Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorthesis

Jan-Ole Giebel

Ressource-Management für eine Testautomationfarm

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Jan-Ole Giebel

Ressource-Management für eine Testautomationfarm

Bachelorthesis eingereicht im Rahmen der Bachelorprüfung
im Studiengang Informations- und Elektrotechnik
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Thomas Klinker
Zweitgutachter : Prof. Dr. Thomas Lehmann

Abgegeben am 05.04.2012

Jan-Ole Giebel

Thema der Bachelorthesis

Ressource-Management für eine Testautomationfarm

Stichworte

Ressourcenverwaltung, Testautomation, Webanwendung, Java, Selenium 2, SauceLabs, Selenium Grid 2

Kurzzusammenfassung

Diese Arbeit umfasst die Auswahl, Entwicklung und Implementierung eines Managementtools, um Ressourcenverwaltung innerhalb einer Testautomatisierungsumgebung zu gewährleisten. Es wird der aktuelle Markt für diese Anwendung analysiert und entsprechende Produkte werden bezüglich ihrer Eignung für den Einsatz in der spezifischen Umgebung analysiert und bewertet. Als Gegenstück zum Einkauf und Anpassung eines Softwareproduktes wird die Entwicklung einer eigenen, unternehmensspezifischen Lösung diskutiert und umgesetzt. Dabei wird sowohl auf architektonische Eigenschaften, als auch Design und Implementierung eingegangen.

Jan-Ole Giebel

Title of the paper

Resource management for a testautomation farm

Keywords

resource management, testautomation, webapplication, Java, Selenium 2, SauceLabs, Selenium Grid 2

Abstract

This thesis describes the development and the implementation of a management tool to ensure the distribution of test resources in a test automation environment. Common products will be inspected focused on their abilities to fit the predefined environment. As a counterpart to a common product, the development of an unique solution with respect to the individual business structure will be discussed.

Danksagung

An dieser Stelle möchte ich zunächst meinen Dank an Herrn Prof. Dr. Thomas Klinker und Prof. Dr. Thomas Lehmann aussprechen, die mir durch ihre Betreuung seitens der HAW und ihre Rollen als Prüfer die Möglichkeit gaben, diese Arbeit anzufertigen. Ebenfalls möchte ich mich bei meinen Betreuern Herrn Jörg Langnickel, Herrn Björn Kellermann und Herrn Dr. Andreas Pelzer seitens der Hamburg Süd bedanken, die mich zu jeder Zeit unterstützten und eine große Hilfe waren. Ein großer Dank auch an Frau Karen Krause für die tatkräftige Hilfe und stets guten Ratschläge.

Ebenfalls möchte ich besonders meiner Familie danken, ohne deren Unterstützung mir dieses Studium nicht möglich gewesen wäre.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listingverzeichnis

1. Einführung

Die Testautomation ist ein zentraler Bestandteil in modernen Softwareentwicklungsprozessen. Sie ermöglicht die einfache Realisierung von verschiedenen Testarten wie beispielsweise Regressiontests oder Stresstests. In dieser Arbeit wird die technische Machbarkeit erörtert, ein Testautomationssystem, welches mit mehreren Testclients arbeitet, mit einem Resourcemanagementserver zu versehen. Dadurch soll die Möglichkeit gegeben werden, dass die Verteilung von Testfällen auf einzelne Testclients gemäß bestimmter Kriterien automatisch erfolgt. Hierbei werden verschiedene Lösungsmöglichkeiten beleuchtet. Bei der technischen Umsetzung wird der Fokus auf Architektur, Design, Implementierung und Test gelegt.

In diesem Abschnitt wird ein Überblick über das technische und organisatorische Umfeld dieser Arbeit gegeben. Dabei wird auf die zu testende Anwendung und das Umfeld dieser Arbeit gegeben. Dabei wird auf die zu testende Anwendung (Application Under Test), das genutzte Testframework und das Testautomationssystem eingegangen. Daraus folgernd wird die Motivation für diese Arbeit dargestellt und die Anforderungen an die zu erarbeitende Lösung formuliert.

1.1. Application Under Test

GLobal Logistic Organisation Business Environment (GLOBE) ist für die Reederei Hamburg Süd aus Sicht der bestehenden IT-Landschaft ein wegweisendes Projekt. Das in GLOBE entwickelte Transport-Management-System ist ein komplexes System, das die operativen Prozesse des Unternehmens, die über viele heterogene Systeme abgebildet werden, zu einem Gesamtsystem zusammenfügt. Dies System wird in mehreren Releases entwickelt. Ein Ausfall der Funktionalität hat massive Auswirkungen auf die Produktivität und daher auch auf das Ansehen des Unternehmens, wodurch der Testprozess während der Entwicklung einen sehr hohen Stellenwert einnimmt.

1.2. Das Testframework AludraTest

AludraTest ist ein auf die AUT[1] zugeschnittenes Testframework, um eine Interaktion mit dem HMI[2], das in Form einer Weboberfläche implementiert ist, durchzuführen. Das Framework bietet die Möglichkeit, einen funktionalen Test zu erstellen, der automatisiert und wiederholt durchführbar ist.

Das Testframework besitzt Datenobjekte und Methoden, die bezüglich der AUT domänenspezifisch sind. Das bedeutet, es wird im Rahmen eines Tests ein domänenspezifisches Datenobjekt generiert und von einer Testoperation, welche die Domäne anspricht und die Attribute des Datenobjektes nutzt, ausgeführt. Die Schnittstelle, gegen die getestet wird, ist das Webinterface. Die domänenspezifischen Datenobjekte und Methoden sind von dieser Schnittstelle losgelöst. Um eine Verbindung zwischen Webinterface und den Datenobjekten und Methoden zu schaffen, werden die Methoden von schnittstellenspezifischen Klassen implementiert, die die Rolle des Bindeglieds zwischen der technischen Schnittstelle und der Domäne einnehmen.

1.3. Das Testautomationssystem

Das System besteht aus 20 virtuellen Clients, welche mit verschiedenen Kombinationen aus Betriebssystem und Webbrowser ausgerüstet sind. Jeder dieser Clients besitzt zudem einen Seleniumserverdienst.

Der Dienst stellt die Möglichkeit zur Verfügung, automatisiert Webanwendungen zu testen. Die Tests arbeiten dabei gegen die GUI einer Webanwendung. Die einzelnen Seleniumserver werden direkt über die jeweilige IP-Adresse des Clients angesprochen. Somit können die Tester direkt einen Server ansprechen und Tests starten. Eine Kontrolle, ob der Server fähig ist den Test auszuführen, gibt es nicht. Somit muss der Tester darauf achten, dass der Seleniumserver mit der vom Test geforderten Kombination aus Betriebssystem und Browser arbeitet. Durch den direkten Zugriff auf die Testclients ist es möglich, dass zwei oder mehr Tester den identischen Client nutzen, wodurch Konkurrenzsituationen zwischen den Tests entstehen können. Daher kann der Testablauf soweit gestört werden, dass die Testergebnisse inkorrekte Ergebnisse liefern.

[1] Application Under Test
[2] Human-Machine-Interface

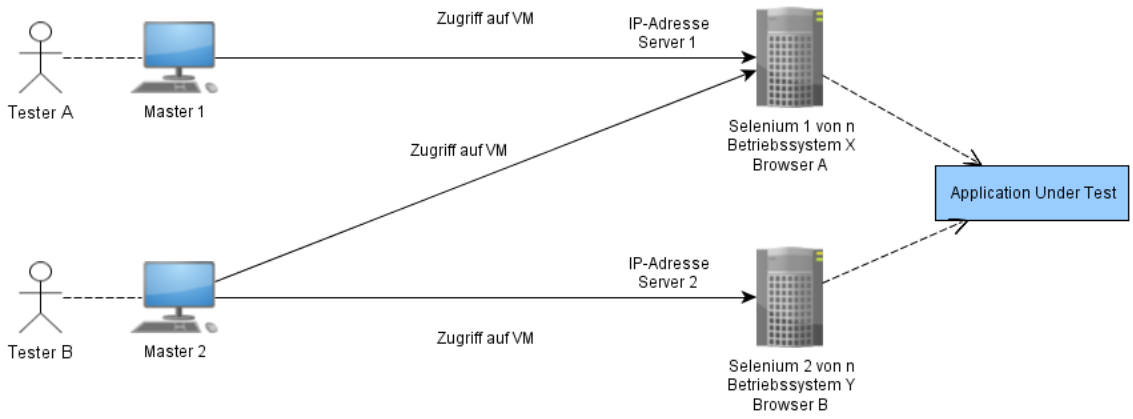

Abbildung 1.: *Ausschnitt - Setup der Testautomation (IST-Zustand)*

1.4. Motivation

Im momentanen Testprozess nimmt der Tester mehrere Rollen wahr. Zum einen ist er für seine Tests verantwortlich und muss zum anderen sicherstellen, dass die Testumgebungen, die von ihm benötigte Umgebung bereitstellt. Zwischen den Testern herrscht ein hoher Kommunikationsbedarf, da es keinen Automatismus gibt, der eine parallele Ausführung von mehreren Tests auf einer Umgebung kontrolliert steuert.

Um den Testprozess diesbezüglich zu optimieren, wird ein Dienst benötigt, der Testern basierend auf gestellten Anforderungen eine Testumgebung zuweist. Dabei hat der Dienst die Kenntnis über alle Testumgebungen und ihre Möglichkeiten zu besitzen. Die Organisation von parallelen Testausführungen ist dabei kein Bestandteil dieser Arbeit, da die Testausführung weiterhin in der Verantwortung des Frameworks AludraTest liegt. Die funktionalen und technischen Anforderungen werden in den Kapiteln 1.5.1, 1.5.2 und 3.1 behandelt.

1.5. Anforderungen

Im Folgenden werden die funktionalen und technischen Anforderungen seitens des Kunden an den Resourcemanagementserver dargestellt.

1.5.1. Funktionale Anforderungen an das Resourcemanagementtool

Der Kunde stellt folgende funktionale Anforderungen an das Resourcemanagementtool.

1 Das Resourcemanagementtool soll eine automatische Zuweisung von Ressourcen (hier Seleniumserver) ermöglichen.

1.1 Als Identifikationskriterien sind Kombinationen aus Browser und Betriebssystem zu verwenden

1.2 Zu der Anfrage soll die Angabe einer Untergrenze für die Anzahl der minimal benötigten Seleniumserver möglich sein.

1.3 Zu der Anfrage soll die Angabe einer Obergrenze für die Anzahl der maximal benötigten Seleniumserver möglich sein.

1.4 Es soll möglich sein, die zu testende Anwendung auch auf mobilen Endgeräten (IPhone und Android) automatisiert zu testen.

1.5 Als Testumgebung auf Windowsbasis wird vorausgesetzt, dass die Lösung die Clientkonfiguration des Kunden unterstützt.

2 Die Vergabe wird nach Prioritäten erfolgen, die von einem Administrator über eine geeignete Schnittstelle konfiguriert werden können.

3 Jeder User hat die Möglickeit einer globalen Mindestanzahl an zugesicherten Seleniumservern.

4 Für den positiven und für den negativen Ausgang der Anfrage sollen Antworten an den Test zurückgegeben werden.

4.1 Bei positivem Ergebnis wird eine eindeutige SessionID zurückgegeben, die ein eindeutiges Identifikationsmerkmal ist, um reservierte Ressourcen einer Anfrage zuzuordnen.

4.2 Bei positivem Ergebnis wird die Anzahl der reservierten Ressourcen zurückgegeben.

4.3 Bei negativem Ergebnis wird eine Nachricht über Gründe des Ausgangs zurückgegeben.

4.3.1 Als Fehlergrund wird zwischen „Kombination aus Browser und Betriebssystem nicht vorhanden“(Rückgabe N1) und „Anzahl der Mindestressourcen nicht verfügbar“(Rückgabe N2) unterschieden.

4.3.2 Bei Rückgabe N2 wird die Auslastung zurückgegeben.

4.3.3 Bei Rückgabe N2 wird eine mögliche Alternative zurückgegeben.

5 Der Test hat die Möglichkeit, eine Ressource an den Server zurückzugeben.

6 Der Test hat die Möglichkeit, eine Session durch Übermittlung der SessionID an den Server zu beenden.

7 Es ist möglich, dass sich die Anzahl der Ressourcen in einer Session ändert.

7.1 Die Anzahl der Ressourcen unterschreitet nie die angefragte Untergrenze.

7.2 Der Test erhält eine Nachricht bei einer Änderung

7.2.1 Die Nachricht enthält den Änderungsstatus (entweder hinzugefügt oder entfernt).

7.2.2 Die Nachricht enthält die aktualisierte Anzahl der Ressourcen innerhalb der Session.

8 Die Ressourcen werden bei Verfügbarkeit dem Test bekannt gegeben.

8.1 Für die Bekanntgabe einer Ressource wird eine Nachricht übermittelt.

8.2 Die Nachricht enthält die Adresse in Form einer URL.

9 Es soll möglich sein, dass mehrere Anfragen von unterschiedlichen Anwendern parallel abgearbeitet werden können.

10 Der User soll, für den Fall dass ein zugewiesener virtueller Rechner beim Test fehlschlägt, in der Lage sein, eine Ressource selbst verwalten zu können.

11 Das Resourcemanagementtool soll eine Schnittstelle für Reports bereitstellen, aus denen mögliche Optimierungen der Kombinationen aus Browser und Betriebssystem, bezogen auf die gestellten Anfragen, hergeleitet werden können.

1.5.2. Technische Anforderungen

Der Kunde stellt, basierend auf dem von den Tests verwendeten Framework AludraTest, folgende technische Anforderungen.

1 Als Entwicklungstechnologie soll Java verwendet werden.

2 Kompatibilität zu AludraTest soll bestehen.

3 Implementierung soll plattformunabhängig sein.

1.5.3. KO-Kriterien

Da die funktionalen Anforderungen aus Abschnitt 1.5.1 vom Kunden ohne Prioritäten gestellt wurden, sind KO-Kriterien definiert worden. Diese Kriterien sind in Tabelle 1 mit den jeweils zugrunde liegenden Anforderungen aufgeführt.

ID	Inhalt	Zugrunde liegende Anforderung
1	Unterstützung von mobilen Endgeräten	1.4
2	Testumgebungen auf Clientkonfiguration des Kunden anpassbar	1.5

Tabelle 1.: *KO-Kriterien*

2. Theoretische Vorbetrachtung

In diesem Kapitel wird die Theoretische Vorbetrachtung bezüglich des Themas Softwaretest und der einsetzbaren Technologien erfolgen.

2.1. Grundlagen Softwaretest

In der theoretischen Vorbetrachtung werden die Grundlagen des Testens kurz dargestellt. Hierbei wird zuerst erläutert, was ein Test ist, welche Arten von Tests es gibt und was die jeweiligen Ziele der einzelnen Testarten sind. In diesem Zusammenhang wird anhand des V-Modells von Boehm der Zusammenhang zwischen Entwicklungsstufen und Teststufen aufgezeigt. Der funktionale Fokus eines Tests auf die zu testende Anwendung lässt sich in drei Teile aufspalten: Blackbox-, Greybox- und Whitebox-Test. Diese Sichtweisen werden im dritten Teil behandelt.

Der Einsatzbereich der Testautomation wird ebenso in der theoretischen Vorbetrachtung bezüglich des Themas Test behandelt. Dabei wird erläutert, was unter einer Testautomation zu verstehen ist. Es werden Methoden erläutert wie im Rahmen der Testautomation Tests erstellt werden können. Darauf aufbauend werden schlussendlich die Vor- und Nachteile der Testautomation diskutiert.

2.1.1. Prozessmodell von Boehm (V-Modell)

Im Gegensatz zum Wasserfallmodell geht das V-Modell nach Boehm nicht davon aus, dass zu jeder Zeit ein komplettes Verständnis und eine vollständige Bekanntheit aller bisherigen Anforderungen vorhanden ist. Somit muss das V-Modell aus iterativen Entwicklungsphasen bestehen. Jeder dieser Entwicklungsstufen steht eine Teststufe gegenüber. Dies wird in Abbildung 2 dargestellt[3].

[3] [vgl. SL05, S.42]

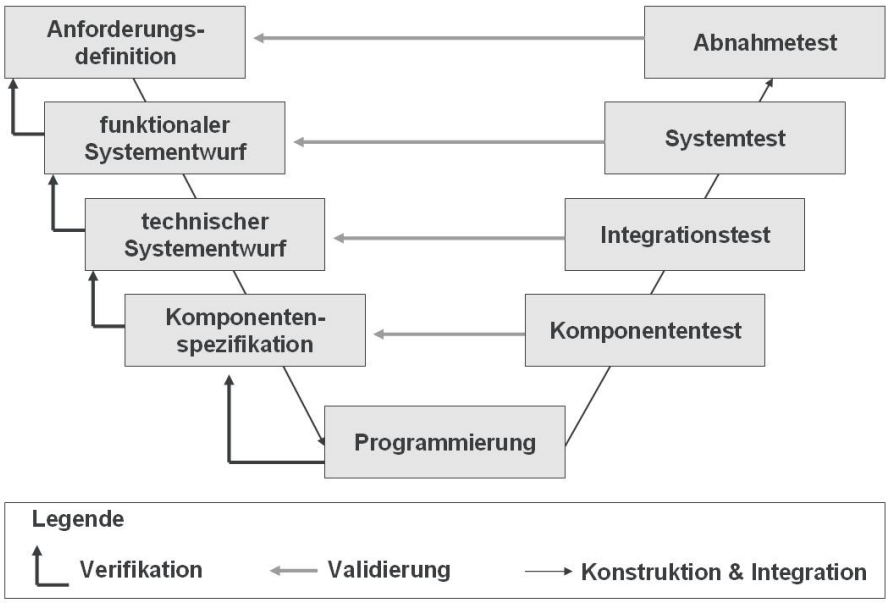

Abbildung 2.: *V-Modell nach Boehm*

Die einzelnen Entwicklungsphasen werden dabei wie folgt definiert.[4]

- Anforderungsdefinition: Mit Kunden abgestimmte Requirements an das Produkt
- Funktionaler Systementwurf: Projektion der Requirements auf neue Funktionalitäten des Systems
- Technischer Systementwurf: Technischer Entwurf des Systems in seiner produktiven Umgebung z.B. durch Schnittstellen oder Teilsysteme
- Komponentenspezifikation: Entwurf der einzelnen Teilsysteme
- Programmierung: Implementierung des Softwaresystems mit Hilfe einer Programmiersprache

Diesen Entwicklungsphasen stehen nach dem V-Modell somit die folgenden Teststufen gegenüber.

- Komponententest
- Integrationstest
- Systemtest
- Abnahmetest

[4] [vgl. SL05, Seite 42]

2.1.2. Die Teststufen im Softwaretest

Softwaretests sind seit langem ein wichtiger Aspekt in der Softwareentwicklung. Dies zeigen eindeutig die Negativbeispiele, die durch unzureichendes Testen von Software entstanden sind. Hierzu soll der Vorfall um den Therac-25 Linearbeschleuniger, der im Rahmen von Strahlentherapien eingesetzt wurde, als kurzes Beispiel dienen.[5]

Der Beschleuniger wurde genutzt, um β- und γ-Strahlung für eine Strahlentherapie zu erzeugen. Es kam bei Behandlungen mit dem Gerät innerhalb von drei Jahren zu sechs Unfällen, von denen einige tödlich endeten. Eine Untersuchung der Vorfälle ergab, dass der Fehler durch Raceconditions[6] zustande kam. Der Therac-25 war so konstruiert, dass der zentrale Computer für die Messung und Steuerungsaufgaben des Geräts zuständig war, aber auch für die Benutzerinteraktion. Da diese Prozesse im Multitasking erfolgten, musste eine Synchronisation sichergestellt werden, damit es zu keinen Raceconditions zwischen dem Prozess für Messung und Steuerung und dem Prozess für Benutzerinteraktion kommt. Diese Synchronisation war unzureichend, sodass eine Strahlenbelastung für den Patienten auftrat, die um den Faktor 20 bis 100 höher ausfiel, als für die Behandlung üblich. Die tödliche Grenze für körperweite Strahlenbelastung wurde um den Faktor 4 bis 20 überschritten. Innerhalb der Entwicklung war der Testverantwortliche für den fehlerhaften Softwareteil gleichzeitig auch der zuständige Entwickler.

Durch dieses sehr drastische Beispiel wird klar, dass Softwaretests ein zentraler Bestandteil in der Softwareentwicklung ist. Der Schaden durch fehlende und unzureichende Tests zeigte sich hier sogar als Gefahr für Leib und Leben. Der Schaden für die Entwicklerfirma wird nicht nur in hohen Maintenancekosten und Schadensersatzforderungen zu Buche schlagen, sondern auch mit einem erheblichen Imageschaden am Markt einhergehen.

Im Folgenden werden die im Kapitel 2.1.1 erwähnten Teststufen erläutert.

Komponententest

Mit einem Komponententest prüft der Tester kleinste Teile der Software auf die zu erwartende Funktionalität. Die Funktionalität kann von einer einzelnen Funktion, einer gesamten Klasse oder einem kleinen Verbund aus Klassen bereitgestellt werden. Somit stellt der Komponententest fest, ob Grundfunktionalitäten vorhanden sind. Da der Fokus dieser Testart nur auf dem komponenteninternen Verhalten liegt, können Fehler

[5] [vgl. Gar05], [vgl. Pre] und [vgl. Por]
[6] Möglichkeit für asynchrone Zugriffe

durch äussere Einflüsse nicht betrachtet werden. Die in diesem Test auffallenden Fehler sind meist in Einzelberechnungen oder Pfadverzweigungen zu finden.[7]

Integrationstest

Der Integrationstest findet nach dem Komponententest statt und stellt Fehler im Zusammenspiel mehrerer Komponenten fest. Somit liegt der Fokus des Integrationtests auch auf den Schnittstellen zwischen den Komponenten und daher sind gefundene Fehler nicht nur in den einzelnen Komponenten, sondern auch an den Schnittstellen zu suchen und zu erwarten.[8]

Systemtest

Für einen Systemtest wird das komplette System in eine Umgebung integriert, die der produktiven Umgebung möglichst entsprechen soll. Der Fokus liegt hierbei auf dem Auffinden fehlerhafter Interaktionen der Software mit ihrer Umwelt, zwischen den einzelnen Teilkomponenten im kompletten Verbund und ob die gestellten Requirements funktional erfüllt werden. Dieser Test wird aus Sicht des Kunden durchgeführt.[9]

Abnahmetest

Im Abnahme- oder auch Akzeptanztest wird das fertige System vom Kunden getestet. Dabei wird verifiziert, ob der Kunde seine Anforderungen als umgesetzt ansehen kann. Dies ist in den meisten Fällen der einzige Test, an dem der Kunde beteiligt oder verantwortlich ist und somit auch der einzige Test, den der Kunde unmittelbar nachvollziehen kann. Dieser Test ist bezogen auf seinen Umfang schwer zu spezifizieren. Der Umfang hängt stark von der Art der Software ab. So wird bei einer sicherheitsrelevanten Individualsoftware (z.B. Steuerungssysteme in medizinischen Anlagen) der Aufwand höher sein, als bei einer unkritischen Standardanwendung[10] (z.B. Office-Anwendung).

Im Rahmen der Abnahmetests existiert folgende Feinunterscheidung. [11]

- Vertragliche Akzeptanz: Die Annahme oder Ablehnung des Produktes durch den Kunden wird anhand eines vorher definierten Entwicklungsvertrages erfolgen.

[7] [vgl. SL05, Seite 44f]
[8] [vgl. SL05, Seite 52]
[9] [vgl. SL05, Seite 60f]
[10] [vgl. SL05, Seite 63ff]
[11] [vgl. SL05, Seite 63ff]

- Benutzerakzeptanz: Das Produkt wird durch den späteren Anwender auf Benutzbarkeit geprüft.
- Akzeptanz durch Systembetreiber: Der Systembetreiber testet, ob das Produkt in die vorhandene IT-Umgebung integrierbar ist. Dabei kann der Fokus unter anderem auf Backup-Routinen und Datensicherung gelegt werden.
- Feldtest: Die Software soll später in verschiedenen Umgebungen lauffähig sein, wobei die Kenntnis über einige der Umgebungen noch unvollständig ist. In diesem Test werden mögliche Umgebungseinflüsse simuliert, um experimentell das Verhalten der Software zu ergründen.

2.1.3. Testverfahren

Testverfahren lassen sich in drei Arten unterteilen, die in diesem Teil dargestellt werden.

- Blackbox-Test
- Whitebox-Test
- Greybox-Test

Blackbox-Test

Ein Blackbox-Test hat keine Kenntnisse über den inneren Aufbau des Testobjekts und entsteht folglich auf Grund einer Spezifikation. Es können nur äussere Schnittstellen angesprochen werden und so mit auch nur Prüfungen zwischen SOLL- und IST-Ausgaben der Schnittstellen gemacht werden.[12] Ein Blackbox-Test kann beispielsweise ein Login in ein System beschreiben. Dabei kennt der Test die Userdaten (Kennung und Passwort) und die Antwort des Systems auf einen Zustand nach dem Login (entweder Erfolg oder Fehlschlag). Um eine Software komplett mit einem Blackbox-Test zu untersuchen, müssen alle Permutationen von Eingabeparametern getestet werden.[13] Bei einer hohen Anzahl an Permutationen ist eine Testausführung bei einem Blackbox-Test daher zeitlich sehr Aufwendig und in der Praxis kaum ausführbar.

[12] [vgl. SL05, Seite 114ff]
[13] [vgl. SL05, Seite 114ff]

Whitebox-Test

Ein Whitebox-Test zeichnet sich durch die Kenntnis der Interna[14] eines Testobjekts aus und wird daher auf Basis des Quellcodes erstellt. Grund für diese Eigenschaft ist das Konzept, jeden Quellcodeteil mindestens einmal ausführen zu können. Somit löst der Whitebox-Test das Problem, dass ein Blackboxtest nie alle möglichen Zustände des Testobjektes testen kann.

Greybox-Test

Ein Greybox-Test verbindet Eigenschaften von White- und Blackbox-Tests. Der Greybox-Test entsteht vor dem Testobjekt auf Basis der Spezifikationen, wodurch er die Sicht von außen auf das Testobjekt hat. Jedoch sind die Interna soweit bekannt, wie es für den Test nötig ist (z.B. Komponentenstruktur)[15]. Diese Testart wird meist bei agiler Softwareentwicklung genutzt, da sie sich für Entwicklungsmethodiken wie Pairprogramming gut eignet.

2.1.4. Testarten

Jeder der unter Kapitel 2.1.2 genannten Tests hat einen eigenen Fokus. Der Fokus lässt sich dabei in verschiedene Testarten unterteilen, wobei jede Testart von einer Teststufe in unterschiedlicher Gewichtung genutzt wird.[16] Die Unterteilung, welche im Folgenden beschrieben wird, sieht dabei wie folgt aus.

- Funktionaler Test
- Nicht-funktionaler Test
- Strukturbezogener Test
- Änderungsbezogener Test

[14] [vgl. SL05, Seite 149ff]
[15] [vgl. QI]
[16] [vgl. SL05, Seite 71]

Funktionaler Test

Ein funktionaler Test umfasst alle Testmethoden, die von außen sichtbare Ein- und Ausgabeverhalten eines Testobjekts prüfen. Das erwartete Verhalten (SOLL-Verhalten) ist in einer funktionalen Anforderung definiert.[17] Eine funktionale Anforderung definiert sich dabei wie folgt.

> „Funktionale Anforderungen spezifizieren das Verhalten, welches das System oder Systemteile erbringen müssen. Sie beschreiben, was das (Teil-)System leisten soll. Ihre Umsetzung ist Voraussetzung dafür, dass das System überhaut einsetzbar ist. Merkmale der Funktionalität nach [ISO 9126] sind: Angemessenheit, Richtigkeit, Interoperabilität, Ordnungsmäßigkeit, Sicherheit.“[18]

Eine solche Anforderung könnte wie folgt aussehen.

> „Der Anwender kann ein Fahrzeugmodell aus dem jeweils aktuellen Modellprogramm zur Konfiguration auswählen.“[19]

Daraus lässt sich ein funktionaler Test folgenden Beispiels ableiten[20].

Aus dem aktuellen Modellprogramm wird vom Anweder ein Fahrzeugmodell zur Konfiguration ausgewählt.

Ein funktionaler Test wird somit primär in Systemtests und den darüber liegenden Tests zu finden sein.

Nicht-funktionaler Test

Ein nicht-funktionaler Test umfasst alle Methoden, die eine nicht-funktionale Anforderung (z.B. Performance) prüfen. Eine nicht funktionale Anforderung ist wie folgt definiert.

> „Nicht funktionale Anforderungen beschreiben Attribute des funktionalen Verhaltens, also wie gut bzw. mit welcher Qualität das (Teil-)System seine Funktionen einbringen soll. Ihre Umsetzung beeinflusst stark, wie zufrieden der Kunde bzw. der Anwender mit dem Produkt ist und wie gerne er es

[17] [vgl. SL05, Seite 72]
[18] [SL05, Seite 72]
[19] [SL05, Seite 72]
[20] [vgl. SL05, Seite 73]

einsetzt. Merkmale nach [*ISO* 9126] sind: Zuverlässigkeit, Benutzbarkeit, Effizienz."[21]

Nicht funktionale Tests können unter anderem in folgender Art auftreten.[22]

- Lasttest: Verhalten abhängig von der Systemlast
- Stabilitäts- oder Zuverlässigkeitstest, z.B. Ausfallzeiten pro Tag
- Performancetest: Lastabhängige Messung von Antwort- und Verarbeitungszeiten bei definierten Anwendungsfällen

Strukturbezogener Test

Strukturbezogene Tests prüfen die innere Struktur einer Software (siehe Whitebox-Test unter Kapitel 2.1.3). Dazu zählen unter anderem der interne Kontrollfluss und die Call-Hierarchie. Ziel dieses Tests ist es, möglichst alle Pfade zu erreichen, also einen Test zu bilden, der eine hohe Zweigabdeckung aufweist. Man findet strukturbezogene Tests hautsächlich in Komponenten- und Integrationstests.[23]

Änderungsbezogener Test

Änderungsbezogene Tests oder Regressiontests kommen bei Software zum Einsatz, die in Teil-Releases entwickelt werden oder wenn Bugfixes bestehender Funktionen entwickelt werden. Dabei muss das neue Softwarerelease alle alten Tests bestehen um nachzuweisen, dass keine bestehende Funktionalität, die vom neuen Softwareteil unberührt ist, beeinträchtigt oder zerstört wurde. Regressiontests sind somit in allen Teststufen, Testarten und Testverfahren vorhanden und eignen sich gut für die Testautomation, da einzelne Testfälle meist häufiger ausgeführt werden müssen.[24]
Ein Regressiontest kann man in vier Kategorien aufteilen.[25]

- Wiederholung fehlgeschlagener Tests
- Test aller geänderten Programmstellen
- Test aller neuen Programmstellen
- Vollständiger Systemtest

[21] [SL05, Seite 74]
[22] [vgl. SL05, Seite 74f]
[23] [vgl. SL05, Seite 76]
[24] [vgl. SL05, Seite 76f]
[25] [vgl. SL05, Seite 77]

Die ersten drei Tests bieten meist keine hinreichende Aussage über die Qualität der Software, da sie einzelne Softwareteile isoliert betrachten. So können kleine Änderungen, die in ihrem Bereich keine Auswirkungen haben, massive Seiteneffekte erzeugen, die wiederum nicht erkannt werden. Um solche Effekte zu vermeiden, müssen komplette Systemtests gefahren werden. Ein vollständiger Regressiontest ist aber meist sehr zeit- und damit auch kostenintensiv. Es muss also eine strategische Abwägung getroffen werden, welche Tests Risiken durch fehlerhafte Software mindern ohne zeitintensiv zu werden.[26] Die Abschätzbarkeit von möglichen Seiteneffekten hängt dabei von der Dokumentation des Systems ab. Bei grob oder gar schlecht dokumentierter Software ist das Abschätzen nahezu unmöglich.

