Bachelorarbeit am Institut für Informatik der Freien Universität Berlin,

Arbeitsgruppe Software Engineering

Einführung einer kontinuierlichen Integrationsumgebung und Verbesserung des Test-Frameworks

Stefan Rossbach
Matrikelnummer: 3782359
rossbach@inf.fu-berlin.de

Betreuer: Dr. Karl Beecher
Eingereicht bei: Prof. Dr. Lutz Prechelt

Berlin, 29.08.2011

Zusammenfassung

Diese Arbeit befasst sich mit der Einführung einer kontinuierlichen Integrationsumgebung für das Saros Projekt. Ziel ist es den Entwicklungsprozess von Saros zu automatisieren. Dieses beinhaltet auch eine vollständige Regression über die vorhandenen Testfälle. Dazu wird das Saros Test-Framework benutzt, welches Rahmen dieser Arbeit verbessert wird.

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.

Güstrow, den 26.08.2011

Stefan Rossbach

Inhaltsverzeichnis

1 Einleitung

Diese Bachelorarbeit beschäftigt sich mit der Verbesserung des Saros Entwicklungsprozesses sowie dessen Test-Framework. In Kapitel 1 werden die Funktionalität und Entstehung von Saros erläutert. In Kapitel 2 wird das Konzept des Test-Frameworks, sowie seine Implementierung, vorgestellt. Innerhalb dieses Kapitels werden notwendige Refaktorisierungsmaßnahmen beschrieben, sowie eine Fehleranalyse durchgeführt und Lösungsansätze zur Korrektur beschrieben. Kapitel 3 befasst sich mit der Erstellung einer kontinuierlichen Integrationsumgebung und dem Aufbau von Eclipse Artefakten.

Typografische Konvention

`Maschinenschrift`
für Markierung von Code, Objekten, Methoden und Programmausgaben

Kursivschrift
für Abkürzungen, Fachwörter, nicht sinnvoll übersetzbare Wörter

1.1 Saros

Saros ist ein Eclipse Plugin zur verteilten Paarprogrammierung und wurde vom Studenten Djemili im Rahmen seiner Diplomarbeit [Dje06] entwickelt. Paarprogrammierung ist eine agile Methode des Extreme Programming[Bec04]. Hierbei sitzen zwei Entwickler an einem Rechner, wobei jeweils immer nur ein Entwickler zum gegebenen Zeitpunkt Code schreibt. Dieses ist der *Driver*. Der andere Entwickler ist in dieser Zeit der *Observer*. Er überwacht die vom *Driver* getätigten Änderungen, überprüft diese auf Richtigkeit und korrigiert gegebenenfalls diese durch mündliche Kommunikation mit seinem Partner. Diese Methodik führt zu einer besseren Qualität der Software, sowie zur Wissensweitergabe innerhalb der Entwickler.

Fällt die örtliche Komponente der Paarprogrammierung weg, so spricht man von verteilter Paarprogramming. Dabei sitzt jeder Entwickler an seinem eigenen Rechner und wird durch Werkzeuge unterstützt, die es erlauben, die *Driver* und *Observer* Rollen zu übernehmen.

Wird es nun ermöglicht, die Anzahl von zwei Entwicklern auf beliebig viele zu erhöhen, bezeichnet man dieses als verteilte Partyprogrammierung. Diese Möglichkeit wurde in Saros durch die Arbeiten[Rie08][Gus07] der Studenten Gustavs und Rieger ermöglicht.

1.1.1 Funktionalität

Saros benutzt das *XMPP*[1] Protokoll zum Aufbauen von Sitzungen. Dabei muss sich der Benutzer zu einem *XMPP* Server verbinden. Dem Benutzer ist es nun möglich, nach Kontakten zu suchen. Dazu wird einen Anfrage an den Server gesendet, die entweder direkt beantwortet oder an weitere Server geschickt wird, die mit dem vom Benutzer verbundenen Server in Kontakt stehen. Diese Kontakte werden dann in eine Kontaktliste gespeichert, welche vom Server verwaltet wird.

Saros ermöglicht es innerhalb der Eclipse Anwendung verschiedene Projekte oder Teile von ihnen innerhalb des aktuellen Arbeitsbereiches mit anderen, in der Kontaktliste vorhandenen, Nutzern zu teilen. Dieses ist über mehrere Wege möglich. Mit den ausgewählten Projekten und Benutzern wird dann eine Sitzung aufgebaut. Dabei wird die Einladung sowie deren Bestätigung über den *XMPP* Server abgewickelt. Im Anschluss wird versucht, eine Direktverbindung mit den Teilnehmern zu erstellen. Dabei fungiert der Rechner des Einladenden als Server beziehungsweise Host. Sollte dieses scheitern, so werden die Daten über einen externen *SOCKS5 Proxy* oder dem vom *XMPP* Server zur Verfügung gestellten *In Band Bytestream* gesendet.

Nachdem die einzuladenden Benutzer den Namen des Zielprojektes ausgewählt haben, findet eine Synchronisation mit dem Host statt. Hierbei werden bereits vorhandene Dateien der gewählten Zielprojekte auf der Einladungsseite mit Prüfsummen versehen und diese an den Host geschickt. Dieser erstellt daraus eine Liste der fehlenden Dateien, welche nicht in den Zielprojekten vorhanden sind oder deren Prüfsumme nicht übereinstimmt. Diese Liste wird wiederum zurückgeschickt. Sie dient dazu, den Einzuladenden darauf hinzuweisen, dass eventuell Dateien überschrieben werden. Sollte dieser Vorgang nun bestätigt werden, werden die fehlenden Dateien komprimiert und in ein Archiv geschrieben. Diese Archive, für jeden eingeladenen Benutzer genau ein Archiv, werden dann übertragen und auf der Empfängerseite dekomprimiert. Dieses ist die Synchronisationsphase und erfolgt innerhalb der bereits aufgebauten Sitzung. Im weiteren ist es möglich weitere Benutzer zu einer bereits bestehenden Sitzung einzuladen, ohne dass die vorhandene Sitzung für diesen Zeitraum angehalten werden muss.

Im Anfangszustand einer Sitzung besitzen alle Teilnehmer das Recht, Dateien zu verändern. Saros ermöglicht aber auch klassische Paarprogrammierung. Dabei ist es dem Host möglich, einzelnen Teilnehmern die Schreibrechte zu entziehen. Im weiteren sorgt ein Wachhund dafür, dass durch vorhandene Eclipse Werkzeuge erzeugte Änderungen, trotz Entzug des Schreibrechtes, gefunden und korrigiert werden können. Diese Benutzer können, auch

[1] Extensible Messaging and Presence Protocol, siehe *RFC* 6120–6122, 3922 und 3923

mit Schreibrechten, dann den in Saros eingebauten Folgemodus aktivieren. Dieser zeigt die momentane Position des zu folgenden Teilnehmers an.

Damit die Teilnehmer sich untereinander austauschen können, steht neben dem Chat eine eingebaute Sprachunterstützung, sowie die Möglichkeit ein Teil seines Bildschirmes zu übertragen, zur Verfügung. Dieses ist in der derzeitigen Saros Version, mit Ausnahme des Chats, nur zwischen zwei Teilnehmern möglich.

Die eigentliche Kernfunktionalität, dass gleichzeitige Editieren von Dateien wird vom Jupiter Algorithmus sichergestellt.

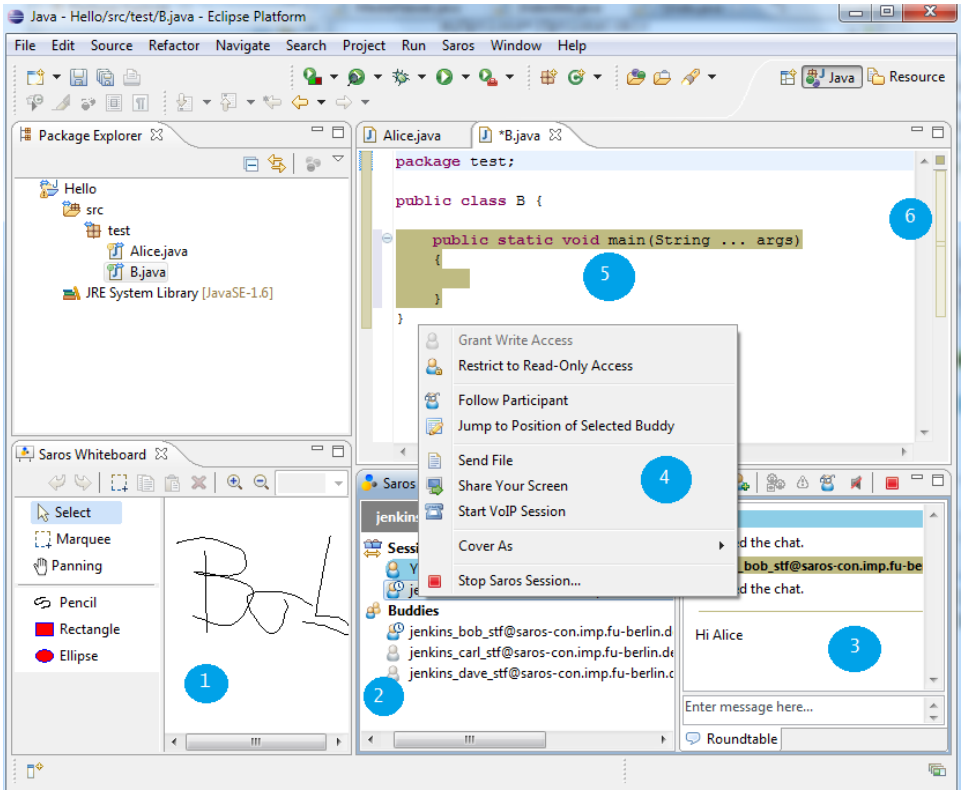

Abbildung 1: Saros Sitzung mit zwei Teilnehmern. Es zeigt das Whiteboard Plugin (1), die Saros View bestehend aus der Kontakliste und den aktuellen Sitzungsteilnehmern (2), sowie dem Chat (3). Beispiel eines Kontextmenü (4) und Teilnehmerinformationen, wie markierter Text (5) und aktueller Sichtbereich (6)

2 Saros Test-Framework

Das Saros Test-Framework, im weiteren Verlauf dieser Arbeit als STF abgekürzt, wurde ursprünglich vom Studenten Szücs[Szu] als Teil seiner Diplomarbeit erstellt und durch die Studentin Chen innerhalb ihrer Diplomarbeit[Che11] weiterentwickelt.

Das Framework ist in Java geschrieben und verwendet ein *Fluent Interface*[2][Fow05] um dem Entwickler einen einfachen Einstieg in das Schreiben von Testfällen zu ermöglichen. Die Testfälle selber sind in Java geschrieben und werden über das JUnit Framework[3] ausgeführt.

Das STF ermöglicht das Erstellen von Testfällen für verschiedene Szenarien der Saros Anwendung. Es ist also möglich mehrere Benutzer, im weiteren als Tester bezeichnet, zu simulieren, die verschiedene Anwendungsfälle automatisiert ausführen können. Dazu muss es einerseits den Testern ermöglicht werden mit der *GUI*[4] von Saros interagieren zu können, andererseits muss eine weitere Komponente diese Tester steuern können.

2.1 SWTBot Framework

SWTBot[5] ist ein, in Java geschriebenes, Framework welches es erlaubt, sogenannte SWT[6] *Widgets*[7] innerhalb einer *SWT* Anwendung, zu der auch die Eclipse *IDE*[8] gehört, zu finden und über einen `SWTBot` zu steuern. Das Framework erlaubt es, im Gegensatz zu klassischen Werkzeugen, welche nur zuvor aufgezeichnete Tastatur- und Mauseingaben ausführen können, sehr präzise über selbstgeschriebenen Java Code die Anwendung zu steuern. Dabei findet die Ausführen dieses Codes nicht im eigentlichen *EDT*[9] der *SWT* Anwendung statt, sondern wird in einem separaten Thread aufgerufen.

Innerhalb einer *GUI* Anwendung ist es unter anderem möglich, dass die Aktualisierung der Elemente nicht sofort nach einer Benutzerinteraktion zur Verfügung steht, da zum Beispiel bestimmte, lang andauernde, Aktionen außerhalb des *EDT* ausgeführt werden müssen. Dieses ist notwendig, da sonst in dieser Zeit die *GUI* nicht mehr auf weitere Eingaben reagieren würde. Hierfür stellt das Framework eine Bedingungsschnittstelle zur Verfügung. Es existieren dabei bereits vorgefertigte Bedingungen innerhalb des SWT-

[2] Entwurfsmuster zur besseren Lesbarkeit des Quellcodes
[3] Java Framework zum Erstellen und Ausführen von Testfällen
[4] Grafische Benutzeroberfläche
[5] `http://www.eclipse.org/swtbot/`
[6] Standard Widget Toolkit
[7] konkretes Anzeigeelement wie zum Beispiel ein Fenster
[8] Integrierte Entwicklungsumgebung
[9] Event Dispatch Thread

Bot Frameworks, jedoch kann man auch eigene Bedingungen nachträglich erstellen. Dazu muss die Bedingung folgende zwei Methoden implementieren, die in der Schnittstelle deklariert sind.

- `String getFailureMessage()`
  Sie gibt eine Fehlermeldung zurück, die beschreibt, weshalb die Bedingung nicht erfüllt werden konnte.
- `boolean test() throws Exception`
  Diese Methode wird von dem Framework in kontinuierlichen Abständen aufgerufen, solange bis die Bedingung erfüllt ist, welche mit dem Rückgabewert `true` signalisiert wird.

Um die Entwicklung zu vereinfachen, benutzt das Framework genau zwei Ausnahmeklassen, welche die Java `RuntimeException` Klasse erweitern und somit nicht explizit in der Methodendeklaration enthalten sein müssen.

Eine `WidgetNotFoundException` wird ausgelöst, wenn das *Widget* nicht gefunden werden konnte. Hierbei ist anzumerken, dass ein *Widget* auch als nicht gefunden gilt, wenn es *disposed* wurde. Das SWT Framework benutzt zum Erstellen seiner *Widgets* die nativen Elemente des Betriebssystems. Im Gegensatz zum AWT[10] Framework, welches die durch das Betriebssystem angeforderten nativen Elemente automatisch wieder zurückgibt, muss dieses in *SWT* manuell durch den Programmierer erfolgen. *Disposed* bezeichnet dabei den Zustand eines *Widgets* nachdem seine Ressourcen wieder an das Betriebssystem zurückgegeben wurden. In diesem Zustand ist jeder Methodenaufruf, welches das Widget zur Verfügung stellt nicht mehr erlaubt und löst eine Ausnahme aus. Die zweite Ausnahme ist die `TimeoutException`. Sie wird ausgelöst, wenn eine Bedingung nicht erfüllt werden konnte.

2.2 Remote Method Invocation

Die Java RMI Technologie[RMIa] ist das Äquivalent zum *RPC*[11]. Sie ermöglicht das Aufrufen von entfernten Methoden. Diese Methodenaufrufe können auf einem entfernten Rechner stattfinden, wobei die Möglichkeit besteht Rückgabewerte an den Aufrufer zu schicken. Dazu muss ein `interface` erstellt werden, welche das `java.rmi.Remote interface` erweitert. Nun kann die eigentliche Klasse erstellt werden, welche dann diese Schnittstelle implementiert. Um nun die Methoden für externe Aufrufer zur Verfügung zu stellen, muss ein Exemplar der Klasse an die *RMI* Registrierung gebunden werden. Dieses erreicht man mit den Aufrufen

[10] Abstract Window Toolkit
[11] Remote Procedure Call

```
Remote remoteProxy = UnicastRemoteObject.
    exportObject(exportedObject, 0);
registry.bind(exportName, remoteProxy);
```

Der erste Aufruf erstellt einen Stellvertreter für unsere Implementation, welcher durch den zweiten Aufruf an die Registrierung gebunden wird und somit nun für die Klienten zur Verfügung steht.

Saros benutzt dieses Verfahren um seine Klassen, welche das *Fluent Interface* bilden, zu exportieren. Nun ist es möglich über diese Schnittstelle auf die Tester zuzugreifen. Dazu wird jeder Tester in einer separaten Eclipse Anwendung gestartet, wobei vor dem Start die Java Eigenschaften `de.fu_berlin.inf.dpp.testmode` und `java.rmi.server.hostname` gesetzt werden müssen. `java.rmi.server.hostname` gibt an, an welcher Netzwerkschnittstelle der Port, welcher durch die Eigenschaft `de.fu_berlin.inf.dpp.testmode` gesetzt wurde, erstellt werden soll. Dieser Port wird dann von der *RMI* Registrierung verwendet.

2.3 Refaktorisierung

2.3.1 Paketnamen

Bei meinem Eintritt in das Saros Team befanden sich die komplette Implementierung des STF, deren dazugehörige Testfälle, sowie weitere JUnit Testfälle, die direkt den Code ohne *GUI* testen konnten, in einem einzigen Unterordner. Dabei war das Framework alleine schon über 4000 Zeilen lang.

Um die Weiterentwicklung zu vereinfachen, sowie die Testfälle besser unterscheiden zu können, wurden drei Quellcode Ordner[12] angelegt auf denen die verschiedenen Java Klassen verteilt wurden. Die JUnit Tests wurden im Ordner `test/junit` abgelegt, die STF Testfälle im Ordner `test/stf` und die Implementierung des STF im Ordner `test/framework/stf/src`. Als nächstes wurden die Paketnamen an die Java Konvention angepasst[13]. Im Gegensatz zur *Camel Case* Schreibweise von Java Klassen ist es für Packetnamen üblich, diese in Kleinbuchstaben zu definieren. Dieses ist von Nöten, da der Paketname im eigentlichen Sinne ein Verzeichnis unterhalb des Klassenpfades beschreibt.

Unter Unix Betriebssystemen ist es daher möglich, die Klassen `a.A.HelloWorld` und `A.a.Helloworld` zu erstellen. Dieses würde unter Betriebssystemen, welche keine Unterscheidung zwischen Groß- und Kleinschreibung innerhalb der Verzeichnisnamen machen, zu nicht behebbaren Fehlern führen. Im

[12] Eclipse Projekte bestehen aus Ordnern die i.d.R auf Verzeichnisse zeigen
[13] http://www.oracle.com/technetwork/java/codeconvtoc-136057.html

übrigen sollte man darauf achten, wenn die Entwicklung auf unterschiedlichen Betriebssystemen stattfindet, die Namensregeln einzuhalten. Das erstellen von Verzeichnissen mit reservierten Namen wie zum Beispiel `NUL`, `CON`, `AUX` sind unter Windows Betriebssystemen nicht erlaubt und deren Löschung ebenfalls nicht. Je nach benutztem Quellcodeverwaltungssystem kann dieses zu unerwarteten Fehlern führen.

2.3.2 STF Klient

Entfernen hart codierter Variablen Ziel dieser Arbeit war es unter anderem das STF in die kontinuierliche Integrationsumgebung einzubinden. Dieses konnte durch den vorhandenen Klienten jedoch nicht ermöglicht werden. Die Konfiguration der Tester, zu der die *JID*, das Passwort, sowie Port und IP[14] Adresse gehörten, waren in einer Java Klasse hart codiert. Dieses hat zur Folge, wenn während einer Regression, die durch den CI Server gerade durchgeführt wird, ein Entwickler eine selbige lokal startet, dass beide Regressionen sich stören würden. Dieses führt zu falschen Ergebnissen, da sich die Tester mehrfach versuchen auf dem *XMPP* Server anzumelden. Dieses hat zur Folge, dass der Server eine vorhanden Verbindung einfach unterbricht. Damit dieses Verhalten ausgeschlossen wird, muss es ermöglicht werden eine externe Konfiguration anzugeben in der die nötigen Informationen hinterlegt sind, mit denen sich der Klient verbinden kann. Dazu erstellte ich ein Java `Enum`, welches die vorhandenen Variablen der vorkonfigurierten Tester ersetzte. Die Felder des Enums werden durch eine Fabrikmethode initialisiert, welche die benötigten Konfigurationsdaten über eine weitere Java Klasse anfordert. Diese Klasse versucht einerseits eine vordefinierte Java Eigenschaftsdatei zu laden, andererseits ist es möglich weitere Dateien mit der Java Eigenschaft `de.fu_berlin.inf.dpp.stf.client.configuration.files` anzugeben. Dabei werden dann bereits eingelesene Werte aus den vorherigen Dateien überschrieben.

