Freie Universität Berlin

Masterarbeit am Institut für Informatik der Freien Universität Berlin

Arbeitsgruppe Software Engineering

Erweiterung des Saros Test Frameworks für die HTML-GUI

Jannis Fey
Matrikelnummer: 5041473
jannis.fey@fu-berlin.de

Betreuer: Franz Zieris
Eingereicht bei: Prof. Dr. Lutz Prechelt
Zweitgutachter: Prof. Dr. Claudia Müller-Birn

Berlin, 13. November 2018

Zusammenfassung

In dem Saros-Projekt werden Plugins für Entwicklungsumgebungen bereitgestellt, welche Software-Entwickler bei der verteilten Paarprogrammierung in Echtzeit unterstützen. Das Saros Test Framework (STF) ist ein eigenständiges Framework, welches Oberflächentest für das Saros Eclipse-Plugin ermöglicht. Die neue HTML-GUI, die für das Eclipse- und IntelliJ-Plugin entwickelt worden ist, lässt sich vor der Arbeit nicht für STF-Tests nutzen, da die notwendige API fehlte.

In dieser Arbeit wird das STF um eine Schnittstelle erweitert, die es ermöglicht, Elemente in der neuen HTML-GUI anzusteuern. Eine neue Projekt-Struktur trägt dazu bei, dass die Komplexität des Frameworks vereinfacht wird. Durch die neue API und weiteren Anpassungen der Test-Infrastruktur ist es nun möglich, bestehende STF-Tests über die neue HTML-GUI durchzuführen. Eine vollständige Integration des STF für das IntelliJ-Plugin konnte jedoch nicht erreicht werden.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides Statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

13. November 2018

Jannis Fey

Inhaltsverzeichnis

1 Einführung

1.1 Saros

Saros ist eine Erweiterung für Entwicklungsumgebungen, welches verteilte Paarprogrammierung ermöglicht. Das Saros Projekt wird von der AG Software Engineering organisiert und hauptsächlich im Rahmen von Abschlussarbeiten weiterentwickelt. Das Open-Source Projekt wurde 2006 gestartet und seitdem stetig erweitert. [3]

Anders als bei klassischer Paarprogrammierung ist die örtliche Nähe der Entwickler durch Saros nicht mehr notwendig. Saros ermöglicht es Softwareentwicklern an zwei verteilten Rechnern gemeinsam den gleichen Quelltext zu bearbeiten. Dazu sind die Rechner über das Netzwerk miteinander verbunden. Saros sorgt dafür, dass der Quelltext geteilt wird und in einem konsistenten Stand bleibt.

Saros wird als eine Multiplatform-Anwendung entwickelt. D.h. es gibt ein Plugin für Eclipse und ein Weiteres für die IntelliJ IDE. Neben der verteilten Paarprogrammierung bietet Saros weitere Möglichkeiten, um mit verbunden Nutzern zu interagieren. Es gibt z.B. einen integrierten Chat zur Kommunikation oder ein virtuelles Whiteboard um Ideen grafisch auszutauschen.

1.2 Saros Test Framework

Das Saros Test Framework (im weiteren Verlauf kurz STF genannt) ist eine, in der Diplomarbeit von Sandor Szuecs [8] entwickelte, Test-Bibliothek um GUI-Tests für Saros zu schreiben. Das STF ist ein eigenständiges Softwareprodukt und somit unabhängig von Saros bzw. den Plugins für die IDEs.

Das Ziel des Frameworks ist es, Ende-zu-Ende-Tests zu ermöglichen. Diese Tests werden in sogenannten STF-Tests realisiert. Ein STF-Test ist ein Funktionstest um Features von Saros zu testen. Dazu stellt das Framework Funktionen bereit, um bestimmte Elemente der GUI anzusteuern wie z.B. einen Button oder ein Eingabefeld. Durch eine Reihe von Operationen kann ein ganzer STF-Test geschrieben werden, welcher Saros über die obersten Ebene (die Benutzeroberfläche) testet.

Das Framework übernimmt dabei zwei Aufgaben:

1. Zugriff auf den jeweiligen Tester: Ein Tester ist im Prinzip eine Instanz einer Entwicklungsumgebung mit installiertem Plugin für Saros. Der Zugriff wird dabei über Java RMI (siehe Abschnitt 2.2) realisiert.

2. Steuerung und Synchronisation der Tester: Als Bibliothek stellt das STF Methoden bereit, um die GUI anzusteuern. Im STF-Test werden diese Methoden in einer logischen Reihenfolge bei den Testern ausgeführt, woraufhin dann das korrekte Verhalten von Saros überprüft wird.

1.3 Ausgangssituation

Zu beginn meiner Arbeit wird das STF nur für Funktionstest verwendet, welche die Eclipse-GUI ansteuern. Das bedeutet, dass es bereits viele STF-Tests gibt die das Plugin für Eclipse testen, aber keine die das Plugin für IntelliJ testen und dazu die IntelliJ-GUI ansteuern.

Im Zuge der Weiterentwicklung von Saros wurde eine neue HTML-GUI für Saros entwickelt. Zuletzt wurde die HTML-GUI in der Arbeit von Marius Schidlack [6] neu implementiert. Die neue GUI wird unabhängig von einer IDE entwickelt, sodass sie sowohl in Saros für Eclipse sowie in Saros für IntelliJ über einen eingebetteten Browser genutzt werden kann. Auch für zukünftige Versionen von Saros, wie z.B. Saros für Netbeans, soll die HTML-GUI verwendet werden. Die HTML-GUI ist noch nicht vollständig und befindet sich weiter in der Entwicklung. Grundlegende Funktionen von Saros wurden noch nicht in der HTML-GUI implementiert.

Die neue HTML-GUI wird noch nicht für die automatisierten Ende-zu-Ende-Tests verwendet. Das Saros Test Framework kann ohne die Änderungen dieser Arbeit nicht einfach auf die neuen HTML-Elemente zugreifen. Der einzige Zugang zu den HTML-Elementen besteht über den eingebetteten Browser. Um HTML-Elemente anzusprechen kann Javascript in diesem Browser ausgeführt werden.

Die Komponenten des Saros Test Frameworks sind nicht sauber vom Saros-Projekt getrennt. Sie liegen verstreut in verschiedenen Teil-Projekten von Saros. So ist z.B. der größte Teil des STF ein Teil des Eclipse-Plugins, wohingegen andere STF-Komponenten bereits Bestandteil der neuen HTML-GUI sind.

1.4 Probleme

Ein Problem besteht darin, dass es noch keine API gibt, um HTML-Elemente in der HTML-GUI über das STF anzusteuern. Zwar lassen sich die HTML-Elemente über Javascript ansteuern, dies stellt aber noch keine strukturierte API dar. Eine API soll dem Java-Entwickler das Schreiben eines STF-Tests erleichtern, ohne dass er dafür wissen muss, wie und welcher Javascript-Code im Browser auszuführen ist, um eine bestimmtes HTML-Element anzusprechen.

Des Weiteren besteht ein Problem in der Strukturierung des Quellcodes im Saros-Projekt. Produktiv-Code, Test-Code, STF-Code, STF-Tests und Test-Code für das STF sind nicht klar von einander getrennt. Insbesondere wurde der Code für das STF sowie die Tests dafür als Teil des Eclipse-Plugins eingeordnet. Mit der Einführung des IntelliJ-Plugins und der neuen HTML-GUI ist das STF allerdings kein Framework mehr, welches ausschließlich zum Testen des Eclipse-Plugins genutzt werden soll. Außerdem sorgt eine neue einheitliche Strukturierung für eine deutlich bessere Lesbarkeit bzw. Übersicht und Wartbarkeit des STF.

Durch das Erweitern des STF um eine neue API für die HTML-GUI, würde es die Möglichkeit geben, alle bestehenden STF-Tests so umzuschreiben, dass diese nun die

Funktionen von Saros über die neue GUI überprüfen. Der Umbau verursacht allerdings einen erheblichen Aufwand, da jeder einzelne STF-Test dazu angepasst werden müsste. Dies stellt keine akzeptable Lösung für das Problem dar. Würde stattdessen nur die zu Grunde liegende Infrastruktur des Test Framework angepasst werden, könnten die bestehenden Tests weitestgehend unangetastet bleiben.

Da Saros eine Multiplattform-Anwendung ist und die HTML-GUI bereits in dem IntelliJ-Plugin zum Einsatz kommt, ist es sinnvoll Funktionstest auch für die IntelliJ-Version von Saros zu starten. Da bisher keine Funktionstests über das Framework für IntelliJ möglich sind, muss auch an dieser Stelle die Infrastruktur des STF angepasst werden.

1.5 Ziele

Das erste Ziel dieser Arbeit ist es, die Interaktion mit HTML-Elementen über eine API zu ermöglichen. Dieses Ziel ist erreicht, wenn ein STF-Test geschrieben werden kann, der Elemente wie z.B. einen Button, in der HTML-GUI ansteuern kann. Genauer gesagt soll sowohl ein lesender als auch schreibender Zugriff auf die HTML-Elemente möglich sein. D.h. der Zustand bzw. die Attribute und Werte eines Elements lassen sich über die API des STF auslesen und verändern.

Die Modularisierung des Saros Test Framework ist das zweite Ziel. Die Zielvorstellung ist es, alle STF-Bestandteile unabhängig vom Produktiv-Code sowie den IDE-Plugins in ein eigenständiges Projekt auszulagern. Dazu soll der STF-Code sowie die Tests dafür in einer neuen Komponente untergebracht werden. Die STF-Tests zum Testen der Features von Saros sollen in einer weiteren separaten Komponente ausgelagert werden.

Das dritte Ziel sichert das Vorhaben die bestehenden STF-Tests für die neue HTML-GUI zu übernehmen. Das Ziel ist erreicht, wenn es die Infrastruktur erlaubt, ohne größere Anpassung des STF-Tests die bestehende Funktionalität von Saros zu überprüfen und dazu statt der alten Eclipse-GUI die neue HTML-GUI verwendet. Das STF soll trotzdem noch in der Lage sein, über die alte GUI zu testen, solange sich die HTML-GUI noch in der Entwicklung befindet.

Sind die ersten drei Ziele erreicht, dann ist das vierte und letzte Ziel die Integration des Saros Test Framework für das IntelliJ-Plugin. Bei der Entwicklung von Saros für IntelliJ wurde das STF zunächst gar nicht berücksichtigt. In dieser Arbeit soll nun versucht werden das Framework im Nachhinein einzubinden, sodass STF-Test auch über das IntelliJ-Plugin für Saros ausgeführt werden können.

1.6 Abgrenzung

In dieser Arbeit geht es darum, die Infrastruktur des Test Framework soweit anzupassen, dass STF-Tests über die HTML-GUI ausgeführt werden können. Bei der Übernahme von bestehenden STF-Tests wird es vorkommen, dass einige unter Verwendung

der HTML-GUI fehlschlagen. Dieser Umstand ist dem unvollständigen Zustand der HTML-GUI geschuldet. Ein STF-Test z.B. für den Chat kann offensichtlich nicht erfolgreich werden, wenn der Chat noch gar nicht über die HTML-GUI funktioniert. Allerdings soll es schon möglich sein, den bestehenden STF-Test zu starten, um dann beobachten zu können, wie dieser Aufgrund von noch nicht implementierten Funktionen der HTML-GUI fehlschlägt. In dieser Arbeit soll es, bis auf kleine Anpassungen, nicht um die (Weiter)Entwicklung der HTML-GUI gehen.

Die volle Unterstützung von STF-Tests in IntelliJ ist ebenfalls kein Bestandteil der Arbeit. Das Framework wird soweit vorbereitet, dass es möglich sein wird STF-Tests für IntelliJ auszuführen. Es soll nur die HTML-GUI über den eingebetteten Browser in IntelliJ angesprochen werden. D.h. die im ersten Ziel eingeführte API soll für die HTML-GUI unter Eclipse sowie auch unter IntelliJ verwendet werden können. GUI-Elemente von IntelliJ wie z.B. das Menü oder der Dateibaum sollen nicht über das Framework ansprechbar sein.

1.7 Umsetzung

Als erster Schritt war es notwendig, alle grundlegenden HTML-Elemente wie z.B. Buttons, Dropdowns oder Eingabefelder via Javascript über das STF in der HTML-GUI anzusprechen. Dafür musste ich entsprechende Klassen und Interfaces zu Ansteuerung der HTML-Elemente in Java, wie in Kapitel 4 zu lesen ist, angelegen. Damit die API auch für später hinzugefügte HTML-Elemente funktioniert, ist in Abschnitt 4.5 ein einfaches Regelwerk für die Implementierung der Saros-HTML-GUI aufgeführt.

Im nächste Schritt sollen die unterschiedlichen Teile des STF in neue Komponenten ausgelagert werden. In Kapitel 5 ist beschrieben, dass zu den bisherigen Komponenten von Saros eine neue STF-Komponente und eine STF-Test-Komponente hinzugefügt wurde. Ich habe zunächst den alten Zustand der Projektinfrastruktur aufgeführt und anschließend in Abschnitt 5.3 und 5.4 die Herausforderungen und Probleme auf dem Weg zum neuen Zustand dokumentiert.

Mein dritter Schritt verfolgte das Ziel die bestehenden STF-Tests zu behalten. Das Framework entscheidet selbständig wie die jeweilige GUI angesprochen werden muss, wodurch eine Anpassung der STF-Tests nicht mehr notwendig ist. In Kapitel 6 wird beschrieben, welche Anpassungen dazu im Saros Test Framework konkret getätigt wurden. Die vorhanden STF-Infrastruktur bietet Interfaces, welche verwendet werden konnten, um mithilfe einer neuen Implementation auf die neue HTML-GUI umzustellen. Der bisherige teilweise sehr unvollständige Zustand der HTML-GUI hat zur Folge, dass viele STF-Tests beim Testen über die neue GUI nun fehlschlagen.

Zum Ende meiner Arbeit habe ich versucht das STF für IntelliJ zu starten, sodass es möglich war, Methoden über RMI in einer IntelliJ Instanz auszuführen. Die neue STF-Komponente aus dem zweiten Schritt ist dazu gleichzeitig ein eigenständiges IntelliJ-Plugin geworden, welches zusätzlich zum IntelliJ-Plugin für Saros startet. In Kapitel 7 beschreibe ich wie diese neue Plugin im Detail aussieht. Dabei gibt es ein großes Problem beim Austausch von Objekten zwischen dem Plugin für das STF und dem

Plugin für Saros. Fünf verschiedenen Varianten, welche versuchen dieses Problem zu lösen sind unter Abschnitt 7.2.1 bis 7.2.5 dokumentiert.

In Kapitel 8 ist meine Arbeit kurz zusammengefasst, bevor ich in Kapitel 9 ein Fazit ziehe. Abschließend gebe ich in Kapitel 10 ein Ausblick für noch anstehende Arbeiten im Saros-Projekt.

2 Grundlagen

2.1 STF Aufbau

Zum Testen von Saros über das STF können gleichzeitig bis zu vier Eclipse-Instanzen gestartet werden. Über die Variablen bzw. Tester `ALICE`, `BOB`, `CARL` und `DAVE` kann jede Instanz eigenständig angesteuert werden.

Um zwischen Testausführung und den jeweiligen Instanzen zu kommunizieren, wird das RMI-Framework (Abschnitt 2.2) verwendet. Dieses basiert auf einer Client-Server-Architektur. Der eigentliche STF-Test wird als klassischer JUnit-Test ausgeführt. Die Testausführung fungiert dabei als Client, die Instanz der Entwicklungsumgebung als Server. Über die RMI-Schnittstelle werden Methoden zur Steuerung der IDE vom Client auf dem Server aufgerufen. Zum Testen von Saros über das STF wird statt einem Server mit mehreren Clients die Architektur auf diese Weise verwendet, dass nur ein Client (Test) mit mehreren Server (IDEs) kommuniziert.

Die Abbildung 1 zeigt den Aufbau der Client-Server-Architektur des Saros Test Framework noch einmal grafisch. In Rot ist der STF-Test als Client dargestellt , welcher über JUnit ausgeführt wird. Dieser ruft über RMI die Server der Tester auf, welche in Blau dargestellt sind. Zur Unterscheidung der Server werden verschiedene Ports verwendet.

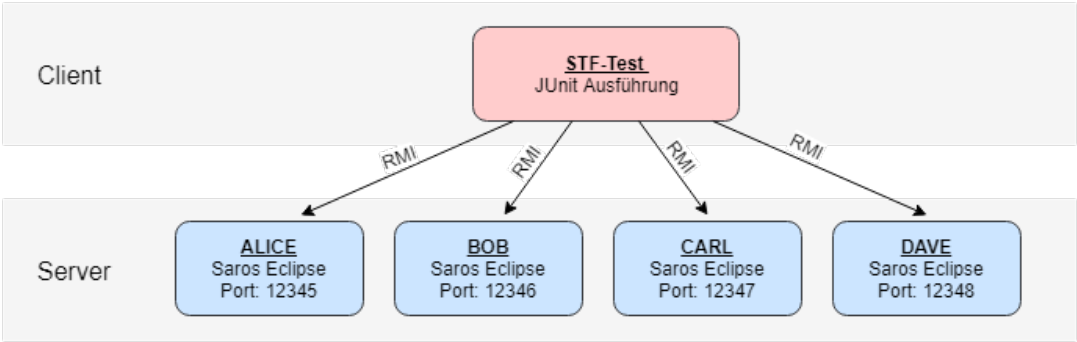

Abbildung 1: Aufbau der Client-Server Architektur des STF mit den Servern ALICE, BOB, CARL und DAVE. Auf diesen kann der Client, welcher den Test ausführt, über RMI Methoden aufrufen.