2.1.5. Testautomation

Nach Dustin ist eine Testautomation wie folgt definiert.

> „The application and implementation of software technology throughout the entire software testing lifecycle (STL) with the goal of improving STL efficiencies and effectiveness.“[27]

Eine Testautomation ist somit definiert als ein Softwaresystem mit dem Ziel, Effektivität und Effizienz des gesamten Softwaretest-Lebenszyklusses zu steigern. In den folgenden Abschnitten wird der Einsatzbereich für eine Testautomation thematisiert und dargestellt, welche Arten der Erstellung von Testfällen es gibt. Abschließend wird eine Diskussion über die Vor- und Nachteile der Testautomation erfolgen.

Einsatzbereiche und Ziele der Testautomation

Die Testautomation eignet sich überall dort, wo aufwändige Testaktivitäten wiederholt durchgeführt werden. Wird zum Beispiel eine Software in vielen aufeinander aufbauenden Releases entwickelt und an den Kunden ausgeliefert, so muss bei jedem Einzelrelease die bisherige Funktionalität der Software gesichert werden, was nur durch ausgiebiges Testen nachgewiesen werden kann (Regressiontest). Dies ist bei komplexer Software nicht trivial und somit bei manueller Ausführung zeit- und kostenintensiv.

Folglich manifestiert sich die Zielsetzung der Effektivitäts- und Effizienzsteigerung in folgende Punkten.[28]

- Zeit- und somit Kostenreduktion durch Reduzierung des manuellen Testens

[26] [vgl. SL05, Seite 77f]
[27] [Du09, Seite 4]
[28] [vgl. Du09, Seite 23]

- Verbesserung der Qualität des Produktes
- Durch manuelle Tests schwer nachzuweisende Fehler (z.B. Memory Leak) und aufwändig durchzuführende Tests (z.B. Performancetests) werden durch die Automatisierung durchgeführt.

Methoden der Testerstellungen

Es werden zwei Methoden der Testerstellung behandelt. Zum einen Capture and Replay, das eine Aufzeichnung von Arbeitsschritten und deren Abspielen beschreibt, und zum anderen das Testautomationsskript, bei dem ein automatisierter Testablauf komplett programmiert wird.

Beim Capture and Replay wird ein Testablauf manuell durchgeführt und aufgenommen (Capture), damit man ihn danach beliebig oft abspielen kann (Replay), um so jedes Mal einen identischen Testablauf zu gewährleisten. Dieser recht einfach scheinende Prozess birgt jedoch Risiken. Zum einen ist es nur selten möglich, ein Capture direkt zu nutzen, so dass manuelle Änderungen am aufgenommenen Testscript erfolgen müssen. Diese Änderungen sind Quellen für Fehler. Die so erstellten Testscripts besitzen selten Validierungsmöglichkeiten, um sicher zu stellen, dass auf einer korrekten Umgebung getestet wird. Es kann also selten erkannt werden, ob beispielweise alle GUI-Elemente oder Testdaten korrekt vorhanden sind. Auch wenn die Skripte einen definierten und realen Use Case abbilden, muss meist Aufwand für das Refactoring aufgebracht werden. Dies ist in der Tatsache begründet, dass die Skripte durch Werkzeuge entstehen, die beispielsweise keine Kenntnis über spezielle Anforderungen eingesetzter Testframeworks besitzen. [29]

Die Methode des Capture and Replays zeigt sich somit als einfache, aber nicht änderungsfreie Methode zur Testerstellung.

Bei der kompletten Erstellung des Testskriptes wird der Test vom Tester programmiert. Hier ist es möglich, Validierung von Umgebung und Daten zu realisieren und Modularisierung in die Tests zu implementieren. Dadurch vermindert sich der Aufwand für Maintenance, wodurch die selbst erstellten Testskripte für den langfristigen Einsatz geeignet sind.

Man muss aber bedenken, dass der initiale Aufwand für die Erstellung sehr hoch ist, da der Testprozess nicht automatisch aufgenommen werden kann, sondern entwickelt werden muss. Wenn sich die Testumgebung oder die zu testende Anwendung an den Schnittstellen kaum ändert, ist der Effizienzgewinn geringer als durch das

[29] [vgl. Du09, Seite 80f]

Capture and Replay, denn der initiale Aufwand wird nicht durch Ersparnis bei Änderungsarbeiten ausgeglichen.[30]

Vor- und Nachteile einer Testautomation

Es lassen sich also folgende Vorteile aus der Testautomation ableiten.[31]

- Aufwändige, sich wiederholende Tests können durch den Tester trivial durchgeführt werden
- Variationen von Daten und Testabläufen lassen sich schneller realisieren im Gegensatz zum manuellen Test (Ausnahme Capture & Replay)
- Testautomation kann Tests ausserhalb der Arbeitszeit der Tester durchführen
- Tester können sich auf komplexen Aufgaben konzentrieren
- Kein Verschleiß des Testers durch monotone und triviale Tests
- Testablauf ist duplizierbar
- Mehrere automatisierte Tests können ganze Anwender simulieren
- Technische Tests, die manuell durchgeführt komplex sind, werden trivial (z.B. Memory Leaks)

Der größte Nachteil, der den zuvor erläuterten Vorteilen entgegen steht, ist der Aufwand bei der initialen Erstellung der Tests. Somit ist eine Testautomation in der Einführung kostenintensiv.[32]

2.2. Technologien

In diesem Teil werden mögliche Technologien, die im Rahmen der Ressourcenverwaltung genutzt werden können, beschrieben. Dabei wird auf die Kommunikation über `Remote Method Invocation (RMI)` und die Auslieferung von Javabibliotheken durch das `Java Network Launching Protocol (JNLP)` eingegangen und ebenso das Testframework `Selenium 2` beschrieben.

[30] [vgl. Mol09, Seite 38]
[31] [vgl. Du09, Seite 40]
[32] [vgl. Du09, Seite 76f]

2.2.1. Serialisierung

Serialisierung ist eine Technologie in Java, um Objekte standardisiert über `ObjectOutputStreams` auszutauschen. Dabei wird ein Objekt anhand der Klasse zum Senden serialisiert und bei Empfang deserialisiert. Dabei muss auf beiden Seiten die gleiche Version einer Klasse vorhanden sein. Serialisierung wird unter anderem bei RMI eingesetzt, um die Rückgabe und Übergabe von Daten zu realisieren.

Versionierung von Klassen bei Serialisierung

Jedes serialisierbare Objekt muss von einem Typ sein, dessen Klasse das Interface `Serializable` implementiert. Jede Klasse, die dieses Interface implementiert, besitzt das statische Attribut `serialVersionUID` vom Typ **long**. Dieses Attribut wird zur Versionierung innerhalb der Serialisierung genutzt. Wird ein Objekt deserialisiert, wird dessen `serialVersionUID` mit der der Klasse verglichen, die als Bauplan für die Deserialisierung genutzt wird. Stimmen die Versionsnummern nicht überein, so wird eine entsprechende Exception geworfen, die in der Implementierung abgehandelt werden sollte. Der Standardwert für das Attribut ist *1L*.

Vergleich auf Identität

Ein wichtiger Aspekt bei der Serialisierung ist, dass nach dem Deserialisieren keine Referenzidentität mit dem Objekt vor der Serialisierung besteht. Somit ist jedes deserialisierte Objekt eine Deep-Copy von dem Objekt vor der Serialisierung. Somit ist es empfehlenswert, die Methode `equals(Object o)` zu überschreiben, sobald das Interface `Serializable` genutzt wird.

Transiente Attribute

In manchen Fällen ist gefordert, dass nicht alle Daten eines Objektes serialisiert werden. Dies ist bei sensiblen Daten wie Passwörtern der Fall. Um dies zu ermöglichen, müssen entsprechende Attribute mit `transient` markiert werden.[33]

[33] [Ull12, Kapitel 12.14.4]

Referenzen und Collections

Oft ist es der Fall, dass nicht ein einzelnes Objekt serialisiert werden soll, sondern z.B. eine Liste. Es ist offenkundig, dass die Liste selber serialisierbar sein muss. Zusätzlich ist jedoch sicherzustellen, dass die enthaltenden Objekte serialisierbar sind. Ist dies nicht der Fall, wird eine entsprechende Exception zu Laufzeit geworfen. Attribute innerhalb von Objekten, die eine Referenz sind, müssen für eine Serialisierung auch das Interface `Serializable` implementieren.

2.2.2. Remote Method Invocation - RMI

RMI ist eine Client-Server-Kommunikationstechnologie, die seit Java 1.1 als Standardbibliothek vorhanden ist und seit Java 5.0 direkt [34] und ohne zusätzliche Spezialcompiler genutzt werden kann[35]. Es wird in diesem Teil zuerst auf die grundsätzliche Funktion von RMI eingegangen. Danach wird erläutert, was für eine Kommunikation als Mindestanforderung implementiert werden muss und wo die Besonderheiten der RMI-Kommunikation liegen.

Grundsätzliche Funktion

RMI basiert auf der Idee des `Remote Procedure Calls (RPC)`. Dabei wird eine Methode eines Objekts, das ausserhalb der eigenen Umgebung lokalisiert ist, aufgerufen. Im Fall von Java kann dies ein Objekt sein, das sich auf einem Rechner in einem anderen Netzwerk befindet oder nur in einer anderen Java Runtime Enviroment beheimatet ist. Bei dem Aufruf einer entfernten Methode erlaubt es RMI unter bestimmten Voraussetzungen auch, Objekte zu übergeben. Die Kommunikationsrichtung ist bidirektional, da die Methoden des Servers Rückgabewerte liefern dürfen. Die Aufrufe hingegen sind unidirektional, da alle Aufrufe vom Client ausgehen. Was für eine Kommunikation über RMI benötigt wird, wird im Folgenden erläutert.

Mindestanforderung von RMI

RMI ist, wie bereits erläutert, eine Server-Client-Technologie.

Als Beispiel für den Aufbau einer RMI-Architektur sollen mehrere auf See ausgesetzte Messbojen ihre Messungen zur Verarbeitung versenden. Eine Messboje soll immer eine möglichst kurze Distanz zu einer Messstation innerhalb eines Messnetzes

[34] [vgl. Ull12, Kapitel 19.3.3]
[35] [vgl. Ull12, Kaitel 19.1.2]

aufweisen. Es weiss also weder Boje noch Empfänger, wer genau an der Kommunikation beteiligt ist. Man benötigt also eine dritte Instanz, die sich um die Lokalisierung kümmert. Das könnte einer der Kommunikationssatelliten realisieren, indem Standortdaten (z.B. GPS) der Boje angefordert werden und sich alle möglichen Empfänger bei dem Satelliten registriert haben. Der Satellit könnte als Vermittlungsstelle somit den nächsten Empfänger ermitteln und der Boje die Adresse zurückgeben.

RMI basiert auch auf einem Konzept mit drei Instanzen. Neben dem Server und dem Client gibt es eine Registry. [36] Der Aufbau einer RMI-Verbindung benötigt folgende Schritte.

1 Holen oder erstellen einer Registry-Instanz

2 Binden einer Serverinstanz an einen Identifikator bei der Registry

3 Client fragt die Registry nach einem Server-Objekt über einen Identifikator

4 Wenn Identifikator bekannt, erhält Client ein Serverobjekt

Im Gegensatz zu einer direkten Verbindung zwischen Server und Client entsteht hier ein hoher Kommunikationsaufwand für die Initialisierung einer Kommunikation. Ersichtlich ist aber der Vorteil, dass der Server wie jedes andere Objekt behandelt werden kann. Dies wird weitere Implementierungen sehr intuitiv machen und somit einen einfach zu lesenden Code ermöglichen. Es gibt jedoch einige weitere Besonderheiten im Umgang mit RMI, die im folgenden Unterkapitel behandelt werden.

Besonderheiten von RMI

Es existieren bei der Kommunikation einige Besonderheiten. Dabei wird auf folgende Punkte eingegangen.

- Freigabe von Objekten
- Serverobjekt als Rückgabewert der Registry
- Exceptionhandling
- Objektreferenzen in Parameterliste

Alle Objekte, die über RMI erreichbar sein sollen, müssen auf der Registry registriert sein. Um ihnen einen Port zuzuweisen, muss ein Export mittels `UnicastRemoteobject` erfolgen. Hierbei fordert der Export, dass alle Methoden eine `RemoteException` werfen können[37]. Ist dies nicht der Fall, wird zur Laufzeit

[36] [vgl. Ull12, Kapitel 19.2.1]
[37] [vgl. Ou]

eine `ExportException` geworfen[38].

Wie erwähnt, bekommt der Client ein Serverobjekt von der Registry. Da die Registry für jede RMI-Kommunikation verwendet werden soll, ungeachtet der teilnehmenden Objekte, kann sie keine Kenntnis über den Typ des zurückzugebenden Serverobjekts haben. Folglich gibt sie immer ein Objekt des identischen Typs (hier `Remote`) zurück, die durch einen Cast in den gewünschten Typ gewandelt werden muss. Daher muss der Client wissen, was für einen Typ von Server er erwartet und dessen Datentyp kennen.[39] Hier ist es empfehlenswert mit Interfaces zu arbeiten, da somit keine Abhängigkeit von der konkreten Implementierung des Servers besteht. Es ist offenkundig, dass der Server dafür jedoch auch das entsprechende Interface implementieren muss.[40] Das Interface muss zudem von `Remote` erben.

```
1 import java.rmi.*;
2 public interface RMIServerInterface extends Remote {
3         public void takeMessage(RMIMessage message) throws RemoteException;
4         public RMIMessage giveMessage() throws RemoteException;
5 }
```

Listing 1: *Server-Interface eines RMI-Aufbaus*

```
1 import java.rmi.*;
2 public class RMIServer extends UnicastRemoteObject implements RMIServerInterface{
3         private static final long serialVersionUID = 1L;
4         private RMIMessage message = null;
5
6         protected RMIServer() throws RemoteException {
7                 super();
8         }
9
10        public void takeMessage(RMIMessage message) throws RemoteException {
11                this.message=message;
12        }
13
14        public RMIMessage giveMessage() throws RemoteException {
15                return message;
16        }
17 }
```

Listing 2: *Server eines RMI-Aufbaus*

[38] [vgl. Ou10a]
[39] [vgl. Ull12, Kapitel 19.4]
[40] [vgl. Ull12, Kapitel 19.3.1]

Die Kommunikation zwischen entfernten Stellen ist immer störanfällig. Es können Verbindungen abbrechen, durch Tests nicht erkannte Fehlerzustände auftreten oder Daten korrumpiert werden, um einige Beispiele zu nennen. Um die Effekte solcher Störungen kontrolliert zu handhaben, benötigt es bei der Kommunikation mittels RMI ein entsprechendes Exceptionhandling. Die RMI-Bibliothek von Java bietet vordefinierte Exceptions.[41] Der Entwickler muss aber die nötige Disziplin aufbringen die Exceptions sinnvoll zu behandeln, damit dem Anwender ein sauberer Ablauf des Programms garantiert ist. Zu beachten ist, dass ein Stack Trace über die Methode `getStackTrace()` nicht mehr möglich ist. Der Trace ist innerhalb der Exception als transient markiert und wird somit nicht serialisiert.[42]

Bei einem Methodenaufruf ist es meist notwendig, bestimmte Informationen und Daten als Parameter zu übergeben. Diese Parameter können sowohl primitive Daten, als auch komplexe Daten in Form von Objekten sein. Objekte werden in Java als Referenz übergeben und sind somit die innerhalb der JRE gültigen Adressen. Bei der Kommunikation über RMI werden aber die Grenzen der eigenen JRE überschritten und somit werden Referenzen ungültig. RMI setzt daher bei der Übergabe von Objekten voraus, dass die Klasse des Objektes das Interface `Serializable` implementiert und somit die Objekte serialisierbar sind.[43]

```
1  import java.io.Serializable;
2  public class RMIMessage implements Serializable{
3
4          private static final long serialVersionUID = 1L;
5          private String message;
6
7          public RMIMessage(String message) {
8                  this.message = message;
9          }
10
11         public String getMessage() {
12                 return this.message;
13         }
14 }
```

Listing 3: *Übertragbares Objekt in einem RMI-Aufbau*

[41] [vgl. Ou10c]
[42] [vgl. Suy00]
[43] [vgl. Ull12, Kapitel 19.5]

Die Serialisierung ist nicht trivial, da Objektreferenzen innerhalb des zu übertragenden Objektes auch serialisierbar sein müssen, damit sie nicht verloren gehen. Besonders beim Einsatz externer Bibliotheken kann es sein, dass das zu übertragende Objekt nicht serialisierbar ist und man als Folge Verantwortlichkeiten neuorganisieren muss. Ebenso ist das Deserialisieren nicht problemfrei. Die Deserialisierung ist ein zeitintensiver Vorgang, da das Objekt in Form des Typ `Object` vorliegt und folglich in seinen Zieltyp gecastet werden muss. Auch hier ist darauf zu achten, dass idealerweise nur Objekte eines Typs übergeben werden, der durch ein Interface definiert ist. Die konkrete Implementierung kann bei den RMI-Teilnehmern untereinander abweichen.

Der RMI-Client kann über zwei Arten die Registry ansprechen. Die erste wäre die Klasse `Naming`, die Funktionalitäten zum Zugriff auf eine Registry bietet. Die zweite Zugriffsart ist über ein Objekt vom Typ `Registry`, bei dem mittels des Konstruktors die Adresse der Registry mitgegeben werden muss. Listing 4 zeigt einen RMI-Client im Zusammenspiel mit der Klasse `Naming`.

```
1  import java.net.MalformedURLException;
2  import java.rmi.*;
3  public class RMIClient {
4          private RMIServerInterface server;
5          public RMIClient(String server) {
6                  try {
7                          this.server = (RMIServerInterface)Naming.lookup("//"+server+
                               "/RMIServer");
8                  } catch (MalformedURLException e) {
9                          e.printStackTrace();
10                 } catch (RemoteException e) {
11                         e.printStackTrace();
12                 } catch (NotBoundException e) {
13                         e.printStackTrace();
14                 }
15         }
16
17         public void sendMessageToServer(RMIMessage message) throws RemoteException{
18                 server.takeMessage(message);
19         }
20
21         public RMIMessage getMessageFromServer() throws RemoteException{
22                 return server.giveMessage();
23         }
24 }
```

