Bachelorarbeit

Automatische Generierung von Testskripten aus Palladio-Modellen

von
Christian Fischer
6512858
Universität Paderborn
Weremboldstraße 5, 46325 Borken
fchristi@mail.upb.de

vorgelegt bei

Erstkorrektor: Prof. Dr. Gregor Engels

Betreuer: Frank Brüseke

Abgabe: 28. Februar 2013

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit selbständig und ohne unerlaubte fremde Hilfe angefertigt, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die den benutzten Quellen und Hilfsmitteln wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Paderborn, 28. Februar 2013

_____

(Vorname Name)

Inhaltsverzeichnis

Abbildungsverzeichnis

Liste der Algorithmen

Tabellenverzeichnis

1 Einleitung

1.1 Problembeschreibung und Motivation

Komponentenbasierte Software-Entwicklung sieht die Architektur einer Software als Zusammenspiel einzelner Komponenten. Auf diese Weise lassen sich auch fremde Software-Komponenten in eine Software-Architektur einbinden. Einzelne Software-Komponenten werden häufig nicht selbst entwickelt, sondern von fremden Anbietern eingekauft. In diesem Fall ist für den Käufer der Einblick in die interne Logik der eingekauften Komponente nicht möglich. Der Käufer kann aber Interesse an der Performance der Komponente haben, die von der internen Logik abhängig ist. Hierzu dienen Palladio Component Models (kurz: PCM). Palladio Component Models dienen der Spezifikation von Performance und Ressourenverbrauch von Software-Komponenten. Der Komponentenhersteller bietet im gegebenen Fall neben der Software-Komponente das entsprechende Palladio Component Model zur Komponente an. Die Aufgabe eines Software-Architekten liegt im Entwickeln einer Software-Architektur bestehend aus einzelnen Komponenten, sowie der Analyse bezüglich der Performance seiner entwickelten Architektur (vgl. [Koziolek *et al.* , 2008]). Im Falle einer unerwarteten Überlastung der Hardware im System auf der Implementierungsebene stellt sich die Frage nach der Ursache für die Überlastung. Diese kann an der Performance einer einzelnen Komponente liegen.

Performance-Tests können dazu dienen, die Performance einer Software-Komponente oder eines Systems zu überprüfen. Da den fremden Software-Komponenten des beschriebenen Falls Performance-beschreibende Palladio-Modelle beiliegen, liegt die Idee nahe, Performance-Tests zu schreiben, die den Angaben in den Palladio-Daten entsprechen. Somit ließen sich miteinander vergleichbare Ergebnisse aus Palladio-Simulationen und Performance-Tests erzeugen. Die zu entwickelnden Performance-Tests sollen dazu als Eingaben die gleiche Anfragelast haben, wie sie in den zugrundeliegenden Palladio-Modellen modelliert ist. Ist die Hardware- und Software-Umgebung beim Performance-Test übereinstimmend mit der in Palladio modellierten, können die Ergebnisse des Performance-Tests mit denen einer Palladio-Simulation verglichen werden. Das bedeutet, bei korrekten Performance-Angaben in Palladio kann von vergleichbaren Antwortzeiten und Ressourcenverbrauchen der Implementierung ausgegangen werden.

Innerhalb Palladios können Anfragelasten modelliert werden, deren manuelle

Umsetzung in einem Test unzählige Testpersonen erfordern würde. Darüber hinaus wäre auf lange Sicht nicht davon auszugehen, dass eine Testperson sich über die volle Dauer des Performance-Tests genau so verhält, wie in Palladio modelliert. Deshalb bietet sich der Einsatz automatisierter Tests an. Benötigt werden also Testskripte, deren Verhalten sich an die Modellierung in Palladio hält.

Manuelle Programmierung ist arbeitsaufwendig und anfällig für Fehler. Unentdeckte Fehler in den Testskripten können verheerende Auswirkungen haben, da ein abweichendes Verhalten der Skripte falsche Ergebnisse liefern und somit den Anwender auf falsche Rückschlüsse bringen könnte. Daher bietet sich der Einsatz automatischer Codegenerierung an. Diese kann den Prozess der Programmierung unterstützen, Fehler vermeiden, Vertrauen in die Ergebnisse gewährleisten und Zeit sparen. Als Datenquelle für die Codegenerierung dienen Palladio-Modelle. Diese sehen ihre Verwendung für Software-Tests jedoch im ursprünglichen Sinne nicht vor.

1.2 Zielsetzung und durchzuführende Arbeiten

Ziel dieser Arbeit ist die Entwicklung einer automatischen Generierung von Performance-Testskripten aus Palladio-Modellen. Die Codegenerierung soll als Eclipse-Plugin realisiert werden. Generierte Testskripte sollen in der Programmiersprache Java geschrieben sein und das JUnit-Framework verwenden.

Um die Entwicklung durchzuführen, ist zunächst die Einarbeitung in die Thematik von Software-Tests, insbesondere Performance-Tests, sowie der Verwendung von JUnit für Performance-Tests nötig. Anschließend muss eine Technik entwickelt oder gesucht werden, durch die sich Palladio-Modelle laden, lesen und in Code umwandeln lassen. Darüber hinaus bedarf es einer Übersicht über den Aufbau und die Semantik von Palladio-Modellen. Kapitel 2 beschreibt technische Grundlagen, die Voraussetzung für den Rest der Arbeit sind. Danach stellt sich die Frage nach der Interpretation der Modell-Daten für Software-Tests und Umsetzung der Modell-Daten in Java-Code. In Kapitel 3 folgt die Beschreibung der entwickelten Konzepte und Lösungen hierzu. Anschließend an den konzeptionellen Teil lässt sich die entwickelte Lösung implementieren und exemplarisch am Software-Projekt CoCoME anwenden. Kapitel 4 befasst sich mit der Evaluation der Implementierung. Schlussendlich gibt Kapitel 5 ein Fazit und Ausblick auf mögliche Erweiterungen der entwickelten Lösung.

2 Grundlagen und Stand der Technik

Für das beschriebene zu entwickelnde Plugin bedarf es einer Reihe an Grundlagen, die in diesem Kapitel näher erläutert werden. Zunächst befasst sich Kapitel 2.1 mit der grundlegenden Struktur des Palladio Component Models, das als Modellgrundlage für die Codegenerierung dienen soll. Danach verschafft Kapitel 2.2 einen Überblick über Software-Tests sowie Test-Frameworks für Java. Kapitel 2.3 befasst sich schließlich mit Methoden der Codegenerierung. Wie sich an Abbildung 1 erkennen lässt, sind somit die drei zentralen Themengebiete der Arbeit abgedeckt.

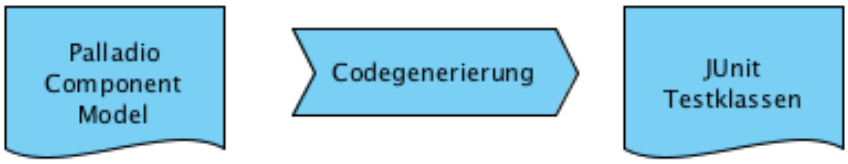


Abbildung 1: Die drei zentralen Themengebiete dieser Arbeit
Quelle: eigene Darstellung

2.1 Palladio Component Model

Das Palladio Component Model ist ein von den Universitäten Karlsruhe und Paderborn entwickeltes und implementiertes Konzept zur modellbasierten Performance Prediction. Zur Bearbeitung von Palladio Modellen stehen dem Nutzer zahlreiche Editoren innerhalb eines Eclipse-Plugins bereit. Die Editoren öffnen hierzu vorliegende Modell-Dateien und interpretieren deren Inhalt als Modelle durch den Abgleich mit zugehörigen Metamodellen. Darüber hinaus lässt sich eine Simulation des aufgestellten Modells durchführen, um die Performance des modellierten Systems zu bewerten. Insgesamt lassen sich 6 Arten von Modellen unterscheiden, für deren Bearbeitung sich unterschiedliche Spezialisten innerhalb eines Software-Projektes anbieten (vgl. [Koziolek *et al.* , 2008]).

Als Definitionsbasis für zu modellierende Software-Komponenten und genutzte Interfaces dient das sogenannte Repository-Modell. Definierte Komponenten können Schnittstellen bedienen (Provided Role) sowie anfordern (Required Role). Im System-Modell werden von den zuvor definierten Komponenten und Schnittstellen zusammenwirkende Instanzen definiert. Auf diese Weise lassen sich im System mehrere Instanzen eines im Repository definierten Elements gleichzeitig definieren (analog zum

Prinzip von Klassen und Objekt-Instanzen in der objektorientierten Programmierung). Das System kann Schnittstellen nach außen bedienen. Diese Schnittstellen ermöglichen modellierten virtuellen Nutzern Service-Anfragen von außen. Anfragen an die definierten äußeren Nutzer-Schnittstellen lassen sich anschließend an Software-Komponenten weiter delegieren. Es sind somit also durch System und Repository die Software-Komponenten spezifiziert. Nun fehlt noch eine Spezifikation der Hardware-Ressourcen im Modell. Dieses geschieht durch das Resource-Modell. Im Allocation-Modell lassen sich die im System-Modell angelegten Software-Komponenten den Recheneinheiten zuweisen, die im Resource-Modell definiert wurden. Für die Methoden, die Software-Komponenten auf den jeweiligen Hardware-Komponenten ausführen, lassen sich sogenannte Resource Demanding Service Effect Specifications (kurz: seff) aufstellen. Diese beschreiben in Kontrollfluss-Form den Ressourcenverbrauch, den der jeweilige Methodenaufruf auslöst. Somit sind die Belastungspotentiale des Modells festgesetzt. Zur Modellierung von Nutzeranfragen im Modell dient das sogenannte Usage-Modell, im weiteren Verlauf auch UsageModel genannt. Dort lassen sich Nutzungsszenarien definieren. Durch den sogenannten Workload wird die Ankunftsrate von Nutzern spezifiziert. Unterschieden wird hier zwischen offenem und geschlossenem Workload (OpenWorkload und ClosedWorkload). Im Falle des offenen Workloads treten Nutzer in einer bestimmten Rate nacheinander in den agierenden Zustand ein und verlassen diesen nach Beendigung ihrer Aktionen dauerhaft. Der geschlossene Workload dagegen definiert eine feste Anzahl von Nutzern, die zugleich beginnen und ihre Aktionen nach einer Denkpause in Endlosschleife wiederholen. Die Abfolge von Dienstanforderungen wird durch das sogenannte ScenarioBehavior spezifiziert. Dieses beinhaltet einen komplexen Kontrollfluss mit Dienstaufrufen und spezifizierten Pausen.

Das Plugin ermöglicht das Bearbeiten der Modell-Dateien in hierarchischer Sicht. Darüber hinaus existiert für jede erstellte Modell-Datei eine verknüpfte Diagramm-Datei. Diese Diagramm-Dateien lassen sich in benutzerfreundlichen Diagramm-Editoren bearbeiten. Abbildung 2 zeigt ein triviales Beispiel für ein Usage-Modell, System-Modell und Repository-Modell in Diagramm-Form.

Das Palladio-Metamodell stützt sich im tieferen Detailsgrad auf weitere Metamodelle, z.B. auf die Metamodelle “stoex” und “probfunction” für die Modellierung stochastischer Ausdrücke, die in Kapitel 3.2.3 aufgegriffen werden. Alle Modelltypen vereint, dass sie sich mithilfe des entsprechend dazugehörigen Metamodells auslesen und so für eigene Zwecke nutzen lassen, wie im Falle dieser Arbeit für das Erstellen

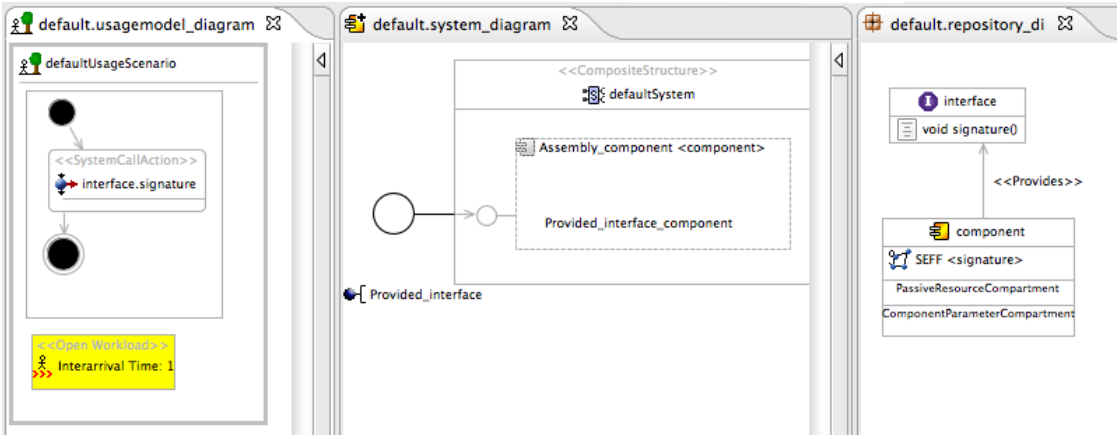


Abbildung 2: Triviales Beispiel für Usage-Modell, System-Modell und Repository-Modell in Palladio-Editor
Quelle: eigene Darstellung

von Testskripten.

2.2 Software-Tests

Wie sich in Softwareentwicklungs-Modellen wie dem V-Modell zeigt, sind Software-Tests elementarer Bestandteil der Softwareentwicklung, die auf unterschiedlichen Ebenen in unterschiedlicher Form durchgeführt werden (siehe Abbildung 3). Sie stellen einen erheblichen Anteil am Gesamtaufwand der Softwareentwicklung dar und sind unerlässlich, um die Qualität von Software zu verbessern und Fehler zu minimieren. Jedoch kann ein erfolgreicher Test niemals die Korrektheit einer Software beweisen, sondern lediglich die Anwesenheit von Fehlern aufzeigen (vgl. [Dijkstra, 1990]), wenn er auch Vertrauen in die Verlässlichkeit der Software herzustellen vermag.

[Thaller, 2000] unterscheidet zwischen unterschiedlichen Arten von Software-Tests. Zum einen existieren sogenannte Funktions-Tests. Diese prüfen die Ergebnisse von Services bzw. Methoden, die von einzelnen Softwaremodulen bereitgestellt werden. Da Fehler in der Entwicklung weniger Kosten verursachen, je früher sie entdeckt und beseitigt werden, ist anzuraten, Funktions-Tests zum frühestmöglichen Zeitpunkt, also direkt nach Erstellung des entsprechenden Moduls durchzuführen. Neben Funktions-Tests existieren auch Testtypen für andere Software-Qualitäten. So gibt es zum Beispiel den Typ des Performance-Tests. Kapitel 2.2.1 bringt den Zweck und die Eigenheiten dieser Tests näher.

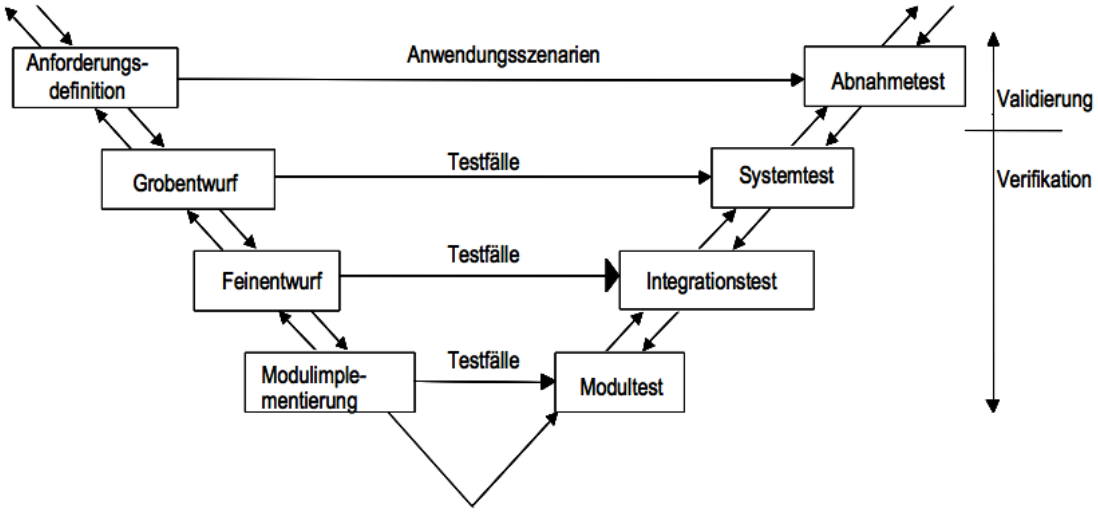


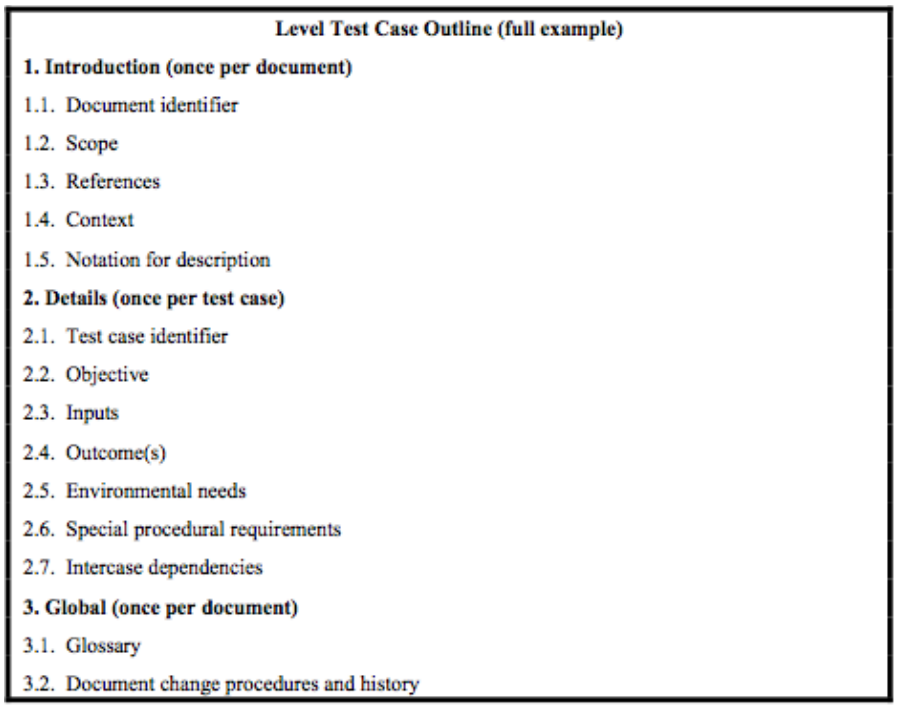
Abbildung 3: Der Aufbau und Ablauf des V-Modells nach Phasen
Quelle: [Balzert, 2002]

Nach [Thaller, 2000] lassen sich Software-Tests auf unterschiedlichen Granularitätsebenen durchführen. Das Spektrum reicht vom fein-granularen Unit-Test bis hin zum System-Test. Ein Unit-Test beschränkt sich auf das Testen einer einzelnen isolierten Software-Komponente. Falls die Komponente zur Arbeit mit weiteren Komponenten kommunizieren muss, werden diese durch trivial gehaltene Dummys ersetzt. Auf diese Weise können auftretende Fehler leicht bestimmten Komponenten zugewiesen werden. Der System-Test dagegen testet das Verhalten einer Software in ihrer Gesamtheit.

Um ein Programm oder eine Software-Komponente zu testen, bedarf es der Aufstellung von Testfällen in einer Testfallspezifikation innerhalb der Testdokumentation. Die Definition eines Testfalls nach IEEE Std. 610.12-1990 lautet: “A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.”[iee, 1990]. Der Aufbau einer Sammlung von Testfällen wird durch den IEEE Std. 829-2008 beschrieben [iee, 2008] (siehe Abbildung 4).

Sämtliche Testfälle einer Sammlung beziehen sich auf den gleichen Kontext (vgl. Abbildung 4). Paragraph 2 der Übersicht bezieht sich auf die einzelnen Bestandteile eines Testfalls, die in IEEE Std. 829-2008 kurz erläutert werden (vgl. [iee, 2008]):

- Test case identifier: Ein eindeutiger Schlüssel zur Identifizierung und Abgrenzung



Level Test Case Outline (full example)

1. Introduction (once per document)

1.1. Document identifier

1.2. Scope

1.3. References

1.4. Context

1.5. Notation for description

2. Details (once per test case)

2.1. Test case identifier

2.2. Objective

2.3. Inputs

2.4. Outcome(s)

2.5. Environmental needs

2.6. Special procedural requirements

2.7. Intercase dependencies

3. Global (once per document)

3.1. Glossary

3.2. Document change procedures and history

Abbildung 4: Der Aufbau einer Testfall-Sammlung nach IEEE Std. 829-2008
Quelle: [iee, 2008]

von anderen Testfällen.

- Objective: Kurze Beschreibung des besonderen Ziels und Zwecks des Testfalls. Die Beschreibung kann auch Informationen über mögliche Risiken und Prioritäten des Testfalls beinhalten.

- Inputs: Die Spezifizierung jedes Eingabeparameters, der für die Durchführung des Testfalls vorgesehen ist. Die Spezifizierung geschieht entweder über den Wert, falls möglich, oder per Namensbenennung. Darüber hinaus werden noch weitere einflussnehmende Elemente wie dazugehörige Datenbanken oder Dateien spezifiziert. Nicht nur der Wert der Eingabeparameter ist von Bedeutung, sondern auch ihre Reihenfolge sowie der Zeitablauf der Eingaben.

- Outcome: Die Spezifizierung jedes erwarteten Ausgabewertes, sowie des erwarteten beobachtbaren Verhaltens. Dazu zählen beispielsweise Antwortzeiten.

- Environmental needs: Die Beschreibung der Testumgebung, die notwendig ist, um den Test vorzubereiten, durchzuführen und seine Ergebnisse festzuhalten. Die Umgebung bezieht sich auf Hardware, Software sowie weitere Voraussetzungen wie zum Beispiel speziell trainierte Benutzer.

- Special procedural requirements: Beschreibt zusätzliche Voraussetzungen für Test-Prozeduren, die den betrachteten Testfall ausführen.

- Intercase dependencies: Listet alle Testfälle auf, die durchgeführt werden müssen, bevor der betrachtete Testfall ausgeführt werden kann.

Eine Quelle für erwartete Ausgabeergebnisse kann beispielsweise die Spezifikation über die Software sein, die Kunde und Entwickler gemeinsam erarbeiten. Diese bietet Anhaltspunkte darüber, welche Dienste die zu entwickelnde Software letztendlich in welchem Umfang besitzen soll.

[Thaller, 2000] unterscheidet zwischen zwei grundlegenden Konzepten zur Bestimmung von Eingabeparametern für Testfälle: White-Box-Testing und Black-Box-Testing. Das Prinzip des White-Box-Testing geht davon aus, dass der Testautor den zu testenden Code kennt. Aufbauend auf der Logik des Kontrollflusses werden Eingabeparameter von Testfällen so ausgewählt, dass diese einen bestimmten Grad der Codeabdeckung erfüllen. Das bedeutet, die einzelnen Schritte des Kontrollflusses sollen unter bestimmten Bedingungen durchlaufen worden sein. Das Black-Box-Testing hingegen setzt keine Kenntnis über den internen Kontrollfluss voraus. Vielmehr werden die Eingabewerte intuitiv bestimmt. Die Funktion des zu testenden Codes lässt oft auf Eingabewerte schließen, die ein hohes Fehlerpotential haben. Meist eignet sich White-Box-Testing eher für fein-granulare Tests wie Unit-Tests, während auf Ebenen groberer Granularität Black-Box-Testing bevorzugt wird (vgl. [Rakitin, 1997]). Das in Kapitel 1 beschriebene Problem bezieht sich auf Black-Box-Testing, da das Innere der zu verwendenden Software-Komponenten dem Software-Architekten fremd sind.

Da Testfälle sich auf Software beziehen, muss der Ablauf eines Testfalls auch in Code implementiert werden. Dieses Code-Segment soll den entsprechenden Testfall repräsentieren und auf der zu testenden Komponente den Test durchführen können. In der objektorientierten Programmierung bietet sich das Anlegen einer eigenen Klasse zur Repräsentation eines Testfalls an. Diese Klasse lässt sich in Entwurfsmustern verwenden, wie sie in der objektorientierten Programmierung üblich sind. Ein verwendetes Entwurfsmuster bei Testklassen kann das Decorator-Pattern sein. Das Decorator-Pattern umschließt eine Klasse, um ihre Funktionalität zu erweitern (vgl. [Gamma *et al.* , 2004]). Kapitel 2.2.2 und Kapitel 2.2.3 greifen das Decorator-Pattern erneut auf.

Software-Tests lassen sich in vielen Fällen auch automatisieren, allerdings oft nicht vollständig. In der Entwurfsphase eines Tests ist häufig die Kreativität und Fähigkeit eines Menschen von Nöten. Die Testausführung an sich jedoch lässt sich wiederum sehr gut automatisieren. Dies gilt besonders für Tests auf Benutzeroberflächen, die einen Menschen für gewöhnlich viel Zeit und Aufwand kosten. Sollte die Oberfläche sich jedoch im Laufe des Projekts noch regelmäßig ändern, bedeutet dies auch erhöhten Aufwand in der Automatisierung. Für den Ergebnisabgleich am Ende eines Tests ist zumindest im Falle eines Fehlers ein menschlicher Einbezug nötig, um zur Analyse des Fehlers überzugehen. An sich lässt sich sagen, dass durch automatisiertes Testen keine zusätzlichen Fehler zum manuellen Testen gefunden werden, sofern die gleichen Testfälle vorliegen. Zudem gestaltet es sich in der Regel recht aufwändig, wenn auch der Nutzen daraus höher ausfallen kann als der Aufwand.