2.2 Remote Method Invocation

Das Java Remote Method Invocation Framework (RMI) ermöglicht es, Methoden über das Netzwerk in einer anderen Java-Virtuellen Maschine (JVM) aufzurufen. Dazu verwendet das Framework eine Client-Server-Architektur, um vom Client aus Methoden eines Objekts auf der Server Seite aufzurufen.

Dabei wird die Methode beim Client zunächst auf einem Stub aufgerufen. Der Stub ist ein Stellvertreter-Objekt beim Client, welcher den Methodenaufruf entgegen nimmt

und ihn via TCP/IP an das richtige Objekt beim Server weiterleitet. Dieses Stub-Objekt ist in der Regel ein Interface, dass nur den Zugriff auf die Methoden bereitstellt. Implementiert werden die Methoden dann in einer Klasse auf Seite des Servers, welche zusätzlich von `java.rmi.Remote` erben muss. Damit die Methoden über RMI aufgerufen werden können, muss beim Starten des RMI-Servers eine Instanz des Objekts an die RMI-Registry gebunden werden. Beim Binden wird ein eindeutiger Name verwendet, über den sich der Client die Instanz zum Aufrufen der Methoden holt. Nachdem der Server die Methode ausgeführt hat, wird der Rückgabewert wieder mit TCP/IP an den Client über das Netzwerk zurück gesendet. [5]

In Abbildung 2 wird der Aufruf einer Methode über RMI anhand einer Grafik veranschaulicht. Dazu wird das bereits oben beschriebenen Vorgehen in vier einzelnen Schritten gezeigt. In der Grafik wird ebenfalls deutlich, dass Server und Client in unterschiedlichen JVMs ausgeführt werden.

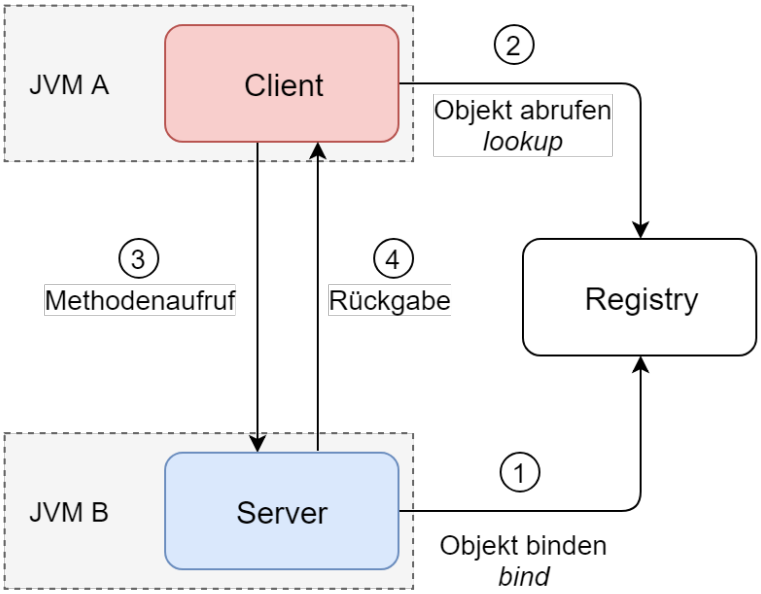

Abbildung 2: 1. Beim Starten des RMI-Servers wird eine Instanz des Objekts an die Registry mit einem eindeutigen Namen gebunden. 2. Der Client beschafft sich über den vergebenen Namen eine Referenz auf das Stellvertretende-Objekt des gebundenen Objekts. Dieser Vorgang wird auch lookup genannt. 3. Der Client kann nun eine Methode des Objekts auf der Server Seite aufrufen. 4. Der Rückgabewert der Methode wird an den Client zurückgegeben.

Das Binden der notwendigen Objekte stellt in Saros der `STFController` sicher. Er sorgt dafür, dass alle Klassen die über RMI ansprechbar sein sollen, über die jeweilige IP und Port erreichbar sind. Für jede Instanz des Plugins wird der `STFController` einzeln gestartet. Dieser fährt dann den Server hoch, um anschließend die Objekte an die Registry zu binden. Für das Saros Test Framework bedeutet dies, dass dadurch vom STF-Test (Client) Methoden auf den jeweiligen Instanzen der Tester (Server) aufgerufen werden können, welche dann wiederum auf die GUI zugreifen, um diese zu steuern.

2.3 STF-Test

Anders als ein Komponententest testet ein STF-Test keine einzelnen Komponenten des Codes, sondern er überprüft Saros aus Sicht eines realen Benutzers. Das heißt, bei einem STF-Test werden Eingaben wie z.B. Mausklicks oder Tastatureingaben simuliert um das Verhalten von Saros zu prüfen. Dabei ist es notwendig, die GUI vom Code aus anzusteuern. Diese Aufgabe übernimmt das Saros Test Framework. Mithilfe verschiedener Bots (siehe Abschnitt 2.4) werden verschiedene Benutzer-Interaktionen mit Saros simuliert, woraufhin dann das korrekte Verhalten von Saros im Test überprüft werden kann.

Im Saros-Projekt erbt jeder STF-Test von der `StfTestCase` bzw `StfHTMLTestCase` Klasse welche Hilfsmethoden zur Steuerung des Test Framework bereitstellt. So ist es z.B. für jede STF-Test Klasse notwendig, über `select(ALICE, BOB, ...)` anzugeben welche Tester für diesen Test verwendet werden. Dabei wird überprüft, ob die Instanz der übergebenen Tester initialisiert und via RMI erreichbar ist. Außerdem wird sicher gestellt, dass die IDE vor dem Start des eigentlichen Test immer gleich initialisiert ist. Dazu werden z.B. immer die gleichen Einstellungen für Saros ausgewählt oder eine bestimmte Ansicht der Entwicklungsumgebung zum Start des Tests festgelegt. Für HTML-STF-Tests wird zusätzlich geprüft, ob alle beteiligten Tester die neue HTML-GUI verwenden.

Neben den STF-Tests gibt es noch STF-Self-Tests. Diese unterscheiden sich vom Aufbau nicht von einem normalen STF-Test, haben aber die Aufgabe das Test Framework an sich zu testen und nicht spezielle Features von Saros. In einem STF-Self-Test wird getestet, ob überhaupt das Framework an sich funktioniert um z.B. das Kontextmenü für eine Datei zu öffnen um dort dann einen Eintrag anzuklicken.

2.4 Bots

Das Saros Test Framework wird mithilfe von verschiedenen Bots (von Roboter) gesteuert, wobei jeder Bot einen eigenen Aufgabenbereich hat. Die Bots werden dann in automatisierten STF-Tests verwendet, um über gezielte Methodenaufrufe, die Oberfläche von Saros anzusteuern. Dieser Abschnitt soll einen Überblick über drei, für das STF relevante, Bots geben und kurz den jeweiligen Aufgabenbereich beschreiben. Neben den unten erwähnten gibt es noch weitere Bots, z.B. zur Steuerung des Netzwerkverkehrs, welche aber für diese Arbeit irrelevant sind.

SuperBot Der `SuperBot` bietet dem STF zum einen den Zugriff auf die verschiedenen Fenster sowie Menüs der Entwicklungsumgebung wie z.B. Menüleiste, Konsole oder Dateiexplorer. Zum Anderen übernimmt der `SuperBot` die Steuerung der Dialoge die von dem Plugin in der Entwicklungsumgebung geöffnet werden können.

RemoteBot Der `RemoteBot` bietet über den `SWTBot` Zugriff auf sämtliche SWT-Elemente wie Buttons, Listen, Checkboxen, usw. Der verwendete `SWTBot` ist kein Teil des

STF, sondern wird über eine externe Bibliothek hinzugefügt. Da der `RemoteBot` nur SWT unterstützt, kann er nur für STF-Test für Eclipse genutzt werden.

HTMLBot Der `HTMLBot` ist im ähnlich wie der `RemoteBot` nur nicht für SWT-Elemente, sondern für HTML-Elemente die in der HTML-GUI innerhalb des eingebetteten Browsers angezeigt werden. Der HTMLBot ist neu und wurde erst im Rahmen dieser Arbeit in das STF integriert bzw. weiter implementiert.

2.5 Javascript API

Da die HTML-GUI vom eigentlichen Saros-Plugin ausgelagert wurde, benötigt es eine Schnittstelle um zwischen Plugin und GUI zu komunizieren. Saros sowie das STF sind in Java geschrieben, wohingegen die HTML-GUI React[1] bzw. Javascript verwendet. Die neue HTML-GUI ist im Grunde eine Single-Page-Application die in jedem Browser angezeigt werden kann.

In der Entwicklungsumgebung wird die HTML-GUI von Saros in einem eingebetteten Browser angezeigt, welcher sich über ein entsprechendes `JQueryBrowser` Objekt in Java ansprechen lässt. Über die Methode `run(String script)` kann Javascript direkt im Browser auf der aktuellen Seite ausgeführt werden. Dadurch ist es dem STF möglich die HTML-GUI zu manipulieren um z.B. einen Button anzuklicken, ein Textfeld mit Inhalt befüllen oder den Zustand einer Checkbox auszulesen. Es ist sogar möglich komplexe Scripte mit Schleifen, Arrays, usw. über die `run(String script)` Methode des eingebetteten Browsers auszuführen.

2.6 PicoContainer

In Saros wird das Entwurfsmuster Dependeny Injection mithilfe der Bibliothek PicoContainer[2] verwendet. Durch den PicoContainer müssen Abhängigkeiten zu Klassen, nicht zwingend bei der Initialisierung festgelegt, sondern können auch erst zur Laufzeit hinzugefügt werden. Dazu müssen zunächst die benötigten Klassen dem PicoContainer als Komponente hinzugefügt werden. Anschließend kann jederzeit auf die Klasse über den Container zugegriffen werden.[7]

In Saros wird der PicoContainer über die Klasse `ContainerContext` gesteuert. Über verschiedene Factorys wie z.B. `SarosCoreContextFactroy` oder `SarosIntellijContext Factroy` werden die benötigten Klassen je nach IDE beim Starten von Saros dem PicoContainer hinzugefügt. Die statische Methode `getComponent(Class class)` kann an jeder Stelle im Code aufgerufen werden, um eine benötigte Klasse aus dem PicoContainer zu laden. Auch das STF benutzt den PicoContainer an vielen Stellen, deshalb ist der `ContainerContext` zum Starten des Test Framework zwingend notwendig.

[1] https://reactjs.org/
[2] http://picocontainer.com/

3 Einarbeitung

In diesem Abschnitt wird insbesondere der Ablauf des Code-Review-Prozess im Saros-Projekt behandelt. Außerdem wird beschrieben wie ich mich in den Code eingearbeitet und mir einen ersten Überblick verschafft habe.

3.1 Gerrit

Gerrit[3] ist ein Code-Review-Tool, welches auf Git aufbaut und das Entwicklerteam bei der Qualitätskontrolle des Codes unterstützt. Änderungen, auch Changes genannt, werden von Git in einem separaten Branch verwaltet. Die Änderung des Codes wird in sogenannten Patch-Sets durchgeführt, die dann über Gerrit anderen Entwicklern zur Durchsicht bereit gestellt werden. Der Reviewer kann ein Patch-Set über eine Weboberfläche begutachten. Über eine Kommentarfunktion können Änderungen vorgeschlagen bzw. diskutiert werden. Der Entwickler kann anschließend mit weiteren Patches auf die Anmerkungen des Reviews eingehen und seinen Code weiter anpassen. Sobald ein Change vollständig akzeptiert wird, kann dieser in den Master-Branch des Projekts integriert werden.

3.2 Umstiegt auf GitHub

Am 16. Juli 2018, also mitten in meiner Arbeit, wurde das Code-Review System im Saros-Projekt von Gerrit auf GitHub umgestellt.

GitHub[4] bietet neben der Versionsverwaltung auch eine Möglichkeit den Review-Prozess in Software-Projekten zu unterstützen. Dazu eröffnet der Entwickler zunächst einen neuen Branch, den er mit seine Änderungen bzw. Commits auf GitHub bereitstellt. Über die Weboberfläche von GitHub kann für diesen Branch ein Pull-Request erstellt werden. Andere Entwickler können dann über diesen Pull-Request die Änderungen, im Vergleich zu einem anderen Branch z.B. dem master, einsehen. Das Code-Review läuft ähnlich wie bei Gerrit ab: Der Code kann kommentiert, abgelehnt oder akzeptiert werden. Um eine Änderung am Code für den Pull-Request vorzunehmen, genügt ein neuer Commit in den entsprechenden Branch. Der neue Commit wird dann automatisch zum Pull-Request hinzugefügt, da sich dieser auf den jeweiligen Branch stützt. Ist ein Pull-Request akzeptiert worden, kann der gesamte Branch in den Master-Branch integriert werden.

Da sich zum Zeitpunkt der Umstellung auf GitHub noch offene und für mich relevante Changes aus der Arbeit von Marius Schidlack [6] in Gerrit befanden, musste ich diese nach GitHub migrieren. D.h. die Commits mussten aus Gerrit gezogen und lokal in einen neuen Branch verschoben werden. Diesen neue Branch habe ich anschließend in das Repository auf GitHub hochgeladen wo ich dann über die Weboberfläche eine Pull-Request für den Branch eröffnet habe.

[3] https://www.gerritcodereview.com/
[4] https://github.com/

3.3 Einstieg

Nach dem Starten des ersten STF-Test für Saros habe ich schnell bemerkt, dass für jede Anpassung im Code die Test Instanzen von Eclipse neu gestartet werden müssen. Dieses Vorgehen ist beim Testen von neuem Code sehr mühselig und zeitraubend aber notwendig.

Durch das Review von älteren Patches aus der Arbeit von Marius Schidlack[6] hatte ich erste Berührungen mit dem mir bisher unbekannten React-Framework.

Als nächstes habe ich mir bereits vorhandene Patches von Franz Zieris zum Thema HTML-STF-Tests angesehen. In diesem ersten Ansatz wurde der `HTMLBot` eingeführt, welcher in dieser frühen Version allerdings nur Buttons ansteuern konnte. Ich habe mich dazu entschieden den `HTMLBot` beizubehalten und auf diesem Grundgerüst aufzubauen.

Meine Idee war es im ersten Schritt noch mehr Zugriffsmöglichkeiten für weitere HTML-Elemente zu implementieren, um untersuchen zu können, ob diese genauso über die Javascript-API ansteuerbar sind. Im nächsten Kapitel 4 ist zu lesen für welche HTML-Elemente ich eine Schnittstelle implementiert habe und wie die Ansteuerung im Detail abläuft.

4 Interaktion mit HTML-Elementen

4.1 Aufbau HTMLBot

Wie zuvor in Abschnitt 2.4 kurz erklärt, dient der `HTMLBot` zur Ansteuerung einzelner HTML-Elemente im Browser. Über den `HTMLBot` ist es möglich, eine beliebige Ansicht der HTML-GUI anzusteuern. Ein Ansicht wie z.B. die Startseite der HTML-GUI ist im Prinzip eine klassische HTML-Seite, die im eingebetteten Browser angezeigt wird. Mit `ALICE.htmlBot().view(MAIN_VIEW)` lässt sich z.B. die Haupt-Ansicht der HTML-GUI über den HTMLBot beim Tester Alice ansteuern. Der Aufruf gibt dabei eine `RemoteHTMLView` Objekt zurück, über welches weitere HTML-Elemente angesteuert werden können.

4.1.1 Aufbau RemoteHTMLView

Die `RemoteHTMLView` Klasse repräsentiert eine Ansicht der HTML-GUI. Über ein View-Enum und einer weiteren Variable für den Browser wird in dieser Klasse festgehalten, welche Ansicht über welchen Browser erreichbar ist, denn es können durchaus mehrere Browser mit unterschiedlichen Ansichten gleichzeitig in einer IDE geöffnet sein. Über verschiedene Methoden können von hier aus einzelne HTML-Elemente in der GUI angesteuert werden. Außerdem stellt die `RemoteHTMLView` Klasse Methoden bereit, um zu überprüfen, ob ein bestimmtes HTML-Element in der Ansicht überhaupt existiert. Des Weiteren kann geprüft werden, ob eine Seite bereits geöffnet ist. Das gezielte Öffnen bzw. Wechseln einer Ansicht kann ebenfalls über die `RemoteHTMLView` gesteuert werden.

