Freie Universität Berlin
Fachbereich für Mathematik und Informatik
Institut für Informatik


# Masterarbeit

## Erstellung eines Testframeworks für das Schreiben von Unit-Tests im Projekt Saros

05. April 2011

Bearbeitet von:
Philipp Cordes
Gubener Str. 7
10243 Berlin
Kontakt: philipp.cordes@fu-berlin.de

Betreut von der Arbeitsgruppe für Softwaretechnik:
Eingereicht bei: Prof. Dr. Lutz Prechelt
Betreuer: Stephan Salinger

# Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe. Diese Arbeit wurde keiner anderen Prüfungsbehörde in gleicher oder ähnlicher Form vorgelegt.

Berlin, den 05. April 2011

.....
Philipp Cordes

# Inhaltsverzeichnis

# 1 Einleitung

## 1.1 Motivation

Bevor wir uns auf eine längere Reise begeben planen und bereiten wir diese vor. Wir entschließen uns für eine geeignete Reisemöglichkeit und Route um sicher anzukommen. Wir versuchen so gut es geht im Voraus zu bedenken was wir alles möglicherweise gebrauchen und vielleicht schief gehen könnte.

Hätte Columbus die Möglichkeit gehabt, sich schon im Voraus sein Ziel genauer anzuschauen oder die beste Route zu wählen, so wäre seine Entdeckung sicher anders verlaufen. Wie wir aber alle wissen standen ihm diese Informationen nicht zur Verfügung. Er war sich zwar darüber sicher, dass es einen Weg von Portugal nach Indien geben würde und dass er auch diesen gefunden hat aber die Erkenntnis darüber was er tatsächlich entdeckt hat kam erst später.

Auf eine sehr ähnlich Reise habe ich mich begeben. Mein Vorhaben war geprägt von von vielerlei Ungewissheiten. Zwar hatte ich das Ziel welches ich zu erreichen vermochte klar vor Augen, aber wie es nun heute tatsächlich aussieht hätte ich nicht erwartet.

Angekommen auf dieser paradiesischen Insel kam es zudem zu einer weiteren Erkenntnis, denn mit der Perspektive von dort aus auf meine Heimat sehe ich heute nun einiges anders wie zuvor.

Meine Vorstellung war es für das studentische Softwareprojekt Saros ein Testframework zu erstellen welches es ermöglicht schnell und einfach automatisierte Tests zu schreiben die zudem nachvollziehbar und gut lesbar sind. Der Titel der Arbeit weist auf die mögliche Route hin die ich damals mir in etwa vorstellte.

Tatsächlich habe ich das Ziel also die paradiesische Insel für testbasiertes Arbeiten im Projekt Saros erreichen können. Die Reise dorthin war allerdings davon geprägt neue geeignete Wege zu finden und Ballast loszuwerden um am Ende nicht zu stranden.

Aber warum dieses gelobte Land überhaupt versuchen zu finden? Grob gesagt ging es wie so oft bei Expeditionen ins Ungewisse darum neue Rohstoffe zu finden die uns neue Möglichkeiten schaffen. Das Gut um welches es hier geht ist Testbarkeit.

## 1.2 Zielstellung

Softwaresysteme oder auch Softwarekomponenten sind in den meisten Fällen komplexe geistige Bauwerke. Ihre Funktion und der grobe Aufbau erschließen sich uns zumeist schon aus der Betrachtung. Ob sie allerdings ihre Funktion erfüllen, können wir ihnen nicht sofort ansehen. Vor dem Bau einer Brücke werden Simulationen durchgeführt um ihre Bestandsfähigkeit im zu testen. Aber auch während ihres Baus werden Tests durchgeführt um zu prüfen, ob man sich dem gegebenem Ziel richtig annähert. Ist die Brücke fertiggestellt werden wiederholt weitere Tests durchgeführt um zu prüfen, ob die Brücke noch immer den Anforderungen genügt.

Auch in der Entwicklung von Software etabliert sich mehr und mehr ein automatisiertes Vorgehen beim Testen dieser. Zu schnell kann es passieren, dass sich während der Entwicklung oder Anpassung das Verhalten einer Funktionalität ungewollt ändert. Ohne automatisierte Tests die immer und immer wieder die Anwendungslogik ausführen und prüfen, ob sie bestimmten Erwartungen genügt, muss dies manuell geprüft werden. Das ist zeitaufwändig und fehleranfällig. Die Definition solcher manuellen Tests erfolgt dann oft auf Basis von schriftlichen Dokumenten die mit der Weiterentwicklung des Softwareproduktes gepflegt werden müssen. Oft kommt es aber vor, dass solche Dokumente im Laufe der Zeit nicht mehr den aktuellen Stand der Software widerspiegeln.

Wünschenswert wären hingegen automatische Tests, die man ständig immer wieder ausführen kann, um zu prüfen ob sich eine Softwarekomponente weiterhin so verhält wie man es erwartet.

Das studentische Softwareprojekt Saros ist ein Softwareprojekt welches ein Maß an Komplexität erreicht hat mit dem es ohne automatisierte Tests immer schwieriger wird Programmcode zu entwickeln der wartbar und nachvollziehbar ist. Ohne ausreichend viele automatisierte Tests die einem den Zustand der Anwendungslogik nach einer Änderungen aufzeigen können werden Änderungen am Programmcode immer schwerfälliger. Das Produkt ist sogar mittlerweile an einem Punkt, an dem es immer zeitraubender wird neue Tests zu schreiben was sich direkt auf die Bereitschaft wiederum auswirkt Tests zu schreiben.

Mein Ziel für diese Arbeit war es, diese Spirale die zu immer weniger getestetem Programmcode führt zu durchbrechen. Die Aufgabe war es also Saros wieder testbar zu machen, damit es den Entwicklern leichter fällt neue Tests zu schreiben.

Unter testbar stellte ich mir damals wie auch heute vor, dass man die Möglichkeit hat im Programmcode ein oder mehrere Saros-Instanzen zu instanziieren und mit diesen auf Programmcode-Ebene Aktionen auszuführen deren Ergebnisse man dann automatisch prüfen kann. Das war wie diese Arbeit auch zeigen wird bislang nicht möglich.

# 2 Voraussetzungen

## 2.1 Allgemeine Voraussetzungen

Folgende Randbedingungen und Definitionen möchte ich dieser Arbeit zu Grunde legen:

**Programmiersprache** Da das Projekt Saros vollständig in der Programmiersprache Java entwickelt wird beschränke ich mich in meiner Arbeit ebenfalls nur auf diese Sprache.

**Framework** Eine Framework stellt eine Sammlung von Funktionalitäten und Komponenten dar, welches für eine Domäne etablierte Funktionalitäten und Konzepte kapselt und dem Benutzer des Frameworks erlaubt unter Einhaltung vorgegebener Bedingungen auf die Funktionalität zurückgreifen zu können.

**Funktionen** Statische Methoden werden als Funktionen bezeichnet.

**Instanz** Als eine Instanz oder auch Exemplar bezeichne ich die die konkrete Ausprägung eines Objektes einer Klasse.

## 2.2 Eclipse

Grundsätzlich ist Eclipse nach [8] eine Open Source Community welche in der Programmiersprache Java unter anderem eine populäre Entwicklungsumgebung die Eclipse IDE baut. Dieser Entwicklungsumgebung liegt das Framework Eclipse Rich Client Platform (RCP) zu Grunde welches ein Rahmenwerk für das Erstellen von Rich Client Anwendungen, also Anwendungen mit einer Benutzerschnittstelle bereitstellt.

Das Framework und seine Funktionalität sind in Komponenten, sogenannten Plug-Ins unterteilt. Durch diesen Komponenten-Ansatz kann man beispielsweise die Eclipse IDE durch neue Funktionalitäten erweitern indem man selbst ein solches Plug-In erstellt, welches dann auf bestehende Plug-Ins und Kernfunktionalitäten von Eclipe zugreifen kann. So stellt das Framework zum Beispiel Abstraktionen für den Umgang mit Java-Projekten und deren Dateien bereit. Projekte werden dabei in einem sogenannte Workspace gelistet welcher die zentrale Schnittstelle für den Zugriff auf die Projekte darstellt.

## 2.3 XMPP

Das Extensible Messaging and Presence Protocoll (XMPP) ist ein XML-basierte Netzwerkprotokoll zum Austausch von Informationen zwischen einer oder mehrerer Parteien ([12]). Dem Protokoll liegen die Prinzipien von IRC-Chatsystemen [1] zu Grunde. Der Austausch der Nachrichten findet dabei über ein dezentrales Netz von XMPP-Servern und Clienten statt. Die Definition des Protokolls zeichnet sich vor allen Dingen dadurch aus, dass diese verhältnismäßig klein ist und zudem explizit vorsieht das man selbiges leicht erweitern kann.

Zu den für diese Arbeit wichtigsten Elemente des Standards gehören:

**Account** Um Nachrichten mit anderen Clienten auszutauschen benötigt man einen Account auf einem XMPP-Server A. Man kann dann mit anderen sogenannten Usern in Kontakt treten, wenn diese ebenfalls direkt auf dem gleichen Server A oder einem anderen XMPP-Server der mit dem Server A diese Daten austauscht.

**Verfügbarkeit** Man kann über einen XMPP-Server die Verfügbarkeit eines anderen Uses prüfen, also ob dieser z.B. erreicht werden möchte oder nicht.

**Roster** Der Roster entspricht einer Kontaktliste von Usern zu einem XMPP-Account. Möchte man einen anderen User als Freund [2] hinzufügen kann es ja nach Umgebung sein, dass dafür eine Zustimmung notwendig ist.

[1] IRC (Internet Relay Chat) ist ein Konzept für die Struktur eines netzwerkbasierten Systems zum Austausch von Textnachrichten.

[2] Die Beziehung „Freund“ sei hier im folgenden als gegenseitiges Hinzufügen des jeweils anderen Users zu dem Roster zu verstehen.

## 2.4 Saros

Saros ist ein studentisches Softwareprojekt welches ein Plug-In für die Entwicklungsumgebung Eclipse erarbeitet. Dieses Plug-In ermöglicht es Entwicklern Paarprogrammierung unabhängig vom Ort durchführen zu können. So ist es beispielsweise möglich, dass ein Entwickler einen anderen zu einer Programmiersitzung einlädt, welcher dann als Beobachter oder Produzent (Driver) an dieser Teilnehmen kann. Alle an einer Sitzung Beteiligten können dann sofort die Änderungen an einem Dokument mit verfolgen und so während der Sitzung Hinweise, Ideen oder Kommentare abgeben ([13]). Saros sorgt während einer Sitzung ebenfalls dafür, dass die Dokumente zwischen den Parteien synchronisiert werden. Zusätzlich zu dieser Kernkompetenz bietet Saros den Teilnehmern einer Sitzung weitere Funktionaliäten wie z.B. Chatten, oder Screensharing.

Es handelt sich also um ein Plug-In welches Informationen über Aktionen und Zustände mehrerer Eclipse-Instanzen über eine Netzwerkverbindung austauscht. Als Kommunikationsschicht hat man sich damals für den offenen Standard XMPP entschiedenen.

Saros ist eine Erweiterung des XMPP-Protokolls speziell für den Austausch von Editor-Aktivitäten. Es ist darauf spezialisiert diese Aktivitäten möglichst effizient zu übermitteln, damit die Änderungen des schreibenden Entwicklers so zeitnah wie möglich bei den Beobachtern zu sehen sind - hierzu wurde der Jupiter Algorithmus aus [11] adaptiert. Dabei musste man für die Austausch von Aktivitäten auf verschiedenste Problemklassen eingehen. Vor allen Dingen der hohe Grad an Nebenläufigkeit die einer Saros-Sitzung zu Grunde liegt sorgt für ein Maß an Komplexität die einem Entwickler ein hohes Maß an Überblick abverlangt um die Abläufe zu verstehen.

Da das Protokoll ständigen Änderungen unterworfen ist, gibt es keine aktuelle Dokumentation der Abläufe, was den Einstieg in das Projekt für jemanden sehr erschwert, der in dem Bereich der Kernfunktionalitäten arbeiten möchte. Zudem gibt es viele Arten von Szenarien die es gilt zu beachten, beispielsweise das verzögerte Erhalten von Nachrichten oder das Vermischen solcher. Nicht zu vergessen spielt auch die Zahl der Teilnehmer und die Aufteilung derer Rollen einen nicht zu verachtenden Faktor.

### 2.4.1 Qualitätssicherung

Wie schon oben dargestellt, sind Teile des Programmcodes von Saros relativ unübersichtlich bzw. schwer verständlich. Gerade für Studenten die oft noch keine Erfahrung mit solchen großen Projekten haben gestaltet es sich schwer, sich allein in den Programmcode einzuarbeiten. Ihnen wird von Beginn an viel Sorgfalt abverlangt.

Um also potentiellen Fehlern entgegen zu treten gibt es zwei hier nennenswerte Prozessschritte die eine Änderung oder ein neue Funktionalität im Projekt durchlaufen müssen, bevor sie in einen Release des Plug-Ins einfließen können. So muss beispielsweise eine Änderung am Code einer gründlichen Review des Teams unterzogen werden. Die Regel hier besagt, dass man erst mit zwei Zustimmungen die Änderung in die Versionskontrolle einchecken darf. Oft sind aber die Änderungen für eine Review zu groß, weshalb es als Gutachter schwer fallen kann eine so durchzuführen, dass man alle Zusammenhänge der Änderung versteht. Des Weiteren werden die Änderungen meist nur manuell getestet. Kommt es trotz Review dazu, dass ein neuer Defekt in den Code-Stand mit einfließt, kann es mitunter bis zum Release und darüber hinaus andauern bis das Versagen zu Tage kommt.

Der zweite wichtige Prozessschritt für die Qualitätssicherung ist die Durchführung von manuellen Tests. Im Projekt gibt es eine Sammlung in Prosa verfasster Tests mit denen man versucht die verschiedenen Szenarien abzubilden die das Plug-In in jedem Fall bestehen muss. Diese Menge von Tests (Testsuite) wird pro Monat einmal komplett zu einem Release durchgeführt. Hat ein Defekt den Weg während eines Monats in den Code-Stand gefunden so fällt er womöglich erst jetzt auf, was Rückschlüsse auf die mögliche Ursache sehr erschwert. Zudem ist die Einrichtung der Testumgebung für diese Tests aufwändig und muss ebenfalls manuell durchgeführt werden.

Die Zahl der vielen manuellen Schritte (auch wenn diese natürlich nicht vollständig wegzudenken sind) lässt darauf schließen, dass hier wenig bis gar nicht testbasiert gearbeitet wird. Tatsächlich gibt es mehrere Hundert automatische Tests. Allerdings bilden diese ganz offenbar viele der manuellen Tests nicht hinreichend ab.

## 2.5 Automatisiertes Testen

Automatisierte Tests im Allgemeinen dienen dazu Anwendungslogik auszuführen und zu prüfen ob die Ergebnisse dieser Ausführung bestimmten Erwartungen gerecht werden.

In [5] erläutert Martin Fowler zwei Arten von automatisierten Tests. Unit-Tests, auch Komponententests genannt beschränken sich auf das Ausführen von Funktionalitäten einzelner Komponenten. Die Granularität nach der man hier eine Komponente definiert kann allerdings danach variieren was man im Einzelfall nach als Komponente bezeichnen möchte. Für die Ausführung von Anwendungslogik mehrerer Komponenten dienen die sogenannten Funktionstests. Diese sind vor allen Dingen dazu gedacht, dass Zusammenspiel der einzelnen Komponenten zu testen. Eine Menge von Funktionstest können wir als Integrationstest bezeichnen.

Idealerweise sollten die Unit-Tests für eine bestimmte Komponente nach jeder Änderung ausgeführt werden um dem Entwickler möglichst früh Feedback darüber zu liefern ob er gerade im Begriff ist die Erwartungshaltung an eine Funktionalität zu ändern. Man spricht hierbei von Regressionstests [1]. Ob alle oder nur eine Teilmenge der automatisierten Tests ausgeführt wird hängt von dem aktuellen Fokus ab. In [9] beispielsweise teilt man die Tests in verschiedene Mengen ein um damit den Deployment-Prozess einer Anwendung in verschiedene Schritte zu zergliedern.

Die extremste Form des testbasierten Arbeitens ist die testgetriebene Softwareentwicklung. Hier formuliert man zuerst einen automatischen Test der bei der ersten Ausführung fehl schlägt. Es sollte dann das Ziel sein möglichst schnell und einfach den notwendigen Programmcode zu schreiben um den Test erfolgreich Ausführen zu lassen.

Im Folgenden werde ich den Begriff „testbasiertes Arbeiten“ verwenden für die Entwicklung von Programmcode die entweder getrieben durch Tests stattfindet oder bei der Tests im Nachhinein entstehen.

Der Vorteil dieser Art der Entwicklung ist, die Perspektive die man auf den zu schreibenden Programmcode einnimmt. Denn bevor man überhaupt mit dem Schreiben einer Zeile beginnen kann muss man sich darüber im klaren werden was man von der Funktionalität für Ergebnisse erwartet und wie man sie benutzen möchte. Diese Perspektive kann große Auswirkungen auf die Architektur der gesamten Software haben. Hält man es genau mit der testgetriebenen Entwicklung so sollte dabei eine Struktur entstehen, die erstens leicht testbar ist und zudem, sofern weitere Bedingungen erfüllt sind, leicht verständlich also nachvollziehbar ist.

Verständlichkeit und Nachvollziehbarkeit sind zwei der wichtigsten Aspekte in der Softwareentwicklung. Sind diese im Programmcode eines Produktes hinreichend gut umgesetzt, sollte das Produkt gegenüber Änderungen auf funktionaler wie auch sozialer Ebene gewappnet sein. Zum einen kann man an solchem Programmcode wesentlich schneller und sicherer Änderungen vornehmen und zum anderen können Neulinge die den Programmcode noch nicht kennen viel eher die Abläufe verstehen und so schneller produktiv werden.

Automatisierte Tests gehören wie andere Artefakte eines Softwareprojektes selbst mit zum Produkt. Beispielsweise können Tests in Prosa geschriebene Dokumentation ersetzen. Durch das Lesen eines Tests kann man die Intention des Entwicklers zu einer Funktionalität besser nach-

vollziehen denn hier ist zu erkennen wie er sich die Benutzung einer Komponente vorstellt. So kann man eher auf eine architekturelle Ideen schließen. Zudem wird so auch das Zusammenspiel von mehreren Komponenten wie „Entitäten“, „Services“ klarer. Vor allen Dingen die Unit-Tests die direkt die Schnittstellen der Anwendungslogik verwenden (also keine Tests die die Benutzerschnittstelle - GUI - benutzen) haben den Vorteil, dass diese zwangsläufig bei der Änderung einer Schnittstelle mit geändert werden müssen. Das heißt diese Tests sind keine tote Dokumentation sondern sie spiegeln ständig die entsprechende Erwartungshaltung an eine Funktionalität aus dem Produkt wider.

Eine sehr abstrakte Form von automatischen Tests sind Tests, die die Benutzerschnittstelle verwenden um eine Anwendung genauso wie der Benutzer auszuführen. Für solche Arten von Tests wurde bereits für Saros ein Testframework entwickelt auf das wir später zu sprechen kommen.

### 2.5.1 JUnit

Komponententests oder auch Unit-Tests genannt, sind je nach Ausprägung die kleinste Einheit von automatisierten Tests die es gibt. Zumeist konzentriert auf kleine Teile des Programmcodes, führen diese Anwendungslogik aus und testen, ob die Ergebnisse dessen vorgegebene Erwartungen erfüllen.

