Rayden

—

Ein System für funktionale Tests mit Spezialisierung auf Abnahmetests

Thomas Fischl

MASTERARBEIT

eingereicht am
Fachhochschul-Masterstudiengang

Software Engineering

in Hagenberg

im Mai 2015

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 31. Mai 2015

Thomas Fischl

Inhaltsverzeichnis

Kurzfassung

TODO !!!

Abstract

TODO !!!

Kapitel 1

Einleitung

1.1 Rayden

Das Wort *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 der Namen *Rayden* abgeleitet wurde.

1.2 Motivation

Viele Software-Firmen haben in den letzten Jahren und Jahrzehnten eine große Testabteilung aufgebaut. Der Fokus in diesen Testabteilungen liegt sehr häufig noch auf dem manuellen Testen der grafischen Oberfläche einer Software. Dabei müssen für jede neue Version einer Software viele manuelle Schritte durchlaufen werden. Dieser Vorgang ist sehr zeit- und kostenintensiv. Durch den Vormarsch von neuen Entwicklungsmethoden und einem starken Kostendruck 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, das einen noch größeren Aufwand bedeutet. In diesem Spannungsfeld überlegen viele Firmen, ihre manuellen Tests zu automatisieren um dadurch langfristig Zeit zu sparen.

Dieser Transformationsprozess stellt die Organisationen vor eine große Herausforderung. Die Firmen 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 Test-*Framework* muss in der Lage sein, auf

Basis der manuellen Tests zu arbeiten. Auf der anderen Seite darf die Lesbarkeit der manuellen Tests aber nicht verloren gehen, damit diese in Ausnahmefällen noch von einer Testerin oder einem Tester manuell durchgeführt werden kann.

1.3 Problemstellung

Viele Testabteilungen arbeiten heutzutage größtenteils mit manuellen Tests. Diese Tests sind über Jahrzehnte gewachsen und es wurden tausende von Stunden in die Erstellung und Wartung investiert. Die Testabteilungen bestehen in solchen Fällen aus vielen manuellen Testern, welche die Tests für jede neue Version einer Software 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 Entwicklung macht es notwendig, dass sich Testabteilungen immer öfter mit dem Thema der Testautomatisierung auseinandersetzen müssen.

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. Das kann der Fall sein, um fehlgeschlagene Ausführungen nachträglich manuell verifizieren zu können.

4. Die bestehenden Automatisierungslösungen sind oft sehr technisch aufgebaut. Jedoch findet man in typischen Testabteilungen nur wenige Entwickler und Techniker, 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 unterstützen.

1.4 Zielsetzung

Das Ziel dieser Arbeit ist es, die Fähigkeiten eines *Keyword-Driven-Testing*-Ansatz 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, welches die Bedürfnisse nach einer einfachen und gut lesbaren Sprache erfühlt. Zusätzlich soll die Sprache eine gute Unterstützung für das *Object-Repository* liefern.

Im nächsten Kapitel 2 werden die Probleme von einer Testabteilung und deren Anforderungen detailliert beschrieben.

Kapitel 2

Problemanalyse

Die Testabteilung wird durch die Umstellung der Softwareentwicklungsprozesse auf Agile-Entwicklungsmethoden auf ein große Probe gestellt. Durch die Verwendung einer Agilen-Entwicklungsmethode werden die Release-Zyklen von Anwendungen deutlich reduziert. Das hast zur Folge, dass die Testabteilung einen deutlich höheren Testaufwand bewerkstelligen muss. Auf der anderen Seite steht die Testabteilung unter einen immer größer werdenden Kostendruck.

Aus diesem Grund entscheiden viele Testmanagerinnen und Testmanager sich dafür, ihre manuellen Testabläufe zu automatisieren. Jedoch können die Testabteilungen bei dem Aufbau von automatisierten Tests nicht von Grund auf neu beginnen. In den manuellen Tests stecken jahrelange Entwicklungszeit und Wissen, welches für die automatisierten Tests wieder verwendet werden muss um bei der Automatisierung erfolgreich zu sein.

Ein anderes Problem von der Testabteilung ist, dass diese über keine bis wenige Entwicklerinnen und Entwickler verfügt. Eine klassische Testabteilung besteht normalerweise aus Personen, welche keine fundierten Programmierkenntnisse besitzen. Aus diesem Grund ist die Testabteilung entweder auf die Mithilfe der Entwicklungsabteilung angewiesen oder muss den Anteil an Entwicklerinnen und Entwicklern aufstocken was wiederum eine Kostensteigerung bedeutet.

Als mögliche Option für eine Testabteilung wäre eine Testmethode, welche es auch Testerinnen und Tester ohne fundierte Programmierkenntnisse automatisierte Tests zu erstellen. Zusätzlich muss es mit der Testmethode möglich sein, bestehende manuelle Tests wieder zu verwenden. Schlussendlich müssen die automatisierten Tests noch immer in einem Format vorliegen, dass diese auch manuell von einer Testerin oder einem Tester ausgeführt werden kann.

Kapitel 3

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, den diese 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 man in Abbildung 3.1 sehen kann. Die unterschiedlichen Methoden werden in diesem Kapitel genau beschrieben und es wird auf die Unterschiede eingegangen.

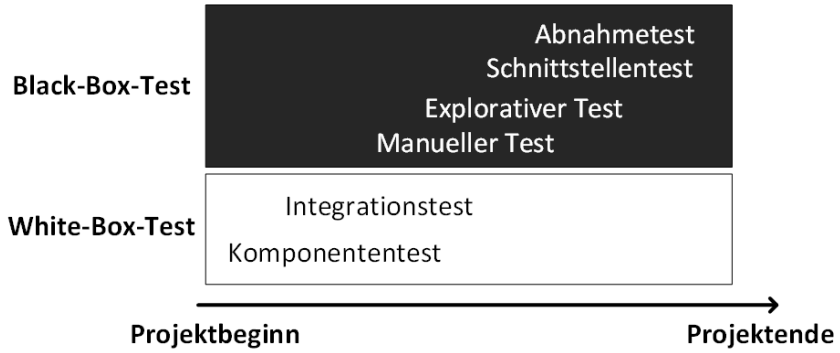


Abbildung 3.1: Testmethoden unterteilt in *White-Box-* und *Black-Box-*Testen

Im zweiten Teil dieses Kapitels werden die verwendeten Technologien beschrieben. Dazu gehören die Werkzeuge und Bibliotheken, welche für die Entwicklung der Sprache Rayden und der Ausführungseinheit verwendet werden. Weiters werden zwei Testtreiber-Bibliotheken beschrieben, welche für automatisierte Abnahmetests verwendet werden können.

3.1 *White-Box*-Test

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

Diese Tests sind typischerweise sehr technisch und verlangen vom Testpersonal Programmierkenntnisse. Daher werden diese Tests von den 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.

3.2 *Black-Box*-Test

Die Gruppe der *Black-Box*-Tests sind klassische Aufgaben eines Qualitätssicherungsteams. Diese Gruppe umfasst alle Testansätze, bei denen der Quelltext der Anwendung nicht vorliegt. Dabei ist die Anwendung eine *Black-Box*. Die Aufgabe des Qualitätssicherungsteams ist es, zu überprüfen, ob alle Anforderungen laut Spezifikation umgesetzt worden sind. Für diese Aufgabe stehen dem Qualitätssicherungsteam eine ganze Reihe an unterschiedlichen Ansätzen zur Auswahl, 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.

3.3 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 worden sind.

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.

3.3.1 Explorativer Test

Eine Speziallform 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 Personen nur eine Aufgabe, die 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 Gegenteil zu manuellen Tests nicht automatisiert werden.

3.4 Automatisierte Testmethoden

Das Ziel von automatisierten Tests ist es, dass man den Testaufwand in einem Software-Projekt 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 Software-Projekt. 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 je-

mandem geschrieben und gewartet werden, auf der anderen Seite sind automatisierte Tests für exploratives Testen ungeeignet. Die Aufgabe des explorativen Testens wird auf absehbare Zeit immer durch eine Person 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 Softwareprozesse sehr wichtig, denn damit bekommt das Entwicklungsteam schneller eine Rückmeldung darüber, ob das System noch korrekt funktioniert.

3.4.1 Komponententest (*Unit Testing*)

Bei einem Komponententest [Mes07] wird genau eine Softwarekomponente getestet. Eine Softwarekomponente ist eine abgeschlossene Einheit in einem Software-Projekt, welche eine definierte Schnittstelle hat. Das kann zum Beispiel eine einzelne Klasse sein, 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 Komponenten 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 für so gut wie jede Programmiersprache ein 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 bekommen, kann man Integrationstests verwenden. Diese werden im nächsten Abschnitt erklärt.

3.4.2 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 worden sind. 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 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 worden ist. Grundsätzlich ist jedoch zu sagen, dass es ein guter Ansatz ist, wenn man mit einer Datenbank die Integrationstests durchführt. 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 steigt.

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 überlegen, wie man für jede Testausführung einen definierten Datenbankzustand herstellen kann. Dieser Datenbankzustand ist sehr wichtig, um reproduzierbare Tests schreiben zu können.

3.4.3 Schnittstellentest (*API Testing*)

Der Schnittstellentest ist die Vorstufe zum Abnahmetest. Dabei werden alle externen Schnittstellen getestet. Dabei kann es sich um eine Schnittstelle in ein externes System handeln oder um eine Web-Service-Schnittstelle. Aber darunter fällt auch die Schnittstelle zwischen Benutzeroberfläche und Geschäftslogik. Diese Schnittstelle ist 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 bei einem Abnahmetest.

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 schlussendlich nur,

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

3.4.4 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.

3.5 Verwendete Technologien

Rayden basiert auf und verwendet eine Vielzahl von unterschiedlichen Technologien, Werkzeugen und Bibliotheken. Dieser Abschnitt gibt einen Einblick in die Technologien und erklärt, in welchen Bereichen diese im Rayden-Framework 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 wurden viele Bibliotheken und Werkzeuge aus dem Eclipse-Umfeld verwendet. Als Testtreiber-Bibliothek wird sowohl eine offene als auch eine kommerzielle Implementierung verwendet.

3.5.1 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-Framework 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.

3.5.2 *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 dieses Konzept baut auch das xText-Projekt auf, welches im nächstem Abschnitt vorgestellt wird.

3.5.3 xText

Das xText-Projekt [Ecl15c] unterstützt das Erstellen von neuen Sprachen. Grundsätzlich ist xText ein Compiler-Generator der aus einer Grammatik einen lexikalischen und Syntax-Analysator generiert. Das besondere an xText ist aber, dass man noch zusätzlich eine 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 nicht nur in Eclipse verwenden kann, sondern auch als eigenständige Anwendung ausführen kann. Somit kann der Aufwand zwei Compiler für seine Sprache zu warten 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.

3.5.4 Selenium

Selenium [Sel15] ist ein Open-Source Projekt, um Webseiten automatisiert testen zu können. Die Bibliothek unterstützt eine Vielzahl an unterschiedlichen Browsern auf allen gängigen Betriebssystemen wie Windows, Linux, Mac 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 propritäre Programmierschnittstelle gesetzt. Seit der Version 2 setzt Selenium auf die standardisierte Programmierschnittstelle WebDriver [W3C15] vom W3C Konsortium. Der Vorteil von WebDriver ist, dass man eine einheitliche Programmierschnittstelle für die unterschiedlichsten Browser hat. Damit erzielt man Unabhängigkeit von einem spezifischen Browser.