2.2.1 Performance-Tests

Software-Tests lassen sich nutzen, um externe Softwarequalitäten zu überprüfen, also Softwarequalitäten, die sich auf die Anwendung der Software beziehen. [Engels & Soltenborn, 2009] nennt folgende externe Softwarequalitäten:

- Korrektheit
- Zuverlässigkeit
- Robustheit
- Effizienz
- Benutzerfreundlichkeit
- (externe) Verständlichkeit

Neben funktionalen Tests, die die Korrektheit, Zuverlässigkeit und Robustheit eines Systems überprüfen, existieren zusätzlich nicht-funktionale Tests. Diese beziehen sich weniger auf die Korrektheit eines Ergebnisses, sondern eher auf die Art, wie es zustande kam. Vertreter nicht-funktionaler Tests sind unter anderem Performance-Tests, die die Leistungsfähigkeit eines Systems unter Benutzerlast testen. Somit sind Performance-Tests nutzbar, um die Softwarequalität “Effizienz” zu überprüfen. Abbildung 5 gibt eine Übersicht über die erwähnten Testtypen und ihren Bezug auf Softwarequalitäten.

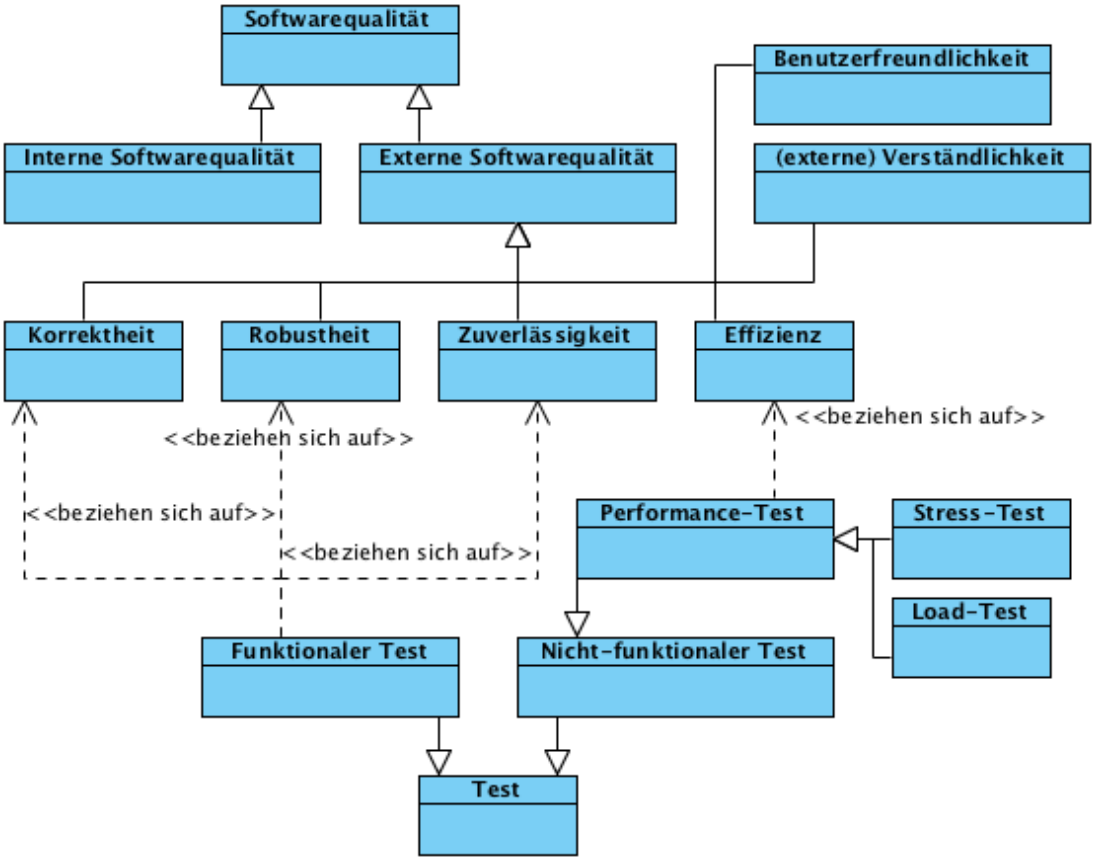


Abbildung 5: Bezug von Software-Tests auf Softwarequalitäten
Quelle: eigene Darstellung

Performance beschreibt nach [Stolze, 2008] die Leistung eines Systems. Die Komplexität und Beschaffenheit dieses Systems kann hierbei variieren. Das betrachtete System kann aus Hardware- und Software-Komponenten bestehen, kann aber genauso gut nur ein einzelner Algorithmus sein. Entscheidend sind nicht-funktionale Indikatoren, die den Nutzen des Systems ausdrücken. Oft sind für den Nutzen zeitliche Indikatoren von Bedeutung. [Molyneaux, 2009] unterscheidet zwischen zwei Kategorien von Performance-Indikatoren. Zum einen sind dies Service-orientierte, zum anderen Effizienz-orientierte Indikatoren. Ein häufiger Schlüsselindikator bei Informationssystemen ist der Datendurchsatz, also die Menge der Daten, die in einer bestimmten Zeit verarbeitet werden können. Der Datendurchsatz ist ein typisches Beispiel für einen Effizienz-orientierten Indikator. Ein anderer zeitlicher Indikator zur Performance-Bewertung kann die Antwortzeit eines Systems sein, ein Beispiel für Service-orientierte Indikatoren. Die Antwortzeit kann zum Beispiel bei Webanwendungen wichtig sein. Für den einzelnen Nutzer ist in diesem Fall der gesamte

Datendurchsatz nicht von Bedeutung. Entscheidend sind für ihn die Antwortzeiten für seine einzelnen Anfragen. Dennoch müssen für die Performance-Bestimmung auch die möglichen Anfragen parallel agierender Nutzer berücksichtigt werden, da diese Einfluss auf die Antwortzeit einer Anfrage haben können.

[Stolze, 2008] hebt zwei Arten von Performance-Tests stark hervor: Den Lasttest und den Stresstest. Für Lasttests werden eine feste Nutzeranzahl und ein fester, längerer Zeitraum zum Testen definiert. Innerhalb dieses Zeitraums führen die Nutzer Anfragen aus. Somit lässt sich erkennen, ob das System unter den gegebenen Bedingungen zufriedenstellende Schlüsselparameter vorweist. Diese Art von Performance-Tests macht ausschließlich Sinn, wenn von einer Beeinflussung der Ergebnisse durch die Nutzermenge ausgegangen wird. Im Gegensatz dazu steht der Stresstest, dem keine Nutzerhöchstzahl und kein fester Zeitraum zugeschrieben wird. Stattdessen wird in kurzer Zeit eine Vielzahl von Anfragen auf das System ausgeführt. Die Anfragelast wird so lange erhöht, bis in der Performance eine bestimmte Grenze überschritten ist.

Insgesamt sieht der Prozess des Performance-Tests nach [Stolze, 2008] folgende Schritte vor:

- Bestimmung der Parameter als Ausdruck für Systemleistung
- Definition der Performancegrenzen
- Durchführung
- Vergleich der Ist-Werte mit den Soll-Werten
- Bestimmung der Schwachstelle
- Optimierung des Systems

Zur Bestimmung von Schwachstellen bei komponentenbasierter Softwareentwicklung beschreibt [Brüseke *et al.* , 2011] das Verfahren der Performance-Blame-Analyse. Die Schwachstelle kann im Design der Softwarearchitektur liegen, die aus einzelnen Komponenten gebildet wurde. Auch kann die Schwachstelle durch mangelhafte Zuweisung von Hardwareressourcen zu einzelnen Komponenten entstehen. Genauso kann die Schwachstelle aber auch einer der verwendeten Software-Komponenten geschuldet sein. Die Performance-Blame-Analyse setzt hier an, indem sie für einzelne Komponenten Performance-Ergebnisse aus Tests mit erwarteten Ergebnissen aus Simulationen der Performance-Prediction vergleicht. Abbildung 6 illustriert das Konzept der

Performance-Blame-Analyse.

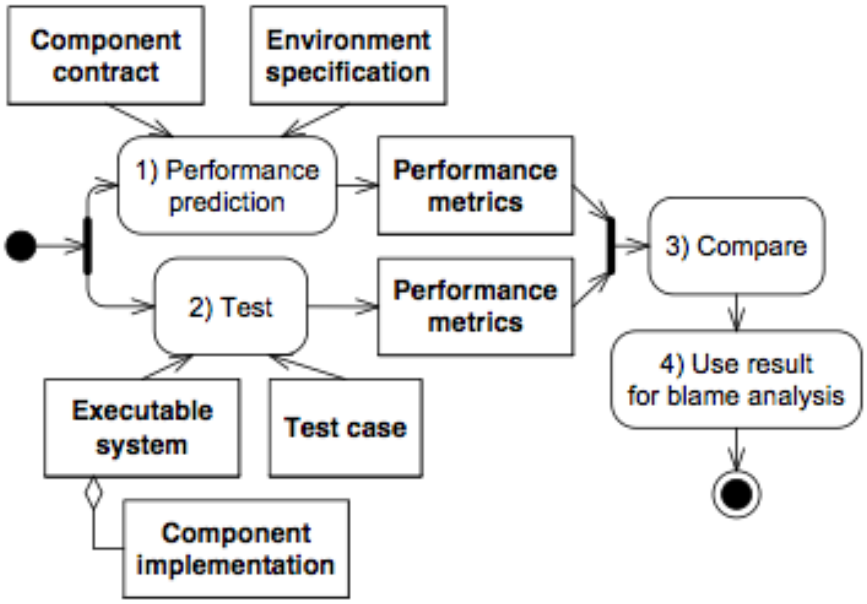


Abbildung 6: Das Konzept der Performance-Blame-Analyse
Quelle: [Brüseke *et al.* , 2011]

Neben parallelen Nutzern können laut [Stolze, 2008] auch Hintergrundprozesse in einem Informations-System das Performance-Ergebnis beeinflussen. Abbildung 7 stellt eine regelmäßige Zusatzbelastung eines Systems durch Hintergrundprozesse dar. Besonders im Falle eines Stresstests kann eine solche variierende Zusatzbelastung das Ergebnis verfälschen. Auch kann das Rauschen unterschiedliche Ergebnisse gegenüber Performance vorhersagenden Simulationen erzeugen, sofern diese die zusätzliche Belastung nicht betrachten. Zur Minimierung der Verfälschung dient die sogenannte Rauschwertermittlung. Das Rauschen, also die Belastung des Systems durch Hintergrundprozesse wird dabei vor dem Performance-Test ermittelt. Anschließend werden die Rauschwerte in die Ergebnisse mit einberechnet. Sind die zusätzlichen Belastungen allerdings im Betrieb mit den selben Effekten zu erwarten, so müssen diese im Performance-Test auch voll mitberücksichtigt werden.

Bei Performance-Tests sind Testautomatisierungen oft notwendig, da die Belastung des Systems durch viele Nutzeranfragen sich ohne Automatisierung meist nicht realisieren lässt. Außerdem gibt [Dustin *et al.* , 2009] zu bedenken, dass Lasttests, die bis zu 72 Stunden dauern können, die Leistungsfähigkeit von Menschen überschreiten und

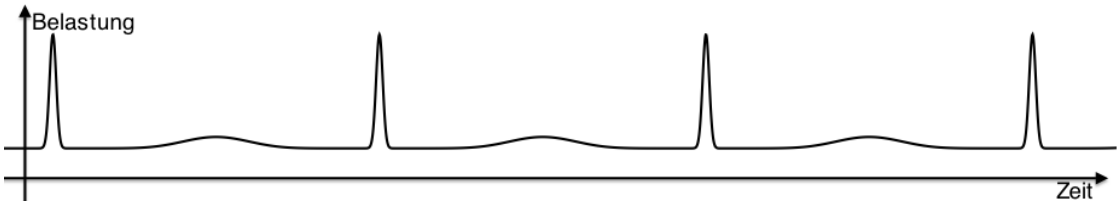


Abbildung 7: Darstellung eines regelmäßigen Belastungsrauschens durch Hintergrundprozesse
Quelle: eigene Darstellung

eine Datenmenge an Ergebnissen erzeugen, die ohne elektronische Unterstützung nicht zu bewältigen ist.

2.2.2 Testen mit JUnit in Eclipse

Das Projekt dieser Arbeit bezieht sich auf Performance-Tests für Software, die auf Java-Code basiert. Ein weit verbreitetes frei verfügbares Entwicklungswerkzeug für Java-Projekte ist das Tool Eclipse (vgl. [Wolmeringer & Klein, 2006]). Gesucht ist also ein Framework, das Performance-Tests für Java-Projekte innerhalb von Eclipse unterstützt. Eclipse beinhaltet standardmäßig das Test-Framework JUnit für Java. Der primäre Zweck von JUnit liegt im zuvor erwähnten Unittest (siehe Kapitel 2.2). Jedoch existieren Erweiterungen für JUnit, die das Framework auch für Performance-Tests praktikabel machen. Kapitel 2.2.3 befasst sich mit einer solchen Erweiterung.

Kapitel 2.2 beschrieb die Verwendung von Testfällen, um Testabläufe zu planen und durchzuführen. Um Testfälle innerhalb von JUnit zu realisieren, bietet das Framework die Klasse TestCase an. Der Nutzer kann eigene Testklassen schreiben, die von dieser Klasse erben. Die Klasse TestCase besitzt einen Konstruktor mit einem zur Identifikation dienenden Namen als Parameter. Innerhalb der Klasse TestCase besteht die Möglichkeit, öffentliche Methoden ohne Eingabeparameter und Rückgabewert anzulegen, deren Bezeichnung mit “test” beginnen. JUnit erkennt diese Methoden automatisch und interpretiert sie als einzelne definierte Testfälle. Existieren mehrere solcher Testmethoden innerhalb einer TestCase-Klasse, werden diese von JUnit im Testdurchlauf in willkürlicher Reihenfolge jeweils einmal ausgeführt (vgl. [Westphal, 2001]). Um die notwendige Umgebung für die Testfälle herzustellen, existieren innerhalb der TestCase-Klasse die zu spezifizierenden Methoden “setUp” und “tearDown”. Diese werden entsprechend ihrem Namen jeweils vor bzw. nach jeder Test-Methode ausgeführt. (vgl. [Westphal, 2001]) Innerhalb der Test-Methoden bietet JUnit den

Aufruf einer Reihe von Methoden, die für die eigentliche Prüfung auf Fehler zuständig sind. Meist geschieht dies über den Abgleich zweier Werte oder Instanzen miteinander. JUnit listet alle Ergebnisse dieser jeweils mit “assert” beginnenden Methoden im Testergebnis auf. Dabei bietet JUnit Ergebnisdarstellung sowohl über eine graphische Benutzeroberfläche als auch über die Konsole.

Um bestehende TestCases einfach um spezielle Funktionen zu erweitern, eignet sich das Prinzip der TestDecorators. Ein TestDecorator bekommt einen Test zugewiesen und besitzt selbst einen eigenen individuell definierbaren Testablauf, in dem wiederum der Aufruf des eingebetteten Tests möglich ist. Ein von JUnit fertig zur Verfügung gestelltes Beispiel ist der RepeatedTest. Dieser DestDecorator führt den eingebetteten Test in festgelegter Anzahl mehrfach hintereinander aus.

Da sich für unterschiedliche Test-Ausgangssituationen das Anlegen unterschiedlicher TestCases anbietet, bedarf es einer Möglicheit größere Mengen von TestCases schnell und regelmäßig automatisiert hintereinander abrufen zu können. Hierzu existiert die Klasse TestSuite von JUnit. Sie dient als Container für Tests, die durch die TestSuite hintereinander ausgeführt werden können. Somit können TestSuites beispielsweise auch alle TestCases sammeln, die Testfälle einer bestimmten Testfall-Sammlung realisieren. In diesem Fall ließe sich redundanter setUp-Code der TestCases untereinander einsparen. Abbildung 8 verdeutlicht die Umsetzung einer Sammlung von Testfällen in JUnit.

TestSuites beinhalten Instanzen, die das Interface “Test” von JUnit implementieren. Dazu gehören insbesondere die Klasse TestCase, sowie die Klasse TestSuite selbst. Somit lässt sich mit Hilfe von TestSuites eine komplexe Baumstruktur aus Tests nach Composite Pattern (siehe Abbildung 9) erzeugen, deren Blätter jeweils TestCases sind. Das Composite Pattern stellt eine Lösung dar, um Objekte zu einer Baumstruktur zusammenzufügen und Methoden rekursiv auf diesem auszuführen (vgl. [Gamma *et al.* , 2004]).

Die Elemente innerhalb einer TestSuite werden zwingend nacheinander abgearbeitet. Ein nebenläufiger Ablauf mehrerer Prozesse ist nicht vorgesehen. Somit wird die Arbeit nebenläufiger Threads nach fertiger Arbeit des Hauptthreads abgebrochen. Dies ergab das ausgiebige Experimentieren mit Threads in JUnit-Tests.

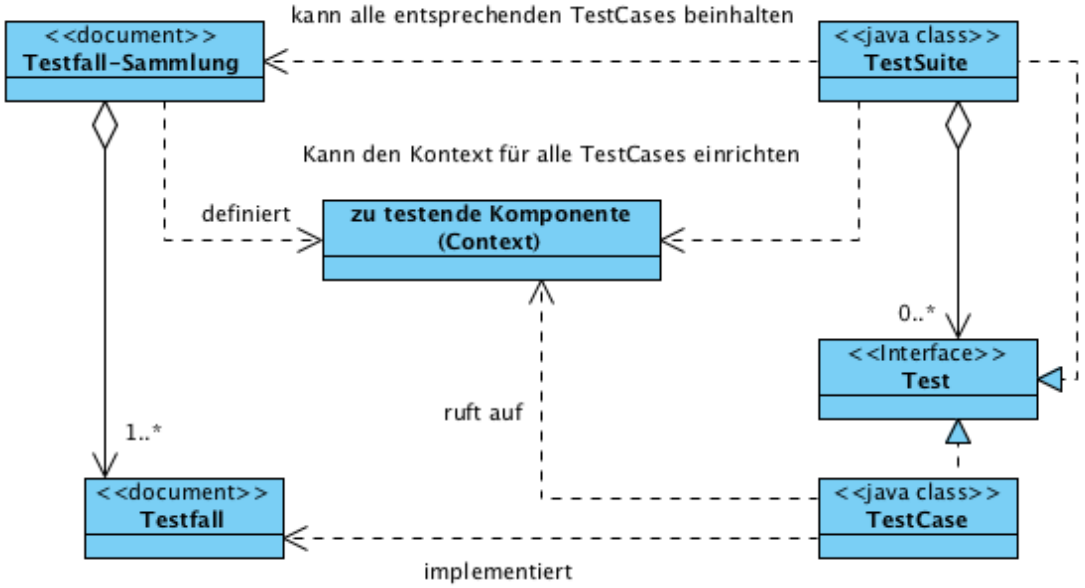


Abbildung 8: Die Umsetzung einer Testfall-Sammlung in JUnit
Quelle: eigene Darstellung

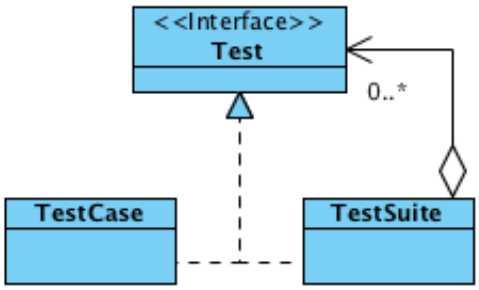


Abbildung 9: Die Verwendung des Composite Patterns in JUnit
Quelle: eigene Darstellung

2.2.3 Performance Testing mit JUnitPerf

Da JUnit selber bei einem Test wie beschrieben also nicht die Beendigung aller Threads abwartet, ist das Zurückgreifen auf weitere Hilfsmittel erforderlich. Eine Möglichkeit zur Problemlösung bildet das Framework JUnitPerf der Firma Clarkware. Das Framework baut auf JUnit auf und bietet vorgefertigte TestDecorators zur Verwendung an. Hierbei stellt ein TestDecorator (siehe Kapitel 2.2.2) selbst nach außen einen TestCase dar, delegiert aber intern unter Berücksichtigung der intern definierten Logik Befehle zu einem inneren TestCase weiter.

Zentraler Vorzug von JUnitPerf für die Realisierung des Projekts ist weniger ein bestimmter TestDecorator, der alle zu lösende Aufgaben übernimmt. Vielmehr erweist sich das Framework als praktisch, da es grundsätzlich die Test-Ausführung auf mehreren gleichzeitig laufenden Threads unterstützt. Die angebotenen Klassen und Interfaces des Frameworks sind speziell auf die Verwendung von Performance-Tests ausgelegt.

Insgesamt beinhaltet JUnitPerf drei TestDecorators. Hierzu zählen der Zeit messende TimedTest und die allgemeiner gehaltene Klasse ThreadedTest. Vor allem zu nennen ist im Kontext dieser Arbeit aber der TestDecorator LoadTest. Dieser ermöglicht den separaten Start eines TestCases mehrfach zur gleichen Zeit. Dabei wird jeder TestCase-Instanz ein virtueller Nutzer zugewiesen, dessen Aktionen in einem separaten Thread laufen (vgl. [jun, n.d.]). Wichtige zu übergebende Parameter sind der innere zu verwendende TestCase, die Anzahl der zu startenden TestCase-Instanzen sowie die Anzahl der zu simulierenden Nutzer. Vorstellbar ist ein RepeatedTest von JUnit als innerer TestCase (siehe Abbildung 10). Die Funktion dieses RepeatedTests lässt sich auch durch das Angeben eines weiteren Parameters schaffen, der die Anzahl an Wiederholungen festlegt. Die Nutzer müssen nicht zwingend zum gleichen Zeitpunkt beginnen. Der LoadTest ermöglicht auch das Starten der Nutzer mit Verzögerung nacheinander. Diese Verzögerung wird durch eine Implementierung des JUnitPerf-eigenen Interfaces Timer definiert. Ein Timer gibt auf Anfrage einen Zahlenwert für die Verzögerung wieder, die so von Anfrage zu Anfrage variieren kann. JUnitPerf liegen mit dem ConstantTimer und dem RandomTimer zwei sehr einfach gehaltene Implementierungen bereits bei.

2.3 Codegenerierung

Die Implementierung von Code – auch zu Testzwecken – gestaltet sich meist zeitaufwändig. Häufig entstehen auf der Implementierungsebene Fehler, deren Beseitigung sehr viel Zeit verschlingen. Daher bietet sich automatische Codegenerierung an, um den Implementierungsprozess zu beschleunigen. Eine entwickelte Automatik zur Codegenerierung kann zwar auch fehlerhaften Code verursachen, die Ursachen für diese Fehler sind dann jedoch in der Automatisierung zu finden. Die auftretenden Fehler sind somit deterministisch. Das bedeutet, dass bei einer erneuten Generierung exakt die gleichen Fehler entstehen. Die Fehlerquellen lassen sich somit schnell durch eine systematische Fehleranalyse in der Automatisierung finden. Manuelle Programmierfehler dagegen entstehen nicht-deterministisch. Dies erschwert das Finden von Fehlern. Abbildung

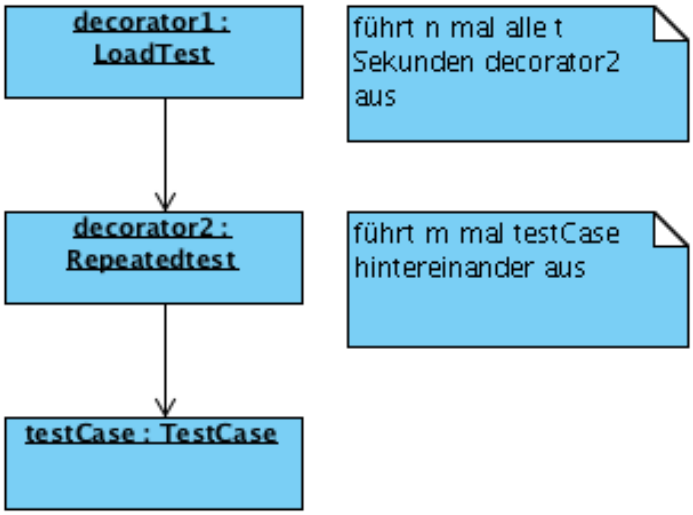


Abbildung 10: Innere Teststruktur des LoadTests als Beispiel mit RepeatedTest
Quelle: eigene Darstellung

11 zeigt, wo automatische Codegenerierung im Prozess der Software-Entwicklung anzusiedeln ist. Auf diese Weise wird aus dem V-Modell ein Y-Modell.

Wird automatische Codegenerierung eingesetzt, so kann es dennoch vorkommen, dass der benötigte Code nicht vollständig generiert werden kann, sondern nach der Generierung manuell vervollständigt werden muss. In diesem Fall rät [Becker, 2010a], unbedingt die Menge an manuellem Code in automatisch generierten Dateien so gering wie möglich zu halten und auszulagern. Folgende Gründe werden für diese Vorgehensweise genannt:

- Bei einer erneuten Generierung besteht die Gefahr, dass alter Code überschrieben wird und manuell geschriebener Code dadurch verloren geht. Diese Gefahr lässt sich bei einigen Technologien zur Softwaregenerierung reduzieren, indem gesicherte Areale im Code angeboten werden, die nicht überschrieben werden können. Ein Beispiel dafür ist openArchitectureWare (siehe Kapitel 2.3.2).

