Fachhochschul-Masterstudiengang
SOFTWARE ENGINEERING
4232 Hagenberg, Austria

Rayden

Ein System für funktionale Tests mit Spezialisierung auf Abnahmetests

Masterarbeit

zur Erlangung des akademischen Grades
Master of Science in Engineering

Eingereicht von

Ing. Thomas Fischl Bsc

Betreuer: Stefan Reiner, Borland Entwicklung GmbH, Linz
Begutachter: FH-Prof. DI Dr. Heinz Dobler

Juni 2015

software engineering
IKM Fakultät Hagenberg

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Hagenberg, am 17. Juni 2015

Thomas Fischl

Inhaltsverzeichnis

Dank

Zu Beginn möchte ich mich ganz herzlich bei meinem Betreuer Heinz Dobler bedanken. Er hat mich bereits bei der Bachelorarbeit betreut und ich bin froh, dass er mich auch wieder bei der Masterarbeit betreut hat. Seine Kompetenz im Bereich des Compilerbaus und sein Verständnis für berufsbegleitend Studierende hat zum Erfolg dieser Masterarbeit erheblich beigetragen.

Weiters möchte ich mich bei meinen Eltern und bei meiner Schwester bedanken. Sie haben mich immer ermutigt weiter zu machen und haben mich in allen meinen Entscheidungen unterstützt. Bei meiner Schwester möchte ich mich auch noch speziell für das Design des Rayden-Logos bedanken.

Aber am meisten möchte ich mich bei meiner Freundin Angelika bedanken. Sie war die gesamte Studienzeit für mich da und hat mich tatkräftig unterstützt. Ich bedanke mich vor allem für das große Verständnis, da ich an vielen Wochenenden mehr Zeit mit dem Studium als mit ihr verbracht habe. Danke.

Kurzfassung

In den letzten Jahren ist das Automatisieren von Tests wieder in den Fokus von Testmanagerinnen und Testmanagern gerückt. Gründe dafür sind die Verkürzung der Release-Zyklen und ein immer größerer Kostendruck. Daher stehen viele Testabteilungen vor dem Problem, ihre manuellen Tests zu automatisieren.

Ein Lösungsansatz dafür ist der *Keyword-Driven-Testing*-Ansatz, welcher sich in letzter Zeit großer Beliebtheit erfreut. Für diesen Testansatz wurden einige Open-Source-Werkzeuge, aber auch kommerzielle Lösungen entwickelt. Jedoch hat dieser Ansatz neben vielen Vorteilen auch einige Nachteile: Je größer die Projekte werden, desto schwieriger wird die Verwaltung der Tests, da es nur wenige Werkzeuge gibt, welche mit großen Testprojekten umgehen können.

Dieser Ausgangspunkt stellt die Motivation für diese Masterarbeit dar, in der ein neues System mit dem Namen Rayden entwickelt wird, welches den *Keyword-Driven-Testing*-Ansatz umsetzt. Jedoch setzt diese Lösung auf einen Compiler, um eine bessere Unterstützung für die Verwenderinnen und Verwender bieten zu können. In Rayden wird auch das Konzept eines *Object Repositories* integriert, das dabei helfen soll, Abnahmetests leichter und besser lesbar zu schreiben.

Um die Fähigkeiten von Rayden zu zeigen, wird eine Web-Anwendung mit diesem System getestet. Dabei wird gezeigt, wie man unterschiedliche Testmethoden mit Rayden vereinen kann und welche besonderen Stärken im Bezug auf Abnahmetests existieren.

Abstract

The automated testing of software became more important for test managers in the last couple of years. The reasons are shorter release cycles and increasing cost pressure. Therefore, many test departments are facing the problem of automating their manual tests.

One approach for this problem is keyword-driven testing, which became very popular in the last few years. Some open source tools and also some commercial solutions have been developed for this approach. However, this approach has many advantages but also some drawbacks: It becomes more and more difficult to maintain large projects. One reason for that is the limited tool support for this approach.

This is the starting point for this master thesis. A new framework called Rayden, which covers the keyword-driven testing approach, is developed. The innovation of Rayden is, that the solution relies on a compiler to provide better tooling support for users. Another benefit of Rayden is an excellent integration of an object repository. All these advantages should help the user to develop better readable and maintainable tests.

A Web application is tested to demonstrate the capabilities of Rayden. Some usages of different testing techniques are shown in the demonstration. This should help users to understand the benefits of Rayden.

Kapitel 1

Einleitung

1.1 Zur Projektbezeichnung Rayden

Der Begriff *Rayden* ist abgeleitet von dem japanischen Wort *Raijin*, welches im japanischen Volksglauben der Name des Donner-Gotts ist. In der westlichen Welt wird der Name aber meist *Raiden* geschrieben, woraus für diese Arbeit die Bezeichnung *Rayden* abgeleitet wurde. Die Abbildung 1.1 zeigt das Logo für Rayden, welches von Agnes Fischl gestaltet wurde.

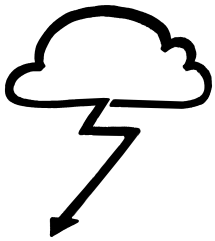

Abbildung 1.1: Rayden-Logo

1.2 Motivation

Viele Softwareunternehmen haben in den letzten Jahren große Testabteilungen aufgebaut. Der Fokus in diesen Abteilungen liegt sehr häufig noch auf dem manuellen Testen der Benutzeroberflächen. Dabei müssen für jede neue Version der Software viele manuelle Schritte durchlaufen werden. Dieser Vorgang ist sehr zeit- und kostenintensiv. Durch den Vormarsch neuer Entwicklungsmethoden und wegen des starken Kostendrucks stehen viele dieser Abteilungen vor einem Problem. Auf der einen Seite müssen sie Kosten einsparen, auf der anderen Seite werden die Release-Zyklen immer kürzer,

was einen noch größeren Aufwand bedeutet. In diesem Spannungsfeld überlegen viele dieser Unternehmen, ihre manuellen Tests zu automatisieren, um dadurch langfristig Zeit und Geld zu sparen.

Dieser Transformationsprozess stellt die Organisationen vor große Herausforderungen. Sie haben tausende Stunden von Expertenwissen in die manuellen Tests investiert. Für die Automatisierung steht jedoch selten derselbe Umfang an Zeit und Geld zur Verfügung. Auch muss der Prozess meistens parallel zu den bestehenden manuellen Tests vollzogen werden, da man kaum eine vollständige Umstellung auf einmal erledigen kann.

Um diesen Prozess für die Testabteilung zu erleichtern, benötigt es ein mehrschichtiges Test-*Framework*, das in der Lage sein muss, bestehende manuelle Tests wiederverwenden zu können. Dabei darf die Lesbarkeit der manuellen Tests aber nicht verloren gehen, da diese in Ausnahmefällen noch von einer Testerin oder einem Tester manuell durchgeführt werden müssen.

1.3 Problemstellung

Viele Testabteilungen arbeiten heutzutage noch größtenteils mit manuellen Tests. Diese Tests sind über Jahrzehnte gewachsen und es wurden tausende von Stunden in die Erstellung und Wartung investiert. Diese Testabteilungen bestehen aus vielen Testerinnen und Testern, welche die Tests für jede neue Version einer Software manuell ausführen. Es kommt nicht selten vor, dass aus Zeitgründen nicht alle Tests für jede Version ausgeführt werden können. Diese Situation hat sich durch den Einsatz von agilen Entwicklungsprozessen und kürzeren Release-Zyklen noch deutlich verschärft.

Diese Entwicklungen machen es notwendig, dass sich Testabteilungen immer öfter mit dem Thema der Testautomatisierung auseinandersetzen müssen.

Das führt zu folgenden Herausforderungen für die Testabteilungen:

1. Für die Automatisierung der Tests steht oft nur ein geringes Budget zur Verfügung.

2. Das Wissen aus den manuellen Test darf nicht verloren gehen.

3. Während des Migrationsprozesses und auch danach muss es möglich sein, dass man automatisierte Tests manuell ausführen kann. Diese Eigenschaft kann notwendig werden, um fehlgeschlagene automatisierte Testausführungen nachträglich manuell verifizieren zu können.

4. Die bestehenden Automatisierungslösungen sind oft sehr technisch aufgebaut. Jedoch findet man in typischen Testabteilungen nur wenige Entwicklerinnen und Entwickler, welche mit diesen Lösungen arbeiten können.

Um alle Herausforderungen dieser Liste zu adressieren, reicht *eine* technische Lösung heutzutage nicht mehr aus. Ein Test-*Framework* in diesem Umfeld muss auf vielen unterschiedlichen Ebenen ansetzen und Unterstützung bieten.

Eine mögliche Lösung für Testabteilungen wäre eine Testmethode, welche es auch Testerinnen und Testern ohne fundierte Programmierkenntnisse ermöglicht, automatisierte Tests zu erstellen. Zusätzlich muss es mit dieser Testmethode möglich sein, bestehende manuelle Tests wieder zu verwenden. Schlussendlich müssen die automatisierten Tests noch immer in einem Format vorliegen, das es ermöglicht, diese auch manuell von einer Testerin oder einem Tester auszuführen.

1.4 Zielsetzung

Das Ziel dieser Arbeit ist es, die Fähigkeiten eines *Keyword-Driven-Testing*-Ansatzes mit einem *Object Repository* zu kombinieren. Im Zuge der Implementierung soll ein neues Test-*Framework* entwickelt werden, welches den Ansatz von *Keyword-Driven Testing* verwendet. Für das *Framework* soll eine neue Sprache entwickelt werden, welche die Bedürfnisse nach einer einfachen und gut lesbaren Sprache befriedigt. Zusätzlich soll die Sprache eine gute Unterstützung für das *Object Repository* liefern.

Im nächsten Kapitel werden die Grundlagen und verwendeten Technologien detailliert beschrieben.

Kapitel 2

Grundlagen und Technologien

In diesem Kapitel werden die grundlegenden funktionalen Testmethoden beschrieben, welche in der Softwareentwicklung angewendet werden. Andere Testbereiche der Softwareentwicklung, wie Performanztest und Penetrationstest, werden hier nicht behandelt, denn die vorliegende Arbeit zielt speziell auf die funktionalen Testmethoden ab und hat in diesem Bereich ihre Stärken, was aber nicht bedeutet, dass man dieselben Konzepte nicht auch für die anderen Testbereiche anwenden könnte.

Funktionale Tests haben die Aufgabe sicherzustellen, dass die Anforderungen aus der Spezifikation korrekt umgesetzt werden. Für die Umsetzung dieser Tests können sowohl manuelle als auch automatisierte Testmethoden verwendet werden, welche in Abbildung 2.1 dargestellt sind. Die unterschiedlichen Methoden werden in diesem Kapitel genauer beschrieben und es wird auf die Unterschiede eingegangen.

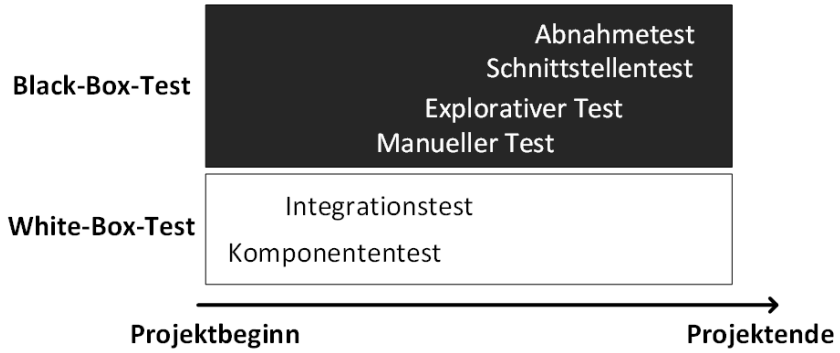

Abbildung 2.1: Testmethoden unterteilt in *White-Box-* und *Black-Box-*Testen

Im zweiten Teil dieses Kapitels werden die verwendeten Technologien beschrieben. Dazu gehören die Werkzeuge und Bibliotheken, welche für die Entwicklung von Rayden verwendet werden. Diese Werkzeuge und Bibliotheken werden benötigt, da Rayden aus einer Sprache, einem Compiler, einem Interpretierer und weiteren Komponenten besteht. Weiters werden zwei Testtreiber-Bibliotheken beschrieben, welche für automatisierte Abnahmetests verwendet werden können.

2.1 *White-Box*-Test

Unter *White-Box*-Tests versteht man Tests, bei denen die Testerinnen und Tester Zugriff auf den Quelltext haben. *White-Box*-Tests werden speziell für bestimmte Codefragmente geschrieben und testen gezielt einzelne Teile einer Anwendung. Diese Tests werden in einer frühen Phase des Entwicklungsprozesses erstellt und liefern sehr bald Qualitätskennzahlen.

White-Box-Tests sind typischerweise sehr technisch und verlangen vom Testpersonal Programmierkenntnisse. Daher werden diese Tests von Personen aus der Entwicklungsabteilung selbst geschrieben und fallen nicht in den Zuständigkeitsbereich des Qualitätssicherungsteams. Das gilt natürlich nur solange, als man sich nicht in einem agilen Entwicklungsprozess befindet. Dort werden sowohl die *White-Box*- als auch die *Black-Box*-Tests im Entwicklungsteam umgesetzt.

2.2 *Black-Box*-Test

Die *Black-Box*-Tests sind klassische Aufgaben eines Qualitätssicherungsteams. Sie umfassen alle Testansätze, bei denen der Quelltext der Anwendung nicht vorliegt. Dabei ist die Anwendung also eine *Black Box*. Die Aufgabe des Qualitätssicherungsteams ist es, zu überprüfen, ob alle Anforderungen laut Spezifikation umgesetzt wurden. Für diese Aufgabe steht dem Qualitätssicherungsteam eine ganze Reihe an unterschiedlichen Ansätzen zur Verfügung, angefangen von manuellen Tests über explorative Tests bis hin zu automatisierten Abnahmetests.

Die *Black-Box*-Tests werden typischerweise im fortgeschritten Projektstadium durchgeführt. Das ergibt sich aus der Tatsache, dass man für die *Black-Box*-Tests eine lauffähige Anwendung benötigt.

2.3 Manuelle Testmethoden

Bei manuellen Tests handelt es sich generell um *Black-Box*-Tests. Dabei überprüft die Testerin oder der Tester, ob sich die Anwendung in Bezug auf

die in der Spezifikation angegebenen Anforderungen korrekt verhält und ob die Funktionalität vollständig vorhanden ist. Die Funktionalität wird typischerweise über die Benutzeroberfläche geprüft. Eine zusätzliche Aufgabe bei manuellen Tests ist es, zu überprüfen, ob die Benutzeroberflächen-Konzepte korrekt und einheitlich umgesetzt wurden.

Bei manuellen Tests ist es typisch, dass die Testmanagerin oder der Testmanager eine textuelle Beschreibung der Testfälle erstellt. Diese Testfälle werden dann von dem Qualitätssicherungsteam für jede neue Version der Anwendung durchgeführt. Bei einer Abweichung der Anwendung muss entschieden werden, ob sich der Anwendungsfall geändert hat oder ob die Anwendung nicht korrekt funktioniert. Im letzteren Fall muss ein Fehlerbericht verfasst und an die Entwicklungsabteilung gesendet werden.

Eine Spezialform des manuellen Testens ist das explorative Testen. Dabei bekommt die Testerin oder der Tester keine genaue Vorgabe, wie ein Anwendungsfall getestet werden soll. In diesem Fall bekommt die Person nur eine Aufgabe gestellt, welche mit der Anwendung gelöst werden muss. Das Ziel ist es, dass man unterschiedlichste Möglichkeiten der Anwendung testen kann. Dieser Ansatz ist gut dafür geeignet, um neue Fehler zu finden.

Grundsätzlich haben manuelle und automatisierte Tests die Limitierung, dass nur festgestellt werden kann, ob eine neue Version einer Anwendung gleich gut funktioniert wie die alte. Es können aber keine neuen Fehler abseits der definierten Tests gefunden werden. Diese Lücke versucht das explorative Testen zu schließen. Es ist auch von Vorteil, wenn nicht immer die gleiche Person dieselbe Aufgabe testet. Jede Person hat neue Ideen, wie man die Aufgabe lösen kann und testet daher neue Bereiche und Kombinationen der Anwendung. Dieser Ansatz ist ein kreativer Prozess und kann daher im Gegensatz zu klassischen manuellen Tests nicht automatisiert werden.

2.4 Automatisierte Testmethoden

Das Ziel von automatisierten Tests ist es, dass man den Testaufwand in einem Softwareprojekt reduziert. Aus wirtschaftlicher Sicht ist es viel besser, wenn das stupide Testen durch einen automatisierten Test erledigt wird. Dadurch reduzieren sich die Kosten für das Softwareprojekt. Bei einer manuellen Ausführung kann es bei mehrmaligen Wiederholungen eines Tests zu Aufmerksamkeitsverlusten kommen, was bei automatisierten Tests nicht der Fall ist.

Durch automatisierte Tests werden Qualitätssicherungsteams jedoch nicht obsolet. Auf der einen Seite müssen die automatisierten Tests auch von jemandem geschrieben und gewartet werden, auf der anderen Seite sind au-

tomatisierte Tests für exploratives Testen ungeeignet. Die Aufgabe des explorativen Testens wird auf absehbare Zeit immer durch Personen erledigt werden.

Schlussendlich gibt es noch einen weiteren wichtigen Vorteil von automatisierten Tests gegenüber manuellen Tests: Man kann automatisierte Tests viel öfter ausführen und sie liefern schneller eine Aussage über die Qualität der Software. Diese Zeitreduktion ist für agile Softwareentwicklungsprozesse sehr wichtig, denn damit bekommt das Entwicklungsteam schneller eine Rückmeldung darüber, ob das System (noch) korrekt funktioniert.

2.4.1 Komponententest (*Unit Testing*)

Bei einem Komponententest [Mes07] wird genau eine Softwarekomponente getestet. Eine Softwarekomponente ist eine abgeschlossene Einheit in einem Softwareprojekt, welche eine definierte Schnittstelle hat. Das kann zum Beispiel eine einzelne Klasse, aber auch ein ganzes Modul sein, wie zum Beispiel in Pascal. Aus diesem Grund wird der Komponententest auch oft als Modul-Test oder Unit-Test bezeichnet. In dem Fall, dass die zu testende Komponente eine Abhängigkeit von einer anderen Komponente hat, werden diese durch eine Test-Implementierung ersetzt. Der Vorteil von Komponententests ist, dass deren Erstellung und Wartung keinen großen Aufwand verursachen. Das ist auch der Grund, warum dieser Testansatz sehr beliebt und weit verbreitet ist. Die Beliebtheit dieser Variante kann man daran ablesen, dass es mittlerweile für so gut wie jede Programmiersprache eine Unit-Test-Bibliothek wie zum Beispiel *JUnit* [KE14] gibt. Der Vorteil ist aber auch der größte Nachteil bei diesem Ansatz: Die Komponenten werden einzeln getestet und man kann daher keine Aussage darüber treffen, wie sich das Gesamtsystem verhalten wird.

Um eine Aussage über das Verhalten des Gesamtsystems zu erhalten, kann man Integrationstests verwenden. Diese werden im nächsten Abschnitt erklärt.

2.4.2 Integrationstest (*Integration Testing*)

Der Integrationstest ist schon deutlich aufwendiger und umfangreicher als ein Komponententest. Bei einem Integrationstest werden alle Komponenten eines Softwaresystems gemeinsam getestet. Das Ziel bei diesen Tests ist es zu gewährleisten, dass alle Komponenten miteinander funktionieren und dass alle Schnittstellen korrekt implementiert wurden. Es werden auch unterschiedliche Fehlersituationen im System simuliert und überprüft, ob diese ausgeglichen werden können. Ein einzelner Fehler in einer Komponente soll nicht das ganze System zum Absturz bringen oder in einen ungültigen Zustand versetzen.

Bei einem Integrationstest stellt sich oft die Frage, ob man mit oder ohne Datenbank testen soll. Diese Frage kann man nicht so einfach beantworten. Auf der einen Seite kann man sagen, dass die Datenbank genauso eine Komponente im Softwaresystem ist, welche getestet werden muss. Auf der anderen Seite kann man argumentieren, dass die Datenbank ein externes System ist, welches bereits getestet wurde. Grundsätzlich ist jedoch zu sagen, dass es ein guter Ansatz ist, wenn man die Integrationstests mit einer Datenbank durchführt. Eine ausführliche Diskussion über diese Thematik kann man in *Der Integrationstest* [Win+12] nachlesen. Es kann immer wieder vorkommen, dass genau bei der Schnittstelle zwischen Softwaresystem und Datenbank Probleme auftreten. Diese Fehler würden sonst erst relativ spät im Projekt-Lebenszyklus auftreten und der Aufwand für die Behebung dieser Fehler würde steigen.

Der Grund, warum über dieses Thema so viel diskutiert wird ist, dass der Aufwand für einen Integrationstest mit Datenbank deutlich höher ist. Man muss eine Strategie finden, wie man für jede Testausführung einen definierten Datenbankzustand herstellen kann. Dieser Datenbankzustand ist sehr wichtig, um reproduzierbare Tests schreiben zu können.

2.4.3 Schnittstellentest (*API Testing*)

Der Schnittstellentest ist die Vorstufe zum Abnahmetest, bei welcher alle externen Schnittstellen getestet werden. Dabei kann es sich um eine Schnittstelle zu einem externen System handeln oder um eine Web-Service-Schnittstelle. Aber darunter fällt auch die Schnittstelle zwischen Benutzeroberfläche und Geschäftslogik. Gerade diese Schnittstelle ist für die Testautomatisierung sehr interessant, da man hierbei die Benutzeroberfläche nicht für das Testen benötigt, jedoch das Gesamtsystem testen kann. Der Vorteil liegt darin, dass dieser Testansatz deutlich stabiler ist als ein Abnahmetest, welcher die Tests über die Benutzeroberfläche ausführt. Auch ist die Durchlaufzeit eines Schnittstellentests deutlich geringer als die eines Abnahmetests.

