die logischen Nachrichten in technische umzuwandeln. Im Adapter werden nur Methodenrümpfe generiert die von den Testfällen aufgerufen werden. Die eigentliche Implementierung muss nach dem Vorliegen der konkreten SUT, bzw. der genauen Schnittstellenbeschreibung, manuell durch Testentwickler erfolgen. Die Adapter-Klasse implementiert die Methoden der Schnittstelle *ISUTAdapter* (siehe Abbildung 24).

Die Methoden *beforeTestCase* und *afterTestCase* werden von der Testumgebung vor, bzw. nach jedem einzelnen Testfall mit dem aktuellen Testfallnamen aufgerufen. Damit besitzt man innerhalb des Adapters Kenntnis über den gerade ausgeführten Testfall. In den Methoden wird jeweils eine *AdapterException* geworfen um zu signalisieren, dass die Methode zurzeit noch nicht implementiert wurde.

Da zur Modellierungszeit meistens noch keine Informationen über die konkreten Schnittstellen der SUT vorliegen, kann es notwendig sein die im Modell verwendeten Objekte an das Typsystem des Testobjektes anzupassen. Dazu müssen die Daten, die von einem Testfall an den Adapter gesendet werden, in eine Form transformiert werden, die der SUT bekannt ist. Dabei kann ein Modellobjekt mit n Attributen in ein SUT-Objekt mit m Attributen umgewandelt werden. Für die Transformation der Rückgabewerte des Testobjekts in das Typsystem

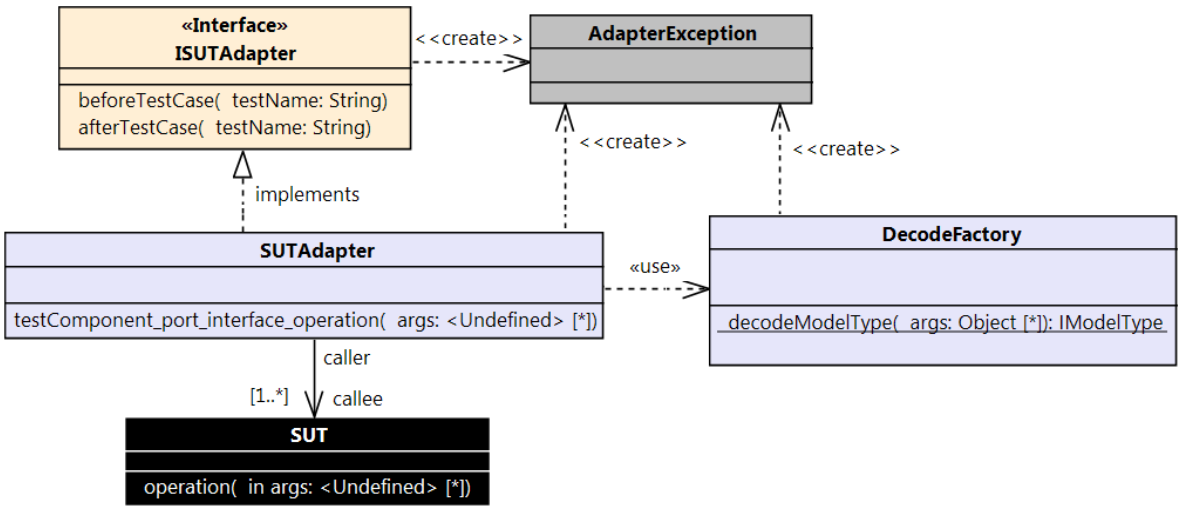

Abbildung 24: Assoziierte Komponenten des Adapters

der Testausführungsumgebung steht die Klasse *DecodeFactory* zur Verfügung. Hier wird für jeden Modelltyp, der von einem Testfall verwendet wird, eine Methode zur Dekodierung bereitgestellt (siehe Listing 10). Als Parameter sind eine beliebige Anzahl von Objekten vom Typ *Object* erlaubt. Die konkrete Implementierung muss, wie im Adapter, manuell erfolgen. Es werden nur die benötigten Methodenrümpfe generiert.

```
public static IStudent decodeStudent(Object... args)
        throws AdapterException {
  // TODO decoding of the Student type
  throw new AdapterException("decodeStudent not yet implemented");
}
```

Listing 10: Dekodierung in den Datentyp *Student*

5.6 Logging

Für die Rückspielung der Testergebnisse müssen diese während der JUnit-Ausführung protokolliert werden. Für die Protokollierung können zwei unterschiedliche Methoden verwendet werden. Zum einen steht die Klasse *Logger* aus dem Paket *java.util.logging* und zum anderen das generierte *logging*-Paket zur Verfügung (siehe Abbildung 25).

Die Klasse *Logger* stellt eine einfache Möglichkeit bereit eine XML-Datei zu erstellen in der Informationen protokolliert werden können (vgl. [Ora13]). Unterschiedliche Meldungen können nach der Wichtigkeit priorisiert werden. So lässt sich z.B. ein unerwarteter Fehler mit einer höheren Priorität loggen als die Initialisierung des Loggers. Der Logger wird innerhalb der Testausführungskomponente verwendet. Vor bzw. nach jedem Testfall werden die Initialisierung und die Finalisierung des aktuellen Arbiter- und Adapter-Objekts protokolliert. Zu Beginn eines Testfalls werden Testfallname und Startzeitpunkt geloggt.

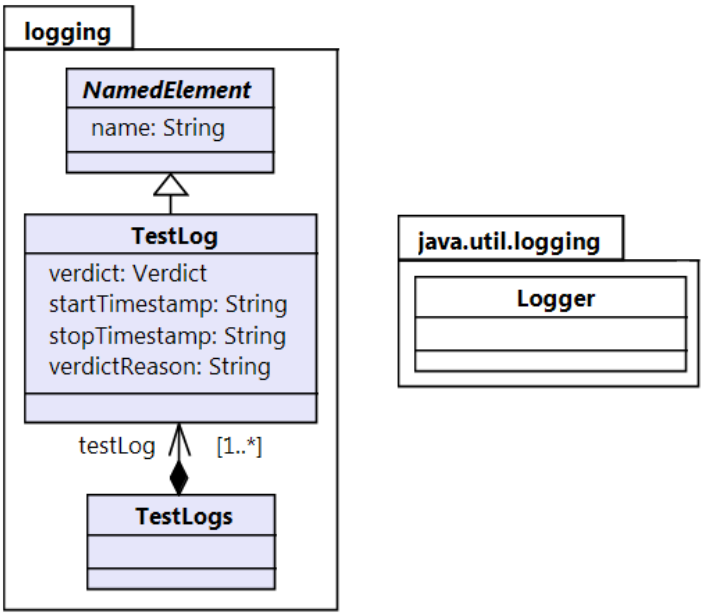

Abbildung 25: Protokollierung der Testausführungsergebnisse

Vor und nach jeder Nachricht werden der Name der Zielmethode und der Portname über den die Kommunikation stattfindet protokolliert. Treten während der Testausführung Ausnahmen auf, werden diese genauso geloggt, wie der Abbruch des aktuellen Testfalls durch den Arbiter.

Für Fokus!MBT existiert ein logisches Testlog-Metamodell, das für den Import von Testausführungsergebnissen verwendet werden kann (vgl. [Ulr12], Kapitel 5.3). Das Logmodell wird bereits für die Rückspielung von TTCN-3-Ausführungsergebnissen der TTworkbench verwendet. Das Testlog-Metamodell besteht aus einem Element *TestLogs* zu dem eine Menge von *TestLog*-Elmenten gehören. Ein *TestLog* repräsentiert genau einen Testfall und beinhaltet das Ausführungsergebnis als Verdict, Start- und Endzeitpunkt der Testausführung und eine Beschreibung, wie das Testergebnis zu Stande gekommen ist (z.B. Fehlermeldungen bei einem Fehlschlag).

Neben diesen Header-Informationen gehören zum Logmodell auch Aktionen, Events und Signaturen. „Aktionen beschreiben Handlungen, die ein Tester in der Entwurfsphase festlegt und die ein spezifisches Verhalten seitens der Testkomponenten dokumentieren“ (vgl. [Ulr12], Seite 30). Mit Events werden Reaktionen der SUT beschrieben. Dazu gehört die beteiligte Testkomponente und die Ports über die die Kommunikation mit der SUT stattfinden. Signaturen spezifizieren die unterschiedlichen Kommunikationsarten z.B. ob eine synchrone oder eine asynchrone Nachricht verwendet wurde. Darüber hinaus werden auch die genutzten Testdaten protokolliert.

Für die Protokollierung der JUnit-Testausführung werden zunächst nur die Elemente *TestLogs* und *TestLog* verwendet (siehe Abbildung 25). Dazu wird vor der Ausführung je-

des Testfalls dessen Name und der Startzeitpunkt protokolliert. Nach jedem Testfall werden der Endzeitpunkt und das finale Verdict des Testfalls geloggt. Tritt bei der Testausführung ein Fehler auf, wird der Grund für diese Ausnahme als *verdictReason* gespeichert. Nach der Ausführung aller Testfälle werden die einzelnen *TestLog*-Elemente in eine XML-Datei geschrieben und gespeichert.

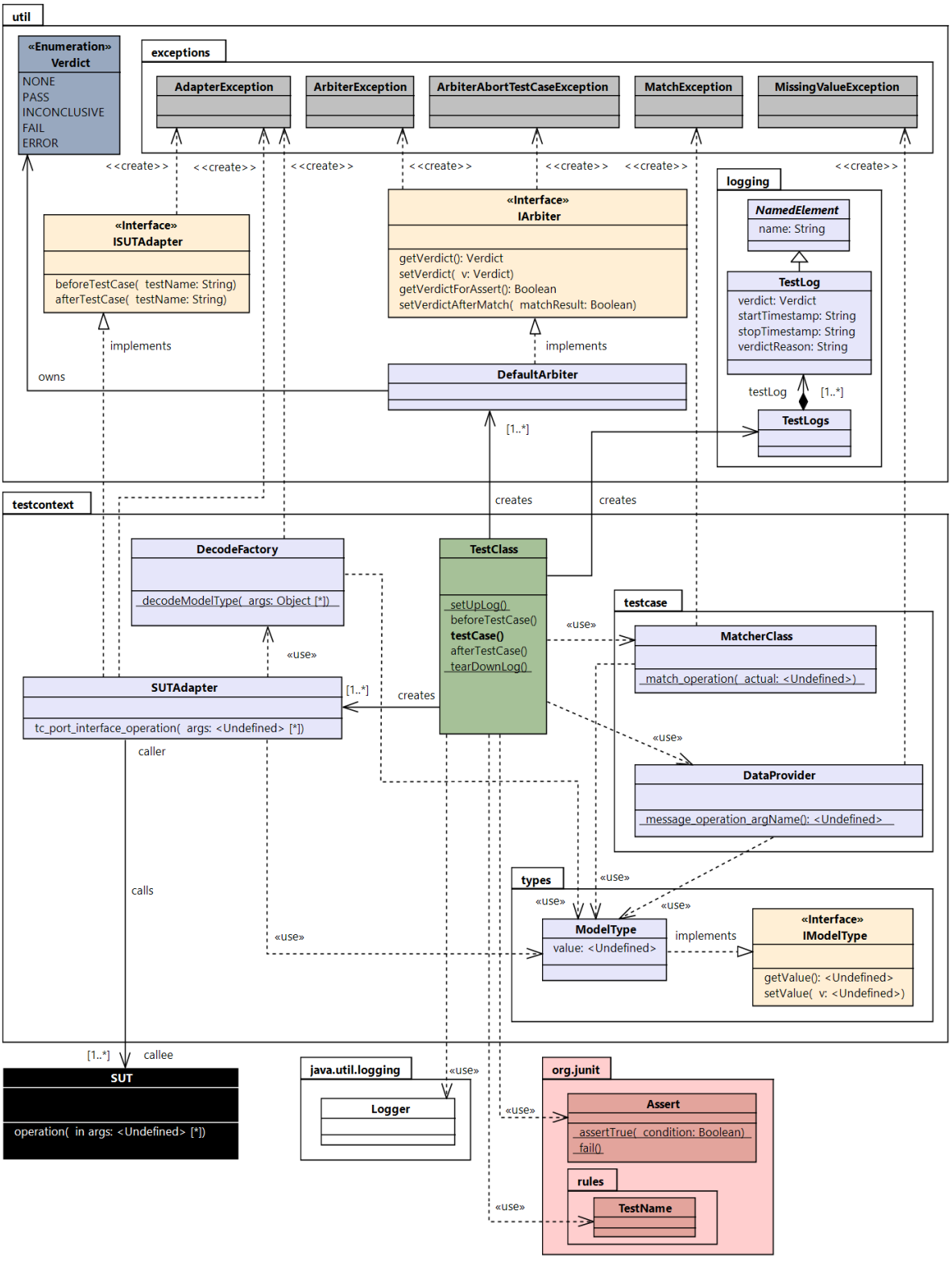

Abbildung 26: Vollständige JUTP-Architektur

6 Implementierung

Der Fokus!MBT-Kern setzt sich aus einer Reihe von Schlüsseltechnologien (u.a. UTP, UML und EMF (*Eclipse Modeling Framework*, vgl. [Ecl13b])) zusammen, mit denen sich testrelevante Dienste implementieren und registrieren lassen, und integriert eclipsebasierte Modellierungsumgebungen wie z.B. Papyrus (vgl. [Ecl13d]).