4.1.2 Aufbau RemoteHTMLElement

Für jeden Typ eines HTML-Elements existiert eine eigene Klasse. Für einen Button gibt es z.B. die `RemoteHTMLButton` Klasse. Diese Klasse dient als Vermittler zwischen der Repräsentation des Elements für das STF und dem eigentlichen HTML-Element in der GUI innerhalb des Browsers. Über die jeweilige Klasse können verschiedene Methoden zum Schreiben und Lesen des HTML-Elements aufgerufen werden. Die Vermittler-Klasse sorgt dann dafür, dass Javascript-Code im Browser ausgeführt wird, wodurch das eigentliche HTML-Element der HTML-GUI angesteuert wird.

In der Klasse für das jeweilige HTML-Elemente, wird beim Ansteuern immer der entsprechende Selector, über den das Element von Javascript aus erreichbar ist, gesetzt. Beim Button ist dies z.B. ein `IDSelector`, d.h. der HTML-Button muss eine ID, haben damit das STF ihn ansprechen kann. Neben dem `IDSelector` gibt es unter anderen auch einen `ClassSelector` oder einen `NameSelector` um HTML-Elemente über die CSS-Klasse bzw. den Namen zu erreichen.

Wie bei der RemoteHTMLView wird außerdem das Browser-Objekt gesetzt, in dem die anzusteuernde GUI gezeigt wird. Dies dient dazu, dass die Vermittler-Klasse den Browser kennt, in dem der Javascript-Code ausgeführt werden soll.

4.1.3 Verwendung von Singeltons im STF

Jede Klasse, die vom STF auf dem Server (Tester) aufgerufen wird, verwendet das Entwurfsmuster Singelton. Über ein Singelton wird sicher gestellt, dass es zur Laufzeit jeweils nur ein Objekt der Klasse gleichzeitig gibt und dieses global erreichbar ist[4].

Für das STF beutet dies konkret, dass der HTMLBot, die RemoteHTMLView sowie alle Klassen die ein HTML-Element vertreten, als Singelton implementiert sind. D.h. sollen aus einem STF-Test z.B. zwei verschiedene Buttons angesprochen werden, dann verwendet das STF intern immer das eine gleiche Objekt der RemoteHTMLButton Klasse. Um zwischen den Buttons zu unterscheiden, wird beim Ansteuern des Elements der jeweilige Selector in dem Objekt gesetzt. Der Javascript-Code wird dann nur auf dem Button mit der jeweiligen ID ausgeführt.

4.1.4 Beispiel Aufruf des HTMLBot

Um den Aufruf über das STF besser zu verstehen, möchte ich an dieser Stelle anhand eines Beispiels den internen Ablauf im Detail erklären. Der in Listing 1 aufgeführte Beispiel-Aufruf führt einen Klick auf einem Button in der HTML-GUI aus.

```
1  ALICE.htmlBot()
2     .view(MAIN_VIEW)
3     .button("add-contact")
4     .click();
```

Listing 1: Beispielhafter Aufruf zum Klicken eines Buttons in der HTML-GUI.

Für den Tester ALICE wird der HTMLBot angesprochen (Zeile 1). Dieser steuert über die view Methode die MAIN_VIEW an (Zeile 2), was der Startseite der HTML-GUI entspricht. Dann wird über button("add-contact") im RemoteHTMLButton Objekt ein IDSelector für die ID add-contact gesetzt (Zeile 3). Außerdem wird an dieser Stelle das Browser-Objekt, welches die entsprechende Seite anzeigt, referenziert. Über den Aufruf der click() Methode (Zeile 4)wird aus dem Selector und einer JS-Funktion der Javascript-Code $('#add-contact').click() zusammengesetzt. Dieser wird dann über die run Methode des Browsers, wie in Abschnitt 2.5 beschrieben, ausgeführt. Der Javascript-Aufruf führt dann dazu, dass der Button in der HTML-GUI tatsächlich angeklickt wird.

Analog zu diesem Aufruf funktionieren alle Methodenaufrufe des HTMLBot. Lediglich die Ansteuerung des HTML-Elements über den Selector und die darauf ausgeführten Methoden sind unterschiedlich.

4.2 HTML-Elemente

Analog zu dem in Abschnitt 4.1.2 beschriebenen `RemoteHTMLButton` habe ich weitere Klassen implementiert, um HTML-Elemente in der HTML-GUI anzusteuern. Alle unten aufgeführte Klassen erben von der RMI-Klasse `Remote` und werden vom Saros Test Framework exportiert, sodass ihre Methoden über die RMI-Schnitstelle, wie bereits in Abschnitt 2.1 erklärt, genutzt werden können. Jede Methode der Klassen kann eine `RemoteException` werfen. Diese wird genau dann geworfen, wenn ein allgemeines Problem bei der RMI-Schnittstelle auftritt oder wenn ein HTML-Element nicht existiert. Wird z.B. bei der Methode `button(Sting id)` eine ID übergeben, die es nicht in der aktuellen View gibt, dann wird die eben genannte `RemoteException` geworfen.

Folgende Klassen zur Ansteuerung von HTML-Elementen wurden in dieser Arbeit implementiert:

- `RemoteHTMLButton` Buttons können angeklickt werden und es lässt sich die Beschriftung zurückgeben.
- `RemoteHTMLTextElement` Der Textinhalt eines Elements kann verändert und gelesen werden.
- `RemoteHTMLInputField` Eingabefelder können befüllt, ausgelesen und zurückgesetzt werden.
- `RemoteHTMLCheckbox` Der Haken einer Checkbox lässt sich setzen und entfernen. Auch ob eine Checkbox ausgewählt ist, kann zurückgegeben werden.
- `RemoteHTMLRadioGroup` Radio-Buttons in einer Gruppe können selektiert und ausgelesen werden.
- `RemoteHTMLSelect` In einer Auswahl-Liste kann eine Option ausgewählt werden, sowie eine Liste aller möglichen Optionen, die Anzahl dieser und die ausgewählte Option ausgelesen werden.
- `RemoteHTMLMultiSelect` Hier stehen die selben Methoden wie bei RemoteHTMLSelect zur Verfügung mit der Besonderheit, dass hier mehrere Optionen ausgewählt und ausgelesen werden können.
- `RemoteHTMLProgressBar` Für einen Fortschrittsbalken kann der Wert gelesen und gesetzt werden.
- `RemoteHTMLTree` Ähnlich wie bei einer Checkbox können hier Knoten in einer Baumstruktur aus- und abgewählt werden.

Wie bereits in Abschnitt 4.1.1 erklärt, spielt die `RemoteHTMLView` eine übergeordnete Rolle der API. Jeder Aufruf um ein HTML-Element zu steuern, geht über die `RemoteHTMLView`. Für jede der oben aufgelisteten Klassen stellt die `RemoteHTMLView` Klasse eine Methode bereit, um den Zugriff auf das jeweilige Element zu ermöglichen. Es gibt neben der Methode `button(String id)`, wie im Beispiel 4.1.4 gezeigt, noch

weitere Zugriffsmethoden wie z.B. `checkbox(String name)` oder `textElement(String id)` um eine Checkbox bzw. ein Text-Element in der HTML-GUI anzusteuern.

4.3 BasicWidgetTestView

Um die in Abschnitt 4.2 aufgeführten Klassen zu testen, habe ich eine `BasicWidgetTestView` implementiert. Diese Ansicht beinhaltet alle HTML-Elemente die über das Test Framework angesprochen werden können und dient als „Spielwiese“ um den `HTMLBot` zu testen. Verwendet wird die Ansicht einzig und alleine von einem STF-Self-Test, siehe dazu Abschnitt 4.3.1. Die neue Test-View kann nicht aus der produktiven HTML-GUI heraus geöffnet werden, sondern lässt sich nur vom STF selbst über `.view(BASIC_WIDGET_TEST).open()` öffnen. Die Ansicht dient nur zu Testzwecken und ist kein Bestandteil der tatsächlichen HTML-GUI. Die Abbildung 3 zeigt die `BasicWidgetTestView` im Browser mit allen ansteuerbaren HTML-Elementen.

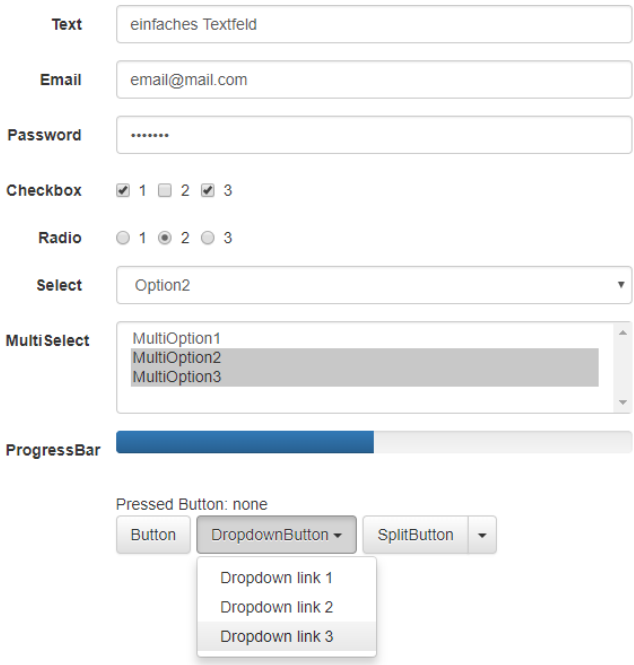

Abbildung 3: Zeigt die Ansicht der `BasicWidgetTestView` im Browser. Die Felder wurden für das Bildschrimfoto mit Beispieldaten befüllt.

Die in Abbildung 3 gezeigte `BasicWidgetTestView` enthält drei Text-Eingabe-Felder zum testen der `RemoteHTMLInputField` Klasse, drei Checkboxen sowie eine Radio-Button-Gruppe mit drei Radio-Buttons zum testen der `RemoteHTMLCheckbox` bzw. `RemoteHTMLRadioGroup` Klasse. Außerdem befinden sich zwei verschiedene Auswahl-Listen, eine davon mit dem Zusatz mehreren Optionen auswählen zu können, in der Ansicht, welche das Testen der `RemoteHTMLSelect` und `RemoteHTMLMultiSelect` Klassen ermöglichen. Über den Fortschrittsbalken kann die Klasse `RemoteHTMLProgressbar` getestet werden. Am Ende der Test-View befinden sich drei unterschiedliche Buttons, mit denen geprüft wird, dass die Klasse `RemoteHTMLButton` auch für unterschiedliche

Arten von Buttons genutzt werden kann. Oberhalb der Buttons befindet sich ein Text-Element, dass beim Anklicken der Buttons den Text ändert. Dadurch kann die Klasse `RemoteHTMLTextElement` getestet werden.

4.3.1 Test der BasicWidgetTestView

Alle Elemente der zuvor gezeigten `BasicWidgetTestView` werden über einen neu erstellten STF-Self-Test einzeln und in vollem Umfang getestet. Der Test überprüft, ob die HTML-Elemente vom STF aus angesteuert werden können und das darauf folgende Verhalten der HTML-GUI auch überprüft werden kann. D.h. der STF-Self-Test interagiert im ersten Schritt mit dem HTML-Element z.B. klickt es an, befüllt es mit Text oder wählt eine Option aus. Im zweiten Schritt wird dann lesend auf das Element zugegriffen, um zu sehen, dass die Interaktion erfolgreich funktioniert hat.

```
1  @Test
2  public void shouldTestInputFields() throws Exception {
3    IRemoteHTMLView view =
4      ALICE.htmlBot().view(BASIC_WIDGET_TEST);
5
6    assertTrue(view.hasElementWithName("text"));
7    view.inputField("text").enter("Some random text");
8    assertTrue(view.inputField("text").getValue()
9      .equals("Some random text"));
10   view.inputField("text").clear();
11   assertTrue(view.inputField("text").getValue()
12     .equals(""));
13 }
```

Listing 2: Beispielhafter STF-Self-Test um zu überprüfen, dass ein Eingabefeld vom STF angesteuert werden kann.

Das Listing 2 zeigt einen Ausschnitt aus dem `HtmlBasicWidgetTest`. Im Detail wird hier geprüft ob ein Element mit dem Namen „text“ existiert (Zeile 6). Im Anschluss wird der Text „Some random text“ in das Feld eingegeben (Zeile 7). Die Eingabe wird als nächstes durch einen lesenden Zugriff auf das Feld überprüft (Zeile 8 & 9). Zuletzt wird noch die `clear()` Methode aufgerufen (Zeile 10) und geprüft ob das Feld danach auch wirklich leer ist (Zeile 11 & 12).

Auf diese Weise werden alle neuen Klassen und deren Methoden des Test-Frameworks analog zu dem Beispiel aus Listing 2 über die speziell dafür entworfene `BasicWidgetTestView` getestet.

4.4 Erster HTML-STF-Test

Als ich alle grundlegenden HTML-Elemente über das Framework auf der `BasicWidgetTestView` ansteuern konnte, war es Zeit für einen ersten richtigen HTML-GUI STF-Test, der nicht die von mir ausgedachte „Spielwiese“ testet, sondern echte Funktio-

nalität von Saros. Bei einem genaueren Anblick der HTML-GUI, kam zu diesem Zeitpunkt der Arbeit nur das Hinzufügen eines neuen Kontaktes als sinnvolles Szenario für einen ersten HTML-STF-Test in Betracht, da die meisten Funktionen von Saros noch nicht in der HTML-GUI implementiert sind.

4.4.1 AddContactTest

Für diesen STF-Test werden die Tester Alice und Bob benötigt. In dem Test soll Alice Bob als neuen Kontakt hinzufügen. Als Vorbedingung muss deshalb gegeben sein, dass Alice und Bob sich noch nicht gegenseitig als Kontakt kennen.

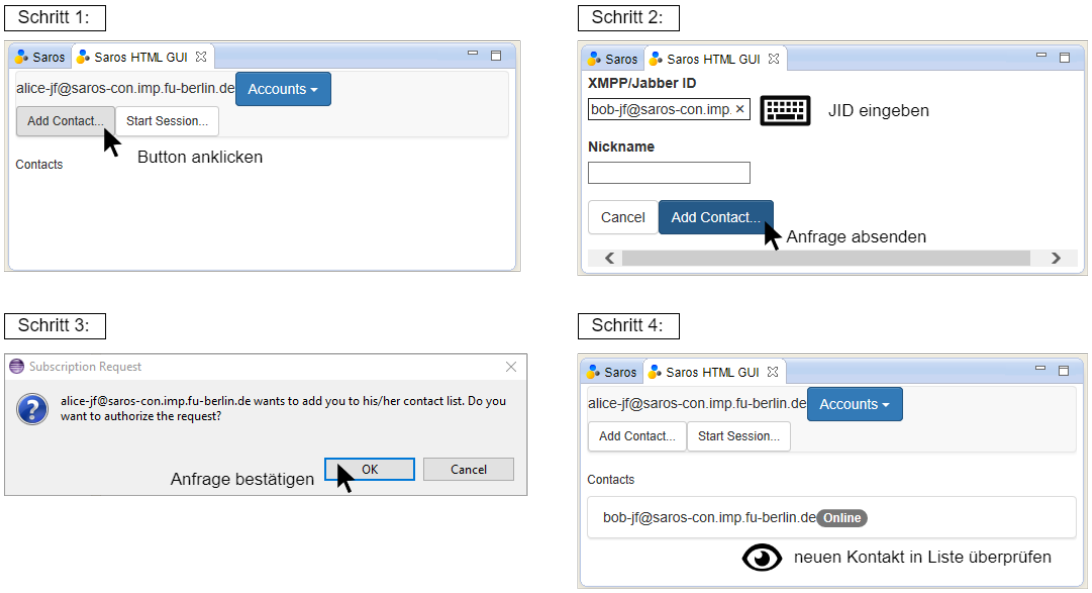

Abbildung 4: Hier ist der Ablauf des AddContactTest in 4 Schritten abgebildet. 1. Der „Add Contact“ Button wird auf der Main-View geklickt. 2. Die JID von Bob wird in das Feld eingegeben und die Anfrage wird abgesendet. 3. Die Anfrage wird von Bob bestätigt (Alice erhält selben Anfragedialog von Bob). 4. Alice hat nun Bob als Kontakt sichtbar in der Kontaktliste.

Der eigentliche Test über die HTML-GUI startet mit einem Klick auf den „Add Contact“ Button in der Main-View. Daraufhin öffnet sich eine neue Add-Contact-View, welche aus zwei Texteingabefeldern, eins für die JID und eins für einen optinalen Nickname, sowie einem Submit-Button besteht. Anschließend wird über das STF die JID von Bob in das vorgesehen HTML-Eingabefeld eingegeben und mit einem Klick auf den Submit-Button eine Kontakt-Anfrage abgesendet. Damit ein Kontakt tatsächlich hinzugefügt wird, muss zuerst Bob die Anfrage annehmen, worauf Alice ebenfalls eine Anfrage bekommt, die sie bestätigen muss. Die Anfrage erscheint noch über die alte SWT-GUI in einem Dialog, welcher aber wie gehabt über den bestehenden Teil des STF bestätigt werden kann. Im letzten Schritt des Tests wird überprüft, ob in der Kontaktliste von Alice, welche auf der Main-View zu sehen ist, Bob als Kontakt erscheint. Ebenso wird geprüft, ob Alice bei Bob als Kontakt angezeigt wird.