Der Unterschied zwischen einem Schnittstellentest und einem Integrationstest ist, dass bei einem Schnittstellentest das Software-System vollständig installiert wird. Für die Tests wird eine vollwertige Datenbank mit realistischen Testdaten verwendet. Bei einem Integrationstest verzichtet man auf diesen Aufwand.

Wie schon die vorhergehenden Testansätze hat auch dieser Ansatz einen großen Nachteil: Bei diesen Tests werden nur die Schnittstellen zwischen externem System und der Benutzeroberfläche getestet. Dabei kann aber nicht sichergestellt werden, dass die Benutzeroberfläche fehlerfrei funktioniert. Für die Benutzerin oder den Benutzer der Anwendung zählt aber schlussendlich

nur, ob die Verwendung über die Benutzeroberfläche korrekt funktioniert. Aus diesem Grund sind all diese Testansätze kein Ersatz für die Abnahmetests.

2.4.4 Abnahmetest (*User Acceptance Testing*)

Abnahmetests sind die aufwendigsten und kostenintensivsten Aufgaben im Testprozess. Bei einem Abnahmetest wird die Anwendung aus Sicht der Benutzerin oder des Benutzers getestet. Das Qualitätssicherungsteam verifiziert, ob alle Anwendungsfälle und Funktionen, welche in der Spezifikation definiert worden sind, vorhanden sind. Dafür muss eine lauffähige Anwendung vorhanden sein, um diese Tests durchführen zu können. Im Wasserfall-Vorgehensmodell kommt dieser Testansatz am Ende des Entwicklungszyklus. Es kommt dabei nicht selten vor, dass die Kundin oder der Kunde diese Tests manuell durchführt. Bei den agilen Vorgehensmodellen werden diese Tests nach jeder Iteration durchgeführt. Durch die kurzen Iterationszyklen können die Abnahmetests nicht mehr manuell durchgeführt werden. In diesem Fall kommen automatisierte Abnahmetests zum Einsatz.

Die große Herausforderung bei diesem Testansatz ist es, die Balance zwischen manuellen und automatisierten Tests zu finden.

2.5 Verwendete Technologien

Rayden basiert auf und verwendet eine Vielzahl von unterschiedlichen Technologien, Werkzeugen und Bibliotheken. Dieser Abschnitt gibt einen Einblick in diese Technologien und erklärt, in welchen Bereichen diese im Rayden-System verwendet werden. Als Basis wird die Programmiersprache Java und deren Laufzeitumgebung verwendet. Die Entscheidung für Java ist essentiell für das Projekt, um eine große Anzahl an unterschiedlichen Test-Szenarien zu unterstützen.

Für die Umsetzung der Sprache Rayden wurden viele Bibliotheken und Werkzeuge aus dem Eclipse-Umfeld verwendet. Als Testtreiber-Bibliotheken werden sowohl die offene Selenium- als auch die kommerzielle *Borland Silk Test*-Implementierung verwendet.

2.5.1 Eclipse

Eclipse [Ecl13] ist eine Entwicklungsumgebung für eine Vielzahl an Programmiersprachen. Ursprünglich wurde Eclipse von IBM für die Sprache Java entwickelt. Im Laufe der Zeit wurde Eclipse zu einer beliebten Entwicklerplattform und es wurden immer mehr Sprachen über *Plug-ins* unterstützt. Auch für das Rayden-System soll ein solches *Plug-in* entwickelt werden, um eine gute Unterstützung bei der Erstellung von Tests bieten zu können.

Neben der Entwicklungsumgebung ist Eclipse aber auch eine Plattform für die unterschiedlichsten Projekte geworden. Diese Projekte werden von der *Eclipse Foundation* [Ecl15a] verwaltet und durch Partnerunternehmen und Freiwillige gepflegt.

Einige dieser Projekte werden in den nächsten Abschnitten separat vorgestellt.

2.5.2 *Eclipse Modeling Framework*

Das *Eclipse Modeling Framework* (EMF) [Ecl15b] ist ein Modellierungswerkzeug für Java. EMF stellte eine Menge an Werkzeugen zur Erstellung, Verwaltung und Weiterverarbeitung zur Verfügung. Dazu gehört auch die Möglichkeit, aus diesen Modellen Code zu generieren. Eine Kernkomponente von EMF ist das *ECore*-Metamodell. Ein Metamodell ist die Schablone für ein spezifisches Modell. Aus einem *ECore*-Modell kann man mithilfe von Code-Generatoren eine Java-Bibliothek generieren.

Auf diesem Konzept baut auch das xText-Projekt auf, welches im nächsten Abschnitt vorgestellt wird.

2.5.3 xText

Das xText-Projekt [Ecl15c] unterstützt das Erstellen von neuen Sprachen. Grundsätzlich ist xText ein Compiler-Generator der aus einer Grammatik einen lexikalischen und einen Syntax-Analysator generiert. Das Besondere an xText ist aber, dass man noch zusätzlich einen Eclipse-Editor für die Sprache bekommt. Der Editor bietet grundlegende Funktionen wie Syntax-*Highlighting*, Fehler- und Validierungsmechanismen. Diese Funktionalität kann man nachträglich noch anpassen und weitere Funktionen hinzufügen. Ein Vorteil von xText ist, dass man den generierten Compiler auch außerhalb von Eclipse als eigenständige Anwendung verwenden kann. Somit kann der Aufwand, zwei Compiler für seine Sprache warten zu müssen, eingespart werden. Der abstrakte Syntaxbaum einer Quelldatei wird im Compiler mit EMF umgesetzt. Das heißt, man bekommt einen vollständigen Syntax-Baum im Hauptspeicher, welchen man sehr einfach verarbeiten kann. Um die Verwendung noch zu vereinfachen, liegt für den Baum ein Metamodell in Form eines *ECore*-Modells vor.

2.5.4 Selenium

Selenium [Sel15] ist eine Open-Source-Bibliothek, um Web-Seiten automatisiert testen zu können. Die Bibliothek unterstützt eine Vielzahl an unterschiedlichen Browsern auf allen gängigen Betriebssystemen wie Windows,

Linux, Macintosh und Google Android. Um die Browser ansprechen zu können, benötigt man einen speziellen Treiber. Dieser wird entweder als separate Anwendung aus- oder bereits mit dem Browser mitgeliefert.

In der ersten Version hat Selenium auf eine proprietäre Programmierschnittstelle gesetzt. Seit der Version 2 setzt Selenium auf die standardisierte Programmierschnittstelle *WebDriver* [W3C15] des W3C Konsortiums. Der Vorteil von *WebDriver* ist, dass man eine einheitliche Programmierschnittstelle für die unterschiedlichsten Browser hat. Damit erzielt man Unabhängigkeit von einem spezifischen Browser.

2.5.5 Borland Silk Test

Borland Silk Test [Bor15] ist eine kommerzielle Testsoftware für native wie auch Web-Anwendungen. *Borland Silk Test* bietet Unterstützung für eine Vielzahl an unterschiedlichen Technologien. Unterstützt werden zum Beispiel die gängigen Browser wie Internet Explorer, Google Chrome und Mozilla Firefox. Neben Web-Technologien werden auch native Windows-, Adobe-Flex-, Windows-Presentation-Foundation- oder Java-Anwendungen unterstützt. Seit kurzem werden auch Browser und Anwendungen auf mobilen Geräten unterstützt. Der Vorteil von *Silk Test* gegenüber von Selenium ist, dass es einen verbesserten *X-Browser Support* gibt. Dabei kann man einen Test, welchen man zum Beispiel mit dem Internet Explorer aufzeichnet, mit einem Mozilla-Firefox- oder dem Google-Chrome-Browser ausführen. Durch diese *X-Browser*-Technologie entfällt die Wartung von Tests für die verschiedenen Browser.

Kapitel 3

Aufbau und Ablauf von Testprojekten

Dieses Kapitel befasst sich mit den Testabläufen in einem Softwareprojekt. Die Abläufe finden in unterschiedlichen Phasen eines Softwareprojekts statt und werden von unterschiedlichen Personengruppen durchgeführt. Dieses Kapitel gibt einen Einblick in diese Abläufe und beschreibt auch die Schwierigkeiten, die es in einem Testprojekt zu bewältigen gibt. Im zweiten Abschnitt wird die Evolution der Testautomatisierung beschrieben. Dabei werden unterschiedliche Testansätze vorgestellt, welche sich über die Zeit entwickelt haben. Einer dieser Testansätze stellt die Basis für das Rayden-System dar.

3.1 Ablauf eines Testprojekts

In einem Softwareentwicklungsprojekt gibt es nicht nur die Testphase, in welcher die Testabteilung eine wichtige Rolle spielt. Die Testabteilung ist in den meisten Phasen eines Entwicklungsprojekts involviert. Um die gesamten Testaufgaben in einem großen Projekt zu koordinieren, wird oft ein Testprojekt aufgesetzt. In einem Testprojekt werden alle Aktivitäten rund um die Qualitätssicherung vereint. Diese Aktivitäten beschränken sich aber nicht nur auf die Testabteilung. Es müssen auch Personen aus der Fachabteilung und der Entwicklungsabteilung eingebunden werden. Diese Schnittstellen zwischen den einzelnen Abteilungen stellen eine große Herausforderung für die Testmanagerin oder den Testmanager dar.

Die Komponenten- und Integrationstests werden in diesem Abschnitt nicht behandelt, da diese Testaktivitäten primär in der Entwicklungsabteilung durchgeführt werden. Der Fokus der Testabteilung liegt auf den manuellen und automatisierten Abnahmetests der Anwendung.

In den nachfolgenden fünf Unterabschnitten werden die einzelnen Aufgaben in einem Testprojekt beschrieben. Es wird beschrieben, wie Testfälle entwickelt werden und zu welchem Zeitpunkt in einem Softwareprojekt welche Testaktivitäten stattfinden.

3.1.1 Rollen in einem Testprojekt

Das Testteam besteht aus einer bunten Mischung an unterschiedlichen Personen. Die Verantwortung in einem Testprojekt trägt die Testmanagerin oder der Testmanager. Diese Person ist für die Koordination des Projekts zuständig und bildet die Schnittstelle zu anderen Abteilungen. Eine dieser Schnittstellen besteht zu der Fachabteilung. Von der Fachabteilung werden die Anwendungsfälle geliefert, welche in weiterer Folge in der Testabteilung umgesetzt werden. Für die Umsetzung der Testfälle sind die Testerinnen und die Tester zuständig. Für die Automatisierung von Testfällen besteht eine Schnittstelle zu der Entwicklungsabteilung, falls die Testabteilung über keine eigenen Entwicklerinnen oder Entwickler verfügt.

3.1.2 Testfall

Während der Konzeptionsphase eines Entwicklungsprojekts werden Anforderungen von der Fachabteilung aufgenommen. Aus diesen Anforderungen werden Anwendungsfälle für das gesamte Projektteam abgeleitet. In der Testabteilung werden aus den Anwendungsfällen Testfälle entwickelt. Die Testfälle werden benötigt, um einen Überblick zu bekommen, welche Bereiche einer Software getestet werden müssen. In einem Testfall wird beschrieben, wie der Anwendungsfall zu testen ist und wie das erwartete Ergebnis aussieht. Da diese Aufgabe wichtig ist und einen hohen Kommunikationsaufwand bedeutet, werden die Testfälle größtenteils von der Testmanagerin oder dem Testmanager erstellt. In großen Projekten wird diese Arbeit auch von erfahrenen Testerinnen oder Testern durchgeführt.

3.1.3 Manuelle Abnahmetests für Testfälle

Die Testfälle bestehen aus einer groben Beschreibung des zu testenden Anwendungsfalls. Weiters umfasst ein Testfall eine Schritt-für-Schritt-Anweisung, wie der Testfall ausgeführt werden soll. Ein Testfall wird in der ersten Phase von einer Testerin oder einem Tester durchlaufen. Für die Zuteilung der Testfälle ist die Testmanagerin oder der Testmanager zuständig.

3.1.4 Automatisierung von manuellen Abnahmetests

In regelmäßigen Abständen sieht sich die Testmanagerin oder der Testmanager die Ausführungshäufigkeit von manuellen Tests an. Werden manuelle

Tests häufig durchgeführt, werden diese Tests automatisiert. Bei der Automatisierung werden die manuellen Schritte mithilfe eines Test-*Frameworks* automatisiert. Durch die Automatisierung spart die Testabteilung Zeit und kann somit schneller Ergebnisse über die Qualität der Anwendung liefern.

3.1.5 Testdokumentation

Der große Vorteil von sauber spezifizierten Testfällen ist, dass man keine zusätzliche Testdokumentation benötigt. Wenn die Testfälle sorgfältig beschrieben sind und auch gewartet werden, dienen diese als Testdokumentation. Wurden Testfälle automatisiert, kann es passieren, dass die Implementierung des Testfalls nach einiger Zeit nicht mehr mit der Beschreibung übereinstimmt. Ein wichtiges Ziel bei der Automatisierung ist es daher, dass man die Testdokumentation aktuell hält. Aus diesem Grund gibt es Automatisierungsansätze, welche versuchen, die manuellen Testfälle direkt zu automatisieren. Einige dieser Ansätze werden im nächsten Abschnitt 3.2 erklärt. Ein anderer Vorteil bei diesen Ansätzen ist, dass man die automatisierten Tests noch immer manuell ausführen kann. Diese Eigenschaft kann für die Verifikation von Ergebnissen sehr wichtig sein.

3.2 Evolution der Testautomatisierung

Die Automatisierung von Abnahmetests war in der Vergangenheit einem starken Wandel unterzogen. In diesem Bereich hat es eine ähnlich starke Entwicklung wie bei den Softwareentwicklungstechniken gegeben. Im Jahr 2009 haben Jeff Hinz und Martin Gijsen einen Artikel [HG09] über die Evolution der Testautomatisierung veröffentlicht. In diesem Artikel teilen die beiden Autoren die Entwicklung der Testautomatisierung in fünf Generationen ein. Jede Generation zeichnet sich durch eine spezielle Technik aus, wie die Tests entwickelt werden.

Die Abbildung 3.1 zeigt die einzelnen Entwicklungsstufen. In den nächsten Unterabschnitten werden die Techniken vorgestellt und es wird auf die Vorteile und Nachteile eingegangen.

3.2.1 Erste Generation - *Record-Replay*

Die erste Generation von Testtechniken sind die *Record-Replay*-Ansätze. Diese Ansätze besteht aus zwei Phasen:

- In der ersten Phase werden mithilfe einer Analysesoftware die Aktionen der Benutzerin oder des Benutzers mit der Anwendung aufzeichnet. Dabei werden typischerweise die Mausbewegungen und

die Tastatureingaben aufgezeichnet.

- In der zweiten Phase werden die aufgezeichneten Aktionen mit einer speziellen Software wieder abgespielt. Die Testsoftware umfasst deshalb spezielle Maus- und Tastatur-Treiber, um die aufgezeichneten Aktionen wiedergeben zu können.

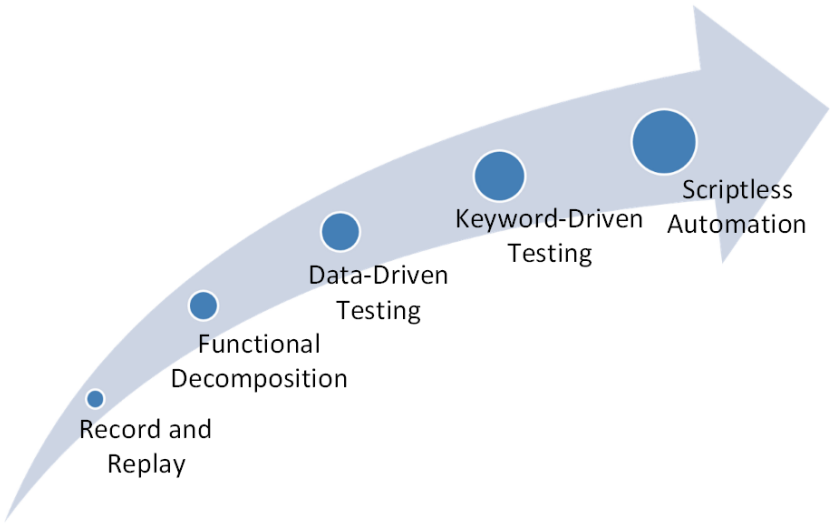

Abbildung 3.1: Die fünf Generationen von Testtechniken

Der große Vorteil bei dieser Methode ist die Einfachheit. Zum Aufzeichnen von Tests muss die Testerin oder der Tester den Anwendungsfall nur durcharbeiten und im Hintergrund werden die Aktionen automatisch aufgezeichnet. Für diese Testtechnik werden keine speziellen Fähigkeiten benötigt. Jedoch hat diese Technik einen schwerwiegenden Nachteil: Sobald sich die zu testende Anwendung nur marginal an der Oberfläche ändert, funktioniert diese Testmethode nicht mehr. Auch müssen die Tests immer mit derselben Bildschirmauflösung ausgeführt werden, um die Aktionen korrekt wiedergeben zu können. Ein weiterer Nachteil ist, dass, sobald man einen Test ändern möchte, der gesamte Test neu aufgezeichnet werden muss.

3.2.2 Zweite Generation - *Functional Decomposition*

Bei *Functional Decomposition* werden die Tests in einzelne Testsequenzen zerteilt. Mit dieser Technik können lange Tests in handliche Sequenzen zerteilt werden. Die Methode erlaubt auch die Wiederverwendung von Sequenzen in anderen Tests. Durch einen hohen Wiederverwendungsgrad kann die

Größe des Testprojekts stark reduziert werden. Einen weiteren positiven Effekt kann man in der Wartbarkeit des Testprojekts feststellen. Durch die Reduktion der Tests wird auch der Wartungsaufwand geringer.

Mit dieser Testmethode ist es nun auch möglich, Bibliotheken mit Testfunktionen für ein Testprojekt anzulegen.

3.2.3 Dritte Generation - *Data-Driven Testing*

In der dritten Generation von Testmethoden wird ein großes Augenmerk auf die Testdaten gelegt. In den vorgehenden Testtechniken lag der Fokus auf der Erstellung und Wartung von Testprojekten. Dabei mussten auch schon Testdaten verwendet werden, aber deren Stellenwert war nicht hoch. Die Testdaten stehen dafür nun in dieser Generation im Mittelpunkt. Man erkannte, dass man oft dieselbe Testsequenz durchläuft, aber jedes Mal andere Daten verwendet. Diese Testtechnik wird häufig für datenzentrierte Anwendungen verwendet.

Bei einem *Data-Driven Testing* werden im Test keine konkreten Werte verwendet. Stattdessen werden Platzhalter (in Form von Variablen) im Test eingebaut. Bei der Ausführung eines Tests werden die Platzhalter mit einem Wert aus einer Datenquelle verbunden. Als Datenquelle können Dateien oder Datenbanken dienen. Mit dieser Technik kann man denselben Test mit unterschiedlichen Parameterwerten ausführen.

3.2.4 Vierte Generation - *Keyword-Driven Testing*

In der vierten Generation von Testmethoden werden die Testdaten noch weiter in den Mittelpunkt gestellt. Bis zu diesem Zeitpunkt wurden die Tests entweder mithilfe eines *Record-Replay*-Ansatzes aufgezeichnet oder in einer Programmiersprache entwickelt. Der *Record-Replay*-Ansatz war einfach und auch von technisch nicht versierten Personen zu benutzen. Jedoch haben die aufgezeichneten Tests ein Problem mit der Zuverlässigkeit. Der zweite Ansatz *Functional Decomposition* bedingt, dass die Personen aus der Testabteilung Programmierkenntnisse benötigen.

Bei dem Ansatz *Keyword-Driven Testing* werden die Tests nun auch als Testdaten angesehen. Der Vorteil davon ist, dass Tests und Testdaten mit den gleichen Werkzeugen erstellt und verwaltet werden. Um einen *Keyword-Driven*-Test ausführen zu können, wird ein spezieller Interpretierer benötigt. Der Interpretierer liest die Tests von einer Datenquelle ein und arbeitet diese ab. Für die Verarbeitung müssen die Tests in einem lesbaren Format für den Interpretierer vorliegen. Eine detaillierte Beschreibung liefert zum Beispiel Pekka Laukkanen von der Universität von Helsinki in seiner Masterarbeit [Lau06].

3.2.5 Fünfte Generation - *Scriptless Automation*

Der modernste Ansatz versucht die Testautomatisierung mit einer *Scriptless Automation* zu vereinfachen. Bei diesem Ansatz wird aus einer abstrakten Repräsentation eines Tests Code erzeugt. Bei der Transformation von der abstrakten Repräsentation zum Quellcode, werden Code-Vorlagen und Code-Generatoren verwendet.

Bei diesem Ansatz wird wiederum versucht, die Größe des Testprojekts zu reduzieren und somit die Wartbarkeit zu erhöhen. Dieser Ansatz befindet sich noch in einer frühen Phase und hat in der Praxis bis jetzt noch keine Relevanz.

Im nächsten Abschnitt wird eine Implementierung des *Keyword-Driven Testings* vorgestellt, welche die Grundlage für das Rayden-System ist.