Der Fokus!MBT-Kern ist mit drei logischen Schichten verbunden. Die *UI Core Extension* Schicht definiert eine Reihe von Service-Schnittstellen, mit denen sich die Oberfläche für unterschiedliche Zwecke und Akteure anpassen lässt. Mit der *Tool Integration* Schicht werden modellierungsspezifische Implementierungen von der Kernkomponente gekapselt. Dadurch kann der Kern von unterschiedlichen Modellierungsumgebungen verwendet werden. Die *Testing Service* Schicht stellt die Verbindung zu testrelevanten Aufgaben her. Zur Testing Service Schicht gehören sechs unterschiedliche Dienste die für die Testautomatisierung benötigt werden (siehe Abbildung 27).

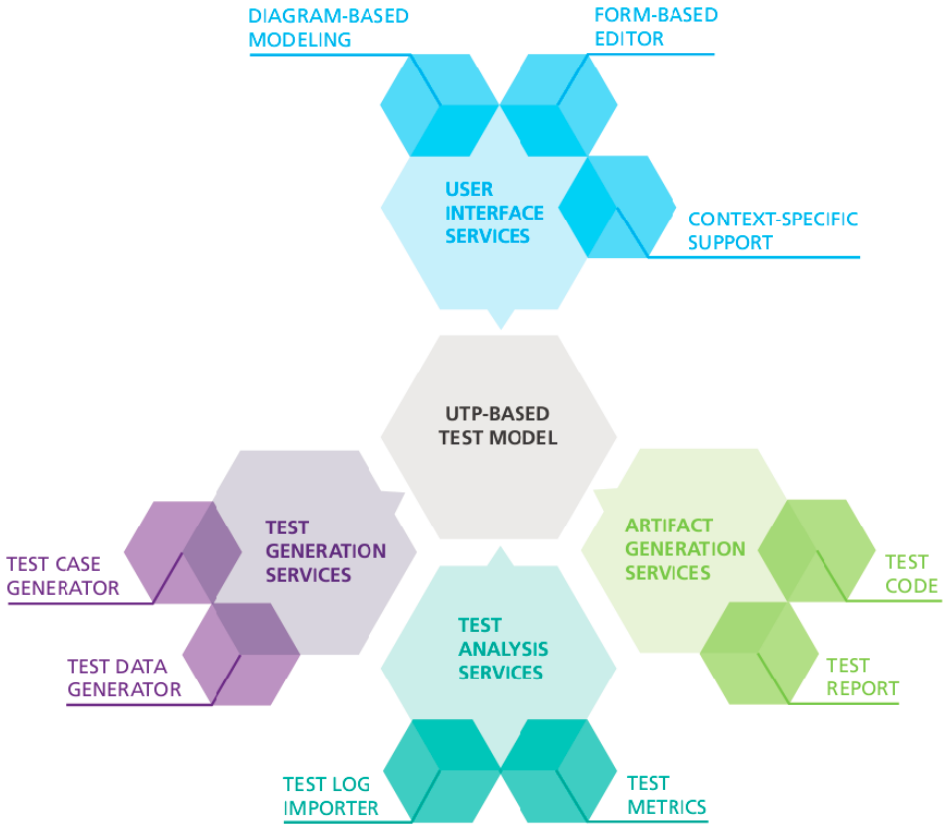

Abbildung 27: Fokus!MBT Testservice-Architektur

Der Testfallgenerator ist ein Service mit dem sich automatisch Testfälle erzeugen lassen, die mit dem Microsofts *Spec Explorer* verwendet werden können. Mit dem Testdatengenerator ist es möglich, aus strukturierten Spezifikationen und Beschränkungen Testdaten zu generieren.

Der *Test Log Importer* ist ein Service zum Wiedereinspielen von Testausführungsergebnissen in das Testmodell, die von einer externen Testausführungsumgebung erstellt wurden. Die zurückgespielten Testlogs werden in Fokus!MBT in das Testmodell integriert. Aus dem Testmodell können dann Testmetriken in Form von Testberichten und Testdokumentation generiert werden.

Generierte oder manuell erstelle Testfälle lassen sich mit dem Testcodegenerator in TTCN-3-Testskripte exportieren (vgl. [WHS13]). Die Testausführungsergebnisse werden Protokolliert und können mit dem Test Log Importer wieder in Fokus!MBT zurückgespielt werden. Ziel dieser Arbeit ist eine Erweiterung des Generators für die Generierung des ausführbaren JUnit-Testscodes und des dazugehörigen Frameworks.

6.1 Acceleo

Die Erzeugung des JUnit-Testcodes und des Testausführungsframeworks aus dem Fokus!MBT-Werkzeug, erfolgt mit dem Codegenerator *Acceleo* (vgl. [Ecl13a]). Acceleo ist ein Eclipse-Plug-in und implementiert die *MOF Model to Text Transformation Language* (MOFM2T) der Object Management Group (vgl. [OMG13a]). Es können beliebige *EMF*-basierte Modelle (z.B. UML, SysML oder eigene Metamodelle) in einen beliebigen Quellcode (z.B. Java, C oder Python) generiert werden. Acceleo bietet darüber hinaus auch einen Editor zur Syntaxhervorhebung, Vervollständigung, Fehlererkennung und Refaktorierung und einen Debugger zum Finden von Fehlerzuständen.

Die Acceleo-Generierung erfolgt standardmäßig innerhalb der Eclipseumgebung. Um den Generator auch außerhalb von Eclipse zu nutzen, generiert Acceleo eine Java-Klasse mit der die Generierung programmatisch gestartet werden kann.

Ein Acceleo-Generator besteht aus einer oder mehreren Modul-Dateien. Ein Modul besitzt einen Namen und die verwendeten EMF *Ecore*-Metamodelle als URI. Es kann von anderen Modulen abgeleitet sein und öffentliche Elemente anderer Module verwenden.

Innerhalb eines Moduls lassen sich Templates definieren, die zur Codegenerierung verwendet werden. Der Typ des Parameters ist ein Typ des verwendeten Metamodells oder ein *OCL*-Typ wie Boolean, String, Integer oder OclAny (vgl. [OMG13b]). Mit einem Template lassen sich außerdem neue Dateien erstellen in denen der generierte Code eingetragen wird. Das sogenannte *Main*-Template ist der Startpunkt für die Generierung. Acceleo erstellt automatisch eine Java-Klasse, die das *Main*-Template aufruft.

Statische Ausdrücke innerhalb eines Templates werden eins-zu-eins in den transformierten Code übernommen. Acceleo-Ausdrücke verwenden Modellelemente um den generierten Text zu berechnen. Die Ausdrücke basieren auf einer Obermenge der OCL-Sprache. Es stehen eine Reihe von Strukturen wie Bedingungen (*if*) und Schleifen (*for*) zur Verfügung und es lassen sich lokale Variablen definieren. Mit *Queries* können komplexe Abfragen an unterschiedlichen Stellen im Generator definiert werden. Das Ergebnis einer Abfrage wird im Cache gespeichert und kann mehrfach aufgerufen werden, ohne dass das Ergebnis mehrmals berechnet werden muss.

Um komplexe Aktionen auszuführen, lassen sich Java-Operationen aufrufen. Die Parameter und Rückgabewerte müssen dabei von einem Typ eines verwendeten Metamodells oder ein primitiver Datentyp wie String, Integer oder Boolean sein. In Abbildung 11 wird innerhalb einer Abfrage mit dem Schlüsselwort *invoke* die Java-Methode *getTestContextName* der Klasse *JUnitHelper* aufgerufen. Als Parameter wird ein *Message*-Objekt übergeben und als Ergebnis eine Zeichenkette erwartet (vgl. [Ecl13a]).

```
[query public getTestContextName(message : Message) : String =
  invoke('de.fraunhofer.fokusmbt.service.junit.JUnitHelper',
         'getTestContextName(org.eclipse.uml2.uml.Message)',
         Sequence{message}) /]
```

Listing 11: Aufruf einer Java-Methode aus Acceleo

6.2 Testcodegenerierung

Grundlage für die Testcodegenerierung, sowohl für TTCN-3 als auch für JUnit, bildet ein EMF *Ecore*-Metamodell (siehe Abbildung 28).

Die Ecore-Klasse *TestCodeGeneratorSettings* besitzt als Attribut das Zielverzeichnis für die generierten Dateien und eine Menge von Testkontexten. Zur *TestContextProxy*-Klasse gehört eine Menge von Testfällen, die durch UML-Operationen repräsentiert werden. Das *Settingsmodell* der TTCN-3- und der JUnit-Generierung sind von der Klasse *TestCodeGeneratorSettings* abgeleitet. Für die Codegenerierung können über das JUnit-Settingsmodell eine Reihe von Einstellungsmöglichkeiten vorgenommen werden.

Mit der Zeichenkette *basePackage*, kann ein Präfix angegeben werden (z.B. *de.fraunhofer.fokus.mbt*), der vor die interne Paketstruktur angehängt wird. Die Zeichenkette wird dann vor alle Paketdefinitionen und Imports in den Java-Klassen gesetzt.

Bei der Modellierung mit der UML stehen für ganze Zahlen der Datentyp *Integer* und für Gleitkommazahlen der Datentyp *Real* zur Verfügung. Ganze Zahlen werden in Java-Integer (32 Bit) und Gleitkommazahlen in Double-Werte (64 Bit) transformiert. Werden für bestimmte Anwendungszwecke Werte mit einer höheren Genauigkeit gefordert, ist die Verwendung von *BigInteger* und *BigDecimal* aus dem *java.math* Paket möglich. Damit lassen sich Zahlen mit einer beliebigen Genauigkeit darstellen (vgl. [Ull11]).

Mit der Option *stopTestCaseAfterFirstFail* kann die Ausführung eines einzelnen Testfalls beendet werden, sobald ein Soll- nicht mit dem entsprechenden Ist-Wert übereinstimmt (vgl. Kapitel 3). Dazu wird eine Ausnahme generiert, die bei einer Abweichung im Komparator von der Standard-Arbiterimplementierung geworfen wird und die aktuelle Testfallausführung unterbricht. JUnit beendet dann den Testfall mit einem Fehlschlag und setzt die Ausführung mit dem nächsten Testfall fort.

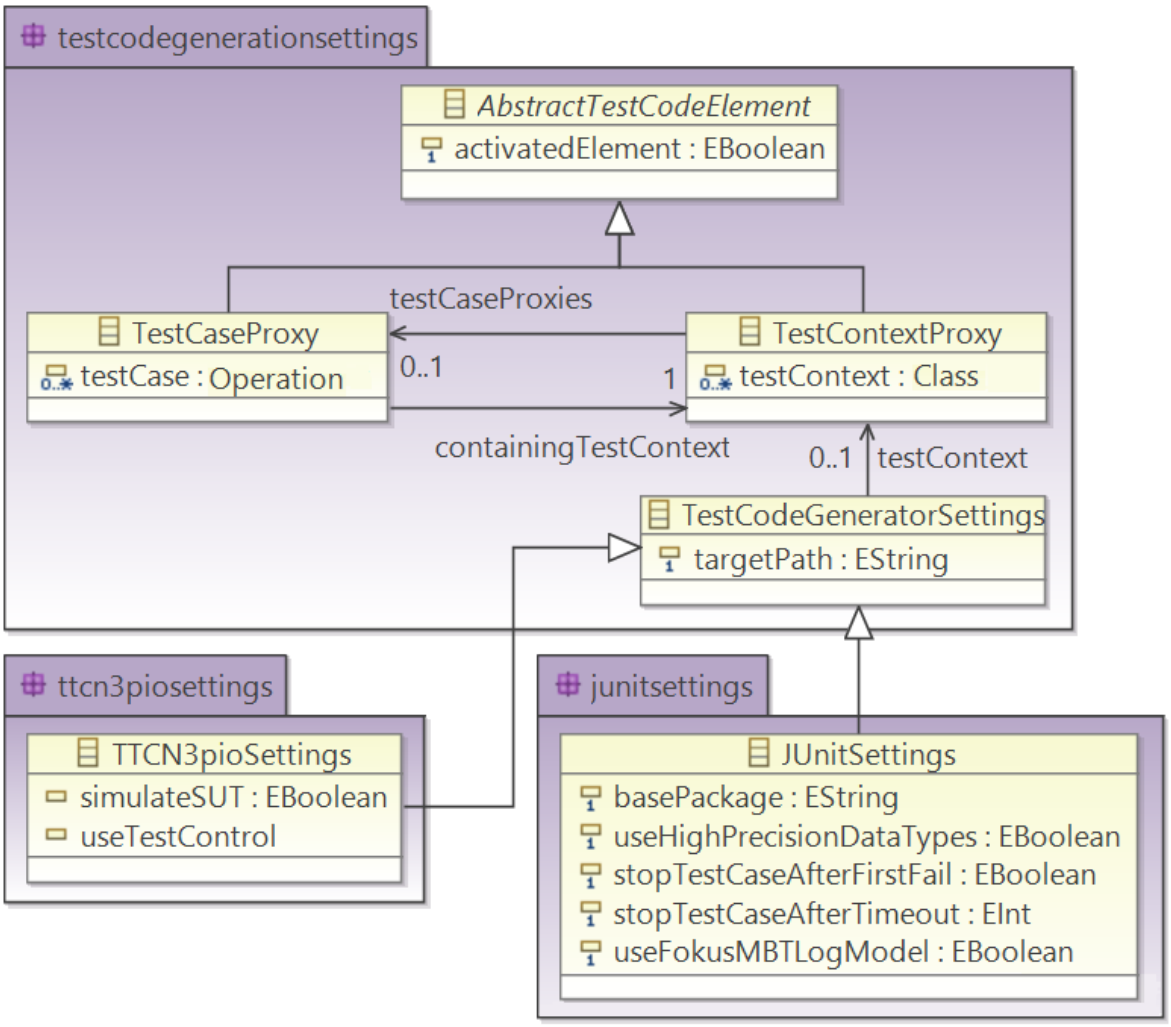

Abbildung 28: Metamodell für die Testcodegenerierung

Um die Ausführungszeit eines Testfalls zu begrenzen, wird diesem eine Zeit in Millisekunden zugewiesen. Innerhalb dieser Zeitspanne muss der Testfall durchlaufen werden. Ist dies nicht der Fall schlägt die Ausführung des aktuellen Testfalls fehl.

