Fakultät für Mathematik, Informatik und Naturwissenschaften

Lehr- und Forschungsgebiet Softwarekonstruktion

Bachelorarbeit

Generierung von Systemtests für Web-basierte Informationssysteme

System Test Generation
for Web-Based Information Systems

Michael Krein

März, 2012

Gutachter:
Prof. Dr. rer. nat. Horst Lichter
Prof. Dr. rer. nat. Bernhard Rumpe

Betreuer:
Dipl.-Inform. Matthias Vianden

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Aachen, den 16. März 2012

— Michael Krein —

Inhaltsverzeichnis

Abbildungsverzeichnis

Quelltextverzeichnis

1. Einleitung

Inhalt

Bei der Erstellung von interaktiven Anwendungen wie z.B. Internetseiten muss der dafür zuständige Softwareentwickler sicherstellen, dass die von ihm programmierte Anwendung so arbeitet wie es in der dazugehörigen Anforderungsspezifikation vorgegeben ist. Eine Überprüfung, ob dies zutrifft, geschieht mit Hilfe von Testfällen, welche die Funktionalität des Programms nachbilden.

1.1. Überblick

Im folgenden Überblick werden die Aufgabenstellung, die Ziele der Arbeit sowie deren Umsetzung vorgestellt.

1.1.1. Aufgabe

Web-basierte Informationssysteme wie z.B. Htmlseiten bestehen aus einer Menge z.T. ineinander verschachtelter Seitenelemente. Je nach Elementtyp ist es möglich, damit durch Eingaben des Benutzers wie z.B. Anklicken zu interagieren, während die darauf folgende Veränderung der Htmlseite als Ausgabe betrachtet wird. Testfälle für solche Informationssysteme bestehen aus einer Abfolge von Interaktionen mit Htmlelementen und einer Überprüfung des daraus entstehenden Htmlseitenaufbaus. Für eine korrekte Ausführung der Testfälle muss die Beschreibung der einzelnen Schritte und vor allem der Überprüfung auf das erwartete Verhalten vollständig spezifiziert sein und darf keinen Interpretationsspielraum zulassen.

Jedes Htmlelement muss eindeutig erkennbar sein und es müsste genau festgelegt werden, was nach welchem Schritt auf der resultierenden Htmlseite vorhanden sein muss. Grundsätzlich wird das erwartete Verhalten eines Programms in der Anforderungsspezifikation festgehalten, welche in der Analysephase der Projektentwicklung geschrieben wurde, also noch vor der Implementierung des

Programms. Für die gegebene Problemstellung wäre eine Anforderungsspezifikation nötig, die für konkrete Testfälle genaustens den Testablauf und das dabei erwartete Sollverhalten beschreibt. Daraus resultiert die Aufgabe, den Inhalt der Anforderungsspezifikation formal so aufzuschreiben, dass sich aus ihm direkt ausführbarer Testcode generieren lässt. Mit diesem Code soll überprüfbar sein, ob das Informationssystem die aufgestellten Anforderungen erfüllt.

1.1.2. Ziele

Ziel der Codegenerierung ist es, dem Programmierer von web-basierten Informationssystemen für deren Testfälle eine Möglichkeit zur Verfügung zu stellen, lauffähigen Programmcode zu erzeugen, welcher dem Verhalten dieser Testfälle entspricht und automatisch ausgeführt werden kann. Die Tests sollen konkrete Anweisungen befolgen und das Resultat der dadurch veränderten Seite durch Überprüfung des Seitentexts und der Seitenelemente kontrollieren.

Um weiterhin möglichst viele verschiedene Testfälle abdecken zu können, muss für die Codegenerierung eine möglichst große Menge an Htmlelementtypen und Interaktionsmöglichkeiten entsprechend definiert sein. Die generierten Codefragmente sollen funktionsmäßig identisch zu den einzelnen Testschritten des Systemtests sein. Damit der Aufbau von Anweisungen möglichst kurz ist und ein und dieselben Codeelemente für unterschiedliche Aufgaben wiederverwendet werden können, sollten einige Anweisungen zu Methoden zusammengefasst werden. Methoden sollten möglichst so konstruiert werden, dass sie für unterschiedliche Testfälle wiederverwendbar sind. Auch eine Möglichkeit für Methoden, sich gegenseitig aufrufen zu können, fördert die Übersichtlichkeit des erstellten Codes.

Die Anforderungsspezifikation wird bereits in der Analysephase eines Projekts erstellt, bevor jeglicher Programmcode implementiert wurde [6]. Damit aus dieser Anforderungsspezifikation Testfälle für das fertige Informationssystem generiert werden können, müssen die Testfälle so aufgebaut sein, dass sie möglichst wenig Wissen über die genaue Projektimplementierung voraussetzen.

1.1.3. Arbeit am EJB-Gen

Die Arbeit wird als Teil des Generators für Eclipse Java Beans (kurz EJB-Gen) realisiert. Der EJB-Gen unter der Bezeichnung „Gargoyle“ ist dafür zuständig, für verschiedene Stadien der Codegenerierung entsprechenden Sourcecode zu generieren [7]. Hierbei wird für die verschiedenen Stadien der Softwareentwicklung jeweils passender Code generiert, der z.T. auf bereits generierten Code vorhergehender Stadien zugreift. Diese Bachelorarbeit beschäftigt sich dabei mit der

Generierung von Fassaden für (System-)Testfälle eines Informationssystems.

1.2. Überblick über die Ausarbeitung

Zunächst wird im Grundlagenteil der Ausarbeitung erläutert, was Systemtests sind und welchen Programme für die Codegenerierung solcher Systemtests benutzt werden. Nach der Vorstellung dieser Programme folgt im Praxisteil schrittweise, wann bei der Generierung welches Programm im Einzelnen benutzt wurde und wie sich die Gestalt der einzelnen Methoden und Konzepte im Laufe der Entwicklung wandelte.

Im Kapitel der Evaluation wird untersucht, inwieweit die Umsetzung der angesetzten Ziele gelang. Schließlich folgt eine Zusammenfassung der Ergebnisse der Bachelorarbeit und ein Ausblick für daran anknüpfende Erweiterungsmöglichkeiten.

1.3. Danksagungen

An dieser Stelle möchte ich all jenen danken, die mir bei dieser Bachelorarbeit geholfen haben. Ich bedanke mich bei Prof. Dr. rer. nat. Horst Lichter für die Erstkorrektur und dafür, dass diese Bachelorarbeit erst ermöglicht wurde.Für die Zweitkorrektur bedanke ich mich bei Prof. Dr. rer. nat. Bernhard Rumpe. Ein sehr großes Danke geht an meinen Betreuer Dipl.Inform. Matthias Vianden, dessen tatkräftige Unterstützung mir bei der Entwicklung der Bachelorarbeit sehr weitergeholfen hat.

2. Grundlagen

Inhalt

Dieses Kapitel erläutert die Grundlagen, auf denen die Codegenerierung für Systemtests im Rahmen dieser Bachelorarbeit basiert. Außerdem wird auf den technischen Aufbau der dafür verwendeten Programme eingegangen.

2.1. Systemtest

Im Rahmen dieser Bachelorarbeit werden Systemtests für ein auf Html basierendes Informationssystem generiert. In der Softwaretechnik beschreiben Systemtests eine Überprüfung des fertig implementierten Systems gegen die vollständige Anforderungsspezifikation [10]. Die betrachteten Testfälle sind typischen Anwendungssituationen nachempfunden, unter denen das fertige Informationssystem vom Anwender benutzt wird. Im Gegensatz zu den ähnlich aufgebauten Abnahmetests, bei welchen der Abnehmer die Software an eigenen Testfällen überprüft, wird beim Systemtest die Überprüfung des Programms durch dessen Entwickler bzw. Tester anhand typischer Testfälle selbst durchgeführt.

Systemtests gehören zu den funktionalen Testverfahren. Dies bedeutet, dass der Benutzer eine Liste von Eingabedaten an das Programm übergibt und deren Ausgabe mit den (vorher in der Anforderungsspezifikation beschriebenen) erwarteten Testdaten vergleicht. Für solche Vergleiche wird ausschließlich die Ausgabe des Programms überprüft. Eine Betrachtung der programminternen Abläufe findet bei Systemtests nicht statt. Tests, die auf diese Art arbeiten, werden als „Black Box“-Tests bezeichnet, zu denen auch der Systemtest gehört [10].

2.2. Systemtest-Codegenerierung

Lauffähiger Code wird generiert, indem zunächst ein abstraktes Metamodell möglicher Code-Komponenten und deren Beziehungen zueinander formal beschrieben wird. Anhand dieses Metamodells werden Regeln aufgestellt, wie aus einzelnen Komponenten fertiger Code entstehen soll. Mit diesen vorgegebenen Regeln und konkreten Modellen vom Typ des Metamodells wird schließlich Programmcode generiert. Modelle beschreiben dabei den Aufbau und die Verwendung eines nachzubildenden Informationssystems [12].

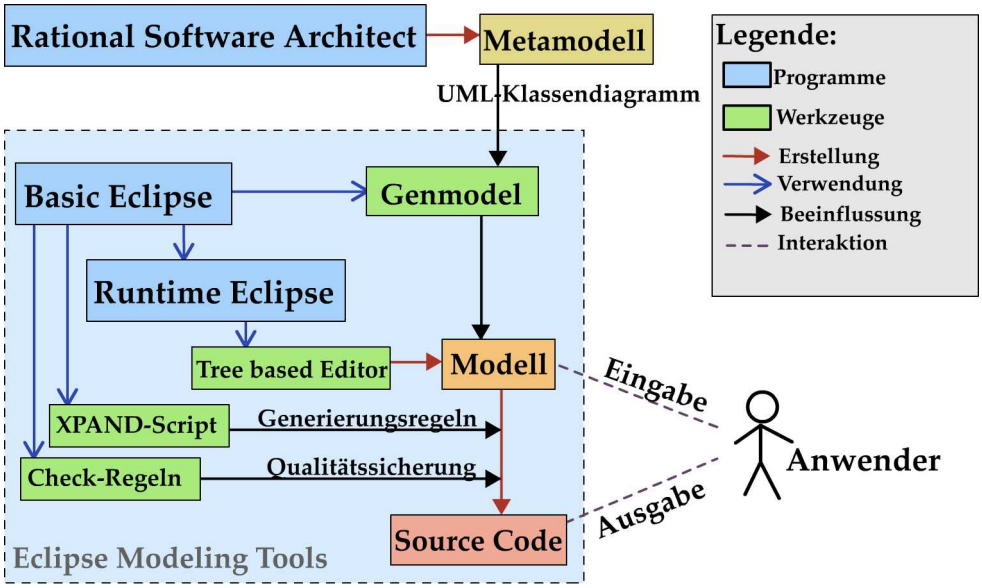


Abbildung 2.1.: Entwicklung der Codegenerierung

Die Entwicklung der Codegenerierung für Systemtests wird in Abb.2.1 bildlich dargestellt. Diese Generierung geschieht dabei im Falle dieser Bachelorarbeit mit einer Reihe von an Java Eclipse angepassten Werkzeugen. Für die Erstellung des Metamodells wird das Programm „Rational Software Architect“ verwendet. Das dabei erzeugte Metamodell in Form eines UML-Klassendeagramms wird vom äußeren Basic Eclipse geladen und mittels Genmodel bearbeitet.

Während der Entwicklung des Projekts wurde aus dem äußeren Eclipse heraus ein inneres Runtime Eclipse gestartet, in dem mit Werkzeugen wie dem Tree-based Editor konkrete Modelle vom vorher bearbeiteten Metamodell erstellt werden. Aus diesen Modellen wird anhand von XPAND-Anweisungen schließlich Java-Sourcecode generiert. Um zusätzlich sicherzugehen, dass der so entstandenen Code bestimmte Vorgaben erfüllt, kann man Regeln aufstellen, an die sich ein erstelltes Modell für die Codegenerierung halten muss. Dies wird hier durch in der Programmiersprache Check geschriebene Regeln umgesetzt, auf die das Modell während der Generierung überprüft wird.

Der Endanwender arbeitet nur mit Modellen, aus denen er nach fertig erstellten

Regeln Programmcode generiert, während er auf den Aufbau des Metamodells bzw. die darauf bezogenen Bearbeitungsregeln keinen Einfluss hat. Auf die genaue Arbeitsweise und Aufgaben der einzelnen hier erwähnten Programme wird in den entsprechenden Kapiteln des Grundlagen- und Praxisteils näher eingegangen.

2.2.1. Testfassade

Beim fertig generierten Code handelt es sich im Rahmen dieser Bachelorarbeit noch nicht um Aufrufe konkreter Testfällen. Stattdessen wird eine Javaklasse mit Methoden generiert, welche mit individuell belegbaren Eingabeparametern von Systemtests aufgerufen werden können. Der Aufbau der generierten Klasse, die statt konkreter Testschritte eine Menge an Methoden ausgibt, die nach konkreten Anforderungen angepasst und aufgerufen werden können, orientiert sich damit nach dem Entwurfsmuster einer „Fassade“ [3]. Daraus folgt die Benennung der erstellten Klasse als „Systemtestfassade“ und ihrer Methoden zu „Fassadenmethoden“.

Fassadenmethoden beschreiben das Verhalten eines Vorgangs innerhalb eines Testfalls. Je nach Belegung ihrer Übergabeparameter können sich die Fassadenmethoden hierbei unterschiedlich verhalten und damit von verschiedenen Testfällen mit entsprechend angepasster Variablenbelegung benutzt werden. Die auf Basis der Systemtestfassade generierte Java-Klasse wird für die Anwendung von Systemtests schließlich von einer Testklasse mit konkreten Testfällen verwendet, welche nacheinander Methoden aus der Systemtestfassadenklasse aufrufen.
Neben einer Liste von Fassadenmethoden enthält die Systemtestfassade zusätzlich eine Definitionen sämtlicher verwendeter Elemente der benutzten Htmlseite wie Buttons oder Links.

2.3. Webbasierte Informationssysteme

Im Rahmen dieser Bachelorarbeit wird Java-Programmcode für sogenannte webbasierte Informationssysteme generiert. Bei einem Informationssystem handelt es sich um eine Anwendung, bei der durch Benutzer zum System gehörende Daten verändert und gespeichert werden können [5]. Webbasierte Informationssysteme sind im Internet verfügbar und können dort z.B. über Webbrowsern angesteuert und bedient werden. Ein Beispiel für ein solches Informationssystem ist ein Forum mit anlegbaren Threads und Usern. Die Bedienoberfläche solcher untersuchten Informationsysteme ist in Html geschrieben.

2.4. Verwendete Werkzeuge

Folgender Abschnitt beleuchtet den Aufbau und die Funktionsweise der Werkzeuge, welche in den einzelnen Stadien der Entwicklung Anwendung fanden.

2.4.1. IBM Rational Software Architect

Der „IBM Rational Software Architect“ (kurz RSA) ist ein auf Eclipse basierendes Werkzeug zur Erstellung von UML-Klassendiagrammen [8]. Die einzelnen Objekte, aus denen ein fertiges Modell besteht, basieren auf einzelnen Klassen eines als Metamodell verwendeten Klassendiagramms. Solche Diagramme für Codegenerierung müssen eine Klasse als Wurzelelement besitzen, aus dem sich bei einem Modell Objekte aller anderen Klassen hierarchisch ableiten lassen.

2.4.2. Eclipse

Sämtliche Programmierung im Rahmen dieser Bachelorarbeit fand mit Hilfe der Eclipse Modeling Tools statt.

Dazu zählt neben dem auf Java basierendem RSA vor allem das eigentliche Eclipse, welche sich grob in zwei Bereiche einteilen lässt. Das standardmäßig startende, „äußere“ Eclipse dient der Verwaltung des aus dem RSA generierten UML-Metamodells. Nachdem ein solches UML-Metamodell geladen wird, werden zu den einzelnen UML-Klassen mit ihren jeweiligen Eigenschaften und Abhängigkeiten äquivalent aufgebaute Javaklassen erzeugt.

Diese Umsetzung erfolgt mittels des Genmodels, welches für die einzelnen Klassen des Metamodells Code für gleichnamige Javaklassen generiert, durch den die Attribute, Relationen und sonstigen Eigenschaften der Klassen nachgebildet werden. Es ist möglich, den automatisch generierten Code zu überschreiben und ihn durch eigene Java-Anweisungen zu ersetzen. Sämtliche Informationen über den vom Genmodel generierten Code werden in einer Ecore-Datei festgehalten, die vom Genmodel während der Codegenerierung miterzeugt wird. In der Ecode-Datei können für die Klassen noch spezielle, für ihre spätere Anwendung im Modell relevanten Eigenschaften bearbeitet werden.

Nachdem die Bearbeitung der einzelnen Klassen abgeschlossen ist, wird das Ecore-File, in welchem sämtliche Eigenschaften für die Bearbeitung des Metamodells abgespeichert sind, in ein Paket kopiert, in dem XPAND-, XTEND- und CHECK-Dateien auf dieses Metamodell angepasste Regeln aufstellen.

Aus dem äußeren Eclipse heraus wird ein inneres „Runtime“-Eclipse gestartet. Im Runtime Eclipse hat der Endanwender die Möglichkeit, Modelle vom Typ des Metamodells mit Hilfe des Treebased Editors für konkrete Testfassaden zu erstellen. Aus den Modellen lässt sich Javasourcecode in ein vorher bestimmtes Verzeichnis generieren.