Für jede bekannte Programmiersprache gibt es Frameworks die einem beim Schreiben von Unit-Tests unterstützen und die gängigen Mittel die man dafür benötigt anbieten. JUnit [3] ist ohne Frage das bekannteste Testframework für Java, es bietet alle nötigen Funktionalitäten um Test-Methoden, Test-Klassen und Test-Suiten also Mengen von Test-Klassen zu erstellen. Des Weiteren können wir leicht Funktionalitäten erstellen die vor oder nach einem Test ausgeführt werden müssen. Ein Beispiel für einen JUnit-Test sieht wie folgt aus:

Listing 2.1: Beispiel für einen einfachen JUnit-Test

```
1  public void TestClass {
2
3      private int i;
4
5      @BeforeClass
6      public static void initialize() {
7          i = 3;
8      }
9
10     @Test
11     public void testSometing() {
12         assertEquals(3,i);
13     }
14 }
```

Wie wir sehen bietet JUnit (hier in der Version 4) per Annotation die Möglichkeit anzugeben in welchem Prozessschritt bei der Ausführung welcher Code ausgeführt werden soll. So werden Methoden die mit *@Before* bzw. *@After* annotiert sind vor bzw. nach jeder Test-Methode ausgeführt. Eine Test-Methode ist mit *@Test* markiert. Innerhalb dieser Testmethode wird der zu testende Code ausgeführt und die entsprechende Erwartungshaltungen formuliert. Die Erwartungshaltungen werden durch statische Hilfsfunktionen wie z.B. *assertEquals* ausgedrückt, die an der Hilfsklasse *Assertion* zu finden sind. Hier finden sich beispielsweise Funktionen um zu prüfen, ob zwei Eingaben gleich sind oder nicht oder ob sie leer oder gleich *null* sind. Das Testframework zeichnet sich vor allen Dingen durch seine Leichtgewichtigkeit und einfache Benutzung aus. Für das Schreiben eines einfachen Tests benötigt man in der Regel nur die Annotation *@Test* und die Einbindung der Assertion-Hilfsklasse.

Diese leichte Verwendbarkeit wollte ich auch für mein Testframework realisieren.

Die Ausführung der Tests findet mittels eines „Test-Runners“ statt welcher die Testergebnisse bereitstellt. Entwicklungsumgebungen wie z.B. Eclipse oder IntelliJ Idea können diese Ergebnisse graphisch ausgeben. Aber auch Build-Tools wie Maven oder Ant können die Ergebnisse des

[3] http://www.junit.org/

TestRunners auswerten um entsprechend mit dem Build-Prozess fort zu fahren. In Verbindung mit einem Werkzeug für „Continuous Integration“ (siehe auch [9]) wie Hudson [4] oder Teamcity [5] können die Testergebnisse so aufbereitet werden, dass man statistische wie auch historische Aussagen über die Testverläufe machen kann. Diese Informationen sind vor allen Dingen für größere Projekte imminent, da man andernfalls Gefahr läuft auf Dauer eine Testsuite zu entwickeln die zum Beispiel zu lange braucht um alle Tests auszuführen. Aber auch für verteilte Teams was auch auf das Team des Saros-Projektes zutrifft könnte ein solch zentraler Ort an dem die Testergebnisse zu finden sind vielerlei Vorteile bieten.

Um sich dieser vielen Möglichkeiten mit meinem Testframework bedienen zu können habe ich JUnit als Grundlage für dieses verwendet.

[4] Hudson Continuous Integration - http://hudson-ci.org/
[5] JetBrains Teamcity - http://www.jetbrains.com/teamcity/

### 2.5.2 Test-Objekte

Häufig kommt es beim Schreiben von Tests vor, dass man eine Funktionalität testen möchte die allerdings als Eingabe andere fertige Objekte benötigt um überhaupt ausgeführt werden zu können und vielleicht auch noch bestimmte Werte erwartet um überhaupt den zu testenden Effekt zu erhalten. Ein Beispiel für eine Methode die ein solches Objekt benötigt wäre folgendes:

Listing 2.2: Beispiel für Objekt (connection) welches für die Verwendung der zu testenden Klasse benötigt wird.

```
1 public class TestClass {
2     @Test
3     public void test() {
4         ...
5         MessageSender service = new MessageSender(connection);
6         service.sendMessage("Testnachricht");
7     }
8 }
```

Um diesen Test zum laufen zu bekommen müsste man nun das Objekt *connection* im Voraus erzeugen und entsprechend initialisieren. Es lässt sich leicht vorstellen, dass die Initialisierung dieses notwendigen Objektes mehr Zeit in Anspruch nehmen kann als das Schreiben des eigentlichen Tests.

Um also hier schnell Abhilfe zu schaffen an ein Objekt zu gelangen welches den Anforderungen des *MessageService* genügt gibt es verschiedene Möglichkeiten auf die man zurückgreifen kann. Im allgemeinen spricht man hierbei von einem Test Double welches die Funktionalitäten des eigentlichen Objektes nachahmt

In [10] identifiziert Gerard Meszaros vier verschiedene Arten von Test Doubles:

1. Dummy-Objekte: Dienen allein für die Übergabe als Parameter; keinerlei Verhalten definiert.
2. Stub-Objekte: Sammeln Werte, die die zu testende Funktionalität an dem Test-Objekt setzen möchte.
3. Mock-Objekte
4. und Fake-Objekte

Auf die beiden letzteren möchte ich im folgenden nun nähre eingehen, da diesen Konzepten während meiner Arbeit eine besondere Rolle zu Teil kam.

#### 2.5.2.1 Mock-Objekte

Mock-Objekte sind Objekte die eine Schnittstellen also im Falle von Java ein Interface implementieren aber zu aller erst nicht weiter genau wissen wie sie sich beim Aufrufen einer Methode verhalten sollen. Man muss also für bestimmte Eingaben entsprechende Rückgabewerte definieren. Auch für das Erstellen von Mock-Objekten gibt es Frameworks auf die man zurückgreifen kann. Im Projekt Saros wird das bekannte Mock-Framework EasyMock verwendet.

Wird an dem Objekt eine Methode aufgerufen, deren Verhalten im Voraus nicht definiert wurde, so erhält man bei der Ausführung eines solchen Tests eine Fehlermeldung darüber, dass der Aufruf der Methode nicht erwartet war. Somit ist im groben sichergestellt, dass bei der Ausführen eines Tests das Objekt sich an den entsprechenden Stellen gewollt verhält.

Um also auf eine Eingabe entsprechend mit einem Mock-Objekt zu reagieren muss man definieren was es für die Eingabe zurückliefern soll.

Listing 2.3: Erstellen eines Mock-Objektes und Definition der Rückgabe als Antwort für den Aufruf der Methode getName.

```
1 IUser mockObject = createMock(IUser.class);
2 expect(mockObject.getName()).andStubReturn("Hans");
3 replay(mockObject);
4 assertEquals("Hans", mockObject.getName());
```

Listing 2.4: Beispiel für die Definition einer komplexeren Rückgabe via IAnswer.

```
1  IUserService mockObject = createMock(IUserService.class);
2
3  expect(mockObject.createNewUserWithName((String) anyObject()))
4      .andStubAnswer(new IAnswer<IUser>() {
5          public IUser answer() throws Throwable {
6          Object[] args = EasyMock.getCurrentArguments();
7          String name = (String) args[0];
8          IUser mockObjectUser = createMock(IUser.class);
9          expect(mockObjectUser.getName()).andStubReturn(name);
10         replay(mockObjectUser);
11         return mockObjectUser;
12     }});
13 replay(mockObject);
14 IUser user = mockObject.createNewUserWithName("Hans");
15 assertEquals("Hans", user.getName());
```

Gerade am letzten Beispiel kann man erahnen, dass über die Zeit hin die Definition solcher Mock-Objekte immer umfangreicher und unübersichtlicher werden kann. Vor allen Dingen wenn man ein Mock-Objekt an verschiedenen Stellen verwenden und zudem vielleicht auch noch dynamisch erzeugen möchte kann es schnell dazu kommen, dass diese immer öfter nicht den Voraussetzungen gerecht werden können die man für die Ausführung eines Tests benötigt [6].

[6] Ähnlich ist mir bei meiner Arbeit ergangen.

#### 2.5.2.2 Fake-Objekte

Für immer komplexere Komponenten die sich entsprechend der Werte anders verhalten sollen und deren Initialisierung aufwändiger ist bietet sich die Verwendung von sogenannten Fake-Objekten an. Im Vergleich zu Mock-Objekten die im allgemeinen dafür vorgesehen sind bestimmte Werte zu liefern, um entsprechende Ausführungsstränge der zu testenden Funktionalität zu bedienen handelt es sich hierbei um echte Implementierungen die sich entsprechend einer Schnittstelle verhalten.

Für die ideale Trennung zwischen Test-Code und Programmcode sollten die Klassen die man für Testimplementierungen verwenden möchte durch Interfaces abstrahiert sein. Diese bieten einem die komfortable Möglichkeit Test-Klassen zu erstellen, die die vorgegebenen Funktionalitäten implementieren können wie es nötig ist. Möchte man ein Fake-Objekt für eine Klasse oder abstrakte Klasse erstellen ist man womöglich dazu gezwungen Funktionalitäten dieser zu verwenden.

Um sich auf die Bereiche konzentrieren zu können die man für die erfolgreiche Ausführung des Tests benötigt sollte man für alle Methoden die einfachste Standardimplementierung wählen. Hier bietet sich es an das Verhalten von Mock-Objekte nach zu ahmen und bei der Verwendung einer nicht implementierten Methode eine *Exception* zu werfen. In Java bieten sich hier die *RunTimeException* an, da man diese erzeugen kann ohne gezwungen zu sein die Schnittstelle der Methode anzupassen. Ein Beispiel für eine solche Klasse wäre folgendes:

Listing 2.5: Simpelste Implementierung eines Fake-Objektes.

```
1 public interface Connection {
2     public String executeCommand(String command);
3 }
4
5 public class ConnectionFake implements Connection {
6     public String executeCommand(String command) {
7         throw new NotYetImplementedException();
8     }
9 }
```

Benötigt die zu testende Logik die Methode *executeCommand* dann sollten wir hier nun eine geeignete Implementierung hinzufügen, die das entsprechende Verhalten geeignet anbietet.

Je nachdem welchen Fokus die Testimplementierungen haben könnten, sollte diese Implementierung so allgemeingültig wie möglich sein, um diese vielleicht später für andere Tests verwenden zu können. Überraschender Weise konnte ich so eine meiner Testimplementierungen sehr flexibel an vielen Stellen wiederholt zum Einsatz kommen lassen.

Auch wenn es scheinen mag, dass man hier unnötiger Weise Logik nachahmt die man doch eigentlich direkt benutzen könnte so bietet dieses Vorgehen vielerlei Vorteile. Denn wäre die Ausführung eines Tests auf eine Datenbank oder einen anderen Dienst angewiesen würde dies die Einrichtung der Entwicklungsumgebung erschweren, was womöglich Auswirkungen auf die Zahl der Tests hat die dann geschrieben werden. Auch nicht zu verachten sei hier der Vorteil in der Ausführungsgeschwindigkeit. So können wir bei einer einfachen Implementierung auf viele Aspekte verzichten die womöglich sonst zu längeren Laufzeiten der Tests führen würden.

Also eine solche Testimplementierung bietet vielerlei Vorteile. Sie ist universeller einsetzbar und zudem flexibler bei der Erfüllung gewisser Voraussetzungen von Tests. Allerdings ist diese Art der Bereitstellung von Test-Objekten initial wesentlich aufwändiger.

#### 2.5.2.3 Beispiel für nützliche Testimplementierung

Für Tests die auf Eclipse-Funktionalitäten zurückgreifen wäre es wünschenswert wenn es bereits eine Testimplementierung gäbe.

Mit sogenannte „JUnit Plug-In Tests“ [7] als erweiterte JUnit-Tests ist es möglich vor der Ausführung eines Tests dafür zu sorgen das die notwendige Umgebung vollständig initialisiert ist. Hierzu gehören zum Beispiel der GUI-Komponenten wie der Workbench oder die Einrichtung eines temporären Workspace. Die Nutzung solcher Plug-In Tests bietet sich bei der Entwicklung von Eclipse-Anwendungen gerade zu an, denn die Initialisierung der Umgebung ist zum einen sehr umfangreich und zum anderen aber oft gar nicht möglich. Im Projekt Saros fand dieser Ansatz schon Gebrauch.

Das Eclipse-Framwork selbst bietet beispielsweise keine nutzbare Implementierung für den Workspace an. So ist es vorgesehen, dass man an einen solche nur über den Aufruf von statischen Funktionen wie die folgende gelangt:

Listing 2.6: Statische Eclipse Funktion für die Rückgabe des aktuellen Workspace.

```
1 ResourcesPlugin.getWorkspace()
```

Das Framework bietet selbst keine anderer Möglichkeit einen Workspace zu erstellen und diesen anwendungsweit verwenden zu können. Die Verantwortlichkeit hierfür wurde klar dem Framework zugeteilt.

Die Verwendung von JUnit Plug-In Tests schloss sich für meine Arbeit aus, da ich mir beispielsweise vorstellte mehr als nur eine Saros-Instanz zu erstellen und es aber unter Verwendung eines solche JUnit Plug-In Tests trotzdem nur ein Workspace zur Verfügung steht. In diesem Fall war also eine Test-Implementierung notwendig.

[7] http://help.eclipse.org/help33/index.jsp?topic=/org.eclipse.hyades.test.doc.user/tasks/tcreatePluginTest.htm

### 2.5.3 Domain-Specific Language

Eine wichtige Bedingung damit automatisierte Tests tatsächliche Mehrwert liefern ist deren Verständlichkeit bzw. Nachvollziehbarkeit. Ist aus dem Test nicht gut zu erkennen wie dieser die Anwendungslogik verwendet so kann man dessen Intention auch nicht nachvollziehen. Allerdings hängt die Lesbarkeit der Tests nicht nur von diesen selbst ab sondern auch von dem zu testenden Programmcode.

Gerade für die Verständlichkeit von automatisierten Tests ist es so wichtig aus dem Lesen das Programmablaufs erkennen zu können was passiert. Andernfalls müsste man um einen Test verstehen zu können, Dokumentation nachlesen oder gar den Programmablauf im Detail nachvollziehen.

Es ist also wichtig, dass man beim Lesen der Tests sich erschließen kann was passiert. Verständliche Tests sollten auch zu besser lesbarem Produktcode führen.

Tests haben aber noch einen weiteren Vorteil im Bezug auf die Struktur des zu testenden Programmcodes. Tests können auch strukturell Anwendungen erläutern in dem sie auf höherer Ebene Programmcode verwenden. Konzeptionell handelt es sich hierbei um Funktionstest. Der Vorteil dabei ist, dass man eine andere Perspektive zu dem Programmcode einnimmt und ähnlich wie man den Test auf abstrakter Ebene gestalten will der Code sich auf dieser dazu ausdrücken muss.

Listing 2.7: Abstrakte Deklarative Formulierung eines Prozessschrittes

```
1 saros.sends().activity(a);
```

Möchte man diese Perspektive einnehmen könne so muss dies der Programmcode auch ermöglichen. Programmcode der auf solch abstrakter Ebene immer noch verständlich ist wird sich wahrscheinlich dem Vokabular der entsprechenden Domäne bedienen um somit die Intention einer Komponente oder der Kompositionen von Komponenten klar zu stellen. Eric Evans bezeichnet dies als Intention-Revealing Interface ([3]) Für die Hauptkompetenz von Saros könnte das zum Beispiel eine Sammlung von Vokabeln sein die das Versenden von Editor-Aktivitäten beschreiben.

Eine deklarative Form eines Intention-Revealing Interfaces ist zum Beispiel eine sogenannte interne Domain-Specific Language (DSL) ([6]). Eine solche ermöglicht es dem Entwicklern sich im Programmcode mit dem Vokabular der Domäne auszudrücken. Man betrachtet so auf abstrakterer Ebene was man genau mit der Entwicklung eine Funktionalität erreichen möchte und schafft außerdem Programmcode der lesbar und leicht verständlich ist. Hält man die Grenzen der DSL ein, kann so man auch die Belange von Mengen von Komponenten abschirmen um klar zu stellen, was in einen Bereich gehört und was nicht. Eric Evans bezeichnet dies als Bounded Context. Gerade dieser Aspekt sollte zu größeren architekturellen Entscheidungen führen. Hätte das Saros Projekt einen oder mehrere solcher abgeschlossener Kontexte geschaffen so hätte man sich womöglich besser gegenüber eclipse-spezifischer Implementierungen abschirmen können was zu besser testbarem Code geführt hätte.

Einer der extremsten Ausprägungen solch einer DSL ist es wenn man versucht die gesprochene Sprache im Programmcode abzubilden. Sogenannte Fluent Interfaces ([4]) bieten eine Schnittstelle für Funktionalitäten die es Benutzern dieser enorm erleichtern die richtigen Funktionalitäten zu finden oder auch richtig zu benutzen. Man kann solche Schnittstellen auf verschiedene Weise anbieten. In Java haben sich bislang die Verkettung von Methoden (Method Chaining) und Ver-

schachtelung von Funktionen (Nested Functions) etabliert. Beide Ansätze haben jeweils ihr für und wider. Die Verkettung von Methoden bietet eine Schnittstelle die Dank der Autovervollständigung von Entwicklungsumgebungen dem Entwickler stets den richtigen Weg weisen können. Solche Schnittstellen müssen Grammatiken realisieren um nicht falsch benutzt werden zu können. Bei der Verschachtelung von Methoden ist dieser Aspekt etwas schwieriger zu realisieren allerdings hat man hier im Vergleich zur Verkettung von Methoden den Vorteil besser erweiterbar zu sein.

Fluent Interfaces können wegen ihrer zu Grunde liegenden Grammatiken schnell sehr komplex werden was negative Auswirkungen auf die Erweiterbarkeit dieser haben kann. Man sollte von diesen Konzepten also immer besonders unter Bedacht Gebrauch machen, sodass andere diese erweitern können (siehe auch [2]).

### 2.5.4 Dependency Injection

Systeme die aus vielen Komponenten bestehen wie auch Saros, müssen für deren Ausführung initialisiert und mit einander verbunden werden. Listener müssen sich bei entsprechenden Diensten anmelden oder ganze Komponenten müssen anderen Komponenten zur Verfügung gestellt werden. In [15] bezeichnet man dies auch als Objektnetz.

Die einfachste Art dem aus dem Weg zu gehen ist es Funktionen zu verwenden die direkt die Funktionalitäten anbieten. Allerdings bedeutet dies, dass man mit diesen Funktionalitäten nur maximal einen Zustand halten kann.

Das folgende Beispiel soll skizzieren wie man beispielsweise eine Verbindung zu einem XMPP-Server der gesamten Anwendungslogik leicht zur Verfügung stellen könnte.

Listing 2.8: statische Klasse

```
1 private static XMPPConnection connection
2         = new XMPPConnection("serveradress");
3
4 public static void connect() {
5     connection.connect();
6 }
```

Möchte man allerdings mehrere Verbindungen auf diese Art verwalten, würde sich für das Beispiel die Änderung ergeben, dass man für die Verwendung alle nötigen Objekte mit übergeben müsste die für die Ausführung der Funktionalität notwendig sind. Auf Dauer würde dies zu langen Schnittstellenbeschreibungen einer Funktion führen.

Listing 2.9: Erweiterung der Funktion um mit verschienden XMPPConnections umgehen zu können.

```
1 public static void connect(XMPPConnection connection) {
2     connection.connect();
3 }
```

Im übrigen wurden statische Funktionen und Felder genau auf solche Art in der Programmlogik von Saros etabliert.

Um also besser strukturierten Code schreiben zu können der dem Konzept der Komponentisierung folgt, sollte man bei der Instanziierung von Komponenten bleiben.

Dependency Injection verallgemeinert die grundlegende Logik für die Initialiserung dieses Geflechts von Komponenten und den Zugriff auf dessen Teile. Durch diese Zentralisierung kehrt man die Kontrolle über die Initialiserung der Komponenten um (Inversion of Control), sodass man nur noch in den Komponenten die Abhängigkeiten zu anderen Komponenten definieren muss und sich darauf verlassen kann, dass diese richtig initialisiert sind.

Es gibt für dieses Konzept diverse Frameworks die alle eine andere Art haben wie man die Abhängigkeiten zwischen Komponenten zum Ausdruck bringen kann. Das Konzept besteht in den meisten Fällen aus zweierlei Schritten. Einmal der Definition der Komponenten und zum anderen dem Verweben dieser.