- Wird die Daten-Quelle für die Codegenerierung nachträglich editiert und der Code neu generiert, entsteht die Gefahr von Inkonsistenzen.

- Kann funktionsfähiger Code vollständig aus einer Daten-Quelle generiert werden, kann im Versionsmanagement der Fokus auf der Daten-Quelle beruhen.

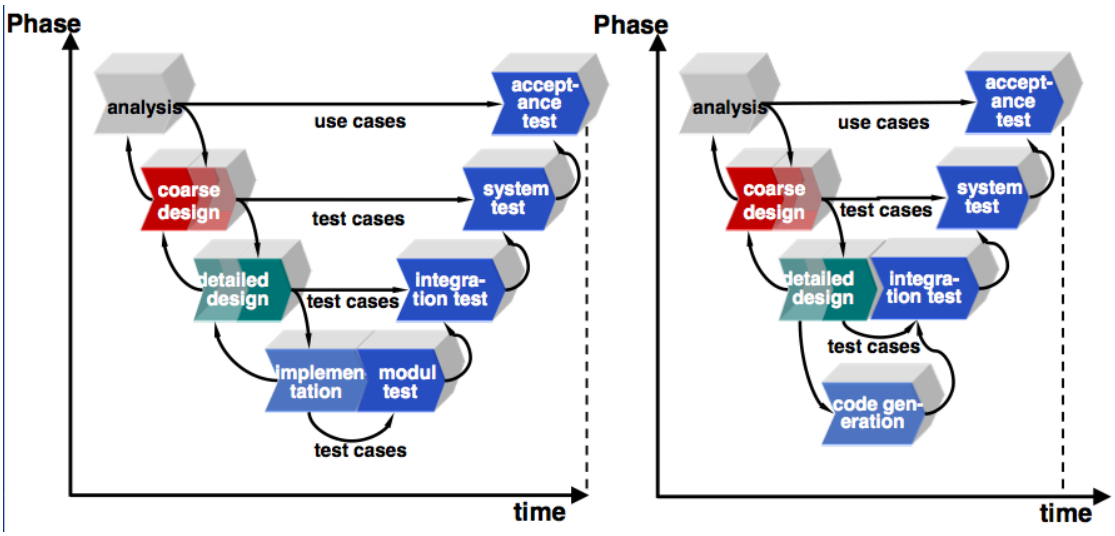


Abbildung 11: Der Aufbau und Ablauf des Y-Modells nach Phasen
Quelle: [Schäfer, 2011]

Zur weiteren Erläuterung von Codegenerierung beschreibt Kapitel 2.3.1 das Prinzip der Modelltransformation als Grundlage von Codegenerierung. Kapitel 2.3.2 befasst sich mit der eingesetzten Technologie openArchitectureWare als Beispiel.

2.3.1 Codegenerierung durch Modelltransformation

Modelltransformation beschreibt Erzeugung oder Editierung eines Modells aus den Informationen und der Struktur eines anderen Modells. Somit sieht jede Modelltransformation ein Quellenmodell und ein Zielmodell vor.

Modelle stellen Instanzen von Metamodellen dar. Da Metamodelle selbst auch Modelle sind, ergibt diese Eigenschaft eine Kette von Modellen, die jeweils Metamodell für ihren Vorgänger sind. Jedes Modell lässt sich durch eine abstrakte Syntax darstellen. Sie beschreibt die Struktur des Modells als Ausprägung ihres Metamodells. Zu einer abstrakten Syntax können unterschiedliche konkrete Syntaxen existieren, die die Struktur des Modells auf unterschiedliche Art modellieren. Auf Grundlage einer konkreten Syntax ist der Nutzer wiederum in der Lage, eine Instanz vom Modell auf der nächsttieferen Metaebene zu modellieren. Abbildung 12 stellt den Zusammenhang von Modell, Metamodell, abstrakter und konkreter Syntax dar. Hierbei ist das Metamodell Grundlage für Modelle, die Beziehungen zwischen Klassen darstellen wollen. Das Modell im Beispiel bildet die Beziehung zwischen einer Eltern-Klasse und einer Kind-Klasse

ab. Für die konkrete Syntax wurde die Notation eines UML-Klassendiagramms gewählt.

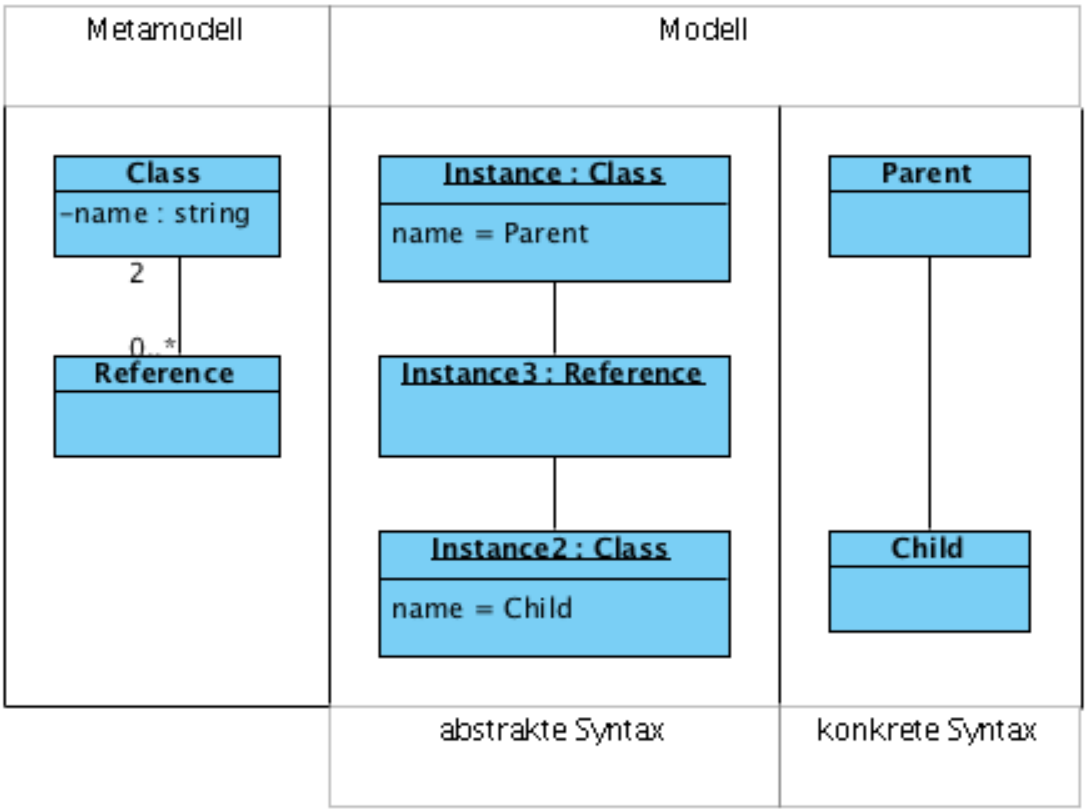


Abbildung 12: Der Zusammenhang von Modell, Metamodell, abstrakter und konkreter Syntax als Beispiel
Quelle: eigene Darstellung

[Becker, 2010b] beschreibt die Strukturierung einer Automatisierung zur Modelltransformation. Abbildung 13 gibt einen Überblick.

Wie zu sehen ist, wird hierbei von Dateien im XML Metadata Interchange-Format (kurz: XMI) als Eingangsdaten ausgegangen. XMI ist ein XML-basierter Standard, der auf die Abbildung von Modellen ausgelegt ist (vgl. [xmi, 2011]). Ein Parser liest die Dateien ein und lädt dadurch das Quellenmodell. Der Modell-Transformator erzeugt anschließend daraus das Zielmodell. [Czarnecki & Helsen, 2003] beschäftigt sich mit charakterisierenden Bestandteilen einer Modelltransformation. Diese werden im Folgenden aufgelistet und kurz erläutert.

- Transformation Rules: Eine Transformationsregel besteht immer aus der Left Handed Side (kurz: LHS) und der Right Handed Side (kurz: RHS). Während die

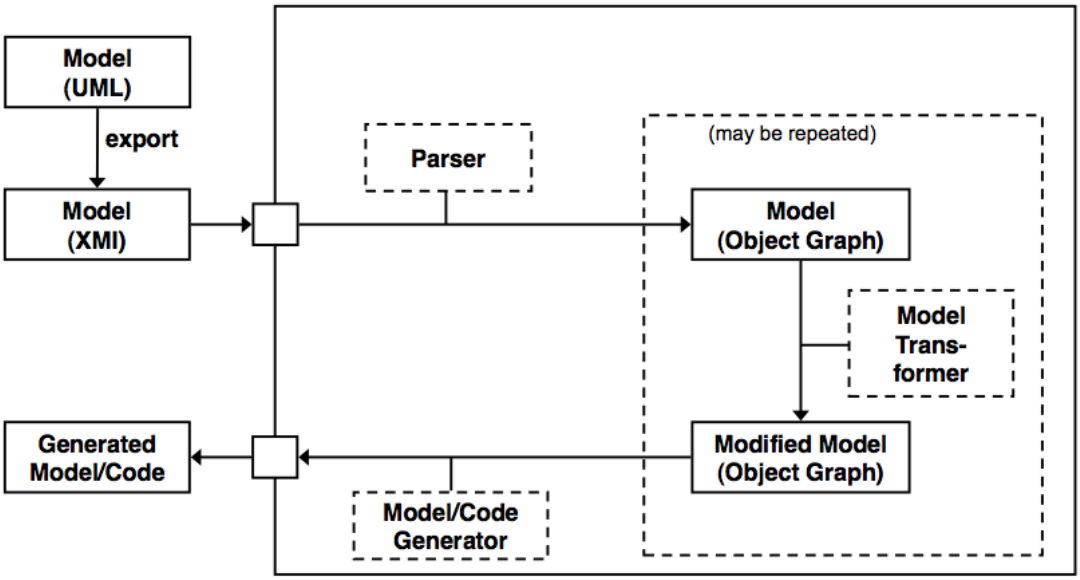


Abbildung 13: Das Prinzip einer Automatisierung zur Modelltransformation
Quelle: [Völter *et al.* , 2006]

LHS den Zugang zum Quellenmodell beschreibt, erläutert die RHS die Umsetzung im Zielmodell.

- Rule Application Scoping: Sowohl für das Quellenmodell als auch für das Zielmodell wird festgeglegt, auf welche Teile der Modelle sich die Transformation beschränkt.

- Source-Target Relationship: Es wird unterschieden, ob das Zielmodell neu erstellt, erweitert oder umgeschrieben wird.

- Rule Application Strategy: Wird das Quellenmodell durchlaufen, wird eine Strategie benötigt, in welcher Reihenfolge die einzelnen Elemente durchlaufen werden. Die Strategie der Transformation kann deterministisch oder auch nichtdeterministisch sein.

- Rule Scheduling: Ebenfalls wird eine Reihenfolge für die Umsetzung von Transformationsregeln festgelegt.

- Rule Organization: Im Falle mehrerer Transformationsregeln müssen diese strukturiert angelegt sein. Es kann möglich sein, dass Transformationsregeln voneinander erben oder einander erweitern.

- Tracing: Um die Arbeit einer Modelltransformation zu überprüfen, hilft das Aufzeichnen von Verbindungen zwischen Elementen in Quelle und Ziel, die an der Umsetzung einer Transformationsregel beteiligt sind.
- Directionality: Um die Synchronisation zweier Modelle zu gewährleisten, können auch beide Modelle sowohl als Quelle als auch als Ziel fungieren. In diesem Fall läge eine bidirektionale Modelltransformation vor, bei einem Quellenmodell und einem Zielmodell eine unidirektionale Modelltransformation.

Nach der Erzeugung des Zielmodells wird dieses anschließend von einem Generator wieder in eine Datei-Form gebracht. Das Format der Ausgabedaten bleibt dadurch unabhängig von den Transformationsregeln des Modell-Transformators. Ausgabedaten einer Automatisierung zur Modelltransformation, wie in Abbildung 13 dargestellt, können wieder im XMI-Format vorliegen. Genauso können aber auch ausführbare Codedateien generiert werden. Modelltransformationen lassen sich somit in Model-To-Model-Transformationen (kurz: M2M) und Model-To-Text-Transformationen (kurz: M2T) bzw. Model-To-Code-Transformationen einteilen. Die Model-To-Code-Transformation stellt genau genommen eine besondere Form der M2M-Transformation dar, bei der die Programmiersprachen-Grammatik des zu generierenden Codes das Metamodell des Zielmodells ist, das durch den Code in einer konkreten Syntax generiert wird (vgl. [Becker, 2010a]).

Im Bereich der M2T-Generierung wird zwischen Visitor-basierten und der häufigen Template-basierten Codegenerierung unterschieden. Visitor-basierte Techniken setzen auf ein Objekt-orientiertes Design. Definierte Visitor-Objekte besuchen Modell-Elemente bestimmten Typs, um Operationen wie das Auslesen von Attributwerten auszuführen. Bei Template-basierten Techniken dagegen werden stattdessen Prototypen typischer Ausgabedateien definiert. Diese Templates zur Definition bestehen aus statischen Codesegmenten und Platzhaltern an Stellen, die je nach Quellenmodell dynamisch zu füllen sind. Kapitel 2.3.2 stellt openArchitectureWare als Vertreter der Template-basierten Techniken vor.

2.3.2 Codegenerierung mit openArchitectureWare

Die entwickelte Lösung basiert auf dem Einsatz von openArchitectureWare (kurz: oAW). Hierbei handelt es sich um ein Template-basiertes Framework zur Codegenerierung für die Entwicklungsumgebung Eclipse. Mittlerweile ist oAW fester Bestandteil

des Eclipse Modelling Projects (vgl. [oaw, 2009]). Dieses Kapitel beschränkt sich jedoch auf die Beschreibung der Bestandteile von oAW. Neben der Codegenerierung eignet sich oAW darüber hinaus zum Model-Checking (vgl. [Lipinski, 2013]).

Herzstück eines oAW-Projektes zur Codegenerierung ist der sogenannte Workflow. Dieser wird in einer Modeling Workflow Engine-Datei (kurz: MWE) definiert. Eine MWE-Datei besteht aus unterschiedlichen Komponenten. Zunächst werden Komponenten vom Typ Reader definiert. Jede Reader-Komponente ist für das Laden der Rohdaten des Modells zuständig. Komponenten des Typs Generator spezifizieren anschließend eine Verwendung der geladenen Modelle. Hierzu werden zunächst alle Metamodelle geladen, die den zu verwendenden Modellen zugrundeliegen. Anschließend wird ein Befehl spezifiziert, der die Arbeit eines Xpand-Templates unter Verwendung geladener Modelle zur Folge hat. Der Ausgabepfad wird im sogenannten Outlet definiert. Dort lassen sich noch Elemente zur Nachbearbeitung des generierten Textes anwählen wie zum Beispiel der JavaBeautifier, der für eine anschauliche Einrückung in Java-Code sorgt. Zusätzlich kann dem Generator noch ein Verzeichnispfad hinzugefügt werden, in dem schreibgeschützte Bereiche berücksichtigt werden. Abbildung 14 veranschaulicht einen Ausschnitt des Aufbaus eines MWE-Workflows. Weitere Themenbereiche wie beispielsweise Model-Checking werden dabei außen vorgelassen.

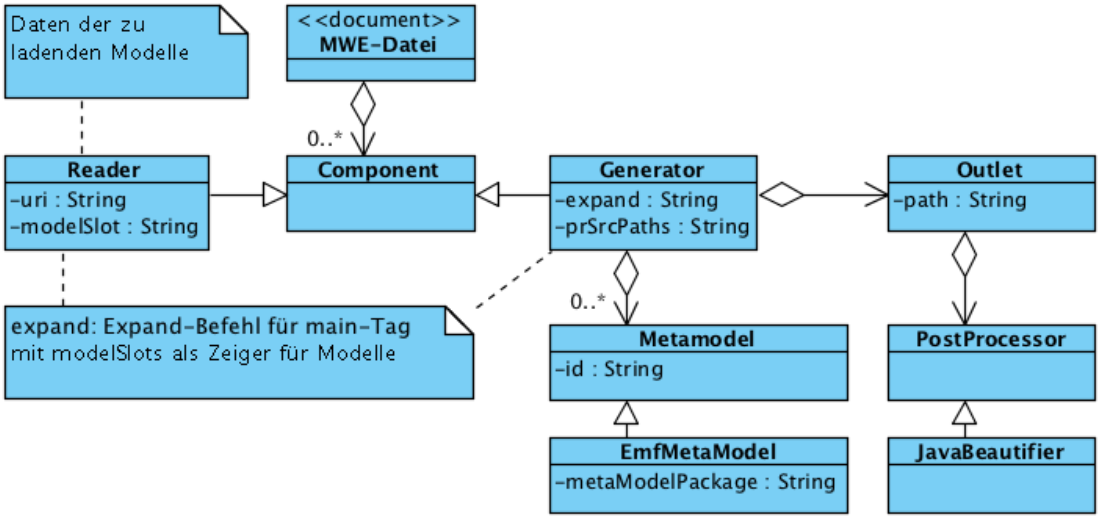


Abbildung 14: Ausschnitt des Aufbaus eines MWE-Workflows
Quelle: eigene Darstellung

In Generator-Komponenten werden Xpand-Templates ausgeführt, denen die geladenen Modelle als Parameter übergeben werden. Xpand ist eine typisierte, Template-basierte Sprache zur M2T-Transformation. Sie ermöglicht das Erstellen neuer Dateien und das Füllen dieser mit Text. Neben statischem Inhalt können die Inhalte auch je nach Modellausprägung dynamisch sein. Der Kontrollfluss eines Xpand-Templates ist analog zu imperativen Programmiersprachen mit sogenannten DEFINE-Tags statt Methoden aufgebaut. Jeder DEFINE-Tag bekommt mindestens ein typisiertes Modellobjekt als Parameter übergeben. Außerdem lassen sich zusätzliche Parameter definieren. Auf Attribute und Referenzen von Parametern kann innerhalb des Tags zugegriffen werden. Zusätzlich lassen sich innerhalb eines Tags wieder neue Tags aufrufen, die an anderer Stelle definiert sind (vgl. [Efftinge & Kadura, n.d.]). Durch den Code-Ausschnitt Algorithmus 1 wird ein Beispiel verbildlicht. Der DEFINE-Tag firstTag bezieht sich auf Elemente der Klasse MetamodelClass und hat einen weiteren Parameter der Klasse MetamodelClass2. Innerhalb von firstTag wird ein Xpand-Befehl für den DEFINE-Tag secondTag aufgerufen.

Algorithmus 1 : Beispielcode für Xpand

```
«DEFINE firstTag(MetamodelClass2 secondParameter) FOR MetamodelClass»
This is static text.
Dynamic value: «this.value».
«EXpand secondTag FOR secondParameter»
«ENDDEFINE»
«DEFINE secondTag FOR MetamodelClass2»
This is static text again.
«ENDDEFINE»
```

Die Entscheidung zum verwendeten Werkzeug zur Codegenerierung in dieser Arbeit fiel auf Xpand wegen der entwicklungsfreundlichen Anwendbarkeit und der Zuschneidung des Funktionsumfangs auf das vorliegende Problem. Dies resultiert aus der konkreten Spezialisierung auf M2T-Transformationen, der fehlenden Ausrichtung auf eine konkrete Sprache des generierten Codes sowie dem unterstützenden Xpand-Editor und der standardmäßigen Einbindung in der Eclipse Modeling Tools-Edition.

3 Entwicklung der Lösung

Nachdem die nötigen Grundlagen beschrieben wurden, befasst sich dieses Kapitel nun mit der Entwicklung der Lösung. Hierzu wird zunächst in Kapitel 3.1 eine detaillierte Anforderungsanalyse betrieben. Anschließend beschreibt Kapitel 3.2 die entwickelten Konzepte zur Umsetzung der Anforderungen. Kapitel 3.3 erläutert die technische Umsetzung des entwickelten Werkzeugs.

3.1 Anforderungsanalyse

Die Aufgabenbeschreibung dieser Arbeit sieht die Generierung von Code für die Durchführung von Tests vor. Für die Ausgabe der Codegenerierung wird von der Aufgabenbeschreibung die Programmiersprache Java festgelegt. Genauso wird die Benutzung des JUnit-Frameworks vorausgesetzt. Der zu generierende Code soll dem Performance-Testing dienen.

Grundlage für die dynamische Generierung sind Daten aus Palladio-Modellen. Da Software-Tests innerhalb des Rollenkonzepts von Palladio keine konkrete Berücksichtigung finden, wird der Software-Architekt somit als typisches Profil eines Nutzers für die zu entwickelnde Lösung herangezogen.

Die Palladio Component Models bieten sich zur Simulation der Komponente vor ihrer Integration in die Software-Architektur an. Der zu generierende Code soll aber dem Testen auf einer existierenden Implementierung dienen. Daher soll die zu entwickelnde Automatisierung Palladio-Modelle durchsuchen und JUnit-Performance-Tests mit Hilfe der gefundenen Informationen generieren. Die Voraussetzungen eines Performance-Tests sollen somit identisch sein mit den Voraussetzungen im gegebenen Palladio Component Model.

Die zu entwickelnde Automatisierung soll dem Nutzer die Auswahl der Eingabedaten, Festsetzung des Ausgabepfades und den Befehl zur Generierung ermöglichen. Eingabe- und Ausgabe-Daten sollen sich innerhalb des aktuellen Eclipse-Workspaces aufhalten. Zusätzlich bedarf es eines Konzepts zum einfachen Zuweisen von Bezeichnungen innerhalb von Palladio auf adequate Bezeichnungen innerhalb der Implementierung, falls diese nicht identisch sein sollten. Abbildung 15 zeigt eine Übersicht über die Produktfunktionen. Die Tabellen 1, 2 und 3 gehen auf die 3 Produktfunktionen für das zu entwickelnde Generierungs-Plugin ein. Ein typisches

Nutzungsszenario ist durch Tabelle 4 zusammengefasst. Anschließend sind die gesammelten Anforderungen in tabellarischer Form aufgelistet.

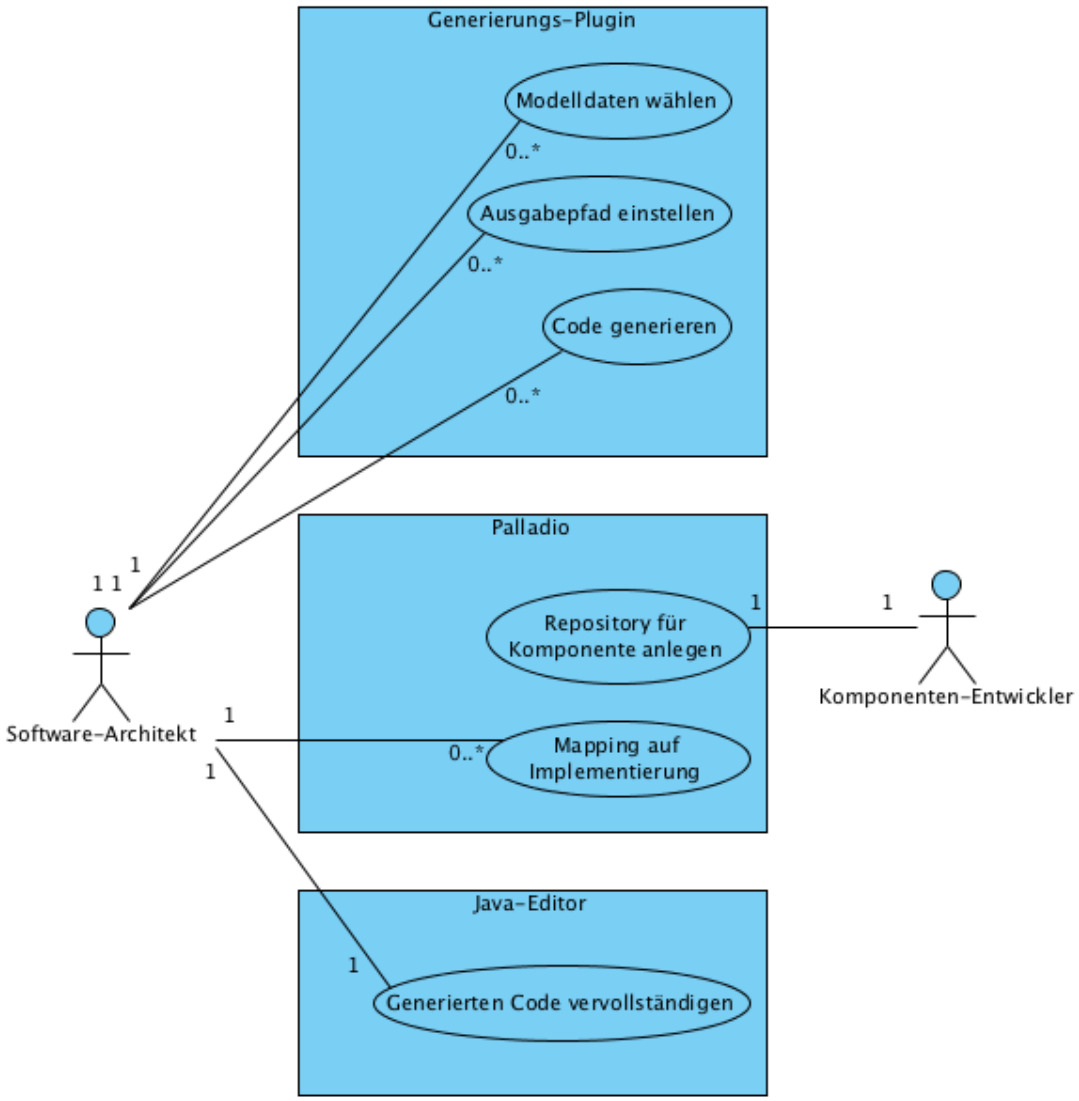


Abbildung 15: Produktfunktionen des beschriebenen Problems
Quelle: eigene Darstellung

Kapitel 3.2 beschreibt die Entwicklung der Automatisierung, die den aufgestellten Anforderungen ensprechen soll.