2.4.3. XPAND

Bei XPAND handelt es sich um das „Modell zu Text“-Transformationsrahmenwerk für die Codegenerierung. Damit ist gemeint, dass in XPAND Anweisungen erstellt werden, nach denen aus der Beschreibung eines Modells im Runtime Eclipse fertiger Java-Code generiert werden kann [11]. Sowohl das Modell im Runtime Eclipse als auch die XPAND-Anweisungen müssen hierfür auf demselben Metamodell basieren.

XPAND-Anweisungen beziehen sich auf einzelne Klassen des Metamodells, für welche sie Regeln aufstellen, wie der aus ihnen generierte Code aussehen soll [4]. Dieser Code kann durch Attribute und Funktionen der jeweiligen Klasse bzw. der für diese Klasse sichtbaren anderen Klassen erweitert werden. Auch ist es währenddessen möglich, andere XPAND-Anweisungen aufzurufen. Sollten hierbei mehrere Anweisungen unter dem selbem Namen für verschiedene Klassen erstellt worden sein, so wird die am meisten spezialisierte noch zum ausgewählten Element passende Anweisungen ausgeführt.

XPAND generiert innerhalb einer Anweisung für eine Beispielklasse beliebigen Text. Es wird in der Anweisung lediglich geprüft, ob sämtliche in der Anweisung verwendeten Attribute und andere Klassen tatsächlich aus Sicht der Beispielklasse verfügbar sind. Prüfungen auf die Korrektheit des erzeugten Codes aus Sicht eines Java-Compilers finden nicht statt.

2.4.4. Modellüberprüfungen mit Check

Durch das im RSA erstellte Metamodell und dessen Umsetzungsanweisungen in XPAND ist es möglich, Programmcode aus einem Modell zu erzeugen. Für diesen Code muss sichergestellt werden, dass er fehlerfrei ist und korrekt die geforderten Funktionalitäten umsetzt. Dies ist jedoch nur der Fall, wenn der Aufbau des Modells, aus dem der Code generiert wurde, sich selbst an bestimmte Vorgaben hält.

Für diese Vorgaben werden der Sprache Check Regeln aufgestellt, wie genau die im Treebased-Editor erstellten Modelle aufgebaut sein sollen. Check besteht aus Aufrufen von einzelnen solchen Regeln, welche Klassen eines Modells nach einer

bestimmten Bedingung untersuchen [11]. Eine Regel betrifft immer eine Klasse und wird auf alle Elemente dieser Klasse sowie der dazugehörigen Unterklassen angewandt.

Einzelne Regeln bestehen aus einer Fehler bzw. einer Warnmeldung, welche Art Fehler vorliegt sowie einer auf die jeweilige Klasse bezogenen logischen Aussage. Ist diese Aussage für ein Objekt der Klasse verletzt, so wird für das Modell bei der Codegenerierung ein vorher festgelegter Fehler- bzw. Warnmeldungstext zusammen mit der betroffenen Klasse ausgegeben. Sowohl beim ausgegebenen Text als auch in der logischen Aussage können für die geprüfte Klasse sichtbare Attribute und Methoden verwendet werden.

2.4.5. XTEND

Bei XTEND handelt es sich um eine funktionale Programmiersprache [11]. Im Rahmen dieser Bachelorarbeit wird XTEND für die Erstellung von Methoden verwendet, welche durch rekursive Aufrufe die Korrektheit bestimmter Aussagen für die Klassen eines Modells überprüfen. Solche XTEND-Methoden werden durch in Check modellierte Regeln verwendet, weil Check selbst über keine Möglichkeiten verfügt, rekursive Tests anzuwenden.

2.4.6. Tree-based-Editor

Im Runtime Eclipse besitzt der Anwender die Möglichkeit, konkrete Modelle vom Typ des zuvor definierten Metamodells zu erstellen. Dies geschieht mit Hilfe eines Tree-based Editors, welcher den hierarchischen Aufbau des Modells festhält. Modelle besitzen ein Wurzelelement von welchem aus neue Klassen als Kinder- oder Geschwisterelemente bestehender Modellelemente eingetragen werden können. Aus dem so entstehenden hierarchischen Aufbau des Modells leitet sich die Bezeichnung des damit arbeitenden, „auf einer Baumstruktur basierenden“ (engl. Tree-based) Editors her.

Für die einzelnen Modellelemente lassen sich die dazugehörigen Attribute belegen und Relationen mit bereits bestehenden Objekten aufbauen. Ein Modellelement basiert auf dem Typen einer zugehörigen Klasse des UML-Metamodell mit all deren Eigenschaften und Relationen. Die einzelnen Elementtypen werden optisch durch verschiedenfarbige Rautenelemente auseinander gehalten. Die einzelnen Elemente werden in der Baumstruktur des Tree-based-Editors unter der Bezeichnung des bei ihnen als „Label Feature“ gesetzten Attributs angezeigt. Mit dem Tree-based Editor erstellt der Anwender ein konkretes Modell. Ein solches Modell ist der Aufbau einer bestimmten Testfassade, welche aus einer Menge von Fassadenmethoden und Seitenelementen besteht.
Bei der abschließenden Generierung von Code kommt das „Model Transforma-

tion Framework“ (kurz MTF) zum Einsatz, welches anhand des Aufbaus des Modells einer Systemtestfassade den dazugehörigen Programmcode nach vorher in XPAND definierten Regeln generiert [9].

2.5. Vergleichbare Arbeiten

Die grundsätzliche Idee, für verschiedene Phasen einer Projekterstellung automatisch Code zu generieren wurde bereits in verschiedenen Projekten neben dem EJB-Gen durchgeführt.

Eine vergleichbare Generierung von Systemtests, also Generierung von Testfällen durch die Beschreibung der verwendeten Elemente und Methoden, erfolgt ebenso beim „Canoo WebTest“ [13]. Mittels WebTest lassen sich konkrete Anweisungen definieren, die einzelne Testschritte beim Interagieren mit einem Informationssystem beschreiben. Bei der Systemtestgenerierung in dieser Bachelorarbeit sollen dagegen Fassadenmethoden definiert werden, die selbst unter einer konkreten Eingabeparameterbelegung aufgerufen werden. Dem Anwender soll ferner bei der Systemtestgenerierung die Möglichkeit zur Verfügung stehen, innerhalb einer Methode andere Methoden aufzurufen. Durch integrierte Checküberprüfungen wird die korrekte Anwendung davon sichergestellt. Die Möglichkeit, andere Methoden in der nach konkreten Anweisungen spezifizierten Generierung aufzurufen, konnte nicht in der geprüften WebTest-Version festgestellt werden. Zusätzlich sollen Befehle und Methodenaufrufe in der Systemtestgenerierung durch Definition von Eingabeparametern leichter für verschiedene Anwendungsfälle wiederverwendbar sein. Folglich soll die Codegenerierung dieser Bachelorarbeit die Funktionalitäten von Informationssystemen möglichst übersichtlich und wiederverwendbar aufgebaute Methoden überprüfen können, weil verglichen mit WebTests mehr Möglichkeiten zur Modellierung dieser Methoden zur Verfügung stehen.

3. Systemtest-Aufbau

Inhalt

Wie bereits in den Grundlagen beschrieben, werden in einem Systemtest das voll funktionsfähige Programm gegen die komplette Liste der aufgestellten Anforderungen getestet. Die Anforderungen sind dabei durch die Form einzelner Testfälle beschrieben, sodass sich aus ihnen direkt entsprechender Programmcode generieren lässt. Die folgenden Abschnitte zeigen hierfür, wie die einzelnen Schritte eines Systemtest aussehen könnten und wie sich daraus ein entsprechender Testfall nachbilden lässt.

3.1. Systemtest für ein Forumsystem

Bei der Umsetzung der Systemtestgenerierung wurde als Beispiel ein Forumsystem herangezogen. In diesem Forumsystem lassen sich Foren und User erstellen sowie für bestehenden Foren User zuweisen. Erstellte User und Foren werden in eigenen Tabellen in der Hauptseite angezeigt und können ausgewählt und bearbeitet werden. Die Benutzeroberfläche des Forumsystems basiert auf Html.

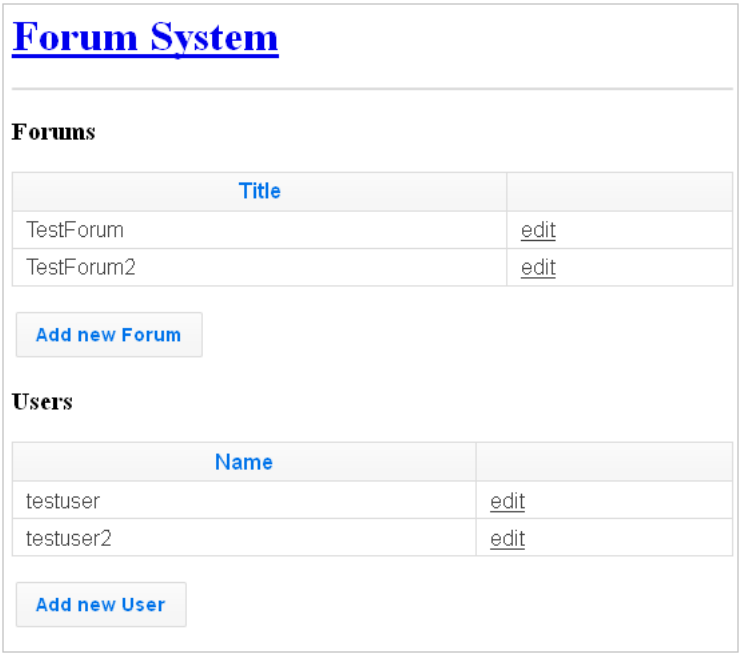


Abbildung 3.1.: Hauptseite eines Forumsystems mit zwei Foren „TestForum“ und „TestForum2“ sowie zwei Usern „Testuser“ und „Testuser2“

Anhand der Beschreibung des Ablaufs einiger Testmethoden für dieses Forumsystems soll gezeigt werden, welche Schritte in einem Systemtest für entsprechende Testfälle nacheinander durchgeführt werden müssten. Daraus ließe sich eine Regelmäßigkeit ableiten, wie Systemtestfälle durchgeführt werden und nach welchen Gesichtspunkten dafür eine entsprechende Systemtestfassade generiert werden kann.

Als Testfall soll im folgenden Beispiel in ein noch leeres Forensystem ein neues Forum unter dem Titel „TestForum“ angelegt werden. Bei den zugehörigen Abbildungen ist der überprüfte Text der Html-Seite orange umrahmt, während mit roten Zahlen die einzelnen Interaktionsschritte vermerkt sind.

Der Testfall leitet sich hierbei aus der Anforderung ab, im Forumsystem neue Foren erstellen zu können. Der dafür benötigte Forumtitel soll vom Anwender beliebig gewählt werden können. In der Anforderungsspezifikation ist zusätzlich festgehalten, dass ein Forumtitel von maximal einem Forum verwendet werden darf. Um dieses Verhalten zu überprüfen, muss in einem Testfall zweimal ein Forum mit gleichen Titel generiert werden. Bei der zweiten Generierung wird dabei eine entsprechende Fehlermeldung erwartet.

3.1.1. Methode „Forum erstellen“

Die erste modellierte Methode befasst sich mit den Schritten zur Erstellung eines Forums. Mit dieser Methode wird überprüft, ob die einzelnen Schritte zur Erstellung eines neuen Forums sich im fertigen Programm fehlerfrei durchführen lassen. Eine Überprüfung, ob das Forum erfolgreich angelegt werden konnte, findet dagegen in anderen Methoden statt.

Die Methode startet im Hauptmenü des Forumsystems, wo durch Anklicken des Buttons „Add new Forum“ ein Pop-up-Dialog zur Forumerstellung aufgerufen wird. In diesem Dialog wird der Forumtitel im dafür vorgesehenen Eingabefeld eingetragen und die Forumerstellung wird durch einen Klick auf einen ebenfalls als „Add new Forum“ beschrifteten Button des Pop-up-Dialogs abgeschlossen.

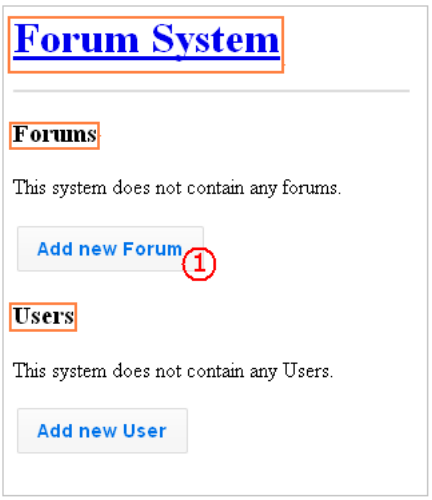


Abbildung 3.2.: Hauptseite eines leeren Forumsystems

Für die Methode wird zunächst angenommen, dass der Anwender im Hauptmenü des Forumsystems startet. Beispiele für den Aufbau einer solchen Hauptseite sind in den Abbildungen 3.1 und 3.2 abgebildet. Als Erstes muss somit in der Methode sichergestellt werden, dass sich der Anwender tätsächlich auf der Hauptseite des Forumsystems befindet. Um dies sicherzustellen, wird die Seite nach den in Abb.3.2 umrahmten Schlüsselwörter „Forum System“, „Forums“ sowie „Users“ durchsucht.

Der nächste Schritt ist das Anklicken des Buttons „Add new Forum“, welcher in Abb.3.2 mit (1) markiert ist.

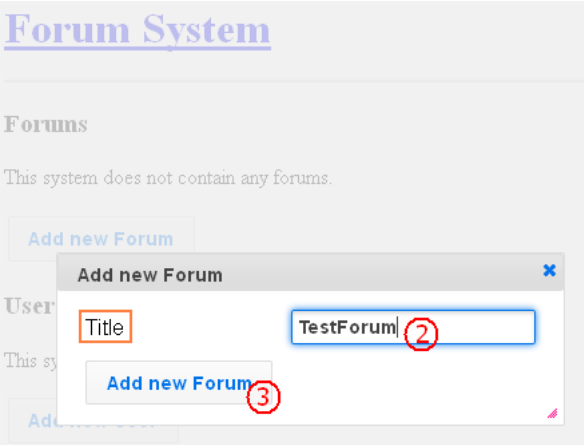


Abbildung 3.3.: Forumerstellung-Popup-Dialog

Es wird erwartet, dass ein Pop-up-Dialog erscheint, in dem der Titel des Forums bei (2) eingegeben werden kann, was durch einen Klick auf den Knopf (3) bestätigt wird. Zunächst wird wieder untersucht, ob das Schlüsselwort „Title“ als Indiz darauf, dass sich der Anwender auf der richtigen Seite befindet, auf der Seite gefunden werden kann. Danach wird das Eingabefeld bei (2) annavigiert und dort der Forumtitel „TestForum“ eingegeben. Abgeschlossen wird die Forenerstellung durch einen Klick auf den „Add new Forum“-Button der Dialogbox bei (3). Trotz gleicher Beschriftung handelt es sich bei den Buttons (1) und (3) um zwei verschiedene Elemente, welche eindeutig auseinander gehalten werden müssen.

3.1.2. Erwartetes Resultat

Im vorhergehenden Abschnitt wurde gezeigt, welche Schritte für eine Methode aus Sicht eines Systemtests vorgenommen werden müssen, um ein Forum zu erstellen. Was in dieser Methode fehlt, ist die abschließende Überprüfung, inwieweit diese Erstellung gelang. Es folgt nun ein Aufbau zweier Methoden, die nach Erstellung eines Forums aufgerufen werden können. Diese Methoden überprüfen, ob das Anlegen des Forums erfolgreich war bzw. ob dabei ein zu erwartender Fehler aufgetreten ist.

3.1.3. Methode „Check ob Forum erfolgreich angelegt“

Diese Methode dient der Sicherstellung, dass durch eine Forumerstellung ein Forum erfolgreich im System angelegt wurde. Sie wird nach der Erstellung eines Forums aufgerufen und überprüft die Seite nach Schlüsselwörtern, aus denen sich der Erfolg der Forumerstellung für den Testfall folgern lässt.

Die resultierende Seite für dieses Beispiel ist in Abb.3.4 dargestellt. Es wird erwartet, dass sich der bestätigende Text „Added a forum to the system.“ auf der Seite befindet, sowie ein Text mit der Aufschrift „Forum:TestForum“.

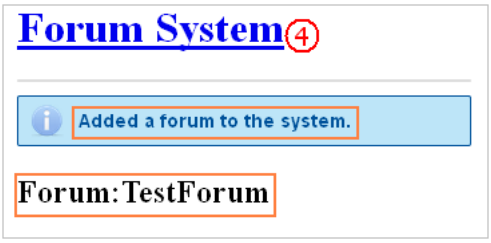


Abbildung 3.4.: Menü des erfolgreich erzeugten Forums (Ausschnitt)

Mit einem abschließenden Klick auf die verlinkte Überschrift „Forum System“ bei (4) kann der Anwender zurück ins Hauptmenü gelangen, wo das „TestForum“ in der Liste vorhandener Foren zu sehen ist (siehe Abb.3.5).