Wird *useFokusMBTLogModel* ausgewählt, wird das in Abschnitt 5.6 beschriebene Fokus!MBT-Logmodell für die Protokollierung der Testausführungsergebnisse verwendet. Ansonsten wird dazu der Java eigene *Logger* verwendet.

6.3 Validierung des Codegenerators

Die Validierung des generierten Quelltexts ist mit einem erheblichen Aufwand verbunden. Als Basis der Generierung dienen UTP-Modelle die mit dem Fokus!MBT-Werkzeug modelliert worden sind. Ein Testkontext besteht aus einer Testkomponente, einer SUT und

Ports über die Testkomponente und SUT verbunden sind. Die Schnittstellen zum Testobjekt werden durch UML-Interfaces beschrieben. Diese Interfaces definieren die Operationen der SUT, über die die Kommunikation mit einer Testkomponente stattfindet. Die Operationen können eine Menge von Parametern besitzen, die ebenfalls als Datentyp, Interface oder Klasse modelliert werden müssen.

Des Weiteren ist es notwendig, die zur Kommunikation benötigten Testdaten zu modellieren. Jede Nachricht kann Daten als Parameter und für den erwarteten Rückgabewert benötigen. Dabei können die Testdaten vom Typ einer in Abschnitt 5.3 beschriebenen Wertspezifikation sein.

Die Testfälle werden in Fokus!MBT in Form von Sequenzdiagrammen definiert. Dazu werden die Nachrichten zwischen einer Testkomponente und der SUT bestimmt und die dazu verwendeten Argumente und die erwarten Rückgabewerte den entsprechenden Testdaten zugewiesen. Für die Beschreibung eines komplexen Verhaltens lassen sich mit kombinierten Fragmenten Bedingungen, Alternativen und Schleifen modellieren.

Im Anschluss an die Modellierung erfolgt die Generierung der JUnit-basierten Testausführungsumgebung. Es wird eine Ordnerstruktur mit ausführbaren Java-Klassen erstellt in denen auch die JUnit-Testfälle enthalten sind. Aus einem Beispiel-Testmodell, das aus einem Testobjekt und zehn Testfällen besteht, werden 51 Java-Dateien in 15 Ordnern mit ungefähr 2000 LOC generiert. Davon besteht die Testausführungskomponente, in der die Testfalllogik enthalten ist, alleine aus einer Java-Klasse mit ca. 700 LOC. Im Vergleich dazu wird von dem TTCN-3-Testcodegenerator aus demselben Testmodell eine einzige Datei mit ungefähr 1300 LOC generiert.

Die so generierten Dateien müssen dann mit der Modellierung verglichen werden. Das bedeutet, dass für die Validierung ein enormer Modellierungs-Overhead notwendig ist. Aus diesem Grund ist eine automatische Validierung nicht praktikabel. Dazu müsste eine Reihe von Modellen erstellt, der erwartete Generierungscode implementiert und der aus den Modellen generierte Code mit dem manuell implementierten automatisch verglichen werden. Den Vergleich könnte man mit Hilfe von JUnit-Tests durchführen. Soll- und Ist-Wert müssten dazu Zeichenketten sein, die jeweils den Code der manuell implementierten bzw. der generierten Java-Klassen enthalten. Dabei kann schon ein einzelnes „falsches“ Leerzeichen als Fehlerzustand interpretiert werden, wobei die Semantik durchaus korrekt sein könnte.

Die Validierung des implementierten JUnit-Testcodegenerators konnte somit nur manuell erfolgen. Dazu wurde eine Reihe von unterschiedlichen Testmodellen erstellt, jeweils mit einigen Testfällen. Innerhalb des Testmodells sowie in den Testfällen wurde dabei versucht die gesamte, mögliche Funktionalität zu verwenden. Anhand dieser Modellierung wurde zunächst manuell eine entsprechende Implementierung erstellt:

- Die Testausführungskomponente.
- Der Adapter als Schnittstelle zum Testobjekt.
- Jeweils eine Schnittstelle und eine entsprechende Implementierung für jeden in der

Modellierung verwendete Datentyp.

- Ein *Komparator*- und eine *DatenProvider*-Klasse für jeden Testfall .

Im Anschluss wurde die Generierung durchgeführt und die generierten Klassen mit den zuvor manuell erstellten verglichen. Dabei wurde nur die syntaktische Korrektheit überprüft. Unter anderem erzeugt die Generierung an manchen Stellen aufgrund der Automatisierung eine Reihe von „unschönen“ Leerzeilen, Leerzeichen oder Einrückungen. Außerdem können die aus dem Testmodell verwendeten Bezeichnungen relativ lang werden, was eine manuelle Implementierung aber auch die Validierung erschwert.

7 Fallstudie

In diesem Kapitel wird die praktische Anwendung des generierten JUnit-Testframeworks beispielhaft demonstriert. Als Testobjekt wird eine reale Komponente des Fokus!MBT-Werkzeugs verwendet. Die Fallstudie erfolgt in den folgenden Schritten:

1. Beschreibung der zu testenden Fokus-MBT-Komponente
2. Definition des Testobjekts mit Fokus!MBT
3. Modellierung der verwendeten Testdaten
4. Spezifikation der benötigten Testfälle
5. Generierung des JUnit-Testframeworks und des entsprechenden Testcodes
6. Einbindung des Testcodes in eine externe Testausführungsumgebung
7. Manuelle Implementierung der logisch-technischen Abbildung
8. Ausführung der Testfälle durch JUnit

Für die Modellierung wird in Fokus!MBT der UML-Editor Papyrus (vgl. [Ecl13d]) verwendet. Diagramminformationen werden in einem Notationsmodell, modellspezifische Informationen in einem EMF-Modell und UML-Informationen in einem UML-Modell gespeichert. Beim Anlegen eines neuen UML-Modells werden von Papyrus drei unterschiedliche XMI-Dateien mit den Endungen *di*, *notation* und *uml* erzeugt. Die *di*-Datei beinhaltet Metadaten über die erstellten Diagramme und einen Pfad zum Notationsmodell. In der XMI-Datei mit der Endung *notation* sind Informationen über die Diagrammelemente gespeichert, z.B. Größe, Position, Farbe oder eine Verbindung zu einem Element im UML-Modell. Die *uml*-Datei enthält alle Informationen über die entsprechenden UML Modellelemente (vgl. [Ein11]).

Beim Erstellen eines Modells mit Fokus!MBT werden die drei unterschiedlichen Dateien und außerdem eine weitere *unicorn*-Datei angelegt. In dieser Datei sind Fokus!MBT-interne Konfigurationsinformationen über den Editor gespeichert. Dazu gehören Verbindungen zu den dazugehörigen Papyrus-Dateien und zu dem entsprechenden *Unicorn*-Modell und Angaben über die unterschiedlichen geöffneten Diagramme. Jedes Diagramm ist in Fokus!MBT in einem eigenen Tab geöffnet und zu jedem dieser Tabs sind dessen Typ und ID gespeichert.

Das Unicorn-Modell befindet sich in dem Paket *de.fraunhofer.fokus.unicorn.core.models* (siehe Abbildung 29). Über einen Modellmanager ist der Zugriff auf ein Unicorn-Modell möglich. Die Registrierung eines Unicorn-Modells erfolgt über ein EMF-*ResourceSet*. Ein ResourceSet ist eine Sammlung zusammengehöriger, persistenter Dokumente. Eine einzelne Ressource wird durch die Angabe einer URI erzeugt.

In dieser Fallstudie werden die korrekte Registrierung und Abmeldung eines Unicorn-Modells mit den Operationen *registerResourceSet* und *deregisterResourceSet* getestet. Außerdem wird überprüft ob das richtige Modell geladen wurde. Dazu erfolgt der Zugriff über

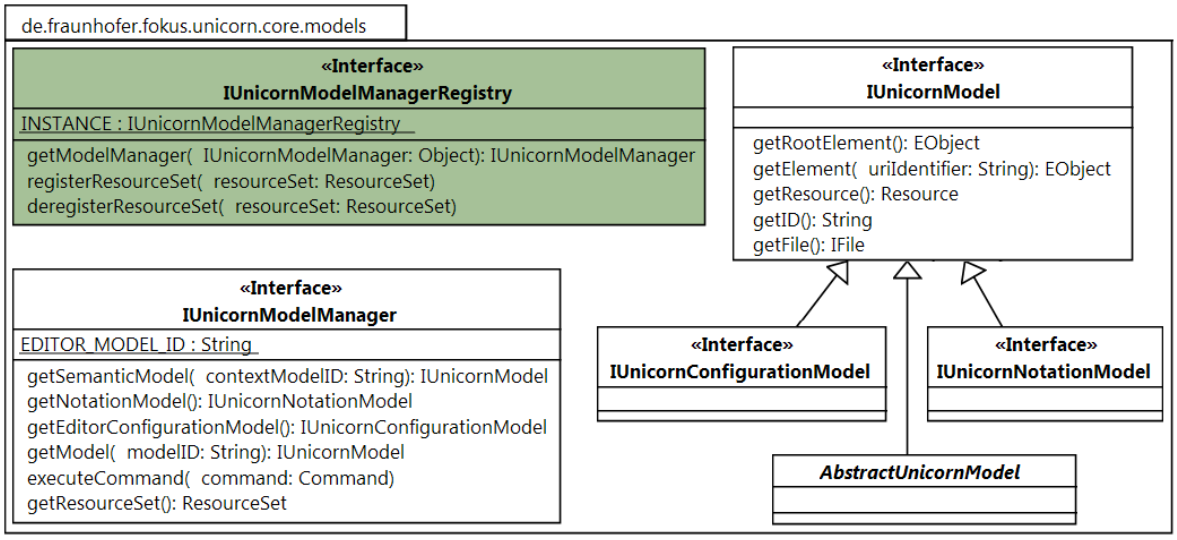

Abbildung 29: Komponenten des Pakets *de.fraunhofer.fokus.unicorn.core.models*

die *getXXXModel*-Methoden des Modellmanagers.

Mit Hilfe des Fokus!MBT-Werkzeugs erfolgt die Modellierung des Testobjekts. Dazu wird als erstes ein neues Fokus!MBT-Modell angelegt, wodurch automatisch die entsprechenden *di-*, *notation-*, *uml-* und *unicorn*-Dateien erstellt werden. Im Anschluss wird das Testobjekt definiert. Dazu werden die öffentlichen Schnittstellen der SUT durch Operationen eines UML-Interface beschrieben. Im Testmodell sind nur solche Informationen nötig, die für den Vergleich des zu testenden Systems relevant sind. Im Modell muss also nicht mit den konkreten Werten und Methodensignaturen gearbeitet werden.

In dieser Fallstudie wird überprüft ob sich ein Fokus!MBT-Modell mit Hilfe der UnicornModelManagerRegistry korrekt registrieren und abmelden lässt. Außerdem soll über den Modellmanager ein Unicorn-Modell geladen, die Dateipfade der *notation-*, *uml-* und *unicorn*-Dateien überprüft und die Anzahl der darin enthaltenen Diagramme und Testfälle ermittelt werden. Für die Modellierung der SUT werden also jeweils eine Methode zum Registrieren und zum Abmelden eins Fokus!MBT-Modells mit einem *Resource*-Objekt benötigt. Für die Verwendung im Testmodell ist es ausreichend eine Ressource mit einer einfachen Enumeration zu beschreiben, in der jedes Literal für eine unterschiedliche Ressource steht. Darüber hinaus wird eine weitere Methode benötigt, mit der überprüft werden kann ob das korrekte Fokus!MBT-Modell registriert bzw. abgemeldet wurde.

Für den Zugriff auf das Unicorn-Modell wird eine Operation definiert, die als Ergebnis ein solches Modell liefert. Dazu wird die benötigte Struktur des Unicorn-Modells wieder selbst als Interface modelliert. Es werden drei String-Attribute für die Dateipfade der *notation-*, *uml-* und *unicorn*-Dateien und zwei Integer-Werte für die Anzahl der enthaltenden Diagramme und Testfälle benötigt. Die resultierenden Schnittstellen und die *Resource*-Enumeration sind in Abbildung 30 dargestellt.