3.3 *Robot-Framework*

Das *Robot-Framework* [KH15] ist die Umsetzung des *Keyword-Driven-Testing*-Ansatzes und wurde ursprünglich von Nokia Siemens Networks entwickelt. Später wurde das Projekt unter die Apache-2-Lizenz gestellt und veröffentlicht. Das *Robot-Framework* stellt nicht nur eine technische Basis zur Verfügung, sondern bietet auch ein Vorgehensmodell an. Das Vorgehensmodell wird *Acceptance Test-Driven Development* (ATDD) genannt und im Artikel *Acceptance Test-Driven Development with Robot Framework* [CB10] erklärt.

```
1 *** Test Cases ***
2 Anmelden an der PetClinic Anwendung
3   [Documentation] Man meldet sich bei der Anwendung PetClinic mit
4   ...             den definierten Daten an. Wenn das Keyword
5   ...             erfolgreich ausgeführt wurde, befindet man
6   ...             sich auf der Hauptseite der Web-Anwendung.
7
8   Open Browser  ${URL}    ${Browser}
9   Input Text    user      TestUser
10  Input Text    password  secret
11  Click Button  login
```

Programm 3.1: Beispiel eines *Robot-Framework*-Testfalls

Das *Robot-Framework* verwendet als Testdaten-Format eine Tabulator-Syntax. Dabei werden die Daten durch Tabulatoren getrennt. Die Abbildung 3.1 zeigt einen Testfall, welcher in der Tabulator-Syntax definiert wurde. In

dem Testfall wurde die Selenium-Bibliothek für das *Robot-Framework* verwendet.

Das *Robot-Framework* unterstützt die Verwendung von Bibliotheken. In einer Bibliothek können *Keywords* zusammengefasst werden. Das *Robot-Framework* und die Entwicklergemeinde dahinter stellen eine große Anzahl an vorgefertigten Bibliotheken zur Verfügung. Die vorgefertigten Bibliotheken erleichtern und beschleunigen das Entwickeln von Test enorm. Somit muss man bei einem neuen Projekt nicht von vorne beginnen, sondern kann auf einen Fundus an *Keywords* zurückgreifen.

Ein anderer Vorteil dieser Bibliotheken ist es, dass auch Personen ohne technischen Hintergrund dieses *Robot-Framework* verwenden können. Die Bibliotheken sind weitestgehend vollständig, sodass man nur selten in die Lage kommt, in der man neue *Keywords* implementieren muss.

Neben den vielen Vorteilen des *Robot-Frameworks* gibt es aber auch Nachteile. Ein Nachteil ist die Tabulator-Syntax. Diese Syntax ist fehleranfällig und ohne einen speziellen Editor nur mühsam zu lesen. Auch fügt sich die Unterstützung von Kontrollstrukturen nicht optimal in das System ein.

Das *Robot-Framework* und die identifizierten Probleme bilden den Startpunkt für das Rayden-System, welches im nächsten Kapitel beschrieben wird.

Kapitel 4

Design von Rayden

Im vorigen Kapitel wurde der Ablauf eines Testprojekts aufgezeigt und eine Einführung in das Thema *Keyword-Driven Testing* gegeben. In diesem Kapitel wird das Rayden-System detailliert erklärt. Zu Beginn werden die Designziele der Sprache Rayden erklärt. Die Sprache Rayden ist eine domänenspezifische Sprache, welche einige Eigenheiten und Überraschungen enthält. In den weiteren Abschnitten wird der Aufbau des Rayden-Systems erläutert und auf die technischen Details eingegangen. Am Ende dieses Kapitels wird noch die Integration in die *Java-Scripting-API* [Ora14] beschrieben. Das Rayden-System bietet die Möglichkeit, einen Test in einer Java-Anwendung über das *Java-Scripting-API* auszuführen.

4.1 Designziele von Rayden

Das primäre Designziel von Rayden ist Offenheit. Rayden soll im gesamten Testprozess einsetzbar sein, darf aber die involvierten Personen nicht überfordern. Um dieses Ziel zu erreichen, setzt das Rayden-System auf mehreren Ebenen an.

Die domänenspezifische Sprache Rayden ist speziell für Personen im Testbereich ausgelegt. Ein wichtiges Ziel der Sprache ist Einfachheit. Die Sprache soll Personen ohne Programmierkenntnisse in die Lage versetzen, Tests in dieser Sprache lesen und bearbeiten zu können. Die Sprache Rayden ist daher stark an der natürlichen Sprache angelehnt, um den Einstieg zu erleichtern. Ein anderes Ziel beim Sprachdesign war die Flexibilität der Sprache. Der Testprozess setzt sich aus einer Vielzahl an unterschiedlichen Aufgaben zusammen. Um möglichst alle Aufgaben mit dieser Sprache abdecken zu können, wird eine hohe Flexibilität benötigt.

Abgesehen von einer geeigneten Sprache gibt es noch weitere wichtige Ziele für das Rayden-System. Rayden muss plattformunabhängig sein, um viele Anwendungsszenarien abdecken zu können. Aus diesem Grund wird die

Programmiersprache Java für die Entwicklung des Rayden-Systems gewählt. Der Interpretierer für Rayden läuft ebenfalls auf der *Java Virtual Machine* (JVM).

Die Einbindung von externen Testtreiber-Bibliotheken wird durch eine offene Schnittstelle ermöglicht. Dadurch können Rayden-Tests für die unterschiedlichsten Anwendungsszenarien entwickelt werden. Rayden kann somit für das jeweilige Projekt und die beteiligten Personen angepasst werden.

4.2 Aufbau des Rayden-Systems

In diesem Abschnitt wird der Aufbau des Rayden-Systems von zwei Blickwinkeln aus beleuchtet. Zuerst wird der konzeptionelle Aufbau erklärt. Dabei wird darauf eingegangen, wie die einzelnen Konzeptebenen miteinander kommunizieren und welche Person für die jeweilige Ebene verantwortlich ist. Im zweiten Teil wird die technische Architektur des Rayden-Systems erläutert. Dazu werden die Komponenten und ihre Beziehungen überblicksweise erklärt. Eine ausführliche Beschreibung findet man in den Abschnitten 4.2.1 und 4.2.2.

4.2.1 Konzeptioneller Aufbau

Wie schon in vorigen Abschnitten erwähnt, ist Rayden an das Konzept von *Keyword-Driven Testing* angelehnt. Bevor der Aufbau von Rayden beschrieben wird, wird der konzeptionelle Aufbau von *Keyword-Driven Testing* erläutert.

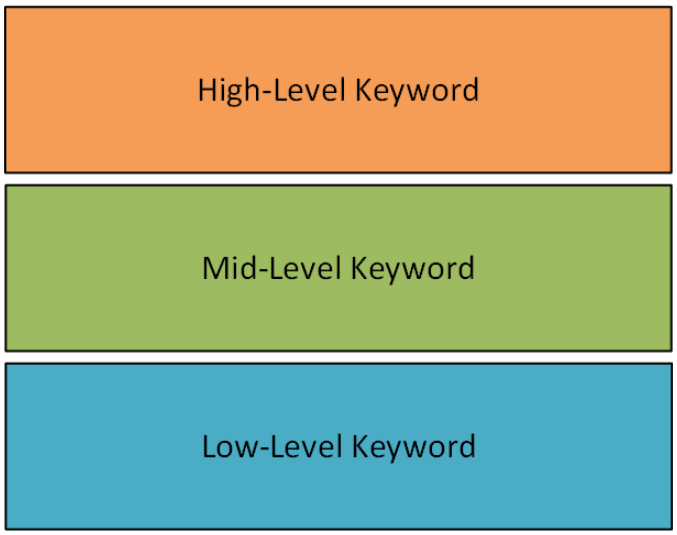

Abbildung 4.1: Aufbau von *Keyword-Driven Testing*

Ein *Keyword-Driven*-Test besteht aus einer Sequenz von *Keywords*. Diese *Keywords* können selbst wieder aus einer Sequenz von *Keywords* bestehen oder mit einem Codestück verbunden sein. Die *Keywords* werden in drei Kategorien, wie in Abbildung 4.1 dargestellt, aufgeteilt. Die *High-Level Keywords* repräsentieren einen Testfall mit einer detaillierten Beschreibung. Diese Gruppe von *Keywords* wird typischerweise von einer Person aus der Fachabteilung oder von einer Testmanagerin oder einem Testmanager erstellt. Dabei wird aber nur der Rumpf des *Keywords* erstellt. Die Implementierung wird erst in der nächsten Phase hinzugefügt. Diese *High-Level Keywords* bilden die Grundlage für die Erstellung der Tests. In einer weiteren Phase werden diese *Keywords* von Testerinnen und Testern implementiert.

Die *High-Level Keywords* bestehen typischerweise aus einer Sequenz von *Mid-Level Keywords*. Diese Sequenz wird in der zweiten Phase erstellt. Normalerweise werden in dieser Sequenz nur *Mid-Level Keywords* verwendet, jedoch können auch in Einzelfällen *Low-Level Keywords* verwendet werden. Ein Beispiel dafür sind Komponententests, da diese sofort mit einem *Low-Level Keyword* implementiert werden. Die *Mid-Level Keywords* bestehen selber wieder aus einer Sequenz von *Keywords*, jedoch befinden sich in dieser Sequenz sowohl *Mid-Level Keywords* als auch *Low-Level Keywords*. Technisch gesehen gibt es keinen Unterschied zwischen *High-Level Keywords* und *Mid-Level Keywords*. Der Unterschied besteht nur in der Art der Verwendung. *High-Level Keywords* beschreiben genau einen Anwendungsfall, der getestet werden soll. Im Gegensatz zu *Mid-Level Keywords* wird bei *High-Level Keywords* kein Wert auf Wiederverwendung gelegt.

In der letzten Phase werden *Low-Level Keywords* mit Code verbunden. Dieser Code kann prinzipiell in jeder Programmiersprache geschrieben sein. Die Wahl der Programmiersprache hängt von dem verwendeten *Keyword-Driven Framework* ab. In diesen *Keyword-Driven Frameworks* werden häufig Skript-Sprachen verwendet. Der Vorteil von Skript-Sprachen liegt darin, dass der Code für die Ausführung des Tests nicht kompiliert werden muss.

Im Gegensatz zu *Keyword-Driven Testing* unterteilt das Rayden-System die *Keywords* in mehr Gruppen, wie in Abbildung 4.2 dargestellt. Die zusätzlichen Gruppen bieten eine bessere Strukturierungsmöglichkeit und geben eine klare Richtung vor, wie ein Rayden-Test-Projekt aufgebaut werden soll.

Rayden führt eine klare Trennung bei *Low-Level Keywords* ein. Diese *Keywords*, welche mit einem Codestück verbunden sind, werden in *Library-* und *Bridge-Keywords* unterteilt. *Library-Keywords* stellen grundlegende Funktionen bereit, welche unabhängig von einem speziellen Anwendungsfall sind. Als Beispiel kann man sich ein *For-* oder *Print-Keyword* vorstellen. Im Gegensatz dazu sind *Bridge-Keywords* speziell für eine Anwendungstechnolo-

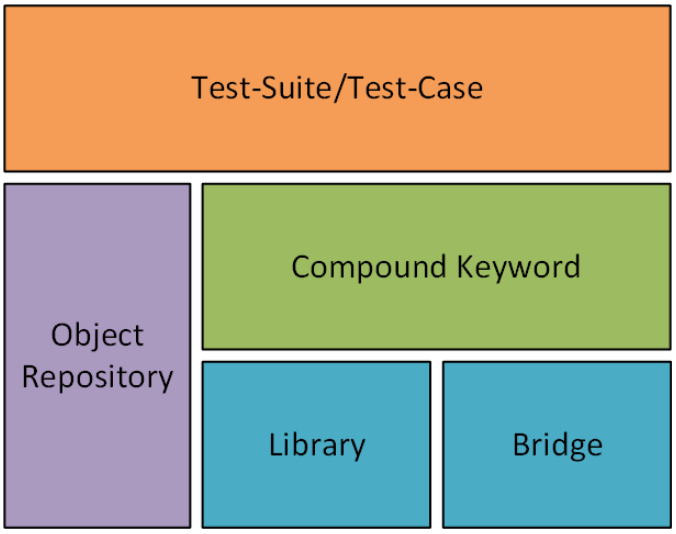

Abbildung 4.2: Aufbau von Rayden

gie angepasst, wie zum Beispiel *Open Browser* oder *Navigate To* für Web-Anwendungen.

Auch bei den *High-Level Keywords* bietet Rayden eine größere Vielfalt. Grob werden diese in Test-Suiten und Testfälle unterteilt. Die Test-Suite dient als Gruppierungselement für Testfälle, um diese gemeinsam ausführen zu können. Bei der Definition von Testfällen können diese mit unterschiedlichen *Keyword*-Arten angelegt werden. Eine nähere Beschreibung findet sich im Abschnitt 4.4.6.

Zum Abschluss ist noch auf das *Object Repository* hinzuweisen. Diese Komponente verwaltet Testobjekte. Testobjekte können für die Beschreibung von Benutzeroberflächen-Komponenten wie Schaltflächen oder Eingabefelder verwendet werden. Dafür wird für jedes Testobjekt ein Bezeichner definiert, mit welchem man die Komponente in der Benutzeroberfläche der zu testenden Anwendung finden kann. Das ist im Fall einer Web-Anwendung ein *XPath*- oder ein *CSS*-Ausdruck. Das *Object Repository* sorgt somit für eine klare Trennung zwischen Test- und Benutzeroberflächen-Beschreibung. Diese Trennung erhöht die Wiederverwendbarkeit von *Keywords* und reduziert den Wartungsaufwand bei Änderungen an der Benutzeroberfläche.

4.2.2 Technische Architektur

Die technische Basis für das Rayden-System ist die Java-Plattform. Auf der Entwicklungsseite wird Java als Programmiersprache für das gesamte Rayden-System verwendet, auf der Ausführungsseite läuft das Rayden-

System in der JVM. Außerdem bietet Rayden die Möglichkeit, dass man einen Rayden-Test über das *Java Scripting API* [Ora14] ausführen kann.

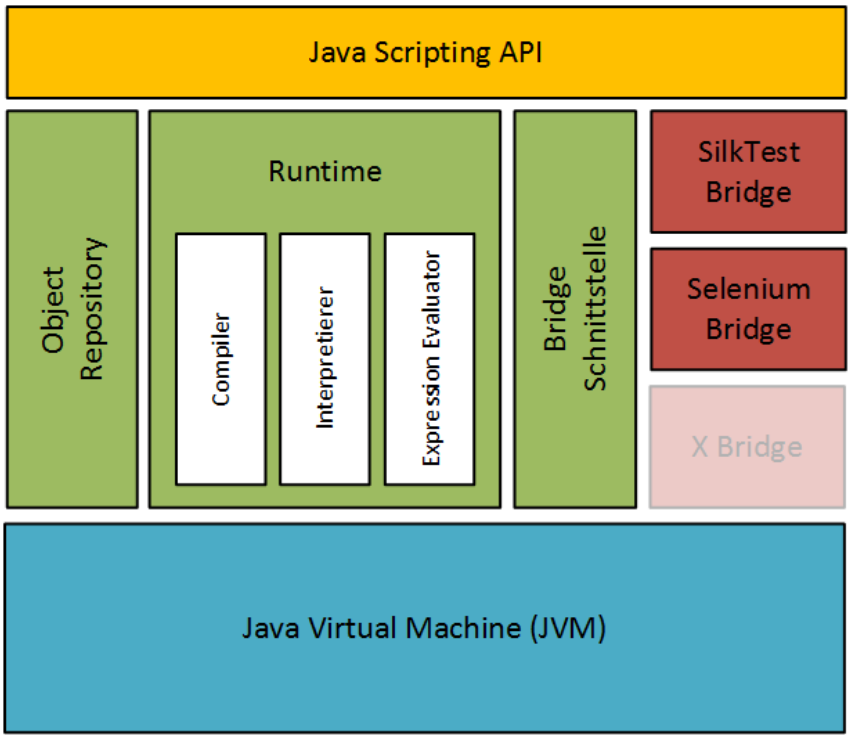

Abbildung 4.3: Rayden-Architektur

Abbildung 4.3 zeigt alle Komponenten des Rayden-Kernsystems in Grün. Diese Komponenten bilden die Grundlage dafür, einen Rayden-Test ausführen zu können. Als Basis dieser Komponenten sieht man in Blau die JVM. Die externen *Bridge*-Implementierungen werden in Rot dargestellt. Diese Komponenten stellen eine Verbindung zwischen dem Test-Treiber und der Rayden-*Runtime* her und werden über die *Bridge*-Schnittstelle hergestellt. Im oberen Abschnitt der Abbildung 4.3 sieht man das *Java Scripting API*, über welches man Rayden-Tests ausführen kann.

Nachfolgend werden die einzelnen Komponenten der Rayden-Architektur beschrieben, um einen groben Überblick über die Funktionsweise von Rayden zu geben.

- ***Runtime***
  Die *Runtime* ist der Einstiegspunkt für die Ausführung von Rayden-Tests. Dazu enthält diese Komponente die Implementierung für die *Java Scripting API*. Wird ein Test ausgeführt, werden zuerst alle

Projektressourcen in die *Runtime* geladen. Das Projektverzeichnis kann man über einen Kontextparameter setzen. Falls dieser nicht gesetzt ist, wird das aktuelle Verzeichnis verwendet. Für das Laden der Ressourcen wird die Compiler-Komponente verwendet, welche ein Bestandteil der *Runtime* ist. Die *Runtime* baut bei diesem Lesevorgang eine *Lookup*-Tabelle für alle *Keyword*-Namen auf. Diese Tabelle wird für einen schnellen Zugriff im Interpretierer benötigt. Der Interpretierer ist eine weitere Komponente der *Runtime*. Der Rayden-Test wird mithilfe des Interpretierers ausgeführt. Das Ergebnis des Tests wird als Resultat über die *Java Scripting API* zurückgegeben.

- **Compiler**
  Der Compiler für die Sprache Rayden wird mit dem Compilerbau-Werkzeug xText [Ecl15c] realisiert. Von dem generierten Compiler wird für die *Runtime* und den Interpretierer nur der lexikalische und syntaktische Analysator verwendet. Der Eclipse-Editor wird für das Rayden-System nicht benötigt. Das Resultat der Compiler-Komponente ist ein EMF-Modell des Tests. Die Basiskomponente *Runtime* verwaltet diese Modelle und stellt sie dem Interpretierer bei Bedarf zur Verfügung.

- **Interpretierer**
  Der Interpretierer, eine Teilkomponente der *Runtime*, ist die wichtigste Komponente im Rayden-System. Der Interpretierer ist dafür verantwortlich, dass die Rayden-Tests ausgeführt werden. Zum Starten des Interpretierers wird der Aufruf eines Test-*Keywords* übergeben. Dieses *Keyword* wird auf den leeren *Stack* geladen. Der Test wird dann mithilfe des Interpretierers abgearbeitet, welcher als *Stack*-Maschine [Wik15a] implementiert ist. Bei jedem Aufruf eines *Keywords* wird die *Keyword*-Implementierung auf den *Stack* geladen. Zusätzlich wird für jeden neuen *Keyword*-Aufruf ein neuer Gültigkeitsbereich (*Scope*) angelegt.

  Der Gültigkeitsbereich ist für die Verwaltung der Parameter und Variablen zuständig. In Rayden haben Gültigkeitsbereiche Zugriff auf andere Gültigkeitsbereiche. Eine detaillierte Beschreibung dazu findet sich im Abschnitt 4.4.7. Die Auswertung von Ausdrücken wird von der *Expression-Evaluator*-Komponente durchgeführt. Für die Auswertung eines Ausdrucks werden der aktuelle Gültigkeitsbereich und ein Teil des abstrakten Syntaxbaums, welcher den Ausdruck repräsentiert, an den *Expression Evaluator* übergeben. Das Ergebnis des Ausdrucks wird wieder auf den *Stack* gelegt. Ruft der Interpretierer ein *Scripted Keyword* (Beschreibung in Abschnitt 4.4) auf, wird

entweder der dazugehörige Code ausgeführt oder es wird der Aufruf an die *Bridge*-Schnittstelle weitergeleitet.

- ***Expression Evaluator***
  Der *Expression Evaluator* ist eine weitere Komponente der *Runtime*. Diese Komponente ist für die Auswertung von Ausdrücken verantwortlich. Damit die Ausdrücke im richtigen Gültigkeitsbereich ausgewertet werden, arbeiten der *Expression Evaluator* und der Interpretierer eng zusammen.

- ***Bridge*-Schnittstelle**
  Die *Bridge*-Schnittstelle ist für die Anbindung von Test-Treibern verantwortlich. Um einen Test-Treiber verwenden zu können, muss eine Rayden-*Bridge* implementiert werden. Die *Bridge* bildet die Verbindung zwischen der Rayden-*Runtime* und dem Test-Treiber.

- ***Object Repository***
  Das *Object Repository* verwaltet Testobjekte, welche von *Keywords* verwendet werden können. Die Testobjekte werden in einer Baumstruktur verwaltet. Die wichtigste Eigenschaft eines Testobjekts ist der Bezeichner (*Locator*). Mit dem Bezeichner kann die Benutzerschnittstellen-Komponente identifiziert werden. Das Konzept ist an der Idee des *Page-Object-Patterns* [Fow13] angelehnt.

4.3 Sprache Rayden

Als Inspiration und Basis für die Sprache Rayden dient das Konzept von *Keyword-Driven Testing*. Das Grundprinzip hinter *Keyword-Driven Testing* ist die Verwendung von *Keywords*. Ein *Keyword* besteht aus einer Sequenz von anderen *Keywords* oder ist mit einem Codestück verbunden. Einen *Keyword-Driven*-Test kann man sich auch als gerichteten Graphen vorstellen, in dem die Knoten die *Keywords* repräsentieren und die Kanten die Abhängigkeit zwischen den *Keywords* beschreiben, wie in Abbildung 4.4 dargestellt.

Der gelbe Knoten repräsentiert ein *High-Level Keyword*. Von diesem Knoten aus werden über gerichtete Kanten die Beziehungen zu den *Mid-Level Keywords* in Grün definiert. Man sieht, dass die *Mid-Level Keywords* entweder wieder in Beziehungen zu anderen *Mid-Level Keywords* stehen oder *Low-Level Keywords* referenzieren. Die *Low-Level Keywords* werden in Blau dargestellt. Bei dem Graphen handelt es um einen gerichteten azyklischen Graphen (DAG).