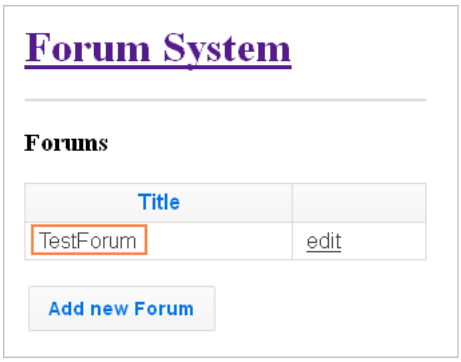


Abbildung 3.5.: Hauptseite nach Forumerstellung (Ausschnitt)

3.1.4. Methode „Check ob Forumname bereits vorhanden“

Nach der Anforderungsspezifikation für das Forumsystem darf jeder Forumtitel nur einmal belegt durch ein Forum werden. Sollte der Anwender versuchen, ein neues Forum unter einem bereits belegtem Titel zu erstellen, so wird erwartet, dass das System in einer Meldung dem Benutzer mitteilt, dass wegen des Namensdublikats kein neues Forum angelegt wurde. Eine Methode, die dieses Verhalten untersucht, besteht folglich aus einer Untersuchung des Seitentexts nach der entsprechenden Fehlermeldung.

Die Methode besteht damit aussschließlich aus der Überprüfung, ob sich der dafür vorgesehene Warntext „There already exists a similar Forum in the database.“ finden lässt.

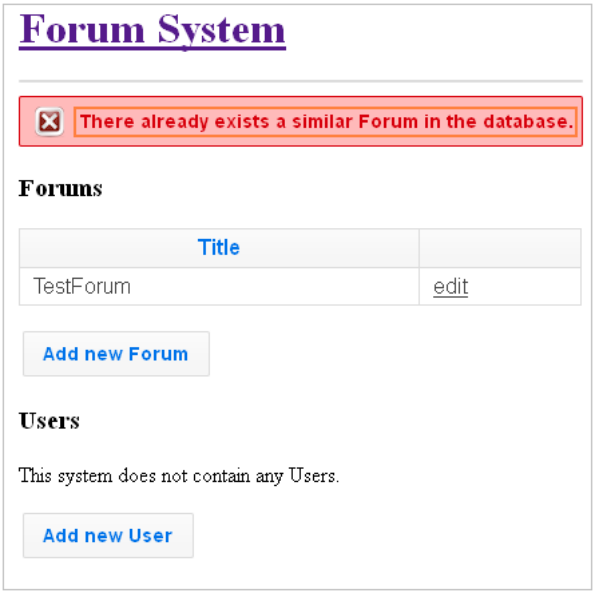


Abbildung 3.6.: Forumdublikatswarnung

3.1.5. Folgerungen für Systemtests

Systemtests rufen nacheinander konkrete einzelne Fassadenmethoden auf, wobei diese Methoden danach geprüft werden, ob ihre einzelnen Schritte ein festgelegtes Resultat tatsächlich erzielen. Nach der vorhergehenden Beschreibung der drei Methoden für das Forumsystem würde ein einzelner Systemtest z.B. daraus bestehen, ein Forum zu erstellen (Abschnitt 3.1.1), den Erfolg der Erstellung durch die entsprechende Methode sicherzustellen (Abschnitt 3.1.3), nach Navigation ins Hauptmenü erneut die Forumerstellung zu durchlaufen (Abschnitt 3.1.1) und danach die Methode zur Forumdublikatserkennung (Abschnitt 3.1.4) aufzurufen.

Bei der Beschreibung der verwendeten drei Methoden wurde deutlich, dass sich bestimmte Muster des Testablaufs fortlaufend wiederholen. In einem Testschritt wird die aktuell ausgewählte Htmlseite zunächst nach bestimmten Schlüsselworten und Elementen abgesucht. Danach werden einzelne Elemente angesteuert und die daraus resultierende Seite als aktuell verwendete Seite geladen.

Der Aufbau eines Informationssystem lässt sich damit anhand einer Menge an Elementen beschreiben, welche von einzelnen Methoden des Systems verwendet werden. Es ist für die Codegenerierung vorteilhaft, wenn Methoden möglichst flexibel definiert sind, sodass eine Methode mit variablen Verhalten von verschiedenen konkreten Testfällen mit entsprechenden Aufforderungen benutzt werden kann. So erstellt die Methode aus Abschnitt 3.1.1 immer ein Forum mit dem

Forumtitel „TestForum“. Wenn es die Anforderungsspezifikation erfordert, Methoden zur Forenerstellung mit anderen Forumtiteln aufzurufen, müsste nach dem Schema für jeden anderen Forumtitel eine eigene Methode definiert werden. Dies kann leicht dadurch verhindert werden, dass Methoden Eingabeparameter besitzen, welche beim Aufruf der Methode an den entsprechenden Testfall angepasst sind. Für die Methode zur Forenerstellung wäre dann ein solcher Eingabeparameter der Titel des Forums, der dann anstelle der konstanten Eingabe von „TestForum“ in das Eingabefeld bei (2) eingetragen wird.

Auch die Methode zur Überprüfung der erfolgreichen Anlegung eines Forums in Abschnitt 3.1.3 ließe sich durch einen Eingabeparameter für unterschiedliche Testfälle wiederverwendbar machen. Dafür müsste in der Methode der Seitentext anstelle von „Forum:TestForum“ nach „Forum:“+den Eingabeparameter untersucht werden.

Im Folgenden wird darauf eingegangen, welche zusätzlichen Informationen über einen Testfall für die Codegenerierung benötigt werden und welche Probleme bei der Generierung einzelner Testschritte entstehen können. Für jeden Schritt eines Systemtestfalls muss zunächst sichergestellt werden, dass er erfolgreich durchgeführt werden konnte. Anderenfalls wird der Testfall abgebrochen und gilt als fehlgeschlagen.

In den beschriebenen Methoden wird mittels einer Überprüfung bestimmteter Schlüsselwörter sichergestellt, dass sich der Anwender auf der richtigen Seite befindet. Auch wenn es durchaus möglich ist, dass auch andere Seiten existieren, in deren Seitentext die notwendigen Schlüsselwörter vorkommen, kann durch ein solches erstes Überprüfen ein Test bereits als fehlgeschlagen abgebrochen werden, falls sich die gesuchten Begriffe doch nicht auf der aktuell annavigierten Seite befinden sollten.

Eine Alternative zur Sicherstellung, dass sich der Anwender auf der Hauptseite des Forumssystems befindet, wäre die Untersuchung der URL der aktuell geladenen Seite. Eine andere Alternative wäre, neben dem angezeigten Seitentext den gesamten Quelltext der Seite nach bestimmten Vorgaben zu überprüfen.

Für die Interaktion mit Seitenelemente müssen diese eindeutig für einen Testfall identifizierbar sein. Der Html-Quellcode der Seite erlaubt hierzu eine Identifikation von Elemente anhand einer ID. Es ist alternativ auch möglich, Elemente anhand anderer eindeutiger Eigenschaften wie ihrer Position in einer Tabelle zu finden.

Es muss sichergestellt werden, dass auf der Htmlseite ein Seitenelement unter der gesuchten ID gefunden wird. Dies ist nicht der Fall, wenn die ID des Sei-

tenelements falsch belegt ist, oder aber auch wenn zu Beginn eine andere Seite annavigiert wurde. Wurde dagegen ein Element gefunden, wird überprüft, ob es sich hierbei um das Htmlelement handelt, welches im konkreten Testschritt tatsächlich ausgewählt werden sollte. Dabei wird geschaut, ob das Element vom korrekten Typ ist und mit einem bestimmten Text beschriftet ist. Erst wenn auch dies zutrifft, geht der Systemtest davon aus, das richtige Element gefunden zu haben.

3.2. Nachbildung der Testfälle

Nachdem nun gezeigt wurde, wie der Aufbau von Fassadenmethoden für Systemtests grundsätzlich funktioniert, folgt die Nachbildung des Forensystembeispiels mitsamt seiner Methoden in die Struktur eines möglichen Modells, wie es im Runtime Eclipse durch Werkzeuge wie den Tree-based Editor erstellt werden kann. Ziel ist hierbei, alle relevanten Elemente für eine Codegenerierung abzubilden.

Zunächst wird betrachtet, mit welchen Elementen (engl. Item) des Informationssystems bei der Erstellung eines Forums interagiert wurde. Es sind die mit „Add new Forum“ betitelten Buttons (1) und (3), sowie das Eingabefeld bei (2). Außerdem existiert zusätzlich der HtmlAnchor(=Link) in (4) zur Navigation an die Hauptseite des Forumsystems.

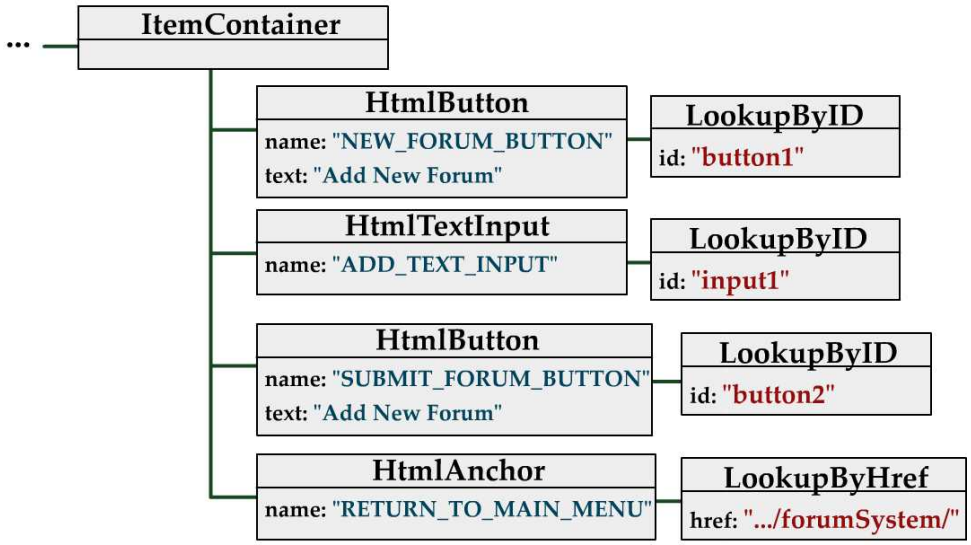


Abbildung 3.7.: Itemdefinition

In Abb.3.7 ist die Umsetzung der vier Interaktionselemente ins Modell zu sehen. Um in der Baumstuktur des Modells leicher gefunden werden zu können, sind Items als Kinderelemente eines Itemcontainers erstellt worden. Die Klasse eines einzelnen so erstellten Items repräsentiert dabei dessen Html-Datentyp (hier HtmlButton, HtmlTextinput bzw. HtmlAchor).

Jedes Htmlelement hat als Attribut einen eindeutigen Variablennamen und einen Beschriftungstext. Damit ein Htmlelement gefunden werden kann, muss dafür außerdem eine Strategie definiert sein, in der festgehalten wird, über welches Merkmal das Element gefunden werden kann.

Jedes Objekt der Elementklasse kann im Modell als Kindelement eine solche

Strategie besitzen. Beim HtmlAnchor zu (4) aus Abb.3.7 ist dies eine Suche nach seinem Ziellink(=Verweisziel [1]), welcher hier als „.../forumsystem“ vereinfacht dargestellt ist. Den anderen Elementen ist eine eindeutige ID zugeordnet, nach der sie gefunden werden können.

Es folgt die Modellierung der einzelnen vorgestellten Methoden. Analog zu Items und Itemcontainern werden Methoden innerhalb einzelner Systemtestfassaden erstellt. Zunächst wird die Methode Zur Erstellung eines Forum nachgebildet.

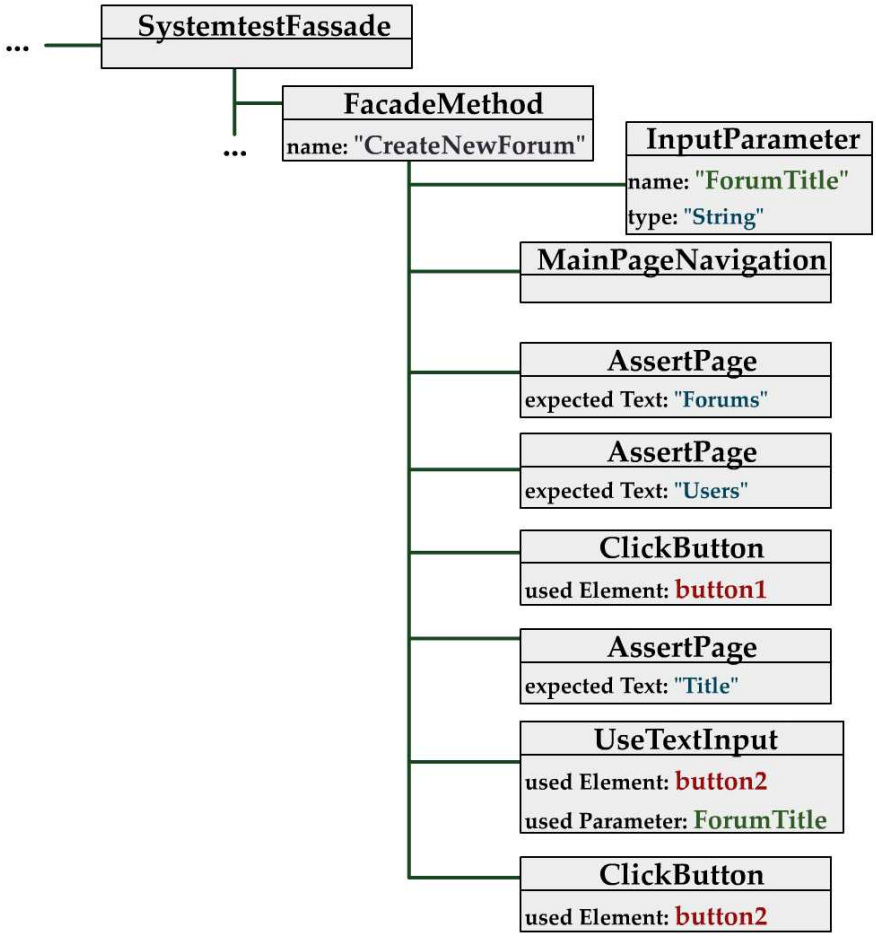


Abbildung 3.8.: Modellierung der Methode „Neues Forum erstellen“

Wie bereits im Kapitel 3.1.1. beschrieben, besteht die Methode „Neues Forum erstellen“ aus der Navigation von der Hauptseite (Abb.3.1) ins Popup-Fenster der Forumserstellung(Abb.3.2), dem dortigen Eintragen des Forumtitels sowie der abschließenden Bestätigung des Vorgangs. Zwischen den Interaktionen finden außerdem Überprüfungen des Seitentexts statt. Diese einzelnen Schritte werden durch Kinderelemente der Methode dargestellt. Der Aufgabenbereich einer einzelnen Anweisung wird durch den Typ ihrer Klasse definiert. Zuerst muss

die Seite mittels „MainPageNavigation“ auf die Hauptseite des Forumsystems annavigiert werden.

Elemente vom Typ „AssertPage“ prüfen hier, ob die Assertion(engl. Behauptung) zutrifft, dass sich im Text der aktuell geladenen Seite der als „expected Text“ definierte String finden lässt. Darauf folgende Interaktionen sind durch „ClickHtmlButton“ und „UseTextInput“ definiert. Dabei fällt auf, dass die in Abschnitt 3.1.1. beschriebenen Überprüfungen bezüglich der Korrektheit eines ausgewählten Seitenelements nicht modelliert werden. Diese Schritte werden automatisch bei einer Interaktion mitgeneriert.

Um die Methode, wie im letzten Abschnitt von Absatz 3.1.5 erwähnt, möglichst flexibel zu halten, wird bei der Texteingabe anstelle eines konstanten Werts der Inhalt des Eingabeparameters „ForumTitle“ eingegeben, welcher als Kindelement der Methode definiert wurde. In den Interaktionen stellen die rot beschrifteten Werte unter „Used Element“ hierbei das verwendete Htmlelement aus Abb.3.7 dar, welches durch dessen ID gekennzeichnet wurde.