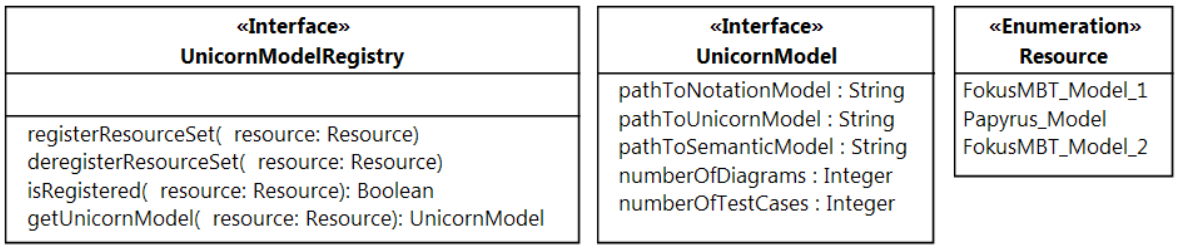

Abbildung 30: Beschreibung der SUT-Schnittstellen und die dazu benötigten Typen

Nach der Modellierung des Testobjektes werden Testfälle und die dazugehörigen Testdaten definiert. Dazu muss zunächst festgelegt werden wie genau die SUT getestet werden soll. Für die Fallstudie werden beispielhaft fünf Szenarien definiert:

1. Registrierung eines Fokus!MBT-Modells und Überprüfung ob dieses Modell korrekt registriert wurde.
2. Registrierung eines Fokus!MBT-Modells, es wird aber überprüft ob ein anderes Fokus!MBT-Modell registriert wurde.
3. Es soll getestet werden ob ein einfaches Papyrus- als Fokus!MBT-Modell registriert werden kann. Es wird erwartet, dass dies nicht funktioniert.
4. Registrierung und Abmeldung eines Fokus!MBT-Modells. Das Modell sollte im Anschluss nicht mehr registriert sein.
5. Registrierung eines Fokus!MBT-Modells und Vergleich der Eigenschaften des geladenen Unicorn-Modells.

Testdaten werden in Fokus!MBT mit Hilfe von Datenpartitionen definiert. Für die Fallstudie werden Werte vom Typ *Boolean*, *Resource* und *UnicornModel* benötigt. Für die booleschen Werte werden zwei Instanzspezifikationen mit *true* und *false* erstellt. Die Ressource ist eine Enumeration mit Literalen für die beiden Fokus!MBT- und das Papyrus-Modelle.

Das Unicorn-Modell wird durch eine Instanzspezifikation mit den entsprechenden Attributen beschrieben (siehe Abbildung 31). Als Ergebnis für die Anzahl der Testfälle wird ein Integer-Wert zwischen 5 und 15 erwartet. Die Anzahl der Diagramme muss genau 16 betragen. Für das semantische Modell wurde der genaue Dateipfad als Zeichenkette definiert. Der Pfad zur *unicorn*-Datei ist mit einem LiteralAny angegeben. Es wird ein beliebiger, nicht-leerer Wert erwartet. Der Pfad zum Notation-Modell ist mit einem regulären Ausdruck beschrieben. Hier werden alle Zeichenketten mit der Endung *.notation* akzeptiert. Die verwendeten Testdaten sind an dieser Stelle nur als Beispiele zu verstehen und wurden nicht nach Testqualitätsmerkmalen sondern nur zu Demonstrationszwecken ausgewählt.

Im Anschluss an die Spezifizierung der Testdaten können die Testfälle modelliert werden. Testfälle werden in Fokus!MBT als Sequenzdiagramme beschrieben. Zum Überprüfen ob eine Ressource korrekt registriert wurde, wird eine synchrone Nachricht mit der Ressource

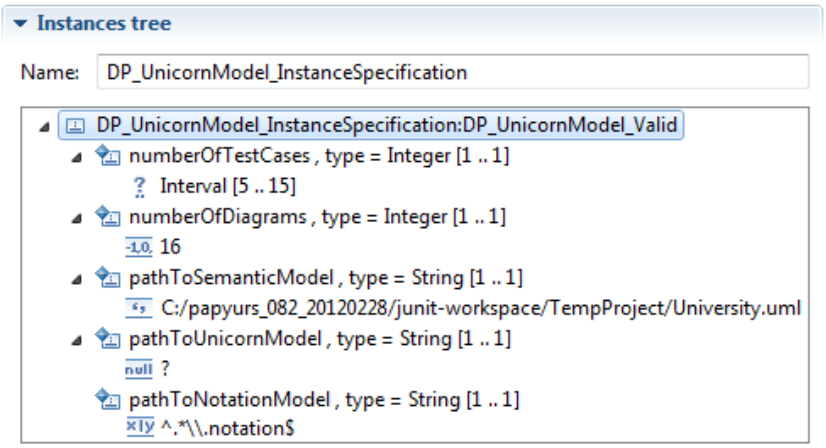

Abbildung 31: Instanzspezifikation des erwarteten Unicorn-Modells

FokusMBT_Model_1 an das Testobjekt gesendet (vgl. Abbildung 32). Da die Methode *registerResourceSet* keinen Rückgabewert besitzt, wird von der SUT mit einer leeren *Reply*-Nachricht geantwortet. Im Anschluss wird über überprüft ob das Modell korrekt registriert wurde. Als Ergebnis wird in diesem Fall der boolesche Wert *true* erwartet. Die konkreten Parameter und Rückgabewerte werden in Fokus!MBT nicht direkt im Modell dargestellt. Es werden die oben beschriebenen Testdaten mit den gesendeten Nachrichten verknüpft.

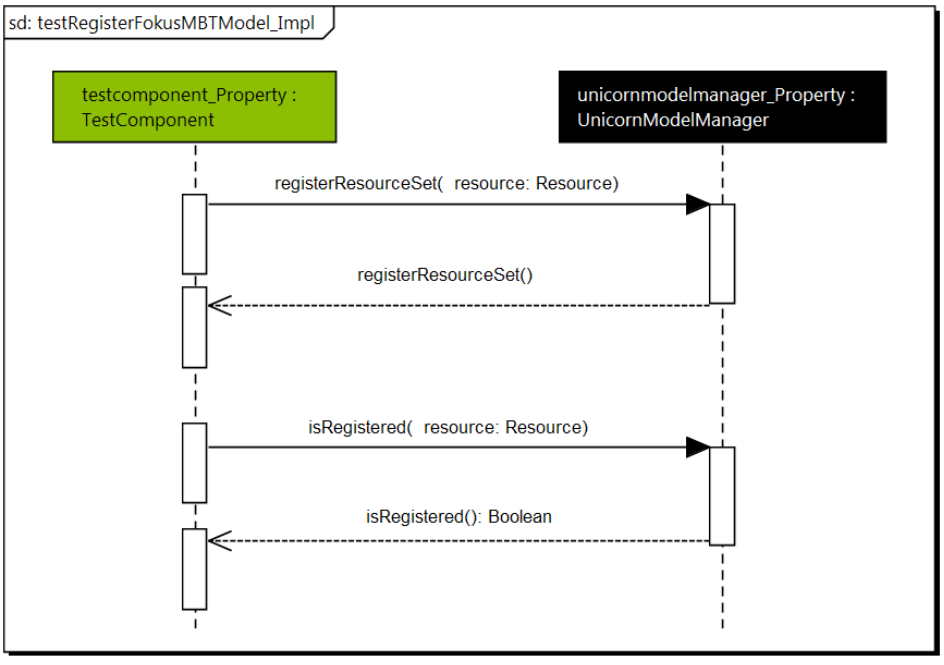

Abbildung 32: Testfall zum Testen der Modellregistrierung

Die Modellierung der Testfälle zwei und drei resultiert im gleichen Modell wie der erste Testfall. Es werden nur unterschiedliche Testdaten verwendet. Einmal wird die Ressource *FokusMBT_Model_1* registriert und auf *FokusMBT_Model_2* überprüft. Im dritten Testfall wird versucht ein Papyrus- statt einem Fokus!MBT-Modell zu registrieren. In beiden Fällen wird als Ergebnis der boolesche Wert *false* erwartet.

Für die Abmeldung wird, genau wie in den ersten drei Testfällen, ein korrektes Fokus!MBT-Modell registriert. Im Anschluss wird dieses Modell wieder abgemeldet und darf nun nicht mehr registriert sein. Es werden somit drei synchrone Nachrichten an die SUT gesendet (siehe Abbildung 33).

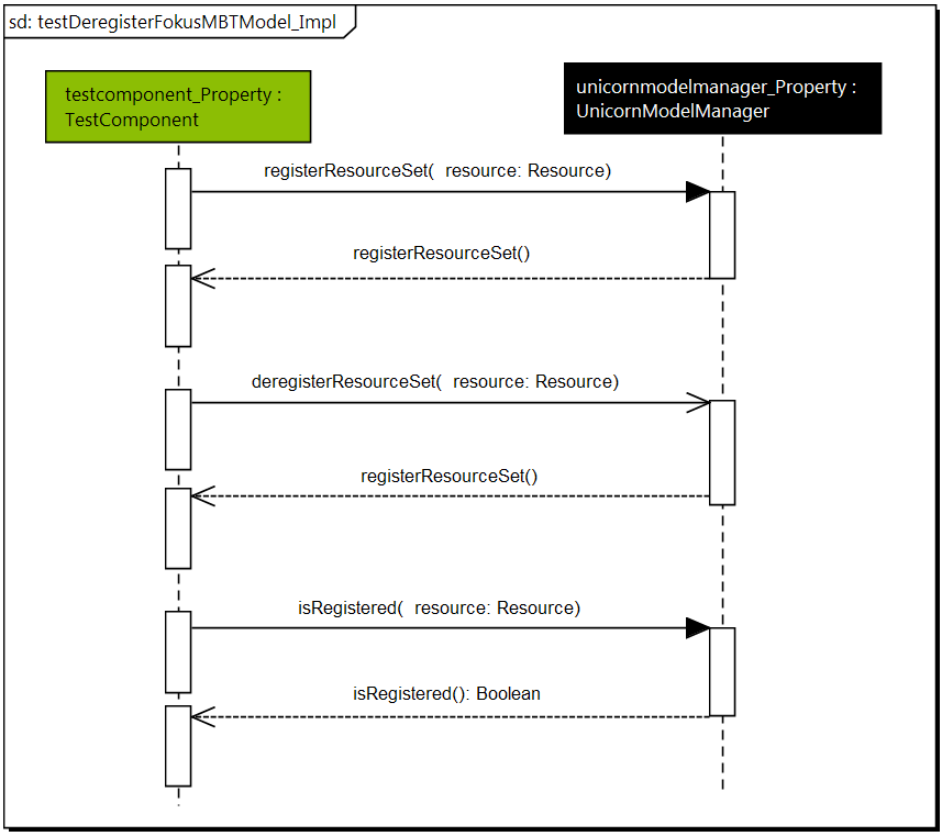

Abbildung 33: Testfall zum Testen der Deregistrierung eines Modells

Der fünfte Testfall soll überprüfen ob das korrekte Modell geladen wurde und ob die entsprechenden Modelldaten den Erwartungen entsprechen. Dazu wird wieder ein Fokus!MBT-Modell registriert und die Unicorn-Modelldaten von diesem ausgelesen (siehe Abbildung 34). Die erwarteten Werte des Unicorn-Modells wurden bereits weiter oben in Abbildung 31 beschrieben.

Nach der Modellierung der Testfälle und den entsprechenden Testdaten, kann das JUnit-Testframework und der entsprechende Testcode aus dem Testmodell generiert werden.

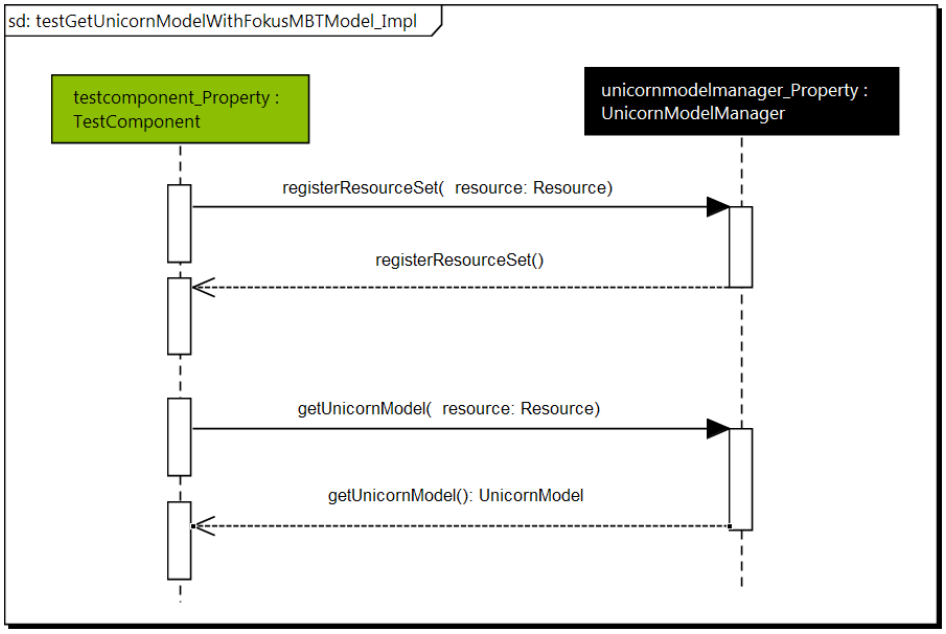

Abbildung 34: Testfall zum Überprüfen der Modelldaten

Dazu können die in Abschnitt 6.2 beschriebenen Einstellungen vorgenommen werden. Für diese Fallstudie wird das Fokus!MBT Log-Modell verwendet und die einzelnen Testfälle werden beendet, sobald eine Nachricht nicht den Erwartungen entspricht.