Abbildung 4 zeigt diese Schritte noch einmal mithilfe von Bildschirmfotos aus dem Eclipse-Plugin für Saros.

Es ist zu erkennen, dass dieser erste HTML-GUI STF-Test noch immer Teile der alten GUI und damit auch Teile des alten Test Frameworks verwendet. Da der Kontaktanfragendialog (Schritt 3 in Abbildung 4) noch nicht in der neuen HTML-GUI implementiert ist, läuft der STF-Test in diesem Fall in einer Art „Mischbetrieb“ - d.h. der STF-Test verwendet sowohl die neue als auch alte GUI. Dies ist möglich, da die neue HTML-GUI die alte noch nicht ersetzt, sondern parallel zur alten GUI läuft.

4.4.2 Problem beim Ansteuern der HTML-Elemente

Beim Entwickeln des Tests hatte ich große Probleme dabei, die JID so in das Feld zu schreiben, dass die Daten beim Anklicken des Submit-Buttons auch korrekt übertragen werden. Wie in Abschnitt 4.1 erklärt, wird bei dem Aufruf der `enter(String text)` Methode intern Javascript im Browser ausgeführt, welches den Text in das Textfeld schreibt. In einem ersten Versuch habe ich den Text mit der JQuery Funktion `val(text)` als Value für das Textfeld gesetzt. Beobachtet man den STF-Test bei der Ausführung, sieht man in der HTML-GUI wie der Text in das Feld für die JID geschrieben wird. Beim Versuch die Daten zu senden geschieht dann nichts weiter. Die View wechselt zurück zur Main-View, aber es wird keine Kontaktanfrage gesendet. Ein Blick in die Log-Datei verrät: `Invalid input: ''. Not a valid JID.` - die Eingabe der JID scheint leer zu sein, obwohl Text im Feld zu sehen ist.

Nach weiterer Recherche habe ich herausgefunden, dass die Implementierung von Eingabefeldern mit React in der HTML-GUI den Fehler verursacht. In der HTML-GUI ist ein onChange-Event-Handler an den Textfeldern gesetzt, welcher dafür sorgt, dass bei jeder Änderung in dem Feld eine globale Variable der React-Komponente mit dem Inhalt des Felds beschrieben wird. Beim Senden der Daten wird dann nicht die Value des Textfelds, sondern der Wert dieser globalen Variable gesendet. Scheinbar löst das Setzen der Value über JQuery kein onChange-Event auf dem HTML-Eingabefeld aus, sodass die globale Variable für dieses Feld nicht befüllt wird. Auch ein manuelles auslösen eines onChange-Events über das JQuery-Framework führte nicht zum gewünschten Ergebnis.

Um das Problem zu lösen, greift das Test Framework nun direkt auf die globale Variable der View zu und befüllt diese mit der Value. Daraus ergibt sich eine Vorgabe nachdem ein HTML-Eingabeelement in React implementiert werden muss, damit das STF die Möglichkeit hat, auf dieses zuzugreifen. Wie sich beim weiteren Testen herausstellt, tritt das Problem nicht nur bei Textfeldern, sondern auch bei anderen Elementen wie Checkboxen, RadioButtons oder Selects auf. Die `BasicWidgetTestView` dient hierzu auch als eine Art Katalog, um zu sehen, wie ein Eingabefeld für die HTML-GUI in React implementiert werden sollte, damit das STF korrekt zugreifen kann.

4.5 Regelwerk für HTML-Elemente

Das in Abschnitt 4.4.2 beschriebene Problem, ist ein generelles Problem was sich für alle HTML-Elemente mit Ausnahme von Buttons für die HTML-GUI ergibt. Die in der HTML-GUI bereits implementierten Features verwenden bei der Weiterverarbeitung von Nutzereingaben nicht die Werte direkt aus dem HTML-Element, sondern lesen diese aus einer globalen Variable. Diese globale Variable wird zuvor über einen onChange-Event-Handler bei einer Interaktion des Nutzers befüllt. Dies kann z.B. ein Mausklick oder eine Eingabe über die Tastatur in ein Feld sein.

```
1  @observable fields = {user: '', password: ''}
2
3  @action onChangeField = (e) => {
4    this.fields[field] = value
5  }
6
7  @action onClickSubmit = () => {
8    this.props.mainUI
9      .doSubmitForm(this.fields.user, this.fields.password)
10 }
```

Listing 3: Bei der Verarbeitung in der onClickSubmit Funktion werden die Variablen user und password verwendet und nicht die Werte direkt aus den Feldern.

In Listing 3 ist zu sehen, dass es zwei Variablen `user` und `password` gibt, welche unter einer globalen Variable `fields` zusammengefasst werden (Zeile 1). Diese Variablen werden über die Funktionen `onChangeField` befüllt (Zeile 4). In der `onClickSubmit` Funktion werden die Variablen gelesen und deren Wert zur weiteren Verarbeitung an eine andere Funktion weitergegeben (Zeile 8 & 9).

```
1 <input type='text' value={this.fields.user}
2 onChange={this.onChangeField} name='user' />
3 <input type='text' value={this.fields.password}
4 onChange={this.onChangeField} name='password' />
5 <button id='submit-button' onClick={this.onClickSubmit}>
6 Submit
7 </button>
```

Listing 4: Die onChangeField Funktion wird als onChange-Event-Handler dem HTML-Element hinzugefügt. Bei einem Event wird die Value aus dem Feld in der globalen Variable gespeichert.

Listing 4 zeigt, dass bei einer Texteingabe in die HTML-Eingabefelder, über den onChange-Event-Handler die Funktion `onChangeField` aufgerufen wird (Zeile 2 & 4). Beim Klick auf den Button in wird dann die Funktion `onClickSubmit` aufgerufen (Zeile 5). Es ist zu erkennen, dass wenn das STF nur die Value an dem HTML-Element ändert, die `onChangeField` Methode nie aufgerufen wird. Die Variablen bleiben leer und werden dementsprechend leer weitergegeben.

Zur Lösung dieses Problems greift das STF direkt über Javascript auf die einzelnen Variablen unter der globale `fields` Variable zu. D.h. beim Beschreiben der Variable wird diese als Text auf der Oberfläche angezeigt und die Weiterverarbeitung über die verwendeten React-Funktionen wird sichergestellt. Damit das STF funktioniert, muss dem HTML-Input-Element über das `name`-Attribut der gleiche Name wie für die Variable vergeben werden. Das STF ließt und schreibt die Variablen der HTML-GUI über Getter und Setter das bedeutet, dass die in Listing 5 aufgeführten Methoden `setFieldValue` und `getFieldValue`, sowie die `fields` Variable Bestandteil jeder Ansicht der HTML-GUI sein müssen, wenn das STF auf Eingabe-Elemente zugreifen soll.

```
1 @observable fields = {var1: '', var2: '', var3: ''}
2
3 @action setFieldValue = (name, value) => {
4   this.fields[name] = value
5 }
6
7 @action getFieldValue = (name) => {
8   return this.fields[name]
9 }
```

Listing 5: Der Setter setFieldValue und der Getter getFieldValue sind in jeder View pflicht wenn das STF auf HTML-Eingabeelemente zugreifen soll

4.6 Dialoge

Neben der Hauptansicht von Saros befinden sich weitere GUI-Elemente innerhalb von Dialogen. Dialoge sind separate Fenster, welche über dem Hauptfenster der Entwicklungsumgebung geöffnet werden. Auch die HTML-GUI, bzw. bestimmte Teile davon, werden in einem Dialog angezeigt.

In der Arbeit von Marius Schidlak [6] wurde die HTML-GUI um eine `StartSession WizardView` erweitert, welche den Vorgang zum Starten einer neuen Session ermöglichen soll. Der Dialog an sich ist ein klassisches GUI-Element von SWT bzw. Swing der jeweiligen Entwicklungsumgebung indem ein weiterer eingebetteter Browser die HTML-View anzeigt. Natürlich sollen über das STF auch die HTML-Elemente innerhalb eines Dialoges angesteuert werden können. Beim Erstellen des Dialogs von Saros speichert der `BrowserManager` die View, die gerade im Browser des Dialogs geöffnet wurde. Vom STF-Test aus können HTML-Elemente in einem Dialog genauso angesteuert werden wie Elemente in der Main-View von Saros. Der `BrowserManager` hilft dem Framework dabei zu entscheiden, in welchem Browser (Browser des Dialogs oder der Main-Saros-View) der Javascript-Code ausgeführt werden soll.

4.6.1 StartSessionWizardTest

Der `StartSessionWizardTest` ist wie der `AddContactTest` ein weiterer STF-Test zum Testen eines Features von Saros über die HTML-GUI. Der StartSessionWizard besteht

aus 2 Schritten. Im ersten Schritt soll der Nutzer über eine Baum-Ansicht auswählen, welche Dateien in der neuen Session geteilt werden sollen. Im zweiten Schritt sind die Kontakte auszuwählen, welche in die neue Session eingeladen werden sollen. Um zwischen den Schritten zu navigieren gibt es einen „Back“, „Next“ sowie einen „Finish“ Button am Ende des Wizards.

In einer ersten Version des `StartSessionWizardTest` wollte ich erst einmal nur den Dialog öffnen, den "Next"Button anklicken und überprüfen, dass der Titel richtig angezeigt wird. Dies hat auf Anhieb ohne Probleme funktioniert. Wie erwartet konnten die HTML-Elemente innerhalb des Dialogs über das STF-angesprochen werden.

4.6.2 Probleme des StartSessionWizardTest

In der nächsten Version des `StartSessionWizardTest` wollte ich über das STF den Datei-Baum ansteuern, um dort Dateien und Ordner auszuwählen. Der Datei-Baum besteht aus Labels mit den Datei- bzw. Ordnernamen und Checkboxen um diese auszuwählen. Um Order ein- bzw. auszuklappen zu können gibt es daneben kleine Buttons mit Plus und Minus. Das ganze Datei-Baum-HTML-Element wird über die „rc-tree“ Bibliothek[5] für React erzeugt und in die HTML-GUI eingebunden.

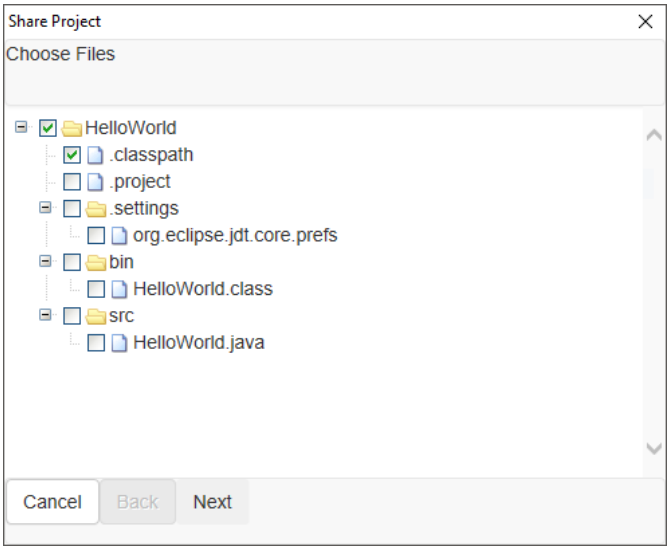

Abbildung 5: Das Bildschrimfoto zeigt den Dialog des StartSessionWizards mit dem Datei-Baum. In diesem Beispiel wurde der oberste Ordner „HelloWorld“ selektiert, aber nur eine der beiden unterliegenden Dateien wurde mit ausgewählt. Außerdem lässt sich der Haken an dem Ordner nicht mehr entfernen.

Bei genauerer Betrachtung des erzeugten Datei-Baumes ist mir aufgefallen, dass die Checkboxen im Baum keine normalen Checkboxen sind, wie ich sie z.B. in der `Basic WidgetTestView` verwendet habe, sondern sie normale `<span>` HTML-Elemente sind. Beim Anklicken dieser „Fake-Checkbox“ wird von der React-Bibliothek nur das Hintergrundbild ausgetauscht, sodass es aussieht wie eine Checkbox in der ein Haken

[5] https://www.npmjs.com/package/rc-tree

gesetzt wurde. Durch diese Entdeckung wird der Einsatz der `RemoteHTMLCheckbox` im `StartSessionWizardTest` hinfällig. Abhilfe habe ich durch die Implementierung der `RemoteHTMLTree` Klasse geschaffen, mit welcher sich über den Selector `span[title='filename']` einen Knoten im Datei-Baum ansteuern lässt.

Allerdings verhält sich der Datei-Baum, welcher in Abbildung 5 zu sehen ist, unabhängig vom STF sehr merkwürdig. Zum Beispiel werden beim Auswählen eines Ordners nicht alle Dateien unter ihm automatisch mit ausgewählt. Oft kommt es auch vor, dass sich Dateien oder Ordner gar nicht auswählen lassen oder daraufhin nicht mehr abwählen lassen. Dieses Verhalten kann ich nur auf einen Fehler innerhalb der Bibliothek oder einer falschen Nutzung dieser aus der Arbeit von Marius Schildlack[6] zurückführen. Ein Beheben des Problems war nicht Teil meiner Arbeit, deshalb wurde es von mir auch nicht weiter untersucht. Das beschriebene Problem macht allerdings ein sinnvolles Testen des Dateibaums über das STF bzw. über die `RemoteHTMLTree` Klasse hinfällig.

Im nächsten Schritt des `StartSessionWizard` sollen die Kontakte ausgewählt werden, die eine Einladung zur Session erhalten sollen. Der aktuelle Stand der GUI lässt ein auswählen des Kontaktes in der Liste jedoch gar nicht zu. Zu diesem Zeitpunkt der Arbeit war es mir nicht möglich, den Kontakt in der HTML-GUI in Eclipse auszuwählen, sodass dieser blau markiert wird. Dadurch ist es nicht möglich, den Wizard abzuschließen, um eine Session zu starten.

4.7 Weitere Probleme

Bei der Entwicklung des Saros Test Framework für die HTML-GUI hatte ich neben den in Abschnitt 4.4.2 und 4.6.2 geschilderten Problemen noch weitere kleine Hürden zu meistern, die ich hier kurz zusammenfassen möchte.

Da die HTML-GUI im Grunde eine klassische HTML-Seite ist, kann sie auch in einem beliebigen Browser außerhalb der Entwicklungsumgebung geöffnet werden. Ich habe z.B. Google Chrome[6] benutzt, um die HTML-GUI zu debuggen. Chrome bietet unter den Entwicklerwerkzeugen eine Konsole, an um Javascript-Code manuell auszuführen. Dies gibt mir die Möglichkeit vorab zu testen, wie ich z.B. auf einen Button zugreifen kann, um dann mit ihm zu interagieren. Allerdings verhält sich die Konsole von Google Chrome nicht genauso wie wenn ich Javascript auf dem eingebetteten Browser über das STF ausführe.

Um einen Button zu klicken reicht es, unter Chrome den Javascript-Code `$('button-id').click()` auszuführen. Wird derselbe Code über das STF auf der HTML-GUI in Eclipse ausgeführt, führt dies nicht zu einem Ankicken des Buttons. Nach einiger Recherche im Internet konnte ich herausfinden, dass es notwendig ist, das HTML-Element eindeutig zu selektieren. Mit `$('button-id')[0].click()` wird auch ein Klick auf dem Button in Eclipse ausgeführt.

[6] https://developers.google.com/web/tools/chrome-devtools/

Die Grund ist, dass `$('button-id')` zunächst eine Liste alle Elemente mit der ID zurückliefert, aus der dann das erste Element genommen werden muss um darauf dann einen Klick auszuführen. Die Entwickler-Konsole von Google Chrome ist scheinbar intelligent genug, dass diese direkt ein Klick auf dem Element ausführt, wenn über den Selector eine Liste mit nur einem Element zurück gegeben wird.

Ein weiteres Problem stellte die Nutzung des JQuery-Frameworks in der Chrome Konsole dar. Das Framework wird beim eingebetteten Browser in Eclipse gleich mit geladen damit sichergestellt ist, dass das STF die Funktionen von JQuery auch nutzen kann. Wenn die HTML-GUI in Chrome geöffnet wird, steht einem beim Ausführen von Javascript-Code über die Konsole kein JQuery mehr zur Verfügung. Dieser Umstand hat mir das Entwickeln des STF zunächst sehr erschwert, da ich die richtigen Selectoren für die HTML-Elemente nicht vorab in der Chrome Konsole testen konnte. Ich habe mich dann dazu entschlossen, einfach manuell das JQuery-Framework über `import $ from 'jquery'` einzubinden. Der Vorteil ist, dass JQuery jetzt auch in der Entwickler-Konsole genutzt werden kann. Der Nachteil allerdings ist, dass das Framework für den produktiven Einsatz unnötig doppelt eingebunden wird, einmal über den eingebetteten Browser und zusätzlich manuell. Der Vorteil, um dadurch bequemer an dem STF Entwickeln zu können, überwiegt aber den Nachteil.