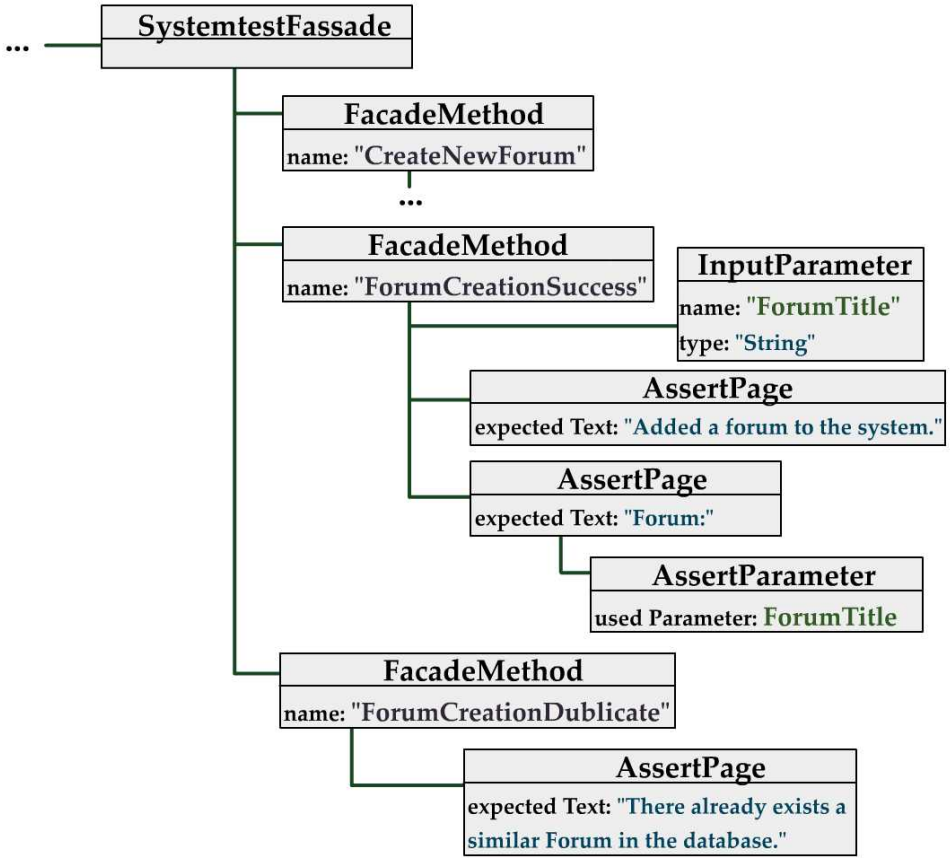


Abbildung 3.9.: Modellierung der verbleibenden Methoden“

Die beiden verbleibenden Methoden zur Überprüfung der Forumerstellung lassen sich in Abb.3.9 nach dem gleichen Prinzip nachbilden. Die Methode „Check

ob Forum erfolgreich angelegt“ besteht aus zwei Überprüfungen des Seitentexts, die durch „AssertPage“-Klassen simuliert werden. Nach dem zunächst die Seite nach dem Text „Added a forum to the system.“ abgesucht wird, soll die Untersuchung nach dem Forumtitel abhängig von der Belegung des Übergabeparameters „ForumTitle“ verlaufen. Um dies zu realisieren, wird für die dazugehörige „AssertPage“-Klasse als Kindelement eine „AssertParameter“-Klasse hinzugefügt, die auf den Übergabeparameter „ForumTitle“ der Methode zugreift. Daraus folgt die Überprüfung, ob die Htmlseite die Zeichenfolge „Forum:“+ForumTitle beinhaltet.

Die Methode „Check ob Forumname bereits vorhanden“ wird durch eine Überprüfung des Seitentexts nach der Fehlermeldung modelliert.

3.3. Forumsystem-Codegenerierung

Sobald ein Modell erstellt wurde, in dem alle benötigten Methoden beschrieben sind, wird aus diesem Modell Java-Code generiert. Für jede definierte Systemtestfassade wird dabei eine einzelne Java-Klasse erstellt. Der grobe Aufbau einer so generierten Java-Klasse hat folgende Form:

Quelltext 3.1: Allgemeiner Codeaufbau

```
package TestPackage;
        //verwendete Imports
Class TestClass{
        //Verwendete lokale Variablen
        // Definition der modellierten Seitenelemente
        // Definition der modellierten Methoden
}
```

Schrittweise wird nun erläutert, wie die einzelnen auskommentierten Bereiche jeweils konstruiert werden, damit der generierte Code lauffähig ist und die Funktionalität der Systemtestfassade umgesetzt wird.

3.3.1. Seitenelemente

Die einzelnen Werte zur Identifikation der einzelnen interaktiven Elemente werden in der erstellten Javaklasse als Konstanten definiert. Bezogen auf die Vorgaben des Modells würde diese Definiton folgendermaßen aussehen:

Quelltext 3.2: Verwendere Konstanten

```
private final String NEW_FORUM_BUTTON = "button1";
private final String ADD_TEXT_INPUT = "input1";
private final String SUBMIT_FORUM_BUTTON = "button2";
private final String RETURN_TO_MAIN_MENU = ".../forumsystem/";
```

3.3.2. Verwendete Variablen

Für die Seitennavigation in Methoden ist es nötig, den Zustand der aktuell geladenen Seite sowie ein Werkzeug zur Seitennavigation an vorgegebene Urls zwischengespeichert zu haben. Dies geschieht mit folgenden Definitionen:

Quelltext 3.3: Intern verwendete Variablen

```
private HtmlPage page;
private WebClient webclient = new WebClient();
```

3.3.3. Fassadenmethoden

Die einzelnen Fassadenmethoden werden durch analoge Java-Methoden abgebildet. Überprüfungen der Seite werden mittels „Assertions“ durchgeführt. Dabei wird geprüft ob eine vorher festgelegte Aussage zutrifft, anderenfalls wird der Test als fehlgeschlagen abgebrochen. Annavigierte Htmlelemente werden in lokalen Variablen der Bezeichung „used_Item“ zwischengespeichert. Auf das Auswählen eines Htmlelements folgen zunächst mehrere Überprüfungen durch „Assertionmethoden“ zur Sicherstellung dass das richtige Element ausgewählt wurde. Abschließend wird die in „page“ zwischengespeicherte Seite, auf die Interaktion mit dem ausgewählten Element aktualisiert. Die Methode zur Forumerstellung hätte damit folgenden Aufbau:

Quelltext 3.4: Javamethode zur Forumerstellung

```
public void CreateForum(String ForumTitle)
throws IOException,InterruptedException
{
    page = webclient.getPage(getApplicationBaseURLString());
    Assert.assertTrue("The Htmlpage expects the text \"Forums\".",
                page.asText().contains("Forums"));
    Assert.assertTrue("The Htmlpage expects the text \"Users\".",
                page.asText().contains("Users"));

    HtmlButton _used_Item_1 = (HtmlButton)
                page.getElementById(NEW_FORUM_BUTTON);
    Assert.assertNotNull("The Button \"NEW_FORUM_BUTTON\"
                was not found.",_used_Item_1);
    Assert.assertEquals("The Button \"NEW_FORUM_BUTTON\"
                needs to be labeled \" Add new User\".",
                "Add new User",_used_Item_1.asText());
    page = _used_Item_1.click();

    Assert.assertTrue("The Htmlpage expects the text \"Title\".",
                page.asText().contains("Title"));

    HtmlTextInput _used_Item_2 = (HtmlTextInput)
                page.getElementById(ADD_TEXT_INPUT);
                Assert.assertEquals("The input for the
                TextInput \"ADD_TEXT_INPUT\"
                must be a text input.",
```

```
                "text",_used_Item_2.getAttribute("type"));
    Assert.assertNotNull("The TextInput \"ADD_TEXT_INPUT\"
                was not found.",_used_Item_2);
    _used_Item_2.setValueAttribute(ForumTitle);

    HtmlButton _used_Item_3 = (HtmlButton)
                page.getElementById(SUBMIT_FORUM_BUTTON);
    Assert.assertNotNull("The Button
                \"SUBMIT_FORUM_BUTTON\" was not found.",
                _used_Item_3);
    Assert.assertEquals("The Button \"SUBMIT_FORUM_BUTTON\"
                needs to be labeled \" Add new User\".",
                "Add new User",_used_Item_3.asText());
    page = _used_Item_3.click();
}
```

Für die verbleibenden zwei Methoden zur Überprüfung des Resultats der Forumerstellung wird analog aufgebauter Javacode erstellt.

Quelltext 3.5: Javamethoden zur Forumüberprüfung

```
public void ForumCreationSuccess(String ForumTitle)
throws IOException,InterruptedException
{
    Assert.assertTrue("The Htmlpage expects the text
                \"Added a forum to the system.\".",
                page.asText().contains("Added a forum
                to the system."));
    Assert.assertTrue("The Htmlpage expects the text
                \"Forum:\""+ForumTitle+".",
                page.asText().contains("Forum:"+ForumTitle));
}

public void ForumCreationDublicate()
throws IOException,InterruptedException
{
    Assert.assertTrue("The Htmlpage expects the text \"There
                already exists a similar Forum in the database.\"
                .",page.asText().contains ("There already
                exists a similar Forum in the database."));
}
```

3.4. Schlussfolgerungen

Wie das Beispiel des Forumsystems gezeigt hat, lassen sich die Vorgänge eines Systemtests durch Aufrufe von Fassadenmethoden nachbilden, welche selbst wiederum aus einer Abfolge von Interaktionen mit Seitenelementen und Seitentextüberprüfungen bestehen. Daraus entstand die Idee, dass der Anwender in einem Baukasten wie dem Tree-based Editor das Modell der Systemtestfassade formal beschreibt, indem er dort die verwendeten Seitenelemente und Funktionen definiert.

Ziel ist es hierbei, beim Anwender so wenig Wissen über die Implementierung der getesteten Seiten wie möglich vorauszusetzen und stattdessen alle Testanforderungen über den logischen Aufbau der Htmlseitenelemente zu beschreiben, weil dieser bereits vor der Implementierungsphase feststeht.

Aus Gründen der Übersicht und Wiederverwendbarkeit ist es sinnvoll, den ausgeführten Code nach einzelnen Fassadenmethoden zu gruppieren, die mit einer konkreten Parameterbelegung aufgerufen werden können und schrittweise entweder den Seitenaufbau untersuchen oder mit Seitenelementen interagieren.

4. Systemtestfassaden-Metamodell

Inhalt

Während sich die vorhergehenden Kapitel mit theoretischen Grundlagen sowie Beispielen zur Veranschaulichung von Systemtests beschäftigen, folgt ab diesem Kapitel die Beschreibung praktischen Umsetzung der Codegenerierung. Es wird auf den Aufbau des verwendeten Systemtestfassaden-Metamodells und der einzelnen, hierfür benötigten Komponenten eingegangen.

4.1. Verwendetes Metamodell

Wie bereits in den Grundlagen geschildert, wurde im RSA das verwendete Systemtestfassaden-Metamodell als UML-Diagramm erzeugt. Für dieses Metamodell werden Regeln zur Codegenerierung und zur Fehlererkennung festgelegt. Der Anwender besitzt die Möglichkeit, Modelle auf Basis des Systemtestfassaden-Metamodells mit Werkzeugen wie dem Tree-based Editor zu erstellen. Hierbei basiert ein Objekt des Modells mit seinen Attributen und Relationen auf den Aufbau einer dazugehörigen Klasse des Metamodells.

4.1.1. Anforderungen an das Metamodell

Ziel ist es, alle einzelnen Komponenten, aus denen ein Systemtest besteht, als eigene Klassen des UML-Diagramms zu definieren, damit diese Komponenten durch Objekte konkreter Modellen spezifiziert werden können.

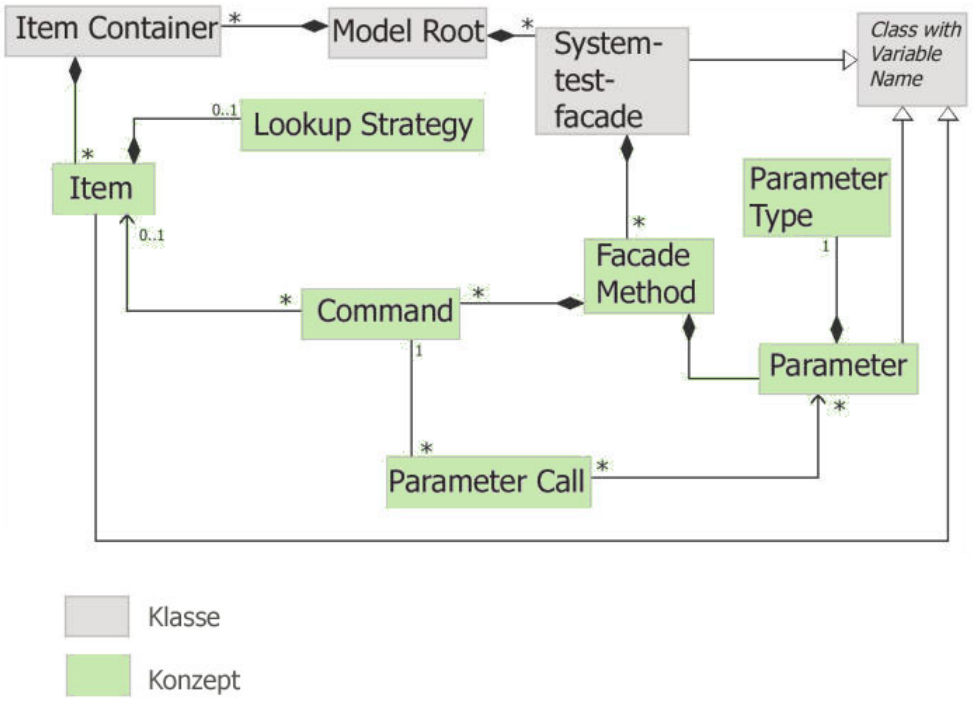


Abbildung 4.1.: Relationen der einzelnen Systemtest-Komponenten

In Abb.4.1 ist das Konzeptmodell des verwendeten Metamodells dargestellt. Bei den abgebildeten einzelnen Konzepten handelt es sich um die Abstrahierung einer Menge von Metamodellelementen.

Es existiert eine Wurzelklasse („Model Root“), in der allgemeine Eigenschaften des getesteten Informationssystems festgehalten werden.
Zu den Basiskomponenten eines Systemtests gehören interaktive Elemente des Informationssystems sowie Methoden von Testfällen. Um sie hierarchisch leichter einteilen zu können, besteht die Wurzelklasse aus Klassen von aus Methoden bestehenden Systemtestfassaden, sowie aus sogenannten „Item Containern“, in denen unter „Items“ die interaktiven Elemente des Informationssystems untergebracht sind.
Um Items eindeutig zu identifizieren, ist pro Item eine entsprechende Strategie nötig. Methoden bestehen aus Befehlen(engl. „Commands“) und aus Eingabeparamtern. Befehle können je nach Typ mit Items interagieren. Außerdem greifen Befehle über spezielle Klassen auf die Eingabeparameter der Methoden zurück. Solche Klassen wurden im Schema als „Parameter Calls“ zusammengefasst. Für Eingabeparameter wurde ihr Typ unter entsprechenden Klassen im Metamodell festgehalten.

Um spätere Überprüfungen des Namensattributs von Systemtestfassaden, Items

und Eingabeparametern zu vereinfachen, wurde eine zusätzliche Klasse „Class with variable Name“ als Oberklasse dieser drei Klassen erstellt. Auf die genaue Funktionsweise der „Class with variable Name“ wird im Kapitel über Check-Überprüfungen genauer eingegangen.

Die einzelnen soeben definierten Komponenten werden im UML-Diagramm der Systemtestfassade als eigene Klassen dargestellt. Um für verschiedene Anforderungen eines Testmodells flexibel unterschiedlich aufgebauten Code generieren zu können, haben viele der Klassen Unterklassen, welche den selben Aufgabenbereich des Modells abdecken, jedoch nach den Eigenarten der Unterklasse aus Modellen eigenen Code generieren. So lassen sich sämtliche interaktiven Elemente wie HtmlButtons, Tabellen oder Eingabefelder als Spezialfälle der abstrakten Itemklasse im Metamodell definieren.

Der finale Aufbau des verwendeten Metamodells befindet sich im Anhang der Bachelorarbeit in der Abbildung A.1. In den Abbildungen A.2 und A.3 sind die einzelnen Klassen des final verwendeten Metamodells nach ihrem Aufgabenbereich jeweils farbig zusammengefasst.

4.1.2. Synchronisation mit dem TestBaseModel

Zeitgleich zu dieser Bachelorarbeit wurde an einer Arbeit geschrieben, welche sich auf Basis des gemeinsam verwendeten EJB-Gens mit konkreten Testfällen beschäftigt [2]. Um die in dieser Arbeit generierten Systemtestfassaden mit konkreten Testfällen zu verwenden, wurden aus dem „TestBaseModel“, dem Metamodell für konkrete Testgenerierung, die Basisklassen importiert. Analoge Systemtestmetamodellklassen wurden daraufhin als Unterklassen der importieren Klassen gesetzt. So ist z.B. die Klasse der Systemtest-Fassade als Unterklasse der importierten Fassade-Klasse definiert. Im angehängten Schema A.2 sind die importierten Klassen mit einer blauen rechteckigen Markierung farbig hervorgehoben.

4.1.3. Entwicklung des Metamodells der Systemtestfassade

Einen vergleichbaren Aufbau zu den angehängten UML-Diagrammn A.1 - A.3 gab es bereits in seinen ersten Versionen des Metamodells. Im Laufe der Entwicklung wurden als wichtigste Änderungen für einzelne Klassen Unterklassen eingeführt, sowie Klassen und Attribute angelegt, um später hinzukommende Sachverhalte des XPAND und Check-Codes besser auszudrücken.