Es wird eine Ordnerstruktur erstellt in der alle Java-Dateien enthalten sind, die für eine Testausführung benötigt werden (vgl. Abbildung 35). Für den Testkontext werden die Testklasse und der Adapter generiert. Für jeden Testfall existieren eine Datenprovider- und eine Matcher-Klasse. Im Ordner *types* sind alle Datentypen enthalten, die für die Testausführung notwendig sind. Außerdem werden alle Hilfskomponenten, wie die verwendeten Ausnahmen, die Logging-Funktionalität, die Standard-Arbiterimplementierung, die Schnittstellen IArbiter und ISUTAdapter und die Verdict Enumeration, erstellt.

Der generierte Testcode muss nun in eine beliebige Testumgebung eingebunden werden. Beliebig bedeutet, dass Java und JUnit zur Verfügung stehen und dass es möglich ist das Testobjekt anzusprechen. In diesem Fall wird eine Eclipse-Entwicklungsumgebung verwendet in welcher der gesamte Fokus!MBT-Code vorliegt. Die von den Testfällen benötigten Fokus!MBT-Modelle und das Papyrus-Modell werden in der Testausführungsumgebung bereitgestellt. Bereits jetzt können die Testfälle von JUnit ausgeführt werden. Alle Tests schlagen zunächst aber fehl, da die entsprechenden Adapterimplementierungen fehlen.

SUTAdapter und *DecodeFactory* sind die einzigen Klassen in denen eine manuelle Anpassung erfolgen muss. Alle anderen, insbesondere die Testfälle und die Komparatoren, sind voll funktionsfähig. Die Testfälle erwarten für die Ausführung kompatible Werte vom Adapter.

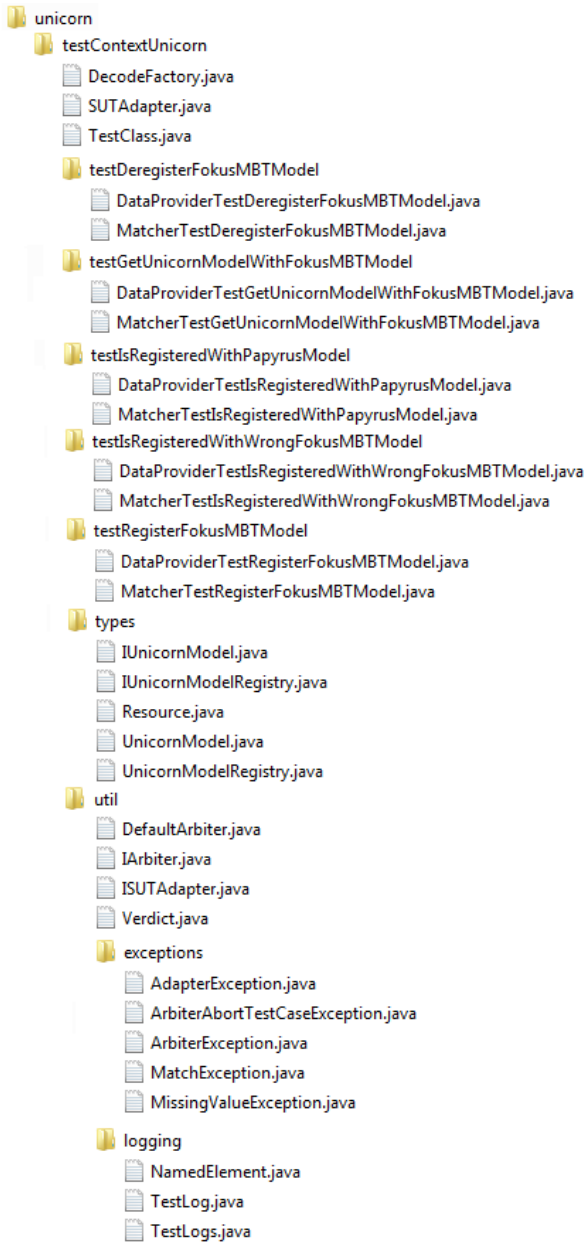

Abbildung 35: Generierte Ordnerstruktur und die darin enthaltenen Java-Dateien

Im Folgenden wird kurz der Aufbau eines generierten Testfalls in Verbindung mit dessen Komparator und der Bereitstellung der Testdaten vorgestellt. Dazu wird der fünfte Testfall betrachtet, der ein Fokus!MBT-Modell registriert und die Eigenschaften des Unicorn-Modells ausliest (vgl. Listing 12).

Durch die *@Test*-Annotation wird die Methode von JUnit als Testfall erkannt. Dauert die gesamte Ausführungszeit des Testfalls länger als zehn Sekunden, wird der Test beendet und schlägt fehl. Die Testfalllogik befindet sich in einem *try*-Block. Treten in der Testumgebung

```
@Test (timeout = 10000)
public void testGetUnicornModelWithFokusMBTModel() {
  try {
    // RegisterResourceSet
    sut.
      testcomponent_Property__tcPort__unicornModelRegistry_
        _registerResourceSet(
          DataProviderTestGetUnicornModelWithFokusMBTModel
            .registerResourceSet__Resource());

    // GetUnicornModel
    UnicornModel sutResult = sut
      .testcomponent_Property__tcPort__unicornModelRegistry_
        _getUnicornModel(
          DataProviderTestGetUnicornModelWithFokusMBTModel
            .getUnicornModel__Resource());

    arbiter.setVerdictAfterMatch(
      MatcherTestGetUnicornModelWithFokusMBTModel
        .match__getUnicornModel(sutResult));

    assertTrue(arbiter.getVerdictForAssert());
  }
  //...
}
```

Listing 12: Ausschnitt aus einem generierten JUnit-Testfalls

oder in der SUT unerwartete Ausnahmen auf, werden diese in der Testmethode aufgefangen und verarbeitet. Der Umgang mit den Ausnahmen wurde in diesem Beispiel ausgeblendet. Die erste gesendet Nachricht ist die Methode *registerResourceSet* (vgl. Abbildung 34). Die Methode wird an den Adapter *sut* weitergeleitet.

Die Methode benötigt als Parameter eine Enumeration vom Typ *Resource*, welche vom Datenprovider bereitgestellt wird (vgl. Listing 13). Es existiert für jeden Testfall eine eigene *DataProvider*-Klasse die für jede Nachricht eine statische Methode besitzt. In diesem Beispiel werden von den beiden im Testfall verwendeten Methoden jeweils ein Literal *FokusMBT_Model_1* der *Resource*-Enumeration benötigt.

Die zweite Methode im Testfall (*getUnicornModel*) liefert als Ergebnis ein Objekt vom Typ *UnicornModel* zurück. Das von der SUT gesendete Ergebnis wird im Komparator mit den erwarteten Werten überprüft (siehe Listing 14). Als Parameter erwartet die Komparator-Methode ein Objekt vom Typ *UnicornModel* und liefert bei Übereinstimmung den Wert *true*, sonst *false*. Tritt bei dem Vergleich ein Fehler auf, wird eine *MatchException* geworfen und im Testfall verarbeitet. Ein Fehler kann z.B. auftreten wenn ein regulärer Ausdruck, der mit der *matches*-Methode überprüft wird, nicht den Java Richtlinien für die Definition von

```
public class DataProviderTestGetUnicornModelWithFokusMBTModel {
  public static Resource registerResourceSet__Resource()
          throws MissingValueException {
    return Resource.FokusMBT_Model_1;
  }

  public static Resource getUnicornModel__Resource()
          throws MissingValueException {
    return Resource.FokusMBT_Model_1;
  }
}
```

Listing 13: Datenprovider für den Testfall *testGetUnicornModelWithFokusMBTModel*

regulären Ausdrücken entspricht.

In dem Komparator ist der Vergleich der in Abbildung 31 definierten Werte zu sehen. Die Anzahl der Testfälle kann ein Integer-Wert von 5 bis 10 sein. Die Testfallanzahl muss genau dem Wert 16 entsprechen. Der Pfad des semantischen Modells ist durch die Zeichenkette exakt angegeben. Der Dateipfad der *unicorn*-Datei kann ein beliebiger, nichtleerer Wert sein. Für das *Notation*-Modell ist ein beliebiger Pfad mit der Endung *.notation*, der als regulärer Ausdruck definiert ist, erlaubt.

```
public static Boolean match__getUnicornModel(UnicornModel actual)
        throws MatchException {
  try {
    if ((actual.getNumberOfTestCases() >= 5
        && actual.getNumberOfTestCases() <= 15)
        && actual.getNumberOfDiagrams().equals(16)
        && actual.getPathToSemanticModel()
            .equals("C:/.../University.uml")
        && actual.getPathToConfigurationModel() != null
        && actual.getPathToNotationModel()
            .matches("^.*\\.notation$")) {
      return true;
    } else {
      return false;
    }
  } catch (Exception Exception) {
    throw new MatchException("MatchException in method ...");
  }
}
```

Listing 14: Komparator für den Testfall *testGetUnicornModelWithFokusMBTModel*

Das Ergebnis des Komparators, entweder *true* oder *false*, wird dem Arbiter des Testfalls übergeben. Der Arbiter ermittelt dann am Ende der Ausführung das globale Verdict des Testfalls. Nach der letzten Nachricht einer Testfallmethode wird das globale Verdict an die JUnit-Methode *assertTrue* übergeben. JUnit wertet das Ergebnis dann als Fehlschlag oder Erfolg aus.

Der Adapter ist dafür zuständig die Daten die in den Testfällen verwendet werden an das konkrete Typsystem der SUT anzupassen. In diesem Fall muss die *Resource*-Enumeration in ein *org.eclipse.emf.ecore.resource.ResourceSet* umgewandelt werden (vgl. Listing 16). Dazu wird das in dem Verzeichnis *testmodelsunicorn* bereitgestellte *Unicorn*-Modell mit dem Namen *University* geladen. Ist das Modell ein gültiges FokusMBT-Modell wird das entsprechende *ResourceSet* ausgelesen. Mit dieser Ressource wird dann die tatsächliche *registerResourceSet* der *UnicornModelManagerRegistry* aufgerufen.

```
public void testcomponent_Property__tcPort_
  _unicornModelRegistry__registerResourceSet(
    Ressource ressource) throws AdapterException, Exception {
  if (ressource == Ressource.FokusMBT_Model_1) {
    model = JUnitTestingHelper
      .loadUnicornModelFromDevelopmentWorkspace(
        "University",
        "testmodelsunicorn",
        FokusMBTKernelHelper.Constants.UNICORN_MODELS__FOKUSMBT,
        InstanceTransformation.class);
  }

  if (model instanceof IFokusMBTModel) {
    fokusMBTModel = (IFokusMBTModel) model;

    set = fokusMBTModel.getResource().getResourceSet();

    IUnicornModelManagerRegistry.INSTANCE.registerResourceSet(set);
  }
}
```

Listing 15: Adapter für die Methode *registerResourceSet*

In der Adapter-Methode *registerResourceSet* muss ebenfalls eine Anpassung vorgenommen werden (vgl. Listing 16). Im Testfall wird ein Objekt vom Typ *UnicornModel* (siehe Abbildung 30) erwartet. Die dazu benötigten Eigenschaften müssen aus unterschiedlichen Stellen im Modell zusammengetragen werden. Über den Modellmanager der *UnicornModelManagerRegistry* hat man Zugriff auf die Dateipfade der *notation-*, *uml-* und *unicorn*-Dateien. Im Unicorn-Modell sind alle Diagramme des Testmodells enthalten. Ein Testfall ist ein Sequenzdiagramm mit der ID *fokusmbt.diagramView.testCase.sequence*. Es werden alle Tabs durchlaufen die im Unicorn-Modell enthalten sind und dabei wird die Anzahl der Diagramme und der Testfälle ermittelt. Die einzelnen Werte werden der

Methode *decodeUnicornModel* übergeben, die aus diesen ein *UnicornModel*-Objekt erzeugt und zurückliefert.

```
public UnicornModel testcomponent_Property__tcPort_
  _unicornModelRegistry__getUnicornModel(
    Ressource ressource) throws AdapterException, Exception {

  String semanticModelFile = IUnicornModelManagerRegistry.INSTANCE
    .getModelManager(null)
    .getSemanticModel("fokusmbt.model.semantic").getFile()
    .getLocation().toString();

  String notationModelFile = IUnicornModelManagerRegistry.INSTANCE
    .getModelManager(null).getNotationModel().getFile()
    .getLocation().toString();

  String unicornModelFile = IUnicornModelManagerRegistry.INSTANCE
    .getModelManager(null).getEditorConfigurationModel().getFile()
    .getLocation().toString();

  int numberOfDiagrams = 0;
  int numberOfTestCases = 0;

  for (EObject tab : IUnicornModelManagerRegistry.INSTANCE
    .getModelManager(null).getEditorConfigurationModel()
    .getRootElement().getUnicorn().getEditorTabConfiguration()
    .getTabConfiguration()) {
      if (tab instanceof DiagramTab) {
        numberOfDiagrams++;

        if (((DiagramTab) tab).getDiagramViewID().equals(
          "fokusmbt.diagramView.testCase.sequence")) {
            numberOfTestCases++;
        }
      }
  }

  return DecodeFactory.decodeUnicornModel(semanticModelFile,
                notationModelFile, unicornModelFile,
                numberOfDiagrams, numberOfTestCases);
}
```