5 Modularisierung

Saros besteht aus mehreren Komponenten bzw. Teilprojekten, die in unterschiedlichen Packages liegen. Durch die Aufteilung des Codes in mehrere Komponenten kann eine bessere Struktur des Codes gewährleistet werden. Logisch zusammenhängende Klassen können in einer Komponente zusammengefasst und von anderen Codebestandteilen getrennt bzw. entkoppelt werden. Damit die Komponenten zusammenarbeiten können, bestehen entsprechende Abhängigkeiten zwischen ihnen. Der Code für das STF sowie die STF-Tests waren bisher nicht durch eine eigene Komponente vom Plugin-Code bzw. GUI-Code getrennt. Eine neue Komponente für den STF-Code sowie eine weitere eigenständige Komponente für die STF-Tests, beschreiben das zweite Ziel meiner Arbeit.

Zu Beginn der Arbeit gab es folgende relevante Komponenten:

de.fu_berlin.inf.dpp Beinhaltet den Entwicklungsumgebung spezifischen Teil von Saros für Eclipse. Diese Eclipse-Komponente enthält zusätzlich alle Klassen für die alte Eclipse-GUI.

de.fu_berlin.inf.dpp.intellij Analog zur Eclipse-Komponente ist hier der Entwicklungsumgebung spezifischen Teil von Saros für IntelliJ zu finden. Hier sind ebenfalls die Klassen der alten IntelliJ-GUI untergebracht.

de.fu_berlin.inf.dpp.core Die Core-Komponente beinhaltet den Teil von Saros welcher unabhängig von einer Entwicklungsumgebung ist. Die Klassen aus dieser Komponente werden von der Eclipse- und IntelliJ-Komponente genutzt.

de.fu_berlin.inf.dpp.ui Diese Komponente bildet den Java Teil der HTML-GUI ab und dient als Schnittstelle zwischen den IDE-Plugins und der HTML-GUI.

de.fu_berlin.inf.dpp.ui.frontend Der React Teil der HTML-GUI ist in dieser separaten Komponente gelagert. Hier sind keine Java Klassen, sondern nur JSX-Datein für React zu finden.

In den folgenden Kapiteln 5.1 und 5.2 werde ich darauf eingehen, wie das Saros Test Framework in den bisherigen Komponenten verteilt war und welche Änderungen ich vornehmen musste, um die Projekt-Struktur in den neuen Ziel-Zustand zu bringen.

5.1 Alter Zustand der Komponenten

Wie bereits in Kapitel 2 den Grundlagen erwähnt, besteht das STF aus drei wesentlichen Komponenten - dem Framework an sich, den STF-Tests und den STF-Self-Tests. Diese drei Teile waren zu Beginn meiner Arbeit in den verschiedenen Komponenten verstreut.

Der Framework-Teil für die Eclipse-GUI lagen in der Eclipse-Komponente, direkt daneben lagen die STF-Self-Tests. Die eigentlichen STF-Tests hingegen lagen ebenfalls in der Eclipse-Komponente. Obwohl sie in einem anderen Ordner lagen, sorgte dies anfänglich für Verwirrung, da STF-Tests und STF-Self-Tests sehr ähnlich aussehen.

Der Framework-Teil für die HTML-GUI war wiederum in der UI-Komponente untergebracht. Die zugehörigen STF-Tests, welche das HTML-STF, nutzen lagen dann allerdings in der Eclipse-Komponente.

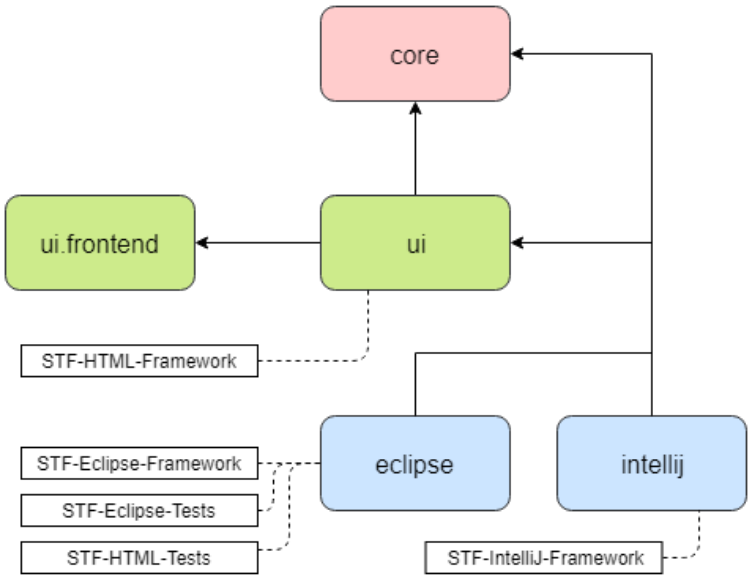

Abbildung 6: Die Teile des STF sind in den verschiedenen Komponenten verstreut. Die Abhängigkeiten der Komponenten untereinander sind durch die Pfeile dargestellt.

Die Abbildung 6 zeigt die zerstreute Verteilung der STF-Teile in den Komponenten noch einmal deutlich. Abhängigkeiten zwischen den Komponenten lassen sich durch die Pfeile erkennen. So ist z.B. die Core-Komponente von der UI-Komponente abhängig. Ein großer Kasten mit abgerundeten Ecken stellt jeweils eine Komponente dar. Von oben nach unten gelesen sind das die „core“-Komponente in Rot, die GUI-Komponenten „ui“ und „ui.frontend“ in Grün sowie die Plugin-Komponenten „eclipse“ und „intellij“ in Blau. An den jeweiligen Komponenten sind durch gestrichelte Linien Bestandteile des Saros Test Frameworks in weißen Kästen angehängt. Die gestrichelte Linie bedeutet, dass dieser Bestandteil unter der jeweiligen Komponente abgelegt ist.

Da diese Form der Strukturierung des Codes für Saros und besonders in Bezug auf den Code für das STF, die STF-Self-Tests und die STF-Tests auch bei zukünftigen Entwicklern im Projekt für Verwirrung sorgen kann, habe ich mich dazu entschieden, für alle Codebestandteile des STF eine neue Struktur zu schaffen, welche alle Teile des Saros Test Frameworks vereint.

5.2 Neuer Zustand der Komponenten

Meine erste Überlegung war es alle drei STF-Teile, also STF-Code, STF-Self-Tests und STF-Tests, aus den bisherigen Komponenten heraus zu nehmen und in eine neue STF-Komponente zu verschieben. Dadurch kann allerdings immer noch schwer zwischen den STF-Self-Tests und den eigentlichen STF-Tests unterschieden werden. Es stellte

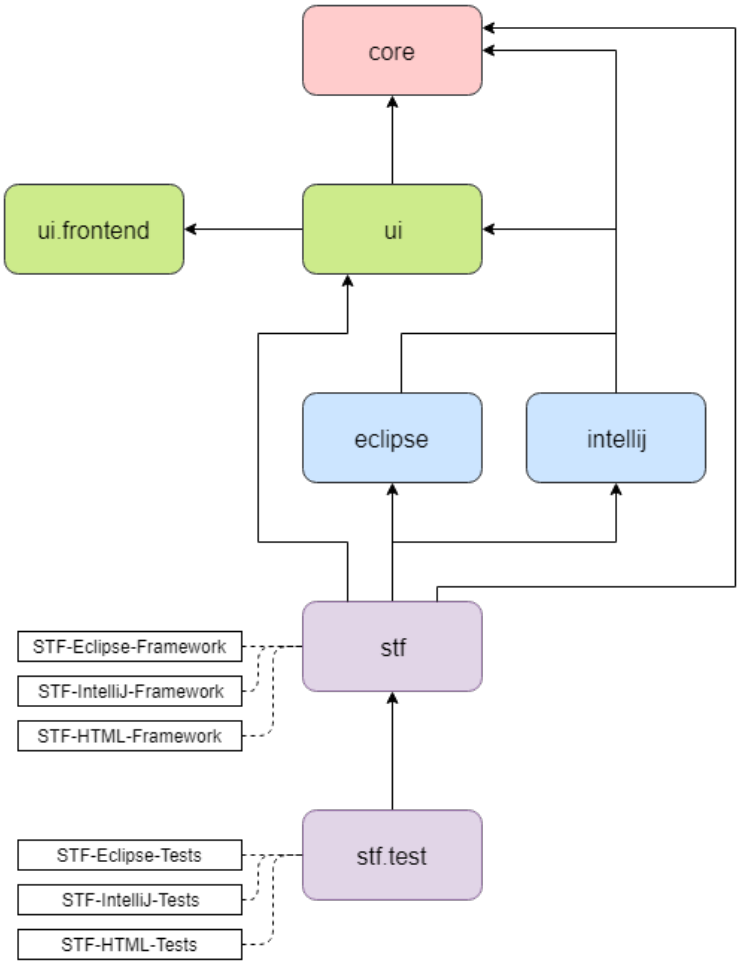

Abbildung 7: Durch die neue Struktur liegen alle Framework-Teile und STF-Self-Tests in der STF-Komponente. Die STF-Tests sind in einer separaten STF-Test-Komponente, welches auf die STF-Komponente zugreift, untergebracht. Damit das Framework funktioniert braucht die STF-Komponente Zugriff auf die Eclipse-, IntelliJ-, UI- und Core-Komponente.

sich raus, dass eine Einteilung des STF in zwei Komponenten doch sinnvoller ist. Die erste Komponente beinhalten den Framework-Teil und die Self-STF-Tests. Die zweite Komponente enthält dann die STF-Tests und verwendet die erste Komponente, also das eigentliche Test-Framework, um die Tests auszuführen.

Dafür mussten die zwei neue Komponenten, stf und stf.test, angelegt werden, um im Anschluss die STF-Teile aus der Eclipse-Komponente bzw. der UI-Komponente in die neuen Komponenten zu verschieben. Intern habe ich die neue STF-Komponente so aufgeteilt, dass es für die HTML-GUI, IntelliJ und Eclipse einen Unterordner gibt. Ich habe mich für diese Struktur entschieden, um beim Verschieben der vielen Klassen einen Überblick zu behalten, welcher Teil bzw. welches Package woher kam. So habe ich z.B. alle STF-HTML-Framework Klassen die zur Steuerung der HTML-GUI da waren, in dem Unterordner „src/html“ verschoben. Alle STF-Eclipse-Framework

Klassen die vorher in der Eclipse-Komponente unter gebracht waren, wurden nun in den Eclipse-Ordner verschoben sowie die STF-IntelliJ-Framework Klassen in einen IntelliJ-Ordner. Neben diesen Source-Ordnern liegen auch die STF-Self-Tests in drei entsprechenden Test-Unterordnern in der neuen STF-Komponente.

Da das Framework auf IDE spezifische und unspezifische Komponenten sowie Teile der HTML-GUI zugreift, ist die STF-Komponente abhängig von den Komponenten eclipse, intellij, core und ui. Der Zugriff auf diese vier Komponenten ist notwendig und auch legitim solange keine dieser vier Komponenten die STF-Komponente selbst benötigt, wodurch eine Abhängigkeit zur STF-Komponente entstehen würde. Denn dann existiert eine zyklische Abhängigkeit der Komponenten im Build-Prozess, was zur Folge hätte, dass das gesamte Projekt nicht kompiliert.

Die zweite STF-Test-Komponente beinhaltet nur die STF-Tests. Dazu hat diese zweite Komponente die gleiche Ordner-Struktur wie die erste, in der die STF-Tests nach IDE bzw. HTML-GUI getrennt werden. Diese zweite Komponente hat keinen Source-Ordner, da sie wie bereits beschrieben nur STF-Tests enthält. Die Abhängigkeit der STF-Test-Komponente besteht nur zur STF-Komponente, dadurch ist sichergestellt, dass die Tests nur Klassen und Methoden aus dem Test Framework benutzen und nicht direkt auf die GUI-Elemente der Entwicklungsumgebung oder die HTML-GUI zugreifen können.

In Abbildung 7 ist zu sehen, welche Abhängigkeiten von den beiden neuen Komponenten zu den bestehenden Komponenten entstehen und wie der Code des STF in diese beiden Komponenten verschoben bzw. aufgeteilt wurde. Im Vergleich zu Abbildung 6 sind die beiden neuen STF-Komponenten in Violett sowie deren neuen Abhängigkeiten hinzu gekommen.

5.3 Herausforderungen der zyklischen Abhängigkeit

Die größte Herausforderung für mich war es, die neuen Abhängigkeiten der einzelnen Komponenten umzusetzen. Es ist zwar möglich, die benötigten Komponenten zum Bauen der neuen STF-Komponente in einer Manifest-Datei unter „Require-Bundle“ explizit anzugeben, allerdings kann dadurch nicht das Problem der zyklischen Abhängigkeit verhindert werden. Als ich alle Teile des Saros Test Framework in die neue STF-Komponente verschoben hatte, bestand am Ende des Vorgangs in einer einzigen Zeile ein Problem bei der Kompilierung des Projekts. Diese eine Zeile, welche in Listing 6 aufgeführt wird, ist entscheidend für den Start des STF für Eclipse.

```
1 STFController.start(port, context);
```

Listing 6: Diese Zeile verhinderte zunächst das kompilieren des Projekts. Der Aufruf startet das STF über den STFController.

Im nächsten Abschnitt 5.3.1 möchte ich weiter auf dieses Problem eingeben und die Ursache beschreiben, damit ich dann in Abschnitt 5.3.2 meine Lösung präsentieren kann.

5.3.1 Problem beim Starten des STF

Um das Problem zu verstehen muss man wissen, dass das STF wie bereits in Abschnitt 2.2 beschrieben, beim Starten der Entwicklungsumgebung einen Server startet. Dieser Server exportiert dann die notwendigen RMI-Objekte damit der STF-Test, welcher auf der Client-Seite sitzt, mit der Entwicklungsumgebung auf der Server-Seite kommunizieren kann. Der `STFController` besitzt eine statische `start` Methode, mit welcher das komplette STF-initialisiert wird und der Server hochgefahren wird. Beim Starten einer Eclipse-Instanz und nachdem das Saros Plugin geladen bzw. initialisiert ist, wird die Methode `StartupSaros.earlyStartp()` ausgeführt, welche das STF über den Aufruf `STFController.start()` startet. Die `StartupSaros` Klasse ist ein Bestandteil des Eclipse Plugins und liegt in der entsprechenden Eclipse-Komponente. Der `STFController` liegt jedoch in der neuen STF-Komponente, da er Zugriff auf das Framework braucht, um die RMI-Objekte zu exportieren. Durch die Unzulässigkeit von zyklischen Abhängigkeiten ist es folglich nicht möglich von der Eclipse-Komponente aus das STF zu starten.

Die Abbildung 8 zeigt den nicht zulässigen Aufruf, welcher eine zyklische Abhängigkeit verursacht. Die Methode `start` wird von der Eclipse-Komponente auf einer Klasse aus der STF-Komponente aufgerufen.

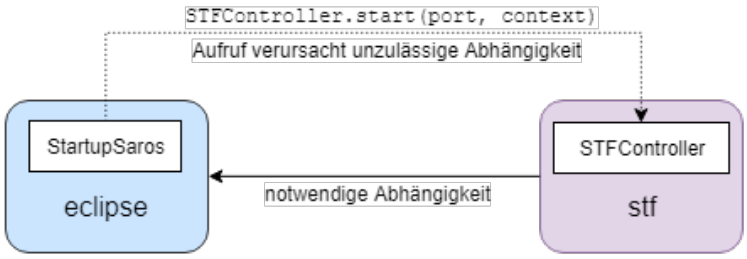

Abbildung 8: Der Aufruf zum Starten des STF verursacht eine unzulässige zyklische Abhängigkeit zwischen der STF-Komponente und der Eclipse-Komponente.

Vor der Verschiebung des STF in eine eigenständige Komponente konnte von der Eclipse-Komponente auf die entsprechenden STF-Start-Methode problemlos zugegriffen werden, da sie in der selben Komponente lag. Nun ist dieser Zugriff nicht mehr möglich, wodurch sich das STF nicht wie bisher starten lässt. Um diese unzulässige Abhängigkeit zu vermeiden, ist es notwendig, das STF von der eigenen STF-Komponente aus zu starten.

5.3.2 Neues Plugin als Lösung

Meine Lösung umgeht das Problem dadurch, dass ich aus der STF-Komponente ein eigenständiges Plugin erstellt habe. Beim Starten von Eclipse wird neben dem Saros-Plugin zusätzlich ein neues STF-Plugin mit geladen. Ich habe den Start des STF aus dem Eclipse-Plugin ausgebaut und in eine neue `StartupSarosSTF` Klasse verlagert. Die `earlysStartup()` Methode in der neuen StartUp-Klasse wird erst beim Laden

des neuen STF-Plugins aufgerufen, welche dann das STF startet. Durch diese Lösung liegt der Aufruf zum Starten des STF nicht mehr in der Eclipse-Komponente, sondern in der STF-Komponente. Der Aufruf des `STFControllers` aus der STF-Komponente heraus ist zulässig bezüglich der Abhängigkeiten, wodurch sich das gesamte Projekt nun kompilieren lässt.