Name:	**Modelldaten wählen**
Primärer Nutzer:	Software-Architekt
Vorbedingung:	Modelldaten sind verfügbar.
Nachbedingung bei erfolgreicher Ausführung:	Modelldaten sind ausgewählt.
Auslösendes Ereignis:	Software-Architekt ruft die Modellauswahl auf.
Umgebende Systemgrenze:	Eclipse-Plugin
Beteiligte Nutzer:	Software-Architekt

Tabelle 1: Produktfunktion 1

Name:	**Ausgabepfad einstellen**
Primärer Nutzer:	Software-Architekt
Vorbedingung:	Keine
Nachbedingung bei erfolgreicher Ausführung:	Ausgabepfad ist festgelegt.
Auslösendes Ereignis:	Software-Architekt ruft die Ausgabepfadauswahl auf.
Umgebende Systemgrenze:	Eclipse-Plugin
Beteiligte Nutzer:	Software-Architekt

Tabelle 2: Produktfunktion 2

Name:	**Code generieren**
Primärer Nutzer:	Software-Architekt
Vorbedingung:	Modelldaten und Ausgabepfad sind ausgewählt.
Nachbedingung bei erfolgreicher Ausführung:	Code wurde korrekt generiert.
Auslösendes Ereignis:	Software-Architekt ruft die Codegenerierung auf.
Umgebende Systemgrenze:	Eclipse-Plugin
Beteiligte Nutzer:	Software-Architekt

Tabelle 3: Produktfunktion 3

Schritt	Nutzer	Beschreibung der Aktivität
1	Komponenten-Entwickler	Entwicklung der Komponente und Dokumentation in Palladio
2	Software-Architekt	Anpassung der Bezeichnungen innerhalb Palladios an Bezeichnungen in Implementierung der Komponente
3	Software-Architekt	Auswahl der Modelldaten zur Eingabe
4	Software-Architekt	Auswahl eines Ausgabepfades für die Generierung
5	Software-Architekt	Generierung des Codes für den Performance-Test
6	Software-Archtiekt	Manuelle Vervollständigung des Codes an notwendigen Stellen

Tabelle 4: Nutzungsszenario

ID	Anforderung	Beschreibung
1	Ausführbarkeit eines Generierungsauftrags	Der Nutzer soll jederzeit und selbstständig den Befehl zur Durchführung der Generierung geben können.
2	Auswählbarkeit der zu verwendenden Dateien	Der Nutzer soll die Dateien angeben und ändern können, die er zur Generierung heranziehen möchte.
3	Eclipse-Plugin als Mensch-Maschine-Schnittstelle	Die Automatisierung soll als Eclipse-Plugin laufen und dem Nutzer dort eine Benutzerschnittstelle bieten.
4	Palladio-Modelle als Eingabedaten	Als Eingabedaten dienen Teile eines vollständigen Palladio Component Models. Die Automatisierung soll diese Einlesen können.
5	Modelle basierend auf existierenden Metamodellen	Die Palladio-Eingabedaten sollen den existierenden Metamodellen entsprechend interpretiert werden.
6	Anfragen in Simulation vergleichbar ablaufend	Die Gesamtheit der Eingabe-Modelle stellen ein System dar, das in Palladio zur Performance-Prediction simuliert werden kann. Modellierte Anfragen sollen im Performance-Test bis auf wenige Millisekunden Toleranz in gleichem Timing ausgeführt werden, wie in der Simulation.

Tabelle 5: Anforderungen 1

ID	Anforderung	Beschreibung
8	Ausgabedaten in Java-Form	Die Ausgabedaten bestehen aus Java-Quelltext.
9	Nutzbarkeit für Performance-Tests	Die Java-Dateien sollen nach nur wenigen manuellen Vervollständigungen des generierten Codes an markierten Stellen das Ausführen von Performance-Tests auf einem System ermöglichen.
10	Benutzung des JUnit-Frameworks in Ausgabedaten	Die Java-Dateien sollen das Test-Framework JUnit verwenden.
11	Ausgabe-Code lauffähig bei validen Eingabedaten	Generierte Performance-Test-Dateien sollen lauffähig sein, wenn die Simulation der Palladio-Daten auch möglich ist.
12	Semantisch korrekte Übertragung von Modell-Elementen	Die semantisch korrekte Übertragung aller wichtigen Modell-Elemente in Code soll zu einem mit der Simulation in Palladio vergleichbaren Ablauf führen.
13	Modelldaten in Eclipse-Workspace	Die Modelldaten sollen im aktuellen Workspace der Eclipse-Instanz auswählbar sein, in der das Plugin läuft.
14	Einstellbarkeit des Ausgabepfads	Der Nutzer soll Ausgabepfad angeben und ändern können, den er zur Generierung verwenden möchte.
15	Ausgabepfad in Eclipse-Workspace	Der Ausgabepfad soll sich im aktuellen Workspace der Eclipse-Instanz befinden, in dem das Plugin läuft.
16	Mapping auf Implementierung	Dem Nutzer soll ein Konzept zur Verfügung stehen, mit dem er Bezeichnungen innerhalb von Palladio auf adequate Bezeichnungen innerhalb der Implementierung mappen kann.
17	Vollständigkeit	Bis auf selten verwendete Ausnahmen sollen alle wichtigen Elemente des Palladio Component Models in Code übertragen werden, um die Automatisierung nutzbar zu machen.

Tabelle 6: Anforderungen 2

3.2 Entwurf des Testskriptkonzepts

Da Software-Tests innerhalb des Palladio Component Models keine Berücksichtigung finden, existieren keinerlei bindende Richtlinien zur Umsetzung von Palladio-Modellen in Testklassen. Daher bedarf es der Entwicklung einer Interpretationsstrategie für die Modelldaten. Zunächst wird ein Konzept zur Auswahl der zu ladenden Palladio-Modelldateien sowie zu Struktur und Aufgaben der zu generierenden Java-Klassen benötigt. Zur Erschließung aller wichtigen Informationen werden nicht sämtliche Daten eines vollständigen Palladio-Projekts benötigt. Dieses sieht mindestens ein Modell für jeden in Kapitel 2.1 vorgestellten Typ vor.

Die in dem Allocation-Modell und Resource-Modell enthaltenen Informationen sind für den Inhalt der Testskripte jedoch uninteressant. Zwar ist die Testumgebung eines Performance-Tests von zentraler Bedeutung, jedoch dienen die Informationen in den genannten Modellen lediglich zur Simulation innerhalb von Palladio. Im Falle eines Performance-Tests ist die technische Umgebung dagegen real existent. Dass die reale technische Testumgebung tatsächlich mit den Definitionen im Palladio Component Modell in jedem Fall übereinstimmt, muss als gegeben angenommen werden, um eine korrekte Arbeitsweise der generierten Testdateien annehmen zu können. Es wird davon ausgegangen, dass alle wichtigen Software-Komponenten durch entfernte Aufrufe auch von einem beliebigen Rechner aus aufrufbar sind, sofern dieser in einem Netzwerk mit den Hardware-Komponenten des Systems verbunden ist. Eine entfernte Aufrufbarkeit ist ebenfalls für Software-Komponenten notwendig, die auf unterschiedlichen Hardware-Ressourcen arbeiten und miteinander kommunizieren. Die generierten Testkripte sollen ein Nutzerverhalten darstellen. Daher sollten diese zumindest auf Maschinen anwendbar sein, die normalerweise für die Bedienung eines Benutzers angedacht sind.

Wichtiger sind an dieser Stelle statt Allocation-Modell und Resource-Modell vor allem die Modelltypen UsageModel und System. Während im UsageModel die Anfragenabläufe der Nutzer abgebildet sind, bietet das System Informationen über die angebotenen äußeren Interfaces des Systems für die Nutzer. In der korrekten Referenzierung der äußeren Interfaces der Implementierung liegt eine zentrale Schwierigkeit der Automatisierung. Um dies möglichst fehlerfrei und somit mit geringem Aufwand zu realisieren, werden auch Bezeichnungsinformationen aus der Diagrammdatei des Repositorys herangezogen. Das Repository definiert verfügbare Komponenten, Interfaces und deren Beziehungen. Genauere Details zur Verwendung des Repository-Diagramms

werden in Kapitel 3.2.4 beschrieben. Insgesamt bedient sich die Automatisierung also der Informationen aus folgenden Datei-Quellen:

- Eine UsageModel-Modelldatei
- Eine System-Modelldatei
- Eine Repository-Diagrammdatei

Nach der Auswahl der zu ladenden Modelle und somit einer ersten groben Festlegung des Rule Application Scoping der Modelltransformation (siehe Kapitel 2.3.1) stellt sich als nächstes die Frage nach Aufgabe und Struktur der zu generierenden Dateien.

Um dem Prinizip von JUnit gerecht zu werden, bedarf es für den zu generierenden Code eines Baums bestehend aus TestSuites und TestCases. Der Testaufruf der Wurzel muss den Testaufruf sämtlicher untergeordneter Elemente zur Folge haben. Abbildung 16 gibt einen vereinfachten Überblick über die zu erzeugenden Elemente und ihre Entsprechung im UsageModel. Da wie erwähnt lediglich eine UsageModel-Datei zugleich zur Generierung vorgesehen ist, lässt sich diese als Repräsentant der als Wurzel dienenden TestSuite betrachten. Die einzelnen Kontrollflüsse des UsageModels werden jeweils durch einzelne TestCases repräsentiert. Kapitel 3.2.1 erläutert und begründet Aufbau und Methodik der als Wurzel dienenden TestSuite. Anschließend befasst sich Kapitel 3.2.2 mit der Übertragung der Arbeitsabläufe innerhalb einzelner UsageScenarios auf Java-Code innerhalb generierter TestCase-Klassen. Als eigenständigen komplexen Teil beschreibt Kapitel 3.2.3 die Übertragung von statischen Zahlenwerten und Zufallsvariablen aus Palladio-Modellen in Code. Zu guter Letzt befasst sich Kapitel 3.2.4 mit den Konzepten zur Editierung der Modelle vor der Generierung, sowie des Codes nach der Generierung.

3.2.1 Generierung der TestSuite

Als Ausgangsdaten für die Generierung der als Wurzel dienenden TestSuite genügen ausschließlich die Informationen, die im UsageModel angegeben sind. Dieses enthält ein bis mehrere UsageScenarios, Ablaufszenarien für unterschiedliche Use Cases. Die Struktur eines UsageModels legt den Gedanken nahe, ein UsageScenario repräsentiere genau einen Testfall. Tatsächlich jedoch existiert keine verbindliche Definition, wie ein UsageModel in Bezug auf Testfälle zu interpretieren ist. Die zu entwickelnde Automatisierung interpretiert die Gesamtheit aller UsageScenarios in einem UsageModel als

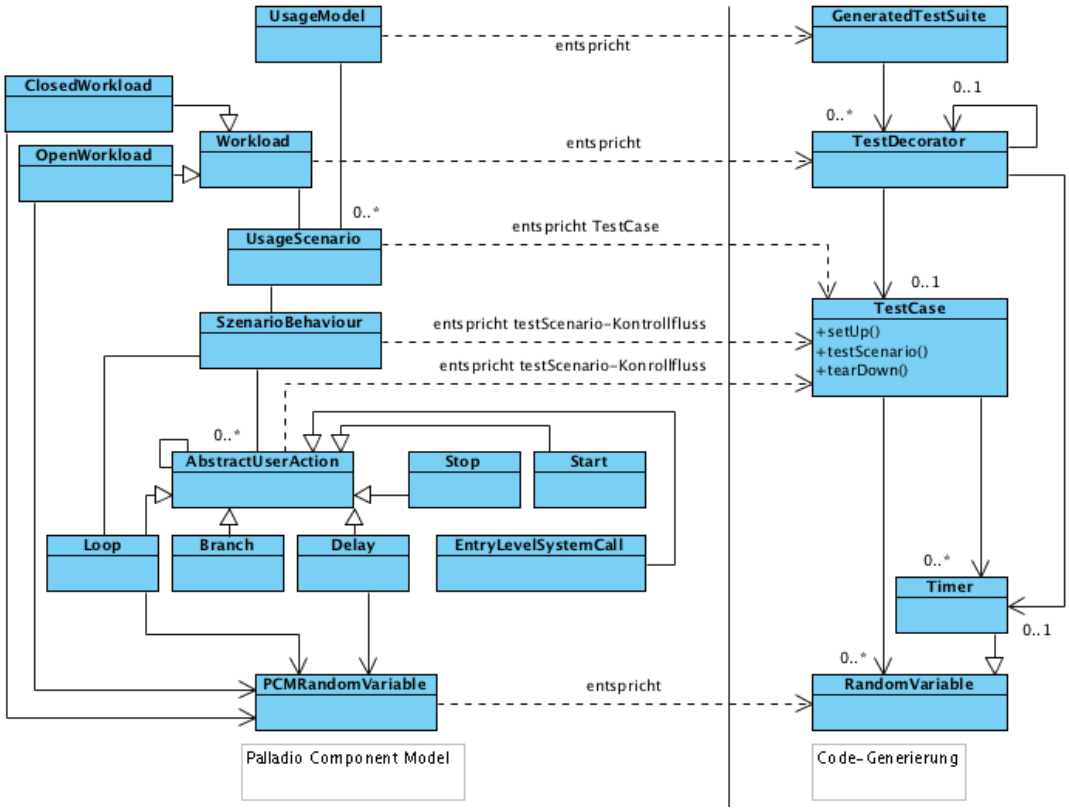


Abbildung 16: Vereinfachte Gesamtübersicht der zu erzeugenden Elemente
Quelle: eigene Darstellung

einen Testfall. Somit sollen die Abläufe in unterschiedlichen modellierten UsageScenarios gleichzeitig anstatt nacheinander stattfinden. Eine Reihe von Argumenten spricht für diese Interpretation:

- Im Falle eines Ablaufs der Szenarien nacheinander folgte als nächstes Problem direkt die Frage nach der Reihenfolge, in der die Szenarien durchlaufen werden sollen. Diese lässt sich im UsageModel nicht konkret angeben.

- Unterschiedliche Use Cases können Kreuzeffekte für die Systembelastung darstellen. Oft finden Aktionen unterschiedlicher Use Cases auch gleichzeitig statt. Ein Performance-Test soll das System auf realistische starke Belastungen hin überprüfen. Daher ist eine Berücksichtigung mehrerer Use Cases zugleich von zentraler Bedeutung für die Aussagekraft des Performance-Tests.

- Wäre für Szenarien innerhalb eines UsageModels die voneinander getrennte Betrachtung vorgesehen, so müsste eine Belastungssituation vollständig durch ein Szenario definierbar sein. Innerhalb einer Belastungssituation können aber unter-

schiedliche Akteurgruppen beteiligt sein, zum Beispiel Kunden und Auftragsbearbeiter. Diesen Akteurgruppen liegen aber unterschiedliche Aktionsabläufe und möglicherweise auch unterschiedliche Workloads zugrunde. Dies lässt sich innerhalb eines einzelnen UsageSzenarios gar nicht abbilden.

Die Entscheidung für einen gleichzeitigen Ablauf aller Szenarien bringt jedoch eine neue Problematik mit sich: Weder JUnit noch JUnitPerf bieten eine TestSuite-Klasse, deren Kinder parallel in verschiedenen Threads laufen. Die JUnit-eigene Klasse TestSuite bietet ausschließlich die Möglichkeit des Startens eines zugewiesenen Tests nach Abschluss des vorherigen Tests. Die in Kapitel 2.2.3 beschriebene JUnitPerf-eigene Klasse LoadTest dagegen ermöglicht das gleichzeitige Starten mehrerer Tests. Jedoch sind dies alles Instanzen des gleichen Tests, während die unterschiedlichen Szenarien innerhalb eines UsageModels von unterschiedlichen Tests abgebildet sein sollen. Es bedarf also der eigenen Entwicklung einer Java-Klasse, die die TestSuite-Eigenschaft des Sammelns unterschiedlicher Tests mit der LoadTest-Eigenschaft des gleichzeitigen Ablaufs vereint, wie in Abbildung 17 verdeutlicht. Dies wird durch die Klasse “SimultaneousLoadTestSuite” realisiert. Die Klasse implementiert das JUnit-Interface “Test”. Wie bei einer TestSuite lassen sich der SimultaneousLoadTestSuite untergeordnete Tests hinzufügen. Abbildung 18 verdeutlicht die Interpretation von UsageScenarios und die Umsetzung durch das entwickelte Klassenkonzept.

Wie in Kapitel 2.1 bereits beschrieben, besitzt jedes UsageScenario jeweils einen offenen oder geschlossenen Workload. Der vom TestDecorator umschlossene Test bildet das Scenariobehavior des Szenarios ab. In derselben Frequenz, in der die definierten Workloads die SzenarioBehaviors starten, sollen die TestDecorators die ihnen zugewiesenen Tests starten. Kapitel 3.2.2 befasst sich genauer mit diesen Blättern in der Baumstruktur, den generierten TestCases.

Der offene Workload lässt sich durch einen einfachen LoadTest realisieren. Die Berücksichtigung der Ankunftszeit zwischen den einzelnen Akteuren gelingt durch die Angabe eines Timers für die Verzögerung, näher beschrieben in Kapitel 3.2.3.

Die Realisierung eines geschlossenen Workloads dagegen bedarf erneut der Entwicklung einer eigenen Klasse. Zumindest die hier benötigte Parallelität der Akteure lässt sich mit einem LoadTest erzeugen. Im Gegensatz zum offenen Workload beginnen im geschlossenen Workload sämtliche Akteure zugleich. Entsprechend muss der Verzögerungswert des LoadTests 0 betragen. Eine Instanz des dem LoadTest zugewie-

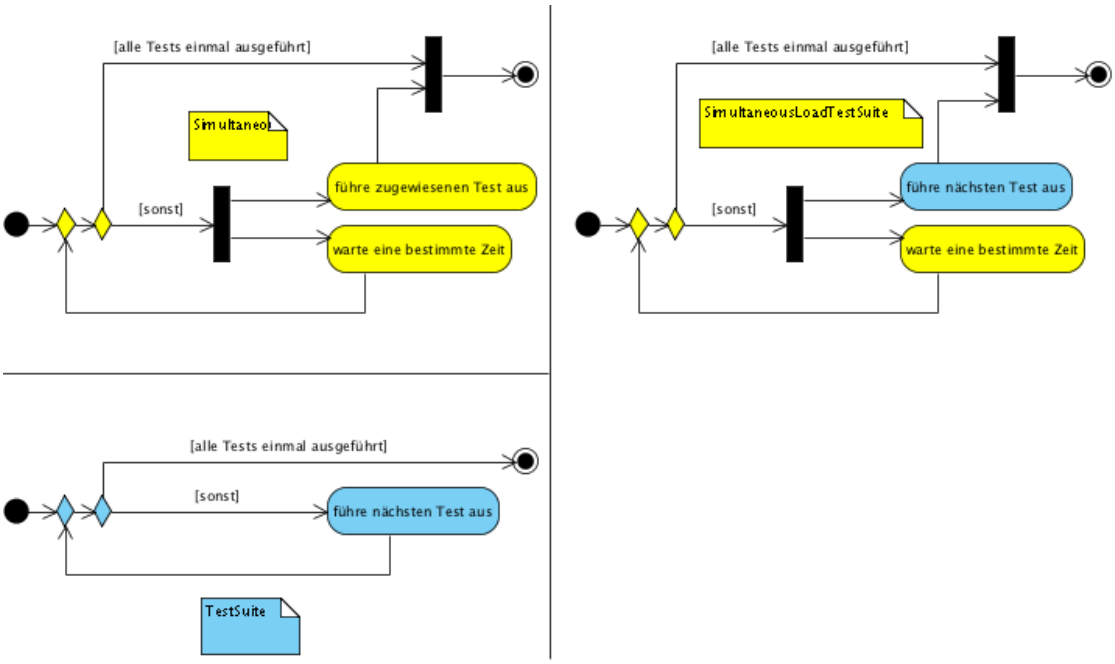


Abbildung 17: Die Arbeitsweise der SimultaneousLoadTestSuite
Quelle: eigene Darstellung

senen Tests repräsentiert nun also einen Akteur. Dieser Test muss also ein weiterer TestDecorator sein, der einen TestCase zur Abbildung des ScenarioBehaviors mit Zwischenpausen wiederholend ausführt (siehe Abbildung 19). Auch dieser TestDecorator muss zunächst selbst entwickelt werden. Zwar kann der TestDecorator RepeatedTest von JUnit einen Test wiederholt starten, allerdings ohne Zwischenpausen. Dies schafft die vom RepeatedTest inspirierte selbstentwickelte Klasse “ThinkTimeRepeatedTest”.

Somit ist festgelegt, welche TestDecorators der SimultaneousLoadTestSuite zugewiesen werden. Abbildung 20 schafft eine Übersicht.

Gewünschte Anfangs- und Endzustände der TestSuite lassen sich unter Umständen durch das Ausführen bereits geschriebener Tests erzeugen, wodurch sich Zeit für eine aufwändige SetUp- und TearDown-Definition vereinfachen lässt. Beispielsweise kann ein TestCase für einen OnlineShop vorsehen, dass ein Kunde eine Bestellung aufgibt. Eine nötige Vorraussetzung für den entsprechenden Test ist, dass der Kunde zum Zeitpunkt der Bestellung im Online-Shop auch eingeloggt ist. Da das Ein- und Ausloggen sowieso ebenfalls getestet werden sollte, existiert vielleicht ein Test für das Einloggen, dessen Endzustand den Anfangszustand des Tests für die Bestellung schafft. Vorbereitende oder nachbereitende Tests können natürlich nicht auch parallel zu den im Fokus

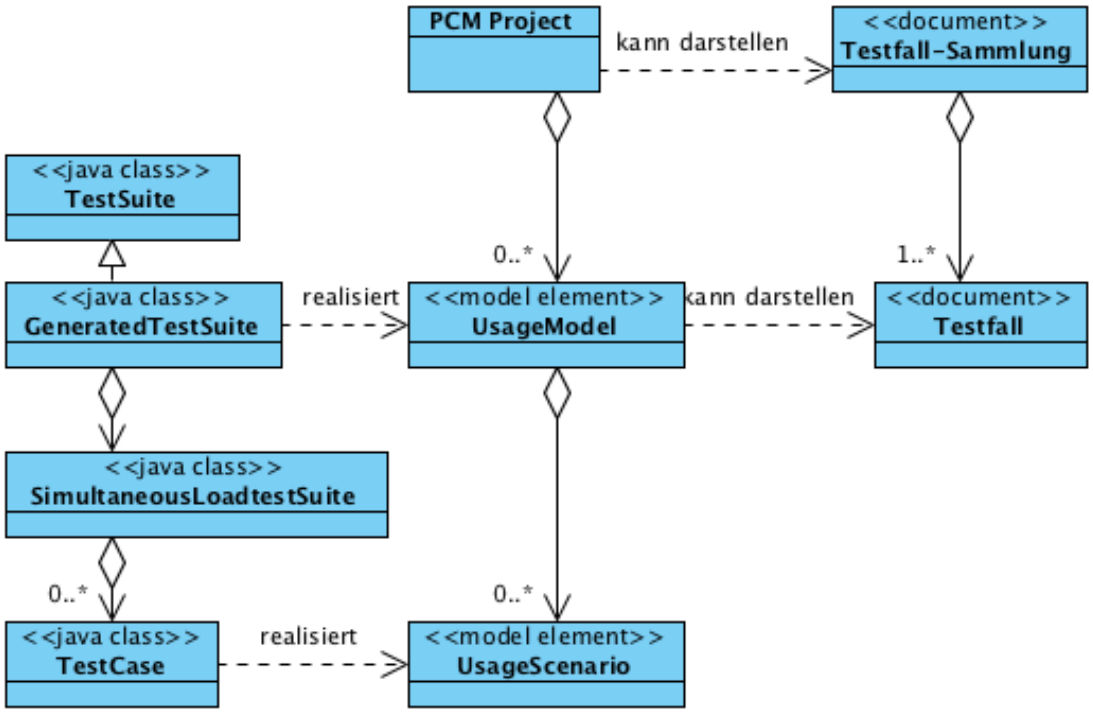


Abbildung 18: Interpretation von UsageScenarios und die Umsetzung durch das entworfene Klassenkonzept
Quelle: eigene Darstellung

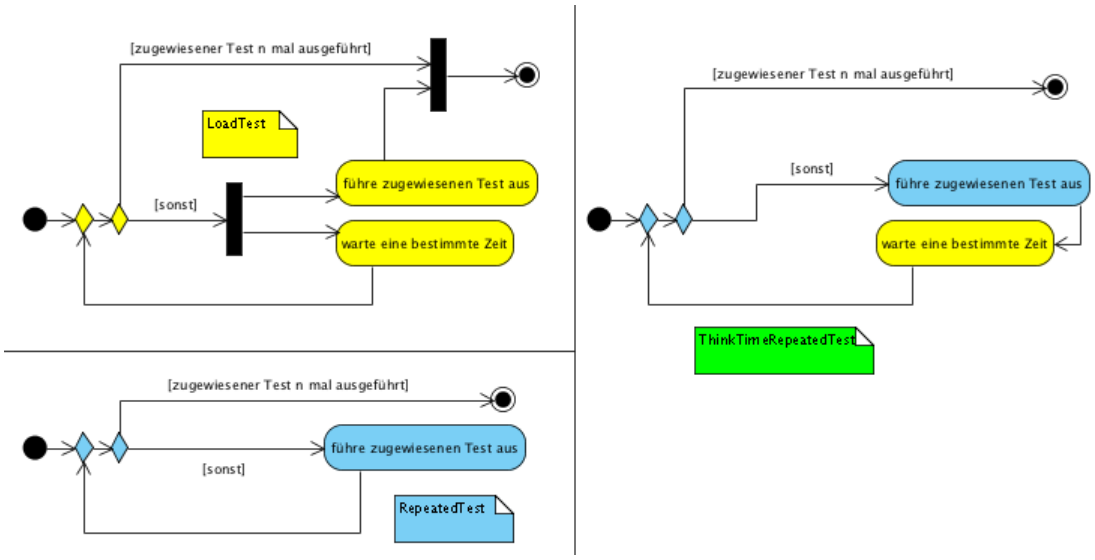


Abbildung 19: Der Arbeitsablauf des selbstentwickelten Decorators ThinkTimeRepeatedTest
Quelle: eigene Darstellung

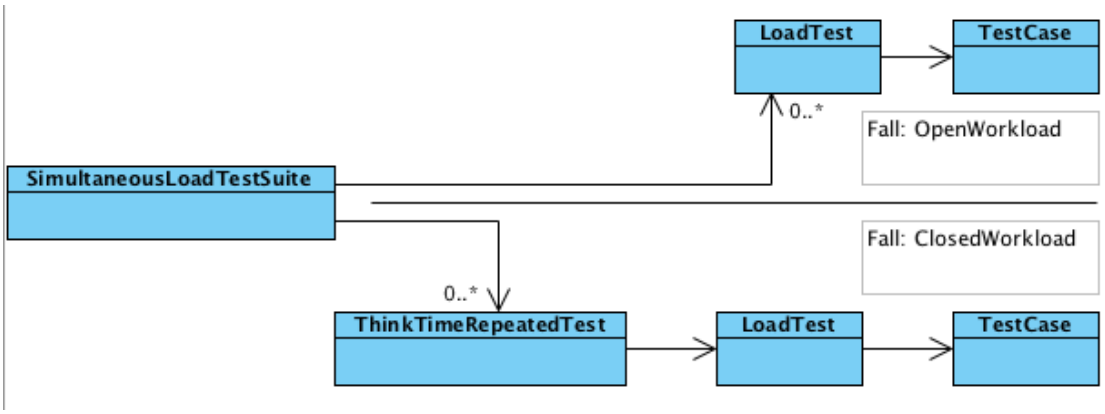


Abbildung 20: Die Umsetzung der Workloads mit TestDecorators
Quelle: eigene Darstellung

stehenden Tests durchgeführt werden. Es muss sichergestellt werden, dass diese vorher beendet sind bzw. erst anschließend starten. Die oberste Wurzel der Teststruktur kann also nicht die gebaute SimultaneousLoadTestSuite darstellen, sondern eine TestSuite bestehend aus den Teilen:

- SetUp-Tests
- SimultaneousLoadTestSuite
- TearDown-Tests

Dem Testautor ist entsprechend anzuraten, TestCases für das SetUp und das TearDown nicht im gleichen UsageModel zu definieren wie andere TestCases. Isoliert in einem jeweils eigenen UsageModel lassen sie sich aber genauso auch separat generieren. Genaueres zur späteren Einstellung von Anfangs- und Endzuständen im generierten Java-Code findet sich in Kapitel 3.2.4.