Listing 16: Adapter für die Methode *registerResourceSet*

In der Klasse *decodeFactory* ist für jeden Modell-Datentyp eine Methode enthalten, die eine Menge von beliebigen Parametern enthält und aus diesen ein bestimmtes Modell-Objekt zu-

sammensetzt und zurückliefert. Die Methode *decodeUnicornModel* (vgl. Listing 17) erstellt ein *UnicornModel* und übergibt dieses zurück an den Adapter. Zunächst wird dazu überprüft, ob die einzelnen Objekte dem erwarteten Typ entsprechen. Ist dies der Fall werden die Eigenschaften im *UnicornModel* gesetzt. Entsprechen die Typen der übergebenen Parameter nicht den Erwartungen, wird eine *AdapterException* geworfen, die im Testfall abgefangen und verarbeitet wird.

```
public static UnicornModel decodeUnicornModel(Object... args)
        throws AdapterException {
  UnicornModel result = new UnicornModel();

 if (args[0] instanceof String && args[1] instanceof String
       && args[2] instanceof String && args[3] instanceof Integer
       && args[4] instanceof Integer) {
    result.setPathToSemanticModel((String) args[0]);
    result.setPathToNotationModel((String) args[1]);
    result.setPathToConfigurationModel((String) args[2]);
    result.setNumberOfDiagrams((Integer) args[3]);
    result.setNumberOfTestCases((Integer) args[4]);

    return result;
  }

  throw new AdapterException("type mismatch");
}
```

Listing 17: Dekodierungsmethode für den Typ *UnicornModel*

Im Anschluss an die Implementierung des Adapters und der Anpassung der Datentypen können die Testfälle von JUnit ausgeführt werden. In dieser Fallstudie wird die Testfallklasse in der Eclipseumgebung als JUnit Plug-in Test gestartet. Eclipse erzeugt dazu alle benötigten Komponenten, die für die Testfallausführung notwendig sind. JUnit durchläuft die Testfälle und wertet diese aus. In diesem Fall sind alle Testfälle erfolgreich (siehe Abbildung 36).

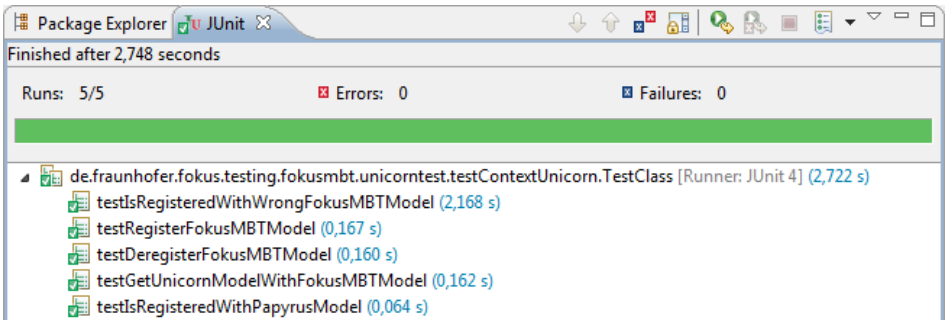

Abbildung 36: Testausführungsergebnisse von JUnit

Die Testausführungskomponente protokolliert die Ergebnisse der einzelnen Testfälle und speichert diese in einer XML-Datei (siehe Abbildung 37). Jeder Testfall entspricht einem *TestLog*-Element in dem der Testname, das Verdict-Ergebnis, der Start- und Endzeitpunkt und der Grund für das Verdict eingetragen sind. *verdictReason* ist in diesem Fall leer, da bei der Testausführung keine Ausnahmen aufgetreten sind. Die XML-Datei kann im Anschluss wieder von Fokus!MBT importiert und in das Testmodell integriert werden. Über den Testfallname kann dann das jeweilige Testausführungsergebnis einer Testanforderung zugeordnet werden.

```
<?xml version="1.0" encoding="UTF-8"?>
- <TestLogs>
  - <TestLog>
      <name>testIsRegisteredWithWrongFokusMBTModel</name>
      <verdict>PASS</verdict>
      <startTimestamp>1385191791669</startTimestamp>
      <stopTimestamp>1385191793856</stopTimestamp>
      <verdictReason/>
    </TestLog>
  - <TestLog>
      <name>testRegisterFokusMBTModel</name>
      <verdict>PASS</verdict>
      <startTimestamp>1385191793857</startTimestamp>
      <stopTimestamp>1385191794024</stopTimestamp>
      <verdictReason/>
    </TestLog>
  - <TestLog>
      <name>testDeregisterFokusMBTModel</name>
      <verdict>PASS</verdict>
      <startTimestamp>1385191794024</startTimestamp>
      <stopTimestamp>1385191794184</stopTimestamp>
      <verdictReason/>
    </TestLog>
  - <TestLog>
      <name>testGetUnicornModelWithFokusMBTModel</name>
      <verdict>PASS</verdict>
      <startTimestamp>1385191794184</startTimestamp>
      <stopTimestamp>1385191794346</stopTimestamp>
      <verdictReason/>
    </TestLog>
  - <TestLog>
      <name>testIsRegisteredWithPapyrusModel</name>
      <verdict>PASS</verdict>
      <startTimestamp>1385191794347</startTimestamp>
      <stopTimestamp>1385191794351</stopTimestamp>
      <verdictReason/>
    </TestLog>
</TestLogs>
```

Abbildung 37: Protokollierte Testausführungsergebnisse als XML-Struktur

8 Fazit

Das Ziel dieser Arbeit bestand aus der Implementierung eines Generators, der aus UTP-Modellen ein testobjektspezifisches Testausführungsframework und den entsprechenden JUnit-Testcode erzeugt.

Der Generator ist in das Fokus!MBT-Werkzeug eingebunden und nutzt für die Generierung dessen Testmodelle, die mit Hilfe von UML und UTP beschrieben werden. Ein Testmodell enthält alle testrelevanten Informationen die zur Codegenerierung notwendig sind. Der Generator erzeugt aus einem Testmodell Tests in einem hohen Abstraktionsniveau. Dadurch lassen sich Testentwurfs- und Testrealisierungsaktivitäten soweit vorbereiten, dass für die Testausführung lediglich die konkreten Schnittstellen zur SUT implementiert werden müssen sobald diese zur Verfügung stehen.

Aufgrund der Trennung von logischen und technischen Informationen, erfolgt die Kommunikationen zwischen den Testfällen und der SUT über eine Adapterschicht. Im Adapter werden Modellinformationen konkretisiert, an die SUT gesendet und deren Antworten zur Verwendung in den Testfällen wieder abstrahiert. Der Adapter ist testobjektspezifisch, was bedeutet, dass für jede Schnittstelle zum Testobjekt eine eigene Adaptermethode generiert wird. Durch die Anpassung an die SUT wird die Komplexität des Adapters reduziert und die Implementierung ist einfacher durchzuführen.

Da JUnit eine Reihe von testbezogenen Konzepten fehlen, wird zusätzlich zum eigentlichen Testcode ein Testausführungsframework erzeugt, das diese Nachteile kompensiert. Die vollständige Testfalllogik, in Form von JUnit-Testmethoden, befindet sich einzig in der Testausführungskomponente. Der Vergleich von Ist- und Sollverhalten wird mit Hilfe von Komparatoren durchgeführt. Die Komparatoren werden entsprechend den modellierten, erwarteten Ergebniswerten generiert. So sind diese in der Lage auch Objekte mit Intervallen, regulären Ausdrücken, komplexen Datentypen und Wildcards zu vergleichen. Durch die Verwendung des bereitgestellten Arbiters, oder einer individuellen Arbiterimplementierung, kann das Testausführungsergebnis je nach Bedarf bestimmt werden.

Darüber hinaus kann das Testframework die Testfälle auch unabhängig von JUnit auswerten, da alle dazu benötigten Informationen im Framework enthalten sind. Dadurch ist eine Erweiterbarkeit des Codegenerators auf andere Java-basierte Testframeworks (z.B. auf TestNG) ohne großen Aufwand möglich.

Werden für bestimmte Anwendungsfälle hoch präzise Testdaten benötigt, ist es möglich, statt den einfachen Java-Datentypen *Integer* und *Double*, die Klassen *BigInteger* und *BigDecimal* aus dem *java.math-Paket* zu verwenden. So können beliebig genaue Werte im Testmodell definiert und im Generierungscode verwendet werden.

Die Ergebnisse einer Testausführung werden konform zum Fokus!MBT Log-Modell protokolliert und als XML-Datei gespeichert. So können die Testausführungsergebnisse in Fokus!MBT zurückgespielt und weiter ausgewertet werden. Bei Bedarf kann für die Protokollierung auch der simple *Logger* aus dem *java.util.logging*-Paket verwendet werden.

8.1 Ausblick

Zur Zeit werden von JUTP lediglich die Header-Informationen des Fokus!MBT Testlog-Metamodells unterstützt. Dazu wird für den Testfall dessen Name, der Start- und Endzeitpunkt der Ausführung, das Testergebnis als Verdict und eine Begründung für das Verdict-Ergebnis protokolliert. Das Logmodell unterstützt aber noch eine Reihe von weiteren nützlichen Informationen, wie die verwendeten Testdaten oder Angaben zu den Nachrichten, die bei der Kommunikation mit der SUT verwendet wurden. Durch die Erweiterung von JUTP um die vollständige Testlog-Funktionalität, wäre im Anschluss an die Fokus!MBT-Rückspielung, eine umfangreiche Analyse der Testausführungsergebnisse möglich.

In einem weiteren Schritt müsste der Fokus!MBT *Test Log Importer* erweitert werden, um die Testausführungsergebnisse von JUTP zurückspielen zu können. Dazu muss die erstellte XML-Logdatei mit Fokus!MBT eingelesen werden. Um sowohl den Import der jetzigen Log-Dateien, in denen lediglich die Header-Informationen protokolliert sind, als auch zukünftig das vollständige Testlog-Metamodell zu unterstützen, könnte die Validität der Log-Dateien mit Hilfe eines XML-Schemas überprüft werden. Eine valide XML-Logdatei wird dann als EMF Ressource geladen und dessen Testausführungsinformationen in das Testmodell für eine weitere Auswertung integriert.

Das Testframework enthält alle Informationen um sowohl das erwartete SUT-Verhalten zu überprüfen, also auch das daraus resultierende Verdict-Testergebnis zu bestimmen. Daher ist auch eine Verwendung von alternativen Java-Testframeworks wie TestNG zusätzlich zu JUnit denkbar. Aufgrund der syntaktischen Ähnlichkeit zwischen Java und C#, könnte der Generator auch ohne großen Aufwand auf die Generierung eines C#-basierten Testframeworks und entsprechenden NUnit-Testcode erweitert werden.

Der generierte Testcode muss momentan von einer externen Testumgebung ausgeführt werden. Da Fokus!MBT auf der Eclipse-IDE basiert, könnte der generierte Testcode direkt in der Fokus!MBT-Umgebung von JUnit zur Ausführung gebracht werden. Dazu ist allerdings erforderlich, dass das Testobjekt in Fokus!MBT eingebunden werden kann. Dadurch wäre die Verwendung einer externen Testumgebung hinfällig und die Zeit für die Testaktivitäten könnte reduziert werden.

Wie in Abschnitt 4.2 beschrieben, müssen eine Reihe von Einschränkungen bei der Modellierung der Testfälle eingehalten werden, damit eine korrekte Generierung des JUnit-Testcodes durchführbar ist. Dadurch ist es unter anderem nicht möglich die UTP-Zeitkonzepte zu verwenden. Mit Hilfe der Zeitkonzepte kann für eine Menge von Nachrichten eine Zeitspanne angegeben werden, die bestimmt wie lange deren Ausführung maximal dauern darf. In JUnit besteht lediglich die Möglichkeit eine Zeitspanne für den gesamten Testfall festzulegen. Es wäre denkbar JUnit zu erweitern, um auch einzelne oder eine Menge von Nachrichten mit einem Timeout zu versehen. In Listing 18 ist eine mögliche Erweiterung der assertTrue-Methode angegeben.

Es wird angenommen, dass die Bedingung *condition* wahr ist. Falls die Ausführung der Methode länger als die angegebene Zeit *time* in Millisekunden dauert oder die Bedingung falsch

```
public static void assertTrue(long time, boolean condition)
```

Listing 18: Mögliche Erweiterung der JUnit assertTrue-Methode

ist, wird ein *AssertionError* geworfen. Für die Verwendung müssten die Stimulation der SUT und der Vergleich des erwarteten Resultates, der als Ergebnis einen Booleschen-Wert liefert, direkt aus der *assertTrue*-Methode erfolgen. Die Zeitangabe könnte auch in den anderen *assert*-Methoden erweitert werden. Außerdem wäre es denkbar *Assert*-Gruppen zu bilden, für die eine gemeinsame, maximale Ausführungszeit angeben werden kann. So wäre man in der Lage die UTP-Zeitkonzepte, zumindest bei der Verwendung von Nachrichten, einzusetzen.