Im Projekt Saros wird zu Teilen von Dependency Injection Gebrauch gemacht. Damals hat man sich für das Framework PicoContainer [8] entschieden, welches alle gängigen Arten für die Injizierung von Komponenten anbietet. Im Falle von Saros hat man insbesondere von der Injektion via Konstruktor und Annotation Gebrauch gemacht.

Listing 2.10: Beispiel für Injizierung via Konstruktor.

```
1 public class SessionService {
2
3     private UserService _userService;
4
5     public SessionService(UserService userService) {
6         _userService = userService;
7     }
8 }
```

Listing 2.11: Beispiel für Injizierung via Annotation

```
1 public class SessionService {
2     @Inject
3     private UserService _userService;
4
5     public SessionService() {}
6 }
```

Damit der PicoContainer weiß von welchen Klassen Exemplare erstellt werden sollen, muss dieser noch entsprechend konfiguriert werden. Im Folgenden werde ich diese Menge und Verflechtung von Komponenten als Kontext bezeichnen.

Listing 2.12: Konfiguration des PicoContainers

```
1 PicoBuilder picoBuilder = new PicoBuilder(new AnnotatedFieldInjection());
2 MutablePicoContainer container = picoBuilder.build();
3 container.addComponent(UserService.class);
4 container.addComponent(SessionService.class);
```

Die Vorteile der Dependency Injection liegen auf der Hand. Möchte ein Entwickler solch einen Dienst wie den *SessionService* erstellen für dessen Funktionalität er auf die Dienste eines *UserService* zugreifen muss, so muss dieser lediglich nur noch via Annotation zum Besipei angeben, dass der *SessionService* eine Instanz von einem *UserService* benötigt. So hat man selbst keinen Aufwand damit die Komponenten an der richtigen Stelle richtig zu erstellen. Das reduziert sich wiederholenden Programmcode und macht das System weniger anfällig für Fehler. Wie wir sehen werden, wird die Verwendung des PicoContainers ein Schlüsselelement dieser Arbeit sein.

[8] http://picocontainer.org/

# 3 Konkretisierung/Abgrenzung

## 3.1 Automatisierte Tests in Saros

Für einige Teile von Saros existierten bereits Unit- wie auch Funktsionstests die sich vieler der oben erwähnten Konzepte bedienen. So wurden beispielsweise Mock-Objekte verwendet oder auch Fake-Objekte [1].

Ein gutes Beispiel für einen simplen Unit-Test ist die Test-Klasse *UnderstandingPico*. Dieser Test dient allein zum Sammeln von Testfällen die für das Verständnis des Verhaltens des PicoContainers dienen.

Besonders hervor zu heben sind aber auch die Tests für den Jupiter-Algorithmus für den verhältnismäßig viele Tests geschrieben wurden. Da es sich hierbei um eine der zentralsten Funktionalitäten von Saros handelt ist dies auch nachvollziehbar. Für diese Tests wurde sogar eine eigene Klasse zur Simulation des Netzwerks erstellt (*NetworkSimulator*). Am Rande sei hier nur erwähnt, dass es sich dabei um eine spezielle Anfertigung für den Jupiter-Algorithmus handelt und somit nicht in Frage kam für mein Vorhaben.

Die Test-Klasse *SimpleClientServerTest* ist ein gutes Beispiel für einen abstrakten Test den man heranziehen kann, wenn man die Funktionalität des Jupiter-Algorithmus verstehen möchte.

Listing 3.1: Beispiel für nachvollziehbaren und gut lesbaren Test

```
1  @Test
2  public void testDeleteStringWithConcurentInsert() {
3      setupClientServer("abcdefg");
4
5      client.sendOperation(new InsertOperation(3, "x"), 100);
6      server.sendOperation(client.getJID(),
7          new DeleteOperation(1, "bcde"), 400);
8
9      network.execute(200);
10     assertEquals("abcxdefg", client.getDocument());
11
12     network.execute(400);
13     assertEqualDocs("axfg", client, server);
14 }
```

[1] z.B. *SarosSessionStub* - zwar wurde im Projekt diese Klasse als Stub bezeichnet allerdings handelt es sich nach der Definition von Gerard Meszaros um ein Fake-Objekt da es auch Logik implementiert die direkt auf Methodenaufrufe mit entsprechendem Verhalten antwortet

### 3.1.1 STF

Aber auch für Integrationstests lag bereits eine Lösung vor. Das von S. Szücs erstellte Testframework - kurz STF - ([14]) ist ein Framework welches speziell für Saros erstellt wurde um es zu ermöglichen Integrationstests zu erstellen die die Benutzerschnittstelle (GUI) wie ein Anwender benutzen. Im Folgenden werde ich diese Art von Tests also Frontend- oder GUI-Tests bezeichnen. Der Vorteil dieser Lösung im Vergleich zu den übrigen Unit-Tests ist, dass man hier Tests erstellen kann die mit mehr als einer Saros-Instanz arbeiten. Konzeptionell funktioniert das so, dass eine oder mehrere Instanzen durch die Tests ferngesteuert werden. Mittels eines RMI-Bots können so Kommandos an die Testinstanzen übermittelt werden, die direkt Aktivitäten auf der GUI auslösen und somit die gesamte Logik über alle Schichten hinweg von Saros zur Ausführung bringen.

Solche Tests können also die gleichen Handlungsschritte machen wie sie ein Anwender durch führen muss, um einen bestimmte Effekt zu erzielen. Man könnte so z.B. die Fehlerbeschreibung von Saros-Anwendern direkt in einem solchem Test formulieren, um das Versagen beheben zu können. Diese Art der Tests sind derzeit die abstrakteste Form solcher mit der man Saros-Logik ausführen kann.

Allerdings gibt es hier einen Aspekt der nicht zu verachten ist und das ist die Wartung dieser. Im Vergleich zu Unit-Tests die direkt Anwendungslogik aufrufen geschieht der Aufruf hier nur indirekt über die GUI und die Nachrichten die für die Ausführung der Kommandos versendet werden. Diese lose Kopplung hat zur Folge, dass man Funktionalitäten ändern kann ohne zu bemerken, dass man damit die Voraussetzungen für einen Test bricht. Das Versagen des Tests würde also erst nach dessen Ausführung auffallen und wäre zudem nicht unbedingt auf eine bestimmte Änderung zurückzuführen.

Der Aspekt das man Änderungen durchführen kann ohne das die Tests direkt davon schon bei der Kompilierung betroffen sind, ist ein grundlegendes Problem von Frontend-Tests. So kann man nicht vor der Laufzeit feststellen, ob ein bestimmter Button den man für die Ausführung eines Tests benötigt tatsächlich noch vorhanden ist. Der eigentliche Programmcode und die Testlogik können also ohne ständige Ausführung der Tests divergieren. Da bislang allerdings kein Continuous Integration System etabliert ist war dies im Verlaufe meiner Arbeit häufiger der Fall. Ein weiterer Nachteil von STF ist, dass die Test-Instanzen laufende Eclipse-Instanzen sind die wie schon erwähnt mittels eines RMI-Bot gesteuert werden. In diesem konkreten Fall laufen die Eclipse-Instanzen in eigenen virtuellen Maschinen. Man kann also derzeit nicht beliebig viele solche Testinstanzen ansteuern ohne das diese entsprechend eingerichtet und konfiguriert werden müssen. Außerdem ist für die Ausführung der Tests eine Netzwerkverbindung notwendig sofern man denn die vorbereiteten Test-Instanzen auf den virtuellen Maschinen verwenden möchte. Eine eigene Einrichtung dieser Testumgebung ist derzeit zu umfangreich als das sie jeder für seine Entwicklungsumgebung vornehmen würde.

## 3.2 Abgrenzung

Mein Testframework soll sich auf einen sehr konkreten Teil der Programmlogik konzentrieren - der sogenannte Business-Logik von Saros. Als Business-Logik zähle ich all die Funktionalitäten die für die Hauptkompetenz von Saros notwendig sind, ausgeschlossen der die für das Frontend notwendig sind - also alle Logik die für das Erzeugen, Austauschen und Verarbeiten von Aktivitäten verantwortlich ist.

Im Vergleich zu GUI-Tests soll man mit Hilfe meines Frameworks Unit-Tests erstellen, die direkt die Funktionalitäten der Anwendungslogik ausführen. Was zur Folge hat, dass wenn ein Entwickler eine Schnittstelle ändern oder entfernen möchte er schon vor der Ausführung der Tests durch die Kompilierung darüber informiert wird, dass es bislang gewisse Erwartungen an die Methode gibt. Der Aufwand die Tests mit der Anwendungslogik im Einklang zu halten ist wesentlich leichter gegenüber GUI-Tests. Idealerweise sollten die Tests keine weitere Konfiguration des Entwicklers bedürfen um ausgeführt zu werden, denn um so geringer die Schwelle für das Schreiben eines Testes ist um so höher sollte auch die Bereitschaft sein einen neuen Test zu schreiben.

Hinzu kommt, dass Unit-Tests wesentlich schnellere Ausführungsszeiten haben sollten als Tests die zusätzlich Leistung aufwänden um die GUI darzustellen. Je schneller die Tests durchlaufen können desto schneller kann dem Entwickler ein Feedback darüber gegeben werden was für Auswirkungen seine Änderungen hatten.

Ein letzter Unterschied zu GUI-Tests ist die Nachvollziehbarkeit der Abläufe die durch den Test angestoßen werden. So sind Abläufe die sich durch Aktivitäten auf der GUI ergeben wesentlich umfangreicher und komplexer als wenn man direkt eine ganz bestimmte Funktionalität aus einem Test heraus startet. Die GUI erzeugt beispielsweise Ereignisse auf die andere Komponenten hören und somit zu Seiteneffekten führen könnten die schwer nachvollziehbar sind.

# 4 Durchführung

Die vorigen Erläuterungen sollten also klargestellt haben, dass ich ein Testframework erstellen wollte welches es ermöglicht schnell und leicht, neue gut lesbare und verständliche Unit-Tests für die Business-Logik von Saros zu erstellen, die keinerlei Konfiguration erfordern sollten und zudem schnellläufig sind. Außerdem wollte ich eine DSL etablieren die abstrakt genug ist, um mit dem Framework umzugehen und eventuell besser auf Saros Logik zurückgreifen zu können.

Dabei gab es mehrere Schwierigkeiten zu bewältigen die vor allen Dingen auf die Architektur von Saros und der Verwendung des Eclipse-Frameworks und einer Bibliothek als Implementierung des XMPP-Protokolls zurückzuführen waren.

## 4.1 Prämissen

Bei der Umsetzung gab es von vornherein Prämissen die Grundlage für einige wichtige strukturelle Entscheidungen waren. Man hätte für die Realisierung der Testbarkeit von Saros mehrere Wege beschreiten können. Zum Beispiel hätte man die Programmlogik soweit umstrukturieren können, bis man die gewünschten Tests hätte schreiben können.

Allerdings wäre dieser Umbau, soviel sei schon vorweggenommen, nicht leicht gewesen. Die Gefahr das mir als Einsteiger in dem Projekt bei der Restrukturierung Fehler unterlaufen wäre zu groß gewesen. Hätte man eine Testsuite gehabt mehr Programmlogik bereits abdeckt hätte man dies sicher in Betracht ziehen können. Es lag also nahe, dass ich mir bei diesem Vorhaben diese Testsuite selber baue und einen anderen Weg beschreite. Die wichtigste Prämisse meines Vorgehens war es, wenig bis keine Änderungen an der aktuellen Anwendungslogik vorzunehmen und es möglich zu machen Tests zu schreiben die das aktuelle Verhalten des Programmcodes festhalten, fixieren. Somit stünde später einem Refactoring nichts mehr im Wege.

Eine andere Prämisse war es, dass mein Framework die wichtigsten Funktionalitäten einfach und verständlich anbietet und zudem leicht erweiterbar ist. Für ersteres gibt es viele Möglichkeiten dies umzusetzen ich denke ich habe eine Möglichkeit gefunden die es einem neuen Benutzer des Frameworks leicht machen einzusteigen und schnell an das zu kommen was für das Schreiben eines Unit-Tests benötigt. Der zweite Punkt der leichten Erweiterbarkeit führte vor allen Dingen zu Entscheidungen die mehrfach die Struktur meiner DSL verändert haben und neue Lösungen hervorbrachte. Einige Ideen habe ich somit bewusst nicht mit einfließen lassen, um so die Etablierung des Framworks zu ermöglichen.

Eine andere Prämisse war es, dass die Tests die mit meinem Testframwork entstehen leicht verständlich sein sollten. Dieser Aspekt zählt auf die Etablierung einer DSL ein die quasi ein Extrakt der wichtigsten Begriffe aus der Saros-Logik sein sollte. Sie soll vor allen Dingen dazu dienen, dass die Benutzer des Frameworks schnell die Funktionalitäten finden die sie suchen und aus dem Lesen von Tests erkennen können, was bei der Ausführung der bereitgestellten Methoden passiert.

Außerdem sollten für die Ausführung der Test keine externen Abhängigkeiten entstehen oder gar die Konfiguration des Entwicklers vorausgesetzt.

Zu guter Letzt sollte die Laufzeit der Test überschaubar bleiben um sicher zu stellen, dass diese so oft es geht ausgeführt werden.

Zusammenfassend wollte ich mich also an folgende Prämissen halten:

1. keine Änderungen an bestehendem Code
2. keine externen Abhängigkeiten für die Ausführung
3. die Tests sollen schnellläufig sein
4. die Tests sollen gut verständlich/lesbar sein
5. notwendige Funktionalitäten für das Schreiben von Tests soll man möglichst einfach wieder finden können.

## 4.2 Die Idee

Für die Realisierung meines Vorhabens hatte ich bereits grob eine Vorstellung die vor allen Dingen im Einklang mit den oben genannte Prämissen war. Die größte Herausforderung sah ich darin die Tests von keinem externen XMPP-Server abhängig zu machen. Ich zog es also in Betracht die Netzwerkschicht von Saros (sofern denn eine solche existiert) durch eine eigene Implementierung zu ersetzen. Des Weiteren sah ich schon im Voraus auf mich zu kommen, dass ich die Ausführung von eclipse-spezifische Funktionalitäten während eines Test deaktivieren müsste.

In wieweit ich diesen groben Vorstellungen folgen konnte soll folgende Analyse zeigen.

## 4.3 Analyse

Bevor ich mit meinem Vorhaben beginnen konnte, musste ich mir einen Überblick über die Struktur des Programmcodes von Saros machen um feststellen zu können wie ich mein Zielsetzung am ehesten erreiche.

Wie schon erwähnt war es ein Gedanke die Netzwerkschicht von Saros durch eine Testschicht zu ersetzen um so das Vorhandensein eines XMPP-Servers nicht für die Ausführung der Tests voraussetzen zu müssen.

Damit wir jedoch überhaupt eine Funktionalität durch eine andere ersetzen können Bedarf es einer Architektur die dies erlaubt. Für Saros könnte man sich beispielsweise folgende Schichten Architektur vorstellen.


Abbildung 4.1: Mögliche Schichtenarchitektur von Saros

In groben kann man eine solche Schichtenarchitektur in der jetzigen Code-Struktur des Projektes erahnen. Allerdings musste ich auf viele Stellen stoßen an denen diesen Schichtenarchitektur unterlaufen wurde.

Für die Netzwerk-Kommunikation hat man sich im Projekt eine Bibliothek Name Smack von Ignite Realtime zu Nutze gemacht. Bei Smack handelt es sich um eine Java-Implementierung des XMPP-Protokolls. Diese bietet alle nötigen Funktionalitäten um mittels Java sich auf einem XMPP-Server einzuloggen oder Nachrichten mit Teilnehmern eines Chats auszutauschen. Abgesehen von den Protokollerweiterungen von XMPP durch Saros ist Smack im Grunde die gewünscht Netzwerkschicht. Allerdings muss man nach weiteren Analysen feststellen, dass diese Bibliothek sehr stark mit Saros verwoben ist was das Ersetzen dieser Schicht nicht trivial machte und zudem nicht mit der ersten meiner Prämissen in Einklang gebracht werden konnte. Die Anwendungslogik von Saros machte über alle Schichten hinweg von Smack-Funktionalitäten Gebrauch. Gerade die beiden Entitäten *Connection* und *Roster* wurden an verschiedenster Stelle verwendet, um beispielsweise auf die Verfügbarkeit von Teilnehmern zu hören oder auch direkt Visualisierungen dieser wie z.B. dem Roster (siehe *RosterView*) anzubieten.

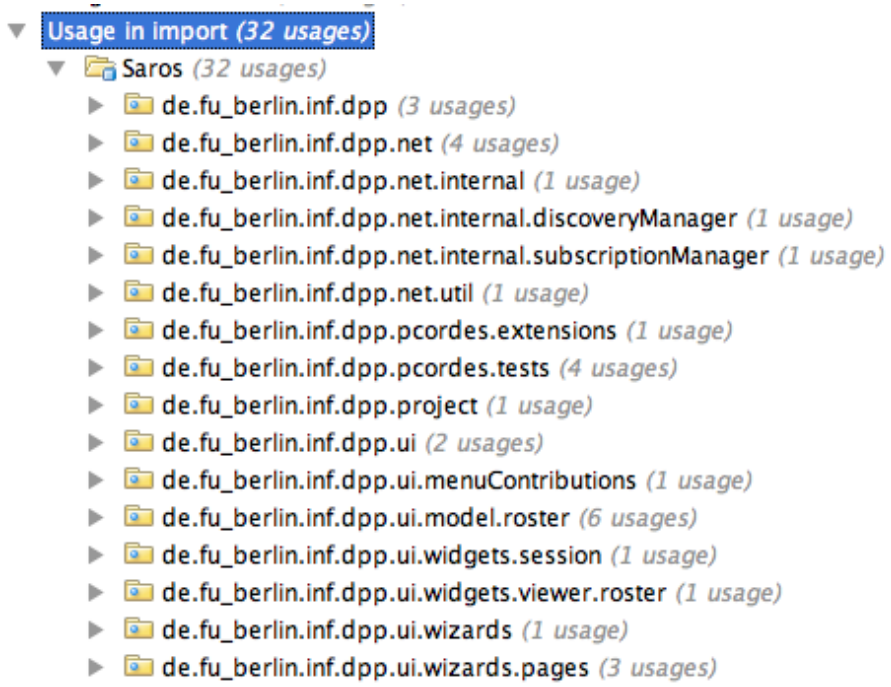


Abbildung 4.2: Die Klasse Roster wird in allen Schichten von Saros verwendet.

Wie ich finde ist die *RosterView* ein gutes Beispiel dafür, dass man offenbar das Vokabular von XMPP respektive Smack mit in die Projektsprache also die eigene DSL aufgenommen hat. Ich sehe hierin einen der entscheidenden Gründe für die unzureichende Trennung zwischen Saros und Smack. Hätte man Saros nicht als direkte Erweiterung von XMPP betrachtet sondern XMPP viel mehr nur als Mittel zur Übertragung von Aktivitäten, dann hätte man vielleicht diese Bibliothek besser gekapselt und sich dafür entschieden ein eigenes Vokabular für die benötigten Entitäten oder Dienste zu entwickeln. So wäre es vielleicht zu Entwicklung einer eigenen User-Entität gekommen die viel eher eine sinnvolle Verwendung in der Anwendungslogik gefunden hätte als die der Smack-Bibliothek. Man hätte Saros somit losgelöst von XMPP-spezifischen Aspekten entwickeln können.

### 4.3.1 Eclipse-spezifische Funktionalitäten

Funktionalitäten aus der Schicht der Benutzeroberfläche sollten durch die angestrebten Unit-Tests nicht ausgeführt werden. Für diese Art der Tests gibt es bereits das eigens dafür entwickelte Testframework - STF. Aber auch hier musste man schnell feststellen, dass es nicht ohne Weiteres möglich war die Anwendungslogik vollständig ohne diese auszuführen.

Häufig finden sich Stellen an denen direkt auf Eclipse-eigene Funktionalitäten zurückgegriffen wird, was eine direkte Abhängigkeit der entsprechenden Komponente zum Eclipse-Framework zur Folge hat. Das hätte keine großen Konsequenzen, wenn man denn als Benutzer des Frameworks die Möglichkeit hätte diese Funktionalitäten durch eigene zu ersetzen. Wie bereits in Abschnitt 2.5.2.3 erläutert ist dies allerdings nicht möglich. Eines der besten Beispiele ist der Zugriff auf Projekte und deren Inhalte im Workspace. Üblicherweise kann man auf Projekt durch Aufruf folgender Funktionen und Methoden zugreifen.

Listing 4.1: Übliche Herangehensweise um an ein Projekt im Workspace zu gelangen.

```
1 IProject project = RessourcePlugin
2     .getWorkspace()
3     .getRoot()
4     .getProject("NameOfProject");
```

Wie wir hier allerdings sehen müssen wird hierzu eine statische Funktion benötigt die uns direkt den Workspace zurückliefert. Die Verwendung solcher Funktionen hat den Nachteil das man sie nicht überschreiben oder anders ersetzen kann. Des Weiteren haben wir in diesem Fall keine weiteren Möglichkeiten auf einen anderen Workspace zuzugreifen was so viel bedeutet, dass einer laufenden Java-Anwendung genau dieser eine Workspace zur Verfügung steht. Das ist eine Bedingung die das Eclipse-Framework vorgibt und zudem einer der Bedingungen die dazu führen das daraus Programmcode entsteht der im Falle des Saros-Projektes schlecht testbar ist.

```
java.lang.IllegalStateException: Workspace is closed.
    at org.eclipse.core.resources.ResourcesPlugin.getWorkspace(ResourcesPlugin.java:340)
    at de.fu_berlin.inf.dpp.pcordes.tests.TempTest.testMock(TempTest.java:66) <3 internal calls>
    at org.junit.runners.model.FrameworkMethod$1.runReflectiveCall(FrameworkMethod.java:44)
    at org.junit.internal.runners.model.ReflectiveCallable.run(ReflectiveCallable.java:15)
    at org.junit.runners.model.FrameworkMethod.invokeExplosively(FrameworkMethod.java:41)
    at org.junit.internal.runners.statements.InvokeMethod.evaluate(InvokeMethod.java:20)
    at org.junit.internal.runners.statements.RunBefores.evaluate(RunBefores.java:28)
    at org.junit.internal.runners.statements.RunAfters.evaluate(RunAfters.java:31)
    at org.junit.runners.BlockJUnit4ClassRunner.runChild(BlockJUnit4ClassRunner.java:73)
```

Abbildung 4.3: Fehlermeldung die man erhält wenn man einen normalen Unit-Test ausführt und dabei statische Eclipse-Funktionen verwendet werden.

Aus damaliger Sicht schien sicherlich die Verwendung dieser Methoden valide. Heute mit einer testbasierten Perspektive würde man sicher diese Funktionalitäten besser kapseln und in einer separate Komponente bündeln. So ist allerdings über die Zeit ein Konstrukt entstanden das schwer bis gar nicht ohne Weiteres mit Hilfe von JUnit Tests testbar war.

Sollte man in Zukunft vorhaben die Struktur des Programmcodes zu überdenken, so sehe ich eine ausgeprägte Testsuite als eine Grundvoraussetzung dafür diese sicher durchführen zu können. Ohne Tests müsste man äußerst behutsam vorgehen. Man hätte nie Gewissheit ob die Kernfunktionalitäten noch so funktionieren wie es erwartet wird. Nach jeder Änderung hätte man mittels manueller Tests prüfen müssen, ob sich etwas ungewünscht geändert hat. Mit einer hinreichenden automatischen Testsuite hätte man ein Gerüst um solche Umbauten sicherer und weniger zeitintensiv durchführen zu können.

### 4.3.2 Netzwerkschicht ersetzen

Wie wir bereits festgestellt haben war es nicht ohne Weiteres möglich die Netzwerkschicht durch eine Testimplementierung zu ersetzen denn Smack als die eigentliche Netzwerk-Schnittstelle wurde nicht gekapselt. Die Verflechtung der Bibliothek ist so stark, dass nicht möglich schien ohne große Veränderung (also Verletzung der obersten Prämisse) des Codes diese zu ersetzen. Zu viel wurde direkt auf Funktionalitäten von der Bibliothek an verschiedensten Stellen zurückgegriffen. Es schien also nicht möglich ohne große Änderungen hier eine Lösung zu finden mit der man die Kontrolle über das Übermitteln von XMPP-Nachrichten erlangen könnte.

Ich versucht also eine andere Lösung zu finden die nicht in der Anwendungslogik lokalisiert ist, sondern außerhalb von dieser - einer eigenen Server Lösung. Die Idee war es einen XMPP-Server bereitzustellen, der vor der Ausführung der Tests lokal automatisch gestartet wird und somit den Test-Instanzen als Netzwerkschicht dienen kann. Der Vorteil war, der dass ich so keinerlei Anwendungslogik ändern musste [1].

[1] In der konzeptionellen Phase meines Vorhabens gab es auch die Idee die Netzwerkschicht konfigurierbar zu machen um beispielsweise Szenarien nachzustellen in denen Paket verzögert oder gar nicht zugestellt werden. Auch mit der Verwendung eines solchen Servers wäre dies möglich.

### 4.3.3 Wahl einer Implementierung

Da bereits Implementierungen von XMPP-Servern existieren habe ich also nur eine Wahl über eine geeignete Lösung treffen müssen.

Bei der Recherche nach fertigen Implementierung stellte ich folgende Voraussetzungen an, die das diese erfüllen müsste um überhaupt in die nähere Betrachtung zu kommen.

Die Implementierung sollte in Java sein, um den oben genannte Punkt der Konfiguration durch die Tests so einfach möglich gestalten zu können. Einen Server zu wählen der nicht der Programmiersprache von Saros entspricht hätte die Entwicklung einer Schnittstelle verlangt und würde zudem den Entwicklern des Projekte mehr Fähigkeiten abverlangen.

Die Implementierung sollte Open Source sein und mit der Lizenz von Saros verträglich sein [2].

Bei der Recherche kamen zwei Java Implementierungen in die enge Auswahl:

1. Openfire (ehemals Wildfire) von Ignite Realtime [3]
2. Tigase [4]

Folgende sehr grobe Kriterien haben dazu beigetragen, dass ich mich für Openfire entschieden habe:

> Im Programmcode von Saros wird die Smack-API verwendet, welche von dem selben Unternehmen (Ignite Realtime) veröffentlicht wird. Ich erhoffe mir hiervon unter anderem auch Vorteile im Bezug auf die Kompatibilität.
>
> Hinter der Entwicklung des Servers steht ein Unternehmen, was darauf schließen ließ das Sichtbarkeit und Pflege des Projektes besser sind. Dies bestätigte sich unter anderem bei der Betrachtung der beiden Versionskontrollsysteme der Projekte.
>
> Im SVN von Tigase tauchen in im vergangenen halben Jahr nur zwei verschiedene Autoren auf. Bei Openfire sind es deutlich mehr verschiedene Autoren. Die Revisionsnummer von Tigase lag bei bei ca. 2400 die von Openfire bei über 12.000 (stand November 2010). Desweitern existiert das SVN von Openfire (seit 2004) länger als das von Tigase (seit 2005).

Schlussendlich kann man hier die Aussage treffen, dass das Projekt Openfire länger existiert, von mehr Entwicklern gewartet wird und mehr Änderungen unterworfen ist was darauf schließen lassen könnte, dass dieses Projekt weiter fortentwickelt ist.

[2] Saros ist eine Open Source Software stehend unter der GNU General Public Licence Version 2
[3] http://www.igniterealtime.org/projects/openfire/
[4] http://www.tigase.org/

### 4.3.4 Umgang mit eclipse-spezifischen Funktionalitäten

Ähnlich bei Untersuchung der Netzwerkschicht galt es auch für die eclipse-spezifischen Funktionalitäten eine Lösung zu finden die deren Ausführung kontrolliert. Allerdings war es hier nicht wie bei Netzwerkschicht so leicht eine Lösung zu entwickeln die keine Änderungen an der Anwendungslogik erfordert. Wie schon weiter oben darauf hingewiesen wurde, hat man an vielen Stellen direkt auf Eclipse-Funktionalitäten zurückgegriffen die nicht ohne Weiteres ersetzbar sind. Um nicht unnötig Änderungen an Programmcode durch zu führen welcher nur über die GUI ausgeführt wird, habe ich die Betrachtung dieses Problems verschoben. Ich hoffte, dass ich durch mein testbasiertes Vorgehen einen besseren Fokus für diese Problematik entwickeln könne.

## 4.4 Testbasiertes Vorgehen

Da wir schon bei der Analyse des Programmcodes für die Ersetzung der Netzwerkschicht und der eclipse-spezifischen Funktionalitäten feststellen mussten, dass eine mögliche Schichtenarchitektur in dem Projekt oft gebrochen wurde und auch die Komplexität des Projektes einen Grad erreicht der für jemanden neues nicht überschaubar war, musste ich versuchen fokussiert vorzugehen um mich nur den Stellen zu widmen die für die Ausführung automatischer Unit-Tests von Belang waren.

Mein Vorgehen war dabei weitesgehend testgetrieben geprägt. Auch wenn der Programmcode in diesem Falle schon vorlag verwende ich trotzdem den Begriff des Treibens, da ich angestoßen durch einen Test ganz bestimmte Änderungen vorgenommen und Entscheidungen getroffen habe.

Bevor ich also Programmcode näher betrachtet habe, habe zunächst meine Erwartungen in einem möglichst einfachen und abstrakten Test beschrieben. In den meisten Fällen glückte die erste Ausführung dessen nicht. Es waren also für den erfolgreichen Verlauf des Tests Anpassungen bzw. Erweiterungen der Anwendungslogik notwendig. Der Vorteil dieses Vorgehens war unter anderem, dass ich mich allein auf die Stellen konzentrieren musste, die für diesen Test notwendig waren, was es mir ermöglichte nur Teile des Programmcode unter dem Aspekt der automatischen Testausführung zu betrachten.

Die Motivation den Test so schnell es ging zum erfolgreichen Verlauf zu bekommen verhalf mir zusätzlich an mancher Stelle zu Entscheidungen, die gerade den Fokus auf ein Problem noch einmal änderten.

Um also getrieben von Tests die Stellen zu identifizieren die einer Änderung bedürfen brauchte ich eine Strategie, einen Weg der dazu führen sollte, dass ich den Großteil aller relevanten Stellen und Aspekte aufdecken konnte. Ich erhoffte mir davon auch, dass ich auf diesem Weg mir selbst bereits geeignete Hilfsmittel bereitstellen würde die ich später in einer DSL verarbeiten könnte. Wie sich herausstellte war dies auch der Fall.

Technisch gesehen war es mein Ziel eine SarosSession [5] mit beliebig vielen Teilnehmern erstellen zu können um über diese Sitzung Aktivitäten unter den Beteiligten austauschen zu können. Nach kurzer Recherche der notwendigen Schritte hierfür habe ich mir folgende Tests überlegt:

1. Saros Test-Instanz erstellen
2. Saros mit XMPP-Server verbinden lassen
3. Zwei Test-Instanzen erstellen
4. Zwei Test-Instanzen mit XMPP-Server verbinden lassen
5. Zwei Instanzen von Saros sich als Freund hinzufügen
6. Eine Saros-Session erstellen
7. Eine Saros-Session erstellen und Freund einladen.
8. Saros-Session mit zwei Teilnehmern starten und eine Aktivität austauschen

Ich werde diese Tests im Folgenden als „strategische Tests“ oder Strategie-Tests bezeichnen.

Wie sich herausstellen sollte ergaben sich während der Realisierung dieser Schritte immer wieder neue Problemstellungen die es galt zu bewältigen. Die meisten der Testfälle zeigten sich als sehr umfangreich. So kann man sicher gut nachvollziehen, dass für den Einladungsprozess in eine SarosSession einige Schritte erforderlich sind [6]. In diesem Fall musste ich Teilschritte einlegen die dann nur Teilaspekte des entsprechenden Ablaufs konkreter testen sollten. Allerdings waren solche Teilschritte auch nicht immer ohne Weiteres möglich, da dazu die Initialisierung des notwendigen Zustandes nötig war, was an manchen Stellen besonders dem Einladungsprozess sehr aufwändig gewesen wäre. In solche Fällen musste ich mittels Debugging die Einzelschritte der Anwendungslogik nachvollziehen was mitunter sehr zeitintensiv war. Vor allen Dingen erschwerten die vielen parallelen Abläufe diesen Prozess. Aber auch der Aspekt, dass man bei nicht initialisierter Eclipse- und GUI-Komponenten oft keine genauen Fehlermeldungen erhält warum der Prozess unterbrochen wurde trug dazu bei, dass man an vielen Stellen sehr sorgfältig nachvollziehen musste wie die Anwendungslogik ausgeführt wird.

Im Folgenden möchte ich nun exemplarisch die Realisierung der ersten 3 Tests schildern und die Aspekte erläutern die sich dabei ergaben.

Zu Beginn habe ich von der automatischen Initialisierung des XMPP-Servers abgesehen und zunächst nur lokal einen solche Dienst gestartet.

[5] Das ist der im Programmcode etablierte Begriff für eine Sitzung über die die Editor-Aktivitäten der Teilnehmer ausgetauscht werden.

[6] Lädt man einen Freund zu einer SarosSession ein so werden die Freundeslisten und Projekte die für diese Session verwendet werden soll miteinander synchronisiert

### 4.4.1 Der erste Test - Konstruktion einer Saros Instanz

Folgenden Test galt es meiner Strategie nach zuerst zu erfüllen:

Listing 4.2: Erstellen einer Test-Instanz

```
1 ...
2 @Test
3 public void testConnect() {
4     Saros saros = new Saros();
5     assertNotNull();
6 }
7 ...
```

Die Formulierung des Tests zeigt klar welche Vorstellungen ich von dem Umgang mit einer Test-Instanz hatte. In dem Projekt nahm die Klasse *Saros* von Beginn an eine sehr tragende Rolle ein. Sie vereinte in sich die Programmlogik für die komplette Initialisierung einer Instanz des Saros Plug-Ins. Um sich für einen Test der bestehenden Funktionalitäten zum Verbinden zu einem XMPP-Server bedienen zu können, benötigte ich eine Saros-Instanz.

Natürlich lag es nahe zu Beginn mit dem Standard-Konstruktor von Saros zu versuchen ein Exemplar dessen zu erstellen. Allerdings traten schon hier die ersten Probleme auf, denn bei der Verwendung des Standard-Konstruktors kommt der Großteil der Initialisierung des PicoContainers zum Tragen. Die Liste der Komponenten die hier erstellt wurden beläuft sich auf über 80. Mit dem Aufbau des PicoContainers wurden für jede dieser Komponenten der entsprechende Konstruktor ausgeführt. Allerdings fand sich auch hier Programmlogik die schon Eclipse-spezifisch war oder Abhängigkeiten zu Komponenten hatte, die selbst nicht richtig initialisiert werden konnte.

Allein der erste Schritt, war somit selbst schon sehr umfangreich und verlangte von Anfang an einen sehr breiten Überblick über die Abläufe der Anwendung beim Starten.

Um mich besser auf die wesentlichen Abläufe für die erfolgreiche Ausführung meines Testes konzentrieren zu können, habe ich Komponenten die den Prozessfluss unterbrachen und offensichtlich nicht für die Verbindung zum XMPP-Server notwendig waren aus dem PicoContainer herausgenommen. So konnte ich zumindest die Masse der zu überblickenden Komponenten reduzieren und konnte die Abhandlung des Konstruktors ohne Unterbrechungen durchlaufen lassen. Es stellte sich später heraus, dass diese Fokussierung sehr nützlich war und ich im Laufe der Zeit die Ausklammerung wieder rückgängig machen konnte da ich bereits an anderer Stelle die Problematik beheben oder ganz explizit darauf eingehen konnte. Um noch einmal darauf hinzuweisen waren es genau diese Stellen die ich versuchte durch das testgetriebene Vorgehen zu identifizieren.

Aus der Realisierung des ersten strategischen Tests wurden also folgende Problemfelder deutlich auf die ich im Verlauf der Arbeit näher eingehen musste:

1. Die Initialisierung des PicoContainers.
2. Aufheben von Abhängigkeit zu eclipse-spezifischem Code

### 4.4.2 Der zweite Test - Verbindung mit XMPP Server

Die erfolgreiche Ausführung des folgenden strategischen Tests war eine Grundvoraussetzung für die weiteren Schritte. Denn um überhaupt eine SarosSession mit mehreren Teilnehmern erstellen zu können mussten diese sich mit dem XMPP-Server zunächst verbinden. Ich hier bereits direkt die Saros-Funktionalitäten verwenden um eine Verbindung zu dem lokal laufendem XMPP-Server herzustellen. Der Test sah wie folgt aus:

Listing 4.3: Erstellen einer Test-Instanz und Verbinden zu XMPP-Server.

```
1 ...
2 public void  testConnect() {
3     Saros saros = new Saros();
4     saros.connect()
5     assertTrue(saros.isConnected());
6 }
7 ...
```

Bei der Benutzung der Methode *connect()* stellte ich fest, dass diese Einstellung benötigt die in einem Eclipse *PreferenceStore* gehalten werden. Dabei handelt es sich um eine Standardimplementierung im Eclipse-Framework mit der ist es möglich ist Paare von Schlüsseln und -Werten ab zu speichern. Auf die Inhalte des *PreferenceStore* wird an vielen Stellen in der Saros-Logik zugegriffen:

Listing 4.4: Beispiel für Verwendung des PreferenceStore.

```
1 public class PreferenceUtils {
2 ...
3     public String getUserName() {
4                 return saros.getPreferenceStore().getString(
5                 PreferenceConstants.USERNAME);
6         }
7 ...
8 }
```

Da der *PreferenceStore* eine Komponente ist die von dem Eclipse-Framework initialisiert und dem Plug-In zu Verfügung gestellt wird, musste ich eine Lösung finden wie ich selbst einen solchen *PreferenceStore* erstellen und anbieten kann.

Um keine Änderungen an der Logik vornehmen zu müssen habe ich mich dafür entschieden eine neue Klasse namens *TestSaros* einzuführen welche eine direkte Ableitung von Saros ist.

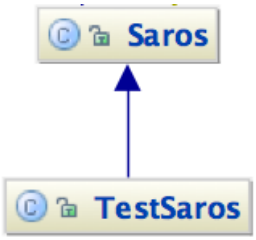


Abbildung 4.4: Einführung einer neuen Unterklasse *TestSaros*

Somit war es mir möglich die Methode die den *PreferenceStore* liefert zu überschreiben und selbst die Einstellungen für die Tests anzugeben.

Listing 4.5: Überschreiben der Saros-Methode die den PreferenceStore zurückliefert.

```
1  public class TestSaros {
2  ...
3  @Override
4  public void getPreferenceStore() {
5      _preferenceStore = new PreferenceStore();
6      _preferenceStore.setValue(PreferenceConstants.SERVER, "");
7      _preferenceStore.setValue(PreferenceConstants.USERNAME, "");
8      _preferenceStore.setValue(PreferenceConstants.PASSWORD, "");
9  }
10 ...
11 }
```

Zusätzlich zum Überschreiben dieser Methode musste ich noch Programmcode aus der Initialisierung des Plug-Ins als neue Methode extrahieren. Somit konnte ich diese Teil in meiner Unterklasse separat aufrufen und war nicht darauf angewiesen die eclipse-spezifischen Methode *start()* aufzurufen.

Bei der Realisierung des zweiten Schrittes bin ich also weitere Aspekte gestoßen die es zu beachten gab:

1. Eclipse-spezfische Funktionalitäten überschreiben bzw. selbst implementieren
2. Extraktion von Anwendungslogik für die Ausführung in Tests

Somit konnte war es nun möglich eine Test-Instanz sich mit einem XMPP-Server verbinden zu lassen.

### 4.4.3 Der dritte Test - mehrere Test-Instanzen erstellen

Schon während der Realisierung der ersten beiden strategischen Tests hätte mir bereits ein weiterer Aspekt auffallen können. Ein Aspekt der nicht einfach durch das Überschreiben von Methoden oder die Extraktion von Programmlogik behandelt werden konnte. Ein Aspekt der die Möglichkeit mehrere Instanzen von Saros zu erstellen in Frage stellte.

Folgender strategischer Test schien zu Beginn einfacher umzusetzen, als es am Ende wirklich war:

Listing 4.6: Erstellen von zwei Test-Instanzen

```
1 ...
2 @Test
3 public void testConnect() {
4     Saros saros1 = new Saros();
5     assertNotNull(saros1);
6     Saros saros2 = new Saros();
7     assertNotNull(saros2);
8 }
9 ...
```

Unsere Erwartungshaltung sollte hier sein, dass bei der mehrfachen Ausführung des Standardkonstruktors die zwei Instanzen *saros1* und *saros2* jeweils einen eigenen PicoContainer besitzen.

Bei näherer Betrachtung der Klasse *Saros* stellte sich allerdings heraus, dass dieser Test nicht ohne Weiteres erfolgreich durchlaufen könnte. Denn wie wir im folgendem Auszug dieser Klasse sehen können wurden hier bei der Implementierung statische Felder verwendet, um beispielsweise die laufende Instanz von Saros zu halten oder einen Zustand darüber zu halten, ob Saros vollständig initialisiert ist.

Listing 4.7: Verwendung von statischen Feldern in der Klasse Saros.

```
1 ...
2     /**
3      * The single instance of the Saros plugin.
4      */
5     protected static Saros plugin;
6
7     /**
8      * True if the Saros instance has been initialized so that calling
9      * reinject() will be well defined.
10     */
11    protected static boolean isInitialized;
12 ...
```

Zusätzlich stellten sich noch statische Funktionen zur Modifizierung des PicoContainers in den Weg und machten es nicht möglich zwei oder mehr Instanzen von Saros zu erstellen. Die Funktion *reinject* hatte zur Aufgabe ein Objekt in den PicoContainer nachträglich einzubinden und existierende Instanzen von dem gleichen Typ zu entfernen. Das neu eingefügt Objekt wurde dann durch den PicoContainer allen interessierten Komponenten injiziert. Wie wir in folgendem Auszug

sehen verwendet diese Methode die in dem statischen Feld *plugin* hinterlegte Instanz von Saros um an den PicoContainer zu gelangen.

Listing 4.8: Verwendung von statischen Funktionen in der Klasse Saros.

```
1  ...
2  public class Saros {
3      /**
4       * Injects dependencies into the annotated fields of the
5       * given object.
6       */
7      public static synchronized void reinject(Object toInjectInto) {
8
9          if (plugin == null || !isInitialized()) {
10             LogLog.error("...");
11             ...
12         }
13
14         try {
15             // Remove the component if an instance of it was
16             // already registered
17             plugin.container.removeComponent(toInjectInto.getClass());
18
19             // Add the given instance to the container
20             plugin.container
21                 .addComponent(toInjectInto.getClass(), toInjectInto);
22 ...
```

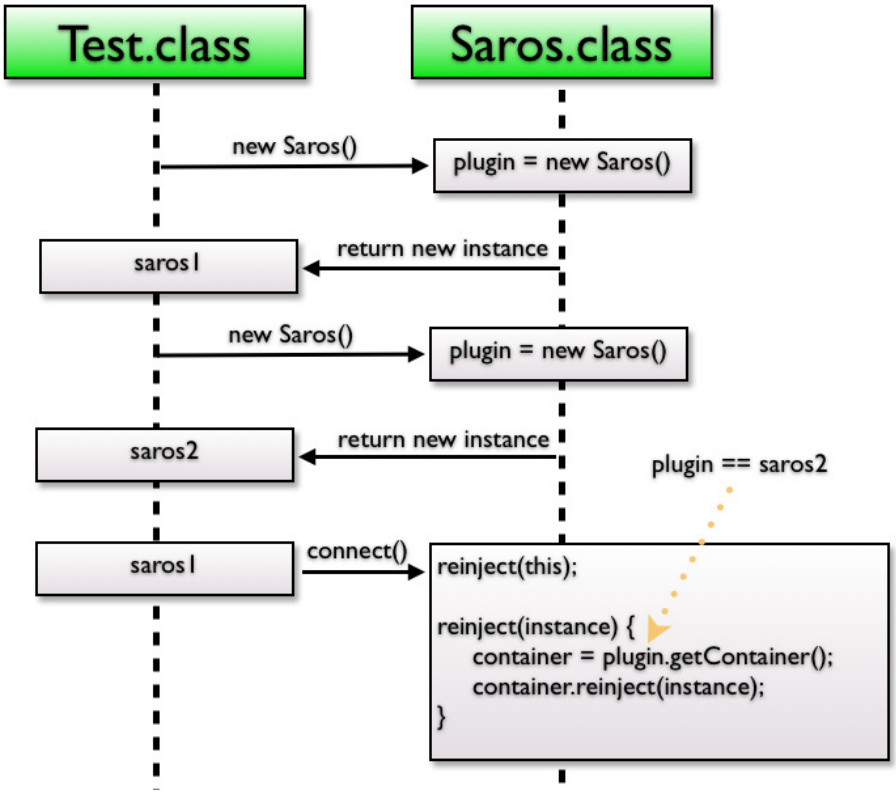


Abbildung 4.5: Schematischer Ablauf wie Isolation zweier Test-Instanzen aufgehoben wird.

Das Flussdiagramm in Abbildung 4.5 soll noch einmal schematisch zeigen was passierte, wenn man zwei Saros-Instanzen erstellen wollte.

Zwar werden beide Instanzen für sich richtig initialisiert allerdings wird bei der zweiten Ausführung des Konstruktors die erste Instanz in dem statischen Feld *plugin* durch die zweite verdrängt. Würde man nun die Methode *connect()* an der Instanz *saros1* aufrufen, so würde dies zum Aufruf der Methode *reinject* sich selbst führen und diese in den PicoContainer der Instanz *saros2* einführen. Damit waren die beiden Test-Instanzen und deren Komponentennetz nicht mehr isoliert voneinander. Aus der Realisierung für den strategischen Test ergab sich also ein sehr wichtiger konzeptioneller Punkt, der sich im späteren Verlauf als einer der gravierendsten herausstellte - Die Beseitigung von Konstrukten die kontextsensitives Verhalten unterlaufen.

Ein Kontext sei im Folgenden beschrieben als die Menge aller Komponenten, die bei der Ausführung des Standardkonstrukturs der Klasse Saros initialisiert werden und welche durch diese Komponenten wiederum während der Laufzeit erzeugt werden. Es darf zu keinem Zeitpunkt an keiner Stelle möglich sein, dass eine Saros-Instanz Kontrolle über eine Komponente einer anderen Saros-Instanz erlangt.

Kontextsensitives Verhalten soll demnach die Einhaltung dieser Grenzen bedeuten. Vor allen Dingen die Verwendung von statische Funktionen war der häufigste Grund für nicht kontextsensitives Verhalten.

Da die Lösung für dieses Problem ein eigenes Thema dieser Arbeit sei hier auf Abschnitt 5.1 verwiesen.

### 4.4.4 Weitere Aspekte

Bei der weiteren Realisierung der strategischen Tests wurden neben den schon oben genannten Aspekten noch weitere Problemfelder deutlich auf, die hier nun im folgenden eingegangen werden soll.

#### 4.4.4.1 Testen asynchroner Abläufe

Eine wichtige Grundvoraussetzung dafür, dass in einem Projekt automatische Tests geschrieben und deren Ergebnisse ernst genommen werden ist, dass sich diese deterministisch also bei jeder Ausführung gleich verhalten.

Bei der Umsetzung des 5. strategischen Tests bei dem sich zwei Test-Instanzen gegenseitig als Freund hinzufügen sollten kam es zu dem Effekt, dass der Test einer Ausführung erfolgreich durchlief aber direkt danach bei einer wiederholten Ausführung fehlschlagen konnte. Viele der Prozesse bei der Kommunikation zwischen mehrerer Saros-Instanzen verlaufen asynchron. Aus diesem Grunde konnten wir hier also nicht direkt nach dem Hinzufügen eines Freundes erwarten, dass die Prüfung ob dieser online ist erfolgreich verlief. Um im gewissen Maße sicherstellen zu können, dass die Erwartungshaltung eingetroffen ist, musste ich also darauf warten. Die einfachste Lösung hierfür wäre es eine feste Zeitspanne zu verwenden:

Listing 4.9: Beispiel für explizites Warten.

```
1 ...
2     saros1.addAsFriend(saros2);
3     TimeUnit.SECONDS.sleep(2);
4     assertTrue(saros1.isOnline(saros2));
5 ...
```

Theoretisch gesehen macht auch das den Test nicht deterministisch, allerdings hat es sich in der Praxis erwiesen, dass das Warten von einigen Millisekunden genügt um vertrauliche Test-Ergebnisse zu erhalten.

Da dieser Aspekt eminent den meisten Tests für Saros zu Grunde liegt habe ich versucht darauf besonders einzugehen, um gerade Einsteigern das Schreiben von asynchronen Tests zu erleichtern. Außerdem würde man durch Streuung von expliziten Warteanweisungen negativ die Lesbarkeit und Ausführungsdauer der Tests beeinflussen.

#### 4.4.4.2 Frontend-Antworten

Der letzte hier erwähnenswerte Aspekt der sich während der Umsetzung der strategischen Tests ergab betraf die Trennung von Business-Logik und Frontend. So gibt es einige Stellen in der Anwendungslogik die direkt GUI-Elemente erzeugen um beispielsweise Abfragen zu machen. Ein Beispiel hierfür ist der *SubscriptionManager* der bei einer eingehenden Freundes-Anfrage über einen MessageDialog den Anwender abfragt ob dieser stattgegeben werden soll oder nicht.

Listing 4.10: Beispiel für eine Methode (askUserForSubscriptionConfirmation) die die Erzeugung von GUI-Elementen direkt provoziert.

```
1 public class SubscriptionManager {
2 ...
3     // ask user for confirmation of subscription
4     if (autoSubscribe
5         || askUserForSubscriptionConfirmation(presence.getFrom())) {
6         ...
7     }
```

Listing 4.11: Implementierung der Methode askUserForSubscriptionConfirmation

```
1  public class SubscriptionManager {
2  ...
3  protected static boolean askUserForSubscriptionConfirmation(
4      final String from) {
5  ...
6      Utils.runSafeSWTSync(log, new Runnable() {
7      public void run() {
8          result.set(
9              MessageDialog.openConfirm(EditorAPI.getShell(),
10                 "Request␣of␣subscription␣received",
11                 "The␣buddy␣" + from + "␣has␣requested␣subscription."));
12 }
13 ...
```

Um auch hier zunächst größeren Änderungen aus dem Weg zu gehen habe ich dafür sorgen müssen, dass das Erstellen von solchen Frontend Komponenten nicht von der Business-Logik direkt provoziert wird und wie in diesem Fall automatisch diese Fragestellungen beantwortet werden.

Um die Verantwortlichkeit für die Erzeugung des Dialoges in die Frontend-Schicht zu verlagern hätte man auf ein Event-System umstellen müssen über welches Frontend-Komponenten darüber informiert werden können, dass eine solche Anfrage eingegangen ist. Die Frontend-Schicht hätte dann je nach Angabe des Anwenders entsprechend Funktionalitäten der Business-Logik benutzen müssen.

## 4.5 isTestSaros

Auf Grund der Tatsache, dass an vielen Stellen in der Anwendungslogik die Verwendung eclipse-spezifischer Funktionalitäten verstreut war bediente ich mich einer einfachen statischen Funktion *isTestSaros* welche während der Ausführung meiner Unit-Tests auf *true* zurücklieferte um diese Stellen von der Ausführung auszuschließen und entsprechend anders reagieren zu können. Diese Methode fand zu Beginn sehr oft Verwendung, um so schnell es ging einen Test zu erfolgreicher Ausführung zu verhelfen.

Natürlich war es das Ziel die Funktion später wieder zu beseitigen, denn in diesem Moment kam es zu einer Vermischung von test- und anwendungsrelevantem Programmcode. Der Vorteil an dieser Reihenfolge war, dass ich dann schon eine Testsuite hatte und so nach dem Zurücksetzen des Programmcodes prüfen konnte, ob noch alles so verläuft wie ich es erwartete. Oder ich konnte so schnell die Stellen identifizieren die einer entsprechenden Behandlung bedurften.

Im Nachhinein stellte sich auch heraus, dass ich durch Streuung dieses Statements einige Schnittpunkte zwischen den Schichten identifiziert habe die vorher nicht so offensichtlich waren.

# 5 Realisierung im Detail

Wie ich schon zu Beginn verdeutlichen wollte war der Weg dieser Arbeit zwar grob abgesteckt, aber über das Aussehen des Ziels, der Lösung konnte ich mir so allerdings nicht im klaren sein.

## 5.1 SarosContext

Der Aspekt der bei der Realisierung der strategischen Tests die größte Rolle spielte war, dass ich kontextsensitives Verhalten innerhalb einer Test-Instanzen erreichen musste. Bislang war in dem Projekt diese Ansicht nicht etabliert was unter anderem dazu führte, dass man oft über statische Funktionen an Komponenten gelangte, was wie wir bereits sehen musste kontextsensitives Verhalten stört. Ein weiteres damals im Projekt sehr etabliertes Beispiel hierfür war ebenfalls eine statische Funktion, die den PicoContainer verwendete um an die Komponenten zu gelangen die durch diesen verwaltet wurden. Eine komfortable und durchaus wünschenswerte Lösung gerade für die Entwicklung auf der Frontend-Ebene.

Listing 5.1: Beispiel für Verwendung injectDependenciesOnly in Frontend-Komponente.

```
1  public class RosterView {
2  ...
3      @Inject
4      protected SarosSessionManager sessionManager;
5  ...
6      public RosterView() {
7              super();
8
9              Saros.injectDependenciesOnly(this);
10     }
11 ...
```

Somit kann man bequem den Mechanismus des PicoContainers zum Injizieren der verwalteten Komponenten verwenden. Hätte diese statische Funktion nur in Frontend Komponenten Verwendung gefunden, so wäre diese theoretisch durch die Unit-Test nie ausgeführt worden. Allerdings wurde diese Funktion auch an Stellen verwendet an denen Entitäten erstellt wurden die sich selbst mit den Komponenten aus dem Kontext initialisierten [1]

Listing 5.2: Beispiel für Verwendung injectDependenciesOnly in Entität.

```
1  ...
2  public ProjectNegotiation(ITransmitter transmitter, JID peer,
3          ProjectNegotiationObservable projectExchangeProcesses,
4          SarosContext sarosContext) {
5          this.transmitter = transmitter;
6          this.peer = peer;
7          this.projectExchangeProcesses = projectExchangeProcesses;
8          this.projectExchangeProcesses.addProjectExchangeProcess(this);
9
10         Saros.injectDependenciesOnly(this);
11 ...
```

Um also kontextsensitives Verhalten zu etablieren genügte es in meinen Augen nicht einfach nur die Verwendung der statische Methode und Felder zu beseitigen. Ich musste dieses Konzept im Programmcode sich widerspiegeln lassen. Da der PicoContainer selbst den Kontext verkörperte

[1] Im übrigen verstößt dieses Vorgehen gehen die Konventionen der Dependency Injection. Richtiger Weise sollte die Komponente nichts von dem Kontext wissen und stattdessen von außen initialisiert werden.

lag es nahe diesen in einer neuen Klasse zu kapseln und alle Funktionalitäten die für das Setzen oder Auslesen von Komponenten im PicoContainer nötig waren dort zu sammeln. Durch die Etablierung der dieser neuen Klasse *SarosContext* konnte ich also die Stellen an denen die statischen Funktionen *reinject()* und *injectDependenciesOnly* verwendet wurden so anpassen, dass diese an einer Instanz vom SarosContext aufgerufen wurden.

Damit man an den entsprechenden Stellen auf den SarosContext zugreifen konnte, musste ich verschiedene Wege gehen. Zu aller erst wird der Kontext selbst auch dem PicoContainer hinzugefügt. Was es für Komponenten die durch den Kontext initialisiert werden ermöglicht per Inject-Annotation sich den SarosContext injizieren zu lassen. Für die Stellen an denen das nicht möglich war wie unter anderem bei der *ProjectNegotiation* musste ich den Konstruktor erweitern, damit es möglich ist den Kontext mit zu übergeben.

Listing 5.3: Verwendung einer Instanz vom SarosContext um Felder zu initialisieren.

```
1  ...
2  public ProjectNegotiation(ITransmitter transmitter, JID peer,
3          ProjectNegotiationObservable projectExchangeProcesses,
4          SarosContext sarosContext) {
5          this.transmitter = transmitter;
6          this.peer = peer;
7          this.projectExchangeProcesses = projectExchangeProcesses;
8          this.projectExchangeProcesses.addProjectExchangeProcess(this);
9
10         sarosContext.initComponent(this);
11 ...
```

Für die Komponenten der GUI-Ebene habe ich die Verwendung von statischen Methoden nur verschoben. Der richtige Umgang mit den Komponenten aus dem Kontext in dieser Schicht sollte separat bearbeitet werden. Also wurde eine Utils-Klasse entworfen *SarosPluginContext* die wie früher die Klasse Saros davon ausgeht das es nur einen Kontext gibt und die gleichen Methoden anbietet. Damit waren die Änderungen auf Ebene des Frontends nur marginal.

Listing 5.4: In Frontend-Komponenten änderte sich nur die Klasse an der die vorigen statischen Funktionen zu finden sind.

```
1  public class RosterView {
2  ...
3  @Inject
4  protected SarosSessionManager sessionManager;
5  ...
6  public RosterView() {
7          super();
8
9          SarosPluginContext.injectDependenciesOnly(this);
10
11     }
12 ...
```

Eine Anmerkung zum Kontext sei an dieser Stelle noch gemacht, denn bei diesem sollte es sich um einen Kontext für die Anwendungslogik handeln. Bei näherer Betrachtung muss man aller-

dings feststellen, dass hier mittlerweile auch Komponenten aus dem Frontend mit initialisiert werden. Als ein Beispiel seien hier die Komponenten *EditorAPI* und *SarosUI* erwähnt. Perspektivisch sollte man versuchen diese Komponenten aus dem Kontext zu herauszulösen.

Der SarosContext war für diese Arbeit unbestritten das wichtigste Teil-Ergebnis. Mit dem Kontext gab es einen zentralen Ort über welchen man testspezifische Anwendungslogik injizieren konnte, ohne den Produktcode direkt anpassen zu müssen.

Für folgende Problematik bot sich mit Hilfe des SarosContext eine denkbar einfache Lösung an. Für die Klasse IBBTransport hat man sich des Singleton-Patterns [7] bedienen wollen. Die Implementierung des Musters wurde mittels einer statischen Factory-Methode vorgenommen die im DataTransferManager verwendet wurde. Da der DataTransferManager bereits durch den SarosContext initialisiert wurde, musste ich für die Etablierung von kontextsensitivem Verhalten nur dafür sorgen, dass eine Instanz des IBBTransport ebenfalls im Kontext initialisiert wird. Diese Instanz musste dann nur über Dependency Injection via dem Kontruktor dem DataTransferManager zur Verfügung gestellt werden.

Listing 5.5: Injizierung der Instanz der Klasse IBBTransport via Konstruktor

```
1  public class RosterView {
2  public class DataTransferManager {
3  ...
4      private IBBTransport ibbTransport;
5  ...
6      public DataTransferManager(Saros saros,
7          SessionIDObservable sessionID,
8          PreferenceUtils preferenceUtils,
9          RosterTracker rosterTracker,
10         IBBTransport ibbTransport) {
11
12             this.ibbTransport = ibbTransport;
13 ...
```

Schlussendlich ist mit dem SarosContext ein Konstrukt entstanden, welches das Konzept eines Kontextes in der Anwendungslogik von Saros verankert und somit im späteren Projektverlauf hoffentlich beibehalten lässt.

## 5.2 Das Framework

Das Testframework stellt eine Zusammenfassung und Verallgemeinerung der Lösungen dar, die ich durch die Bearbeitung der strategischen Tests erarbeitet habe.

Es zeichnet sich dadurch aus, dass es leichtgewichtig ist und sich konzeptionell an JUnit anlehnt.

Zum einen wären da einige Hilfsmethoden, die man für das Schreiben eines Testes sehr gut benötigen kann. Diese Methoden finden sich in der Klasse SarosTestUtils [2]. Zum Beispiel erfordern häufig Methoden die Übergabe eines Eclipse *IProgressMonitor*, um den Verlauf der Aktion in der GUI darstellen oder auf das Abbrechen eines Prozesses durch den Anwender reagieren zu können. In den meisten Fällen darf dieser Parameter nicht *null* sein. Aus diesem Grunde findet sich hier eine statische Funktion, mit der man sich ein Exemplar eines Eclipse *SubMonitor* erstellen kann. Für meine Arbeit hätte ich mir eine solche Methode gewünscht, da ich erst im späteren Verlauf diese Varianten fand mit der man eine solche Entität erstellt. Zuvor hatte ich versucht einen IProgressMonitor mittels Mock-Objekt zu erstellen.

Einen weiteren Teil des Frameworks machen Testimplementierungen, in diesem Fall Fake-Objekte, aus. Ganz besonders hervorzuheben ist hier die Implementierung aller notwendigen Entitäten für den Dateizugriff über den Workspace und Projekte. Dies ist Recherchen nach die erste öffentliche Implementierung dieser Art und erwies sich bereits als sehr flexibel und hilfreich für das Erzeugen, Auslesen und Modifizieren von Projekten deren Dateien.

Hinzu kommen Hilfsmethoden die den Umgang von asynchrone Abläufe beim Testen verallgemeinern sollen. Mit Hilfe dieser soll es möglich sein Erwartungen zu formulieren ohne, dass man explizit daran denken muss auf das Eintreten von Bedingungen warten zu müssen.

Die meisten Teile des Frameworks sind über Methoden der abstrakten Klasse *AbstractSarosUnitTest* benutzbar. Möchte man einen Unit-Test schreiben der auf diese Funktionalitäten zurückgreifen soll, muss man nur noch eine Unterklasse dafür erstellen. Die Ausführung eines solche Tests führt dann auch dazu, dass automatisch lokal ein XMPP-Server gestartet wird über den die Testinstanzen kommunizieren können.

Für jeden dieser Teilbereiche [3] gibt es Tests aus denen man erkennen kann, wie diese Funktionalitäten benutzt werden und wie sie funktionieren sollen.

[2] Diese Klasse existierte schon vorher.
[3] ausgeschlossen einiger Hilfsmethoden

### 5.2.1 XMPP-Server

Wie schon in Abschnitt 4.3.3 erläutert habe ich mich für den Einsatz der XMPP-Server-Implementierung OpenFire entschieden. Hintergrund für dessen Einsatz war es, dass für die Ausführung der Tests keine Einrichtung eines XMPP-Servers durch den Entwickler notwendig ist.

Einer der wichtigsten Bedingungen die von automatisierten Tests immer gewährleistet sein muss ist, dass diese isoliert sind und stets von den gleichen Voraussetzungen ausgehen können. Für den Einsatz des XMPP-Servers bedeutet dies, das Nutzer-Accounts die für die Tests erstellt auch nach jedem Test gelöscht werden müssen.

Genau genommen müsste man dafür sorgen, dass auch der XMPP-Server mit zurückgesetzt wird. Allerdings reicht es in diesem Kontext aus nur die Account-Daten zurück zu setzen um von einer Isolierung der Tests untereinander sprechen zu können.

Für das Erzeugen und Löschen der Account-Daten habe ich eine Fassade etabliert. Durch den Aufruf der Methode *getNextUser()* wird ein neuer Account auf dem Server eingerichtet und ein XMPPUser-Object zurückgeliefert welches die Anmeldedaten enthält. Diese Daten werden später benötigt um die Saros Test-Instanzen zu initialiseren. Mit Aufruf der Methode *deleteAllCreatedAccounts()* werden alle erstellten Accounts wieder gelöscht.

In der ersten Version dieser Fassade war es noch möglich für einen Account selbst den gewünschten Namen anzugeben. Es erwies sich als äußerst sinnvoll einprägsame Namen wie „Alice“ und „Bob“ zu verwenden da diese in den Logmeldungen von Saros häufig mit ausgegeben werden und es so erlauben die Abläufe besser nachzuvollziehen zu können. Mit der Zeit stellte sich allerdings heraus, dass die manuelle Angabe des Namens keinen nennenswerten Mehrwert hatte. Zudem hätte dies auch eine mögliche Fehlerquelle sein können, wenn man versucht zwei Instanzen mit dem gleichen Benutzernamen zu erzeugen. Also habe ich mich dafür entschieden, dass die Fassade selbst für die Vergabe der Namen verantwortlich ist. Wie man jedoch an der folgenden Log-Meldung sehen kann liegt der Fassade eine Liste von Namen zu Grunde um trotzdem lesbare Namen für die Tests zu verwenden.

Listing 5.6: Log-Meldung der Fassade nach Erstellen eines User-Account auf dem XMPP-Server

```
1 **** XMPP-FACADE: Created User-Account: alice
```

### 5.2.2 Fake-Implementierung

Für die Realisierung der Strategie-Tests die eine SarosSession voraussetzen, musste ich eine Lösung finden mit der es den Test-Instanzen möglich ist Operationen auf dem Workspace oder Projekten und ihren Dateien auszuführen.

Zuerst habe ich damals versucht mit Hilfe von Mock-Objekten hier vorzugehen. Allerdings musste ich feststellen, dass die Anforderungen an die Mock-Objekte immer größer wurden und diese mehr und mehr Verhalten implementieren mussten um entsprechende Antworten zu liefern.

Listing 5.7: Beispiel für komplizierte Definition eines Mock-Objektes

```
1
2  IProject project = createMock(IProject.class);
3  ...
4  expect(project.getProject()).andStubReturn(project);
5  expect(project.isOpen()).andStubReturn(true);
6  expect(project.getName()).andStubReturn(projectName);
7  expect(project.exists()).andStubReturn(true);
8  ...
9  List<File> javaFiles = getJavaProjectFiles(path);
10 IResource[] ressources = new IResource[javaFiles.size()];
11
12 int i = 0;
13 for (final File file : javaFiles) {
14     final IFile aFile = createMock(IFile.class);
15     expect(aFile.getName()).andStubReturn(file.getName());
16     expect(aFile.getContents()).andStubAnswer(
17     new IAnswer<InputStream>() {
18         public InputStream answer() throws Throwable {
19             return new FileInputStream(file.getPath());
20         }
21     });
22     ...
23     replay(aFile);
24
25     expect(project.getFile(new Path(file.getPath()))).andReturn(aFile);
26
27     ressources[i] = aFile;
28     i++;
29 }
30
31 expect(project.members()).andStubReturn(ressources);
32 replay(project);
33 ...
```

Wie man in Zeile 9 sehen kann habe ich damals versucht auf Test-Dateien zurückzugreifen um die Test-Objekte richtig zu initialisieren. Nach einiger Zeit war die Verwendung dieser Mock-Objekte nicht mehr praktikabel. Immer unübersichtlicher wurde die Definition dieser. Also entschied ich mich dazu nachdem ich zu Beginn der Arbeit dies wegen des Aufwandes gescheut habe, selbst

eine eigene Implementierung der Interfaces *IWorkspace*, *IWorkspaceRoot*, *IProjekt* und andere eclipse-spezifischer Entitäten die das File-System abstrahieren anzubieten. Ich baute also Fake-Objekte für diese Interfaces.

Da ich bereits durch die Definition der Mock-Objekte Erfahrung im Umgang mit diesen Entitäten gesammelt habe war die Implementierung dieser dann nicht mehr so aufwändig wie ursprünglich befürchtet. Ich wusste zu diesem Zeitpunkt relativ genau welche Erwartungen die Anwendungslogik an die Implementierung hatte. Dabei sei erwähnt das die jetzige Implementierung generisch einsetzbar und nicht vom Saros-Projekt abhängig ist.

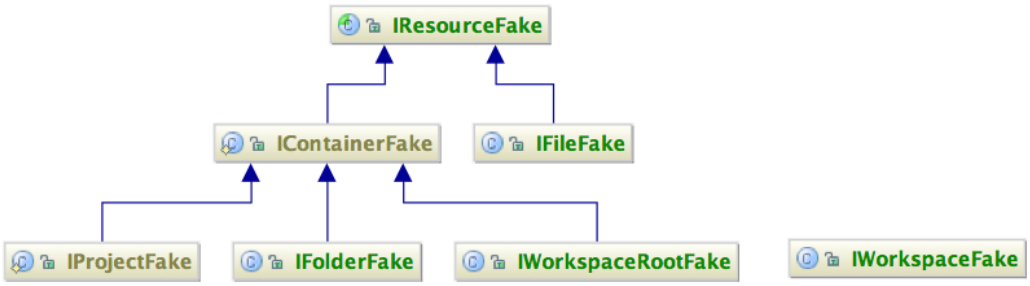


Abbildung 5.1: Klassendiagram der Fake-Objekte

In Abbildung 5.1 kann man die Struktur der Implementierung erkennen. Es verwundert nicht, dass die Hierarchie genau der aus dem Eclipse-Framework entspricht. Um ähnlich wie im Eclipse-Framwork durch Abstraktion Funktionalitäten wieder verwenden zu können, habe ich ebenfalls die Interfaces *IRessource* und *IContainer* als Oberklassen implementiert.

Die Implementierung dieser Interfaces ist keinesfalls vollständig. Ähnlich wie wir bei Mock-Objekten nur das Verhalten definieren, welches tatsächlich benutzt wird, so sind auch hier nur die Funktionalitäten implementiert die ich für die Realisierung meiner Tests gebraucht habe. Alle anderen Methoden liefern eine *NotYetImplementedException* welches eine *RuntimeException* [4] ist und den Entwickler der diese Funktionalität benutzen möchte darauf hinweisen soll, dass diese noch implementiert werden muss.

Da es sich bei all den Interfaces um Abstraktionen von Dateien handelt lag es nahe hier das Adapter-Pattern zu verwenden. Jede der Implementierung delegiert die Methodenaufrufe an eine Instanz vom Typ *java.io.File* weiter. bzw. nutzt dessen Funktionalitäten um das gewünschte Verhalten zu erreichen.

[4] Der Vorteil dieser Art von Excepton ist, dass man hierfür nicht die Schnittstellenbeschreibung der Methode ändern muss.

Listing 5.8: Einsatz des Adapter-Musters

```
1
2  abstract public class IResourceFake implements IResource {
3
4      protected File wrappedFile;
5      ...
6      protected IResourceFake(File wrappedFile) {
7          this.wrappedFile = wrappedFile;
8      }
9      ....
10     public boolean exists() {
11         return wrappedFile.exists();
12     }
13
14 }
```

Der Vorteil dessen ist, dass wir so auf eine Vielzahl von Funktionalitäten zurückgreifen können ohne diese selbst noch einmal implementieren zu müssen.

Wie schon beim Beispiel für die Definition des Mock-Objektes zu erkennen ist hatte ich ursprünglich vorgesehen ein Referenz-Testprojekt zu verwenden, welches aus einigen wenige Dateien bestand und für jede Test-Instanz von Saros kopiert werden musste. Mit Hilfe der Fake-Objekte konnte ich diesen Ansatz durch eine dynamische Projekterzeugung aus dem Programmcode heraus ablösen.

Ähnlich wie für XMPP-Server-Fassade bietet die Workspace-Fassade Methoden zum erzeugen und löschen von Workspaces. Um beispielsweise eine Referenz auf eine Datei in einem Workspace zu erhalten muss man wie folgt vorgehen:

Listing 5.9: Erstellen eines Workspaces und Zugriff auf ein Projekt und eine Datei

```
1 IWorkspace workspace = WorkspaceFacadeForTests.createWorkspace("alice");
2 IProject project = workspace.getRoot().getProject("testproject");
3 project.getFile("src/Person.java");
```

Damit man während der Ausführung der Tests auf den entsprechenden Workspace einer Test-Instanz zugreifen kann, wurde der Unterklasse TestSaros entsprechend Methoden hinzugefügt. Bei der Erzeugung eines TestSaros wird also der Instanz ein Workspace zugeteilt, sodass man später hier Projekte hinzufügen, modifizieren oder auslesen kann. Außerdem musste dafür gesorgt werden, dass sich die Stellen der Anwendungslogik die einen Workspace benötigen sich kontextsensitiv verhalten. Hierfür wird durch den SarosContext jeder Test-Instanz ein *EclipseHelper* initialisiert der, je nachdem ob das Frontend vorhanden ist oder nicht, den Workspace der entsprechenden Test-Instanz oder den von Eclipse initialisierten Workspace zurück liefert.

Damit ist die Etablierung der Fake-Objekte sehr einfach geworden. So brauchte ich beispielsweise in der IncomingProjectNegotiation [5] nur noch dafür sorgen, dass der EclipseHelper injiziert wird und die Verwendung der Eclipse-Funktionen durch den Aufruf der Methode vom EclipseHelper ersetzen.

[5] Diese Klasse ist verantwortlich den Einladungsprozess der eingeladenen Instanz zu behandeln.

Listing 5.10: Verwendung von statischen Funktionen (vorher)

```
1  public class IncomingProjectNegotiation extends ProjectNegotiation {
2  ...
3  public void accept(Map<String, String> projectNames,
4      SubMonitor subMonitor,
5      Map<String, Boolean> skipSyncs,
6      boolean useVersionControl)
7      throws SarosCancellationException {
8  ...
9          IWorkspace ws = ResourcesPlugin.getWorkspace();
10 ...
```

Listing 5.11: Verwendung des EclipseHelpers (nachher)

```
1  public class IncomingProjectNegotiation extends ProjectNegotiation {
2  ...
3  public void accept(Map<String, String> projectNames,
4      SubMonitor subMonitor,
5      Map<String, Boolean> skipSyncs,
6      boolean useVersionControl)
7      throws SarosCancellationException {
8  ...
9          IWorkspace ws = eclipseHelper.getWorkspace();
10 ...
```

### 5.2.3 Asynchrones Testen

Um beim Schreiben von Unit-Tests mit Saros Test-Instanzen nicht ständig auf die Aspekte der Nebenläufigkeit achten zu müssen habe ich versucht eine Standardlösung zu etablieren, bei deren Verwendung man nicht mehr explizit auf das Eintreten eines Zustandes warten muss.

Nachdem ich für die Realisierung der ersten Tests zunächst feste Wartezeiten verwendete und dadurch die Laufzeiten der Tests immer länger wurden habe ich mir zunächst eine statische Funktion *waitFor* bereitgestellt. Die Funktion hatte zur Aufgabe zyklisch eine Bedingung zu testen und erst den weiteren Testablauf zu ermöglichen wenn diese eintraf.

Listing 5.12: Die waitFor-Funktion zum Warten auf das Eintreten einer Bedingung.

```
1 waitFor(new TCondition() {
2     public boolean isFullfilled() throws Exception {
3         return saros.receivedOneMessage();
4     }
5 });
```

Diese Methode erwartete die Implementierung einer *TCondition*, ein einfaches Interface welches die Methode *isFullfilled* deklarierte. *isFullfilled* wird mehrfach mit kurzen Wartezeiten dazwischen aufgerufen. Die Implementierung dieser Methode sieht in etwa wie folgt aus:

Listing 5.13: Implementierung der waitFor-Funktion

```
1  public static void waitFor(TCondition testCondition) {
2      boolean fullfilled = false;
3      for (int i = 0; i < 100; i++) {
4          try {
5              TimeUnit.MILLISECONDS.sleep(100);
6              if (testCondition.isFullfilled()) {
7                  fullfilled = true;
8                  break;
9              }
10         } catch (Exception e) {
11             // do nothing ...
12         }
13     }
14     assertTrue(fullfilled);
15 }
```

Mit Hilfe dieses Vorgehens konnte ich die Laufzeit der Tests wesentlich verkürzen und hatte mir zudem ein generisches Konstrukt geschaffen, um auf das Eintreten von Erwartungen zu warten.

Streng genommen kann natürlich die Erwartung auch noch nach der maximalen Wartezeit eintreten, allerdings hat sich es im Umgang mit diesen Methoden gezeigt, dass diese fehlschlugen wenn auch tatsächlich etwas nicht wie erwartet verlief.

Im Verlauf der Arbeit habe ich dieses Konstrukt weiter abstrahiert, um damit auf Code-Ebene Erwartungen auszudrücken die den von JUnit ähneln sollten.

Nach vielen Versuchen kann die Formulierung einer solchen Erwartung nun wie folgt aussehen:

Listing 5.14: Beispielhafte Formulierung einer Erwartung mit der assertThat-Methode der abstrakten Klasse AbstractSarosUnitTest.

```
1 assertThat(new ValueProducer<Boolean>() {
2     public Boolean value() {
3         return false;
4     }
5 }).isTrue();
```

Man beginnt die Formulierung einer Erwartung mit der Funktion *assertThat* welche einen *ValueProducer* erwartet. Anschließend kann man via Methodenverkettung (MethodChaining) angeben welche Erwartungen man an den Wert des *ValueProducers* hat. Das Konstrukt verhält sich ähnlich wie *waitFor*. So wird der Wert des ValueProducers zyklisch erzeugt und mit der Erwartungshaltung verglichen. Tritt die Bedingung nicht ein erhält man eine aussagekräftige Fehlermeldung die der des JUnit-Frameworks gleicht:

Listing 5.15: Fehlermeldung die man erhält, wenn eine mit asserThat formulierte Erwartung nicht eintritt.

```
1 java.lang.AssertionError:
2     expected element containing string <Testnachricht.>
3     but was:<Eine andere Nachricht!>
```

Im Vergleich zur *waitFor*-Funktion bei der wir nur angeben konnten ob eine Bedingung erfüllt war oder nicht, können wir hier genauer zum Ausdruck bringen weshalb eine Erwartung nicht eingetroffen ist.

Ich habe mich bewusst für die jetzige Bezeichnung der Methode entschieden damit es genau einen Anhaltspunkt gibt um mit dem Schreiben einer Erwartungen zu beginnen.

### 5.2.4 Hilfsklassen

Bei der Verwendung der waitFor-Funktion zeigte sich mit der Zeit gewisse Muster. Als Beispiel sei hier folgender abstrakte Test aufgeführt:

Listing 5.16: Abstrakte Formulierung eines Tests

```
1 saros1.sendsMessageWithContent("Testinhalt");
2 saros2.receivesMessageWithContent("Testinhalt");
```

Um tatsächlich testen zu können ob bei *saros2* eine Nachricht mit dem angegebenen Inhalt angekommen ist, musste ich vorher bei der Instanz *saros2* einen entsprechenden Listener initialisieren welcher dies prüfte. In Wirklichkeit hätte der Test in etwa so aussehen müssen:

Listing 5.17: Tatsächliche Implementierung des abstrakten Tests

```
1  XMPPReceiver xmppReceiver2 =
2      saros2.getContext().getComponent(XMPPReceiver.class);
3
4  final AtomicBoolean received = new AtomicBoolean(false);
5  xmppReceiver2.addPaketListener(
6      new PaketListener() {
7          public void processPaket(Paket packet) {
8              if (packet.toXML().contains("Testinhalt")) {
9                  received.set(true);
10             }
11     }}, new PaketFilter() {
12         public boolean accept(Paket packet) {
13             return true;
14         }
15     });
16
17 saros2.receivesMessageWithContent("Testinhalt");
18
19 waitFor(new TCondition() {
20     public boolean isFullfilled() throws Exception {
21         return received.get();
22     }
23 });
```

Es lässt sich wohl nicht bestreiten, dass dieser Test wesentlich komplizierter und somit schwerer zu verstehen ist als dessen abstrakte Formulierung. Des Weiteren würde es nicht unbedingt jedem Entwickler sofort einleuchten, dass man für das Testen solcher Ereignisse schon im Voraus einen Listener installieren muss.

Um dieses Muster zu abstrahieren und zudem für die richtige Reihenfolge [6] zu sorgen habe ich eine Hilfsklasse entworfen. Mit Aufruf der Methode *receives()* an einer Test-Instanz erhält man nun einen *ReceiverHelper*.

[6] Installieren eines Listeners und Ausführung der Methode.

#### 5.2.4.1 ReceiverHelper

Damit wir in einem Test erkennen können, ob eine TestInstanz eine Aktivität erhalten hat müssen wir vor dem Senden einen Listener bei dem XMPPReceiver [7] der entsprechenden Test-Instanz installieren. Würden wir dies erst nach dem Senden machen könnte vielleicht es schon zu spät sein. Damit also Entwickler die einen entsprechenden Test schreiben diesen Sachverhalt nicht vergessen habe ich die Methoden des *EclipseHelper* mit einem zweiten erforderlichen Parameter ausgestattet. Jede Methode erwartet eine Instanz des Interfaces *When* welches einem *Callable* gleicht und von der Methode erst dann ausgeführt wird, wenn der entsprechende Listener fertig initialisiert ist.

Ein Beispiel für die Verwendung dieser Methoden ist folgendes:

Listing 5.18: Verwendung des ReceiverHelpers

```
1 saros2.receives().packetContaining("Testinhalt", new When() {
2     public void running() throws Exception {
3         saros1.sendsPaketContaining("Testinhalt");
4     }
5 });
```

Die Methoden liefern zudem gleich die entsprechende Aktivität als Ergebnis zurück, sofern denn auch eine solche in einer bestimmten Zeitspanne eingetroffen ist. Hierfür habe ich mich an dem Konzept der *Futures* [8] aus Java orientiert.

Listing 5.19: Verwendung von FutureResult

```
1  public Paket packetContaining(final String content, When when) {
2      final FutureResult<Paket> result = new FutureResult<Paket>();
3
4      // install listener which sets value in result ...
5
6      try {
7          when.running();
8      } catch (Exception e) {
9          e.printStackTrace();
10     }
11     return result.getValue();
12 }
```

Wie wir in diesem Beispiel sehen liefert die Methode das Ergebnis eines sogenannten *FutureResult*. Hierbei handelt es sich um eine Entität die beim Aufruf der Methode *getResult* blockiert und maximal n Sekunden wartet bis der Wert des Ergebnisses vorliegt. Trifft dieser ein wird mit dem Programmverlauf sofort fortgefahren. Das Setzen des Wertes vom *FutureResult* geschieht im installierten Listener.

[7] Der XMPPReceiver ist in einer Saros-Instanz die zentrale Stelle an der Pakete eingehen und beispielsweise als Aktivitäten weiter verteilt werden.

[8] http://download.oracle.com/javase/6/docs/api/java/util/concurrent/Future.html

#### 5.2.4.2 SendHelper

Um die Formulierung eines Unit-Tests noch näher an die im Abschnitt 5.2.4 zu findende abstrakte Form zu bringen, habe ich eine Hilfsklasse für das Senden von Aktivitäten und Paketen etabliert. Sie soll dem Entwickler auch dazu dienen die Stellen die für das Senden von Aktivitäten oder Nachrichten verantwortlich sind schneller zu finden. Andernfalls müsste dieser in der gesamten Anwendungslogik danach suchen sofern es keine geeignet Dokumentation gibt.

Ein Test unter Verwendung beider Hilfsklassen würde wie folgt aussehen:

Listing 5.20: Verwendung von ReceiverHelper und SenderHelper

```
1 Paket packet =
2     saros2.receives().packetContaining("Testinhalt", new When() {
3         public void running() throws Exception {
4             saros1.sends().messageTo(saros2, message);
5         }
6     });
7 assertNotNull(packet);
```

### 5.2.5 Szenarien

Im weiteren Verlauf der Arbeit habe feststellen müssen, dass ich immer wieder die gleiche Initialisierung von Test-Instanzen benötigte.

Die Initialisierung einer oder mehrerer Test-Instanzen und deren Beziehung untereinander soll im folgenden als Szenario bezeichnet werden. Test-Instanzen können in folgenden Beziehung stehen:

1. eine Test-Instanz einzeln
2. mehrere Test-Instanzen
3. mehrere Test-Instanzen die sich gegenseitig als Freund hinzugefügt haben
4. mehrere Test-Instanzen die sich gemeinsam eine SarosSession teilen

Für jede dieser Beziehungstypen bietet die Hilfsklasse *ScenarioInitializer* eine entsprechende Methode an. In einem SarosUnitTest steht ein solcher *ScenarioInitializer* über die Methode *sarosSetup* zur Verfügung.

Um also eine SarosSession mit drei Teilnehmern zu erstellen muss man lediglich folgenden Aufruf verwenden:

Listing 5.21: Erstellen eines kompletten Test-Szenarios mit dem ScenarioInitializer

```
1 List<TestSaros> group = sarosSetup().getSession(3);
```

Alle Methoden [9] liefern eine Liste als Ergebnis die die initialisierten Test-Instanzen enthalten. Um auf die einzelnen Test-Instanzen zugreifen zu können muss man nur noch die Liste entsprechend auslesen.

Die initialisierten Saros-Instanzen erfüllen dabei folgende Konventionen.

Jede Test-Instanz:

- hat einen eingerichteten Account auf dem XMPP-Server
- ist auf dem XMPP-Server eingeloggt
- besitzt einen eigenen Workspace
- und hat ein Testprojekt mit zwei 2 Testdateien

Listing 5.22: Die Methoden für das Erstellen eines Szenarios sind lesbar gehalten um Konventionen nachvollziehen zu können.

```
1 public List<TestSaros> getSession(int numOfUsers) {
2     List<TestSaros> users = getSarosList(numOfUsers);
3     loginSarosList(users);
4     subscribeUsersInRoster(users);
5     createSessionFor(users);
6     return users;
7 }
```

[9] bis auf die Methode die eine einzelne Test-Instanz zurückliefert

### 5.2.6 TestSarosContext

Zu Beginn der Arbeit habe ich häufig versucht bei der Realisierung der Tests voranzukommen in dem ich vorerst Programmcode der den Ablauf der Unit-Tests störte von der Ausführung ausklammerte. Dazu gehörten unter anderem kleine Programmteile bis hin zu ganzen Komponenten die nicht mehr mit dem Kontext initialisiert werden sollten. Für Frontend-Komponente machte dies auch durchaus Sinn.

Also verallgemeinerte ich die Initialisierung des PicoContainers im SarosContext und fügte die Möglichkeit hinzu der Initialisierungsmethode eine Liste von Klassen zu übergeben die nicht mit in den PicoContainer aufgenommen werden sollen [10].

Um aber auch die Änderungen in den einzelnen Klassen wollte ich wieder rückgängig machen um den testrelevanten Code wieder von dem Produktcode zu trennen. Für diese Fälle entschied ich mich dafür Unterklassen der entsprechenden Klassen zu erstellen in denen ich dann die entsprechenden Funktionalitäten überschreiben konnte. Ich musst dann nur noch dafür sorgen, dass für die Erstellung eines Test-Kontextes diese Implementierungen verwendet werden. Somit gibt

```
SarosContext result = new SarosContext();
result.saros = this.saros;
result.dotMonitor = this.dotMonitor;
result.isTestContext = this.isTestContext;
if (isTestContext) {
    result.init(excludedComponentsForTestContext);
    addTestImplemantations(result);
```

Abbildung 5.2: Initialisierung eines Test-Kontextes

es also zwei verschiedene Typen eines SarosContext. Zum einen den vollständigen SarosContext der alle bisherigen Komponenten enthält und einen Test-Kontext der nur eine Teilmenge der Komponenten vom vollständigen SarosContext benutzt und zusätzlich noch andere Implementierung von bestimmten Klassen verwendet.

Auf diese Weise konnte ich die größten Eingriffe im Produktcode weitesgehend wieder zurückbilden.

[10] Da die Zahl der Komponenten die für Unit-Tests nicht benötigt werden sehr klein ist habe ich mich dafür entscheiden ein Liste derer zu erstellen die nicht aufgenommen werden sollen.

# 6 Fazit

## 6.1 Fallstudie

Nach mehr als zwei dritteln der Bearbeitungszeit habe ich eine kleine Fallstudie mit einer Versuchsperson aus dem Saros-Projekt durchgeführt um bewerten zu können ob die Funktionalitäten meines Frameworks für jemand anderes auffindbar, nachvollziehbar und einsetzbar sind.

### Aufgabe

Die Aufgabe für die Versuchsperson war es einen Test für die bestehende Anwendungslogik zu erstellen und testgetrieben Änderungen an dieser zu vollziehen die im Kontext eines größeren Projektes notwendig gewesen wären.

Als ein solches Test-Projekt entwarf ich die konzeptionelle Idee eines sogenannten Saros-Loggers. Dabei soll es sich um eine Ausprägung der Saros-Anwendungslogik handeln die keine GUI benötigt und sich an einer SarosSession nur passiv beteiligen kann. Sinn soll es sein, dass wenn eine SarosSession erstellt wird automatisch eine solche Instanz zu der Sitzung mit eingeladen werden soll. Der Saros-Logger könnte dann so beispielsweise alle Aktivitäten der SarosSession mit protokollieren um den Entwicklern neue Informationen über den Verlauf von SarosSesssions liefern zu können.

Die Aufgabe des Probanden bestand zum einen darin sich mittels eines Tests die notwendigen Funktionalitäten zu eigen zu machen die er für das Protokollieren der Aktivitäten benötigte. Für den zweiten Teil der Aufgabe sollte dieser testgetrieben die notwendige Änderung beim Erstellen einer SarosSession vornehmen um automatisch eine Saros-Instanz zu einer Sitzung einzuladen.

An dieser Stelle sei erwähnt das zum damaligen Zeitpunkt das Framework anders aussah. So habe ich damals die Prämisse verfolgt die Klasse TestSaros so wenig wie möglich mit Funktionalitäten auszustatten die man bei der Klasse Saros nicht findet. Außerdem habe ich viele Funktionalitäten u.a. auch die für das gegenseitige Hinzufügen von Test-Instanzen als statische Funktionen in der SarosTestUtils angeboten.

### Durchführung

Zeitlich waren für die beiden Schritte insgesamt 30 Minuten angesetzt. Zu Beginn habe ich das Framework konzeptionell vorgestellt und mittels beispielhafter Tests gezeigt wie man Test-Instanzen erstellt, wo wichtige Funktionalitäten zu finden sind und wie man den SarosContext benutzt.

Damit der Proband sich auf die eigentliche Aufgabe fokussieren konnte habe ich je nach Lage das weitere Vorgehen grob skizziert.

Für die Realisierung des ersten Testes sah der Proband sich mit folgendem Szenario konfrontiert. Zu erstellen waren zwei Test-Instanzen, wobei eine den Saros-Logger repräsentieren sollte und die zweite dazu dienen sollte ein Paket an den Saros-Logger zu senden. Ziel der Aufgabe war es testweise zu skizzieren wo man bei dem Saros-Logger ansetzen müsste um Aktivitäten mit zu protokollieren. Dafür musste der Proband an dem XMPPReceiver des Saros-Loggers einen Listener installieren der prüft ob das gesendete Paket angekommen ist.

Der vollständige Test hierfür sah damals so aus:

Listing 6.1: Resultat für den ersten Test der Fallstudie

```
1  @Test
2  public void testLogger() {
3      TestSaros alice = getSarosForUsername("alice");
4      TestSaros logger = getSarosForUsername("logger");
5
6      alice.connect(false);
7      logger.connect(false);
8
9      final List<Boolean> receivedPackages = new ArrayList<Boolean>();
10     XMPPReceiver xmppReceiver =
11         logger.getComponent(XMPPReceiver.class);
12     xmppReceiver.addPaketListener(new PaketListener() {
13         public void processPaket(Paket packet) {
14             receivedPackages.add(true);
15         }
16     }, new PaketFilter() {
17         public boolean accept(Paket packet) {
18             return true;
19     }});
20
21     XMPPTransmitter transmitter =
22         alice.getComponent(XMPPTransmitter.class);
23     Message message = new Message();
24     message.setSubject("Das ist ein Test");
25     transmitter.sendMessageToUser(logger.getMyJID(), message, false);
26     SarosTestUtils.waitFor(new Assertion("Receive package") {
27         @Override
28         public boolean isFullfilled() {
29             return receivedPackages.size() > 0;
30         }
31     }, 5);
32 }
```

Für die Umsetzung dieses Tests habe ich Hinweise darüber gegeben welche Komponente für das Versenden von Nachrichten (in diesem Fall der *XMPPTransmitter*) und welche Komponente für das Verarbeiten von eingehenden Nachrichten verantwortlich ist (hier der *XMPPReceiver*). Außerdem musste ich Hilfestellungen dazu geben wo Funktionalitäten von meinem Framework zu finden sind. Zwar erkannte der Proband die Problematik der Nebenläufigkeit, allerdings vermutete er die Lösung für dieses nicht in den SarosTestUtils [1]. Insgesamt haben wir für die Realisierung dieses Tests ca. 20 Minuten benötigt.

Für den zweiten der Teil der Aufgabe sollte der Proband testgetrieben vorgehen, also zu erst einen Test schreiben und dann zur erfolgreichen Ausführung bringen.

[1] damals lag die *waitFor*-Funktion hier

Auch hier habe ich Hilfestellung geleistet und mögliche Stellen (in diesem Fall der *SarosSession-Manager*) vorgegeben an der Änderungen durchgeführt werden mussten. Des Weiteren habe ich die Schritte die für das Starten einer SarosSession nötig sind vorgegeben.

Wir wollten in diesem Fall zunächst zwei Test-Instanzen starten die sich gegenseitig als Freund hinzufügen sollten. Anschließend sollte eine der Test-Instanzen eine SarosSession starten und später die zweite Test-Instanz zu dieser Sitzung einladen.

Listing 6.2: Resultat für den zweiten Test der Fallstudie

```
1  @Test
2  public void testLoggerIsParticipant() throws XMPPException {
3      TestSaros alice = getSarosForUsername("alice");
4      TestSaros bob = getSarosForUsername("bob");
5
6      loginAndSubscribeUsersInRoster(asList(alice, bob));
7
8      SarosSessionManager sarosSessionManager = alice
9          .getComponent(SarosSessionManager.class);
10
11     sarosSessionManager.startSession();
12
13     final ISarosSession sarosSession = sarosSessionManager
14         .getSarosSession();
15     SarosTestUtils.waitFor(new Assertion("Logger␣added␣to␣session") {
16         @Override
17         public boolean isFullfilled() {
18             return sarosSession.getParticipants().size() == 2;
19         }
20     });
21 }
```

Das Formulieren dieses Testes fiel dem Probanden merklich leichter. Die *waitFor*-Funktion verwendete dieser hier nun selbstständig und formulierte die Bedingung auf die gewartet werden sollte allein. Allerdings musste ich auch in diesem Fall wieder auf die *SarosTestUtils* hinweisen als es darum ging eine bestehende Funktionalität wieder zu verwenden um die Test-Instanzen sich gegenseitig als Freund hinzuzufügen.

Da die Durchführung der Implementierung für diesen zweiten Test die restliche verbleibende Zeit überschritten hat, haben wir nach der erfolgreichen Ausführung des obigen Tests das Experiment beendet. Für Realisierung dieses Tests haben wir insgesamt ca. 20 Minuten benötigt.

## Beobachtungen

Während der Durchführung des Experimentes konnte ich folgende Beobachtungen machen, die für das Schreiben von Unit-Tests mit meinem Framework relevant sind.

**Auffindbarkeit von Funktionalitäten** Funktionalitäten die ich in den *SarosTestUtils* untergebracht habe wurden bei der Überlegung einen Test zu schreiben wenig bis gar nicht nicht in Betracht gezogen als die Funktionalitäten die direkt an der *AbstractSarosUnitTest* zu finden waren.

**Höhere Erwartungen an Test-Instanz** Der Proband erwartete Funktionalitäten erweiterte Funktionalitäten direkt an der Test-Instanz.

**Hoher Aufwand** Der größte Aufwand um einen Test zu formulieren bestand meist darin die Funktionalitäten die dafür notwendig waren richtig zu benutzen bzw. zu initialisieren.

**Vergabe von Namen** Der Proband sah keinen Sinn darin beim Erstellen einer Test-Instanz einen Namen mit anzugeben.

**waitFor** Das Konzept hinter der *waitFor*-Funktion war für den Probanden nachvollziehbar.

**Intention von Funktionen** Der Proband erwartete beispielsweise nicht, dass die Funktion *subscribeUsers* [2] diese auch auf dem XMPP-Server einloggt.

## Schlussfolgerungen

Durch das Experiment wurde deutlich, dass meine bis dahin erstellten Funktionalitäten noch nicht ausreichend gut benutzbar waren. Zum einen das Auffinden dieser, aber auch die Erwartungshaltung die jemand an diese haben entsprachen nicht meinen Vorstellungen. So versuchte ich also die Zahl der statischen Funktionen zu reduzieren und über Hilfsklassen anzubieten.

Da der Proband Funktionalitäten direkt an der Test-Instanz erwartete folgerte ich daraus die Klasse TestSaros als Ort zu nutzen um mein Wissen über die Funktionsweise für das Senden und Empfangen von Paketen und Aktivitäten zu aggregieren. Hieraus entstanden unter anderem die Hilfsklassen *ReceiverHelper* und *SenderHelper* die nun die Funktionalitäten aus ehemaligen statischen Funktionen sammeln.

Um einer leichteren Initialisierung der Testinstanzen gerecht zu werden versuchte ich den Begriff des Testszenarios durch den *ScenarioInitializer* zu etablieren. Hiermit habe ich einen zentralen Ort geschaffen, der deutlich macht, dass mehrere Testinstanzen in diversen Arten von Beziehungen zu einander durch einen Methodenaufruf erstellt werden. Zudem sind die einzelnen Methoden so kurz gehalten, dass man schon durch Überfliegen erkennen sollte, welche Konventionen dabei die Test-Instanzen erfüllen (siehe 5.2.5).

[2] Diese Funktion hatte damals zur Aufgabe mehrere Test-Instanzen auf einem XMPP-Server einzuloggen und diese sich gegenseitig als Freund hinzufügen zu lassen.

## 6.2 Selbstversuch

Für einen geeigneten Selbstversuch habe ich mich der Testfälle bedient die für jeden Release von Saros manuell durchgeführt werden [3]. Ich habe dabei drei Testfälle verschiedenster Kategorien gewählt, welche nicht auf das Frontend der Anwendung angewiesen waren und zudem noch nicht durch andere von mir bereits erstellte Tests abgedeckt waren.

### Aufgbae

Folgende Tests habe ich für den Selbstversuch gewählt:

**Editieren während Einladung** Die Test-Instanz Alice soll Bob zu einer SarosSession einladen. Während Alice Carl zu der Session einlädt soll Bob Textänderungen vornehmen. Am Ende des Testes sollten bei allen drei Test-Instanzen die Änderung von Bob zu finden sein.

**Session wird automatisch beendet** Alice startet eine SarosSession mit Bob. Wenn sich Alice vom XMPP-Server abmeldet sollte Bob an keiner SarosSession mehr teilnehmen.

**Ordner Operationen** In einer Projekt-Struktur mit einem großen Ordner soll Alice diesen Umbenennen und Unterordner in diesem erstellen. Abschließend soll der Ordner von Alice gelöscht werden. Die Erwartung ist die, dass für die Umbenennung des Ordners keine Synchronisation dessen stattfinden soll, sondern der Ordern lokal bei Bob nur verschoben wird. Für die restlichen Aktionen von Alice gilt, dass das diese in der Projekt-Struktur von Bob nachvollziehbar sind.

### Durchführung

Da ich für die ersten beiden Tests die notwendigen Funktionalitäten bereits kannte benötigte ich für das Erstellen beider insgesamt fünf Minuten. Ich konnte hier auf die Funktionalitäten zum Erstellen des Testszenarios zurückgreifen und verwendete ausschließlich *assertThat*-Funktion der Oberklasse *AbstractSarosUnit*.

Für die Realisierung des dritten Tests musste ich mir allerdings zunächst einen Überblick darüber verschaffen, wie ich das Verschieben eines Ordners in der Schicht der Business-Logik zum Ausdruck bringe. Nach kurzer Recherche wusste ich, dass ich hierfür eine *FolderActivity* benötige. Nachdem ich diese richtig initialisieren konnte, habe ich die Hilfsmethode zum Versenden von Aktivitäten an der entsprechenden Test-Instanz verwenden können. Als ich soweit war erste Teile des Test auszuführen erhielt ich eine *NotYetImplementedException* von der *IRessourceFake*-Implementierung. Schnell wurde klar, dass die Methode für das Verschieben von Dateien noch nicht implementiert war. Ich unterbrach also meine Arbeit an dem Test und versuchte nun die notwendige Funktionalität testgetrieben zu entwickeln. Nach dem ich die das Verschieben einer Datei bzw. Ordners implementiert hatte, konnte ich den Rest des Testes ohne weitere Zwischenfälle fertigstellen. Für die Realisierung dieses Tests benötigte ich mehr als 20 Minuten.

[3] Im Projekt verwendet für die Verwaltung dieser Testsuite TestLink - http://www.teamst.org/

## Beobachtungen

Folgende Aspekte wurden bei der Verwendung des Testframworks für die Bewältigung dieser Aufgabenstellung deutlich:

**Framework half** Ich konnte auf viele bestehende Funktionalitäten zurückgreifen.

**Kein langwieriges Debugging mehr** Ich musste nicht via Debugging nachvollziehen wie der Programmablauf genau aussieht.

**Hoher Aufwand** Ich benötigte wieder einmal viel Zeit um mir überhaupt einen Überblick zu verschaffen um den dritten Test umzusetzen.

## Schlussfolgerungen

Aus der Umsetzung dieser Tests ergaben sich keine weiteren Änderungen an dem Framework. Allerdings lässt die jetzige Formulierung des 3. Tests darauf schließen, dass es für zukünftige Tests die dateirelevante Aktionen ausführen sicher wünschenswert wäre, wenn man Funktionalitäten verwenden könnte, mit denen man komfortabler Testdaten erzeugt.

## 6.3 Eigenschaften der Tests

### 6.3.1 Laufzeit

Die Laufzeit aller mit dem Framework erstellten Tests und der Tests die Funktionalitäten des Frameworks selbst testen beträgt ca. 90 Sekunden.

Ein Faktor der sich direkt auf die gesamte Laufzeit auswirkt ist die Wartezeit für das zyklische Prüfen von Bedingungen. Ich habe hier mit verschiedenen Werten jeweils 3 Probeläufe durchlaufen lassen und bin zu folgendem Ergebnis gekommen.

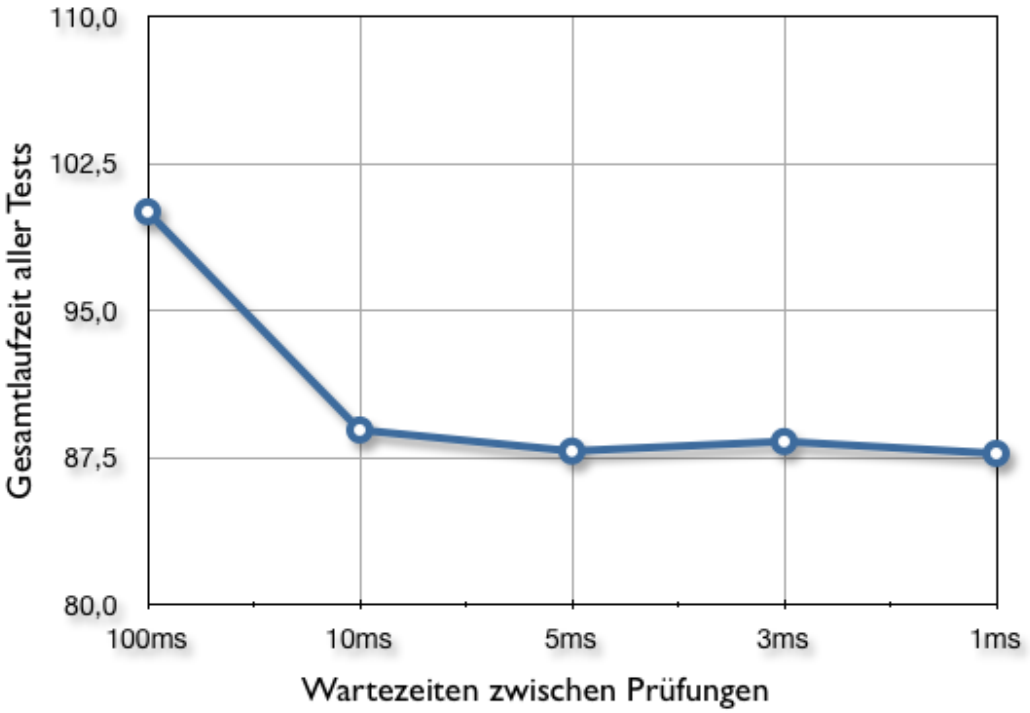


Abbildung 6.1: Auswirkungen der zyklischen Wartezeit auf die Gesamtlaufzeit aller Tests.

Wir wir sehen sinkt die gesamte Laufzeit bei geringeren Wartezeiten. Allerdings stagniert diese ab einem Bereich von 1-5 Millisekunden. Hier ist der Grad an Einfluss des zyklischen Wartens erschöpft. Würde man noch öfter Testen hätte dies sicher negativen Einfluss auf die Ausführungsgeschwindigkeit der Test-Instanzen, da man dann immer mehr Ressourcen der CPU in Anspruch nehmen würde.

Eine Möglichkeit diese Laufzeiten noch zu reduzieren wäre für die einzelnen Szenarien eigene Testklassen anzufertigen die dann nur noch einmal die Initialisierung aller Test-Instanzen vornimmt und pro Test nur noch die Projekte der Test-Instanzen zurücksetzt. Somit würde man sich das wiederholende Erstellen eines Accounts auf XMPP-Server und das gegenseitige Hinzufügen als Freunde ersparen.

### 6.3.2 Lesbarkeit/Nachvollziehbarkeit der Tests

Mit dem erstellten Framework sollte es allerdings nicht nur möglich sein schnell und einfacher Tests zu schreiben sondern auch die entstehenden Tests sollten auch gut nachvollziehbar und lesbar sein.

Betrachten wir zum Beispiel folgenden Test:

Listing 6.3: Üblicher Form eines Test die ich mit dem Framework erstellt habe.

```
1  @Test
2  public void testInviteToSession() throws XMPPException {
3      List<TestSaros> group = sarosSetup().getSarosGroup(2);
4      TestSaros saros1 = group.get(0);
5      final TestSaros saros2  = group.get(1);
6
7      SarosSessionManager sarosSessionManager1 =
8          saros1.getContext().getSarosSessionManager();
9
10     sarosSessionManager1.startSession();
11     sarosSessionManager1.invite(saros2.getMyJID(), "Please␣join.");
12
13     assertThat(new ValueProducer<Integer>() {
14         public Integer value() throws Exception {
15             SarosSession session2 = saros2.getCurrentSession()
16             return session2.getParticipants().size();
17         }
18     }).isEquals(2);
19
20 }
```

Dieses Beispiel stellt die gängigste Struktur eines Unit-Tests dar. Zu Beginn wird das Szenario definiert, dann wird die zu testende Funktionalität ausgeführt und anschließend wird eine Erwartung über das Ergebnis formuliert.

Zum einen kann man bereits beim Lesen des Tests erkennen was passieren soll und zum anderen kann man schnell nachvollziehen um welche Funktionalität denn hier getestet werden soll.

Betrachten man nur den Teil der Erwartung und blendet man Wörter aus die durch die Syntax von Java vorgegeben sind, so ergibt sich beinah ein Satz was sich auch positiv auf die Verständlichkeit der Tests auswirken sollte.

Auch wenn die Menge der „störenden“ Wörter ein wenig groß erscheint, so würde ich diese Art der deklarativen Schnittstelle trotzdem noch als Fluent Interface bezeichnen, welches sich beider Ansätze der Methodenverkettung und einer etwas anderen Version der Funktions-Verschachtelung bedient.

```
assertThat(new ValueProducer<Integer>() {
    public Integer value() throws Exception {
        SarosSession session2 = saros2.getCurrentSession()
        return session2.getParticipants().size();
    }
}).isEquals(2);
```

Abbildung 6.2: Blendet man die durch die Syntax von Java erforderlichen Wörter aus erhält mein satzähnliche Formulierung des Vorgangs.

### 6.3.3 Testabdeckung

Insgesamt habe ich während der Arbeit über 50 Tests verfasst. Die gering scheinende Zahl dieser Tests verschleiert allerdings den Aufwand der damit verbunden war diese überhaupt zur erfolgreichen Ausführung zu bringen [4]. Allerdings mögen sie vielleicht auch ein Anzeichen dafür sein, dass ich in meinen Schritten zu Groß vorgegangen bin.

Die Zahl der Tests genügen um insgesamt ca. 24% Testabdeckung[5] auf Zeilenebene zu erreichen. Vormals waren es ca. 7% auf Zeilenebene.

Wie wir in Abbildung 6.3 sehen wurde in den Paketen *ui* und *videosharing* nicht nennenswert Programmcode ausgeführt, was auf Grund des Fokus der Unit-Tests durchaus einleuchtet, denn die Ausführung von Programmcoude des Frontends gehört nicht zu dessen Bereich.

Um also die Testabdeckung für die Anwendungslogik exklusive dieser Frontend-lastigen Pakete zu ermitteln, müssen wir die Menge von Zeilen dieser Pakete von der Gesamtzahl der Zeilen abziehen. Übrig bleiben dann von den insgesamt ca. 83.000 Zeilen Saros-Logik ca. 57.000 Zeilen[6]. Bei einer Testabdeckung von 24% über das ganze Projekt hin macht das ca. 20.000 Zeilen Programmcode die durch die Unit-Tests ausgeführt werden.

Exklusive der Pakete *ui* und *videosharing* kommen wir so auf eine Testabdeckung von ca. 35% für den Bereich der Business-Logik.

[4] An dieser Stelle sei noch einmal auf die Fallstudie aus Abschnitt 6.1 verwiesen. Der Proband musste damals die gleichen Erfahrungen im Umgang mit der Anwendungslogik von Saros machen.

[5] In diesem Fall sollten wir Testabdeckung als die Menge der Code-Zeilen verstehen die durch die Testsuite ausgeführt wurden. Davon ausgeschlossen sind derzeit alternative Programmflüsse oder gar das Provozieren von Fehlern.

[6] Diese Zahl könnte auch weiter durch das Herausnehmen von Frontend-spezifischem Programmcode minimiert werden. Ein gutes Beispiel wäre das Paket *editor*. Allerdings liegt hier bereits einige Programmlogik, die man mit zur Business-Logik zählen muss.

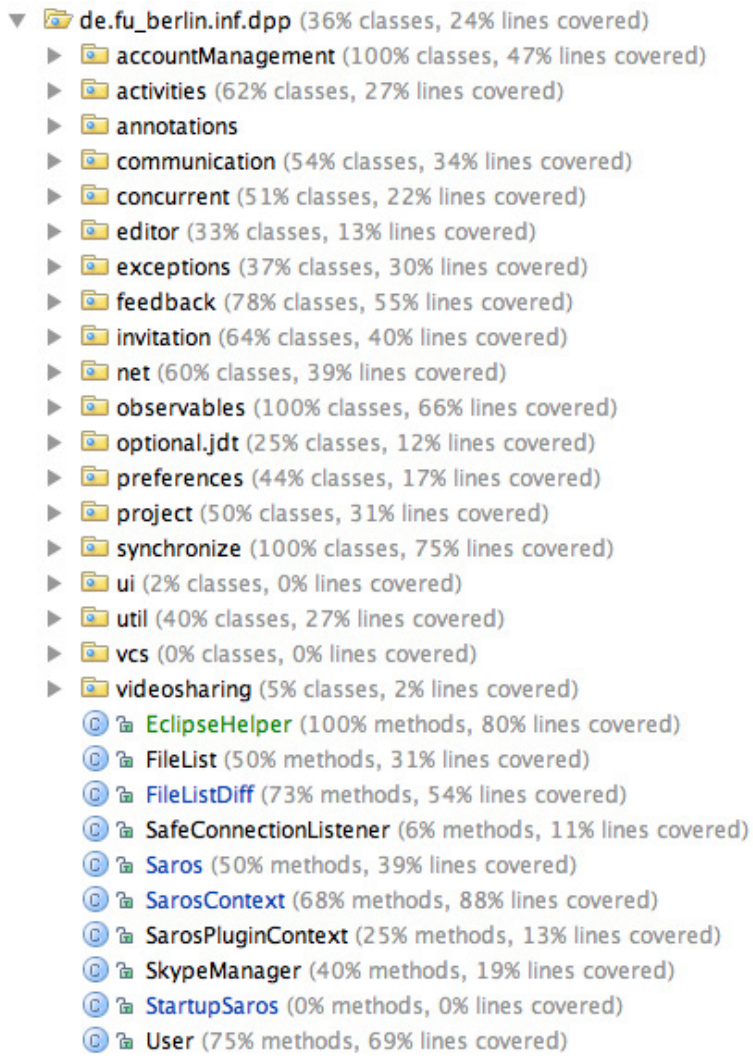


Abbildung 6.3: Testabdeckung in den einzelnen Paketen

### 6.3.4 Offene Punkte

Um mein Ziel der Testbarkeit von Saros überhaupt in dem zeitlichen Rahmen zu erreichen musste ich mich auf dem Weg dorthin von Ballast befreien. So habe ich leider an mancher Stelle nicht die idealste Lösung realisieren können.

Ein Beispiel ist die Ausführung von Frontend-spezifischem Programmcode. Da es keine etablierte Schicht für diese Funktionalitäten gibt musste ich an Stellen an denen es nicht möglich war die Logik zu überschreiben die Ausführung abhängig von der Kondition machen ob das Frontend überhaupt aktiviert ist. Vorrangig statische Funktionen sind hiervon betroffen. Ein gutes Beispiel sind hierfür die Funktionen der Klasse *EditorAPI*. Auf Grund der vielen statischen Funktionen ist diese Klasse im Kontext nicht ersetzbar. Allerdings wäre es wünschenswert wenn genau dies möglich wäre. Man könnte so für Tests ein Fake-Objekt bereitstellen. Ich sehe in dieser Klasse das große Potential die meisten der Frontend-spezifischen Funktionalitäten zu abstrahieren. Man sollte also versuchen die statischen Funktionen zu Methode umzuwandeln und die Verwendung von *Saros.isShellAvaible* zu minimieren.

Gleiches gilt auch für die Funktionalitäten die auf Datei-Ebene arbeiten. Hierfür ist überwiegend

die Klasse *FileUtils* zuständig.

Außerdem habe ich gewisse Bereiche der Anwendungslogik konfigurativ von der Ausführung ausgeklammert. So kommt beispielsweise die Logik für die Synchronisation über die Versionskontrolle nicht zur Ausführung. Als Verbindungstyp für die Übertragung von Daten wird bislang nur der IBB-Channel verwendet. Die Synchronisation ganzer Projekte scheitert bislang noch wegen ungültig übertragender ZIP-Archive.

Das explizite Herunterfahren des XMPP-Servers ist ebenfalls noch nicht realisiert [7].

Des Weiteren gilt es noch die Änderungen die ich an dem Produktcode durchführen musste einer Review durch das Team zu unterziehen.

[7] Bislang gab allerdings keine negativen Effekte dadurch. Der Openfire-Server zeigte sich demgegenüber als robust aus.

## 6.4 Zusammengefasst

Als Resultat dieser Arbeit liegt dem Projekt Saros nun die Möglichkeit bereit schnell und einfach Unit-Tests zu schreiben die es ermöglichen weite Teile der Business-Logik zur Ausführung zu bringen.

Als wichtigstes Teilergebnis ist dabei der SarosContext entstanden der zudem auch Auswirkungen auf zukünftige strukturelle Entscheidungen haben wird.

Die abstrakte Klasse *AbstractSarosUnitTest* ein gelungener Einstieg um komplette Szenarien zu erstellen und sich nicht über den notwendigen XMPP-Server Gedanken zu machen. Neuen Projektteilnehmern sollte dies und die etablierte DSL, sprich die Hilfsklassen *ReceiverHelper* und *SenderHelper* oder auch die *assertThat*-Funktion zum Formulieren von Erwartungen die asynchrones Testen erleichtern, es somit wesentlich leichter fallen sich in der Anwendungslogik von Saros zurecht zu finden und mit dieser zu experimentieren.

Viele der jetzt zur Verfügung stehenden Funktionalitäten waren sehr aufwändig zu etablieren, da es auch für mich als Einsteiger in dem Projekt gerade zu Beginn sehr schwer war die Abläufe zu verstehen.

Im übrigen konnte ich im Verlauf meiner Arbeit die meiste Zeit die Entwicklungsumgebung IntelliJ Idea verwenden, was deutlich machen sollte, dass wir hiermit einen großen Schritt in Richtung Unabhängigkeit von Eclipse gemacht haben.

Alles in allem konnte ich mit dieser Arbeit einen wichtigen Schritt machen um nun endlich wieder bequemer Unit-Tests für das Saros-Projekt schreiben zu können. Im Grunde steht dem Schreiben von neuen Tests nur noch der Aufwand im Weg, den man benötigt um bestehende Funktionalität im Nachhinein zu verstehen und richtig zu benutzen. Ich hoffe, dass hiermit die Voraussetzungen für testbasiertes Arbeiten im Projekt Saros geschaffen wurden und man vielleicht irgendwann so weit gehen kann eine Prozessregel daraus zu machen, dass für jede Review einer neuen Funktionalität ein entsprechender Unit-Test mit abzuliefern ist.

# Abbildungsverzeichnis

# Literaturverzeichnis


[1] Beck, Kent: *Test-Driven development by Example*. Pearson, 2004.

[2] Cordes, Philipp: *Automatisierte Generierung von deklarativen Frameworks - Modellgetriebene Realisierung von Fluent Interfaces*. Januar 2009.

[3] Evans, Eric: *Domain-Driven Design - Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004.

[4] Fowler, Martin: *Fluent Interface*, Dezember 2005. http://martinfowler.com/bliki/FluentInterface.html Zuletzt aufgerufen am 04.04.2011.

[5] Fowler, Martin: *Refactoring. Oder wie Sie das Design vorhandener Software verbessern*. Addison-Wesley, 2005.

[6] Fowler, Martin: *Domain-Specific Languages*. Addison-Wesley, 2010.

[7] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, Munchen, Juli 2004.

[8] Jeff McAffer, Jean-Michel Lemieux: *Eclipse Rich Client Platform*. Addison-Wesley, 2007.

[9] Jez Humble, David Farley: *Continuous delivery: reliable software releases through build, test and deployment automation*. Pearson Education, 2010.

[10] Meszaros, Gerard: *Mocks, Fakes, Stubs and Dummies*, Februar 2009. http://xunitpatterns.com/Mocks,%20Fakes,%20Stubs%20and%20Dummies.html Zuletzt aufgerufen am 04.04.2011.

[11] Nichols, Curtis, Dixon: *High-latency, low-bandwidth windowing in the jupiter collaboration system.* In UIST '95: Proceedings of the 8th annual ACM symposium on User interface and software technology, pages 111-120, New York, NY, USA, 1995.

[12] Peter Saint-Andre, Kevin Smith, Remko Troncon: *XMPP: The Definitive Guide*. O'Reilly, 2009.

[13] Riad Djemili, Christopher Oezbek, Stephan Salinger: *Saros: Eine Eclipse-Erweiterung zur verteilten Paarprogrammierung*. Software Engineering 2007 - Beitrage zu den Workshops, Gesellschaft fur Informatik, Marz 2007. http://www.inf.fu-berlin.de/inst/ag-se/pubs/saros-2007.pdf.

[14] Szucs, Sandor: *Behandlung von Netzwerk- und Sicherheitsaspekten in einem Werkzeug zur verteilten Paarprogrammierung*. Februar 2010.

[15] Wolff, Eberhard: *Spring 2 - Framework fur die Java-Entwicklung*. dpunkt.verlag, 2007.