3.5.5 Borland Silk Test

Borland Silk Test [Bor15] ist eine kommerzielle Testsoftware für native wie auch Web-Anwendungen. 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-, WPF- 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 X-Browser-Support gibt. Dabei kann man einen Test, welchen man 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 verschiedensten Browser.

Kapitel 4

Aufbau und Ablauf von Testprojekten

Dieses Kapitel befasst sich mit den Testabläufen in einem Softwareprojekt. Diese Abläufe finden in unterschiedlichen Phasen eines Softwareprojekts statt und werden von unterschiedlichen Personengruppen durchgeführt. Diese 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 diese Testansätze stellt die Basis für das Rayden-System dar.

4.1 Ablauf eines Testprojekts

In einem Softwareentwicklungsprojekt gibt es nicht nur die Test-Phase, 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, wir 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 bieten eine große Herausforderung für die Testmanagerin oder dem Testmanager.

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

In den nachfolgenden fünf Unterabschnitten werden die einzelnen Aufgaben

in eine Testprojekt beschrieben. Es wird beschrieben, wie Testfälle entwickelt werden und zu welchem Zeitpunkt in einem Softwareprojekt welche Testaktivitäten ablaufen.

4.1.1 Rollen in einem Testprojekt

Das Testprojekt besteht aus einer bunten Mischung an unterschiedlichen Personen. Die Verantwortung in einem Testprojekts trägt die Testmanagerin oder der Testmanager. Diese Person ist für die Koordination des Projekts zuständig und bildet die Schnittstellen zu anderen Abteilungen. Eine 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 eigene Entwicklerinnen oder Entwickler verfügt.

4.1.2 Testfall

Während der Konzeptionsphase in einem Entwicklungsprojekts werden Anforderungen von der Fachabteilung aufgenommen. Aus diesen Anforderungen werden Anwendungsfälle für das gesamte Projektteam abgeleitet. In der Testabteilung werden aus den Anwendungsfälle 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 durchgeführt. In großen Projekten wird diese Arbeit auch von erfahrenen Testerinnen oder Testern durchgeführt.

4.1.3 Manuelle Abnahmetests für Testfälle

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

4.1.4 Automatisieren von manuellen Abnahmetests

In regelmäßigen Abständen sieht sich die Testmanagerin oder der Testmanager die Ausführungshäufigkeit von manuellen Tests an. Werden manuelle Tests sehr oft durchgeführt, wird dieser Test automatisiert. Bei der Automatisierung werden die manuellen Schritte mithilfe von einem Test-*Framework*

automatisiert. Durch die Automatisierung spart die Testabteilung Zeit und kann somit schneller Ergebnisse über die Qualität der Anwendung liefern.

4.1.5 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 Testfall über die 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 Automatisierungsansätze, welche Versuchen, die manuellen Testfälle direkt zu automatisieren. Einige dieser Ansätze werden im nächste Abschnitt 4.2 erklärt. Ein anderer Vorteil bei diesen Ansätzen ist, dass man die automatisieren Tests noch immer manuell ausführen kann. Diese Eigenschaft kann für Verifikationen von Ergebnissen sehr wichtig sein.

4.2 Evolution der Testautomatisierung

Die Automatisierung von Abnahmetests hat sich über die Zeit einen starken Wandel unterzogen. In diesem Bereich hat es eine ähnliche starke Entwicklung wie bei den Softwareentwicklungstechniken gegeben. Im Jahr 2009 haben Jeff Hinz und Martin Gijsen eine 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 Test entwickelt werden.

Die Abbildung 4.1 zeigt die einzelnen Entwicklungsstufen. In den nächsten Abschnitten werden die Techniken vorgestellt und auf die Vorteile und Nachteile eingegangen.

4.2.1 Erste Generation - *Record-Replay*

Die erste Generation von Testtechniken sind die *Record-Replay*-Ansätze. Dieser Ansatz besteht aus zwei Phasen. In der ersten Phase wird mithilfe einer Analyse-Software 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 beinhaltet dazu spezielle Maus- und Tastatur-Treiber, um die aufgezeichneten Aktionen wiedergeben zu können.

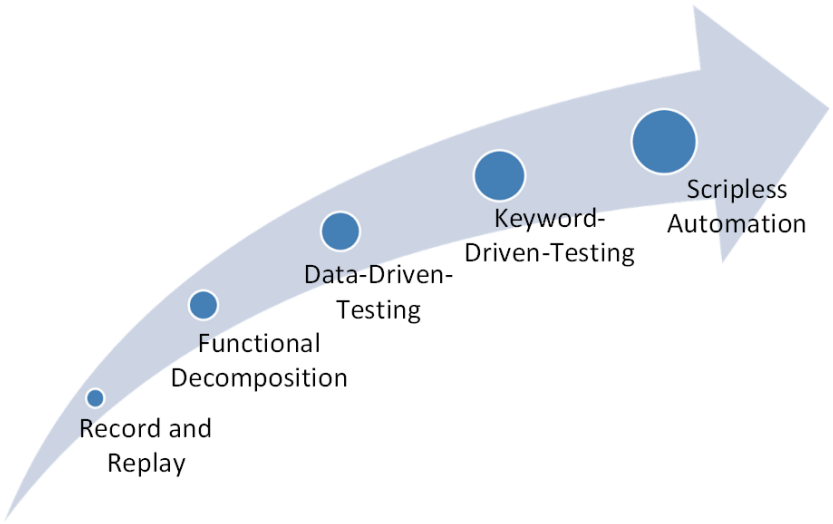


Abbildung 4.1: Die fünf Generationen von Testtechniken

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

4.2.2 Zweite Generation - *Functional Decomposition*

Bei *Functional Decomposition* werden die Tests in einzelne Testsequenzen zerteilt. Mit dieser Technik konnten langen unleserliche Test 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. Ein weiterer positive Effekt kann man in der Wartbarkeit des Testprojekts feststellen. Durch die Reduktion der Test wird auch der der Wartungsaufwand geringer.

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

4.2.3 Dritte Generation - *Data-Driven Testing*

In der dritten Generation von Testmethoden wurde 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 der Stellenwert war nicht hoch. Die Testdaten stehen dafür nun in dieser Generation im Mittelpunkt. Man erkannte, dass man oft die selbe Testsequenz durchläuft, aber jedes Mal andere Daten verwendet. Diese Testtechnik wird stark für datenzentrierten Anwendungen verwenden.

Bei einem *Data-Driven-Testing* werden im Test keine konkrete Werte verwendend. Statt dessen werden Platzhalter (Variablen) im Test eingebaut. Bei der Ausführung eines Tests werden die Platzhalter mit einem Wert aus einer Datenquelle verbunden. Als Datenquelle können Dateien wie auch Datenbanken dienen. Mit dieser Technik, kann man ein und denselben Test mit unterschiedlichen Daten ausführen.

4.2.4 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 Test entweder mithilfe eines *Record-Replay*-Ansatzes aufgezeichnet oder in einer Programmiersprache entwickelt. Der *Record-Replay*-Ansatz war einfach und auch von nicht technisch versierten Personen zu benutzen. Jedoch haben diese aufgezeichneten Test ein Zuverlässlichkeitsproblem. Der zweite Ansatz bedingt, dass die Personen aus der Testabteilung Programmierkenntnisse benötigen.

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

4.2.5 Fünfte Generation - *Scriptless Automation*

Die letzte Methode versucht die Testautomatisierung mit einem *Scriptless-Automation*-Ansatz zu vereinfachen. Bei diesem Ansatz wird versucht, dass man aus einer abstrakten Repräsentation eines Tests den Code zu erzeugen. Bei diesem Transformationsvorgang 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. Diese Ansatz befindet noch in einer frühen Phase und hat in der Praxis bis jetzt noch keine Relevanz.