Glossar

Anforderung (requirement)
„Eine vom Benutzer benötigte Eigenschaft oder Fähigkeit, die eine Software erfüllen oder besitzen muss, um einen Vertrag, einen Standard, eine Spezifikation oder ein anderes formales Dokument zu erfüllen“ ([IG13], Seite 46).

Black-Box-Test (black box testing)
Funktionales oder nicht-funktionales Testen ohne Nutzung von Informationen über die interne Struktur des Systems oder einer Komponente (vgl. [IG13], Seite 6).

Debugging (debugging)
„Tätigkeit des Lokalisierens/Identifizierens, Analysierens und Entfernens der Ursachen von Fehlerwirkungen in der Software“ ([IG13], Seite 16).

Fehlerwirkung (failure)
„Abweichung einer Komponente/eines Systems von der erwarteten Lieferung, Leistung oder dem Ergebnis“ ([IG13], Seite 24).

Fehlerzustand/Defekt (fault/defect)
Fehlerhafter Zustand eines Teilprogramms, welcher die geforderte Funktion beeinträchtigt und zu einer Fehlerwirkung führt bzw. führen kann (vgl. [IG13], Seite 17).

Fehlhandlung (error)
„Die menschliche Handlung, die zu einem falschen Ergebnis führt“ ([IG13], Seite 21).

Formale und semi-formale Modelle (formal and semi-formal models)
Formale Modelle sind sehr Abstrakt, leicht zu verstehen und besitzen eine präzise Syntax und Semantik, z.B. Graphen. Dagegen sind semi-formale Modelle weniger abstrakt, lassen sich aber besser implementieren. Sie besitzen zwar eine weitgehend, präzise Syntax aber die Semantik ist meist unpräzise. Ein bekanntes Beispiel ist die semi-formale UML (vgl. [Wer08], Seite 25).

Funktionales Testen (functional testing)
„Testen, das auf der Analyse der funktionalen Spezifikation einer Komponente oder eines Systems basiert“ ([IG13], Seite 26).

Klassifikationsbaum (classification tree)
„Ein Baum, der Äquivalenzklassen hierarchisch gliedert, und der in der gleichnamigen Verfahren zum Entwurf von Testfällen genutzt wird“ ([IG13], Seite 9).

Klassifikationsbaumverfahren (classification tree method)
„Ein Black-Box-Testentwurfsverfahren, bei dem die durch einen Klassifikationsbaums dargestellten Testfälle so entworfen werden, dass Kombinationen der Repräsentanten von Eingabe- und/oder Ausgabebereichen (Äquivalenzklassen) ausgeführt werden“ ([IG13], Seite 9).

Komponententest (component testing)
„Testen einer (einzelnen) Komponente“ ([IG13], Seite 10).

Modellbasiertes Testen (model-based testing)
„Testen, das auf einem Modell der zu testenden Komponente oder des zu testenden Systems basiert, z.B. Zuverlässigkeitswachstumsmodelle, Nutzungsmodelle wie Nutzungsprofile oder Verhaltensmodelle wie Entscheidungstabellen oder Zustandsübergangsdiagramme“ ([IG13], Seite 35). „umbrella of approaches that generate tests from models“ ([ETS11], Seite 6). „An umbrella of techniques that use (semi-)formal models as engineering artifacts in order to specify and/or generate test-relevant artifacts, such as test cases, test scripts, reports, etc“ ([OMG13f], Seite 5).

Nicht-funktionaler Test (non-functional testing)
„Testen der Eigenschaften eines System, die nicht direkt mit der Funktionalität in Verbindung stehen, z.B. Zuverlässigkeit, Effizienz, Benutzbarkeit, Änderbarkeit und Übertragbarkeit“ ([IG13], Seite 43).

Qualität (quality)
„Der Grad, in dem ein System, eine Komponente oder ein Prozess die Kundenerwartungen und -bedürfnisse erfüllt“. „Der Grad, in dem ein Satz inhärenter Merkmale Anforderungen erfüllt“ ([IG13], Seite 43).

Qualitätsmerkmal (quality attribute)
„Fähigkeit oder Eigenschaft, welche die Qualität einer Einheit beeinflusst“. „Ein Satz von Eigenschaften eines Softwareprodukts, anhand dessen seine Qualität beschrieben und beurteilt wird. Ein Softwarequalitätsmerkmal kann über mehrere Stufen in Teilmerkmale verfeinert werden. Qualitätsmerkmale sind Funktionalität, Zuverlässigkeit, Benutzbarkeit, Effizienz, Änderbarkeit und Übertragbarkeit“ ([IG13], Seite 44).

Qualitätssicherung (quality assurance)
„Teil des Qualitätsmanagements, das darauf gerichtet ist, Vertrauen in die Erfüllung der Qualitätsanforderungen zu erzeugen“ ([IG13], Seite 43).

Retest (re-testing)
Auch Fehlernachtest. „Die Wiederholung aller Testfälle, die vor der Fehlerkorrektur eine Fehlerwirkung erzeugt haben. Sie dient der Überprüfung, ob die Korrektur des ursächlichen Fehlerzustands erfolgreich war“ ([IG13], Seite 10).

Platzhalter (stub)
„Eine rudimentäre oder spezielle Implementierung einer Softwarekomponente, die verwendet wird, um eine noch nicht implementierte Komponente zu ersetzen bzw. zu simulieren“ ([IG13], Seite 55).

Systemmodell (system model)
Ein Systemmodell stellt eine interne Sicht auf das System, seine Komponenten, Schnittstellen und Datentypen bereit. Es wird beschrieben, wie das System aufgebaut ist und spezifiziert was eine tatsächliche Umsetzung erfüllen muss. (vgl. [Wen13], Seite 11).

Test (test)
„Eine Menge von einem oder mehreren Testfällen“ ([IG13], Seite 56).

Testablaufspezifikation (test prodedure specification)
Auch Testskript oder Testdrehbuch. „Bezeichnet ein Dokument, das eine Folge von Schritten zur Testausführung festlegt“ ([IG13], Seite 62).

Testanforderung (test requirement)
Auch Testbedingung (test requirement) oder Testsituation (test situation). „Eine Einheit oder ein Ereignis, z.B. eine Funktion, eine Transaktion, ein Feature, ein Qualitätsmerkmal oder ein strukturelles Element einer Komponente oder eines Systems, welche bzw. welches durch einen oder mehrere Testfälle verifiziert werden kann“ ([IG13], Seite 58).

Testart (test type)
„Eine Gruppe von Testaktivitäten, mit der Absicht, eine Komponente oder ein System auf einige zusammenhängende Qualitätsmerkmale zu prüfen. Eine Testart ist auf ein bestimmtes Testziel fokussiert, wie z.B. Zuverlässigkeitstest, Regressionstest, Benutzbarkeitstest. Die Testart kann sich auch auf eine oder mehrere Teststufen oder -phasen beziehen“ ([IG13], Seite 64).

Testbasis (test basis)
„Alle Dokumente, aus denen die Anforderungen ersichtlich werden, die an eine Komponente oder ein System gestellt werden, bzw. die Dokumentation, auf der die Herleitung oder Auswahl der Testfälle beruht“ ([IG13], Seite 57).

Testbasismodell (test basis model)
Konzeptuelle Modelle die eine Testbasis beschreiben ([Pro13], Seite 11).

Testdirektive (test directive)
„Eine Testdirektive steuert oder beschreibt die Ableitung (manuell oder automatisiert) eines Systemmodells in ein oder mehrere Testkontexte des Testmodells“ ([WSH10], Seite 2).

Testen (testing)
„Der Prozess, der aus allen Aktivitäten des Lebenszyklus besteht (sowohl statisch als auch dynamisch), die sich mit der Planung, Vorbereitung und Bewertung eines Softwareprodukts und dazugehöriger Arbeitsergebnisse befassen. Ziel des Prozesses ist sicherzustellen, dass diese allen festgelegten Anforderungen genügen, dass sie ihren Zweck erfüllen, und etwaige Fehlerzustände zu finden“ ([IG13], Seite 65).

Testentwurf (test design)
„Siehe auch Testentwurfsspezifikation. Der Vorgang, allgemeine Testziele in handfeste Testbedingungen und Testfälle zu überführen“ ([IG13], Seite 59).

Testentwurfsspezifikation (test design specification)
Ëin Ergebnisdokument, das die Testbedingungen für ein Testobjekt, die detaillierte Testvorgehensweise und die zugeordneten abstrakten Testfälle identifiziert. Siehe auch Testspezifikation.“ ([IG13], Seite 59).

Testfall (test case)
„Umfasst folgende Angaben: die für die Ausführung notwendigen Vorbedingungen, die Menge der Eingabewerte (ein Eingabewert je Parameter des Testobjekts), die Menge der vorausgesagten Ergebnisse, sowie die erwarteten Nachbedingungen. Testfälle werden entwickelt im Hinblick auf ein bestimmtes Ziel bzw. auf eine Testbedingung, wie z.B. einen bestimmten Programmpfad auszuführen oder die Übereinstimmung mit spezifischen Anforderungen zu prüfen (wie Eingaben an das Testobjekt zu übergeben und Sollwerte abzulesen sind)“ ([IG13], Seite 57).

Testfallspezifikation (test case specification)
„Ein Dokument, das eine Menge von Testfällen für ein Testobjekt spezifiziert (inkl. Testdaten und Vor-/Nachbedingung), bei dem die Testfälle jeweils Ziele, Eingaben, Testaktionen, vorausgesagte Ergebnisse und Vorbedingungen für die Ausführung enthalten ([IG13], Seite 57).

Testkonzept (test plan)
„Ein Dokument, das u.a. die Vorgehensweise, die Ressourcen und die Zeitplanung der beabsichtigten Tests mit allen Aktivitäten beschreibt“. Es identifiziert die Testobjekte und beschreibt die Testumgebung ([IG13], Seite 62).

Testmodell (test model)
Ein Testmodell beschreibt, wie ein System zu testen ist. Es werden interne Aspekte vernachlässigt und die Priorität liegt auf dem extern zu beobachteten Verhalten. Ein Testmodell kann für die Testfallgenerierung verwendet werden und es können auch Artefakte des System- oder von anderen Modellen verwendet werden (vgl. [Wen13], Seite 11).

Testobjekt (test object)
Auch SUT (system under test). „Die Komponente oder das System, welches getestet wird“ ([IG13], Seite 61).

Testphase (test phase)
„Eine abgegrenzte Menge von Testaktivitäten, die einer Projektphase zugeordnet sind, z.B. Ausführungsaktivitäten einer Teststufe“ ([IG13], Seite 62).

Testplanung (test planning)
„Eine Aktivität im Testprozess zur Erstellung und Fortschreibung des Testkonzepts“ ([IG13], Seite 62).

Testrahmen (test harness)
„Eine Testumgebung, die aus den für die Testausführung benötigten Treibern und Platzhaltern besteht“ ([IG13], Seite 60).

Testprozess (test process)
„Der fundamentale Testprozess umfasst die folgenden Aktivitäten: Planung und Steuerung, Analyse und Design, Realisierung und Durchführung, Bewertung und Berichterstattung sowie den Abschluss der Testaktivitäten“ ([IG13], Seite 63).

Testskript (test script)
„Bezeichnet üblicherweise eine Testablaufspezifikation, insbesondere eine automatisierte“ ([IG13], Seite 63).

Testspezifikation (test specification)
„Ein Dokument, das aus der Testentwurfspezifikation, der Testfallspezifikation und/oder der Testablaufspezifikation besteht“ ([IG13], Seite 64).

Teststeuerung (test control)
„Als Teststeuerung bezeichnet man die Managementaufgabe zur Entwicklung und Anwendung von Korrekturmaßnahmen, um in einem Testprojekt eine Abweichung vom geplanten Vorgehen zu beherrschen“ ([IG13], Seite 58).

Teststrategie (test strategy)
„Abstrakte Beschreibung der vorgesehenen Teststufen und der Art und Weise, wie innerhalb dieser Teststufen vorzugehen ist, für eine Organisation oder ein Programm - gültig für ein oder mehrere Projekte“ ([IG13], Seite 64).

Teststufe (test level)
„Eine Teststufe ist eine Gruppe von Testaktivitäten, die gemeinsam ausgeführt und verwaltet werden. Teststufen sind mit Zuständigkeiten in einem Projekt verknüpft. Beispiele für Teststufen sind der Komponententest, der Integrationstest, der Systemtest und der Abnahmetest“ ([IG13], Seite 61).

Testsuite (test suite)
„Die Zusammenstellung (Aggregation) mehrerer Testfälle für den Test einer Komponente oder eines Systems, bei der Nachbedingungen des einen Tests als Vorbedingungen des folgenden Tests genutzt werden können“ ([IG13], Seite 64).

Testumgebung (test environment)
„Eine Umgebung, die benötigt wird, um Tests auszuführen. Sie umfasst Hardware, Instrumentierung, Simulatoren, Softwarewerkzeuge und andere unterstützende Hilfsmittel“ ([IG13], Seite 59).

Testziel (test objective)
„Ein Grund oder Zweck für den Entwurf und die Ausführung von Tests“ ([IG13], Seite 61).

Treiber (driver)
„Ein Testwerkzeug, das eine zu testende Komponente/ein System aufruft und/oder steuert“ ([IG13], Seite 20).