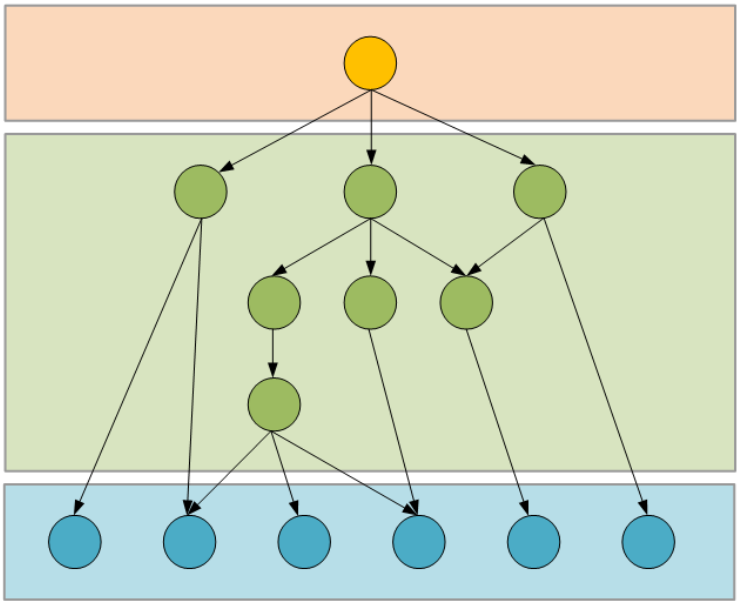

Abbildung 4.4: Graph-Repräsentation eines Tests

Die Sprache Rayden setzt auch auf dieses Konzept von *Keywords*. Im Unterschied zu *Keyword-Driven Testing* setzt Rayden auf eine größere Vielfalt an unterschiedlichen *Keywords*, welche im nächsten Abschnitt 4.4 detailliert beschrieben werden. Ein weiterer Unterschied ist die Benennung von *Keywords*. Normalerweise besteht der Name eines *Keywords* nur aus einem Wort, damit die Verarbeitung der Tests für den Compiler erleichtert wird. In Rayden wird großes Augenmerk darauf gelegt, dass man nicht nur auf ein Wort beschränkt ist, sondern auch ganze Sätze als *Keywords* verwenden kann. Diese Eigenschaft ist sehr nützlich, um die Testfälle wie in einer natürlichen Sprache beschreiben zu können. Der wesentliche Vorteil ist, dass man somit ohne weiteren Aufwand eine ordentliche Dokumentation des Tests bekommt.

Eine andere interessante Eigenschaft der Sprache Rayden ist, dass in ihr keine Sprung-Operationen vorgesehen sind. Die Konsequenz daraus ist, dass in der Sprache auch keine Schleifen- oder Verzweigungs-Konstrukte enthalten sind. Die einzige Möglichkeit, um ähnliche Konstrukte zur Verfügung zu stellen, sind *Scripted Compound Keywords*, wobei man bei diesem Metatyp von *Keyword* auch nur entscheiden kann, ob eine Sequenz von *Keywords* ausgeführt werden soll. Das Konzept der Metatypen wird im Abschnitt 4.4 beschrieben.

Da Sprung-Operationen vermieden werden und die Sprache blockstrukturiert ist, gewinnt man die Fähigkeit, Tests visuell darstellen zu können. Diese

Fähigkeit ist hilfreich, um eine bessere Unterstützung und einen leichteren Einstieg in die Sprache zu ermöglichen. Das ist vor allem von Vorteil, wenn Personen aus einer Fachabteilung nur unregelmäßig damit arbeiten müssen.

4.4 *Keywords* von Rayden

Das *Keyword* ist die Schlüsselkomponente der Sprache Rayden. In diesem Abschnitt werden die unterschiedlichen Arten und Metatypen erklärt und gezeigt, wofür diese verwendet werden können. Zu Beginn werden die vier Metatypen von *Keywords* erklärt. Die Metatypen bilden die Basis für den Funktionsumfang der Sprache. Ferner werden die unterstützten Arten beschrieben und wofür diese verwendet werden können. Zum Schluss werden noch Themen wie Gültigkeitsbereich, Datentypen und Parameter erläutert.

4.4.1 Metatypen

Der Metatyp definiert die Funktionsweise eines *Keywords*. Rayden unterscheidet zwischen vier Metatypen, wobei einer dieser Metatypen nur eine Kurzform ist.

4.4.2 Metatyp: *Compound Keyword*

Ein *Compound Keyword* ist die einfachste Variante eines *Keywords*. Bei einem *Compound Keyword* wird eine Sequenz von *Keywords* zu einem neuen *Keyword* zusammengefasst. Der Beispiel-Code 4.1 zeigt die Verwendung des *Compound Keywords Anmelden an der PetClinic Anwendung*. In dem Beispiel kann man gut sehen, dass dieser Metatyp hauptsächlich für die Strukturierung von Tests verwendet wird. Um die Funktionalität von *Keywords* zu erläutern, kann am Beginn eines *Keywords* eine mehrzeilige Beschreibung hinzugefügt werden. Eine Beschreibung beginnt und endet mit drei Hochkommas. Ein Vorgehen zur Strukturierung von einem Testfall kann sein, dass man diesen immer weiter und weiter in *Compound Keywords* zerlegt, bis man am Ende die Aufgabe auf einzelne Aktionen heruntergebrochen hat. Für diese Aktionen werden dann *Scripted Keywords* verwendet wie im Code-Beispiel die beiden *Keywords Type Text* und *Click Left*.

Ein klares Ziel bei der Erstellung von *Compound Keywords* ist die Wiederverwendung. Ein *Compound Keyword* soll als eine logische Einheit aufgebaut werden, sodass man diese auch wieder für andere Tests verwenden kann.

4.4.3 Metatyp: *Inline Keyword*

Der Metatyp *Inline Keyword* ist eine Kurzform des *Compound Keywords*. Dabei kann man in einem *Compound Keyword* ein neues *Keyword* erstellen.

```
keyword Anmelden an der PetClinic Anwendung {
  '''Man meldet sich bei der Anwendung PetClinic mit den definierten
     Daten an. Wenn das Keyword erfolgreich ausgeführt wurde,
     befindet man sich auf der Hauptseite der Web-Anwendung.'''

  parameter in username as string
  parameter in password as string

  Type Text(@PetClinic.LoginPage.Username, username)
  Type Text(@PetClinic.LoginPage.Password, password)

  Click Left(@PetClinic.LoginPage.LoginButton)
}
```

Programm 4.1: Beispiel für das *Compound Keyword Anmelden an der PetClinic Anwendung*

Daher kommt auch der Name *Inline Keyword*, weil es innerhalb eines anderen *Keywords* erstellt wird. Im Beispiel 4.2 wird im *Keyword Anmelden an der PetClinic Anwendung* das *Inline Keyword Besitzer anlegen* definiert. In diesem *Inline Keyword* werden alle Schritte zum *Anlegen eines neuen Besitzers* zusammengefasst. Ein *Inline Keyword* muss nicht explizit aufgerufen werden. Die Definition eines *Inline Keywords* ist auch automatisch auch der Aufruf dieses *Keywords*. Der Anwendungsfall dieses Metatyps ist wiederum die Strukturierung, aber in diesem Fall innerhalb eines *Keywords*.

```
testcase Anlegen eines neuen Besitzers {
  '''Der Testfall überprüft den Anwendungsfall um einen
     neuen Besitzer anlegen zu können.'''

  Anmelden an der PetClinic Anwendung ("max", "secret")

  Besitzer anlegen {
    Oeffnen der Besitzerseite
    Neuen Besitzer in der Anwendung anlegen("Huber", "Mayr")
    Daten von Besitzer ueberpruefen
  }

  Abmelden von der Anwendung
}
```

Programm 4.2: Beispiel *Inline Keyword*

Der Nachteil bei dieser Variante ist, dass man dieses *Keyword* nicht wiederverwenden kann. Das *Inline Keyword* ist nur innerhalb des *Compound Keywords* bekannt.

4.4.4 Metatyp: *Scripted Keyword*

Das *Scripted Keyword* ist im Gegensatz zum *Scripted Compound Keyword* der einfachere Metatyp, mit welchem man Code an ein *Keyword* binden kann. Ein *Scripted Keyword* wird wie ein *Compound Keyword* definiert. Im Unterschied dazu besitzt das *Scripted Keyword* keine Sequenz von *Keywords*, sondern einen Hinweis auf die Implementierung. Im Beispiel-Code 4.3 sieht man eine Variante mit einer Java-Implementierung. Die Anweisung *implemented in java* definiert die Implementierungssprache. Nach dem Pfeil folgt ein Bezeichner, welcher die Implementierung referenziert. Im Fall von Java wird der Klassenname mit dem *Package*-Namen verwendet.

```
1 keyword Print {
2   '''Der Parameter 'text' wird in den Test-Report geschrieben.'''
3
4   parameter text
5   implemented in java -> "com.github.thomasfischl.rayden.runtime.
      keywords.impl.PrintKeyword"
6 }
```

Programm 4.3: Beispiel *Scripted Keyword*

Um die Java-Klasse als *Keyword*-Implementierung verwenden zu können, muss die Klasse das *Interface ScriptedKeyword* implementieren. Das *Interface* hat nur die Methode *execute*. Kommt der Interpretierer zu einem Aufruf eines *Scripted Keywords*, wird ein neues Objekt der *Keyword*-Implementierung über den *Java-Reflection*-Mechanismus angelegt. Von diesem Objekt wird dann die Methode *execute* mit dem Namen des aktuellen *Keywords*, dem Gültigkeitsbereich und einem *Reporter*-Objekt aufgerufen. Auf die Parameter des *Keywords* kann man über den Gültigkeitsbereich zugreifen, wie man im Beispiel-Code 4.4 sehen kann. Als Ergebnis liefert die Methode ein *KeywordResult*-Objekt. Dieses Objekt signalisiert dem Interpretierer, ob das *Keyword* erfolgreich ausgeführt wurde.

Der Parameter *keyword* bei der Methode *execute* wird benötigt, weil es in Rayden möglich ist, eine Implementierung an mehrere *Keyword*-Definitionen zu binden. Mit diesem Parameter kann man den Namen der aktuellen *Keyword*-Definition abfragen.

Über das *Reporter*-Objekt kann man Einträge in den Test-Report hinzufügen. Die Instanz bietet unterschiedliche Granularitätsstufen für Nachrichten. Es werden spezielle Methoden für die Stufen Fehler, Warnung und Information angeboten. Diese Nachrichten können in der Folge von den jeweiligen *Reporter*-Implementierungen unterschiedlich behandelt werden.

```
1 public class PrintKeyword implements ScriptedKeyword {
2
3   @Override
4   public KeywordResult execute(String keyword,
5       KeywordScope scope, RaydenReporter reporter) {
6     reporter.log(scope.getVariable("text").toString());
7     return new KeywordResult(true);
8   }
9
10 }
```

Programm 4.4: Java-Implementierung des *Print Keywords*

4.4.5 Metatyp: *Scripted Compound Keyword*

Das *Scripted Compound Keyword* ist die komplizierteste Variante der vier Metatypen, jedoch ist diese Variante essentiell für die Flexibilität der Sprache Rayden. Mit dem Konzept von *Scripted Compound Keywords* ist eine Entwicklerin oder ein Entwickler in der Lage, die Sprache um Kontrollstrukturen zu erweitern. Dafür werden die Eigenschaften von *Compound Keywords* und *Scripted Keywords* kombiniert, wie in den Codeausschnitten 4.5 und 4.7 dargestellt.

```
1 keyword If {
2   parameter in condition as boolean
3   implemented in java -> "com.github.thomasfischl.rayden.runtime.
      keywords.impl.IfKeyword"
4 }
```

Programm 4.5: Beispiel *Scripted Compound Keyword*

Das *Scripted Compound Keyword* ist mit einem Codestück verbunden und hat zusätzlich noch eine *Keyword*-Liste. In der Implementierung des *Scripted Compound Keywords* hat man die Möglichkeit, die Ausführung der *Keyword*-Liste zu beeinflussen. Man kann damit eine bedingte bzw. mehrmalige Ausführung der Liste realisieren. Es ist aber zu beachten, dass man die Liste nur als Ganzes ausführen kann. Eine teilweise Ausführung der Liste ist nicht möglich.

Das Beispiel 4.5 zeigt die Definition für ein *If Keyword*. Dabei werden, wie bei einem *Scripted Keyword*, die Programmiersprache und der Bezeichner definiert. Im Fall eines *Scripted Compound Keywords* muss die Klasse das *Interface ScriptedCompoundKeyword* implementieren. Dieses *Interface* ist

deutlich aufwendiger zu implementieren, wie man im Beispiel-Code 4.6 sehen kann. In der Methode *executeBefore* wird mithilfe des Parameters *condition* entschieden, ob die *Keyword*-Liste ausgeführt wird. Die Definition des Parameters *condition* zeigt das Codebeispiel 4.5.

```
1 public class IfKeyword implements ScriptedCompoundKeyword {
2
3   private KeywordScope scope;
4
5   @Override
6   public void initializeKeyword(String keyword,
7     KeywordScope scope, RaydenReporter reporter) {
8     this.scope = scope;
9   }
10
11  @Override
12  public boolean executeBefore() {
13    return scope.getVariableAsBoolean("condition");
14  }
15
16  @Override
17  public boolean executeAfter() {
18    return false;
19  }
20
21  @Override
22  public KeywordResult finalizeKeyword() {
23    return new KeywordResult(true);
24  }
25
26 }
```

Programm 4.6: Java-Implementierung des *If Keywords*

Das *Interface* enthält für jede der vier Phasen eines *Scripted Compound Keywords* eine Methode, in der man die Ausführung steuern kann.

- **Phase 1: Initialisierung (*initializeKeyword*)**
  In der Initialisierungsphase wird der aktuelle Zustand von dem Interpretierer an die *Keyword*-Implementierung übergeben. Falls die Informationen für die Ausführung benötigt werden, können diese im Objekt der *Keyword*-Implementierung gespeichert werden. Für jede Ausführung eines *Keywords* wird ein neues Objekt der *Keyword*-Implementierung angelegt. Das heißt, man kann keinen globalen Zustand für zukünftige Ausführungen speichern. Falls man diese Funktionalität benötigt, muss man diese Daten in Klassenda-

tenkomponenten speichern.

- **Phase 2: Beginn der Auswertung (*executeBefore*)**
  Die Auswertung der *Keyword*-Liste kann in dieser Phase beeinflusst werden. Diese Methode wird vor jeder Auswertung der *Keyword*-Liste aufgerufen. Wenn diese Methode *false* liefert, wird die Liste nicht ausgewertet und es wird zur Phase 4 gesprungen.

- **Phase 3: Beendigung der Auswertung (*executeAfter*)**
  Nach der Ausführung der *Keyword*-Liste wird die Methode *executeAfter* aufgerufen. In dieser Phase wird entschieden, ob die Liste ein weiteres Mal ausgeführt werden soll. Wenn die Methode in dieser Phase *true* liefert, wird die Ausführung bei der zweiten Phase fortgesetzt. Ansonsten wird die vierte Phase ausgeführt.

- **Phase 4: Beendigung des Keywords (*finalizeKeyword*)**
  In der letzten Phase können noch Abschlussarbeiten vorgenommen werden, wie beispielsweise die Berechnung des Status für das *Keyword*. Der Status signalisiert, ob die Ausführung erfolgreich war oder nicht. Diese Funktionalität kann zum Beispiel für Validierungen verwendet werden.

Die Verwendung eines *Scripted Compound Keywords* sieht wie ein *Inline Keyword* aus. Der große Unterschied ist, dass ein *Inline Keyword* im Gegensatz zu einem *Scripted Compound Keyword* keine Parametersignatur hat. Ein Beispiel für die Verwendung des *If Keywords* findet man im Codeausschnitt 4.7.

```
1 keyword If Keyword Beispiel {
2   If (a == 1) {
3     Print("Condition is true")
4   }
5   If (b == "b") {
6     Print("Condition is false")
7   }
8 }
```

Programm 4.7: Verwendung des *If Keywords*

4.4.6 Arten

Neben den Metatypen gibt es in Rayden auch unterschiedliche Arten von *Keywords*. Die Arten liefern keine zusätzliche Funktionalität für die Sprache, sondern dienen als Strukturierungselement für Testprojekte. Mithilfe

der Arten kann man Testfälle unterscheiden und eine klare Zuordnung zu einer Testmethode treffen. Durch die Arten wird es auch möglich, eine Aussage über die Verteilung der Testmethoden in einem Testprojekt treffen zu können.

Rayden unterstützt die folgenden *Keyword*-Arten:

- Test-Suite (*TestSuite*),
- Testfall (*TestCase*),
- Komponententest (*UnitTest*),
- Integrationstest (*IntegrationTest*),
- Schnittstellentest (*APITest*),
- automatisierter Abnahmetest (*AUTest*) und
- manueller Abnahmetest (*MAUTest*).

In Rayden ist es aber nicht zwingend notwendig, diese Arten zu verwenden. Man kann anstelle einer Art einfach das Schlüsselwort *keyword* verwenden. Damit verliert man aber die Auswertungsmöglichkeit in einem Testprojekt.

4.4.7 Gültigkeitsbereiche

In Rayden wird für jeden Aufruf eines *Keywords* ein neuer Gültigkeitsbereich angelegt. In diesem Gültigkeitsbereich befinden sich alle Parameter, welche für das *Keyword* definiert sind. Nachdem Parameter in einem Gültigkeitsbereich definiert wurden, verhalten sich diese gleich wie Variablen. Variablen aus dem aktuellen Gültigkeitsbereich können von einem *Keyword* mit einem Wert belegt werden. Der Wert einer Variablen kann entweder in einem Ausdruck oder in einem *Keyword* verwendet werden. In Rayden müssen Variablen nicht deklariert werden und können den Datentyp während der Ausführung ändern. Sobald das erste Mal eine Variable mit einem Wert belegt wurde, ist diese im Gültigkeitsbereich vorhanden.

In Rayden gibt es aber noch eine Besonderheit in Bezug auf Gültigkeitsbereiche. Rayden verwendet für die Gültigkeitsbereiche das Konzept von *Dynamic Scoping*. Bei einem Aufruf eines *Keywords* wird der Gültigkeitsbereich mit den Parametern angelegt. Das Besondere ist, dass der neue Gültigkeitsbereich eine Referenz auf den alten Gültigkeitsbereich hat. Das Resultat ist, dass jeder Kind-Gültigkeitsbereich Zugriff auf den Eltern-Gültigkeitsbereich hat.

Ein Beispiel dazu sieht man in der Abbildung 4.5. Beim Starten eines Tests wird der Gültigkeitsbereich *G1* angelegt. Auf diesen Gültigkeitsbereich haben später alle anderen Gültigkeitsbereiche Zugriff. Daher eignet sich dieser Gültigkeitsbereich gut für globale Variablen.

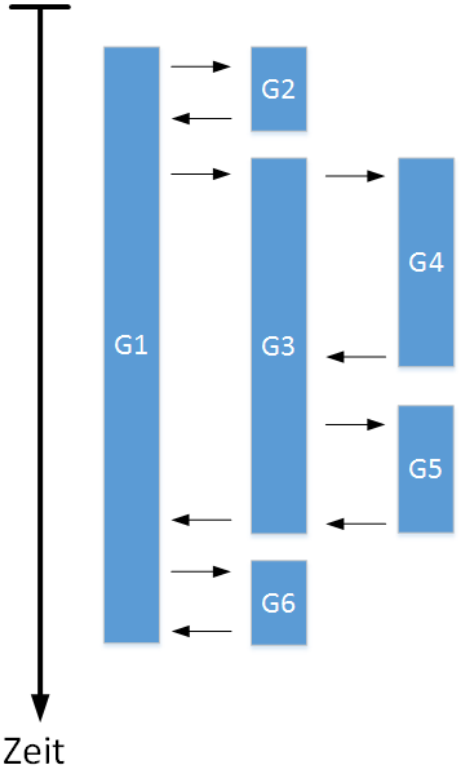

Abbildung 4.5: Gültigkeitsbereiche in einem Rayden-Test

In dieser Abbildung sieht man weiter, dass jeder neue Gültigkeitsbereich eine Beziehung zu einem Eltern-Gültigkeitsbereich hat. Der Pfeil von einem Kind- zu einem Eltern-Gültigkeitsbereich mag am Anfang ungewöhnlich wirken. Der Pfeil erklärt sich aber damit, dass man in Rayden *out*- und *inout*-Parameter definieren kann. Damit können Werte aus *G2* an *G1* übertragen werden. Eine detaillierte Beschreibung zu den Parametern findet sich im Abschnitt 4.4.8.

Der Vorteil von vererbten Gültigkeitsbereichen ist, dass nicht alle Variablen übergeben werden müssen, welche bei einem Test zahlreich vorkommen können. Die Parameter bieten die Möglichkeit der expliziten Definition von Variablen. Das wird verwendet, um sicherzustellen, dass eine Variable zur Verfügung steht bzw. erleichtert auch die Verwendung eines *Keywords*.

4.4.8 Parameter

Keywords unterstützen das Definieren von Parametern für eine einfachere Verwendung. Grundsätzlich werden Parameter in Rayden nicht zwingend

benötigt, da Rayden das Konzept von vererbten Gültigkeitsbereichen verwendet. Parameter ermöglichen jedoch eine explizite Schnittstelle für *Keywords*.

```
1 keyword Parameter Beispiel {
2
3   parameter in    param1
4   parameter in    param2 as string
5   parameter out   param3 as boolean
6   parameter inout param4 as number
7
8   Test1
9 }
```

Programm 4.8: Verwendung von Parametern

Rayden unterstützt sowohl typisierte als auch untypisierte Parameter. Sind die Parameter typisiert, werden diese vom Rayden-Interpretierer überprüft. Sind keine Werte für einen Parameter vorhanden, wird die Ausführung mit einem Fehler abgebrochen.

Neben einem Typ kann man bei einem Parameter auch noch die Richtung definieren. Die Richtung bezieht sich auf die Gültigkeitsbereiche. In Rayden werden die Richtungen *in*, *out* und *inout* unterstützt, wie das Code-Beispiel 4.8 zeigt.

- ***in*-Parameter**
  Ein *in*-Parameter transferiert einen Wert aus dem Eltern-Gültigkeitsbereich in den Kind-Gültigkeitsbereich. Das ist auch das Standardverhalten, falls keine Richtung bei einem Parameter definiert ist.

- ***out*-Parameter**
  Ein *out*-Parameter ist das Gegenteil zum *in*-Parameter. Dabei wird ein Wert aus dem Kind-Gültigkeitsbereich in den Eltern-Gültigkeitsbereich transferiert.

- ***inout*-Parameter**
  Die dritte Variante ist eine Kombination aus einem *in*-Parameter und einem *out*-Parameter.

4.5 Datentypen von Rayden

Die Sprache Rayden unterstützt die folgenden Datentypen:

- *number*,
- *string*,
- *boolean*,
- *variable*,
- *location* und
- *enumeration*.

Darunter befinden sich einige Standard-Datentypen wie *number*, *string* und *boolean*.

```
1 keyword Open Browser {
2   parameter in browserType as enumeration (IE | FF | CHROME)
3
4   implemented in java -> "selenium.OpenBrowserKeyword"
5 }
```

Programm 4.9: Verwendung eines *enumeration*-Parameters

Der Typ *enumeration* wird intern als *string* repräsentiert. Die Laufzeitumgebung sorgt dafür, dass nur die vordefinierten Werte zugewiesen werden dürfen. Diese Überprüfung wird aber nur bei einem Übergang von einem Gültigkeitsbereich in einen anderen Gültigkeitsbereich vorgenommen. Diese Einschränkung ist damit zu erklären, dass ein *enumeration*-Datentyp genau für ein *Keyword* definiert wird. Ein Beispiel dazu ist im Codeausschnitt 4.9 dargestellt.

Ein weiterer spezieller Datentyp ist *location*. Mit diesem Datentyp kann man ein Objekt in einem *Object Repository* referenzieren. Ein Wert dieses Datentyps beginnt immer mit dem @-Symbol. Nachfolgend kann man einen Pfad im *Object Repository* beschreiben, wie in Beispiel 4.10 gezeigt. Für Abnahmetests ist das Referenzieren von Testobjekten essentiell. Daher bietet Rayden dafür eine Erleichterung.

Falls der erste Parameter eines *Keywords* vom Datentyp *location* ist, kann man diesen Parameter vor das *Keyword* schreiben. Somit wird das Lesen eines Tests erleichtert. Eine Verwendung davon findet man ebenfalls im Beispiel 4.10. Diese Spracheigenschaft wird von der Rayden-Laufzeitumgebung wieder in einen klassischen *Keyword*-Aufruf umgebaut.

```
1 '''Standardverwendung von Parametern'''
2 Click Left( @PetClinic.PetClinicWeb.Login.Go )
3
4 '''Spezielle Verwendung eines location Parameters'''
5 @PetClinic.PetClinicWeb.Login.Go :: Click Left
```

Programm 4.10: Verwendung des Datentyps *location*

```
1 keyword For Keyword Beispiel {
2   For (i, 0, 2) {
3     Print("Hello - " + i)
4   }
5 }
6
7 keyword For {
8   parameter in var as variable
9   parameter in from as number
10  parameter in to as number
11
12  implemented in java -> "com.github.thomasfischl.rayden.runtime.
      keywords.impl.ForKeyword"
13 }
```

Programm 4.11: Verwendung des Datentyps *variable*

Der letzte Datentyp ist *variable*. Dieser Datentyp wird verwendet, wenn man den Namen einer Variable an ein *Keyword* übergeben will. Dieser Datentyp beeinflusst die Auswertung von Ausdrücken. Wird ein Ausdruck mit dem Datentyp *variable* typisiert, werden alle Verwendungen von Variablen in diesem Ausdruck nicht ausgewertet. Ein gutes Beispiel dazu ist das *For Keyword* aus dem Codeausschnitt 4.11.

In diesem Beispiel ist der Parameter *var* als *variable* deklariert. Dadurch wird der Ausdruck *i* nicht ausgewertet, sondern als Zeichenkette der *Keyword*-Implementierung übergeben. Somit kann die Implementierung eine neue Variable mit dem Namen *i* anlegen. Würde man den Parameter *var* mit einem anderen Datentyp versehen, würde der *Expression Evaluator* versuchen, diese Variable mit einem Wert aus dem Gültigkeitsbereich zu ersetzen. Wird kein Wert für *i* gefunden, wird die Ausführung mit einem Fehler abgebrochen.

Eine Typumwandlung ist in der Sprache Rayden nicht vorgesehen. Es besteht zwar die Möglichkeit, alle Datentypen in einen *string* umzuwandeln, aber andere Kombinationen sind nicht möglich. In der Implementierung ei-

nes *Keywords* können die Werte beliebig konvertiert werden. Die Laufzeitumgebung stellt den Datentyp nur innerhalb der Gültigkeitsbereiche sicher.

4.6 Verarbeitung von *Keywords* und Ausdrücken

Im Rayden-System sind die *Runtime* und die Teilkomponenten Interpretierer, Compiler und *Expression Evaluator* für die Ausführung eines Tests zuständig. Dabei wird die Ausführung von *Keywords* und Ausdrücken voneinander getrennt. Die *Keywords* werden vom Interpretierer ausgeführt.

Die Ausdrücke werden im *Expression Evaluator* behandelt. Diese Komponente verwendet keine explizite *Stack*-Maschine, sondern einen *Post-Order*-Baumdurchlauf für die Auswertung. Dabei kann diese Komponente entweder untypisiert oder typisiert ausgeführt werden. Diese Eigenschaft zur Typisierung von Parametern wird benötigt, um die Funktionalität einiger Datentypen zu ermöglichen. Darunter fallen die Datentypen *variable* und *enumeration*. Für diese beiden Datentypen muss sich der *Expression Evaluator* entweder anders verhalten oder zusätzliche Überprüfungen durchführen.

4.7 *Library* und *Bridge*

Um mit Rayden auch große Testprojekte verwalten zu können, gibt es das Konzept von Bibliotheken (*Libraries*). Eine Bibliothek besteht aus einer Menge von *Keywords*. Es können sowohl *Scripted-*, *Scripted-Compound-* also auch *Compound-Keywords* in einer Bibliothek enthalten sein. Wobei man wahrscheinlich eher *Scripted-* und *Scripted-Compound-Keywords* in einer typischen Bibliothek finden wird.

In einer Bibliothek werden *Keywords* thematisch zusammengefasst. Es wäre beispielsweise möglich, dass es eine Standard-Bibliothek gibt, wie im Code-Beispiel 4.12 zu sehen ist. In diesem Beispiel sind *For-, If-* und *Print-Keyword*-Definitionen enthalten. Die Datei *stdlibrary* und das dazugehörige Java-Archiv bilden eine Rayden-Bibliothek.

Um eine Bibliothek verwenden zu können, muss man diese über die Direktive *import library* einbinden. Codeausschnitt 4.13 zeigt ein Beispiel. Nachdem die Bibliothek eingebunden wurde, können alle *Keywords* daraus verwendet werden. Für *Keywords* gibt es nur einen Namensraum. Falls es durch das Einbinden von Bibliotheken zu Namenskonflikten kommen sollte, wird die erste Implementierung, die gefunden wird, verwendet. In der Reihenfolge kommen zuerst die *Keywords* aus der aktuellen Datei und danach die *Keywords* aus den Bibliotheken.

```
keyword For {
  parameter in var as variable
  parameter in from as number
  parameter in to as number

  implemented in java -> "com.github.thomasfischl.rayden.runtime.
    keywords.impl.ForKeyword"
}

keyword If {
  parameter in condition as boolean

  implemented in java -> "com.github.thomasfischl.rayden.runtime.
    keywords.impl.IfKeyword"
}

keyword Print {
  parameter text
  implemented in java ->"com.github.thomasfischl.rayden.runtime.keywords
    .impl.PrintKeyword"
}
```

Programm 4.12: Bibliothek: *stdlibrary*

```
import library "stdlibrary"

keyword Library Beispiel {
  If (1 == 1) {
    Print("Condition is true")
  }
  For ("i", 0, 2) {
    Print("Hello - " + i)
  }
}
```

Programm 4.13: Verwendung der *StdLib*-Bibliothek

In Rayden wird zwischen einer *Library* und einer *Bridge* unterschieden. In dieser Ausbaustufe des Rayden-Systems ist die Unterscheidung jedoch nur semantisch.

Unter einer *Library* versteht man grundlegende Funktionen wie Schleifen, Verzweigungen und Validierungen. Im Gegensatz dazu besteht eine *Bridge* aus *Keywords*, welche spezifisch für eine Anwendungstechnologie sind. Dazu ist eine *Bridge* auch meistens mit einem Test-Treiber gekoppelt, welcher die Basisfunktionen zur Verfügung stellt. Die *Keywords* kapseln die Funktionalität aus dem Test-Treiber und stellen diese zur Verfügung. Eine *Bridge* kann

zum Beispiel das Steuern eines Browsers unterstützen und verwendet dazu Selenium.

4.8 *Object Repository*

Das *Object Repository* stellt eine Abstraktion zur Test-Anwendung her. Alle Testobjekte, welche in einem Test verwendet werden, können in einem *Object Repository* verwaltet werden. In den Tests muss nicht jedes Mal der volle Bezeichner für das Testobjekt verwendet werden, sondern nur eine Referenz darauf.

```
1 objectrepository PetClinic {
2
3   application PetClinicWeb {
4     location absolute "/browser"
5
6     page Login {
7       location "/body/div/div[text='bla']"
8
9       button Go {
10        location "/btn[text='GO']"
11      }
12
13      control<Special Button> Cancel {
14        location "/div[text='Cancel']"
15      }
16
17      textfield Username {
18        location  "/input[id='username']"
19      }
20
21      textfield Password {
22        location  "/input[id='password']"
23      }
24    }
25
26  }
27
28 }
```

Programm 4.14: *Object Repository*

Die Testobjekte werden im *Object Repository* in einem Baum verwaltet und können über diesen auch referenziert werden. Das Beispiel 4.14 zeigt ein *Object Repository* für eine Web-Anwendung, welche als Bezeichner einen *XPath*-Ausdruck verwendet. Der Vorteil davon ist, dass der Bezeichner *loca-*

tion über die Baumstruktur zusammengebaut wird. Das spart viel an Wartungsaufwand.

Typischerweise werden in einem *Object Repository* nur Testobjekte von einer Test-Anwendung zusammengefasst. Werden in einem Test mehrere Anwendungen getestet, sollten dafür unterschiedliche *Object Repositories* angelegt werden.

```
1 page Main Page {
2   list Owners (index) {
3     location  "/ul/ur[$index]"
4   }
5 }
```

Programm 4.15: Parametrisiertes Testobjekt

Es ist auch möglich, dass man ein Testobjekt in einem *Object Repository* parametrisiert. Damit können zum Beispiel Listen abgebildet werden, indem der Index als Parameter definiert wird. Ein Beispiel dazu findet man im Codeausschnitt 4.15. Im Bezeichner *location* werden keine Ausdrücke unterstützt. Die Parameter werden über eine Substituierung ersetzt und benötigen daher keinen Datentyp.

4.9 *Java-Scripting-API*

Das Rayden-System implementiert die *Java-Scripting-API*. Durch diese Implementierung kann man in jedem Java-Programm einen Rayden-Test ausführen. Somit lässt sich das Rayden-System in viele unterschiedliche Szenarien einbinden.

```
1 ScriptEngineManager manager = new ScriptEngineManager();
2 manager.registerEngineName("RaydenLangScriptEngine",
3     new RaydenScriptEngineFactory());
```

Programm 4.16: Registrierung der *ScriptEngineFactory* für Rayden

Um einen Rayden-Test in ein Java-Programm einbinden zu können, muss man zuerst die *RaydenScriptEngineFactory* registrieren. Damit gibt man dem *ScriptEngineManager* eine neue Sprache bekannt. Im Code-Beispiel 4.16 sieht man eine Möglichkeit, wie man die Rayden-Sprache über einen

```
1 ScriptEngineManager manager = new ScriptEngineManager();
2 ScriptEngine engine = manager.getEngineByName("RaydenLangScriptEngine");
3 Object result =  engine.eval(new FileReader("./test/simple-test.rlg"));
4 RaydenScriptResult resultObj = (RaydenScriptResult) result;
```

Programm 4.17: Ausführen eines Rayden-Tests

Namen registrieren kann. Es gibt auch noch andere Möglichkeiten, wie etwa das Registrieren über die Dateiendung. Für Rayden-Tests gibt es die Dateiendung *rlg*.

Über die *ScriptEngineFactory* kann der *ScriptEngineManager* eine neue Instanz einer *ScriptEngine* anlegen. Das Code-Beispiel 4.17 zeigt, wie man einen Test aus einer Datei einliest und diesen über die *ScriptEngine* ausführen lassen kann.

In diesem Kapitel wurden der Aufbau und das Design des Rayden-Systems beschrieben. Im nächsten Kapitel werden ausgewählte Teile der Rayden-Implementierung gezeigt.

Kapitel 5

Implementierung von Rayden

Dieses Kapitel beschreibt die Implementierung von ausgewählten Komponenten des Rayden-Systems.

Die Abschnitte 5.1 und 5.2 zeigen die Grammatik der Sprache Rayden und den Interpretierer für die Ausführung von *Keywords*. Die Abschnitte enthalten Codeausschnitte der Implementierung und Teile der Grammatik.

Der Abschnitt 5.3 befasst sich mit der Umsetzung und Auswertung von Ausdrücken im Rayden-System. Dazu werden in diesem Abschnitt Auszüge aus der Grammatik der Sprache Rayden und Teile der *RaydenExpressionEvaluator*-Klasse erklärt. Die Klasse *RaydenExpressionEvaluator* ist für die Auswertung der Ausdrücke zuständig.

Im Abschnitt 5.4 wird die Validierung von Rayden-Tests gezeigt. Dafür wird das Validierungssystem von xText verwendet.

Abgeschlossen wird dieses Kapitel mit dem Abschnitt 5.5, welcher die Integration des Rayden-Systems in die *Java-Scripting-API* zeigt.

5.1 Umsetzung der *Keyword*-Grammatik

Die Sprache Rayden wurde mit dem xText-Compilerbauwerkzeug umgesetzt. Die Abbildung 5.1 zeigt einen Auszug aus der Grammatik für die Sprache Rayden. Die Regel *KeywordDecl* beginnt eine Definition eines neuen *Keywords*. Am Beginn der Regel wird die Art des *Keywords* definiert. Eine Beschreibung und Auflistung der Arten ist in Abschnitt 4.4.6 enthalten. Danach folgt ein Name für das *Keyword* (*IDEXT*), auf welchen eine geöffnete geschwungene Klammer folgt.

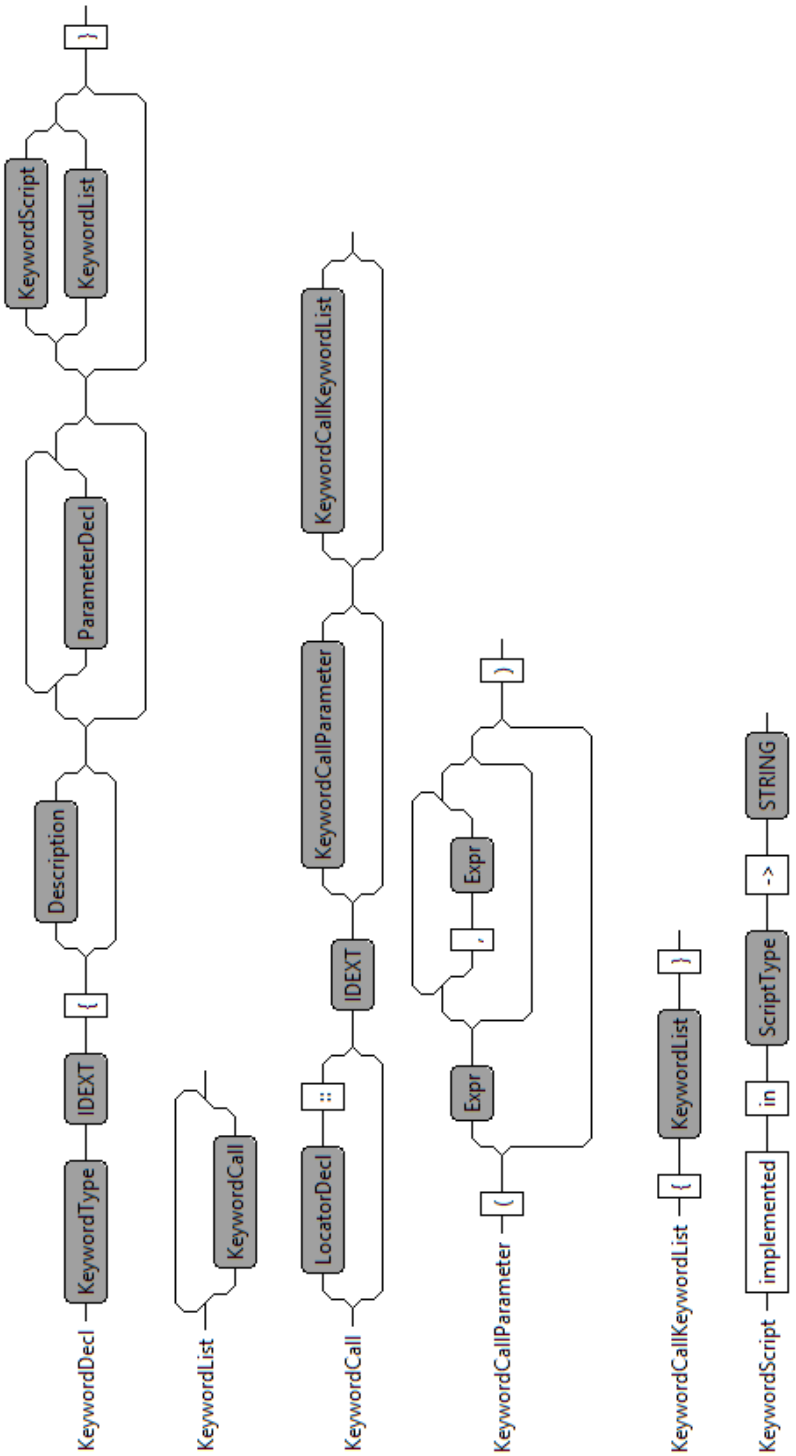

Abbildung 5.1: Auszug aus der Grammatik für *Keywords*

Die geschwungenen Klammern definieren den Bereich der *Keyword*-Implementierung. Am Anfang der Implementierung kann eine optionale Beschreibung angeführt werden. Auf diese folgt eine optionale Parameterliste. Ein Parameter wird mit der Regel *ParameterDecl* beschrieben und kann 0 bis *N* Mal wiederholt werden. Eine Parameter-Definition besteht aus dem Schlüsselwort *parameter*, einem Namen, einem optionalen Datentyp und einer Richtung.

Danach folgt entweder die Bindung an ein Codestück mit der Regel *KeywordScript* oder im Fall eines *Compound Keywords* die *Keyword*-Liste mit der Regel *KeywordList*. Die beiden Angaben sind wiederum optional, um *Keyword*-Rümpfe anlegen zu können. Diese Eigenschaft ist hilfreich, wenn die Testmanagerin oder der Testmanager nur die Struktur festlegen möchte, die Umsetzung des *Keywords* jedoch von anderem Testpersonal vorgenommen wird.

Die Regel *KeywordCall* definiert den Aufruf eines *Keywords* in einer *Keyword*-Liste. Die Regel fängt normalerweise mit dem Namen des aufzurufenden *Keywords* an. Danach folgt optional die Parameterliste für den Aufruf eines *Keywords*. Die Regel *KeywordCallParameter* definiert die Parameterliste, welche durch runde Klammern umschlossen ist. Die Parameter können als Liste von *Expr*-Regeln definiert werden und werden durch einen Beistrich separiert. Für die einfachere Verwendung und besserer Lesbarkeit enthält die Regel *KeywordCall* auch noch syntaktischen Zucker. Falls der erste Parameter eines *Keywords* vom Typ *location* ist, kann dieser Parameter vor das *Keyword* geschrieben werden. Somit lässt sich die Implementierung leichter lesen. Der Codeausschnitt 5.1 zeigt dazu die Verwendung dieses syntaktischen Zuckers im Vergleich zur klassischen Verwendung. Am Ende der *KeywordCall*-Regel ist es noch möglich, eine *Keyword*-Liste zu definieren. Diese wird benötigt, falls es sich um ein *Scripted Compound Keyword* oder um ein *Inline Keyword* handelt.

```
1  Type Text (@PetClinic.PetClinicWeb.Login.Username ,  "max.mustermann")
2  @PetClinic.PetClinicWeb.Login.Username :: Type Text ("max.mustermann")
3
4
5  Click Left( @PetClinic.PetClinicWeb.Login.Go )
6  @PetClinic.PetClinicWeb.Login.Go :: Click Left
```

Programm 5.1: Syntaktischer Zucker für die Verwendung von *location*-Datentypen

5.2 Ausführung von *Keywords* mit dem Interpretierer

Im vorigen Abschnitt 5.1 wurde die Grammatik eines *Keywords* der Sprache Rayden erklärt. Dieser Abschnitt beschäftigt sich mit der Ausführung von *Keywords*. Damit der Interpretierer arbeiten kann, benötigt dieser den Zugriff auf den abstrakten Syntaxbaum.

Das Compilerbauwerkzeug xText stellt für den abstrakten Syntaxbaum ein *Eclipse-ECore*-Modell zur Verfügung. Der generierte Compiler ist so konzipiert, dass dieser die gesamte Quelltextdatei einliest und daraus einen abstrakten Syntaxbaum erzeugt. Dieser abstrakte Syntaxbaum steht für die weitere Verarbeitung als *ECore*-Modell zur Verfügung. Einen Auszug aus dem Modell zeigt die Abbildung 5.2. Diese Abbildung zeigt die Modell-Repräsentation der Grammatik-Regeln von Abbildung 5.1. Dieser Ausschnitt aus dem Modell stellt die Basis für den Interpretierer dar.

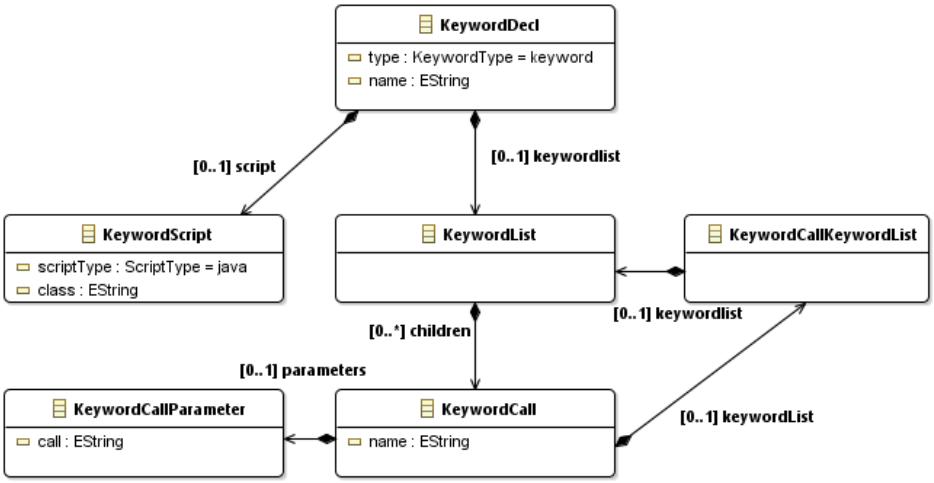

Abbildung 5.2: Ausschnitt aus dem Grammatik-Modell

Der Interpretierer für das Rayden-System ist in der Klasse *RaydenRuntime* implementiert, da der Interpretierer eine Teilkomponente der *Runtime* ist. Der Codeauszug 5.2 zeigt die essentielle Methode *executeKeyword*, welche für die Ausführung von *Keywords* verantwortlich ist. Die Methode wird mit einem *KeywordCall*-Objekt aufgerufen. Dieses Objekt bezeichnet das erste *Keyword*, welches von dem Interpretierer ausgeführt wird. Zuerst werden in der Methode übriggebliebene Elemente vom *Stack* entfernt. Danach werden alle *Reporter*-Objekte über den Start eines neuen Testfalles benachrichtigt. Im nächsten Schritt wird ein neuer Gültigkeitsbereich (*RaydenScriptScope*)

angelegt und mit dem *KeywordCall*-Objekt initialisiert. Der Gültigkeitsbereich wird dann auf den leeren *Stack* geladen.

Nach der Initialisierung des Interpretierers wird die Abarbeitung gestartet. Es werden nun solange die *Keywords* am *Stack* abgearbeitet, bis der *Stack* leer oder ein Fehler bei der Ausführung eines *Keywords* aufgetreten ist. Der Gültigkeitsbereich repräsentiert einen Aufruf eines *Keywords* und die dazugehörigen Parameter und Variablen. Der Gültigkeitsbereich speichert zusätzlich die aktuelle Position in der *Keyword*-Liste, falls es sich um ein *Compound Keyword* oder *Scripted Compound Keyword* handelt. Über die Methode *getNextKeyword* kann der Interpretierer das nächste *Keyword* aus dem aktuellen Gültigkeitsbereich lesen. Liefert die Methode keinen Wert, ist die Ausführung der *Keywords* für diesen Gültigkeitsbereich zu Ende und wird daher vom *Stack* entfernt.

Wurde jedoch ein Wert zurückgeliefert, wird mit der Ausführung fortgefahren. Handelt es sich bei dem Wert um ein *KeywordCall*-Objekt, wird die Methode *executeKeywordCall* aufgerufen. Diese Methode löst den Aufruf des *Keywords* über eine *Lookup*-Tabelle auf. Wurde die passende *Keyword*-Implementierung gefunden, wird ein neuer Gültigkeitsbereich angelegt und auf den *Stack* geladen. Wird in der *Lookup*-Tabelle keine passende Implementierung gefunden, wird die Anwendung in einen Fehlerzustand versetzt und die Ausführung abgebrochen. Handelt es sich jedoch um ein *KeywordDecl*-Objekt wird das *Keyword* ausgeführt.

Dabei muss der Interpretierer überprüfen, ob es sich um ein *Scripted Compound Keyword* handelt. Bei einem *Scripted Compound Keyword* muss eine andere Ausführung gewählt werden, da sowohl eine Code-Implementierung, als auch eine *Keyword*-Liste vorhanden sind. Bei allen anderen *Keyword*-Metatypen wird die Methode *executeKeywordDecl* ausgeführt. Diese Methode führt bei einem *Scripted Keyword* das spezifizierte Codestück aus. Bei einem *Compound Keyword* wird die *Keyword*-Liste in den Gültigkeitsbereich geladen.

Wurden alle Gültigkeitsbereiche am *Stack* erfolgreich abgearbeitet, werden am Ende noch alle *Reporter*-Objekte aufgerufen. Danach wird die Ausführung des Interpretierers beendet.

```
1 public class RaydenRuntime {
2
3   private final Stack<RaydenScriptScope> scopeStack = new Stack<>();
4
5   ...
6
7   private void executeKeyword(KeywordCall keywordCall) {
8     scopeStack.clear();
9
10    try {
11      reporter.reportTestCaseStart(keywordCall.getName());
12      scopeStack.push(new RaydenScriptScope(null,
13          Lists.newArrayList(keywordCall)));
14
15      Object currKeyword = null;
16      RaydenScriptScope currScope = null;
17      while (!scopeStack.isEmpty()) {
18
19        currScope = scopeStack.peek();
20        currKeyword = currScope.getNextKeyword();
21        if (currKeyword == null) {
22          scopeStack.pop();
23          continue;
24        }
25
26        if (currKeyword instanceof KeywordCall) {
27          KeywordCall keyword = (KeywordCall) currKeyword;
28          executeKeywordCall(keyword, currScope);
29        } //if
30
31        if (currKeyword instanceof KeywordDecl) {
32          KeywordDecl keyword = (KeywordDecl) currKeyword;
33          if (currScope.getKeywordCall().getKeywordList() != null
34            && currScope.getKeywordCall().getParameters() != null) {
35            executeScriptedCompoundKeywordDecl(keyword, currScope);
36          } else {
37            executeKeywordDecl(keyword, currScope);
38          }
39        } //if
40
41      } //while
42    } finally {
43      reporter.reportTestCaseEnd(keywordCall.getName());
44    }
45  } //executeKeyword
46
47  ...
48 }
```

Programm 5.2: Codeauszug aus der *RaydenRuntime*-Klasse

5.3 Auswertung von Ausdrücken

Dieser Abschnitt befasst sich mit den Grammatik-Regeln und der Auswertung von Ausdrücken. Die Sprache Rayden unterstützt in einigen Bereichen Ausdrücke. Ein Ausdruck wird in der Grammatik mit der Regel *Expr* beschrieben. Die Abbildung 5.3 zeigt einen Überblick über die Grammatik-Regeln von Ausdrücken.

Die Regeln für den Ausdruck sind klassisch aufgebaut. Die Operationen sind nach ihrem Vorrang in den Regeln eingearbeitet. Die am stärksten bindenden Operationen befinden sich dadurch in der Nähe der Blätter des Ausdrucksbaums. Die schwach bindenden Operationen befinden sich in der Nähe des Wurzelknotens. Die Blätter repräsentieren die Werte in einem Ausdruck, welche in der Regel *Fact* definiert werden. Die Werte können entweder Konstanten oder Variablen sein und haben einen definierten Datentypen. Die Regel *Fact* hat jedoch keine spezielle Behandlung für *Enumerations*. Der Grund dafür ist, dass *Enumerations* intern als *Strings* verarbeitet werden. Die Validierung der *Enumerations* wird nur beim Initialisieren von Gültigkeitsbereichen durchgeführt.

Für die Auswertung von Ausdrücken ist im Rayden-System die Klasse *RaydenExpressionEvaluator* zuständig. Der Codeausschnitt 5.3 gibt einen Überblick über die Klasse *RaydenExpressionEvaluator*. Ein Objekt dieser Klasse wird mit einem Gültigkeitsbereich initialisiert. Der Gültigkeitsbereich wird benötigt, um Variablen bei der Abarbeitung auswerten zu können.

Die Auswertung eines Ausdrucks wird mit der Methode *eval(Expr expression, String resultType)* gestartet. Als Parameter für diese Methode werden ein *Expr*-Objekt und eine Zeichenkette übergeben. Das *Expr*-Objekt ist im *ECore*-Modell das Wurzelobjekt für einen Ausdruck. Mit dem zweiten Parameter *resultType* kann man die Auswertung des Ausdrucks typisieren. Als Wert für den Parameter *resultType* werden die Namen der Datentypen verwendet, welche als Konstanten in dieser Klasse vorhanden sind. Wurde ein Typ definiert, wird am Ende der Auswertung noch überprüft, ob das Ergebnis dem geforderten Typ entspricht. Passt der Wert nicht zum Datentyp, wird eine Ausnahme geworfen. Es gibt auch noch einen Spezialfall bei der Typisierung: Wird ein Ausdruck mit dem Typ *variable* parametrisiert, werden keine Variablen im Ausdruck ausgewertet. Diese Eigenschaft wird benötigt, um Variablennamen an ein *Keyword* übergeben zu können.

Der Codeausschnitt 5.3 enthält am Ende die Implementierung der *eval*-Methode für ein Objekt vom Typ *Fact*. Diese Methode zeigt, wie die Werte aus dem *ECore*-Modell nach Java konvertiert werden. Die Methode zeigt auch, wie Variablen mithilfe des Gültigkeitsbereichs ausgewertet werden können.

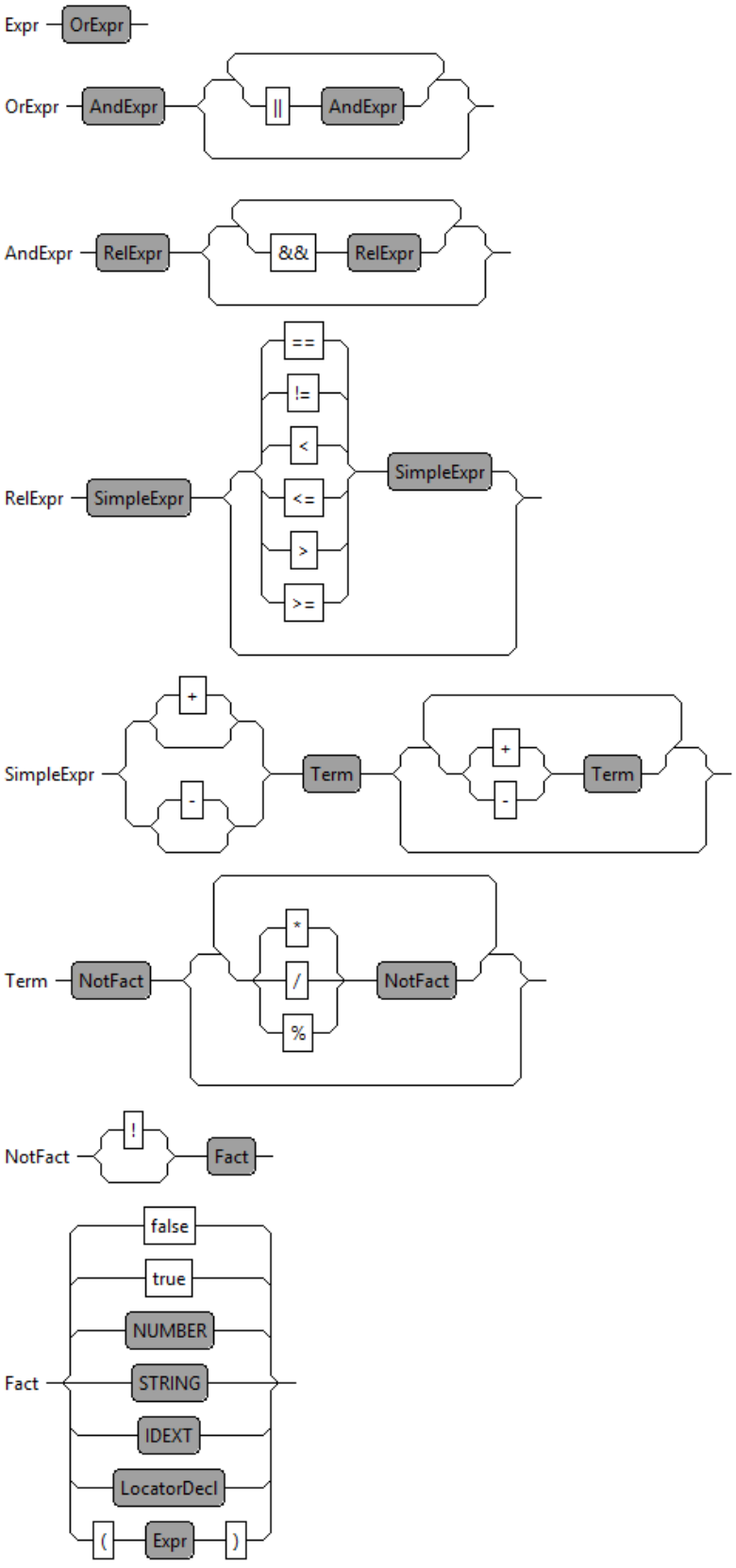

Abbildung 5.3: Grammatik-Regeln für Ausdrücke

```
1 public class RaydenExpressionEvaluator {
2
3   private static final String RESULT_TYPE_STRING = "string";
4   private String resultType;
5   ...
6
7   public RaydenExpressionEvaluator(RaydenScriptScope scope) {
8     this.scope = scope;
9   }
10
11  public Object eval(Expr expression, String resultType) {
12    ...
13  }
14
15  private Object eval(OrExpr expr) {
16    ...
17  }
18
19  private Object eval(AndExpr expr) {
20    ...
21  }
22
23  private Object eval(RelExpr expr) {
24    ...
25  }
26
27  private Object eval(SimpleExpr expr) {
28    ...
29  }
30
31  private Object eval(Term expr) {
32    ...
33  }
34
35  private Object eval(NotFact expr) {
36    ...
37  }
38
39  private Object eval(Fact expr) {
40    if (expr.getBool() != null) {
41      return "true".equals(expr.getBool());
42    } else if (expr.getString() != null) {
43      return expr.getString();
44    } else if (expr.getIdent() != null) {
45      if (RESULT_TYPE_VARIABLE.equals(resultType)) {
46        return expr.getIdent();
47      }
48      return scope.getVariable(expr.getIdent());
49    } else if (expr.getExpr() != null) {
50      return eval(expr.getExpr(), resultType); //recursion
51    } else if (expr.getLocator() != null) {
52      return evalLocator(expr.getLocator());
53    } else {
54      return expr.getNumber();
55    }
56  } //eval
57 } //RaydenExpressionEvaluator
```

Programm 5.3: Codeauszug aus dem *RaydenExpressionEvaluator*

5.4 Validierung eines Rayden-Tests

Um die Entwicklung und Wartung von Rayden-Tests zu unterstützen, wurden neben einer syntaktischen Validierung von Tests zusätzliche Validierungen hinzugefügt. Für die Umsetzung der Validierungen wurde eine Schnittstelle des xText-Frameworks verwendet. Nachdem eine Datei erfolgreich durch den xText-Compiler geladen wurde, können zusätzliche Validierungen vorgenommen werden. Der Vorteil bei diesem Vorgehen ist, dass in dieser Phase bereits das gesamte *ECore*-Modell zur Verfügung steht. Die Validierungen können somit für Überprüfungen auf das gesamte Modell zugreifen. In dieser Phase ist auch schon sichergestellt, dass es keine syntaktischen Fehler gibt, da diese Fehler bereits vom Syntaxanalysator gefunden wurden.

```
1 public class RaydenDSLJavaValidator extends
2   AbstractRaydenDSLJavaValidator {
3
4   public static final String KEYWORD_NOT_EXISTS = "KEYWORD_NOT_EXISTS";
5
6   @Check
7   public void checkKeywordCallExists(KeywordCall keyword) {
8
9     // check if this instance is an inline keyword
10    if (RaydenModelUtils.isInlineKeyword(keyword)) {
11      return;
12    }
13
14    List<KeywordDecl> keywords = RaydenModelUtils.getAllKeywords(
15      keyword);
16    boolean keywordExists = false;
17    for (KeywordDecl keywordDecl : keywords) {
18      String name1 = RaydenModelUtils.normalizeKeyword(
19        keyword.getName());
20      String name2 = RaydenModelUtils.normalizeKeyword(
21        keywordDecl.getName());
22      if (name1.equals(name2)) {
23        keywordExists = true;
24      }
25    }
26
27    if (!keywordExists) {
28      warning("Keyword does not exist",
29        RaydenDSLPackage.Literals.KEYWORD_CALL__NAME, //generated code
30        KEYWORD_NOT_EXISTS);
31    }
32  }
33
34 } //RaydenDSLJavaValidator
```

Programm 5.4: Codeauszug aus dem *RaydenDSLJavaValidator*

Um Validierungen implementieren zu können, wird von xText eine Klasse generiert, welche mit *JavaValidator* endet. Im Fall von Rayden heißt diese Klasse *RaydenDSLJavaValidator*. In dieser Klasse können nun sprachspezifische Validierungen hinzugefügt werden. Jede Methode in dieser Klasse, welche mit einer *@Check*-Annotation gekennzeichnet ist, wird zur Validierung ausgeführt.

Das Codebeispiel 5.4 zeigt die Validierung für die Verwendung von *Keywords*. Diese Validierung wird für alle *KeywordCall*-Modellelemente aufgerufen. Ein *KeywordCall* stellt einen Aufruf eines *Keywords* dar. Die Validierung überprüft, ob für jeden Aufruf eines *Keywords* auch eine Implementierung vorhanden ist. Das Traversieren des Modells und Suchen aller Modellelemente entfällt, da diese Aufgabe vom xText-Framework bei jedem neuen Aufruf des Compilers durchgeführt wird.

Im ersten Schritt überprüft die Validierung aus dem Codebeispiel 5.4, ob es sich um ein *Inline Keyword* handelt:

- Falls das *KeywordCall*-Modellelement ein *Inline Keyword* repräsentiert, kann die Validierung beendet werden, da ein *Inline Keyword* direkt in einem *KeywordCall*-Element implementiert ist.

- Falls es sich nicht um ein *Inline Keyword* handelt, wird im nächsten Schritt in den *Lookup*-Tabellen gesucht, ob eine Implementierung für das *Keyword* vorhanden ist. Wird keine Implementierung gefunden, wird eine Warnung ausgegeben.

Das Fehlen einer Implementierung liefert nur eine Warnung. Würde diese Validierung einen Fehler liefern, würde der Compiler aufgrund dieses Fehlers abbrechen und es könnten keine Tests ausgeführt werden, obwohl diese nicht von dem Fehler betroffen wären. Diese Warnung soll vielmehr eine Unterstützung für die Testerinnen und Tester sein, um fehlende *Keyword*-Implementierungen zu finden.

5.5 Integration von Rayden in die *Java-Scripting-API*

Das Rayden-System bietet eine Integration in die *Java-Scripting-API*. Die *Java-Scripting-API* ist eine standardisierte Schnittstelle für das Ausführen von Skriptsprachen in Java. Über diese Schnittstelle kann man direkt in einem Java-Programm ein Skript in einer beliebigen Sprache ausführen. Die

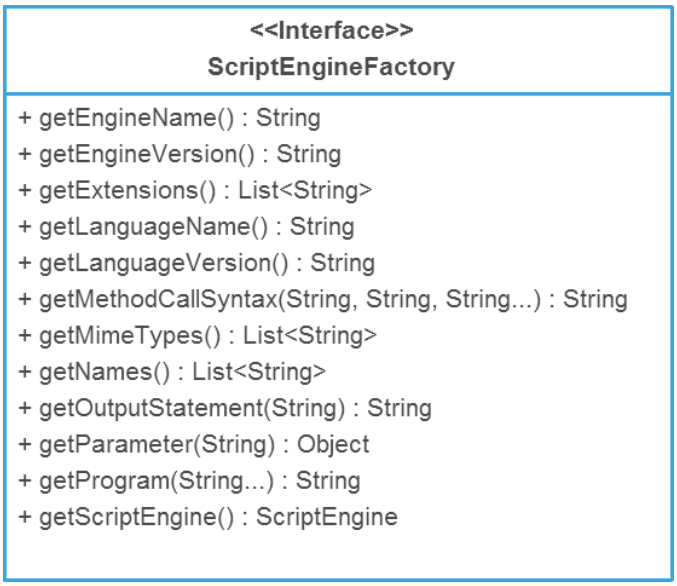

Abbildung 5.4: *ScriptEngineFactory* UML-Klassendiagramm

einzige Einschränkung dabei ist, dass für die Skriptsprache eine *ScriptEngineFactory* registriert werden muss. Die *Java-Scripting-API* ist vergleichbar mit der *Dynamic Language Runtime* [Mic15] in *Microsoft .Net*. Mit der *Dynamic Language Runtime* ist es zum Beispiel in *C#* möglich, ein Python-Skript [Fou15] auszuführen.

Für die Integration einer neuen Skriptsprache in die *Java-Scripting-API*, muss man die Schnittstellen *javax.script.ScriptEngineFactory* und *javax.script.ScriptEngine* implementieren. Diese beiden Schnittstellen bilden das Bindeglied zwischen der Java- und der Skriptsprachen-Welt. Die *ScriptEngineFactory*-Schnittstelle liefert Metadaten zu einer Skriptsprache, wie den Namen oder die Versionsnummer. Einen Überblick über die Schnittstelle gibt die Abbildung 5.4. Die wichtigste Methode der Schnittstelle ist aber *getScriptEngine()*. Diese Methode liefert ein *Script-Engine*-Objekt.

Über ein *Script-Engine*-Objekt können Skripts ausgeführt werden. Für die Ausführung stehen unterschiedliche Ausprägungen der Methode *eval()* zur Verfügung. Der Skriptcode kann entweder als Zeichenkette oder als *Reader* übergeben werden. Mit einem *Reader* kann man ein Skript direkt aus einer Datei einlesen. Der Codeauszug 5.5 zeigt die Hauptimplementierung der *eval()*-Methode aus der *RaydenScriptEngine*-Klasse. Zu Beginn der Methode wird die Rayden-*Runtime* instanziiert und initialisiert. Die *Runtime* wird benötigt, um einen Rayden-Test ausführen zu können. Im nächsten Schritt wird überprüft, ob ein spezieller *Reporter* verwendet werden soll.

```
1 public class RaydenScriptEngine extends AbstractScriptEngine {
2
3   ...
4
5   @Override
6   public Object eval(Reader reader, ScriptContext context)
7     throws ScriptException {
8
9     RaydenRuntime runtime = RaydenRuntime.createRuntime();
10    if (reporter != null) {
11      runtime.setReporter(reporter);
12    }
13
14    if (context.getAttribute(WORKING_FOLDER, ScriptContext.ENGINE_SCOPE)
15        != null) {
16      runtime.setWorkingFolder(new File(String.valueOf(context.
17        getAttribute(WORKING_FOLDER, ScriptContext.ENGINE_SCOPE))));
18    }
19
20    runtime.loadRaydenFile(reader);
21    RaydenScriptResult result = runtime.executeAllTestSuites();
22    getContext().setAttribute(TEST_RESULT, result,
23      ScriptContext.ENGINE_SCOPE);
24    return result;
25  } //eval
26
27  ...
28 } //RaydenScriptEngine
```

Programm 5.5: Codeauszug aus der *RaydenScriptEngine*

Standardmäßig wird die *RaydenXMLReporter*-Implementierung verwendet, welche die gesamten Ausgaben einer Test-Ausführung in einer XML-Datei protokolliert.

Damit man in einem Rayden-Test eine Bibliothek verwenden kann, ist es für die Rayden-*Runtime* entscheidend, dass spezifiziert ist, wo die Bibliotheken gefunden werden. Dafür wird ein Arbeitsbereich (*Working Folder*) definiert, in welchem Bibliotheken und andere externe Ressourcen gesucht werden. Der Arbeitsbereich ist standardmäßig das Ausführungsverzeichnis der Java-Anwendung. Der Wert kann aber über einen Kontextparameter verändert werden. Kontextparameter werden verwendet, um Parameter an die *Script Engine* übergeben zu können.

Nachdem alle Einstellungen für die Rayden-*Runtime* vorgenommen wurden, wird das Skript mit allen Abhängigkeiten geladen. Dazu wird das Skript mit der Methode *loadRaydenFile()* geladen. Die Implementierung der Methode zeigt der Codeauszug 5.6. Wie im Codeauszug dargestellt, gibt es zwei Me-

```
1 public class RaydenRuntime {
2   ...
3   public void loadRaydenFile(Reader reader) {
4     loadFile(reader, definedKeywords);
5   }
6   public void loadLibraryFile(Reader reader) {
7     loadFile(reader, definedImportedKeywords);
8   }
9
10  private void loadFile(Reader reader,
11    Map<String, KeywordDecl> keywordStore) throws ParseException {
12    IParseResult result = parser.parse(reader);
13    if (result.hasSyntaxErrors()) {
14      for (INode error : result.getSyntaxErrors()) {
15        reporter.error(error.getSyntaxErrorMessage().toString());
16      }
17      throw new ParseException("Provided input contains syntax errors");
18    } //if
19
20    if (result.getRootASTElement() instanceof Model) {
21      reporter.log("Model loaded successfully.");
22      Model model = (Model) result.getRootASTElement();
23
24      EList<KeywordDecl> keywords = model.getKeywords();
25      reporter.log("Loading " + keywords.size() + " keywords ...");
26      for (KeywordDecl keyword : keywords) {
27        keywordStore.put(RaydenModelUtils.normalizeKeyword(
28          keyword.getName()), keyword);
29      } //for
30
31      EList<ImportDecl> imports = model.getImports();
32      for (ImportDecl importDecl : imports) {
33        try {
34          loadLibraryFile(new FileReader(
35            new File(workingFolder, importDecl.getImportLibrary())));
36        } catch (Exception e) {
37          reporter.error("Error during loading library '" +
38            importDecl.getImportLibrary() + "'");
39        }
40      } //for
41    } //if
42  } //loadFile
43  ...
44 } //RaydenRuntime
```

Programm 5.6: Laden von Rayden-Dateien

thoden für das Laden von Rayden-Dateien. Eine Rayden-Datei wird durch die Dateiendung *rlg* identifiziert. Die erste Methode *loadRaydenFile()* ist für das Laden der primären Rayden-Datei zuständig. Mit der zweiten Methode

loadLibraryFile() werden Bibliotheken geladen. Der Hauptgrund für die zwei unterschiedlichen Methoden ist der, dass zwei *Lookup*-Tabellen für *Keywords* vorhanden sind. Es werden alle *Keywords* aus der primären Rayden-Datei in eine spezielle *Lookup*-Tabelle geladen, sodass nur *Keywords* aus dieser Datei ausgeführt werden können.

Zum Laden der Rayden-Datei wird der generierte xText-*Parser* verwendet. Im nächsten Schritt wird das Ergebnis des *Parsers* abgefragt und auf Fehler überprüft. Bei einem Fehler wird die Ausführung sofort beendet. Konnte die Datei ohne Fehler geladen werden, liefert der *Parser* eine Instanz des *ECore*-Modells. Das Modell wird durchlaufen und alle *Keyword*-Namen, die gefunden werden, in der *Lookup*-Tabelle gespeichert. Im letzten Schritt werden noch alle Bibliotheken geladen. Das Laden einer Bibliothek funktioniert identisch wie das Laden der primären Rayden-Datei, nur werden in diesem Fall die *Keywords* in eine andere *Lookup*-Tabelle gespeichert.

Nachdem das Skript und alle externen Ressourcen erfolgreich geladen wurden, können die Rayden-Tests ausgeführt werden. Dazu stellt die Rayden-*Runtime* die Methode *executeAllTestSuites()* zur Verfügung. Die Methode sucht in der primären *Lookup*-Tabelle nach *Keywords* mit der *Keyword*-Art *testcase*.

Die Sprache Rayden unterstützt folgende Arten:

- Test-Suite (*TestSuite*),
- Testfall (*TestCase*),
- Komponententest (*UnitTest*),
- Integrationstest (*IntegrationTest*),
- Schnittstellentest (*APITest*),
- automatisierter Abnahmetest (*AUTest*) und
- manueller Abnahmetest (*MAUTest*).

Alle gefunden Tests werden im nächsten Schritt sequenziell ausgeführt. Als Resultat der Skriptausführung wird das kumulierte Ergebnis der Tests geliefert. Das Ergebnis kann dann entweder in der Java-Anwendung, welche die Tests gestartet hat, oder nachträglich über die XML-Datei ausgewertet werden.

In diesem Kapitel wurde gezeigt, wie einige essentielle Komponenten des Rayden-Systems umgesetzt wurden. Im nächsten Kapitel wird erläutert, wie man Tests mit dem Rayden-System schreiben und ausführen kann. Dazu werden unterschiedliche Testmethoden verwendet, um die Stärken von Rayden zu demonstrieren.

Kapitel 6

Umsetzung eines Testprojekts mit Rayden

In diesem Kapitel wird ein Testprojekt mit dem Rayden-System umgesetzt. Dabei wird gezeigt, wie man unterschiedliche Testmethoden mit dem Rayden-System verwenden kann. Um sinnvolle Tests schreiben zu können, wird eine Beispielanwendung benötigt, welche in Abschnitt 6.1 kurz beschrieben wird. In den nächsten drei Abschnitten 6.2, 6.3 und 6.4 wird die Umsetzung von unterschiedlichen Testmethoden mit dem Rayden-System gezeigt. Ein besonderes Augenmerk wird dabei auf die Abnahmetests gelegt.

6.1 Beispielanwendung *PetClinic*

Für die Evaluierung des Rayden-Systems wird eine Anwendung zum Testen benötigt. Bei der Anwendung sollte es sich um eine Web-Anwendung handeln, um die Unterstützung von Selenium zeigen zu können. Für die Evaluierung von Rayden wird die *PetClinic*-Web-Anwendung verwendet, welche eine Beispielanwendung des Spring-Projekts ist. Die Abbildung 6.1 zeigt die Startseite dieser Web-Anwendung. Die *PetClinic* mit allen Ressourcen ist öffentlich auf Github unter der Adresse *https://github.com/spring-projects/spring-petclinic/* zugänglich.

Bei der *PetClinic*-Anwendung handelt es sich um eine Verwaltungssoftware für eine Tierklinik. Mit *PetClinic* können Besuche bei einer Tierärztin oder einem Tierarzt protokolliert werden. Dazu gehört die Erfassung der Tierbesitzerinnen und -besitzer mit ihren Haustieren. Zu jedem Haustier werden alle Arztbesuche gespeichert, damit der Krankheitsverlauf dokumentiert ist. Neben den Besitzerinnen und Besitzern mit ihren Tieren werden zusätzlich Tierärztinnen und -ärzte verwaltet.

Da der Funktionsumfang von *PetClinic* überschaubar ist, eignet sich diese Anwendung ausgezeichnet als Beispielanwendung für die Evaluierung des

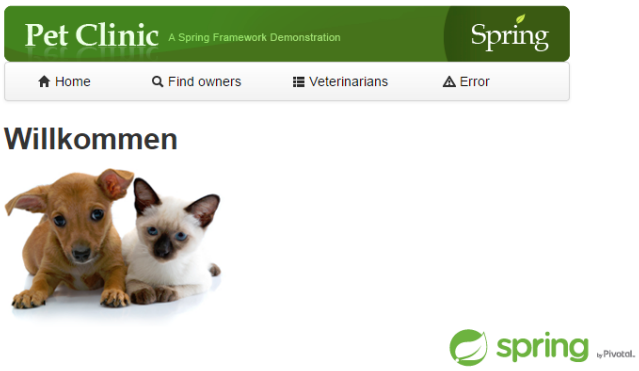

Abbildung 6.1: Startseite der Web-Anwendung *PetClinic*

Rayden-Systems. In den folgenden Abschnitten 6.2, 6.3 und 6.4 werden Tests für die *PetClinic*-Anwendung vorgestellt. Diese Tests wurden mit drei unterschiedlichen Testmethoden umgesetzt, um zu zeigen, wie man die Testmethoden mit dem Rayden-System vereinen kann.

6.2 Komponententest

Dieser Abschnitt zeigt die Umsetzung eines Komponententests mit dem Rayden-System. Für einen Komponententest wird in Rayden zuerst ein *Keyword* angelegt. Der Codeauszug 6.1 zeigt die Definition dieses *Keywords*. Ein Komponententest wird normalerweise als *Scripted Keyword* umgesetzt und mit der *Keyword*-Art *unittest* gekennzeichnet. In diesem Beispiel wird die Komponente *PetTypeFormatter* getestet. Diese Komponente der *PetClinic* ist dafür verantwortlich, für eine Zeichenkette das dazugehörige Domänenobjekt zu liefern und umgekehrt.

```
1 unittest Test PetTypeFormatter {
2   ''' This unittest verifies the functionality of the
3       formatter class PetTypeFormatter '''
4   implemented in java -> "petclinic.TestPetTypeFormatterKeyword"
5 }
```

Programm 6.1: Komponententest *Test PetTypeFormatter*

Der Codeausschnitt 6.2 zeigt die Implementierung des *Scripted Keywords*. Die Implementierung des Komponententests ist grundlegend gleich mit einem *JUnit*-Test. Die wesentlichen Unterschiede sind, dass die Methoden

nicht mit *@Test* annotiert werden und dass es nur eine Testmethode pro Klasse geben kann.

```
1 public class TestPetTypeFormatterKeyword implements ScriptedKeyword {
2
3   @Override
4   public KeywordResult execute(String keyword, KeywordScope scope,
5     RaydenReporter reporter) throws  AssertionError {
6     ClinicService service = new MockClinicService();
7     PetTypeFormatter formatter = new PetTypeFormatter(service);
8
9     try {
10     Assert.assertEquals("dog", formatter.parse("dog", null).getName());
11     Assert.assertEquals("cat", formatter.parse("cat", null).getName());
12     Assert.assertEquals("fish",formatter.parse("fish",null).getName());
13    } catch (ParseException e) {
14      throw new AssertionError(e);
15    }
16    try {
17      formatter.parse("hamster", null);
18      Assert.fail("No ParseExeption was thrown!");
19    } catch (ParseException e) {
20    }
21    try {
22      formatter.parse(null, null);
23      Assert.fail("No ParseExeption was thrown!");
24    } catch (ParseException e) {
25    }
26
27    return new KeywordResult(true);
28  } //execute
29
30 } //TestPetTypeFormatterKeyword
```

Programm 6.2: Implementierung des *Test PetTypeFormatter Keywords*

Eine nützliche Erweiterung des Rayden-Systems in der Zukunft wäre zum Beispiel eine bessere Integration mit *Unittest-Frameworks* wie *JUnit* oder *TestNG*.

6.3 Schnittstellentest

Bei einem Schnittstellentest werden öffentliche Schnittstellen, wie eine *RESTful*-Schnittstelle [Wik15b], getestet. Der Codeausschnitt 6.3 zeigt einen Test, welcher alle Tierärztinnen und -ärzte über eine *RESTful*-Schnittstelle abfragt. Schnittstellentests können entweder als *Compound Keywords* oder als *Scripted Keywords* definiert werden. Es hängt ganz davon

ab, ob die Implementierung des *Scripted Keywords* in einem anderen Test wieder verwendet werden kann.

```
1 apitest Test Veterinarians RESTful Service {
2   '''This keyword checks the restful service for veterinarians.'''
3
4   Verify JSON("http://localhost:9966/petclinic/vets.json",
5               "./demodata/vets.json")
6 }
7
8 keyword Verify JSON {
9   '''The keyword downloads the content from the given url. A second
10     content is loaded from the file. The both contents are parsed
11     into a JSON object tree. If the two trees are equal, the
12     keyword finishes successfully'''
13
14  parameter url
15  parameter file
16
17  implemented in java -> "petclinic.VerifyJsonKeyword"
18 }
```

Programm 6.3: Integrationstest *Test Veterinarians RESTful Service*

Bei diesem Beispiel liefert die Schnittstelle das Ergebnis als *JSON*-Text (*JavaScript Object Notation*). Bei *JSON* handelt es sich um ein Datenformat, bei dem die Daten in Attribute-Wert-Paaren angeordnet werden. Um zu überprüfen, ob die Schnittstelle korrekt funktioniert, wird dieser Text mit einem Text aus einer Demodaten-Datei verglichen. Damit der Test erfolgreich durchläuft, müssen die beiden Texte semantisch gleich sein. Zwei *JSON*-Texte sind semantisch gleich, wenn die beiden Objektgraphen gleich sind. Die serialisierte Form ist dafür aber nicht entscheidend.

Wenn mehrere Schnittstellen dieser Art getestet werden, ist es sinnvoll, dass man die Funktionalität zum Abfragen und Vergleichen der Daten in ein separates *Keyword* kapselt. Dadurch können andere Tests dieses *Keyword* wiederverwenden. Das Codebeispiel 6.4 zeigt die Implementierung des *Verify Json Keywords*. Am Beginn des Codestücks wird zuerst über einen *HttpClient* der *JSON*-Text von einem Server abgefragt. Danach wird der *JSON*-Text mit einem *Parser* in einen Objektgraph transformiert. Dafür wird eine *Parser*-Implementierung aus der *Google-Guava*-Bibliothek [Goo15] verwendet. Derselbe Prozess wird auch mit der Demodaten-Datei durchlaufen. Am Ende gibt es zwei Objektgraphen für die *JSON*-Texte. Für den semantischen Vergleich der beiden Graphen kann die Methode *equals()* der Klasse *JsonElement* verwendet werden. Diese Klasse stammt wiederum aus der *Google-Guava*-Bibliothek.

```
1 public class VerifyJsonKeyword implements ScriptedKeyword {
2
3   @Override
4   public KeywordResult execute(String keyword, KeywordScope scope,
5     RaydenReporter reporter) throws RuntimeException {
6     String url = scope.getVariableAsString("url");
7     String file = scope.getVariableAsString("file");
8
9     try {
10      CloseableHttpClient client = HttpClientBuilder.create().build();
11      CloseableHttpResponse response = client.execute(new HttpGet(url));
12      if (response.getStatusLine().getStatusCode() != 200) {
13        return new KeywordResult(false);
14      }
15      String json = IOUtils.toString(response.getEntity().getContent());
16
17      JsonParser parser = new JsonParser();
18      JsonElement o1 = parser.parse(json);
19      JsonElement o2 = parser.parse(IOUtils.toString(
20        new FileInputStream(file)));
21
22      return new KeywordResult(o1.equals(o2));
23    } catch (IOException e) {
24      throw new RuntimeException(e);
25    }
26  } //execute
27
28 } //VerifyJsonKeyword
```

Programm 6.4: Implementierung des *Verify Json Keywords*

6.4 Abnahmetests

In diesem Abschnitt wird die letzte Testmethode für die Evaluierung erläutert. Dabei handelt es sich um den Abnahmetest, die wohl wichtigste Testmethode für das Rayden-System. Ein Großteil des Rayden-Systems ist primär für die Unterstützung dieser Testmethode entwickelt worden. Aus diesem Grund enthält dieser Abschnitt auch zwei Umsetzungen von Testmethoden mit Rayden. Als erstes wird der Testfall *Find a pet owner and check the details* 6.4.1 umgesetzt. Bei diesem Testfall wird die Suche der *PetClinic*-Anwendung getestet. Im zweiten Testfall 6.4.2 wird das Anlegen einer neuen Tierbesitzerin oder eines neuen Tierbesitzers gezeigt.

Für das Steuern der Browser wurde für beide Abnahmetests die Selenium-Bibliothek verwendet. Aus diesem Grund wird im dritten Teil 6.4.3 dieses Abschnittes die Bindung zwischen dem Rayden-System und der Selenium-Bibliothek gezeigt.

6.4.1 Abnahmetest *Find a pet owner and check the details*

Bei dem Abnahmetest im Codeausschnitt 6.5 wird die Suchfunktion der *PetClinic*-Anwendung getestet. Dafür wird der Testfall zu erst in die zwei *Compound Keywords Find a specific Pet Owner* und *Check Owner Details* aufgeteilt. Neben den beiden *Keywords* werden noch die zwei weiteren *Keywords Prepare Browser* und *Cleanup Browser* benötigt, welche für das Starten und Stoppen des Browsers zuständig sind.

```
1 uatest Find a pet owner and check the details {
2   Prepare Browser
3   Find a specific Pet Owner("Davis")
4   Check Owner Details ("Betty Davis", "638 Cardinal Ave.",
5                        "Sun Prairie", "6085551749")
6   Cleanup Browser
7 }
8
9 keyword Find a specific Pet Owner {
10  parameter in petOwner as string
11
12  @PetClinicWeb.Find Owners :: Click
13  @PetClinicWeb.Find Owners Page.Title :: Verify Text("Find Owners")
14  @PetClinicWeb.Find Owners Page.Find :: Click
15  @PetClinicWeb.Find Owners Result Page.Title :: Verify Text("Owners")
16  @PetClinicWeb.Find Owners Result Page.Result:: Count
17  Verify(itemCount, 10)
18  @PetClinicWeb.Find Owners Result Page.Search:: Type Text (petOwner)
19  @PetClinicWeb.Find Owners Result Page.Result :: Count
20  Verify(itemCount, 2)
21  @PetClinicWeb.Find Owners Result Page.Result.Item :: Click
22 }
23
24 keyword Check Owner Details  {
25  parameter name as string
26  parameter address as string
27  parameter city as string
28  parameter telephone as string
29
30  @PetClinicWeb.Owner Detail Page.Title :: Verify Text("Owner
      Information")
31  @PetClinicWeb.Owner Detail Page.Name :: Verify Text(name)
32  @PetClinicWeb.Owner Detail Page.Address :: Verify Text(address)
33  @PetClinicWeb.Owner Detail Page.City :: Verify Text(city)
34  @PetClinicWeb.Owner Detail Page.Telephone :: Verify Text(telephone)
35 }
```

Programm 6.5: Abnahmetest *Find a pet owner and check the details*

Das *Keyword Find a specific Pet Owner* führt die Suche nach einer Tierbesitzerin mit dem Namen *Davis* aus. Dafür navigiert das *Keyword* auf die

```
objectrepository PetClinic {

  application PetClinicWeb {
    location absolute "//body"

    button Find Owners {
      location "//ul[contains(@class, 'nav')]/li[2]/a"
    }

    ...

    page Find Owners Result Page {
      control Title { location "//h2" }

      textfield Search {
        location "//div[@id='owners_filter']/label/input"
      }

      list Result {
        location absolute "//*[@id='owners']"
        control Item { location "/tbody/tr[1]//a" }
      }
    }

    page Owner Detail Page {
      location "/div/table[1]"

      control Title { location absolute "//body/div/h2[1]" }
      control Name { location "/tbody/tr[1]/td" }
      control Address { location "/tbody/tr[2]/td" }
      control City { location "/tbody/tr[3]/td" }
      control Telephone { location "/tbody/tr[4]/td" }
    }

    ...
  }
}
```

Programm 6.6: Codeauszug aus dem *Object Repository* für den Testfall *Find a pet owner and check the details*

Seite für Tierbesitzerinnen und Tierbesitzer und startet die Suche. Danach werden auf der Ergebnisseite der Suche alle Besitzerinnen und Besizter aufgelistet, welche in der Anwendung vorhanden sind. Danach wird mithilfe der integrierten Suche nach dem Namen *Davis* gesucht und die Detailseite der Tierbesitzerin geöffnet. Wurde die Detailseite erfolgreich geöffnet, ist das *Keyword* fertig abgearbeitet. Die Validierung der Daten wird von dem nächsten *Keyword Check Owner Details* durchgeführt.