Listing 4: *RMI-Client im Zusammenspiel mit der Klasse* `Naming`

2.2.3. Java Network Launching Protocol (JNLP)

JNLP ist eine Technologie, um Applets oder Anwendungen mit Java Web Starter auf einen Client auszuliefern. Dabei wird eine jnlp-Datei erzeugt, die in XML-Notation das Deployment beschreibt. Im Folgenden soll das Beispiel[44] in Listing 5 das Grundkonzept der JNLP-Datei beleuchten.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <jnlp spec="1.0+" codebase="" href="">
3      <information>
4          <title>Dynamic Tree Demo</title>
5          <vendor>Dynamic Team</vendor>
6          <icon href="sometree-icon.jpg"/>
7          <offline-allowed/>
8      </information>
9      <resources>
10         <!-- Application Resources -->
11         <j2se version="1.6+" href=
12            "http://java.sun.com/products/autodl/j2se"/>
13         <jar href="DynamicTreeDemo.jar"
14              main="true" />
15
16     </resources>
17     <application-desc
18           name="Dynamic Tree Demo Application"
19           main-class="webstartComponentArch.DynamicTreeApplication"
20           width="300"
21           height="300">
22      </application-desc>
23      <update check="background"/>
24 </jnlp
```

Listing 5: *Beispiel für eine JNLP-Datei*

In Zeile 2 wird spezifiziert, welche Version von JNLP genutzt werden soll. In diesem Fall ist es die Version 1.0 oder höher, da das Pluszeichen ein Wildcard für eine Mindestanforderung zu interpretieren ist. Durch `codebase` wird ein Stammverzeichnis als URL angegeben, von der alle folgenden `href`-Verzweigungen mit relativem Inhalt als Basis ausgehen. Das in dieser Zeile vorkommende `href`-Attribut kann genutzt werden, um auf die eigene JNLP-Datei als URL zu zeigen.

[44] [vgl. Ou12b]

Im Tag `information` werden die generellen Informationen definiert. Darunter fallen in dem Beispiel der Titel und der Anbieter der Anwendung, sowie ein Bild, dass der Anwendung als Desktopicon, Java Application Icon und Startbild zugeordnet ist. Zusätzlich ist hier durch `offline-allowed` angegeben, dass die Anwendung offline gestartet werden kann. Dennoch wird die Anwendung online nach Updates suchen.

Im Tag `resources` werden die Ressourcen der Anwendung beschrieben. Ähnlich wie beim Attribut `sec` im `jnlp`-Tag wird hier im `j2se`-Tag beim Attribut `version` mittels Wildcard vorgegeben, dass die Applikation mindestens die Java-Version 1.6 benötigt. Mit der folgenden Referenz wird ein Verweis auf den entsprechenden Download dieser Version gesetzt. Im Tag `jar` wird über das Attribut `href` auf das zu benutzende Jar-Paket verwiesen. Mit dem Attribut `main` wird angegeben, ob in dem Paket eine Klasse mit der Main-Methode zu finden ist.

Im Tag `application-desc` wird die Anwendung selbst beschrieben und definiert, dass diese JNLP-Datei zu einer Anwendung gehört. Das Attribut `name` definiert dabei den Namen und `main-class` gibt den Pfad zu der Klasse innerhalb des vorher definierten Pakets an, welche die Main-Methode beinhaltet. Mit den Attributen `width` und `height` wird die Größe der GUI festgelegt.

Der Tag `update` definiert das Verhalten beim Auftreten eines Updatevorgangs der Anwendung. Mit der Zuweisung `check="background"` wird definiert, dass das Update im Hintergrund laufen soll und somit der Anwender mit der älteren Version parallel weiterarbeiten kann.

Für eine Auflistung aller XML-Elemente im Rahmen von JNLP sei auf die Dokumentation von Oracle verwiesen.[45]

2.2.4. Selenium 2

Selenium 2 ist ein Testframework, um die Oberfläche von Webseiten zu testen. Dabei wird auf dem entsprechenden System, auf dem Selenium läuft, im Browser ein Zustandsautomat durchlaufen, dessen Ausgaben durch z.B. JUnit ausgewertet werden können. Im Folgenden wird die Ansprache der Websites und des Browsers mittels Selenium erläutert.

[45] [siehe Ou12b]

Ansprache des Webbrowsers

Websites können mit verschiedenen Browsern betrachtet werden, wobei jeder Browser eine eigene API zur Steuerung bietet. Um die entsprechende Flexibilität zu liefern, wurde in der Vorgängerversion (Selenium RC) die Browseransprache über ein einziges Objekt gelöst, dass den entsprechenden Browser als Parameter bekommen hat. Selenium 2 stellt nun für verschiedene Browser Treiberobjekte zur Verfügung, die die Steuerung des Browsers kapseln. Um die einzelnen Elemente der Weboberfläche zu identifizieren und somit eine Ansprache zu ermöglichen, stellt Selenium eine Identifikatorschnittstelle zur Verfügung. Die Schnittstelle kann die Elemente über deren Parameter wie zum Beispiel `Name` oder `Id` ansprechen, aber auch über deren Inhalt (z.B. Text eines Links) oder über die technische Abfragesprache XPath identifizieren.[46]

Technischer Aufbau der Treiberobjekte

Selenium 2 stellt über die Treiberobjekte Schnittstellen zur Browsersteuerung bereit, wobei durch das Interface `WebDriver` die grundlegenden Funktionalitäten für alle Treiberobjekte vorgegeben werden. Das Klassendiagramm aus Abbildung 3 stellt den Zusammenhang zwischen den WebDriver-Klassen dar, wobei aus Gründen der Übersicht hier auf die Darstellung der Methoden verzichtet wurde. Für eine Auflistung der zugänglichen Methoden, sei hier auf die WebDriver-API von Selenium verwiesen[47].

Es zeigt sich, dass das Interface `WebDriver` nur von zwei Klassen direkt implementiert wird. Die Klasse `RemoteWebDriver` erweitert das Interface dabei um eine Schnittstelle, die es ermöglicht, entfernte Seleniumserver anzusprechen. Die Klasse `HtmlUnitDriver` stellt einen WebDriver dar, der auf hohe Geschwindigkeit im Testablauf zielt. Dies ist möglich, da dieser WebDriver eine reine Java-Implementierung ist und somit keinen Browser startet. Neben dem `RemoteWebDriver` und dem `HtmlUnitDriver` sind folgende Treiberobjekte instanzierbar.

- FirefoxDriver
- InternetExlorerDriver
- ChromeDriver
- AndroidDriver
- IPhoneDriver
- IPhoneSimulatorDriver

[46] [XPath-Tutorial der w3school: Dat12]
[47] [vgl. Web11]

- OperaDriver

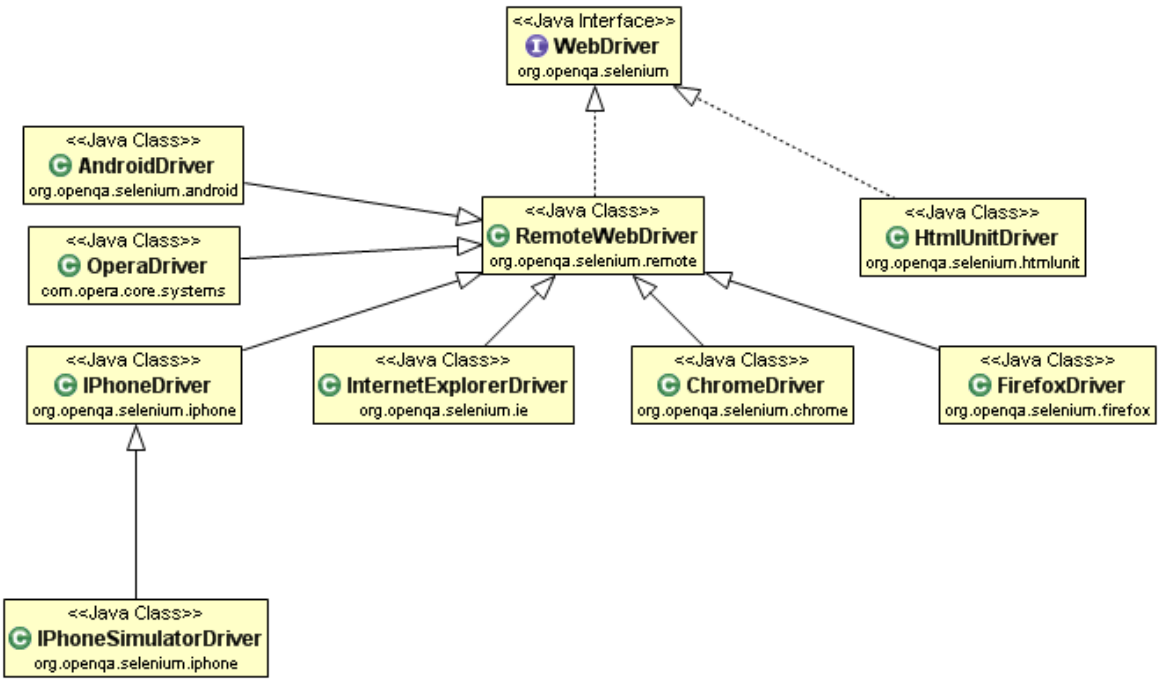

Abbildung 3.: *Klassendiagram der Webdriver ohne deren Methoden*

Da diese Treiberobjekte alle von `RemoteWebDriver` ableiten, ist es mit allen möglich, einen entfernten Seleniumserver anzusprechen[48].

[48] [vgl. Tol11]

3. Analyse

In diesem Teil werden die funktionalen Anforderungen aus Kapitel 1.5.1 und die technischen Anforderungen aus Kapitel 1.5.2 untersucht. Dabei wird der Blick auf eine Analyse ausgewählter, am Markt präsenter Lösungen gesetzt, um darauf aufbauend die Realisierbarkeit der Anforderungen des Kunden zu diskutieren. Dabei wird in der Marktanalyse der kostenpflichtige Dienst SauceLab und das freie Verwaltungstool Selenium Grid untersucht und mit einer Eigenimplementierung verglichen. Abschließend wird eine Empfehlung bezüglich einer Lösung gegeben.

3.1. Analyse der Anforderungen

Im Folgenden werden die unter Abschnitt 1.5.1 und 1.5.2 seitens des Kunden definierten Anforderungen analysiert und für die Erarbeitung einer Lösung spezifiziert. Dabei wird auf die technische Machbarkeit im Allgemeinen und im Rahmen dieser Arbeit eingegangen.

3.1.1. Realisierbarkeit der funktionalen Anforderungen

In diesem Abschnitt werden die funktionalen Anforderungen untersucht, die auf Grund ihres Inhaltes besonders betrachtet werden müssen.

Die in der Anforderung 1.1 erwartete Anfrage nach Kombinationen aus Browser und Betriebsystem muss im Zusammenhang mit der Anforderung 1.5 angepasst werden. Es existiert beim Kunden eine Clientkonfiguration basierend auf dem Betriebssystem Linux. Hierbei wird aber keine grafische Oberfläche wie Gnome oder KDE installliert, sondern ein terminalbasierter Client ausgeliefert. Da der einzige in Selenium vorhandene WebDriver ohne grafische Oberfläche (`HtmlUnitDriver`) nicht von `RemoteWebDriver` erbt, ist eine entfernte Nutzung dieser Linuxclients nicht möglich und wird daher im Rahmen dieser Arbeit nicht behandelt.

In Anforderung 1.4 erklärt der Kunde, dass als Testumgebung für mobile Endgeräte auch ein Apple IPhone möglich sein soll. Apple bietet für Entwickler die Möglichkeit

das Betriebssystem iOS und somit ein IPhone mittels der XCode-IDE zu emulieren. Die IDE verlangt als Betriebssystem Apples OSX, welches mit der entsprechenden Lizenzbedingung[49] wiederum nur auf Hardware von Apple installiert werden darf. Für eine Realisierung einer IPhone-Testumgebung gibt es also zwei Möglichkeiten.

1. Anschaffung eines Rechners auf Apples Hardwarebasis, um virtuelle Rechner mit OSX und XCode aufzusetzen.
2. Anschaffung von mehreren IPhones, um die Tests direkt auf der Hardware laufen zu lassen.

Da die Realisierung einer dieser Lösungen einen zusätzlichen finanziellen Aufwand und ein zusätzliches Evaluierungsprojekt erzeugen würde, wird die Anforderung 1.4 nur im Rahmen von Androidsystemen realisiert. Google stellt für Android eine kostenlose Entwicklungsumgebung zur Verfügung die auch einen kostenlosen Emulator enthält, der aus Lizenzsicht nicht an eine Plattform gebunden ist.[50] Diese Anforderung gibt jedoch indirekt vor, dass in einer Lösung Selenium 2 verwendet werden muss, da für eine Kommunikation mit Android der entsprechende WebDriver benötigt wird.

Die Anforderungen 2 und 2.1 können im simpelsten Fall über eine Konfigurationsdatei realisiert werden.

Requirement 2.1 verlangt eine zugesicherte Anzahl an Clients für einen User. Dabei ist die Testumgebung der Clients durch Anforderung 1.5 definiert. Die userabhängige Mindestanzahl lässt sich in den Userdaten hinterlegen.

Anforderung 4.1 verlangt nach einer eindeutigen SessionID. Um diese ID möglichst einfach zu gestalten, sollte eine Implementierung als Ganzzahl genutzt werden.

Anforderung 4.3.3 fordert einen alternativen Vorschlag an den Tester bezüglich seiner angeforderten Umgebung. Da Anforderung 1.5 eine auf jedem Testclient identische Testumgebung vorgibt, ist ein alternativer Vorschlag für eine Testumgebung nicht nötig.

Anforderung 10 erzeugt ein Problem, das sich in der parallelen Ausführung von Tests manifestiert, sobald auf einem Testrechner drei oder mehr Tests parallel in verschiedenen Browserinstanzen ablaufen. Bricht einer dieser Tests mit einem Fehler ab, oder hängen zwei oder mehr in einem identischen Zustand fest, kann nicht sichergestellt werden, dass der Anwender auch den durch ihn gestarteten und sich annormal verhaltenden Test findet.

[49] [Siehe 2. Permitted License Uses and Restrictions APP]
[50] [siehe Goo] und [siehe Flo07]

Ein zweites Problem zeigt sich, wenn Tests von verschiedenen Anwendern fehlschlagen und die Anwender den Zustand auf dem Testrechner prüfen wollen. Alle Testrechner haben, wie vorher bereits erläutert, als Betriebssystem nur Windows installiert. Besitzen nun beide Anwender keine Administratorrechte, so blockiert der zuerst angemeldete User den zweiten und es kann passieren, dass der angemeldete User versehentlich den Browser mit dem Test des zweiten Anwenders beendet. Besitzt nun einer der Anwender Administratorrechte, so kann er sich immer anmelden, womit er andere angemeldete User aus dem System wirft und die fremde Testausführung beendet. Somit wäre der Zustand des Testrechners verloren. Dieses Requirement wird noch gesondert im Rahmen der finalen Lösung betrachtet werden.

Anforderung 11 kann im einfachsten Fall über ein Mitzählen der Anfragen erfolgen, die zeitgesteuert in einer Datei gespeichert werden. Die dabei zu protokollierenden Messgrößen werden gesondert im Rahmen der finalen Lösung diskutiert.

3.1.2. Realisierbarkeit der technischen Anforderungen

Die technischen Anforderungen sind realisierbar. Die Plattformunabhängigkeit wird durch die vorgegebene Entwicklungssprache Java geliefert. Die Unterstützung von AludraTest ist auch realisierbar, da AludraTest in Java geschrieben ist und somit auf eine identische technische Basis aufgebaut werden kann.

3.2. Analyse vorhandener Lösungen

In diesem Teil werden zwei ausgewählte Produkte zum Resourcemanagement in einer Testautomation auf ihre Eignung für eine mögliche Lösung hin untersucht. Hierbei wird sich exemplarisch auf einen kommerziellen Dienst und ein fertiges freies Softwareprodukt beschränkt, um einen konzeptionellen Überblick zu gewinnen.

3.2.1. Selenium Grid

Das Selenium Grid ist eine Anwendung, um mehrere Selenium-Testumgebungen zu gruppieren und zentral als Ressource zu verwalten. Es besteht im Wesentlichen aus zwei Komponenten: dem Hub als Verwaltungsinstanz und dem Grid als Gruppierung der Testumgebungen, welche aus einzelnen Selenium-Servern (hier Nodes) besteht. Im

Folgenden wird der technische Aufbau des Grids sowie eine exemplarische Beschreibung, wie die Anforderung einer Ressource realisiert wird und wo die Grenzen des Selenium Grids zu finden sind, dargestellt.

Technischer Aufbau eines Selenium Grids

Eine Testumgebung mit Selenium Grid besteht, wie vorher erwähnt, aus zwei Teilen. Das Grid umfasst alle Selenium-Testserver und der Hub kennt deren Umgebungen. Um diese Verbindung zu realisieren, muss im ersten Schritt ein Selenium-Server über einen entsprechenden Parameter als Hub deklariert werden. An diesem Hub müssen sich die Selenium-Server als Nodes mit ihren entsprechenden Umgebungen registrieren. Damit ist eine Verbindung zwischen Hub und Node existent, um Tests auf einer definierten Umgebung der Node auszuführen.

Um dem Tester die entsprechende Verbindung zur Node zu ermöglichen, muss der Testclient eine Anfrage an den Hub stellen und seine gewünschte Umgebung als Anforderung mitteilen (im Seleniumkontext Capability). Wenn eine entsprechende Umgebung vorhanden ist, kann der Client ein RemoteWebDriver-Objekt erstellen. Im Fehlerfall gibt es eine `UnreachableBrowserException`.

Selenium Grid Node

Eine Node ist im Kontext von Selenium Grid jeder einzelne Testserver. Dabei kann die Umgebung der Node spezifiziert werden. Als Standardspezifikation gilt, dass maximal elf Browser, wovon je fünf Browser Firefox- und Chromeinstanzen sind und eine Instanz für den Internet Explorer reserviert ist. Die maximale Anzahl der parallel laufenden Instanzen ist in der Standardspezifikation auf fünf beschränkt. Diese Parameter können bei der Registrierung einer Node am Hub geändert und durch weitere Optionen ergänzt werden. Für die Browserumgebung existieren die in Tabelle 2 genannten Parameter.[51]

[51] [vgl. Ros12a]

Parameter	Beschreibung	Erlaubte Werte
browserName	Name des Browsers	android, chrome, firefox, htmlunit, internet explorer, iphone, opera
version	Version des Browsers	Versionsnummer des Browsers (kann je nach Hersteller unterschiedlich sein)
firefox_binary	Pfad zu Firefox (wichtig bei mehreren Firefox-Versionen)	Pfad in plattformabhängiger Notation
maxInstances	maximale Anzahl der Instanzen für diese Browserumgebung	$\mathbb{N}$
platform	Betriebssystemplattform auf der die Umgebung gestartet wird	LINUX, WINDOWS, MAC

Tabelle 2.: *Parameter des Command Line Arguments* `-browser`

Der einzige auf einer Node spezifizierbare Timeout regelt die Dauer bis zu einer Neuregistrierung, wenn der Hub nicht erreichbar ist. Dies ist über den Aufrufparameter `-registerCycle=`[*Zeit in ms*] realisiert. Die sonstigen Timeouts können auf dem Hub als zentrale Verwaltungsinstanz konfiguriert werden.[52]

Selenium Grid Hub

Der Selenium Grid Hub ist die zentrale Verwaltungsinstanz des aus allen Nodes bestehenden Grids. Aus technischer Sicht unterscheiden sich Node und Hub in erster Linie durch die unterschiedliche Rolle. Beide Services sind Selenium Server. Als Hub stehen nun mit der Verwantwortlichkeit als Verwaltungsinstanz andere Konfigurationsmöglichkeiten zur Verfügung. Wichtig sind hierbei die Timeouts, da sie die einzige Möglichkeit sind, das zeitliche Verhalten des Hubs zu spezifizieren. Der Selenium Hub kennt zwei Timeouts, die über die Command Line oder über die Methode `webDriver.manage().timeouts()` konfiguriert werden können.[53] Die Timeouts sind in Tabelle 3 aufgezeigt.

[52] [Ros12a]
[53] [Ros12a]

Parameter	Beschreibung	Erlaubte Werte
-timeout	Leerlaufzeit des Browsers	Ganzzahlige Zeit in Sekunden größer 0
-browserTimeout	Maximale Laufzeit des Browsers	Ganzzahlige Zeit in Sekunden größer 0

Tabelle 3.: *Timeouts im Selenium Grid*

Ablauf einer Ressourenanfrage mit Selenium Grid

Folgende Schritte sind für eine Anfrage einer Ressource notwendig, nachdem eine Node am Hub registriert wurde.

1. Definieren der Anforderung als `DesiredCapabilities`-Objekt
2. Erstellen eines neuen RemoteWebDriver-Objekts mit den definierten DesiredCapabilities und dem Selenium Grid Hub
3. Eventuell `UnreachableBrowserException` fangen

Das Activity Diagram in Abbildung 4 zeigt den kompletten Ablauf (Registrierung einer Node und Ressourcenanfrage) für ein definiertes Beispiel. Dabei soll folgende Umgebung gelten.

- Browser Firefox 11
- Plattform Windows
- maximal 3 Testinstanzen

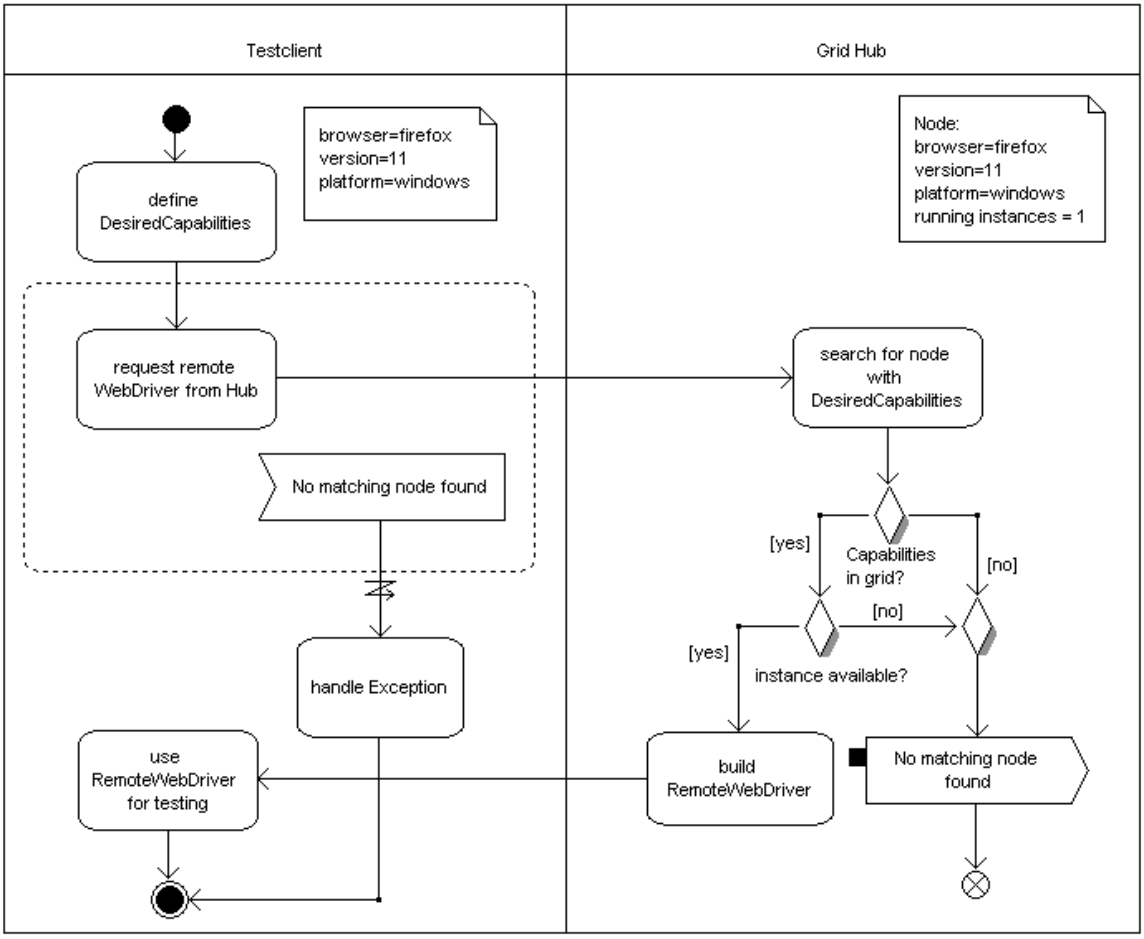

Abbildung 4.: *Activity Diagram - Ablauf einer Ressourcenanfrage im Grid*

Grenzen des Selenium Grids

Eine große Schwäche des Selenium Grids ist die Unterstützung von Androidsystemen im Androidemulator. Ein Androidemulator kann beliebig viele Systeme parallel emulieren, wobei jedes System eine eindeutige ID besitzt. Auf jedem Androidsystem muss eine WebDriver-App installiert und gestartet sein. Diese App stellt dann den Selenium-Testserver. Um einen Zugriff von aussen zu realisieren, wird der Port, an den die Teststeuerungsanweisungen geschickt werden, zum Androidemulator weitergeleitet.[54]

Diese emulierten Androidsysteme sind nicht im Grid nutzbar. Problem ist hierbei, dass die App ausschließlich eine lokale Seleniuminstanz bereitstellen kann und

[54] [vgl. Kem12, „Using the Remote Server“]

keine Anmeldung an einen anderen Seleniumserver oder an einen Hub ermöglicht. [55] Man kann nun aber einen Seleniumserver, auf dem der Emulator installiert ist und eine Androidemulation läuft, als Node mit dem Browser „Android“ am Hub registrieren. Dies führt jedoch zu dem Effekt, dass auf der Node ein Firefox geöffnet wird und dort alle Tests ablaufen.

Das Verhalten von IPhone-Umgebungen wurde auf Grund der Realisierbarkeit aus Kapitel 3.1.1 nicht weiter untersucht.

3.2.2. SauceLabs OnDemand

SauceLabs OnDemand ist ein kommerzieller Dienst, der das Testen von Websites in der Cloud ermöglicht. Im Folgenden werden die technischen Schnittstellen, die technische Basis und möglichen Testkonfigurationen dargestellt und das Kostenmodell kurz erläutert. Abschließend erfolgt eine Bewertung SauceLabs als Kandidat für das Resourcemanagement der Testautomation.

Technischer Aufbau von SauceLabs

SauceLabs setzt auf Selenium auf, wodurch auf Seiten des Kunden der Selenium Client benötigt wird. SauceLabs unterstützt sowohl Selenium 2, als auch den Vorgänger Selenium RC. Technisch unterscheiden sich die Versionen durch die Ansprache der Seleniuminstanzen. Während Selenium 2, wie in Kapitel 2.2.4 beschrieben, die Anfrageparameter in einem eigenen Objekt kapselt, arbeitet Selenium RC mit Stringparametern, um die Testumgebung zu spezfizieren. Dabei werden die Stringparameter direkt beim Konstruktoraufruf übergeben. Da diese Arbeit auf Selenium 2 basiert (siehe Anforderung 1.2 in Kapitel 3.1.1), wird Selenium RC in dieser Arbeit nicht weiter behandelt werden.

Der Ablauf einer Ressourcenanfrage ist hier ähnlich zu dem allgemeinen Ablauf im Selenium Grid (siehe Kapitel 3.2.1). Der einzige Unterschied besteht in den möglichen Testumgebungen und in der Authentifizierung bei SauceLabs, da diese über die URL erfolgt. Das Reporting, die Account- und SSH-Konfiguration kann über eine REST API erfolgen. REST ist eine webbasierte Schnittstelle, die eine Anfrage nach definierten Parametern über Http ermöglicht.[56]

[55] [siehe Sem12, Zeile 105-109]
[56] [Sau11]

Testkonfigurationen von SauceLabs

SauceLabs stellt zwei Plattformen (Windows und Linux) zum Test zur Verfügung, auf denen verschiedene Browser in diversen Versionen genutzt werden können. SauceLabs stellt die in Tabelle 4 aufgezeigten Testumgebungen zur Verfügung. Hierbei sei erwähnt, dass SauceLabs OSX momentan noch nicht unterstützt, es aber laut FAQ an einer Integration dieses Systems gearbeitet wird[57]. Mobile Endgeräte und Systeme (siehe Anforderung 2 in Kapitel 3.1.1) werden momentan nicht unterstützt.

Plattform	Browser	Version
Windows XP	Internet Explorer	6, 7, 8
	Firefox	3.0, 3.5, 3.6, 4, 5, 6, 7, 8, 9, 10, 11
	Google Chrome	aktuellste stabile Version
Windows 7 und Vista	Internet Explorer	9
	Firefox	4, 5, 6, 7, 8, 9, 10, 11
	Google Chrome	aktuellste stabile Version
Linux	Firefox	3.0, 3.6, 4, 5, 6, 7, 8, 9, 10, 11
	Google Chrome	aktuellste stabile Version
	Opera	11

Tabelle 4.: *Spezifizierbare Testumgebungen*

Desweiteren können keine userspezifischen Umgebungen genutzt werden, die nicht den in Tabelle 4 spezifizierten Umgebungen genügen. Somit können beispielsweise keine speziellen JavaScript Engines oder bestimmte Firefox-Plugins genutzt werden.

Kostenmodell von SauceLabs OnDemand

SauceLabs OnDemand unterteilt sein Kostenmodell nach zwei Kriterien.

- Anzahl paralleler Tests
- Testdauer in $\frac{Minuten}{Monat}$

Tabelle 5 zeigt das aufgeschlüsselte Kostenmodell.

[57] [Sau12]

Modell	Anzahl paralleler Tests	Testdauer in $\frac{Minuten}{Monat}$	Preis in $\frac{\$}{Monat}$
The Small Team	4	1000	49
The Team	16	4000	149
Commercial	32	8000	279
Commercial+	48	16000	499

Tabelle 5.: *Kostenmodell von SauceLabs OnDemand*

Jede weitere Testminute über der monatlichem Maximalmenge schlägt mit 0,05$ zu Buche.

3.3. Bewertung der lösungskandidaten Empfehlung einer Lösung

Basierend auf den Kapiteln 3.1 und 3.2 wird in diesem Teil eine Empfehlung für eine Lösung des Resource-Mangement gegeben. Als erstes werden für einen Gesamtüberblick die realisierbaren Anforderungen mit den daraus resultierenden KO-Kriterien dargelegt und die vorhandenen Lösungen dagegen dargestellt.

3.3.1. Abgrenzungen

Tabelle 6 stellt die Abgrenzungen zu den Anforderungen im Rahmen dieser Arbeit dar.

Anforderung	Inhalt	Abgrenzung im Rahmen dieser Arbeit
1.1	Als Identifikationskriterien sind Kombination aus Betriebsystem und Browser zu nehmen	Betriebssystem nur Windows
1.4	Testumgebungen sollen mobile Endgeräte (IPhone und Android-Devices) beinhalten	Lizenzbedingter Ausschluss von Testumgebung IPhone
4.3.3	Bei Rückgabe N2(Anzahl der Mindestclients nicht verfügbar) wird eine mögliche Alternative zurückgegeben	entfällt, da alle Testumgebungen identisch sind
10	Bei Fehlgeschlagenem Test soll der User in der Lage sein, Zugriff auf den Zustand der ihm zugeteilten Ressource zu erhalten	Wird gesondert für die finale Lösung betrachtet
11	Reportingschnittstelle für die Optimierung der Ressourcenkonfiguration in Abhängigkeit der Anfragen	Entfällt in dieser Arbeit, da eine grundlegende Auswertung auch mit Hilfe der Logdateien möglich ist.

Tabelle 6.: *Abgrenzungen im Rahmen dieser Arbeit*

3.3.2. Selenium Grid und SauceLabs bewertet an den Anforderungen

Tabelle 7 zeigt die von SauceLabs und Selenium Grid 2 jeweils erfüllten Anforderungen. Hierbei wird sich auf die kritischen Anforderungen beschränkt.

Anforderung	1	1.1	1.4	1.5	2	3	10
Selenium Grid 2	(X)	X		X	X	X	X
SauceLabs	(X)	X			X	X	X

Tabelle 7.: *Matrixaufschlüsselung von Selenium Grid 2 und Saucelabs nach Anforderungen*

Es zeigt sich, dass SauceLabs und Selenium Grid 2 als Lösung nicht gewählt werden können, da kritische Anforderungen von beiden Kandidaten nicht erfüllt werden. Im

Fall von Selenium Grid 2 mangelt es an einer Unterstützung von mobilen Testumgebungen. SauceLabs ermöglicht zusätzlich nicht, dass die Windows-Testumgebungen nicht mit den Clientimages des Kunden konfiguriert werden können.

3.3.3. Bewertung von Selenium Grid 2 als Lösungskandidat

Bei Selenium Grid schlägt sich das erste KO-Kriterium nach Tabelle 1 nieder. Da auf Grund der AndroidWebDriver-App keine Testumgebungen mit Android aus dem Grid geliefert werden kann, scheidet Selenium Grid 2 als Lösungskandidat aus. Trotzdem stellt das Selenium Grid in der aktuellen Version 2.20 für alle nicht-mobilen Tests eine sinnvolle und einfache Lösung für ein Resourcemanagementtool dar.

3.3.4. Bewertung von SauceLabs OnDemand als Lösungskandidat

SauceLabs OnDemand eignet sich nicht als Lösung für das Resourcemanagement. Hier schlagen beide KO-Kriterien nach Tabelle 1 zu buche. SauceLabs unterstützt nach Tabelle 4 keine mobilen Testumgebungen und durch die fest definierten Kombinationen von Betriebssystem und Browser zu Testumgebungen durch SauceLabs, ist es nicht möglich, dass Clientimages des Kunden in der Cloud als Testumgebungen genutzt werden können.

3.3.5. Empfehlung

Es empfiehlt sich, eine eigene Implementierung als Lösung zu wählen, da man die Anforderungen des Kunden an die Umgebung angepasst realisieren kann. Durch die Nutzung der `AndroidWebDriver` ohne das Grid ist es möglich, auf Instanzen im Androidemulator zuzugreifen. Damit wäre der WebDriver die freizugebene Ressource. Die Testclients können in einer Eigenimplementierung beim Kunden lokalisiert werden und daher auch mit individuellen Umgebungen ausgestattet werden.

Eine Erweiterung des AndroidWebDrivers um die Funktionalität einer Node wäre auch möglich, wird jedoch nicht durchgeführt. Eine Node hat neben der Aufgabe des Selenium Servers zur Browsersteuerung noch die Aufgabe, eine Verbindung zwischen Hub und Node sicherzustellen und zu überwachen. Die emulierten Androidsysteme auf den virtuellen Testsystemen sind bezogen auf die Geschwindigkeit recht langsam. Würde man der Webdriver-App zusätzliche Funktionalitäten hinzufügen, die parallel ausgeführt werden müssen, würde die Geschwindigkeit der emulierten Androidinstanzen weiter sinken. Dies führt unweigerlich zu einer Erhöhung der Ausführungszeit

einzelner Tests.

Anforderung 10 in der empfohlenen Lösung

Die in Abschnitt 3.1.1 dargestellten Probleme beim Zugriff auf Testumgebungen, auf denen Tests parallel ablaufen, besteht in dieser Lösung weiterhin. Die virtuellen Testclients liegen in der IT-Landschaft des Kunden und sind mit den kundenspezifischen Windowssystemen ausgestattet. Im Rahmen dieser Arbeit werden parallele Tests auf einem Testclient nicht weiter betrachtet.

Es ist auf Windowssystemen nicht möglich, dass zwei Benutzer zur gleichen Zeit auf einem System angemeldet sind. Würde sich ein Anwender auf dem Client anmelden, so brechen parallel ausgeführte Tests ab, da der zu dem Zeitpunkt angemeldete User automatisch abgemeldet wird.

4. Architektur

Im folgenden werden die architektonischen Grundsteine des Resource-Mangement-Systems beschrieben. Dabei wird auf die Kommunikation, die Kernkomponente, die Konfigurationskomponente und die Loggingkomponente eingegangen.

4.1. Kommunikation zwischen AludraTest und Resourcemanagementserver

In diesem Kapitel wird die Kommunikation zwischen AludraTest und dem Resourcemanagementserver untersucht. Nach der Darstellung der Ausgangssituation wird ein Überblick über zwei gängige Kommunikationskonzepte gegeben. Diese Konzepte werden bezüglich ihrer Eignung zur Kommunikationsgrundlage der hier beteiligten Kommunikationspartner untersucht.

4.1.1. Ausgangssituation

Durch die Analyse der technischen Anforderungen in Kapitel 3.1.1 zeigt sich ein hoher Kommunikationsaufwand zwischen AludraTest und dem Managementserver. Die Kommunikation zwischen AludraTest und Resourcemanagementserver ist eine Kommunikaiton zwischen entfernten Rechnern in einem Server-Client-Verhältnis. Um zu gewährleisten, dass die Kommunikation zwischen allen entfernten Stellen auf identischem Wege möglich ist, muss eine einheitliche Technologie eingesetzt werden. Im Folgenden wird die technische Basis für die Kommunikation zwischen AludraTest und dem Resourcemanagementserver diskutiert.

4.1.2. Kommunikationskonzepte

Um die Kommunikation aus Sicht der Architektur zu implementieren, bieten sich zwei Techniken an. Eine Möglichkeit für die Implementierung ist ein eventgetriebenes System, das über Nachrichten kommuniziert. Die andere Möglichkeit ist das zeitgetriebene System in Form von Polling, wobei auf einer Schnittstelle in definierten zeitlichen

Abständen ein Zustand erfragt wird. Im Folgenden wird ein kurzer Überblick über die jeweiligen Charakteristika der Konzepte gegeben und eine Entscheidung über den Einsatz im Rahmen dieser Arbeit getroffen.

Zeitgetriebenes System

Beim zeitgetriebenen System wird zu definierten Zeitpunkten (z.B. Takt) eine Aktion durchgeführt. Bei einer Kommunikation zwischen zwei Systemen würde daher ein System den Zustand des Kommunikationspartners erfragen. Eine mögliche Implementierung dieses Konzeptes ist das Polling. Dabei wird auf einer definierte Schnittstelle der Zustand des Kommunikationspartners erfragt und je nach erkanntem Zustand eine Aktion ausgeführt. Vorteil dieses Konzepts ist eine simple Implementierung und eine schnelle Reaktionsmöglichkeit, da Änderungen sehr schnell bemerkt werden. Diese Reaktionsgeschwindigkeit zu realisieren ist jedoch aufwändig. Um die Änderung an einer Schnittstelle schnellstmöglich zu erkennen, muss die Abfrage in den kleinstmöglichen Zeitabständen erfolgen. In einem getakteten technischen System wäre das beispielsweise mit jedem einzelnen Takt des Prozessors, wobei die Prozessorzeit in diesem Fall zu 100% für das Polling genutzt wird. Dies ist in der Praxis jedoch selten erwünscht, da ein Programm mehr als eine Sache erledigen soll und somit nicht alle Taktzyklen für das Polling genutzt werden können. Ein weiterer Nachteil ist die Latenzzeit bei entfernten Systemen. Hierbei spielt die Übertragungsgeschwindigkeit des Kommunikationsmediums eine Rolle. Das Polling besteht aus zwei Teilschritten: Anfrage und Rückgabe. Sind die entfernten Systeme nun mittels eines Kommunikaitonsmediums (z.B. Ethernet) verbunden, benötigt das Abfragesignal eine gewisse Zeit, um das entfernte System zu erreichen, und somit verzögert sich das Aussenden des Rückgabesignals. Das Rückgabesignal muss ebenso das Kommunikationsmedium passieren und somit ist der früheste Reaktionszeitpunkt des pollenden Systems nach zwei, durch das Kommunikationsmedium beschränkte Signallaufzeiten. Zusätzlich ist es möglich, dass ein Zustandswechsel nicht erkannt wird, wenn zwischen zwei Abfragezeitpunkten ein Statusübergang der Form $A \rightarrow B \rightarrow A$ stattfindet.[58]

Eventgetriebenes System

Ein eventgetriebenes System basiert auf definierten Ereignissen, welche zu einer Reaktion führen. Ist ein Ereignis bei einem der Kommunikationsteilnehmer eingetreten, so wird eine Nachricht an den anderen Teilnehmer gesendet. Wie die Nachricht gesendet werden muss, wird über Schnittstellen definiert. Diese Schnittstellen sind beiden Kommunikationspartnern bekannt. Die Kommunikationsteilnehmer wissen in dieser Architektur nichts über den inneren Aufbau und die inneren Zustände des anderen.

[58] [vgl. Bra]

Der damit verbundene Informationsverlust wird durch die versendeten Nachrichten kompensiert. Jede Nachricht enthält Daten für den Kommunikationspartner, mit denen er die für ihn definierten Aufgaben erledigen kann. Da eine Nachricht nicht verlangt oder angefragt werden muss, wird bei entfernten Kommunikationspartnern immer nur eine durch das Kommunikationsmedium bedingte Signallaufzeit für die Kommunikation benötigt. Zusätzlich wird Prozessorlast für eine Kommunikation nur erzeugt, wenn eine Nachricht versendet wird. Im Gegenzug ist durch die lose Kopplung der Teilnehmer die Möglichkeit des Nachrichtenverlusts gegeben, da keine ständige Verbindung sichergestellt ist.[59]

4.1.3. Entscheidung für ein Kommunikationskonzept

Im Rahmen dieser Arbeit wird für die Kommunikation zwischen Tester und Resourcemanagementserver, die für die Verwaltung der Ressourcen benötigt wird, ein eventgetriebenes System genutzt. Die Dauer der einzelnen Testfälle kann untereinander stark variieren. Dadurch würde im Fall des zeitgesteuerten Systems bei langer Belegung einer Ressource sehr viel Netzwerklast durch das Polling entstehen. Das eventgetriebene System bietet hier den Vorteil, dass eine Benachrichtigung nur dann gesendet wird, wenn eine Änderung an dem Zustand einer Ressource erfolgt ist. Somit wird das Netzwerk nur belastet, wenn auch eine Aktion auf die Belastung folgt. Eventgetriebene Systeme sind zudem schneller, da eine Nachricht nur einmal das Kommunikationsmedium passieren muss. Beim Polling dagegen muss das Signal das Medium zweimal durchqueren. Folglich ist ein eventgetriebenes System für die Kommunikation betreffend der Ressourcenverwaltung von Vorteil.

Um den Nachteil des möglichen Nachrichtenverlusts zu begegnen, der z.B. bei einem Netzwerkausfall auftreten kann, wird ein Signal benötigt, mit dem Informationen über die Erreichbarkeit des Kommunikationsteilnehmers übertragen werden. Hierfür eignet sich eine Mischung aus zeit- und eventgetriebenen System.

Es wird mit einem definierten Takt eine Nachricht an den Kommunikationspartner versendet. Die Nachricht dient als Lebenssignal. Diese Nachricht wird in einem bestimmten Zeitfenster erwartet, so dass bei ausfallender Nachricht die Verbindung als unterbrochen angesehen wird und entsprechend reagiert werden muss.

[59] [vgl. Bra]

4.2. Kernfunktionalitäten und Kernkomponenten

Dieser Teil befasst sich mit den Kernfunktionalitäten und Kernkomponenten. Dabei wird auf die eingesetzten Technologien und deren Auswirkung eingegangen. Die Kernfunktionalitäten umfassen:

- Definition einer Ressource
- Anfrage einer Ressource
- Identifikation einer Ressource
- Kommunikation mit den Clients
- Zuweisung einer Ressource
- Administration des Resourcemanagements

Daraus ergeben sich die Kernkomponenten.

- Ressource
- Ressourcenidentifikator
- Ressourcenanfrage
- Kommunikationskanal
- Administrationsschnittstelle

Zunächst werden die Kernkomponenten inhaltlich definiert.

4.2.1. Inhaltliche Definitionen

Im folgenden werden die Kernkomponenten inhaltlich offengelegt.

Ressource

Eine Ressource ist eine Kombination aus exakt einem Betriebssystem (entweder Windows oder Android) mit exakt einem aus n verschiedenen Browsern. Anforderung 1.1 und 1.3 (siehe Tabelle 6) verlangen zum einen, dass die Clientkonfiguration des Kunden auf den virtuellen Testrechnern verwendet, wird und zum anderen die Kombination von Betriebssystem und Browser als Identifikation. Die Analyse bezüglich der Realisierbarkeit (siehe Tabelle 6) gibt als Betriebsystem Windows vor, da die Linuxclients ohne grafische Oberflächen wie KDE oder Gnome ausgeliefert werden. Zulässige Ressourcen sind in Tabelle 8 definiert.

Betriebssystem	Browser
Windows	Internet Explorer
	Firefox
Android	versionsabhängiger Standardbrowser

Tabelle 8.: *Valide Ressourcen*

Ressourcenidentifikator

Eine Ressource ist mindestens über exakt ein Betriebssystem mit exakt einem aus n verschiedenen Browsern identifiziert (siehe Tabelle 6 Anforderung 1.1 und 1.3) und muss ein Element aus der Menge valider Ressourcen gemäß Tabelle 8 sein.

Ressourcenanfrage

Die Anfrage einer Ressource seitens des Testers erfolgt nach erfolgreicher Authentifizierung über einen Ressourcenidentifikator. Der Resourcemanagementserver wird in der Lage sein, diese Anfrage auszuwerten, da die Anfrage über eine gemeinsame Schnittstelle erfolgt.

Kommunikationskanal

Der Kommunikationskanal stellt eine standardisierte Verbindung zwischen Managementserver und Tester und ist abhängig von der genutzten Entwicklungstechnologie. Durch die Ausrichtung auf die Entwicklungstechnologie Java soll eine homogene Umgebung geschaffen werden.

4.2.2. Ablauf einer Ressourcenanfrage

Die Ressourcenanfrage folgt definierten Schritten. Dabei soll Folgendes für den erfolgreichen Fall gelten.

1. Definition der DesiredCapabilities und der Anzahl der benötigten Ressourcen
2. Anfrage beim Resourcemanagementservers
3. Prüfung der Authentifizierung seitens des Resourcemanagementserver
4. Prüfung seitens des Resourcemanagementserver, ob Ressourcen verfügbar
5. Rückgabe der Session, der die Ressourcen zugeordnet sind
6. Übergabe der Ressourcen an den Tester

Im Fehlerfall können zwei Signale an den Tester gesendet werden.

- Authentifizierung fehlgeschlagen
- Ressource nicht verfügbar

Auf dieser Basis lässt sich ein Activity Diagram erstellen, dass beide Abläufe (Erfolgs- und Fehlerfall) abbildet.

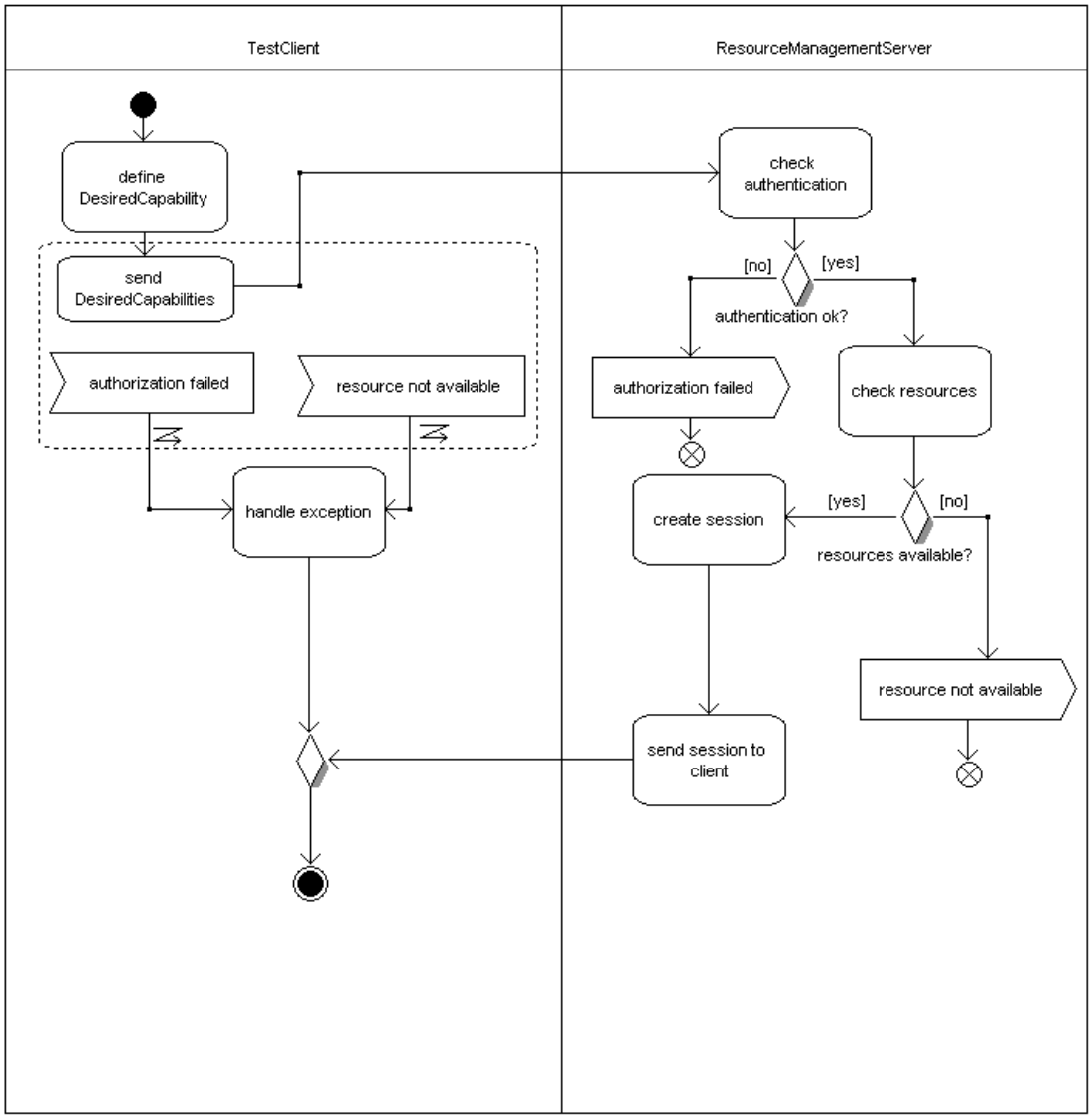

Abbildung 5.: *Activity Diagram - Ablauf einer Ressourcenanfrage*

4.2.3. Technologische Definition der Kernkomponenten

In diesem Teil wird auf die technologische Definition der einzelnen Komponenten eingegangen. Als grundlegende Technologie wird Java eingesetzt, da als Testframework Selenium 2 vorgegeben ist. Selenium bietet Schnittstellen zur Ansprache mit Java an.

Ressource

Eine Ressource besteht aus mehreren Teilen. Um eine Ressource aus Sicht des Resourcemanagements zu beschreiben, werden Informationen über die installierten Umgebungen und die auf jeder Umgebung mögliche Anzahl an paralellen Tests hinterlegt. Der Tester benötigt nur die Information, wie die Ressource zu erreichen ist. Da die Testautomation auf Selenium 2 basiert und die Steuerung der Browser über WebDriver realisiert ist, wird eine URL zur Adressierung der Ressource genutzt.

Ressourcenanfrage

Die Ressourcenanfrage beinhaltet vier Informationen. Neben den Authentifizierungsdaten des Testers enthält diese die vom Tester benötigte Umgebung, die minimal und maximal benötigte Anzahl der jeweiligen Testumgebung. Die Spezifikation der Testumgebung erfolgt über eine `DesiredCapabilities` aus Selenium 2. Der Vorteil hierbei ist, dass in Selenium ein Comparator existiert, der `DesiredCapabilities` miteinander vergleicht. Damit entfällt der Aufwand, einen eigenen Comparator zu entwickeln.

Kommunikationsprotokoll

Als Entwicklungstechnologie wird Java benutzt, weshalb die Kommunikation über RMI erfolgt. RMI stellt ein in Java standardisiertes Kommunikationsprotokoll dar, mit dem Zugriffe auf entfernte JVMs realisiert werden. Dies erfolgt über Serialisierung. Im Rahmen dieser Arbeit wird RMI für die Kommunikation verwendet, da es die Freigabe der Objekte über das Netzwerk mit den Mitteln von Java erlaubt und dabei im Gegensatz von z.B. CORBA eine einfache server- und clientseitige Implementierung ermöglicht.

Für eine umfassendere Darstellung sei hier auf Kapitel 2.2.2 verwiesen.

4.2.4. Architektur mit clientseitigem Framework AludraTest

Da als Basis ein eventgetriebenes System verwendet wird, muss auf Seiten des Frameworks AludraTest[60] eine Schnittstelle existieren, die entsprechende Ereignisse erkennt und die dadurch vermittelten Ressourcen nutzt. Auf seiten des Resourcemanagement-Servers muss eine Schnittstelle existieren, die eine Ausgabe der Ereignisse ermöglicht. Um die Verbindung zwischen dem Framework und dem Server zu realisieren, wird

[60] siehe Kapitel 1.2

auf Grund des RMIs eine Komponente für die Registry benötigt. Da RMI aus sicherheitstechnischen Gründen verbietet, von fremden Hosts aus Objekte an der Registry anzumelden, wird diese Komponente auf Seiten des Servers vorzufinden sein. Als einziges, auf der Registry angemeldetes Objekt, wird die Schnittstelle zwischen Framework und Server sein, da Ressourcen und User als serialisierte Daten über RMI ausgetauscht werden. Die Abbildung 6 zeigt die Einteilung in Komponenten als Komponentendiagramm.

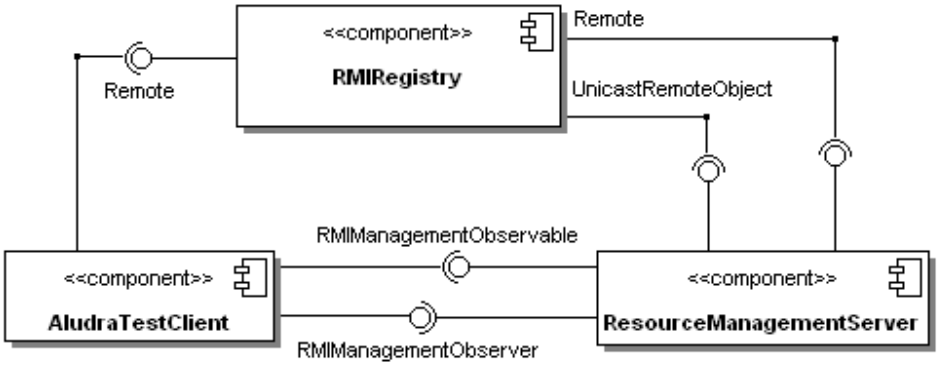

Abbildung 6.: *Komponentendiagramm reduziert auf die Kommunikation zwischen Testframework und Resourcemanagement-Server*

Komponenten

Es existieren nach Abbildung 6 drei zentrale Komponenten bezüglich des AludraTest Frameworks und des Resourcemanagementservers.

- AludraTest
- Resourcemanagement-Server
- RMI-Registry

AludraTest fordert über RMI Ressourcen beim Resourcemanagementserver an und führt auf der Ressource mittels Selenium 2 seine JUnit-Tests aus. Der Resourcemanagementserver ist an der RMI-Registry registriert und wird nur über die Registry angesprochen. Die RMI-Registry muss dabei, wie vorher erwähnt, auf dem gleichen Host sein wie der Resourcemanagementserver.

Schnittstellendefinition

Zwischen dem AludraTest und dem Managementserver existieren zwei Schnittstellen, die folgende Interaktionen ermöglichen.

- Authentifizierung des Testers
- Anfrage nach Ressourcen
- Übergabe von Sessions und Ressourcen
- Auflösen einer Session
- Rückgabe der Ressourcen

Die beidseitige Kommunikation wird über Nachrichten erfolgen. Grund hierfür ist, dass nicht sichergestellt ist, dass zwischen Resourcemanagementserver und AludraTest eine ständige Verbindung besteht.

4.3. Konfigurationsschnittstellen

Die Konfiguration des Management-Tools erfolgt über Textdateien mit eindeutig definierten Strukturen. Im Folgenden wird auf das Format der Konfigurationsdateien und die enthaltene Syntax eingegangen.

4.3.1. Format der Konfigurationsdateien

Um eine strukturierte Abbildung der von der Anwendung genutzten Daten sicherzustellen, wird die Konfiguration in XML-Dateien[61] abgelegt.

XML ermöglicht, komplette Datenstrukturen in einer selbst definierten Syntax abzulegen. Somit können die in der Anwendung genutzten Datenstrukturen nachgebildet werden.

[61] Extensible Markup Language

Konfiguration der Testclients

Ein Testclient ist ein definierter virtueller Rechner mit einer eindeutigen IP-Adresse und kann eine definierte Menge an Browsern beinhalten, die wiederum in einer definierten Menge an Instanzen genutzt werden können. Daraus lässt sich für die Darstellung eines Testclients in XML eine Struktur ableiten.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <clients>
3         <client ip="128.0.0.1" platform="windows" pool="dynamic">
4                 <capability>
5                         <browser>firefox</browser>
6                         <version>11</version>
7                         <maxInstances>5</maxInstances>
8                 </capability>
9                 <capability>
10                        <browser>chrome</browser>
11                        <version>5</version>
12                        <maxInstances>1</maxInstances>
13                </capability>
14        </client>
15 </clients>
```