Überdeckungsgrad (coverage)
„Der Grad, ausgedrückt in Prozent, zu dem ein spezifiziertes Überdeckungselement (z.B. Zweig) durch eine Testsuite ausgeführt wurde.“ ([IG13], Seite 14).

Unittest-Framework (unit test framework)
„Ein Werkzeug, das eine Umgebung für einen Komponententest bereitstellt. In dieser Umgebung wird die Komponente isoliert oder mit geeigneten Treibern und Platzhaltern getestet. Darüber hinaus wird dem Entwickler zusätzliche Unterstützung (z.B. Debugging) zur Verfügung gestellt“ ([IG13], Seite 67).

Validierung (validation)
„Bestätigung durch Bereitstellung eines objektiven Nachweises, dass die Anforderungen für einen spezifischen beabsichtigten Gebrauch oder eine spezifische beabsichtigte Anwendung erfüllt worden sind“ ([IG13], Seite 68).

Verifizierung (verification)
„Bestätigung durch Bereitstellung eines objektiven Nachweises, dass festgelegte Anforderungen erfüllt worden sind“ ([IG13], Seite 68).

White-Box-Test (white box testing)
„Ein Test, der auf der Analyse der internen Struktur einer Komponente oder eines Systems basiert“ ([IG13], Seite 69).

Literatur

[BCW12] BRAMBILLA, M., J. CABOT und M. WIMMER: *Model-Driven Software Engineering in Practice*. Synthesis Lectures on Software Engineering. Morgan & Claypool Publishers, 2012.

[Bec98] BECK, KENT: *Kent Beck's Guide to Better Smalltalk*. Cambridge University Press, New York, NY, USA, 1998.

[BGSC13] BECK, KENT, ERICH GAMMA, DAVID STAFF und MIKE CLARK: *JUnit. A programmer-oriented testing framework for Java.*, 2013. http://junit.org/. Letzer Aufruf: Dezember 2013.

[BHK04] BORN, M., E. HOLZ und O. KATH: *Softwareentwicklung mit UML 2: Die neuen Entwurfstechniken UML 2, MOF 2 und MDA*. Programmer's choice. Addison Wesley, 2004.

[BS08] BEUST, CÉDRIC und HANI SULEIMAN: *Next generation Java testing : TestNG and advanced concepts*. Addison-Wesley, 2008.

[Cou13] COUNTERSOFT: *Gemini*, 2013. http://www.countersoft.com/solutions/project-tracking/. Letzer Aufruf: Dezember 2013.

[Dai04] DAI, ZHEN RU: *Model-Driven Testing with UML 2.0*. Technischer Bericht, Computing Laboratory, University of Kent, 2004.

[Ecl13a] ECLIPSE: *Acceleo - transforming models into code*, 2013. http://www.eclipse.org/acceleo/. Letzer Aufruf: Dezember 2013.

[Ecl13b] ECLIPSE: *Eclipse Modeling Framework Project (EMF)*, 2013. http://www.eclipse.org/modeling/emf/. Letzer Aufruf: Dezember 2013.

[Ecl13c] ECLIPSE: *Eclipse Test & Performance Tools Platform Project (TPTP)*, 2013. http://www.eclipse.org/tptp/. Letzer Aufruf: Dezember 2013.

[Ecl13d] ECLIPSE: *Papyrus*, 2013. http://www.eclipse.org/papyrus/. Letzer Aufruf: Dezember 2013.

[Ein11] EINARSSON, HAFSTEINN ÞÓR: *Refactoring UML Diagrams and Models with Model-to-Model Transformations*. Magister Scientiarum, Faculty of Industrial Engineering, Mechanical Engineering and Computer Science, University of Iceland, 2011.

[ETS11] ETSI: *European Telecommunications Standards Institute ES 202 951 v 1.1.1: Methods for Testing and Specification (MTS); Model-Based Testing (MBT); Requirements for Modelling Notations*, 2011.

[Fak13] FAKHROUTDINOV, KIRILL: *The Unified Modeling Language (UML)*, 2013. http://www.uml-diagrams.org/. Letzer Aufruf: Dezember 2013.

[Fis13] FISCHE, STEFAN: *Vorlesung „Programmieren“: Java Grundlagen*, 2013. Vorlesung „Programmieren“, Institut für Telematik, Universität zu Lübeck. http://media.itm.uni-luebeck.de/teaching/ws2013/prog/Programmieren-06-Java%20Grundlagen.pdf. Letzer Aufruf: Dezember 2013.

[Fra13] FRAUNHOFER: *ModelBus*, 2013. Fraunhofer-Institut für Offene Kommunikationssysteme, Kompetenzzentrum MOTION. http://www.modelbus.org/modelbus/. Letzer Aufruf: Dezember 2013.

[GVV08] GRAHAM, D. und E. VAN VEENENDAAL: *Foundations of Software Testing: ISTQB Certification*. Cengage Learning, 2008.

[Hof08] HOFFMANN, DIRK W.: *Software-Qualität*. Springer, 2008.

[HP13] HEWLETT-PACKARD: *HP Quality Center-Software*, 2013. http://www8.hp.com/de/de/software-solutions/software.html?compURI=1172141#.UnzZP_kz2So. Letzer Aufruf: Dezember 2013.

[IBM13] IBM: *Rational Quality Manager*, 2013. http://www-03.ibm.com/software/products/de/de/ratiqualmana/. Letzer Aufruf: Dezember 2013.

[IG13] ISTQB / GTB, WORKING PARTY GLOSSARY: *ISTQB/GTB Standardglossar der Testbegriffe*, April 2013. http://www.german-testing-board.info/service/information/glossar.html. Letzer Aufruf: Dezember 2013.

[Kar12] KARAMAN, MEHMET EMIN: *Realisierung von Testmustern mit dem UML Testing Profile im Kontext modellbasierten Testens*. Bachelorarbeit, Beuth Hochschule für Technik Berlin, 2012.

[KK09] KOORD, Y. und V. KRAUTER: *WinfWiki, Überblick Vorgehensmodelle im Projektmanagement*, 2009. http://winfwiki.wi-fom.de/index.php/%C3%9Cberblick_Vorgehensmodelle_im_Projektmanagement. Fachhochschule für Oekonomie und Management (FOM) Hamburg, Letzer Aufruf: Dezember 2013.

[MM03] MILLER, JOAQUIN und JISHNU MUKERJI: *MDA Guide*, 2003. http://www.omg.org/mda/mda_files/MDA_Guide_Version1-0.pdf. Letzer Aufruf: Dezember 2013.

[OMG13a] OMG: *MOF Model To Text Transformation Language (MOFM2T)*, 2013. http://www.omg.org/spec/MOFM2T/1.0/. Letzer Aufruf: Dezember 2013.

[OMG13b] OMG: *Object Constraint Language (OCL)*, 2013. http://www.omg.org/spec/OCL/. Letzer Aufruf: Dezember 2013.

[OMG13c] OMG: *Object Management Group (OMG)*, 2013. http://www.omg.org/. Letzer Aufruf: Dezember 2013.

[OMG13d] OMG: *OMG Model Driven Architecture (MDA)*, 2013. http://www.omg.org/mda/. Letzer Aufruf: Dezember 2013.

[OMG13e] OMG: *Systems Modeling Language (SysML), Version 1.3*, 2013. http://www.omg.org/spec/SysML/1.3/. Letzer Aufruf: Dezember 2013.

[OMG13f] OMG: *UML Testing Profile (UTP), Version 1.2*, 2013. http://www.omg.org/spec/UTP/1.2/. Letzer Aufruf: Dezember 2013.

[OMG13g] OMG: *Unified Modeling Language (UML), Version 2.4.1*, 2013. http://www.omg.org/spec/UML/2.4.1/. Letzer Aufruf: Dezember 2013.

[Ora13] ORACLE: *java.util.logging.Logger*, 2013. http://docs.oracle.com/javase/6/docs/api/java/util/logging/Logger.html. Letzer Aufruf: Dezember 2013.

[Pro13] PROJECT, MIDAS: *MIDAS Glossary - Terms, Definitions, and Acronyms*, 2013. http://www.midas-project.eu/wp-content/uploads/2013/08/MIDAS_Glossary_v1.1.pdf. Letzer Aufruf: Dezember 2013.

[SBB12] SEIDL, R., M. BAUMGARTNER und T. BUCSICS: *Basiswissen Testautomatisierung: Konzepte, Methoden und Techniken*. dpunkt, 2012.

[Sch08] SCHIEFERDECKER, INA: *Testautomatisierung leicht gemacht*. OBJEKTspektrum, 2008.

[Sch11] SCHIEFERDECKER, INA: *Basiswissen Softwaretest Certified Tester*, 2011. Vorlesung Grundlagen des Softwaretestens, Freie Universität Berlin.

[Sch12] SCHIEFERDECKER, INA: *Model-Based Testing*. IEEE Software, 29(1):14–18, 2012.

[SDGR03] SCHIEFERDECKER, I., Z. R. DAI, J. GRABOWSKI und A. RENNOCH: *The UML 2.0 testing profile and its relation to TTCN-3*. In: *Proceedings of the 15th IFIP international conference on Testing of communicating systems*, TestCom'03, Seiten 79–94, Berlin, Heidelberg, 2003. Springer-Verlag.

[SMJU10] SARMA, M., P. V. R. MURTHY, S. JELL und A. ULRICH: *Model-based testing in industry: a case study with two MBT tools*. In: *Proceedings of the 5th Workshop on Automation of Software Test*, AST '10, Seiten 87–90, New York, NY, USA, 2010. ACM.

[Spi02] SPILLNER, ANDREAS: *Das W-Modell: Testen als paralleler Prozess zum Software-Entwicklungsprozess*. Technischer Bericht, Hochschule Bremen, 2002.

[Sta73] STACHOWIAK, HERBERT: *Allgemeine Modelltheorie*. Springer, 1973.

[SVEH07] STAHL, T., M. VÖLTER, S. EFFTINGE und A. HAASE: *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. Programmer's choice. dpunkt, Heidelberg, 2 Auflage, 2007.

[Sys13] SYSTEMTECHNIK, BERNER & MATTNER: *CTE XL*, 2013. http://www.cte-xl-professional.com/de/cte-xl-professional/index.html. Letzer Aufruf: Dezember 2013.

[Tea13] TEAMTEST: *TestLink*, 2013. http://sourceforge.net/projects/testlink/. Letzer Aufruf: Dezember 2013.

[Tec13a] TECHNOLOGIES, TESTING: *TTworkbench - The Reliable Test Automation Platform*, 2013. http://www.testingtech.com/products/ttworkbench.php. Letzer Aufruf: Dezember 2013.

[Tec13b] TECHNOLOGY, VERIFYSOFT: *Conformiq Tool Suite*, 2013. http://www.verifysoft.com/de_conformiq_automatic_test_generation.html. Letzer Aufruf: Dezember 2013.

[Tes13] TESTNG: *TestNG*, 2013. http://testng.org/doc/index.html. Letzer Aufruf: Dezember 2013.

[Tho13] THOUGHTWORKS: *Selenium*, 2013. http://docs.seleniumhq.org/. Letzer Aufruf: Dezember 2013.

[UL10] UTTING, M. und B. LEGEARD: *Practical Model-Based Testing: A Tools Approach*. Elsevier Science, 2010.

[Ull11] ULLENBOOM, CHRISTIAN: *Java ist auch eine Insel - Das umfassende Handbuch*. Programmer's choice. Galileo Computing, 9. Auflage, 2011.

[Ulr12] ULRICH, STEVEN: *Formalisierung von Testausführungsergebnissen in Konzepte des UML Testing Profils*. Bachelorthesis, Beuth Hochschule für Technik Berlin, 2012.

[UPL06] UTTING, M., A. PRETSCHNER und B. LEGEARD: *A taxonomy of model-based testing*. Technischer Bericht, University of Waikato, Hamilton, Neuseeland, April 2006.

[Wen13] WENDLAND, MARC-FLORIAN: *Fokus!MBT – The Test Modeling Environment – Technical and Methodological Insights*, 2013. Fraunhofer Institute for Open Communication Systems.

[Wer08] WERNER, MATTHIAS: *Grundlagen Modelle*, 2008. Vorlesung Analyse und Modellierung von Betriebssystemen, Technische Universität Chemnitz, http://osg.informatik.tu-chemnitz.de/lehre/old/ws0809/am/am-02-Model-a4.pdf. Letzer Aufruf: Dezember 2013.

[Wes05] WESTPHAL, FRANK: *Testgetriebene Entwicklung mit JUnit & FIT: Wie Software änderbar bleibt*. dpunkt, 2005.

[WHS13] WENDLAND, MARC-FLORIAN, ANDREAS HOFFMANN und INA SCHIEFERDECKER: *Fokus!MBT - a multi-paradigmatic test modeling environment*. In: *Proceedings of the workshop on ACadeMics Tooling with Eclipse*, ACME '13, Seiten 3:1–3:10, 2013.

[WSH10] WENDLAND, MARC-FLORIAN, INA SCHIEFERDECKER und ANDREAS HOFFMANN: *Modellbasiertes Testen mit Fokus!MBT*. OBJEKTspektrum, Seiten 1–6, 2010.

[Zec03] ZECH, JENS: *TTCN-3 - Testing and Test Control Notation*. Seminararbeit, Technische Universität Berlin, 2003.