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

4.3 *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-Framwork* stellt nicht nur eine technische Basis zur Verfügung, sondern bietet auch ein Vorgehensmodell dafür an. Das Vorgehensmodell wird *Acceptance test-Driven Development* (ATDD) genannt und im Artikel [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 worden ist, befindet man
6    ...             sich auf der Hauptseite der Webanwendung.
7
8    Open Browser  ${URL}    ${Browser}
9    Input Text    user      TestUser
10   Input Text    password  secret
11   Click Button  login
```

Programm 4.1: Beispiel von einem *Robot-Framework*-Testfall

Das *Robot-Framework* verwendet als Testdaten-Format eine Tabulator-Syntax. Dabei werden die Daten durch Tabulatoren getrennt. Die Abbildung 4.1 zeigt einen Testfall, welcher in der Tabulator-Syntax definiert worden ist. 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 Entwickler dahinter stellen eine große Anzahl an vorgefertigten Bibliotheken zur Verfügung. Die vorgefertigten Bibliotheken erleichtern und beschleunigen das Entwickeln von Test enorm. Somit muss man nicht bei

jedem neuen Projekt von Null beginnen, sonder kann auf einen Fundus an *Keywords* zurück greifen.

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

Neben den vielen Vorteil des *Robot-Framework*, gibt es aber auch Nachteile. Ein Nachteil wäre die Tabulator-Syntax. Diese Syntax ist Fehleranfällig und ohne einem speziellen Editor nur mühsam zum 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 5

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 erklärt 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, dass man einen Test in einer Java-Anwendung über das *Java-Scripting-API* ausführen kann.

5.1 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 von Rayden ist speziell für Personen im Testbereich ausgelegt. Das wichtigste Ziel der Sprache ist Einfachheit. Die Sprache soll Personen ohne Programmierkenntnisse in die Lage versetzen, Tests in dieser Sprache zu lesen und zu bearbeiten. Die Sprache Rayden ist daher stark an der natürlichen Sprache angelehnt, um den Einstieg zu erleichtern. Ein anderes Ziel bei dem Sprachdesign ist 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 Interpreter für Rayden selbst läuft auch wieder auf der virtuellen *Java*-Maschine (JVM).

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

5.2 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 5.2.1 und 5.2.2.

5.2.1 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.

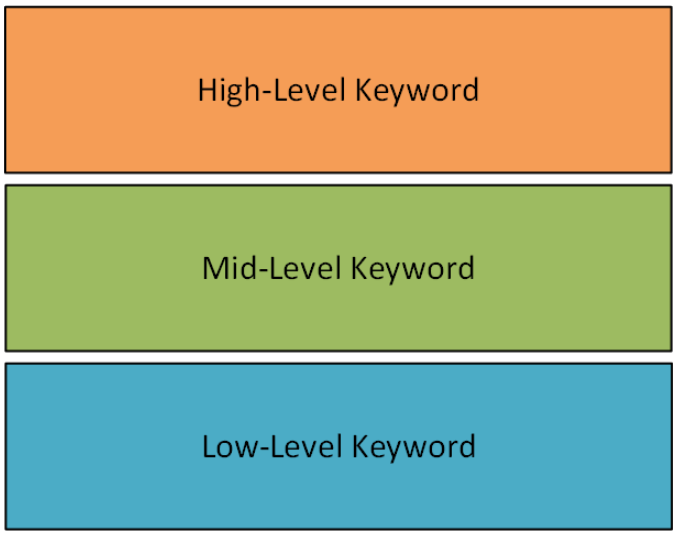


Abbildung 5.1: Aufbau von *Keyword-Driven Testing*

Ein *Keyword-Driven*-Test besteht aus einer Sequenz von *Keywords*. Diese *Keywords* können wiederum aus einer Sequenz von *Keywords* bestehen oder mit einem Codestück verbunden sein. Die *Keywords* werden in drei Kategorien, wie in Abbildung 5.1 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 der 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 finden sich *Mid-Level Keywords* in dieser Sequenz, es können aber auch *Low-Level Keywords* verwendet werden. Die verwendeten *Mid-Level Keywords* bestehen wiederum aus einer Sequenz von *Mid-Level Keywords* und *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 Gegenteil zu *Mid-Level Keywords* wird hier kein Wert auf Wiederverwendung gelegt.

In der letzten Phase werden *Low-Level Keywords* mit Code verbunden. Der 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 5.2 dargestellt. Die zusätzlichen Gruppen bieten einen bessere Strukturierung 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 eine *For-* oder *Print-Keyword* vorstellen. Im Gegensatz dazu sind *Bridge-Keywords* speziell für eine Anwendungstechnologie 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

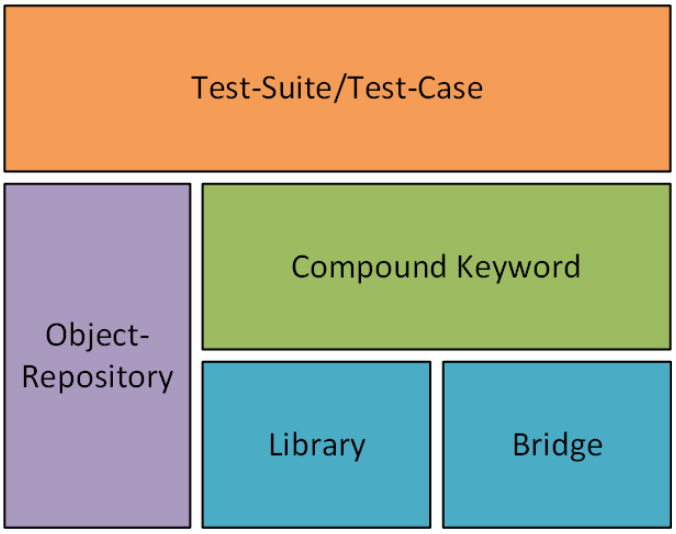


Abbildung 5.2: Aufbau von Rayden

Gruppierungselement für Testfälle, um diese gemeinsam ausführen zu können. Bei der Definition von Testfällen können diese mit unterschiedlichen Testtypen angelegt werden. Eine nähere Beschreibung findet sich im Abschnitt 5.4.6.

Zum Abschluss ist noch auf das *Object-Repository* hinzuweisen. Diese Komponente verwaltet Test-Objekte. Test-Objekte können für die Beschreibung von Benutzeroberflächen-Komponenten wie Schaltflächen oder Eingabefelder verwendet werden. Dafür wird für jedes Test-Objekt ein Bezeichner definiert, mit welchem man die Komponente in der Benutzeroberfläche 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 Benutzerschnittstelle.

5.2.2 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 auf der virtuellen Java-Maschine (JVM). Außerdem bietet Rayden die Möglichkeit, dass man einen Rayden-Test über das *Java Scripting API* [Ora14] ausführen kann.

Abbildung 5.3 zeigt alle Komponenten des Rayden-Kernsystems in Grün.

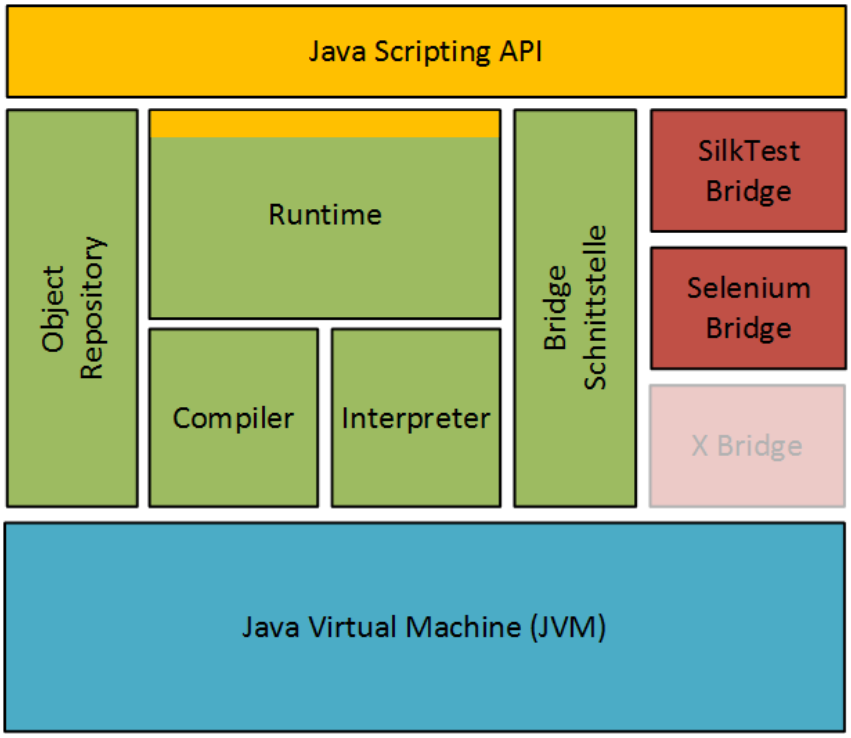


Abbildung 5.3: Rayden-Architektur

Diese Komponenten bilden die Grundlage dafür, einen Rayden-Test ausführen zu können. Als Basis dieser Komponenten sieht man in Blau die virtuelle Java-Maschine (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 oberem Abschnitt der Abbildung 5.3 sieht man das *Java Scripting API*, über welches man Rayden-Tests ausführen kann.

Im nächsten Absatz 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 der *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. Die *Runtime* baut bei

diesem Lesevorgang eine *Lookup*-Tabelle für alle *Keywords* auf. Diese Tabelle wird für einen schnellen Zugriff im Interpreter benötigt. Der Rayden-Test wird mithilfe des Interpreters ausgeführt. Das Ergebnis des Tests wird als Resultat über die *Java Scripting API* zurückgegeben.

- **Compiler**
  Der Compiler für die Rayden-Sprache wird mit dem Compiler-Werkzeug xText [Ecl15c] realisiert. Von dem generierten Compiler wird für die Ausführungseinheit 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 *Runtime*-Komponente verwaltet die Modelle und stellt diese dem Interpreter bei Bedarf zur Verfügung.

- **Interpreter**
  Der Interpreter ist die wichtigste Komponente im Rayden-System. Der Interpreter ist dafür verantwortlich, dass die Rayden-Tests ausgeführt werden können. Zum Starten des Interpreters wird der Aufruf eines *Test-Keywords* übergeben. Dieses *Keyword* wird auf den leeren *Stack* geladen. Der Test wird mithilfe einer *Stack*-Maschine [Wik15a] abgearbeitet. 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. Eine Besonderheit in Rayden ist, dass Gültigkeitsbereiche Zugriff auf andere Gültigkeitsbereiche haben. Eine detaillierte Beschreibung dazu findet sich im Abschnitt 5.4.7. Die Auswertung von Ausdrücken wird in einem separaten Teil des Interpreters vorgenommen. Für die Auswertung wird der aktuelle Gültigkeitsbereich und der Ausschnitt aus dem Modell an die Evaluierungskomponente übergeben. Das Ergebnis des Ausdrucks wird wieder auf den *Stack* gelegt. Ruft die *Stack*-Maschine ein *Scripted-Keyword* (Beschreibung in Abschnitt 5.4) auf, wird entweder der dazugehörige Code ausgeführt oder es wird der Aufruf an die *Bridge*-Schnittstelle weitergeleitet.

- **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* mit der Schnittstelle bildet die Verbindung zwischen der Rayden-*Runtime* und dem Test-Treiber.

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

5.3 Sprache von Rayden

Als Inspiration und Basis für die Sprache 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 Graph vorstellen, in dem die Knoten die *Keywords* repräsentieren und die Kanten die Abhängigkeit zwischen den *Keywords* beschreiben, wie in Abbildung 5.4 dargestellt.

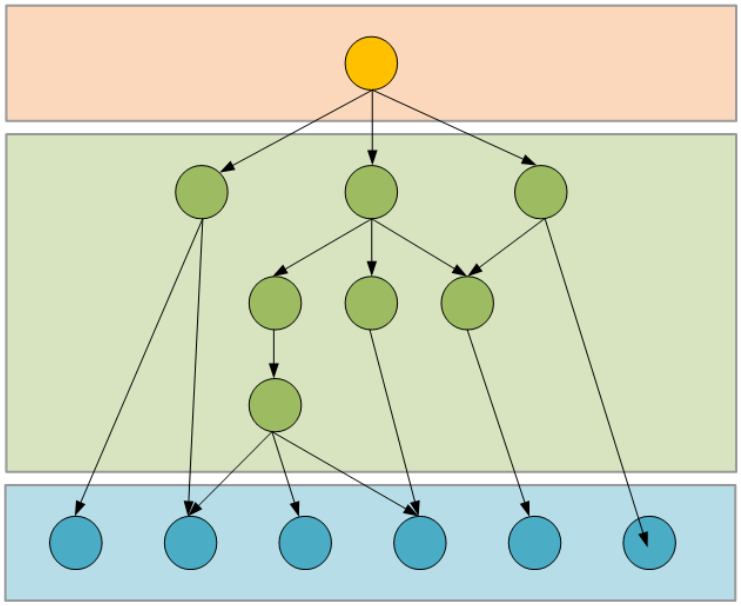


Abbildung 5.4: Graph-Repräsentation eines Tests

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 den Graphen handelt es um einen gerichteten azyklischen Graph (DAG).

Die Rayden-Sprache 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 5.4 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 der Rayden-Sprache wird ein großes Augenmerk darauf gelegt, dass man nicht nur auf ein Wort beschränkt ist, sondern auch ganze Sätze als Namen verwenden kann. Diese Eigenschaft ist sehr nützlich, um die Testfälle wie in einer natürlichen Sprache beschreiben zu können. Der Vorteil ist, dass man somit ohne weiteren Aufwand eine ordentliche Dokumentation des Tests bekommt.

Eine andere interessante Eigenschaft der Sprache ist, dass in der Sprache keine Sprung-Operationen enthalten sind. Die Konsequenz daraus ist, dass es 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 5.4 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.

5.4 *Keywords* von Rayden

Das *Keyword* ist die Schlüsselkomponente der Sprache Rayden. In diesem Abschnitt werden die unterschiedlichen Typen und Metatypen erklärt und gezeigt, wofür diese verwendet werden können. Am Anfang werden die vier Metatypen von *Keywords* erklärt. Die Metatypen sind die Basis für den Funktionsumfang der Sprache. Ferner werden die unterstützten Typen beschrieben und wofür diese verwendet werden können. Als Abschluss werden noch Themen wie Sichtbarkeit, Benennung und Parameter erläutert.

5.4.1 Metatypen

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

5.4.2 Metatype: *Compound Keyword*

Das *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 5.1 zeigt die Verwendung eines *Compound Keywords*. In dem Beispiel kann man gut sehen, dass dieser Metatyp hauptsächlich für die Strukturierung von Tests verwendet wird. Ein mögliches Vorgehen kann dabei sein, dass man einen Testfall immer weiter und weiter in *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*.