Mit Hilfe dieser neuen Implementation ist es möglich, mittels einer Zeile Java Code einen neuen Tester dem Framework bekannt zu machen, welcher beliebig konfiguriert werden kann. Im übrigen muss der Klient nicht mehr neu kompiliert werden, sobald man die Konfigurationsparameter ändern möchte.

Refaktorisierung der abstrakten Testfallklasse Um einer STF Testfall zu erstellen, muss die Klasse, welche diesen implementiert, von der abstrakten Basisklasse `StfTestCase` erben. Zu Beginn meiner Arbeiten enthielt diese viele Utility Methoden, die ich in eine separate Klasse auslagerte. Dasselbe wurde mit allen konstanten Variablen gemacht, welche ich in den

[14] Internet Protokoll

Quellcodeordner `test/stf` verschob. Im weiteren entfernte ich nicht genutzte Methoden oder fügte semantisch ähnliche zu einer Methode mit mehreren Parametern zusammen. Im weiteren dokumentierte ich die alten und neuen Methoden. Dieses ist wichtig, da bei der Rückkehr von diversen Methodenaufrufen bestimmte Nachbedingung noch nicht gegeben sind.

```
public static void buildSessionSequentially(String
   projectName,
       TypeOfCreateProject usingWhichProject,
          AbstractTester inviter,
       AbstractTester... invitees) throws
          RemoteException
```

Bei dieser Methode zum Beispiel, welche eine Sitzung zwischen mehreren Testern aufbaut, ist es nicht gewährleistet, dass nach deren Beendigung die Dateien des Projektes schon bei den eingeladenen Teilnehmern auf das Speichermedium geschrieben worden sind, da eventuell die Daten noch über die Netzwerkschicht übertragen werden. Als Resultat ist die ursprünglich ca. 900 Zeilen lange abstrakte Klasse auf nun mehr ca. 510 Zeilen verkürzt worden. Die neue Utility Klasse umfasst ca. 480 Zeilen inklusive Spezifikation des Verhaltens der einzelnen Methoden.

Verbesserung der Vor- und Nachbedingung Wie bereits in der Arbeit von der Studentin Chen beschrieben, ist es nötig das jeder Testfall in der selben Umgebung ausführt wird, wie sie sein Vorgänger vorgefunden hat. Dazu muss vor und nach jedem Testfall ein fest definierter Zustand erreicht werden. Das JUnit Framework bietet hier in der Version 3 vordefinierte Methoden an, die es während der Ausführung des Tests aufruft. In der JUnit Version 4 ist es außerdem möglich, Methoden mit Annotationen zu markieren, so dass die Namen der Methoden selber gewählt werden können, was zu einer besserern Aussagekraft der einzelnen Methoden führt.

JUnit 3	JUnit 4	Beschreibung
setUpBeforeClass	@BeforeClass	statische Methode, welche beim Laden der Testklasse aufgerufen wird
tearDownBeforeClass	@AfterClass	statische Methode, welche beim Beenden der Testklasse aufgerufen wird
setUp	@Before	Methode, welche vor dem Start eines Testfalls aufgerufen wird
tearDown	@After	Methode, welche nach Beendigung eines Testfalls aufgerufen wird

Ich entfernte zunächst die `@Before` und `@After` Methoden aus der abstrakten Testklasse. Dieses musste in Betracht gezogen werden, da es sonst zu Fehlverhalten in den Testfällen kommen kann, sollte das Verhalten dieser Methoden nachträglich in der abstrakten Basisklasse geändert werden. Danach überarbeite ich die `@AfterClass` Methode. In der `@BeforeClass` Methode wird dafür gesorgt, dass die *Saros View* erscheint und alle am Test beteiligten Tester sich gegenseitig in ihrer Kontaktliste ohne Nickname[15] vorfinden. In der `@AfterClass` Methode müssen alle, durch den Testfall, verursachten Veränderungen beseitigt oder Rückgängig gemacht werden. Dafür ist es wichtig, dass der Tester sich zunächst vom *XMPP* Server trennt und damit die Saros Sitzung unterbricht. Würde dieser Schritt vergessen, so können Seiteneffekte durch Saros verursacht werden, wenn man versucht den Arbeitsbereich zu löschen. Danach wird die Version von Saros auf ihren Ursprung zurückgesetzt und im Anschluss der Arbeitsbereich gelöscht. In der ursprünglichen Implementierung wurde dieses über die *GUI* erledigt. Jedoch zeigten sich nach zahlreichen Tests, dass es immer wieder zu Problemen mit Saros kam. Irgendeine Saros Komponente gibt beim Beenden einer Saros Sitzung seine allokierten Ressourcen nicht sofort frei. Dieses hatte zur Folge, dass die Eclipse *IDE* ein weiteres Fenster mit einer Fehlermeldung öffnete.

Diese Fehlermeldung und der eigentliche Fehler verursachten dann weitere Fehlschläge in den nächsten Testfällen. Um diesen Fehler zu beseitigen, implementierte ich eine Methode innerhalb der STF Server Komponente, welche nicht das *GUI* benutzt, sondern direkt die Eclipse *API*[16] aufruft. Der Vorteil dieser Methodenaufrufe ist der eindeutige Rückgabewert. Er gibt an, ob eine Datei erfolgreich gelöscht werden konnte. Eine weitere positive Eigenschaft ist die Ausführungsgeschwindigkeit. Es ist nun möglich den kompletten Arbeitsbereich, je nach Geschwindigkeit des Speichermediums, in wenigen Sekunden zu löschen.

Um den gefundenen Fehler beim Löschen von Dateien zu korrigieren, wird nun innerhalb von einer Minute mehrmals versucht, den Arbeitsbereich zu löschen. Die Analyse der Ereignisprotokolldatei zeigte, dass meistens nach 20 Sekunden alles erfolgreich gelöscht werden konnte.

Verbesserung der Fehleranalyse Bei einer automatisierten Regression ist es üblich, dass diese ohne Beisein eines Entwicklers abläuft. Hierbei ist es wichtig, dass möglichst viele Daten während der Regression gesammelt und gespeichert werden, um eine gute Analyse beim Auftreten eines Fehlers

[15] In Saros werden Kontakte mit ihrer JID angezeigt, diese kann durch einen Nicknamen ersetzen werden

[16] Application Programming Interface

zu ermöglichen. Dazu muss innerhalb der Ereignisprotokolldatei erkenntlich sein, wo ein Test begonnen und aufgehört hat. Da außerdem die Tests über eine *GUI* stattfinden, ist es wünschenswert ein graphisches Ergebnis zu erhalten, welches zur besseren Analyse beiträgt.

Das JUnit Framework in der Version 4 verfügt über eine `@Rule` Annotation. Diese Annotation, welche an eine Variable angehängt wird, gibt dem Junit Framework einen Hinweis, dass diese ein Exemplar einer Klasse enthält, welche das `interface MethodRule` implementiert. Dieses Exemplar erhält dann von JUnit Informationen über den momentanen Ablauf des Tests. JUnit stellt bereits einige Klassen mit der Implementierung dieser Schnittstelle zur Verfügung.

Die vom STF Klienten genutzte Klasse ist der `TestWatchman`. Sie besitzt die Methoden:

- `void failed(Throwable e, FrameworkMethod method)`
- `void succeeded(FrameworkMethod method)`
- `void finished(FrameworkMethod method)`
- `void starting(FrameworkMethod method)`

Diese Methoden können in der vererbten Klasse überschrieben werden. Um diese Informationen besser umzusetzen, erstellte ich in der STF Server Komponente eine Methode, welche es erlaubt Zeichenketten in die entfernte Ereignisprotokolldatei zu schreiben. Durch Nutzung der oben beschriebenen Methoden ist es nun möglich, den Start und das Ende eines Testfalls in dieser Datei zu erkennen. Außerdem wird nun automatisch ein Screenshot des Bildschirms von jedem Tester, der an dem Testfall beteiligt ist, erstellt, sobald die Methode `failed` aufgerufen wird.

Diese geschaffenen Möglichkeiten können dem Entwickler nun eine einfachere Analyse des Fehlers innerhalb des Testfalls bieten.

2.3.3 STF Server

Implementierung Die STF Server Komponente besteht, neben wenigen Hilfsklassen, nur aus Einzelstücken. Dieses ist notwendig, da jeweils immer nur ein Exemplar einer Klasse an die *RMI* Registrierung gebunden werden kann. Jede dieser Klassen implementiert eine Schnittstelle, welche dem

Klienten zur Verfügung steht, um mit ihr über das *RMI* Protokoll die Methoden der Einzelstücke aufzurufen.

Dabei kann man die Komponente in zwei Unterkomponenten gliedern. Die eine Unterkomponente ist der `RemoteWorkbenchBot`. Sie besteht aus Einzelstücken, welche jeweils ein Exemplar einer Klasse aus dem SWTBot Framework enthält. Die über die Schnittstelle aufgerufenen Methoden werden dann an dieses Exemplar weiter delegiert.

Die zweite Unterkomponente ist der `SuperBot`, welches eine Fassade um die `RemoteWorkbenchBot` Implementierung bildet.

Die *RMI* Technologie liefert immer eine Referenz auf ein Exemplar, welches die Schnittstelle implementiert, zurück, nie das Exemplar selber. Im Gegensatz dazu werden Argumente und Rückgabewerte vor beziehungsweise nach dem Aufruf serialisiert und dann übertragen. Sollten die Objekte nicht serialisierbar sein, so wird eine Ausnahme auf der Seite des Klienten ausgelöst.

Transient Schlüsselwort Laut *RMI* Spezifikation[RMIb] muss keine Variable innerhalb einer Klasse, die die Schnittstelle `Remote` implementiert, mit dem Java Schlüsselwort `transient` deklariert werden, da nur eine Referenz auf das Exemplar zurückgegeben wird. Das `transient` Attribut bewirkt, dass diese Variable vom Serialisierungsprotokoll ignoriert wird und bei der Deserialisierung eigenständig neu erzeugt werden muss. In jeder Einzelstückimplementation war die Referenz auf das Exemplar der Klasse als `transient` deklariert. Dieses ist unnötig und zudem auch falsch, da jede dieser Klassen ihre Methodenaufrufe an ein `SWTBotWidget` delegiert, welches in der Tat nicht serialisierbar ist. Um weitere Missverständnisse in Hinsicht auf die Weiterentwicklung der STF Server Komponenten zu vermeiden, entfernte ich jedes `transient` Attribut.

Superbot Das SWTBot Framework besteht nicht nur aus einem einzigen *Bot*. Für bestimmte *SWT* Widgets ist es möglich einen *Bot* zu erstellen, der nur auf diesem *Widget* arbeitet. Dieses ist zum Beispiel für ein *SWT* `Shell` *Widget*[17] vorgesehen. Arbeitet man mit dem `SWTBot`, so ist folgender Aufruf möglich:

```
SWTBot bot = new SWTBot().shell("HelloWorld").bot();
bot.label("HelloWorldLabel").setText("HelloWorld");
```

Da die STF Server Komponente auf Einzelstücken basiert, analysierte ich die dafür zugehörige Implementation.

[17] Eine Shell stellt ein Fenster innerhalb des SWT Frameworks da

```
public IRemoteBot bot() {
RemoteBot botImp = RemoteBot.getInstance();
// botImp.setBot(swtBotShell.bot());
botImp.setBot(SarosSWTBot.getInstance());
return botImp;
}
```

Es ist zu sehen, dass hier nicht wie gefordert ein neuer *Bot* für die `Shell` benutzt wird, sondern ein bereits existierendes Exemplar verwendet wird. Die entsprechende Korrektheit hätte eine Analyse des Quellcodes vom SWTBot Framework erfordert. Ich entschloss mich daher, das Verhalten korrekt zu implementieren. Dazu musste jedoch der `Superbot` umgeschrieben werden, da dieser intern den `RemoteWorkbenchBot` benutzt. Damit die geforderte Funktionalität unterstützt werden konnte, mussten Änderungen am `Superbot` gemacht werden. Dazu wurden alle Methodenaufrufe an den `RemoteWorkbenchBot` durch Aufrufe an das SWTBot Framework ersetzt. Da bereits im Vorfeld eine Testsuite mit 90 Selbsttests für das STF vorhanden war, erleichterte dieses das Überarbeiten. Dabei wurden unter anderem auch Fehler in der Implementierung gefunden, welche korrigiert werden konnten.

Nach Abschluss der Arbeiten konnte dann die korrekte Umsetzung in den `RemoteWorkbenchBot` integriert werden. Im weiteren wurde innerhalb der `RemoteWorkbenchBot` Klasse die Methode `resetBot` eingeführt. Diese Methode wird automatisch von dem Klienten aufgerufen und setzt vor dem Methodenaufruf `remoteBot()` den SWTBot zurück. Es muss dabei jedoch beachtet werden, dass Aufgrund der Architektur des STF bestimmte Aufrufe zu Fehlern führen.

```
IRemoteBotShell a = ALICE.remoteBot().shell("Shell1")
IRemoteBotShell b = ALICE.remoteBot().shell("Shell2");

List<String> treeItems = a.tree("ProjectListing").
    getAllItems();
```

Durch die interne Implementierung wird beim zweiten Aufruf des `RemoteWorkbenchBot` der zuvor gespeicherte `SWTBot` für die `Shell a` durch den `SWTBot` für die `Shell b` ersetzt. Alle weiteren Aufrufe von Methoden der Variablen `a` verweisen nun auf `b`. Eine Lösung hätte die Änderung der Architektur benötigt, deshalb wurde dieses Verhalten in der STF Anleitung festgehalten.

2.4 Abstürze

Die Tester können auf dem lokalen Rechner über sogenannte *Eclipse Launch Configuration* gestartet werden. Dabei können unter anderem Argumente an

die virtuelle Java Maschine übergeben werden. Dieses betrifft neben diversen Java Eigenschaften auch Einstellung für die virtuelle Java Maschine.

Während einer Regression über ca. 100 Testfälle kam es dabei regelmäßig zu einer `OutOfMemoryException` innerhalb der permanenten Generation[18] der JVM[19], welche die Eclipse Anwendung in einem nicht definierbaren Zustand hinterließ und die Regression zum Abstürzen brachte. Mit dem Diagnosewerkzeug `jconsole`, welches in der Java Oracle Distribution enthalten ist, kann man während der Ausführung einer Java Applikation diverse Statusinformationen der JVM abfragen. Ein empirischer Test ergab, dass der Wert für den reservierten Speicher, die der permanenten Generation zur Verfügung stand, zu gering gewählt wurde. Dieser war ursprünglich auf 128 MiBytes eingestellt. Der Test zeigte jedoch, dass mindestens 160 MiBytes benötigt wurden. Nach entsprechender Korrektur des Wertes konnte dieser Fehler beseitigt werden.

Ein anderer Fehler betraf das Einfrieren der Eclipse *GUI*. Dieser Fehler trat in sehr seltenen Abständen auf und war nicht manuell reproduzierbar. Da das Diagnosewerk `jconsole` es auch erlaubt sich die Stacktraces der einzelnen Threads innerhalb einer Java Anwendung anzuschauen, konnte ich den Fehler eingrenzen. Das Problem war, dass der *EDT* versuchte eine Nachricht in die Ereignisprotokolldatei zu schreiben, jedoch bereits ein anderer Thread zu diesem Zeitpunkt die `Logger` Klasse benutze. Der Thread der diesen Aufruf blockierte war der *RMI* Thread des STF Servers. Eine genaue Analyse ergab, dass dieser Thread durch den *EDT* Thread blockiert war und somit zu einer Verklemmung führte. Die den Fehler verursachende Klasse befand sich innerhalb des SWTBot Frameworks. Um bessere Leistung zu erzielen, erzeugt diese Klasse die zu protokollierende Zeichenkette erst während des Protokollierungsvorgangs. Dazu wird in dem Log4J[20] Framework eine überladene Methode benutzt, der anstatt einer Zeichenkette ein Exemplar einer Klasse übergeben wird. Dadurch wird intern ein Schloss angefordert und im Anschluss wird die `toString` Methode des Exemplars aufgerufen, welche versucht über den *EDT* Daten eines *Widgets* auszulesen. Dieses erzeugte das oben beschriebene Verhalten. Als momentan einzige Lösung musste die Protokollierungsstufe für das SWTBot Framework auf `WARNING` herabgestuft werden. Dieses hat zur Auswirkung das Informationen fehlen, wenn das STF sich selbst testet und dabei Fehler auftreten.

[18] Bereich zum Speichern der Metainformation von Java Klassen
[19] Java Virtuell Machine
[20] http://logging.apache.org/log4j/

2.5 Nichtdeterministisches Verhalten

Das schwerwiegendste Problem des STF ist es, dass es regelmäßig für ein und den selben Test verschiedene Ergebnisse liefern kann. Solch ein Verhalten ist für ein Test-Framework nicht akzeptabel, da sich die Entwickler auf kurze oder lange Sicht weigern würden, dieses zu benutzen, da der Aufwand für das Schreiben eines Testfalls in keiner Relation zu dem gelieferten Mehrwert steht.

Der Studentin Chen war dieses Verhalten bereits bekannt. Sie kam, wie ich, zu dem Entschluss, dass es viele Faktoren gibt die Fehlschläge einzelner Tests auslösen können, weil sie nicht berücksichtigt wurden. Hierbei stehen zum Beispiel Veränderungen an der *GUI* selbst. Diese haben den größten Einfluss auf die Tests. Dabei muss durch eine Änderung an der *GUI* es nicht unbedingt gewährleistet sein, dass die Testfälle, welche diese Änderung betreffen, sofort fehlschlagen. So kann es möglich sein, dass nur unter gewissen Umständen zum Beispiel ein neues Fenster erscheint und dieses vom Testfall oder der STF Implementierung nicht berücksichtigt wurde und dadurch fehlschlägt.

Da es momentan nicht möglich ist die Ablaufverfolgungsstufe des SWTBot Frameworks zu ändern, ohne Abstürze in Kauf zu nehmen, mussten nachträglich zusätzliche Protokollierungsanweisungen in das STF an markanten Stellen, die Probleme zu versursachen schienen, eingefügt werden.

Um die Fehler besser zu analysieren teilte ich sie in mehrere Kategorien ein.

Fehlschlag des Testfalls	mögliche Fehlerquellen (nach priorität)
nur während der ersten Ausführung	Zeitverhalten, Defekt im STF, Defekt in Saros
nur nach Ausführung eines anderen Tests innerhalb der Testklasse	Testfälle falsch geschrieben, Defekt im STF, defekt in Saros
nach der Ausführung eines anderen Tests	Defekt im STF, defekt in Saros
zufällig	Defekt im STF, Defekt in Saros, Zeitverhalten

Um Probleme, welches das Zeitverhalten innerhalb einer *JVM* und der Synchronisation des *SWT EDT* betrifft, besser eingrenzen zu können, ist es notwendig deren Verhalten zu analysieren.

Java Bytecode Im Gegensatz zu Programmen die bereits in Maschinencode vorliegen, etwa in Assembler oder C/C++ geschriebene Programme,

wird der Quellcode einer Java Klasse durch den Kompilierer in einen Zwischencode, dem sogenannte Java Bytecode übersetzt. Dieser Bytecode wird zusammen mit Metainformation, welche die Klasse betreffen, in eine `class` Datei geschrieben. Dabei es ist es auch möglich, mehrere `class` Dateien in einem Jar Archiv zu speichern.

Dynamisches Laden von Klassen Da der Bytecode nicht direkt durch die *CPU* ausführbar ist, muss er über eine *JVM* interpretiert werden. Da vor der Ausführung des Java Programms der *JVM* nicht bekannt ist, welche Klassen sie benötigt, lädt die *JVM* per Spezifikation diese Klassen erst, wenn sie das erste Mal im Programmfluss referenziert werden.

Damit die *JVM* die entsprechenden Klassen finden kann, muss ihr vor dem Start mitgeteilt werden, wo sie diese Klassen suchen soll. Dazu wird ihr ein `Classpath` übergeben. In diesem kann man mehrere Verzeichnisse, sowie Jar Archive angeben, die durch die *JVM* durchsucht werden sollen. Hierbei ist zu beachten, dass die *JVM* diesen Klassenpfad in der angegebenen Reihenfolge durchsucht und immer die zuerst gefundene Klasse lädt.

Ein Nachteil dieser Methode ist, dass sehr viele Zugriffe auf den Speichermedien stattfinden, in welchen die benötigten `class` Dateien oder Jar Archive liegen. Dabei können die Zeiten zum Laden der Dateien, je nach Geschwindigkeit des Mediums und Pufferstrategie des Betriebssystem, schwanken.