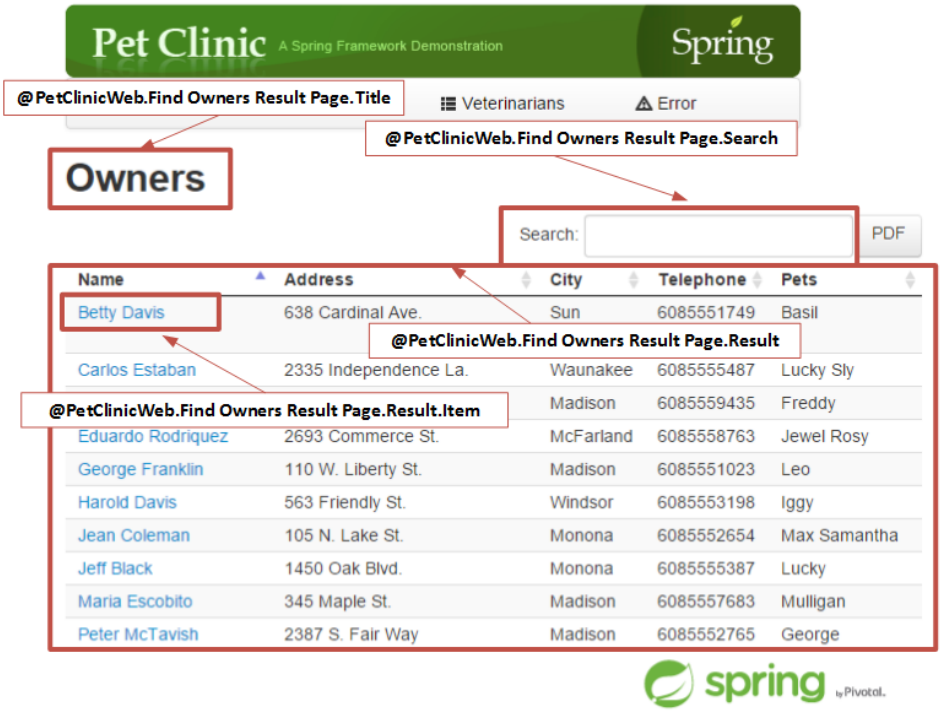

Abbildung 6.2: Darstellung der Verbindung eines Objekts aus dem *Object Repository* mit Elementen auf der Web-Seite

Dem *Keyword Check Owner Details* werden die zu überprüfenden Daten als Parameter übergeben. Das *Keyword* überprüft jedes Datum mit dem *Scripted Keyword Verify Text*. Dem *Scripted Keyword* werden zwei Parameter übergeben. Der erste Parameter ist ein *Locator*, welcher ein Element auf der Web-Seite definiert. Von diesem Element wird der Text abgefragt und mit dem zweiten Parameter verglichen. Der zweite Parameter ist eine Zeichenkette mit dem erwarteten Wert. Sind die Werte nicht gleich, wird der Test mit einem Fehler abgebrochen.

Damit in den *Keywords* keine *XPath*-Ausdrücke vorkommen müssen, wird ein *Object Repository* verwendet. Einen Auszug mit den wichtigsten Objekten für diesen Abnahmetest zeigt das Codebeispiel 6.6. Dieses Codebeispiel enthält die Objekte für die Suchergebnisseite und die Detailseite für Tierbesitzerinnen und Tierbesitzer. Die Abbildung 6.2 zeigt die Verbindung von Web-Seiten-Elementen auf Objekte im *Object Repository*. In der Grafik sind jene Elemente rot hinterlegt, welche einen Eintrag im *Object Repository* besitzen. Neben jedem Element steht auch der vollständige *Locator*, mit welchem man das Objekt aus *Object Repository* abfragen kann.

6.4.2 Abnahmetest *Add new pet owner*

Der zweite Abnahmetest testet den Anwendungsfall *Add new pet owner*. Dieser Abnahmetest wird direkt im Abnahmetest-*Keyword* spezifiziert. Zuerst wird wieder die Testumgebung mit dem *Keyword Prepare Browser* vorbereitet. Im nächsten Schritt navigiert der Test auf die Tierbesitzerinnen- und besitzerseite und betätigt die Schaltfläche *Add Owner*.

Danach werden alle benötigten Daten für eine neue Tierbesitzerin oder einen neuen Tierbesitzer im Formular eingegeben. Dazu wird das *Keyword Type Text* verwendet. Dieses *Scripted Keyword* schreibt mithilfe der Selenium-Bibliothek einen Text in ein Textfeld auf der Web-Seite. Der Codeauszug 6.7 zeigt die dazugehörigen Objekte im *Object Repository*. Der Auszug zeigt die Seite *Edit Owner Page*, welche das Formular für das Anlegen und Ändern einer Besitzerin oder eines Besitzers zeigt.

Das *Add new pet owner Keyword* ist ein gutes Beispiel, um zu zeigen, wie man mithilfe des *Object Repositories* einen leserlichen Test schreiben kann. Der Test setzt ausschließlich auf *Locators* und verzichtet auf die direkte Verwendung von *XPath*-Ausdrücken.

```
1 objectrepository PetClinic {
2
3   application PetClinicWeb {
4     location absolute "//body"
5
6     ...
7
8     page Edit Owner Page {
9       control Title { location "//h2" }
10      textfield First Name { location "//*[@id='firstName']" }
11      textfield Last Name { location "//*[@id='lastName']" }
12      textfield Address { location "//*[@id='address']" }
13      textfield City { location "//*[@id='city']" }
14      textfield Telephone{ location "//*[@id='telephone']" }
15      button Add Owner { location "//button[text() = 'Add Owner']" }
16    }
17
18    ...
19  }
20 }
```

Programm 6.7: Codeauszug aus dem *Object Repository* für den Testfall *Add new pet owner*

6.4.3 *Keywords* aus der Selenium-Bibliothek

In diesem Abschnitt werden ausgewählte Selenium-*Keywords* aus den Abnahmetests beschrieben. Diese *Keywords* wurden alle in der Datei *selenium.rlg* angelegt. Die Datei *selenium.rlg* stellt somit die *Bridge* zwischen Rayden und Selenium dar. Das Codestück 6.8 zeigt drei *Scripted Keywords*, welche einen guten Überblick in die Integration von Selenium in das Rayden-System zeigen.

```
1 keyword Open Browser  {
2   parameter in browserType as string
3   parameter in url as string
4
5   implemented in java -> "petclinic.selenium.OpenBrowserKeyword"
6 }
7
8 keyword Click {
9   parameter in locator as location
10
11   implemented in java -> "petclinic.selenium.ClickKeyword"
12 }
13
14 keyword Verify Text  {
15   parameter in locator as location
16   parameter in text as string
17
18   implemented in java -> "petclinic.selenium.VerifyTextKeyword"
19 }
```

Programm 6.8: Codeauszug aus der Selenium-*Keyword*-Bibliothek

Zuerst wird im Codeausschnitt 6.9 das *Open Browser Keyword* beschrieben. Dieses *Scripted Keyword* ist essentiell für die Verwendung von Selenium, da dieses *Keyword* die Testumgebung vorbereitet. Als Parameter werden der Browsertyp und eine *URL* übergeben. Der Browsertyp definiert den Browser, welcher gestartet werden soll. Das kann zum Beispiel der *Internet Explorer*, *Firefox* oder *Google Chrome* sein. Mit dem Browsertyp-Parameter wird ein *Webdriver*-Objekt angelegt. Dieses Objekt dient als Schnittstelle zwischen dem Test und der Browser-Instanz. Jedes neue *Webdriver*-Objekt startet einen Browser. Der zweite Parameter *URL* definiert die Startseite. Über die Methode *navigate()* der Klasse *Webdriver* kann man auf eine neue Seite im Browser navigieren.

Bei dem nächsten *Keyword* handelt es sich um das *Click Keyword*, welches im Codestück 6.10 gezeigt wird. Mit diesem *Keyword* kann man einen Klick auf der Web-Seite simulieren. Um die richtige Position für den Mauszeiger

```
1 public class OpenBrowserKeyword implements ScriptedKeyword {
2
3   @Override
4   public KeywordResult execute(String keyword, KeywordScope scope,
5     RaydenReporter reporter) {
6     String browserType = scope.getVariableAsString("browserType");
7     String url = scope.getVariableAsString("url");
8     WebDriver driver = Selenium.getInstance()
9       .initializeDriver(browserType);
10    driver.navigate().to(url);
11    return new KeywordResult(true);
12  } //execute
13
14 } //OpenBrowserKeyword
```

Programm 6.9: Implementierung des *Open Browser Keywords*

herausfinden zu können, wird dem *Keyword* ein *Locator* übergeben. Dieser *Locator* beschreibt ein Element auf der Web-Seite. Die Auswertung dieses *Locators* wird von der Methode *findElement()* übernommen. Die Methode greift auf das aktuelle *WebDriver*-Objekt zu, um das Element zu finden. Diese Aktion könnte man auch direkt über das *WebDriver*-Objekt ausführen, jedoch enthält die Methode *findElement()* aus der Selenium-Klasse eine zusätzliche Wiederholungsfunktion im Fall eines Fehlers. Die Funktionalität ist für einen stabilen Abnahmetest wichtig, da diese Synchronisierungsprobleme mit der Web-Anwendung reduziert.

```
1 public class ClickKeyword implements ScriptedKeyword {
2
3   @Override
4   public KeywordResult execute(String keyword, KeywordScope scope,
5     RaydenReporter reporter) {
6     RaydenExpressionLocator locator =
7       (RaydenExpressionLocator) scope.getVariable("locator");
8     reporter.log("Click on '" + locator + "'");
9     Selenium.getInstance().findElement(locator.getEvalLocator())
10      .click();
11    return new KeywordResult(true);
12  } //execute
13
14 } //ClickKeyword
```

Programm 6.10: Implementierung des *Click Keywords*

Man spricht von einem Synchronisierungsfehler bei einer Web-Seite, wenn eine Aktion ausgeführt wird, obwohl die Anwendung noch nicht fertig gela-

den wurde. Dieser Fehler tritt häufig bei stark dynamischen Web-Seiten auf. Die einfachste Lösung für das Problem ist die mehrmalige Auswertung des *XPath*-Ausdrucks, falls dieser kein Element findet. Genau dieser Ansatz ist auch in der *findElement()*-Methode der Selenium-Klasse implementiert.

```
1 public class VerifyTextKeyword implements ScriptedKeyword {
2
3   @Override
4   public KeywordResult execute(String keyword, KeywordScope scope,
5     RaydenReporter reporter) {
6     RaydenExpressionLocator locator =
7       (RaydenExpressionLocator) scope.getVariable("locator");
8     String text = scope.getVariableAsString("text");
9     WebElement element = Selenium.getInstance().findElement(
10      locator.getEvalLocator());
11    String elementText = element.getText();
12    reporter.log("Verify Text: '" + text + "'='" + elementText + "'");
13    return new KeywordResult(text.equals(elementText));
14  } //execute
15
16 } //VerifyTextKeyword
```

Programm 6.11: Implementierung des *Verify Text Keywords*

Bei dem *Keyword Verify Text* handelt es sich um eine Überprüfung. Mit diesem *Keyword* kann ein Text auf einer Web-Seite mit einem vordefinierten Text verglichen werden. Bei einem Fehlerfall wird das *Keyword* mit einem Fehler beendet und die Ausführung des Tests abgebrochen. Die Implementierung des *Scripted Keywords* zeigt der Codeausschnitt 6.11. Das Ergebnis des Vergleichs der beiden Texte wird an ein *KeywordResult*-Objekt übergeben.

Dieses Kapitel hat gezeigt, wie man ein Testprojekt mit Rayden umsetzen kann. Dafür wurden drei unterschiedliche Testmethoden gezeigt und erläutert, wie diese mit Rayden umgesetzt werden können. Die Testbeispiele haben auch gezeigt, welche Vorteile die Verwendung des *Object Repositories* hat. Das nächste Kapitel bietet eine Zusammenfassung dieser Masterarbeit und gibt einen Ausblick auf mögliche Erweiterungen des Rayden-Systems.

Kapitel 7

Zusammenfassung, Ausblick und Erfahrungen

7.1 Zusammenfassung

Diese Masterarbeit hat zum Ziel, die Erstellung und die Wartung von Tests im Allgemeinen und von Abnahmetests im speziellen zu vereinfachen. Ein weiteres Ziel ist es, die Zusammenarbeit von Fach-, Entwicklungs- und Testabteilungen zu erleichtern. Dafür wurden im Kapitel 2 die gängigsten Testmethoden und Technologien vorgestellt, welche in der Softwareentwicklung eingesetzt werden. In Kapitel 3 wurde der Ablauf eines typischen Testprojekts skizziert. Dabei wurde darauf eingegangen, welche Personengruppen in einem Testprojekt involviert sind und welche Aufgaben diese übernehmen.

Das Kapitel 4 beschäftigte sich mit dem Design von Rayden. Am Anfang des Kapitels wurden die Ziele von Rayden beschrieben. In den nächsten Abschnitten wurde die Sprache Rayden detailliert erklärt und die Verwendung eines *Object Repositories* gezeigt. Die Implementierung ausgewählter Komponenten wurde in Kapitel 5 gezeigt. Es wurde erläutert, wie der Interpretierer, welcher für die Ausführung von *Keywords* verantwortlich ist, implementiert wurde. Ein weiterer interessanter Aspekt der Implementierung ist die Integration des Rayden-Systems in die *Java-Scripting-API*, die auch behandelt wurde.

Im Kapitel 6 wurde das Rayden-System mithilfe einer Beispielanwendung evaluiert. Es wurden Tests mit unterschiedlichen Testmethoden für die Beispielanwendung geschrieben. Dabei wurde gezeigt, welche besonderen Vorteile Rayden bei der Umsetzung von Abnahmetests hat.

7.2 Ausblick auf weitere Arbeiten

Im Zuge der Erstellung der Masterarbeit wurden einige Bereiche identifiziert, welche zukünftig erweitert werden können:

- **Implementierung eines grafischen Editors für Rayden-Tests**
  Um die Verwendung des Rayden-Systems zu erleichtern, wäre eine visuelle Repräsentation von Tests hilfreich. Dazu müsste ein grafischer Editor für die Sprache Rayden entwickelt werden. Das Ziel dieses Editors wäre es, neue *Compound Keywords* anzulegen und bestehende zu warten. Auch eine visuelle Darstellung der Ausführung eines Tests in dem Editor würde die Handhabung erleichtern.

- ***Debugger* für Rayden**
  Da Rayden eine eigene *Runtime* besitzt, können keine bestehenden *Debugger* verwendet werden, um einen Rayden-Test zu analysieren. Darum wäre die Implementierung eines eigenen *Debuggers* für die Sprache vorteilhaft. Zusätzlich könnte man mit dieser Erweiterung einen Test Schritt für Schritt ausführen. Somit könnten Fehler schneller aufgespürt und behoben werden.

- **Integration weiterer Sprachen**
  In der derzeitigen Ausführung von Rayden können *Scripted Keywords* und *Scripted Compound Keywords* nur in Java implementiert werden. Als Erweiterung könnte man zusätzliche Sprachen unterstützen. Spezielle Skriptsprachen wie JavaScript, Python oder Ruby würden sich eignen. Der Vorteil wäre, dass die Kompilierungsphase für die *Scripted Keywords* wegfallen würde.

- **Integration mit Test-Bibliotheken**
  Bei der Umsetzung von Komponententests hat sich gezeigt, dass man bestehende Komponententests nicht so einfach wiederverwenden kann. Daher würde es helfen, wenn man nicht nur Klassen mit dem Interface *ScriptedKeyword* als Implementierung für *Scripted Keywords* verwenden könnte. Diese Erweiterung könnte dafür sorgen, dass zum Beispiel bestehenden *JUnit*-Tests als Implementierung verwendet werden könnten. Es müsste ein Adapter erstellt werden, der die Verbindung zwischen Rayden und einem Test-*Framework* wie *JUnit* herstellt. Diese Erweiterung würde die Akzeptanz von Entwicklerinnen und Entwicklern steigern, da diese weiterhin mit ihren gewohnten Werkzeugen arbeiten könnten.

7.3 Erfahrungen

Meine größte und wichtigste Erfahrung bei dieser Masterarbeit war die Erkenntnis, dass das Designen einer Sprache ein höchst komplexer Prozess ist. Das Design der Sprache Rayden war äußerst anspruchsvoll, da diese Sprache wenig Ähnlichkeiten zu bestehenden Sprachen hat. Daher gab es für das Sprachdesign etliche Iterationen bis das finale Konzept fertig war.

Das xText-Framework war das wichtigste Werkzeug bei der Erstellung der Sprache und des gesamten Systems. Ich habe schon gute Erfahrungen mit dem xText-Framework im Zuge der Bachelorarbeit gemacht. Aus diesem Grund habe ich dieses Werkzeug auch für die Masterarbeit wiederverwendet. Jedoch habe ich hier die Erfahrung gemacht, dass ich mit der Komplexität und den Besonderheiten der Sprache Rayden die Grenzen des xText-Frameworks erreicht habe.

Quellenverzeichnis

Literatur

[CB10] Larmann Craig und Vodde Bas. „Acceptance Test-Driven Development with Robot Framework“. In: (2010). URL: http://wiki.robotframework.googlecode.com/hg/publications/ATDD_with_RobotFramework.pdf (siehe S. 17).

[HG09] Jeff Hinz und Martin Gijsen. „Fifth Generation Scriptless and Advanced Test Automation Technologies“. In: (2009). URL: http://www.testars.com/docs/5GTA.pdf (siehe S. 14).

[Lau06] Pekka Laukkanen. „Data-Driven and Keyword-Driven Test Automation Frameworks“. Masterthesis. Helsinki University of Technology, Aug. 2006 (siehe S. 16).

[Mes07] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007 (siehe S. 7).

[Win+12] Mario Winter u. a. *Der Integrationstest: Von Entwurf und Architektur zur Komponenten- und Systemintegration*. Carl Hanser Verlag, 2012 (siehe S. 8).

Online-Quellen

[Bor15] Borland. 2015. URL: http://www.borland.com/silktest (besucht am 03. 05. 2015) (siehe S. 11).

[Ecl13] Eclipse. 2013. URL: http://eclipse.org/eclipse (besucht am 28. 04. 2015) (siehe S. 9).

[Ecl15a] Eclipse. 2015. URL: https://eclipse.org/org/foundation (besucht am 03. 05. 2015) (siehe S. 10).

[Ecl15b] Eclipse. 2015. URL: https://www.eclipse.org/modeling/emf (besucht am 03. 05. 2015) (siehe S. 10).

[Ecl15c] Eclipse. 2015. URL: http://eclipse.org/Xtext/ (besucht am 28. 04. 2015) (siehe S. 10, 24).

[Fou15] Python Software Foundation. 2015. URL: https://www.python.org (besucht am 24. 05. 2015) (siehe S. 54).

[Fow13] Martin Fowler. 2013. URL: http://martinfowler.com/bliki/PageObject.html (besucht am 28. 04. 2015) (siehe S. 25).

[Goo15] Google. 2015. URL: https://github.com/google/guava (besucht am 28. 05. 2015) (siehe S. 61).

[KE14] Beck Kent und Gamma Erich. 2014. URL: http://junit.org/ (besucht am 03. 05. 2015) (siehe S. 7).

[KH15] Pekka Klärck und Janne Härkönen. 2015. URL: http://robotframework.org/ (besucht am 17. 05. 2015) (siehe S. 17).

[Mic15] Microsoft. 2015. URL: https://msdn.microsoft.com/en-us/library/dd233052(v=vs.110).aspx (besucht am 24. 05. 2015) (siehe S. 54).

[Ora14] Oracle. 2014. URL: https://www.jcp.org/en/jsr/detail?id=223 (besucht am 28. 04. 2015) (siehe S. 19, 23).

[Sel15] Selenium. 2015. URL: http://www.seleniumhq.org (besucht am 03. 05. 2015) (siehe S. 10).

[W3C15] W3C. 2015. URL: http://www.w3.org/TR/2013/WD-webdriver-20130117 (besucht am 03. 05. 2015) (siehe S. 11).

[Wik15a] Wikipedia. 2015. URL: http://en.wikipedia.org/wiki/Stack_machine (besucht am 28. 04. 2015) (siehe S. 24).

[Wik15b] Wikipedia. 2015. URL: http://de.wikipedia.org/wiki/Representational_State_Transfer (besucht am 28. 05. 2015) (siehe S. 60).