```
1 keyword Anmelden an der PetClinic Anwendung {
2   '''Man meldet sich bei der Anwendung PetClinic mit den definierten
3      Daten an. Wenn das Keyword erfolgreich ausgeführt worden ist,
4      befindet man sich auf der Hauptseite der Webanwendung.'''
5
6   parameter in username as string
7   parameter in password as string
8
9   Type Text(@PetClinic.LoginPage.Username, username)
10  Type Text(@PetClinic.LoginPage.Password, password)
11
12  Click Left(@PetClinic.LoginPage.LoginButton)
13 }
```

Programm 5.1: Das Beispiel zeigt das *Compound Keyword* Anmelden an der PetClinic Anwendung

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.

5.4.3 Metatype: *Inline Keyword*

Der Metatype *Inline Keyword* ist eine Kurzform des *Compound Keywords*. Dabei kann man in einem *Compound Keyword* ein neues *Keyword* erstellen. Daher kommt auch der Name *Inline Keyword*, weil es innerhalb eines anderen *Keywords* erstellt wird. Im Beispiel 5.2 wird im *Keyword Anmelden an*

der PetClinic Anwendung das *Inline Keyword Besitzer anlegen* definiert. Es werden alle Schritte zum Anlegen eines neuen Besitzers zusammengefasst. Der Anwendungsfall dieses Metatyps ist wiederum die Strukturierung, aber in diesem Fall innerhalb eines *Keywords*.

```
1 testcase Anlegen eines neuen Besitzers {
2   '''Der Testfall überprüft den Anwendungsfall um einen
3      neuen Besitzer anlegen zu können.'''
4
5   Anmelden an der PetClinic Anwendung ("max", "secret")
6
7   Besitzer anlegen {
8     Oeffnen der Besitzerseite
9     Neuen Besitzer in der Anwendung anlegen("Huber", "Mayr")
10    Daten von Besitzer ueberpruefen
11  }
12
13  Abmelden von der Anwendung
14 }
```

Programm 5.2: Beispiel von einem *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.

5.4.4 Metatype: *Scripted Keyword*

Das *Scripted Keyword* ist 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 5.3 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 vollständige Name der Klasse verwendet.

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 die *Stack*-Maschine zu einem Aufruf eines *Scripted Keywords*, wird ein neues Objekt der *Keyword*-Implementierung über den *Java-Reflection*-Mechanismus angelegt. Von dem 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 5.4 sehen kann. Als Ergebnis liefert die

```
keyword Print {
  '''Der Parameter 'text' wird in den Test-Report geschrieben.'''

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

Programm 5.3: Rayden: Beispiel *Scripted Keyword*

Methode ein *KeywordResult*-Objekt. Dieses Objekt signalisiert der *Stack*-Maschine, ob das *Keyword* erfolgreich ausgeführt worden ist.

```
public class PrintKeyword implements ScriptedKeyword {

  @Override
  public KeywordResult execute(String keyword,
      IKeywordScope scope, IRaydenReporter reporter) {
    reporter.log(scope.getVariable("text").toString());
    return new KeywordResult(true);
  }
}
```

Programm 5.4: Rayden: *Java*-Implementierung des *Print Keywords*

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.

5.4.5 Metatype: *Scripted Compound Keyword*

Das *Scripted Compound Keyword* ist die komplizierteste Variante der vier Metatypen, jedoch ist diese Variante essentiell für die Flexibilität der Sprache. 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.

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

Programm 5.5: Beispiel für ein *Scripted Compound Keyword*

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

Das Beispiel 5.5 zeigt die Definition für ein *IF Keyword*. Dabei wird wie bei einem *Scripted Keyword* die Programmiersprache und der Bezeichner definiert. Im Fall von einem *Scripted Compound Keyword* muss die Klasse das *Interface ScriptedCompoundKeyword* implementieren. Dieses *Interface* ist deutlich schwieriger zu implementieren, wie man im Beispiel-Code 5.6 sehen kann.

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 der *Stack*-Maschine an die *Keyword*-Implementierung übergeben. Falls die Informationen für die Ausführung benötigt werden, können diese im Objekt gespeichert werden. Eine Instanz der *Keyword*-Implementierung wird genau für eine Ausführung verwendet. Das heißt, man kann keinen globalen Zustand für zukünftige Ausführungen speichern. Falls man diese Funktionalität benötigt, muss man diese Daten in Klassenvariablen speichern.

- **Phase 2: Beginn der Auswertung (*executeBefore*)**
  Die Ausführung 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 *executeAf-*

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

Programm 5.6: *Java*-Implementierung des *IF Keywords*

ter 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 für Validierungen verwendet werden.

Die Verwendung eines *Scripted Compound Keyword* sieht wie ein *Inline Keyword* aus. Der große Unterschied ist, dass ein *Inline Keyword* keine Parametersignatur im Gegensatz zu einem *Scripted Compound Keyword* hat. Ein Beispiel für die Verwendung findet man im Code-Ausschnitt 5.7.

```
keyword If Keyword Bespiel {
  If (a == 1) {
    Print("Condition is true")
  }
  If (test == "b") {
    Print("Condition is false")
  }
}
```

Programm 5.7: Verwendung des *IF Keywords*

5.4.6 Typen

Neben den Metatypen für *Keywords* gibt es in Rayden auch unterschiedliche Typen von *Keywords*. Die Typen liefern keine zusätzliche Funktionalität für die Sprache, sondern dienen als Strukturierungselement für Testprojekte. Mithilfe der Typen kann man Testfälle unterscheiden und eine klare Zuordnung zu einer Testmethode treffen. Durch die Typen wird es auch möglich, eine Aussage über die Verteilung der Testmethoden in einem Testprojekt treffen zu können.

Rayden unterstützt die folgenden *Keyword*-Typen:

- Test-Suite (*TestSuite*),
- Testfall (*TestCase*),
- Komponententest (*UnitTest*),
- Integrationstest (*IntegrationTest*),
- Schnittstellentest (*APITest*),
- Automatisierter Abnahme-Test (*AUTest*) und
- Manueller Abnahme-Test (*MAUTest*).

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

5.4.7 Gültigkeitsbereich

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 worden sind, verhalten sich diese gleich wie Variablen. Variablen können von einem *Keyword* in einem Gültigkeitsbereich 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. Sobald das erste Mal eine Variable mit einem Wert belegt worden ist, ist diese im Gültigkeitsbereich vorhanden.

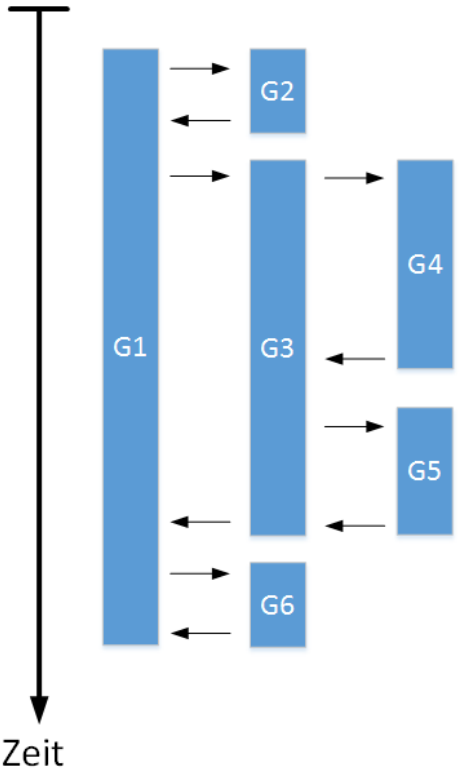


Abbildung 5.5: Gültigkeitsbereiche in einem Rayden-Test

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 5.5. 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.

In der 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 komisch wirken. Den Pfeil kann man aber damit erklären, dass es in Rayden möglich ist, einen *Out*- bzw. *InOut*-Parameter zu definierten. Damit können Variablen aus G2 an G1 übertragen werden. Eine detaillierte Beschreibung zu den Parametern findet man im Abschnitt 5.4.8.

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, für eine explizite Definition von Variablen. Das wird verwendet, um sicherzustellen, dass eine Variable definitiv zur Verfügung steht bzw. erleichtert auch die Verwendung eines *Keywords*.

5.4.8 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 Bespiel {
2
3   parameter in    parm1
4   parameter in    parm2 as string
5   parameter out   param3 as boolean
6   parameter inout param4 as number
7
8   Test1
9 }
```

Programm 5.8: Verwendung von Parametern

Rayden unterstützt sowohl typisierte als auch untypisierte Parameter. Sind die Parameter typisiert, werden diese vom Rayden-Interpreter 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 Gültigkeitsbereich. In Rayden werden die Richtungen *In*, *Out* und *InOut* unterstützt, wie man im Code-Beispiel 5.8 sehen kann.

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

- ***Out*-Parameter**
  Der *Out*-Parameter ist das genaue Gegenteil zum *In*-Parameter. Dabei wird ein Wert aus dem Kinder-Gültigkeitsbereich in den Eltern-Gültigkeitsbereich transferiert.

- ***InOut*-Parameter**
  Der dritte Variante ist eine Kombination aus dem *In*-Parameter und dem *Out*-Parameter.

5.5 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 5.9: Verwendung von einem *enumeration*-Parameter

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 sieht man im Code-Ausschnitt 5.9.

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 einem @-Symbol. Nachfolgend kann man einen Pfad im *Object-Repository* beschreiben, wie man im Beispiel 5.10 sehen kann. Für Abnahme-Tests ist das Referenzieren von Test-Objekten essentiell. Daher bietet Rayden dafür eine Erleichterung.

```
1 Click Left( @PetClinic.PetClinicWeb.Login.Go )
2 @PetClinic.PetClinicWeb.Login.Go :: Click Left
```

Programm 5.10: Verwendung vom Datentyp *location*

Falls der erste Parameter von einem *Keyword* vom Datentyp *location* ist, kann man diesen Parameter vor das *Keyword* schreiben. Somit wird das Lesen eines Tests erleichtert. Ein Verwendung dazu findet man ebenfalls im Beispiel 5.10. Dieses Sprachfeature wird von der Rayden-Laufzeitumgebung wieder in einen klassischen *Keyword*-Aufruf umgebaut.

```
 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 5.11: Verwendung vom Datentyp *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 Code-Ausschnitt 5.11.

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 die Ausführungseinheit für Ausdrücke versuchen, diese Variable mit einem Wert aus dem Gültigkeitsbereich zu ersetzen. Wird kein Wert für gefunden, wird die Ausführung mit einem Fehler abgebrochen.