Listing 6: *Beispiel einer Clientkonfiguration*

Da als eindeutiges Identifikationsmerkmal jeder virtuelle Rechner immer nur eine eindeutige IP-Adresse und ein Betriebssystem besitzt, eignen sich diese festen Eigenschaften als Attribute des Client und werden auch so implementiert. Das Attribut Pool wird genutzt, um den Client entweder als reserviert für einen Nutzer (static), als frei verfügbar (dynamic) oder als fehlerhaft (no pool) zu markieren. Da jeder Client mehrere Browser in verschiedenen Versionen beherbergen kann, werden diese analog zu Selenium 2 in Capabilities zusammengefasst. Jede Capability kann somit aus allen Elementen einer Selenium 2 `DesiredCapability` bestehen. Um dies darzustellen, können Eigenschaften einer Capability als Element in einem Element `Capability` genutzt werden. Zusätzlich wird das Element `maxInstances` eingeführt, mit dem definiert wird, wieviele Instanzen dieser Capability möglich sind.

Anhand von Listing 6 wird also der folgende Client erstellt.

- IP 128.0.0.1
- Betriebssystem Windows

- frei verfügbar
- Browser Firefox 11 im Standardpfad
- mit maximal 5 Instanzen parallel möglich
- Browser Chrome 5 im Standardpfad
- mit maximal 1 Instanz möglich

Konfiguration der User

Ein User besitzt neben den Authentifizierungsdaten (Namen und Passwort) zusätzlich eine Priorität und eine Mindestanzahl an global zugesicherten Ressourcen. Daraus lässt sich eine XML-Struktur für die Darstellung des Users ableiten.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <users>
3         <user username="Test01" password="test1234">
4                 <prio>2</prio>
5                 <reservedclients>4</reservedclients>
6         </user>
7 </users>
```

Listing 7: *Beispiel einer Userkonfiguration*

Um einen User eindeutig zu identifizieren, wird die Kombination aus Nutzername und Passwort genutzt und daher in den Attributen des Userelements gespeichert. Das Userelement beinhaltet Informationen über die Priorität und Anzahl global zugesicherter Ressourcen als eigene Elemente, da sie nicht den User selbst, sondern seine Möglichkeiten beschreiben.

Das Beispiel in Listing 7 lässt sich ein User ableiten.

- Name Test01
- Passwort test1234
- Priorität 2
- global 4 Ressourcen zugesichert

Eine Definition der Daten ist in Kapitel 5.3.2 erläutert.

4.3.2. Einlesen der Konfigurationsdateien

Java stellt für das Einlesen von XML-Dateien das Document Object Model (DOM) und die Simple API for XML (SAX) zur Verfügung. DOM stellt eine einfache Schnittstelle bereit, die das Parsen eines XML-Dokumentes in eine Datenstruktur in Form eines Baums, das Navigieren in dieser Stuktur, das Editieren der Daten innerhalb der Struktur und das Serialisieren der Daten in eine Datei ermöglicht. Hierbei wird jedoch immer die komplette Datei eingelesen.

SAX arbeitet hingegen streambasiert mit Events und Handlern. Dabei erzeugt jeder XML-Baustein ein Event. Die vom Entwickler erstellten Handler verarbeiten diese Events. Daher eignet sich SAX sehr gut für Suchen nach Änderungen und kleinen Datenmengen und erzeugt daher einen geringeren Ressourcenaufwand als DOM.

Da die Konfigurationsdateien immer komplett eingelesen und in Javaobjekte umgesetzt werden sollen, wird für die Implementierung DOM genutzt.

Abfrage der XML-Elemente

Die Abfrage der XML-Elemente ist mit zwei Techniken realisierbar. Da die Daten in einer mit DOM organisierten Baumstruktur vorliegen, kann die Navigation durch die Knoten mit dem im DOM definierten Methoden vorgenommen werden. Dies ist jedoch sehr aufwändig. Die API ist sehr allgemein formuliert, da sie natürlich für alle mit DOM erzeugten Bäume gelten soll. Im Kapitel 4.3.1 ist die Konfigurationssyntax definiert, worduch ein Spezialfall des allgemeinen Baumes vorliegt. Es sind somit alle möglichen Konfigurationselemente bekannt und eine direkte Ansprache der Elemente sinnvoll. Hierfür eignet sich die Abfrage über XPath.

Abfragen über XPath

XPath ist eine von der *W3C* standardisierte Abfragesprache, um Inhalte von XML-Dokumenten zu identifizieren. Dabei wird die Baumstruktur von XML genutzt, um eindeutige Pfade zu Elementen zu definieren. Als Basisdatei für eine beispielhafte Anfrage wird das Listing 6 genutzt. Dabei soll nun der Browsername der zweiten Capability ermittelt werden.

Als erstes müssen alle Capabilities des Clients gefunden werden, wofür eine Abfrage formuliert werden muss.

```
1 //clients/client[1]/capability
```

Listing 8: *Abfrage nach allen Capabilities des erste Clients*

Mit dieser Anfrage werden aus Sicht des XML-Baums alle Capability-Elemente des ersten Client-Elements unter dem Element Clients ausgewählt. Folglich muss für das zweite Capability-Element `[2]` zu der Abfrage hinzugefügt werden.

```
1 //clients/client[1]/capability[2]
```

Listing 9: *Abfrage nach der zweiten Capability des Clients*

Um nun den Browsernamen zu erhalten, muss zum einen die Ergänzung `/browser` erfolgen und zum anderen der Text selektiert werden. Die Ergänzung um das Element Browser selektiert nur das Element im Baum, jedoch nicht seinen Inhalt.