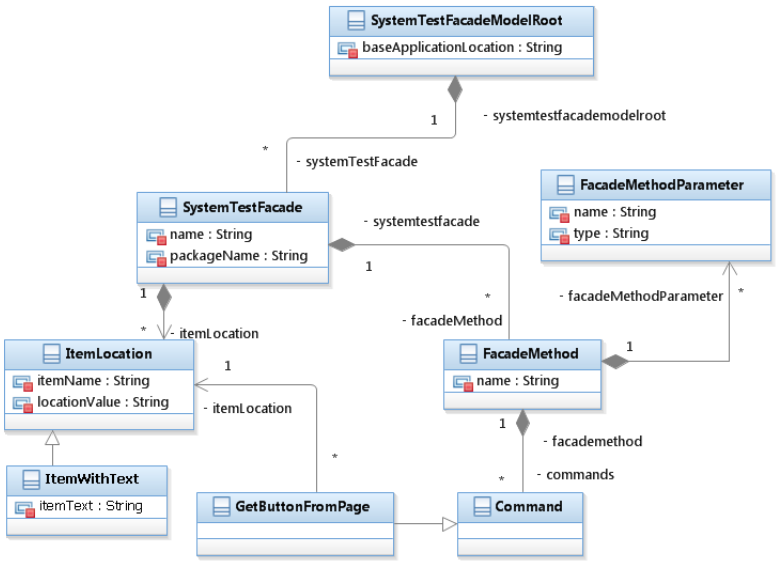


Abbildung 4.2.: Erste lauffähige Version des UML-Metamodells

Auch die erste Version des UML-Diagramms verfügt als Komposition der Systemtestfassade über Fassadenmethoden, die über „Command“-Befehle mit als „Itemlocation“ benannten Seitenelementen interagieren und zusätzlich über Eingabeparameter verfügen. Jedoch gehören die Seitenelemente in der ersten Version noch direkt zu einer Systemtestfassade und nicht zu einem davon unabhängigen „ItemContainer“. Bei der Modellerstellung sollte es vermieden werden, den Anwender zu zwingen, bereits einmal definierte Elemente an anderen Stellen des Modells erneut definieren zu müssen. Dies ist z.B. der Fall, wenn in der ersten Version mehrere verschiedene Systemtestfassaden erstellt werden, die mit den gleichen Seitenelementen interagieren.

In der ersten Version des UML-Diagramms müsste für jede Systemtestfassade die Menge an verwendeten Elementen neu definiert werden. Durch später von den Fassaden unabhängig aufgebauten „ItemContainer“, deren Items von allen Fassadenmethoden benutzt werden können, konnte dieses Problem gelöst werden.

Der Aufbau der zu den einzelnen Komponenten gehörenden Klassen aus Abbildung A.3 wird in den folgenden Abschnitten näher erläutert. Nach der Beschreibung ihrer einzelnen Funktionen und Aufgabenbereiche wird außerdem noch auf den Entwicklungsprozess eingegangen.

4.1.4. Basiselemente/Wurzelelemente

Das UML-Modell besitzt ein Wurzelelement, in dem unter dem Attribut „baseApplicationLocation“ die Basis-URL der anzuvagigierenden Hauptseite gespeichert ist. Später hinzukommende URl-Navigationen nutzen diese BasisURL und erweitern diese.

Das Wurzelelement besitzt seit der ersten Metamodell-Version als Komponenten eine Menge an Systemtestfassaden, welche Testfälle auf die in der Wurzelklasse spezifizierte URL bereitstellen. Später kamen noch ItemContainer als Komponenten hinzu.

Die Fassade ihrerseits besitzt als Attribute ihren Namen, welcher als Klassenname bei der Generierung benutzt wird sowie einen Paketnamen, welcher als Beschreibung des Verzeichnisses dient, in das der Code generiert wird.
Pro Systemtestfassade wird eine entsprechende Javaklasse erzeugt. Das Systemtestfassadenattribut „name“ gibt den Namen der zu generierenden Klasse an, während in „packageName“ das Verzeichnis festgehalten wird. Zur Angleichung an den Aufbau vom „TestBaseModell“ wurden die Klasse der Systemtestfassade als Unterklasse der importierten Fassadenklasse definiert, woraufhin die Attribute der Fassadenklasse übernommen wurden. Die Fassadenklasse besteht aus mehreren Fassadenkonfigurationen, die jeweils Name und Paketname als Attribute führen [2].

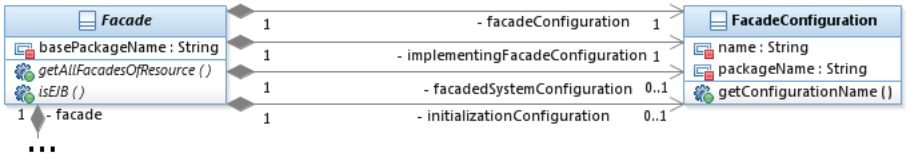


Abbildung 4.3.: Relationen zwischen Fassade und Fassadenkonfiguration im TestBaseModel [2]

Für den Systemtest sind hierbei vor allem die unter „facadeConfiguration“ und „implementingFacadeConfiguration“ benannten Elemente relevant. Während unter „implementingFacadeConfiguration“ Name und Paketname einer Fassade für die fertige Generierung eingetragen werden, regelt die „facadeConfiguration“ die Belegung der beiden Attribute die optische Benennung der Fassade im TreeBased-Editor, was keine Auswirkungen auf die eigentliche Codegenerierung hat.

Neben dem Namen der Basis-URL können in der Wurzelklasse noch andere Eigenschaften über den Aufbau des Informationssystems festgehalten werden. Ein

solches Attribut ist „usesPrimfaces“, in dem festgelegt wird, ob im Informationssystem die Htmlstruktur der einzelnen Seiten mit oder ohne die Verwendung der Komponentebibliothek „Primefaces“ geschieht, welche Auswirkungen auf den strukturellen Aufbau von Seiten eines Informationssystems hat.

4.1.5. Fassadenmethoden und Parameteraufrufe

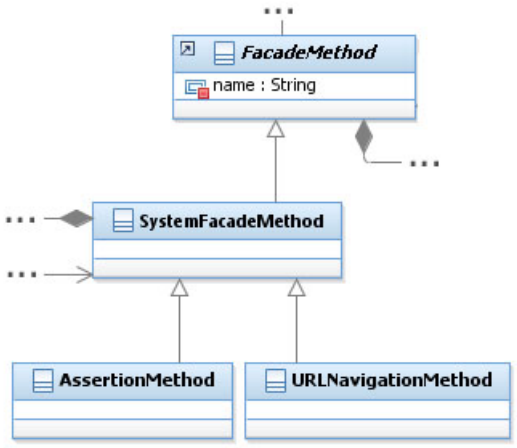


Abbildung 4.4.: Fassadenmethoden, Ausschnitt des verwendeten UML-Diagramms

Aus Elementen der Klasse der Fassadenmethoden werden Java-Methoden generiert, die von konkreten Testfällen ausgeführt werden. Methoden können Eingabeparameter besitzen und führen nacheinander eine vorgegebene Folge an einzelnen Anweisungen(engl. Command) durch, in denen die Eingabeparameter der Methode mitverwendet werden können. In der UML-Modellierung bedeutet dies, dass Fassadenmethoden Parameter und Anweisungen als Komponenten besitzen. Die Reihenfolge, in der die Anweisungen im Modell als Kinderelemente einer Methode modelliert werden, gibt die Reihenfolge ihrer Durchführung an. In einer Methode verwendete Eingabeparameter werden ebenfalls als Kinderelemente ins Modell eingefügt.

Um erstellten Code möglichst kurz zu halten und bereits definierte Testschritte wiederzuverwenden, können Methoden andere Methoden als Anweisung aufrufen. Aus Testfällen am Beispiel des Forumsystems wurde deutlich, dass es im Rahmen des Systemtests zwei Arten an Anweisungen gibt: Interaktionen mit Seitenelementen und Untersuchungen der Htmlseite nach Schlüsselwörtern.

Daraus entstand die Idee, Untersuchungen der Htmlseite nach Schlüsselwörtern als separate Klasse von „Assertionmethoden“ festzuhalten, welche dann von den eigentlichen Fassadenmethoden aufgerufen werden.

Durch die Zweiteilung in Assertionmethoden und Fassadenmethoden entstanden im Metamodell immer wieder parallel ähnlich aufgebaute Konstrukte wie der zuvor beschriebene Aufbau von Methodenaufrufen. Das UML-Diagramm wurde unnötig komplex, weil für jede einzelne Methode einzelne Klassen modelliert werden mussten. So gab es Assertionmethodenaufrufe für Assertionmethoden, Assertionmethodenaufrufe für Fassadenmethoden und Fassadenmethodenaufrufe für Fassadenmethoden. Für all diese drei Anweisungen waren eigene Überprüfungen und Codegenerierungsanweisungen nötig, was neben der Komplexität des UML-Modells auch die der XPAND- und Check-Regeln erhöhte.

Aus diesem Grund wurde die Klasse der Asserionmethoden als zu Fassadenmethoden paralleles Konstrukt vollständig entfernt. Die Untersuchung der Htmlseite nach Schlüsselwörtern wurde als Unterklasse der anderen Befehle einer Fassadenmethode eingefügt. Damit können innerhalb einer Fassadenmethode sowohl Iteminteraktionen als auch Korrektheitsüberprüfungen durchgeführt werden.

Für eine spätere optionale Einteilung wurden als Unterklassen von Fassadenmethoden noch Assertionmethoden sowie URL-Navigationsmethoden eingefügt. Semantische Unterschiede unter diesen Methoden existieren jedoch nicht, die Unterteilung erfolgt vor allem aus Gründen der Übersicht auf Initiative des Anwenders.

4.1.6. Commands

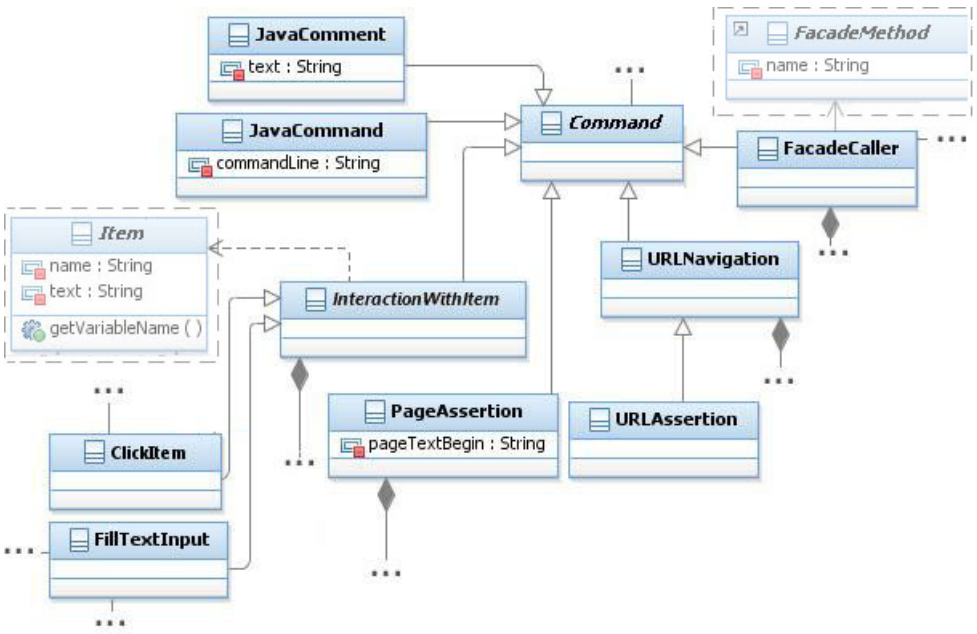


Abbildung 4.5.: Command-Klassen, Ausschnitt des verwendeten UML-Diagramms

Commandklassen sind die einzelnen Befehle von Fassadenmethoden. Sie werden in der Reihenfolge generiert, in der sie innerhalb einer Methode ausgeführt werden. Die abstrakte Command-Oberklasse besitzt dabei verschiedene Unterklassen, welche die einzelnen Arten der möglichen Befehle darstellen.

Zur direkten Interaktion mit Seitenelementen existieren die Klassen „ClickItem“ und „FillTextInput“, die wegen ihres z.T. ähnlichen Aufbau zur Oberklasse „InteractionWithItem“ gehören. Diese Klassen stehen in Relation mit einem zu interagierenden Seitenelement. In den anfänglichen Versionen konnte dabei im Treebased-Editor jeder Itemtyp als zu interagierendes Element ausgewählt werden. Durch Überprüfungen im Check bei der Modellgenerierung wurde überprüft, ob die Art der Interaktion mit der Itemklasse nicht sinnvoll ist, wie z.B. Texteingabe für Elemente ohne entsprechendes Eingabefeld wie Buttons. Durch Einteilung der Seitenelemente in Klassen anklickbarer Items und Items mit Textfeldeingabe wurden solche Überprüfungen jedoch unnötig. Die Interaktionen stehen direkt mit Items der passenden Oberklasse in Relation, wodurch im Modell nur Objekte von Itemklassen ausgewählt werden können, die ein Spezialfall dieser Oberklasse sind.

Um dem Anwender die Möglichkeit zu geben, manuell geschriebenen Code einzufügen, für den noch keine formale Beschreibung vorgesehen ist, gibt es die Klasse der „Javacommands“, in deren „text“-Attribut im Modell eine beliebige Codezeile eingegeben werden kann. Dies ist vor allem dann nötig, wenn Anweisungen nicht durch das Metamodell beschrieben werden können. Durch die stetige Erweiterung an möglichen Befehlen sank die Bedeutung der „Javacommand“-Klasse. Sie wurde aber dennoch beibehalten, um bei unvorhergesehenen Situationen den Anwender eine manuelle Codegenerierung zu ermöglichen. Daneben gibt es eine analoge Möglichkeit Kommentare anhand der Klasse „JavaComment“ zu realisieren.

Die Aufrufe von anderen Methoden werden in der aktuellen Version des Metamodells in der Klasse „FacadeCaller“ definiert, welcher die aufgerufene Fassadenmethode unter der Bezeichnung „callerMethodUsed“ übergeben wird. „FacadeCaller“ verfügten zunächst über eine Menge an Übergabeparametern, die sie in der Reihenfolge ihres Auftretens im Aufruf der verwendeten Methode einsetzten. Dadurch entsteht das Problem, dass ein Parameter höchstens einmal in einem Methodenaufruf belegt werden kann. Der Aufruf einer Methode, bei der mehrere ihrer Eingabeparameter durch ein und denselben Eingabeparameter belegt sind, ist dadurch nicht möglich.

Zur Behebung des Problems wurde zu sogenannten Parametermappings gewechselt. Bei einem Parametermapping besteht der Methodenaufruf nicht direkt aus einer Menge an Parametern, sondern aus einer Menge an „Mappings“, welche genau einen Parameter verwenden, wodurch ein und derselbe Parametern durch

mehrere Parametermappings benutzt werden kann.

Damit leichter für jeden einzelnen „FacadeCaller“ geprüft werden kann, ob die übergebenen Parameter des Mappings den tatsächlichen Parametern der Funktion entsprechen, ist die Kompositionsbeziehung zwischen Commands und der Fassadenmethode bidirektional. So können Commands ihre als „basemethod“ gekennzeichnete Methode sehen und ihre Parameter mit der Parameterliste aus der Methode vergleichen.

Die Klasse der „URL-Navigation“ bezieht sich auf die Möglichkeit, in Testfällen nicht nur über Klicks der Seitenelemente zu navigieren, sondern auch durch die direkte Eingabe einer URL. Bevor ein solcher Befehl im Metamodell existierte, wurde im generierten Code zu Beginn jeder Fassadenmethode die aktuell benutzte Htmlseite automatisch mit der Hauptseite initialisiert. Dies erwies sich spätestens jedoch dann als sehr sperrig, als von Fassadenmethoden aufgerufene andere Methoden bereits einzelne kleine Schritte durchführen sollten, die nicht zwangsläufig an der URL-Hauptseite begangen.

Durch das Verwenden einer URL-Navigation wird die in der Wurzelklasse definierte Hauptseite als aktuell verwendete Seite geladen. Für die Navigation an eine bestimmte Unterseite kann die URL-Navigation als Komponente beliebig viele URL-Elemente besitzen, die mit einen Fassadenmethodenparameter in Relation stehen und zusätzlich einen spezifischen Text als Attribut besitzen. Damit lässt sich eine URL definieren, die aus der Basis-URL und einer beliebigen Menge an Erweiterungen in Form eines Parameters und eines darauf folgenden Textes besteht. Neben der URL-Navigation gibt es noch die Klasse der „URL-Assertions“, die wegen gleichen Aufbaus der verwendeten URL als Unterklasse der URLNavigationen definiert sind. In der Codegenerierung dienen „URL-Assertions“ für Tests, ob die URL der aktuell geladenen Htmlseite mit der erwarteten, zusammengesetzten URL übereinstimmt.

Überprüfungen des Seitentexs nach bestimmten Schlüsselwörtern lassen sich mittels der „PageAssertion“-Klasse realisieren. Die daraus generierte Assertionmethode besteht aus einer Aussage und einen zu werfenden Fehlertext. Wenn die Aussage sich in einem konkreten Testfall als unwahr erweisen sollte, wird der Fehlertext ausgegeben und gleichzeitig der Testfall als fehlgeschlagen abgebrochen.