Eine Typumwandlung ist in der Sprache Rayden nicht vorgesehen. Es ist zwar möglich, dass man alle Datentypen zu einem *string*-Datentyp umwandeln kann, aber alle anderen Kombinationen sind nicht möglich. In der Implementierung von einem *Keyword* können die Werte beliebig konvertiert werden. Die Laufzeitumgebung stellt den Datentyp nur innerhalb der Gültigkeitsbereiche sicher.

5.6 Verarbeiten von *Keywords* und Ausdrücken

Im Rayden-System ist der Interpreter und die *Runtime* für die Ausführung eines Tests zuständig. Dabei wird die Ausführung von *Keywords* und Ausdrücken von einander getrennt. Die *Keywords* werden von einer *Stack*-Maschine ausgeführt.

Die Ausdrücke werden in einer eigenen Ausführungseinheit behandelt. Die Ausführungseinheit verwendet keine *Stack*-Maschine, sondern den rekursiven Abstieg für die Auswertung. Dabei kann diese Einheit 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 die Ausführungseinheit entweder anders verhalten oder zusätzliche Überprüfungen durchführen.

5.7 *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.

```
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 5.12: Bibliothek: *stdlibrary.rlg*

In einer Bibliothek werden *Keywords* thematisch zusammengefasst. Man kann sich zum Beispiel vorstellen, dass es eine Standard-Bibliothek gibt, wie im Code-Beispiel 5.12 zu sehen ist. In diesem Beispiel sind *For-, If-* und *Print-Keyword*-Definitionen enthalten. Die Datei *stdlibrary.rlg* und das dazugehörige *Java*-Archiv bilden eine Rayden-Bibliothek.

Um eine Bibliothek verwenden zu können, muss man diese über eine *import library* Direktive einbinden. Ein Beispiel sieht man dazu im Code-Ausschnitt 5.13. 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 aktuellen *Keywords* aus der Datei und danach die Bibliotheken in der Reihenfolge, in welcher diese definiert wurden.

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 Testtreiber gekoppelt, welcher die

```
1 import library "stdlibrary"
2
3 keyword Library Beispiel {
4   If (1 == 1){
5     Print("Condition is true")
6   }
7   For ("i", 0, 2){
8     Print("Hello - " + i)
9   }
10 }
```

Programm 5.13: Verwendung der *StdLib* Bibliothek

Basisfunktionen zur Verfügung stellt. Die *Keywords* kapseln die Funktionalität aus dem Testtreiber und stellen diese zur Verfügung. Eine *Bridge* kann zum Beispiel das Steuern eines Browsers unterstützen und verwendet dazu Selenium.

5.8 *Object Repository*

Das *Object Repository* stellt eine Abstraktion zur Test-Anwendung her. Alle Test-Objekte, 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 Test-Objekt verwendet werden, sondern nur eine Referenz darauf.

Die Test-Objekte werden im *Object Repository* in einem Baum verwaltet und können über diesen auch referenziert werden. Das Beispiel 5.14 zeigt ein *Object Repository* für eine Webanwendung, welche als Bezeichner einen *XPath*-Ausdruck verwendet. Der Vorteil davon ist, das der Bezeichner *location* über die Baumstruktur zusammengebaut wird. Dadurch erspart man sich viel Wartungsaufwand.

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

Es ist auch möglich, das man ein Test-Objekt 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 Code-Ausschnitt 5.15. Im Bezeichner *location* werden keine Ausdrücke unterstützt. Die Parameter werden über eine Substituierung ersetzt und benötigen daher keinen Datentyp.

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

Programm 5.14: *Object-Repository*

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

Programm 5.15: *Parametrisiertes Test-Objekt*

5.9 *Java-Scripting-API*

Das Rayden-System implementiert das*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.

Um einen Rayden-Test in ein *Java*-Programm einbinden zu können, muss man zuerst die *RaydenScriptEngineFactory* registrieren. Damit gibt man

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

Programm 5.16: Code-Beispiel: *ScriptEngineFactory* für Rayden registrieren

dem *ScriptEngineManager* eine neue Sprache bekannt. Im Code-Beispiel 5.16 sieht man eine Möglichkeit, wie man die Rayden-Sprache über einen Namen registrieren kann. Es gibt auch noch andere Möglichkeiten, wie etwa das Registrieren über die Dateiendung.

```
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 5.17: Code-Beispiel: Ausführen eines Rayden-Tests

Über die *ScriptEngineFactory* kann der *ScriptEngineManager* eine neue Instanz einer *ScriptEngine* anlegen. Das Code-Beispiel 5.17 zeigt, wie man einen Test aus einer Datei einliest und diesen über die *ScriptEngine* ausführen lassen kann.

In diesem Kapitel wurde der Aufbau und die Verwendung des Rayden-Systems beschrieben. Im nächsten Kapitel wird ein Testprojekt mit dem Rayden-System umgesetzt. Dazu werden alle vorher beschriebenen Testmethoden angewendet.

Kapitel 6

Implementierung von Rayden

Diese Kapitel beschreibt die Implementierung von ausgewählten Komponenten des Rayden-System. Die Abschnitte 6.1 und 6.2 zeigen die Grammatik und die *Stack*-Maschine für die Ausführung von *Keywords*. Die Abschnitte enthalten Codeauschnitte der Implementierung und Teile der xText-Grammtik.

Der Abschnitt 6.3 befasst sich mit der Umsetzung von Ausdrücken im Rayden-System. Dazu werden in diesem Abschnitt Auszüge der Grammatik und Teile der *RaydenExpressionEvaluator* Klasse erklärt. Die Klasse *RaydenExpressionEvaluator* ist für die Auswertung der Ausdrücke zuständig.

Im Abschnitt 6.4 wird die Validierung von Rayden-Tests gezeigt. Dafür wird das Validierungssystem von *xText* verwendet. Abgeschlossen wird diese Kapitel mit dem Abschnitt 6.5, welcher die Integration des Rayden-Systems in das *Java-Scripting-API* zeigt.

6.1 Umsetzung der *Keyword*-Grammatik

Die Rayden-Sprache wurde mit dem xText [Ecl15c] Compilerwerkzeug umgesetzt. Die Abbildung 6.1 zeigt einen Auszug aus der Grammatik für die Rayden-Sprache. Der Auszug zeigt die Grammatikregeln für *Keywords*. Die Regel *KeywordDecl* beginnt die Definition eines neuen *Keywords*. Am Beginn der Regel wird der Typ für das *Keyword* definiert. Eine Beschreibung und Auflistung der Typen ist in Abschnitt 5.4.6 enthalten. Danach folgt ein Name für das *Keyword* welcher von einer öffneten geschwungenen Klammer gefolgt wird.

Die geschwungenen Klammer definieren den Bereich der *Keyword*-Implementierung. Am Anfang der Implementierung kann eine optionale Beschreibung ange-

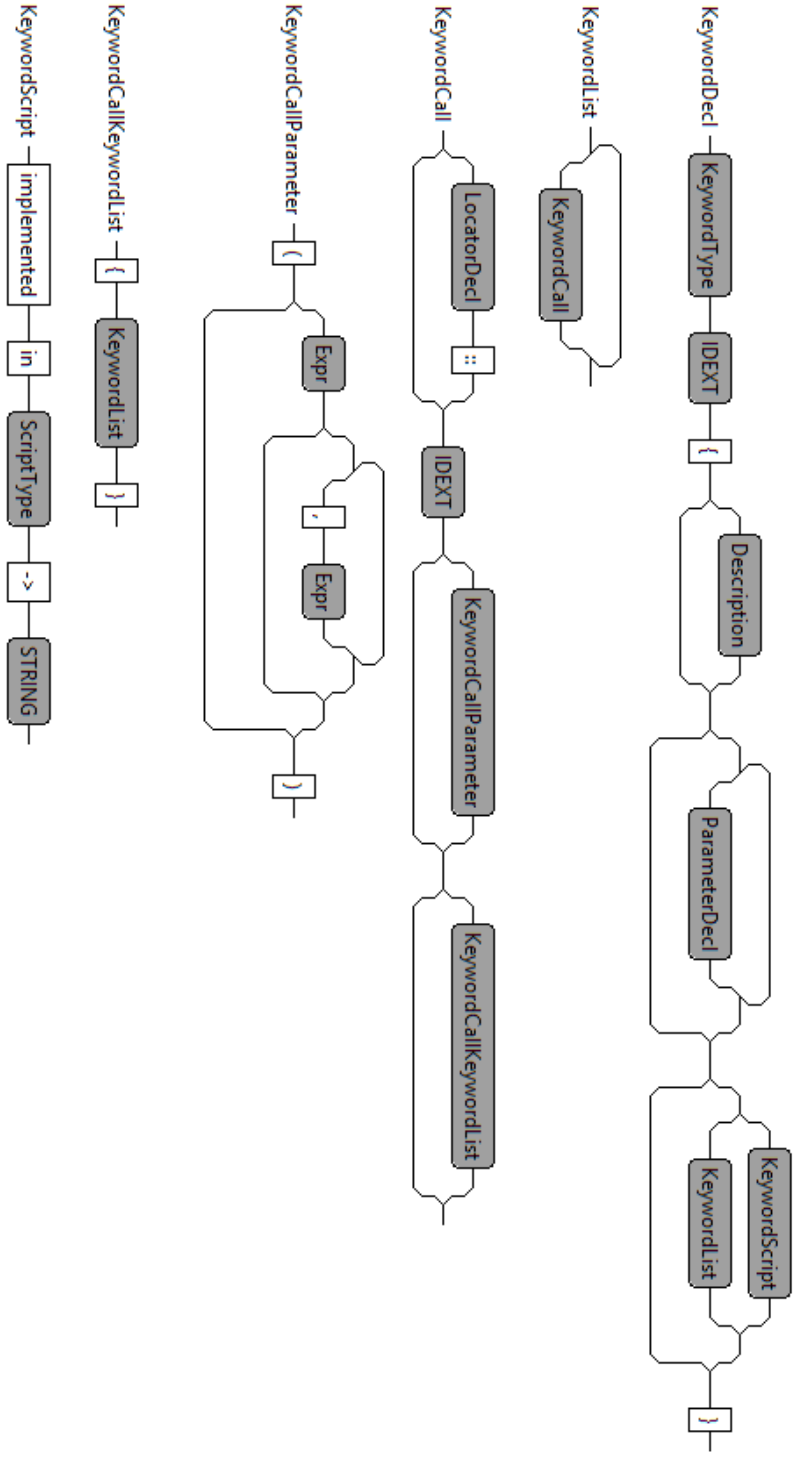


Abbildung 6.1: Auszug aus der Grammatik für *Keywords*

führt werden. Diese wird von einer Parameterliste gefolgt. Ein Parameter wird mit der Regel *ParameterDecl* beschrieben und kann 0 bis N Mal wiederholt werden. Eine Parameter-Definition besteht aus dem Schlüsselwort *parameter*, einen Namen, einen Datentyp und einer Richtung.