JIT Kompilierer Moderne *JVM* verfügen über sogenannte Just in Time Kompilierer[Wik]. Ihre Aufgabe ist es zur Laufzeit aus Java Bytecode nativen Maschinencode für die unterliegende Architektur zu erstellen und diesen dann auszutauschen. Durch ihre Einführung ist es überhaupt erst möglich die Geschwindigkeitseinbußen, welche durch das Interpretieren des Java Bytecodes entstehen zu eliminieren. Da die *JIT* Kompilierer Zugriff auf Informationen haben, die den statischen Kompilierern nicht zur Verfügung stehen, ist es ihnen zum Teil möglich, deutlich besseren Maschinencode zu erstellen.

Innerhalb der JVM ist es möglich sich diesen Vorgang über die Option `-XX:+PrintCompilation` anzeigen zu lassen. Die meisten *JIT* Kompilierer sind sogenannte Hotspot Kompilierer, welche die Anzahl der Methodenaufrufe mitverfolgen. Steigt die Anzahl über einen bestimmten Grenzwert, so wird mit der Umsetzung des Bytecodes in Maschinencode begonnen.

Es ist jedoch dabei zu beachten, dass dieser Vorgang Rechenzeit benötigt und somit die Programmausführung unterbrechen oder sie in ihre Geschwindigkeit während Kompilierungsphase reduzieren kann. Dieses hängt davon

ab, ob die Kompilierung nebenläufig ausgeführt wird und ob genügend Rechenzeit durch die *CPU* zu Verfügung steht.

Automatische Speicherbereinigung Die *JVM* benutzt zum Verwalten ihres benötigten Speicher automatische Speicherbereinigung. Diese ist dafür zuständig, dass der genutzte Speicher von Variablen beziehungsweise die Datenstrukturen auf welche sie verweisen, automatisch und ohne Zutun des Programmierers, wieder freigegeben wird. Für die automatische Speicherbereinigung stehen verschiedene Algorithmen zu Verfügung. Die Funktionsweise soll Anhand der Sun Hotspot JVM[Sun06] erklärt werden.

Die *Hotspot JVM* besitzt zwei Speicherbereiche, eine junge Generation und eine alte Generation. Auf jeder Generation arbeitet dabei ein anderer Speicherbereinigungsalgorithmus. In der jungen Generation wird versucht sämtliche neu erzeugte, in der Regel kurzlebige, Objekte zu erstellen. Ist das Objekt zu groß für diese Generation, so wird das Objekt sofort in der alten Generation erstellt. Ist bei einer Neuerstellung eines Objektes nicht mehr genügend Speicherplatz vorhanden, so wird eine Speicherbereinigung auf dieser Generation durchgeführt. Die junge Generation ist dazu in drei Speicherbereiche unterteilt. Dem *eden* Bereich in dem die neuen Objekte angelegt werden und zwei *Survior Spaces*, *from* und *to* . Bei einer Speicherbereinigung werden dann alle überlebenden Objekte aus dem *eden* und *from* Speicherbereich, die noch referenziert werden können, in den Speicherbereich *to* kopiert. Dabei werden alle Objekte die nicht mehr in diesen Bereich passen in die alte Generation verschoben. Im Anschluss werden die Bereiche *from* und *to* vertauscht. Überlebt ein Objekt diesen Vorgang nun in einer zuvor festgelegten Anzahl, so wird dieses Objekt ebenfalls in die alte Generation verschoben.

In der alten Generation befinden sich, je nach Erfolg der in der jungen Generation durchgeführten Speicherbereinigung, überwiegend sehr langlebige Objekte. Sollte nun auch diese Generation überlaufen, so löst die JVM eine *Full Garbage Collection* aus. Dieser Vorgang bewirkt, dass alle laufenden Java Threads innerhalb der JVM angehalten werden und die gesamte Anwendung für diesen Zeitraum nicht reagiert. Die Länge dieser Operation bestimmt sich aus der Anzahl der vorhanden Objekte und dem verwendeten Speicherbereinigungsalgorithmus[21]. Es ist anzumerken, dass Java Anwendungen die Speicherlecks erzeugen über eine lange Programmausführung hinweg mehr und immer mehr *Full Garbage Collections* erzeugen. Dieses zeichnet sich durch eine sehr hohe Ausnutzung der verfügbaren Rechenressourcen ab, wobei die Anwendung jedoch selber ihre Eingaben nur noch sehr langsam verarbeitet oder kaum noch reagiert.

[21] Die Sun Hotspot JVM bietet die Möglichkeit parallelle Speicherbereinigungalgorithmen zu benutzen

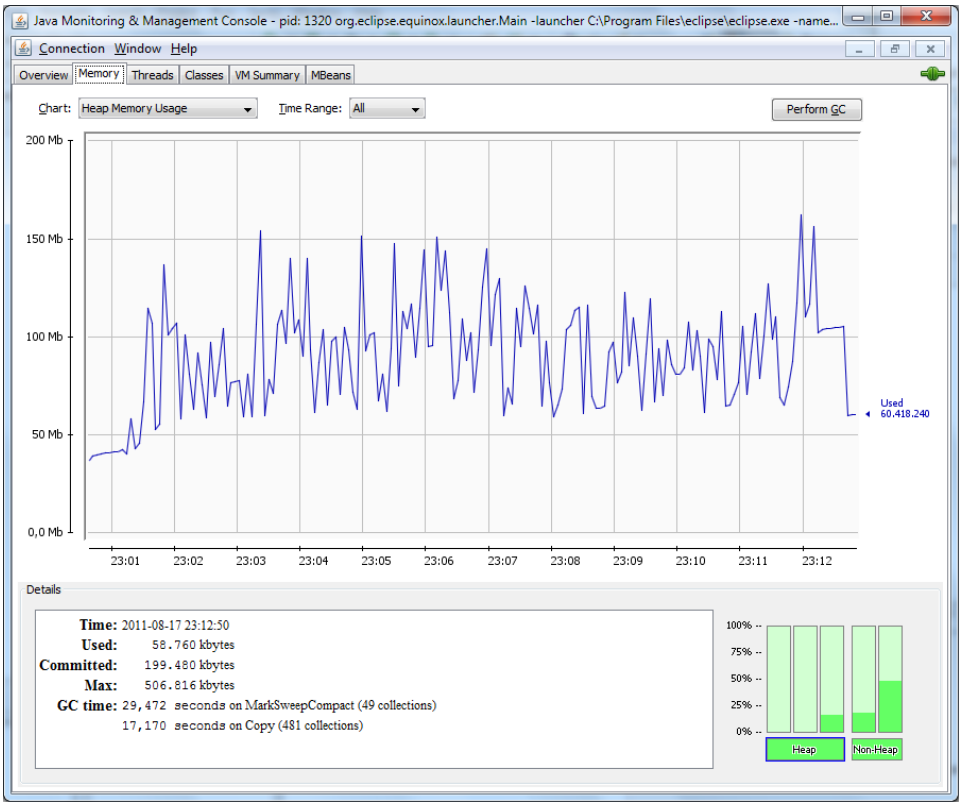

Abbildung 2: Vebrauchte Zeit durch die Garbage Collectors während einer STF Regression

Virtuelles Speichermanagement Jedes moderne Betriebssystem bietet die Möglichkeit virtuellen Arbeitsspeicher zu benutzen. Bei diesem Verfahren wird der physikalisch verfügbare Arbeitsspeicher in sogenannte logische Seiten unterteilt. Fordert nun eine Anwendung Speicher an, der physikalisch nicht mehr zur Verfügung steht, so kann das Betriebssystem Teile des Arbeitsspeichers, welche durch andere Anwendungen belegt sind, auf ein Speichermedium verschieben und diesen dann für andere Anwendungen bereitstellen. Dabei wird das Betriebssystem durch die *CPU* unterstützt, die entsprechende Ausnahmen auslöst, sollte sich die logische Seite momentan nicht im Arbeitsspeicher befinden.

Wie bei der automatischen Speicherbereinigung beschrieben, ist es möglich, dass die *JVM* ihren ganzen genutzten Arbeitsspeicher durchsucht. Dieses kann zu einer hohen Aktivität auf dem Speichermedium führen, sollten viele Seiten ausgelagert sein. Je nach Geschwindigkeit des Mediums kann der Pro-

grammablauf dadurch für einen bestimmten Zeitraum angehalten werden.

2.5.1 Event Dispatching Thread

Das SWT Framework, sowie auch das SWING Framework basieren auf einer ereignisgesteuerten Architektur. Dabei werden die Ereignisse von dem *EDT* ausgeführt. Innerhalb von *SWT* ist dieser der Java Main Thread. Alle Operationen, welche Veränderung an der *GUI* vornehmen möchten, müssen innerhalb des *EDT* ausgeführt werden. Um dieses auch anderen Threads zu ermöglichen, stellt das SWT Framework die Methoden `void asyncExec(Runnable runnable)` und `void syncExec(Runnable runnable)` zur Verfügung.

Die Methode `asyncExec` bewirkt, dass das übergebende `Runnable` in die Eventschlange eingefügt wird und dann zurückkehrt. Es wird dabei keine Garantie gegeben, wann die zeitliche Ausführung des `Runnable` erfolgt. Möchte man sicherstellen, dass die Ausführung stattgefunden hat, so muss die Methode `syncExec` benutzt werden. Diese Methode wartet solange bis die Ausführung erfolgt ist und kehrt dann zum Aufrufer zurück. Es ist im weiteren auch möglich innerhalb des *EDT* die Methode `asyncExec` zu benutzen.

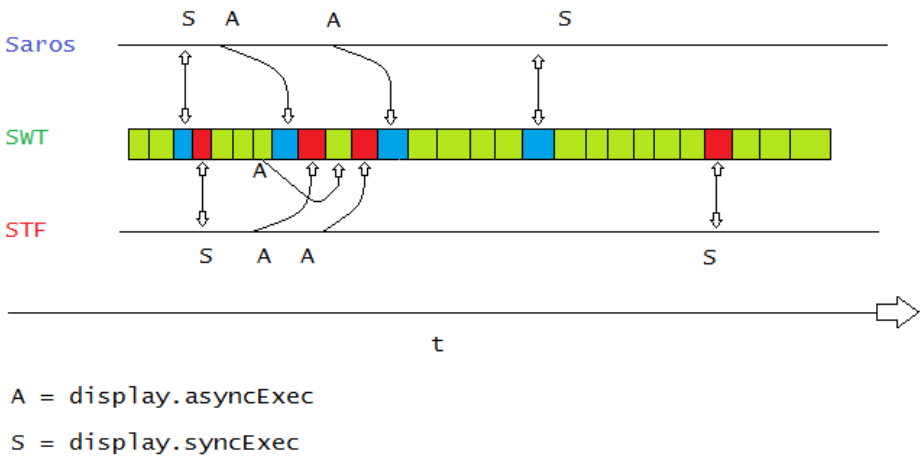

Abbildung 3: Zugriff von Saros und dem STF auf den SWT EDT

2.6 Fehlerauswertung

2.6.1 Falsche Wartewerte

Viele Tests, die Datei Operationen nutzten, schlugen deshalb Fehl, weil bestimmte Operationen, wie zum Beispiel das Bewegen oder Umbenennen einer Datei, unterschiedlich viel Zeit kosten können. Dieses Problem konnte man optisch analysieren während die Testfälle ausgeführt wurden. Zur Lösung dieses Problems wurde der Zeitüberschreitungswert dieser Operationen auf eine Minute angehoben. Dieser Wert wurde durch empirische Versuche auf einem Rechner mit normaler Hardware Ausstattung[22] ermittelt.

2.6.2 Zwischenspeichern von Widgets

Innerhalb der einzelnen Methodenaufrufe, die das *Fluent Interface* bereitstellt, kommt es an diversen Stellen vor, dass *Widgets* über den `SWTBot` angefordert und in einer Variable zwischengespeichert werden. Dieses stellte auf den ersten Hinblick auch für mich kein Problem dar. Bei zahlreichen Regressionen über das STF selbst kam es jedoch immer wieder zu *Widget is disposed* Ausnahmen. Dieses betraf allerdings nur das JFace[23] `Treeviewer` *Widget* aus der Saros *View*. Da innerhalb dieses *Widgets* die Visualisierung der Kontaktliste und aktuellen Saros Sitzung implementiert ist, werden über das STF dort zahlreiche Aktionen ausgeführt. Auch nach mehrfacher Analyse der Ereignisprotokolldateien konnte der Fehler nicht gefunden werden. Deshalb programmierte ich einen Test, in dem über den SWTBot versucht wurde, bestimmte Blätter, welches selber wieder *Widgets* sind, aus dem JFace `Treeviewer` *Widget* zu finden. Dabei stellte sich heraus, dass sobald die Methode `setInput` auf dem `Treeviewer` *Widget* aufgerufen wird, alle Kind Elemente des Baumes *disposed* werden. Dieses geschieht in der Saros *View* beim initiieren einer Saros Sitzung viele Male. Ich korrigierte den entsprechenden Codeabschnitt, so dass dieser Aufruf nur noch einmal ausgeführt werden muss. Zudem wurde die Zwischenspeicherung dieser *Widgets* aus den Superbot Klassen entfernt. Es wird nun bei jedem Zugriff auf ein `TreeItem`, dieses jedesmal neu gesucht.

Es bleibt anzumerken, dass dieses den Fehler nicht komplett ausschließt, sondern dessen Auftreten nur verringert. Eine korrekte Implementierung konnte nicht erreicht werden, da bei manchen Aktionen auf deren Bearbeitung durch Saros gewartet werden muss. Es ist deshalb nicht möglich, die gesamte Logik auf einmal innerhalb des *EDT* auszuführen. Der Fehler kann jedoch Umgangen werden, wenn man es unterlässt, während des Synchronisationsprozesses irgendwelche Aktionen auf der Saros *View* durchzuführen.

[22] AMD Athlon 64 X2 Dual Core Prozessor 5200 @ 2.6 GHz, 3 GB DDR 3 RAM, 7200 RPM UDMA 5 HDD

[23] Framework das eine Abstrakstionsebene zu SWT Widgets bereitstellt

2.6.3 Tastatur

Um die Hauptfunktionalität von Saros, das gleichzeitige Editieren, zu testen, muss es möglich sein Tastatureingaben zu simulieren. Dieses wird im SWTBot Framework durch Keyboardstrategien ermöglicht. SWTBot unterstützt dabei das Erzeugen von Eingaben mittels *AWT* oder *SWT* und einer eigenen `MockKeyboard` Klasse. Die ersten beiden Strategien benutzen native Methoden, welche die Tastendrücke in die Eventwarteschlange[Zie09] des Betriebssystems einfügen, wo sie dann ausgewertet werden. Damit der `SWTBot` weiß, welche Zeichen über welche Tastatureingaben erzeugt werden, müssen diese zudem in einer Datei vermerkt werden. Im Gegensatz dazu sendet die `MockKeyboard` Klasse die erzeugten Zeichen direkt an ein *Widget*, welches aber nicht für alle Arten von *Widgets* möglich ist.

In der ursprünglichen Version wurde die *AWT* Keyboardstrategie verwendet, in dessen Konfigurationdatei kaum Zeichendefinitionen standen oder sogar falsch waren. Als Resultat konnte man außer dem Alphabet in Groß- und Kleinschreibung, sowie den Zahlen 0 bis 9 keine anderen Zeichen eingeben.

Ein wesentlich schwerwiegenderes Problem war jedoch, dass es nicht möglich war gleichzeitig mehrere Texteingaben von unterschiedlichen Testern machen zu lassen. Dieses trat immer dann auf, wenn man mehrere Eclipse Anwendungen auf dem selben Rechner startete. Dieses ist damit begründet, dass durch das Senden von Eingaben an das Betriebssystem, dieses die Eingaben an die Anwendung schickt, welche den derzeitigen Fokus hat.

Da es dem Entwickler ermöglicht werden soll, dass STF mit nur einem Rechner zu benutzen, ersetzte ich die *AWT* Strategie durch die *Mock* Strategie. Diese hat allerdings den Nachteil, dass sie nicht mit allen *Widgets*, zum Beispiel dem *Widget* `Text`, funktioniert und bestimmte Tasten, wie zum Beispiel die Pfeiltasten, nicht unterstützt. Der große Vorteil ist jedoch, dass es durch diese Strategie ermöglicht wird, paralleles Schreiben innerhalb verschiedener Eclipse Anwendungen gleichzeitig auszuführen. Die nicht zur Verfügung stehenden Tasten können dabei durch *Workarounds* umgangen werden. So ist es zum Beispiel möglich, das *Scrollen* durch eine Datei im Eclipse Editor zu simulieren, in dem man zur gewünschten Zeile springt und dort einen Buchstaben eingibt.

Die Konfigurationdatei für die Zeichen hat den Namem `default.keyboard` und ist im Java Paket `de.fu_berlin.inf.dpp.stf.server.bot` abgespeichert. Damit der `SWTBot` diese Strategie benutzt müssen vor dem Start zwei Java Eigenschaften gesetzt werden. Dieses ist zu einem die Eigenschaft `org.eclipse.swtbot.keyboard.strategy`, welche mit dem Wert

`org.eclipse.swtbot.swt.finder.keyboard.MockKeyboardStrategy` initialisiert wird. Die andere Eigenschaft ist `org.eclipse.swtbot.keyboard.layout` und wird mit dem Wert `de.fu_berlin.inf.dpp.stf.server.bot.default` initialisiert.

2.6.4 Saros

Viele Fehler verursachte Saros jedoch zum Teil selbst. Dieses war immer dann gegeben, wenn durch irgendeinen internen Fehler die *GUI* Elemente nicht aktualisiert oder Fenster geöffnet wurden, die nicht durch das SWTBot Framework geschlossen werden konnten. Ein Beispiel dafür war das vorzeitige Beenden des Einladungsdialoges. Brach man diesen ab, öffnete sich ein neues Fenster was um eine Bestätigung fragte. Schließt man dieses Fenster ohne dabei auf den OK Knopf zu drücken, befindet man sich wieder im Einladungsdialog. Da das STF versucht alle Fenster am Ende einer Testfallklasse zu schließen, musste für diesen und andere Fälle neue Logik implementiert werden. Sie soll sicherstellen, dass diese Dialoge korrekt geschlossen werden.

Ein anderes Problem stellt der Synchronisationsmonitor dar, der Information über den Einladungsprozess gibt. Dieser ist ebenfalls nicht automatisch schließbar und kann, aus nicht geklärten Umständen, manchmal nicht sofort geschlossen werden, so dass nachfolgende Testfälle fehlschlagen. Der seltenste zu reproduzierende Fehler betraf den Folgemodus. In ungefähr jeder 20. Regression kam es zu einem Fehler. Dieses Betraf die Kontextmenüs. Dabei kam es vor, dass das Kontextmenü der Saros *View* falsche Einträge hatte. Eine Analyse ergab, dass der Folgemodus in der Lage ist den Fokus von der Saros *View* auf den Editor zu verlagern.

Betrachtet man zum Beispiel die Implementierung um den Folgemodus zu aktiveren,

```
treeItem.select();
ContextMenuHelper.clickContextMenu(tree,
    CM_START_FOLLOWING);
```

so ist dieses keine atomare Operation sondern zwei getrennte Operationen, wobei die Implementierung der letzten Methode wieder aus mehreren nicht atomaren Operationen besteht. Alle diese Operationen greifen auf den *EDT* zu. Saros ist in der Lage, zwischen diesen Operationen den Fokus zu wechseln und verursacht damit, dass das Kontextmenü nicht erscheint oder einen falschen Inhalt hat. Dieses Verhalten ist innerhalb der Superbot Komponente korrigierbar, jedoch nicht in RemoteWorkbenchbot Komponente, da diese nicht die notwendigen Methoden für die Synchronisation mit dem *EDT* bereitstellen kann.

2.7 Offene Probleme

Die Analysen haben gezeigt, dass gerade Zeitprobleme auf leistungsschwachen Rechnern in Betracht gezogen werden müssen. Dazu bietet das SWTBot Framework Java Eigenschaften[24] an, mit denen die voreingestellten Wartewerte geändert werden können. Diese Werte müssen dann vom Entwickler empirisch ermittelt werden.

Ein weiteres Problem tritt auf, wenn zu viele Tester auf einem Rechner gestartet werden. Der Speicherverbrauch der Eclipse Anwendung beträgt derzeit bis zu 700 MiBytes pro Tester. Es ist möglich den Speicherverbrauch zu ändern, jedoch kann dieses wiederum zu erhöhter Anzahl von *Garbage Collections* führen.

Ein weiteres Problem stellt die Latenz da. Es ist zum Teil einfacher innerhalb eines Testfalls kurz zu warten, anstelle bestimmte Ereignisse konstant abzufragen, welche unter Umständen zusätzlichen Code innerhalb des Testfalls erfordern, wenn das STF diese Möglichkeit für das Problem nicht bietet. Da sowohl das lokale Testen, sowie die kontinuierliche Integrationsumgebung, eine Latenz von weniger als 10 Millisekunden zu den anderen Eclipse Anwendungen haben, ist diese Lösung momentan Aufgrund ihrer einfachen Implementierung vorzuziehen. Sie hat jedoch den Nachteil, dass der Zeitraum nicht zu kurz gewählt werden darf. Ein anderer Nachteil ist, dass dadurch der Testfall unter Umständen unnötig Zeit verbraucht.

Die Testumgebung innerhalb der kontinuierlichen Integration bietet zwar eine Firewall, die solch ein Verhalten simulieren kann, jedoch wurde bereits in der Diplomarbeit[Gus11] vom Studenten Gustavs gezeigt, dass durch das benutzte TCP[25] Protokoll keine Auffälligkeiten zu erkennen waren. Um alle Fehler, die mit hohen Latenzen zu tun haben könnten, auszugrenzen, wäre es empfehlenswert dieses Verhalten einerseits mit JUnit Tests direkt in der Netzwerkschicht zu testen, oder eine separate Testsuite für das STF mit Tests zu erstellen. Für die Regression über die Testsuite müsste die Firewall dann mit extrem Werten, wie zum Beispiel einer Latenz von 10 Sekunden oder einem Paketverlust von 90 Prozent vorkonfiguriert werden. Dazu muss aber das Verhalten von den einzelnen Netzwerkkomponenten innerhalb dieser Schicht spezifiziert werden, damit dieses Verhalten korrekt getestet werden kann.

[24] http://wiki.eclipse.org/SWTBot/FAQ
[25] Transmission Control Protocol

3 Kontinuierliche Integration

3.1 Begriffserklärung

Der Begriff kontinuierliche Integration[Fow] entstand mit dem Aufkommen des *Extreme Programmings*. Er bezeichnet eine Methodik, die es erlaubt, Software nach festgelegten Kriterien automatisiert zu produzieren. Hierzu wird das Gesamtsystem wenn möglich in mehrere Teilkomponenten unterteilt. Die einzelnen Teilkomponenten werden nun unabhängig voneinander gebaut und gegebenenfalls mittels automatischen Tests getestet. Im Anschluss können diese dann in weiteren Integrationstests mit anderen Komponenten getestet werden oder mit ihnen weitere Komponenten erzeugt werden.

Dazu ist es von den Entwicklern[DMG07] erforderlich, dass sie ihre Änderungen möglichst nach abgeschlossener Arbeit in das Quellcodeverwaltungssystem einspielen. Nur so ist es möglich frühzeitig Fehler zu finden und die Vorteile einer kontinuierlichen Integration zu nutzen.

3.2 Verfügbare Software

Kontinuierliche Integration wird auf separaten Servern ausgeführt. Im Lauf der Zeit entwickelten sich mehrere Plattformen, welche die kontinuierliche Integrationsmethodik implementieren. Solche Server werden im weiteren verlauf dieser Arbeit CI Server genannt.

Für einen CI Server standen CruiseControl[26], Hudson[27], Jenkins[28] und BuildMaster[29] zur Auswahl. Während die ersten drei Anwendungen größtenteils in Java geschrieben sind, ist BuildMaster eine *.NET* Applikation und schied somit im Vorfeld aus, da der CI Server unter einem Unix Betriebssystem laufen sollte. Als Kandidaten für die nähere Auswahl wählte ich dann Hudson und Jenkins.

Die Software Hudson wurde ursprünglich von Kohsuke Kawaguchi[30], welcher damals bei Sun Microsystems als Entwickler tätig war, geschrieben. Sie hat eine pluginbasierte Architektur und läuft entweder als eigenständige Anwendung in einem GlassFish Container[31] oder kann direkt mit der vorhandenen *Web Application Archive* Datei in einen Apache Tomcat Server[32]

[26] http://cruisecontrol.sourceforge.net/
[27] http://hudson-ci.org/
[28] http://jenkins-ci.org/
[29] http://inedo.com/
[30] http://kohsuke.org/about/
[31] http://glassfish.java.net/
[32] Webserver der die Java Servlet und JavaServer Pages Technologien implementiert,

eingespielt werden.

Nach der Übernahme von Sun Microsystems durch Oracle kam es allerdings zu Streitigkeiten mit der Hudson Entwickler Gemeinschaft. Einerseits sicherte Oracle sich den Namen Hudson, andererseits wurde das Versionsverwaltungssystem umgestellt, so dass tagelang die Weiterentwicklung von Hudson still stand. Durch diese Streitigkeiten entstand letztenendes eine Abspaltung des Hudson Projekts, sowie dessen Umbenennung in Jenkins.

Da viele Entwickler dem Projekt Jenkins gefolgt sind, entschloss ich mich also dazu diesen CI Server zu verwenden.

3.3 Installation des CI Servers

3.3.1 Virtuelle Maschinen Konfiguration

Das Saros Projekt lässt ihre Server in virtuellen Maschinen laufen. Dazu wird spezielle Software benötigt, welche es ermöglicht eine virtuelle Rechnerarchitektur innerhalb einer realen Rechnerarchitektur zu emulieren. Dazu stellt der Fachbereich die Software VMWare zur Verfügung, welche auf den Gaia Clustern installiert ist.

Der AG Softwareengineering stehen hierzu vier virtuelle Rechner zur Nutzung bereit.

1. saros-build
   dieser Rechner ist zuständig für die Verwaltung diverser Dienste wie zum Beispiel das Reviewboard oder dem SVN Spiegel

2. saros-eclipse1
   Testrechner für das Saros Test-Framework

3. saros-eclipse2
   Testrechner für das Saros Test-Framework

4. saros-firewall
   Firewall Server der es ermöglicht, verschiedene Netzwerkeigenschaften (hohe Latenz, hoher Packetverlust, niedriger Datendurchsatz etc ...) zu simulieren.

Das folgende Bild veranschaulicht die Netzwerkkonfiguration zwischen den einzelnen Rechnern.

---

http://tomcat.apache.org/

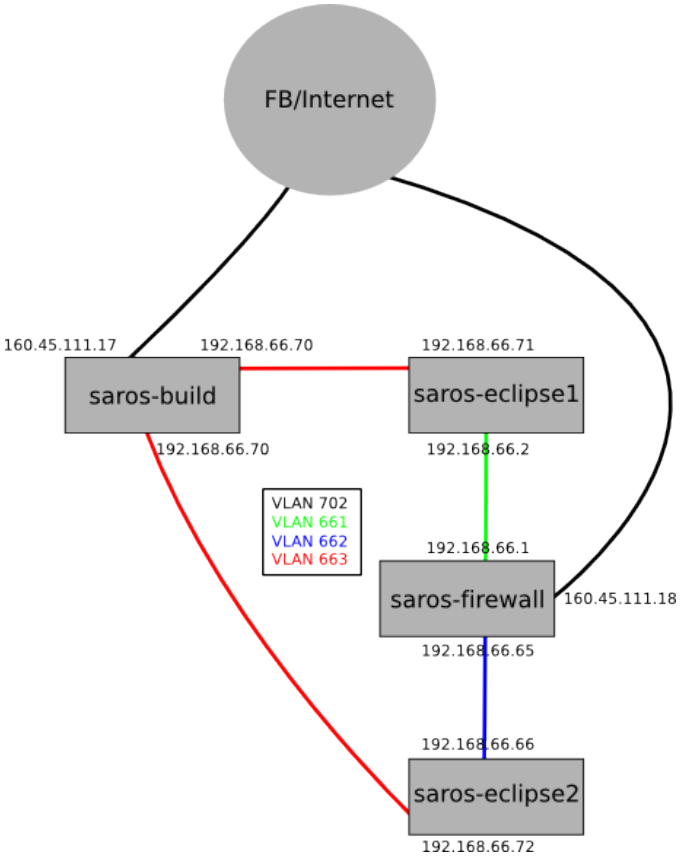

Abbildung 4: Netzwerkkonfiguration der virtuellen Maschinen

3.3.2 Einspielung von Jenkins in den Tomcat Server

Das Einspielen der Jenkins Software in den Tomcat Server ist normalerweise eine sehr einfache Angelegenheit. Dazu muss lediglich das *Web Application Archive* in das Verzeichnis `/var/lib/tomcat6/webapps` abgelegt werden. Da zuvor bereits eine Hudson Applikation installiert war, entfernte ich diese zugleich.

Da der Tomcat Server als *Daemon*[33] Prozess läuft muss man vor seinem Start noch eine Umgebungsvariable mit dem Namen `JENKINS_HOME` setzten, die auf ein Verzeichnis zeigen muss, in dem die Jenkins Applikation ihre Daten speichern kann.

Die Startskripte für Prozesse, die als *Daemon* gestartet werden, liegen unter Unix Betriebssystemen normalerweise im Verzeichnis `/etc/init.d`. In dem dort vorhandenen Skript `tomcat6` war bereits die Umgebungsvariable `HUDSON_HOME` gesetzt. Ich änderte also die Variable zu `JENKINS_HOME` und korrigierte den Pfad.

Ein Neustart des Tomcat Servers ergab, dass der Jenkins CI Server immer noch das alte Hudson Verzeichnis benutzte. Nach weiterer Analyse im *Daemon* Skript viel mir auf, dass dort noch die Datei `/etc/default/tomcat6` inkludiert wurde. In dieser Datei wurde abermals die Umgebungsvariable `HUDSON_HOME` gesetzt. Ich korrigierte also die entsprechende Zeile und löschte aus dem anderen Skript die zuvor definierte Umgebungsvariable. Nach einem weiteren Neustart zeigte sich jedoch immer noch das gleiche Verhalten. Erst nach mehreren Stunden weiterer Suche ergab sich, dass in den Tomcat Konfigurationsdateien zum dritten mal diese Variable gesetzt wurde. Nach deren Entfernung aus der Konfigurationsdatei und Neustart des Tomcat Servers wurde das neue Verzeichnis schließlich benutzt.

3.3.3 Konfiguration des Apache Web Server

Durch die Restriktion der Zedat sind auf dem Server `saros-build` nur die Ports 22, 80 und 443 erreichbar. Der Tomcat Server ist jedoch so konfiguriert, dass er Port 8080 benutzt. Dadurch ist der Tomcat mit seinen Applikationen nicht aus dem Internet erreichbar. Um dennoch den Zugriff zu ermöglichen muss der Apache Web Server so konfiguriert werden, dass er bestimmte URL[34] Anfragen an den AJP[35] Port des Tomcat Servers weiterleitet. Dazu müssen in die Apache Web Server Konfigurationdatei folgende

[33] Bezeichnet unter Unix ein Programm das im Hintergrund ausgeführt wird
[34] Uniform Resource Locator
[35] Apache JServ Protocol

Zeilen hinzugefügt werden:

```
# Continuous Integration
<Location /jenkins>
ProxyPass ajp://localhost:8009/jenkins
</Location>
```

Dadurch ist der CI Server über die URL http://saros-build.imp.fu-berlin.de/jenkins/ erreichbar.

3.4 Software für den CI Betrieb

Die Jenkins Applikation wird standardmäßig nur mit einem vorinstallierten CVS[36] und SVN[37] Plugin und einer Maven[38] Unterstützung ausgeliefert. Für das Saros Software wird zusätzliche Software benötigt, um sie bauen zu können.

3.4.1 Cobertura

Cobertura[39] ist ein Werkzeug zum Erstellen von Codeüberdeckungsberichten. Um dieses zu ermöglichen wird der fertig kompilierte Java Bytecode um weitere Befehle ergänzt. Diesen Vorgang bezeichnet man als Instrumentierung. Dazu werden vor und nach jedem Befehl im eigentlichen Programm Sprunganweisungen mit den dazugehörigen Parametern eingefügt, die auf Methoden der Cobertura Bibliothek zeigen. Diese Methoden sammeln dann diese Daten und werten sie bei Beendigung des Programmes aus.

Neben Cobertura existieren noch weitere Werkzeuge zum Erstellen einer Codeüberdeckung. Zu einem Emma[40], welches auch auf Bytecode Ebene arbeitet, jedoch nur auf Blockebene. Ein Block bezeichnet dabei eine Menge von Anweisungen in der der Kontrollfluss des Programmes nicht durch Sprunganweisungen und ähnlichen Befehlen, die ihn ändern können, beeinflusst wird. Eine andere Möglichkeit besteht darin, dass auf Quellcodeebene zusätzlicher Code hinzugefügt wird. Dieser Ansatz wird zum Beispiel vom Werkzeug Clover[41] benutzt.

[36] Concurrent Versions System, Versionsverwaltungssystem
[37] Apache Subversion, Versionsverwaltungssystem
[38] Werkzeug zum bauen von Software
[39] http://cobertura.sourceforge.net/
[40] http://emma.sourceforge.net/
[41] http://www.atlassian.com/software/clover/

3.4.2 Ant

Ant[42] ist ein Werkzeug zum Bauen von Software. Dabei wertet Ant eine sogenannte *Build* Datei aus, welche eine *XML*[43] Datei ist. Dabei werden die einzelnen Schritte, sogenannte *Tasks*, die benötigt werden, in Form von *XML* Elementen in diese Datei geschrieben. Hierbei kann man einzelne Abfolgen von Schritten zu Zielen, in Ant als *Target* bezeichnet, aggregieren. Beim Aufruf von Ant muss dann ein solches angegeben werden. Im weiteren ist möglich sogenannte Abhängigkeiten zwischen den Zielen zu definieren. Zum Beispiel kann man ein Ziel angeben, welches den Bytecode der Java Klassen instrumentieren soll. Damit dieses möglich ist muss der Bytecode natürlich vorhanden sein. Dazu gibt man ein weiteres Ziel an, welches für die Kompilierung des Quellcodes zuständig ist und verweist auf dieses als Vorbedingung. Ruft man nun Ant mit dem Instrumentierungs-Ziel auf, so wird automatisch erst die Kompilierung des Quellcodes veranlasst und im Anschluss der entstandene Bytecode instrumentiert. Ant ist durch Bibliotheken erweiterbar, welche neue *Tasks* definieren können.

3.4.3 Ant4Eclipse

Ant4Eclipse[44] erweitert das Ant Werkzeug um Befehle, die es ermöglichen OSGi[45] *Bundles* außerhalb der Eclipse *IDE* Umgebung zu erstellen.

3.4.4 FindBugs

FindBugs[46] ist ein Werkzeug das Bytecode von Java Klassen nach bestimmten Mustern durchsucht und bei Fund diesen Mustern ein bestimmten Eintrag aus einem Fehlerkatalog zuweist.

3.4.5 PMD

PMD[47] ist ein Werkzeug das den Quellcode von Java Klassen nach zuvor angegebenen, auch selbst definierten, Regeln durchsucht. Es wird in der Regel dazu verwendet den Quellcode nach Abweichungen von zuvor definierten *Coding conventions* zu durchsuchen. PMD besitzt bereits eine Menge vordefinierter Regeln, die vor allem syntaktisch schlecht geschriebenen Code finden können.

[42] http://ant.apache.org/
[43] Extensible Markup Language
[44] http://www.ant4eclipse.org/
[45] Open Services Gateway initiative, http://www.osgi.org/Main/HomePage
[46] http://findbugs.sourceforge.net/
[47] http://pmd.sourceforge.net/

3.5 Speicherort der Software

Hierbei stehen verschiedene Ansätze zur Verfügung. Einerseits ist es möglich alle benötigte Software, die zum Kompilieren der Komponenten oder des Systems benötigt werden, in das Versionsverwaltungssystem einzuspielen. Eine andere Möglichkeit ist die Installation der Software in ein Verzeichnis das lokal unter dem `JENKINS_HOME` Verzeichnis liegt. Diese Option ermöglicht es auf einfache Weise Jenkins so zu konfigurieren, dass ein Jenkins Job auf einem *Slave*[48] Rechner ausgeführt werden kann, der dann die selbe Verzeichnisstruktur aufweisen muss.

Die benötige Software um die Saros Anwendung zu kompilieren besteht aus dem Eclipse Frame Work mit Version 3.6 oder höher, Ant, Ant4Eclipse und dem JUnit Framework um die Saros Software zu testen.

Als weitere Software installierte ich außerdem das Dokumentationswerkzeug Doxygen[49] in der Version 1.7.4, welches von der Studentin Johannsen für ihre Diplomarbeit benötigt wurde. Ich installierte außerdem noch den Java Profiler JIP[50], welcher allerdings noch nicht in der CI genutzt wird.

3.6 Konfiguration des Jenkins Server

Wie im letzten Abschnitt beschrieben besteht die Standardinstallation des Jenkins Server nur aus drei Plugins. Deshalb installierte ich zunächst weitere Plugins. Als erstes jene, die die zuvor installierten Werkzeuge unterstützen sowie die Plugins *Lock and Latches* und *Copy Artifact*, welche für die STF Regression benötigt werden.

Des Weiteren wurden verschiedene Umgebungsvariablen definiert, welche den Jobs dann zur Verfügung stehen, sowie die Einstellung des Email Accounts damit der Jenkins Server bei auftretenden Fehlern das Saros Team benachrichtigen kann. Hierbei ist anzumerken, dass der Name des Email Account frei gewählt werden kann und nicht existieren muss, solange der angegeben *SMTP*[51] Server keine Authentifizierung benötigt.

Im letzten Schritt wurde der Jenkins Server gegen unerlaubte Benutzung abgesichert. Prinzipiell stehen hier mehrere Möglichkeiten zur Auswahl. Als geeignete Lösung wäre die Wahl auf die *LDAP*[52] Authentifizierung gefallen, jedoch gab es im Vorfeld bereits massive Probleme mit der Reviewboard

[48] Jenkins ermöglicht das verteilen von Jobs auf verschiedene Rechner
[49] http://www.doxygen.org/
[50] http://jiprof.sourceforge.net/
[51] Simple Mail Transfer Protocol
[52] Lightweight Directory Access Protocol

Software[53], welches auch das *LDAP* zur Anmeldung verwendete. Daher entschied ich mich für das eingebaute Benutzerverzeichnis.

3.7 Konfiguration der Testrechner

Um das Testen mittels des STF zu ermöglichen müssen auf Seiten der Testrechner bestimmte Bedingungen gegeben sein. Dieses erfordert zu einem eine lauffähige Eclipse Installation, einen SSH[54] Daemon, einen VNC[55] Server, eine Java JDK[56] Distribution und einen rsync[57] Daemon.

Als VNC Server wählte ich TightVNC[58], welcher bereits in der Diplomarbeit vom Studenten Szücs benutzt wurde. Für den Rechner `saros-eclipse1` wählte ich als JDK die Distribution von Oracle. Auf dem Rechner `saros-eclipse2` wurde das Open JDK installiert.

Als nächstes musste für die Eclipse Software noch ein SVN[59] Adapter installiert werden. Dazu muss ein *VNC* Server gestartet werden. Dieses geschieht durch den Aufruf von `vncserver`. Als Rückgabe wird auf der Konsole die *Display Nummer* angegeben. Jetzt muss man Eclipse noch mitteilen welches *Display* benutzt werden soll. Dazu exportiert man die Umgebungsvariable `DISPLAY` mit der dazugehörigen *Display Nummer*. Wurde zum Beispiel durch den Befehl `vncserver` ein *Display* mit der Nummer 5 erstellt, so lautet der dazugehörige Befehl in einer BASH[60] Shell[61] `export DISPLAY=:5` . Danach kann man Eclipse über die Konsole starten.

Um nun Zugriff zu erhalten, muss man sich mittels eines VNC Klienten mit dem VNC Server verbinden. Da die beiden Testrechner nur durch den Server `saros-build` zu erreichen sind, muss über einen *SSH* Tunnel eine Verbindung hergestellt werden. Dieses erreicht man, indem man auf dem `saros-build` Rechner den Befehl `ssh -L A:B:C:D administrator@saros-build` ausführt, wobei `A` die Netzwerkschnittstelle auf dem `saros-build` Rechner angibt. Diese kann mit * angegeben weden. `B` bezeichnet einen beliebigen freien Port. `C` gibt den Testrechner an. Dieses ist entweder die IP Adresse des Testrechners, oder ein Alias der in der `hosts` Datei gespeichert ist. `D` bezeichnet den Port an den die Daten von außerhalb gesendet werden sollen. Der Wert ergibt sich aus 5900 + Nummer des *Displays*.

[53] Werkzeug für Codedurchsichten, wird vom Saros Team verwendet
[54] Secure Shell, Netzwerkprotokoll zum erstellen Verschlüsselter Verbindungen
[55] Virtual Network Computing, Software zum steuern entfernter Rechner
[56] Java Development Kit
[57] Netzwerkprotokoll zu synchronisation von Dateien zwischen zwei Rechnern
[58] http://www.tightvnc.com/
[59] Apache Subversion
[60] Bourne-again shell, eine Unix Shell Implementierung
[61] Kommandozeileninterpreter

Die Standardinstallation des *SVN* Adapter benutzt native Komponenten. Dieses sind unter Windows Betriebssystemen DLL[62] und auf Unix Derivaten SO[63] Dateien. Diese können jedoch nicht über Eclipse installiert werden. Unter Unix Systemen, die mit dem Packet Manager `apt` arbeiten, können die nativen Komponenten mit dem Befehl `apt-get install libsvn-java` nachinstalliert werden.

3.8 Eclipse Features und Plugins

In normalen Java Projekten ist es üblich, den geschrieben Code mit Hilfe des Java Kompilierer zu kompilieren und diesen dann meistens im Anschluss in einem Jar Archiv zu speichern. Jedes Archiv kann dabei eine `MANIFEST.MF` Datei enthalten. Diese muss per Spezifikation[Jar] im Verzeichnis `META-INF` innerhalb des Archives liegen. In ihr sind Metainformation enthalten, die für die Ausführung des Codes benötigt werden. Im einfachsten Fall steht dort die auszuführende Java Klasse, welche eine `main` Methode enthält.

Die Eclipse Plattform unterscheidet jedoch wird zwischen *Product*, *Feature* und *Plugin*[Ecl].

Ein *Product* bezeichnet eine *RCP*[64] Applikation, die außerhalb der Eclipse Plattform lauffähig ist, jedoch einige Eclipse Komponenten benötigt.

Ein *Feature* bezeichnet eine Aggregation verschiedener Plugins. Das *Feature* wird durch eine *XML* Datei beschrieben in der die Versionen der Plugins vermerkt sind, sowie deren Abhängigkeiten zu anderen Plugins von Drittentwicklern. Diese Datei ist eine Art Installationsanleitung für die Eclipse Plattform, wenn man nachträglich Software Komponenten innerhalb der *IDE* installieren will.

Ein Plugin bezeichnet im eigentlichen Sinne ein Jar Archiv beziehungsweise dessen extrahierte Form innerhalb des Verzeichnisses `plugins` in der Eclipse Installation. Da Eclipse jedoch auf die OSGi Plattform aufbaut[Del06], wird solch ein Plugin im OSGi Kontext als *Bundle* bezeichnet. Ein Plugin enthält eine `plugin.xml` Datei. In dieser Datei stehen *GUI* spezifische Information für die Eclipse *IDE*. Die andere relevante Datei ist die `MANIFEST.MF` Datei. Hier stehen neben Abhängigkeiten zu anderen *Bundles*, welche bereits über die `feature.xml` Datei installiert sein sollten, Export Regeln und der komplette Klassenpfad für das *Bundle*. Dabei wird dieser Klassenpfad nicht

[62] Dynamic Link Library, enthält ausführbaren Maschinencode und kann dynamisch in den Arbeitsspeicher geladen werden
[63] Shared Object
[64] Rich Client Platform

durch den Java `SystemClassLoader` bedient, sondern vom OSGi Framework selbst erstellten Klassenladern. Sie ermöglichen es Klassen aus Jar Archiven zu laden, die sich wiederum in einem Jar Archiv innerhalb dessen befinden.

Beispiel:

```
Connection connection = new XMPPConnection();
```

Die Klassen `org.jivesoftware.smack.XMPPConnection` und `org.jivesoftware.smack.Connection` sind zur Zeit des Aufrufs noch nicht geladen worden. Der *OSGi* Klassenlader, der für das Bundle zuständig ist, sucht nun in der `MANIFEST.MF` nach dem Klassenpfad und durchsucht ihn in der angebenen Reihenfolge. Die zu ladenen Klassen befinden sich in dem Jar Archiv `smack.jar`, welches sich wiederum im Unterverzeichnis `lib` des Saros *Bundles* befindet.

Damit während der Produktion des *Bundles* referenzierte Bibliotheken innerhalb des Klassenpfades in das *Bundle* aufgenommen werden, müssen diese in der `build.properties` Datei eingetragen sein. Vergisst man diese Einträge, so wird das *Bundle* zwar ohne Beanstandungen kompiliert und gebaut, jedoch fehlen dann unter Umständen Klassen zur Laufzeit, was in der Regel in unerwarteten Verhalten resultiert[65].

3.9 Erzeugen von OSGi Bundles mit Ant4Eclipse

Normalerweise wird die Erstellung von *OSGi Bundles* unter Eclipse mit Hilfe der PDE[66] durchgeführt. Auf *IDE* Ebene ist dieser Prozess sehr einfach durchzuführen. Da aber innerhalb einer CI alles automatisiert ablaufen muss, steht demnach keine *IDE* zur Verfügung. Es muss also auf Konsolenebene gearbeitet werden, wo Zeile für Zeile bestimmte Anweisungen abgearbeitet werden. Ein sogenannter *PDE Build* ohne die Eclipse *IDE* ist relativ kompliziert durchzuführen[Ecl], weshalb ich auf die bereits existierende Lösung des Studenten Thiel zurückgriff.

Er benutzte dafür die Software Ant4Eclipse, welche neue Befehle für Ant zur Verfügung stellt. Mit Ant4Eclipse ist es sehr einfach ein *OSGi* Bundle zu erstellen. Innerhalb der Ant *Build* Datei muss lediglich der Ant4Eclipse *XML* Namensraum angeben und die dazugehörigen Makros importiert werden. Danach stehen eine Reihe neuer Befehle für Ant zur Verfügung.
Für die Produktion des *Bundles* benötigt man dem Befehl `<ant4eclipse:targetPlatform id=saros.target.platform>`. In diesem Befehl kann man per `<location />` Element angeben, wo die referenzierten *Bundles* liegen. Hierbei sind mehrere Verzeichnisse möglich. Der Inhalt des

[65] Der Student Szücs beschrieb solch ein Verhalten in seiner Diplomarbeit
[66] Plug-in Development Environment

XML Attributs `id` ist frei wählbar und wird für den darauf folgenden Befehl

```
<buildPlugin workspaceDirectory="${workspaceDirectory}
    " projectName="${plugin.name}" targetplatformid="
    saros.target.platform" destination="${build.dir}" /
    >
```

benötigt. Dieser Befehl baut dann das OSGi *Bundle*. Es muss dabei das Verzeichnis angegeben werden, in dem sich der Arbeitsbereich der Eclipse *IDE* befindet. Danach muss angegeben werden welches Projekt man bauen möchte. In diesem Fall wäre das unser Saros Plugin, dessen Name sich in der Datei `.project` innerhalb des *XML* Elements `<name />` befindet. Der nächste Parameter gibt die Zielplattform an die wir zuvor definiert haben. Es ist hierbei zu beachten, dass die *Bundles*, welche in den Verzeichnissen liegen mit der Architektur des Rechners übereinstimmen müssen. Das Ausführen des Befehl würde zum Beispiel scheitern, wenn man versuchen würden das *Bundle* gegen eine Unix x64 Architektur zu produzieren, der Produktionsrechner jedoch eine Sparc Architektur besitzt. Als letztes muss noch das Verzeichnis angegeben werden in dem das fertig produzierte *Bundle* gespeichert werden soll.

3.10 Jenkins Build Jobs

Der Jenkins CI Server benutzt sogenannte Jobs um seine Aufgaben zu erledigen. Da es durch die Pluginunterstützung möglich ist, die für die Jobs verfügbaren Optionen zu erweitern, beschreibe ich hier deshalb nur die Grundeigenschaften.

Jeder Job hat einen eindeutigen Namen und ein von Jenkins zugewiesenen Arbeitsbereich. Als vordefinierte Jobs stehen zur Auswahl:

Free Style-Softwareprojekt bauen Dieses ist der Standardjob. Er ermöglicht die volle Nutzung der installierten Plugins und beliebiges Ausführen von Skripten.

Maven 2/3 Projekt bauen Dieser Job ist nur für Projekte gedacht die Maven als Produktionswerkzeug einsetzen.

Externen Job überwachen Dieser Job ermöglicht es externen Programmen wie zum Beispiel `cron`[67] ihre Ausgabe an den CI Server zu schicken.

Multikonfigurationsprojekt bauen Dieser Job ist in erster Linie identisch mit dem Free Style Job. Allerdings wird hier eine Matrix mit

[67] ein Unix Daemon Prozess der periodisch andere Prozesse startet

Konfigurationsdaten erzeugt. So ist es zum Beispiel möglich eine Java Software die JDBC[68] benutzt auf verschiedenen Betriebssystemen und Rechnerarchitekturen gegen bestimmten DBMS[69] zu bauen und zu testen.

Die auf dem Jenkins CI Server laufenden Jobs sind ohne Ausnahme alles *Free Style Jobs*, so dass der volle Optionsumfang zu Verfügung steht.

3.11 Gliederung der Jobs

Die einfachste Möglichkeit einen Job zu erstellen besteht daraus in ihm die Software zu bauen, zu testen und sie dann gegebenenfalls auf dem Zielrechner zu installieren. Solch einen Job hatte ich auf dem früheren Hudson CI Server gefunden, wobei die Installation, welche in diesem Fall die Veröffentlichung des *Saros Features* auf der Sourceforge Plattform gewesen wäre, entfiel. Im Hinblick auf das STF und das kommende Framework vom Studenten Cordes wurden zuerst die JUnit Tests aus dem Job ausgelagert. Dieses sollte man immer in Betracht ziehen, sobald die Zeit, die für das Testen aufgewendet muss, ein gewisses Maß überschreitet. Außerdem wird dadurch eine bessere Auslastung des CI Servers erzielt, da mehrere Jobs nebenläufig ausgeführt werden können. Die Zeitspanne, die das Maß beschreibt, ist dabei keine feste Konstante sondern kann selbst spezifiziert werden.

Ich teilte zunächst das *Saros Feature* in seine einzelnen Plugins auf und wies jedem Plugin einen einzelnen Job zu. Als nächstes erfolgte die Erstellung von drei weiteren Jobs, die zum Testen des Saros Plugins dienten.

3.12 Konfiguration der Build Jobs

Die Build Jobs dienen dazu, dass die einzelnen Komponenten des *Saros Feature* getrennt gebaut werden können. Hierzu wird ein Job im *Webinterface* des Jenkins CI Servers ausgewählt, worauf hin die Möglichkeit erscheint diesen Job zu konfigurieren. Hierbei sollte man neben einer detaillierten Beschreibung auch darauf achten, wie lange die *Build* Artefakte gespeichert werden sollen. Vergisst man diese Einschränkung resultiert das auf lange Zeit hin in einem Speichermedium das keine Kapazität mehr für neue Daten hat.

Als nächstes muss angegeben werden, woher die Quelldateien bezogen werden sollen. Da dieses in der Regel durch ein Versionsverwaltungssystem geschieht, kommen die dafür nötigen Jenkins Plugins zur Verwendung, welche es ermöglichen den Quellcode aus dem Versionsverwaltungssystem zu

[68] Java Database Connectivity, die Datenbankschnittstelle der Java Plattform
[69] Datenbankmanagementsystem

beziehen. Für die einzelnen Plugins des *Saros Features* ist hierbei darauf zu achten, dass ein Verzeichnis mit angegeben werden muss, das unterhalb des Arbeitsbereiches liegt, da ansonsten das Ant4Eclipse Werkzeug das Projekt, welches gebaut werden soll, nicht finden kann. Zusätzlich kann optional noch angegeben werden, ob die Abfrage des Versionsverwaltungssystems in regelmäßigen Abständen automatisch stattfinden soll. Diese Option muss für eine CI natürlich aktiviert sein. Die Abfrage Intervalle sind mittel `cron` Syntax sehr variabel einstellbar.

Obwohl eher unüblich muss als nächster Schritt eine Semaphore benutzt werden, die man über das *Lock and Latches* Plugin anfordert. Dieses ist leider erforderlich, da Ant4Eclipse auf gemeinsamen Verzeichnissen unterhalb des, unter dem Betriebssystems festgelegten, temporär Verzeichnis arbeitet. Eine parallele Ausführung kann somit unter Umständen in einem Fehlschlag des *Builds* enden.

Im nächsten Schritt wird dann Ant aufgerufen, welches das OSGi *Bundle* baut. Sollte der Vorgang erfolgreich Verlaufen sein, so werden in den nächsten Schritten die Analysewerkzeuge aufgerufen um die nötigen Berichte zu erstellen.

Am Ende des ganzen Vorganges werden diese Berichte über die dazugehörigen Jenkins Plugins ausgewertet und in eine, für den Entwickler, lesbare Repräsentation überführt. Die erzeugten Artefakte, in diesem Fall das Saros Plugin, sowie dessen Quellcode werden archiviert und mit einer MD5[70] Prüfsumme versehen. Die Prüfsumme ermöglicht die Verfolgung der Artefakte, dass heißt man kann in anderen Jobs die Herkunft dieser Datei erkennen. Im Anschluss können dann weitere Jobs gestartet werden. Für das *Saros Feature* sind dieses die Jobs für den Bau des Whiteboard und Widget Gallery Plugins, sowie die Test Jobs für Saros.

3.13 Überarbeitung des FAD Skriptes

Der Student Szücs entwickelte im Rahmen seiner Diplomarbeit das STF. Damit man mit dem STF auf den Testrechnern testen konnte, schrieb er dazu mehrere Skripte, die den Start der Eclipse Anwendung auf den beiden Testrechnern ermöglichte.

Da die Skripte nicht gewartet wurden, konnten sie dem aktuellen Stand des STF nicht mehr genüge werden. Anstatt die Skripte zu überarbeiten übernahm ich nur das Startup Skript, welches auf den Testrechnern aus-

[70] Message-Digest Algorithm 5

geführt werden sollte. Als nächstes erstellte ich ein Skript, was nicht die eigentlich auszuführenden Schritte enthält, sondern selber ein Skript erzeugt. Diese Methode wird Metaprogrammierung genannt. Hierzu bekommt das Skript eine Reihe von Eingabeparametern, aus denen es dann Code für den Shell Interpreter erzeugt.

Nebenbei überarbeite ich noch das *Startup* Skript. Diesem müssen nun 3 Parameter übergeben werden. Zu einem die *Display Nummer* eines laufenden *VNC* Servers, der Name des Testers, welcher frei gewählt werden kann, sowie die Port Nummer, welche für die *RMI* Verbindung benötigt wird.

Im weiteren modifizierte ich das Skript in soweit, als dass nun das benötigte Jar Archiv, welches zum Starten der Eclipse Anwendung benötigt wird, automatisch gesucht wird. Ein anderer wichtiger Aspekt war es, das nach dem Start und bei Beendigung der Eclipse Anwendung keine unerwünschten Fenster geöffnet werden. Dieses betraf konkret die Nachfrage, ob der *SVN* Adapter Statistiken sammeln darf und den Bestätigungsdialog, wenn die Eclipse *IDE* beendet werden soll. Dieses erreicht man, wenn man im zu benutzenden Arbeitsbereich unter dem Verzeichnis `.metadata/.plugins/...` Dateien erstellt, in deren Inhalt ein Wert stehen muss, welcher die entsprechende Option deaktiviert. Dieses muss zuvor manuell ermittelt werden.

Damit die *JVM* die nativen Komponenten zur Laufzeit findet, muss die Umgebungsvariable `LD_LIBRARY_PATH` entsprechend angepasst werden. Hierbei ist es auch möglich direkt die Java Eigenschaft `java.library.path` mit den entsprechenden Verzeichnissen zu setzten, jedoch wird dadurch der zuvor von der *JVM* eingelesene Pfad aus der `LD_LIBRARY_PATH` Umgebungsvariable komplett überschrieben was unter Umständen zu Fehlverhalten führen kann.

Damit die durch Cobertura instrumentierten Java Klassen innerhalb eines instrumentierten *OSGi Bundle* fehlerfrei ausgeführt werden können, muss man der *OSGi* Plattform mitteilen, wo es diese Klassen finden kann, da sie nicht im *Bundle* enthalten sind. Dazu muss man zunächst den Java Klassenpfad so setzten, dass die benötigten Cobertura Bibliotheken in ihm enthalten sind. Damit nun die Plattform die Klassen aus den Bibliotheken finden kann, muss beim Start der Eclipse Anwendung die Java Eigenschaft `osgi.parentClassloader` auf den Wert `app` gesetzt werden.

Als letztes erstellte ich noch ein Skript mit dem es möglich ist, einzelne Eclipse Anwendungen auf den Testrechnern zu beenden.

3.14 Konfiguration der Test Jobs

3.14.1 Vorbedingungen

Ziel dieser Arbeit ist es, dass Tests automatisiert durchgeführt werden können. Um diesen Aspekt einzuhalten muss die Software Möglichkeiten zur Konfiguration bieten. Für einen automatisierten Testablauf ist es hinderlich, wenn im Programm bestimmte Elemente, wie zum Beispiel die Einstellung der Ereignisprotokollierung fest voreingestellt sind. Andere Merkmale sind fest voreingestellte Pfade zu Konfigurationsdateien, oder Werte innerhalb des Programms in Bezug auf *URL* Angaben, verwendetes *DBMS* und ähnliches.

Im weiteren müssen auch Namensregelungen in Betracht gezogen werden. Uneinheitlicher Aufbau von Verzeichnispfaden oder Dateinamen kann zum Beispiel zur unnötigen Verkomplizierung der zu verwendeten Skripte führen, da unter Umständen mehrere Möglichkeiten berücksichtigt werden müssen, welche sich auf deren Quellcodelänge auswirkt. Zum Teil werden deswegen komplizierte reguläre Ausdrücke als Abhilfe genutzt die eine Wartung der Skripte erschweren können.

3.14.2 Saros Junit Test

Dieser Test Job wird immer im Anschluss nach einem erfolgreichen *Build* des Saros Plugins ausgelöst. Er fordert, über das *Copy Artefact* Plugin, das letzte erfolgreich gebaute Artefakt, sowie dessen Quellcode an und speichert diese dann in dessen Arbeitsbereich. Wie bereits vorher beschrieben finden diese Tests außerhalb der Eclipse Entwicklungsumgebung statt. Da sich jedoch die notwendigen Bibliotheken innerhalb des *Bundles* befinden, müssen diese vor dem Start der Regression zuerst extrahiert werden. Dieses geschieht mit dem Programm `jar` welches in jeder Java JDK[71] Distribution enthalten ist. Damit man später im Codeüberdeckungsbericht auch den Quellcode sehen kann, muss dieser zusätzlich entpackt werden. Im nächsten Schritt wird eine Ant *Build* Datei mit den notwendigen Parametern aufgerufen. Hier werden über über verschiedene Ant *Tasks* zuerst der Bytecode instrumentiert und im Anschluß eine Regression über alle vorhandenen Testfälle gestartet. In diesem *JUnit Task* wird auf die Angabe einer Testsuite Klasse verzichtet. Anstelle der Angabe einer Testsuite wird der sogenannte Stapel Modus verwendet. JUnit sucht dabei nach Testklassen die mit einem bestimmten regulären Ausdruck übereinstimmen und führt diese aus.

Wie in den Vorbedingung beschrieben, sind hier Namensregeln für die Dateien und Verzeichnisse erforderlich. Dieses gilt nicht nur für die Suche nach Testfällen, sondern auch für die zu instrumentierenden Java Klassen. Im

[71] Java Development Kit

einfachsten Fall kann es passieren, dass die Testklassen einfach mit instrumentiert werden und dann später im Bericht auftauchen, was ihn verfälschen kann.

Wie bereits in den Build Jobs beschrieben, werden am Ende der Regression wieder die entstandenen Daten durch die Jenkins Plugins ausgewertet. Dieses sind Codeüberdeckungsdaten, sowie die Ergebnisse der Regression selber.

3.14.3 Saros STF Tests

Diese beiden Jobs sind in sofern identisch, in dass sie sich nur von der Menge der instrumentierten Klassen und auszuführenden Testfälle unterscheiden. Dabei testet der Job `Test STF` das Framework selber und `Test Saros STF` führt eine Regression der Testfälle durch, die verschiedene Anwendungsszenarien durchspielen.

Im Gegensatz zum JUnit Test Job ist es nicht möglich den gesamten Vorgang durch eine *XML* Datei zu beschreiben. Theoretisch könnten man sogar komplett auf den Einsatz von Ant verzichten. Allerdings erstellt das JUnit Framework von sich aus keine *XML* Dateien aus den Testabläufen, welche aber von Jenkins benötigt werden, so dass zumindestens die Tests über Ant ausgeführt werden müssen.

Als ersten Schritt muss dazu eine Datei erstellt werden, die die Konfigurationsdaten der Tester enthält. Diese kann theoretisch einmal erstellt und abgespeichert werden. Innerhalb der Jobs wird diese aber jedesmal neu erzeugt.

```
# STF RUN CONFIGURATION

echo "ALICE_JID = jenkins_alice_stf@saros-con.imp.fu-
    berlin.de/Saros" > stf_config
echo "ALICE_PASSWORD = jenkins" >> stf_config
echo "ALICE_HOST = 192.168.66.129" >> stf_config
echo "ALICE_PORT = 12345" >> stf_config

echo "BOB_JID = jenkins_bob_stf@saros-con.imp.fu-
    berlin.de/Saros" >> stf_config
echo "BOB_PASSWORD = jenkins" >> stf_config
echo "BOB_HOST = 192.168.66.130" >> stf_config
echo "BOB_PORT = 12346" >> stf_config

echo "CARL_JID = jenkins_carl_stf@saros-con.imp.fu-
    berlin.de/Saros" >> stf_config
echo "CARL_PASSWORD = jenkins" >> stf_config
echo "CARL_HOST = 192.168.66.129" >> stf_config
```

```
echo "CARL_PORT = 12347" >> stf_config
```

Danach müssen für die Codeüberdeckung die Klassen instrumentiert werden. Dieses erreicht man durch den Aufruf:

```
"${JENKINS_HOME}/tools/cobertura/cobertura-instrument.
   sh" --basedir "${WORKSPACE}" --destination "${
   WORKSPACE}/instr" --includeClasses 'de\.fu_berlin\.
   inf\.dpp\.stf\.server\..*' de.fu_berlin.inf.dpp_*
```

Die Umgebungsvariablen `JENKINS_HOME` und `WORKSPACE` werden vom Jenkins Server vor der Ausführung des Jobs mit Werten initialisiert. Der Parameter `--basedir` gibt hierbei an, wo Cobertura Java Klassen und Jar Archive suchen soll. `--destination` beschreibt das Zielverzeichnis in dem die instrumentierten Klassen und Jar Archive gespeichert werden sollen. Als weitere Optionen hat man die Möglichkeit mit den Befehlen `--includeClasses` und `--excludeClasses` durch reguläre Ausdrücke Cobertura mitzuteilen, welche Klassen instrumentiert werden sollen und welche nicht. Die nachfolgenden Parameter verweisen auf die zu instrumentierenden Java Klassen und Jar Archive.

Im nächsten Schritt wird aus den Konfigurationsdaten der Tester das benötigte Skript für die Regression erstellt. Dieses geschieht mit dem Aufruf:

```
"${JENKINS_HOME}/scripts/stf/stf_script_gen" --
   cobertura "${JENKINS_HOME}/tools/cobertura" "${
   JENKINS_HOME}/ssh_keys/saros-build" instr/de.
   fu_berlin.inf.dpp_* stf_config | tee stf_regression
   .sh
```

Der Parameter `--cobertura` gibt an, wo die Cobertura Bibliotheken zu finden sind. Der nächste Parameter muss auf eine Datei mit einem privaten *SSH* Schlüssel verweisen. Der nächste Parameter auf die Saros Plugin Datei und der letzte auf die Konfigurationsdatei für die Tester.

Mit diesem Skript kann nun die Regression begonnen werden. Der Ablauf sieht dann wie folgt aus:

```
. ./stf_regression.sh

set +e

deploy

stop_vnc_server

start_vnc_server

start_remote_bots
```

```
sleep 30
echo "STARTING REGRESSION: TIMEOUT IS 30 MINUTES"
"${JENKINS_HOME}/tools/ant/bin/ant" -Dsrc.dir=src -
   Dlib.dir=lib -Declipse.dir=/home/build/eclipse -
   Djunit.dir=junit "-Dsaros.plugin.dir=${WORKSPACE}"
   "-Dstf.client.config.files=${WORKSPACE}/stf_config"
    -lib ${JENKINS_HOME}/tools/junit -f
   saros_stf_self_test.xml &

ANT_PID=$!

wait_until_timeout 60 30 $ANT_PID

TIMEOUT=$?

if [ $TIMEOUT -eq 0 ]
then
"${JAVA_HOME}/bin/java" -cp "${ECLIPSE_HOME}/plugins
   /*:lib/*:*" "-Dde.fu_berlin.inf.dpp.stf.client.
   configuration.files=${WORKSPACE}/stf_config" de.
   fu_berlin.inf.dpp.stf.client.ShutdownRemoteEclipse
sleep 60
fi

stop_remote_bots KILL

kill_ssh_connections

stop_vnc_server

fetch_code_coverage

fetch_screen_shots

exit $TIMEOUT
```

Der Funktionsaufruf `deploy` bewirkt, dass das Saros Plugin, die Cobertura Bibliotheken und die beiden Skripte auf die Testrechner kopiert werden. Mit der Funktion `stop_vnc_server` werden danach alle laufenden *VNC* Server auf beiden Testrechnern gestoppt. `start_vnc_server` startet dann die nötige Anzahl an *Displays*. Als nächstes wird `start_remote_bots` aufgerufen. Damit werden die Eclipse Anwendungen auf den Testrechner gestartet.

Jetzt erfolgt der Aufruf für die Regression. Dazu wird Ant im Hintergrund mit den nötigen Parametern gestartet. Dieser Prozess wird mittels eines Wachhundes überwacht. Sollte der Prozess nun in der vorgegebenen Zeit terminieren, wird versucht die Eclipse Anwendungen normal zu beenden

damit Cobertura eine Codeüberdeckung erzeugen kann. Um unter Unix Betriebssystemen einem Prozess mitzuteilen, dass er sich beenden soll, sendet man ihm das Signal `SIGTERM`.

Jedoch musste ich feststellen, dass es unter dem Unix Derivat Ubuntu egal war, ob man dem Eclipse Prozess ein `SIGTERM` oder `SIGKILL` schickte. Dieses hatte zur Folge, dass Cobertura seine Daten nicht erstellen konnte. Um dieses Problem zu umgehen, schrieb ich eine Java Klasse die für alle Tester versucht, über das STF, die Eclipse Anwendung über ihr Menu zu beenden. Die Ausführung dieser Klasse findet in dem `if` Block des Skriptes statt. Es ist hierbei anzumerken, dass diese Berechnung der Codeüberdeckung unter Umständen einige Minuten dauern kann. Der genaue Wert muss deshalb empirisch ermittelt werden.

Im Anschluss werden alle genutzten Ressourcen freigegeben und die Codeüberdeckungsdaten sowie die Screenshots von den Testrechnern im Arbeitsbereich des Jobs gespeichert. Sollte eine Zeitüberschreitung stattgefunden haben, so wird an dieser Stelle nun der Job abgebrochen. Dieses ist dann optisch durch einen roten Ball in der Jenkins Übersicht zu erkennen.

Damit ein Codeüberdeckungsbericht erstellt werden kann, muss man Cobertura noch die gesammelten Daten auswerten lassen, dieses geschieht mit den Aufrufen:

```
"${JENKINS_HOME}/tools/cobertura/cobertura-merge.sh"
   coverage*
"${JENKINS_HOME}/tools/cobertura/cobertura-report.sh"
   --format xml --destination "${WORKSPACE}" "${
   WORKSPACE}/src"
```

3.14.4 TestLink

TestLink[72] ist ein Werkzeug zum Erstellen, Speichern und Verwalten von Testfallspezifikationen. Es kann mehrere Projekte verwalten und Unterstützt die Aggregation von einzelnen Testfällen zu einem Testplan.

Da mit wachsender Anzahl an STF Testfällen auch die Ausführungszeit steigt, ist es von Vorteil wenn man nur die Ergebnisse der Tests des aktuellen Testplan überprüfen kann. Dieses spart Zeit und gibt dem Entwickler die Möglichkeit, schneller seine Korrekturen oder Erweiterungen zu überprüfen.

Damit der Jenkins CI Server diese Testfälle automatisch ausführen kann,

[72] http://www.teamst.org/

wird das TestLink Plugin[73] für Jenkins benötigt. Das Plugin benutzt die *XMLRPC* Schnittstelle von TestLink, wodurch es möglich ist Testpläne abzufragen und sie zu aktualisieren.

Damit die Testfallspezifikationen Implementierungen zugewiesen und ausgeführt werden können, wird an jede Spezifikation ein anpassbares Feld angehängt. In diesem soll die dazugehörige Java Klasse angegeben werden. Dieses hätte aber zum Hindernis, dass man die in Java geschriebenen Testfälle nicht ohne weiteres refaktoriersieren könnte. Eine Änderung des Klassennamens oder das Verschieben in ein anderes Packet hätte zur Folge, dass man dieses Feld neu anpassen müsste. Dieses ist jedoch bei bereits ausgeführten Testfallspezifikationen nicht mehr möglich und kann nur durch Änderung zur einer neuen Version ermöglicht werden.

Ich beschloss deshalb Java `Annotations`[Jav] zu benutzten. Diese Annotationen stellen Metadaten bereit und können entweder an einer Klasse, Methode oder Variable angehängt werden. Des weiteren kann man noch bestimmen, zu welchem Zeitpunkt die Annotation nicht mehr verfügbar sein soll. Dieses kann man über die Werte `Class`, `Runtime` und `Source` angeben. `Source` bewirkt, dass die Annotation nach der Kompilierung der Klasse nicht mehr vorhanden ist. Der Java Kompilierer benutzt zum Beispiel die Annotation `@Override` während der Kompilierung um festzustellen, ob man eine Methode der Basisklasse überschreibt. Ist dieses nicht der Fall so bricht der Kompilierungsvorgang mit einer entsprechenden Fehlermeldung ab. `Class` gibt an, dass die Annotation erst beim Laden der Klasse entfernt werden soll. Dieses ist nützlich für Frameworks die ihre eigene Klassenlader benutzen und vor dem Laden der Klassen noch den Bytecode ändern müssen. Die Option `Runtime` bewirkt, dass die Annotationen zur Laufzeit über die Java Reflection Schnittstelle ausgelesen werden können.

Anhand dieser Optionen erstellte ich eine `@TestLink` Annotation, welche an eine Klasse angehängt werden kann. Sie ist zur Laufzeit verfügbar und enthält ein ID Feld. Der Wert dieses Feldes muss mit dem Wert übereinstimmen, der in der Testfallspezifikation angegeben ist.
Bevor das TestLink Plugin innerhalb des Jobs gestartet werden kann, wird ein Java Klasse ausgeführt, welches alle Java Klassen des Saros Plugins durchsucht und bei Fund eine Abbildung zwischen *ID* und vollständigen Paketnamen erstellt und auf dem Standardausgabekanal ausgibt. Mit Hilfe dieser Datei können nun den Testfallspezifikationen konkrete Java Klassen zugeordnet werden. Damit nach Abschluss eines Testfalls das TestLink Plugin in der Lage ist die Daten auszuwerten, muss die durch den Ant JUnit Befehl erstellte *XML* Datei noch einmal nachträglich manipuliert werden.

[73] https://wiki.jenkins-ci.org/display/JENKINS/TestLink+Plugin

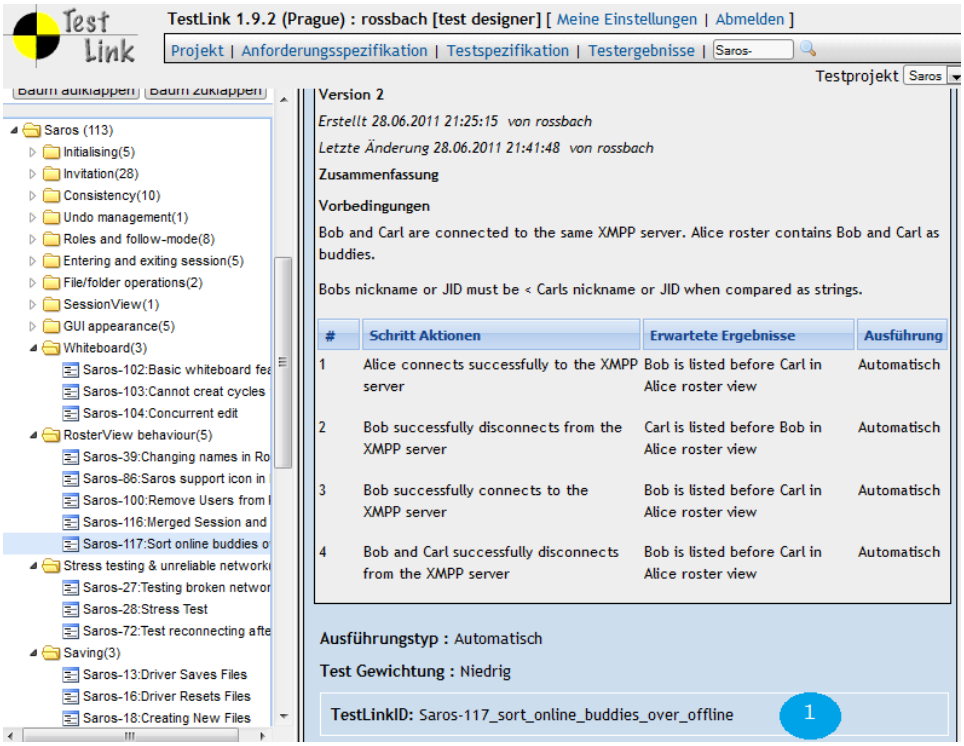

Abbildung 5: ID Feld (1) innerhalb einer TestLink Testfallspezifikation

Dieses ist notwendig, da in den *XML* Dateien der vollständige Klassenname vermerkt ist aber das TestLink Plugin nach der *ID* Zeichenkette sucht. Dieses wird durch ein Skript ermöglicht, was nach jedem Testfall die *XML* Datei des Testfalls automatisch anpasst.

4 Arbeiten im Saros-Team

Im Rahmen dieser Bachelorarbeit war ich auch Mitglied des Saros Entwickler Teams. Innerhalb dieses Teams ist man dafür verantwortlich, die Saros Anwendung zu verbessern, Fehler zu korrigieren und im Entwicklungsprozess mitzuhelfen.

Saros wird in iterativen Zyklen, die jeweils einen Monat in Anspruch nehmen entwickelt. In den ersten drei Wochen wird die Weiterentwicklung von Saros vorangetrieben. Diese beinhaltet unter anderem Integration von neuer Funktionalität, Verbesserung solcher und das beseitigen von Fehlern. Die letzte Woche ist die sogenannte *Release* Woche. Hier werden innerhalb des Teams verschiedene Rollen auf die einzelnen Mitglieder verteilt.

Der Release Manager, sowie der ihm zu assistierende Release Manager sind dafür verantwortlich die Änderungen innerhalb des aktuellen Zyklus zu erfassen und diese in einem Changelog[1] festzuhalten. Dazu wird der aktuelle Entwicklungszweig in einen separaten Zweig verschoben. Aus diesem Zweig wird am Abschluss der *Release* Woche das Saros Feature produziert und auf die Sourceforge Plattform eingespielt. Danach werden in diesem Zweig eingespielte Fehlerkorrekturen zurück in den Entwicklungszweig gespielt.

Der Test Manager und der assistierende Test Manager sind dafür verantwortlich aus der Testlink Datenbank Testfälle auszuwählen oder neu zu erstellen und mit ihnen einen Testplan zu generieren. Dieser muss dann abgearbeitet werden. Dabei kann es vorkommen, dass weitere Mitglieder gebraucht werden, da manche Tests bis zu vier Tester benötigen. Nach Abschluss der Tests wird das Ergebnis den anderen Mitgliedern mitgeteilt.

Der Dokument Manager ist dafür verantwortlich Dokumentation in bereits vorhandener oder mit dem *Release* neu dazugekommener Form auf Fehler zu analysieren und deren Korrektur auf die zuständigen Mitglieder aufzuteilen.

Am letzten Tag der*Release* Woche findet zusätzlich ein Akzeptanztest statt, der durch Mitglieder der AG Software Engineering durchgeführt wird. Sollte dieser positiv Verlaufen, also keine gravierenden Mängel in der Software auftreten, ist damit der aktuelle Zyklus beendet.

Durch die tägliche STF Regression konnten viele Fehler innerhalb von Saros gefunden werden. Dieses waren zum größten Teil `NullPointerExceptions` oder Ausnahmen die mit Nebenläufigkeit zu tun hatten. Da ich mich nicht in alle Teile des Codes einarbeiten konnte, war es mir jedoch nicht möglich

[1] Datei die die Veränderungen zwischen verschiedenen Versionen beinhaltet

die `NullPointerExceptions` zu beheben, da eine bloße `if` Abfrage unter Umständen die Semantik verändert hätte. Zudem fehlte zum Teil in der Dokumentation, dieser Methoden, die Spezifikation der zulässigen Wertebereiche der einzelnen Parameter. Die Fehler, welche die Nebenläufigkeit betrafen wurden von mir behoben. Diese betrafen unter anderen einen Fehler im `StreamServiceManager`, der bei Beendigung einer `StreamSession` die gesamte Eclipse Anwendung einfrieren ließ.

Andere Korrekturen betrafen unter anderem die Unterstützung von Java 7 für Saros. Dazu musst eine neue Version der XStream Bibliothek benutzt werden. Nach dem Einspielen erfolgte jedoch ein neuer Fehler der daraus resultierte, dass in dieser Bibliothek auf Klassen referenziert wurde, die innerhalb des Jar Archivs nicht vorhanden waren. Nach dem Einspielen der benötigten Klassen, die in einer anderen Bibliothek zu finden waren und Anpassung des Klassenpfades konnte Saros mit Java 7 benutzt werden. Ein anderer Fehler betraf das Hinzufügen von Kontakten. Dieser trat jedoch nur auf dem von Saros zur Verfügung gestellten eJabberd Server auf. Durch hinzufügen der Server Ressource zur JID konnte dieses Problem gelöst werden. Dieses war jedoch innerhalb der XEP-30 Spezifikation nicht beschrieben, so dass es möglich ist, dass ein Defekt oder Fehlkonfiguration innerhalb des eJabberd Servers vorliegt.

Die letzte Verbesserung betraf die Überarbeitung der `Screensharing` Funktionalität welche durch den Studenten Lau in seiner Bachelorarbeit[Lau10] eingeführt wurde. In seiner Arbeit verweist er dabei auf Literatur die besagt, dass Java langsam sei. Hier ist anzumerken, dass diese Literatur aus dem Jahre 2000 stammt und veraltet ist. Techniken zur Steigerung der Ausführungszeit wurden bereits durch JIT Kompilierer beschrieben.

Um die Geschwindigkeit zu verbessern, implementierte ich den von ihm vorgeschlagenen Algorithmus noch einmal neu. Eine Änderung betraf die Kachelgröße, welche von 32x32 Bildpunkten auf 8x8 Bildpunkten reduziert wurde. Zum vergleichen der Bilder wurde im weiteren eine von mir damals im Computer Graphik Praktikum benutzte Methode aufgerufen. Diese ermöglicht den direkten Zugriff auf die Bildpunktdaten, so dass zusätzliches kopieren von Speicherbereichen entfällt. Die Daten der beiden Bilder werden nun jeweils in 8x8 Kacheln unterteilt und verglichen. Damit dieser Vorgang beschleunigt wird, basiert die von mit dazu geschriebene Methode nur aus arithmetischen Operationen und einer einzigen Vergleichsfunktion. Sie soll damit Sprungvorhersage möglichst eliminieren, welche zu *Pipeline Stalls* führen kann.

Im Anschluss wird versucht die veränderten Kacheln zu größeren Kacheln zu aggregieren. Den dafür benötigten Algorithmus von Breu[DSS89] konn-

te ich aus Zeitmangel nicht implementieren. Danach werden diese Kacheln komprimiert und an den Zielrechner übertragen. Dabei entfernte ich die im Code vorhandene Java Klasse, welche zur Reduktion der Bilddaten zuständig war. Sie war überflüssig, da im Anschluss eine `JPEG`[1] Komprimierung über die `ImageIO API` statt fand. Der `JPEG Kompressor` ist Bestandteil der Java Spezifikation und somit in allen Distributionen enthalten. Innerhalb der Oracle Distribution ist dieser zudem als native Komponente implementiert, so dass davon auszugehen ist, dass der erweiterte Befehlssatz zur Multimedia Unterstützung, wie `SSE` oder `3DNow`, genutzt wird.

[1] Norm die Methoden zur Bildkompression beschreibt

5 Zusammenfassung

In dieser Arbeit wurde der Entwicklungsprozess von Saros um eine kontinuierliche Integrationsumgebung erweitert, sowie Änderungen an dem Saros Testframework vorgenommen, welche die Stabilität und Konfigurierbarkeit verbessern.

Für die kontinuierliche Integration wurde dazu ein CI Server und die benötigten Werkzeuge installiert und konfiguriert. Es wurden Build Jobs für die einzelnen Plugins des Saros Features erstellt. Diese werden nach Änderungen im Versionsverwaltungssytem automatisch gestartet, bauen dann die Komponenten und analysieren diese dann statisch auf Fehler.

Im Anschluss findet eine automatisierte Regression statt, die die Qualität von Saros sichern soll. Dazu kommen JUnit Tests und das Saros Test-Framework zum Einsatz. Für das Framework wurden zwei virtuelle Maschinen konfiguriert auf denen dann die Testexemplare gestartet werden können.

Um eine Verbesserung der Test-Automatisierung zu erreichen, kann der Entwickler nun seine Testfälle mit einer Testlink Annotation versehen. Diese ermöglicht es Testpläne aus komplett automatisierten Tests zu erstellen und diese dann von dem CI Server ausführen zu lassen.

Die Probleme, welche für das Versagen des Saros Test-Frameworks verantwortlich sind wurden analysiert und sofern möglich korrigiert. Die gewonnenen Kenntnisse sollen es späteren Entwicklern ermöglichen, die zum Teil komplexen Zusammenhänge besser zu verstehen und noch bestehende Probleme schneller zu lösen.

Im weiteren wurde das Test-Framework verbessert. Es wurde Logik eingefügt, die es ermöglicht dem Entwickler bessere Daten für die Analyse zu liefern. Im weiteren erfolgt nun die Konfiguration der Tester über Eigenschaftsdateien. Dadurch ist dem Entwickler möglich, die benötigten XMPP Accounts für die Regression oder einzelne Tests frei zu wählen. Dieses ermöglicht paralleles Testen innerhalb des Teams und im weiteren auch die Unterstützung für ein dynamisches Erstellen von Testrechnern, sowie deren Ansteuerung über den Klienten des Frameworks.

Literatur

[Bec04] Kent Beck. *Extreme Programming Explained, Second Edition: Embrace change.* Addison-Wesley Professional, 2004.

[Che11] Lin Chen. Einführung eines Testprozesses. Diplomarbeit, Freie Universität Berlin, Inst. für Informatik, 2011.

[Del06] Scott Delap. Understanding how Eclipse plug-ins work with OSGi. http://www.ibm.com/developerworks/library/os-ecl-osgi/index.htmll, 2006.

[Dje06] R. Djemili. Entwicklung einer Eclipse-Erweiterung zur Realisierung und Protokollierungverteilter Paarprogrammierung. Diplomarbeit, Freie Universität Berlin, Inst. für Informatik, 2006.

[DMG07] Paul Duval, Steve Matyas, and Andrew Glover. *Continuous Integration: Improving Software Quality and Reducing Risk.* Addison-Wesley, 2007.

[DSS89] F. Dehne, J. Sack, and N. Santoro. *Algorithms and Data Structures Workshop WADS '89 Ottawa, Canada, August 17–19, 1989 Proceedings*, chapter An efficient algorithm for finding all maximal square blocks in a matrix. Springer Berlin / Heidelberg, 1989.

[Ecl] Plug-in Development Environment Guide. http://help.eclipse.org/galileo/index.jsp?nav=/4.

[Fow] Martin Fowler. Continuous Integration. http://martinfowler.com/articles/continuousIntegration.html.

[Fow05] Martin Fowler. Fluent Interface. http://martinfowler.com/bliki/FluentInterface.html, 2005.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software.* Addison-Wesley Professional, 1995.

[Gus07] Björn Gustavs. Weiterentwicklung des Eclipse-Plug-Ins Saros zur Verteilten Paarprogrammierung. http://www.inf.fu-berlin.de/inst/ag-se/theses/Gustavs_Saros_DPPII.pdf, 2007.

[Gus11] Björn Gustavs. Stabilitäts- und Testbarkeitsverbesserungen der Netzwerkschicht in Saros. Diplomarbeit, Freie Universität Berlin, Inst. für Informatik, 2011.

[Jar] JAR File Specification. http://download.oracle.com/javase/1.5.0/docs/guide/jar/jar.html.

[Jav] Annotations. http://download.oracle.com/javase/1.5.0/docs/guide/language/annotations.html.

[JVM] The Java HotSpot Performance Engine Architecture. http://java.sun.com/products/hotspot/whitepaper.html.

[Lau10] Stephan Lau. Verbesserte Präsenz durch Screensharing für ein Werkzeug zur verteilten Paarprogrammierung. Bachelorarbeit, Freie Universität Berlin, Inst. für Informatik, 2010.

[LY] Tim Lindholm and Frank Yellin. The Java Virtual Machine Specification Second Edition. http://java.sun.com/docs/books/jvms/second_edition/html/VMSpecTOC.doc.html.

[Rie08] Oliver Rieger. Weiterentwicklung einer Eclipse-Erweiterung für verteilte Paar-Programmierung im Hinblick auf Kollaboration und Kommunikation. Diplomarbeit, Freie Universität Berlin, 2008.

[RMIa] Java Remote Method Invocation - Distributed Computing for Java. http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138781.html.

[RMIb] Java™ Remote Method Invocation Specification. http://java.sun.com/j2se/1.5/pdf/rmi-spec-1.5.0.pdf.

[Sun06] Sun Microsystems. *Memory Management in the Java HotSpot™ Virtual Machine*, 2006.

[Szu] Sandor Szuecs. Behandlung von Netzwerk- und Sicherheitsaspekten in einem Werkzeug zur verteilten Paarprogrammierung. Diplomarbeit, Freie Universität Berlin, Inst. für Informatik.

[Wik] Wikipedia: Just-in-time-Kompilierung. http://de.wikipedia.org/wiki/Just-in-time-Kompilierungl.

[Zie09] Franz Zieres. Das A. Bachelorarbeit, Freie Universität Berlin, Inst. für Informatik, 2009.

A SWTBot Fehler

A.1 Email

```
Subject: Re: Deadlock on Eclipse EDT if logging is enabled
From: Ketan Padegaonkar <ketanpadegaonkar@gmail.com>
To: Stefan Rossbach <rossbach@inf.fu-berlin.de>

Hi,

This information is quite useful! Thanks for investigating.

Would you mind opening a
bug(https://bugs.eclipse.org/bugs/enter_bug.cgi?product=3DSWTBot) so
someone can take a look at it. I think I'd likely have a workaround
for you in a while.

Ketan
studios.thoughtworks.com | twitter.com/ketanpkr




On Fri, Jun 24, 2011 at 6:22 AM, Stefan Rossbach

<rossbach@inf.fu-berlin.de> wrote:
> Hi Ketan,
>
> I am currently one of the developers of the Saros project. (
> http://www.saros-project.org/ )
>
> To ensure the quality of our product, we use your SWTBOT framework for
> automated gui testing.
> We are currently using Jenkins CI server for the regression and so i had =
to
> enable the logging for
> SWTBOT because no one is actually watching the screen(s) while the
> regression is run.
>
> Our current problem is that sometimes Eclipse freezes randomly. I tracked
> down the problem
> to this level.
>
```

```
> Name: main
> State: BLOCKED on org.apache.log4j.spi.RootLogger@4c1a841e owned by: RMI =
TCP
> Connection(10)-127.0.0.1
> Total blocked: 15.962 =A0Total waited: 229
>
> Stack trace:
> org.apache.log4j.Category.callAppenders(Category.java:204)
> org.apache.log4j.Category.forcedLog(Category.java:391)
> org.apache.log4j.Category.debug(Category.java:260)
> de.fu_berlin.inf.dpp.editor.EditorManager.lockAllEditors(EditorManager.ja=
va:1739)
> de.fu_berlin.inf.dpp.editor.EditorManager$2.block(EditorManager.java:198)
> de.fu_berlin.inf.dpp.synchronize.StopManager.lockProject(StopManager.java=
:378)
> de.fu_berlin.inf.dpp.synchronize.StopManager$4.run(StopManager.java:324)
> de.fu_berlin.inf.dpp.util.Utils$6.run(Utils.java:408)
> org.eclipse.swt.widgets.RunnableLock.run(RunnableLock.java:35)
> org.eclipse.swt.widgets.Synchronizer.runAsyncMessages(Synchronizer.java:1=
34)
> =A0 - locked org.eclipse.swt.widgets.RunnableLock@2ed070a4
> org.eclipse.swt.widgets.Display.runAsyncMessages(Display.java:4041)
> org.eclipse.swt.widgets.Display.readAndDispatch(Display.java:3660)
> org.eclipse.ui.internal.Workbench.runEventLoop(Workbench.java:2640)
> org.eclipse.ui.internal.Workbench.runUI(Workbench.java:2604)
> org.eclipse.ui.internal.Workbench.access$4(Workbench.java:2438)
> org.eclipse.ui.internal.Workbench$7.run(Workbench.java:671)
> org.eclipse.core.databinding.observable.Realm.runWithDefault(Realm.java:3=
32)
> org.eclipse.ui.internal.Workbench.createAndRunWorkbench(Workbench.java:66=
4)
> org.eclipse.ui.PlatformUI.createAndRunWorkbench(PlatformUI.java:149)
> org.eclipse.ui.internal.ide.application.IDEApplication.start(IDEApplicati=
on.java:115)
> org.eclipse.equinox.internal.app.EclipseAppHandle.run(EclipseAppHandle.ja=
va:196)
> org.eclipse.core.runtime.internal.adaptor.EclipseAppLauncher.runApplicati=
on(EclipseAppLauncher.java:110)
> org.eclipse.core.runtime.internal.adaptor.EclipseAppLauncher.start(Eclips=
eAppLauncher.java:79)
> org.eclipse.core.runtime.adaptor.EclipseStarter.run(EclipseStarter.java:3=
69)
> org.eclipse.core.runtime.adaptor.EclipseStarter.run(EclipseStarter.java:1=
79)
```

```
> sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
> sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java=
:39)
> sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorI=
mpl.java:25)
> java.lang.reflect.Method.invoke(Method.java:597)
> org.eclipse.equinox.launcher.Main.invokeFramework(Main.java:620)
> org.eclipse.equinox.launcher.Main.basicRun(Main.java:575)
> org.eclipse.equinox.launcher.Main.run(Main.java:1408)
> org.eclipse.equinox.launcher.Main.main(Main.java:1384)
>
> ---------------------------------------------------------------------=
---------------
>
> Name: RMI TCP Connection(10)-127.0.0.1
> State: WAITING on org.eclipse.swt.widgets.RunnableLock@1f6855ac
> Total blocked: 7.945 =A0Total waited: 7.932
>
> Stack trace:
> java.lang.Object.wait(Native Method)
> java.lang.Object.wait(Object.java:485)
> org.eclipse.swt.widgets.Synchronizer.syncExec(Synchronizer.java:186)
> org.eclipse.ui.internal.UISynchronizer.syncExec(UISynchronizer.java:150)
> org.eclipse.swt.widgets.Display.syncExec(Display.java:4584)
> org.eclipse.swtbot.swt.finder.finders.UIThreadRunnable.run(UIThreadRunnab=
le.java:76)
> org.eclipse.swtbot.swt.finder.finders.UIThreadRunnable.syncExec(UIThreadR=
unnable.java:142)
> org.eclipse.swtbot.swt.finder.utils.SWTUtils.getText(SWTUtils.java:114)
> org.eclipse.swtbot.swt.finder.utils.SWTUtils.toString(SWTUtils.java:156)
> org.eclipse.swtbot.swt.finder.utils.SWTUtils.toString(SWTUtils.java:145)
> org.eclipse.swtbot.swt.finder.utils.WidgetTextDescription.describeTo(Widg=
etTextDescription.java:33)
> org.hamcrest.BaseDescription.appendDescriptionOf(BaseDescription.java:21)
> org.hamcrest.StringDescription.toString(StringDescription.java:28)
> org.eclipse.swtbot.swt.finder.widgets.AbstractSWTBot.toString(AbstractSWT=
Bot.java:322)
> java.text.MessageFormat.subformat(MessageFormat.java:1246)
> java.text.MessageFormat.format(MessageFormat.java:836)
> java.text.Format.format(Format.java:140)
> org.eclipse.swtbot.swt.finder.utils.MessageFormat.toString(MessageFormat.=
java:54)
> org.apache.log4j.or.DefaultRenderer.doRender(DefaultRenderer.java:36)
> org.apache.log4j.or.RendererMap.findAndRender(RendererMap.java:80)
```

```
> org.apache.log4j.spi.LoggingEvent.getRenderedMessage(LoggingEvent.java:36=
2)
> org.apache.log4j.helpers.PatternParser$BasicPatternConverter.convert(Patt=
ernParser.java:403)
> org.apache.log4j.helpers.PatternConverter.format(PatternConverter.java:65=
)
> org.apache.log4j.PatternLayout.format(PatternLayout.java:502)
> org.apache.log4j.WriterAppender.subAppend(WriterAppender.java:302)
> org.apache.log4j.WriterAppender.append(WriterAppender.java:160)
> org.apache.log4j.AppenderSkeleton.doAppend(AppenderSkeleton.java:251)
> =A0 - locked org.apache.log4j.ConsoleAppender@3887c7e
> org.apache.log4j.helpers.AppenderAttachableImpl.appendLoopOnAppenders(App=
enderAttachableImpl.java:66)
> org.apache.log4j.Category.callAppenders(Category.java:206)
> =A0 - locked org.apache.log4j.spi.RootLogger@4c1a841e
> org.apache.log4j.Category.forcedLog(Category.java:391)
> org.apache.log4j.Category.debug(Category.java:260)
> org.eclipse.swtbot.swt.finder.widgets.SWTBotMenu.click(SWTBotMenu.java:61=
)
> de.fu_berlin.inf.dpp.stf.server.rmi.remotebot.widget.impl.RemoteBotMenu.c=
lick(RemoteBotMenu.java:58)
> de.fu_berlin.inf.dpp.stf.server.rmi.superbot.component.contextmenu.peview=
.impl.ContextMenusInPEView.open(ContextMenusInPEView.java:75)
> sun.reflect.GeneratedMethodAccessor65.invoke(Unknown Source)
> sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorI=
mpl.java:25)
> java.lang.reflect.Method.invoke(Method.java:597)
> sun.rmi.server.UnicastServerRef.dispatch(UnicastServerRef.java:305)
> sun.rmi.transport.Transport$1.run(Transport.java:159)
> java.security.AccessController.doPrivileged(Native Method)
> sun.rmi.transport.Transport.serviceCall(Transport.java:155)
> sun.rmi.transport.tcp.TCPTransport.handleMessages(TCPTransport.java:535)
> sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run0(TCPTransport.ja=
va:790)
> sun.rmi.transport.tcp.TCPTransport$ConnectionHandler.run(TCPTransport.jav=
a:649)
> java.util.concurrent.ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor=
.java:886)
> java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.jav=
a:908)
> java.lang.Thread.run(Thread.java:662)
>
> After investigating the code of the
> org.eclipse.swtbot.swt.finder.utils.MessageFormat class i found out that =
```

```
the
> string creation for the debug message is performed lazy.
>
> So the following scenario holds:
>
> Main-Thread (EDT): runs some code
> SWTBOT-Thread: begins to log a message and acquire the lock on the logger
> Main-Thread (EDT): tries to log a message and is blocked on the current l=
ock
> of the logger
> SWTBOT-Thread: is running a Display.(A)Sync to gather the information for
> the string to be logged.
> SWTBOT-Thread: is blocked by Main-Thread (EDT) which is blocked by
> SWTBOT-Thread:
>
> DEADLOCK
>
> The current solution is to turn off logging but that is hitting us hard
> without having any trace files.
>
> I hope you have the time to investigate this problem and would fix it.
>
> Best regards,
> Stefan Rossbach
>
```

B STF Skripte

B.1 Metaskript

```
#!/usr/bin/lua

--[[
   This script depends on some hard coded pathes in
      the script start_eclipse.sh
   Lua manual can be found at: http://www.lua.org/
      manual/5.1/
   Author: Stefan Rossbach <rossbach@inf>

   Currently the firewall does not work
  ]]

local host_to_user_mapping =
{
```

```
    ["192.168.66.129"] = "saros-eclipse",
    ["192.168.66.130"] = "saros-eclipse"
}

local supported_emulators =
{
    ["none"]           = true,
    ["802.11a"]        = true,
    ["802.11b"]        = true,
    ["802.11g"]        = true,
    ["adsl"]           = true,
    ["sdsl"]           = true,
    ["highpacketloss"] = true
}


local function to_remote_ssh_command(user, host,
    sshkey, command)
    return "ssh -n -i " .. sshkey .. " " .. user .."@"
        .. host .. " \"" .. command .. "\""
end

local function to_scp_command(user, host, sshkey, from
    , to)
    return "scp -o BatchMode=yes -i " .. sshkey .. " "
        .. user .."@" .. host .. ":\"" .. from .. "\""
        .. " \"" .. to .. "\""
end

local function e_print(t)
    io.stderr:write(t)
    io.stderr:write("\n")
    io.stderr:flush()
end

local function usage()
    e_print("Usage: stf_script_gen [options] sshprivkey
        pluginfile configfile")
    e_print("")
    e_print("   --cobertura <cobertura_dir>  directory
       to cobertura if code coverage is desired, plugin
        file must be already instrumented")
    -- theses options are not implemented yet
    e_print("   --sniffer                    Start a
       sniffer on the gateway. You find the file /tmp/
       packet.pcap on the gateway")
    e_print("   --emulator <name>            Use given
       emulator configuration")
    os.exit(1)
```

```
end

local function parse_config_file(config)
   local client, host, port
   local t = {}

   for line in io.lines(config) do
      client, host = line:match("%s-([^_]+)_HOST%s-=%s
         -([%w%p]+).-")

      if client and host then
         if not t[client] then t[client] = {} end
         t[client].host = host;
      end

      client, port = line:match("%s-([^_]+)_PORT%s-=%s
         -([%d]+).-")

      if client and port then
         if not t[client] then t[client] = {} end
         t[client].port = port;
      end
   end

   return t
end

local function process_args()

   local t, i, j = {}, 1, 0

   while i <= table.getn(arg) do

      local a = arg[i]

      if a == "--sniffer" then
         t.sniffer, i = true , i + 1
      elseif a=="--emulator" then
         t.emulator, i = arg[i+1], i + 2
      elseif a =="--cobertura" then
         t.cobertura, i = arg[i+1], i + 2
      elseif a:sub(1, 2) ~= "--" then
         if j == 0 then
            t.sshkey = arg[i]
         elseif j == 1 then
            t.plugin = arg[i]
         elseif j == 2 then
            t.config = arg[i]
         end
```

```
            i, j = i + 1, j + 1
        else
            i = i + 1
        end
    end

    return t
end

if table.getn(arg) == nil or arg[1] == "--help" then
    usage() end

local args = process_args()

if args.emulator and (not supported_emulators[args.
    emulator]) then e_print("error, unknown emulator
    type: " .. args.emulator) os.exit(1) end

if not (args.sshkey and args.plugin and args.config)
    then usage() end

local config = parse_config_file(args.config)

local hosts = {}

for _, v in pairs(config) do
    if v.port == nil or v.host == nil then e_print("
        error in config file, host or port is missing
        for: " .. k) os.exit(1) end
    hosts[v.host] = host_to_user_mapping[v.host]
    if hosts[v.host] == nil then e_print ("error, the
        host " .. v.host .. " is not known") os.exit(1)
        end
end

print("#!/usr/bin/sh")

print("function wait_until_timeout ()")
print("{")
print("POLL=$1")
print("DELAY=$2")
print("PID_TO_WAIT_FOR=$3")
print("while [ $POLL -gt 0 ]")
print("do")
print("sleep $DELAY")
print("POLL=$(( $POLL - 1 ))")
print("kill -0 $PID_TO_WAIT_FOR > /dev/null 2>&1")
print("STATUS=$?")
```

```
print("if [ $STATUS -ne 0 ]")
print("then")
print("return 0")
print("fi")
print("done")
print("return 1")
print("}")


print("function stop_vnc_server ()")
print("{")
print("echo
   \"################################################\"")
print("echo \"#             STOP VNC SERVERS
                  #\"")
print("echo
   \"################################################\"")

for host, user in pairs(hosts) do
   for i = 1, 9 do
  print(to_remote_ssh_command(user, host, args.sshkey,
      "vncserver -kill :" .. i))
  print(to_remote_ssh_command(user, host, args.sshkey,
      "rm -rf /tmp/.X" .. i .. "-lock"))
  print(to_remote_ssh_command(user, host, args.sshkey,
      "rm -rf /tmp/.X11-unix/X" .. i))
   end
end
print("}")


print("function start_vnc_server ()")
print("{")
print("echo
   \"################################################\"")
print("echo \"#             START VNC SERVERS
                #\"")
print("echo
   \"################################################\"")

for host, user in pairs(hosts) do
   for _, net in pairs(config) do
  if host == net.host then print(to_remote_ssh_command
     (user, host, args.sshkey, "vncserver")) end
   end
end
print("}")
```

```
print("function start_sniffer ()")
print("{")
print("echo
   \"################################################\"")
print("echo \"#            START SNIFFER
                    #\"")
print("echo
   \"################################################\"")
print("}")


print("function configure_emulator ()")
print("{")
print("echo
   \"################################################\"")
print("echo \"#            CONFIGURE EMULATOR
               #\"")
print("echo
   \"################################################\"")
print("}")

print("function deploy ()")
print("{")
print("echo
   \"################################################\"")
print("echo \"#                  DEPLOY
                      #\"")
print("echo
   \"################################################\"")
for host, user in pairs(hosts) do
   local stf_scripts = os.getenv("JENKINS_HOME") .. "/
      scripts/stf"
   print(to_remote_ssh_command(user, host, args.sshkey
      , "rm -rf /home/" .. user .. "/plugins"))
   print(to_remote_ssh_command(user, host, args.sshkey
      , "rm -rf /home/" .. user .. "/bin"))
   print(to_remote_ssh_command(user, host, args.sshkey
      , "mkdir -p /home/" .. user .. "/plugins"))
   print(to_remote_ssh_command(user, host, args.sshkey
      , "mkdir -p /home/" .. user .. "/bin"))
   print("rsync -v -e 'ssh -i " .. args.sshkey .. "'"
      .. " -a " .. args.plugin .. " " .. user .. "@"
      .. host .. ":plugins/")
   print("rsync -v -e 'ssh -i " .. args.sshkey .. "'"
      .. " -a " .. stf_scripts .. "/start_eclipse.sh"
      .. " " .. user .. "@" .. host .. ":bin/")
   print("rsync -v -e 'ssh -i " .. args.sshkey .. "'"
      .. " -a " .. stf_scripts .. "/stop_eclipse.sh"
      .. " " .. user .. "@" .. host .. ":bin/")
```

```
   if args.cobertura then
      print(to_remote_ssh_command(user, host, args.
         sshkey, "rm -rf /home/" .. user .. "/
         cobertura"))
      print(to_remote_ssh_command(user, host, args.
         sshkey, "mkdir -p /home/" .. user .. "/
         cobertura"))
      print("rsync -v -e 'ssh -i " .. args.sshkey .. "
         '" .. " -a " .. args.cobertura .. "/ " ..
         user .. "@" .. host .. ":cobertura/")
   end
end

print("}")

print("function start_remote_bots ()")
print("{")
print("echo
   \"################################################\"")
print("echo \"#              START REMOTE BOTS
                #\"")
print("echo
   \"################################################\"")

local display_numbers = {}
for saros_user, net in pairs(config) do
   display_numbers[net.host] = 1
end

for saros_user, net in pairs(config) do
   local current_display_number = display_numbers[net.
      host]
   print(to_remote_ssh_command(host_to_user_mapping[
      net.host], net.host, args.sshkey, "./bin/
      start_eclipse.sh :" .. current_display_number ..
       " " .. saros_user .. " "  .. net.port) .. " > "
       .. saros_user:lower() .. ".log 2>&1 &");
   print("echo $! >> ssh.pids")
   print("# SLEEP TO AVOID HIGH LOAD ON THE MACHINES
      AND ECLIPSE INITIALIZATION ERRORS")
   print("sleep 30")
   display_numbers[net.host] = current_display_number
      + 1
end
print("}")

print("function stop_remote_bots ()")
print("{")
```

```
print("echo
   \"##############################################\"")
print("echo \"#              STOP REMOTE BOTS
                #\"")
print("echo
   \"##############################################\"")
for saros_user, net in pairs(config) do
   print(to_remote_ssh_command(host_to_user_mapping[
      net.host], net.host, args.sshkey, "./bin/
      stop_eclipse.sh " .. "${1}" .. " " .. saros_user
       .. " "  .. net.port));
end
print("}")

print("function kill_ssh_connections ()")
print("{")
print("cat ssh.pids | while read LINE")
print("do")
print("kill -SIGKILL $LINE")
print("done")
print("}")

print("function fetch_code_coverage ()")
print("{")
print("echo
   \"##############################################\"")
print("echo \"#           FETCH CODE COVERAGE
                #\"")
print("echo
   \"##############################################\"")

if args.cobertura then
   for saros_user, net in pairs(config) do
      print(to_scp_command(host_to_user_mapping[net.
         host], net.host, args.sshkey, "/home/" ..
         host_to_user_mapping[net.host] .. "/
         workspace_" .. saros_user .. "/coverage_" ..
         saros_user .. ".ser", "coverage_" ..
         saros_user .. ".ser"))
   end
end

print("}")

print("function fetch_screen_shots ()")
print("{")
print("echo
   \"##############################################\"")
```

```
print("echo \"#             FETCH SCREENSHOTS
                #\"")
print("echo
   \"##############################################\"")

if args.cobertura then
   for saros_user, net in pairs(config) do
      print(to_remote_ssh_command(host_to_user_mapping
         [net.host], net.host, args.sshkey, "cd /home/
         " .. host_to_user_mapping[net.host] .. "/
         workspace_" .. saros_user .. "/.metadata &&
         tar czvf " .. saros_user .. "screen_shots.tar
         .gz saros_screenshots"))
      print(to_scp_command(host_to_user_mapping[net.
         host], net.host, args.sshkey, "/home/" ..
         host_to_user_mapping[net.host] .. "/
         workspace_" .. saros_user .. "/.metadata/" ..
          saros_user .. "screen_shots.tar.gz",
         saros_user .. "screen_shots.tar.gz"))
      print("tar xzvf " .. saros_user .. "screen_shots
         .tar.gz")
   end
end

print("}")

print("echo
   \"##############################################\"")
print("echo \"#     this script was auto generated by
       #\"")
print("echo \"#
                                              #\"")
print("echo \"#             stf_script_gen
                  #\"")
print("echo \"#
                                              #\"")
print("echo
   \"##############################################\"")
```

B.2 Eclipse Startskript

```
#! /bin/sh

# params DISPLAY, USER(e.g: Alice, Bob), PORT

set -x

DISPLAY=$1
```

```
USER=$2
RMI_PORT=$3

JAVA=`/usr/bin/which java`

if [ ! -x $JAVA ]; then
  echo "error, no java executable found"
  exit 1
fi

HOSTIP=`/sbin/ifconfig eth1 | grep "inet addr" | cut -
    d ":" -f 2 | cut -d " " -f 1`

ECLIPSE_DIR="${HOME}/eclipse"
ECLIPSE_PLUGIN_DIR="${ECLIPSE_DIR}/plugins"

WORKSPACE="${HOME}/workspace_${USER}"

PLUGIN_ID_PREFIX="de.fu_berlin.inf.dpp"
SAROS_PLUGIN_DIR="${HOME}/plugins"

# determine (versioned) filename of plugin (_
    suppresses .source versions)
SAROS_PLUGIN_FILENAME=`ls -1 $SAROS_PLUGIN_DIR | grep
    ${PLUGIN_ID_PREFIX}_*`

if [ -z $SAROS_PLUGIN_FILENAME ]; then
  echo "cannot find plugin with prefix
      $PLUGIN_ID_PREFIX in $SAROS_PLUGIN_DIR"
  exit 1
fi

echo "deleting workspace: ${WORKSPACE}"
rm -rf "${WORKSPACE}"

if [ ! -e "${SAROS_PLUGIN_DIR}/lock}" ]; then
  touch "${SAROS_PLUGIN_DIR}/lock}"
  echo "deleting old Saros plugin(s)"
  rm -f "${ECLIPSE_PLUGIN_DIR}/de.fu_berlin.inf.dpp"*
  echo "installing Saros plugin $SAROS_PLUGIN_FILENAME
      "
  cp "${SAROS_PLUGIN_DIR}/${SAROS_PLUGIN_FILENAME}" "$
      {ECLIPSE_PLUGIN_DIR}"
fi

mkdir -p "${WORKSPACE}"
mkdir -p "${WORKSPACE}/.metadata/.plugins/org.eclipse.
    core.runtime/.settings"
```

```
# enable auto close of eclipse

echo EXIT_PROMPT_ON_CLOSE_LAST_WINDOW=false > "${
    WORKSPACE}/.metadata/.plugins/org.eclipse.core.
    runtime/.settings/org.eclipse.ui.ide.prefs"

# keep SVN quite
echo ask_user_for_usage_report_preference=false > "${
    WORKSPACE}/.metadata/.plugins/org.eclipse.core.
    runtime/.settings/org.tigris.subversion.subclipse.
    tools.usage.prefs"

echo "grant{\\npermission java.security.AllPermission
    ;\\n};" > "${WORKSPACE}/stf.policy"

# get path to equinox jar inside eclipse home folder

CLASSPATH=$(find "${ECLIPSE_PLUGIN_DIR}" -name "org.
    eclipse.equinox.launcher_*.jar" | sort | tail -1);

CLASSPATH="${CLASSPATH}:${HOME}/cobertura/cobertura.
    jar"

LD_LIBRARY_PATH="/usr/lib/jni:${LD_LIBRARY_PATH}"

export LD_LIBRARY_PATH
export DISPLAY
export CLASSPATH

echo "starting Eclipse for user ${USER}"

$JAVA -version

# be sure to set -Dosgi.parentClassloader=app
    otherwise instrumented classes would throw a class
    not found exception

$JAVA \
-XX:MaxPermSize=192m -Xms384m -Xmx512m -ea \
-Djava.rmi.server.codebase="file:${
    SAROS_PLUGIN_FILENAME}" \
-Djava.security.manager \
-Djava.security.policy="file:${WORKSPACE}/stf.policy"
    \
-Djava.rmi.server.hostname="${HOSTIP}" \
-Dde.fu_berlin.inf.dpp.testmode="${RMI_PORT}" \
-Dde.fu_berlin.inf.dpp.sleepTime=200 \
-Dorg.eclipse.swtbot.keyboard.strategy=org.eclipse.
    swtbot.swt.finder.keyboard.MockKeyboardStrategy \
```

```
-Dorg.eclipse.swtbot.keyboard.layout=de.fu_berlin.inf.
    dpp.stf.server.bot.default \
-Dfile.encoding=UTF-8 \
-Dnet.sourceforge.cobertura.datafile="${WORKSPACE}/
    coverage_${USER}.ser" \
-Dosgi.parentClassloader=app \
org.eclipse.equinox.launcher.Main \
-name "eclipse_${USER}" \
-clean \
-consoleLog \
-data "${WORKSPACE}"
```

B.3 Eclipse Stopskript

```
#! /bin/sh

set -x

# params KILL_MODE USER(e.g: Alice, Bob), PORT

PID=`ps aux | grep -v grep | grep testmode=${3}.*
    workspace_${2} | sed 's/ \+/ /g' | cut -d " "  -f
    2`

kill -${1} ${PID}
```

C Ant Build Dateien

C.1 Plugins

```
<?xml version="1.0"?>

<project name="build.plugin" basedir="." default="
    build" xmlns:ant4eclipse="antlib:org.ant4eclipse">

  <!-- define ant-contrib macros -->
  <taskdef resource="net/sf/antcontrib/antcontrib.
      properties"/>

  <!-- define ant4eclipse tasks -->
  <taskdef uri="antlib:org.ant4eclipse" resource="org/
      ant4eclipse/antlib.xml" />

  <!-- import the ant4eclipse pde macros -->
  <import file="${ant4eclipse.dir}/macros/a4e-pde-
      macros.xml" />
```

```
  <!-- define the workspace location here -->
  <property name="workspaceDirectory" value="${basedir
     }/.." />

  <property name="src.dir" value="${basedir}/src" />
  <property name="build.dir" value="${basedir}/build"
     />

  <ant4eclipse:jdtClassPathLibrary name="org.eclipse.
     jdt.junit.JUNIT_CONTAINER/4">
      <fileset dir="${eclipse.plugin.dir}">
      <include name="**/junit.jar"/>
      </fileset>
  </ant4eclipse:jdtClassPathLibrary >

  <!-- (define eclipse.plugin.dir using -D syntax) the
      target platform
        location (can be an Eclipse plugin dir or
           another set of JARs
        satisfying the Saros platform dependencies) --
           >
  <ant4eclipse:targetPlatform id="saros.target.
     platform">
    <location dir="${eclipse.plugin.dir}" />
    <location dir="${additional.plugin.dir}" />
  </ant4eclipse:targetPlatform>

  <!-- Targets -->
  <target name="build">
    <delete dir="${build.dir}" />
    <mkdir dir="${build.dir}" />
    <buildPlugin workspaceDirectory="${
       workspaceDirectory}" projectName="${plugin.name
       }" targetplatformid="saros.target.platform"
       destination="${build.dir}" />
  </target>
</project>
```

C.2 Junit Test

```
<project name="saros.test" basedir="." default="test">

  <path id="cobertura.classpath">
    <fileset dir="${cobertura.dir}">
      <include name="cobertura.jar" />
      <include name="lib/**/*.jar" />
    </fileset>
```

```
  </path>

  <taskdef classpathref="cobertura.classpath" resource
     ="tasks.properties" />

  <target name="instrument">
    <delete file="${cobertura.datafile}" />
    <mkdir dir="${instrumented.dir}" />
    <cobertura-instrument todir="${instrumented.dir}"
       datafile="${cobertura.datafile}">

      <!-- only instrument Saros proper code -->
      <includeClasses regex="de\.fu_berlin\.inf\.dpp.*
         " />

      <!-- exclude classes ending in Test -->
      <excludeClasses regex=".*Test" />

      <!-- exclude static classes in Test classes -->
      <excludeClasses regex=".*Test\$.*" />

      <!-- exclude classes starting with Test -->
      <excludeClasses regex=".*\.Test.*" />

      <!-- exclude packages container test -->
      <excludeClasses regex=".*\.test\..*" />

      <!-- exclude classes containing TestSuite -->
      <excludeClasses regex=".*TestSuite.*" />

      <!-- exclude stf classes  -->
      <excludeClasses regex=".*\.stf\..*" />

      <fileset dir=".">
        <include name="de.fu_berlin.inf.dpp_*.jar" />
      </fileset>
    </cobertura-instrument>
  </target>


  <target name="test" depends="instrument">
    <mkdir dir="${junit.dir}" />
    <mkdir dir="${src.dir}" />

    <junit printsummary="yes" fork="yes" forkmode="
       perBatch">
      <sysproperty key="de.fu_berlin.dpp.test.xmpp.
         useExternalServer" value="true" />
```

```
      <sysproperty key="de.fu_berlin.dpp.test.xmpp.
         serverAddress" value="saros-eclipse1" />
      <sysproperty key="net.sourceforge.cobertura.
         datafile" value="${cobertura.datafile}" />
      <classpath>

        <!-- include the test lib directory first
           because of some library issues -->
        <fileset dir="${test.lib.dir}" />
        <fileset dir="${lib.dir}" />

        <fileset dir="${instrumented.dir}">
          <include name="de.fu_berlin.inf.dpp_*.jar" /
             >
        </fileset>
        <fileset dir="${eclipse.dir}/plugins">
          <include name="*.jar" />
        </fileset>

      </classpath>

      <classpath refid="cobertura.classpath" />

      <formatter type="xml" />

      <batchtest todir="${junit.dir}">
        <fileset dir="${src.dir}">
          <exclude name="**/*PluginTest.java"/>
          <exclude name="**/stf/**"/>
          <include name="**/*Test.java"/>
        </fileset>
      </batchtest>
    </junit>
    <cobertura-report format="xml" datafile="${
       cobertura.datafile}" destdir="${report.dir}"
        srcdir="${src.dir}" />
  </target>
</project>
```

D STF Testfall

```
package de.fu_berlin.inf.dpp.stf.test.
   rosterviewbehaviour;

import static de.fu_berlin.inf.dpp.stf.client.tester.
   SarosTester.ALICE;
```

```
import static de.fu_berlin.inf.dpp.stf.client.tester.
    SarosTester.BOB;
import static de.fu_berlin.inf.dpp.stf.client.tester.
    SarosTester.CARL;
import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;

import java.rmi.RemoteException;
import java.util.List;

import org.junit.BeforeClass;
import org.junit.Test;

import de.fu_berlin.inf.dpp.stf.annotation.TestLink;
import de.fu_berlin.inf.dpp.stf.client.StfTestCase;

@TestLink(id = "Saros-117
    _sort_online_buddies_over_offline")
public class SortBuddiesOnlineOverOfflineTest extends
    StfTestCase {

    @BeforeClass
    public static void selectTesters() throws
        Exception {
        select(ALICE, CARL, BOB);
    }

    @Test
    public void testSortBuddiesOnlineOverOffline()
        throws RemoteException {

        // wait for roster update
        ALICE.remoteBot().sleep(1000);

        List<String> buddies = ALICE.superBot().views
            ().sarosView()
            .getAllBuddies();

        assertTrue("corrupted size on roster", buddies
            .size() >= 2);
        assertEquals(BOB.getBaseJid(), buddies.get(0))
            ;
        assertEquals(CARL.getBaseJid(), buddies.get(1)
            );

        checkBuddiesOrder(buddies, 2);

        BOB.superBot().views().sarosView().disconnect
            ();
```

```
        BOB.superBot().views().sarosView().
            waitUntilIsDisconnected();

        // wait for roster update
        ALICE.remoteBot().sleep(1000);

        buddies = ALICE.superBot().views().sarosView()
            .getAllBuddies();

        assertTrue("corrupted size on roster", buddies
            .size() >= 2);
        assertEquals(CARL.getBaseJid(), buddies.get(0)
            );

        checkBuddiesOrder(buddies, 1);

        BOB.superBot().views().sarosView()
             .connectWith(BOB.getJID(), BOB.getPassword
                ());
        BOB.superBot().views().sarosView().
            waitUntilIsConnected();

        // wait for roster update
        ALICE.remoteBot().sleep(1000);

        buddies = ALICE.superBot().views().sarosView()
            .getAllBuddies();

        assertTrue("corrupted size on roster", buddies
            .size() >= 2);
        assertEquals(BOB.getBaseJid(), buddies.get(0))
            ;
        assertEquals(CARL.getBaseJid(), buddies.get(1)
            );

        checkBuddiesOrder(buddies, 2);

        BOB.superBot().views().sarosView().disconnect
            ();
        CARL.superBot().views().sarosView().disconnect
            ();

        // wait for roster update
        ALICE.remoteBot().sleep(1000);

        buddies = ALICE.superBot().views().sarosView()
            .getAllBuddies();
```

```
        assertTrue("corrupted size on roster", buddies
            .size() >= 2);
        checkBuddiesOrder(buddies, 0);

    }

    private void checkBuddiesOrder(List<String>
        buddies, int s) {
        for (int i = s; i < buddies.size() - 1; i++)
            assertTrue("roster is not sorted asc. : "
                + buddies.get(i) + " > "
                + buddies.get(i + 1),
                buddies.get(i).compareTo(buddies.get(i
                    + 1)) <= 0);
    }
}
```