4.1.7. Parameter und Parametertypen

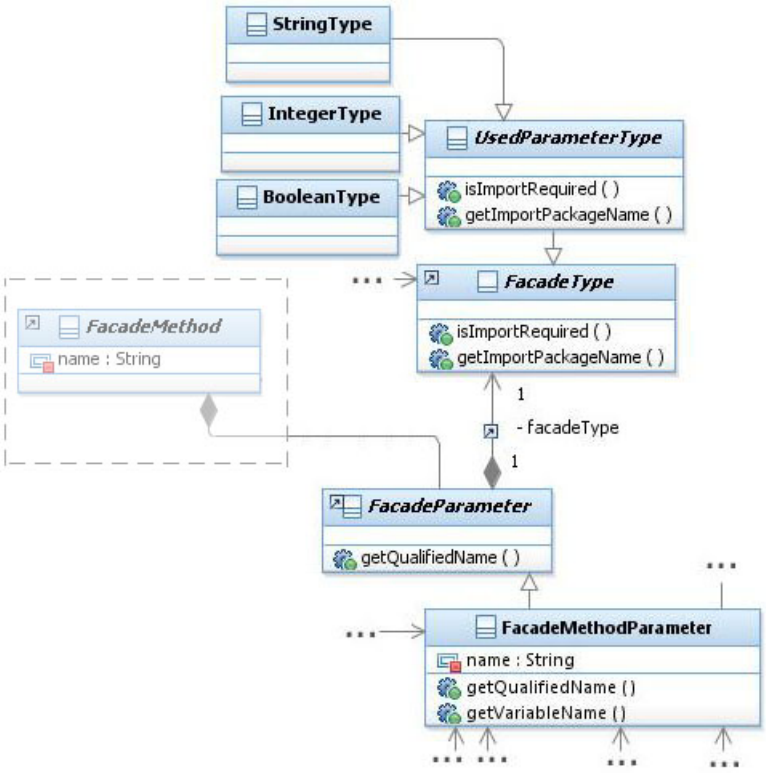


Abbildung 4.6.: Parameter und Parametertypen, Ausschnitt des verwendeten UML-Diagramms

Die Klasse der Fassadenparameter dient als Platzhalter für Eingabeparameter der Fassadenmethode. Als die Methoden nach Fassaden und „Assertionmethoden“ unterteilt waren, besaß jede einzelne Methode ihre eigene Parameterklasse. Als Attribute besitzt ein Parameter einen Namen sowie einen Typ.

Der Typ des Parameters war zunächst als Attribut definiert und konnte im Modell als String eingetragen werden. Aus Gründen der Übersicht und zur Anpassung an den Aufbau vom „TestBaseModel“ wurde der Typ des Parameters in die separate abstrakte Klasse „used Parameter Type“ ausgelagert, welche als Unterklassen die gängigen Parametertypen String, Integer und Boolean besitzt.

4.1.8. Item

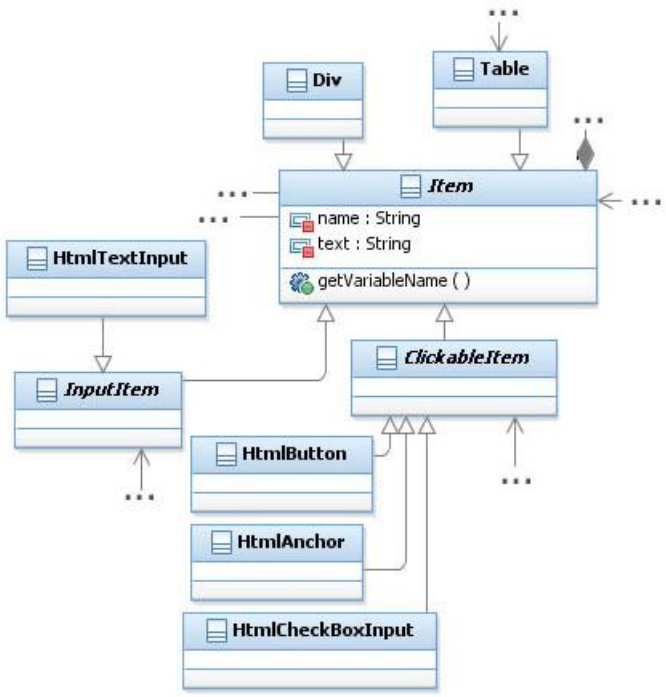


Abbildung 4.7.: Verwendete Itemklassen, Ausschnitt des verwendeten UML-Diagramms

Im Metamodell definierte Klassen von Items sind Platzhalter für die verschiedenen interaktiven Elemente von Htmlseiten. Als Attribute besitzen sie einen Text, sowie einen Namen.

Zunächst waren Items der Systemtestfassade selbst zugeordnet und wurden nicht nach Ihrem Elementtyp in Html, sondern lediglich nach ihrem Verwendungszweck grob eingeteilt. Danach wurden Items als Komponenten für eigene Itemcontainer zugeordnet, um von unterschiedlichen Systemtestfassaden verwendet werden zu können.

Als weitere Änderung wurden Items passend zu den zwei Interaktionsmöglichkeiten in klickbare Items und Items mit Texteingabe eingeteilt. Je nach Interaktionsmöglichkeit wurden darauf alle Typen von Items als Unterklassen von den beiden Spezialfällen von Items oder von der Itemklasse selbst definiert. Unterklassen von klickbaren Items sind hierbei HtmlButton, HtmlAnchor und HtmlCheckboxInput. Unterklasse für EingabeItems ist die Klasse der HtmltextInputs.

Daneben gibt es Items, die nicht zur direkten Interaktion vorgesehen sind, sondern zur Beschreibung der Struktur einer Seite. Dazu zählen als „Div“ gekennzeichtene HtmlDivisionen, sowie Tabellen in denen jeweils andere Items enthalten sein können.

4.1.9. Item-Lookupstrategie

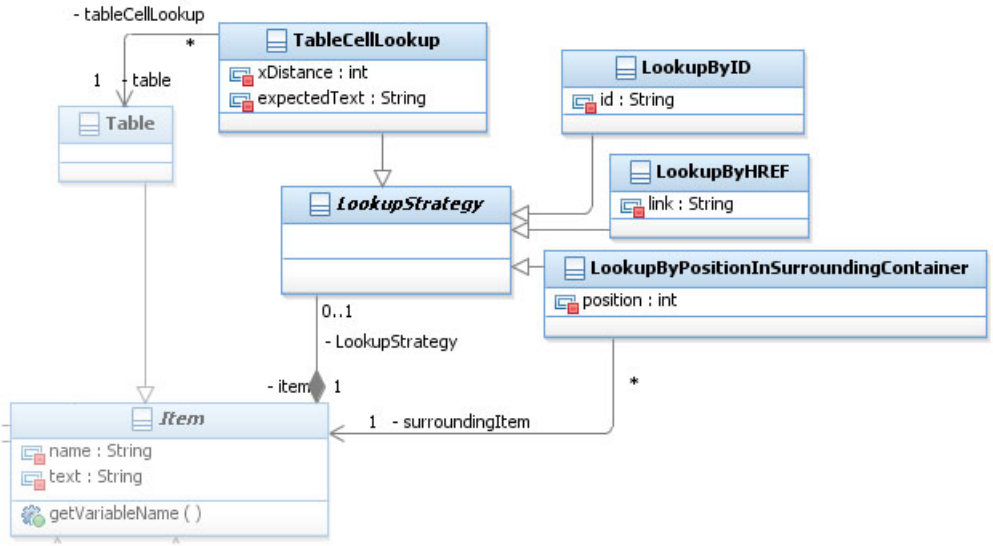


Abbildung 4.8.: Lookupstrategien und dazugehörige Items, Ausschnitt des verwendeten UML-Diagramms

Mit der Lookupstrategie (engl. lookup=nachschlagen) wird für ein Item definiert, wie es auf der Htmlseite gefunden werden soll.

Zunächst war für jedes Item allein aufgrund seines Metatyps eine Strategie definiert worden, die den Typ seiner Identifikationsmöglichkeit bestimmte. Diese Identifikationsmöglichkeit wurde dabei immer im String-Format als „Locationvalue"-Itemattribut gespeichert. Jedoch können Items auf verschiedene Art und Weise identifiziert werden. So kann z.B. ein HtmlAnchor entweder über seine ID oder über seinen Link gefunden werden. Um dies zu realisieren, wurde das Locationvalue-Attribut entfernt und stattdessen eine abstrakte Klasse der Lookupstrategie als Komponente der Itemklasse erstellt, von der jedes Item maximal eine besitzen kann. Im Falle des Item-Aufrufs würde der Code dann entsprechend der Art dieser Strategie aufgebaut sein.

Einzelne Unterklassen der Lookupstrategien unterscheiden sich durch die Art, wie das Item gefunden werden kann. Darunter gibt es die bereits erwähnte Suche anhand einer ID bzw. anhand eines Links. Später kam das Ziel hinzu, Items zu finden, welche in einer Tabelle während eines Tests erstellt wurden und deswegen deren ID zu Testbeginn nicht bekannt ist.

Als Lösung wurde die Strategie „TableCellLoolup" definiert. Mit dieser Strategie wird eine Tabelle zellenweise nach einem bestimmten Schlüsselwort durchsucht, welches mit dem gesuchten Element im Zusammenhang steht. Wurde eine Tabellenzelle gefunden, die dieses Schlüsselwort beinhaltet, wird angenommen, dass sich das gesuchte Item in einer vorher festgelegten Anzahl an Spalten entfernt

von dieser Zeile befindet.

Beim Arbeiten mit fertigen Informationssystemen fiel auf, dass es bei Htmlelementen wie Tabellen vorkommen kann, dass sie keine ID besitzen. Dennoch musste eine Möglichkeit gefunden werden, mit der ein in die Tabelle generiertes Objekt gefunden werden konnte. Die erste Idee, dieses Problem zu lösen, war eine automatische Berechnung, wie die ID eines dynamisch erstellten Items lauten müsste. Jedoch wurde dieser Ansatz sofort wieder verworfen, weil er Wissen über die Implementierung voraussetzt und damit der Idee dieser Bachelorarbeit widerspricht, ohne spezifische Kenntnisse des Programmaufbaus Systemtests erstellen zu können.

Um stattdessen Elemente ohne ID zu finden, wurde die Lookupstrategie-Unterklasse „LookupbyPostionInSurroundingContainer“ erstellt. In dieser Strategie wird in einem „umhüllenden“ Item (meistens vom Typ Div) ein enthaltenes Item an einer vorgegebenen Position ausgewählt. So konnten Tabellen eindeutig anhand ihrer Position im hierarchischen Aufbau der Seite bestimmt werden. Ist die Tabelle gefunden, wird mittels „TableCellLoolup“ aus ihr das gesuchte Item ausgewählt.

5. Codegenerierung aus Modellen

Inhalt

Dieses Kapitel beschäftigt sich damit, wie Modelle auf Basis des Metamodells erstellt werden und wie genau die einzelnen Programmwerkzeuge arbeiten, um aus solchen Modellen Programmcode zu generieren.

5.1. Modellerstellung im Tree-based Editor

Im Tree-based Editor werden konkrete Modelle vom Typ der Systemtestfassade definiert, aus denen mittels MTF Programmcode nach in XPAND definierten Regeln generiert wird. Das Modell besteht aus einzelnen Objekten, welche als Kinder- oder als Geschwisterelemente an bereits definierte Objekte angeknüpft werden. Sowohl die Objekte als auch ihre Relationen basieren auf dem Aufbau der einzelnen Klassen des Metamodells. Jedes Objekt basiert auf einer Klasse des Metamodells, wobei die im Metamodell definierten Attribute und Relationen durch konkrete Werte und Elemente belegt werden.

Für Relationen eines Objekts zu anderen Elementen wird aus einer Liste von Objekten der dafür passenden Klassen das konkrete Zielelement ausgewählt. So wird z.B. für Befehle, die mit einem Objekt der Parameter-Klasse interagieren, aus der Liste aller vorhandenen Parameterelemente einer dieser Parameter als Belegung des dafür vorgesehene „inputParameter“-Attribut ausgewählt. Übergebene Parameter müssen im Allgemeinfall zur selben Methode wie der Befehl gehören. In der Liste stehen allerdings alle Parameter des Modells zur Auswahl, sodass der Anwender auch ungültige Parameter auswählen kann. Ein solcher Fehler wird mit entsprechenden Check-Regeln abgefangen.

Sollten jedoch zwei Parameter gleich benannt sein, können sie in der Parameterliste optisch vom Anwender nicht auseinandergehalten werden. Objekte des Tree-based-Editors werden über ihr als „Label Feature“ gekennzeichnetes Attribut anzeigt, das Standardmäßig der Name des Objekts ist. Als Lösung hierfür

wurden anstelle des Namens als „Label Feature“ bei Parametern das automatisch errechnete Attribut „Qualifiedname“ benutzt, welches neben dem Namen den gesamten hierarchischen Pfad eines Objekts anzeigt. Durch diesen eindeutigen Pfad kann eine optische Unterscheidung der einzelnen Objekte sichergestellt werden.

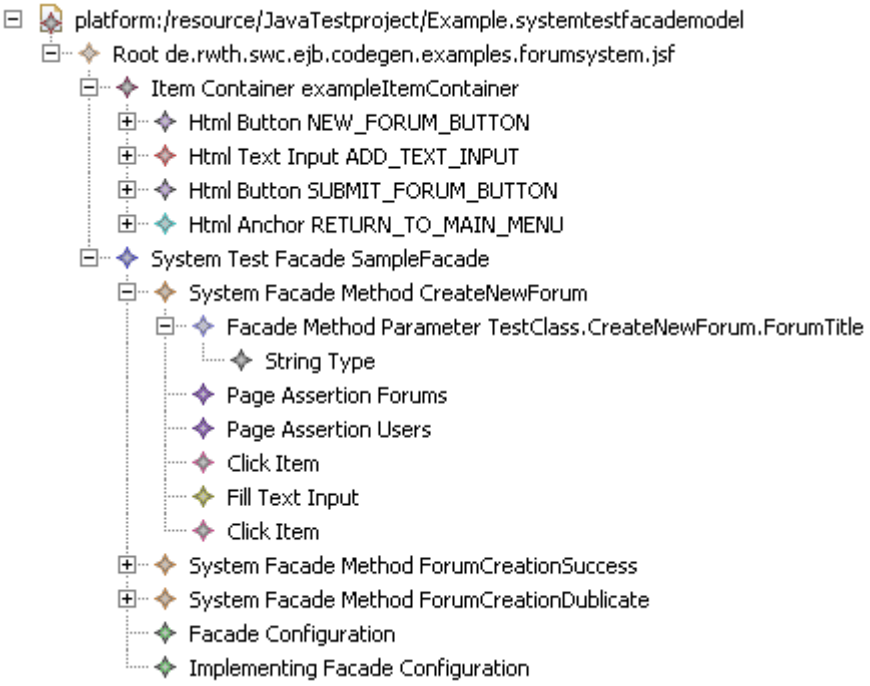


Abbildung 5.1.: Nachbildung des Forumsystembeispiels durch den Tree-based Editor

5.2. Systemfassadenmodellierung in XPAND

In XPAND werden Anweisungen für Elemente des Metamodells geschrieben, nach denen für analoge, konkrete Objekte eines Modells Programmcode generiert wird. Das Modell des Tree-based Editors besteht, wie bereits in vorhergehenden Abschnitten erwähnt, aus einem WurzelElement. Dieses besitzt als Komponenten ItemContainer-Elemente mit Items als Kinderelementen und Fassaden mit einzelnen Methoden als Kinderelementen.

XPAND-Anweisungen haben die grobe Form:

«DEFINE method FOR SampleClass»
sampleTest «SampleClass.name»

«ENDDEFINE»

In diesem Beispiel wurde die als „method“ benannte Anweisung für die Klasse „SampleClass“ definiert, bei der „sampleText “ sowie der Wert des in „SampleClass“ definierten Attributs „name“ als Fließtext bei einem Aufruf dieser Anweisung generiert werden. Die Text- bzw. Codegenerierung von XPAND-Anweisungen kann durch Generierungsregeln anderer XPAND-Anweisungen erweitert werden. Deshalb wird für die Nachbildung eines Modells zunächst die Anweisung für eine Wurzelklasse definiert und durch Anweisungen zur Codegenerierung für die Kinderelemente der Wurzelklasse erweitert. Durch einen entsprechenden XPAND-Befehl wird für jedes als Wurzelelement verwendete Systemtestfassadenelement des Modells eine neue Datei vom Typ einer Javaklasse angelegt. Name und Verzeichnis der Datei entsprechen der Attributbelegung dieses Objekts.

Der XPAND-Code erzeugt für dieses Wurzelelement den Rumpf einer Javaklasse. Ein solcher Klassenrumpf besteht aus Importen von verwendeten Elementen, aus der durch geschweifte Klammern eingegrenzten Klassendefinition sowie aus einer Menge sich darin befindender Konstanten, Methoden sowie lokaler Variablen. Als lokale Variablen exisitieren für den Systemtest die aktuell geladene Htmlseite unter der Variable „page“ sowie ein „webclient“ zur Seitennavigation. Importiert werden in erster Linie die einzelnen Htmlelemente mit denen bei konkreten Testfällen interagiert wird.
Um flexibel in XPAND die Codegenerierung für eine Beispielklasse um Generierungsanweisungen für ihre einzelnen Komponenten zu erweitern, wird mit Befehlen der Form «EXPAND sampleMethod FOR this.sampleElement» für jedes Kindelement der Beispielklasse vom Typ „sampleElement“ die dafür definierte XPAND-Anweisung „sampleMethod“ aufgerufen.