Nach der Betrachtung des Palladio-Beispiel-Projekts “CoCoME”, auf das in Kapitel 4.2 genauer eingegangen wird, zeigt sich, dass ein Projekt auch durchaus legitim über mehrere UsageModels verfügen kann. Im erwähnten Beispiel existiert für jeden Use Case ein eigenes Modell. Ist in einem solchen Fall ein Test unter Berücksichtigung mehrerer Use Cases aus unterschiedlichen UsageModels gewünscht, ist der Testautor gezwungen entweder ein neues UsageModel zu bauen oder zunächst für jedes betroffene UsageModel jeweils die Generierung zu starten und die einzelnen generierten TestSuites manuell zusammen zu fügen.

Ist bisher nun also die Abbildung eines UsageModels, seiner UsageScenarios und deren Workloads in einer TestSuite beschrieben, folgt in Kapitel 3.2.2 die Umsetzung von ScenarioBehaviors in TestCase-Klassen.

3.2.2 Generierung der TestCases

Nachdem die generierte TestSuite mit Hilfe von Decorators die Workflows der einzelnen UsageScenarios in einem UsageModel abbildet, muss noch der zweite wichtige Teil eines UsageScenarios abgebildet werden. Ein UsageScenario entspricht einem Use Case. Durch den Workload wird die Anzahl der Nutzer und deren Ankunftsrate für den Use Case definiert. Das ScenarioBehavior dagegen beschreibt den Kontrollfluss der Nutzer im Use Case. Zur Abbildung des ScenarioBehaviors in jedem UsageScenario wird zusätzlich zu jedem ScenarioBehavior eine weitere Klasse generiert. Generierte Klassen dieser Art erben von der JUnit-Klasse TestCase, was der TestSuite ermöglicht, sie in ihre unterliegenden TestDecorators einzubinden, wie in Kapitel 3.2.1 beschrieben wurde.

Die generierten TestCase-Klassen müssen sich an die in Kapitel 2.2.2 beschriebenen Syntax-Regeln für JUnit halten. Entsprechend werden die Methoden setUp und tearDown implementiert, die für die Vorbereitung bzw. Nachbereitung des Anfangs- bzw. Endzustands eines Testfalls sorgen. Des weiteren enthält eine TestCase-Klasse die generierte Methode testScenario, die den Kontrollfluss des ScenarioBehaviors abbildet.

ScenarioBehaviors können die folgenden Kontrollfluss-Elemente beinhalten, deren Abbildung in der Methode wichtig ist:

- Start: Der Beginn des Kontrollflusses.
- Stop: Das Ende des Kontrollflusses.
- Delay: Lässt eine Pause abwarten, die eine bestimmte Zeit dauert.
- Loop: Beinhaltet ein weiteres ScenarioBehavior in Kontrollfluss-Form, das für eine bestimmte Anzahl wiederholt wird.
- Branch: Beinhaltet mehrere ScenarioBehaviors in Kontrollfluss-Form, von denen eines zufällig ausgeführt wird. Die Wahl ergibt sich aus definierten Wahrscheinlichkeitswerten für jede Alternative. Entsprechend ergibt die Summe aller Wahrscheinlichkeitswerte 1.

- EntryLevelSystemCall: Der Aufruf einer Methode.

Ist das Start-Element gefunden, können die Nachfolger Schritt für Schritt in Code abgebildet werden. Ein Delay lässt sich hierbei noch sehr einfach durch den „sleep“-Befehl des derzeit laufenden Threads realisieren. Wie aus Kapitel 2.2.3 hervorgeht, erzeugen LoadTests von JUnitPerf für jeden Durchlauf des zugewiesenen Tests einen eigenen Thread. Auf gleiche Weise arbeitet der selbstentwickelte ThinkTimeRepeatedTest. Somit lässt sich wieder sicherstellen, dass der Schlaf-Befehl für den einzelnen Kontrollfluss keine Auswirkungen auf die nebenläufigen Prozesse hat.

Das Loop-Element lässt sich entsprechend durch eine Ausführungs-Schleife im Code übertragen. Bevor der Generierungsalgorithmus anschließend mit dem nächsten Element fortfahren kann, muss zunächst der Algorithmus für das innere ScenarioBehavior des Loop-Elements ausgeführt und vollständig abgeschlossen sein.

Herausforderung beim Abbilden des Branch-Elements im Code ist, dass dieses nicht zwingend genau 2 Alternativen beinhalten muss. Diese ließen sich einfach mit einem If-Else-Statement übertragen, wie es in den meisten imperativen Programmiersprachen vorhanden ist. Stattdessen muss die Auswahl undefiniert vieler Alternativen gewährleistet sein. Voraussetzung, die für die Realisierung angenommen wird, ist, dass die Summe der Wahrscheinlichkeiten dieser Alternativen 1 beträgt. Zur Realisierung muss zunächst ein Zufallswert zwischen 0 und 1 initialisiert werden. Zusätzlich bedarf es eines Zählwerts, der bei 0 anfängt. Für jede Alternative wird dem Zählwert die Eintrittswahrscheinlichkeit der Alternative aufsummiert. Ist der Zählwert höher als der Zufallswert, wird die entsprechende Alternative ausgewählt. Würde der Algorithmus in derselben Form für alle folgenden Alternativen weitergeführt, so gälte für diese auch die Bedingung zur Auswahl. Daher muss nach einmaliger Auswahl einer Alternative der Zählwert um mindestens 1 gesenkt werden. Da die Summe aller Wahrscheinlichkeitswerte 1 beträgt, wird anschließend garantiert keine weitere Alternative mehr ausgewählt. Algorithmus 2 verdeutlicht den Algorithmus zur Alternativen-Auswahl in Pseudocode.

Da das End-Element keiner Implementierung bedarf, sondern lediglich der Beendigung des Generierungs-Algorithmus, bleibt als letztes zu übertragendes Element der EntryLevelSystemCall. Die Abbildung von Elementen diesen Typs bringt vier zentrale Probleme mit sich:

Algorithmus 2 : Die logische Umsetzung des Branch-Elements in Pseudocode

```
Data : List of Element Tuple, consisting of alternative and its probability, tuples
rnd ← random() ;
count ← 0 ;
forall tuple in tuples do
    count ← count + tuple.probability ;
    if rnd ≤ count then
        execute(tuple.alternative) ;
        count ← count − 1
    end
end
```

- Die Bezeichnungen der Methoden-Namen im Palladio-Modell müssen als übereinstimmend mit den Methoden-Namen im tatsächlichen Software-Projekt angenommen werden, um eine fehlerfreie automatische Code-Generierung zu garantieren. In der Realität jedoch kommt es im Entwicklungs-Prozess oft und schnell zu Veränderungen von Methoden-Namen. Die Automatisierung muss daher eine komfortable Anpassung der Methoden-Namen im Palladio-Modell an die Methoden-Namen im Software-Projekt ermöglichen. Näheres hierzu wird in Kapitel 3.2.4 beschrieben.

- Methoden können Parameter besitzen. Diese müssen ebenfalls im Code abgebildet werden. [Brüseke *et al.* , 2011] beschreibt ein Konzept zur Parametermodellierung in Palladiomodellen zur späteren Performance-Blame-Analyse. In UsageModels lassen sich an EntryLevelSystemCalls sogenannte VariableUsages anbringen, die jeweils auf einen bestimmten Parameter der betrachteten Methode referenzieren. Diese VariableUsages lassen sich durch VariableCharacterisations spezifizieren, die Informationen über den Parameter-Wert wie auch über andere Parameter-Charakteristika beinhalten können. Beinhaltet ein EntryLevelSystemCall für einen Parameter genau eine VariableUsage mit genau einer VariableCharacterisation, so wird der angegebene Wert als später editierbare Voreinstellung im generierten Code verwendet. Näheres hierzu findet sich in Kapitel 3.2.4.

- Typen von Parametern können auch in Palladio eigens definierte sein. Kapitel 3.2.3 befasst sich mit dieser Problematik näher.

- Neben der Kenntnis über die Methoden ist auch der Zugriff auf die entsprechenden Komponenten von Nöten, die die entsprechenden User-Interfaces implementieren. Das Generierungskonzept teilt den Zugriff auf die Komponenten in drei

Teilschritte:

- Definition
- Referenzierung/Initialisierung
- Methodenaufruf

Als statisch typisierte Sprache sieht Java eine eindeutige Typdefinition von Variablen vor. Zur Typdefinition kann entweder die Bezeichnung des nach außen bereitgestellten Interfaces oder die Bezeichnung der Komponente dienen, die das entsprechende Interface implementiert. Letztere offenbart keinerlei Vorteile gegenüber der ersten Alternative. Die Automatisierung typisiert Zeiger für Methoden-Aufrufe nach den entsprechenden Interfaces, da dies erstens leichter umzusetzen ist und zweitens auch für den Testautor den Code leichter nachvollziehbar macht. Darüber hinaus ist anzunehmen, dass Bezeichnungen von Interfaces eher die Entwicklung überdauern als Komponenten-Namen, da die Aufgabe von Interfaces schließlich darin besteht, den Zugriff von außen zu gestalten und die Austauschbarkeit von Komponenten zu ermöglichen.

Nachdem der Typ eines Zeigers festgelegt ist, bedarf es in einem nächsten Schritt der Referenzierung auf eine Instanz bzw. möglicherweise ihre Initialisierung. Die Initialisierung bleibt zwangsläufig Aufgabe des Testautors, da die Form der notwendigen Initialisierung nicht aus den Palladio-Modelldaten hervor geht. Beispielsweise ist nicht bekannt, ob die entsprechenden Komponenten bereits initialisiert sind, ob ein bestimmter Konstruktor dazu aufgerufen werden muss oder eine bestimmte Methode einer Factory.

Da ein Methodenaufruf innerhalb eines ScenarioBehaviors mehrfach ausgeführt werden kann, empfiehlt sich die einmalige Referenzierung zu Beginn der Ausführung statt der erneuten Referenzierung vor jedem Methodenaufruf. Hierzu bietet sich die Methode setUp als geeigneter Ort für Initialisierungen an. Somit sind alle Initialisierungen an einem Ort zur Übersicht für den Testautor gesammelt.

Anders als die Definition und die Referenzierung findet der Methodenaufrauf selbst im Kontrollfluss statt. Die Rückgabe-Parameter der in ScenarioBehaviors aufgerufenen Methoden - sofern diese überhaupt über welche verfügen - werden dabei vom generierten Code nicht weiter berücksichtigt. Kapitel 3.2.3 befasst sich genauer mit der

Umsetzung von stochastischen Ausdrücken und ihren Limitationen.

3.2.3 Umsetzung von stochastischen Ausdrücken

Innerhalb eines UsageScenarios treten eine Reihe von Zahlenwerten auf, die wichtige Informationen für den Ablauf darstellen. Insgesamt lassen sich diese Werte semantisch in fünf Gruppen einteilen:

- Zeitliche Dauer (RandomVariable mit Zahlenwert)
- Häufigkeit (RandomVariable mit Zahlenwert)
- Wahrscheinlichkeit (Double-Wert (Gleitkommazahl))
- Personenanzahl (Integer-Wert (ganze Zahl))
- Primitive Parameterwerte (RandomVariable mit allen in Palladio möglichen Typen)

Je nach Art lassen sich die Werte auf unterschiedliche Weise modellieren und besitzen innerhalb von Palladio je nach Semantik unterschiedliche Wertebereiche. Die folgenden Abschnitte beschreiben Schritt für Schritt die einzelnen Wert-Typen und ihre Wertebereiche.

Werte für die zeitliche Dauer beschreiben Ankunfts- und Wartezeiten einzelner simulierter Nutzer. So benötigen OpenWorkloads beispielsweise eine Definition für die jeweilige Dauer zwischen den Ankünften zweier Nutzer. ClosedWorkloads dagegen benötigen die Definition der jeweiligen Wartezeit vor der erneuten Ausführung. Das Delay-Element innerhalb eines ScenarioBehaviors besitzt ebenfalls einen Wert für die zeitliche Dauer der Verzögerung. Innerhalb von Palladio besitzen diese Zahlenwerte zunächst keine feste Einheitsangabe zur zeitlichen Größenordnung. Die entwickelte Automatisierung interpretiert die Zahlenwerte als Sekunden, da diese für Wartezeiten zwischen Aktionen von Computernutzern als intuitive Größe erscheinen.

Neben der Wartezeit benötigen ClosedWorkloads auch eine Angabe zur Anzahl der teilnehmenden Personen. Für diesen Wert erwartet Palladio einen festen Dezimalwert.

Häufigkeit spielt bei Loops eine Rolle. Hier wird die Anzahl der Iterationen festgelegt, die eine solche Schleife durchlaufen soll. Dagegen treten Wahrscheinlichkeitswerte

nur bei Branches als Eintrittswahrscheinlichkeit unterschiedlicher Alternativen auf.

Diese Wahrscheinlichkeitswerte betragen einen Wert zwischen 0 und 1. Die Summe aller ergibt 1. Entsprechend müssen diese Werte eindeutig sein und werden als Fließkommazahlen modelliert. Anders verhält sich dies bei den Werten für zeitliche Dauer, Häufigkeit und primitive Parameterwerte. Diese müssen lediglich Ergebnis eines stochastischen Ausdrucks sein. Für zeitliche Dauer und Häufigkeit kommen hier semantisch nur numerische Werte in Frage. Primitive Parameterwerte dagegen sind nicht zwangsläufig auf numerische Typen beschränkt, sondern können theoretisch auch Zeichen oder Zeichenketten sein. Dies alles wird ermöglicht durch die Verwendung des „stoex“ Metamodells, auf das Palladio zurückgreift. Die Zahlenwerte werden als sogenannte RandomVariables definiert. Hierbei stellt der Element-Typ “RandomVariable“ die Brücke vom Palladio- zum stoex-Metamodell dar. Abbildung 21 und Abbildung 22 veranschaulichen Einsatz und Aufbau des Metamodells.

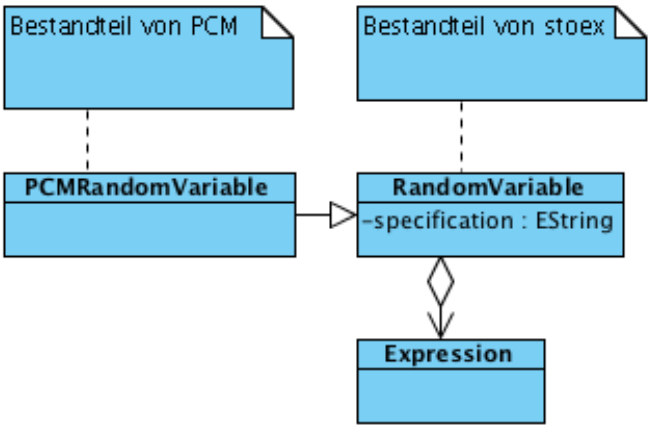


Abbildung 21: Die Verwendung des Metamodells stoex in Palladio
Quelle: eigene Darstellung

Für jede vorliegende RandomVariable generiert die Automatisierung eine eigene Klasse und referenziert im Code an den entsprechenden Stellen auf diese. Diese Klassen erben von der selbstentwickelten Klasse ProbabilityFunction oder im Falle von zeitlichen Angaben von der selbstentwickelten Klasse AbstractTimer.

Alle erwähnten generierten Klassen besitzen eine Methode namens “calcValue”. Dessen Return-Statement stellt den stochastischen Ausdruck der RandomVariable dar. Diese stochastischen Ausdrücke können sehr komplex modelliert sein. Unter anderem

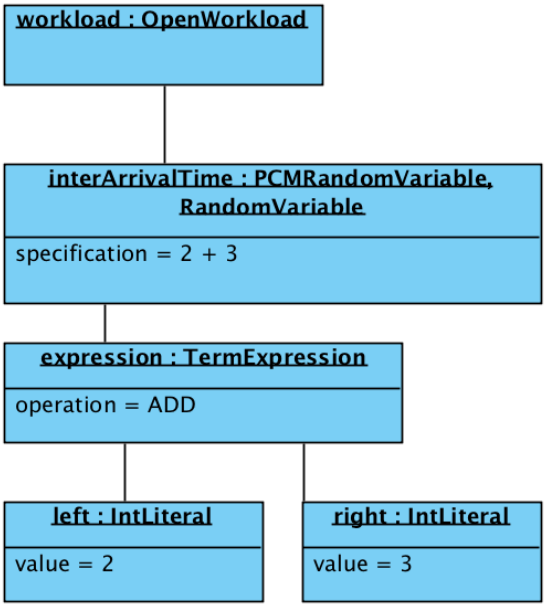


Abbildung 22: Beispiel für den Aufbau eines stochastischen Ausdrucks
Quelle: eigene Darstellung

bestehen diese aus konstanten Werten primitiven Typs. Diese lassen sich durch eine Reihe von Operationen verbinden. Zum einen sind dies die Operationen für Addition, Subtraktion, Multiplikation, Division und Modulo-Berechnung. Darüber hinaus lassen sich auch Elemente durch Größen-Vergleiche verbinden. Das Ergebnis eines solchen Vergleichs ein boolescher Wert . Abhängig von booleschen Ergebnissen lassen sich auch Elemente als Alternativen verbinden. Auf diese Weise lassen sich „If-Else“-Strukturen modellieren. Boolesche Werte lassen sich zudem logisch miteinander verknüpfen. Strukturiert ist ein stochastischer Ausdruck als Baum, wie in Abbildung 22 dargestellt.

Weiterer wichtiger Bestandteil von stochastischen Ausdrücken sind Wahrscheinlichkeitsfunktionen. Diese werden in Palladio unter Einbezug des weiteren Metamodells “probfunction” modelliert. Wahrscheinlichkeitsfunktionen werden in Palladio in zwei Typen unterschieden:

- Wahrscheinlichkeitsfunktion (engl.: Probability Mass Function, kurz: PMF)
- Wahrscheinlichkeitsdichtefunktion (engl.: Probability Density Function, kurz: PDF)

[Wunsch & Schreiber, 1992] beschreibt den Zusammenhang zwischen Wahrscheinlichkeit und Dichte. Sei Ω die Gesamtheit aller möglichen Ereignisse und $A \subseteq \Omega$ ein

einzelnes Ereignis, so ist $P(A)$ die Wahrscheinlichkeit für das Eintreten des Ereignisses. Eine Wahrscheinlichkeitsfunktion lässt sich folgendermaßen definieren:

$$\begin{aligned} P : A &\mapsto P(A) \\ P(A) &\geq 0 \\ P(\Omega) &= 1 \\ P(\bigcup_{i=1}^{\infty} A_i) &= \sum_{i=1}^{\infty} P(A_i) \end{aligned}$$

Für eine Dichtefunktion bedarf es der Abbildung der einzelnen Ereignisse auf numerische Werte. Diesem Zweck dient die sogenannte zufällige Veränderliche $X : \Omega \mapsto \mathbb{R}$. Eine Wahrscheinlichkeitsdichtefunktion wird verwendet, um die Wahrscheinlichkeit zu errechnen, mit der die zufällige Veränderliche einen Wert im Intervall $[\xi, \xi'[$ hat. Eine Dichtefunktion $f(x)$ wird laut [Wunsch & Schreiber, 1992] so definiert, dass diese Wahrscheinlichkeit ihr Integral über das Intervall $[\xi, \xi'[$ ist.

$$\begin{aligned} f(x) &: \mathbb{R} \mapsto \mathbb{R}^+ \\ P(X \in [\xi, \xi'[) &= \int_{\xi}^{\xi'} f_x(x)dx \end{aligned}$$

Um die Benutzung dieser beiden Funktionen im Code zu ermöglichen, besitzt die selbstentwickelte abstrakte Klasse ProbabilityFunction entsprechende Methoden. Wie in Kapitel 2.2.3 erwähnt, beinhaltet das Framework JUnitPerf das Interface Timer, um zeitliche Abstände für seine TestDecorators zu definieren. Dazu dient die selbstentwickelte abstrakte Methode AbstractTimer, die auf ProbabilityFunction aufbaut und das Interface Timer implementiert. Die Algorithmen 3 und 4 verdeutlichen die Umsetzung der beiden Wahrscheinlichkeitsfunktionen im generierten Code anhand von Pseudocode. Hierbei sei zu erwähnen, dass das entwickelte Konzept zur PDF auf der PMF aufbaut.

Neben PDF und PMF sieht Palladio auch die Benutzung von vordefinierten Funktionen vor. Diese Funktionen, zum Teil ebenfalls Verteilungen (z. B. Exponentialverteilung), werden nicht von der Automatisierung unterstützt, da ihre Implementierung den Rahmen dieser Arbeit sprengen würde. Als zukünftige Erweiterung wäre die Anbindung

Algorithmus 3 : Die Umsetzung der PMF in Pseudocode

```
Data : List of Element Tuple, consisting of value and probability, tuples
Result : result
rnd ← random() ;
count ← 0 ;
result ← 0 ;
resultFound ← false ;
forall tuple in tuples do
    count ← count + tuple.probability ;
    if rnd ≤ count ∧ ¬resultFound then
        resultFound ← true ;
        result ← tuple.value ;
    end
end
return result ;
```

Algorithmus 4 : Die Umsetzung der PDF in Pseudocode

```
Data : List of Element Tuple, consisting of value and probability, tuples
Result : result
domainMax ← calcPMF(tuples) ;
domainMin ← domainMax ;
for i = 0 to tuples.length do
    if domainMax == tuple[i].value then
        if i > 0 then
            domainMin ← tuple[i − 1].value ;
        end
        else
            domainMin ← 0 ;
        end
    end
end
result ← domainMin + ((domainMax − domainMin) ∗ random()) ;
return result ;
```

einer Bibliothek mit gängigen Verteilungsfunktionen denkbar, um den Mangel zu beheben. Notfalls kann der Nutzer spezielle angestrebte Verteilungen als PDF- oder PMF-Element umformen, um sich diesen anzunähern. Weitere nicht unterstützte Features der stochastischen Ausdrücke in Palladio sind das Einbauen von Rückgabeparametern und das Angeben von Instanzen selbstdefinierter Typen in Palladio. Sollte der Nutzer auf diese Features angewiesen sein, besitzt er die Möglichkeit nach der Generierung den erstellten Code nach seinen Wünschen anzupassen.