```
1 //clients/client[1]/capability[2]/browser/text()
```

Listing 10: *Abfrage nach dem Browsernamenen*

Mit der Abfrage aus Listing 10 erhalten wir nun den Text des Elements Browser unter dem zweiten Capability-Element des ersten Clients unter dem Wurzelelement Clients.

XPath stellt noch weitere Funktionalitäten im Rahmen der Abfrage von XML-Elementen bereit.

- Anzahl vorkommender Elemente
- Abfrage auf Attribute
- logische Verknüpfungen
- Substring-Operationen

Dies ist nur eine Auswahl der Möglichkeiten, da XPath ein zu mächtiges Werkzeug ist, um alle möglichen Funktionalitäten aufzulisten. Eine detailiertere Auflistung weiterer Funktionalitäten ist auf der Website der W3C-School[62] zu finden.

[62] [vgl. Dat12]

4.4. Logging

In diesem Kapitel wird die Einordnung des Loggings aus architektonischer Sicht beschrieben. Dabei werden Technologien miteinander verglichen und für die Implementierung im Rahmen dieser Arbeit bewertet.

4.4.1. Loggingframeworks Java Logging und Logback

In diesem Abschnitt wird die in Java vorhandene API Java Logging und das Framework Logback erläutert und verglichen.

Java Logging

Java Logging ist eine seit Java 1.4 vorhandene API, die eine Loggingfunktionalität bereit stellt. Architektonisch setzt sich das Logging zwischen die Anwendung und die Benutzerschnittstelle, um Möglichkeiten zur Filterung und Formatierung der Lognachrichten zu ermöglichen.

Um eine Filterung zu ermöglichen, sind Loglevels definiert, die die Wichtigkeit der Meldung beschreiben. Die Loglevels sind in der Tabelle 9 hierarchisch zusammengefasst[63].

Hierarchiestufe	Loglevel	Standardausgabekanal
höhste Stufe	SEVERE	Konsole
	WARNING	Konsole
	INFO	Konsole
	CONFIG	keiner
	FINE	keiner
	FINER	keiner
niedrigste Stufe	FINEST	keiner

Tabelle 9.: *Prioritäten und vordefinierte Ausgabekanäle der Loglevels von Java Logging*

Jeder Logger kann auf einen Loglevel gesetzt werden, um eine Filterung zu realisieren. Dabei lässt der Logger alle Meldungen passieren, die dem zugewiesenen Loglevel und allen höher eingestuften Levels entsprechen.

[63] [vgl. BW03]

Um den Ausgabekanal zu bestimmen, existieren entsprechende Handler, die eine Umleitung realisieren. Tabelle 10 fasst diese Handler zusammen.[64] [65]

Handler	Ausgabekanal
`StreamHandler`	`OutputStream`
`ConsoleHandler`	System.err
`FileHandler`	einzelne Datei oder eine Sammlung abwechselnder Zieldateien
`SocketHandler`	Entfernter TCP-Port
`MemoryHandler`	Im Speicher vorhandener Puffer

Tabelle 10.: *Handler für Ausgabekanäle für Meldungen von Java Logging*

Die Handler und Filter müssen im Programm konfiguriert werden, da die Java Logging API keine externe Konfigurationsdatei vorsieht.

Java Logging unterstützt von sich aus zwei mögliche Formatierungen. Der `SimpleFormatter` stellt dabei eine reine textuelle Ausgabe der Meldung, die mit Hilfe eines Handlers an den entsprechenden Ausgabekanal gegeben werden kann. Dabei wird neben dem Zeitpunkt der Meldung und dem Loglevel auch der nach Klasse und Methode aufgeschlüsselte Ort angegeben. Zusätzlich wird die vom Entwickler zusätzliche mögliche Meldung hinzugefügt. Die zweite Formatierungsmöglichkeit stellt der `XMLFormatter`. Dabei wird eine in Java Logging vorgegebene Struktur für einen XML-Baum genutzt, um die Logmeldung in eine XML-Datei zu schreiben. Die vorgegebene Struktur ist in einer DTD[66] verfasst[67].

Logback

Logback ist ein Framework, das als Nachfolger vom viel verwendeten Log4j entwickelt wurde. Es liefert eine Implementierung von SLF4J (Simple Logging Facade for Java) mit, so dass eine einfach Ansprache über ein Facade-Pattern[68] möglich ist. Wie auch Java Logging bietet Logback Möglichkeiten, Meldungen zu filtern und zu formatieren.

Die Filterung erfolgt auch hier über Loglevels. Tabelle 11 stellt die Loglevels in einer Hierarchie zusammen.

[64] [vgl. BW03]
[65] [vgl. Ou10b]
[66] Document Type Definition
[67] [vgl. Ou10b]
[68] [vgl. GHJV95]

Hierarchiestufe	Loglevel
höhste Stufe	ERROR
	WARN
	INFO
	DEBUG
niedrigste Stufe	TRACE

Tabelle 11.: *Prioritäten der Loglevels von Logback*

Auch bei Logback schreibt ein Logger alle Meldungen, die sein definiertes Loglevel und höher besitzt. Im Gegensatz zu Java Logging werden alle Ausgaben als Standard auf die Konsole geschrieben.

Logback ermöglicht die Änderung eines Ausgabekanals mit Hilfe einer Konfigurationsdatei[69]. In der im XML-Format geschriebenen Datei wird eine Zuordnung zwischen Loglevel und Ausgabekanal realisiert. Dabei kann ein Loglevel auf mehrere Ausgabekanäle verweisen, um beispielsweise bei Meldungen des Levels *ERROR* in einer Datei zu protokollieren und per E-Mail eine Meldung an die Administratoren zu senden[70].

SLF4J bietet in der Konfigurationsdatei die Möglichkeit, Ausgaben mit Hilfe von Patterns zu formatieren. Die Formatierung ist dabei in der Konfiguration einem Ausgabekanal zugeordnet. Somit kann ein Loglevel verschieden formatierte Meldungen an unterschiedliche Kanäle senden. Beispielsweise lässt sich damit die Benachrichtigung per Mail mit anderen Informationen gestalten als die Benachrichtigung in einer Logdatei[71].

Vergleich von Java Logging und Logback

Sowohl Java Logging, als auch Logback besitzen eine hierarchisch organisierte Menge von Loglevels und Möglichkeiten, nach den Levels zu filtern. Dabei ist die Zuweisung von Ausgabekanälen mittels Logback vielseitiger, da über die Konfiguration einem Loglevel mehrere Ausgabekanäle zugewiesen werden können. Die Formatierung der Ausgabe ist mit Java Logging und Logback vielseitiger gestaltbar. Jedoch benutzt Logback im Vergleich zu Java Logging definierte Patterns, wodurch alle Implementierungen einem einzigen Standard folgen. Die Formatierung mittels des DTD von Java Logging ermöglicht zwar jeder Implementierung ein eigene Struktur, jedoch definiert

[69] [vgl. GP12]
[70] [vgl. GP12]
[71] [vgl. GP12]

dadurch jede Implementierung einen eigenen Standard, der im Zusammenspiel mit anderen Loggern zu Kollisionen führen kann.

4.4.2. Bewertung für eine mögliche Implementierung

Java Logging ist nützlich, um schnell ein Logging zu realisieren, das eine enge Bindung an Java hat. Da es aber keine Konfiguration über eine Datei ermöglicht, muss für eine externe Konfiguration eine externe Anwendung genutzt werden. Logback hingegen ist auf die Anwendung und Umgebung einfacher zu konfigurieren und somit ist bei Änderungen des Loggings keine Änderung an der Anwendung nötig. Um eine entsprechende Anpassungsfähigkeit zu ermöglichen, wird daher im Rahmen dieser Arbeit eine Implementierung des Loggings mit Logback erfolgen.

5. Design

Im Rahmen des Designs werden die Verantwortlichkeiten innerhalb des Resourcemanagements aufgezeigt und in Form von Interfaces als Konzept entworfen oder in Form einer konkreten Klasse ausgestaltet. Ebenfalls wird die Modellierung der Daten vorgenommen.

5.1. Aufteilung der Verantwortlichkeiten

Die Aufteilung der Verantwortlichkeiten ist in der objektorientierten Entwicklung eine der Kernentscheidungen bezüglich des Designs. In diesem Teil wird zunächst die grundlegende Idee dargestellt, nach der die Zuordnung einer Verantwortlichkeit zu der daraus resultierenden Klasse vorgenommen wird.

5.1.1. Single Responsibility Principle

Das Single Responsibility Principle sagt aus, dass für eine Klasse oder ein Modul immer nur ein Grund für eine Änderung existieren soll. Folglich soll eine Klasse oder ein Modul nach Möglichkeit auch immer nur eine Verantwortlichkeit besitzen, die eine Änderung auslöst[72]. Dieses Prinzip erzeugt einige Vorteile.

- Aufwand für Maintenance geringer
- Modularisierung der Software
- höhere Wiederverwendbarkeit

Der Aufwand für Maintenance sinkt, da funktionale Fehler schneller zu finden sind. Jede dieser fehlerhaften Funktionalitäten ist einer Verantwortlichkeit zuzuordnen und somit direkt mit einem Modul oder einer Klasse in Verbindung zu bringen. Somit entfällt der Aufwand, mehrere Klassen, die für die identische Funktionalität verantwortlich sind, zu analysieren, um die Fehlerquelle zu identifizieren.

[72] [vgl. Mar09]

Die Modularisierung wird dadurch erzeugt, dass ein Modul und die enthaltenen Klassen nur eine Verantwortlichkeit tragen und somit in dieser Verantwortlichkeit von keinem anderen Modul abhängig sind. Dadurch ist es möglich, einzelne Module auszutauschen, um etwa ein Modul mit gleicher Verantwortung und höherer Performance zu integrieren.

Die höhere Wiederverwendbarkeit ist eine Folge der Modularisierung. Da ein Modul oder eine Klasse immer nur eine Verantwortlichkeit trägt, kann es möglich sein, dass ein Modul oder eine Klasse in anderen Softwareprojekten eingesetzt werden kann. Hierbei ist jedoch zu prüfen, wie stark eventuelle Änderungen an den Schnittstellen des Moduls oder der Klasse ausfallen.

5.1.2. Verantwortlichkeiten innerhalb der Ressourcenverteilung

Innerhalb der Ressourcenverteilung existieren mehrere Aufgaben und somit Verantwortlichkeiten. Eine Ressource wird in Abhängigkeit von aktueller Belegung, Nutzerberechtigung und der auf ihr installierten Umgebung vergeben. Zur Interaktion mit dem User muss eine Anmeldefunktionalität, sowie die Rückgabefunktionalität von Ressource und Session und der Heartbeat bereitgestellt werden. Diese Aufgaben lassen sich in zwei Verantwortlichkeiten einteilen. Die erste Gruppe befasst sich mit der Ressource an sich und ihren Zuständen im Zusammenhang mit der Anfrage und beinhaltet somit folgende Aufgaben.

- Untersuchung der Ressource auf ihren Zustand (*FREE* oder *INUSE*)[73]
- Analyse der in Form von Capabilities gespeicherten Umgebung einer Ressource
- Prüfung der momentanen Ressourcenbelegung für mögliche Freigaben an höher priorisierte Sessions
- Sicherstellung des Zugriffs auf die Ressource vor der Vergabe an den User
- Reservierung der Ressource
- Zuordnung von Ressourcen zu den einzelnen Sessions

Die Verantwortlichkeit wird im Rahmen des Resourcemanagementservers abgebildet, der somit für die globale Verwaltung der Ressourcen zuständig ist.

Die zweite Verantwortlichkeit befasst sich mit der Userinteraktion und der Ausführung einer Anfrage. Um eventuelle Lock-Zustände bei der Vergabe von Ressourcen vorzubeugen, wird die Ausführung mehrerer Anfragen sequentiell und nicht parallel

[73] siehe Kapitel 5.2.2

ausgeführt. Somit liegen folgende Aufgaben in der Verantwortlichkeit der Userinteraktion.

- Anmeldefunktionalität
- Rückgabe einer Session an den Server
- Rückgabe einzelner Ressourcen an eine Session

Diese Verantwortlichkeit wird im Rahmen des `ResourceManagementServers` abgebildet, der somit die Funktionalität zur Ausführung der Anfrage und zur Userinteraktion trägt. Die Darstellung des `ResourceManagmentServers` ist im Abschnitt 5.7 zu finden.

Die Verantwortung der redundanzfreien Datenspeicherung wird an entsprechende Storages gegeben, die genau einmal im System vorhanden sein dürfen. Um die Einmaligkeit zu gewährleisten, eignet sich das Singleton-Pattern[74]. Es existieren Storages für die zu verwaltenden Ressourcen (`SeleniumResourceStorage` in Abbildung 7), User (`SeleniumUserStorage` in Abbildung 8) und Sessions (`SeleniumResourceManagerSessionStorage` in Abbildung 9). Eine Besonderheit ist hierbei der `SeleniumResourceManagerSessionStorage`, da er nicht nur die Sessions hält, sondern auch neue Sessions generiert. Da eine Session im System immer eindeutig sein muss, darf sie auch nur an einer Stelle erzeugt und verwaltet werden. Da die Speicherung schon in einem Singleton realisiert wird, bietet sich dieser Singleton auch für die Erzeugung der Sessions an.

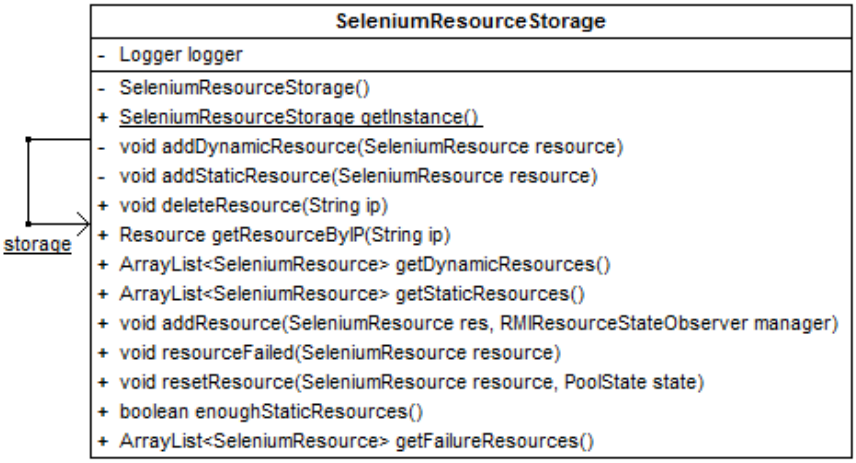

Abbildung 7.: *Klassendiagramm des SeleniumResourceStorage*

[74] [vgl. GHJV95]

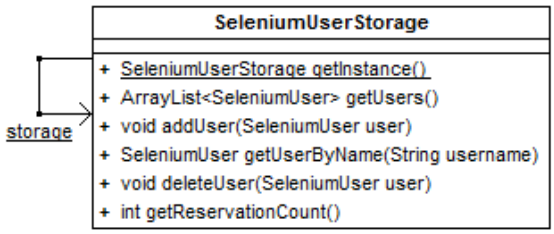

Abbildung 8.: *Klassendiagramm des SeleniumUserStorage*

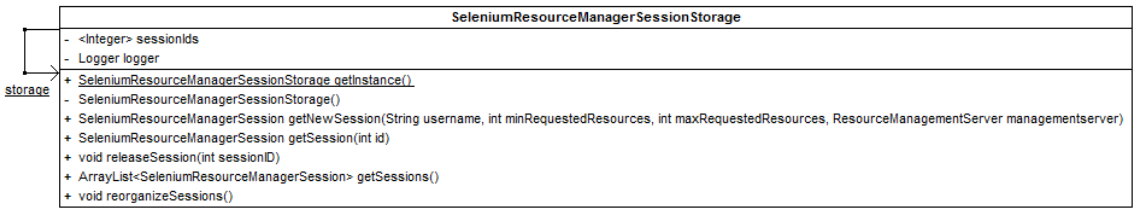

Abbildung 9.: *Klassendiagramm des SeleniumResourceManagerSessionStorage*

5.2. Ressourcen

In diesem Teil wird auf das Design der Ressourcen eingegangen. Dabei wird das Datenmodell bezüglich der installierten Testumgebung und der Zustände, welche eine Ressource innerhalb des Resourcemanagementservers einnehmen kann, erläutert.

5.2.1. Datenmodell einer Ressource

Eine Ressource beinhaltet sowohl Daten bezüglich der installierten Umgebung des virtuellen Rechners, als auch Informationen über die Verfügbarkeit innerhalb der Ressourcenverwaltung. In erster Linie wird eine Ressource nach ihrer Verfügbarkeit unterschieden. Dies ist der Anforderung geschuldet, dass ein Anwender sessionübergreifend eine Mindestanzahl von Ressourcen zugewiesen haben kann. Daher ist eine Ressource entweder in der Menge der fest zugeschriebenen Ressourcen oder in der Menge der frei zu vergebenen Ressourcen einzuordnen. Zudem ist es immer möglich, dass der virtuelle Rechner aus technischen Gründen nicht erreichbar ist (z.B. durch Netzwerkfehler oder abgestürzte Seleniuminstanz). In diesem Fall muss die Ressource einer Menge der defekten Ressourcen zugeordnet werden.
Um die auf dem virtuellen Rechner installierte Testumgebung darzustellen, besitzt

jede Ressource eine Beschreibung in Form von `DesiredCapabilities`, welche in Selenium 2 implementiert ist. Ebenfalls besitzt Selenium Vergleichsobjekte für diesen Datentyp in Form von `CapabilityMatcher`. Somit wird in der Implementierung kein zusätzlicher Aufwand erzeugt, um einen Datentyp mit Vergleichsoperation für die Repräsentation der installierten Umgebung zu entwickeln.

5.2.2. Zustandsmodell einer Ressource

Die aktuelle Nutzung einer dynamischen Ressource kann mit Zuständen beschrieben werden. Dabei sind folgende Zustände denkbar.

- Frei zur Vergabe an eine Session (*FREE*)
- Im Rahmen einer Session in Benutzung (*INUSE*)

Sobald eine Ressource frei zur Vergabe ist, kann sie jeder Session zugeordnet werden. Bei einer Zuordnung erfolgt eine Transition in den Zustand *INUSE*. *INUSE* beschreibt den Zustand, dass die jeweilige Ressource für eine Testausführung in Benutzung ist.

5.2.3. Klassendiagramm

In Abbildung 10 ist das Konzept einer Ressource als Klassendiagramm des Interfaces dargestellt. Auf die konkrete Implementierung wird im Kapitel 6 eingegangen.

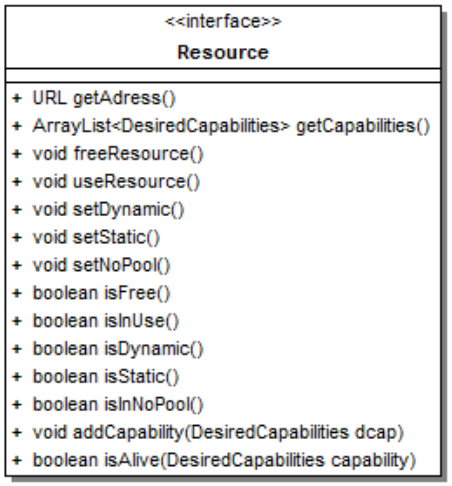

Abbildung 10.: *Klassendiagramm einer Ressource als Interface*

5.3. User

Der User ist aus Sicht des Designs trivial, da er im Kontext des Resourcemanagementservers als Datum zu sehen ist. Ein User dient nur als Repräsentation des Testers und hat somit keine Verantwortung und Funktionalität. Im Folgenden wird das Datenmodell, welches einen User darstellt, kurz erläutert. Ein besonderer Fokus wird dabei auf die Vergabe der Priorität gelegt.

5.3.1. Daten eines Users

Ein User besteht aus einer Mehrzahl von Daten, die ihn im Rahmen der Ressourcenvergabe identifizieren und beschreiben.

Zunächst muss ein User eindeutig identifizierbar sein. Dies kann durch seinen Benutzernamen sichergestellt werden, da in der allgemeinen IT-Landschaft des Anwenders jeder Anwender einen eindeutigen Usernamen besitzt. Da ein Userdatum im Rahmen der Ressourcenvergabe auch zur Authentifizierung genutzt wird, ist es sinnvoll, die Eindeutigkeit eines Users über die Kombination aus Benutzername und Passwort sicherzustellen.

Da bei einer Ressourcenvergabe auf einen Pool von vorreservierten und freien Ressourcen zugegriffen werden kann, muss sichergestellt werden, dass die Ressourcen aus den korrekten Pools gesammelt werden. Ebenso muss beachtet werden, dass ein User nur eine bestimmte Anzahl an festen Ressourcen besitzt. Diese Daten werden als ein Attribut im User abgelegt. Grund hierfür ist, dass bei keiner zugesicherten Ressource die Frage nach dem richtigen Ressourcenpool trivial ist; alle Ressourcen müssen aus dem freien Pool entstammen. Ist die Anzahl der zugesicherten Ressourcen größer null, so muss diese Anzahl an Ressourcen erst aus dem statischen Pool reserviert werden. Um zu wissen, wieviel feste Ressourcen schon für einen User reserviert wurden, wird diese Anzahl im User gespeichert. Dies ist dadurch zu begründen, dass alle Informationen über die Vergabe von fest reservierten Ressourcen an einer Stelle gesammelt sind und somit nicht über mehrere Klassen gesucht werden müssen.

5.3.2. Priorität eines Users

Die Priorität eines Users wird im Userobjekt selbst hinterlegt. Dabei gibt es zwei Möglichkeiten: Zum einen kann eine geringe Menge von festen Prioritätsstufen definiert werden. Die zweite Möglichkeit ist die Festlegung einer Höchstpriorität mit beliebig

vielen Unterprioritäten. Im Design der Prioritäten wird die zweite Möglichkeit genutzt. Bei wenigen festen Prioritätsstufen muss von Anfang an eine klare hierarchische Struktur im abgebildeten Prozess existieren. Eine Änderung an diesen Stufen zieht eine Neuorganisation und Neueinstufung der bisher zugeordneten Prioritäten nach sich. Ebenso kann zur Laufzeit nie eine tiefere Priorität vergeben werden, als die als niedrigst definierte.

Bei der Festlegung einer Höchststufe mit beliebig vielen Unterstufen hingegen entfällt die Einführung von neuen Stufen und die damit verbundene Neuorganisation. Ebenso kann zur Laufzeit eine Herabstufung durch Änderung der Priorität erfolgen, da es keine festen Stufen gibt. Dies ist beispielsweise in Unix in Form der Nicevalues bei Prozessen implementiert.

Im Rahmen des Resourcemanagementservers wird die höchste Priorität für einen User mit 1 angegeben und die Herabstufung erfolgt durch eine Inkrementierung des Wertes der Priorität.

5.3.3. Klassendiagramm

Das Klassendiagramm in Abbildung 11 stellt das Konzept eines Users in Form des Interfaces `TestautomationUser` dar.

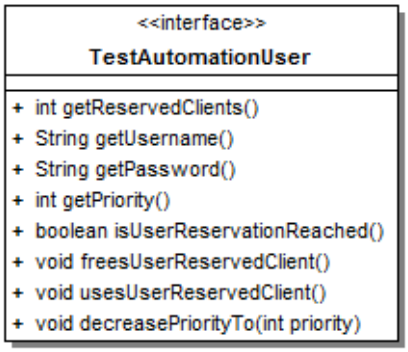

Abbildung 11.: *Klassendiagramm eines TestautomationUsers*

5.4. Session

In diesem Teil wird das Datenmodell einer Session, die Gültigkeit einer Session und die Verantwortlichkeit und Funktion erläutert.

5.4.1. Verantwortlichkeiten und Aufgaben

Die Session ist mehr als nur ein reines Datum. Sie ist dafür verantwortlich, Nachrichten über die Änderung der ihr zugeordneten Ressourcen zu melden. Dafür wird eine Funktionalität benötigt, die eingegangene Nachrichten auswertet und entsprechend die Benachrichtgung ihres Besitzers einleitet. Die Auswertung muss in einer Art Filterfunktion implementiert werden, da zwei Arten von Nachrichten existieren, die jeweils einen Zustand der Ressource abbilden. Bei einer freien Ressource hat die Session die Aufgabe, die Ressource dem Managementserver zur Neuverteilung bekannt zu geben. Somit ist die Ressource für die Verwaltung der ihr zugewiesenen Ressourcen verantwortlich.

5.4.2. Datenmodell

Innerhalb einer Session werden mehrere Daten gehalten. Klar ersichtlich ist, dass der Nachrichtenempfänger gespeichert werden muss, um die Nachrichten zuzustellen. Ebenso muss eine Session eindeutig identifizierbar sein. Hierfür wird eine zu jedem Zeitpunkt einmalige ID vergeben. Die ihr zugeordnete Anzahl an Ressourcen wird auch gespeichert, da für diese Information sonst über alle vorhandenen Ressourcen mit Kenntnis der SessionID iteriert werden müsste, um festzustellen, wieviel Ressourcen einer Session zugeordnet sind. Dies ist bei vielen Ressourcen zeitintensiv. Ebenfalls kennt die Ressource die für sie minimal und maximal angefragte Anzahl an Ressourcen. Ebenfalls wird die Priorität der Session, die sie durch ihren User erhält, und der Name des Users gespeichert. Der User könnte auch als Referenz übergeben werden. Dies wird hier aber nicht realisiert, da bei einem Neuladen des referenzierten Users in den Userstorage die Referenz verloren gehen würde. Somit wäre die Session verwaist und müsste aufgeräumt werden, wodurch der Test des Users abbrechen könnte. Ein solcher Abbruch ist bei langen Tests (z.B. Regressiontests) sehr problematisch, da im schlimmsten Fall ein Großteil der Testdauer des wiederholt durchzuführenden Tests für das Erzeugen bekannter Ergebnisse genutzt werden muss. Aus einem ähnlichen Grund wird die Userpriorität in der Session hinterlegt. Dadurch muss bei einem Entfernen einzelner Ressourcen der User nicht aufgelöst werden, um einen Vergleich von Prioritäten durchzuführen. Zusätzlich kann ein User oder ein Administrator die Session zur Laufzeit neu priorisieren, um anderen Tests Vorzüge bei der Ressourcenzuweisung zu ermöglichen.

5.4.3. Gültigkeit einer Session

Eine Session besitzt immer einen Zeitraum, in dem sie gültig ist. Dies ist besonders bei Kommunikation über Netzwerke wichtig, da Verbindungen unterbrochen und somit

Sessions verwaisen können. Um dem entgegen zu wirken, werden zwei Gültigkeiten für eine Session definiert.

- Eine Session ist solange gültig, bis ein User sie auflöst
- Eine Session ist solange gültig, bis ein Timeout im Heartbeat erfolgt

Der Heartbeat sichert ab, dass eine Verbindung zwischen Tester und Resourcemanagementserver existiert. Zwar basiert die Kommunikation auf RMI und somit ist eine Verbindung über TCP vorhanden, jedoch ist dies nur ein Timeout auf Netzwerkebene. Da die Verbindung für das Versenden der Nachrichten via RMI über Methodenaufrufe und somit mittels Objektreferenzen stattfindet, greift der TCP-Timeout bei einem Referenzverlust nicht. Daher muss der Tester in einem definierten zeitlichen Abstand (Heartbeat-Timeout) Nachrichten an den Resourcemanagementserver schicken. Fällt eine Nachricht aus, erfolgt ein Timemout und die Session ist ungültig.

5.4.4. Klassendiagramm

Im Folgenden wird das Konzept der Session in Form des Interfaces als Klassendiagramm dargestellt. Auf die konkrete Session wird im Kapitel 6 eingegangen.

Abbildung 12.: *Klassendiagramm einer Session*

5.5. Kommunikation über Nachrichten

In diesem Teil wird das Design der Kommunikation über Nachrichten behandelt. Dabei wird das Observer-Pattern im allgemeinen und in der Implementierung innerhalb Javas erläutert. Da die Kommunikation über RMI implementiert wird (s. Kapitel 4.2.3), werden die nötigen Anpassungen des Observer-Patterns für eine Kommunikation über RMI ebenfalls in diesem Kapitel diskutiert werden.

Die Ausgestalltung der Nachrichten sowie die Aktionen und Reaktion der einzelnen Kommunikationsteilnehmer werden ebenfalls im folgenden Kapitel dargelegt.

5.5.1. Das Observer-Pattern

Das Observer-Pattern ist ein, in der objektorientierten Softwareentwicklung bekanntes Standardmodell[75], um Softwaresysteme zu entwickeln, die auf Nachrichten basieren. Im Folgenden werden die Eigenschaften des Observer Patterns dargelegt.

Allgemeiner Aufbau des Observer-Patterns

Es existieren im Observer-Pattern zwei Typen von Klassen; der Observer und das Subject. Hierbei gilt es zunächst Beziehungen der Form 1 zu n zu identifizieren, da das Observer-Pattern besagt, dass eine Anzahl von n Observern eine Nachricht oder Änderung von einem Subject erwarten. Um die Implementierung wiederverwendbar zu gestalten, muss die Verbindung zwischen Observer und Subject über einheitliche Interfaces erfolgen. Daher muss ein Subject folgende Funktionen für einen Observer bereitstellen.

- Anmelden zum Nachrichtenempfang
- Abmelden vom Nachrichtenempfang
- Benachrichtigung der Observer über Änderung

Es zeigt sich also, dass ein Subject alle Observer kennt und sie somit aus technischer Sicht referenziert. Um die Benachrichtigung der Observer durch das Subject zu ermöglichen, muss der Observer eine Funktionalität dafür bereitstellen. Die Ausgestalltung dieser Funktionalität unterscheidet sich in zwei Konzepten.

Das erste Konzept ist das Pull-Konzept. Hierbei hat der Observer die Verantwortung, die Änderung des Subjects zu erfragen. Das Subject schickt daher nur eine

[75] [vgl. GHJV95]

Nachricht, dass eine Änderung stattgefunden hat. Dies Konzept ist sehr sinnvoll, wenn die konkrete Änderung des Subjects losgelöst von dem Status des Observers ist. Dies wäre zum Beispiel bei einem getaktetem System nützlich, da der Observer bei einem neuen Taktsignal einen Automaten durchlaufen könnte. Die internen Daten des Subjects, also des Taktes, sind dabei für den Observer nicht von Interesse.

Das entgegengesetzte Konzept ist das Push-Konzept. Hierbei wird der Observer seitens des Subjects nicht ausschließlich über eine Änderung informiert, sondern auch darüber, was sich geändert hat. Dies könnte beispielsweise bei zwei asynchronen Automaten vorteilhaft sein, sobald der Observerautomat vom inneren Zustand seines Subject-Automaten abhängt.

Für das allgemeine Observer-Pattern bedeutet das nun, dass ein Subject bei einer Änderung seine Observer je nach Benachrichtigungskonzept mit den entsprechenden Daten aufrufen muss. In einem Klassendiagramm stellt sich diese Beziehung wie folgt dar[76].

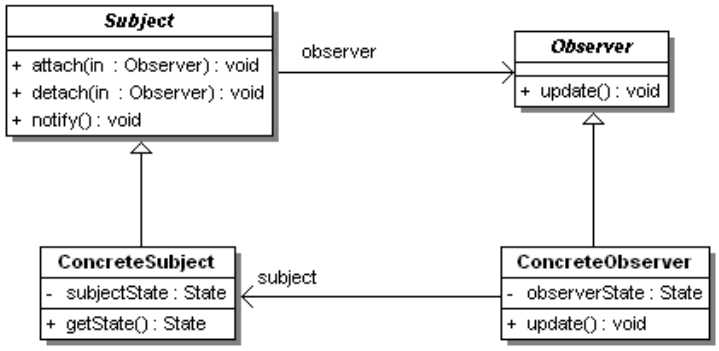

Abbildung 13.: *Klassendiagramm Observer-Pattern mit Pull-Konzept*

Implementierung in Java

Das Observer-Pattern wird in Java besonders in der Entwicklung der GUI genutzt. Hierbei werden sogenannte Listener implementiert, die innerhalb von MVC[77] die Rolle eines Controlers übernehmen und somit das Datenmodell über Änderungen der Daten innerhalb GUI informieren. Zudem stellt Java eine Möglichkeit bereit, ein javaseitig standartisiertes Observer-Pattern zu implementieren. Hierfür wird das Interface `Observer` und die abstrakte Klasse `Observable` genutzt. Das Interface schreibt die Methode `update(Observable o, Object arg)` vor. Wie zu erkennen ist,

[76] [vgl. GHJV95]
[77] Model-View-Controler

legt sich Java nicht auf Pull- oder Push-Konzept fest, sondern gibt dem Entwickler die Möglichkeit, durch entsprechendes Nutzen der Argumente das gewünschte Verhalten zu implementieren.

Die Klasse `Observable` beschreibt das Subject und bietet konkrete Methoden für die Verwaltung und Benachrichtigung der Observer an. Wenn ein Entwickler diese Methoden überschreiben will, sollte man die zusätzliche Methode, die die Zustandsänderung des Subjects beschreibt (`setChanged()`) beachten. Die Abbildung 14 zeigt das Klassendiagramm des Observer-Patterns in der Javaimplementierung.

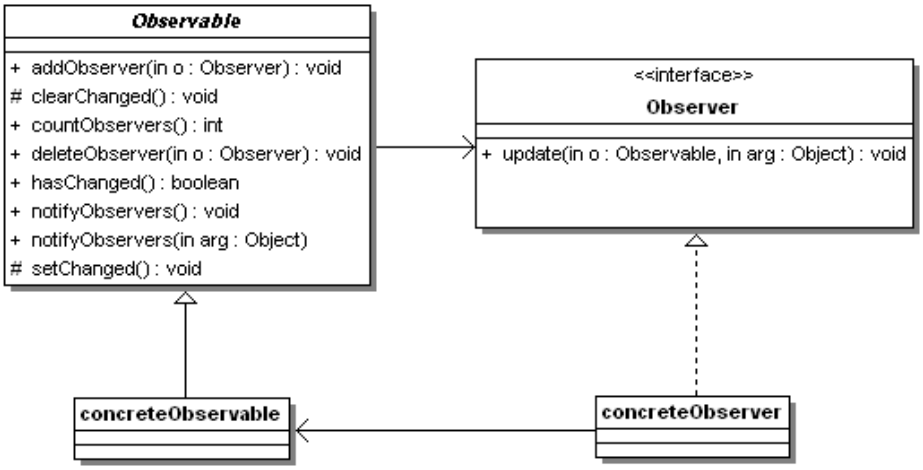

Abbildung 14.: *Klassendiagramm Observer-Pattern in der Javaimplementierung*

Die Implementierung seitens Java ist im Rahmen dieser Arbeit nicht hilfreich, da sie nicht für eine Kommunikation mit entfernten Anwendungen geschaffen ist. Die Gründe dafür und notwendigen Anpassungen werden im folgenden Teil erläutert.

Anpassung an Kommunikation mit entfernten Anwendungen

Die Implementierung des Observer-Patterns in Java ist nicht für eine Kommunikation über RMI geeignet. Dies ist technisch begründet. In der allgemeinen Beschreibung des Patterns wird dargelegt, dass ein Subject seine Observer für die Benachrichtigung referenzieren muss. Soll nun über RMI kommuniziert werden, muss der Observer ein Object vom Typ `Remote` und serialisierbar sein. Javas Observer-Interface implementiert aber diese Eigenschaft nicht. Somit muss entweder zu Lasten der Wiederverwendbarkeit die konkrete Implementierung, welche die notwendigen Interfaces implementiert, referenziert werden oder eine eigene Implementierung des Observer-Patterns entwickelt werden.

Observer-Pattern innerhalb des Resourcemanagementservers

Innerhalb des Servers kann für den Nachrichtenaustausch zwischen Ressource und Session ebenfalls das Observer-Pattern genutzt werden. Da diese Komponenten in der gleichen JRE lokalisiert sind, wird für die Kommunikation kein RMI verwendet.

Die Menge der Datenänderung ist sehr klein. Es können sich nur die zugeordneten Sessions und der interne Status der Ressource ändern. Somit kann auch hier mit dem Push-Konzept gearbeitet werden. Die Java-Implementierung des Patterns wird jedoch nicht genutzt. Die Abfragen, ob genügend Änderungen für eine Nachricht vorhanden sind, ist bei der geringen Menge an geänderten Daten pro Nachricht nicht sinnvoll. Daher wird eine simple und spezialisierte Variante als Eigenimplementierung vorgenommen.

In dieser Anwendung werden zwei Ziele observiert. Auf dem Resourcemanagementserver beobachten die Sessions und der Manager selber immer den Zustand einer Ressource. Somit wird hier ein `ResourceStateObserver` implementiert, der Methoden zur Signalisierung der Zustände besitzt. Von ausserhalb des Servers wird der Resourcemanagementserver durch das Framework AludraTest beobachtet. Der Server muss das Framework benachrichtigen, wenn eine Session verfügbar ist, wenn eine Ressource innerhalb einer Session verfügbar ist und wenn eine Anfrage abgelehnt wird.

Die dazugehörigen Interfaces und die jeweiligen Observables sind in den Abbildungen 15 und 16 als Klassendiagramm dargestellt.

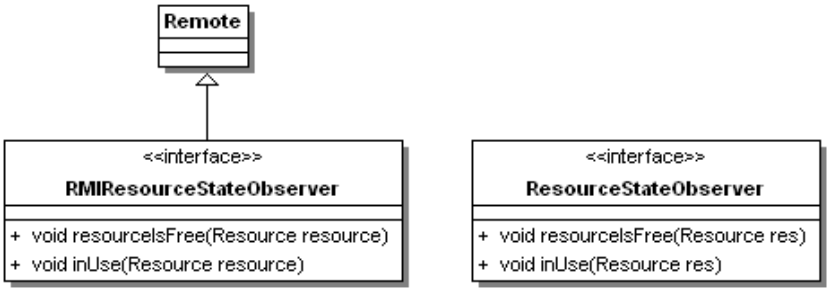

Abbildung 15.: *Klassendiagramm des Interfaces vom internen Observers*

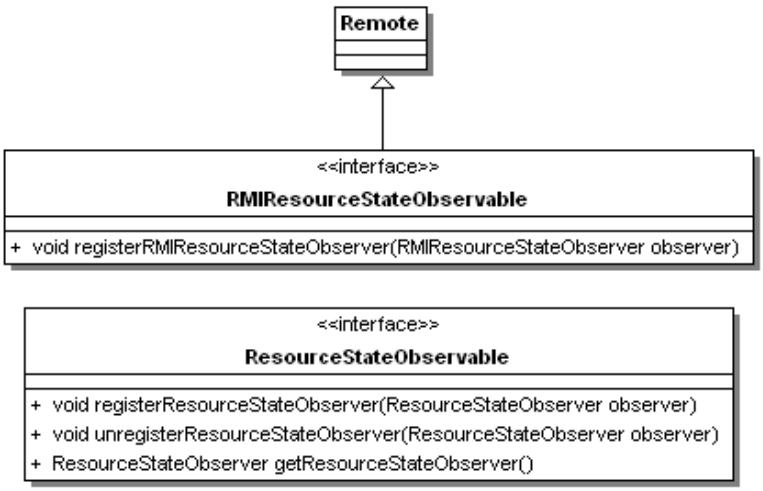

Abbildung 16.: *Klassendiagramm des Interfaces vom internen Observable*

Observer-Pattern ausserhalb des Resourcemanagementservers

Kommunikation zwischen dem Server und der Aussenwelt soll mittels RMI und einem angepassten Observerpattern erfolgen. Dabei wird der Resourcemanagementserver das `RMIManagementObservable` Interface aus Abbildung 17 implementieren, dass ihm die entsprechenden Eigenschaften zuweist.

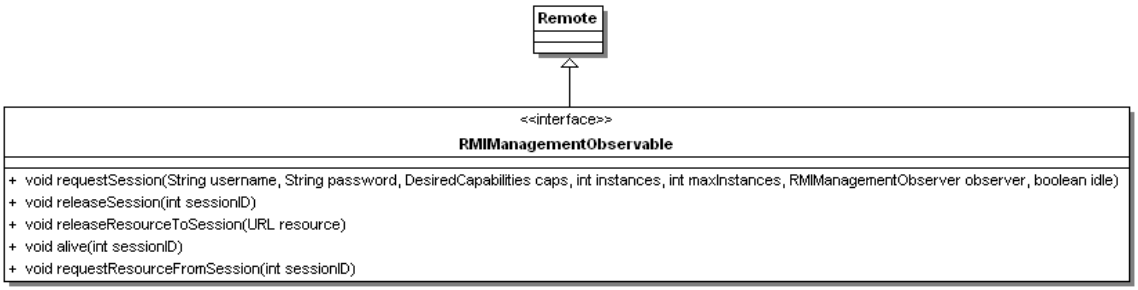

Abbildung 17.: *Klassendiagramm des Interfaces RMIManagementObservable*

Der Beobachter muss das Interface `RMIManagementObserver` implementieren, um Nachrichten vom Resourcemanagementserver erhalten zu können.

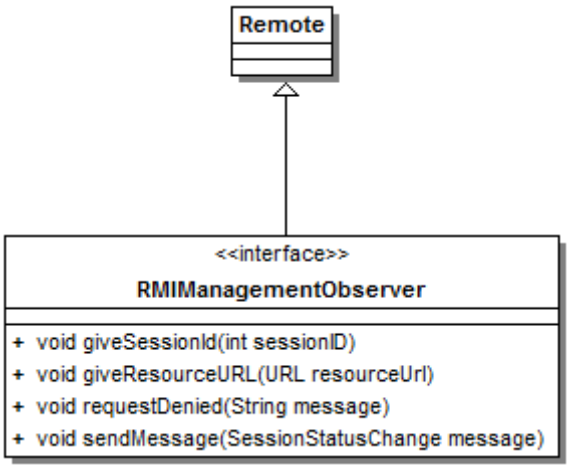

Abbildung 18.: *Klassendiagramm des Interfaces RMIManagementObserver*

5.5.2. Nachrichten

Um die Änderungen eines Observables für die Observer zu dokumentieren, werden Methoden definiert. Diese Methoden sind die zu übertragende Nachrichten und somit je nach Empfänger unterschiedlich implementiert.

Fokus des Testers

Der Tester benötigt drei Informationen, die durch Nachrichten übermittelt werden müssen.

- ID einer Session
- URL der durch die Session belegten Ressource
- Ablehnung einer Anfrage

Diese Nachrichten werden durch die folgenden Methoden dargestellt.

- `giveSessionID(int sessionID)`
- `announceResourceURL(URL resourceURL)`
- `requestDenied(SessionStatusChange message)`

Der Typ `SessionStatusChange` beinhaltet hierbei die ID einer Session des Observers, sowie die momentane Anzahl der gespeicherten Ressourcen. Dies ermöglicht dem Observer auf Änderungen der ihm zugewiesenen Menge an Ressourcen zu reagieren.

Fokus des Resourcemanagementservers

Der Managementserver benötigt Informationen von zwei Quellen. Eine Quelle sind die Ressourcen, die nicht in einer Session verwaltet werden. Die zweite Quelle sind die angemeldeten Observer. Der Server benötigt die folgenden Informationen.

- Status der Ressource
- Lebenszeichen für die Sessions eines Observers

Die Nachrichten, die diese Informationen enthalten, werden durch die folgenden Methoden dargestellt.

- `resourceInUse(Resource resource)`
- `resourceIsFree(Resource resource)`
- `alive(int sessionID)`

5.5.3. Verteilung der Rollen

Um das Observer-Pattern zu implementieren, muss zunächst eine Einteilung der Kommunikationsteilnehmen in die Rollen der Observer und Observables vorgenommen werden. Da hier zwei Observable-Rollen vorliegen, muss eine Unterscheidung nach `ResourceStateObservable` und `RMIManagementObservable` bzw. deren Observern vorgenommen werden. Die Einteilung wird in Tabelle 12 dargestellt.

Observable	Observer
ResourceState	Session
	Resourcemanagementserver
ManagementObservable	AludraTest-Instanz

Tabelle 12.: *Aufteilung der Kommunikationsteilnehmer in die jeweiligen Observable und Observer*

Ein Sonderfall existiert bei der Implementierung der Ressource in das Pattern. Die Ressource besitzt in der detaillierten Ausformulierung zwei mögliche Observer. Der `ResourceStateObservable` in Form des Managementservers muss für die Kommuniaktion über RMI erreichbar sein, wodurch er nur Methoden besitzen darf, die `RemoteExceptions` werfen können. Daher muss auch seine Implementierung des `ResourceStateObservers` diese Exceptions werfen können, wodurch ein `RMIResourceStateObserver` benötigt wird. Somit hat die Ressource zwei Observerreferenzen. Eine für den `ResourceStateObserver` und eine für den `RMIResourceStateObserver`.

5.6. Logging

In diesem Kapitel wird das Logging aus Sicht des Designs behandelt. Dabei wird auf die Ansprache des Loggers und die Ausgabe der jeweiligen Loglevels eingegangen.

5.6.1. Ansprache des Loggers mit SLF4J

SLF4J (Simple Logging Facade for Java) ist eine API, die eine Fassade zur Ansprache des Loggers bereitstellt. Im Rahmen dieser Arbeit wird nur ein Logger für das gesamte System genutzt, der in alle Instanzen weitergereicht wird. Dadurch wird auch nur eine Konfiguration benötigt und somit steigt die Komplexität des Gesamtsystems nur minimal.

SLF4J unterstützt mehrere Logging-APIs, zu denen auch Logback gehört. Um einen Logger zu erhalten, wird eine Factory genutzt, die einen Logger mit einem vom Entwickler festgelegtem Namen liefert. Um sicherzustellen, dass alle Objekte im Resourcemanagementserver den identischen Logger benutzen, wird der Logger von einer zentralen Klasse erzeugt, gespeichert und als Referenz weitergegeben. Die Ansprache erfolgt dann direkt auf dem Logger-Objekt, dem dann Nachrichten für entsprechende Loglevels übergeben werden können.

5.6.2. Konfiguration der Loglevels

Die Loglevels werden in einer XML-Datei konfiguriert. Dabei wird ein globales Rootlevel angegeben, das den Mindestloglevel angibt. In diesem System wird der Level *DEBUG* als Rootlevel genutzt. Mit *TRACE* wäre noch ein niedrigerer Level möglich, dieser ist jedoch in dieser Anwendung nicht genutzt, da eine Verfolgung einer Anfrage oder Ressource nicht geloggt werden soll.

Die Ausgabe der Loglevels erfolgt für alle Levels auf der Konsole und im zentralen Logfile `logfile.log`. Loggingeinträge mit dem Level *ERROR* erfolgen zusätzlich in einem speziellen Logfile `errorlog.log` und werden dort mit Auftrittsort im Code dargestellt. Loggingeinträge mit dem Level `WARN` werden mit Angabe der Java-Datei, in der sie auftreten, im zentralen Logfile gespeichert. Um Änderungen am Logger selber vorzunehmen, wird Logback über das Attribut `scan` so konfiguriert, dass die Konfigurationsdatei alle drei Minuten neu eingelesen wird. Die Konfigurationsdatei ist im Anhang durch das Listing 22 dokumentiert.

5.7. Resourcemanagementserver

Der Resourcemanagementserver stellt die zentralen Funktionalitäten zur Verwaltung der Ressourcen und der Kommunikation mit AludraTest zur Verfügung.

5.7.1. Datenmodell des Resourcemanagementservers

Der Resourcemanagementserver selber besitzt eine Liste aller Observer, eine Map für die Auflösung der Verbindung zwischen Session und Usernamen, und eine Map zur Auflösung der Verbindung zwischen Usernamen und Observer. Um Zugriff auf alle Daten innerhalb des Resourcemanagements zu haben, existieren Referenzen zu den einzelnen Storages. Diese Daten sind in der konkreten Implementierung in Form der Klasse `SeleniumResourceManagementServer` zu finden.

Die enthaltenen Methoden werden von drei Interfaces vorgeschrieben. Das Interface `ResourceManagementServer` schreibt die Methoden für die Ressourcenverwaltung vor. Selbst erweitert das Interface zwei weitere Interfaces. Dabei gibt das Interface `RMIManagementObservable` die Methoden zur clientseitigen Ansprache des Servers vor. Das im Abschnitt 5.5.3 erwähnte Interface `RMIResourceStateObserver` stellt eine Variante des `ResourceStateObservers` da, die für die Implementierung von Objekten vorgesehen ist, die im RMI-Kontext stehen.

5.7.2. Klassendiagramm des Resourcemanagementservers

Die Abbildungen 19 und 20 zeigen die Klassendiagramme des Resourcemanagementservers.

Abbildung 19.: *Klassendiagramm des* `ResourceManagementServer`*-Interfaces*

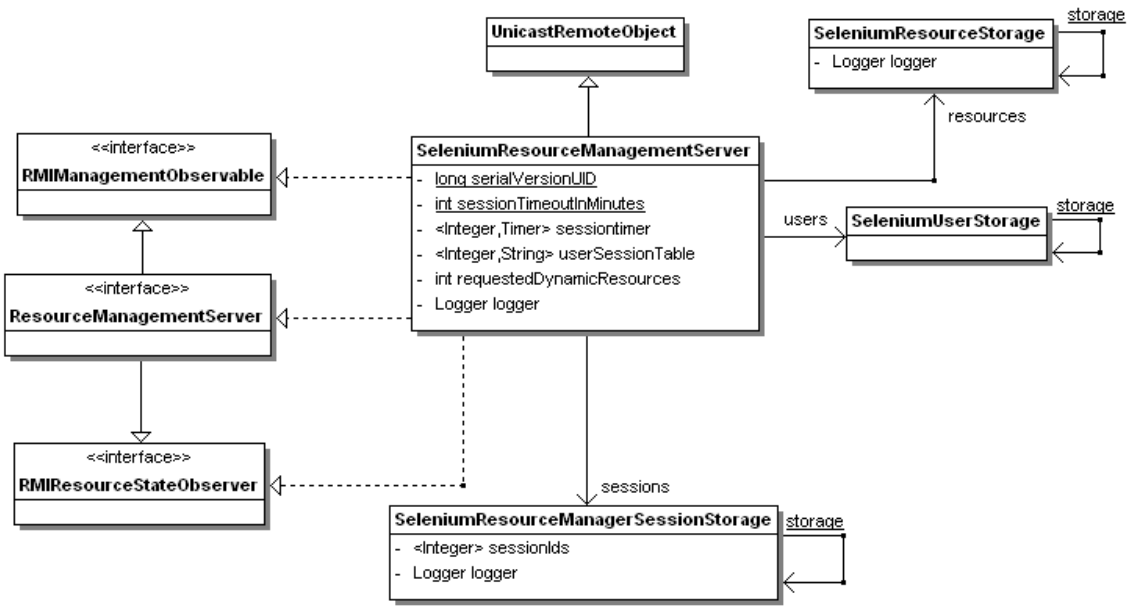

Abbildung 20.: *Klassendiagramm des konkreten* `SeleniumResourceManagementServer` *dargestellt ohne seine Methoden*

6. Realisierung

In diesem Kapitel wird die Umsetzung des Designs in die Implementierung behandelt. Dabei wird die Integration der einzelnen Teile des Systems zu einem Gesamtsystem, die Implementierung von RMI und die Implementierung der Konfigurationsschnittstellen behandelt.

6.1. Integration zu einem Gesamtsystem

Die im Kapitel 5 definierten Klassen und Interfaces bilden im Zusammenspiel ein System zum Resourcemanagement, das hier behandelt werden soll. Das komplette Klassendiagram ist im Anhang in Abbildung 31 dargestellt.

6.1.1. Storages

Die Storages sind als Singleton-Pattern[78] entworfen, da sie exakt einmal im System vorhanden sein dürfen, um eine redundante Speicherung der enthaltenen Daten zu vermeiden.

Jeder Storage besitzt eine Methode `getInstance()`, welche die Referenz auf das Singleton zurück gibt. Diese Methode ist klassengebunden (**static**) und somit an nur einer Stelle zu erreichen. Intern hält die Klasse eine Referenz auf ein Objekt ihres Storagetyps. Diese Referenz wird nur beim ersten Aufruf von `getInstance()` gesetzt. Zu jedem folgenden Zeitpunkt wird die Referenz nur zurückgegeben. Das folgende Listing beschreibt anhand des `SeleniumResourceStorages` diese Methode.