Sollten für verschiedene Klassen als „sampleMethod“ benannte Anweisungen in XPAND existieren, so wird diejenige Anweisung ausgeführt, die in der Modellstruktur am nächsten zum auszuführenden Element passt.

Der Rumpf der Anweisung zur Generierung der Systemtestfassade wird so um XPAND-Anweisungen für Konstantendefinitionen und Methodenerstellungen erweitert. Codegenerierungsanweisungen für Methoden bestehen aus der durch den Methodennamen und ihre verwendeten Parameter definierten Rumpf, in dem für jeden Befehl der Methode entsprechender Code generiert wird. Damit der Code für die richtige Methode modelliert wird werden gleichnamige Anweisungen für alle einzelnen Methoden und die abstrakte Methodenklasse erstellt. Bei einem Aufruf wird die Anweisung umgesetzt, die den Typ der nachzubildenden Methode verwendet.

Für die Umsetzung einzelner Commandklassen werden neben der eigentlichen Anweisung zusätzlich automatische Überprüfungen mitmodelliert, welche die

korrekte Anweisungsausführung sicherstellen sollen. Zu solchen zusätzlichen Überrüfungen gehört z.B. bei Iteminteraktionen die Sicherstellung, dass ein Item gefunden werden konnte und dass das Item korrekt beschriftet ist. Weil diese Schritte immer nach bestimmten Anweisungen folgen und immer nach einem Schema aufgebaut sind, werden sie im Modell nicht einzeln definiert. Bei speziellen Befehlen wie bei der URL-Assertion ist noch zu beachten, dass die Namen von verwendeten Parametern durch entsprechende XPAND-Anweisungen an die URL-Formatierungen angepasst werden müssen. So wird z.B. ein Leerzeichen zu einem „%20“ umgewandelt.

Viele der Befehle, welche mit Items interagieren, arbeiten intern mit automatisch generierten lokalen Variablen, unter denen das verwendete Item zwischengespeichert wird. Der generierte Name solcher Variablen lautet dabei immer „used_Item“+die Position des Commandos in der Methode.

Es hat sich gezeigt, dass manche oft verwendeten Schritte verschiedener Anweisungen platzsparend in eigenen Methoden ausgelagert werden können, um durch Methodenaufrufe innerhalb dieser Anweisungen verwendet zu werden.

Zu solchen Anweisungen gehört das Auslesen der akuell geladenen URL und das Finden von Seitenelementen anhand vorgegebener Eingabedaten. Diese Methoden werden automatisch von XPAND für eine Fassade mitkonstruiert und dann von anderen Methoden mit entsprechenden Eingabewerten aufgerufen.

5.3. Modellqualitätssicherung durch Check

Wie bereits in den Grundlagen beschrieben, handelt es sich bei der Programmiersprache Check um ein Werkzeug, mit dem der Aufbau von Modellen überprüft werden kann. Ziel ist es, mit dem Aufbau von Regeln so weit wie möglich Situationen zu erkennen, bei denen aus einem Modell fehlerhafter Programmcode generiert wird.

5.3.1. Attributprüfungen

Einer der häufigsten Auslöser für fehlerhaften Code sind undefinierte Attribute und Relationen eines Objekts. Um solche Fehler zu erkennen, wird in Check bei vielen Klassen für jede zu ihr in Relation stehende Klasse geprüft, ob diese definiert ist und damit nicht „null“ beträgt. Bei Klassenattributen wird zusätzlich noch überprüft, ob das Attribut nicht mit einem leeren String belegt ist. Neben einer Belegung durch einen leeren String gibt es viele andere Situationen, in denen der Wert eines Attributs zu ungültigen Programmcode führen kann. Dies passiert z.B. in Fällen, bei denen der verwendete Name syntaktisch falsch ist (z.B. enthalten von Sonderzeichen im Attribut, die nicht vom Java-Compiler

erkannt werden). Dies wird durch Regeln überprüft, die den Namen nach einer bestimmten (verbotene) Zeichenfolge untersuchen. Ein weiterer Grund für potentiell ungültigen Programmcode ist die Benutzung von bereits verwendeten Namen für andere, im Modell definierte Elemente. Auf dieses Problem wird im folgenden Abschnitt näher eingegangen.

5.3.2. Namensprüfung

Um Überprüfungen nach dem Namen einer Klasse zu vereinfachen, wurde gegen Ende des Entwicklungsprozesses für Items, die Systemtestfassade und Fassadenmethodenparameter die Methode „getVariableName()“ sowie die abstrakte Oberklasse „Class With Variable Name“ erstellt, welche ebenfalls die Methode „getVariableName()“ besitzt. Die Ausgabe der Methode ist der Klassenname. Regeln, welche die „Class With Variable Name“ bezüglich ihres Namensaufbaus untersuchen, werden dabei auf alle ihre Unterklassen mitangewandt, wodurch sie nur für die Oberklasse beschrieben werden müssen.

Um Konflikte bezüglich gleichem Namen unter Elementen zu erkennen, werden alle Elemente paarweise auf Namensgleichheit überprüft. Dabei gilt grundsätzlich, dass nicht nur für Elemente einer Klasse Namensgleichheit verboten ist, sondern auch fehlerhafter Code entsteht, wenn ein Parameter denselben Namen wie ein Item oder eine Fassade hat.

Neben Kollision von Namen anderer Modellelemente wird geprüft, ob der untersuchte Name einen reservierten Wert für in XPAND generierte Hilfsvariablen annehmen könnte. Zu den reservierten Werten zählen die Namen lokalen Variablen für die Htmlseite „page“ und den Webclient-Wrapper „webclient“. Reserviert ist ebenfalls die Zeichenfolge „used_Item“, mit denen generierte Variablen für Iteminteraktionen beginnen. Die Prüfung nach solchen Werten findet innerhalb der Oberklasse „Class With Variable Name“ statt.

Bei Methoden ist es im Javacode unproblematisch, wenn ihr Name mit dem einer Variable oder Konstante identisch ist, da Methoden bereits durch ihren semantischen Aufbau eindeutig von anderen Konstrukten aus Sicht des Javacompilers unterschieden werden können. Für Methoden wurden deshalb nur Überprüfungen definiert, ob sich in der Liste von Methoden zwei Methoden im Namen gleichen und ob eine der Methoden reservierte Methodennamen nutzt.

5.3.3. Parameterprüfungen

Für alle Befehle der Fassadenmethoden, welche auf Parameter zugreifen, sind Überprüfungen nötig, ob die ausgewählten Parameter aus derselben Methode wie der Befehl selbst stammen.

Zur Überprüfung betrachtet die entsprechende Regel für einen Befehl die Parameterliste seiner Methode und untersucht, ob diese Liste den verwendeten Parameter beinhaltet.

Besondere Beachtung findet bei der Parameterüberprüfung der „FacadeCaller"-Befehl zum Aufruf anderer Fassadenmethoden, der eine Liste von Parametern übergeben bekommt, die als Belegungen für die Parameter der aufgerufenen Zielmethode eingesetzt werden.

Es wird geprüft, ob die Anzahl übergebener Parameter der Parameteranzahl der aufgerufenen Methode entspricht.
Zum Zeitpunkt, wo Typen eines Parameters als eigenständige Klassen fest definiert wurden, entstand zusätzlich die Idee einer Überprüfung, ob der Typ der übergebenen Parameter in den Typ des eingesetzten Methodenparameters umgewandelt werden kann.

Es wird somit geprüft, ob der Parameter der aufgerufenen Methode und der an dessen Stelle eingesetzte Parameter vom gleichen Typ sind oder ob es sich beim Typ des Parameters der aufgerufenen Methode um einen „String" handelt. In letzterem Fall wird der übergebene Parameter durch einen Typecast selbst in einen String umgewandelt, was in Java bei allen Parametertypen durchführbar ist.

5.4. XTEND Methoden

In XTEND erstellte Methoden finden für Überprüfungen Verwendung, bei denen durch rekursives Überprüfen der Beziehungen eines Modellobjekts bestimmte Eigenschaften festgestellt werden müssen. Eine solche Eigenschaft ist Zykelfreiheit. Weil Methoden sich gegenseitig aufrufen können, würde bei einem Zykel sich aufrufender Methoden der entsprechend generierte Code eine Endlosschleife beinhalten.

Eine andere XTEND-Methode untersucht den Typ der als erstes in einer Methode aufgerufenen Anweisung. Um Interaktionen auf einer möglicherweise falsch definierten Seite vorzubeugen, wird hierbei erwartet, dass die erste angewandte Anweisung einer Fassadenmethode entweder die Url-Navigation auf einer Seite darstellt oder eine Untersuchung beinhaltet, ob die aktuell geladene Seite bestimmte Bedingungen erfüllt.

Beide hier vorgestellten XTEND-Methoden werden von zugehörigen Check-Regeln zur Fehlererkennung angewandt.

6. Evaluation

Inhalt

Nach Fertigstellung des für die Codegenerierung relevanten Programms wurde geprüft, inwieweit die für die Bachelorarbeit angesetzten Ziele erfüllt werden konnten.

6.1. Evaluation des Konzeptes

Die Idee, durch Beschreibung einer Anwendung und der damit möglichen Methoden Code zu generieren, blieb im Laufe der Bachelorarbeit unverändert. Überlegungen nach dem Konzept der Bachelorarbeit befassten sich anfangs mit dem Ansatz, dass anstelle einer aufrufbaren Testfassade direkt sämtliche möglichen Testfälle für den Aufbau einer Seite automatisch generiert werden. Hierfür wäre für die einzelnen Elemente des Modells noch eine genaue Beschreibung nötig, welche mit diesen Elementen ausführbaren Interaktionen aufeinander folgen könnten und in welche Äquivalenzklassen sich die einzelnen Eingabewerte für Attribute einteilen ließen. Eine automatische Erzeugung aller Testfälle aus der Beschreibung der Struktur eines Informationssystems würde jedoch auf die Erstellung von Pfadüberdeckungstests hinauslaufen. Von Pfadüberdeckungstests ist allerdings bekannt, dass die Menge möglicher Pfade sehr schnell unübersichtlich groß wird und in vielen Situationen für ein Informationssystem unbegrenzt viele Pfade entstehen können [10].

Die Idee, Tests anhand von Funktionalitäten automatisch zu konstruieren, wurde damit verworfen und die Codegenerierung wurde auf die Erzeugung von einer Klasse von extern aufrufbaren Fassadenmethoden eingeschränkt.

So kann der Anwender für konkrete Testfälle eines Systemtests ein Set an Methoden definieren, die mit jeweils für den Test vorgesehenen Eingabewerten nacheinander aufgerufen werden können. Die Erstellung solcher konkreter Tests erfolgt dabei außerhalb des Rahmens dieser Bachelorarbeit manuell oder automatisch durch andere Codegenerierungsmechanismen des EJB-Generators.

Eine andere verworfene Konzeptidee befasst sich mit der Suche nach dynamisch

erstellten Tabellenelementen. Die Suche nach konkreten IDs war bereits fertig implementiert und Ziel war es, in Tabellen Items finden zu können, deren ID für den Testfall unbekannt ist. Das erste Lösungskonzept dafür bestand aus der Definition eines programminternen Algorithmus, mit welchem die ID für Elemente errechnet wird. So wäre es möglich, anhand der Position des neu erstellten Elements dessen ID zu errechnen und es danach aufzurufen. Dieses Konzept wurde verworfen, weil es dem Ziel der Bachelorarbeit widerspricht, für Systemtests als Blackbox-Tests ein Minimum an Wissen über die programminterne Implementierung vorauszusetzen. Hier müsste der Anwender etwa wissen, mit welchen Algorithmen IDs für Elemente des Informationssystems erzeugt werden.Als alternative, umgesetzte Lösung wurden entsprechende Suchstrategien für Items nach einer Position bzw. Schlüsselwörtern in anderen Items (wie Tabellen) festgelegt.

6.2. Evaluation des Programms

Der zu generierende Code der Bachelorarbeit hat die Aufgabe, die Funktionalität von webbasierten Informationssystemen so weit wie möglich abbilden zu können und dabei gleichzeitig beim Anwender ein Minimum an Implementierungswissen über das System vorauszusetzen. Diese Idee ließ sich bis auf bestimmte Ausnahmen erfolgreich implementieren. In folgenden Abschnitten wird näher beleuchtet, an welchen Stellen der Codegenerierung eine solche Umsetzung nicht gelungen ist.

6.2.1. Elementverschachtelung

Anhand generierter Systemtestfassaden für Informationssysteme wurde deutlich, dass je nach Testfall unterschiedlich viel Wissen über den Aufbau der untersuchten Seite nötig ist.

Je nach Situation ist bestimmtes Vorwissen über die eigentliche Implementierung unvermeidlich. Als Beispiel hierfür dient die Strategie des Findens von Htmlelementen anhand ihrer Position. Eine solche Methode überprüft die aufgerufene Htmlseite nach ihrem hierarchischen Aufbau und sucht dabei das benötigte Element in einer bestimmten Verschachtelungstiefe des anderen Elements. Je nach Aufbau der Htmlseite werden Unterelemente entweder direkt im Rumpf eines Elements definiert, oder auf eine bzw. mehreren HtmlDivisions aufgeteilt. In der vorgelegten Version des Programms muss jede solche Verschachtelung manuell durch Definition entsprechender HtmlDivisions geschehen, in denen sich das gesuchte Element befinden soll.

Die Art der Verschachtelung von Items ist innerhalb eines einzelnen Informationssystems konsistent und basiert darauf, auf welche Art und Weise die Htmlsei-

te aufgebaut ist. Ein Beispiel dafür ist die Verwendung der Komponentenbibliothek Primefaces, welche automatisch HtmlDivisions für alle Html-CodeElement-Rümpfe erstellt, in denen die einzelnen Elemente gelagert sind.

Es wäre möglich, anstelle manuell erstellter Elementverschachtelungen den Aufbau des Systems in der Wurzelklasse des Modells zu definieren, woraus in XPAND bei der Positionssuche von Elementen deren Verschachtelungen automatisch konstruiert werden könnten. Provisorisch wurde dies durch das nicht implementiere „usesPrimfaces"-Attribut in der Wurzelklasse angedeutet.

In jedem Fall bleibt für die Suche nach Elementen bestimmtes Vorwissen über die genaue Implementierung notwendig. Für die Elementsuche nach einer ID wird erwartet, dass die vergebene ID des Elements bekannt ist, für eine Suche nach der Position des Elements muss der Anwender dagegen wissen, nach welcher Struktur die Htmlseite aufgebaut ist.

6.2.2. Attributbenennung

Neben der Elementsuche wurde auch ein anderes Limit der Codegenerierung deutlich: Die Überprüfung des Codes nach ungültigen Namen von Attributen. Durch eine ungültige Bezeichnung eines Attributs, aus dem die Belegung einer Variable entsteht, wird der generierte Code fehlerhaft, entweder weil die Art der Bezeichnung mit den Erwartungen an die Grammatik von Java kollidiert (z.B. Sonderzeichen) oder auch wenn z.B. durch Verwendung von Leerzeichen nur ein Teil des Variablennamens tatsächlich als Variable interpretiert wird. Eine Lösung des Problems besteht aus einzelnen Checks auf das Attribut, welche es nach solchen Fehlerquellen überprüfen.Dabei wurde deutlich, dass die Abdeckung aller Fehlerfälle zu einer sehr großen Menge einzeln zu defnierender Regeln führen würde.Auch wenn durch Zusammentragen der geprüften Klassen unter „Class with variable Name" nur diese Klasse auf Unstimmigkeiten geprüft wird und nicht jede ihrer einzelnen Unterklassen, entsteht dennoch eine große Menge an Überprüfungsregeln, weil jede mögliche Fehlerquelle in einer einzelnen Regel festgehalten werden müsste, um komplette Sicherheit zu gewährleisten.

Es folgte der Entschluss, nur nach den häufigsten und wichtigsten Ursachen für Fehler wie Namensdublikaten oder Leerzeichen im Namen zu suchen. Die restliche Fehlervermeidung wurde an die Fähigkeiten des bedienenden Anwenders anvertraut. Weil vorausgesetzt werden kann, dass die Anwender der Systemtest-Software sich mit Softwareentwicklung und Programmierung auskennen, ist auch davon auszugehen, dass sie offensichtliche Benennungsfehler wie z.B. nicht unbeabsichtigt verursachen würden bzw. von selbst solche Fehler erkennen könnten. Dadurch sank folglich die Priorität, die Abdeckung aller nur irgendwie möglicher Namensfehler weiter umzusetzen.

Selbiges gilt in noch extremerer Form für Freitexteingabefelder, bei denen dem

Anwender die Möglichkeit geboten wird, beliebigen Text einzutragen. Dort wird komplett auf eine Überprüfung des eingegebenen Texts verzichtet.

Bei der Umsetzung von Methoden zur Untersuchung der URLs der aktuell geladenen Seite entsteht noch ein weiteres Problem. In der URL werden vom ursprünglich übergebenen String bestimmte Zeichen durch url-taugliche Äquivalente ersetzt. So wird etwa ein „ „-Leerzeichen in ein „%20“ umgewandelt. In der aktuellen Version sind solche Umwandlungen und darauf basierende Überprüfungen nur unzureichend implementiert.

6.2.3. Seitenelemente

Für die meisten oft verwendeten Htmlelemente wurden entsprechende Metamodellklassen definiert und in XPAND Regeln aufgestellt, die das Verhalten dieser Elemente erfolgreich nachbilden. Es fehlen weiterhin Definitionen für bestimmte Htmlelemente wie z.B. RadioButtons. Auch beschränkt sich die Untersuchung der Seite nur ihren Seitentext, während Überprüfungen von anderen Seitenelementen wie Bildern nicht vorgesehen sind.

7. Zusammenfassung und Ausblick

Inhalt

Dieser Abschnitt fasst die Ergebnisse der Arbeit zusammen und gibt daraufhin einen Ausblick, wie sich die weitere Entwicklung einzelner Bereiche gestalten ließe.

7.1. Zusammenfassung

In der Bachelorarbeit wurde ein Werkzeug erstellt, mit dem sich Java-Code für Systemtestfassaden zur Überprüfung der Funktionalität von webbasierten Informationssystemen generieren lässt.

Dafür wird der Aufbau eines webbasierten Informationssystems als Modell nachgebildet. Zu dieser Nachbildung zählt die Definition der einzelnen interaktiven Seitenelemente und von Methoden, welche den Aufbau der Seite nach Schlüsselwörtern untersuchen und mit den definierten Elementen interagieren.

Durch Konstrukte wie rekursive Methodenaufrufe sowie Übergabeparameter besteht die Möglichkeit, diese Methoden möglichst flexibel und kurz zu halten. Bevor der Code generiert wird, finden Überprüfungen statt, ob das aufgebaute Modell fehlerhaften Programmcode wegen ungültiger Wertebelegung produzieren könnte. Der generierte Code wird schließlich in Form einer Java-Klasse in einem vorher spezifizierten Verzeichnis abgespeichert. Auf die Methoden der Klasse kann von anderen Klassen mit jeweils konkreter Parameterbelegung zurückgegriffen werden, um mit den Informationssystemen nach den vorher spezifizierten Regeln interagieren zu können.

Mit den Methoden des generierten Codes kann für Informationssysteme durch Aufrufen der Systemtestfassadenmethoden überprüft werden, ob die Funktionalität des Informationssystems den durch die einzelnen Testfälle definierten funktionalen Anforderungen tatsächlich entspricht.

7.2. Ausblick

Die Systemtestgenerierung ist zwar bereits funktionsfähig, jedoch wurden nicht für alle Elemente von Htmlseiten entsprechende Codegenerierungsregeln aufgestellt. Eine Erweiterung könnte daraus bestehen, noch nicht definierte Seitenelemente wie z.B RadioButtons im Metamodell zu definieren und dafür XPAND-Regeln aufzustellen. Auch Erweiterungen des Metamodells um weitere Datentypen für Parameter sowie Lookupstrategien für Items sind denkbar.

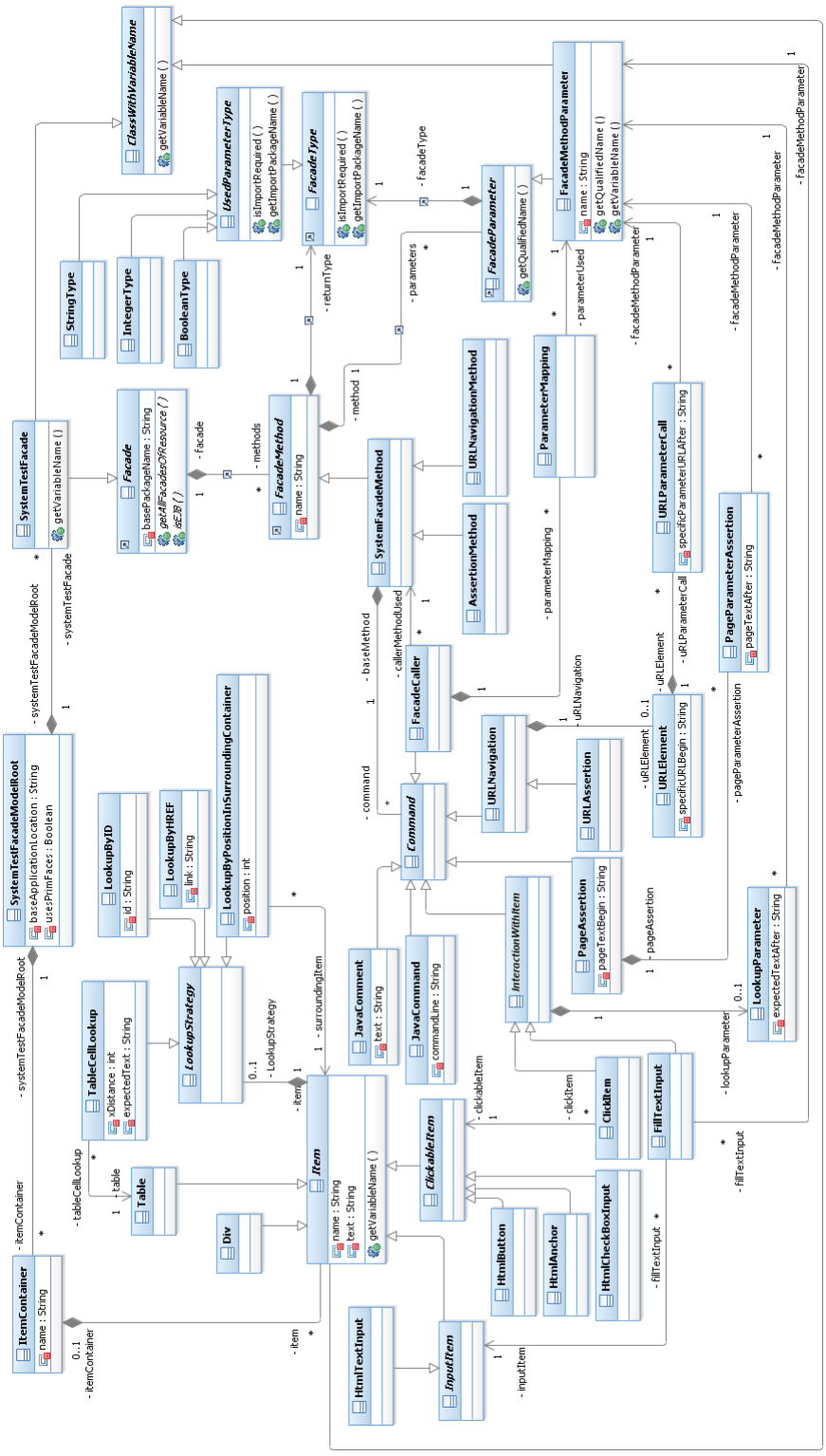
Bei den bereits implementierten Lookupstrategien hat vor allem die Lookupstrategie „LookupBySurroundingContainer“ noch Erweiterungspotential durch eine angepasste Elementsuche anhand des Seitenaufbaus.

Während in Check bereits durch viele Regeln festgehalten wird, ob sich bei der Benennung von Modellattributen Fehler einschleichen können, ist diese Liste an Regeln nicht komplett. Dafür ließen sich zusätzliche Überprüfungen einbauen, falls die Priorität steigen sollte, maximal Fälle ausschließen zu können, in denen der Anwender durch unkorrekte Attributbenennung fehlerhaften Code generieren lässt, ohne dass dazu eine entsprechende Warnung ausgegeben wurde.

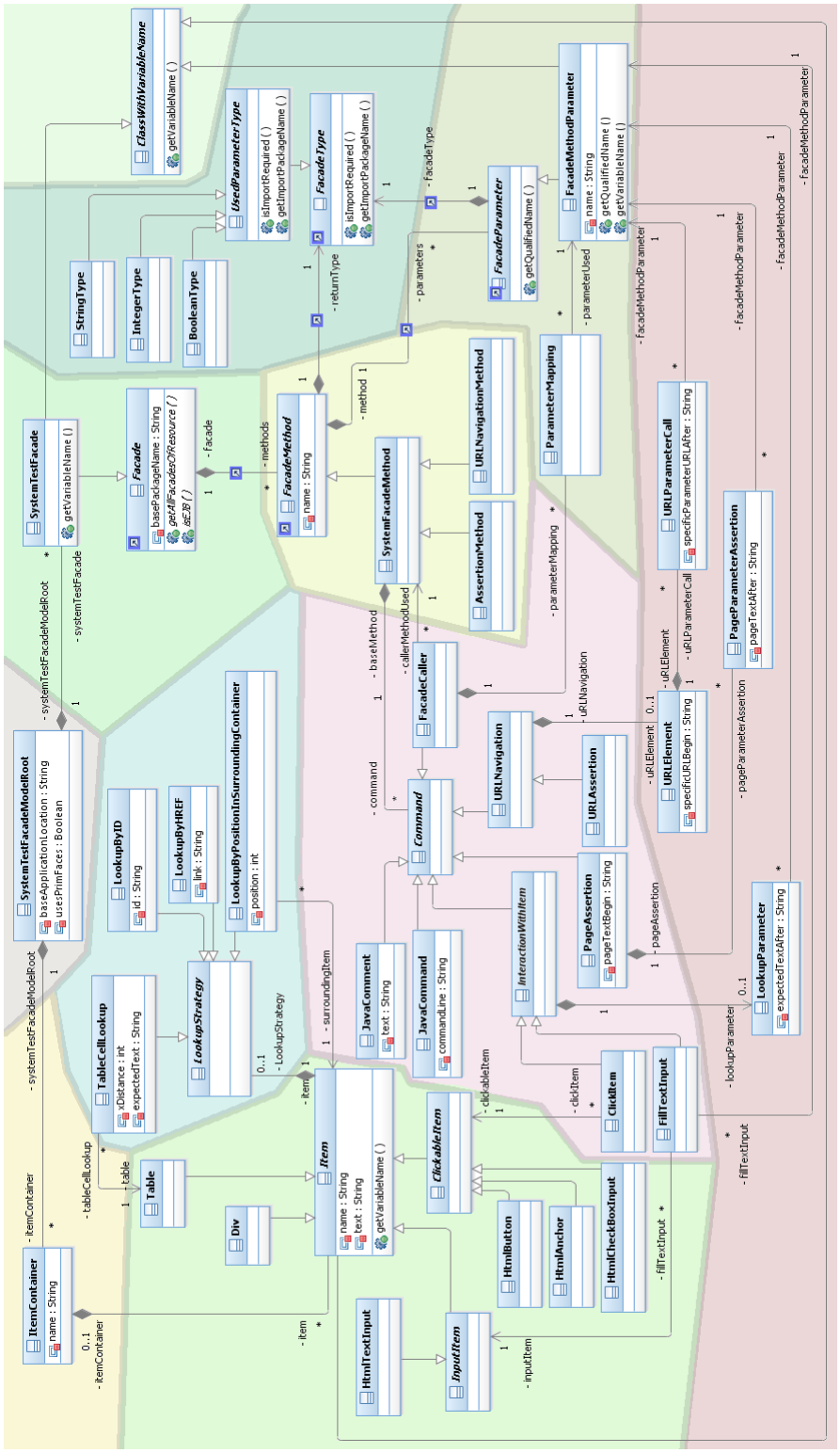
A. Dateien des Programms

Inhalt

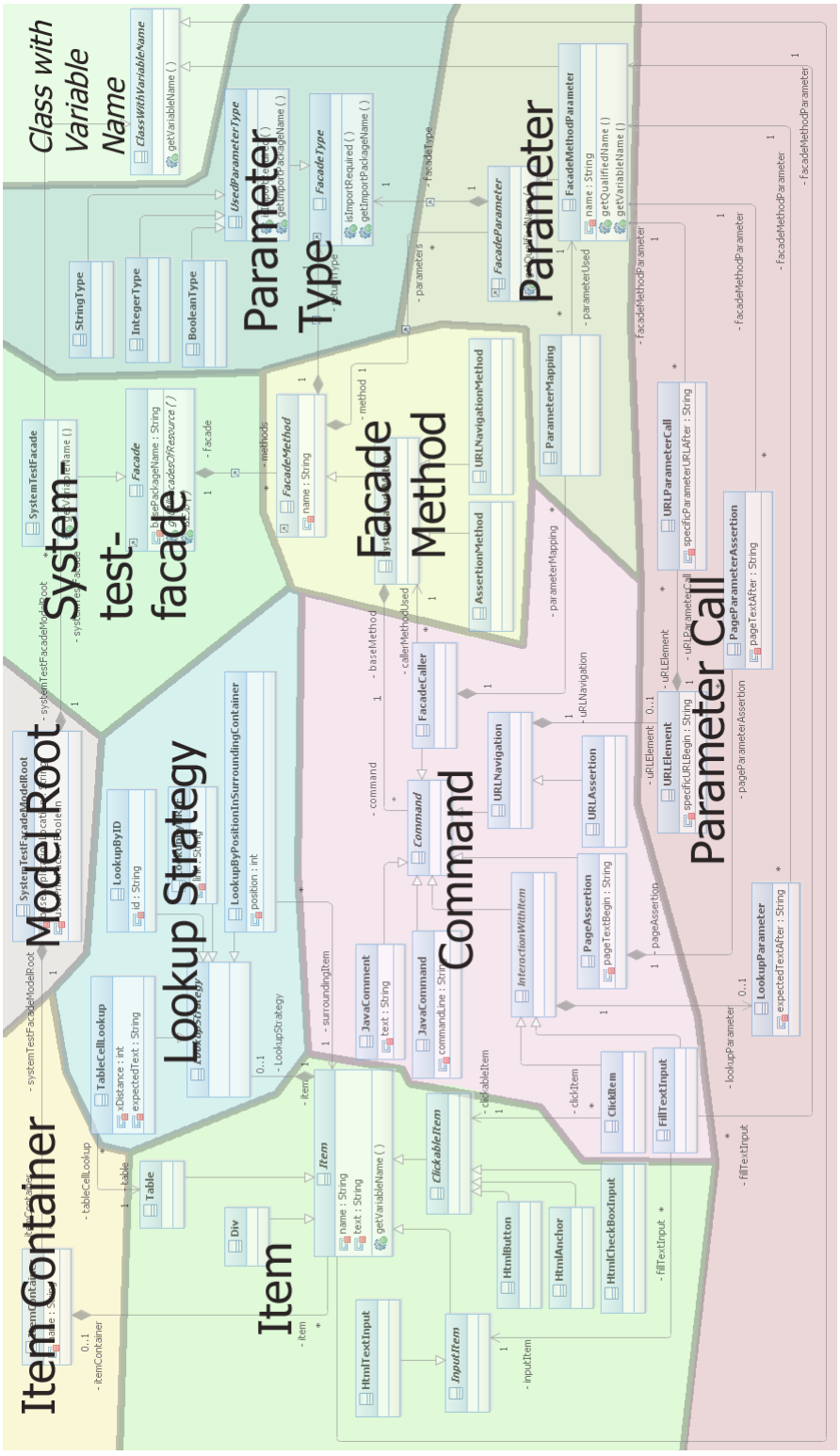
A.1. UML-Diagramm des verwendeten Metamodells



A.2. UML-Diagramm des verwendeten Metamodells nach Klassentypen gruppiert



A.3. Einteilung des Klassendiagramms nach Aufgabenbereichen



Literaturverzeichnis

[1] Michael Brocksch. *http://www.ahref.de/*, 2006.

[2] Steffen Conrad. Ein Ansatz zum modellgetriebenen Test von EJB-basierten Informationssystemen. Diplomarbeit, RWTH Aachen University, 2012.

[3] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[4] Peter Friese. Getting started with Code Generation with Xpand. www.peterfriese.de/getting-started-with-code-generation-with-xpand/, 12.03.2012.

[5] Golo Haas. Webbasierte Informationssysteme Einführung. Seminararbeit, Technische Universität Kaiserslautern, 2004.

[6] Horst Lichter. Software-Praktikum - Grundlagen, Folien, 2010.

[7] Tobias Löwenthal. Generierung von web-basierten Prototypen für Geschäftsanwendungen. Diplomarbeit, RWTH Aachen University, 2011.

[8] Kunal Mittal. Introducing IBM Rational Software Architect. http://www.ibm.com/developerworks/rational/library/05/kunal/, 2005.

[9] Eclipse Modeling Project. http://www.eclipse.org/modeling/, 12.03.2012.

[10] Bernhard Rumpe. *Script zur Vorlesung Softwaretechnik*. WS 2011/2012.

[11] Sven Efftinge, Clemens Kadura. Check / Xtend / Xpand Reference . www.openarchitectureware.org/pub/documentation/4.3.1/html/contents/core_reference.html, 12.03.2012.

[12] Horst Lichter und Jochen Ludewig. *Software EngineeringEngineering: Grundlagen, Menschen, Prozesse, Techniken.* dpunkt.Verlag GmbH, 2007.

[13] Canoo WebTest. http://webtest.canoo.com/webtest/manual/WebTestHome.html, 12.03.2012.