Danach folgt entweder die Bindung an ein Codestück mit der Regel *KeywordScript* oder die *Keyword*-Liste mit der Regel *KeywordList* im Fall eines *Compound-Keywords*. Die beiden Regeln 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 einem anderen Testpersonal vorgenommen wird.

```
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 6.1: Syntaktischer Zucker für die Verwendung von *location*-Datentypen

Die Regel *KeywordCall* definiert den Aufruf von einem *Keyword* in einer *Keyword*-Liste. Die Regel fängt normalerweise mit dem Namen des aufzurufenden *Keywords* an. Danach folgt optional die Parameterliste für den Aufruf von dem *Keyword*. Die Regel *KeywrodCallParameter* definiert die Parameterliste, welche durch runde Klammer umschlossen ist. Die Parameter können als Liste von *Expr*-Regeln definiert werden und werden durch einen Beistrich separiert werden. Für die einfachere Verwendung und besserer Lesbarkeit enthält die Regel *KeywordCall* auch noch syntaktischen Zucker. Falls der erste Parameter von einem *Keyword* vom Typen *location* ist, kann diese Parameter vor das *Keyword* geschrieben werden. Somit lässt sich die Implementierung leicht lesen. Der Codeausschnitt in Abbildung 6.1 zeigt dazu die Verwendung des 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.

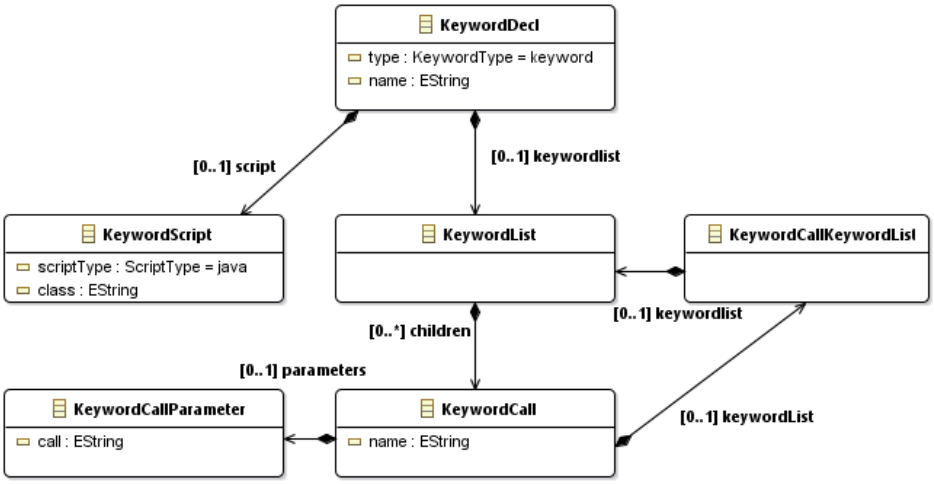


Abbildung 6.2: Ausschnitt aus dem Abstrakte Syntaxbaum

6.2 Ausführung von *Keywords* mit einer *Stack*-Maschine

Im vorigen Abschnitt 6.1 wurde die Grammatik von einem *Keyword* in der Sprache Rayden erklärt. Dieser Abschnitt beschäftigt sich mit der Ausführung von *Keywords*. Damit die *Stack*-Maschine arbeiten kann, benötigt diese einen Zugriff auf den abstrakten Syntaxbaum.

Das Compilerwerkzeug *xText* stellt dafür ein *Eclispe-ECore*-Modell zur Verfügung. Der generierten Compiler ist so konzipiert, dass dieser die gesamte Datei einliest und daraus einen abstrakten Syntaxbaum generiert. Der Syntaxbaum steht für die weitere Verarbeitung als *ECore*-Modell zur Verfügung. Einen Auszug aus dem Modell zeigt die Abbildung 6.2. Die Abbildung zeigt die Modell-Repräsentation der Grammatik-Regeln von Abbildung 6.1. Dieser Ausschnitt aus dem Modell stellt die Basis für die *Stack*-Maschine dar.

Die *Stack*-Maschine für das Rayden-System ist in der Klasse *RaydenRuntime* implementiert. Der Codeauszug 6.2 zeigt die essentielle Methode *executeKeyword*, welche für die Ausführung verantwortlich ist. Die Methode wird mit einem *KeywordCall*-Objekt aufgerufen. Diese Objekt bezeichnet das erste *Keyword*, welches von der *Stack*-Maschine aufgerufen wird. Als erstes wird in der Methode übriggebliebene Element von dem *Stack* entfernt. Danach wird über das *Reporter-Interface* alle Objekte notifiziert, dass ein neuer Testfall gestartet worden ist. 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 der *Stack*-Maschine wird die Abarbeitung gestartet. Es werden nun solange die Gültigkeitsbereiche am *Stack* abgearbeitet, bis der *Stack* leer ist oder ein Fehler bei der Ausführung von einem *Keyword* aufgetreten ist. Der Gültigkeitsbereich repräsentiert einen Aufruf von einem *Keyword* 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 *getNextOpt* kann die *Stack*-Maschine das nächste *Keyword* aus dem aktuellen Gültigkeitsbereich laden. Liefert die Methode kein Wert, ist die Ausführung des Gültigkeitsbereiches am 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 ein Fehler geworfen und die Ausführung abgebrochen. Handelt es sich jedoch um ein *KeywordDecl*-Objekt wird das *Keyword* ausgeführt.

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

Wurden alle Gültigkeitsbereich am *Stack* erfolgreich abgearbeitet wird am Ende noch das *Reporter-Interface* aufgerufen. Danach wird die Ausführung der *Stack*-Maschine beendet.

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

Programm 6.2: Codeauszug aus der *RaydenRuntime*-Klasse

6.3 Auswertung von Ausdrücken

Dieser Abschnitt befasst sich mit den Grammatik-Regeln und der Ausführung von Ausdrücken. Die Rayden-Sprache unterstützt in einigen Bereichen der Sprache Ausdrücke. Ein Ausdruck kann in der Grammatik mit der Regel *Expr* aufgerufen werden. Die Abbidlung 6.3 zeigt einen Überblick über die Grammatik-Regeln von Ausdrücke.

Die Regeln für den Ausdruck sind klassisch aufgebaut. Die Operationen sind nach der Ausführungsreihenfolge in den Regeln eingearbeitet. Die am stärksten bindenden Operationen befinden sich in der Nähe der Blätter des Ausdrucksbaum. 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 eine definierten Datentypen. Die Regel *Fact* hat jedoch keinen 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 Ausführung von Ausdrücken ist im Rayden-System die Klasse *RaydenExpressionEvaluator* zuständig. Der Codeausschnitt 6.3 zeigt eine Überblick über die Klasse *RaydenExpressionEvaluator*. Die 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 die Methode wird ein *Expr*-Objekt und eine Zeichenkette übergeben. Das *Expr*-Objekt ist das Wurzelobjekt im *ECore*-Modell für eine Ausdruck. Mit dem zweiten Parameter kann man die Auswertung des Ausdrucks typisieren. Wurde ein Typ definiert, wird am Ende der Auswertung noch überprüft, ob das Ergebnis vom selben Typ ist. Stimmt der Wert nicht überein, wird eine Fehler geworfen. Es gibt auch noch eine 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 6.3 enthählt am Ende die Implementierung der *eval*-Methode für das Objekt *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.

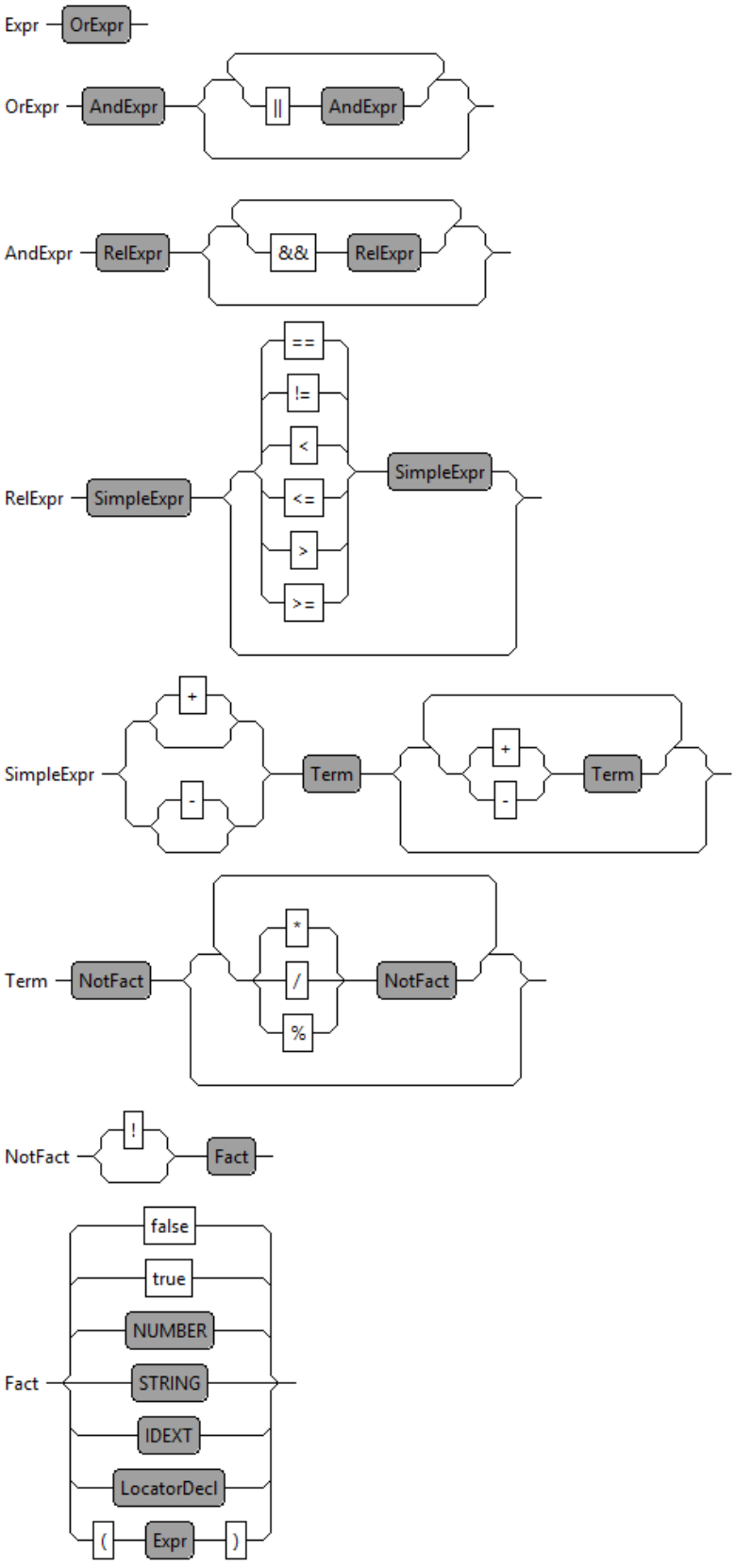


Abbildung 6.3: Grammatik-Regeln für Ausdrücke

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

Programm 6.3: Codeauszug aus dem *RaydenExpressionEvaluator*

6.4 Validierung eines Rayden-Tests

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

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

Programm 6.4: Codeauszug aus dem *RaydenDSLJavaValidator*

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

Das Codebeispiel 6.4 zeige eine Validierung für die Verwendung von *Keywords*. Diese Validierung wird für alle *KeywordCall* Modellelemente aufgerufen. Ein *KeywordCall* stellt einen Aufruf von einem *Keyword* dar. Die Validierung überprüft, ob für jeden Aufruf von einem *Keyword* auch eine Implementierung vorhanden ist. Das Traversieren des Modell und suchen alle Modellelementen entfällt, da diese Aufgabe vom *xText-Framework* durchgeführt wird.

Im ersten Schirtt überprüft die Validierung aus dem Codebeispiel 6.4 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, werden im nächsten Schritt mithilfe der *Lookup*-Tabellen gesucht, ob eine Implementierung für das *Keyword* vorhanden ist. Wurde keine Implementierung gefunden, wird eine Warnung ausgegeben. Das Fehlen einer Implementierung liefert nur eine Warnung. Würde diese Validierung einen Fehler liefern, würde der *Compiler* mit diesem Fehler abbrechen und es können 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.

6.5 Integration von Rayden in das *Java-Scripting-API*

Das Rayden-System besitzt eine Integration in das *Java-Scripting-API*. Das *Java-Scripting-API* ist eine standardisierte Schnittstelle für das Ausführen von Skriptsprachen in *Java*. Über diese Schnittstellen kann direkt in einem *Java*-Programm ein Skript in einer beliebigen Sprache ausführen. Die einzige Einschränkung dabei ist, dass für die Skriptsprache eine *ScriptEngineFactory* registriert worden ist. Das *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 das *Java-Scripting-API* muss man die Schnittstellen *javax.script.ScriptEngineFactory* und *javax.script.ScriptEngine*

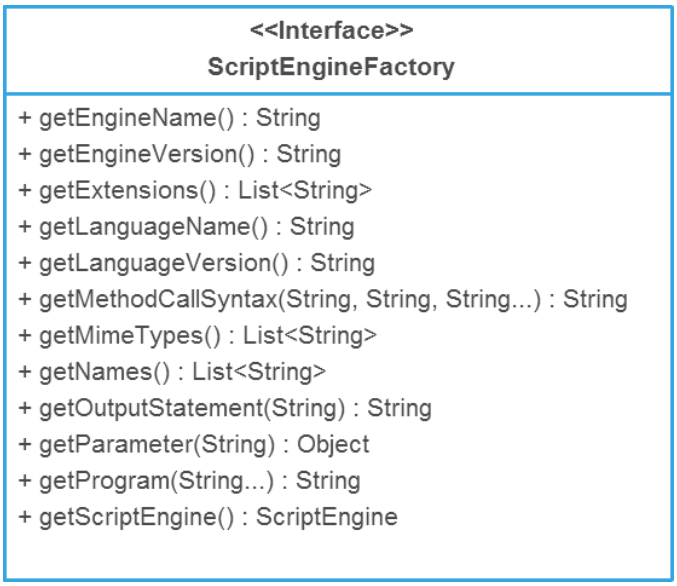


Abbildung 6.4: ScriptEngineFactory UML-Klassendiagramm

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 6.4. 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 6.5 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. Standardmäßig wird die *RaydenXMLReporter*-Implementierung verwendet, welche die gesamte Ausgaben einer Test-Ausführung in eine XML-Datei protokolliert.

Damit man in einem Rayden-Test eine Bibliothek verwenden kann, ist es für die *Runtime* entscheiden, dass spezifiziert ist, wo die Bibliotheken gefunden werden. Dafür wird ein Arbeitsbereich (*Working Folder*) definiert, in welchem Bibliotheken und andere externe Ressourcen gesucht werden.

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

Programm 6.5: Codeauszug aus der *RaydenScriptEngine*

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 worden sind, wird das Skript mit allen Abhängigkeiten geladen. Dazu wird das Skript mit der Methode *loadRaydenFile()* geladen. Die Implementierung der Methode zeigt der Codeauszug 6.6. Wie der Codeauszug zeigt, gibt es zwei Methoden für das Laden von Rayden-Dateien. 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 es zwei *Lookup*-Tabellen für *Keywords* vorhanden sind. Es werden alle *Keywords* aus der primären Rayden-Datei in eine spezielle *Lookup*-Tabelle geladen, dass nur *Keywords* aus dieser Datei ausgeführt werden können.

Zum Laden der Rayden-Datei wird der generierte xText-*Parser* verwendet.