```
1 private static AludraResourceStorage storage=null;
2 public static synchronized AludraResourceStorage getInstance(){
3         if(storage == null){
```

[78] [vgl. GHJV95]

```
4                   storage = new AludraResourceStorage();
5           }
6           return storage;
7   }
```

Listing 11: *Beispiel einer getInstance()-Methode*

Zur Absicherung, dass durch zeitgleichen, systemweiten Erstzugriff von zwei oder mehr Threads mehrere Storages des gleichen Typs instanziert werden, ist die Methode durch das Schlüsselwort **synchronized** thread-safe implementiert.

Alle Storages speichern die Objekte eines Typs, für den sie im Rahmen ihrer Verantwortlichkeit definiert worden. Eine Ausnahme hierbei ist der `SeleniumResourceManagerSessionStorage`.

Dieser hält zusätzlich zu den Sessions eine Liste der vergebenen IDs. Dies ist sinnvoll, da der Storage als einziger neue Sessions erstellen darf. Somit muss intern eine Kenntnis über vorhandene IDs bestehen. Eine ID wird nach folgendem Activitydiagram erstellt.

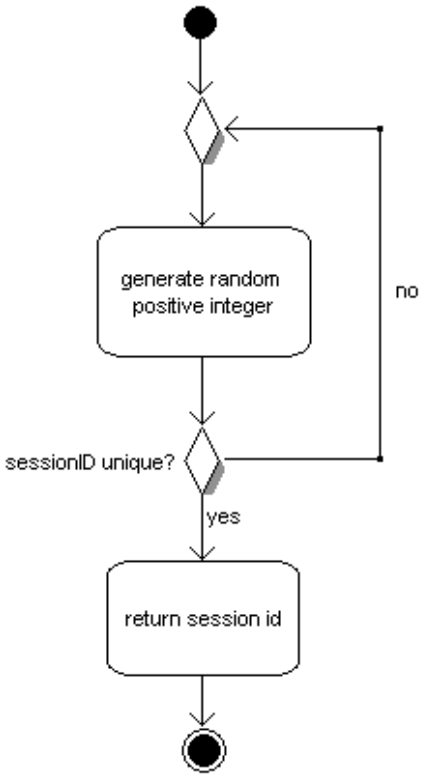

Abbildung 21.: *Activity Diagram - Ablauf der Erstellung einer SessionID*

Alternativ könnte die SessionID bei jeder neuen Session um 1 inkrementiert werden. Ein Nachteil ist, dass freigegebene IDs nicht neu vergeben werden, wenn keine zusätzliche Suche nach Lücken in der sortierten Reihenfolge der IDs erfolgt. Eine Suche nach bestimmten IDs wird auch im dargestellten Activitydiagram der Abbildung 21 benötigt. In beiden Lösungsmöglichkeiten ist ein Abbruch der Suche bei einer gefundenen Lücke bzw. der generierten ID ausreichend, wodurch nur im Extremfall die gesamte Liste durchlaufen werden muss. Hierbei ist es aber gleichgültig, ob die Liste der vergebenen IDs sortiert ist, wodurch die Implementierung einer geeigneten Sortierung entfällt. Da in Java bei Objekten ein modifizierter Mergesort zum Einsatz kommt[79], in der Liste jedoch in Objekten gespeicherte primitive Daten vorliegen, wäre es sinnvoll, in der alternativen Lösung einen performanteren Suchalgorithmus zu implementieren. Dieser Aufwand entfällt aber bei dem im Activitydiagram dargestellten Verfahren.

[79] [vgl. Ou11]

6.1.2. Ressourcen

Das Interface `Resource` wird für das Framework Selenium in der konkreten Klasse `SeleniumResource` implementiert. Im Folgenden werden die Kernpunkte der Realisierung dargelegt. Dazu zählt die Implementierung des Observer-Patterns und die Implementierung des Wechsels einer Ressource zwischen zwei Sessions.

Implementierung des Observer-Patterns

Das Observer-Pattern setzt voraus, dass ein Observable seine Observer kennt. Dies ist für eine Ressource durch eine einfache Referenz je Observertyp implementiert. Eine Ressource ist immer entweder einer Session oder Server zugeordnet, wodurch keine Liste an Observern benötigt wird.

Die Benachrichtigung der Observer durch den entsprechenden Methodenaufruf wird beim Setzen eines Zustandes erfolgen. Beim Setzen eines Zustands muss jedoch vorher ein Observer gesetzt sein. Ist dies nicht der Fall wird ein Eintrag durch das Logging mit dem Loglevel *ERROR* erfolgen. Abbildung 22 zeigt die konkrete Ressource als Klassendiagramm ohne Darstellung der enthaltenen Methoden.

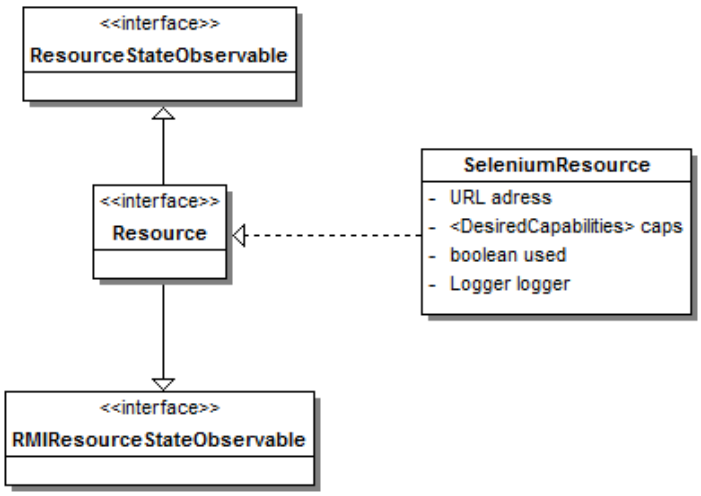

Abbildung 22.: *Klassendiagramm der konkreten Ressource als Klasse SeleniumResouce*

Wechsel einer Ressource zwischen Sessions

Um eine Ressource zwischen Sessions zu tauschen, sind zwei zentrale Probleme zu beachten.

- Es dürfen nur dynamische Ressourcen zwischen Sessions wechseln
- Ressourcen dürfen nur aus Sessions geringerer Priorität entnommen werden
- Es dürfen nur Ressourcen im Status *FREE* repriorisiert werden
- In einer Session muss die minimal angefragte Menge an Ressourcen enthalten bleiben

Um diese Probleme zu lösen, wird die Session jede dynamische Ressource, die ihr den Zustand *FREE* signalisiert, an den Resourcemanagementserver zur Repriorisierung geben, solange mehr Ressourcen in der Session enthalten sind, als minimal angefragt wurden. Dieser prüft dann bei der Ressource, ob Sessions mit besserer Priorität vorliegen. Ist dies der Fall, wird die Session informiert, die entsprechende Ressource an die wichtigere Session zu übermitteln. Existiert keine besser priorisierte Session, so wird die Ressource an die Session zurückgegeben, bzw. zu den freien Ressourcen hinzugefügt, wenn die Session aufgelöst wurde.

6.1.3. Session

Das Interface `Session` wird für diese Anwendung in der Klasse `SeleniumResourceManagerSes` konkretisiert. Im Folgenden wird die Implementierung der Klasse `SeleniumResourceManagerSe` im Rahmen der lokalen Ressourcenverwaltung erläutert. Abbildung 23 zeigt das Klassendiagramm der `SeleniumResourceManagerSession` ohne ihre Methoden.

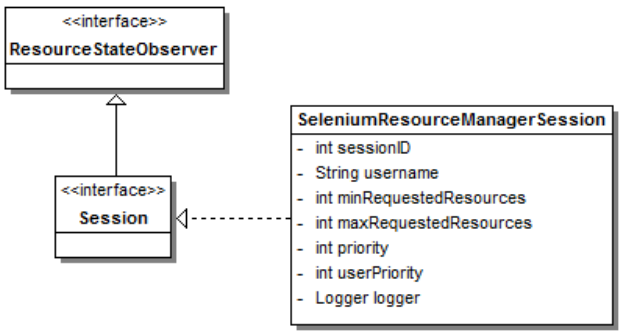

Abbildung 23.: *Klassendiagramm der konkreten Session durch die Klasse* `SeleniumResourceManagerSession`

Lokale Ressourcenverwaltung

Die Session verwaltet die ihr zugewiesenen Ressourcen. Dabei gelten definierte Regeln.

- Eine Session ist als voll markiert, wenn ihr Ressourcenmaximum erreicht ist
- Eine Session hat, bis die minimale Anzahl an Ressourcen vorhanden ist, die maximale Priorität
- Freie Ressourcen sind zur Repriorisierung zu geben, bis die Untergrenze der zu verwaltenden Ressourcen erreicht ist

Um Ressourcen an den Manager zurückzugeben, wird er als direkte Referenz in jeder Session gespeichert. Um die Untergrenze der in einer Session gespeicherten Ressourcen nicht zu unterschreiten, wird die Session vor jeder Freigabe einer Ressource zur Repriorisierung prüfen, ob es sich um eine Minimalressource handelt. Ist dies der Fall, wird sie AludraTest über den Manager direkt als verwendbar bekanntgegeben.

Änhlich ist das Verhalten für die Priorität einer Session definiert. Wird eine Session erstellt, ist die Priorität solange maximal, bis die Mindestanzahl an Ressourcen einer Session hinzugefügt wurde. Ist diese Anzahl erreicht, wird die aktuelle Priorität durch die Priorität des Users ersetzt. Dieser Mechanismus sorgt dafür, dass bei der Repriorisierung einer Ressource immer die Sessions im Wartemodus bevorzugt befüllt werden.

Die für die Ordnung benötigte Vergleichsmethode basiert daher auf der Priorität. Hier wird nach umgekehrter Priorität sortiert, da ein niedriger Wert eine höhere Priorität darstellt. Die Implementierung ist in Listing 12 dargestellt.

```
1 public int compareTo(Session o) {
2         if((o==null) || (priority > o.getPriority())){
3                 return 1;
4         }
5
6         if(priority < o.getPriority()){
7                 return -1;
8         }
9         return 0;
10 }
```

Listing 12: *Gleichheit zwischen zwei Sessions*

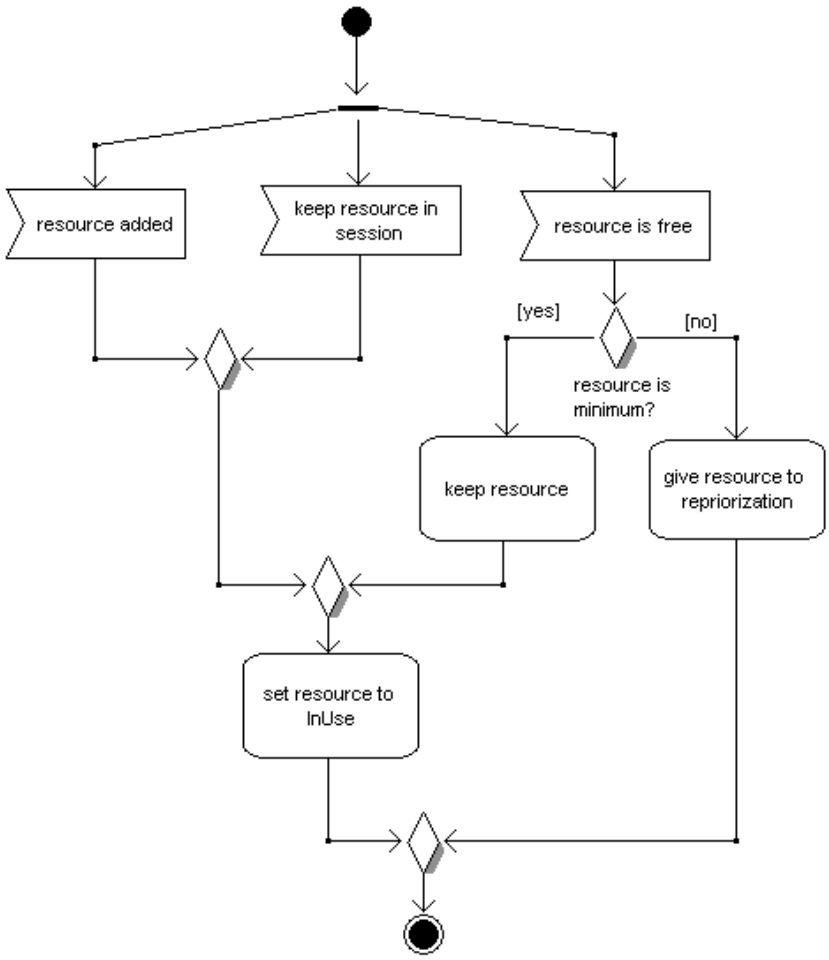

Abbildung 24.: *Activity Diagram - lokales Ressourcemanagement durch eine Session*

Benachrichtigung der AludraTest-Instanzen

Eine Session kann die ihr zugeordnete AludraTest-Instanz nur indirekt mittels des Resourcemanagementservers informieren. Eine Session hat nur einen einzigen Grund für eine Benachrichtigung. Wird ihr eine Ressource zugewiesen oder darf sie eine freie Ressource behalten, muss sie der AludraTest-Instanz diese Ressource mitteilen.

6.1.4. Nachrichten

Die Nachrichten, die innerhalb der Ressourcenverwaltung versendet werden, sind durch Methoden implementiert. Dabei reagieren Sessions und der Resourcemanage-

mentserver unterschiedlich auf die Nachrichten.

Reaktion auf Nachricht einer freien Ressource

Wird eine Ressource ihre Observer über den Zustand *FREE* benachrichtigen, so erfolgt eine unterschiedliche Reaktion. Eine Session gibt eine freie Ressource an den Resourcemanagementserver zurück, der die Ressource entweder neu verteilt oder der Session zurückgibt. Erhält der Resourcemanagementserver jedoch die Nachricht von einer Ressource, dass sie frei ist, wird er nichts unternehmen. Dies ist sinnvoll, da der Resourcemanagementserver nur von Ressourcen benachrichtigt werden darf, die im Zustand *FREE* sind.

Reaktion auf Nachricht einer genutzten Ressource

Sendet eine Ressource eine Nachricht betreffend des Zustands *INUSE* an ihre Observer, so erfolgen unterschiedliche Reaktionen. Die Session prüft in diesem Fall, ob die Nachricht von einer der ihr zugeordneten Ressourcen stammt und schreibt im Fehlerfall einen Eintrag ins Errorlog. Erhält der Resourcemanagementserver die Nachricht, so wird ein Fehler bei der Ressourcenzuteilung erkannt und durch das Logging mit dem Loglevel *ERROR* vermerkt. Der Resourcemanagementserver kennt als einzig erlaubten Zustand einer Ressource in seiner Verwaltung *FREE*. Somit kann über diese Implementierung mit dem Observer-Pattern eine Fehlererkennung durchgeführt werden.

6.1.5. User

Das Interface `TestAutomationUser` wird für das Framework Selenium in der Klasse `SeleniumUser` konkretisiert. Die Implementierung des Users ist als reiner Datentyp realisiert und besitzt somit keine Interaktionsfunktionen mit anderen Klassen. Wichtig hierbei ist, dass die Implementierung der Methode `equals(Object o)` basierend auf den Attributen `username` und `password` vorgenommen wurde. Wie im Kapitel 5.3.1 dargestellt sind zwei User identisch, wenn Name und Password zwischen den Usern identisch ist.

Abbildung 25 zeigt das Klassendiagramm des konkreten `SeleniumUser` ohne die Darstellung seiner Methoden. Der Konstruktor mit den Parametern `username`

und `password` wird genutzt um Vergleichsobjekte zu erzeugen um beispielsweise die Loginfunktion zu vereinfachen.

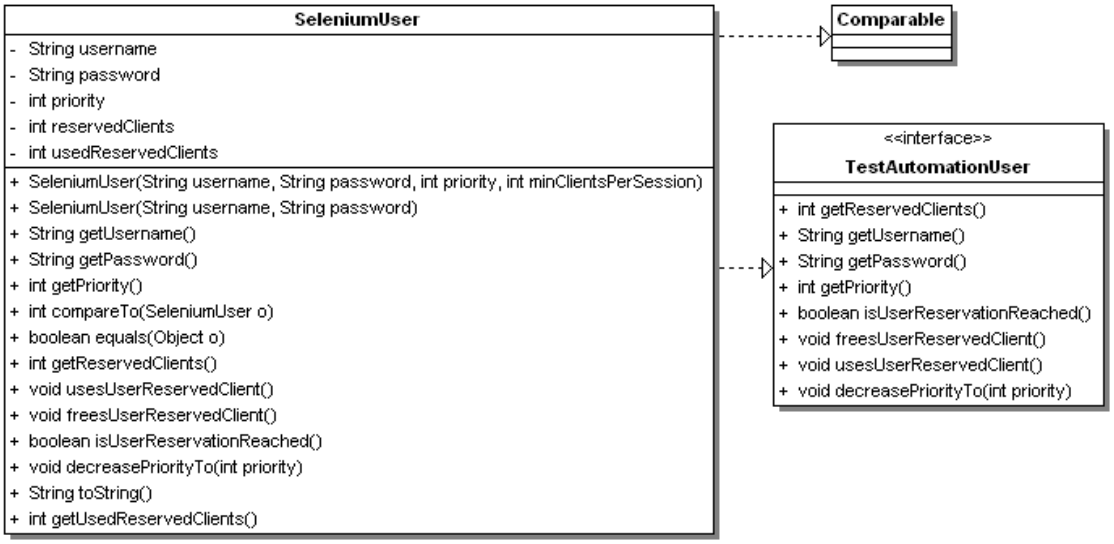

Abbildung 25.: *Klassendiagramm des konkreten* `TestautomationUser` *durch die Klasse* `SeleniumUser`

6.1.6. Anfrage nach Ressourcen

Die Anfrage nach Ressourcen liegt in der Verantwortung des Resourcemanagementservers. In diesem Abschnitt wird die Kommunikationsschnittstelle zwischen AludraTest und dem Resourcemanagementservers beschrieben. Es wird ebenso auf den Ablauf einer Suche nach Ressourcen deren Repriorisierung eingegangen.

Kommunikationsschnittstelle zwischen User und Ressourcemanagementserver

Das Interface `ResourceManagementServer` wird für das Framework Selenium in der Klasse `ResourceManagementServer` konkretisiert. Die Kommunikation mit den Instanzen von AludraTest erfolgt mittels des Observer-Patterns. Um eine Anfrage nach Ressourcen starten zu können, muss die AludraTest-Instanz sich an dem Resourcemanagementserver anmelden. Der Resourcemanagementserver kennt aus der Konfiguration alle User und kann mittels der Methode `equals(Object o)` der User die Anmeldefunktionalität stellen.

Mit der Implementierung von `alive(int sessionID)` wird ein Heartbeat implementiert, der die Verbindung zwischen AludraTest und dem Resourcemanagementserver sicherstellen soll.

Suche nach Ressourcen

Für die Anfrage nach Ressourcen teilt die AludraTest-Instanz dem Resourcemanagementserver die gewünschte Umgebung, Ober- und Untergrenze der Ressourcenanzahl in der Session und ob er die Session sofort erstellt werden soll mit.

Die Suche nach Ressourcen läuft nach einem definiertem Schema ab.

1. Suche nach passenden statischen, für den User verfügbaren Ressourcen
2. Suche nach passenden dynamischen Ressourcen
3. Zuordnung der Ressourcen zur Session

In Schritt 1 und 2 wird bei jeder passenden Ressource ein Test ausgeführt, ob die Ressource erreichbar ist. Ist sie es nicht, wird ein Logeintrag mit dem Level *ERROR* erzeugt und die Ressource als fehlerhaft markiert. Schritt 1 und 2 besitzen zwei Abbruchbedingungen.

- Session besitzt maximale Anzahl an angefragten Ressourcen
- Alle Ressourcen sind abgefragt worden

Die Benachrichtigung der AludraTest-Instanz über die SessionID erfolgt nach der Suche. Die Benachrichtigung über verfügbare Ressourcen, wird von der Session aus erfolgen.

Bei der Anfrage gibt es ein Unterscheidungskriterium, da AludraTest vorgeben kann, ob das Ressourcenminimum und somit die Session sofort verfügbar sein soll, oder ob auf das Ressourcenminimum gewartet werden kann. Kann nicht gewartet werden, so wird nach einer Analyse der verfügbaren und nachgefragten Ressourcen entweder die Session erstellt, oder die Anfrage abgelehnt. Soll jedoch gewartet werden, so wird die Session mit den verfügbaren Ressourcen gefüllt und mit der höchsten Priorität 0 versehen.

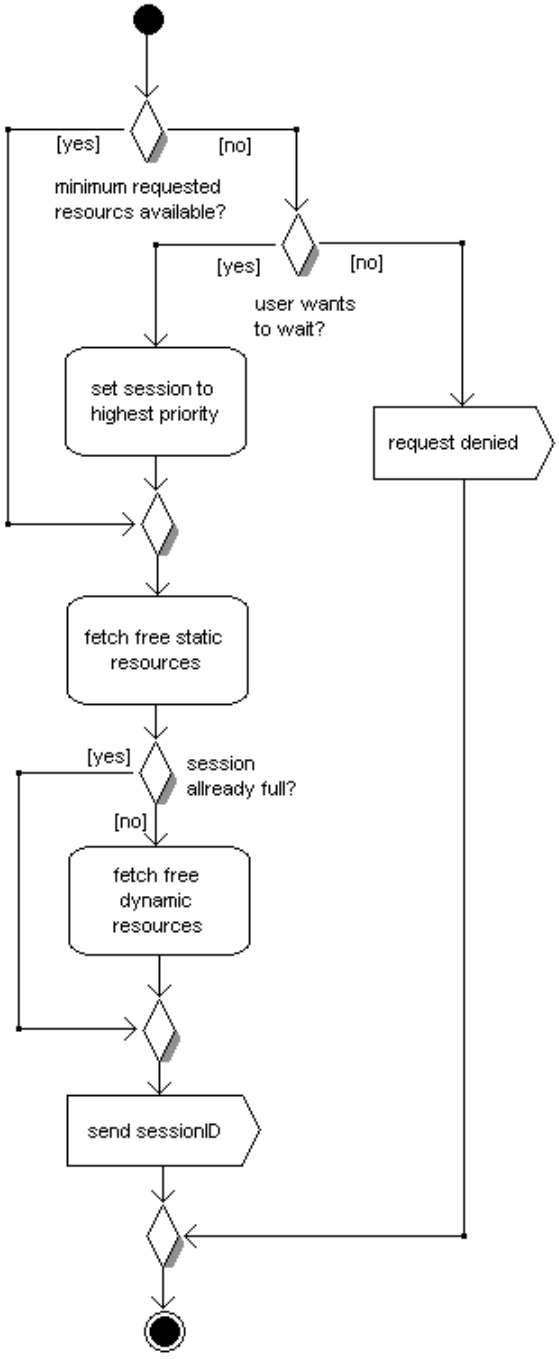

Abbildung 26.: *Activity Diagram - Ablauf einer Ressourcenanfrage*

Repriorisierung einer Ressource

Um eine hohe dynamik in der Verteilung der Ressourcen zu gewährleisten, muss zur Laufzeit eine Repriorisierung von Ressourcen stattfinden.

Jede Session muss nach den in Abschnitt definierten 6.1.3 Regeln eine Ressource zur Repriorisierung geben. Die Repriorisierung iteriert über alle Sessions absteigend nach Priorität. Ist eine Session gefunden, die eine bessere Priorität besitzt als die, aus der die Ressource freigegeben wurde, so wird die Ressource in die besser priorisierte Session überstellt und die Besitzer der geänderten Sessions über den aktuellen Status ihrer jeweiligen Session informiert.

Durch die Repriorisierung ist somit sichergestellt, dass wartende Sessions mit Priorität 0 bevorzugt mit Ressourcen aufgefüllt werden.

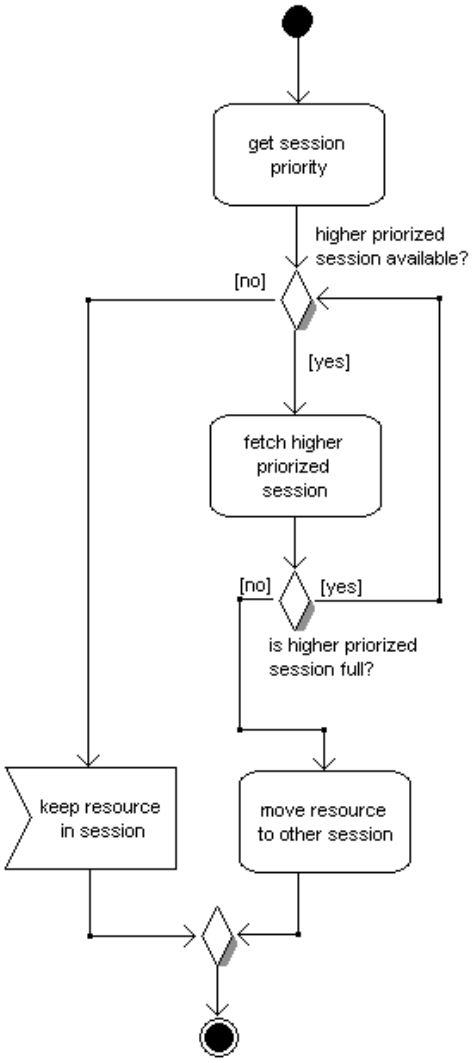

Abbildung 27.: *Activity Diagram - Ablauf einer Repriorisierung*

6.1.7. Timeout einer Session

Um zu signalisieren, dass die Session eines Testers gültig ist, muss der Tester dies in einem definierten Zeitfenster bestätigen. Passiert dies nicht, tritt ein Timeout auf.

Um sicherzustellen, dass der Timeout nach einer vorgegebenen Zeit zuschlägt, eignet sich die Implementierung der Javaklassen `Timer` und `TimerTask`.

Timer Task

Eine Klasse, die `TimerTask` erweitert, bekommt die Funktionalität eines Threads, der nach einer zeitlichen Verzögerung und zusätzlich in bestimmten Zeitabständen gestartet werden kann. Dafür muss die Methode `run()` implementiert werden, die beim Start des Threads ausgeführt wird. Somit muss `run()` alle Arbeitsschritte beinhalten, die zur Abarbeitung der vorgesehenen Aufgabe benötigt werden. Die Zuordnung zu einer zeitlichen Verzögerung und dem Ausführungsintervall wird über einen Timer vorgenommen. Im Rahmen eines Sessiontimeouts wird nun in der Methode `run()` ein Sessionrelease mit der entsprechenden ID beim Managementserver durchgeführt.