Die Abbildung 9 zeigt den neuen Aufruf zum Starten des STF. Der Aufruf verlässt die STF-Komponente nicht mehr und verursacht dadurch keine Abhängigkeit zur Eclipse-Komponente.

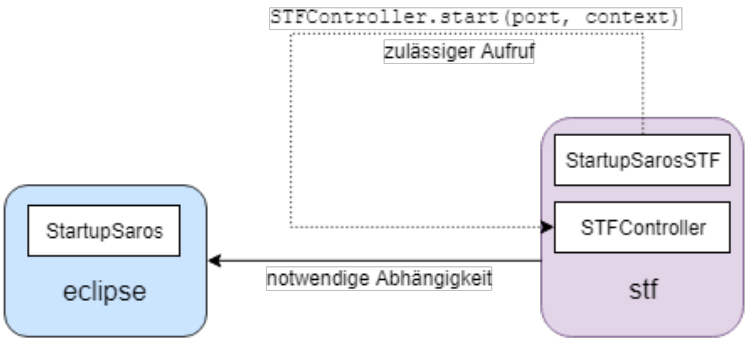

Abbildung 9: Der Aufruf zum Starten des STF aus Sicht der STF-Komponente ist zulässig.

Theoretisch besteht das selbe Problem auch für das IntelliJ-Plugin. Allerdings wurde zu diesem Zeitpunkt meiner Arbeit das STF noch nicht für IntelliJ eingebunden. Somit gab es erstmal kein Problem, was sich aber wie in Abschnitt 7.2 zu lesen ist, noch geändert hat.

5.4 Probleme bei der Verschiebung des STF-IntelliJ-Framework

In diesem Abschnitt möchte ich auf die Probleme eingehen, welche nach der Verschiebung der STF-Code-Teile für IntelliJ folgten.

Für IntelliJ gab es bereits die `IntelliJHTMLWorkbenchBot` Klasse, die Teil des STF ist und lediglich die HTML-GUI von Saros in IntelliJ öffnen bzw. schließen soll. Erst beim Verschieben dieser Klasse fiel mir auf, dass die neue STF-Komponente nun auch von der IntelliJ-Komponente abhängig war, damit der `IntelliJHTMLWorkbenchBot` auf die IntelliJ-GUI zugreifen kann, um dort z.B. ein Menü anzuklicken. Dazu war es notwendig, die IntelliJ-Komponente mit Eclipse zu kompilieren, was eigentlich kein Problem darstellen sollte, da auf der Saros Website[7] bereits dokumentiert ist wie dies funktioniert.

Im Gegensatz zu den anderen Komponenten muss das IntelliJ-Plugin für Saros mit Java 1.7 und nicht mit 1.6 kompiliert werden. Allerdings ließ dies mein Setup von Eclipse 3.7 Indigo nicht zu, da erst ab Version Eclipse 4.2 Juno, Java 1.7 unterstützt

[7] https://www.saros-project.org/saros-for-intellij/dev-environment

wird. Durch ein Update von Eclipse auf die neuere Version konnte ich das IntelliJ-Plugin unter Eclipse kompilieren, jedoch hatte das Update auch zur Folge, dass der SWTBot nicht mehr funktioniert hat, da der bisher verwendete SWTBot nur mit Eclipse bis Version 3.7 kompatibel ist. Beim Ausführen eines alten STF-Tests, welcher noch die Eclipse-GUI verwendet, wurden plötzlich Elemente nicht mehr gefunden und der SWTBot konnte nicht mehr auf diese zugreifen. Für mich bedeutete das, dass der SWTBot auch ein Update bekommen muss, wenn ich das STF auch in IntelliJ einbinden möchte.

Nach dem Update des SWTBot von 2.0.5 auf die aktuellste Version 2.7.0 kompilierte das Projekt leider nicht mehr. Das Problem lag dieses Mal nicht an der IntelliJ-Komponente, sondern an der geänderten API des SWTBot. Einige Methoden des SWTBot-Frameworks verlangten nun statt zwei Parametern eine Array als Parameter. Nachdem ich die entsprechenden Stellen im Code auf die neue API korrigiert hatte, kompilierten alle Komponenten wieder und die STF-Tests konnten wie gewohnt mit Eclipse Juno ausgeführt werden.

6 Bestehende STF-Tests

In diesem Kapitel beschreibe ich die durchgeführten Schritte, die zur Umsetzung des dritten Ziels notwendig waren. Da bereits viele Funktionen von Saros mit Hilfe von bestehende STF-Test über die Eclipse-GUI getestet werden, ist es sinnvoll, diese Tests für die HTML-GUI wieder zu verwenden. In Abschnitt 6.1 zeige ich anhand von Code-Beispielen, welche Möglichkeiten der `HTMLBot` nach den Änderungen aus Kapitel 4 liefert ohne weitere Anpassung vorzunehmen. Der darauf folgenden Abschnitt 6.2 beschreibt, was ich implementiert habe, damit ein STF-Test die HTML-GUI verwendet, ohne das er dazu umgeschrieben werden muss. Am Ende des Kapitels gebe ich in Abschnitt 6.3 noch einen Überblick darüber, warum der Zustand der HTML-GUI daran Schuld ist, dass nach den Umbauten im STF nicht alle STF-Tests mit der neuen HTML-GUI funktionieren.

6.1 Möglichkeiten des HTMLBots

Mit den in Kapitel 4 neu eingeführten Klassen und Methoden des Saros Test Framework ist es möglich, Funktionalitäten von Saros, wie z.B. das Hinzufügen eines neuen Kontakts, über die HTMl-Oberfläche zu testen. Viele Funktionen von Saros werden bereits in einem STF-Test auf ihre Korrektheit getestet, allerdings wird hierzu ausschließlich die SWT-GUI von Saros für Eclipse verwendet. Es gibt bereits eine Test-Klasse `ContactsByAliceBobTest`, welche in dem Test `addContact()` überprüft, ob der Tester Alice den anderen Tester Bob als Kontakt hinzufügen kann.

Das Listing 7 zeigt den vollständigen Code des `addContact()` Test.

```
1  @Test
2  public void addContact() throws Exception {
3  Util.removeTestersFromContactList(ALICE, BOB);
4
5  assertFalse(ALICE.superBot().views().sarosView()
6    .isInContactList(BOB.getJID()));
7  assertFalse(BOB.superBot().views().sarosView()
8    .isInContactList(ALICE.getJID()));
9
10 ALICE.superBot().views().sarosView().addContact(BOB.getJID());
11
12 BOB.superBot().confirmShellRequestOfSubscriptionReceived();
13 ALICE.superBot().confirmShellRequestOfSubscriptionReceived();
14
15 assertTrue(ALICE.superBot().views().sarosView()
16   .isInContactList(BOB.getJID()));
17 assertTrue(BOB.superBot().views().sarosView()
18   .isInContactList(ALICE.getJID()));
19 }
```

Listing 7: Der bereits bestehende STF-Test testet, ob Alice einen neuen Kontakt über die Eclipse-GUI hinzufügen kann.

Der Tester Bob wird im ersten Schritt von der Kontaktliste von ALICE entfernt. Der Zustand der Liste wird nach der Entfernung bzw. dem Hinzufügen der Kontakte durch `assertFalse` bzw. `assertTrue` Anweisungen überprüft (Zeile 5-8 & 15-18). Der Test greift über den `SuperBot` auf die `SarosView` zu. Hinter der `SarosView` verbirgt sich eine STF-Klasse, welche über den `SWTBot` Operationen auf der Eclipse-GUI ausführt. Neben der hier ausgeführten `addContact(BOB.getJID())` Methode stellt die SarosView auch viele weitere Operationen zur Verfügung, welche die Interaktion mit der Eclipse-GUI ermöglichen, wie z.B. die Methoden `connect()` oder `selectChatroom(String name)`.

Soll dieselbe Funktionalität des `addContact()` STF-Test über die HTML-GUI getestet werden, wird dies über den in Listing 8 gezeigten Test erreicht.

```
1  @Test
2  public void addContactViaHTML() throws Exception {
3  assertFalse(ALICE.htmlBot().getContactList(MAIN_VIEW)
4    .contains(BOB.getBaseJid()));
5  assertFalse(BOB.htmlBot().getContactList(MAIN_VIEW)
6    .contains(ALICE.getBaseJid()));
7
8  ALICE.htmlBot().view(MAIN_VIEW)
9    .button("add-contact").click();
10 ALICE.htmlBot().view(ADD_CONTACT)
11   .inputField("jid").enter(BOB.getBaseJid());
12 ALICE.htmlBot().view(ADD_CONTACT)
13   .button("add-contact").click();
14
15 BOB.superBot().confirmShellRequestOfSubscriptionReceived();
16 ALICE.superBot().confirmShellRequestOfSubscriptionReceived();
17
18 assertTrue(ALICE.htmlBot().getContactList(MAIN_VIEW)
19   .contains(BOB.getBaseJid()));
20 asserttrue(BOB.htmlBot().getContactList(MAIN_VIEW)
21   .contains(ALICE.getBaseJid()));
22 }
```

Listing 8: Der Test überpüft, ob Alice einen neuen Kontakt über die HTML-GUI hinzufügen kann.

In diesem Test werden die HTML-Elemente wie Buttons und Eingabefelder direkt über den `HTMLBot` angesprochen (Zeile 8-13). Wie bereits in Abschnitt 4.4.1 verwendet der Test neben dem `HTMLBot` noch den `SuperBot`, um die Kontaktanfrage-Dialoge zu bestätigen. Da diese Dialoge noch nicht in der HTML-GUI existieren, müssen noch die alten Dialoge aus der Eclipse-GUI angesteuert werden, damit der Test erfolgreich durchläuft.

Mein Ziel in diesem Schritt der Arbeit war es, die Infrastruktur des STF soweit anzupassen, dass kein zusätzlicher STF-Test für die HTML-GUI mehr notwendig ist, sondern der bestehende Test weiter verwendet werden kann. Das Framework soll dazu

selbständig entscheiden, ob zum Testen der Saros-Funktionalität die alte Eclipse-GUI oder die neue HTML-GUI verwendet werden soll.

6.2 Bestehende Klassen neu implementieren

Zur Umsetzung des Ziels habe ich mir die bestehenden STF-Tests einmal genauer angesehen. Dabei ist mir aufgefallen, dass die meisten Funktionalitäten über die `SarosView` Klasse angesteuert werden.

Die Methoden der `SarosView` wie z.B. `addContact(BOB.getJID())` aus dem Listing 7 verwenden intern den `SWTBot`, um die entsprechenden Elemente in der Eclipse-GUI anzusteuern. Die Idee ist es, eine neue Klasse zu implementieren, die statt dem `SWTBot` für die alte Eclipse-GUI den `HTMLBot` für die neue HTML-GUI verwendet. Da es zu der `SarosView` Klasse bereits ein passendes Interface `ISarosView` gibt, eignet sich dieses Interface ebenfalls, um eine neue Klasse `SarosHTMLView` zu implementieren.

6.2.1 SarosHTMLView

Die neue `SarosHTMLView` Klasse implementiert auch das `ISarosView` Interface, denn sie muss dieselben Methoden bereit stellen wie die `SarosView` Klasse. Die `addContact( JID jid)` Methode der neuen `SarosHTMLView` verwendet den HTMLBot, wie in dem STF-Test zuvor, um einen neuen Kontakt über die HTML-GUI hinzuzufügen.

Die neue addContact() Methode, welche die View öffnet, die JID in das Feld eingibt und abschließend den Button zum Absenden der Anfrage anklickt ist in Listing 9 zu sehen.

```
1 @Override
2 public void addContact(JID jid) throws RemoteException {
3 htmlBot.view(ADD_CONTACT).open();
4 htmlBot.view(ADD_CONTACT).inputField("jid")
5   .enter(jid.getBase());
6 htmlBot.view(ADD_CONTACT).button("add-contact").click();
7 }
```

Listing 9: Die neue addContact() Methode in der SarosHTMLView

Auch die im STF-Test verwendete `isInContactList(JID jid)` Methode habe ich angepasst, sodass sie über den HTMLBot die Kontakte aus der HTML-GUI ausließt. Nur durch diese zwei Methoden ist es schon möglich, den alten STF-Test für die Eclipse-GUI wiederzuverwenden, um damit den Test über die HTML-GUI auszuführen. Das STF musste dahingehend angepasst werden, dass die Ansteuerung der `SarosView` über `view().sarosView()` ein Instanz der neuen `SarosHTMLView` Klasse zurückgibt. Wie genau das STF zwischen alter und neuer Implementierung unterscheidet, ist im nächsten Abschnitt 6.2.2 zu lesen.

Das Ziel, die HTML-GUI in einem STF-Test zu verwenden ohne diesen dabei selbst anpassen zu müssen, war damit im Grunde erreicht, aber nur exemplarisch für diesen einen STF-Test umgesetzt. Um weitere bestehende STF-Tests über die HTML-GUI zu nutzen, ist es notwendig, weitere Methoden in der neuen `SarosHTMLView` zu implementieren.

6.2.2 Einbau der SarosHTMLView

Wie im Abschnitt zuvor erwähnt, kann das STF nun zwei verschiedene Implementierungen des `ISarosView` Interface in der `view().sarosView()` Methode zurückgeben. Die alte `SarosView` Klasse soll zurückgegeben werden, wenn die HTML-GUI für den Tester noch nicht aktiviert wurde. Wenn die HTML-GUI aktiv ist, dann soll das Test Framework die neue `SarosHTMLView` Klasse verwenden.

Über die statischen Methode `Saros.useHtmlGui()` kann das STF abfragen, ob die HTML-GUI für diese Instanz aktiviert wurde. Daraufhin kann in der `sarosView()` Methode entschieden werden, welche Implementierung zurückgegeben werden soll. Die neue intelligentere `sarosView()` Methode ist in Listing 10 gezeigt. Der Inhalt der alten Methode ist nun in den `else`-Block (Zeile 9-12) verschoben worden. Statt der alten Ansicht von Saros wird die neue Ansicht mit dem eingebetteten Browser geöffnet und der HTMLBot in der `SarosHTMLView` gesetzt.

```
1  @Override
2  public ISarosView sarosView() throws RemoteException {
3  if (Saros.useHtmlGui()) {
4    IHTMLBot htmlBot = HTMLBotImpl.getInstance();
5    EclipseHTMLWorkbenchBot.getInstance().openSarosBrowserView();
6    htmlBot.view(View.MAIN_VIEW).open();
7    return SarosHTMLView.getInstance().setBot(htmlBot);
8  } else {
9    SWTWorkbenchBot bot = new SWTWorkbenchBot();
10   RemoteWorkbenchBot.getInstance().openViewById(VIEW_SAROS_ID);
11   bot.viewByTitle(VIEW_SAROS).show();
12   return SarosView.getInstance().setView(bot.viewByTitle(
         VIEW_SAROS));
13 }}
```

Listing 10: In dieser Methode wird die Weiche zwischen der neuen und der alten GUI gestellt. Nutzt der Tester die HTML-GUI, dann wird das SarosHTMLView Objekt zurückgegeben. Nutzt er sie nicht, kommt wie zuvor das SarosView Objekt zurück.

6.3 Zustand der HTML-GUI

Aufgrund des noch unvollständigen Zustandes der HTML-GUI war es nicht möglich, alle Methoden der `SarosHTMLView`, wie in Abschnitt 6.2.1 gezeigt, zu implementieren. Es ist z.B. nicht möglich, eine aktive Verbindung über die HTML-GUI zu trennen.

Auch der Chat wurde in der HTML-GUI noch gar nicht berücksichtigt. Die Folge ist, dass nur 5 von 35 Methoden aus dem `ISarosView` Interface in dieser Arbeit implementiert werden konnten.

Selbst Methoden, die beim Starten vor jedem STF-Test notwendig sind, konnten nicht für die `SarosHTMLView` implementiert werden. Das ist ein deutlich größeres Problem, da z.B. ein Kontakt über die HTML-GUI zu diesem Zeitpunkt meiner Arbeit noch nicht umbenannt werden kann. Folglich können auch nicht die Nicknames vor jedem STF-Test zurück gesetzt werden. Daher lässt sich über diesen Stand der HTML-GUI kein einziger STF-Test ausführen ohne dabei die Initialisierung der Tests schwerwiegend zu verändern.

Es wäre denkbar, Funktionen, die noch nicht über die HTML-GUI möglich sind, über die alte Eclipse-GUI auszuführen. Dies widerspricht jedoch dem allgemeinen Ziel von Saros, in Zukunft komplett auf die neue HTML-GUI umzustellen, sodass die alte GUI ausgebaut werden kann. Wird vom STF-Test eine Methode der `SarosHTMLView` aufgerufen, die nicht implementiert wurde bzw. aufgrund der unvollständigen HTML-GUI noch nicht implementiert werden kann, dann wird vom STF eine `UnsupportedOperationException` geworfen. Der STF-Test bricht anschließend mit der Fehlermeldung „Method is not yet implemented“ ab.

7 STF für IntelliJ

Bisher wurde das STF nur für das Eclipse Plugin gestartet, nicht aber für das IntelliJ-Plugin. Mit dem vierten Ziel meiner Arbeit soll das STF nun auch für IntelliJ integriert werden. D.h. mit dem Start des IntelliJ-Plugins soll auch das STF starten, sodass die Möglichkeit besteht, STF-Tests über IntelliJ auszuführen.