```
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, Map<String, KeywordDecl>
      keywordStore) {
11    IParseResult result = parser.parse(reader);
12    if (result.hasSyntaxErrors()) {
13      for (INode error : result.getSyntaxErrors()) {
14        reporter.error(error.getSyntaxErrorMessage().toString());
15      }
16      throw new ParseException("Provided input contains syntax error.");
17    }
18
19    if (result.getRootASTElement() instanceof Model) {
20      reporter.log("Successful loaded model.");
21      Model model = (Model) result.getRootASTElement();
22
23      EList<KeywordDecl> keywords = model.getKeywords();
24      reporter.log("Loading " + keywords.size() + " keywords ...");
25      for (KeywordDecl keyword : keywords) {
26        keywordStore.put(RaydenModelUtils.normalizeKeyword(keyword.
      getName()), keyword);
27      }
28
29      EList<ImportDecl> imports = model.getImports();
30      for (ImportDecl importDecl : imports) {
31        try {
32          loadLibraryFile(new FileReader(new File(workingFolder,
      importDecl.getImportLibrary())));
33        } catch (Exception e) {
34          reporter.error("Error during loading library '" + importDecl.
      getImportLibrary() + "'");
35        }
36      }
37    }
38  }
39  ...
40 }
```

Programm 6.6: Laden von Rayden-Dateien

Im nächsten Schritt wird das Ergebnis des *Parser* überprüft, ob ein Syntaxfehler aufgetreten ist. Bei einem Fehler wird die Ausführung sofort beendet. Konnte die Datei ohne Fehler geladen werden, liefert der *Parser* eine Instanz

des *ECore*-Modells. Es wird das Modell durchlaufen und alle *Keywords* die gefunden werden in die *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 wird in diesem Fall die *Keywords* in eine andere *Lookup*-Tabelle gespeichert.

Nachdem das Skript und alle externe Ressourcen erfolgreich geladen worden sind, 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 einem Testtypen. Die Rayden-Sprache unterstützt folgende Testtypen:

- Test-Suite (*TestSuite*),
- Testfall (*TestCase*),
- Komponententest (*UnitTest*),
- Integrationstest (*IntegrationTest*),
- Schnittstellentest (*APITest*),
- Automatisierter Abnahme-Test (*AUTest*) und
- Manueller Abnahme-Test (*MAUTest*).

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

In diesem Kapitel wurden einige essentielle Komponenten des Rayden-Systems gezeigt und wie die Komponenten umgesetzt worden sind. Im nächsten Kapitel 7 wird gezeigt, wie man Tests mit dem Rayden-System schreiben kann. Dazu werden unterschiedliche Test-Methoden verwendet um die Stärken von Rayden zu zeigen.

Kapitel 7

Umsetzung eines Testprojektes mit Rayden

Es wird gezeigt, wie man ein Testprojekt mithilfe von Rayden umsetzten kann. Dabei wird eine einfache Webanwendung getestet. Zum Beispiel ein Rechner oder kleine Task Anwendung. Dafür werden alle Ebenen von funktionalen Tests durchgeführt.

7.1 Beispielanwendung

Für das Evaluieren des Rayden-Systems wird eine Anwendung zum Test benötigt. Bei der Anwendung sollte es sich um eine Webanwendung handeln, um die Unterstützung von Selenium zeigen zu können. Für die Evaluierung hat man sich für die *PetClinic*-Webanwendung entschieden, welche eine Beispielanwendung des Spring-Projekts ist. Die Anwendung 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 der Anwendung können Besuche bei einem Tierarzt protokolliert werden. Dazu gehört die Erfassung der Tierbesitzer mit ihren Haustieren. Zu jedem Haustier werden alle Arztbesuche gespeichert, damit der Krankheitsverlauf dokumentiert ist. Neben den Besitzer und ihren Tieren werden Tierärztinnen und Tierärzte verwaltet.

Da der Funktionsumfang der Anwendung überschaubar ist, eignet sich diese ausgezeichnet als Beispielanwendung für die Evaluierung des Rayden-System. In den nächsten Abschnitten werden Test mit unterschiedlichen Testmethoden für die *PetClinic*-Anwendung gezeigt.

In den folgenden Abschnitte 7.2, 7.3 und 7.4 werden Tests für die *Petclinic*-Anwendung vorgestellt. Diese Tests wurden mit drei unterschiedlichen

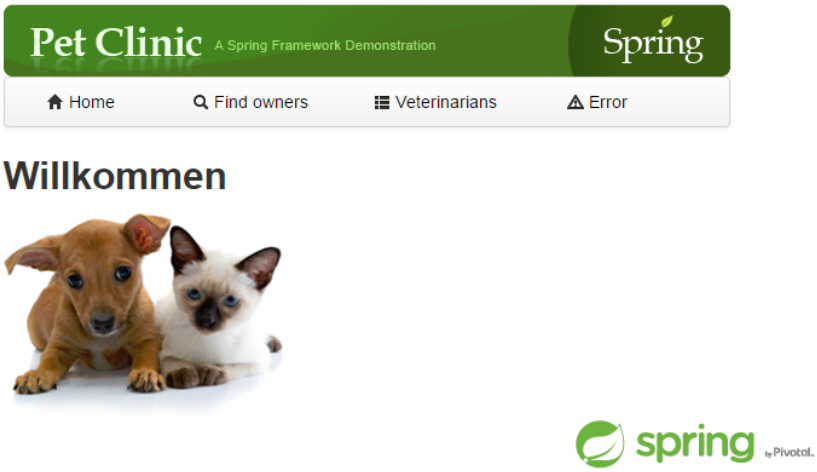


Abbildung 7.1: Startseite der Webanwendung PetClinic

Testmethoden umgesetzt um zu zeigen, wie man die Testmethoden mit dem Rayden-System vereinen kann.

7.2 Komponententests

Dieser Abschnitt zeigt die Umsetzung eines Komponententests mit dem Rayden-System. Für einen Komponententest wir in Rayden zuerst ein *Keyword* angelegt. Der Codeauszug 7.1 zeigt die Definition des *Keywords*. Ein Komponententest wird normalerweise als *Scripted-Keyword* umgesetzt und mit dem *Keyword*-Typ *unittest* gekennzeichnet. In diesem Beispiel wird die Komponente *PetTypeFormatter* getestet. Die Komponente ist in der Webanwendung dafür verantwortlich, aus einer 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 7.1: Komponententest *Test PetTypeFormatter*

Der Codeausschnitt 7.2 zeigt die Implementierung des *Scripted-Keywords*. Die Implementierung des Komponententests ist grundlegend gleich mit einem normalen *JUnit*-Test. Die großen Unterschiede sind, dass die Methode

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,
      RaydenReporter reporter) {
5     ClinicService service = new MockClinicService();
6     PetTypeFormatter formatter = new PetTypeFormatter(service);
7
8     try {
9      Assert.assertEquals("dog", formatter.parse("dog", null).getName());
10     Assert.assertEquals("cat", formatter.parse("cat", null).getName());
11     Assert.assertEquals("fish",formatter.parse("fish",null).getName());
12    } catch (ParseException e) {
13      throw new AssertionError(e);
14    }
15
16    try {
17      formatter.parse("hamster", null);
18      Assert.fail("No ParseExeption was thrown!");
19    } catch (ParseException e) {
20    }
21
22    try {
23      formatter.parse(null, null);
24      Assert.fail("No ParseExeption was thrown!");
25    } catch (ParseException e) {
26    }
27
28    return new KeywordResult(true);
29  }
30 }
```