```
1 public void run() {
2         try {
3                 server.releaseSession(sessionID);
4                 server.removeTimer(sessionID);
5
6         } catch (SessionNotFoundException e) {
7                 e.printStackTrace();
8         } catch (Exception e) {
9                 e.printStackTrace();
10        }
11 }
```

Listing 13: *Reaktion auf einen Timeout*

Wie zu erkennen ist, besitzt der Timeout eine Referenz zum Server, um zum einen die Session aufzulösen und zum anderen, um sich selber zu stoppen. Dies ist notwendig, da der Timer Task parallel zum Hauptthread läuft und somit keine Synchronität zwischen Hauptprogramm und Timeouts herrscht.

Timer

Die Klasse `Timer` beschreibt einen Scheduler, der Objekte vom Typ `TimerTask` nach einer bestimmten Verzögerung startet. Dabei existiert zudem die Möglichkeit, dass nach der Verzögerung der Task in Intervallen gestartet wird. Über die Methoden `schedule(TimerTask, int delay_in_millis)` und `schedule(TimerTask, int delay_in_millis, int interval_millis)` wird eine Zuordnung von einem Task zu einem Ausführungszeitpunkt vorgenommen. Dabei sind die Zeitangaben in Millisekunden definiert. Um einen Timer abzubrechen, ist die Methode `cancel()` vorhanden.

Um einen Timer eindeutig einer Session zuzuordnen, wird eine assoziative Liste genutzt, wobei die eindeutige SessionID als Key genutzt wird um den entsprechenden Timer zu identifizieren. Wird das erste mal die Alive-Methode für eine Session aufgerufen, muss ein neuer Eintrag erzeugt werden. Das Timeout ist auf drei Minuten definiert. Beim Aufruf von `alive(int sessionID)` wird nun der vorhandene Timer angehalten um ihn neu zu starten und somit einen Reset des Timeouts zu realisieren.

```
1  public synchronized void alive(int sessionID) throws RemoteException,
       SessionTimedOutException {
2          //check session available
3          logger.info("Alive for session {}", sessionID);
4          Timer timeout;
5          if(!sessiontimer.containsKey(sessionID)){
6                  throw new SessionTimedOutException();
7          }
8          timeout = sessiontimer.get(sessionID);
9          timeout.cancel();
10         timeout.purge();
11
12         try {
13                 logger.info("Reschedule timeout for session {}", sessionID);
14                 setupTimeout(sessions.getSession(sessionID));
15                 logger.info("Reschedule done for session {}", sessionID);
16         } catch (SessionNotFoundException e) {
17                 logger.error("Session {} not found for reschedule", sessionID);
18         }
19 }
```

Listing 14: *Reset eines Timeouts in der Methode* `alive(int sessionId)`

6.2. Implementierung von RMI

In diesem Teil wird die Implementierung von RMI als Kommunikationsprotokoll zwischen `SeleniumResourceManagementServer` und den anfragenden AludraTest-Instanzen behandelt.

Für die Implementierung von RMI ist zuerst die Registry für die Objektverwaltung zu konfigurieren. Dabei spielt die Platzierung der Registry in dem Aufbau mit RMI ein große Rolle. Ebenso wichtig ist die Integration in das bestehende Netzwerk.

6.2.1. Registry

Die Registry wird ist auf dem gleichen Host realisiert, auf dem auch der Server läuft. Um sicherzustellen, dass von fremden Hosts keine eventuell schadcodebehaftete Objekte auf der Registry abgelegt werden können, sieht RMI vor, dass nur vom Host der Registry aus ein Objekt registriert werden kann. Wird von einem fremden Host versucht ein Objekt anzumelden, wird eine entsprechende Exception von der JRE geworfen.

Um eine Kommunikation über das Netzwerk zu realisieren, müssen sowohl Registry, als auch Clients über den gleichen Port kommunizieren. Der RMI-Standardfall sieht dafür den Port 1099 vor. Dies ist in der Implementierung im Rahmen dieser Arbeit nicht möglich, da durch interne Sicherheitsrichtlinien ein Port über 16000 genutzt werden muss. In diesem Fall ist es der Port 16022.

Um die Registry zu starten, wird eine statische Methode in der Klasse `TestAutomationRMIRegistry` erstellt.

```
1 /**
2  * Creates a RMI registry on port 16022
3  * @return a RMI registry on port 16022
4  */
5 public static Registry createRegistry(){
6         Registry reg = null;
7         try {
8                 reg = LocateRegistry.createRegistry(RMI_PORT);
9         } catch (RemoteException e) {
10                e.printStackTrace();
11        }
12        return reg;
13 }
```

Listing 15: *Erstellung einer Registry*

Diese Methode versucht, eine Registry auf dem Port 16022 zu erstellen. Gelingt dies nicht, wird eine Exception gefangen. Ebenfalls existiert eine Methode, um eine

existierende Registry anzufordern.

```
1 /**
2  * Searches for an existing RMI registry on port 16022
3  * @return a RMI registry
4  */
5 public static Registry retrieveRegistry(){
6         Registry reg = null;
7         try {
8                 reg = LocateRegistry.getRegistry(REGISTRY_HOST, RMI_PORT);
9         } catch (RemoteException e) {
10                e.printStackTrace();
11        }
12        return reg;
13 }
```

Listing 16: *Anfordern einer Registry*

Der Parameter `RMI_HOST` enthält die Hostadresse als String und der Parameter `RMI_PORT` den Port als Ganzzahl. Sie sind als statisches Attribut der Klasse implementiert.

6.2.2. Export mit UnicastRemoteObject

Jedes über RMI erreichbare Objekt wird durch einen Proxy repräsentiert. In der Standardimplementierung gibt die RMI-Registry diesen Proxy über einen zufälligen Port frei, den sie dem Anfragenden mitteilt. Um dies zu ermöglichen, muss ein über RMI erreichbares Objekt die Klasse `UnicastRemoteObject` erweitern und in seinem Konstruktor den Konstruktor von `UnicastRemoteObject` nutzen. Hierbei kann ein Port übergeben werden, an dem die Proxyobjekte freigegeben werden. Dieser kann bei allen Objekten identisch sein, da RMI in dem Fall ein Multiplexing realisiert. Als Beispiel soll hierfür der Konstruktor der Klasse `SeleniumResourceManagementServer` dienen.

Für diese Klasse wird der Port 16044 genutzt. Da auf Grund von Sicherheitsrichtlinien im Netzwerk des Kunden ein Zugriff mit zufälligen Ports nicht zulässig ist, wird ein dedizierter Port für alle zu exportierenden Objekte genutzt.

```
1 public SeleniumResourceManagementServer() throws RemoteException{
2                 super(16044);
```

```
3                 resources = SeleniumResourceStorage.getInstance();
4                 users = SeleniumUserStorage.getInstance();
5                 sessions = SeleniumResourceManagerSessionStorage.getInstance();
6 }
```

Listing 17: *Export eines RMI-Objektes mit Hilfe des Kunstruktors von* `UnicastRemoteObject`

6.3. Implementierung der Konfigurationsschnittstelle

Um den User- und Resourcestorage mit Inhalten zu füllen, existieren zwei Konfigurationsdateien im XML-Format (siehe Kapitel 4.3.1). Die enthaltenen Daten werden mit Hilfe von XPath ausgelesen (siehe Kapitel 4.3.2) und in dem entsprechenden Storage gespeichert. In diesem Teil wird auf die Konfiguratoren der Storages eingegangen.

6.3.1. Ressourcenkonfigurator

Der Ressourcenkonfigurator hat die Aufgabe, die in der XML-Datei definierten Ressourcen einzulesen und im Storage zu speichern. Diese Verantwortlichkeit ist in der Klasse `ResourceConfigurator` implementiert. Dabei werden die Ressourcen sequentiell eingelesen. Die als Capabilities hinterlegten Umgebungen werden in dieser Implementierung nur auf den Browser hin gesondert überprüft, da ein Test auf Linux noch nicht unterstützt wird. Ist eine unbekannte Browserangabe in einer Capability zu finden, wird die gesamte Ressource verworfen. Im Activitydiagram der Abbildung 28 ist der Ablauf skizziert. Das Listing 18 zeigt die Implementierung der Browseranalyse.

```
1 /**
2  * Sets the browser (firefox, internet explorer, chrome or android)
3  * @param browser name of the browser
4  * @param dcap capability object containing client capabilities
5  * @throws NoValidBrowserSpecifiedInConfigException
6  */
7 private void setBrowserCapability(String browser, DesiredCapabilities dcap) throws
      NoValidBrowserSpecifiedInConfigException {
8         if(browser.equals("firefox")){
9                 dcap.setBrowserName(DesiredCapabilities.firefox().getBrowserName());
10        }
11
```

```
12          else if(browser.equals("iexplorer")){
13                  dcap.setBrowserName(DesiredCapabilities.internetExplorer().
                        getBrowserName());
14          }
15
16          else if(browser.equals("android")){
17                  dcap.setBrowserName(DesiredCapabilities.android().getBrowserName());
18          }
19
20          else if(browser.equals("chrome")){
21                  dcap.setBrowserName(DesiredCapabilities.chrome().getBrowserName());
22          }
23
24          else{
25                  throw new NoValidBrowserSpecifiedInConfigException(browser);
26          }
27  }
```

Listing 18: *Browseranalyse im ResourceConfigurator*

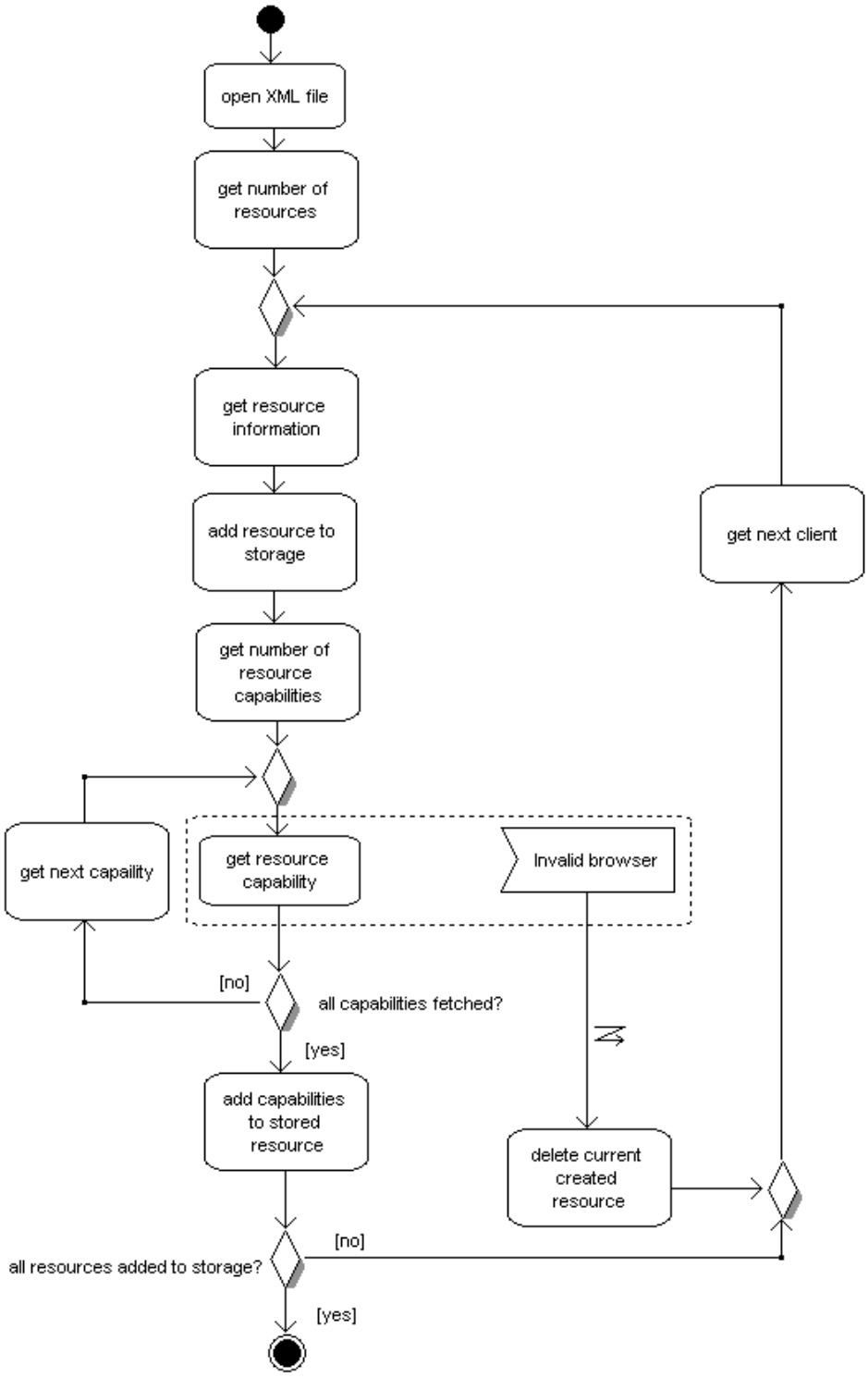

Abbildung 28.: *Activity Diagram - Ablauf einer Ressourcenkonfiguration*

6.3.2. Userkonfigurator

Der Userkonfigurator arbeitet ähnlich wie der Ressourcenkonfigurator. Er liest aus einer XML-Datei die Charakteristika der User ein und füllt mit den daraus erstellten Userobjekten den Userstorage. Die Attribute werden auch hier mit Hilfe von XPath ausgelesen. Die Abbildung 29 stellt den Ablauf der Userkonfiguration da.

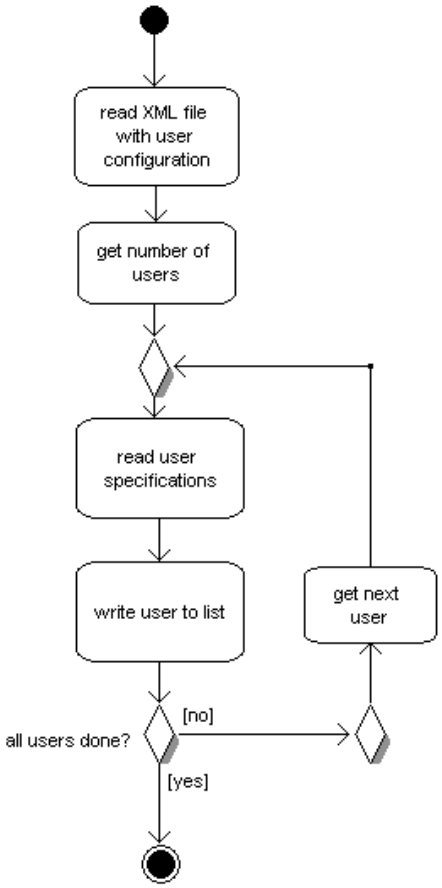

Abbildung 29.: *Activity Diagram - Ablauf einer Userkonfiguration*

In Listing 19 ist ein Ausschnitt des Erstellens der User gezeigt. Mit der XPath-Funktion `count` wird die Anzahl der erfolgreichen Suchergebnisse eines XPath-Ausdrucks ermittelt. In diesem Fall ist das die Anzahl der User in der Konfigurationsdatei. Da die Evaluation des Ausdrucks einen String zurückliefert, muss das Ergebnis geparst werden, um den Integerwert zu erhalten. Ist auf Grund eines Fehlers das Parsen nicht möglich, wird eine Exception geworfen und der Stacktrace ausgegeben. Die Variable `users` wird vor diesem Listing mit dem Wert 0 initialisiert, sodass die Schleife

nicht durchlaufen und keine User geschrieben werden. Ist in der Konfiguration eines einzelnen Users ein Fehler, wird die dadurch entstehende Exception gefangen und mit dem nächsten User fortgefahren.

```
1  ArrayList<SeleniumUser> seleniumusers = new ArrayList<SeleniumUser>();
2  try{
3          expr = xPath.compile("count(//users/user)");
4          users = Integer.parseInt((String) expr.evaluate(doc, XPathConstants.STRING))
               ;
5  }
6  catch(XPathExpressionException xpee){
7          xpee.printStackTrace();
8  }
9  for(int i = 0;i<users;i++){
10         try{
11                 expr = xPath.compile("//users/user["+(i+1)+"]/@username");
12                 username = (String)expr.evaluate(doc, XPathConstants.STRING);
13
14                 expr = xPath.compile("//users/user[@username='"+username+"']/
                       @password");
15                 password = (String)expr.evaluate(doc, XPathConstants.STRING);
16
17                 expr = xPath.compile("//users/user[@username='"+username+"']/prio");
18                 prio = Integer.parseInt((String)expr.evaluate(doc, XPathConstants.
                       STRING));
19
20                 expr = xPath.compile("//users/user[@username='"+username+"']/
                       reservedclients");
21                 reservedClients = Integer.parseInt((String)expr.evaluate(doc,
                       XPathConstants.STRING));
22
23                 if(prio < 1){
24                         throw new InvalidUserPriorityException(username);
25                 }
26                 else if(reservedClients < 0){
27                         throw new InvalidUserReservedClientsException(username);
28                 }
29                 else{
30                         seleniumusers.add(new SeleniumUser(username, password, prio,
                               reservedClients));
31                 }
```

```
32          }
33          catch(XPathExpressionException xpee){
34                  xpee.printStackTrace();
35          }
36          catch(NumberFormatException nfe){
37                  nfe.printStackTrace();
38          }
39  }
```

Listing 19: *Usererstellung im UserConfigurator*

7. Test

In diesem Kapitel wird der Test des Resourcemanagement-Servers beschrieben. Es wird in Unittests und Systemtests unterschieden. Dabei wird erläutert, warum die Software nur in Teilen durch Unittest getestet werden kann.

7.1. Unittest

In diesem Abschnitt werden die durchgeführten Unittests erläutert. Dabei wird beleuchtet, welche Teile der Software mit einem Unittest testbar sind. Bei den durchgeführten Tests werden neben den ausformulierten JUnit-Tests, die zugrunde liegenden Daten erläutert.

7.1.1. Testbarkeit der Software durch Unittests

Eine kompletter Test aller Funktionalitäten der im Rahmen dieser Arbeit erstellten Software anhand von Unittests mittels JUnit ist nicht möglich. Im Kapitel 4.1 wurde die Entscheidung getroffen, aus Sicht der Architektur ein eventgetriebenes System zu implementieren. Im Design wurde diese Architektur mit Hilfe des Observerpatterns umgesetzt (siehe Abschitt 5.5). Eine Anfrage nach Ressourcen wird mittels RMI vom Tester zum Resourcemanagementserver übertragen (siehe Abschnitt 4.2.3). Der Resourcemanagementserver wird somit über das Netzwerk erreicht und somit ist die Antwort auf die Anfrage verzögert. Die Dauer der Verzögerung kann nicht vorhergesehen werden, da die Verfügbarkeit des Netzwerkes nie vollständig zugesichert werden kann.
Im Resourcemanagementserver ist die gesamte Ressourcenanfrage asynchron. Somit gibt es keine Rückgaben darüber, ob alle nötigen Nachrichten versendet wurden. Durch die Kommunikation über das Netzwerk und die nicht vorhersehbaren Verzögerungen bei der Übertragung, kann auch keine zeitliche Zusicherung erfolgen, wann eine Nachricht übermittelt wurde.

JUnit wertet Methodenaufrufe anhand des Rückgabewertes oder einer Exception

aus, somit arbeitet JUnit mit Hilfe von synchronen Nachrichten. Dies führt nun zu den folgenden Problemen bei der Testbarkeit mit JUnit[80].

- Asynchrone Nachrichten liefern keinen Rückgabewert.
- Für den Nachrichtenempfang kann keine zeitliche Zusicherung erfolgen.

Folglich können mittels JUnit nur die Testfälle abgedeckt werden, die ausserhalb der asynchronen Kommunikation erfolgen. Das sind daher alle Testfälle, die nicht das Observerpattern abfragen und nur auf einem Host stattfinden.

7.1.2. Testfälle

Die Anzahl der Testfälle ist durch die Testbarkeit der Software begrenzt. Es können also nur die folgenden Bereiche durch Testfälle mittels JUnit abgebildet werden.

- Storages
- Configurators
- Aufrufe der Registry

Test der Klassen SeleniumResourceStorage und des ResourceConfigurator

Die Klasse `SeleniumResourceStorage` wird basierend auf der im Listing 20 konfigurierten Ressourcen getestet.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <clients>
3         <client ip="127.0.0.1" platform="windows" pool="dynamic">
4                 <capability>
5                         <browser>firefox</browser>
6                         <version>11</version>
7                         <path></path>
8                         <maxInstances>5</maxInstances>
9                 </capability>
10                <capability>
11                        <browser>iexplorer</browser>
12                        <version>6</version>
13                        <path></path>
```

[80] [vgl. Eva05]

```
14                      <maxInstances>5</maxInstances>
15              </capability>
16              <capability>
17                      <browser>android</browser>
18                      <version>4</version>
19                      <path></path>
20                      <maxInstances>5</maxInstances>
21              </capability>
22      </client>
23      <client ip="128.0.0.1" platform="windows" pool="static">
24              <capability>
25                      <browser>firefox</browser>
26                      <version>11</version>
27                      <path></path>
28                      <maxInstances>5</maxInstances>
29              </capability>
30              <capability>
31                      <browser>iexplorer</browser>
32                      <version>6</version>
33                      <path></path>
34                      <maxInstances>5</maxInstances>
35              </capability>
36              <capability>
37                      <browser>chrome</browser>
38                      <version>4</version>
39                      <path></path>
40                      <maxInstances>5</maxInstances>
41              </capability>
42      </client>
43 </clients>
```

Listing 20: *Testdaten für den SeleniumResourceStorage*

Die Funktionalitäten werden durch sieben Tests überprüft.

(1) Die Ressourcen mit den IP-Adressen 127.0.0.1 und 128.0.0.1 sind im Storage enthalten.

(2) Die Ressource mit der IP-Adresse 127.0.0.1 ist im dynamischen Pool.

(3) Die Ressource mit der IP-Adresse 128.0.0.1 ist im statischen Pool.

(4) Die Ressource mit der IP-Adresse 127.0.0.1 besitzt 3 Capabilities.

(4.1) Die Ressource mit der IP-Adresse 127.0.0.1 besitzt einen FireFox.

(4.2) Die Ressource mit der IP-Adresse 127.0.0.1 besitzt einen InternetExplorer.

(4.3) Die Ressource mit der IP-Adresse 127.0.0.1 besitzt eine Androidumgebung.

Der Unittest überprüft die Tests 4.1, 4.2 und 4.3 mit Hilfe eines Matchers von Selenium (siehe Abschnitt 5.2.1). Die dafür nötigen Casts in eine Map der Form `Map<String,Object>` sind nicht kritisch. Jede Objekt vom Typ `DesiredCapability` kann die gespeicherten Capabilities als Map der Form `Map<String,?>` zurückgeben. Dabei weisst die *?*-Wildcard auf irgendein Objekt unbekannten Typs. Object ist die Basisklasse aller Klassen und somit ist die Bedingung für die Wildcard erfüllt.

