Freie Universität Berlin

Einführung eines Testprozesses

Diplomarbeit am Institut für Informatik der Freien Universität Berlin,

Arbeitsgruppe Software Engineering

Lin Chen

Matrikelnummer: 4002161

lchen@zedat.fu-berlin.de

Berlin, den 17. April 2011

Eingereicht bei: Prof. Dr. Lutz Prechelt

Betreuer: Karl Beecher und Stephan Salinger

Eidesstattliche Erklärung

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

Berlin, 17. April 2011

Lin Chen

Inhaltsverzeichnis

1 Einleitung 5

Problemstellung....6

Zielstellung....7

Vergehensweise....7

2 Software -Test 11

Warum Überhaupt Testen?....12

Teststufen....13

GUI-Tests....16

3 SAROS 21

Einführung Von Saros....22

4 Ist-Analyse 29

Der Saros Entwicklungsprozess....30

Schwachstellen-Analyse....32

Bisherige Infrastruktur Und Tools....36

Zusammenfassung....42

5 Erstellung einer Soll-Konzeption 43

Automatisierter Testprozess....44

Umsetzungsstrategien....45

Voll Automatisierung....52

6 Ist-Analyse von STF 57

Hintergrund....58

Was Ist STF?....61

Einführung Von SWTBot....63

Verteilung Und Synchronisation....69

Zusammenfassung....77

7 STF Reise 78

Page-Object Als Designpattern....79

Bots Familie....87

SarosSWTBot....88

RemoteBot....92

SuperBot....99

Zukunft....112

8 Fazit 115

Wir Sind Auf Dem Richtigen Weg!....116

Literaturverzeichnis 121

Anhangsverzeichnis 125

1 Einleitung

Problemstellung

Das Softwareprojekt Saros wird von der Arbeitsgruppe Software Engineering des FB für Informatik der Freien Universität entwickelt. Saros ermöglicht die Umsetzung der Entwicklungsmethoden Distributed-Party Programmierung, wodurch zwei oder mehrere Entwickler kollaborativ am selben Software-Projekt von verschiedenen Orten aus über das Netzwerk zusammenarbeiten können. Dieses Projekt ist im Rahmen einer Diplomarbeit im Jahre 2006 [Dje06] entstanden und wurde seitdem stetig weiterentwickelt. Seit meinem Eintritt bei Saros bezieht sich die Entwicklung nicht mehr nur auf die technische Verbesserung, sondern werden im Rahmen der Hauptfunktionalität auch noch zusätzlich In-Built Methoden wie IM (Instant Messaging), VoIP (Voice over IP), Remote-Screen-Sharing und Whiteboard entwickelt, die zur audiovisuellen Kommunikation beitragen. Eine große Änderung findet auch im Hinblick auf die Saros-Benutzeroberfläche statt. Bei fast jedem Release präsentierte Saros sich in einem neuen schicken Look, der nicht nur gut aussieht, sondern auch höhere Benutzerfreundlichkeit bietet. Das bringt gewaltige Mengengerüste an Anforderungen, Code und Testfällen hervor und erhöht somit die Komplexität der Software. Es kann schnell passieren, dass unerwünschte Seiteneffekte nach Codeeingriffen integriert werden. Unentdeckte Softwarefehler können durch die Vielzahl von Erweiterungen noch tiefer in den Code-Ozean versinken, wodurch es sich schwer beurteilen lässt, ob die Fehler wegen der neuen Änderungen hinzugekommen oder bereits in früheren Projekt-Phasen entstanden sind. Dadurch ist der Aufwand der Fehlerbehebung wesentlich gestiegen. Wenn der ganze Entwicklungsprozess nicht mit einem angemessenen Testprozess ausgestattet wäre, würde man schnell das Selbst-Vertrauen in das entstandene Softwareprodukt verlieren. Nach jedem Release des Produkts werden dann frei nach dem Motto “Wird schon gut gehen!” die Daumen gedrückt. Deswegen ist es dringend notwendig einen strategischen Test-Prozess einzuführen, um weiterhin Softwarequalität zu garantieren.

Zielstellung

Im Rahmen der Arbeit soll die Ist-Situation bzgl. des Testens des Saros-Projektes evaluiert und anschließend Saros mit einem angemessenen Testprozess ausgestattet werden. Dabei soll untersucht werden, welche Arten von Tests sinnvoll eingesetzt werden können. Ein weiterer Schwerpunkt der Arbeit liegt darin, dass GUI basierte Test-Framework STF (Saros Test Framework) weiter zu entwickeln, das im Jahre 2010 vom Absolventen Sandor Szücs durch seine Diplomarbeit [Szü10] eingeführt wurde.

Vergehensweise

Zunächst wird auf das Thema Softwaretest im Allgemeinen eingegangen. Es werden verschiedene Testarten wie Modul-, Integrations- und Regressionstest erläutert. Darauf folgt eine Vorstellung über GUI-Test, die als Theoriegrundlage für die spätere Einführung von STF dient.

Anschließend wird die Software Saros vorstellt. Da in später vorgestellten Testprozessen und STF sich auf Saros bezieht, ist ein grundlegendes Verständnis über diese Software notwendig. Aus diesem Zusammenhang heraus wird erläutert, wie die aktuelle Test-Situation von Saros aussieht und warum und an welcher Stelle sie verbessert werden soll.

Der praktische Teil dieser Arbeit enthält die Erweiterung und Implementierung des auf dem GUI basierten prototypischen Test Framework STF. Es wird zuerst gezeigt, wie STF entstanden ist und wie es funktioniert. Aus dieser Kenntnis heraus beginnt die “Reise”, die vier Umgestaltungen der Entwurfsmuster von STF vorstellt, die zum stabileren und leichter zu bedienenden Framework führt.

Zum Ende dieser Arbeit wird ein Fazit gezogen und ein Ausblick auf die Zukunft dargestellt.

SOFTWARE TEST

2 Software -Test

„Softwareentwicklung ohne Tests ist wie Klettern ohne Seil und Haken. [...] Stellen Sie sich einen Kletterer vor, der jeden seiner Schritte durch einen Haken absichert. Mit jedem gesetzten Sicherheitshaken reduziert er ganz bewusst sein Risiko, wie tief er bei einem Fehltritt fallen kann. [...] Der bis zum Haken erkletterte Weg gehört ihm in jedem Fall, selbst wenn ihm ein Fehler unterläuft.“, (vgl. [Wes06: S.4])

Warum überhaupt Testen?

“Das einzige Mittel, den Irrtum zu vermeiden, ist die Unwissenheit.”[1] Für uns bietet diese Erkenntnis kaum eine Möglichkeit, Fehler zu vermeiden. Heutzutage wird Software immer komplexer und unübersichtlich. Hohe Komplexität erschwert dabei das Wissen um die Zusammenhänge und die Auswirkungen von Änderungen am Softwaresystem. Besonders bei der iterativen Softwareentwicklung werden Änderungen an der Software sehr häufig durchgeführt. Wenn in solchen Projekten eine sorgfältig definierte Teststrategie fehlt, ist die Gefahr sehr hoch, dass unerwünschte Seiteneffekte nach Codeeingriffen ins neue Release integriert werden. Solche unentdeckten Bugs werden von Release zu Release immer tiefer im Code versteckt, so dass ihre Behebung vielmehr aufwendiger und teurer wird als am Anfang. Um Vertrauen in das entstandene Softwareprodukt zu schaffen, ist Testen einer der direkten Nachweismittels, indem es Informationen darüber liefert, welche Anforderungen erfolgreich realisiert worden sind.

[1] [Rous62]

Teststufen

(vgl. [Ou87, S.5]; vgl.[Noe06: S.180]; vgl. [Stein05]; vgl. [wiki11]; vgl. [Vig10: S.30])

Fehler entstehen prinzipiell in jeder Phase eines Softwareprojekts, beim Aufnehmen der Anforderungen, beim Erstellen des Designs und bei der Kodierung der Software. Somit ist es sinnvoll, in jeder Stufen passende Tests zuzuordnen, um stufenweise eine Software zu überprüfen, ob sie richtig implementiert und die Anforderung gemäß dem Fachkonzept umgesetzt wurde. Darunter fällt die Funktionsfähigkeit einzelner Komponenten bis hin zum fehlerfreien Zusammenspiel derer im Gesamtsystem. Auf dieser Grundlage werden die Tests untergliedert in Modultests, Integrationstests, Systemtests und Abnahmetests. In diesem Kapitel wird auf verschiedene Teststufen eingegangen, die für Saros sinnvoll erscheinen.

Modultests

Modultests werden üblicherweise von den Entwicklern durchgeführt. Dabei handelt es sich um Tests kleiner abgeschlossener Funktionen. Die Tests werden automatisiert ausgeführt und können jederzeit wiederholt werden. Mit Hilfe von Modultests wird die mögliche Funktionsfähigkeit geprüft. Für die testgetriebene Entwicklung werden die Tests bereits vor der Implementierung geschrieben. Dazu werden zuerst die leeren Klassen, Schnittstellen und Methoden für die Applikation programmiert. Mit Hilfe dieser werden die Testfälle erstellt. Daraufhin werden die Testfälle ausgeführt und die Implementierung so lange angepasst, bis die Test erfolgreich abschließen. Ist dies der Fall, werden die Quellen auf der Entwicklungsumgebung freigegeben.

Integrationstest

„Die Erfahrung lehrt, dass funktionierende Einzelteile keineswegs immer ein funktionierendes Ganzes ergeben.“ [2]

Nachdem die einzelnen Module getestet wurden, wird die Kompatibilität verschiedener Module zu- und untereinander getestet. Die einzelnen Features werden vernachlässigt und das Zusammenspiel wandert in den Betrachtungsraum. Es entstehen sogenannte Subsysteme bzw. ein einheitliches Gesamtsystem. Sukzessive werden die Einheiten hinzufügt. Dadurch wird die Überprüfung der korrekten Interaktionen der einzelnen Module ermöglicht.

Systemtest

Der Systemtest ist die Teststufe, bei der das gesamte System gegen der gesamten Anforderungen (funktionale und nicht funktionale Anforderungen) getestet wird. Gewöhnlich findet der Test auf einer Testumgebung statt und wird mit Testdaten durchgeführt. Die Testumgebung soll die Produktivumgebung des Kunden simulieren. In der Regel wird der Systemtest durch die realisierende Organisation durchgeführt.

Regressionstest

Ein Regressionstest ist ein Test, der nach Änderungen am System zu wiederholen ist. Damit wird sichergestellt, dass es keine Seiteneffekte gibt. Zumindest nicht dort, wo wir vorher bereits getestet haben. Die verschiedenen Testarten können prinzipiell alle für einen Regressionstest eingesetzt werden. Gerade in der iterativen Softwareentwicklung ist der Druck enorm hoch, diese Tests zu automatisieren. Es ist hierbei besonders wichtig, dass die Regressionstests auch wirklich immer wieder laufen. Nur so erreichen wir die angestrebte Qualität in unseren Softwareprodukten. Zusätzlich zum

[2] Alper, 1994, Seite 102

Aufwand für die Ausführung des Tests sind daher immer wieder Anpassungen der Testfälle an Veränderungen am Testobjekt notwendig, um Regressionstests stets ausführbar zu haben.

GUI-Tests

(vgl. [Köh10]; vgl. [Coh10 S.342]; vgl. [Vig10: S.251-256]; vgl. [laut03])

Für die Applikationslogik sind Automatisierungen des Testens bereits verbreitet. Die graphische Oberfläche (GUI) der Anwendung bietet genauso Potential für die Automatisierung der Tests. Die graphische Oberfläche einer Anwendung sollte besonders intensiv getestet werden, denn sie gilt als "Face to Costumer", da sie die Anwendung nach außen präsentiert, jedoch nur eine Hülle der gesamten Anwendung mit ihren Funktionalitäten ist. Trotzdem testen GUI-Tests nicht allein die graphische Oberfläche, sondern aus Sicht des Anwenders das System als Ganzes.

Für GUI-Tests sind drei Testarten üblich. 1) Konsistenzchecks testen, ob die Benutzeroberfläche der Anwendung mit der Spezifikation übereinstimmt. 2) Plausibilitätschecks untersuchen, ob die Reaktionen auf Aktionen gemäß der Spezifikation erfolgen und 3) bei datengesteuerten Akzeptanztests bekommt die Anwendung fixe Eingabedaten, deren berechnete Werte der GUI mit den erwarteten übereinstimmen müssen.

Zur Durchführung der Tests sind zwei Testkonzepte verbreitet, die sich mit Testwerkzeugen realisieren lassen. Mit dem Capture/Replay-Verfahren werden Benutzerinteraktionen zunächst aufgezeichnet und können dann wiedergegeben und als Testfälle verwendet werden. Ein weiteres Konzept stellen die GUI-Unit-Tests dar, die sich an klassischen Unit-Tests orientieren und graphische Oberflächen in kleine Einheiten unterteilen, die systematisch getestet werden.

Was bringen GUI-Tests?

Automatisierte GUI-Tests stellen ein zuverlässiges Mittel dar, die Qualität in der Softwareentwicklung zu erhöhen. Insbesondere wegen der Tatsache, dass die GUI von den Benutzern als das System selbst wahrgenommen wird, führt zu der Erkenntnis, dass die Sicherung der Qualität dieser Schnittstelle nicht mehr nur Testbenutzern überlassen werden kann, sondern mit programmatischen Mit-

teln erweitert werden muss. Das beste System ist wertlos, wenn es beispielsweise aufgrund von unzureichendem Fehlerhandling in der GUI von den Benutzern nicht akzeptiert wird. Mit automatisierten Mitteln kann insbesondere bei Weiterentwicklungen wesentlich gründlicher und öfter getestet werden, als dies manuell möglich ist.

Probleme bei GUI-Tests

Trotzdem dürfen die Kosten und Aufwände für das automatisierte Testen nicht vernachlässigt werden, da GUI-Tests eine ganze Reihe ungünstiger Eigenschaften mitbringt:

- Labilität. Eine kleine Änderung in der Oberfläche kann viele Tests unbrauchbar machen. Wenn dies im Verlauf eines Projekts mehrfach geschieht, besteht die Gefahr, dass die Teams aufgeben und damit aufhören, die Tests nach jeder Änderung der Oberfläche anzupassen.

- Großer Aufwand beim Schreiben. Ein schneller Capture-and-Playback-Ansatz für UI-Tests kann funktionieren, aber solche Tests sind gewöhnlich am instabilsten. Es braucht Zeit eine guten UI-Test zu schreiben, der auch im weiteren Verlauf nützlich und gültig bleibt.

- Zeitaufwand bei der Durchführung. Tests, die in der Benutzeroberfläche ausgeführt werden, erfordern häufig viel Zeit.

Teststrategie

Die Teststrategie kommt daher bei Oberflächentests eine besondere Bedeutung zu: Was wollen wir wie an welcher Stelle testen? Worauf wollen wir uns konzentrieren? Welche Masken testen wir und welche Aspekte darauf? Was können beziehungsweise möchten wir visuell inspizieren? Welche Aspekte testen wir gemeinsam beziehungsweise in einer Abfolge und welche besser isoliert beziehungsweise getrennt? Neben den Antworten auf diese Frage ist das Timing wichtig. Wann beginnen wir mit der Programmierung von Oberflächentests? Darstellungsaspekte ändern sich oft und kurz-

fristig. Erst wenn sich zumindest für einzelne Masken eine Beständigkeit entwickelt hat, ist es sinnvoll, dort und nur dort mit Oberflächentests zu beginnen. Eine Wichtige Information dafür ist es zu erkennen, an welchen Stellen Änderungen häufig auftreten und wo eher seltener. Mit diesem Wissen können wir eine Oberflächenteststrategie entwickeln. Übrigens können Tests zur fachlichen Logik besser über Schnittstellen direkt unterhalb der GUI implementiert und automatisiert werden. Deren Wartbarkeit sollte deutlich besser sein als gegenüber den GUI-Automatisierungen.

TEST PROZESS

3 SAROS

Da nun die benötigten Grundlagenkenntnisse über Tests geschaffen sind, wird in diesem Kapitel das Testobjekt Saros vorgestellt.

Einführung von Saros

Die in der Programmierung sehr verbreitete „Agile Softwareentwicklung“ - beispielsweise Extreme Programming (XP) [Bec99] - ist beim Pair Programming ein unverzichtbarer Bestandteil. Wie der Name Pair Programming verrät, heißt das, dass die Programmierer/innen zu zweit am Code arbeiten und intensiv über den Entwurf diskutieren können. Aber was ist, wenn die wenigen, mit denen man zusammen am Code arbeiten will, zu weit entfernt sind ? Um dies zu ermöglichen, werden Werkzeuge benötigt, die eine virtuelle Arbeitsumgebung schaffen. So sollen die Entwickler, obwohl räumlich getrennt, dieselben Möglichkeiten haben, als wenn sie nebeneinander an einem Computer sitzen. Saros ist genau so ein Werkzeug [vgl. Sal10].

Saros ist ein Plugin für die Entwicklungsumgebung Eclipse und wird als Open Source Projekt von der Arbeitsgruppe Software Engineering des Instituts für Informatik an der Freien Universität Berlin entwickelt. Das Plugin ermöglicht verteilte, kollaborative Softwareentwicklung sowie verteilte Paarprogrammierung innerhalb der Eclipse-Umgebung. Entwickler können gleichzeitig an gemeinsamen Dokumenten über ein Netzwerk arbeiten. Dabei wird der Editor von Eclipse mit verschiedenen Saros-Teilnehmern über das Internet verbunden und es werden Änderungen in Echtzeit gesehen.

Entstanden ist Saros durch Riad Djemili im Rahmen einer Diplomarbeit [Dje06] im Jahre 2006. Es wurde seitdem durch weitere Studien-, Bachelor- und Diplomarbeiten erweitert und verbessert. Seit meinem Eintritt bei Saros bezieht sich die Entwicklung nicht mehr nur auf die technische Verbesserung, sondern es werden im Rahmen der Hauptfunktionalität auch noch zusätzlich In-Built Methoden wie IM, VoIP, Remote-Screen-Sharing und White-Board-Funktonalität entwickelt, die zur audiellen und visuellen Kommunikation beitragen. Eine große Änderung findet auch im Hinblick auf die Saros-Benutzeroberfläche statt. Bei fast jedem Release präsentierte Saros sich in einem neuen “Look”, der nicht nur gut aussieht, sondern auch viel mehr Benutzerfreundlichkeit bietet.

Bei meiner Arbeit geht es um die Verbesserungen des Saros-Testprozesses und die Weiterentwicklung des GUI-basierten Test Framework STF, welches im letzten Kapitel vorgestellt wird. Somit war für mich interessant zu analysieren, wie der Ablauf eines normalen Szenario von Saros aussieht und welche Eigenschaften Saros als Testobjekt hat. Dies würde dann als Bedingung für die STF-Entwurfmaßnahme dienen. Es werden hier erst mal ein paar Szenen[3] aufgenommen, die zeigen wie Saros hauptsächlich funktioniert.

Saros Szenarien

Um gemeinsam über ein Netzwerk programmieren zu können, ist eine gemeinsame Saros-Sitzung aller Teilnehmer erforderlich. Nachfolgend wird der Ablauf eines kleinen Szenarios zwischen den Benutzern Alice und Bob demonstriert, bei dem Alice Bob zu einer Saros-Sitzung einlädt.

1. Vorbereitung

Zur Benutzung von Saros benötigen Alice und Bob jeweils eine Entwicklungsumgebung Eclipse mit installiertem Saros-Plugin. Nach der Installation sind bei einem Neustart die Eclipse Saros-Features zu sehen, die wesentlich aus der View und Menü "Saros" bestehen, mit denen man die Kollaboration verwalten kann. Wobei Saros-View als das Haupt-Control-Panel für shared Projekt Kollaboration dient (Abb. 3.1).

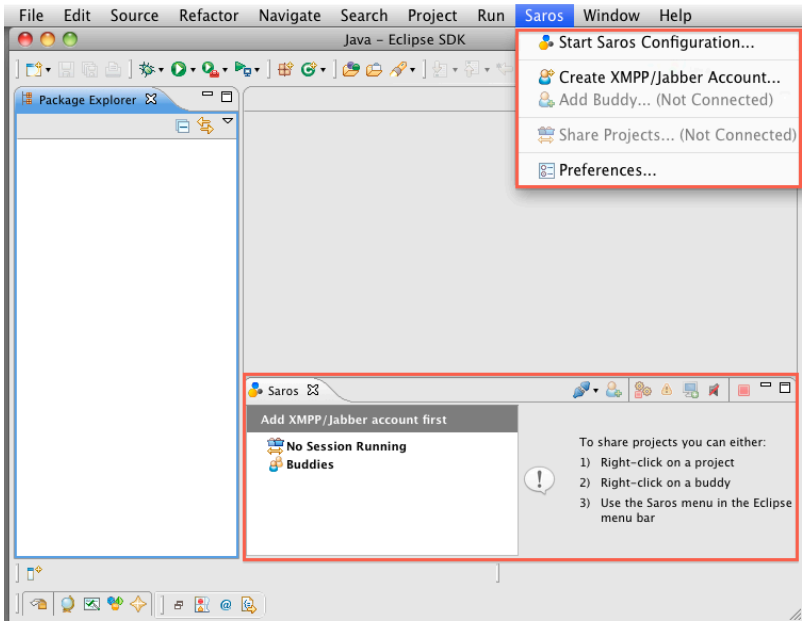


Abb. 3.1: Eclipse-Workbench with Saros-Features

[3] In dieser Arbeit aufgenommen Screenshots entstammen der Saros-Version 11.3.25 unter Mac OS.

2. XMPP/Jabber Account erstellen

Bevor Alice und Bob loslegen können, müssen Sie beim ersten Programmstart erst einmal einen neuen Jabber-Account auf einem XMPP Server anlegen, welcher als eigene Identifikation dient. Das kann durch unterschiedliche Klick-Pfade erreicht werden. Entweder klickt man das Menü `Saros -> Create XMPP /Jabber Account...` in der Menüleiste oder aktiviert das Popupfenster über das Menü `Saros -> Start Saros Konfiguration...`und klickt dann den Button `Create New Account`. Beide Wege lösen das Popupfenster `Create XMPP/Jabber Account` aus, in welchem man die Account-Informationen eintragen und anschließend bestätigen kann (Abb. 3.2).

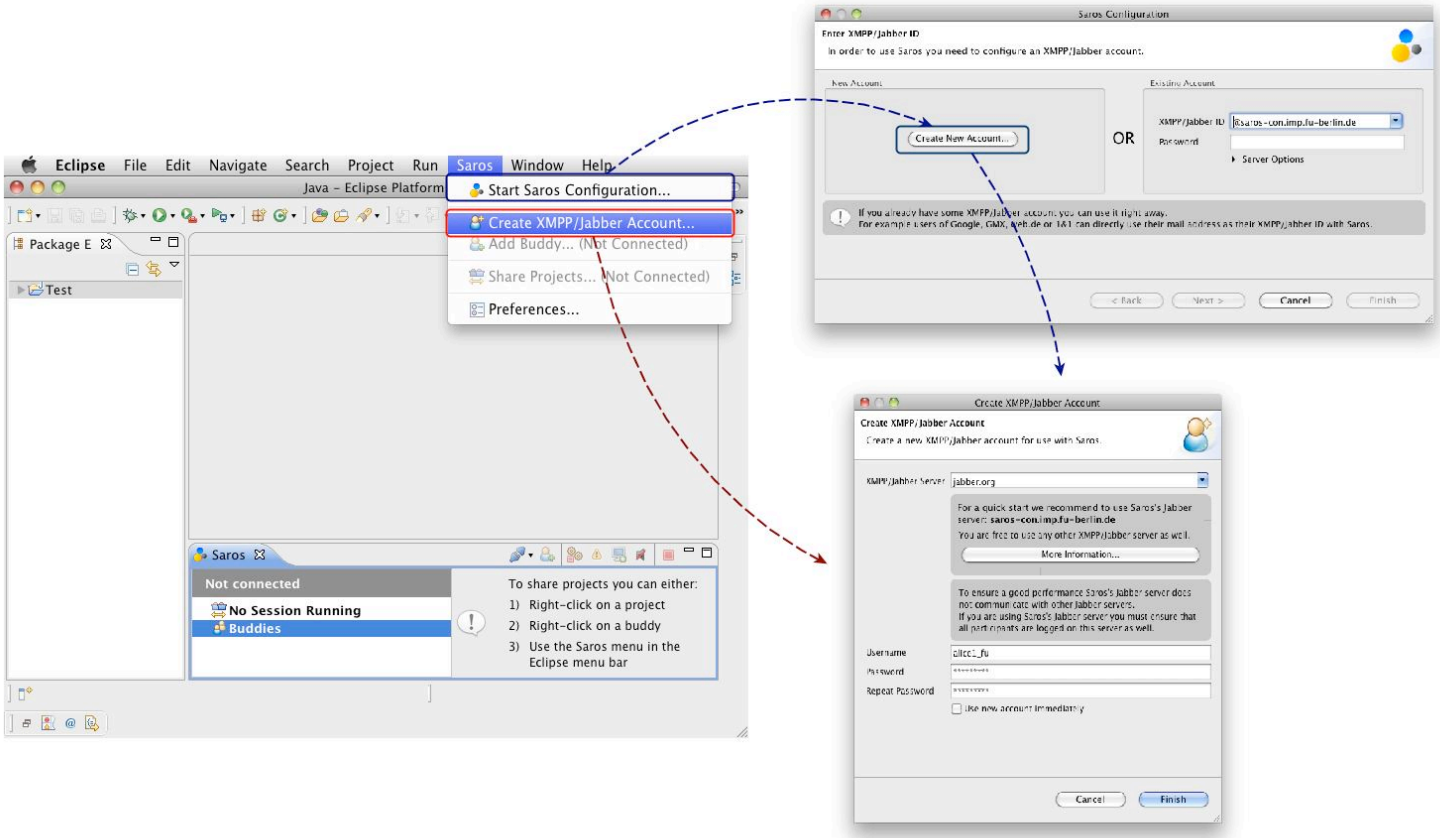


Abb. 3.2: Klick-Pfade bei Erstellung eines neuen Jabber Account

3. Verbindung herstellen und Buddy einfügen

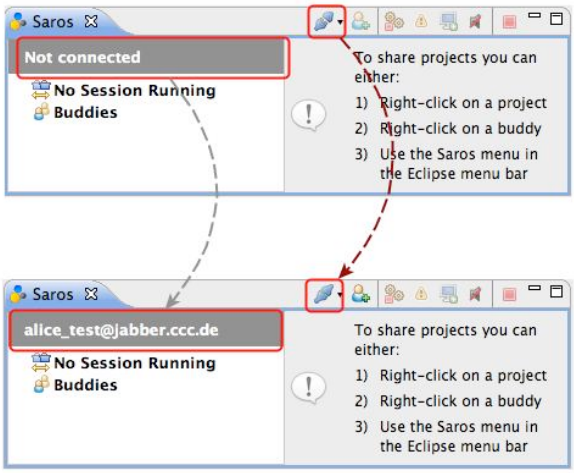


Abb. 3.3: GUI-Element präsentiert die Verbindung

Ähnlich wie bei alle anderen VoIP/Chat-Softwares müssen Alice und Bob online sein und gegenseitig in der Kontaktliste (wird auch Buddiesliste genannt) stehen, bevor sie kommunizieren können. Die Verbindung kann einfach durch das Klicken des Toolbarbutton `Connect` hergestellt werden. Wobei durch die Label mit grau markierter Hintergrundfarbe den Benutzer sofort informieren, ob er gerade online ist und mit welchem Account (Abb. 3.3).

Im Connect-Zustand kann Alice jetzt Bob in der Buddiesliste hinzufügen. Das kann man durch unterschiedliche Klick-Pfade erreichen. Beide Wege lösen das Popupfenster `Add Buddy` aus. Damit kann Alice die Kontakt-Anfrage zu Bob schicken. Nach der Bestätigung des aktivierten Popupfensters bei Bob ist der Kontakt zwischen den beiden hergestellt (Abb. 3.4).

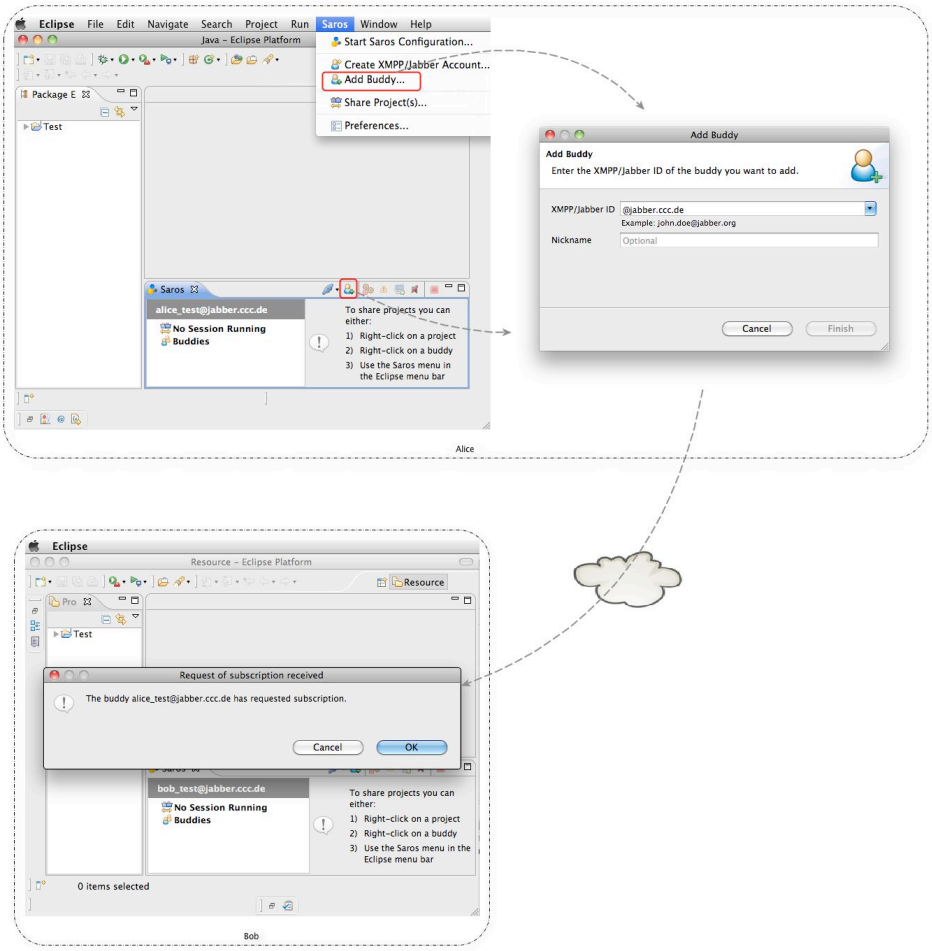


Abb. 3.4: Die möglichen Event-Flows beim Szenario Buddy einfügen

4. Saros-Sitzung erstellen

Alles ist bereit! Jetzt kann Alice endlich eine Saros-Sitzung mit Bob herstellen. Alice braucht nur das Projekt, welches sie mit Bob gemeinsam bearbeiten will zu markieren und auf Kontext `Share With -> bob_test@jabber.ccc.de` zu klicken. Damit wird eine Einladungs-Anfrage zu Bob gesandet. Nach Bestätigung der Anfrage durch Bob wird die Sitzung hergestellt. In der Sitzung können beide nun anfangen den Code gemeinsam zu editieren. Aber wie? Wie bekommt Bob mit, was Alice gerade bearbeitet bzw. an welcher Stelle im Code sie sich befindet? Dafür bietet Saros viele nützliche Awareness-Informationen. Lassen wir zusammen uns erst mal betrachten, wie das Saros-Layout jetzt bei Bob aussieht. Die Saros-View ist auf einmal blau und grün dekoriert, wobei blau für Alice steht und grün für Bob. Die View ist in drei Hauptbereiche aufgeteilt. Das linke Feld enthält die Benutzerinformationen, wer in dieser Saros-Sitzung und der Buddiesliste ist. Auf der rechten Seite liegt der Chatroom, welcher automatisch geöffnet wird, sobald die Saros-Sitzung erstellt wird. Damit können Alice und Bob untereinander Nachrichten austauschen. Die Farben simulieren die Präsenz und Bewegung der Teilnehmer. Damit kann Bob Informationen über Alice ablesen: (Abb. 3.5)

1. Im Package Explorer wird angezeigt, in welcher Datei (mit gelbfarbigem Punkt markiert) sich Alice gerade befindet und welche Dateien (mit Grün markiert) sie geöffnet hat.

2. Wenn Alice etwas geändert hat, bekommt Bob dies durch die entsprechende Farbe angezeigt.

3. Wenn Alice einen Teil des Codes markiert, erscheint eine Blau-

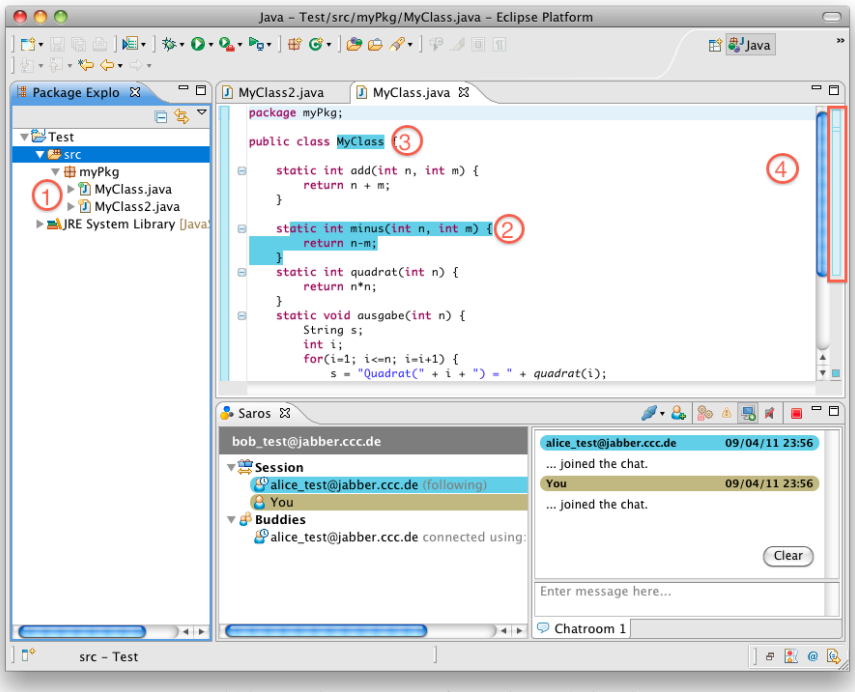


Abb. 3.5: Awareness-Informationen bei Bob

Markierung auch bei Bob.

4. Anhand der Scrollbalken ist jederzeit ersichtlich, welchen Ausschnitt dieser Datei Alice gerade sehen kann.

Saros bietet auch einen sogenannten Verfolgermodus, mit dem es möglich ist, dass Bob genau das angezeigt bekommt, was Alice tut. Dieser Modus wechselt also automatisch von einer Datei zu nächsten oder scrollt simultan mit dem Teilnehmer durch den Quellcode[vgl. Sal10]. Mit dieser Art und Weise hat Saros so eine virtuelle Arbeitsumgebung geschaffen, auf der Alice und Bob problemlos zusammen Code reviewen, diskutieren und programmieren können. (Abb. 3.6)

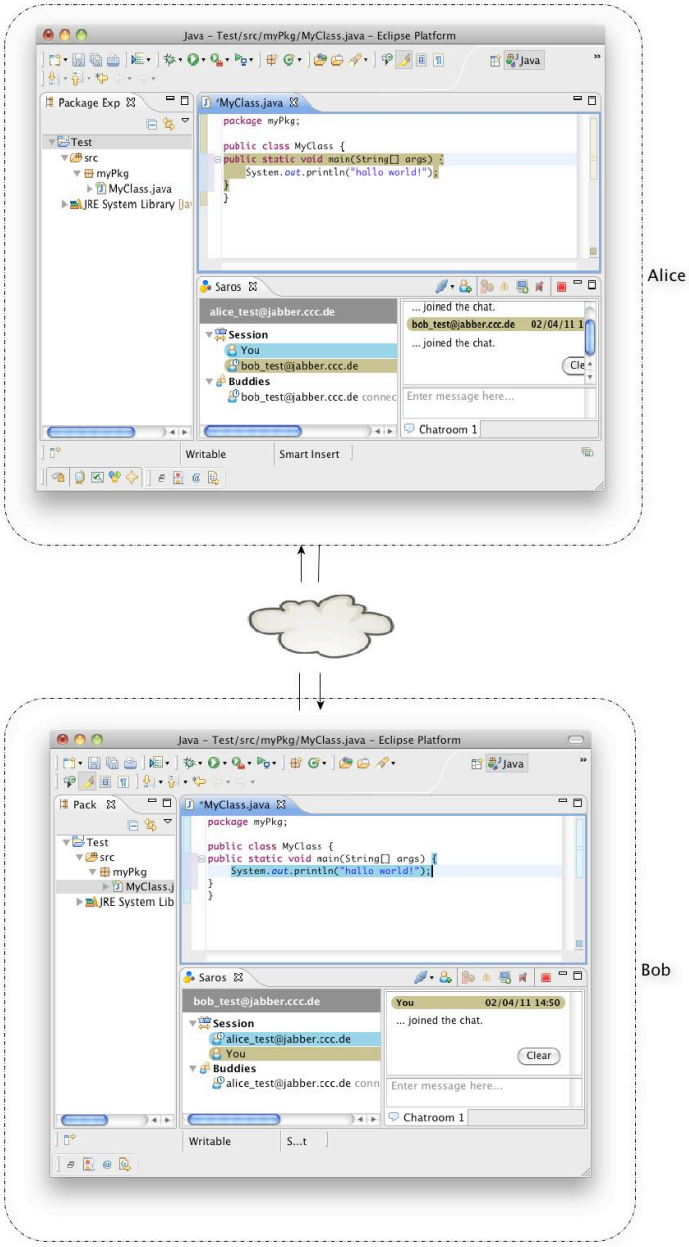


Abb. 3.6: Die virtuelle Arbeitsumgebung in Saros

Fazit

Nach dem oben beschriebenen Ablauf ist deutlich zu sehnen, dass das Einladungsszenario über zwei Saros-Instanzen verläuft. Manche dabei von Alice ausgelöste Ereignisse wurden durch Bob verifiziert. Dazu müssen sie zuvor rechtzeitig an Bob durch das Netzwerk übertragen worden sein. Nur so kann Bob mitbekommen, was Alice gerade tut. Alice und Bob können sich in unterschiedlichen Netzwerk-Umgebungen wie im Internet, hinter einem NAT oder einem Portfilter befinden. Schlechte Leitungsqualität, kabellose Netze oder große Entfernungen können zu hohen Latenzen der Verbindung führen. Um sicherzustellen, dass sich Saros auch in unterschiedlichen Netzwerk-Umgebungen stabil verhält, ist Testen von großer Bedeutung.

4 Ist-Analyse

Um systematisch eine angemessene Prozessverbesserung (SOLL-Konzeption) vorzunehmen, muss zunächst der Ist-Zustand festgestellt werden. An diesen lehnt sich die zu entwerfende Testkonzeption an. Im ersten Schritt wird der allgemeine Entwicklungsprozess von Saros beschrieben. Hierbei wird der zu begleitende Testprozess genau betrachtet und die Schwachstellen analysiert, welche die Neukonzeption des Testprozesses rechtfertigen. Zum Schluss wird noch die vorhandene Infrastruktur vorgestellt und wie diese zum Testprozess beiträgt.

Der Saros Entwicklungsprozess

Es gibt eine Vielzahl von verschiedenen Prozessmodellen die dazu beitragen, eine strukturierte und steuerbare Softwareentwicklung durchzuführen. Je nachdem welches Prozessmodell verwendet wird, sollte man das entsprechendes Testen dem Vorgehensmodell zuordnen. Aus diesem Grund soll hier als erstes der bisherige Saros-Entwicklungsprozess vorgestellt werden.

Saros ist ein studentisches Softwareprojekt und wird durch Studien-. Bachelor- und Diplomarbeiten erweitert und verbessert. Jeder Student im Team arbeitet während eines Zeitraums von 3 bis 6 Monaten an dem vereinbarten Ziel, welches zur Erweiterung und Verbesserung eines bestimmten Teilbereichs des Produkts beiträgt. Das Projekt wird in einem iterativen Prozess entwickelt. Jede Iteration dauert einen Monat und wird in zwei Phasen (Implementierung und Testen) unterteilt. Während der Implementierungsphase werden selten konkrete gemeinsame Ziele vereinbart, sondern die Entwickler arbeiten eher an einem eigenen Teilbereich. Welche neuen Features und Verbesserungen in einem Release eingebracht werden ist davon abhängig, welche Annäherungen eigens gesetzter Teilziele in diesem Release verwirklicht werden (können). Es kann passieren dass manches Release große Änderungen und Fortschritte bietet und manche eher wenige. Neben der eigenen Aufgabe sind Entwickler auch dafür verantwortlich Bugs zu fixen. Kritische Bugs wurden am Anfang der Implementierungsphase vom technischen Leiter dem darauf spezialisierten Entwickler zugewiesen. Es ist zwar wünschenswert, dass alle zugewiesenen Bugs vor dem Release gefixt werden, es kann aber durchaus vorkommen, dass manche Bugs sich wegen des hohen Schwierigkeitsgrades auf mehrere Iterationen beziehen.

Ein Patch (welche klein gehalten werden sollte) muss vor Einbringung in den Programmcode von zwei weiteren Entwicklern gegengeprüft werden. Reviews sind zwar als analytische Qualitätsmaßnahme sehr notwendig, bremsen allerdings auch den Arbeitsfortschritt der Entwickler, da man ständig warten muss, bis die in einer Warteschlange stehende Patches endlich “ge-reviewed” sind. Aus diesem Grund haben manche Entwickler sich dazu entschlossen, erst einmal unter eigenem Branch

zu entwickeln, um dann als Endeffekt einen großen Patch zu erstellen, der “ge-reviewed” werden muss.

Die Test und Release Phase befindet sich in den letzten Wochen einer Iteration. Hierbei werden vier Entwicklern vom technischen Leiter ausgewählt und ihnen die Rollen Testmanager (TM) und Releasemanager (RM) sowie deren jeweiligen Assistenten ATM und ARM übertragen.

RM und ARM sind für den Ablauf des Release-Prozesses verantwortlich. Dazu gehört die Erstellung eines sogenannten “Change-Log”, der alle in diesem Release eingebrachten neuen Features und Verbesserungen enthält, sowie die Vorbereitung eines separaten Entwicklungszweig für Tests und Release und die Veröffentlichung des Releases.

TM und ATM sind eher für den Ablauf der Testphase verantwortlich. Sie führen die Testplanung durch. Dazu gehört der Entwurf und das Schreiben von Testfällen. Testfälle werden dabei basierend auf den “Change-Log” nach Priorität erstellt, welche zentral in dem Test-Management-Tool TestLink verwaltet werden.

Das intensive Testen findet am Mittwoch statt, am dem die von TM und ATM erstellten Testfälle manuell getestet werden müssen. Theoretisch sollten die gesamten Team-Mitglieder an diesem Testtag anwesend sein, um Tests gemeinsam durchzuarbeiten. Aber in der Wirklichkeit habe ich leider noch nie erlebt, dass alle Entwickler anwesend waren. Am Ende des Tages schickt der TM eine Liste, der vor dem Release zu behebender Defekte an alle Teilnehmer. Donnerstags und Freitags müssen alle gefundenen Defekte behoben werden. Am Freitagnachmittag wird noch der letzte Sanity-Test per Hand vor dem Release durchgeführt.

Die rechte Abbildung veranschaulicht nochmals das Prozessmodell bei Saros. Hierbei ist deutlich zu sehen, dass ein Integrationstest nur am Ende einer Iteration stattfindet.

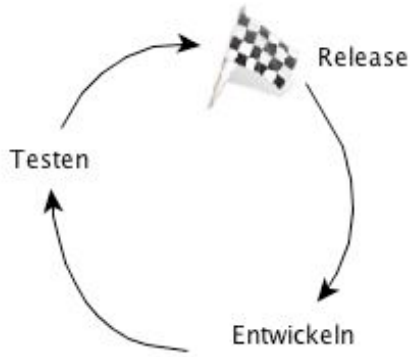


Abb. 4.1: Ist-Prozessmodell bei Saros

Schwachstellen-Analyse

([vgl. Few94 S.9-10]; [vgl. Her09])

Wir haben einen kurzen Entwicklungszyklus, bei dem vor jedem Zyklus ein intensives manuelles Testen stattfindet. Dies sollte eigentlich ausreichend sein, oder? Bevor man diese Frage beantwortet, zeigen ein paar typische Szenen, was während der manuellen Testphasen im Saros-Raum passiert.

Manuelles Testen

Schlechte Ressourcennutzung

Die meisten Testfälle von Saros wie im Kapital Saros geschildert, benötigen mindestens 2 teilweise bis zu 5 Entwickler. Des weiteren ist manuelles Testen eine sehr aufwendige, monotone und fehlerbehaftete Tätigkeit. Nur ein Beispiel: Um Features von Saros zu testen, folgen die Tester immer wieder dem Vorgehensmuster: Verbindung mit anderen Teilnehmern aufbauen, Java-Projekt und -Klassen erstellen, Saros-Sitzung erstellen usw.. Dies führt zu Verschwendung von Personalressourcen.

Ineffizient

Bevor mit den Tests begonnen werden kann muss jeder teilnehmende Entwickler die Testumgebung lokal manuell einrichten. Es kann somit passieren, dass die Testumgebung versehentlich falsch eingerichtet oder falsch konfiguriert wurde. Auch wenn alles klappt ist es die Regel, dass Tests ständig unterbrochen werden. Einerseits, weil z.B. ein Entwickler wegen eines Bug „emotional“ wird und anfängt mit dem Entwicklungsteam zu diskutieren oder zum anderen, weil Unterbrechungen durch Telefon-Anrufe Konzentrationsschwächen hervorrufe und ein erneutes Hineindenken in den Quellcode erschweren. Man spricht in diesem Zusammenhang auch vom sog. „Sägeblatt-Effekt”[4].

[4] http://www.rhetorik.ch/Saegeblatteffekt/Saegeblatteffekt.html

Manchmal braucht man für 5 Tests schon einen ganzen Nachmittag. Effiziente Tests und mehr Abdeckung sind auf diese Weise unmöglich.

Sehr fehleranfällig

Weil Saros eine verteilte Programmierumgebung ist, braucht man schon für den einfachsten Testfall mindestens zwei Saros Instanzen. Manche von einem Tester ausgelösten Ereignisse wurden durch einen anderen verifiziert. Um den inhaltlichen Ablauf genau zu verfolgen, müssen sich Tester gut abstimmen. Aber Menschen sind keine Roboter. Eine exakte Zusammenarbeit verlangt konstant hohe Konzentration und Abstimmung. Es ist ausgeschlossen, dass man über einen so langen Zeitraum fehlerfrei arbeitet.

Schwer reproduzierbar

Beim Testen tritt oft ein Szenario auf, dass manche Bugs nur gelegentlich erscheinen und somit schwer zu reproduzieren sind. Besonders Defekte auf Netzwerkschicht wurden in den meisten Fällen nur bei bestimmten Netzwerkeigenschaften und Betriebssystemen sichtbar.

Gesunkenes Vertrauen

Wegen mangelhafter Testabdeckung und fehlenden Integrationstests tauchen zeitweise unangenehme Überraschungen vor dem Release aus. Es wird laut im Saros Raum: “Bei mir läuft’s aber ” - “Ich habe dort nichts geändert!”. sicherlich müssen vor dem Release solche Fehler behoben werden, aber in so einer angespannten Situation ist es sehr wahrscheinlich, dass das Beheben der Bugs wieder andere Fehler verursacht. Hierfür gibt es leider keinen Regressionstest. Deswegen werden nach jedem Release die Daumen gedrückt, ganz nach dem Motto: “Wird schon gut gehen!”

Review

Bei Reviews sollten Kollegen konkrete Feedbacks geben und codierte Stellen aufdecken, die gegen diese Richtlinien verstoßen, so dass keine unnötigen Fehler in den Source-Code einfließen. Aber das Saros -Entwicklungsteam leidet ziemlich stark unter einer Entwickler Fluktuation. In der Zeit, seit

dem ich bei Saros angefangen habe, sind es insgesamt 5 Studenten, welche sich in unterschiedlichen Phasen befanden. Manche davon sind frisch ins Saros Team eingestiegen, wissen noch nicht wie die Entwicklung bei Saros vorangeht und kennen das Projekt nur vom Hören. Andere von uns sind fast am Ende ihrer Arbeiten und befinden sich gerade in einer anstrengenden Schreibphase. Diese sind zwar im Projekt drin, aber wegen der Doppelbelastung nicht mehr wirklich produktiv. In solchen Situationen ist der Review-Prozess manchmal nicht so gelaufen wie er sollte. Neue Mitglieder sind noch nicht in der Lage, konkrete und hilfreiche Feedbacks zu geben und "alte Hasen" haben dafür einfach keine Zeit. Dies führt dazu, dass Patches nicht rechtzeitig "ge-reviewed" wurden. Somit haben sich Entwickler dazu entschlossen, erst mal unter eigener Branch zu entwickeln, um dann als Endeffekt ein großes Patch zu erstellen. Große Patches sind aber schwer zu reviewen. Deswegen ist die Möglichkeit sehr hoch, dass unentdeckte Bugs in den Source Code eingecheckt werden bzw. schon eingecheckt worden sind.

Kein Regressionstest

Zur Zeit gibt es keinen Regressionstest bei Saros. Das Review ist die einzige Prüfung, die der Code bestehen soll, um in das Programm einzufließen. Es gibt zwar JUnit-Tests, aber diese decken nicht mehr als die Hälfte des Source Code ab. Deswegen haben JUnit-Tests-basierte Regressionstest keine wirkliche Aussagekraft. In der iterativen Entwicklung von Saros wird ständig etwas geändert und erweitert. Die Gefahr hierbei ist sehr hoch, dass unerwünschte Seiteneffekte nach Codeeingriffen ins neue Release integriert werden.

Was schließen wir daraus?

Die Fehlerquellen wie menschliches Versagen, Schwankungen und mangelnde Konsistenz lassen keinen Zweifel daran aufkommen, dass manuelle Prozesse nur eine geringe Chance haben, die schnellen und reproduzierbaren Ergebnisse zu liefern. Ganz zu schweigen davon, dass bei einer großen Code-Basis das manuelle Testen in aller Regel den für diese Iteration bemessenen Zeitrahmen bei weitem sprengt.

Außerdem werden Iterations Test sehr viel seltener ausgeführt als eigentlich empfehlenswert. Somit steigt das Risiko, dass Fehler erst ziemlich spät gefunden werden. Wie ich schon im ersten Kapitel erwähnte ist üblicherweise der Aufwand, einen Fehler zu beheben, umso größer je später ein Fehler gefunden wird. Letztendlich führen lange Testlaufzeiten und die daraus resultierenden Reaktionen zu langsameren Entwicklungszyklen, weniger Flexibilität bei der Entwicklung und einem Produkt mit mehr Fehlern als nötig.

Bisherige Infrastruktur und Tools

Die Infrastruktur für das Testen beinhaltet alle notwendigen Einrichtungen und Mittel, um den Anforderungen entsprechend testen zu können. Test-Umgebungen werden benötigt, um den Test durchzuführen, und Test-Tools bieten zusätzliche Unterstützung [Pol00].

Ein automatisierter Testprozess ist sicherlich wünschenswert für Saros, aber wegen der speziell verteilten Eigenschaft von Saros sind normale Test-Tools wie JUnit nicht ausreichend, so dass die automatisierten Verfahren nicht wirklich im Saros-Projekt integriert werden können. Um sich dem Ziel zu nähern, werden mehrere Forschungsarbeiten vergeben, die unterschiedliche Perspektiven zur Realisierung des automatisierten Testprozesses beitragen. Zu dem Zeitpunkt, als ich angefangen habe, gab es schon Test Infrastruktur und Tools, von denen manche bereits zum Testprozess beitrugen, manche jedoch noch nicht im Prozess integriert waren. In folgendem Subkapitel werden alle vorhandenen Testinfrastrukturen und Tools und deren Zustand vorgestellt.

Test-Tools

Zur Unterstützung aller Phasen des Test-Prozesses sind einige Tools verfügbar. Die wichtigsten darunter sind diejenigen für Planung und Verwaltung, für Fehler-Verfolgung, für automatische und für die automatische Testdurchführung. Hier liste ich alle vorhandenen Tools von Saros auf. Manche sind schon im Einsatz, manche müssen noch weiter entwickelt werden.

Planung und Verwaltung

TestLink

TestLink ist das web-basierende Test-Management-System, welches strukturiertes Software-Testen und die Prozesse dahinter gut unterstützt [Cam10]. Die folgende Tabelle zeigt die Informationen von TestLink, welche wir bei Saros verwenden.

Version	184
Quelle	http://testlink.sourceforge.net
Lizenz	GNU GPL 2.0
Sprache	PHP
Datenbank	MySQL

Unsere Testfälle wurden zentral durch TestLink verwaltet. Während der Release-Woche erstellt der Testmanager einen systematischen Testplan für den Release mit dem Tool TestLink und ordnet dem Testplan die Test Fälle zu, welche für diese Release wichtig sind. Sind diese Vorarbeiten abgeschlossen, kann begonnen werden die Test auszuführen. Die Benutzung von TestLink ist für Einsteiger nicht immer intuitiv und einige der Funktionen sind recht schwer zu finden. Es würde sehr hilfreich sein, ein kurze Einführung über TestLink in Release -Plan einzufügen.

Fehler-Verfolgung

Bug Tracker in Sourceforge

Saros verwendet direkt den in Sourceforge integrierten Bug-Tracker, welcher dafür geeignet ist, Programmfehler zu erfassen und zu dokumentieren. Mit Ihnen werden auch Zustands - oder Feature-Berichte geschrieben. Dieses Tool spielt eine große Rolle während des Entwicklungsprozesses, da damit alle gefundenen Bugs verwaltet wurden.

Automatische Testdurchführung

JUnit

JUnit ist ein Framework zum Testen von Java-Programmen, das besonders für automatisierte Unit-Tests einzelner Units (meist Klassen oder Methoden) geeignet ist. Es ist das Kernprogramm für die Automatisierung. Die in der Zukunft eingesetzten Framework Saros-JUnit-Framework (in meiner Arbeit wird dieses Framework als SJF benannt) und Saros-Test-Framework (STF) werden auf JUnit basieren.

STF

Das Saros Test Framework (STF) basiert auf dem Framework SWTBot[5], einem UI-Testtool für SWT und Eclipse basierter Applikation. Dieses Framework sollte dazu beitragen, den Integrationstest und GUI-Test in den Testprozess zu integrieren. Der große Vorteil an diesem Framework ist, dass Szenarien unter einer konfigurierbaren Testumgebung getestet werden kann und es damit möglich ist, die in der Netzschicht oft auftauchenden Fehler zu identifizieren und zu reproduzieren, was normalerweise bei lokalem Testen unmöglich ist. Wobei konfigurierbar bedeutet, dass man die Testumgebung so konfigurieren kann, dass eine bestimmte Zielumgebung der Anwendung simuliert wird. Sandor Szücs hat das prototypische Grundgerüst von STF bereitgestellt. Die Weiterentwicklung des Framework, um es einsetzbar zu machen, ist Kern meiner Arbeit. Mehr über STF wird im letzten Kapitel vorgestellt.

SJF

[5] http://www.eclipse.org/swtbot/

SJF ist ein mockbasiertes Testframework für Saros und wurde in der Zeit, als ich angefangen habe, von Student Philipp Cordes entwickelt. Anders als STF wird die Netzwerkschicht mit Mocks abstrahiert, so dass für Tests keine Netzwerkverbindung notwendig ist. Es wird dazu beitragen, dass man ohne aufwendiges Setup schnell einfache JUnit Tests erstellen und schnell ausführen kann. Somit wäre ein testbasiertes Arbeiten möglich. Man könnte zum Beispiel etablieren, dass jeder, der Code zum Review abgibt, auch einen Test dazu schreibt. Die Tests können wiederum dazu beitragen um Regressionstest durchzuführen. Bislang ist es noch nicht möglich vor allen Dingen schnell einen Test zu schreiben, was die Leute abschreckt. Würde man das vereinfachen und das Testframework im Team etablieren, könnte man sogar erreichen, dass die Testabdeckung auf einem bestimmten Niveau bleibt.

Hudson

Hudson ist ein CI (Continuous Integration) Server. Dieser sorgt für die regelmäßige Ausführung dieser Integrationstest, etwa nach jeder Codeänderung oder einmal pro Nacht (Nightly Build). Bei neuen Problemen würde sofort Alarm geschlagen. Hudson wurde von Florian Thiel[6] eingerichtet, aber als er das Team verließ, wurde die Sache noch nicht zum Laufen gebracht, so dass der CI-Server nicht zum Testprozess beitrage konnte.

TESTUMGEBUNG

VMware

Um die unterschiedliche Ziel-Umgebung der Applikation zu simulieren und somit die Netzschicht testen zu können, hat Florian Thiel eine Testumgebung gebaut, in der drei virtuelle Maschinen saros-eclipse1, saros-eclipse2 und saros-firewall (Abb. 4.2) installiert sind. Zum Einsatz kommt das Produkt “VMWare Player”. Die Basis-Installationen der einzelnen Betriebsysteme wird von den Technikern zur Verfügung gestellt. Diese Testumgebung ist für STF gedacht, so dass STF-Tests

[6] Florian Thiel war früherer Mitarbeiter der AG Software Engineering

nicht nur unter der lokalen Entwicklungsumgebung, sondern auch unter dieser konfigurierbaren Testumgebung durchgeführt werden können. Wobei Machine saros-firewall die XMPP-Server enthält und der Simulierung von verschiedenen Netzwerkeigenschaften dient. Die andere sind für Zielrechner gedacht, worauf Saros später installiert wird. Für das Durchführen dieser Tests in dieser Testumgebung ist das CI-Tool Hudson verantwortlich, welches auf Machine Saros-Build installiert ist. Zum Testen werden die beiden Machine saros-eclipse1 und saros-eclipse2 jeweils neu gestartet. Darauf wird dann das aktuelle Saros-Plugin installiert und mit den zugewiesenen Ports gestartet. Tests werden dann über die Beiden Rechner durchgeführt.

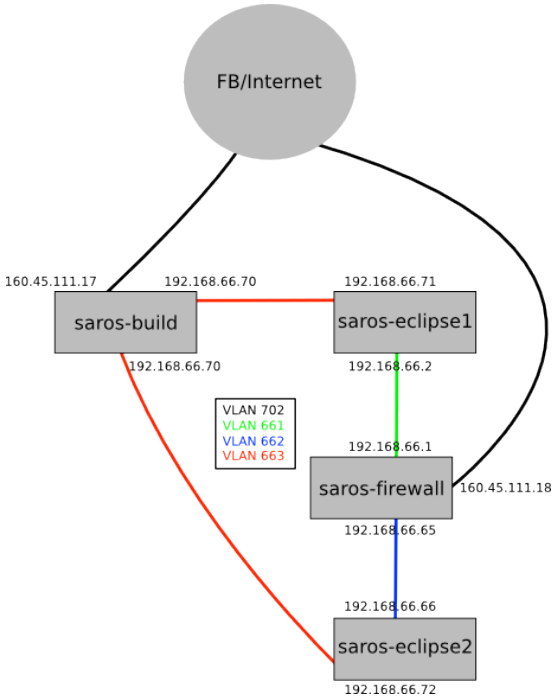


Abb. 4.2: Network Layout. Quelle: [Abb2]

Eigentlich ist die TestUmgebung fast fertig eingerichtet und man kann schon STF-Tests unter der Testumgebung automatisch durch Hudson durchführen lassen. Die Forschung über die Gestaltung der Testumgebung und die Erkenntnis des eigentlichen Ablaufs waren aufgrund der fehlenden Dokumentation sehr zeitaufwendig. Schlussendlich bin ich am Ende zu einfachen Tests wie "eine Verbindung herstellen" in der Testumgebung gelangt.

Nach dem Zwischenerfolg begann ich versuchsweise einen etwas komplizierteren Testfall wie "die Herstellung einer Saros-Sitzung" zu testen. Leider tauchte bei der Durchführung der Tests das Problem auf, dass der Einladungsprozess unendlich lang dauerte. Um die Ursache herauszufinden, habe ich mit Sebastian Bauch[7] alle Möglichen untersucht. Schließlich waren wir sicher, dass Saros unter Ubuntu 64Bit nicht wirklich funktioniert (der gesamte Analyse Prozess wird in Sebastian's Bachelor Arbeit[8] erläutert, weshalb ich hier nicht weiter ins Details gehe).

[7] damalige Team-Mitglieder mit dem Thema *Verbesserung von Saros in unzuverlässigen Netzwerken mit hoher Latenz*

8 http://www.inf.fu-berlin.de/inst/ag-se/theses/Bauch10-saros-verbesserung-netzwerke.pdf

Unsere VMware hat genau dieses OS installiert. Nach langer Diskussion mit unserem technischen Leiter haben wir entschieden, VM noch mal mit anderem OS, nämlich Ubuntu 32Bit, zu bespielen. Nach der Neuinstallation (es hat mehr als zwei Wochen gedauert, bis Techniker die neue Installation durchgeführt haben) scheint es, dass die Verbindung zwischen Saros-Build und Saros-eclipse1/2 nicht mehr existiert. Da diese Schwierigkeiten mehr als einen Monat Zeitverzug verursachten, entschied ich STF-Framework erst mal unter einer lokalen Entwicklungsumgebung zu entwickeln.

Zusammenfassung

Obwohl der Testprozess zur Zeit nur manuell ausgestattet ist, sind wir schon auf dem richtigen Weg zu einem modernen und voll automatisierten Testprozess. Das Diagramm (s. unten) zeigt den momentanen Testablauf und noch nicht integrierte Werkzeuge. Was wir tun müssen ist, die vorhandenen Tools zum Laufen zu bringen und strategisch an die passende Stelle des Test Prozesses einzuordnen. Der mögliche Lösungseinsatz wird dann in der folgenden SOLL-Konzeption genau vorgestellt.

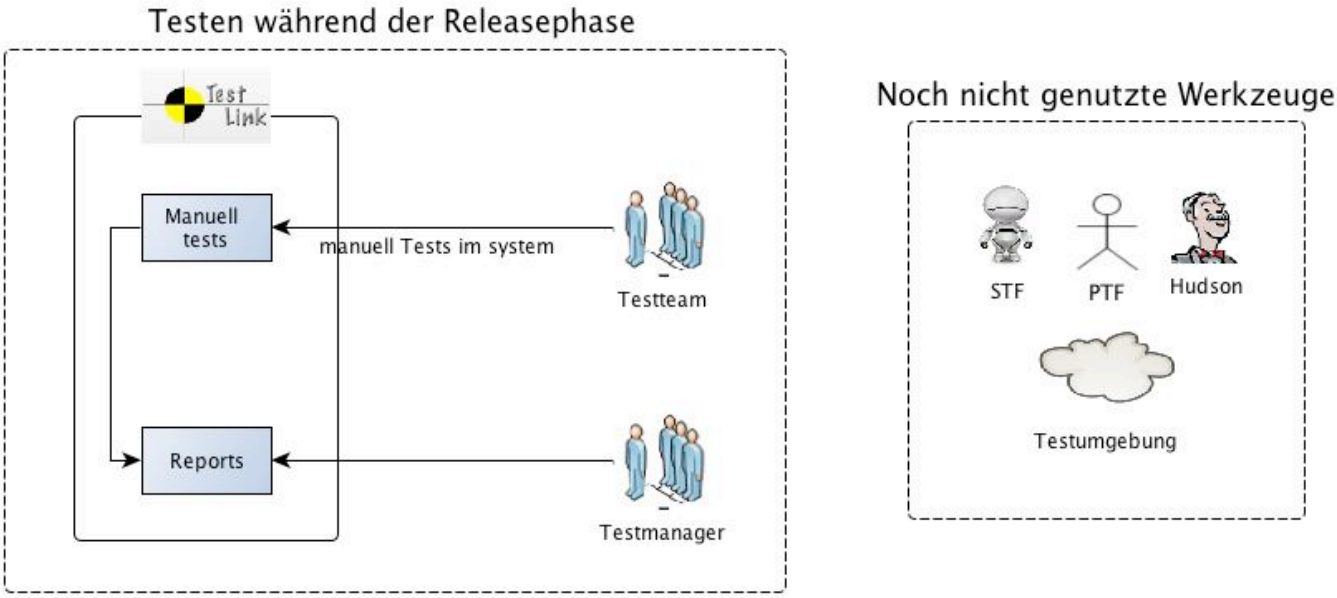


Abb. 4.3: Bisheriger Testablauf und nicht eingesetzte Tools

5 Erstellung einer Soll-Konzeption

Dieses Kapitel enthält die auf der IST-Analyse basierende SOLL-Konzeption von Tests bei Saros. Die Schwächen im Umgang mit den bestehenden manuellen Testprozess sollen dabei durch Automatisierung minimiert werden. Zusätzlich wird genau beschrieben wie sich Entwicklung- und Testprozess miteinander "verzahnen". Im Anschluss wird analysiert, ob andere Technologien zum Einsatz kommen können. Des weiteren werden Verbesserungsvorschläge zum voll automatisierten Testprozess unterbreitet.

Automatisierter Testprozess

Durch die Schwachstellen-Analyse können wir feststellen, dass es vom entscheidender Wichtigkeit ist, einen automatisierten Testprozess in dem Entwicklungsprozess von Saros einzubinden. Die dadurch eingebrachten Vorteile sind erheblich. Fast alle zuvor erwähnten Probleme können durch Automatisierung einfach gelöst werden. Automatisiertes Testen bringt mehr Flexibilität und Reproduzierbarkeit, ermöglicht Regressionstest in jeder Iteration und erleichtert Testern maßgeblich die Arbeit. Somit ist eine Steigerung der Effizienz und der Qualität in der Softwareentwicklung gewährleistet.

Okay; wir haben schon verstanden. Saros braucht ein automatisierten Testprozess. Aber Automatisierung entsteht nicht von allein. Wie kann dies organisatorisch und strategisch in den entstandenen Testprozess eingebettet werden, wie sieht die Vorgehensweise aus, welche Tools können wir dafür benutzen? In welcher Phase werden sie eingesetzt? Welche Regeln sollten wir dabei beachten, so dass die Automatisierung nicht noch mehr Chaos bringt.

Im folgendem Subkapitel wird ein möglicher Lösungsansatz, bezogen auf die vorher gestellten Fragen vorgestellt, der aber wegen des Umfangs und noch nicht vollständig implementierter Infrastruktur in der Praxis nicht eingesetzt und evaluiert wird. Aber ich denke, es ist schon sinnvoll, in meiner Arbeit die von mir gesammelten Erkenntnisse zu kommentieren, so dass Nachfolger sich an diesen Ausgangspunkt orientieren können.

Umsetzungsstrategien

([vgl. Coh10 S.341-344]; [vgl. Her09]; [vgl. Wien07])

Automatisierung auf verschiedenen Ebenen

Es kann schon passieren, dass automatisierte Tests oft erst nach Monaten und sogar Jahren nach der Programmierung eines Features geschrieben ist, wenn die Automatisierung auf der falschen Ebene vorgenommen ist. Eine Wirkungsvolle Testautomatisierung muss auf drei verschiedenen Ebenen stattfinden, wie die in der Abbildung 5.1 dargestellte Testautomatisierungspyramide zeigt.

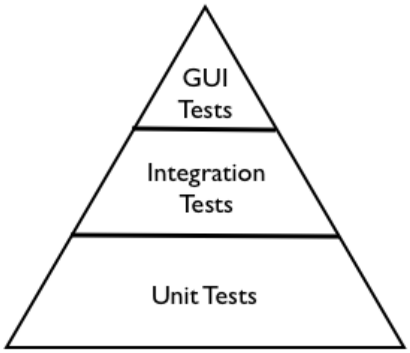


Abb. 5.1: Testautomatisierungspyramide

Unit Tests

Das Fundament dieser Pyramide bilden die Unit-Tests. Sie sollten die Grundlage jeder soliden Vorgehensweise zur Testautomatisierung sein und bilden daher den größten Teil der Pyramide. Für diese Ebene kommen das Test Framework SJF zum Einsatz, welches den Entwicklern ein testbasiertes Arbeiten ermöglicht.

Integration Tests

Die Integration-Tests-Ebene der Testautomatisierungspyramide ist dafür verantwortlich, dass das Systemverhalten zu prüfen ist, unabhängig von der Benutzeroberfläche. Wenn Testfälle auf dieser Ebene erledigt werden können, sollte dies nicht in der Oberfläche der Anwendung durchgeführt werden, weil UI-Test aufwändig zu schreiben, aufwendig durchzuführen und labil sind. In diesem Sinne ist STF gar nicht für diese Ebene geeignet. Aber bezogen auf Saros, eine auf Netzwerk basierte Anwendung, ist es entscheidend wichtig die Fehler während der Interaktion zwischen mehre-

ren Saros-Instanzen in einer bestimmten Ziel-Umgebung aufzudecken, was mit PTF wegen der mit Mocks abstrahierten Netzwerkschicht nicht möglich ist. Im Prinzip können wir für diese Ebene die beiden Frameworks gut kombinieren. PTF wird dann verwendet, wenn die Netzwerkschicht eines Testfall nicht im Vordergrund steht, andernfalls kommt STF zum Einsatz.

GUI Tests

Die automatisierten Tests der Benutzeroberfläche befinden sich an der Spitze der Pyramide, da wir wegen Labilität und zu hohen Aufwand wenig davon durchführen möchten. An der Ebene der Benutzeroberfläche müssen wir nur prüfen, ob die hinter GUI stehenden Services mit den richtigen Oberflächen verbunden sind. Automatisierte Tests für das User Interface (UI) müssen möglichst naturgetreu die Interaktionen von Benutzern mit dem Client simulieren, d.h. Tastatur- und Mauseingaben nachahmen. Diese Ebene wird mit getestet wenn ein Integrationstest mit STF realisiert ist.

Wie gesagt, gut organisierte Automatisierung spart viel Zeit und sorgt nachweislich für Qualität. Ansonsten bringt es mehr Chaos als Nutzen. Deswegen ist es empfehlenswert, die richtige Balance von GUI-Tests, Integrations-Tests und Unit Tests zu behalten.

Tests während der Implementierung

Während der Implementierung haben wir außer manuellen Testen keine andere Testtätigkeit durchgeführt. Frei nach dem Prinzip, je früher ein Fehler gefunden wird, desto geringer ist der Aufwand, den Fehler zu beheben. Deswegen sollten Entwickler während der Implementierung schon anfangen Tests zu schreiben. Dabei gelten folgende Regeln und Praxis:

Unit Tests

Mit PTF ist es möglich, einfach und schnell Unit Test zu schreiben, welches die ganze Implementierungsphase begleiten sollte. Es ist empfehlungswert, dass neue Funktion mit dem Unit Test vorangehend entwickelt wird. Die Entwicklung der Funktion ist erst abgeschlossen, wenn der entspre-

chende Unit Test korrekt abläuft. Solch ein Test wird auch als Spezifikation der entsprechenden Funktionalität bezeichnet. Damit erspart man sich Dokumentation zu schreiben. Außerdem könnte man den neuen Beteiligten einen Anhaltspunkt geben, über den man sich mit der Struktur des Programmcodes vertraut machen kann.

Bug == Neu Test

Ein gefundener Fehler sollte die Entwicklung eines Test nach sich ziehen, der die Fehlerfreiheit nachweist, nachdem der Fehler gefixt würde. Außerdem kann man überprüfen, ob der gleiche Fehler wieder auftaucht oder nicht.

Änderungen am Code (Refactorings)

Die Änderungen, die am Code durchgeführt werden, sind minimal und systematisch, um die Wahrscheinlichkeit, Fehler einzuschleppen, möglichst gering zu halten. Wobei die Regressionstests eine Hauptrolle im Refactoring spielen. Durch sie wird sichergestellt, dass die gemachten Änderungen am Code keine semantische Änderung des Programms hervorgerufen haben. Die Test-Suite sollte vollautomatisiert ablaufen, um dem Entwickler möglichst viel Arbeit abzunehmen, damit er sich auf das Wesentliche Konzentrieren kann. Die Ergebnisse sollten kompakt ausgegeben werden, etwa wie "Alle Tests bestanden" oder "Test 5 nicht bestanden". Sollten die Regressionstest andere Ergebnisse als "Alle Tests wurden bestanden" ergeben, so werden die Änderungen rückgängig gemacht.[vgl. Hoh02]

Rollen während der Implementierung

Saros ist ein studentisches Softwareprojekt. Jeder Entwickler muss außer der eigenen Tätigkeit nebenbei mehrere Rollen ausführen. Um eine klare Aufgabenverteilung zu schaffen, werden den Entwicklern Rollen mit entsprechenden Testaufgaben zugewiesen (Abb. 5.2).

Der **Bugreporter** ist dafür verantwortlich, den gefunden Bug mit automatisiertem Test zu reproduzieren, wobei man ein Integrationstest mit der Unterstützung von STF schreiben kann , wenn dies mit PTF nicht möglich ist.

Der **Bugfixer** fixt dann den Bug so lang bis der zugewiesene Test erfolgreich durchgeführt wird.

Der **Developer** ist dafür verantwortlich, die entsprechenden Unit Tests für den eigenen Code zu schreiben. Eigentlich wird in der Praxis vermieden, dass Entwickler für eigenen Code Tests testen. Aber aus Mangel an Personalressourcen muss man Kompromisse eingehen. Der erstellte Komponententest ist wie eine dynamische Spezifikation für den entsprechenden Komponenten (Klasse, Modell), die hohe Abdeckung für die Komponente schaffen soll.

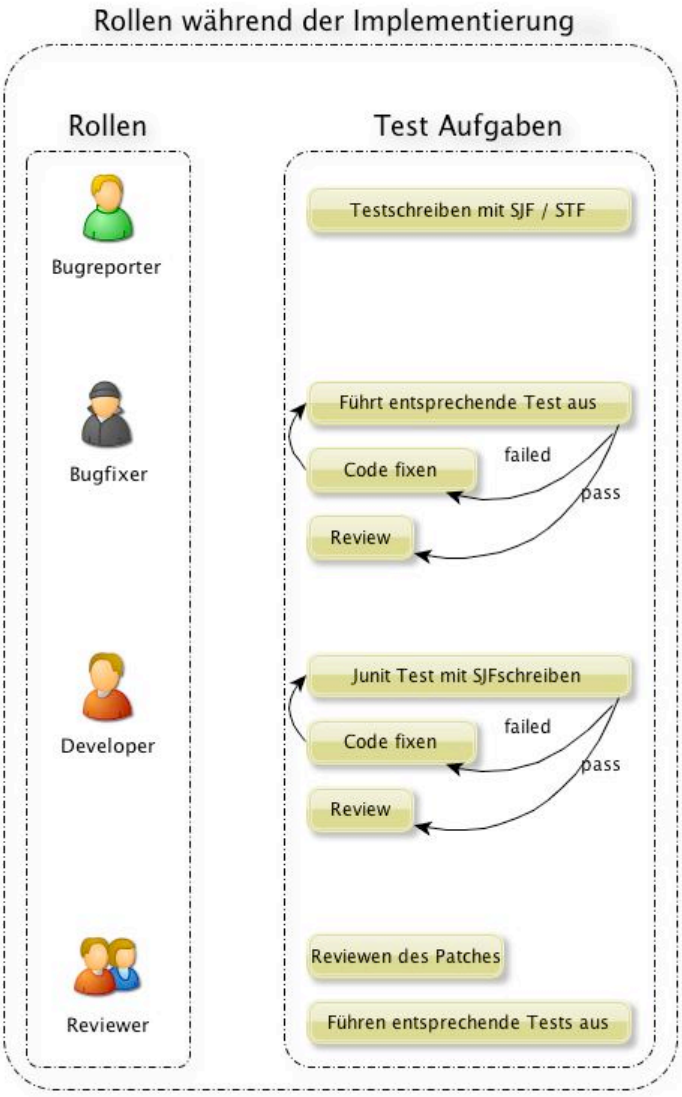


Abb. 5.2: Rolle und Test Aufgaben

Die **Reviewer** sollten im Rahmen der Code Reviews überprüfen, ob die dazu gehörigen JUnit Tests vorhanden und erfolgreich bestanden sind.

Tests während der Release Phase

Integrationstest

Integrationstest sind Black Box System Tests. Jeder Integrationstest repräsentiert ein zu erwartendes Ergebnis von dem System. Integrationstest werden vom Testmanager erstellt, bezogen auf neue Features in diesem Release. Eine Feature kann mehrere Integrationstests haben. Ein Feature ist nicht als vollständig angesehen bis er seine Integrationstests bestanden hat.

Regressionstest

In der Entwicklung sollte nach jeder Änderung ein Regressionstest ausgeführt werden. Bei Regressionstest müssen Sie darauf achten, dass 1) Sie immer die gleichen Tests ausführen, wenn Sie einen bestimmen Code-Abschnitt testen und 2) das betreffende Tests die Akzeptanzkriterien der jeweiligen Anforderung abgleicht. Wenn nun später aber Hudson in den Testprozess integriert würde, könnte man sich diese Arbeit sparen. Mit Hudson würde es dann möglich sein, die Regressionstest regelmäßig auszuführen, etwas nach jeder Codeänderung oder einmal pro Nacht (Nightly Build). Falls es Probleme dabei geben würde, würde das Team per Email informiert.

Alle Tests (Modultests und Integrationstests) müssen vor dem Release erfolgreich sein

Es ist wichtig, diese Regeln einzuhalten, so dass man sicher sein kann, dass ein gefundene Fehler nicht erst in der Release-Version erkannt wird.

Rollen während der Release Phase

Während der Release Woche würde der Vorgang des Testens in verschiedene Teilaktivitäten (Phasen) aufzuteilen und diese Aktivitäten Personen mit verschiedene Rollen zuzuteilen sein. Um das ganze Team kreativ in die Testphase einzubringen und die Lasten vom Testmanager und seiner Assistent zu reduzieren, würden noch zwei Rollen wie Testentwickler und Tester hinzugefügt.

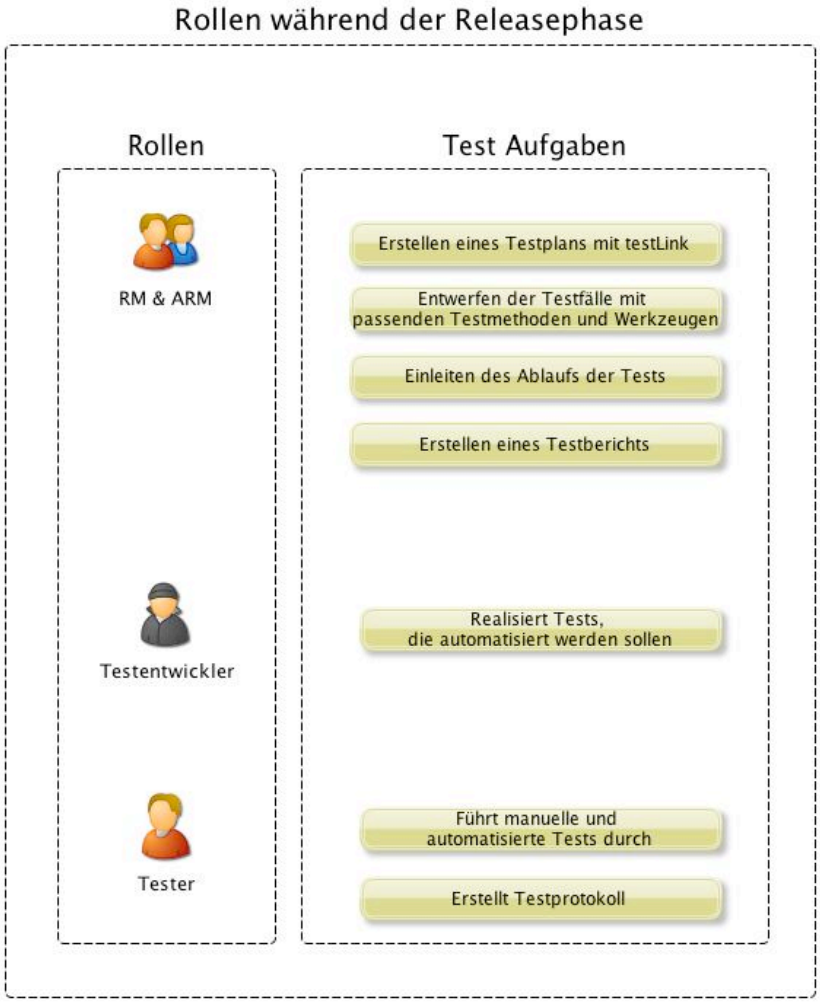


Abb. 5.3: Testen während der Entwicklungsphase

TM und **RTM** sind für den funktionierenden Ablauf des Tests verantwortlich. Sie führen die Testplanung durch und entwickeln Teststrategien. Dazu gehört die Auswahl von geeigneten Testmethoden und Testwerkzeugen sowie Entwurf und Schreiben von Testfällen. Testfälle werden basierend auf den neuen Features in diesen Release nach Priorität sorgfältig erstellt. Wobei die automatisierbaren Testfälle Testentwicklern zugewiesen werden. Wenn all dies fertig ist, werden Testberichte erstellt.

Der **Testentwickler** realisiert die vom Testmanager entworfenen Testfälle mittels zugewiesen Testmethoden und Werkzeugen.

Der **Tester** ist derjenige, der einen Test tatsächlich durchführt. Jedes Teammitglied kann sich als Tester bezeichnen. Als Grundlage für seine Arbeit dienen die vom Testentwickler erzeugten Testfäl-

le samt vorgegebener Infrastruktur und Testfälle, die nicht automatisierbar sind. Seine Testprotokolle sind die Basis für Korrekturmaßnahmen der Entwicklerteams sowie für die Forschrittskontrolle und das Berichtswesen des Testmanagers.

Ideale Vorgehensweise

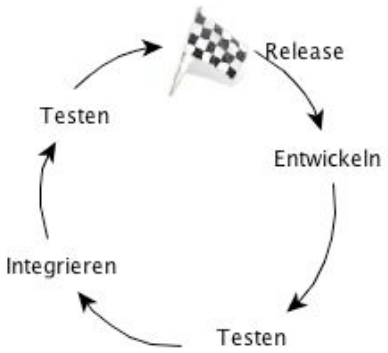


Abb. 5.4: Soll-Entwicklungsprozess

Nach den neu eingefügten Teststufen ist deutlich zu sehen wie sich die Testtätigkeiten im Lebenszyklus des Entwicklungsprozess "verzahnen". Im Vergleich mit vorher sind Tests nicht nur am Ende einer Iteration sondern schon in der ganzen Implementierungsphase bei Saros beteiligt. (Abb. 5.4)

Voll Automatisierung

Viele Tests bringen viele Probleme mit sich. Es ist schon oft passiert, dass Testmanager den Überblick verloren haben, ob ein Testfall schon definiert und dazu ein automatisierter Test generiert wurde. Es führt oft dazu, dass zu viel redundante Tests definiert sind und somit unnötiger Wartungsaufwand entsteht. Ist es möglich verteilte Tests zentral ohne großen Aufwand zu verwalten?

Eigentlich bietet TestLink noch viele Möglichkeiten, die wir noch nicht entdeckt haben. Beispielsweise der von Bruno P. Kinoshita am 10 Dezember 2010 veröffentlichte Artikel[9], der berichtet, dass es mit dem sogenannten TestLink Plugin[10] möglich ist, Hudson mit TestLink zu integrieren, so dass alle Tests zentral in TestLink verwaltet sind. Wobei der Ausführungsstatus des automatisierten Tests durch Hudson aktualisiert wurde. Die genaue Vorgehensweise ist durch die Abbildung 5.5 veranschaulicht. Zuerst bekommt Hudson den Testauftrag vom Testteam. Dann lädt er die automatisierten Tests von SCM (in diesem Fall auch SVN gemeint), die wiederum von Testentwicklern geschrieben sind. Die geladenen Tests werden dann von Hudson durchgeführt und die Ausführungsergebnisse direkt in TestLink übertragen. Diese Vorgehensweise sieht schon ziemlich ansprechend aus. Ob dieser

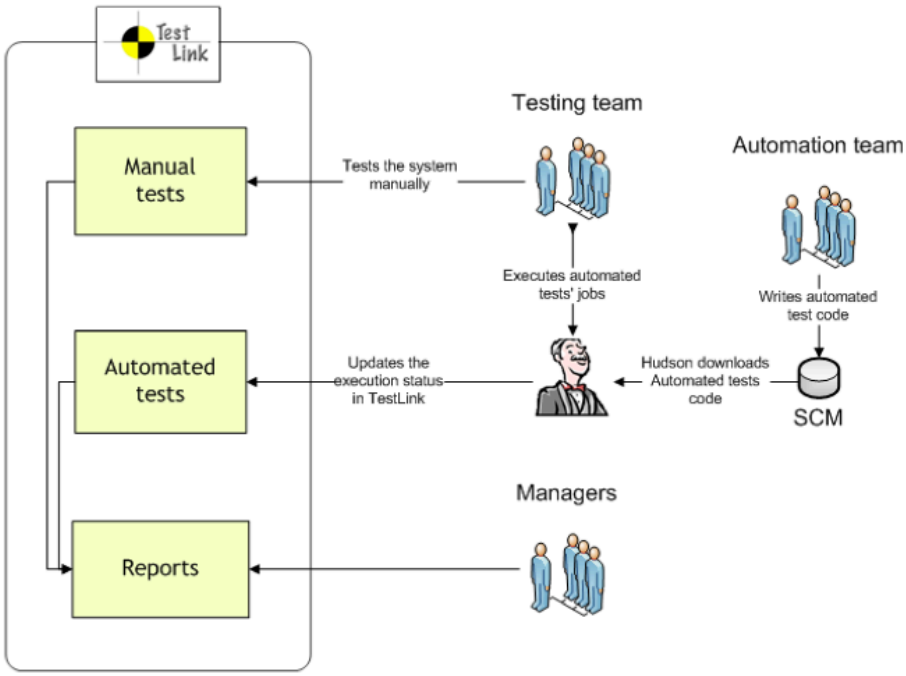


Abb. 5.5: TestLink mit Hudson, Quelle: [Pk11]

[9] [Pk11]

[10] http://wiki.hudson-ci.org/display/HUDSON/TestLink+Plugin, 2010

Lösungsansatz leicht realisierbar und ideal für Saros ist, gehört nicht weiter zur meiner Forschungsgebiet, so dass ich hier nicht mehr weiter ins Details gehe. Aber wäre keine schlecht Idee, Tests zentral und automatisch zu verwalten. Es würde sicherlich viel Wartungsaufwand ersparen.

STF

6 Ist-Analyse von STF

In diesem Kapitel wird zuerst gezeigt, wie STF entstanden ist und wie es funktioniert. Aus dieser Kenntnis heraus beginnt die “Reise”, wobei hier vier Umgestaltungen der Entwurfsmuster von STF vorstellt werden, die zu einem stabileren und leichter zu bedienenden Framework führt.

Hintergrund

Wie in Kapitel Soll-Konzeption erläutert, sollte dieses Framework zum Integration- und GUI-Tests des Testprozesses bevorzugt für die Netzwerkschicht beitragen. Aber wie STF zustande gekommen ist, wird in diesem Unterkapitel erklärt.

Motivation

Die Motivation zur Entwicklung eines Testframework für Saros resultiert daraus, dass ständig unregelmäßige Fehler in der Netzschicht auftraten, die durch manuelles Testen schwer zu identifizieren waren und somit die Möglichkeit verhinderten, die Fehler zu beheben. Deswegen ist es wünschenswert eine "produktionsnahe" Umgebung einzurichten, die die verschiedenen Netzwerkschichten simulieren können. Auf dieser Umgebung Tests durchzuführen bietet weiterhin die Möglichkeit, die bislang unbekannten Defekte in der Netzwerkschicht aufzudecken. Aus dieser Motivation heraus kam Sandor Szücs auf die Idee, eine Netzwerk Testumgebung einzurichten und zusätzlich ein Testframework zu entwickeln, welches dazu beiträgt, dass mit dem Framework geschriebene Tests unter dieser Testumgebung lauffähig sind.

Anforderungen

Eine richtige Testsoftware kann wesentlich zur Effizienz und zur Qualitätssteigerung beitragen. Um die richtige Entscheidung zu treffen und ein maßgeschneidertes Testframework für Saros zu erstellen, ist es erst mal sehr wichtig, die Eigenschaften und Anforderungen unseres Testobjekt Saros zu kennen, die als Kriterien Maßnahme zur späteren Entscheidung beitragen. Hier sind die entscheidende Anforderungen von Saros zusammengefasst.

1. **Verteilung**: Um ein minimales Szenario durchzuführen sind mindestens zwei Saros Instanzen erforderlich, die nicht auf dem selben Rechner laufen. D.h., der Ablauf des Szenario ist über mindestens zwei Saros Instanzen verteilt, die über ein Netzwerk verbunden und miteinander kommunizieren können. Daraus können wir schließen, dass das SOLL-Testframework das verteilte Testen unterstützen sollte.

2. **Synchronisation**: Manche von einem Tester ausgelösten Ereignisse wurden durch einen anderen verifiziert. Um den inhaltlichen Ablauf genau zu kontrollieren, sollte das SOLL-Testframework in der Lage sein, den Testablauf synchronisiert zu streuen.

3. **Eclipse-Plugin**: Saros ist ein Eclipse Plugin Projekt. Somit sollte das SOLL-Testframework die Operationen von Eclipse-Widgets unterstützen.

4. **JUnit**: Saros-Entwickler sind vertraut mit Java. Deswegen sollte berücksichtig werden, dass das Test-Script mit Java ausgestattet ist, e.g. JUnit-Test.

Evaluation

In Bezug auf die vorherigen Anforderungen versuchte Sandor erst mal herauszufinden, ob es die Möglichkeit gibt, passende Software am Markt direkt zu verwenden. Basierend auf verteilte Eigenschaften sind 4 Kandidaten jtrunner[11], Grinder[12], GridGain[13] und STAF[14] "im Rennen". Wobei jtrunner als erstes aus der näheren Auswahl verschwand, weil der Einsatzzweck vom Framework damit nicht unterstützt wurde. Grinder und GridGain bieten zwar verteiltes Testen, unterstützen aber keine Synchronisierung, so dass diese beiden auch schnell ausschieden. Der Kandidat STAF ist somit derjenige, der beide Bedingungen Verteilung und Synchronisation erfüllt. Aber bei dem Wett-

[11] http://jtrunner.sourceforge.net/

[12] http://grinder.sourceforge.net/

[13] http://www.gridgain.com/

[14] http://staf.sourceforge.net/

bewerb tauchte wieder ein anderer Kandidat SWTBot auf, der wiederum völlig anders ist. Er unterstützt zwar keinen verteilten Ablauf und Synchronisation, erfüllt aber die letzten beide Bedingungen (Eclipse-Plugin und J-Unit, siehe Seite 58 oben). STAF hat wegen der Komplexität und unzureichenden Einsatzmöglichkeit in der letzten Runde gegen SWTBot verloren. Der Gewinner ist somit SWTBot.

Folgerung

SWTBot erfüllt die letzten beiden Bedingungen vollständig. Was bei SWTBot fehlt ist eine Unterstützung für Synchronisierung und Verteilung. Schließlich hat Sandor Szücs beschlossen ein Framework selber zu entwickeln, welches die Mängel von SWTBot beseitigt. Das Ergebnis ist STF.

Was ist STF?

Aus der Erkenntnis können wir Folgendes schließen: STF ist ein maßgeschneidertes Test Framework für Saros, welches basierend auf SWTBot entwickelt wird und sich zum Ziel gesetzt hat, die Interaktionen zwischen verteilten Saros Instanzen testen zu können.

Die Vorgehensweise von STF hat eine normale Testszene simuliert, in der mehrere Tester anhand eines vorliegenden Testfalls Saros zusammen manuell testen. Einziger Unterschied ist, dass simulierte Tester auch Saros-Instanzen entfernt bedienen können, die jeweils auf den unterschiedlichen virtuellen Maschinen in der Testumgebung laufen. Der vorliegende Testfall dient dann als zentrale Teststeuerung, in der definiert wird, von wem und wie der Testfall durchgeführt werden soll. Die folgenden Diagramme präsentieren wie Tester Alice, Bob und Carl normalerweise Saros zusammen testen und vergleichsweise mit der Vorgehensweise von STF.

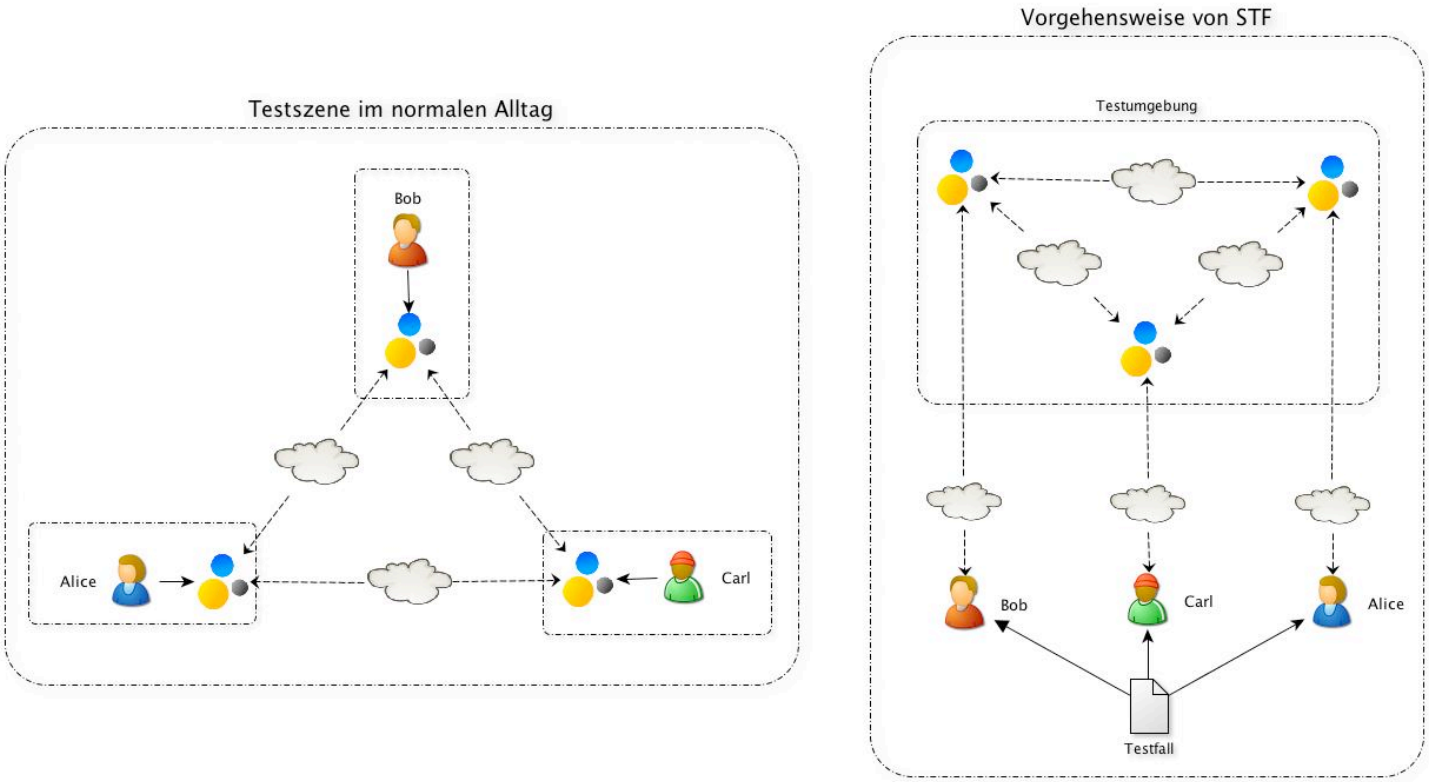


Abb. 6.1: STF simuliert Testalltag von Saros

Sicherlich ist diese Beschreibung immer noch zu abstrakt um zu verstehen, was STF ist und wie es eigentlich funktioniert. Um konkreter zu werden wird nachfolgend auf folgende Fragen näher eingegangen:

- Was ist eigentlich SWTBot und wie funktioniert es?
- Wie wird die Verteilung und Synchronisation in diesem Sinne realisiert?

Einführung von SWTBot

(vgl. [Zel09]; vgl. [Ebert09])

Dieser Abschnitt stellt das Java basierte funktionale UI Testtool gezielt für SWT und die auf Eclipse basierende Anwendung SWTBot vor. SWTBot ist der Kernteil von STF und somit es nötig, dass wir es zuerst kennenlernen. Es wird die grundsätzliche Struktur von SWTBot aufgezeigt, welche als theoretische Grundlage für zukünftige Entwurfsmuster von STF dient.

Was ist SWTBot?

SWTBot ist ein Open-Source Projekt und wurde von Ketan Padegaonkar entwickelt. Es wird als ein "Klick-Roboter" bezeichnet, der die Fähigkeit besitzt, auf das UI der SWT und Eclipse Anwendung zuzugreifen und somit Benutzeraktionen simulieren zu können. Intern werden dabei Events ausgelöst, die effektiv denen eines Benutzers "aus Fleisch und Blut" entsprechen. Weiterhin können momentane UI-Eigenschaften wie `button.isActive()` abgefragt werden, um sie mit Erwartungen zu vergleichen. Mit SWTBot ist es möglich, Tests auf eine natürliche und vertraute Art zu automatisieren. Wie bei JUnit-Test werden SWTBot-Tests in Java geschrieben und als normale JUnit-Tests (SWT) oder als SWTBot-Tests (Eclipse-Anwendung) ausgeführt.

Tests erstellen und ausführen

Bevor näher auf die Theorie eingegangen wird, betrachten wir, wie ein Test mit SWTBot überhaupt aussieht? Eigentlich kann man allein mit SWTBot schon Saros-Szenerios testen, die keine Kommunikation mit anderen Saros-Instanzen benötigen. Beispielsweise läuft der Testfall "Neuen Account einfügen" normalerweise in folgenden 3 Schritten ab.

1. Klicke den Main Menü: Saros -> Start Saros Configuration…

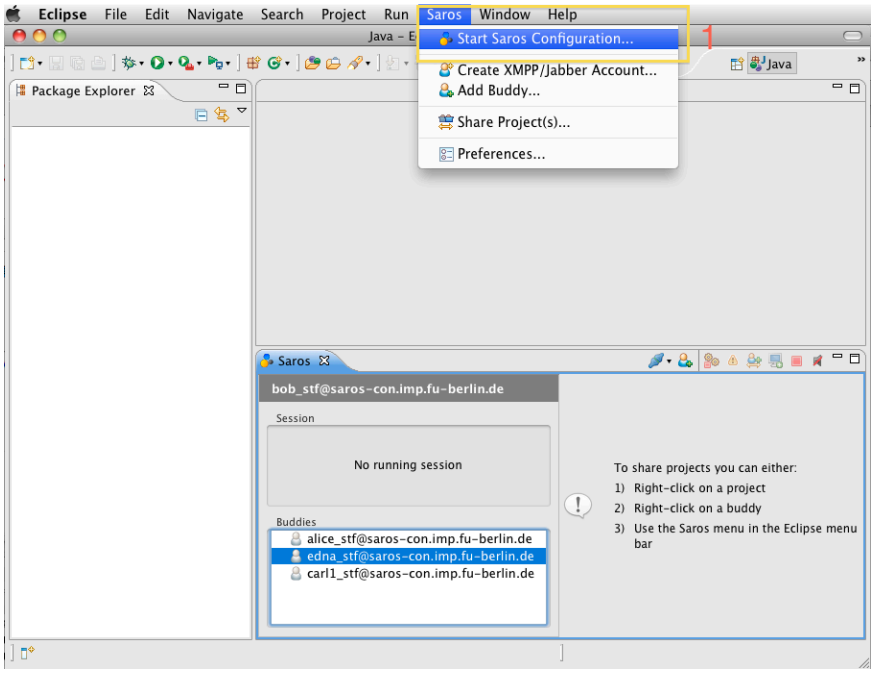


Abb. 6.2: Klicke Menü Saros-> Start Saros Configuration...

2. Trage die XMPP/Jabber ID und das dazugehörige Passwort in die Felder im Popupfenster ein, welche durch den ersten Schritt ausgelöst wird.

3. Klicke den Button "Finish" zum Bestätigen.

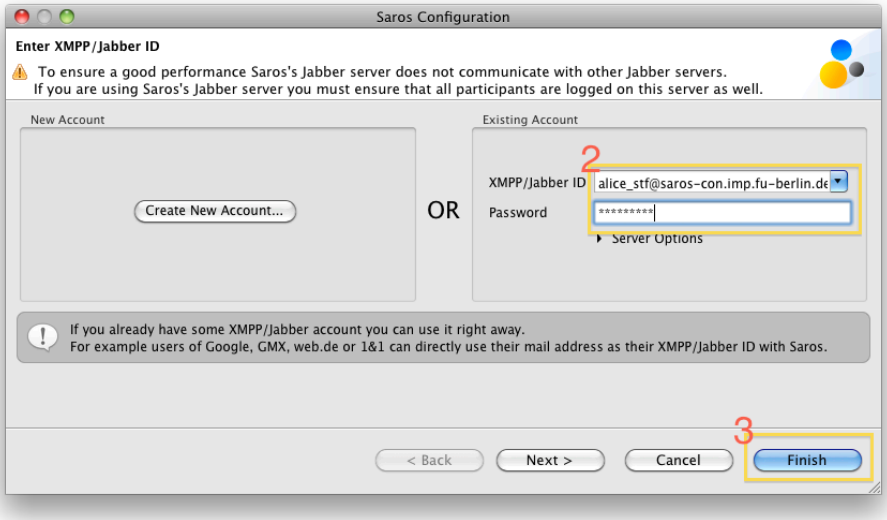


Abb. 6.3: Bestätige Popupfenster "Saros Configuration"

Mit ähnlicher Art und Weise wird die Interaktion vom Benutzer mit der Saros-Instanz durch SWTBot simuliert, und zwar mit dem Prinzip "Finden, Ausführen und Abfragen". Die genaue Implementierung sieht dann folgendermaßen aus:

```
public class AccountTest {
     @Test
     public void addAccount() {
      SWTWorkbenchBot bot = new SWTWorkbenchBot();
      //Klicke das Main Menü: Saros -> Start Saros Configuration…
      bot.menu("Saros").menu("Start Saros Configuration").click();

      //Trage die XMPP/Jabber ID und das Passwort ein
      SWTBotShell shell =  bot.shell("Saros Configuration");
      shell.activate();
      shell.bot().comboBoxWithLabel("XMPP/Jabber ID").setText("alice_test@jabber.ccc.de");
      shell.bot().textWithLabel("Password").setText("alicePassword");

      //Klicke den Button "Finish"
      shell.bot().button("Finish").click();
      }
      ...
}
```

Der Eintrittspunkt in das SWTBot-Framework ist die Klasse `SWTBot`. Diese enthält ein umfangreiche APIs für die Bedienung der SWT-Oberfläche. Für Eclipse-Anwendungen verwendet man die Unterklasse `SWTWorkbenchBot`, die Erweiterungen zur Bedienung der Eclipse Workbench-Oberfläche bereitstellt. Daher wird ein `SWTWorkbenchBot` in der Testklasse `AccountTest` deklariert. Mit Methodenaufrufen auf den `bot` kann man nach Widgets suchen und Aktionen auf diesen auslösen. Mit den Methoden von `SWTWorkbenchBot` können gezielt Elemente der Eclipse Workbench angesteuert werden. So können Views oder Editoren anhand ihrer ID oder dem Reiter-Titel aufgefunden werden. Mit einem so erreichbaren View- oder Editor- Objekt kann man mit der bot()-Methode einen `SWTBot` erhalten, der auf die Inhalte dieses View bzw. Editor-Reiters beschränkt ist. So können alle Methoden von `SWTBot` wiederum verwendet werden, um über die Inhalte des Reiters zu navigieren. SWTBot ist durch das Entwurfsmuster Fluent-Interface[15] implementiert, so dass die entstehenden Tests leicht verständlich und einfach zu pflegen sind. In dem obigen Beispiel ist deut-

[15] http://martinfowler.com/bliki/FluentInterface.html

lich zu sehen, dass die Klick-Pfade durch die Methoden-Verkettung lebensnah reflektiert wurde. Bei der Ausführung werden SWTBot Tests nicht als JUnit Plugin Test sondern als SWTBot Test (einen speziellen SWTBot Ausführungsmechanismus) gestartet. Dieser stellt die Umgebung her, in der die Tests in einem Hintergrund-Thread ausgeführt werden und SWTBot die Anwendung aus diesem heraus fernsteuern kann. Nach der Ausführung der Test-Methode wird die Anwendung automatisch wieder beendet.

Funktionsweise

Es sieht schon intuitiv aus, aber was versteckt sich hinter der Kulisse? Wie ich vorher schon erwähnt habe, heißt die zentrale Klasse so wie das Testframework selbst: `SWTBot`. Für Eclipse-Anwendungen verwendet man die Unterklasse `SWTWorkbenchBot`, die Erweiterungen zur Bedienung der Eclipse Workbench-Oberfläche bereitstellt. Die Grundkonstruktion besteht aus Wrappern, Findern und Matchern. Für jedes unterstützte Widget gibt es in SWTBot eine entsprechende Wrapper-Klasse, z.B. `SWTBotButton` für Widget `Button` und `SWTBotTree` für Widget `Tree`. Darin befinden sich diejenigen Methoden, die das gewünschte Widget anhand von Kriterien wie dem Textattribut finden und fernsteuern. Wegen der Flexibilität bietet SWTBot verschiedene Möglichkeiten, um auf Widgets zuzugreifen. Die Einfachste ist, sie anhand des Textattributs bzw. zugehörigen Labels zu identifizieren, beispielsweise `bot.text("Name")` oder `bot.textWithLabel("Password")`. Oder man kann auch durch IDs auf ein bestimmtes Widget lokalisieren wie `bot.viewById("testkey")`. Die rechte Abbildung zeigt deutlich wie viele Möglichkeiten SWTBot bietet, um einen Button anhand von Kriterien zu lokalisieren.

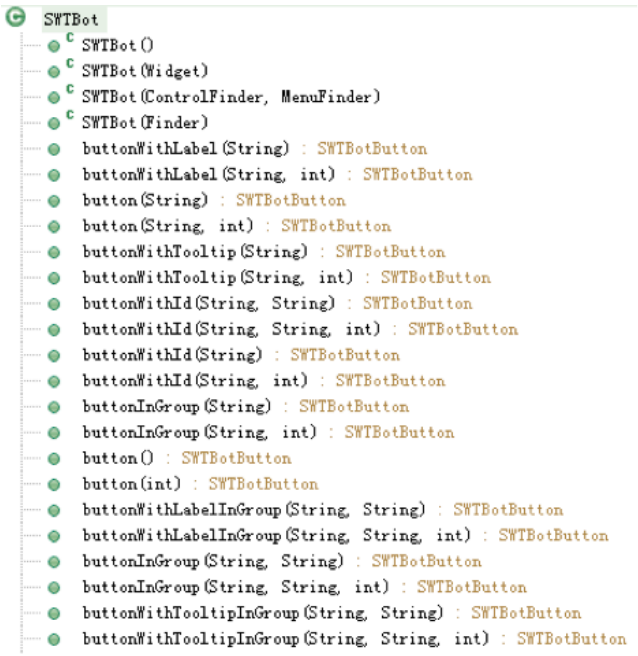


Abb. 6.4: Mögliche Methoden für die Lokalisierung eines Button

Das unten abgebildete Diagramm veranschaulicht die Beziehungen zwischen Widgets, SWTBot-Widgets und SWTBot.

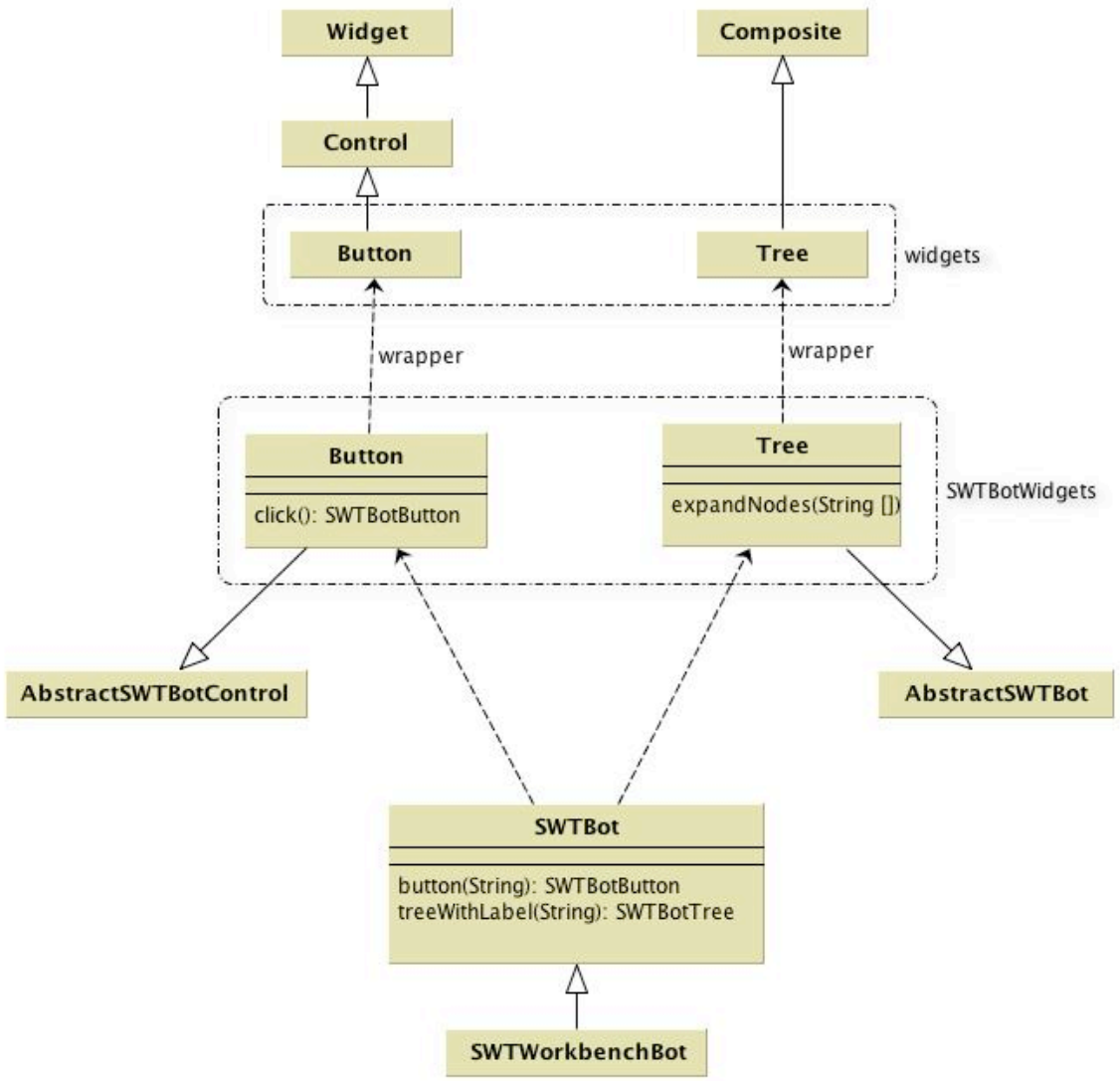


Abb. 6.5: Ausschnitt des Klassendiagramms von SWTBot

Fazit

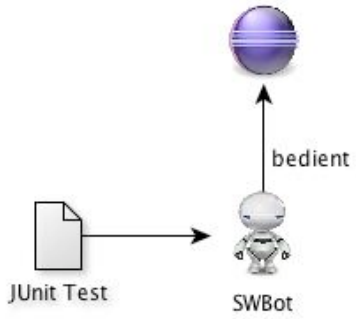


Abb. 6.6: Vorgehensweise mit SWTBot

Die Verhaltensweise von SWTBot entspricht einem Klick-Roboter. Er übernimmt einfach die monotone Klick-Aufgabe von uns. Es reicht aus ihm einen Befehl zu geben. Das Bild auf der linken Seite visualisiert die Vorgehensweise, wobei bezogen auf SWTBot's Eigenschaft durch einen Roboter präsentiert wird.

SWTBot ist ein noch sehr junges Projekt und enthält noch einige Stolpersteine und Restarbeiten. Aber es wird sicherlich weiterhin verbessert. Seit Ende 2008 werden der Roadmap zufolge sämtliche SWT Widgets unterstützt. Die weiter geplanten Features unterstützen sogar GEF (Graphical Editing Framework). Damit ist es auch möglich, dass neue Feature "Whiteboard" von Saros zu testen, das im Jahre 2011 vom Absolventen Michael Jurke durch seine Diplomarbeit [Jur11] eingeführt wurde.

Verteilung und Synchronisation

Um Saros testen zu können, müssen mehrere Saros-Instanzen ferngesteuert und gleichzeitig die einzelnen Testschritte inhaltlich aufeinander abgestimmt sein. Dafür stößt SWTBot eher an seine Grenze. Wie wäre es wenn alle Saros-Instanzen jeweils entsprechende SWTBots zugewiesen würden, die dann zentral in einem JUnit-Test ferngesteuert sind? Dieses wird mit der Technologie RMI (Remote Methode Invocation) verwirklicht.

Wie funktioniert RMI?

(vgl. [wiki10]; vgl. [Bag05])

Um die nachfolgende Erzählung nachvollziehen zu können, wird hier kurz eine Einführung von RMI erläutert.

RMI stellt einen Mechanismus zur sog. verteilten Anwendungsprogrammierung bereit, d.h. es ermöglicht dem Client auf entfernte Objekte zuzugreifen und deren Dienste zu nutzen. Als entfernte Objekte (Remote Objects) werden Objekte bezeichnet, die sich auf einer anderen JVM befinden. Diese kann (muss aber nicht) auf einem anderen Rechner laufen und ermöglicht damit die Kommunikation über Netzwerke zwischen verschiedenen Rechnern. Der konkrete Ablauf besteht aus folgenden Schritten:

- Der Server registriert ein Remote Object bei der RMI-Registry unter einem eindeutigen Namen.
- Der Client sieht bei der RMI-Registry unter diesem Namen nach und bekommt eine Objektreferenz, die seinem Remote Interface entsprechen muss.

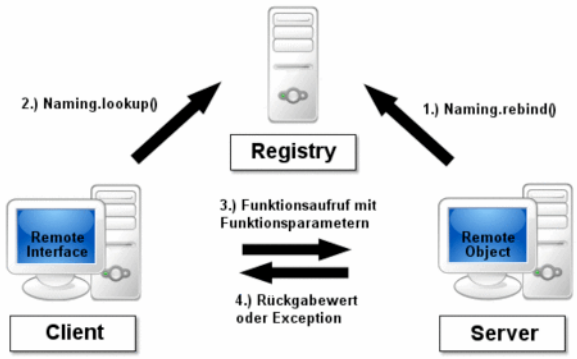


Abb. 6.7: Die vier Schritte der RMI. Quelle: [wikiRmi]

- Der Client ruft eine Methode aus der Objektreferenz auf (dass diese Methode existiert, wird durch das Remote Interface garantiert). Dabei kann ein Objekt einer Klasse X übergeben werden, die der JVM des Servers bisher nicht bekannt ist (das ist möglich, wenn X ein dem Server bekanntes Interface implementiert). In diesem Fall lädt die Server-JVM die Klasse X dynamisch nach, beispielsweise vom Webserver des Client.

- Die Server-JVM führt die Methode auf dem Remote Object aus, wobei evtl. dynamisch geladener Fremdcode benutzt wird (z. B. Methoden von X), der im Allgemeinen Sicherheitsrestriktionen unterliegt. Dem Client werden die Rückgabewerte dieses Aufrufes gesendet, oder der Client bekommt eine Fehlermeldung (z. B. bei einem Verbindungsabbruch).

Wie funktioniert RMI + SWTBot?

Mit dem gleichem Prinzip kann es so übertragen werden, dass die Saros-Instanz als das Server Programm bezeichnet wird. Und die vom Server Programm angebotenen Services sind die SWTBot APIs. D.h. jeder Saros-Instanz wird ein SWTBot zugewiesen, welches dafür zuständig ist, die Saros-Instanz zu steuern. Auf der Client Seite wird eine Klasse Tester definiert, welcher die RMI-Registry mit den Schlüsselpaar (Port Nummer und IP-Address) identifizieren und somit die Objektreferenz erhalten kann. Mit Objekt-Referenz kann Tester SWTBot-APIs entfernt aufrufen. Damit ist es möglich, die entsprechende Klick-Aktion auf der Saros-Anwendung durchzuführen. In JUnit-Test werden dann soviel Testers erzeugt, wie die Anzahl der benötigen Saros-Instanzen für den Testszenario. Mit dieser Idee kann das erste Prototypen-Modell dann so aussehen. (siehe Bild)

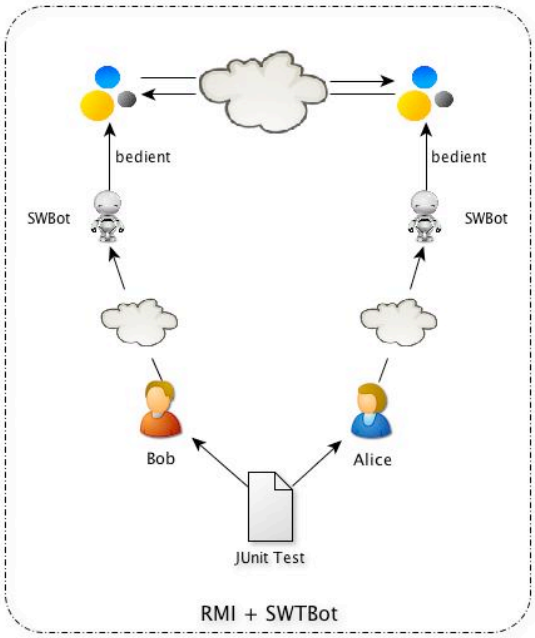


Abb. 6.8: Das erste Prototypen-Modell von STF

Wie exportiert SWTBot APIs?

Wie werden SWTBot-APIs als Remote Services exportiert? Ein erster Lösungsansatz dafür könnte sein, eine Methode in einer Remote Klasse zu definieren, die `SWTWorkbenchBot` Object als Rückgabewerte zurückliefet. Damit kann Tester SWTBot'APIs bei Aufrufe der Methode verwenden. Aber das Problem dabei ist, dass `SWTWorkbenchBot` nach dem Entwurfsmuster Fluent Interface implementiert ist und deshalb bei der Benutzung meistens SWTBot[Widget]-Objekte zurückliefert. SWTBot[Widget]-Objekte können nicht per RMI exportiert werden, weil sie plattformabhängig sind und daher nicht serialisiert werden können. Somit ist der Ansatz selbstverständlich gescheitert.

Der andere Lösungsansatz ist, dass man eine Delegation-Klasse `RmiSWTWorkbenchBot` für `SWTWorkbenchBot` erstellt. In dieser Delegation-Klasse werden dann Funktionen definiert, die als Vertreter für den Aufruf der Methoden-Verkettungen mit SWTBot dienen. Beispielsweise definiert man eine Methode `clickButton(String buttonName)` in der Klasse, die dann `bot.button("Finish").click()` aufruft. Für die Außenwelt scheint es, dass die RmiSWTWorkbenchBot-Klasse diejenige ist, die die Aktionen durchgeführt hat. Aber in Wirklichkeit ist `SWTWorkbenchBot` ist der Täter. Solche Vertreter-Methode, welche nur eine Aktion oder Zustand-Anfrage bezogen auf ein Widget enthält, wird auch als Low-Level Methode bezeichnet. Der folgende Code zeigt einen Ausschnitt von Low-Level Methoden in der Klasse `RmiSWTWorkbenchBot`, welche exportfähig sind.

```
SWTWorkbenchBot bot = new SWTWorkbenchBot();
 //Klicke Button
public void clickButton(String mnemonicText) throws RemoteException {
   bot.button(mnemonicText).click();
}
//Überprüft den Zustand des Buttons
public boolean isButtonEnabled(int num) throws RemoteException {
   return bot.button(num).isEnabled();
}
```

In dieser Delegation-Klasse wurden meistens High-Level Methoden beinhaltet, die jeweils aus mehrere Low-Level Methoden besteht und ein bestimmte Service von Eclipse präsentiert. Z.B die Methode `newClass` steht dafür, dass nach dem Aufruf dieser Methode eine Klasse erzeugt wird, und Methode `addANewBuddy` heißt aber nicht dass eine neue Kontakt eingefügt wird. Denn um eine Kontakt vollständig einzufügen muss noch die Bestätigung von anderes Seite vorliegen sein. In diesem Fall diese Methode bedeutet eher eine Kontakt-Anfrage zu senden. Solche Methode wird meistens durch Services' Name benannt. Hier ist noch die genaue Implementierung von Methode `addANewBuddy`.

```
public void addANewBuddy(String jabberID) throws RemoteException {
   bot.viewByTitle("Saros").toolbarButton("Add a New Buddy").click();
   SWTBotShell shell =  bot.shell("Add Buddy");
   shell.activate();
   shell.bot().textWithLabel("XMPP/Jabber ID").setText(jabberID);
   shell.bot().button("Finish")click();
}
```

Die Remote-Objekts bei der ersten STF-Entwurfsmuster sind grundsätzlich auf diesem Delegation-Muster entwickelt. Weiterhin wurde eine Klasse `SarosRmiWorkbenchBot` von `RmiSWTWorkbenchBot` abgeleitet, womit spezifisch auf Saros-Features bezogene Aktionen zentral in diesem Subklasse verwaltet werden können. Damit die erweiterte Modell sieht dann so aus:

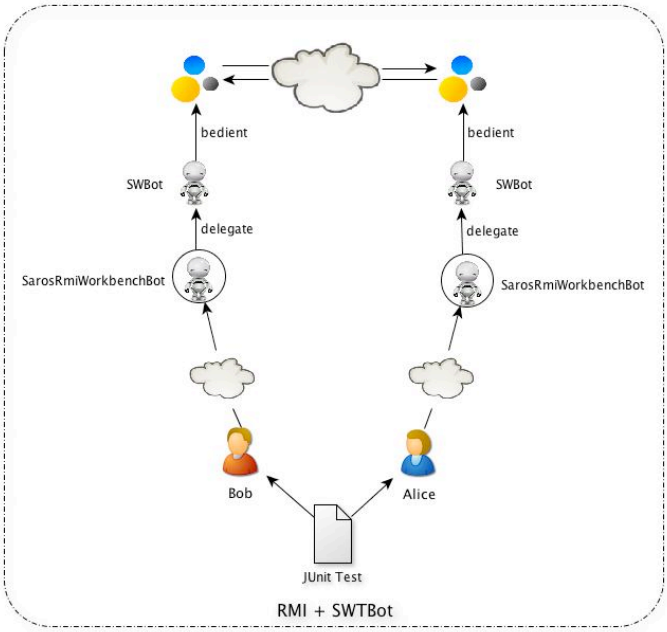


Abb. 6.9: Die erweiterte Modell von STF

Bau des Testframeworks

Bei dem Aufruf einer Saros-Instanz wurden das RMI-Objekt `SarosRmiWorkbenchBot` bei der RMI-Registry registriert und somit die definierte APIs (später STF-APIs genannt) für Tester zugänglich sind. Das Klassendiagramm (Abb. 6.7) verdeutlicht das Verhältnis der Klassen innerhalb des Testframeworks.

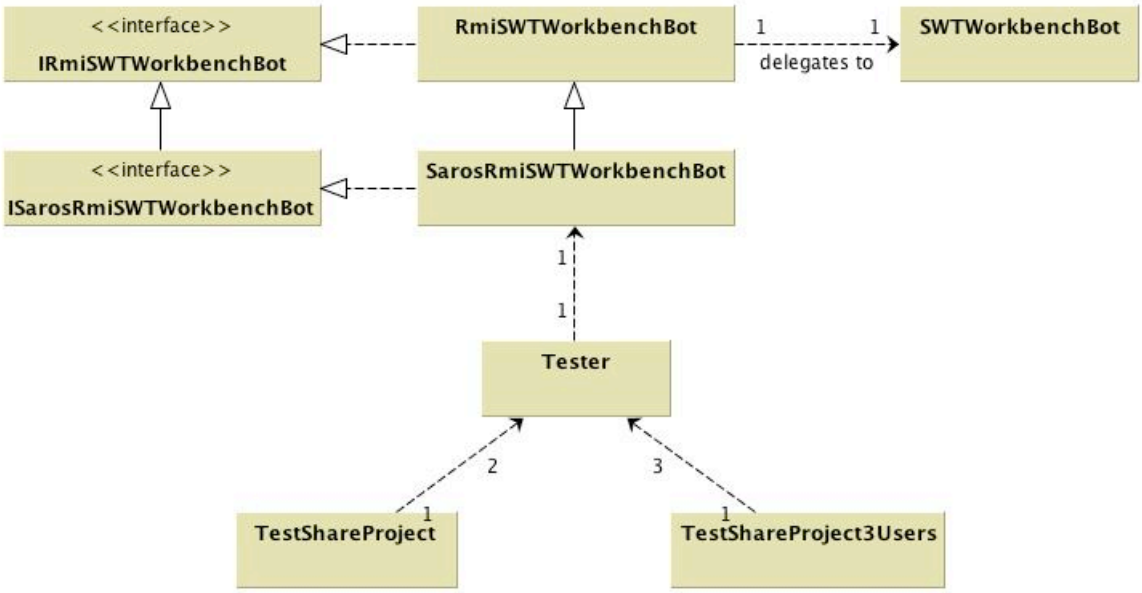


Abb. 6.10: Der erste Entwurf von STF

Wie werden RMI-Objekte bei Start eines Saros exportiert?

Der Start eines Saros-Instanz wird über die Klasse `StartupSaros` ermöglicht. Sie ist eine Implementierung von Eclipse `IStartup`, die nach Initialisierung der Workbench nach dem Laden eines Plugins automatisch die Methode `earlyStartup()` aufruft. In `StartupSaros#earlyStartup()` wird auf den entscheidenden Startparameter `-dDe.fu_berlin.inf.dpp.testmode` geprüft, um Saros normal oder als Test-Model zu starten. Das Start als Test-Model umfasst den Start einer RMI-Registry und den Export von zwei verschiedenen RMI-Objekten. Der unterliegende Beispiel zeigt eine mögliche Run Konfiguration um Saros als Test-Model zu starten. Wobei mit dem Port Nummer und dem IP-Address von der Host Machine der Server Programm kann Client auf RMI-Registry zugreifen. In diesem Beispiel wird die IP-Address auf `localhost` gesetzt. D.h. Server Programm und Client laufen in gleichen Maschine. Diese Art von Konfiguration ist für lokale Test gedacht.

```
-Xmx512m
-ea
-XX:MaxPermSize=128m
//Der Pfad zum bin Ordner vom Projekt Saros
-Djava.rmi.server.codebase=file:${workspace_loc}/Saros/bin
-Djava.security.manager
//Der Pfad zur Datei “all.policy”, welcher für das Exportieren benötigt wird
RMI-Objects-Djava.security.policy=file:${workspace_loc}/Saros/test/STF/all.policy
//Die Portnummer zuweisen, welche zum Erstellen der RMI Registry benutzt wird
-Dde.fu_berlin.inf.dpp.testmode=12345
-Dde.fu_berlin.inf.dpp.sleepTime=200
//Die IP-Adressen der Host Maschine der Server Anwendung, welche vom Client zum Bekommen der
//RMI-Registry benutzt wird
-Djava.rmi.server.hostname=localhost
```

STF-Test erstellen und ausführen

Wie sieht dann ein Test aus? Mit STF können wir jetzt schon verteilte Saros-Szenerie testen, wie z.B. das Szenario "Alice fügt Bob in der Kontaktliste ein". Sie passiert normalerweise in folgenden 3 Schritten:

1. Alice klickt Toolbarbutton mit Tooltip "Add a New Buddy";

2. Alice trägt die Jabber ID von Bob im Popup-Fenster mit Title "Add Buddy"ein, welches vom ersten Schritt ausgelöst wird, und bestätigen;

3. Bob wartet auf die Popupfenster "Request of Subskription received" und bestätigen den Anfrage von Alice.

Die genaue Implementierung mit STF sieht dann folgendermaßen aus:

```
public class sarosViewTest {
    private Tester alice;
    private Tester bob;
   @Before
   public void configureAliceAndBob() throws RemoteException{
        alice = new initAlice();
        alice.openSarosView();
        alice.connect();
        bob = new initBob();
        bob.openSarosView();
        bob.connect();
   }
   @Test
   public void addANewBuddy() {
        alice.rmiBot.addANewBuddy("bob_test@jabber.ccc.de");
        bob.rmiBot.confirmShellRequestOfSubskriptionReceived();
   }
}
```

Nach der RMI-Erweiterung ist STF-Test nicht wesentlich anders als SWTBot-Test. Die größte Unterschiede dabei sind Initialisierung und Ausführung. Statt den `SWTWorkbenchBot` wird `Tester` in der Testklasse deklariert. In der mit `@BeforeClass` annotierten JUnit-Setup-Methoden werden soviel Tester Initialisiert wie die Anzahl der benötigen Saros-Instanzen für den Testszenario. Mit der Initialisierung wird sichergestellt, dass Tester den Zugang jeweils zur dem von Saros-Instanz angebotene STF-APIs durch den Objekt-Referenz `rmiBot` hat.

Um verteilte Testlauf auch synchronisiert zu halten wird die komplette Teststeuerung in einem separaten Prozess durchgeführt. Bei Ausführung Tests werden nicht als SWTBot Tests gestartet, da sie einen unabhängigen Ablauf zur Konsequenz hätte. In unserem Fall besteht die Ausführung aus zwei Schritten. Es müssen Saros-Instanzen als Test-Model konfiguriert und manuell gestartet werden. Danach startet man STF-Test als JUnit Test.

Zusammenfassung

Bis hierher wurde das erste STF-Entwurfsmuster vorgestellt. Aber die „Abenteuerreise“ von STF ist noch nicht am Ende, Sie wurde von mir weitergeführt. Durch den am Anfang bitteren Rückschlag und die daraus gesammelten Erfahrungen in der Praxis hat STF insgesamt vier Veränderungen der Entwurfsmusters erlebt. Somit ist es von Release zu Release immer stärker, stabiler und leichter bedienbar geworden. Wohin die Reise von STF geht wird im späteren Kapitel vorgestellt.

7 STF Reise

Page-Object als Designpattern

Am Anfang meiner Arbeit ging es sehr hektisch und planlos zu. Der Ehrgeiz, dass STF möglichst schnell zum Testprozess beitragen könnte, hatte viel Chaos verursacht. Das Gewicht meiner Arbeit fokussiert sich eher auf dem Testschreiben als auf der Verbesserung und Erweiterung des Frameworks. Nach der immer weiter steigenden Komplexität und dem somit immer höheren Wartungsaufwand wurde mir bewusst, dass eine strategische Architektur-Umgestaltung bei STF durchgeführt werden muss. Wie das sogenannte Chaos entstanden und was die Gegenmittel bzw. Lösungsansätze dafür sind, werden in diesem Unterkapitel vorgestellt.

Planlose Erweiterung

Sandor Szücs hat eigentlich das Grundgerüst von STF bereitgestellt. Wegen der fehlenden APIs und Implementierungen ist STF noch nicht wirklich einsetzbar. Es sind zwar schon ein paar High-Level Methoden vorhanden, die aber nur aus Testgründen definiert wurden. Auf dieser Basis habe ich bezogen auf Saros-Szenarien viele High-Level Methoden wie deleteClass, renameFile, connectWith, renameBuddy, usw. erstellt, die dann einfach in die vorhandenen Remote-Objekte eingepackt wurden. Normalerweise sollte nach den Faustregeln (vgl. [Lor93]) eine Klasse nicht mehr als 15 Methoden umfassen, sodass es leicht fällt, potentielle Problembereiche zu identifizieren. Aber zwischendurch ist die Zahl der Klasse `SarosRmiSWTWorkbenchBot` enthaltenen Methoden sogar auf 100 angewachsen. Da ist es unvermeidlich, dass man redundante Methoden versehentlich definiert hat. Außerdem ist nach der GUI-Änderung eine schnelle Anpassung schwer zu erreichen.

Trotz dieser Ausgangslage wurde das STF eingesetzt, um die in Testlink definierten Testfälle zu automatisieren. Inzwischen gibt es fast mehr als 24 Testfälle, die mit STF übersetzt wurden. Denn am Anfang bestand mein Hauptziel eher darin, das Test-Management-Tool Testlink durch automatisierte STF-Tests abzulösen. Im Nachhinein merkte man dann, dass es eher eine vorschnelle Entscheidung war. Wie ich schon im Kapitel SOLL-Konzeption erwähnt habe, spielt Testlink eine sehr

zentrale Rolle in dem gesamten Testprozess. Damit können nicht nur organisatorische Testpläne erstellt, sondern auch alle manuellen bzw. automatisierten Tests dynamisch mit Hudson zentral verwaltet werden. Außerdem ist es auch nicht wirklich nötig, alle Testfälle mit STF zu übersetzen, wenn sie einfach mit JUnit-Tests automatisiert werden können.

Die Vorgehensweise beim Testschreiben ist ziemlich anstrengend und begrenzt. Denn mit der High-Level Methode kann man zwar schnell und bequem Tests schreiben, sie bietet aber zu wenig Flexibilität. Wenn ein Testschreiber z.B. beim Testfall „Einfügen eines neuen Buddy“ auch testen möchte, ob eine Warnmeldung angezeigt wird, wenn der hinzufügte Buddy schon in der Kontaktliste existiert, ist die Methode `addANewBuddy` eher nutzlos. Dafür muss man wiederum eine Methode wie `addAExistedBuddy` definieren. Somit entsteht viel redundanter Code, der die Wartungsarbeit erschwert.

Sicher ist, dass durch die begrenzten High-Level Methoden bestimmte Benutzerwünsche nicht immer erfüllt werden. Die APIs, die von SWTBot angeboten werden, wurden wegen der fehlenden Implementierung nicht wirklich auf der Client-Seite sicht- und damit nutzbar gemacht. D.h., dass der Testschreiber nur das sieht, was STF bietet. Wenn er keine passende Methode findet, muss er erst mal auf der Server-Seite nachschauen, was SWTBot anbietet, um dann die gefundenen APIs von SWTBot umschreiben und die ganze Remote-Umgebung wieder anpassen zu können. Wenn dann alles richtig gemacht wurde, darf er im JUnit-Test testen, ob die von ihm definierte Funktion das Angestrebte erfüllt. Wenn er Glück hat, wird es beim ersten Mal klappen, ansonsten muss er alles wieder von Neuem beginnen. Deswegen hat es mich am Anfang nicht überrascht, dass Testschreiber sich andauernd beschwerten, dass damit Tests zu schreiben zu langwidrig und nicht gerade trivial ist.

Noch schlimmer ist, dass jeder Testschreiber STF APIs nach Belieben so erweitert, dass es am Ende viele Funktionen gibt, die zwar unterschiedlich benannt sind, aber das gleiche tun. So entsteht viel “doppelte” Arbeit und somit ein hoher Wartungsaufwand.

Nach dem Rückschlag habe ich aufgehört, weitere Tests zu automatisieren. Denn dadurch würde mehr Unordnung geschaffen werden, als das es helfen würde. Mein Focus sollte mehr darauf liegen, dass ich Entwicklern die Möglichkeit biete, schnell und effektiv Tests mit STF zu schreiben. Um sich schneller dem Ziel zu nähern, ist anfangs eine Architektur-Umgestaltung für STF notwendig; STF braucht mehr Struktur, Ordnung und genaue Richtlinien.

Lösungsansatz

Wie wäre es denn, wenn man mit den sogenannten Designmuster "Page-Object"[16] unsere STF-APIs verwaltet? Das Muster stammt von Google und ist vor allem bei Webseite-Tests mit Selenium[17] bekannt. Zum Zugriff auf die Oberfläche werden spezielle Klassen erstellt, sogenannte Page-Objects, und diese dann in den Tests verwendet. Die Page-Objects enthalten eine fachliche API für Benutzeraktionen pro "Seite". So wird die Anzahl der redundanten Codezeilen verringert. Falls sich im UI etwas ändert, muss nur im jeweiligen Page-Object eine Anpassung stattfinden. Analog dazu können Eclipse-Komponenten wie Views, Editoren, Perspektiven, Wizards, Menubar usw. im übertragenen Sinn als "Seiten" verstanden und dafür Page-Objects erstellt werden (vgl. [Zel09]). Genauer gesagt heißt das, dass es innerhalb der Eclipse-GUI-Oberfläche Komponenten gibt, mit der die Tests interagieren. Diese wird dann als Page-Object modelliert. Wobei die in einem Page-Object definierten Methoden (sogenannte High-Level Methoden) als "Services" interpretiert werden können, die die entsprechende Eclipse-Komponente bietet. Zum Beispiel wird Saros-View als Page-Object modelliert. Die davon angebotenen Services wie Herstellen einer Verbindung, Hinzufügen eines neuen Buddy, usw. können jeweils als Funktion interpretiert werden.

Basierend auf dieser Idee wurden die vorhandenen Remote-Objekte `RmiSWTWorkbenchBot`, `SarosRmiSWTWorkbenchBot` auf mehrere Page-Objects zerkleinert, welche mit Remote-Interface erweitert und exportiert wurden. Somit kann ein Client auf die Services der Page-Objects zugreifen.

16 http://code.google.com/p/selenium/wiki/PageObjects

17 http://seleniumhq.org/

In der folgenden Abbildung ist deutlich zu sehen, wie viele Page-Objects (mit Java-Icon gekennzeichnete Knoten) erstellt und welche Eclipse-Komponenten damit modelliert wurden. Wobei Page Objects im selben Paket eingeordnet werden, welche Komponenten mit gleichem Typ modellieren.

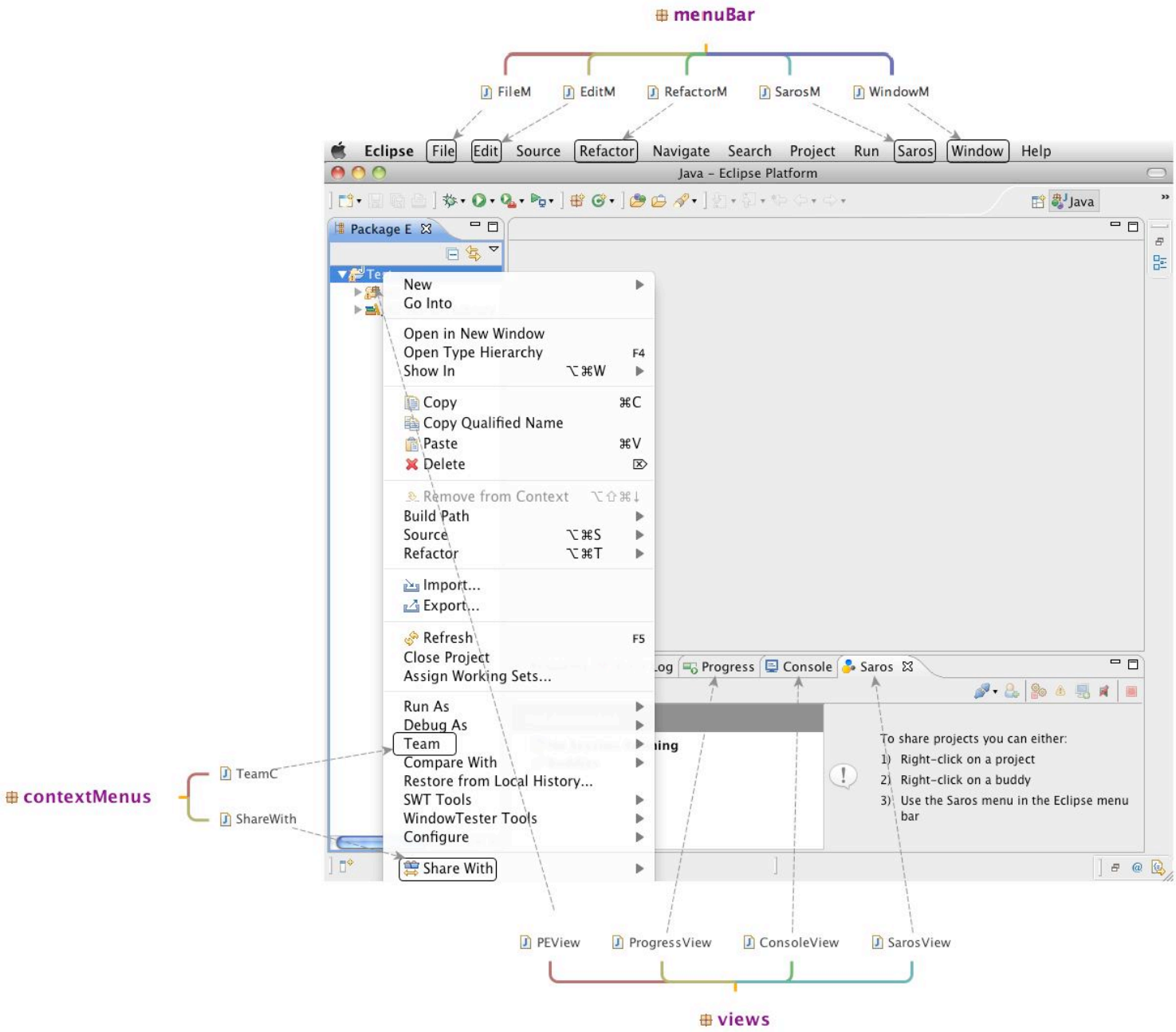


Abb. 7.1: Modellierte Eclipse-Komponenten

Wie ich vorher erwähnte, ist es auch wünschenswert, dass ein Client SWTBot-APIs direkt verwenden kann, damit er mehr Flexibilität beim Testschreiben hat. Dafür müssen mehrere Low-Level Methoden definiert werden. Aber wie verwaltet man denn solche Methoden? In diesem Fall werden SWT-Widgets satt Komponenten als Page Objects modelliert. Die von einem Widget, z.B Button, angebotenen "Services" wie `click()`, `isEnabled()` usw. werden dann als Low-Level Methoden interpretiert. Auf Saros-Szenarien bezogen werden nur die oft verwendeten Widgets modelliert. Die rechtsliegende Abbildung zeigt die bisher erstellten Page-Objects für Widgets und die Low-Level Methoden in Page-Object Button.

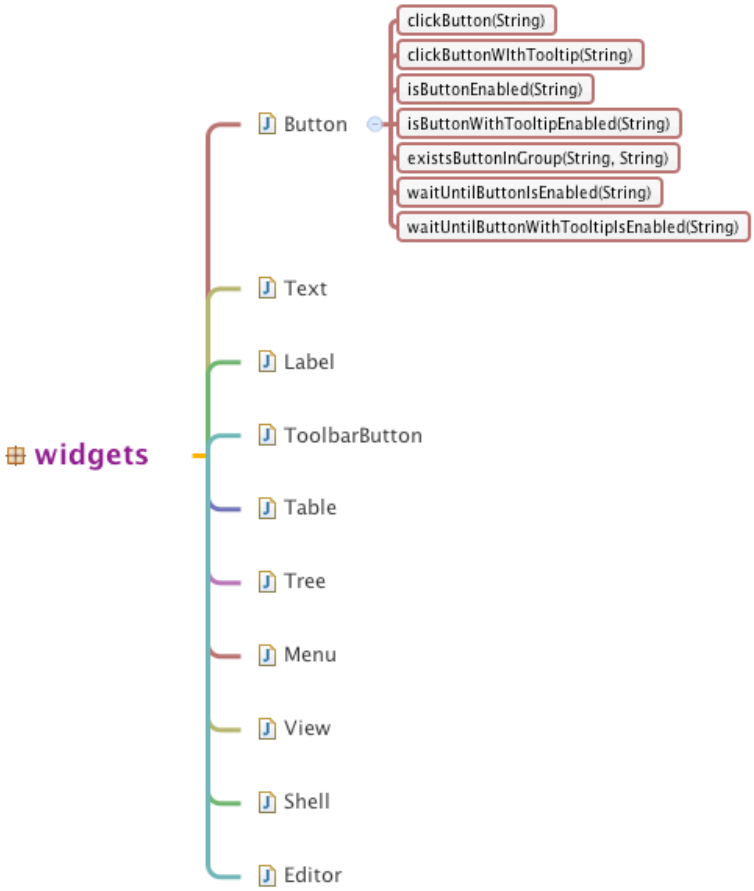


Abb. 7.2: Modellierte Widgets

Struktur

Durch diese Umgestaltung sind 21 Page-Objekts zustande gekommen, die jeweils Eclipse-Komponenten oder Widgets modellieren. Diese werden zentral durch die Klasse `STFController` bei der RMI-Registry registriert. Für Page-Objects, die in der gleichen Group (Package) liegen, wird jeweils eine Oberklasse definiert, die dazu dient, dass die von allen Unterklassen gemeinsam verwendeten Services nur einmal definiert werden müssen. Die Klasse `Menubar`, `Views`, `ContextMenus` und `Widgets` erben wiederum von der Klasse `Components`, die als die Delegation-Klasse für

`SWTWorkbenchBot` dient. Wegen der Zerlegung besitzt ein Client nicht nur einen Eintrittspunkt, sondern mehrere, die jeweils auf das entsprechende Page-Objekt referenzieren. Somit muss man sich beim Testscheiben Gedanken darüber machen, wo sich ein Service befindet. Das Klassendiagramm (Abb. 7.3) verdeutlicht die nach dem Lösungsansatz (Page-Object) entstandenen Klassen des Testframeworks und ihren Zusammenhang mit Test `TestSchareProject`.

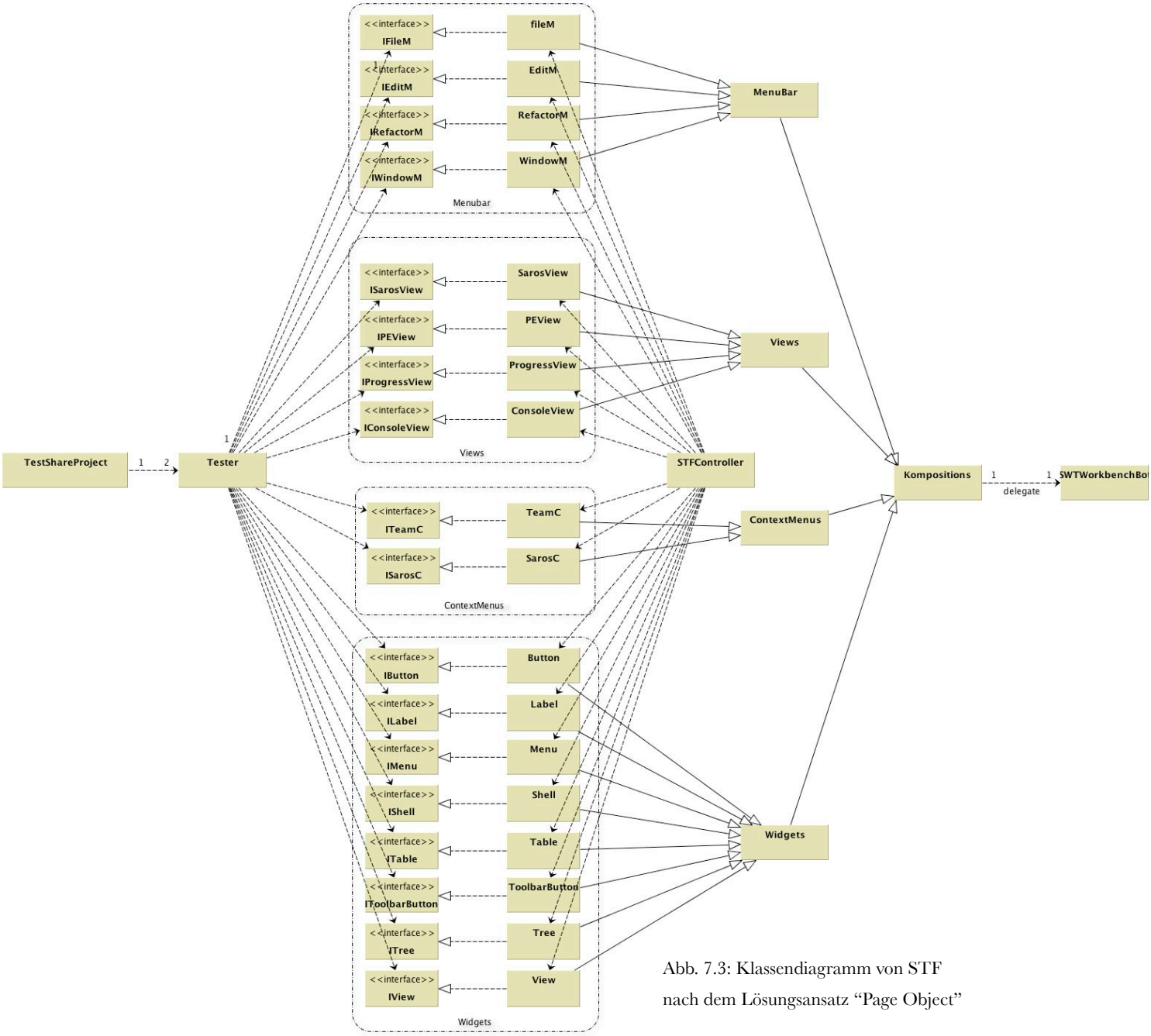


Abb. 7.3: Klassendiagramm von STF nach dem Lösungsansatz "Page Object"

Fazit

Mit dieser Version bin ich für eine Weile sehr zufrieden gewesen. Durch die klaren Struktur und die Aufteilung bietet sie uns viele Vorteile:

1. Man bekommt schnell einen Überblick, welche Page Objects angepasst und welche erweitert werden müssen.

2. Es fördert Wiederverwendung und reduziert somit die duplizierten Codezeilen.

3. Es macht Tests lesbar.

4. Es macht Tests robuster und weniger spröde.

5. Die Wartbarkeit wird wesentlich verbessert. Dies ist besonders wichtig für solche Applikationen wie Saros, die sich rasch weiterentwickeln und viele Änderungen auf der GUI-Oberfläche haben.

Man kann variabel sein, ob Tests mit High-Level oder Low-Level Methoden oder beiden kombiniert gestaltet werden. Das Problem dieser Version ist aber, dass STF-APIs in unterschiedlichen Page-Objects verteilt wurden und somit auf der Client Seite mehrere Objects-Referenzen als Eintrittspunkte vorhanden sind. Testschreiber müssen erst überlegen, welcher Service in welchem Page-Objekt definiert ist. Insgesamt gibt es zur Zeit 21 Page-Objects, somit ist es schon nicht einfach herauszufinden, welcher Service wo liegt.

Das andere Problem, das meiner Implementierung anzumerken wäre, betrifft die Vorgehensweise, wie ein Low-Level Methode einen Zustand bekommt. Wie ich im letzten Kapitel schon erwähnt habe, muss man eine Low-Level-Methode definieren, um SWTBot APIs exportieren zu können, die als Vertreter für den Aufruf einer Methoden-Verkettung mit SWTBot dient. Diese Transformation versucht einfach, leicht lesbares Methoden-Chaining mit SWTBot durch eine langnamige Methode

mit mehreren Parametern zusammenzufassen, was sehr schlechte Lesbarkeit und Bedienbarkeit verursacht.

Aus diesem Problem heraus fragte ich mich, wie STF bloß eine Erweiterung von SWTBot verteilte Systeme testen kann. Wieso müssen SWTBot APIs komplett anders definiert werden? Besonders die Einfachheit und leichte Lesbarkeit, die SWTBot uns bietet, sind durch die Transformierung wieder zurückgesetzt. SWBot bietet einen einzigen Eintrittspunkt für die ganzen APIs, so dass Benutzer schnell passende Methoden finden können. Aber STF versucht wieder, Benutzer durch mehrere Eintrittspunkten zu verwirren. Es läuft irgendwie nicht richtig. Aus diesem Gedanken heraus ist folgende Bots-Familie entstanden, deren Mitglieder nach dem gleichen Designmuster wie SWTBot entwickelt wurden und jeweils eine Erweiterung von SWTBot darstellen. In folgendem Kapitel wird genau erläutert, aus welcher Motivation die Bots entstanden sind und wie sie funktionieren.

Bots Familie

Die Bots-Familie (Abb. 7.4) besteht aus vier Klick-Robotern: SWTBot, SarosSWTBot, RemoteBot und SuperBot. Jeder von ihnen hat eigene Stärken und Schwächen. Aber gemeinsam sind sie eine richtig starke Mannschaft. Sie zusammen können dazu beitragen, dass Testschreiben sehr schnell und intuitiv vonstatten geht. Wobei SarosSWTBot eine Erweiterung von SWTBot ist und die Möglichkeit bietet, dass man auch auf die speziell für Saros definierten Widgets zugreifen kann. RemoteBot und SuperBot können zwar nicht selbst Mausklicks simulieren, bieten aber die Möglichkeit, den Remote-Befehl von Testern zu SarosSWTBot weiterzuleiten, so dass ein verteiltes Testen verwirklicht werden kann. Der Unterschied zwischen SWTBot und SuperBot liegt darin, dass SuperBot viel schneller, aber weniger flexibel als RemoteBot ist.

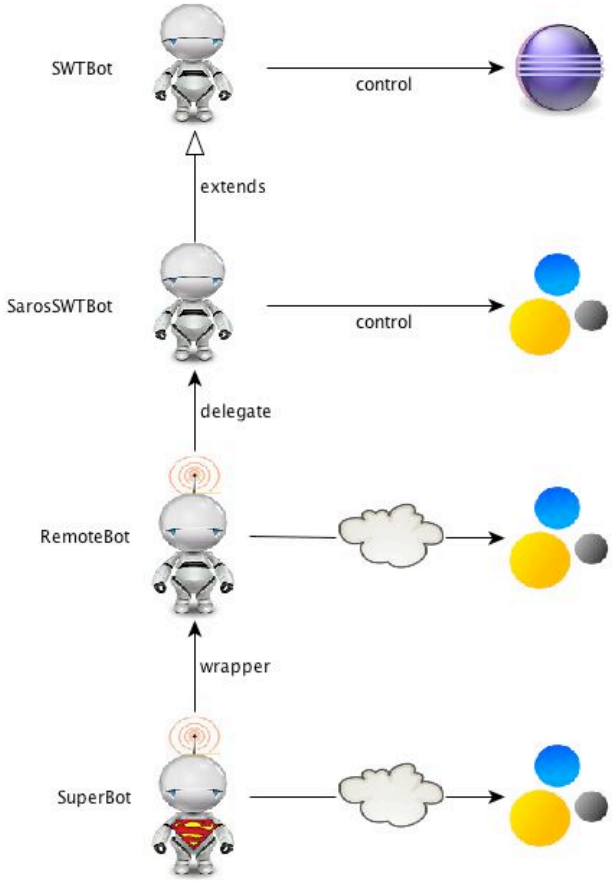


Abb. 7.4: Bots Familie

SarosSWTBot

SarosSWTBot ist ein “Klick-Roboter”. Er besitzt nicht nur die Fähigkeit die SWTBot hat, sondern kann auch auf die speziell für Saros definierten Widgets zugreifen und somit Benutzeraktionen simulieren.

Motivation

Was SWTBot bietet sieht schon ziemlich ansprechend aus, aber SWTBot ist noch ein sehr junges Projekt. Deshalb wird man mit Sicherheit hin und wieder an die Grenzen des Machbaren stoßen. Manchmal steht eine Funktion tatsächlich nicht zur Verfügung, die in der Dokumentation aber erwähnt ist. Dies ist dann besonders ärgerlich und zeitaufwendig. Zum Beispiel hat vor ein paar Monaten ein Student versucht einen einfachen Test mit STF zu schreiben, bei dem es nur darum ging eine Datei zu einem anderen Session-Teilnehmer zu schicken, wobei ein ausgelöster File Dialog bestätigt werden musste. Es vergingen ein paar Tage bis er schliesslich zu mir kam und sagte: ”Ich habe kein Lust mehr, es kostet mich zu viel Zeit! SWTBot unterstützt kein File Dialog!”. Es hatte so lang gedauert, bis er feststellte, dass SWTBot keine nativen Widgets wie File Dialog, Color Dialog usw. unterstützen, weil man volles Vertrauen an die Third-Party-Software und weniger an STF hatte.

Während ich an STF arbeitete, wurden große Umgestaltungen an der GUI Oberfläche vollzogen. Wobei Saros-Mitglied Björn Kahlert den Chatroom (Abb. 7.5) komplett „umgestyled“, die Usability und die visuellen Effekte wesentlich verbessert hat. Aber das Problem dabei ist, dass er eigene GUI-Widgets definierte, die nicht weiter von SWTBot unterstützt wurden.

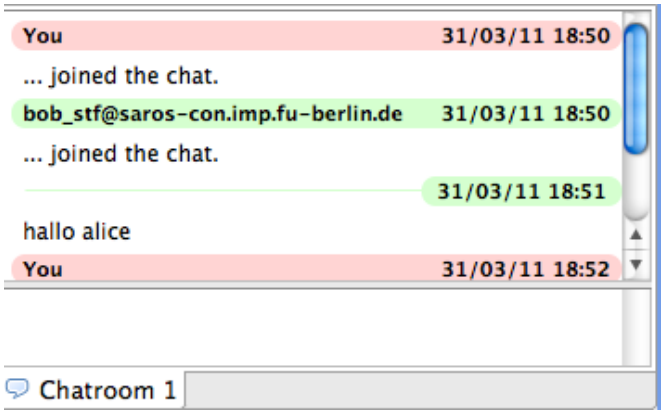


Abb. 7.5: Chatroom

Außer den oben erwähnten Problemen hat SWTBot auch andere Schwächen, wie z.B. die Tastatureingabe korrekt und lebensnah zu simulieren. Besonders bei der Tastaturausstattung der unterschiedlichen Betriebssysteme und der Sprache verhält es sich unterschiedlich.

Korrektur

Zur Zeit wird bei STF die SWTBot Version 2.0.0 verwendet. In dieser Version existieren noch ein paar kritische Bugs, die die Entwicklung von STF blockieren. Außer den vorher erwähnten Problemen mit Native Dialogs gibt es noch andere SWT-Widgets, die nicht wirklich von SWTBot unterstützt werden. Dazu gehören auch die Widgets Radio und Kontextmenü. Beide Widgets sind aber Hauptbestandteile der GUI-Oberfläche von Saros. Somit können die meisten Testfälle von Saros gar nicht getestet werden. Um die Entwicklung von STF voranzutreiben, habe ich in meiner Arbeit die beiden kritische Bugs so weit beseitigt, dass man problemlos die beiden Widgets finden und steuern kann. Vermutlich werden manche Bugs auch in der kommenden Version von SWTBot beseitigt werden (z.B. in dem Release-Notes v2.0.4[18] von SWBot vom Mar 04, 2011 wurde berichtet, dass das Problem mit SWTBotRadio.click() gefixt wurde).

Erweiterung

Um SWTBot mit den neu definierten Widgets eine Verbindung aufbauen zu lassen, sind zwei Schritte erforderlich. Erstens muss man für solche neuen Widgets entsprechende Wrapper-Klassen (sogenannte SWTBot[Widget]) definieren. Beispielsweise wurde Widget `ChatInput` mit `SWTBot-ChatInput` verpackt. Darin sind außer dem Konstruktor noch Funktionen wie `setText()` definiert, die die tatsächlichen Aktionen am Widget auslösen (siehe unteres Beispiel).

18 http://download.eclipse.org/technology/swtbot/helios/dev-build/RELEASE_NOTES.txt

```
public class SarosSWTBotChatInput extends AbstractSWTBot<ChatInput> {
    public SarosSWTBotChatInput(ChatInput w, SelfDescribing description)
            throws WidgetNotFoundException {
        super(w, description);
    }
    public SarosSWTBotChatInput setText(final String text) {
        waitForEnabled();
        asyncExec(new VoidResult() {
            public void run() {
                widget.setText(text);
            }
        });
        return this;
    }
}
```

Dann muss noch die Klasse `SarosSwTBot` erstellt werden, die von der Klasse `SWTWorkbenchBot` abgeleitet ist. Darin befinden sich diejenigen Methoden wie `chatInput`, die das neue Widget `ChatInput` anhand von Kriterien wie dem Indexattribut finden.

```
public class SarosSWTBot extends SWTWorkbenchBot {
...
      public SarosSWTBotChatInput chatInput(int index) {
        Matcher matcher = allOf(widgetOfType(ChatInput.class));
        return new SarosSWTBotChatInput((ChatInput) widget(matcher, 0), matcher);
    }
...
}
```

Mit dieser Vorgehensweise können alle neuen Widgets für SWTBot zugänglich gemacht werden. Das folgende Klassen-Diagramm veranschaulicht wie das neue Widget ChatInput ins SWTBot integriert wurde.

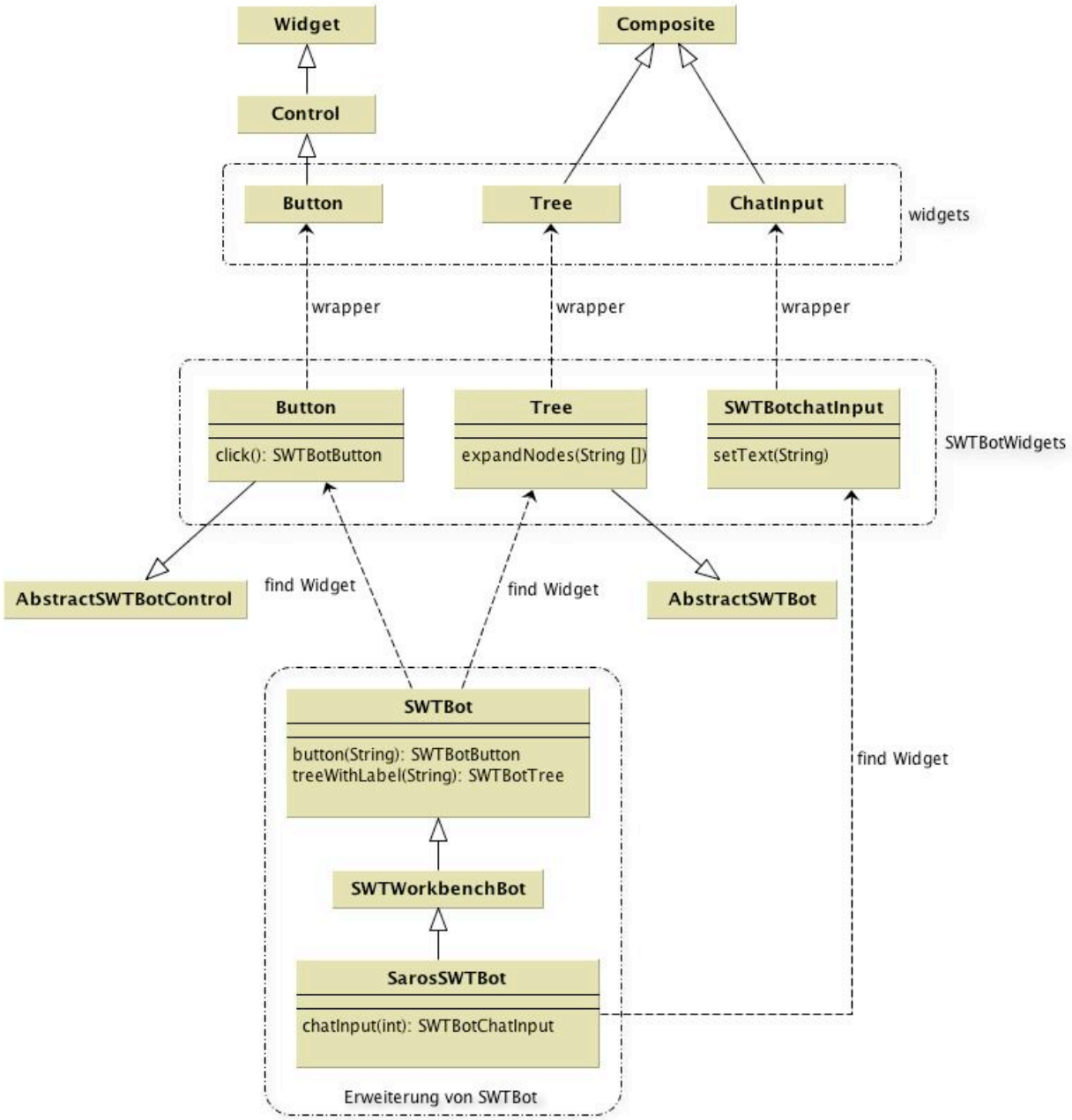


Abb. 7.6: Abschnitt der Klassendiagramm von SarosSWTBot

RemoteBot

RemoteBot ist ein “Remote-Klick-Roboter”. Wie der Name schon verrät kann er auf die UI der SWT und Eclipse Anwendung zugreifen, welche sich auf einem anderen Rechner befindet. Eigentlich ist RemoteBot bloß ein Vertreter von SWTBot; seine Aufgabe ist es den Auftrag von Client zum SWTBot weiterzuleiten. Er bietet ähnliche APIs wie SWTBot, somit “denkt” der Client immer, dass er SWTBot verwenden würde.

Motivation

Um zu verwirklichen, dass der Client direkt SWTBot-APIs verwenden kann, zielt der bisherige Lösungsansatz eher darauf ab, dass die möglichen Kombinationen einer Methoden-Kette jeweils in einer einzigen Methode fusioniert wurden. Diese Variante funktioniert zwar, bringt aber viele Probleme mit sich. Man kann sich gut verstellen, dass die komplette SWTBot-APIs in dieser Art umzuschreiben sehr aufwendig ist. Außerdem beträgt die Zahl der durch die Umwandelung der Methoden-Kette zur Low-Level-Methode entstandenen STF-APIs ein Vielfaches. Beispielsweise bietet die Klasse `SWTBot` 20 unterschiedliche Methoden um das Widget `Button` zu lokalisieren, welche Rückgabewerte mit gleichem Type `SWTBotButton` zurückliefern. Die Klasse `SWTBotButton` bietet wiederum ungefähr sechs Funktionen wie `click()`, `isActive()`, etc., somit gibt es insgesamt 120 Kombinationen, die fusioniert werden können. D.h. genauer, dass man für diesen Fall schon 120 entsprechende Methoden definieren muss. Diese expandierende Vorgehensweise ist in folgendem Diagramm deutlich zu sehen.

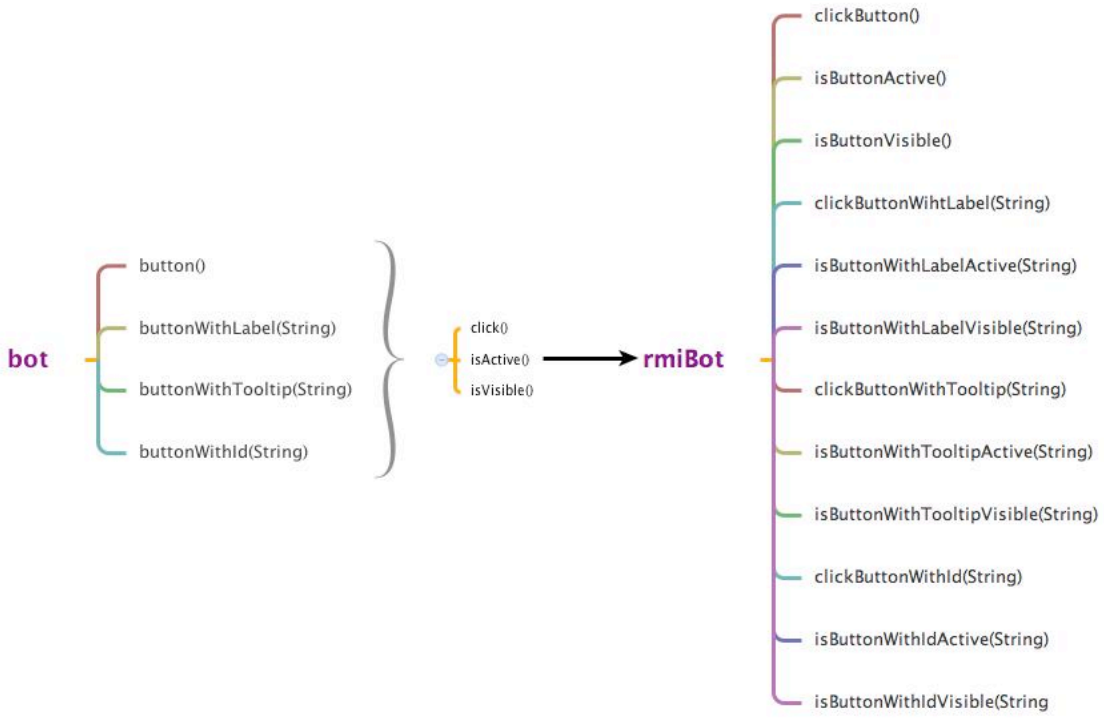


Abb. 7.7: Expandierende Low-Level Methode

Zusätzlich sind die entstehenden STF-APIs sehr verwirrend, unübersichtlich und somit fehleranfällig. Denn um die Klick-Pfade in einer Spur zu halten, haben die Methoden meistens sehr lange Namen und mehrere Parameter. Wenn ein Benutzer z.B. das Kontext Menü "Follow this buddy" von einem Session-Teilnehmer in der View "Saros" klicken möchte, kann er umgehend folgende Methode Verkettung durch SWTBot aufrufen, die den Klick-Pfad simuliert.

```
view("Saros").bot().tableInGroup("Session").getTableItem("bob_stf@saros-con.imp.fu-b
erlin.de").contextMenu("Follow this buddy").click()
```

Vergleichsweise sieht die daraus transformierte Methode ungefähr wie folgt aus:

```
clickContextMenuOfTableItemWithNameInViewWithTitle("Saros", "alice_stf@Saros-con.imp.fu-
Berlin.de", "Following this Buddy")
```

Wie wir sehen können ist es schwer genau zu wissen, wofür jeder der übergebenen Werte steht. So können wir beim bloßen Lesen des Codes fälschlicher Weise davon ausgehen, dass der Wert "Saros" für das Kontext-Menü steht. Außerdem gibt es ziemlich viele solcher Methoden, bei welcher Namen

sehr lang und ähnlich sind, wie bspw. `clickContextMenuOfTreeItemWithNameInViewWithTitle` und `clickContextMenuofTreeWithNameInViewWithTitle`, Somit ist es beinahe unvermeidbar, dass man beim Testschreiben versehentlich falsche Methoden benutzt.

Manchmal fragte ich mich wie es wäre, wenn STF genauso wie SWTBot mit dem Muster Fluent Interface gebaut wäre. Damit würden die vorher erwähnten Probleme automatisch beseitigt. SWTBot-Kenner wären dann automatisch mit STF-APIs vertraut. Übrigens, Fluent Interfaces ermöglicht beinahe in Satzform Funktionalität aufzurufen. Der daraus resultierende Methoden-Chaining spiegelt dann die Klick-Pfade der Benutzer wider. Somit ist es leicht lesbar und macht das Verständnis über dessen Intention leicht. Mit solchen sehr ausdrucksvollen Aufbau kann man sich ersparen zusätzliche Kommentare im Programmcode zu schreiben. Aus dieser Motivation heraus bin ich wieder auf den Gedanken gekommen, STF neu zu gestalten.

Lösungsansatz

Wie ich im früheren Kapitel schon erwähnt habe liegt das Hindernis darin, dass wir SWTBot APIs nicht direkt verwenden können, da bei der Benutzung der SWTBot-APIs meistens SWTBot[Widget]-Objekte zurückgeliefert werden, die nicht exportiert werden können. Aber wie wäre es, wenn wir für alle SWTBot[Widget]-Klassen jeweils entsprechende Wrapper-Klassen mit Remote-Interface definieren, die dann per RMI exportiert würden?

Mit diesem Ansatz kann man SWTBot-APIs komplett wrappen. Wobei `RemoteBot` die Wrapper-Klasse für `SWTBot` ist. Darin befinden sich diejenigen Methoden, die dann jeweils die entsprechenden (gleichnamigen) Methoden in der Klasse `SWTBot` aufrufen und Rückgabewerte vom Typ RemoteBot[Widget] liefern. Für die Unterklasse `SWTWorkbenchBot` von `SWTBot` gibt es auch eine entsprechende Wrapper-Klasse, in der sich gleichnamige Methoden wie die vom `SWTWorkbenchBot` befinden. Mit dem gleichen Prinzip wird für jedes SWTBot[Widget] eine entsprechende Wrapper-Klasse erstellt, e.g. `RemoteBotButton` für `SWTBotButton` und `RemoteBotTree` für `SWTBotTree`.

Struktur

Die nach dem Lösungsansatz entstandene Struktur ist fast gleich wie die bei SWTBot. `RemoteBot` ist dann die zentrale Klasse in dem RemoteBot-Framework; das durch dieses Verfahren entstandene Programmiergerüst wird als RemoteBot-Framework bezeichnet, welches ein Bestandteil von STF-Framework ist. Die Unterklasse `RemoteWorkbenchBot` wird für Eclipse-Anwendungen verwendet. Das Klassen-Diagramm (Abb. 7.8) zeigt einen Ausschnitt vom Ganzen wie „eins zu eins" SWTBot-APIs zu RemoteBot-APIs transformiert wurde.

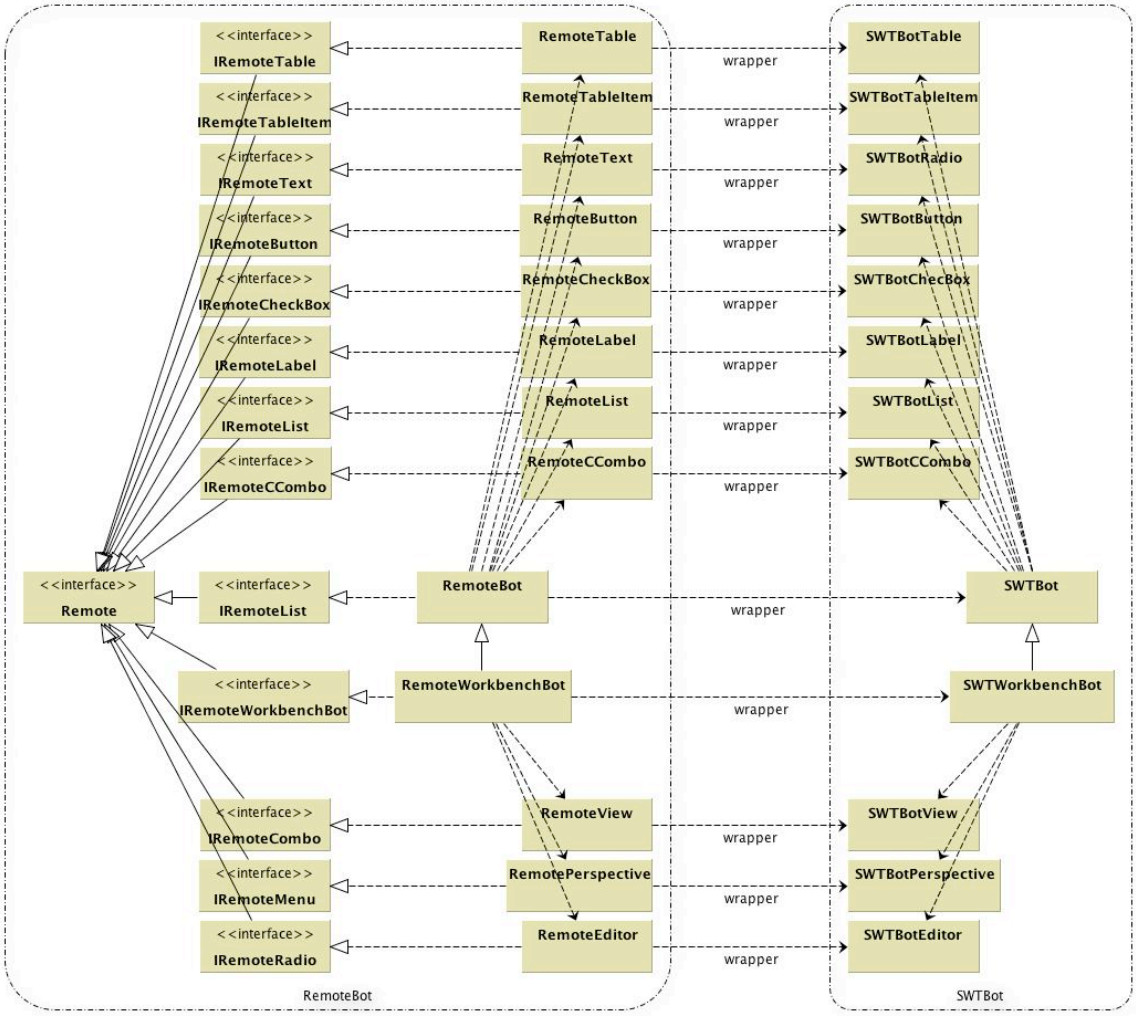


Abb. 7.8: Transformation von SWTBot zu RemoteBot

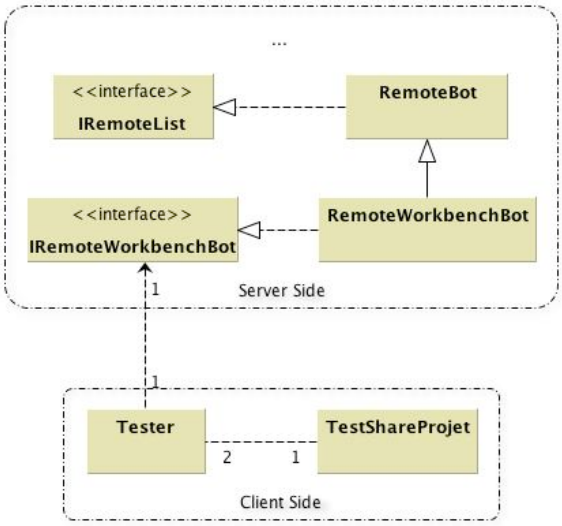


Abb. 7.9: Die Verbindung zwischen Client und Server

Anders als bei der Version "Page Object" haben wir hier zwar auch viele Remote-Objects definiert, die exportiert werden müssen, aber hier ist nur ein Eintrittspunkt für den Client nötig, um auf die ganzen RemoteBot-APIs zuzugreifen. Der Eintrittspunkt für den Client ist dann die Object-Referenz remoteBot, welche dem Remote-Interface `IRemoteWorkbenchBot` entspricht. Somit ist die Verbindung zwischen Client und Remote Objects einfach gehalten (Abb. 7.9).

Wenn wir uns direkt die Implementierung anschauen, dann ist deutlich zu sehen, dass RemoteBot bloß ein Vertreter von SWTBot ist. Das folgende Beispiel zeigt, dass die Methode `button` in der Klasse `RemoteBot` nicht mal versucht selbst das Widget Button mit übergebenem Parameter zu finden, sondern die Aufgabe direkt an SWTBot weiterreicht. Als Rückgabewert wird statt `SWTBotButton` dann `RemoteBotButton` Object zurückgeliefert.

```
public class RemoteBot extends STF implements IRemoteBot {
    private static RemoteBot button;
    private static SWTBot bot;
    …...
    public RemoteBotButton button(String mnemonicText) throws RemoteException {
        button.setWidget(bot.button(mnemonicText, 0));
        return button;
    }
……
}
```

Ebenfalls ist jedes Remote[Widget] auch bloß ein Vertreter von SWT[Widget]. Hier ist noch ein Ausschnitt der Klasse `RemoteBotButton` zu sehen.

```
public class RemoteBotButton extends AbstractRmoteWidget implements IRemoteBotButton {
    ……...
    private SWTBotButton widget;
    ……...
    public void click() throws RemoteException {
        widget.click();
    …...
    }
```

Test erstellen

Aus Client Sicht besitzt RemoteBot die gleiche Struktur wie SWTBot, wodurch man beim Test-schreiben nach dem gleichen Prinzip "Finden, Ausführen und Abfragen" vorgehen kann.

Finden der Widgets

```
Tester alice = initAlice();
IRemoteWorkbenchBot remoteBot = alice.remoteBot();

// find the Textbox that has a Label 'Username:'
IRemoteBotText text = remoteBot.textWithLabel("Username:")

//Easily identify buttons using the tooltip
IRemoteToolbarButton toolbarButton = remoteBot.toolbarButtonWithTooltip("Save");

//find windows using the window title
IRemoteBotShell shell = remoteBot.shell("New Project Wizard");
```

Ausführung der Aktion auf Widgets

```
text.typeText("Alice");
toolbarButton.click();
shell.activate();
```

Abfrage des Zustands der Widgets

```
String name = text.getText();
toolbarButton.isEnabled();
shell.isActive();
```

Fazit

Mit RemoteBot'APIs kann man jetzt fast alle möglichen Testfälle ohne Umweg direkt im JUnit-Tests automatisieren. Besonders vorteilhaft ist dabei, dass die Umgestaltung und die Verhaltensweise zwischen den beiden RemoteBot und SWTBot so ähnlich sind, dass man sich nur mit SWTBot A-PIs vertraut machen muss. Außerdem haben die RemoteBot-APIs wegen der Ähnlichkeit alle Vorteile wie Einfachheit und Lesbarkeit von SWTBot-APIs beibehalten. Zugleich sind die vorhin erwähnten Nachteile automatisch beseitigt.

SuperBot

SuperBot ist eine schnelle Version von RemoteBot und baut auf den beiden Mustern Page Object und Fluence Interface auf. Seine Arbeitsweise ist ähnlich der Vorgehensweise eines Menschen. Er fokussiert zuerst auf den Testbereich, wo sich eine Aktion befinden könnte, wählt die gewünschte Aktion aus und führt sie durch. Mit SuperBot ist Testschreiben so intuitiv und blitzschnell, da man ihm nur einmal "sagen" braucht, wo eine Aktion liegt und wie diese heißt.

Motivation

RemoteBot ist zwar flexibel, aber nicht effizient. Der Grund dafür ist, dass man sich beim Testschreiben immer Gedanken machen muss, welchen Typ das zu suchende Widget hat und welche Attribute als Suchkriterien zu übergeben sind. Je nachdem wo sich das Widget befindet, müssen alle davor liegenden Widgets auch nach gleicher Vorgehensweise fokussiert werden. Um bspw. das Szenario "Alice entfernt Bob von ihrer Buddiesliste" erfolgreich automatisieren zu können, muss man die Kenntnis über die präsentierten GUI-Bestandteile und ihre Struktur haben (siehe untenstehendes Beispiel).

```
 alice.remoteBot().view("Saros").bot().tree().selectTreeItem("Buddies",
"bob_test@jabber.ccc.de").contextMenu("Delete").click();
```

Außerdem ist es sehr wichtig zu überprüfen, ob die AUT[19] sich gerade in einem günstigen Zustand befindet, bevor man eine lange Methoden-Kette baut. Andernfalls bekommt man sofort die sogenannte `WidgetNotFoundException`, wenn der Testbereich gar nicht offen ist oder das zu suchende Widget sich nicht in diesem Bereich befindet. Um nun sicherzustellen, dass der vorliegende Test problemlos durchgeführt werden kann, muss noch folgender Testcode hinzugefügt werden, der dafür sorgt, dass die View "Saros" offen und aktiv ist.

19 Application Under Test

```
if(!alice.remoteBot().isViewOpen("Saros"))
        alice.remoteBot().openViewById("id of saros view")
alice.remoteBot().view("Saros").activate():
```

Wenn man jetzt andere Aktionen testen möchte, die sich im selben Testbereich befinden, muss man die ganzen Schritte erneut schreiben. Dies ist unnötige, sich wiederholende Arbeit, welche wiederum viele Code-Duplikate produziert.

Es ist besonders typisch, dass die Saros-Anwendung in einen ausreichend definierten Zustand gebracht werden muss, bevor ein Szenario erfolgreich getestet werden kann. Zum Beispiel muss, um den Verfolgermodus zu testen, vorher eine Saros-Sitzung hergestellt werden. Um diese herzustellen, müssen vorher alle Teilnehmer miteinander in der Buddiesliste stehen und online sein. Dafür muss man aber erst mal prüfen, ob alle Teilnehmer überhaupt in Kontakt sind. Um dies zu überprüfen, muss wiederum eine Verbindung hergestellt werden. Um eine Verbindung herzustellen, muss wiederum die Saros View geöffnet sein. So geht es weiter bis zum Startzustand der Saros-Instanz.

Es ist viel sinnvoller, wenn STF noch einen Mechanismus bietet, der dazu beiträgt, dass man ohne großen Aufwand schnell einen ausreichend definierten Zustand der Saros-Anwendung herstellen und ohne Wissen über Details sich schnell auf einen Testbereich oder Testelement fokussieren kann.

Eigentlich ist die früher definierte High-Level Methode genau dafür da, jeweils die sich oft wiederholenden Testschritte zusammenzustellen. Das Problem dabei ist, dass sie so unterschiedlich implementiert sind und man daraus keine feste Grammatik und Regeln schließen kann. Somit ist es nicht möglich genau zu wissen, welche Testschritte eine High-Level Methode genau abdeckt, ohne dass man vorher die entsprechende Implementierung überschaut. Das andere Problem ist, dass man eine passende High-Level Methode manchmal nicht schnell finden kann. Wie ich vorher schon erwähnt habe, sind die High-Level Methoden jeweils dem entsprechenden Page Object zugeordnet. Dieser Verwaltungsmechanismus bringt zwar Strukturierung und Ordnung, liefert aber auch die Schwierigkeit, schnell die passende Methode zu finden. Wegen der Usability kann der gleiche Service (Aktion) jeweils an unterschiedlichen Stelle erscheinen. Zum Beispiel bietet Saros drei Möglichkeiten,

um ein Buddy hinzuzufügen (Abb. 7.10). Der Dialog “AddBuddy“ kann aus dem Menü Saros > mit “Add Buddies“, mit dem ToolbarButton > “Add a New Buddy“ oder aus dem KontextMenü der Buddies Node ausgelöst werden. In so einer Situation ist es dann schwer zu entscheiden, ob die dem Service entsprechende Methode `addBuddy()` in Page Object `SarosM` (welches Menü “Saros” modelliert) oder in `SarosView` (welches View „Saros“ modelliert) eingeordnet werden soll. Wenn dies nur in einem PageObject definiert würde, so ist die Gefahr hoch, dass der Testschreiber die Methode nicht findet. Andererseits, wenn die Methode in beiden Page Objects definiert würde, entsteht auch viel redundanter Code.

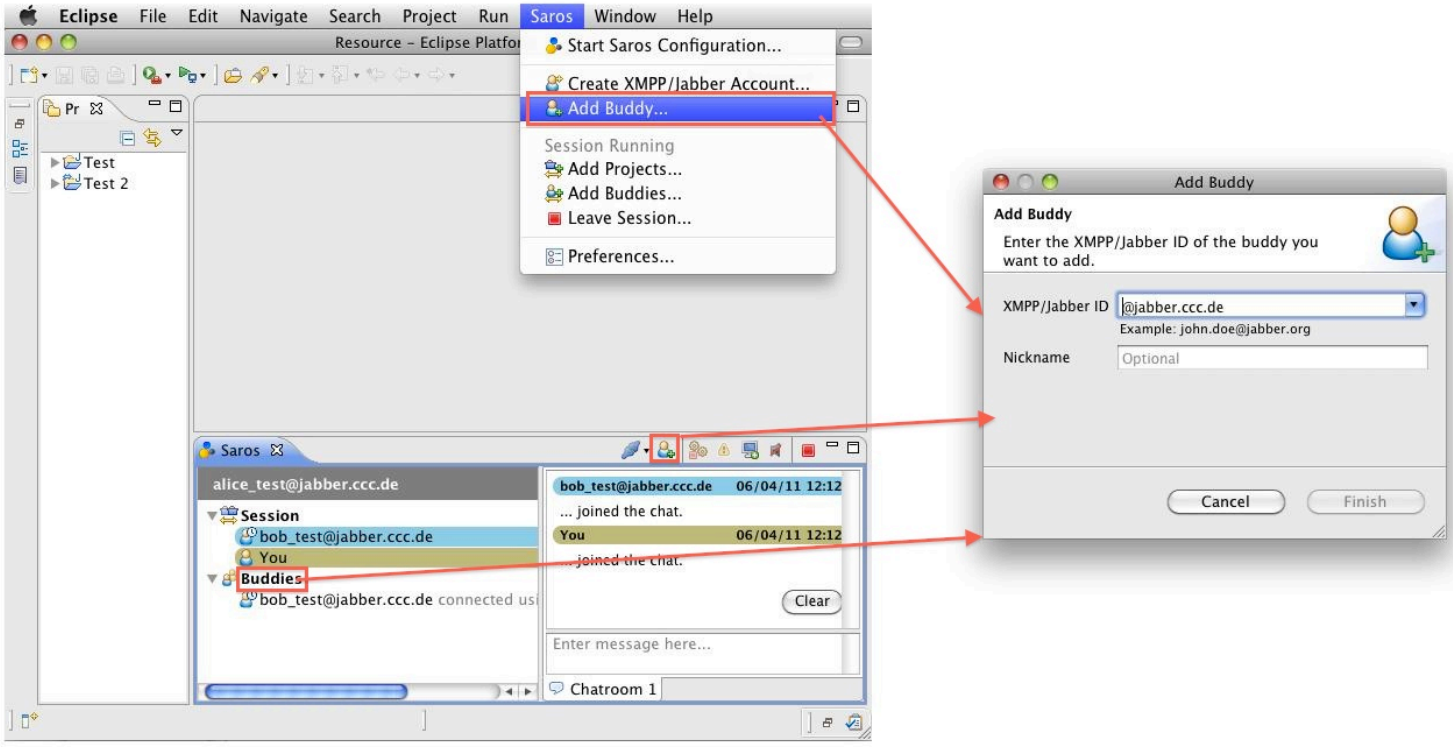


Abb. 7.10: Die mögliche Klick-Pfade zum Auslösen des Dialog “Add Buddy”

Lösungsansatz

Wie wäre es dann, wenn die High-Level Methode analog zu RemoteBot auch mit dem Muster Fluent Interface verwaltet werden könnte?

Die Vorgehensweise ist eigentlich relativ einfach. Page Objects werden jeweils durch eine Methode interpretiert. D.h., diese Methode liefert das Page Object als Rückgabewert zurück und sorgt dafür, dass die vom Page Object modellierte Eclipse-Komponente open und aktiv ist. Beispielsweise wird für Page Object `SarosView` die Methode `sarosView()` definiert, die sicherstellt, dass nach dem Aufruf der Methode diese Saros View geöffnet und aktiviert ist. Bei Page Objects, die im selben Package liegen, werden deren entsprechenden Methoden auch in derselben Klasse eingeordnet. Z.B, die Methode `sarosView()`, `packageExplorerView()`, `consoleView()`, `progressView()` werden in der Klasse `Views` eingeordnet und die Methoden `sarosM()` und `windowM()` in der Klasse `Menubar`. Die Objekte `Views` und `Menubar` kann man jeweils durch den Methodenaufruf `views()` und `menubar()` erhalten, die dann in der zentralen Klasse `SuperBot` definiert sind. Damit werden Page Objects nach dem Muster Fluent Interface umgestellt. Das unten stehende Bild verdeutlicht, woher die Methoden-Ketten-Zustände kommen und welche Komponente durch welche Methode präsentiert wird.

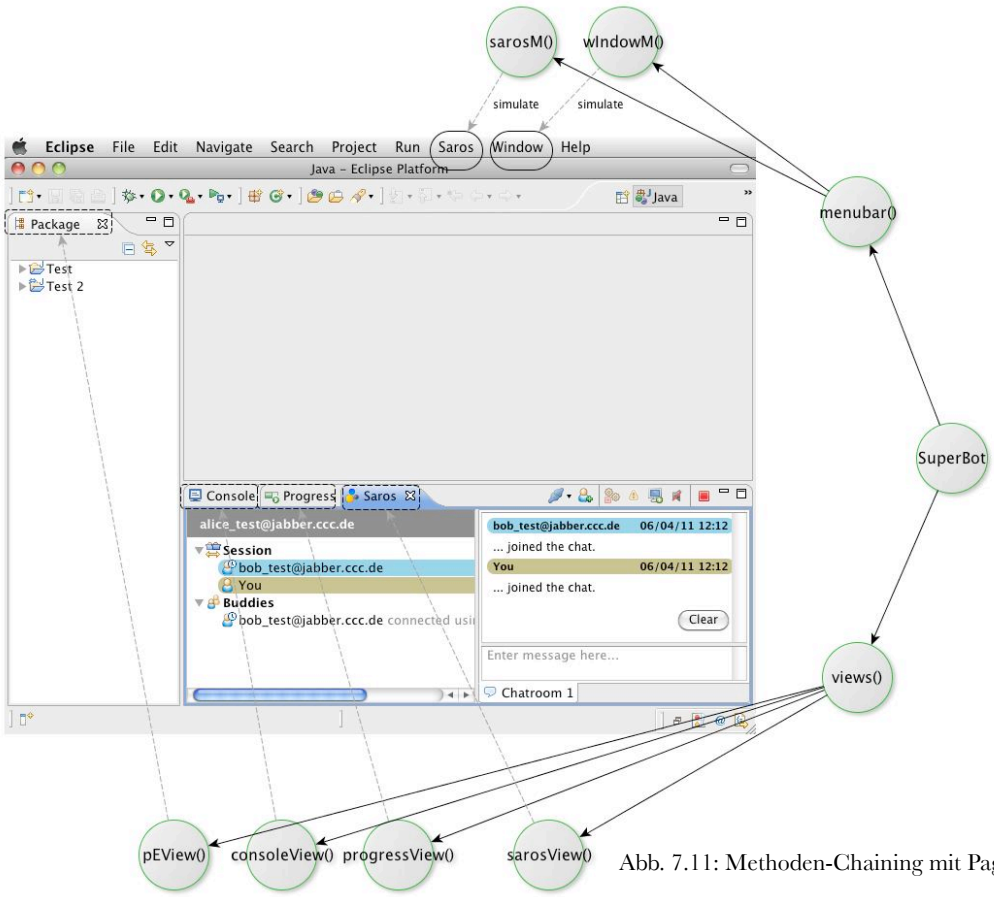


Abb. 7.11: Methoden-Chaining mit Page-Objects

Um die Komplexität des Frameworks zu minimieren und somit sich mehr auf Saros-Features zu fokussieren, wird bei dem Entwurf nach folgenden Regeln vorgegangen.

- Nur die zum Saros-Tests beitragenden Eclipse-Aktionen werden implementiert. Denn das STF-Framework ist auf Saros-Feature und nicht auf Eclipse ausgerichtet.

- Eclipse-Aktionen, die jeweils mit unterschiedlichen Stellen wie Menubar oder KontextMenü verbunden sind, werden nur einmal implementiert, wobei die Variante KontextMenü höchste Priorität genießt. Deswegen werden in diesem Modell nur das Menü "Saros" und "Window" modelliert. Aktionen, die sich in anderen Menüs befinden, werden zentral in der Klasse `ContextMenuWrapper` definiert.

- Alle in Saros-Features beinhaltenden Komponenten müssen modelliert werden.

Vorgehensweise

Mit dieser neu "eingebauten" Struktur kann der Client von dem Eintrittspunkt SuperBot durch Aufruf der Methoden-Kette z.B. `views().sarosView()` auf die View "Saros" fokussieren, ohne dass man vorher überprüfen muss, ob diese View offen ist oder nicht. Entlang dieser Methode `SarosView()` (die Page Object `SarosView` zurückliefert) sind viele High-Level Methoden zur Auswahl vorhanden, die jeweils eine Aktion in der View-Symbolleiste interpretieren. Z.B. die Aktionen "Connect", "Add a New Buddy", "Lease Session", die als Elemente in der Symbolleiste vorhanden sind, werden durch die entsprechenden High-Level Methoden `connect, addAnewBuddy,leaveSesion` interpretiert. Eine solche High-Level Methode behandelt nicht nur Klick-Aktionen, sondern auch die dadurch ausgelösten Events, falls es welche gibt. Zum Beispiel behandelt die Methode `addAnewBuddy` die Klick-Aktion des ToolbarButton mit dem TooltipText "Add a New Buddy" und bestätigt auch das ausgelöste Popupfenster. Da ein Popupfenster durch unterschiedliche Klick-Pfade ausgelöst werden kann, wird die Behandlung eines Popupfenster extra in einer einzelnen Methode definiert, die dann in der Klasse `SuperBot` eingeordnet wird. Aus diesen Regeln sieht die High-Level Methode `addANewBuddy` beispielsweise dann so aus:

```
public void addANewBuddy(JID jid) throws RemoteException {
        if (!hasBuddy(jid)) {
            clickToolbarButtonWithTooltip("Add a New Buddy");
            superBot().confirmShellAddBuddy(jid);
        }
}
```

Bezogen auf die View "Saros" können momentane GUI-Eigenschaften mit der Methode `isConnect`, `isInSession`, `existsBuddy` abgefragt werden.

So weit, so gut. Lassen wir den Test "Alice fügt den Buddy Bob ein" mal mit SuperBot versuchen.

```
alice.superBot().views().sarosView().addBuddy(bob.getJID());
bob.superBot().confirmShellRequestOfSubscriptionReceived();
assertTrue(alice.superBot().views().sarosView().existsBuddy(bob.getJID()));
```

In diesem Beispiel ist deutlich zu sehen wie intuitiv es mit SuperBot geht! Wir brauchen nur SuperBot zu sagen, wo eine Aktion liegt und wie sie heißt, ohne uns dabei Gedanken zu machen, ob Saros View offen und aktiv ist und ohne selbst überprüfen zu müssen, ob der übergebende Buddy schon existiert oder nicht. Diese werden federführend von SuperBot übernommen. Interessant dabei ist, dass die daraus resultierende Methodenkette irgendwie die Klick-Pfade der Benutzer widerspiegelt, die aber nicht durch ein Widget, sondern durch den konkreten Komponenten- oder die GUI Elementenamen präsentiert wird. Somit ist der entstandene Test-Code sehr klar und verständlich. Auch beim Schreiben kann man mit Hilfe der Autovervollständigung[20] sehen, welche Funktionen als nächstes zulässig sind. Wenn wir das Szenario durch andere Klick-Pfade wie Menü "Saros" -> "Add a New Buddy" automatisieren möchten, brauchen wir dem Bot nur mitteilen, wie der andere Weg heißt.

```
alice.superBot().menuBar().saros().addANewBuddy(bob.getJID());
bob.superBot().confirmShellRequestOfSubscriptionReceived();
```

20 http://de.wikipedia.org/wiki/Autovervollständigen

Dieses Szenario kann aber auch über den Weg des Kontextmenü-Befehls “Add a New Buddy“ der Buddies Knoten getestet werden. Aber wie greift man mit SuperBot auf die Kontextmenüs zu? Wir wissen, dass das Kontextmenü nur erscheint, wenn man ein GUI-Element oder einen Bereich markiert und dann die rechte Maustaste klickt. Um diese Verhaltensweise zu simulieren, muss die Markierung und damit interaktive Kontextmenüs modelliert werden. Bevor wir ins Detail gehen, lassen Sie uns erst mal schauen, welche Elemente in der View “Saros” kontextsensitiv sind. Um die Komplexität zu minimieren, werden nur Kontextmenüs in den Views “Package Explorer” und “Saros” betrachtet, da diese meistens bei Saros-Szenarios verwendet werden.

1. Der View-Reiter

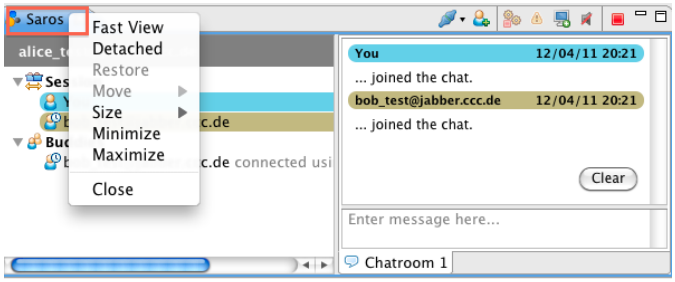


Abb. 7.12: View-Reiter “Saros”

2. Der Bereich Innerhalb der Saros View

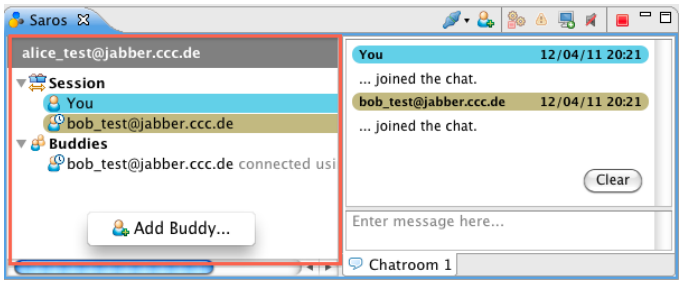


Abb. 7.13: Bereich innerhalb des Saros View

3. Ein GUI-Element innerhalb der Saros View

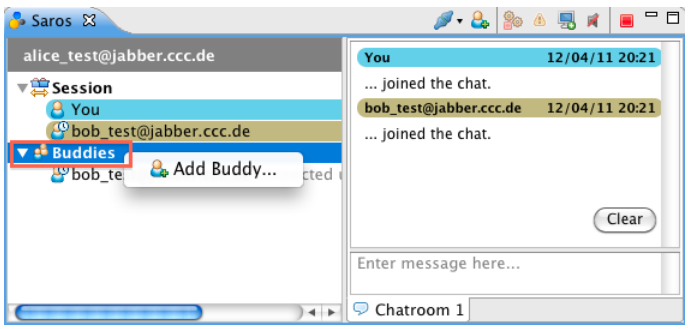


Abb. 7.14: TeeItem “Buddies”

Mit dieser Kenntnis können wir dann folgendes modellieren:

- Die Methode `bot()` interpretiert die Markierung des Bereiches innerhalb eines View.
- Die Methode mit dem Muster `select[GUI-Element-Name]` wie z.B. `selectBuddies()` interpretiert dann die Markierung eines GUI-Element innerhalb eines View.

Wobei das Kontextmenü auf dem View-Reiter nicht weiter von mir betrachtet wird, weil es wenig mit Saros-Szenarien zu tun hat.

Um die kontextsensitiven Interaktionen mit einem markierten Element oder Bereich zu simulieren, werden zwei Klassen, `ContextMenuWrapper` und `SarosContextWrapper` definiert, die die Kontextmenüs jeweils innerhalb der PackageExplorer View und Saros View modellieren. Die in der Klasse befindlichen Services (Public-Methode) können nur erreichbar sein, wenn die Methode `bot()` oder eine Methode mit dem Muster `select[GUI-Element-Name]` vorher aufgerufen wurde. Jetzt lassen wir das gleiche Szenario noch mal über den Weg im Kontextmenü "Add a New Buddy" der Buddies Knoten gehen.

```
alice.superBot().views().sarosView().selectbuddies().addBuddy(bob.getJID());
bob.superBot().confirmShellRequestOfSubscriptionReceived();
```

In diesem Beispiel ist sind Klick-Pfade: View "Saros" -> TreeItem "Buddies" -> Kontextmenü "Add Buddy" durch das Methoden-Chaining deutlich zu erkennen. Diese Modellierung verschafft nicht nur die Möglichkeit, leicht lesbaren Code zu generieren, sondern reduziert auch Duplizität. Denn die Zwischenschritte wie z.B. sarosView() und selectBuddies() müssen nur einmal definiert werden.

Diese Regeln gelten auch für andere Views. Beispiel: Entferne ein Java-Projekt über das Kontextmenü "Delete" innerhalb der Package Explorer View. Dafür sind folgende Schritte nötig:

1. Öffnen und Aktivieren der Package Explorer View

2. Markieren des Java Projekts, das zu löschen ist

3. Klicken des KontextMenüs “Delete”

4. Ausgelöstes Popupfenster mit OK bestätigen

Diese vier Schritte können mit folgendem Methoden-Chaining interpretiert werden, wobei die letzten beiden Schritte der Methode `delete` behandelt werden.

```
superbot().views().packageExplorerView().selectJavaProject(“Foo-Bar”).delete();
```

Um die Klick-Pfade noch lebensnaher zu simulieren und gleichzeitig eine bessere Ordnung und Strukturierung zu schaffen, wird für das Kontextmenü, welches noch Submenüs enthält, zusätzlich eine Klasse definiert, in der sich Methoden befinden, die entsprechende Submenüs interpretieren. Nach unserem Testbedarf werden momentan insgesamt vier Kontextmenüs “Share With”, “New”, “Refactor” und “Team” in der Package Explorer View modelliert. Z.B. sorgt die Methodenkette `superBot().views().PackageExplorerView().bot().new().javaProject("Foo-bar")` dafür, ein Java-Projekt mit Namen “Foo-Bar” zu erzeugen. Mit `superBot().views().PackageExplorerView().selectJavaProject("Foo-bar").refactor().rename("Foo-bar-new")` wird der Projektname wieder auf “Foo-bar-new” geändert. Sollte Bob sich an diesem Projekt beteiligen, ist mit `superBot().views().PackageExplorerView().selectJavaProject("Foo-bar-new").shareWith().buddy("bob_test@jabber.ccc.de")` dann eine Einladung sofort zu Bob geschickt. Auf diese Art und Weise Tests zu schreiben ist nicht nur intuitiv, sondern macht richtig Spaß!

Die folgende Abbildung ist ein Ausschnitt der Implementierung, wie Klick-Pfade über Kontextmenüs durch das Methoden-Chaining simuliert wird.

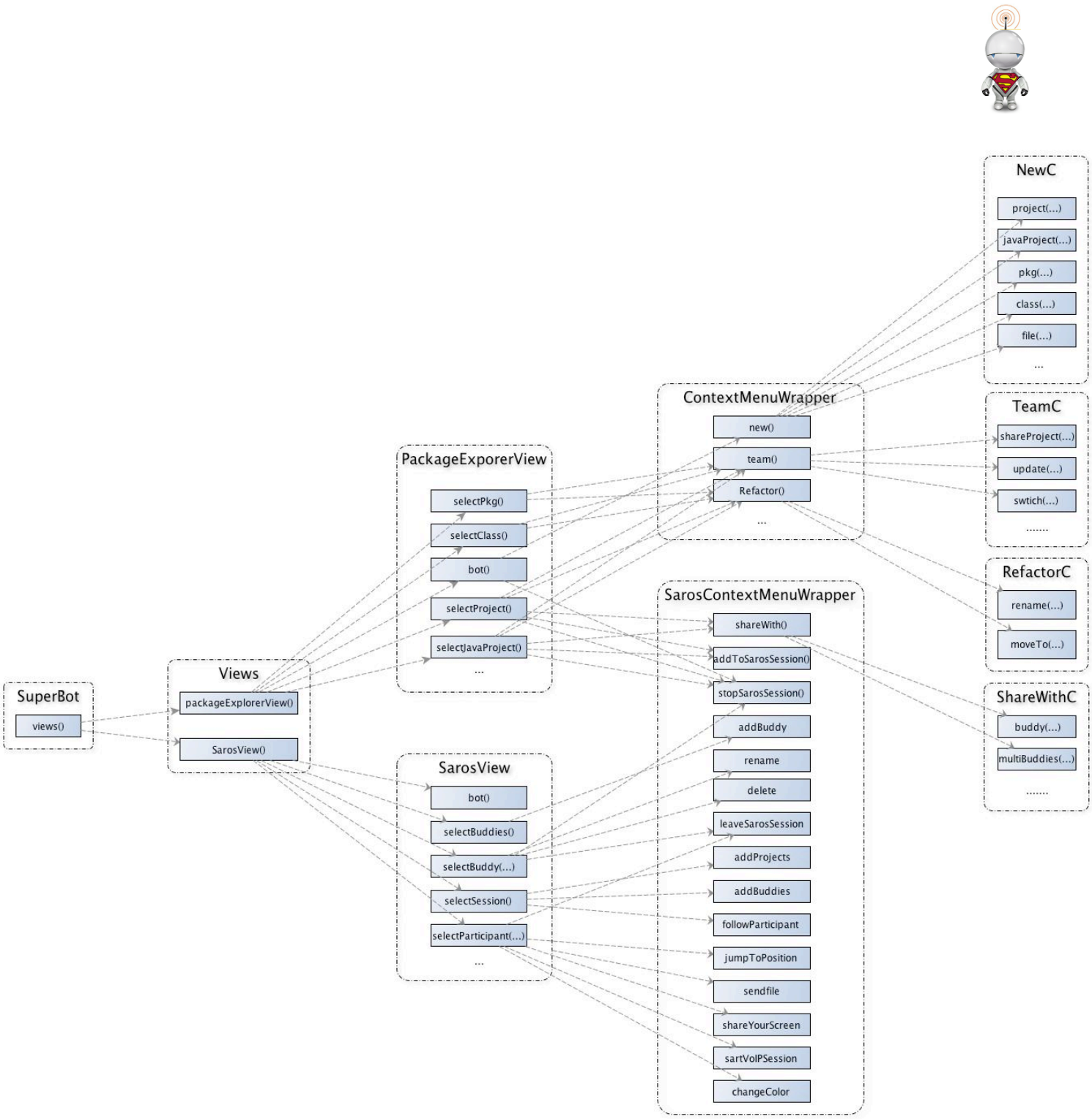


Abb. 7.15: Methoden-Chaining über Kontextmenü

Struktur

Die nach dem Lösungsansatz entstandene Struktur ist wie ein Baum (Abb. 7.16). Wobei `SuperBot` den Startknoten darstellt. Daraus kann man iterativ entlang eines Weges auf eine bestimmte SuperBot-API zugreifen. In diesem Diagramm ist noch die Klasse `SarosPreference` zu sehen, die den Saros-Preferences-Dialog präsentiert, welches durch den Pfad Menü Saros -> Preferences… ausgelöst werden kann. Saros-Preferences-Dialog ist für die Konfiguration zuständig. Damit kann man Aktionen wie "Activate Account", "Add Account", "Delete Account", "Edit Account" usw.. durchführen, welche durch Methoden in der Klasse `SarosPreference` interpretiert werden. Mit dieser Modellierung kann man jetzt mit `superbot().menubar().saros().perferences().addAccount(…)` einen Account einfügen oder mit `superbot().menubar().saros().perferences().deleteAccount(…)` diesen Account wieder löschen.

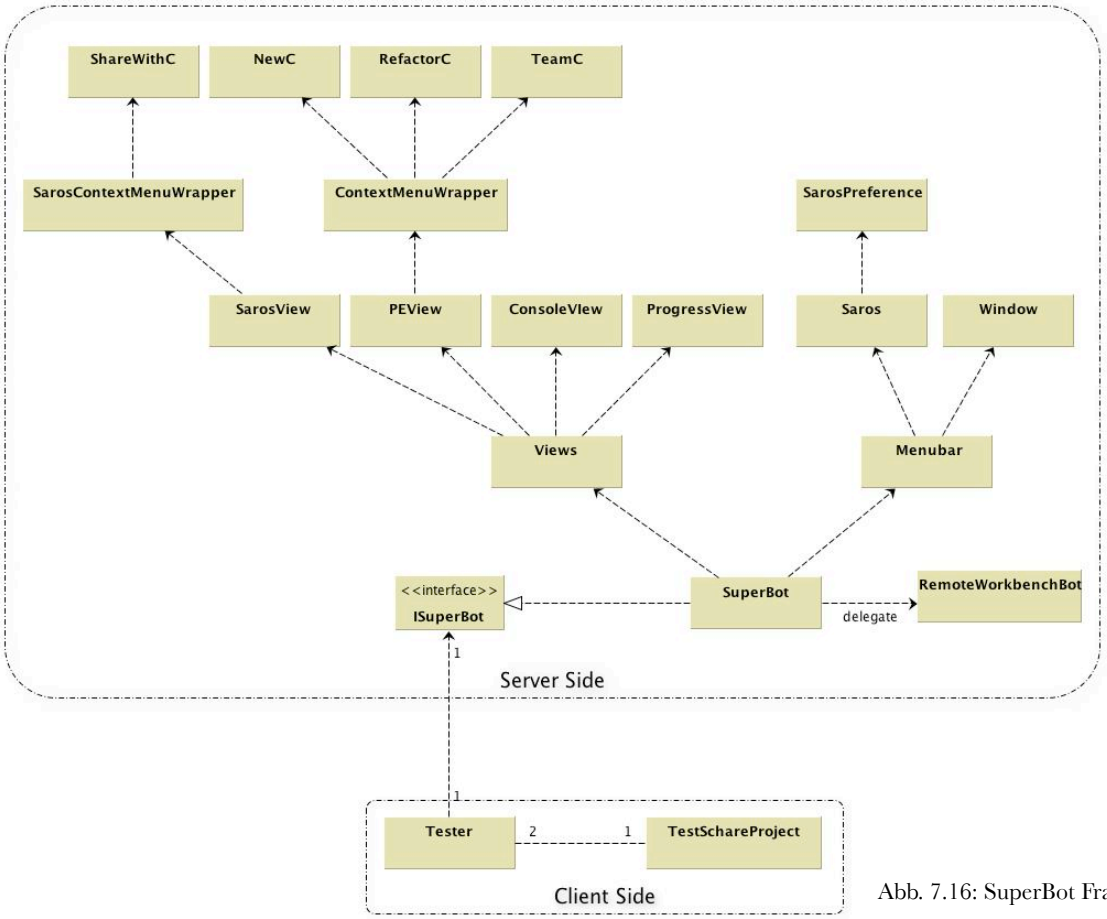


Abb. 7.16: SuperBot Framework

Für den Client ist hier auch nur ein Eintrittspunkt nötig, nämlich die Objekt-Referenz superBot, welches dem Remote-Interface `ISuperBot` entspricht.

Fazit

Was SuperBot kann, kann RemoteBot auch, aber nicht umgekehrt, da die Implementierung von SuperBot begrenzt ist. SuperBot ist zwar weniger flexibel, dafür aber viel schneller programmierbar als RemoteBot. Um nachvollziehen zu können, lassen wir die beiden Bots einfach das gleiche Szenario "alice add new Buddy bob" testen.

Mit RemoteBot:

```
alice.bot().view("Saros buddies").toolbarButton("Add a New Buddy").click();
alice.bot().waitUntilShellIsOpen("Add Buddy");
alice.bot().shell("Add Buddy").activate();
alice.bot().shell("Add Buddy").bot()
alice.bot().shell("Add Buddy").bot().comboBoxWithLabel("XMPP/Jabber ID").setText(bob.getJID().getBase());
alice.bot().shell("Add Buddy").bot().button("Finish").waitUntilIsEnabled();
alice.bot().shell("Add Buddy").bot().button("Finish").click();
bob.bot().waitUntilShellIsOpen("Request of Subskription Received");
bob.bot().shell("Request of Subskription Received") .activate();
bob.bot().shell("Request of Subskription Received").bot().button("Ok").click();
```

Mit SuperBot:

```
alice.superBot().views().buddiesView().addANewBuddy(peer.getJID());
bob.superBot().confirmShellRequestOfSubscriptionReceived();
```

Hier ist deutlich zu sehen, wie schnell es mit SuperBot geht. Damit entstandene Testcodes sind auch super prägnant. Aber wegen der fest codierten High-Level Methode ist SuperBot eher dafür geeignet, die Saros-Anwendung schnell in einen ausreichend definierten Zustand zu bringen und darauf gewöhnliche Aktionen durchzuführen, wie z.B. das Erstellen einer Saros-Sitzung, Löschen aller Projekte, usw. . Wenn man mehr Details testen will, ist dann RemoteBot sehr hilfreich. Zum Beispiel, ob

eine Warnungsmessage erscheint, wenn man versucht einen ungültigen Server-Namen bei einem Account einzugeben. Wie gesagt, die beiden Bots können sich sehr gut gegenseitig ergänzen. Zusammen bieten Sie eine starke Leistung.

Zukunft

Nach vier Veränderungen ist STF wesentlich stabiler und leichter bedienbar geworden. Aber es gibt sicherlich noch viele Stellen, welche STF weiter optimieren können. In diesem Abschnitt werden drei weitere mögliche Verbesserungsvorschläge aufgelistet, deren Umsetzung wegen der vorgesehenen Zeit nicht möglich waren.

Robustheit

Das Problem der "zerbrechlichen" Oberflächentests durch die Iterationsvorgänge tritt bei Saros besonders deutlich in Erscheinung. Da STF-Tests direkt auf der Benutzerschnittstelle aufsetzen, sind sie äußerst anfällig gegen Veränderungen. Der Wartungsaufwand, die Tests stets lauffähig und damit aussagekräftig zu halten, ist gewaltig hoch. Die Auseinandersetzung des vorliegenden Problems führte mich zu dem Gedanken, ob es möglich wäre, nicht ständig den Testcode anpassen zu müssen, auch wenn die Maske der Anwendung umgebaut wird.

Eine Lösung für dieses Problem besteht darin, für die Widgets der Anwendung IDs zu vergeben, die SWTBot den Weg weisen:

```
Label passwordLabel = new Label(this, SWT.NONE);
paswordLabel.setText("password");
passwordLabel.setData("de.fu_berlin.inf.dpp.widget.key", "passwordId")
```

Mit `setData` werden einem Widget Zusatzinformationen angefügt. Der Key kann mit einer Konstanten wie. z.B. `de.fu_berlin.inf.dpp.widget.key` definiert werden. So markierte Widgets können mit dem Muster [Widget]WithId lokalisiert werden. Man kann bspw. statt `bot().label("password")` auch mit `bot().labelWithId("passwordId")` Label identifizieren (vgl. [Ebert09]).

Dies funktioniert besonders robust, da das Widget eindeutig markiert und identifiziert wurde. Auch wenn die Beschriftungen der Oberfläche geändert werden, sind keine Änderungen am Testcode mehr notwendig. Aber das Problem dabei ist, dass nachträglich alle Widgets IDs zu vergeben sehr aufwendig ist. Es wäre einfacher gewesen, wenn Entwickler es während der Implementierung schon getan hätten. Ein anderer Nachteil dieser Variante liegt darin, dass der entstandene Test-Code nicht so lesbar wie unter zu Hilfenahme der Textattribute ist.

Eine andere Lösungsmöglichkeit wäre, dass Saros-Anwendungen über die Auslagerung der Texte in Property-Dateien (Java Resource Bundes) übersetzt werden. D.h., alle GUI-Labels der Saros-Anwendung werden zentral in Property-Dateien verwaltet, die wiederum von STF-Tests referenziert sind. Der Vorteil ist, dass man die Beschriftung nur einmal ändern müsste - nämlich in den Property-Dateien - und es überall geändert wird, wo es auf diese Stelle referenziert. Dieser Lösungsansatz ist meiner Meinung nach nicht nur aus Testsicht vorteilhaft, sondern bietet auch die Möglichkeit doppelte Label-Definitionen zu vermeiden und einen positiven Ausgangspunkt bereitzustellen, die zur Lokalisierung beitragen könnte, falls in Zukunft Saros auch in weitere Sprachen unterstützen werden sollen. Übrigens ist die Realisierung dieses Verfahrens nicht sehr aufwendig, denn Eclipse bietet unterstützende Tools für die Auslagerung der Texte.

STF als einziges Plugin

Am Anfang habe ich versucht das ganze STF-Projekt in ein separates Plug-in abzulegen, um sicherzustellen, dass die Abhängigkeiten der Anwendung frei von den Tests bleiben. Leider ist dieser Versuch gescheitert, da der Startmechanismus der Saros-Instanz als Test-Mode dazu führt, dass die Saros-Anwendung von STF abhängig ist, so dass eine Abhängigkeitszyklus entstanden wäre. Um diese Aufgabe zu meistern wird vorausgesetzt, dass keine Abhängigkeitszyklus existieren dürfen. Aus diesem Problem heraus habe ich entschlossen, dass der bisherige Startmechanismus anders ablaufe sollte.

Tests-Ausführung

Neben den zuvor erwähnten Problemen wurde dieser Startmechanismus mehrmals kritisiert, da die halb automatisierte Vorgehensweise sehr verwirrend ist. Um STF-Test unter der lokalen Entwicklungsumgebung durchzuführen, müssen zuerst die Saros-Instanzen im Test-Modus per Hand gestartet werden, bevor STF-Tests als JUnit Test ausgeführt werden darf.

Für nicht verteilte Plugin-Anwendung bietet das JUnit die Möglichkeit, Plugin bezogene Unit Tests in einem speziellen Modus zu starten. Dieser entspricht weitestgehend dem Starten eines Moduls um diese bei der Entwicklung zu testen. Hierbei wird durch JUnit mit einem speziellen Launcher in einer eigenen VM eine neue Instanz von Eclipse mit eben diesem neuen Plugin gestartet. In dieser Instanz wird dann der Test gestartet (vgl. [Fern10]). Im übertragenden Sinne überlege ich, ob es möglich wäre, einen eigenen Launcher für STF-Tests zu implementieren, der dafür sorgt, bei der Tests-Ausführung mehrere Instanzen automatisch zu starten. Eine weitere Überlegung ist, ob es einen Weg gibt, die Eclipse Launch-Konfiguration programmatisch zu starten. Wegen der nicht vorhersehbaren Komplexität und dem Mangel an Zeit zur Evaluation kann ich schwer sagen, ob meine Überlegungen realisierbar sind.

8 Fazit

Wir sind auf dem richtigen Weg!

Der technische Leiter Karl Beecher bemüht sich, den Build-Server Hudson wieder zum Laufen zu bringen, welcher für die regelmäßige Ausführung des Integrationsprozesses sowie dem Regressionstests sorgt.

Student Phillip Cordes erstellt zur Zeit im Rahmen seiner Masterarbeit einen JUnit Framework für Saros, der dazu beiträgt, dass ohne aufwendiges Setup einfache JUnit Tests schnell erstellt und ausgeführt werden können.

Ich bemühe mich STF so zu erweitern, dass man durch simulierte Klick-Pfade schnell, stabile und leicht lesbare Integrationstests schreiben kann. STF bietet, vorbereitet durch klar strukturierte Designmuster, gute Abwehrmechanismen beim Kampf gegen die Labilität von GUI-Tests. Der Einsatz von STF sollte die Lücken zwischen Unit- und IT-Tests schliessen und die Fehler in der Netzwerkschicht abdecken.

Die neuen in den Testprozess einfliessenden Beiträge werden sicherlich positive Auswirkungen auf die Häufigkeit der Testdurchführung und die Arbeitsmoral der Entwickler haben. Das Szenario, dass der Entwickler das Testen seiner Arbeit während der Implementierungsphase als unnötiges Übel ansieht, wird sicherlich bald der Vergangenheit angehören. Denn automatisierte Tests werden durch Hudson häufiger durchgeführt und somit wird es sofortige Rückmeldungen an die Entwickler geben. Erfolgreiche Änderungen und neue Funktionen werden zeitnah von CI-System positiv bestätigt. Es lässt sich mutiger entwickeln und gefahrlos Neues ausprobieren.

Wir sind schon auf dem richtigen Weg zu einem modernen und voll automatisierten Testprozess!

Motivation

How to click the toolbar button with the tooltip text “Connect” in the saros view with SWTBot?

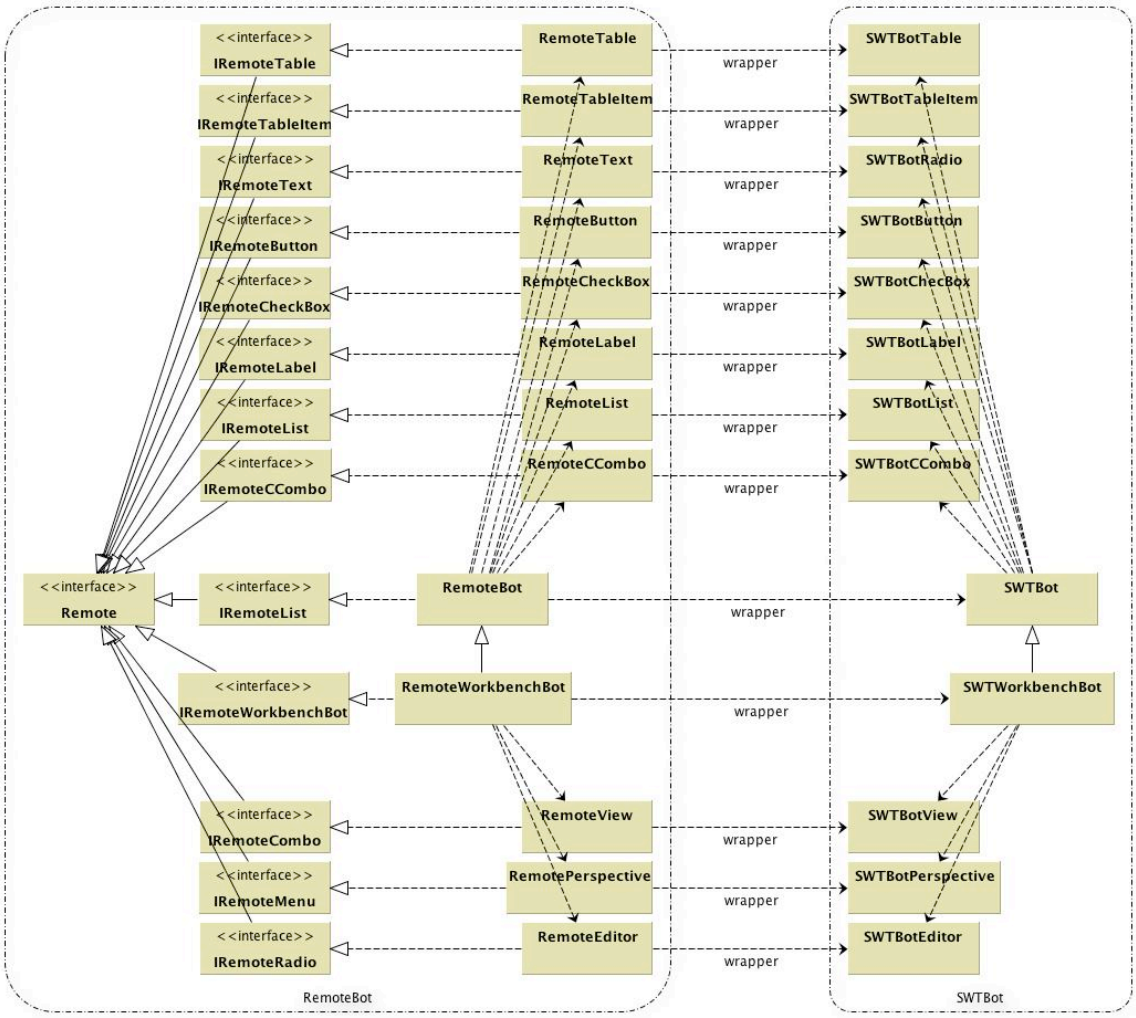
```
//SWTWorkbenchBot is the Subclass of SWTBot
SWTWorkbenchBot bot = new SWTWorkbenchBot();

@Test
public connect() {
    //bot find the view with the name “Saors” and find the toolbar button with the tooltip
    //text “Connect” Ono the view,then click the toolbarbutton.
    bot.viewByTitle(“Saros”).toolbarButton(“Connect”).click();
}
...
```

In this example viewByTitle(“Saros”) returns the object SWTBotView, through which you can call the method toolbarButton(String tooltipText). This method return the Object SWTBotToolbarButton again, with which you may call the method click(); With the method call chain you can find the goal widget and run the action on it. All the returned SWTBot[Widget] objects can not be exported by RMI, because it is platform dependent and therefore can not be serialized.

How to use SWTBot’ APIs remotely?

The Solution is that I've wrapped the whole SWTBot' APIs. E.g. Every wrapper-class is a singleton and would be registered by RMI-Registery.



How to click the toolbar button with the tooltip text “Connect” in the saros view with **RemoteBot**?

```
Tester alice = initTesters(TypeOfTester.ALICE);

@Test
public connect() {
alice.remoteBot.viewByTitle(“Saros”).toolbarButton(“Connect”).click();
}
...
```

By calling the method `viewByTitle` two things would be done:
1. let the real offender SarosSWTBot (Subclass of SWTBot) find the view with the passed view title.

2. pass the object SWBotView returned by calling the method viewByTitle to the class RemoteView, the wrapper-class to SWTBotView.

```
public class RemoteWorkbenchBot extends RemoteBot implements
    IRemoteWorkbenchBot {
    private static transient RemoteWorkbenchBot self;
    private static RemoteBotView view;
    private static SarosSWTBot sarosSwtBot;

    /**
     * {@link RemoteWorkbenchBot} is a singleton, but inheritance is possible.
     */
    public static RemoteWorkbenchBot getInstance() {
        if (self != null)
            return self;
        self = new RemoteWorkbenchBot();
        view = RemoteBotView.getInstance();
        sarosSwtBot = SarosSWTBot.getInstance();
        ...
        return self;
    }

  public IRemoteBotView viewByTitle(String viewTitle) throws RemoteException {
        view.setWidget(sarosSwtBot.viewByTitle(viewTitle));
        return view;
    }
}
...
```

```
 private static transient RemoteBot self;

    private static SWTBot swtBot;
    private static RemoteBotToolbarButton toolbarButton;
    ...

    /**
     * {@link RemoteBot} is a singleton, but inheritance is possible.
     */
    public static RemoteBot getInstance() {
        if (self != null)
            return self;
        self = new RemoteBot();
        swtBot = SarosSWTBot.getInstance();
        toolbarButton = RemoteBotToolbarButton.getInstance();
        ...
        return self;
    }
```

source: [Abb1]

Literaturverzeichnis

[Noe06] Noé, Manfred: Projektbegleitendes Qualitätsmanagement: *Der Weg zu besserem Projekterfolg*. 1. Auflage, Erlangen, 2006

[Stein05] Steinweg, Carl: *Management der Software-Entwicklung: Projektkompass für die Erstellung von leistungsfähigen IT-Systemen*. 6. Auflage, Wiesbaden, 2005

[Bag05] Stephan Bagel: *RMI - Remote Method Invocation*, online Referenz: http://www.sbgl.de/rmi, 2005.

[wiki10] *Remote Method Invitation,* Wikipedia, online Referenz: http://de.wikipedia.org/wiki/Remote_Method_Invocation, 2010.

[Pol00] Martin Pol, Tim Koomen, Andreas Spillner: *Management und Optimierung des Testprozesses: ein praktischer Leitfaden für Testen von Software, mit TPI und TMap*, dpunkt.verlag, 2000, ISBN 3-932588-65-7

[köh10] André Köhler: *Automatisierung von GUI-Tests - Themendossier für die User Group - SoftwareTest und Qualitätssicherung*, SOftwareforen Leipzig . Online Referenz: http://www.versicherungsforen.net/fs/vfl/media/themen/versicherungsforenthemendossiers/2010_2/2010_nr17_sep_it/2010_td_softwareforen_tests.pdf, 2010.

[Lor93] Lorenz, M: *Object-Oriented Software Development – Practical Guide*. Prentice Hall, Englewood Cliffs, 1993.

[Ou87] Ould, Martyn A.; Unwin, Charles: *Testing in Software Development* (British Computer Society Monographs in Informatics). Cambridge University Press, 1987

[Few99] FEWSTER Mark, GRAHAM Dorothy *Software test automation - effective use of test execution tools* (Addison-Wesley-Verlag, London, 1999)

[Coh10] Mike Cohn: *Agile Softwareentwicklung -Mit Scrum zum Reflag!* (Addison-Wesley-Verlag, 2010)

[Bec99] Beck, Kent: *Extreme Programming Explained: Embrace Change*. (Addison-Wesley, 1999) http://www.mip.sdu.dk/~brianj/Extreme%20Programming%20Explained%20-%20Kent%20Beck%3B%20Addison-Wesley,%201999.pdf. – ISBN 0201616416

[Dje06] Djemili, Riad: *Entwicklung einer Eclipse-Erweiterung zur Realisierung und Protokollierung verteilter Paarprogrammierung*, Freie Universität Berlin, Diplomarbeit, 2006.
https://www.inf.fu-berlin.de/w/SE/ThesisDPPI

[Szü10] Sandor Szücs: *Behandlung von Netzwerk- und Sicherheitsaspekten in einem Werkzeug zur verteilten Paarprogrammierung*, Frei Universität Berlin, Diplomarbeit, 2010.
http://www.inf.fu-berlin.de/inst/ag-se/theses/Szuecs10-DPPVI.pdf

[Jur11] Michael Jurke, *Interative, prototypgesteuerte Entwicklung einer Whiteboard-Funktionalität für Saros,* Frei Universität Berlin, Diplomarbeit, 2011. https://www.inf.fu-berlin.de/w/SE/ThesisDPPXVII

[Rous62] Jean-Jacques Rousseau (1712-1778), Schriftsteller und Kulturphilosoph, aus: Emilie, ca. 1762.

[Wes06] Frank Westphal: *Testgetriebene Entwicklung mit JUnit & FIT – Wie Software änderbar bleibt*. dpunkt.Verlag, 2006.

[Vig10] Uwe Vigenschow: *Testen von Software und Embedded Systems.* dpunkt.Verlag, 2010.

[Wien07] *Der Testprozess,* Technische Universität Wien, Online Referenz:
http://qse.ifs.tuwien.ac.at/courses/QS/ws0304/07S_Testprozess_wid_20031116.pdf, 2007

[Sal10] Stephan Salinger; Christopher Oezbek: *Distributed Party Programming*. Online Referenz: http://it-republik.de/jaxenter/artikel/Distributed-Party-Programming-3037.html, 2010

[Cam10] Campus02: *Testlink Tutorial*. Online Referenz: http://itmkb.campus02.at/index.php/Testlink_Tutorial, 2010

[Zel09] Zeljko Markovic: *SWTBot für automatisierte SWT- und RCP-Tests. Online Referenz:* http://it-republik.de/jaxenter/artikel/SWTBot-fuer-automatisierte-SWT--und-RCP-Tests-2275.html, 2009

[Fern10] *JUnit in Eclipse,* FernUni, online Referenz: http://wiki.fernuni-hagen.de/eclipse/index.php/JUnit_in_Eclipse, 2010

[wiki11] Softwaretest, Wikipedia, online Referenz: http://de.wikipedia.org/wiki/Softwaretest, 2011

[Laut03] Siegfried Lautenbacher: *Automatisiertes Testen graphischer Benutzeroberflächen*, online Referenz: http://www.bea-projects.de/files/BlueNotes/BlueNote%20-%20GUI-Tests.pdf, 2003

[Hoh02] Jörg Hoh,: *Refactorings*, Abteilung Programmiersprachen und Compilerbau Fakultät Informatik Universität Stuttgart,

[Pk11] Bruno P. Kinoshit: *Manage automated tests with TestLink and Hudson/Jenkins* http://www.kinoshita.eti.br/2011/02/25/belgium-testing-days-2011-manage-automated-tests-with-testlink-and-hudsonjenkins, 2011.

[Ebert09] Ebert, Ralf: *Eclipse Magazin: Eclipse RCP Anwendungen testen mit SWTBot*. online Referenz: http://www.ralfebert.de/articles/swtbot/, 2009

[Abb1] Alessandro Rei: *Dark Glass*, online Referenz: http://kde-look.org/content/show.php/Dark-Glass+reviewed?content=67902

[Abb2] Florian Thiel: *Page SarosTech*, Frei Universität Berlin, online Referenz: https://www.mi.fu-berlin.de/wiki/bin/viewauth/SE/SarosTech

[Her09] Joachim Herschmann: *Agile Testprozesse – mit großem „A" wie Automatisierung*, Borland (A MICRO FOCUS COMPANY), online Referenz: http://www.sigs-datacom.de/fileadmin/user_upload/zeitschriften/os/2009/Testing/herschmann_OS_testing_09.pdf, 2009.

[Few94] Fewster, Mark and Graham, Dorothy: *Software Test Automation. Effective use of test automation tools*. New York, USA. ACM Press, 1994.

Anhangsverzeichnis

Anhang A: STF-Handbuch