3.2.4 Einstellungsmöglichkeiten des Nutzers

Nicht alle Informationen, die zum Erstellen lauffähiger Testskripte notwendig sind, können aus Palladio-Modellen gewonnen werden. Wichtige Informationslücken sind:

- Tatsächliche Namen von Interfaces und Methoden in der Implementierung
- Package-Struktur der Implementierung
- Notwendige SetUp- und TearDown-Aktionen
- Dauer der Testdurchläufe
- Komponenten-Initialisierung/Referenzierung

Da einige notwendige Informationen zur Generierung fehlen, müssen also einige Stellen im generierten Code leer bleiben. Diese Stellen bedürfen einer manuellen Ausfüllung durch den Testautor. Um die Arbeit für den Testautor möglichst leicht zu machen, sollten die manuell zu füllenden Stellen nahe beieinander liegen und gekennzeichnet sein. Dies erleichtert das Zurechtfinden im fremden Code.

Wie in Kapitel 2.3.2 bereits erwähnt, hält Xpand die Möglichkeit bereit für den Nutzer vorgesehene Bereiche im generierten Code zu definieren. Diese Bereiche sind im Falle einer erneuten Code-Generierung vor Überschreibung gesichert. Das entwickelte Werkzeug sieht die Verwendung solcher schreibgeschützten Bereiche für die manuell zu füllenden Lücken der Generierung vor. Die manuelle Vervollständigung ist in folgenden generierten Klassen notwendig:

- GeneratedTestSuite (Die Wurzel aller generierten Klassen, beschrieben in Kapitel 3.2.1)

- TestCase-Klassen (Die Abbildungen einzelner ScenarioBehaviors, beschrieben in Kapitel 3.2.2)
- Sammelklasse für Durchlaufdauer (beschrieben im weiteren Verlauf)
- Sammelklasse für Eingabeparameter (beschrieben im weiteren Verlauf)

Für sämtliche dieser Klassen ist zu Beginn ein Bereich für manuelle Importe benötigter zusätzlicher Java-Klassen- und Packages vorhanden. Unter anderem betrifft dies auch den Import von Elementen der gegebenen Software-Implementierung.

In der GeneratedTestSuite kann der Testautor den Einbezug von Tests festlegen, die sich für ein nötiges SetUp oder TearDown vor bzw. nach dem eigentlichen Performance-Test eignen (siehe Kapitel 3.2.1). Genauso ist in den jeweiligen TestCase-Klassen die manuelle Angabe der SetUp- und TearDown-Aktionen notwendig. Wie in Kapitel 3.2.2 beschrieben, betrifft dies besonders die Referenzierung von Variablen auf die korrekten Komponenten bzw. die eventuelle Initialisierung dieser, damit im anschließenden Durchlauf des Kontrollflusses der Methodenaufruf auf diesen möglich ist.

Während einzelne TestCase-Klassen ScenarioBehaviors im Palladio-Modell abbilden, werden die dazugehörigen Workloads durch TestDecorators abgebildet, die die umschlossenen Tests wiederholt abrufen. Innerhalb von UsageScenarios beschreiben Workloads nicht die Dauer ihrer Arbeit. Um die Dauer von Performance-Tests zu definieren, bedarf es im Falle eines OpenWorkloads eines Wertes für die Anzahl der durchlaufenden Nutzer, im Falle eines ClosedWorkloads eines Wertes für die Anzahl der zu durchlaufenden Iterationen. Alle Werte dieses Typs sind in der Klasse “TestDuration” gesammelt und einstellbar. Somit hat der Testautor an dieser Stelle die Möglichkeit die Dauer des Performance-Tests zu kalibrieren.

Wie in Kapitel 3.2.2 beschrieben, ermöglicht Palladio mittels VariableUsages Werte für Eingabeparameter festzulegen. Eingabe-Sets sind ein zentrales Thema von Software-Tests. Diese lassen sich in der Klasse “ParameterInput” angeben. Standardmäßig ist dort die im Palladio-Modell angegebene RandomVariable Grundlage für den Eingabe-Wert.

Auch wenn die manuellen Eingaben im Code in den dazu vorgesehenen Bereichen grundsätzlich sicher vor Überschreibung sind, ist dennoch die weitestgehende Auslagerung selbstgeschriebener Codefragmente aus dem generierten Code zu empfehlen (vgl.

[Becker, 2010a]). Die Anbindung externer Codefragmente wird durch die erwähnten schreibgeschützten Import-Bereiche unterstützt.

Die bisher beschriebenen manuellen Anpassungen geschehen anschließend an die Generierung im Code. Die Automatisierung ermöglicht für die Anpassung von Interface- und Methodennamen jedoch auch eine erweiterte Eingabemöglichkeit innerhalb des Palladio-Modells. Hierzu wird das Diagramm des Repository-Modells verwendet. Innerhalb der Diagramme lassen sich Notizen an Elemente heften. Diese Notizen werden verwendet, um mittels einer einfachen Sprache notwendige Namensersetzungen durchzuführen. Solche Ersetzungen werden notwendig, wenn die Namen von Interfaces oder Methoden in der Software-Implementierung nicht mit denen im Palladio-Modell übereinstimmen und somit eine korrekte Referenzierung des generierten Codes auf die Software-Implementierung nicht möglich ist. Um innerhalb einer Notiz eine Zeile als Befehl zur Namensersetzung zu markieren, muss diese mit dem Erkennungs-Schlüssel “pcm2junit_correct” beginnen. Anschließend daran folgen jeweils durch einen Doppelpunkt getrennt der im Modell verwendete Name und der im Code zu verwendende Name. Algorithmus 5 definiert die Grammatik der Sprache. Abbildung 23 zeigt ein Beispiel.

Algorithmus 5 : Die formale Grammatik zur Ersetzung von Bezeichnern

text ::= (line “;”)*
line ::= “pcm2junit_correct” “:“ palladioBezeichner “:“ javaBezeichner
palladioBezeichner ::= string
javaBezeichner ::= string

3.3 Implementierung des Plugins

Die entwickelte Automatisierung zur Code-Generierung basiert auf einem MWE-Workflow. Dieser lässt sich durch das entwickelte Eclipse-Plugin PCM2JUnit_Plugin ausführen. Läuft das Plugin auf der aktuellen Eclipse-Instanz, ist das Öffnen einer View möglich, die in Abbildung 24 dargestellt ist. Innerhalb dieser View lassen sich alle wichtigen Optionen einstellen und die Generierung starten.

Wichtige Informationen für die Ausführung sind die Pfade der drei zu verwendenden Palladio-Modell-Dateien der Typen “.usagemodel”, “.system” und

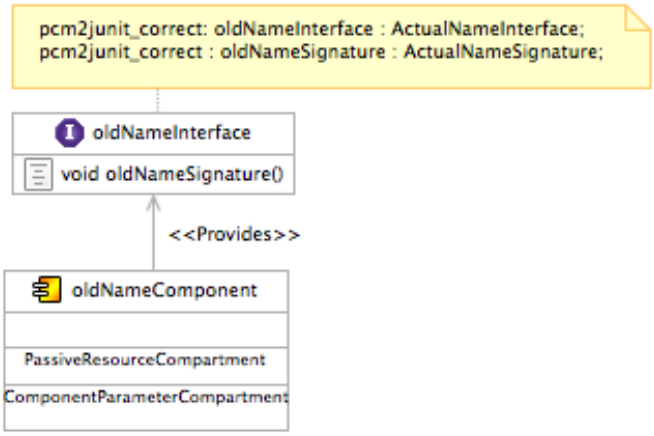


Abbildung 23: Beispiel für eine Namenskorrektur im Repository-Diagramm
Quelle: eigene Darstellung

".repository_diagram". Das Plugin ermöglicht die bequeme Auswahl der Dateipfade durch einen Selection-Dialog, der den aktuellen Workspace in Baum-Form zur Auswahl stellt. Auf dieselbe Weise lässt sich auch der Ausgabepfad definieren. Durch den Button "Quick Configuration…" kann ein Verzeichnis gewählt werden. Beinhaltet dieses Verzeichnis Palladio-Modelle der genannten Typen, wird automatisch eines dieser Modelle ausgewählt. Durch Betätigen des Knopfs "Generate" werden die gesammelten Pfade der Workflow-Engine übergeben und der Befehl zur Ausführung des MWE-Workflows "generator_Plugin.mwe" gegeben. Nach Durchführung der Generierung lässt sich der Vorgang der Generierung im Log auf der Console nachvollziehen. Im Falle eines Abbruchs gibt dieser jedoch keine Fehlermeldung.

Für diesen Fall eignet sich die alternative Durchführung der Codegenerierung über den Befehl "Run As…MWE-Workflow" auf der MWE-Datei "standalone.mwe". Diese ist konsistent zur anderen Workflow-Datei mit dem Unterschied, dass diese die Pfad-Informationen statisch beinhaltet, die manuell in der Datei editiert werden müssen. Abbildung 25 zeigt, an welchen Stellen die Pfade in der Datei editiert werden können. Der direkte Ablauf ohne das Plugin ist zwar für den Anwender weniger komfortabel, ermöglicht jedoch eine informativere Log mit Fehlermeldungen im Falle eines Abbruchs.

Der MWE-Workflow startet eine Reihe von Xpand-Templates auf die geladenen Modelle. Die genaue Strukturierung des Plugin-Projekts wird in Anhang A.3 beschrieben.

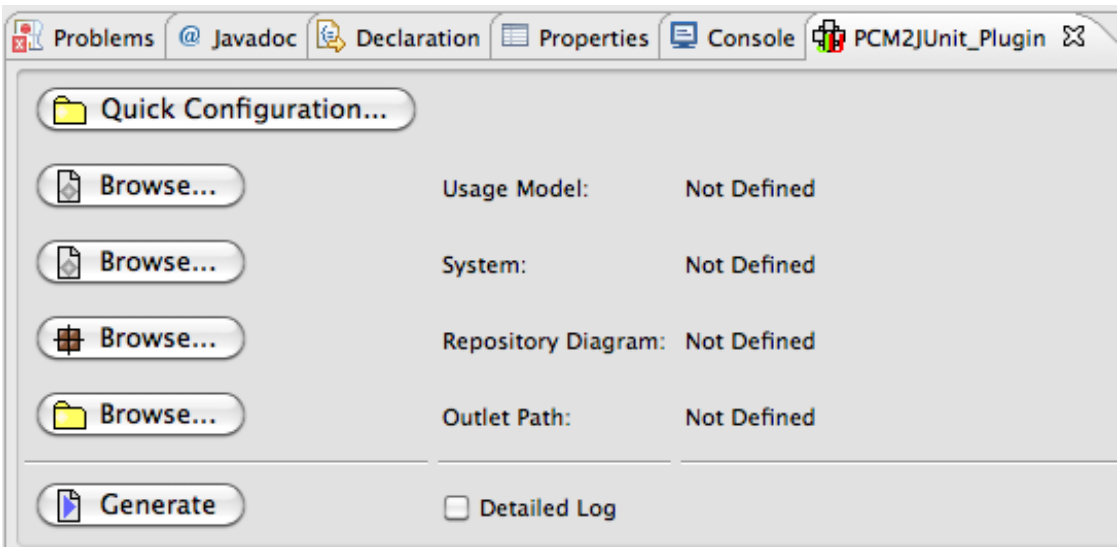


Abbildung 24: Die View des entwickelten Plugins für Eclipse
Quelle: eigene Darstellung

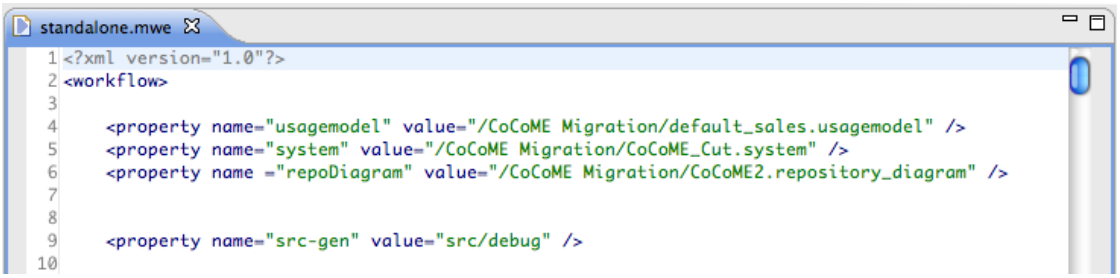


Abbildung 25: Die statische Pfadangabe in einer MWE-Datei
Quelle: eigene Darstellung

4 Evaluierung

Nachdem in Kapitel 3 die Entwicklung der Automatisierung beschrieben wurde, befasst sich Kapitel 4 nun mit der Validierung der entwickelten Lösung. Kapitel 4.1 befasst sich mit den Daten, die zum Testen der Automatisierung verwendet wurden und erklärt, wie die Testergebnisse bezüglich der Korrektheit der Automatisierung zu interpretieren sind. Anschließend erörtert Kapitel 4.2 die Verwendbarkeit der Automatisierung an einem gegebenen Beispiel. Zu guter Letzt greift Kapitel 4.3 die in Kapitel 3.1 erhobenen Anforderungen auf und überprüft die Erfüllung dieser.

4.1 Testmodell-Set

Entscheidend für die Nutzbarkeit der entwickelten Automatisierung ist die vollständige Korrektheit der Code-Generierung. Sämtliche wichtigen Informationen aus den Palladio-Daten müssen korrekt übertragen werden und zu einem lauffähigen Gesamt-Code beitragen. Wegen der hohen Komplexität der Modelle würde ein formaler Beweis für die Korrektheit jedoch den Rahmen dieser Arbeit sprengen. Daher muss für die Validierung des entwickelten Werkzeugs auf Tests zurückgegriffen werden, die keine Korrektheit beweisen, sondern lediglich die Anwesenheit von Fehlern aufzeigen können (vgl. [Dijkstra, 1990]). Es muss ein Testdaten-Set herangezogen werden, das möglichst viele Fehler findet und somit nach deren Beseitigung ausschließt. Das Testdaten-Set sollte auf jeden Fall sämtliche wichtigen Elemente beinhalten, deren Übertragung von der Automatisierung unterstützt wird. Wird das Testdaten-Set in korrekt arbeitsfähiger Form in Java-Code übertragen, zeigt das Set, dass alle Elemente zumindest in einer Ausprägung korrekt verarbeitet werden. Im Code der verwendeten Xpand-Generierung (siehe Kapitel 2.3.2) werden somit alle Zeilen einmal durchlaufen. Dies entspricht einer Testabdeckung vom Typ C1, die [Thaller, 2000] folgendermaßen definiert und als Standard bezeichnet:

“Alle Segmente eines Moduls oder Programms werden mindestens einmal ausgeführt. Es werden also alle Pfade durch das Programm wenigstens einmal durchlaufen.“

Auf die korrekte Arbeit mit anderen Modell-Ausprägungen lässt sich schließen, wenn deren Modell-Struktur aus Teilstrukturen besteht, die selbst wiederum mit erwartetem Ergebnis erfolgreich getestet wurden. Die Elemente werden in vier Bereiche eingeteilt, mit denen sich jeweils ein bzw. zwei eigens angefertigte UsageScenarios befassen. Tabelle 7 zählt diese angelegten UsageScenarios auf. Sämtliche Palladio-

Modelle des Testdaten-Sets sowie die Ausgabe-Ergebnisse sind in Anhang A.1 zu finden.

Bereich	**Name des bzw. der UsageScenarios**
Stochastische Ausdrücke	usageScenarioExpression
Workloads	usageScenarioClosedWorkload usageScenarioOpenWorkload
Signaturen von Methoden	usageScenarioSignatures
Kontrollfluss-Elemente in ScenarioBehaviors	usageScenarioActions usageScenarioBranch

Tabelle 7: Die UsageScenarios des TestSets zur Validierung

Kapitel 3.2.3 beschrieb und kategorisierte die berücksichtigten Elemente innerhalb stochastischer Ausdrücke folgendermaßen:

- Konstante Werte primitiven Typs (ganze Zahlen, Gleitkommazahlen, boolesche Ausdrücke): Rot unterstrichen in der Abbildung
- Addition, Subtraktion, Multiplikation, Division und Modulo-Berechnung: Blau unterstrichen in der Abbildung
- Vergleiche (<, >, ==, <=, >=): Gelb unterstrichen in der Abbildung
- If-Else-Alternativen: Grün unterstrichen in der Abbildung
- Logische Verknüpfungen (“and”, “or”, “not”, “xor”): Orange unterstrichen in der Abbildung
- Wahrscheinlichkeitsfunktionen: Violett unterstrichen in der Abbildung

Das UsageScenario mit Namen “usageScenarioExpression” beinhaltet einen OpenWorkload. Die RandomVariable seiner InterarrivalTime wird benutzt, um einen komplexen stochastichen Ausdruck zu definieren, der sämtliche unterstützte Elemente beinhaltet. Abbildung 26 zeigt ein Bild des stochastischen Ausdrucks. Das ScenarioBehavior des UsageScenarios ist an dieser Stelle nicht von Bedeutung und kann daher einfach bleiben.

Bei Workloads wird zwischen den Ausprägungen des OpenWorkloads und des ClosedWorkloads unterschieden. Da jedes UsageScenario nur einen Workload beinhalten

```
BoolPMF[ (true;0.5) (false;0.5) ] AND ( 1 < 2 AND ( 3 > 4 AND ( 5 ==
6 AND ( 7 <= 8 AND ( 9 >= 10 ) ) ) ) ) ? ( ( ( 11 + 12.5 * 13 / 14 %
15 ) - ( -16 ) < 17 ) AND true XOR ( false OR NOT false ) ? 18 : 19 +
20 ^ 1 * DoublePDF[ (0.2; 0.20000000) (0.3; 0.30000000) (0.5;
0.50000000) ] * IntPMF[ (21;0.2) (22;0.3) (22;0.5) ] * DoublePMF
[ (0.2;0.2) (0.3;0.3) (0.5;0.5) ] ) : 23
```

Abbildung 26: Der komplexe stochastische Ausdruck zum Test
Quelle: eigene Darstellung

kann, sind die zwei UsageScenarios mit den Namen “usageScenarioOpenWorkload” und “usageScenarioClosedWorkload” erforderlich, um die korrekte Übertragung beider Workload-Typen in Code zu testen. Da an dieser Stelle ausschließlich der Umgang mit den Workload-Typen in der TestSuite getestet werden soll, kann das ScenarioBehavior hier ebenfalls einfach bleiben.

Beim UsageScenario mit Namen “usageScenarioSignatures” dagegen stehen Elemente innerhalb des ScenarioBehaviors im Mittelpunkt. Hier wird auf die erwartete Übertragung von Methoden-Signaturen ohne Parameter, mit einem oder mehreren Parametern getestet. Außerdem wird die korrekte Behandlung unterschiedlicher Eingabeparameter-Typen getestet. Die benutzten Methoden, Komponenten und Interfaces werden im beiliegenden Repository und System definiert (siehe Anhang A.1). Im Repository-Diagram wird die in Kapitel 3.2.4 beschriebene Methode der Namensersetzung für Interfaces und Methoden benutzt. Somit wird diese auch getestet. Insgesamt werden vier Methoden aufgerufen:

- void signatureVoid()
- void signatureDoubleByte(double parameterDouble, byte parameterByte)
- void signatureInt(int parameterInt)
- void signatureStringBooleanChar(string parameterString, boolean parameterBoolean, char parameterChar)

Das UsageScenario mit Namen “usageScenarioActions” testet eine Abfolge unterschiedlicher Kontrollfluss-Elemente aller Typen hintereinander. Die Abfolge lautet: “Start, Delay, EntryLevelSystemCall, Loop, Branch, Stop”. Diese Abfolge wird nicht nur im obersten ScenarioBehavior des UsageScenarios getestet, sondern auch in den ScenarioBehaviors des definierten Loops und des definierten Branches. Im Falle eines

erfolgreichen Tests kann davon ausgegangen werden, dass im Baum des Kontrollflusses tiefer liegende ScenarioBehaviors ebenfalls wie vorgesehen in Code übertragen werden. Die Praxis zeigte einen wichtigen Kontrollfluss-Fall, der von dem UsageScenario bisher nicht abgedeckt wurde. Alle ScenarioBehaviors in „usageScenarioActions“ haben die gleiche Reihenfolge der Kontrollfluss-Elemente mit einem Branch an letzter Stelle vor dem Stop-Element. Somit wird niemals auf das korrekte Weiterführen des Codes nach einem Branch-Element getestet. Diesen Sonderfall testet das UsageScenario “usageScenarioBranch”, das einen EntryLevelSystemCall auf einen Branch folgen lässt.

Die beschriebenen UsageScenarios werden als Test-Set für die Codegenerierung herangezogen um zu überprüfen, ob der generierte Code dem gewünschten Ergebnis entspricht. Damit ist jedoch noch nicht gesagt, dass das generierte Ergebnis auch tatsächlich als Performance-Test wie gewünscht abläuft. Dies betrifft vor allem das Umsetzungskonzept der Workloads in der generierten TestSuite. Für einen solchen Ablauf-Test ist das beschriebene UsageModel ungeeignet, da es zu viele gleichzeitig laufende UsageScenarios besitzt, die den Ablauf unübersichtlich machen würden. Außerdem sind einige Werte semantisch zur Nutzung in einem Performance-Test ungeeignet, wie zum Beispiel der komplexe stochastische Ausdruck in „usageScenario-Expression“. Daher bedarf es zum Testen des Ablaufs eines übersichtlicheren Testfalls. Dieser sieht nur zwei UsageScenarios vor, jeweils mit einem OpenWorkload und einem ClosedWorkload. Innerhalb der beiden ScenarioBehaviors wird jeweils auf eine von zwei unterschiedlichen Signaturen zugegriffen. Dadurch lässt sich der Zugriff auf eine Methode einem bestimmten Workload zuordnen. Die Modelle und Testergebnisse hierzu sind in Anhang A.2 zu finden.

Wie die Daten im Anhang zeigen, liefern alle Tests die erwarteten Ergebnisse und legen somit keine weiteren Fehler offen. Einen zusätzlichen Hinweis auf die Robustheit der Automatisierung kann die Anwendung am konkreten Beispiel geben, die nun im folgenden Kapitel 4.2 beschrieben wird.

4.2 Anwendungsbeispiel: CoCoMe

Als konkretes Anwendungsbeispiel für die Automatisierung wird das Common Component Modeling Example (kurz: CoCoME) herangezogen. Das Projekt wurde als Paradebeispiel für komponenten- und modellbasierte Softwareentwicklung entworfen.

Die Idee dazu entstand während des Workshop of Component Oriented Programming (kurz: WCOP) auf der European Conference of Object-Oriented Programming (kurz: ECOOP) 2005 in Glasgow. Nachdem das Projekt ins Leben gerufen wurde, meldeten sich weltweit 19 Teams zur Teilnahme (vgl. [Rausch *et al.* , 2008]).

CoCoME beschreibt ein typisches Handels-Informationssystem einer Supermarktkette. Die Kassen einer einzelnen Filiale sind mit einem filial-eigenen Server verbunden. Die Server der Filialen wiederum sind mit einem großen Hauptserver verbunden. Verschiedene Software-Komponenten sind auf die einzelnen Hardware-Komponenten verteilt (vgl. [Herold *et al.* , 2008]). Abbildung 27 zeigt die vorliegenden Use Cases unter Einbezug von Akteuren, die in einem Supermarkt vorkommen.

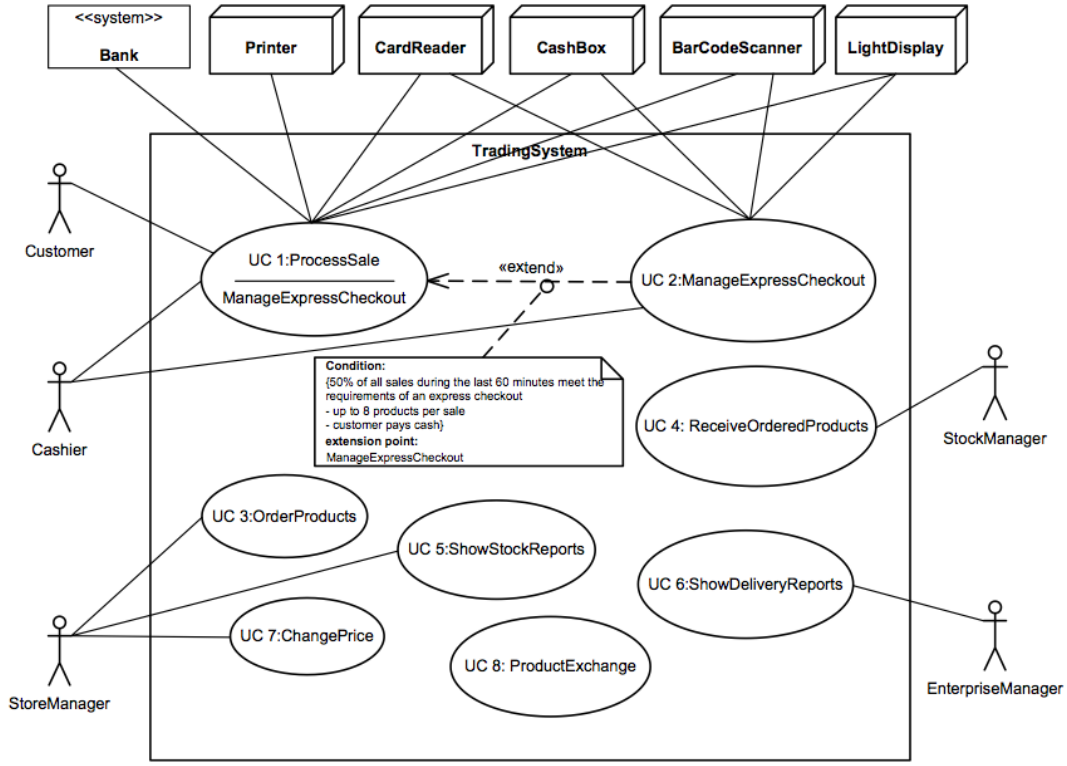


Abbildung 27: Die Use Cases des Common Component Modeling Examples
Quelle: [Herold *et al.* , 2008]

Das CoCoME-Projekt ist auch in Palladio modelliert. Zu jedem der gezeigten acht Use Cases existiert ein UsageModel mit jeweils einem UsageScenario. Da das Testen einer CoCoME-Implementierung auf einem verteilten Hardware-System aufwändig ist, lassen sich sogenannte QoS-Prototypes (Quality of Service) erstellen. Dieser

lässt sich aus den Palladio-Modellen generieren und verspricht, das gesamte verteilte System auf einem Rechner zu simulieren. Hierbei soll sich das Antwortzeit- und simulierte Ressourcenverbrauchsverhalten an die Modellierung in Palladio halten (vgl. [Krogmann & Reussner, 2008]).