Test der Klasse SeleniumUserStorage und des UserConfigurator

Die Klassen `SeleniumUserStorage` und `UserConfigurator` werden basierend auf der im Listing 21 konfigurierten Daten getestet.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <users>
3          <user username="Test01" password="test1234">
4                  <prio>2</prio>
5                  <reservedclients>4</reservedclients>
6          </user>
7          <user username="Test02" password="test1234">
8                  <prio>3</prio>
9                  <reservedclients>0</reservedclients>
10         </user>
11 </users>
```

Listing 21: *Testdaten für den SeleniumUserStorage*

Die Funktionalitäten werden durch vier Tests überprüft.

1 Es sind 2 User im Storage enthalten.

2 Die User mit den Namen Test01 und Test02 sind im Storage enthalten.

3 Der User Test01 hat 4 reservierte global reservierte Ressourcen.

4 Der User Test02 besitzt die Priorität 3.

7.1.3. Ergebnisse der Testfälle

In diesem Abschnitt werden die Testergebnisse zu den Testfällen des Unittests dargestellt und die Korrektheit der Funktionalität für den jeweiligen Testfall verifiziert.

Klasse SeleniumResourceStorage

Die Ergebnisse des Tests zeigen, dass der `ResourceConfigurator` die geforderten Funktionalitäten erfüllt. Der `SeleniumResourceStorage` ermöglicht den Zugriff auf die Ressourcen, wie er im Rahmen dieses Tests benötigt wird.

Klasse SeleniumUserStorage

Das Ergebnis des Tests zeigt, dass der `UserConfigurator` den `SeleniumUserStorage` mit den erwarteten Usern füllt. Der Storage erlaubt auch die erwarteten Zugriffe. Die Funktionalität ist somit korrekt implementiert.

7.2. Systemtest

Im Systemtest wird das Gesamtsystem auf seine Funktionalität hin durch Blackboxtests überprüft. Dabei wird zunächst auf die Testfälle eingegangen, die sich mit der Ressourcenanfrage befassen.

7.2.1. Testfälle

In diesem Abschnitt werden die Testfälle erläutert. Dabei wird kurz auf die zu testende Funktionalität eingegangen und darauf folgend die Testdaten und der erwartete Testausgang definiert.

Test der Anmeldefunktionalität

In diesem Testfall wird die Anmeldefunktionalität gestestet. Dabei wird für den positiven Fall eine valide Kombination von Usernamen und Passwort genutzt, die im `SeleniumUserStorage` definiert ist. Tabelle 13 zeigt die für den Test genutzten Daten und die zu erwarteten Testergebnisse.

Username	Passwort	Erwarteter Ausgang
Test03	1234	Login erfolgreich
Test03		Login fehlgeschlagen

Tabelle 13.: *Testdaten und erwarteter Ausgang für den Test der Anmeldefunktion*

Test der Erfüllbarkeit von Reservierungen

In diesem Testfall werden zwei dynamische und drei statische Ressourcen konfiguriert sein. Die User haben insgesamt vier Reservierungen von statischen Ressourcen. Somit sind mehr Reservierungen gefordert, als der Resourcemanagementserver erfüllen kann. Erwartet wird ein Loggingeintrag mit dem Loglevel *ERROR* und der Resourcemanagementserver darf nicht starten.

Vergabe von reservierten Ressourcen

Als erstes sollen die reservierten Ressourcen an die Session eines Users vergeben werden. Um diese Funktionalität zu prüfen, werden wieder drei statische und zwei dynamische Ressourcen konfiguriert. Ein User mit einem Reservierungslimit von zwei Ressourcen stellt eine Anfrage nach zwei Ressourcen. Erwartet wird, dass nur statische Ressourcen vergeben werden.

Vergabe von statischen und dynamischen Ressourcen

In diesem Test soll geprüft werden, ob Sessions mit dynamischen Ressourcen befüllt werden, wenn schon statische vorhanden sind.

Es existieren 3 statische und eine dynamische Ressourcen. Es sind drei User vorhanden. User1 besitzt eine Priorität von 2 und zwei reservierte Ressourcen. User2 besitzt eine Priorität von 3 und hat keine reservierten Ressourcen. User3 besitzt eine Priorität von 100 und eine reservierte Ressource.

Im ersten Teil dieses Tests darf keine der Sessions verzögert gefüllt werden. Zuerst fragt User1 nach exakt 3 Ressourcen, so dass in diesem Fall Minimum und Maximum identisch sind. Erwartet wird, dass eine Session mit zwei statischen und einer dynamischen Ressource erstellt wird. Als zweites fragt User3 nach exakt zwei Ressourcen. Erwartet wird, dass eine statische und eine dynamische Ressource geliefert wird. Als Letztes fragt User2 nach exakt einer Ressource. Da keine Ressourcen verfübar sind, wird erwartet, dass diese Anfrage abgelehnt wird.

Im zweiten Teil fragt User1 eine Session mit drei Ressourcen an. Hier wird auch erwartet, dass zwei statische und eine dynamische Ressource vergeben wird. User1 fragt für einen weiteren Test eine Ressource an. Erwartet wird, dass die letzte dynamische Ressource geliefert wird, da das Limit für statische Ressourcen erreicht ist. User2 fragt nun nach zwei Ressourcen, wobei die Session verzögert gefüllt werden darf (Idle-Flag auf `true`). Erwartet wird, dass eine statische Ressource geliefert wird und die Session mit Priorität 0 auf eine freie dynamische Ressource wartet.

Dynamische Ressourcenvergabe zur Laufzeit

Um die dynamische Ressourcenvergabe zu testen werden drei User gemäß Tabelle 14 konfiguriert.

Name	Reservierte Ressourcen	Angefragte Untergrenze	Angefragte Obergrenze	Warten erlaubt	Priorität
User 1	0	1	3	Nein	1
User 2	0	1	3	Nein	2
User 3	1	3	4	Ja	500

Tabelle 14.: *Konfigurierte User für die dynamische Ressourcenvergabe*

Es sind insgesamt sechs Ressourcen verfügbar, von denen eine statisch ist. Erwartet wird folgender Ablauf bei der dynamischen Ressourcenzuteilung. User 1 wird als erster seine Anfrage senden. Da noch keine Ressourcen vergeben sind, wird er die angefragte Obergrenze mit dynamischen Ressourcen erhalten. Seine Sessions wird mit Priorität 1 versehen. Als zweiter fragt User 2 nach einer Session. Da lediglich noch zwei dynamische Ressourcen für ihn verfügbar sind und er als Untergrenze 1 gewählt hat, wird er diese zwei Ressourcen bekommen. Seine Sessions wird mit Priorität 2 versehen. Er wird zwei lange Testfälle ausführen und daher keine seiner erhaltenen Ressourcen zur Repriorisierung zurückgeben. Als letztes fragt User 3 nach einer Session für seinen Ressourcenbedarf. Er erlaubt seine Session verzögert befüllen zu lassen. Somit erhält er als Erstes seine statische Ressource. Seine Session wird mit Priorität 0 versehen.

User 1 wird nacheinander je eine Ressource zur Repriorisierung freigeben. Da User 3 eine wichtigere Session besitzt, wird seine Session mit den frei gewordenen Ressourcen befüllt, so dass er schlussendlich auf zwei dynamische und eine statische Ressource kommt. Zu diesem Zeitpunkt wird seine Session mit seiner Priorität von 500 versehen.

User 3 löst nun seine Session auf. Die freien dynamischen Ressourcen gehen an

User 1, da seine Session die aktuell höchste Priorität (1) besitzt. User 2 bekommt somit erst Ressourcen zu seiner Session zugeordnet, wenn User 1 seine Session auflöst. Dieser Ablauf ist im Sequenzdiagramm in Abbildung 30 dargestellt.

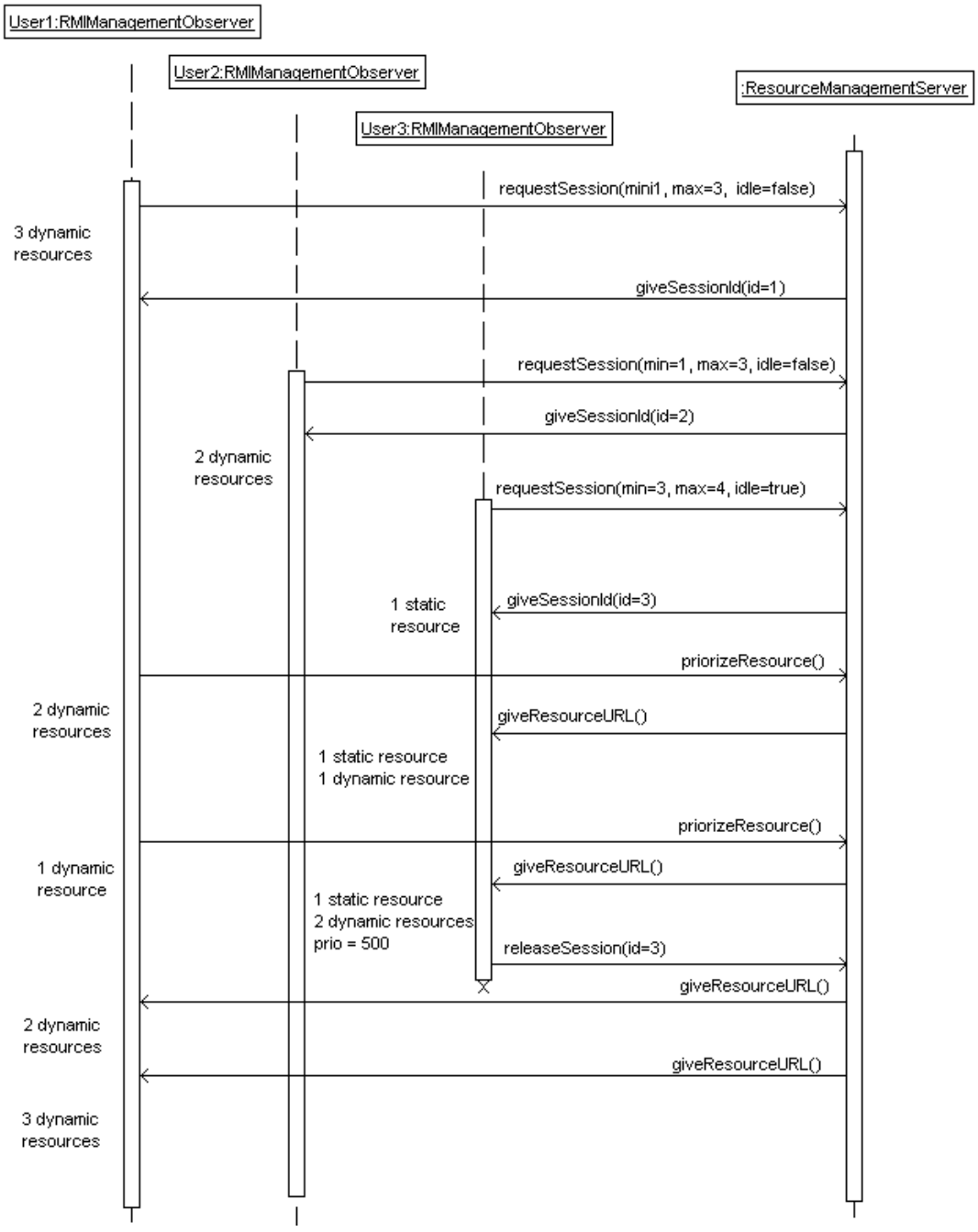

Abbildung 30.: *Sequenzdiagramm - Ablauf des komplexen Testfalls*

7.2.2. Ergebnisse der Tests

In diesem Abschnitt werden die Testergebnisse zu den Testfällen des Systemtests dargestellt und die Korrektheit der Funktionalität für den jeweiligen Testfall verifiziert.

Anmeldefunktion

Die Anmeldefunktion hat für beide Fälle den erwarteten Ausgang geliefert. Daraus lässt sich schließen, dass die Anmeldung auf dem Resourcemanagementserver und die Equals-Methode der User korrekt implementiert ist.

Erfüllbarkeit der von Reservierungen

Der Resourcemanagementserver konnte wie erwartet nicht gestartet werden und eine entsprechende Fehlermeldung wurde in das Errorlog geschrieben. Somit ist die Prüfung auf eine korrekte Konfiguration des statischen Pools gemessen an der Summe aller Reservierungen korekt implementiert.

Vergabe von reservierten Ressourcen

Die Session wurde mit zwei statischen Ressourcen gefüllt. Somit zeigt sich, dass zuerst die statischen Ressourcen vergeben werden. Die Funktionalität ist somit korrekt implementiert.

Vergabe von statischen und dynamischen Ressourcen

Die Anfrage von User2 wurde wie erwartet abgelehnt, da nicht genügend Ressourcen zur befriedigung der Anfrage zum Zeitpunkt der Anfrage vorhanden sind. Ebenfalls erhält User3 die erwarteten Ressourcen. User1 besitzt am Ende des Tests nur die erwarteten zwei statischen Ressourcen und zusätzlich eine dynamische Ressource. Somit ist die Vergabe von statischen und dynamischen Ressourcen ohne Wartemodus korrekt implementiert.

Im Wartemodus erhält User2 eine Session, die maximal priorisiert ist, da User1 alle Ressourcen zu seinen zwei Ressourcen zugeordnet hat. Dies war das erwartete Ergebnis und somit ist die Funktionalität der Verzögerten Sessionbefüllung korrekt implementiert.

Dynamische Ressourcenvergabe zur Laufzeit

es ist zu beobachten, dass User2 nie seine angefragte Obergrenze an Ressourcen zugewiesen bekommt. Dies war auch so erwartet, da die Session von User3 im Wartemodus bevorzugt behandelt wird. Als User3 seine Session auflöste, wurden die Ressourcen zum Auffüllen der aktuellen Sessions nach absteigender Priorität genutzt. Da alle Ressourcen benötigt wurden, um die Sessin von User1 zu befüllen, wurde User2 keine Ressource mehr zusätzlich gegeben. Erst wenn User1 seine Session auflöst, können User2 zusätzliche Ressourcen gewährt werden. Dieses Testergebnis deutet auf die Korrektheit der Implementierung der dynamischen Ressourcenvergabe hin.

8. Fazit und Ausblick

In diesem Kapitel wird die Arbeit rückblickend bewertet und auf mögliche Erweiterungen hin untersucht. Zum Abschlusszeitpunkt dieser Arbeit befand sich die Anwendung noch im Testprozess, weshalb die Bewertung auf dem Verhalten zu diesem Zeitpunkt basiert.

8.1. Fazit

Ziel dieser Arbeit ist die Einführung eines Tools zur dynamischen Verwaltung von Ressourcen innerhalb einer Testautomation.

Anfangs wurde im Rahmen dieser Arbeit das einzigartige Umfeld, in dem diese Arbeit durchgeführt wurde dargelegt, die Anforderungen und Ausschlusskriterien seintens des Kunden und die theoretischen Grundlagen auf denen diese Arbeit basiert dargelegt. Es wurde am Beispiel des Vorfalls um den Therac-25 Linearbeschleuniger die Wichtigkeit des Softwaretests in der Softwareentwicklung vermittelt und darauf folgend ein Gefühl für die Komplexität des Themas Softwaretest gegeben.

In der Analyse wurden die Anforderungen an die Arbeit erörtert und, basierend auf den Ergebnissen dieser Erörterung, eine Analyse des Marktes für Ressourcenverwaltung in der Testautomatisierung vorgenommen. Im Rahmen der Marktanalyse stellten sich die Software Selenium Grid und der Dienstleister SauceLabs als mögliche Lösungskandidaten heraus. Eine genauere Betrachtung der Kandidaten zeigte jedoch, dass beide für einen Einsatz nicht geeignet sind. SauceLabs konnte keine Konfiguration der Testumgebungen auf spezielle Kundenanforderungen liefern und unterstützt keine Testumgebungen für mobile Endgeräte. Selenium Grid zeigte auch einen Mangel in der Unterstützung mobiler Endgeräte, der jedoch nicht im Grid selber zu finden war, sondern im entsprechenden Treiber für die Fernsteuerung des Browsers auf den Endgeräten. Somit wurde der Weg einer Eigenentwicklung eingeschlagen.

Im Kapitel Architektur zeigte sich, dass für die Server-Client-Beziehung zwischen Tester und Resourcemanagementserver eine grundlegende Entscheidung für das

Kommunikationsprinzip erfolgen muss. Dabei wurde sich auf Grund der geringeren Belastung des Netzwerkes für eine eventgetriebenes System entschieden. Diese Entscheidung beeinflusste die Definition der Kernkomponenten und besonders die Architektur im Zusammenspiel mit AludraTest. Es wurde abschließend die Konfigurationsschnittstelle definiert und eine Analyse von zwei möglichen Logging-APIs vorgenommen.

Im Design wurden die im Rahmen der Architektur getroffenen Entscheidungen, in Form von Interfaces und Klassen spezifiziert. Eine besondere Problemstellung war dabei die Integration des Observer-Pattern in eine Kommunikation über RMI. Dabei stellte sich heraus, dass es sinnvoll ist, drei auf die Anwendung spezialisierte Formen des Observer-Pattern zu definieren, um das Exceptionhandling überschaubar zu halten. Zwei Pattern sind dabei für die innere Observerstruktur des Resourcemanagementservers definiert und eins stellt die Funktionalität des Observer-Patterns für die Netzwerkkommunikation über RMI bereit. Für die innere Struktur wurden die Pattern auf den Zustand von Ressourcen ausgelegt. Nach aussen hin wurde der Fokus auf das Observieren des Resourcemanagementservers gelegt. Eine wichtige Entscheidung im Design war die Definition der Priorität. Die Entscheidung wurde für ein Prioritätsmodell getroffen, das sich an dem Prinzip der Nicevalues von UNIX orientiert.

In der Implementierung zeigte sich der größte Aufwand bei der Implementierung der dynamischen Ressourcenvergabe durch Repriorisierung im Zusammenhang mit dem Observer-Pattern. Die Identifizierung der Zustände, die zum Vergabezeitpunkt existieren können, war nicht trivial. Hier spielen viele Faktoren bezüglich der existierenden Sessions, der Befugnissen der User und der momentanen Belegung der Ressourcen eine Rolle, die über eine Vergabe der Ressourcen an Sessions entscheiden. Es musste stets darauf geachtet werden, wer zu welchem Zeitpunkt der Observer des Zustands einer Ressource ist.

Im Test wurden die wichtigsten Funktionalitäten durch Blackbox-Tests überprüft. Dabei war durch das eventgetriebene System ein Unittest mit JUnit nur bedingt möglich. Der Systemtest bezüglich der gesamten Entwicklung wurde anhand definierter Testfälle durchgeführt. Insgesamt wurde sich im Rahmen dieser Arbeit auf funktionale Tests beschränkt. Nicht-funktionale und strukturbezogene Tests wurden auf Grund des Aufwands im Rahmen dieser Arbeit nicht mehr durchgeführt. Die durchgeführten Tests zeigten jedoch positive Ergebnisse.

Der aktuelle Stand der Implementierung bietet einen guten Überblick über die Basisfunktionen einer Ressourcenverwaltung. Trotz der im Abschnitt 8.2 erläuterten Erweiterungs- und Verbesserungsmöglichkeiten, steht dem Unternehmen nun eine Anwendung zur dynamsichen Ressourcenverwaltung in der Testautomation zur Ver-

fügung und eine Reduzierung der Kommunikation zwischen den Testern betreffend der Organisation ihrer Testausführungen ist möglich.

8.2. Ausblick

Die im Rahmen dieser Arbeit erfolgte Implementierung stellt eine Grundimplementierung dar. Es sind dabei mögliche Erweiterungen denkbar.

8.2.1. Grafische Administrationsschnittstelle mit automatischem Deployment

Die Administrationsschnittstelle kann im simpelsten Fall mit Hilfe einer Konsolenanwendung realisiert werden. Dabei wäre es möglich mittels JNLP ein automatisches Deployment zu ermöglichen. Dabei könnte auf der zentralen Website des Testteams ein Link zu dem JNLP-File gesetzt werden, der dann die aktuellste Version automatisch herunterlädt und installiert.

8.2.2. Automatische Prüfung nach validen Konfigurationen

Die Konfigurationsdateien könnten schon beim Einlesen auf Korrektheit geprüft werden. Sinnvolle Techniken hierfür ist die Definition einer XML-Document-Type-Definition (DTD). Die DTD setzt dabei die genaue Struktur der XML-Datei fest, wodurch die Anzahl der möglichen Fehleingaben verringert wird.

8.2.3. Einsatz von JINI bei unterschedlichen AUT oder redundanten Registries

Jede zu testende Anwendung kann unterschiedliche Testumgebungen benötigen. Im Rahmen dieser Arbeit wurden ausschließlich Seleniumserver als Ressourcen verwaltet. Es kann jedoch der Fall eintreten, dass Anwendungen getestet werden sollen, die kein Selenium benötigen, sondern eine Datenbankumgebung auf einem Unix-Betriebsystem. Da dies andere Ressourcen sind, wäre ein weiterer Managementserver, der auf diese Ressourcen spezialisiert ist, denkbar.

Ebenfalls können zur Steigerung der Verfügbarkeit mehrere Registries existieren, die die Managementserver halten. Dabei verwalten die Managementserver gleichen Typs die gleichen Ressourcen. Ziel ist es dabei, beim Ausfall einer Registry oder eines

Servers auf einen anderen wechseln zu können.

JINI bietet da eine Möglichkeit, den entsprechenden Resourcemanagementserver aus der erreichbaren Registry anzubieten und zu unterscheiden, welchen Typ des Managementservers benötigt wird. Zusätzlich muss der Client den Registrierungsnamen des Servers bei er Registry nicht kennen, da die Auflösungsverantwortung bei JINI liegt.

Literaturverzeichnis

[APP] APPLE INC.: *SOFTWARE LICENSE AGREEMENT FOR MAC OS X*. PDF, . – http://images.apple.com/legal/sla/docs/macosx106.pdf Letzter Zugriff 13.04.2012

[Bra] BRAUN, Stephan: *Ausarbeitung Zeit- und ereignisgesteuerte Echtzeitsysteme*. – http://ess.cs.tu-dortmund.de/Teaching/PGs/autolab/ausarbeitungen/Braun-Zeit-_und_ereignisgesteuerte_Echtzeitsysteme-Ausarbeitung.pdf

[BW03] BERENDES, Marc ; WEVER, Philipp: *Logging in Java*. Seminararbeit, 2003. – http://berrendorf.inf.h-brs.de/lehre/ss03/vups1/Seminar/3_LoggingInJava.pdf Letzter Zugriff 21.06.2012

[Dat12] DATA, Refsnes: *XPath Tutorial*. Website, 2012. – XPath Tutorial der W3Schools http://www.w3schools.com/xpath/default.asp Letzter Zugriff 10.04.2012

[Du09] DUSTIN, Elfride ; U.A.: *Implementing Automated Software Testing*. Pearson Education Inc., 2009

[Eva05] EVANS, Simon: *Simon Evans' Blog: Design pattern for writing unit tests with event driven architectures*. Website, 2005. – http://consultingblogs.emc.com/simonevans/archive/2005/06/06/Design-pattern-for-writing-unit-tests-with-event-driven-architectures.aspx Letzter Zugriff 18.06.2012

[Flo07] FLOURNOY, Jamie: *Journalists, Developers Puzzled by Android SDK's License*. Website, 2007. – http://www.pervasivecode.com/blog/2007/11/21/journalists-developers-puzzled-by-android-sdks-license/ Letzter Zugriff 13.04.2012

[Gar05] GARFINKEL, Simon: *History's Worst Software Bugs*. Website, 2005. – http://www.wired.com/software/coolapps/news/2005/11/69355?currentPage=all Letzter Zugriff 11.04.2012

[GHJV95] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns Elements of Resuable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1995

[Goo] Google: *Content License/Android Developers*. Website, . – http://developer.android.com/license.html Letzter Besuch 13.04.2012

[GP12] Gülcü, Ceki ; Pennec, Sébastien: *Logback Manual*. Website, 2012. – http://logback.qos.ch/manual/index.html Letzter Zugriff 21.06.2012

[Kem12] Kemparaj, Ajay: *AndroidDriver Getting Started With Android Driver*. Website, 2012. – Selenium Wiki Kapite AndroidDriver http://code.google.com/p/selenium/wiki/AndroidDriver Letzter Zugriff 10.4.2012

[KH07] Krüger, Guido ; Hansen, Heiko: *Handbuch der Java-Programmierung 5.Auflage*. Addison-Wesley, 2007

[Mar09] Martin, Robert C.: *Clean Code A handbook of Agile Software Craftmanship*. Prentice Hall, 2009

[Mol09] Mollhauer, Uwe: Jenseits von Capture & Replay. In: *OBJEKTspektrum* (2009), Oktober

[Ou] Oracle ; u.a.: *Designing a Remote Interface*. Website, . – http://docs.oracle.com/javase/tutorial/rmi/designing.html Letzter Zugriff 10.04.2012

[Ou10a] Oracle ; u.a.: *Export (Java 2 Platform SE v.1.4.2)Exception*. 2010. – http://docs.oracle.com/javase/1.4.2/docs/api/java/rmi/server/ExportException.html Letzter Zugriff 10.04.2012

[Ou10b] Oracle ; u.a: *Java TM Logging Overview*. Website, 2010. – http://docs.oracle.com/javase/1.4.2/docs/guide/util/logging/overview.html Letzter Zugriff 21.06.2012

[Ou10c] Oracle ; u.a.: *Package java.rmi*. 2010. – Java RMI Package Overview http://docs.oracle.com/javase/1.4.2/docs/api/java/rmi/package-summary.html Letzter Zugriff 10.04.2012

[Ou11] Oracle ; u.a.: *Arrays (Java Platform 6)*. Website, 2011. – http://docs.oracle.com/javase/6/docs/api/java/util/Arrays.html Letzter Zugriff 10.06.2012

[Ou12a] Oracle ; u.a.: *Java Network Launch Protocol*. Website, 2012. – http://docs.oracle.com/javase/tutorial/deployment/deploymentInDepth/jnlp.html Letzter Zugriff 10.4.2012

[Ou12b] Oracle ; u.a.: *Structure of the JNLP File*. Website, 2012. – http://docs.oracle.com/javase/tutorial/deployment/deploymentInDepth/jnlpFileSyntax.html Letzter Zugriff 11.4.2012

[Ou12c] Oracle ; u.a.: *Trail: RMI*. Website, 2012. – http://docs.oracle.com/javase/tutorial/rmi/index.html Letzter Zugriff 10.4.2012

[Por] Porrello, Anne M.: *Death and Denial: The Failure of the THERAC-25, A Medical Linear Accelerator*. Website, . – http://users.csc.calpoly.edu/~jdalbey/SWE/Papers/THERAC25.html Letzter Zugriff 11.04.2012

[Pre] Prechelt, Lutz: *Sicherheit: Therac-25*. Online PDF, . – http://www.inf.fu-berlin.de/inst/ag-se/teaching/V-AWS-2011/22_Sicherheit.pdf Letzter Zugriff 11.04.2012

[QI] QM-Infocenter: *Testen von Software*. Website, . – http://www.qm-infocenter.de/qm/o_bs.asp?task=4&basic_id=212199319-46&bt=00025.00050 Letzter Zugriff 11.04.2011

[Ros12a] Rosenvold, Kristian: *Grid2 - selenium - Browser automation framework - Google Project Hosting*. Website, 2012. – Selenium Wiki Grid 2 http://code.google.com/p/selenium/wiki/Grid2 Letzter Zugriff 16.04.2012

[Ros12b] Rosenvold, Kristian: *RemoteWebDriverServer Setting up the remote webdriver server*. Website, 2012. – Selenium Wiki Kapitel Remote WebDriver Server http://code.google.com/p/selenium/wiki/RemoteWebDriverServer Letzter Zugriff 10.4.2012

[Sau11] SauceLabs: *Sauce REST API - Sauce Labs*. Website, 2011. – Version 1.0 http://saucelabs.com/docs/saucerest Letzter Zugriff 17.04.2012

[Sau12] SauceLabs: *Frequently Asked Questions - Sauce Labs*. Website, 2012. – http://saucelabs.com/tech-resources Letzter Zugriff 17.04.2012

[Sem12] Semerau, Luke: *org.openqa.selenium.android.app.MainActivity*. Subversion Checkin, 2012. – Source Code der AndroidWebdriver App in der Revision r15471 http://code.google.com/p/selenium/source/browse/trunk/android/src/org/openqa/selenium/android/app/MainActivity.java?r=15471 Letzter Zugriff 16.04.2012

[SL05] Spillner, Andreas ; Linz, Tilo: *Basiswissen Softwaretest 3.Auflage*. dpunkt.verlag, 2005

[Suy00] Suydam, Terren: *Java Tip 91: Use nested exceptions in a multi-tiered environment*. Website, 2000. – http://www.javaworld.com/javaworld/javatips/jw-javatip91.html Letzter Zugriff 12.04.2012

[Tol11] Tolfsen, Andreas: *RemoteWebDriver Information about the RemoteWebDriver*. Website, 2011. – Selenium Wiki Kapitel Remote WebDriver http://code.google.com/p/selenium/wiki/RemoteWebDriver Letzter Zugriff 10.4.2012

[Ull12] Ullenboom, Christian: *Java ist auch eine Insel 10.Auflage*. 2012. – http://openbook.galileocomputing.de/javainsel/ Letzter Zugriff 30.06.2012

[Web11] *Interface WebDriver*. Website, 2011. – Selenium-API WebDriver http://selenium.googlecode.com/svn/trunk/docs/api/java/index.html Letzter Zugriff 10.04.2012

A. Anhang

A.1. Datenträger

Auf dem hier zu findenden Datenträger ist der Sourcecode der Anwendung zu hinterlegt. Bei der digital veröffentlichten Version dieser Arbeit, befindet sich der Sourcecode auf der CD der Veröffentlichung.

A.2. Abhängigkeiten des Resourcemanagementservers

Zum kompilieren der Anwendung werden drei Pakete benötigt, zu denen eine Abhängigkeit besteht.

- Selenium Server 2.21.0 (http://www.seleniumhq.org/)
- SLF4J API 1.6.6 (http://www.slf4j.org/)
- Logback Core und Classic 1.0.6 (http://logback.qos.ch)

Diese Abhängigkeiten können mit Maven mittels der auf dem Datenträger mitgelieferten pom.xml aufgelöst werden.

A.3. Klassendiagramm Gesamtsystem

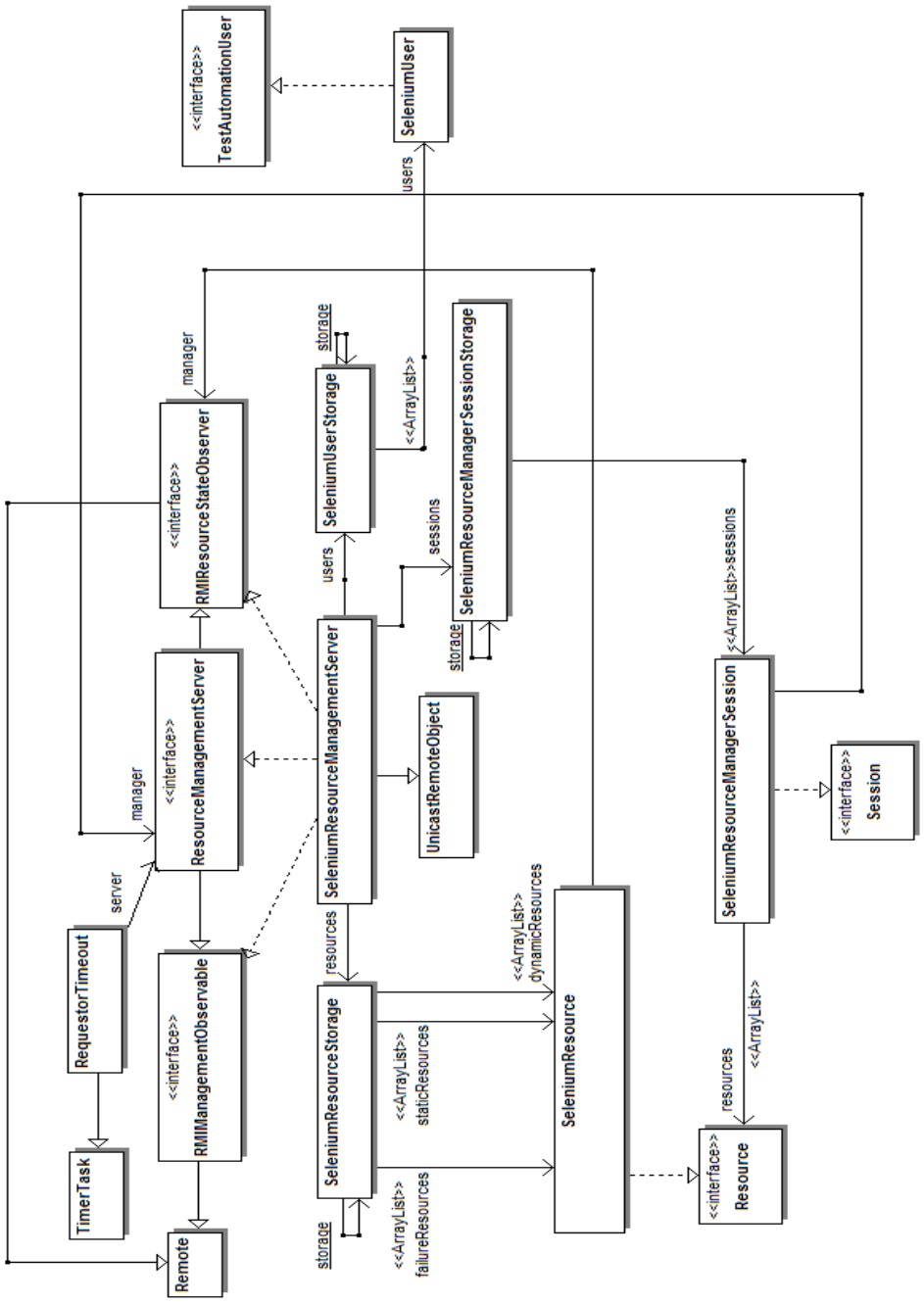

Abbildung 31.: *Klassendiagramm des Gesamtsystems dargestellt ohne seine Methoden*

A.4. Konfigurationsdatei des Logback-Loggers

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <configuration scan="true" scanPeriod="180 seconds">
3          <!-- Basic logging for INFO loggings -->
4          <appender name="STDOUT_INFO" class="ch.qos.logback.core.ConsoleAppender">
5                  <encoder>
6                          <pattern>%-5level: %date{dd.MM.yyyy HH:mm:ss} %message %n</
                               pattern>
7                  </encoder>
8                  <filter class="ch.qos.logback.classic.filter.LevelFilter">
9                          <level>INFO</level>
10                         <onMatch>ACCEPT</onMatch>
11                         <onMismatch>DENY</onMismatch>
12                 </filter>
13         </appender>
14
15         <!-- Extended logging for WARN and ERROR loggings -->
16         <appender name="STDOUT_WARN" class="ch.qos.logback.core.ConsoleAppender">
17                 <encoder>
18                         <pattern>%-5level: %date{dd.MM.yyyy HH:mm:ss} at [%file] %
                               message %n</pattern>
19                 </encoder>
20                 <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
21                         <level>WARN</level>
22                 </filter>
23         </appender>
24
25         <appender name="LOGFILE" class="ch.qos.logback.core.FileAppender">
26                 <file>logfile.log</file>
27                 <append>true</append>
28                 <encoder>
29                         <pattern>%-5level: %date{dd.MM.yyyy HH:mm:ss} %message %n</
                               pattern>
30                 </encoder>
31                 <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
32                         <level>INFO</level>
33                 </filter>
34
35         </appender>
36         <appender name="ERROR_LOGFILE" class="ch.qos.logback.core.FileAppender">
```

```
37                  <file>errorlogfile.log</file>
38                  <append>true</append>
39                  <encoder>
40                          <pattern>%-5level: %date{dd.MM.yyyy HH:mm:ss} at [%file:%
                                method:%line] %message %n</pattern>
41                  </encoder>
42                  <filter class="ch.qos.logback.classic.filter.ThresholdFilter">
43                          <level>ERROR</level>
44                  </filter>
45          </appender>
46
47          <root level="DEBUG">
48                  <appender-ref ref="STDOUT_WARN"/>
49                  <appender-ref ref="STDOUT_INFO"/>
50                  <appender-ref ref="LOGFILE"/>
51                  <appender-ref ref="ERROR_LOGFILE"/>
52          </root>
53  </configuration>
```

Listing 22: *Konfiguration des Logback-Loggers*

Glossar

GUI Grahical User Interface (grafische Oberfläche)

JNLP Java Network Launching Protocol

RMI Remote Method Invocation

SLF4J Simple Logging Facade for Java

Versicherung über die Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16(5) APSO-TI-BM ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

Hamburg, 4. Juli 2012	
Ort, Datum	Unterschrift