In Abschnitt 7.1 beschreibe ich, wie das STF aus der neuen STF-Komponente heraus für IntelliJ gestartet werden kann. Analog zum Eclipse-Plugin musste ich dafür die STF-Komponente zu einem weiteren IntelliJ-Plugin anpassen.

Wie das STF dann konkret gestartet wird behandelt der Abschnitt 7.2. Hier führe ich auch die Schwierigkeiten bei der Umsetzung des Ziels auf und erkläre im Detail die verschiedenen Varianten in Abschnitt 7.2.1 bis 7.2.5, welche ich zum Starten des STF näher betrachtet habe.

7.1 STF-Modul als eigenes Plugin

Wie ich bereits in Abschnitt 5.3.2 erläutert habe, ist die STF-Komponente gleichzeitig auch ein eigenständiges Eclipse-Plugin, welches nach dem Start des eigentlichen Saros-Eclipse-Plugins gestartet wird. Der Start des STF wird dazu von einer eigenen Startup-Klasse für das Eclipse-Plugin ausgeführt. Diese Anpassung im Startvorgang des Plugins war notwendig, da durch die neue Struktur der STF-Komponente, das Starten des STF aus der Eclipse-Komponente nicht mehr möglich war. Das selbe Problem ergibt sich nun auch beim Startvorgang des STF für das IntelliJ-Plugin.

Konkret bedeutet dies, dass die STF-Komponente zusätzlich zu einem Eclipse-Plugin auch ein IntelliJ-Plugin ist. Dieser Zustand der Komponente klingt auf den ersten Blick widersprüchlich, ist in der Praxis jedoch umsetzbar. In der STF-Komponente liegt unter dem `META-INF` Ordner eine `MANINFEST.MF` Datei, welche unter anderem dafür sorgt, dass diese Komponente ein Eclipse-Plugin ist. Um aus der STF-Komponente gleichzeitig ein IntelliJ-Plugin zu machen, genügt eine `plugin.xml` Datei im selben `META-INF` Ordner. In dieser Datei werden per XML die Eigenschaften des STF-IntelliJ-Plugins beschrieben. Dazu gehört z.B. auch das Definieren einer StartUp-Klasse, welche in unserem Fall eine sogenannte `Project-Component` ist. Eine `Project-Component` ist eine Klasse, die immer genau dann ausgeführt wird, sobald von IntelliJ ein Projekt geladen wird. Das Saros-IntelliJ-Plugin startet ebenfalls über solch eine `Project-Component`. Damit das STF-Plugin erst nach dem IntelliJ-Plugin läd, konnte ich in der `plugin.xml` Datei ein `depends` XML-Tag vergeben, welches definiert, dass das STF-Plugin abhängig vom IntelliJ-Plugin ist. Dieses Tag sorgt dafür, dass die beiden Plugins in der richtigen Reihenfolge nacheinander starten.

Analog zu der `StartupSarosSTF` für Eclipse habe ich eine `SarosComponentSTF` Klasse für IntelliJ implementiert, die den Start des Saros Test Frameworks über den `STF Controller` ausführt.

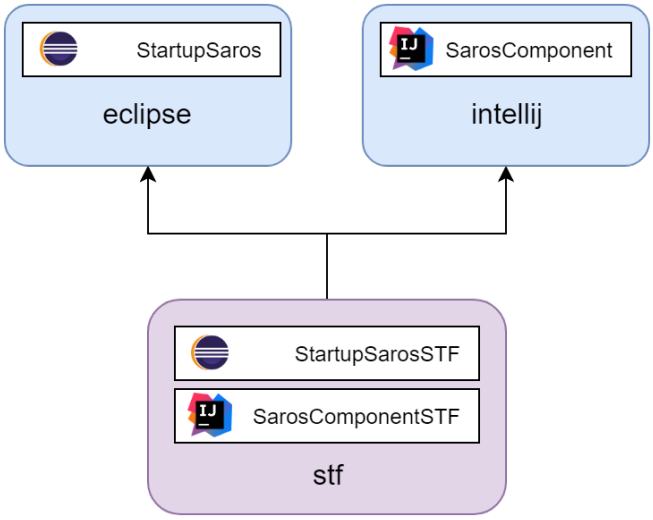

Abbildung 10: Die drei Komponenten eclipse, intellij und stf sind Plugins, welche von unterschiedlichen Starter-Klassen gestartet werden. Die Starter-Klassen sind durch weiße Kästen in den Komponenten dargestellt. Durch die Logos ist sichtbar welche Starter-Klasse zu einem Eclipse-Plugin und welche zu einem IntelliJ-Plugin gehört. Die Pfeile zeigen die Abhängigkeit der Komponenten.

Die Struktur der Eclipse-, IntelliJ- und STF-Komponente mit den Starter-Klassen in dem jeweilige Plugins ist in Abbildung 10 zu sehen. Die STF-Komponente besitzt zwei Starter-Klassen, da sie gleichzeitig Eclipse- und IntelliJ-Plugin ist. Beim Starten von Saros für Eclipse wird zunächst die Starter-Klasse `StartupSaros` des Eclipse-Plugins und danach die `StartupSarosSTF` Klasse des STF-Plugins ausgeführt. Analog dazu wird für einen Start von Saros für IntelliJ zuerst die `SarosComponent` Klasse des IntelliJ-Plugins und dann die Starter-Klasse `SarosComponenetSTF` aus dem STF-Plugin ausgeführt.

Der Vorteil dieser Lösung ist, dass keine weitere Komponente zum Starten des STF für die jeweilige IDE notwendig ist. Der gesamte Startvorgang für beide Entwicklungsumgebungen sowie das Test Framework selbst, sind in einer STF-Komponente untergebracht. Dadurch ergibt sich der Nachteil das der Code für Eclipse und IntelliJ nicht sauber getrennt wird. Da es sich nur um eine einzelne Start-Klasse für jede IDE handelt, ist dieses in Kauf zu nehmen, da zwei weitere Komponenten, welche dann zum Starten des STF implementiert werden müssten, einen deutlich höheren Aufwand verursachen würden. Auch die Komplexität der Gesamtstruktur würde sich dadurch nicht verbessern.

7.2 Starten des STF

Wie im Abschnitt 7.1 zuvor beschrieben ist, wird der Start des STF für IntelliJ über die Klasse `SarosComponenentSTF` angestoßen. Zum Starten wird für den `STFController` der Port und der Kontext des IntelliJ-Plugins in Form eines Objekts der `Container Context` Klasse benötigt.

Über den ContainerContext hat das STF Zugriff auf den PicoContainer, welchen ich bereits in Abschnitt 2.6 erklärt habe. Das STF nutzt den PicoContainer um zur Laufzeit Instanzen von Klassen zu laden, damit diese die GUI ansteuern. Für den HTMLBot bedeutet dies z.B., dass über den PicoContainer die Instanz des BrowserManager zur Laufzeit geladen wird, um abzufragen, auf welchem Browser-Objekt der entsprechende JS-Code zur Steuerung der HTML-GUI ausgeführt werden soll.

Der ContainerContext mit PicoContainer wird beim Starten des IntelliJ-Plugins erzeugt. Sobald der PicoContainer gestartet wurde, ist es möglich, über die statische Methode getComponent(Class class) eine Instanz z.B. die des BrowserManager zu laden. Es ist auch möglich über die @Inject Annotation eine globalen Variable über den PicoContainer zu befüllen. Die StartupSarosSTF Klasse startet das STF für das Eclipse-Plugin auf genau diese Weise.

```
1  public class StartupSarosSTF implements IStartup {
2
3  @Inject
4  private IContainerContext context;
5
6  public StartupSarosSTF() {
7    SarosPluginContext.initComponent(this);
8  }
9
10 @Override
11 public void earlyStartup() {
12   Integer port = Integer.getInteger("de.fu_berlin.inf.dpp.
         testmode");
13   STFController.start(port, context);
14 }
15
16 }
```

Listing 11: Verkürzte From der StartupSarosSTF Klasse die zeigt, wie das STF für das Eclipse-Plugin gestartet wird.

Das Listing 11 zeigt die auf das wesentliche Beschränkte StartupSarosSTF Klasse. Durch den Aufruf der Methode initComponent(this) (Zeile 7) wird der ContainerContext samt PicoContainer initialisiert, wodurch anschließend die Variable context (Zeile 4) mit der Instanz des ContainerContext aus dem PicoContainer befüllt werden kann. Diese Objektinstanz wird dann zum Starten des STF an den STFController weitergegeben (Zeile 13). Für das Eclipse-Plugin funktioniert dieser Startvorgang des STF ohne Probleme.

Wird der zuvor beschrieben Startvorgang in die SarosComponenetSTF Klasse übertragen, um das STF für das IntelliJ-Plugin zu starten, funktioniert dies nicht so einfach. Der Aufruf zum Initialisieren des Kontextes (Zeile 7) bricht für IntelliJ mit einer NullPointerException ab, da der ContainerContext zum Initialisieren in der SarosPluginContext Klasse nicht gesetzt wurde. Mithilfe des Debuggers konnte ich herausfinden, dass die entsprechende Methode zwar aufgerufen wird, die statische

Variable aber nur für das IntelliJ-Plugin gesetzt wurde und nicht für das neue STF-Plugin. Dies hatte zur Folge, dass der Kontext nicht über eine `@Inject` Annotation an der Variable geladen werden konnte.

In den folgenden Abschnitten 7.2.1 bis 7.2.5 möchte ich auf die verschiedene Varianten eingehen, welche ich implementiert bzw. ausprobiert habe, um vom STF-Plugin aus auf den `ContainerContext`, der von IntelliJ-Plugin erzeugt wurde, zuzugreifen. Konkreter geht es darum, den Kontext dementsprechend zu laden, dass der PicoContainer verwendet werden kann, um z.B. den `BrowserManager` im STF zu laden.

Um weitere Hilfe bei dem Problem zu erhalten, habe ich ein minimales Beispiel dieser Plugin-Konstellation nachgebaut und dieses zusammen mit der Problembeschreibung auf StackOverflow[8] und dem JetBrains-Forum[9] veröffentlicht. Leider habe ich auch über diesen Weg keinen nützlichen Hinweis zur Lösung des Problems erhalten.

7.2.1 Variante 1: Kontext statisch laden

Der `ContainerContext` wird beim Starten des IntelliJ-Plugins von der Klasse `IntellijProjectLifecycle` erzeugt und in einer Variable gespeichert. Mit dem Aufruf in Listing 12 habe ich über die statische Methode `getInstance()` Zugriff auf das Lifecycle-Objekt bekommen und somit auch auf das Kontext-Objekt.

```
1 IContainerContext context = IntellijProjectLifecycle
2   .getInstance(project)
3   .getSarosContext();
```

Listing 12: Zugriff auf das Kontext-Objekt über statische Methoden

Auf den ersten Blick sah es so aus, als würde diese Alternative funktionieren, da der zurückgegeben `ContainerContext` genau das Objekt war, welches vom IntelliJ-Plugin erzeugt wurde. Dies konnte ich über den Debugger durch einen Vergleich der Hash-Values der beiden Objekte genau beobachten. Das STF startete zunächst ohne weitere Probleme. Beim Ansteuern der HTML-GUI über das STF wird versucht, den `BrowserManager` über den Kontext bzw. PicoContainer zu laden. Genau an dieser Stelle bricht das Plugin mit einem `IncompatibleClassChangeError` ab. In der Fehlermeldung heißt es weiter: „ContainerContext does not implement the requested interface IContainerContext“. Das Problem liegt hierbei an den unterschiedlichen ClassLoadern der Plugins, d.h. die Klasse bzw. der Typ `ContainerContext` ist für das STF-Plugin nicht der selbe Typ wie für das IntelliJ-Plugin. Für das STF-Plugin sieht es so aus, als würde die Implementation der Klasse nicht zum geforderten Interface passen.

Was ich beobachten konnte ist, dass keine einzige Instanz einer Klasse über den `ContainerContext` geladen werden konnte. Obwohl das Objekt des `ContainerContext` in dem STF-Plugin dasselbe ist, wie das was in dem IntelliJ-Plugin erzeugt wurde,

[8] https://stackoverflow.com/questions/52555806/use-picocontainer-with-two-intellij-plugins

[9] https://intellij-support.jetbrains.com/hc/en-us/community/posts/360001204760-Get-object-from-one-Plugin-in-an-other-Plugin

kann kein Objekt über den PicoContainer geladen werden. Wird versucht ein Objekt über das Kontext-Objekt aus dem IntelliJ-Plugin zu laden, funktioniert dies einwandfrei. Der Fehler tritt erst dann auf, wenn versucht wird den PicoContainer aus dem STF-Plugin heraus zu verwenden.

7.2.2 Variante 2: Kontext über ServiceManager laden

Auf meinen Eintrag bei Stackoverflow hat ein Nutzer geantwortet, dass ich Services für mein Vorhaben verwenden soll. Für IntelliJ-Plugins wird über das IntelliJ-Plattform-SDK[10] ein `ServiceManager` angeboten der sicherstellt, dass für einen Service jeweils nur eine Instanz geladen wird, auch wenn dieser Service mehrmals aufgerufen wird[2].

Ich habe versucht die Instanz des `ContainerContext` über den `ServiceManager` wie in Listing 13 gezeigt zu laden.

```
1 IContainerContext context = ServiceManager
2   .getService(ContainerContext.class);
```

Listing 13: Laden der ContainerContext-Klasse über den ServiceManager

Dieser Aufruft bricht mit der Fehlermeldung „ContainerContext has unstatisfied dependency [java.util.List]“ ab. Eine Liste von `IContextFactory` konnte beim Laden der Instanz über den `ServiceManager` nicht mit aufgelöst werden, was zum Abbruch des Plugins führt.

Bei näherer Betrachtung des `ServiceManager` ist diese Variante auch nicht sinnvoll, da der `ServiceManager` ebenfalls über einen PicoContainer versucht, die Instanz der Klasse zu laden. Das würde für meinen Fall bedeuten, dass wir den PicoContainer für Saros in den PicoContainer von IntelliJ verschachteln würden. Diese Variante erschien mir dann doch deutlich zu Komplex, weshalb ich das Problem ab diesem Punkt nicht weiter untersucht habe. Den `ServiceManager` habe ich dann aber in Variante 5 (Abschnitt 7.2.5) nochmal verwendet, um nur den `BrowserManager` auf diesem Weg zu laden.

7.2.3 Variante 3: Verwendung von ExtensionPoints

In der Dokumentation des IntelliJ-Plattform-SDK bin ich neben den Services auf ein weiteres Konzept, den Extension-Points, gestoßen. Mithilfe von Extension-Points ist es laut Dokumentation möglich, von einem IntelliJ-Plugin mit einem anderen IntelliJ-Plugin zu interagieren bzw. dieses zu erweitern[1].

Dazu ist es notwendig, bei einem Plugin einen `ExtensionPoint` anzulegen, um anderen Plugins zu erlauben dieses zu erweitern bzw. auf dieses zuzugreifen. Für Saros

[10] http://www.jetbrains.org/intellij/sdk/docs/welcome.html

wäre dies das IntelliJ-Plugin, da ich auf dieses ja vom STF-Plugin zugreifen möchte. Auf der anderen Seite, in meinem Fall bei dem STF-Plugin, muss eine `Extension` definiert werden, welche den `ExtensionPoint` des ersten Plugins erweitert.[1]

Die `Extension` bzw. `ExtensionPoints` werden in der `plugin.xml` Datei des jeweiligen Plugins konfiguriert. Laut Dokumentation müsste das XML dafür wie in Listing 14 aussehen.

```
1  <extensionPoints>
2    <extensionPoint name="MyContextExtensionPoint"
3     interface="de.fu_berlin.inf.dpp.context.IContainerContext">
4  </extensionPoints>
5
6  <extensions defaultExtensionNs="de.fu_berlin.inf.dpp.intellij">
7    <MyContextExtensionPoint
8     implementationClass="de.fu_berlin.inf.dpp.context.
          ContainerContext">
9     </MyContextExtensionPoint>
10 </extensions>
```

Listing 14: Konfiguration der ExtensionPoints und Extensions in der plugin.xml laut Dokumentation von JetBrains

Diese Dokumentation scheint nicht mit der Version des IntelliJ-Plugins für Saros kompatibel zu sein, da bereits die XML-Konfiguration der Extension-Points nicht valide in der `plugin.xml` Datei ist. Demnach war es mir nicht möglich die Extension-Points zu verwenden, um Zugriff vom STF-Plugin auf das IntelliJ-Plugin zu bekommen.

Für mein selbst konstruiertes Minimal-Beispiel habe ich ebenfalls über Extension-Points versucht, zwischen zwei Plugins zu interagieren. Hier war die Konfiguration über XML zwar möglich, führte aber nicht zum gewünschten Ergebnis. Es gibt allgemein sehr wenige Beispiele, Tutorials oder Dokumentationen zu dem Thema, weshalb ich keine weiteren Ideen für diese Variante hatte und sie deshalb nicht weiter verfolgt habe.

7.2.4 Variante 4: Neuer Lifecycle für STF

Diese Variante geht das Problem etwas anders an, als die bisherigen Varianten. Bisher wollte ich den Kontext aus dem eigentlichen IntelliJ-Plugin wo er erzeugt wurde laden. Statt diesen in das STF-Plugin zu laden, habe ich in dieser Variante versucht, den Kontext erst in dem STF-Plugin zu erzeugen. Dafür war es notwendig, den gesamten Startvorgang des Plugins in das STF-Plugin zu verlagern.

Die Klasse `IntellijProjectLifecycle` sorgt dafür, dass das IntelliJ-Plugin initialisiert, d.h. der Kontext erzeugt wird, um dann das Plugin zu starten. Für das STF-Plugin habe ich eine neue Klasse `IntellijProjectLifecycleSTF` angelegt, die genau diese Aufgaben jedoch vom STF-Plugin aus übernimmt. Damit wollte ich vermeiden, dass die Erzeugung der `ContainerContext` Instanz und die Verwendung dieser in

unterschiedlichen Plugins stattfindet. Auf diese Weise wäre sicher gestellt, dass die Instanz des Kontextes immer die selbe ist.

Beim Starten des Plugins über den neuen STF-Lifecyle kann der `ContainerContext` gar nicht erst gebaut werden. Beim Erzeugen des Kontextes tritt die Fehlermeldung „Cannot Cast SarosIntelliJContextFactory to IContextFactory“ auf. Diese Meldung sorgt für Verwirrung, denn im Code kann eindeutig nachvollzogen werden, dass die `SarosIntelliJContextFactory` das Interface der `IContextFactory` implementiert. Ein Umwandeln zu diesem Typen sollte daher ohne Probleme möglich sein. Nach weiterer Recherche bin ich zu der Vermutung gekommen, dass hier wieder ein Problem mit unterschiedlichen ClassLoadern besteht. Erst zur Laufzeit wird versucht eine Klasse in eine andere Klasse zu „casten“, die aber über einen anderen ClassLoader geladen wurde. Dies führt zum oben genanntem kuriosen Fehler.

Auch in dieser Variante stoßen wir vermutlich an das gleich Grundproblem wie in Variante 1 (Abschnitt 7.2.1). Die Unterschiedlichen ClassLoader der Plugins lassen eine Interaktion zwischen den Plugins nicht zu.

7.2.5 Variante 5: Nur BrowserManager laden

In Variante 2 (Abschnitt 7.2.2) habe ich versucht den gesamten `ContainerContext` samt PicoContainer über den `ServiceManager` zu laden. Bei genauerer Betrachtung des Codes für das STF zur Steuerung der HTML-GUI habe ich bemerkt, dass der `BrowserManager` die einzige Klasse ist, die für den HTML-Teil des STF wirklich notwendig ist.

```
1 BrowserManager browserManager = ServiceManager
2   .getService(BrowserManager.class);
```

Listing 15: Laden des BrowserManager-Objekts über den ServiceManager

Listing 15 zeigt den Aufruf der gezielt versucht den `BrowserManager` zu laden. Ausgeführt wird dies nicht in der Starter-Klasse für das STF, wie in Variante 2 (Abschnitt 7.2.2), sondern intern im STF an der Stelle an der der `BrowserManager` tatsächlich erst benötigt wird.

Diese Methode schien zunächst erfolgversprechend, da der `ServiceManager` es schafft, eine Instanz des `BrowserManagers` zur Laufzeit an der benötigten Stelle im Code nachzuladen. Allerdings handelt es sich dabei nicht um die beim Erstellen bzw. Öffnen der Browser-Ansicht erzeugten Instanz, sondern um eine vollkommen neu erstellte Instanz. Daraus ist abzuleiten, dass diese neue Instanz lediglich eine leere Liste über die geöffneten Browser beinhaltet. Benötigt wird dies Liste aber genau aus dem ursprünglichem `BrowserManagers`-Objekt, welches die Referenzen zu den Browsern hat, auf die das STF zugreifen soll.

Ich vermute, dass das Konzept der Services für IntelliJ-Plugins nicht dafür entworfen ist, über Plugin-Grenzen hinweg auf Services zuzugreifen. Diese Variante löst damit leider auch nicht das Problem um zwischen zwei IntelliJ-Plugins zu interagieren.

8 Zusammenfassung

Innerhalb der Bearbeitungszeit von 23 Wochen für diese Arbeit habe ich drei von vier Ziele vollständig erreicht und das vierte Ziel teilweise. Etwa zwei Drittel des Zeitraums benötigte ich für die Einarbeitung und Implementierungsarbeiten. Ein Drittel der Zeit wurde für das Schreiben dieser Arbeit aufgewandt. Die Ziele 2 und 4, die Modularisierung und den Start des STF für IntelliJ, stellten für mich die technisch größten Herausforderungen dar und verlangten dementsprechend auch am meisten Zeit.

Mit dem ersten Ziel, welches ich in Kapitel 4 beschrieben habe, wurde der Grundstein für diese Arbeit gelegt. Über den `HtmlBot` ist es nun möglich, STF-Tests für die HTML-GUI zu schreiben, sodass Funktionstests für Saros über die neue Oberfläche durchgeführt werden können. Die neue API macht es möglich, gezielt HTML-Elemente anzusteuern, damit der Entwickler dafür nicht selber Javascript-Code verwenden muss.

Durch eine neue Struktur der Projekt-Komponenten konnte das STF vom eigentlichen Quellcode der Saros-Plugins getrennt und in eine eigenständige Komponente ausgelagert werden. Die STF-Tests, welche ebenfalls in eine neue Komponente verschoben wurden, nutzen die STF-Komponente wie ein Framework. Die neuen Komponenten wurden mit dem zweiten Ziel in Kapitel 5 eingeführt. Die Trennung der Komponenten bietet einen Vorteil für nachfolgende Saros-Entwickler. Jemand der keine Änderung am STF vornehmen möchte, braucht diese Komponente fortan gar nicht mehr zu importieren. Die Menge des für ihn sichtbaren Quellcodes reduziert sich somit, wodurch der Entwickler einen besser Überblick behält.

Eine Voraussetzung für das STF war es, die bestehenden STF-Tests für die neue HTML-GUI anzupassen. Somit war mein drittes Ziel durch möglichst wenig Änderungen des Codes die STF-Test auf die HTML-GUI umzuschreiben. In Kapitel 6 ist beschrieben, wie die Infrastruktur des Saros Test Frameworks angepasst wurde, sodass bereits vorhandene STF-Tests einfach wiederverwendet werden können. Eine neue Implementierung der `SarosView` Klasse sorgt dafür, dass statt dem `SWTBot` der `HTMLBot` verwendetet wird, weshalb nicht mehr die SWT-Eclipse-GUI, sondern die neue HTML-GUI vom selben Test angesteuert wird. In der HTML-GUI fehlen zentrale Funktionen von Saros, was zur Folge hat, dass bestehende STF-Tests über die HTML-GUI nicht durchlaufen würden.

Das vierte Ziel, das STF für IntelliJ zu starten, wurde in dieser Arbeit nur teilweise erfüllt. Das STF konnte für IntelliJ wie in Kapitel 7 gezeigt grundsätzlich gestartet werden, allerdings nicht ohne Einschränkungen. Durch die neuen Komponenten aus dem zweiten Ziel konnte der PicoContainer des IntelliJ-Plugins nicht mehr von dem STF genutzt werden. Dadurch war der Zugriff auf notwendige Objekte, wie z.B. dem `BrowserManager`, nicht mehr möglich. Somit konnte z.B. kein STF-Test, welcher den `HTMLBot` zur Steuerung der HTML-GUI in IntelliJ verwendet, gestartet werden. Das Ziel ist teilweise erfüllt, da das Saros Test Framework prinzipiell mit dem IntelliJ-Plugin gestartet wird, es aber in einem STF-Test für IntelliJ aufgrund des genannten Problems nicht wie gefordert verwendet werden kann.

9 Fazit

Durch meine Arbeit am Saros-Projekt ist es nun möglich, STF-Tests über die HTML-GUI auszuführen. Der neue `HtmlBot` ermöglicht es, die HTML-GUI bzw. die angezeigten Elemente anzusteuern. Keine meiner Änderungen sind für den Benutzer von Saros sichtbar. Im Wesentlichen profitieren die aktuellen sowie zukünftigen Entwickler von Saros durch meine Arbeit. Den Frontend-Entwicklern wird nun ermöglicht, die neue HTML-GUI direkt in Eclipse über das STF zu testen. Die Backend-Entwickler kommen durch die neuen Komponenten nicht mehr direkt mit dem STF-Code in Berührung. Entwickler, die das STF weiterentwickeln wollen, bekommen durch die neue Strukturierung einen besseren Überblick über das Framework und die dazugehörigen Tests.

Das Resultat der fünf Versuche das Problem zu lösen ist, das keine der in 7.2.1 bis 7.2.5 gezeigte Variante es geschafft hat, das vierte Ziel dieser Arbeit zu erfüllen. Das STF kann nur mit Einschränkungen über die in Abschnitt 7.2 beschriebenen Klassen für das IntelliJ gestartet werden. Es ist jedoch nicht möglich benötigte Objekte über den PicoContainer zu laden. Das Kernproblem besteht darin, dass es mir nicht gelungen ist, vom STF-Plugin aus auf den `BrowserManager` zuzugreifen.

Die Varianten 1 und 4 führen beide zum gleichen Problem. Durch die zwei Plugins gibt es Konflikte zwischen den unterschiedlichen ClassLoadern, wodurch eine übergreifende Interaktion der Klassen verhindert wird. In Variante 2 und 5 wird versucht das Konzept der Services zu benutzen, um benötigte Objekte zur Laufzeit des Programmes nachträglich zu laden. Allerdings sind Services nicht dazu ausgelegt Plugin-Grenzen zu überwinden. Für die in Variante 3 verwendeten ExtensionPoints gibt es zu wenig Dokumentation, sodass ich es nicht geschafft habe, das STF auf diese Art und Weise für das IntelliJ-Plugin zu starten. Laut der offiziellen Dokumentation von Jetbrains[1] sind Extensions bzw. ExtensionPoints jedoch genau dazu entwickelt worden, um zwischen verschiedenen Plugins zu interagieren.

Dass ich das vierte Ziel nicht geschafft habe, ist für mich persönlich ziemlich schade. Allerdings stellte sich auch erst während der Arbeit an Saros heraus, dass das Problem für IntelliJ doch etwas komplizierter ist, als zu Beginn der Arbeit erahnt. Die selbe Problematik ließ sich für Eclipse durch zwei Zeilen lösen. Dass für IntelliJ keine ähnlich einfache Lösung gefunden werden kann, ließ sich im Vorfeld nicht absehen.

10 Ausblick

Aufbauend auf den vom mir versuchten Varianten aus Kapitel 7 muss eine Lösung gefunden werden, damit das STF vollständig für IntelliJ genutzt werden kann. Auch wenn ich es in der Bearbeitungszeit dieser Arbeit nicht geschafft habe, einen funktionierenden Weg zu finden, denke ich, dass es eine Lösung gibt, um Objekte zwischen zwei IntelliJ-Plugins auszutauschen.

Für eine aufbauende Arbeit würde ich eine weitere Analyse in die korrekte Verwendung von ExtensionPoint aus Variante 3 (Abschnitt 7.2.3) empfehlen, da mir diese am erfolgversprechendsten erscheint. Bei allen anderen Varianten sehe ich weniger Chancen. Wenn das Problem mit den unterschiedlichen ClassLoadern aus Variante 1 und 4 gelöst werden kann, dann besteht die Möglichkeit auch diese Ansätze zum Starten des STF weiter zu entwickeln. Natürlich ist auch eine komplett neue Variante denkbar.

Ein großes Ziel im Saros Projekt ist es, die alte SWT-GUI auszubauen und nur die neue HTML-GUI zu verwenden. Ist dieses Ziel geschafft, können viele Klassen aus dem STF entfernt werden. Bis dahin müssen noch viele weitere Funktionen in der HTML-GUI implementiert werden. Bei der Weiterentwicklung der HTML-GUI sollten vom Entwickler auch sofort die entsprechenden Methoden im STF angepasst werden, sodass direkt über das STF die neuen Funktionen der GUI getestet werden können. Insbesondere sind damit die Methoden in der `SarosHTMLView` Klasse gemeint. In dieser befinden sich nach meiner Arbeit einige Platzhalter, da die entsprechende Funktionen noch nicht in der HTML-GUI existieren. Diese müssen in Zukunft durch sinnvolle Anweisungen zur Steuerung der HTML-GUI ersetzt werden.

Sollten in der HTML-GUI weitere Elemente verwendet werden als bisher, müssen analog zu den STF-Klassen für HTML-Buttons, Input-Elementen, usw. neue Klassen geschrieben werden, um diese neuen HTML-Elemente über die API vom STF anzusteuern. Dabei muss das Regelwerk aus Abschnitt 4.5 für HTML-Eingabeelemente beachtet werden. Denkbar sind auch gesonderte STF-Klassen für spezielle Elemente der HTML-GUI wie z.B. Kontakte oder Chat-Nachrichten.

Literaturverzeichnis

[1] IntelliJ Platform SDK DevGuide - Plugin Extensions and Extension Points. https://www.jetbrains.org/intellij/sdk/docs/basics/plugin_structure/plugin_extensions_and_extension_points.html.

[2] IntelliJ Platform SDK DevGuide - Plugin Services. https://www.jetbrains.org/intellij/sdk/docs/basics/plugin_structure/plugin_services.html.

[3] R. Djemili. Entwicklung einer Eclipse-Erweiterung zur Realisierung und Protokollierungverteilter Paarprogrammierung. Diplomarbeit, Freie Universität Berlin, Inst. für Informatik, 2006.

[4] Erich Gamma, Ralph Johnson, Richard Helm, and John Vlissides. *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Pearson Deutschland GmbH, 2011.

[5] Frank Müller-Hofmann, Martin Hiller, and Gerhard Wanner. Remote method invocation. In *Programmierung von verteilten Systemen und Webanwendungen mit Java EE*, pages 57–80. Springer, 2015.

[6] Marius Schidlack. Neuimplementierung und Weiterentwicklung der HTML-GUI von Saros. Bachelorarbeit, 2017.

[7] David Sungaila. Verbesserung und erweiterung der core-bestandteile von saros. Bachelorarbeit, 2016.

[8] Sandor Szuecs. Behandlung von Netzwerk- und Sicherheitsaspekten in einem Werkzeug zur verteilten Paarprogrammierung. Diplomarbeit.

Abbildungsverzeichnis

Quellcodeverzeichnis

A Anhang

A.1 Angenommen Pull-Requests

- [FEATURE][HTML] Implement Configuration Wizard UI
  https://github.com/saros-project/saros/pull/174
- [STF] Add RemoteHTMLInputField and use it in AddContactTest
  https://github.com/saros-project/saros/pull/186
- [STF] Open HTML view via SarosApi and InputField improvement
  https://github.com/saros-project/saros/pull/187
- [STF] Added ComponentTestView for HTML component testing
  https://github.com/saros-project/saros/pull/188
- [STF] Add checkbox, radio group, select, multi select and progressbar
  https://github.com/saros-project/saros/pull/189
- [STF] Add RemoteHTMLTextElement to test functionality of buttons
  https://github.com/saros-project/saros/pull/190
- [STF][RREFACTOR] Extract view enum
  https://github.com/saros-project/saros/pull/191
- [STF] Add StartSessionWizardTest
  https://github.com/saros-project/saros/pull/192
- [STF][REFACTOR] remove duplicate code, move/rename test and packages
  https://github.com/saros-project/saros/pull/194
- [STF] Add RemoteHTMLTree and use it in StartSessionWizardTest
  https://github.com/saros-project/saros/pull/196
- [STF] Add ContactListItem accessible and complete the test
  https://github.com/saros-project/saros/pull/197
- [STF] Add checkbox, radio group, select, multi select and progressbar
  https://github.com/saros-project/saros/pull/228
- [STF] Add RemoteHTMLTextElement to test functionality of buttons
  https://github.com/saros-project/saros/pull/229
- [STF][RREFACTOR] Extract view enum
  https://github.com/saros-project/saros/pull/230
- [STF] Add StartSessionWizardTest
  https://github.com/saros-project/saros/pull/231

A.2 Offene Pull-Requests

- [FEATURE][HTML] Implement 'Rename' and 'Delete' operations for contacts #173 https://github.com/saros-project/saros/pull/173
- [UI][FIX] Set size of EclipseBrowserDialog #193
  https://github.com/saros-project/saros/pull/193
- [BUILD] Update the swtbot from 2.0.5 to 2.7.0 #220
  https://github.com/saros-project/saros/pull/220
- [BUILD] Clean up hamcrest libraries #221
  https://github.com/saros-project/saros/pull/221
- [STF] Add SarosHTMLView for superBot #222
  https://github.com/saros-project/saros/pull/222
- Außerdem die Commit in folgenden Branches:
  - pr/stf_move_all_stf_parts
  - draft/stf_intellij