Im Rahmen dieser Arbeit wird das entwickelte Plugin auf den “Use Case 8: Product Exchange” angewendet. Dieser beschreibt die Überprüfung des Lagerbestandes eines Produktes in einer Filiale. Sollte dieser nur noch gering sein, werden andere Filialen angefragt, ob diese den Bestand erhöhen können. Obwohl der Use Case nach Abbildung 27 ohne menschliche Aktionen auskommt, wird im zugehörigen Palladio-UsageScenario der Kauf eines Produktes an der Kasse modelliert und somit vorausgesetzt. Abbildung 28 zeigt das entsprechende UsageScenario.

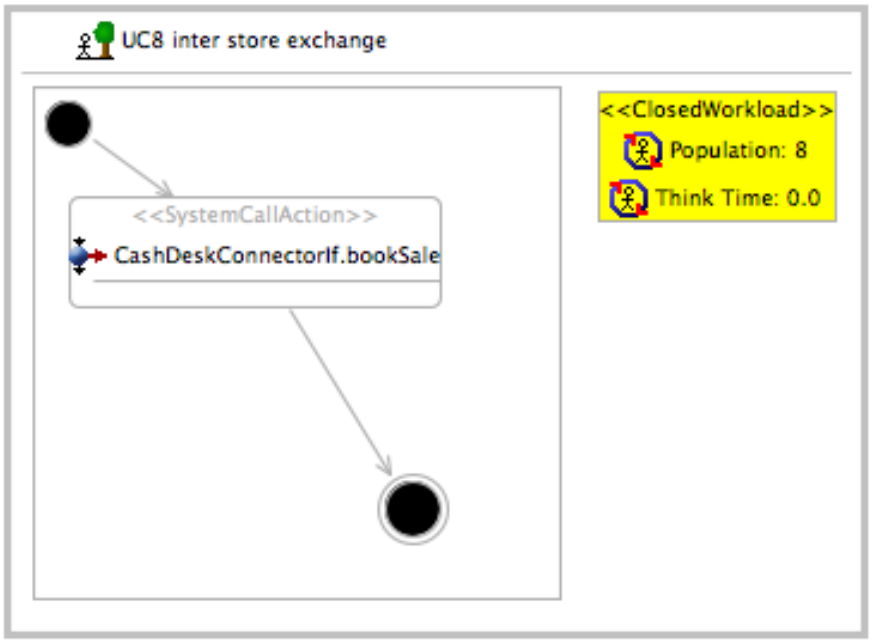


Abbildung 28: Das Palladio-UsageScenario zum CoCoME-Use Case 8: Product Exchange
Quelle: eigene Darstellung

Um den generierten Code zur Lauffähigkeit zu vervollständigen, wurde dieser durch Vererbung mit der bereits bestehenden Grundlagen-Testklasse “TestScenarioBase” für generelle Tests an CoCoME erweitert. Trotzdem ist der generierte Code anschließend noch mit einer weiteren großen Menge an manuellem Code zu füllen. Auffällig ist hier die weitaus aufwändigere und komplexere Befehlsfolge mit größerem Funktions-

umfang, als dies im UsageModel modelliert ist. Vor allem befasst sich der zusätzlich benötigte Code mit der Vorbereitung der Umgebung und der Initialisierung des als Eingabeparameter dienenden SaleTO-Objekts. Normalerweise wäre der passende Ort für diesen Code in der Methode für die Parameter-Übergabe. Die Initialisierung des Objekts hängt jedoch auch von Code aus der Klasse "TestScenarioBase" ab. Der strukturelle Aufbau des generierten Codes lässt einen Zugriff der Methode "getValue" in der Klasse ParameterInput (siehe Kapitel 3.2.4) auf die objektbezogenen Attribute der TestCase-Klasse nicht zu. Entsprechend muss der erwähnte manuelle Code in der Methode "setUp" der TestCase-Klasse eingefügt werden. Dieser setUp-Teil muss ein "static"-Attribut füllen, das der Methode "getValue" später den Zugriff auf den Parameter ermöglicht. Eine Vervollständigung der Klasse erweist sich somit im gegebenen Fall als umständlich, aber realisierbar. Die Umständlichkeiten ergeben sich aus der Entscheidung, sich auf die Klasse "TestScenarioBase" zu beziehen. Der Fremdcode ist in einer eigenen Klasse ausgelagert.

Während das Setup und die Parameterinitialisierung an den dafür vorgesehen Stellen noch stark manuell erweitert sind, zeigen andere Stellen im generierten Code, dass sie keiner weiteren manuellen Editierung nach der Generierung mehr bedürfen. Hierzu zählen die Klasse "GeneratedTestSuite", die den Ablauf von Workloads abbildet (siehe Kapitel 3.2.1), und die Methode "testScenario", die die Struktur des Kontrollflusses abbildet (siehe Kapitel 3.2.2). Durch die Einfachheit des UsageModels, besonders durch die Einfachheit des Kontrollflusses und des Workloads, kann die Codegenerierung ihre Stärken in diesem Beispiel entsprechend wenig zum Tragen bringen. Ein Ausblick für den erzeugten Testcode ist das tatsächliche Ausführen.

4.3 Überprüfung der Anforderungen

Nachdem die erfolgreich durchgeführten Tests dokumentiert wurden, überprüft dieses Kapitel, ob die in Kapitel 3.1 aufgestellten Anforderungen eingehalten wurden.

- Wie in Kapitel 3.3 beschrieben, wurde ein lauffähiges Eclipse-Plugin zur Durchführung der Codegenerierung entwickelt. Das Plugin ermöglicht das Auswählen von Modelldateien sowie eines Ausgabepfades innerhalb des aktuellen Eclipse-Workspaces. Somit sind die Anforderungen 1-3 und 13-15 erfüllt.

- Das in Kapitel 4.1 beschriebene Testset zeigt, dass sich eine Codegenerierung erfolgreich auf Grundlage von Palladio-Modellen durchführen lässt (Anforderung 4). Alle verwendeten Modellelemente wurden in erwarteter Form korrekt in Code übertragen (Anforderung 12). Die generierten Dateien sind in Java-Form (Anforderung 8). Das Testset nimmt für sich in Anspruch alle Elemente zu testen, die für die Codegenerierung relevant sind (Anforderung 17). Ausnahme davon sind die in Kapitel 3.2.3 beschriebenen nicht berücksichtigten Elemente in stochastischen Ausdrücken.

- Der in Kapitel 4.1 beschriebene Test zum Verhalten des generierten Codes bei Ausführung zeigte eine vergleichbare Ausführung mit der Simulation in Palladio (Anforderung 6). Somit ist auch gewährleistet, dass der generierte Code lauffähig ist (Anforderung 11).

- Wie Kapitel 4.2 zeigt, lässt sich der generierte Code auch tatsächlich für Software-Tests an komplexen Implementierungen verwenden, unter anderem auch für Performance-Tests (Anforderung 9).

- Kapitel 3.2.1 beschrieb die Interpretation und Umsetzung von Modell-Elementen in JUnit-Klassen. Somit ist die Nutzung des JUnit-Frameworks ebenfalls gewährleistet (Anforderung 10).

- In Kapitel 3.2.4 wurde das entwickelte Konzept zum Mapping von Bezeichnern innerhalb Palladios auf Bezeichner für die Implementierung beschrieben (Anforderung 16).

Wie sich zeigt, wurden sämtliche aufgestellten Anforderungen unter Berücksichtigung kleiner Limitationen erfüllt. Kapitel 5 zieht ein Schlussfazit und gibt Ausblick auf weitere Möglichkeiten.

5 Zusammenfassung und Ausblick

5.1 Zusammenfassung

Automatische Codegenerierung auf Basis von Modellen kann den Software-Entwickler unterstützen. Jedoch konnte sie im konkreten Fall die manuelle Programmierung nicht vollständig ersetzen. Vorgesehen war die Generierung von Performance-Test-Code aus Palladio-Modellen. Diese aber haben Performance-Prediction zum Ziel. Obwohl sich Palladio-Modelle für unterschiedliche Entwicklerrollen anbieten, werden Software-Tests für die Nutzung von Palladio nicht berücksichtigt. Entsprechend ist es notwendig, Palladio-Modelle bezüglich ihrer Semantik für Performance-Tests zu interpretieren. Außerdem sind die notwendigen Daten für einen Performance-Test nicht vollständig in einem Palladio-Modell vorhanden, sodass der Test-Entwickler gezwungen ist, restliche Lücken im generierten Code manuell zu schließen.

Im Rahmen dieser Arbeit wurde ein Eclipse-Plugin entwickelt, das Palladio-Modelle einliest und Performance-Test-Code generiert. Der generierte Code ist Java-basiert und verwendet die Test-Frameworks JUnit und JUnitPerf. Um die Codegenerierung zu realisieren, verwendet das Plugin die openArchitectureWare-Technologie mit Xpand-Templates zur Model-To-Text-Transformation. Mit Hilfe dieses Plugins wurde Performance-Test-Code am Beispiel von Palladio-Modellen für das Software-Projekt CoCoME generiert. Dieser ließ sich an dazu wenigen vorgesehenen Stellen manuell vervollständigen. Neben der Einarbeitung in die technischen Grundlagen wurde auch eine Strategie zur Interpretation von Palladio-Modellen für Performance-Tests und ihre Umsetzung in Java erarbeitet.

5.2 Ausblick

Diese Arbeit lässt als Ausblick die tatsächliche Testausführung für das Projekt CoCoME offen. Ergebnisse eines Performance-Tests, der auf automatisch generiertem Code basiert, könnten mit Palladio-Simulationsergebnissen verglichen werden. Neben Performance-Tests lässt sich der generierte Code auch verwenden, um regelmäßige und zahlreiche Service-Anfragen zu automatisieren und somit den in Palladio modellierten Fall tatsächlich zu nutzen und nicht bloß zu testen. Denkbar wäre zum Beispiel die regelmäßige Abfrage von statischen Websites zur Aktualisierung.

Die entwickelte Lösung stößt bei der Umsetzung von vordefinierten Funktionen (z. B. Exponentialverteilung) im Metamodell stoex an seine Grenzen der Umsetzbarkeit im Rahmen dieser Arbeit. Die Anbindung einer Java-Bibliothek, die die Berechnung der erwähnten Verteilungen ermöglicht, wäre eine sinnvolle Erweiterung des Projekts zur Vervollständigung. Die Berücksichtigung von Rückgabeparametern und deren anschließende Verwendung in stochastischen Ausdrücken allerdings dürfte sich durch Codegenerierung deutlich komplizierter darstellen, als die Alternative des Testautors, nachträglich den Code selbst anzupassen.

Für die Zukunft wäre auch denkbar, sämtliche Informationslücken innerhalb von Palladio-Modellen in einem weiteren Modell zu erfassen. Somit wäre der generierte Code nach seiner Erzeugung vollständig. Dies hätte den Vorteil höherer Konsistenz zwischen Modell und Code. Anstatt Zusatzinformationen in einer kryptischen Sprache in angeheftete Notizen zu schreiben, wäre die Entwicklung eines eigenen Metamodells denkbar. Modell-Instanzen dieses Metamodells könnten auf Palladio-Modellelemente zurückgreifen und die wichtigen Zusatzinformationen mit Link zum betreffenden Palladio-Element speichern. Dieses Modell würde dann in der entwickelten Codegenerierung an die Stelle des derzeit leicht zweckentfremdeten Repository-Diagramms treten. Auch das Erstellen von Metamodellen lässt sich mit Eclipse leicht bewerkstelligen.

Literatur

[jun, n.d.] *JUnitPerf API Reference*. http://underpop.free.fr/j/java/professional-java-tools-for-extreme-programming/LiB0191.html, visited 2013-02-28.

[iee, 1990] 1990. *IEEE Standard Glossary of Software Engineering Terminology*.

[iee, 2008] 2008. *IEEE Standard for Software and System Test Documentation*.

[oaw, 2009] 2009. *Official openArchitectureWare Homepage*. http://openarchitectureware.org/, visited 2013-02-28.

[xmi, 2011] 2011 (August). *OMG MOF 2 XMI Mapping Specification*.

[Balzert, 2002] Balzert, H. 2002. *Lehrbuch der Softwaretechnik*. Spektrum Akademischer Verlag.

[Becker, 2010a] Becker, Jun.-Prof. Dr.-Ing. Steffen. 2010a. *Model-Driven Software Development*. Vorlesungsfolien WS 2010.

[Becker, 2010b] Becker, Jun.-Prof. Dr.-Ing. Steffen. 2010b. *Model-Driven Software Development*. Vorlesungsfolien WS 2010.

[Brüseke *et al.* , 2011] Brüseke, Frank, Engels, Prof. Dr. Gregor, & Becker, Jun.-Prof. Dr.-Ing. Steffen. 2011. *Palladio-based Performance Blame Analysis*. Proceedings of the 16th international workshop on Component-oriented programming.

[Czarnecki & Helsen, 2003] Czarnecki, Krzysztof, & Helsen, Simon. 2003. Classification of Model Transformation Approaches. *In: Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*. University of Waterloo, Canady.

[Dijkstra, 1990] Dijkstra, E.W. 1990. *Structured Programming in Software Engineering Techniques*. ARTECH HOUSE INC.

[Dustin *et al.* , 2009] Dustin, Elfriede, Garret, Thom, & Gauf, Bernie. 2009. *Implementing Automated Software Testing*. Addison-Wesley.

[Efftinge & Kadura, n.d.] Efftinge, Sven, & Kadura, Clemens. *OpenArchitectureWare 4.1 Xpand Language Reference*. http://www.openarchitectureware.org/pub/documentation/4.1/r20_XpandReference.pdf, visited 2013-02-28.

[Engels & Soltenborn, 2009] Engels, Prof. Dr. Gregor, & Soltenborn, Christian. 2009. *Softwareentwurf*. Vorlesungsfolien WS 2009.

[Gamma *et al.* , 2004] Gamma, Erich, Helm, Richard, Johnson, Ralph, & Vlissides, John. 2004. *Entwurfsmuster*. Addison-Wesley.

[Herold *et al.* , 2008] Herold, Sebastian, Klus, Holger, Welsch, Yannick, Deiters, Constanze, Rausch, Andreas, Reussner, Ralf, Krogmann, Klaus, Koziolek, Heiko, Mirandola, Raffaela, Hummel, Benjamin, Meisinger, Michael, & Pfaller, Christian. 2008. CoCoME - The Common Component Modeling Example. *In: The common component modeling example : comparing software component models*. Springer-Verlag Berlin Heidelberg.

[Koziolek *et al.* , 2008] Koziolek, Heiko, Happe, Jens, Becker, Jun.-Prof. Dr.-Ing. Steffen, & Reussner, Ralf. 2008. Evaluating Performance of Software Architecture Models with the Palladio Component Model. *Model-Driven Software Development: Integrating Quality Assurance, IDEA Group Inc*.

[Krogmann & Reussner, 2008] Krogmann, Klaus, & Reussner, Ralf. 2008. Palladio - Prediction of Performance Properties. *In: The Common Component Modeling Example : Comparing Software Component Models*. Springer-Verlag Berlin Heidelberg.

[Lipinski, 2013] Lipinski, Klaus. 2013. *IT Wissen*. http://www.itwissen.info/definition/lexikon/oAW-openArchitectureWare.html, visited 2013-02-28.

[Molyneaux, 2009] Molyneaux, Ian. 2009. *The Art of Application Performance Testing*. O'Reilly Media, Inc.

[Rakitin, 1997] Rakitin, Steven R. 1997. *Software Verifikation and Validation: A Practioner's Guide*. NATO Science Committee.

[Rausch *et al.* , 2008] Rausch, Andreas, Reussner, Ralf, Mirandola, Raffaela, & Plasil, Frantisek. 2008. Introduction. *In: The common component modeling example : comparing software component models*. Springer-Verlag Berlin Heidelberg.

[Schäfer, 2011] Schäfer, Prof. Dr. Wilhelm. 2011. *Modellbasierte Softwareentwicklung*. Vorlesungsfolien WS 2011.

[Stolze, 2008] Stolze, Jirk. 2008. *Performancetests und Bottleneck-Analyse in Multischichtarchitekturen*. http://www.performance-test.de/, visited 2013-02-28.

[Thaller, 2000] Thaller, Georg Erwin. 2000. *Software-Test*. Heinz Heise.

[Völter *et al.* , 2006] Völter, M., Stahl, T., Bettin, J., Haase, A., Helsen, S., Czarnecki, K., & von Stockfleth, B. 2006. *Model-Driven Software Development: Technology, Engineering, Management*. Wiley Software Patterns Series. John Wiley & Sons.

[Westphal, 2001] Westphal, Frank. 2001. *Unit Tests mit JUnit*. http://www.frankwestphal.de/UnitTestingmitJUnit.html, visited 2013-02-28.

[Wolmeringer & Klein, 2006] Wolmeringer, Gottfried, & Klein, Thorsten. 2006. *Profikurs Eclipse 3 : Mit Eclipse 3.2 und Plugins professionell Java-Anwendungen entwickeln - Von UML bis JUnit*. 2., verbesserte und erweiterte auflage edn. Wiesbaden : Friedr. Vieweg Sohn Verlag / GWV Fachverlage GmbH, Wiesbaden.

[Wunsch & Schreiber, 1992] Wunsch, Gerhard, & Schreiber, Helmut. 1992. *Stochastische Systeme*. 3., neubearb. u. erw. aufl. edn. Berlin [u.a.] : Springer.

A Anhang

A.1 Ergebnisse des Testsets zur Überprüfung der Codegenerierung

Die folgenden Unterkapitel beinhalten jeweils Abbildungen von Ausschnitten aus Palladio-Modellen. Diese Palladio-Modelle wurden zur Codegenerierung verwendet. Der jeweils folgende Java-Code ist ein Ausschnitt aus dem generierten Code und soll die korrekte Umsetzung der entsprechenden Modell-Ausschnitte verdeutlichen.

A.1.1 Test: Stochastische Ausdrücke

```
BoolPMF[ (true;0.5) (false;0.5) ] AND ( 1 < 2 AND ( 3 > 4 AND ( 5 ==
6 AND ( 7 <= 8 AND ( 9 >= 10 ) ) ) ) ) ? ( ( ( 11 + 12.5 * 13 / 14 %
15 ) - ( -16 ) < 17 ) AND true XOR ( false OR NOT false ) ? 18 : 19 +
20 ^ 1 * DoublePDF[ (0.2; 0.20000000) (0.3; 0.30000000) (0.5;
0.50000000) ] * IntPMF[ (21;0.2) (22;0.3) (22;0.5) ] * DoublePMF
[ (0.2;0.2) (0.3;0.3) (0.5;0.5) ] ) : 23
```

Abbildung 29: Der komplexe stochastische Ausdruck zum Test
Quelle: eigene Darstellung

Listing 1: Die Abbildung des stochastischen Ausdrucks in Java

```
1 /*
2 ...
3 */
4 public class RV_usageScenarioExpression_Workload extends AbstractTimer {
5         public double calcValue() {
6                 return (boolPMF(true, 0.5, false, 0.5) && ((1 < 2 && ((3 > 4 && ((5 == 6 && ((7 <= 8 && (9 >= 10))))
                        ))))) ? ((((((11 + (((12.5 * 13) / 14) % 15))) - ((-16))) < 17) && (true && !((false || (!
                        false)))) || (!true && ((false || (!false))))) ? 18 : (19 + (((Math.pow(20, 1) * calcPDF(0.2,
                        0.2, 0.3, 0.3, 0.5, 0.5)) * calcPMF(21, 0.2, 22, 0.3, 22, 0.5)) * calcPMF(0.2, 0.2, 0.3, 0.3,
                        0.5, 0.5)))) : 23;
7         }
8 }
```

A.1.2 Test: Workload

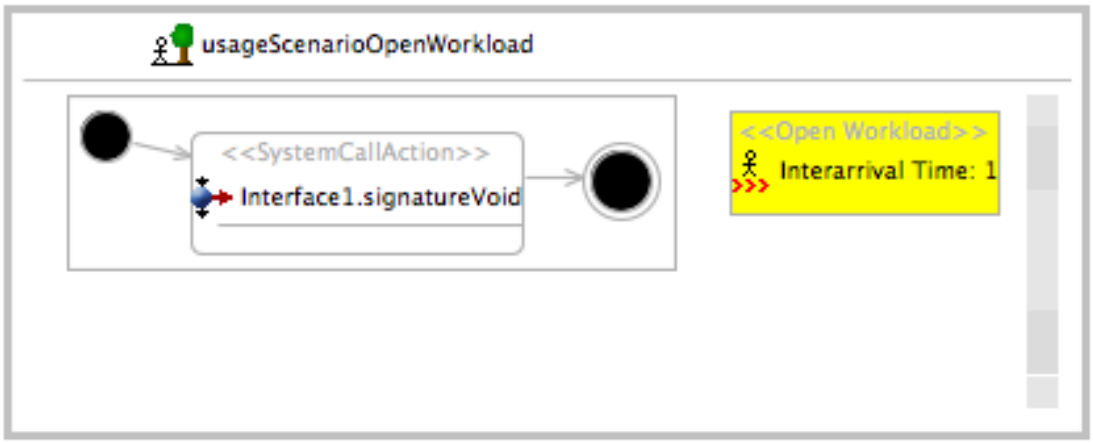


Abbildung 30: Der OpenWorkload im UsageScenario zum Test
Quelle: eigene Darstellung

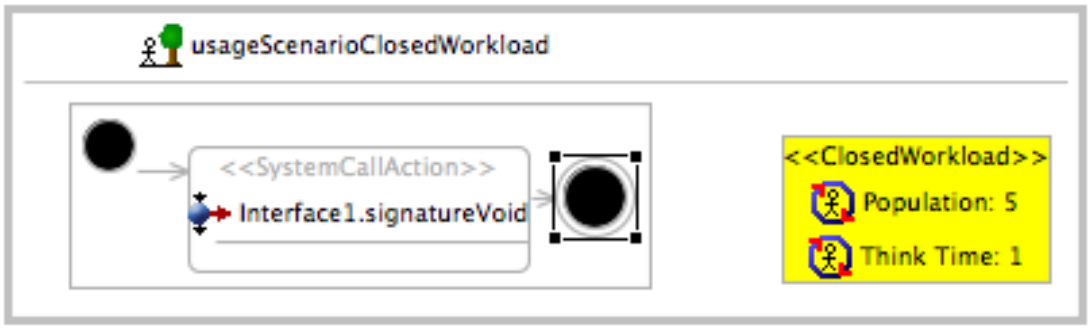


Abbildung 31: Der ClosedWorkload im UsageScenario zum Test
Quelle: eigene Darstellung

Listing 2: Die Abbildung des OpenWorkloads und des ClosedWorkloads in Java

```
1  /*...*/
2  public class GeneratedTestSuite {
3          /*...*/
4          public static Test suite() {
5                  int users = 0;
6                  LoadTest workloadTest = null;
7                  Test usageScenarioTest = null;
8                  ThinkTimeRepeatedTest thinkTimeRepeatedTest = null;
9                  Timer interArrivalTimer = null;
10                 Timer thinkTimer = null;
11
12                 SimultaneousLoadTestSuite simultaneousLoadTestSuite = new SimultaneousLoadTestSuite();
13                 /*...*/
14                 //building TestCase for Scenario: usageScenarioOpenWorkload
15                 usageScenarioTest = new TestMethodFactory(TestCase_usageScenarioOpenWorkload.class, "testScenario");
16
17                 interArrivalTimer = new RV_usageScenarioOpenWorkload_Workload();
```

```
18                  workloadTest = new LoadTest(usageScenarioTest, TestDuration.getUsers_usageScenarioOpenWorkload(),
                        interArrivalTimer);
19
20                  simultaneousLoadTestSuite.addTest(workloadTest);
21
22                  //building TestCase for Scenario: usageScenarioClosedWorkload
23                  usageScenarioTest = new TestMethodFactory(TestCase_usageScenarioClosedWorkload.class, "testScenario"
                        );
24
25                  users = 5;
26                  thinkTimer = new RV_usageScenarioClosedWorkload_Workload();
27                  thinkTimeRepeatedTest = new ThinkTimeRepeatedTest(usageScenarioTest, TestDuration.
                        getIterations_usageScenarioClosedWorkload(), thinkTimer);
28                  workloadTest = new LoadTest(thinkTimeRepeatedTest, users);
29
30                  simultaneousLoadTestSuite.addTest(workloadTest);
31                  /*...*/
32                  TestSuite suite = new TestSuite();
33                  addSetUpTests(suite);
34                  suite.addTest(simultaneousLoadTestSuite);
35                  addTearDownTests(suite);
36                  return suite;
37          }
38
39          public static void main(String[] args) {
40                  junit.textui.TestRunner.run(suite());
41          }
42  }
```

A.1.3 Test: Methodenaufrufe

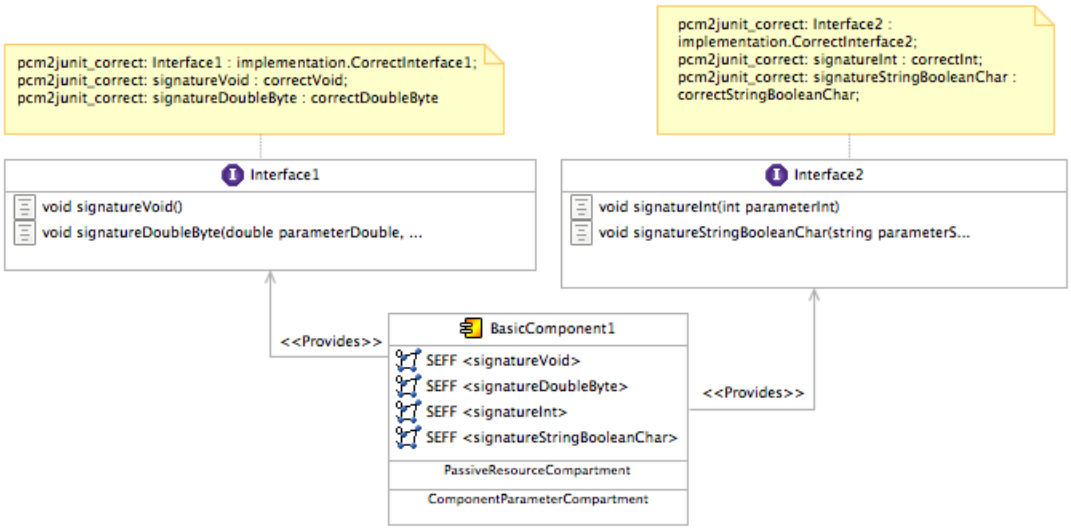


Abbildung 32: Die Ersetzung von Bezeichnern in Palladio zum Test
Quelle: eigene Darstellung

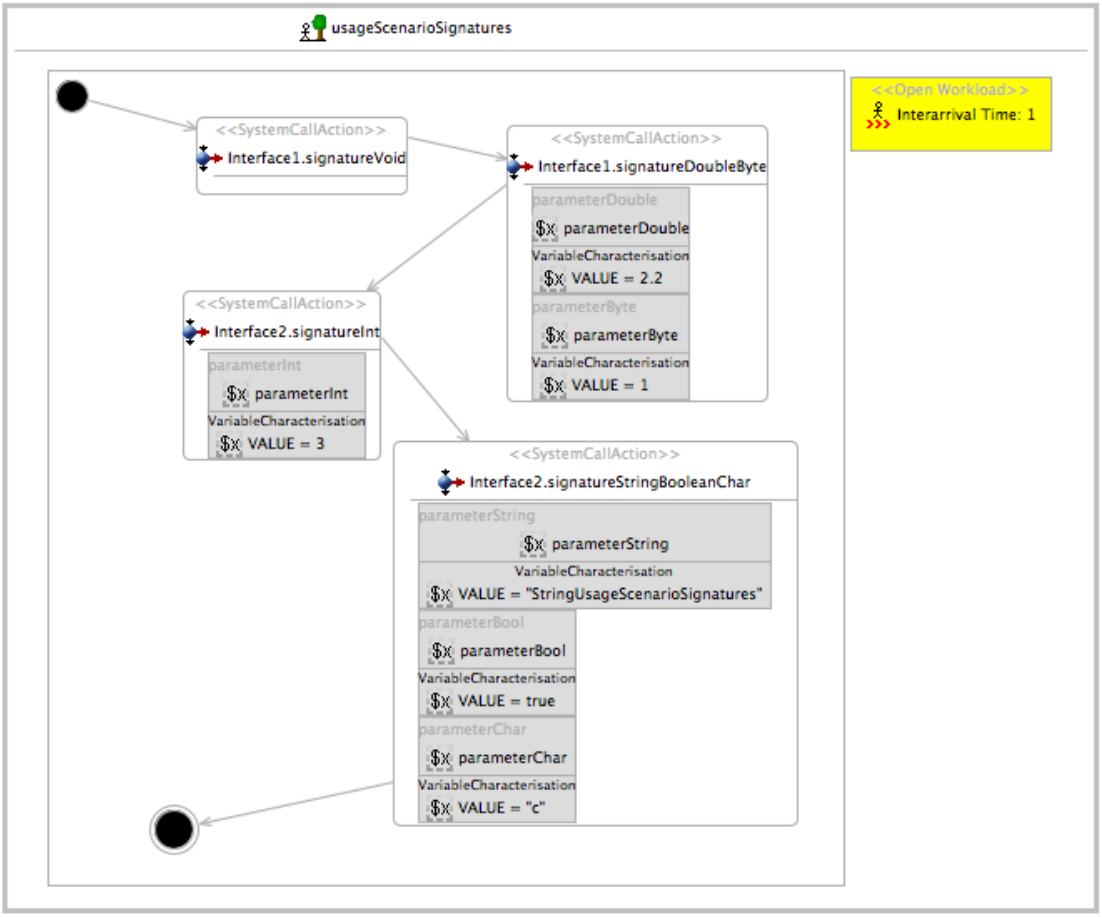


Abbildung 33: Der Aufruf unterschiedlich parametrisierter Methoden zum Test
Quelle: eigene Darstellung

Listing 3: Die Abbildung der Methodenaufrufe mit ersetzten Bezeichnern in Java

```
1  /*...*/
2  public class TestCase_usageScenarioSignatures extends TestCase {
3          public TestCase_usageScenarioSignatures(String name) {
4                  super(name);
5          }
6
7          implementation.CorrectInterface1 providedRole_Interface1 = null;
8
9          implementation.CorrectInterface2 providedRole_Interface2 = null;
10         /*...*/
11         public void testScenario() {
12
13                 try {
14                         providedRole_Interface1.correctVoid();
15                 } catch (Exception e) {
16                         e.printStackTrace();
17                 }
18
19                 try {
20                         providedRole_Interface1.correctDoubleByte(ParameterInput.
                               getValue_usageScenarioSignatures_aName__KBoTUAMKEeKwv7t2GRc5og_parameterDouble(),
                               ParameterInput.
                               getValue_usageScenarioSignatures_aName__KBoTUAMKEeKwv7t2GRc5og_parameterByte());
21                 } catch (Exception e) {
22                         e.printStackTrace();
```

```
23                  }
24
25                  try {
26                          providedRole_Interface2.correctInt(ParameterInput.
                                getValue_usageScenarioSignatures_aName__NZY5UAMKEeKwv7t2GRc5og_parameterInt());
27                  } catch (Exception e) {
28                          e.printStackTrace();
29                  }
30
31                  try {
32                          providedRole_Interface2.correctStringBooleanChar(ParameterInput.
                                getValue_usageScenarioSignatures_aName__QruMwAMKEeKwv7t2GRc5og_parameterString(),
                                ParameterInput.
                                getValue_usageScenarioSignatures_aName__QruMwAMKEeKwv7t2GRc5og_parameterBool(),
                                ParameterInput.
                                getValue_usageScenarioSignatures_aName__QruMwAMKEeKwv7t2GRc5og_parameterChar());
33                  } catch (Exception e) {
34                          e.printStackTrace();
35                  }
36
37          }
38          /*...*/
39  }
```

A.1.4 Test: Kontrollfluss-Elemente

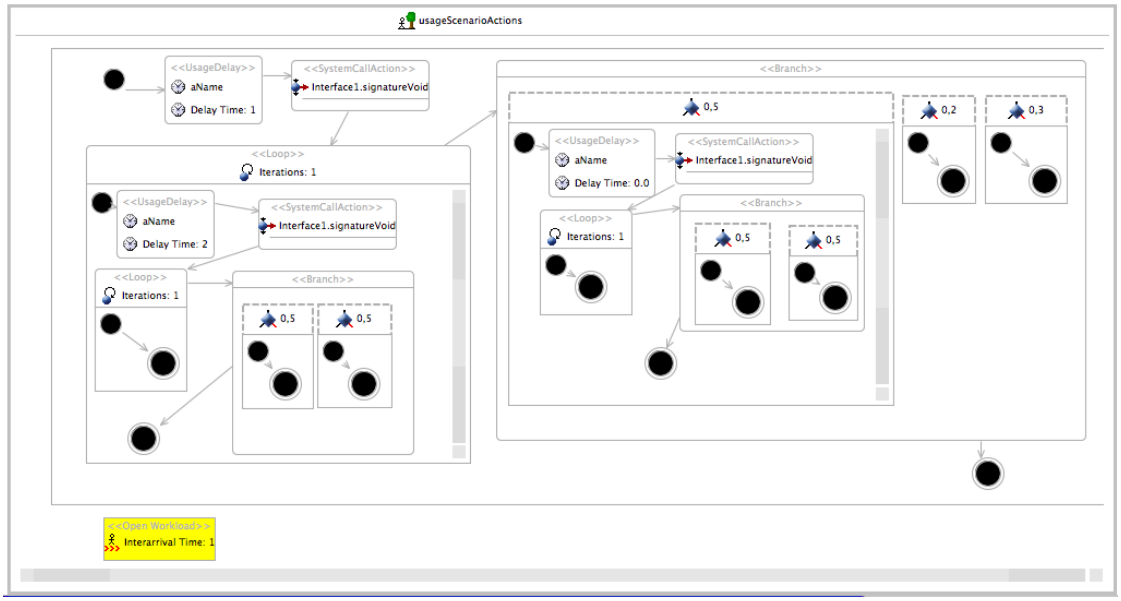


Abbildung 34: Der Durchlauf unterschiedlicher Kontrollfluss-Elemente zum Test
Quelle: eigene Darstellung

Listing 4: Die Abbildung der Kontrollfluss-Elemente in Java

```
1  /*...*/
2  public class TestCase_usageScenarioActions extends TestCase {
3          public TestCase_usageScenarioActions(String name) {
4                  super(name);
5          }
6
7          implementation.CorrectInterface1 providedRole_Interface1 = null;
8
```

```
9         implementation.CorrectInterface2 providedRole_Interface2 = null;
10        /*...*/
11        public void testScenario() {
12
13                try {
14                        Thread.sleep((int) new RV_usageScenarioActions_aName__QmL6kAMMEeKwv7t2GRc5og()
15                                        .getDelay());
16                } catch (InterruptedException e) {
17                        e.printStackTrace();
18                }
19
20                try {
21                        providedRole_Interface1.correctVoid();
22                } catch (Exception e) {
23                        e.printStackTrace();
24                }
25
26                double loopIterations_loop1__sS4rIAMMEeKwv7t2GRc5og = new
                        RV_usageScenarioActions_loop1__sS4rIAMMEeKwv7t2GRc5og().calcValue();
27
28                for (int index_loop1__sS4rIAMMEeKwv7t2GRc5og = 0; index_loop1__sS4rIAMMEeKwv7t2GRc5og <
                        loopIterations_loop1__sS4rIAMMEeKwv7t2GRc5og; index_loop1__sS4rIAMMEeKwv7t2GRc5og++) {
29
30                        try {
31                                Thread.sleep((int) new RV_usageScenarioActions_aName__cTnyEAMMEeKwv7t2GRc5og()
32                                                .getDelay());
33                        } catch (InterruptedException e) {
34                                e.printStackTrace();
35                        }
36
37                        try {
38                                providedRole_Interface1.correctVoid();
39                        } catch (Exception e) {
40                                e.printStackTrace();
41                        }
42
43                        double loopIterations_aName__1sPnMAMMEeKwv7t2GRc5og = new
                                RV_usageScenarioActions_aName__1sPnMAMMEeKwv7t2GRc5og()
44                                        .calcValue();
45
46                        for (int index_aName__1sPnMAMMEeKwv7t2GRc5og = 0; index_aName__1sPnMAMMEeKwv7t2GRc5og <
                                loopIterations_aName__1sPnMAMMEeKwv7t2GRc5og; index_aName__1sPnMAMMEeKwv7t2GRc5og++) {
47
48                        }
49
50                        double rnd_aName__20pZsAMMEeKwv7t2GRc5og = Math.random();
51                        double sum_aName__20pZsAMMEeKwv7t2GRc5og = 0;
52
53                        sum_aName__20pZsAMMEeKwv7t2GRc5og += 0.5;
54                        if (sum_aName__20pZsAMMEeKwv7t2GRc5og >= rnd_aName__20pZsAMMEeKwv7t2GRc5og) {
55
56                                sum_aName__20pZsAMMEeKwv7t2GRc5og -= 1;
57                        }
58
59                        sum_aName__20pZsAMMEeKwv7t2GRc5og += 0.5;
60                        if (sum_aName__20pZsAMMEeKwv7t2GRc5og >= rnd_aName__20pZsAMMEeKwv7t2GRc5og) {
61
62                                sum_aName__20pZsAMMEeKwv7t2GRc5og -= 1;
63                        }
64
65                }
66
67                double rnd_aName__Occ8AAMOEeKwv7t2GRc5og = Math.random();
68                double sum_aName__Occ8AAMOEeKwv7t2GRc5og = 0;
69
70                sum_aName__Occ8AAMOEeKwv7t2GRc5og += 0.5;
71                if (sum_aName__Occ8AAMOEeKwv7t2GRc5og >= rnd_aName__Occ8AAMOEeKwv7t2GRc5og) {
72
73                        try {
74                                Thread.sleep((int) new RV_usageScenarioActions_aName__ZO2uoAMOEeKwv7t2GRc5og().
                                        getDelay());
75                        } catch (InterruptedException e) {
76                                e.printStackTrace();
77                        }
```

```
78
79                          try {
80                                  providedRole_Interface1.correctVoid();
81                          } catch (Exception e) {
82                                  e.printStackTrace();
83                          }
84
85                          double loopIterations_aName__R89_8AMPEeKwv7t2GRc5og = new
                                RV_usageScenarioActions_aName__R89_8AMPEeKwv7t2GRc5og().calcValue();
86
87                          for (int index_aName__R89_8AMPEeKwv7t2GRc5og = 0; index_aName__R89_8AMPEeKwv7t2GRc5og <
                                loopIterations_aName__R89_8AMPEeKwv7t2GRc5og; index_aName__R89_8AMPEeKwv7t2GRc5og++) {
88
89                          }
90
91                          double rnd_aName__ShIc8AMPEeKwv7t2GRc5og = Math.random();
92                          double sum_aName__ShIc8AMPEeKwv7t2GRc5og = 0;
93
94                          sum_aName__ShIc8AMPEeKwv7t2GRc5og += 0.5;
95                          if (sum_aName__ShIc8AMPEeKwv7t2GRc5og >= rnd_aName__ShIc8AMPEeKwv7t2GRc5og) {
96
97                                  sum_aName__ShIc8AMPEeKwv7t2GRc5og -= 1;
98                          }
99
100                         sum_aName__ShIc8AMPEeKwv7t2GRc5og += 0.5;
101                         if (sum_aName__ShIc8AMPEeKwv7t2GRc5og >= rnd_aName__ShIc8AMPEeKwv7t2GRc5og) {
102
103                                 sum_aName__ShIc8AMPEeKwv7t2GRc5og -= 1;
104                         }
105
106                         sum_aName__Occ8AAMOEeKwv7t2GRc5og -= 1;
107                 }
108
109                 sum_aName__Occ8AAMOEeKwv7t2GRc5og += 0.2;
110                 if (sum_aName__Occ8AAMOEeKwv7t2GRc5og >= rnd_aName__Occ8AAMOEeKwv7t2GRc5og) {
111
112                         sum_aName__Occ8AAMOEeKwv7t2GRc5og -= 1;
113                 }
114
115                 sum_aName__Occ8AAMOEeKwv7t2GRc5og += 0.3;
116                 if (sum_aName__Occ8AAMOEeKwv7t2GRc5og >= rnd_aName__Occ8AAMOEeKwv7t2GRc5og) {
117
118                         sum_aName__Occ8AAMOEeKwv7t2GRc5og -= 1;
119                 }
120
121         }
122         /*...*/
123 }
```

A.1.5 Test: Branch-Element

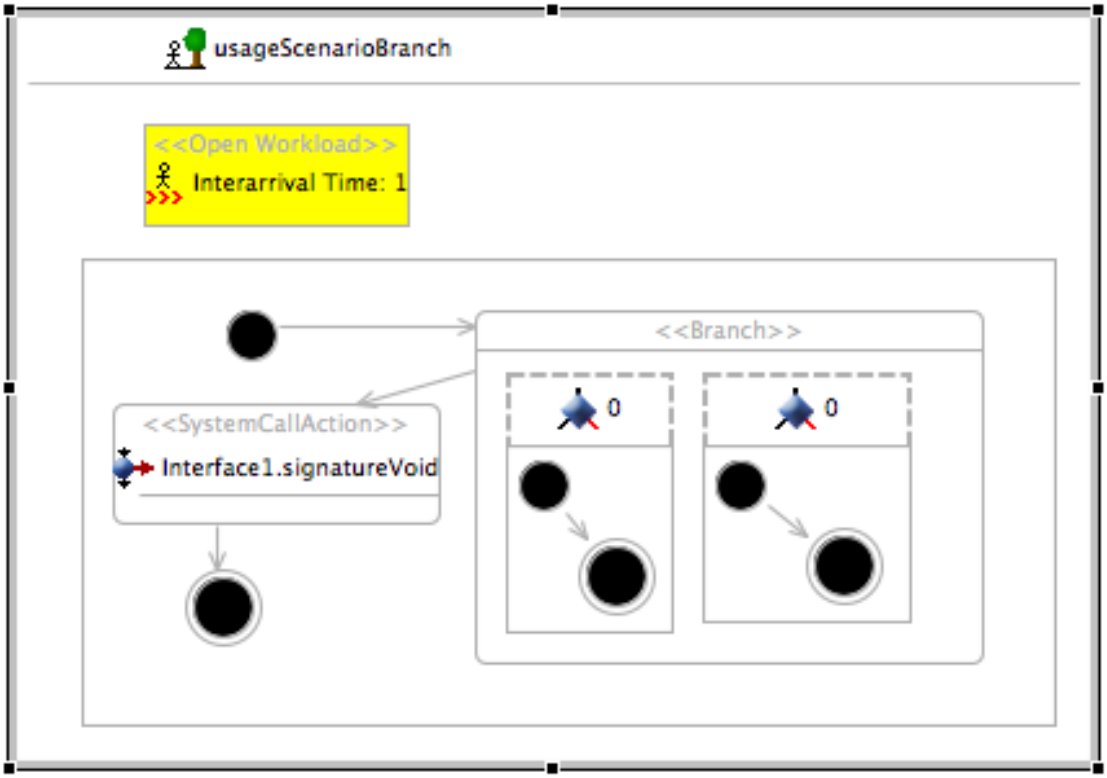


Abbildung 35: Die Verwendung des Branch-Elements zum Test
Quelle: eigene Darstellung

Listing 5: Die Abbildung des Branch-Elements in Java

```
/*...*/
public class TestCase_usageScenarioBranch extends TestCase {
        public TestCase_usageScenarioBranch(String name) {
                super(name);
        }

        implementation.CorrectInterface1 providedRole_Interface1 = null;

        implementation.CorrectInterface2 providedRole_Interface2 = null;
        /*...*/
        public void testScenario() {

                double rnd_aName__kpOcEH7UEeKIvq0EC7DB5w = Math.random();
                double sum_aName__kpOcEH7UEeKIvq0EC7DB5w = 0;

                sum_aName__kpOcEH7UEeKIvq0EC7DB5w += 0.5;
                if (sum_aName__kpOcEH7UEeKIvq0EC7DB5w >= rnd_aName__kpOcEH7UEeKIvq0EC7DB5w) {

                        sum_aName__kpOcEH7UEeKIvq0EC7DB5w -= 1;
                }


                sum_aName__kpOcEH7UEeKIvq0EC7DB5w += 0.5;
                if (sum_aName__kpOcEH7UEeKIvq0EC7DB5w >= rnd_aName__kpOcEH7UEeKIvq0EC7DB5w) {

                        sum_aName__kpOcEH7UEeKIvq0EC7DB5w -= 1;
                }
```

```
28
29                  try {
30                          providedRole_Interface1.correctVoid();
31                  } catch (Exception e) {
32                          e.printStackTrace();
33                  }
34
35          }
36          /*...*/
37  }
```

A.2 Ergebnisse des Testprojekts für den Abgleich mit der Simulation

Es folgen Simulationergebnisse des Palladio-Projekts “RunTest” und Testergebnisse der generierten TestSuite aus selbigem Projekt. Der “time”-Wert in Abbildung 37 entspricht dem “Event Time”-Wert in Abbildung 36.

Signature 1:			**Signature 2:**	
Event Time	Time Span		Event Time	Time Span
0.0	0.0		2.0	0.0
1.0	0.0		2.0	0.0
2.0	0.0		4.0	0.0
3.0	0.0		4.0	0.0
4.0	0.0		6.0	0.0
5.0	0.0		6.0	0.0
6.0	0.0		8.0	0.0
7.0	0.0		8.0	0.0
8.0	0.0			
9.0	0.0			

Abbildung 36: Die aggregierten Simulationsergebnisse des Palladio-Projekts “RunTest”
Quelle: eigene Darstellung

```
starting thread
starting thread
...RunComponent: signature2 executed (time:0.0)
RunComponent: signature2 executed (time:0.0010)
RunComponent: signature1 executed (time:0.0010)
.RunComponent: signature1 executed (time:0.998)
.RunComponent: signature1 executed (time:1.999)
.RunComponent: signature2 executed (time:2.001)
.RunComponent: signature2 executed (time:2.001)
.RunComponent: signature1 executed (time:2.999)
.RunComponent: signature1 executed (time:3.999)
.RunComponent: signature2 executed (time:4.001)
.RunComponent: signature2 executed (time:4.001)
.RunComponent: signature1 executed (time:5.0)
.RunComponent: signature1 executed (time:6.0)
.RunComponent: signature2 executed (time:6.001)
.RunComponent: signature2 executed (time:6.002)
.RunComponent: signature1 executed (time:7.0)
.RunComponent: signature1 executed (time:8.001)
..RunComponent: signature2 executed (time:8.002)
RunComponent: signature2 executed (time:8.002)
.RunComponent: signature1 executed (time:9.001)

Time: 11,005

OK (20 tests)
```

Abbildung 37: Der Ablauf der generierten TestSuite zu “RunTest”
Quelle: eigene Darstellung

A.3 Aufbau des Plugin-Projekts

Es folgt eine Beschreibung des Source-Ordners im Projekt “PCM2JUnit_Plugin”.

debug enthält Daten zum Auffinden von Fehlern innerhalb der Xpand-Templates.

- **standalone.mwe** führt die Codegenerierung ohne Starten des Plugins durch.

pcm2junit_plugin enthält Java-Code für das Eclipse-Plugin.

- **Activator.java** wird beim Start des Plugins aufgerufen und und startet die Initialisierung.
- **ConsoleProgressMonitor.java** führt das Logging auf der Konsole aus.
- **DirectorySelection.java** ermöglicht die Schnellauswahl.
- **PluginManager.java** leitet den Gesamtablauf des Plugins.
- **PluginView.java** visualisiert die View des Plugins.
- **ResourceSelection.java** visualisiert die Auswahldialoge.
- **ValueChangedEvent.java** wird geworfen, falls eine Resource ausgewählt wurde.
- **ValueChangedListener.java** gibt den Befehl zur Aktualisierung der View.
- **WorkflowManager** startet den MWE-Workflow mit gegebenen Parametern.

template enthält die verwendeten Xpand-Templates.

- **CommonTemplate.xpt** leistet unterstützende Aufgaben für die andren Templates.
- **DurationTemplate.xpt** generiert die Klasse “TestDuration”.
- **HelpclassTemplate.xpt** generiert die Statischen Hilfsklassen.
- **ParameterTemplate.xpt** generiert die Klasse “ParameterInput”.
- **SuiteResourceTemplate.xpt** generiert die “GeneratedTestSuite”, also die oberste Wurzel der Testklassen.
- **TestCaseTemplate.xpt** generiert die TestCase-Klassen.
- **TimerTemplate.xpt** generiert die RandomVariable-Klassen für stochastische Ausdrücke.

workflow beinhaltet den MWE-Workflow.

- **generator_Plugin.mwe** führt die Generierung für das Plugin durch.

A.4 Inhalt der CD

diplomarbeit.pdf ist dieses Dokument im PDF Format,

report/ enthält die LaTeX-Quellen zu dieser Arbeit,

docs/ enthält Zitierte Arbeiten, soweit elektronisch vorhanden,

PCM2JUnit_Plugin enthält das entwickelte Softwareprojekt.

testresult/ enthält Testdaten und Ergebnisse.

- **generationTest/** enthält Palladio-Eingabedaten und Generierungsergebnisse.
- **runTest/** enthält Palladio-Eingabedaten, Palladio-Simulationsergebnisse, Generierungsergebnisse und Ablaufs-Logs zum Abgleich.
- **cocome/** enthält die verwendeten Palladio-Eingabedaten, Generierungsergebnisse und manuell vervollständigten Code.