Programm 7.2: Implementierung des *Test PetTypeFormatter Keywords*

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

7.3 Schnittstellentest

Bei einem Schnittstellentest werden öffentliche Schnittstellen wie ein *Restful*-Schnittstelle [Wik15b] getestet. Im Schnittstellentest 7.3 wird die *Restful*-Schnittstelle für Tierärzte getestet. 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 restfull service for veterinarian.'''
3
4   Verify Json("http://localhost:9966/petclinic/vets.json",
5               "./demodata/vets.json")
6 }
7
8 keyword Verify Json {
9   '''The keyword download 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 were equals, the
12     keyword finish successfully'''
13
14  parameter url
15  parameter file
16
17  implemented in java -> "petclinic.VerifyJsonKeyword"
18 }
```

Programm 7.3: Integrationstest *Test Veterinarians Restful Service*

Bei diesen Beispiel liefert die Schnittstelle das Ergebnis als einen JSON-Text. 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. Semantisch Gleich heißt bei einem JSON-Text, dass die enthaltenen Daten gleich sein müssen, aber nicht in welcher Reihenfolge diese serialisiert worden sind.

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. Die Implementierung des *Verify Json Keywords* zeigt das Codebeispiel 7.4. Das Codestück zeigt, dass zuerst über einen *HttpClient* der JSON-Text von einem Server abgefragt wird. Danach wird der JSON-Text mit einem *Parser* in einem Objektbaum transformiert. Dafür wird eine *Parser*-Implementierung aus der *Google-Guava*-Bibliothek verwendet. Der selbe Prozess wird auch mit der Demodaten-Datei durchlaufen. Am Ende gibt es zwei Objektbäume für die JSON-Texte. Für den semantischen Vergleich der beiden Bäume kann die Methode *equals()* der Klasse *JsonElement* verwendet werden. Diese Klasse stammt wiederum aus der *Google-Guava*-Bibliothek.

7.4 Abnahmetests

TODO

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

Programm 7.4: Implementierung des *Verify Json Keywords*

7.4.1 Abnametest *Suchen nach einen Tierbesitzer*

```
1 uatest Find a pet owner and check the details {
2   parameter in petOwner as string
3
4   Prepare Browser
5   @PetClinicWeb.Home :: Click
6   Find a specific Pet Owner(petOwner)
7   Check Owner Details ("Betty Davis", "638 Cardinal Ave.", "Sun Prairie
      ", "6085551749")
8   Cleanup Browser
9 }
10
11 keyword Find a specific Pet Owner {
12   parameter in petOwner as string
13
14   @PetClinicWeb.Find Owners :: Click
15   @PetClinicWeb.Find Owners Page.Title :: Verify Text("Find Owners")
16   @PetClinicWeb.Find Owners Page.Find :: Click
17   @PetClinicWeb.Find Owners Result Page.Title :: Verify Text("Owners")
18   @PetClinicWeb.Find Owners Result Page.Result:: Count
19   Verify(itemCount, 10)
20   @PetClinicWeb.Find Owners Result Page.Search:: Type Text (petOwner)
21   @PetClinicWeb.Find Owners Result Page.Result :: Count
22   Verify(itemCount, 2)
23   @PetClinicWeb.Find Owners Result Page.Result.Item :: Click
24 }
25
26 keyword Check Owner Details  {
27
28   parameter name as string
29   parameter address as string
30   parameter city as string
31   parameter telephone as string
32
33   @PetClinicWeb.Owner Detail Page.Title :: Verify Text("Owner
       Information")
34   @PetClinicWeb.Owner Detail Page.Name :: Verify Text(name)
35   @PetClinicWeb.Owner Detail Page.Address :: Verify Text(address)
36   @PetClinicWeb.Owner Detail Page.City :: Verify Text(city)
37   @PetClinicWeb.Owner Detail Page.Telephone :: Verify Text(telephone)
38 }
```

Programm 7.5: Codeauszug: *Suchen nach einen Tierbesitzer*

TODO

```
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 7.6: Codeauszug aus dem *Object-Repository* für den Testfall *Suchen nach einen Tierbesitzer*

7.4.2 Abnametest *Anlegen eines neuen Tierbesitzer*

```
1 uatest Add new pet owner {
2   parameter firstname
3   parameter lastname
4   parameter address
5   parameter city
6   parameter telephone
7
8   Prepare Browser
9
10  @PetClinicWeb.Find Owners :: Click
11  @PetClinicWeb.Find Owners Page.Add Owner :: Click
12  @PetClinicWeb.Edit Owner Page.Title :: Verify Text ("New Owner")
13  @PetClinicWeb.Edit Owner Page.First Name :: Type Text (firstname)
14  @PetClinicWeb.Edit Owner Page.Last Name :: Type Text (lastname)
15  @PetClinicWeb.Edit Owner Page.Address :: Type Text (address)
16  @PetClinicWeb.Edit Owner Page.City :: Type Text (city)
17  @PetClinicWeb.Edit Owner Page.Telephone :: Type Text (telephone)
18
19  Cleanup Browser
20 }
```

Programm 7.7: Codeauszug: *Anlegen eines neuen Tierbesitzer*

TODO

```
1 objectrepository PetClinic {
2
3   application PetClinicWeb {
4     location absolute "//body"
5
6     ...
7
8     page Find Owners Page{
9       control Title { location "//h2" }
10
11      textfield Search Lastname { location "//*[@id='lastName']" }
12
13      button Find {
14        location "//*[@id='search-owner-form']/fieldset/div[2]/button"
15      }
16
17      button Add Owner {  location "//a[text() = 'Add Owner']" }
18    }
19
20    page Edit Owner Page {
21      control Title { location "//h2" }
22
23      textfield First Name { location "//*[@id='firstName']" }
24
25      textfield Last Name { location "//*[@id='lastName']" }
26
27      textfield Address { location "//*[@id='address']" }
28
29      textfield City { location "//*[@id='city']" }
30
31      textfield Telephone{ location "//*[@id='telephone']" }
32
33      button Add Owner { location "//button[text() = 'Add Owner']" }
34    }
35
36    ...
37  }
38 }
```

Programm 7.8: Codeauszug aus dem *Object-Repository* für den Testfall *Anlegen eines neuen Tierbesitzer*

7.4.3 Selenium *Keywords*

```
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 Type Text  {
15   parameter in locator as location
16   parameter in text as string
17
18   implemented in java -> "petclinic.selenium.TypeTextKeyword"
19 }
20
21 keyword Close Browser  {
22   implemented in java -> "petclinic.selenium.CloseBrowserKeyword"
23 }
24
25 keyword Verify Text  {
26   parameter in locator as location
27   parameter in text as string
28
29   implemented in java -> "petclinic.selenium.VerifyTextKeyword"
30 }
31
32 keyword Count {
33   parameter in locator as location
34   parameter out itemCount as number
35   implemented in java -> "petclinic.selenium.CountKeyword"
36 }
```

Programm 7.9: Codeauszug aus der Selenium *Keyword*-Bibliothek

TODO

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

Programm 7.10: Implementierung des *Open Browser Keywords*

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

Programm 7.11: Implementierung des *Click Keywords*

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

Programm 7.12: Implementierung des *Verify Text Keywords*

7.5 Testdokumentation

TODO !!!

Kapitel 8

Zusammenfassung

TODO

Diskussion, Erfahrungen, weitere Arbeiten TODO !!!
Erweiterungen: Vernünftige JUNIT Unterstützung

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. 18).

[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. 15).

[Lau06] Pekka Laukkanen. „Data-Driven and Keyword-Driven Test Automation Frameworks“. masterthesis. Helsinki University of Technology, Aug. 2006 (siehe S. 17).

[Mes07] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007 (siehe S. 8).

Online-Quellen

[Bor15] Borland. 2015. URL: http://www.borland.com/silktest (besucht am 03. 05. 2015) (siehe S. 12).

[Ecl13] Eclipse. 2013. URL: http://eclipse.org/eclipse (besucht am 28. 04. 2015) (siehe S. 10).

[Ecl15a] Eclipse. 2015. URL: https://eclipse.org/org/foundation (besucht am 03. 05. 2015) (siehe S. 11).

[Ecl15b] Eclipse. 2015. URL: https://www.eclipse.org/modeling/emf (besucht am 03. 05. 2015) (siehe S. 11).

[Ecl15c] Eclipse. 2015. URL: http://eclipse.org/Xtext/ (besucht am 28. 04. 2015) (siehe S. 11, 25, 43).

[Fou15] Python Software Foundation. 2015. URL: https://www.python.org (besucht am 24. 05. 2015) (siehe S. 53).

[Fow13] Martin Fowler. 2013. URL: http://martinfowler.com/bliki/PageObject.html (besucht am 28. 04. 2015) (siehe S. 26).

[KE14] Beck Kent und Gamma Erich. 2014. URL: http://junit.org/ (besucht am 03. 05. 2015) (siehe S. 8).

[KH15] Pekka Klärck und Janne Härkönen. 2015. URL: http://robotframework.org/ (besucht am 17. 05. 2015) (siehe S. 18).

[Mic15] Microsoft. 2015. URL: https://msdn.microsoft.com/en-us/library/dd233052(v=vs.110).aspx (besucht am 24. 05. 2015) (siehe S. 53).

[Ora14] Oracle. 2014. URL: https://www.jcp.org/en/jsr/detail?id=223 (besucht am 28. 04. 2015) (siehe S. 20, 23).

[Sel15] Selenium. 2015. URL: http://www.seleniumhq.org (besucht am 03. 05. 2015) (siehe S. 11).

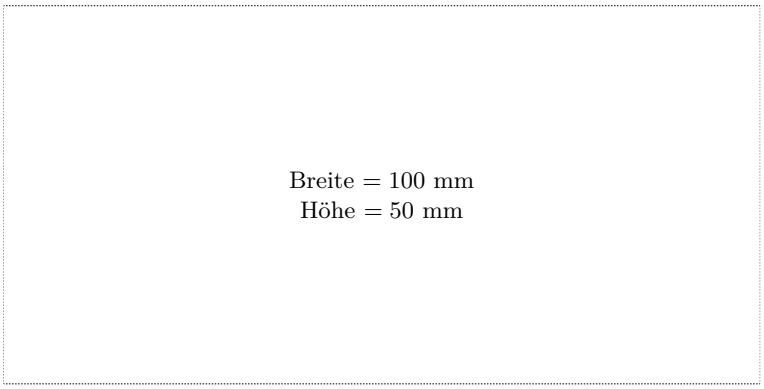
[W3C15] W3C. 2015. URL: http://www.w3.org/TR/2013/WD-webdriver-20130117 (besucht am 03. 05. 2015) (siehe S. 12).

[Wik15a] Wikipedia. 2015. URL: http://en.wikipedia.org/wiki/Stack_machine (besucht am 28. 04. 2015) (siehe S. 25).

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

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —