Freie Universität Berlin

Diplomarbeit zur Erlangung des akademischen Grades Diplom-Informatiker

Verbesserung der Architektur der DPP-Software Saros durch Einführung einer dokumentierten Modulsicht

Fachbereich Mathematik und Informatik
Institut für Informatik
Arbeitsgruppe Software Engineering

Wjatscheslaw Belousow
Matrikelnummer: 3346330
belousow@inf.fu-berlin.de

Eingereicht bei: Prof. Dr. Lutz Prechelt
Zweitgutachter: Prof. Dr. Elfriede Fehr
Betreuer: Karl Beecher und Stephan Salinger

Berlin, 04. Juli 2011

Eidesstattliche Erklärung

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

Berlin, 04. Juli 2011 _____

Wjatscheslaw Belousow

„ Im Prozess der Entwicklung der DPP-Software Saros, welche in den letzten Jahren stark gewachsen ist, treten strukturelle Probleme auf, die die Arbeit an dem Programm erschweren. Im Rahmen meiner Diplomarbeit werde ich untersuchen, wie die Architektur der DPP-Software Saros verbessert werden kann. Der erste Teil der Arbeit beschäftigt sich mit der Analyse des aktuellen Quelltextes, der verwendeten Architektur, der Dokumentation, der Vorgehensweisen und der zu Verfügung stehenden Ressourcen.

Anhand der Ergebnisse wird im zweiten Teil ein Modell zu Verbesserung der Architektur und der dazugehörigen Dokumentation entworfen. Dabei werden Eigenschaften der vereinfachten Wartung, Implementierung neuer Features und der Einstieg von neuen Entwicklern besonders berücksichtigt. Im dritten Teil erfolgt eine Bewertung des Modells anhand einer szenariobasierten Architekturbewertung. “

Inhaltsverzeichnis

1 Einführung 10

1.1 Beschreibung des Themas ... 10
1.2 Vorgehensweise ... 10
1.3 Methode ... 11
1.4 Aufbau der Arbeit ... 11
1.5 Typographische Konventionen ... 11

2 Grundlagen 12

2.1 Saros ... 12
2.2 Softwarearchitektur ... 13
2.2.1 Sichten ... 13
2.2.2 Entwurf ... 14
2.2.3 Dokumentation ... 14
2.3 Software Architecture Analysis Method (SAAM) ... 15
2.4 OSGi ... 15
2.4.1 OSGi und Java 7 bzw. 8 ... 17
2.5 Dependency Injection ... 17
2.5.1 PicoContainer ... 18
2.6 XMPP ... 19
2.6.1 Smack ... 19

3 Saros Architektur 20

3.1 Architekturkonzepte im Saros-Projekt ... 20
3.1.1 Niedergeschriebenes Wissen ... 20
3.1.2 Quelltext ... 23
3.1.3 Wissen im Team ... 24
3.1.4 Wissen aus Prozessen und Werkzeuge ... 26
3.2 Aktuelle Saros Architektur ... 26
3.2.1 Komponentensicht ... 27
3.2.2 Programmaufbausicht ... 28
3.3 Probleme der aktuellen Saros Architektur ... 29
3.3.1 Historisch gewachsen ... 29
3.3.2 Strukturelle Probleme ... 29
3.3.3 Organisatorische Probleme ... 30
3.3.4 Probleme der Dokumentation ... 30
3.3.5 Zusammenfassung ... 31
3.4 Ansätze zu Verbesserung ... 32

4 Saros verbesserte Architektur 33

4.1 Abstraktion ... 33
4.2 Modularisierung ... 33
4.2.1 Basis ... 34
4.2.2 Basis Implementierung ... 35
4.2.3 Verwendung ... 38
4.3 Konstruktion und Konfiguration ... 39
4.4 ModuleDoc (Dokumentation) ... 40

5 Integration in Saros 42

5.1 Eine neue Fassade für Saros ... 42
5.1.1 Phase 1 - ISaros Schicht ... 43
5.1.2 Phase 2 - ICoreModule ... 45
5.1.3 Phase 3 - Saros-Builder ... 46
5.1.4 Phase 4 - Eclipse Plugin-Mechanismus abtrennen ... 47
5.1.5 Phase 1-4 - Zusammenfassung ... 51
5.2 Phase 5 - Inkrementelle Integration ... 51
5.2.1 Funktionalität identifizieren ... 52
5.2.2 Modulschicht definieren ... 53
5.2.3 Abhängigkeiten auflösen ... 53
5.3 Weitere Module ... 54
5.4 Besonderheiten des Saros-Projektes ... 55
5.4.1 Dependency Injection von Eclipse ... 55
5.4.2 Test ... 55
5.4.3 Portabilität ... 55

6 Bewertung 56

6.1 Vorgehen ... 56
6.2 Szenario 1 - Neuer Entwickler ... 56
6.3 Szenario 2 - Fehlerbehebung ... 57
6.4 Szenario 3 - neue Features ... 57
6.5 Szenario 4 - Tests ... 57
6.6 Szenario 5 - Portierung ... 57
6.7 Gesamtbewertung ... 58

7 Ausblick 59

7.1 Umsetzung ... 59
7.2 Einschränkungen des Designs ... 59
7.3 Umstellung auf OSGi oder das Modularisierungskonzept von Java 8 ... 59
7.4 ModuleDoc ... 59

A ModuleDoc 61

A.1 Struktur 61
A.2 Services, Objects und Values 63
A.3 Ressource 63
A.4 Implementierungen 64
A.5 Zusammenfassen der Beschreibung mehrerer Module 64
A.6 Textgestaltung 65
A.7 Darstellung im Browser 65

B Modul Beschreibungen 66

B.1 ICoreModule 66
B.2 IAccountManagerModule 67

Quellenangaben 69

1 Einführung

1.1 Beschreibung des Themas

Als ich meine Diplomarbeit im Rahmen des Saros-Projektes angefangen habe, stand noch nicht fest in welche Richtung es am Ende gehen würde. Das Saros-Projekt hatte Probleme mit der Quelltextbasis, mit Seiteneffekten während der Fehlerbehebung und der Integration von neuen Features. Über mehrere Jahre intensiver Entwicklung ist das System stark und schnell gewachsen, wobei strukturelle Probleme auftraten, welche die Weiterentwicklung erschweren. So war die grundlegende Frage, wie und in welcher Art Saros in dieser Hinsicht verbessert werden kann.

Meine Motivation für diese Arbeit bestand darin, ein bestehendes größeres Softwareprojekt zu untersuchen, mögliche Probleme aufzudecken und Lösungsvorschläge dafür zu erarbeiten. Ich hatte schon durch andere fremde und eigene Projekte die Erfahrung gemacht, dass ab einer gewissen Größe oder Zeitspanne die Entwicklung sich verlangsamt und verschiedene Probleme auftreten, an die vorher gar nicht gedacht wurde. Jede kleine Änderung produziert weitere Probleme, nach deren Behebung wieder was an anderer Stelle nicht geht. Die Motivation an so einem Projekt zu arbeiten sinkt und am Ende kann es sogar dazu führen, dass es über längere Zeit brach liegt, gar nicht vorankommt und irgendwann alle vergessen haben, wie es eigentlich von innen aussieht. Dazu fällt mir ein Satz ein, welchen ich mal gelesen habe (leider weis ich nicht mehr wo): „zuerst entwickeln wir Software, danach beten wir diese an“ :)

Das Ziel der Arbeit war eine Möglichkeit zu Verbesserung von Saros zu untersuchen, zu beschreiben und eventuell auch zu realisieren.

1.2 Vorgehensweise

Ich habe schon zuvor die Erfahrung gemacht, dass Probleme nicht nur am Quelltext liegen. Es ist immer das ganze Projekt mit allen Beteiligten und Prozessen, welches über den Erfolg einer Software entscheidet. So habe ich von Anfang an versucht mich in das Projekt zu integrieren, die Arbeitsweise zu verstehen, die Abläufe zu verfolgen, selber daran teilzunehmen und so Informationen zum Projekt und den Problemen zu sammeln.

Schon am Anfang habe ich bemerkt, dass es Saros an Projektdokumentation mangelt. Das meiste Wissen wird von Person zu Person weitergegeben. Eine andere Besonderheit des Saros-Projektes ist, dass es bereits von vielen Personen entwickelt wurde, die für eine begrenzte Zeit am Projekt gearbeitet haben. So könnte man sagen, dass an Saros schon mehrere Generationen von Entwicklern dran waren.

Eine andere Eigenschaft von Saros ist, dass Fehler innerhalb einer überschaubaren Zeit gefunden und behoben werden können. Besonders dank der automatischen Tests, welche in den letzten Monaten massiv verbessert wurden. So ist der Quelltext selber gut zu lesen.

Doch wenn es um größere Änderungen am Design der Software geht, wird es schwierig. Es fehlt einfach ein vereinfachter und aktueller Blick auf die Software, wie aus einer Vogelperspektive.

Ich hatte viele Ansätze für die Verbesserung der Software angesehen. Darunter waren eine Modularisierung von Saros, eine umfassende Dokumentation des Systems und die Verbesserung der automatischen Prozesse während der Entwicklung (Continuous Integration[1]). Ich hatte die Möglichkeit während meiner Arbeit die BerlinExpertDays (BED)[2] Konferenz zu besuchen, bei der Continuous Integration ein Schwerpunkt war. Dort habe ich auch einen interessanten Vortrag zu historisch gewachsenen Architekturen gehört und wie diese dokumentieren werden können[19].

1.3 Methode

An Ende habe ich mich auf die Modularisierung von Saros konzentriert, genauer auf die Einführung einer **dokumentierten Modulsicht**. Dazu habe ich eine neue Architektur für Saros entworfen und versucht diesen mit dem „Software Architecture Analysis Method (SAAM)“ (Kapitel 2.3) Verfahren zu bewerten. Ich habe verschiedene Szenarios entwickelt und beschrieben wie die verbesserte Architektur darauf reagieren kann.

1.4 Aufbau der Arbeit

In Kapitel 2 werden die Grundlagen beschrieben, auf welchen diese Arbeit aufbaut. In Kapitel 3 wird die aktuelle Saros Architektur untersucht und beschrieben. Kapitel 4 widmet sich dem Design einer verbesserten Architektur, welche in Kapitel 5 teilweise in Saros integriert wird. In Kapitel 6 erfolgt eine Bewertung dieser Architektur anhand des SAAM-Verfahrens. Kapitel 7 schließt die Arbeit mit einem Ausblick ab.

1.5 Typographische Konventionen

Mit **hervorgehoben** wurden wichtige Begriffe markiert, welche eine Rolle in dem jeweiligen Abschnitt spielen. Mit *kursiv* werden Eigennamen dargestellt. Elemente eines Quelltextes im Fließtext wurden mit einer `nichtproportionale Schriftart` geschrieben. Der Einfachheit wegen habe ich in der Arbeit den `de.fu_berlin.inf.` Teil eines Saros Java-Packages weggelassen, die Packages beginnen mit `dpp`. Bei der Erstellung der UML Diagramme habe ich mich an „UML 2.0 - kurz & gut“[13] orientiert, mit einer Ausnahme. Obwohl `abstract` nach der UML-Definition kein Stereotyp ist, habe ich es als Stereotyp verwendet, wegen der Einfachheit und Symmetrie.

[1] ein Prozess bei dem die Software immer wieder neu erstellt, getestet und release-fertig aufbereitet wird

[2] https://bed-con.org/

2 Grundlagen

2.1 Saros

Saros[3] ist eine Erweiterung für die Eclipse Plattform (Plugin), welche es mehreren Anwendern, an verschiedenen Rechnern, ermöglicht gemeinsam an einem Text zu arbeiten. Innerhalb von Sitzungen können mehrere Teilnehmer gleichzeitig am selben Projekt arbeiten, Saros synchronisiert alle Änderungen im Hintergrund. Primär wird es für die Unterstützung der Softwareentwicklung eingesetzt, doch damit können auch andere Arten von textbasierten Dokumenten gemeinsam bearbeitet werden. Saros verwendet das sogenannte Distributed Party Programming (DPP) Verfahren, deshalb auch der Name DPP-Software Saros, welches auf dem Konzept der Paarprogrammierung (PP) berührt. Entwickelt wurde Saros 2006 von Riad Djemili[20], in Rahmen seiner Diplomarbeit. Inzwischen wurde es durch zahlreiche weitere Abschluss-/Studienarbeiten erweitert und verbessert.

Paarprogrammierung kommt aus der Welt der Agilen Softwareentwicklung([6], S. 70). Dabei arbeiten zwei Entwickler gemeinsam an der selben Aufgabe und an dem selben Rechner zusammen. Während eine Person die Maus und Tastatur benutzt (das ist der Driver) sitzt die anderen Person daneben und begleitet diesen Prozess (das ist der Observer). Das Ziel ist, dass die erste Person ihre gerade gemachten Handlungen erklärt, während die zweite nachvollzieht, korrigiert, hinterfragt und ergänzt. Dabei wird davon ausgegangen, dass wenn zwei Entwickler gemeinsam an einer Aufgabe arbeiten und darüber kommunizieren, einerseits die Aufgabe besser verstanden und erledigt wird (Vier-Augen-Prinzip), andererseits ein Wissenstransfer zwischen den Entwicklern stattfindet. Die Rollen des Driver und des Observer sollten sich regelmäßig abwechseln, damit beide Personen die Aufgabe aus verschiedenen Blickwinkeln betrachten können.

Distributed Party Programming erweitert dieses Konzept von einem Rechner auf mehrere, welche in einen Netzwerk miteinander verbunden sind. Räumliche Nähe ist nicht mehr erforderlich. Jeder Entwickler kann vor seinem eigenen Rechner sitzen, die Maus und Tastatur in der Hand haben und damit am einen Projekt arbeiten. Es gibt zwei Ansätze das zu ermöglichen. Einerseits kann während einer Sitzung der Bildschirminhalt eines Entwicklers als Videostream zu den anderen übertragen werden und alle haben die selbe Ansicht. Doch dabei teilen sich die Entwickler den selben Desktop und dieselben Werkzeuge, welche im Normalfall nur von einer Person gleichzeitig bedient werden können. Andererseits kann jeder Entwickler seine eigene Entwicklungsumgebung mit eigener Kopie des Projektes haben, wo nur noch die geänderten Fragmente (bearbeitete Stellen im Quelltext, Dateien im Projekt) zwischen den Teilnehmern einer Sitzung synchronisiert werden. So kann jede Person zu gleicher Zeit mit ihren eigenen Werkzeugen am gleichen Projekt gemeinsam arbeiten.

[3] http://www.saros-project.org

2.2 Softwarearchitektur

„Die Architektur eines Systems beschreibt die Strukturen des Systems, dessen Bausteine, Schnittstellen und deren Zusammenspiel. Die Architektur besteht aus Plänen und enthält meistens Vorschriften oder Hinweise, wie das System erstellt oder zusammengebaut werden sollte.“ ([2], S. 10). Das ist eine der vielen Definitionen für Softwarearchitektur. Das Software Engineering Institute (SEI) hat versucht den Begriff exakt zu definieren und hat schon über 50 verschiedene Definitionen dazu gesammelt[15]. Eine andere kürzere Definition, welche ich auf der BerlinExpertDays Konferenz kennengelernt habe lautet: „Softwarearchitektur ist die Summe aller wichtigen Entscheidungen“[19]. Von bestimmten Entscheidungen kann später der (Miss-)Erfolg einer Software abhängen und manche Entscheidungen können später nur sehr schwer wieder korrigiert werden. So beschäftigt sich die Softwarearchitektur aus meiner Sicht damit, wie gute und leicht zu pflegende/ wartende Software für den jeweiligen Einsatzzweck entstehen kann und mit der Rolle des Software-Architekten.

2.2.1 Sichten

Die Softwarearchitektur sollte einen abstrakten Blick auf ein System bieten. Die wichtigsten Elemente eines Systems sollten übersichtlich dargestellt werden, Implementierungsdetails sind wegzulassen. Zu besseren Erklärung werden Sichten verwendet. Eine Sicht betrachtet immer nur einen bestimmten Aspekt der Architektur. Für die Erstellung der Sichten werden Modellierungssprachen verwendet, der bekannteste Vertreter ist UML. Die vier wichtigsten Arten der Sichten([1], S. 82-83, Abbildung 2.1) sind:

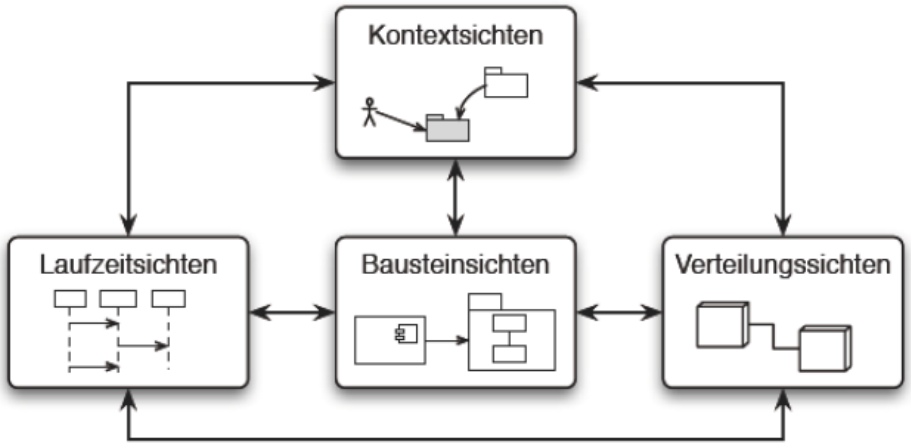

Abbildung 2.1 *Vier Arten von Sichten*

- **Kontextsichten** - wie die Software in die Umgebung, wo sie ausgeführt werden soll, integriert wird. Das ist der größte Überblick über das System, welcher z.B. auch den Kunden/Anwendern vorgestellt werden kann
- **Baustein/-Komponentensichten** - zeigen die internen statischen Strukturen der Bausteine/Komponenten eines Systems, sowie ihrer Schnittstellen und Beziehungen zu einander

- **Laufzeitsichten** - beschreiben das Laufzeitverhalten eines Systems, welche Bausteine wann und wie mit anderen interagieren. Das ist eine dynamische Betrachtung eines Systems
- **Verteilungssichten** - bilden das Hardwaresystem ab, auf welchen die Software läuft. Hier werden Rechner, Prozessoren, Netztopologien und -protokolle dokumentiert

2.2.2 Entwurf

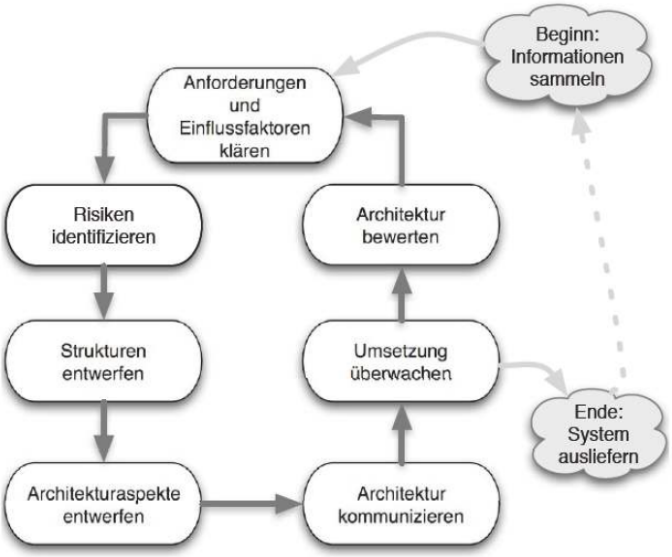

Abbildung 2.2 *Systematischer Entwurf von Softwarearchitekturen*

Ein Verfahren zum Entwerfen von Architekturen ist nach ([1], S. 36-37, Abbildung 2.2) ein iteratives Sammeln von Anforderungen und Einflussfaktoren; Identifizierung und Klärung der möglichen Risiken mit maßgeblichen Beteiligten am Projekt; Entwicklung einer grundlegenden Systemidee und ersten Strukturen; Entwurf der Sichten; Kommunikation mit verschiedenen Projektbeteiligten, schriftlich und mündlich; Beobachtung der Implementierung und Hilfestellung bei Problemen; Bewertung der Architektur.

2.2.3 Dokumentation

Softwarearchitekturen werden von *Architecture Description Languages* (ADL) beschrieben. Diese können sowohl grafisch, als auch in Textform ein System darstellen. UML ist z.B. eine ADL. Auf der Seite[16] des Software Engineering Institute (SEI) gibt es eine Übersicht dieser Sprachen. Zwei der ADLs möchte ich kurz vorstellen. *xADL*[17] basiert auf XML und bietet die Möglichkeit eine Beschreibung der Architektur noch in der Entwurfsphase anzufertigen. *xADL* stellt Elemente zur Verfügung, um Komponenten, Konnektoren, Interfaces und deren Zusammenspiel zu beschreiben. *arc42*[18] ist eine Vorlage (kommentiertes Inhaltsverzeichnis) für die Beschreibung von Softwarearchitekturen. Es wurde aus der Praxis entwickelt und basiert auf Erfahrungen internationaler Architekturprojekte und Rückmeldungen vieler Anwender.

2.3 Software Architecture Analysis Method (SAAM)

Die *Software Architecture Analysis Method* (*SAAM*)[31] ist eine szenariobasierte Architekturbewertung[32]. Mit diesem Vorgehensmodell erfolgt die Bewertung einer Architektur anhand von Szenarios. Unter einem Szenario wird eine kurze Beschreibung einer Anforderung an die Architektur verstanden.

SAAM definiert dabei 5 Schritte für eine Architekturbewertung:

1. **Beschreibung der Architektur** - der Entwurf der Architektur sollte durch Dokumentation und Sichten für alle Beteiligten klar und einfach verständlich dargestellt werden. Die wichtigsten Systemkomponenten und deren Zusammenspiel sollte beschrieben werden
2. **Entwicklung der Szenarios** - welche Anforderungen an die Architektur gerichtet werden und wie sich das System über die Zeit verändern kann. An der Entwicklung dieser Szenarios sollten möglichst alle beteiligten Rollen (Anwender, Kunden, Administratoren, Entwickler) teilnehmen
3. **Bewertung der einzelnen Szenarios** - die Szenarios werden in zwei Kategorien eingeteilt:
   - **Direkte Szenarios** - können mit der vorhandenen Architektur ohne Änderungen direkt ausgeführt werden
   - **Indirekte Szenarios** - es müssen Änderungen an der Architektur durchgeführt werden, um die Szenarios zu ermöglichen

   und danach den entsprechenden Teilen der Architektur zugeordnet. Bei den direkten Szenarios erfolgt eine Beschreibung der Schritte, um das Szenario auszuführen. Bei den indirekten Szenarios wird eine Beschreibung der notwendigen Änderungen an der Architektur angefertigt, mit Schätzung des Aufwandes
4. **Betrachtung der Szenariointeraktionen** - es wird untersucht, ob mehrere Szenarios Änderungen am selben Teil eines Systems erfordern. Das könnte auf Probleme in der Architektur hinweisen, weil ein Teilsystem entweder mehrere verschiedene Aufgaben übernimmt oder nicht ausreichend definiert/beschrieben ist. In so einem Fall sollte dieser Teil der Architektur noch mal überarbeitet werden
5. **Gesamtbewertung** - die bewerteten Szenarios werden gewichtet und verglichen. Damit lässt sich eine Priorität der Szenarios für Änderungen aufstellen. Dieser Prozess ist subjektiv, welcher die Beteiligung aller Rollen im Projekt erfordert

2.4 OSGi

Die *OSGi Alliance*[4] (Open Services Gateway initiative) ist eine Standardisierungsorganisation, im Jahr 1999 gegründet. Das Ziel dieser Organisation ist eine hardwareunabhängige dynamische Softwareplattform, die OSGi-Plattform, zu spezifizieren, um Anwendungen über das Komponentenmodell zu modularisieren. Die OSGi-Plattform setzt auf Java als Ausführungsumgebung auf. Die *OSGi Alliance* stellt

[4] http://www.osgi.org

keine Implementierung der OSGi-Plattform bereit. Zu den bekanntesten Open Source Implementierungen zählen *Equinox*, von der Eclipse Foundation und die Basis für Eclipse, und *Apache Felix*, von der Apache Software Foundation. Weiterhin gibt es eine Reihe kommerzieller OSGi-Frameworks.

Eine der Schwächen von Java ist die logische Strukturierung von Anwendungen. Dazu kennt Java Klassen und Packages. Doch für größere Anwendungen ist es zu fein. JAR-Dateien bieten hier auch keine bessere Lösung, denn nach dem Laden befinden sich alle Klassen im selben Raum. OSGi versucht diesen Umstand durch Modularisierung zu verbessern. „Hinter dem Konzept Modularität/Modularisierung steht das Bestreben, Softwareeinheiten (Module/Komponenten) unabhängig vom Systemkontext verändern und pflegen zu können. Auswirkungen der Veränderung auf andere Softwareeinheiten sollten begrenzt sein und keine ungewöhnlichen Nebeneffekte auslösen“([11], S. 10). Das Komponentenmodell von OSGi versucht eine klare Abgrenzungen zwischen den Komponenten zu erreichen. Nur die öffentlichen Schnittstellen (Services) sind sichtbar, die Implementierungsdetails bleiben verborgen. Abhängigkeiten zu anderen Komponenten werden deklariert, die Verbindung erfolgt durch die OSGi-Plattform (Inversion-of-Control Prinzip). Weiterhin ermöglicht OSGi das dynamische Laden und Entladen der Komponenten zur Laufzeit. In Abbildung 2.3 ist die OSGi-Architektur dargestellt.

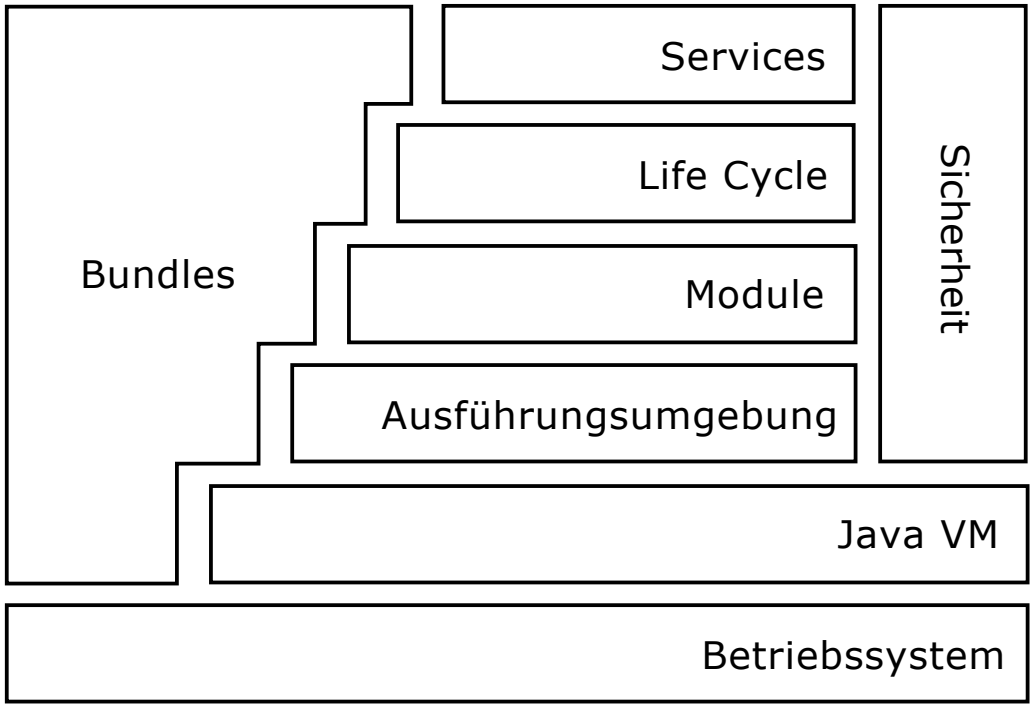

Abbildung 2.3 *OSGi-Architektur ([11], S. 14)*

Eine OSGi Komponente wird unabhängig von anderen Komponenten in eine JAR-Datei verpackt. Dazu kommen alle benötigten Ressourcen (wie Konfigurationsdateien oder Grafiken) hinzu. Dieses Paket wird in OSGi als „Bundle“ bezeichnet. Im Bundle-Manifest (basierend auf dem JAR-Manifest) wird definiert welche Services aus anderen Bundles benötigt werden und welche Services das Bundle selber anderen zu Verfügung stellt. Ein Bundle kann eine bestimmte Ausführungsumgebung voraussetzen (z.B. Java Enterprise

Edition), um ausführbar zu sein. Das OSGi-Frameworks muss sicherstellen, dass alle benötigten Ressourcen zu Verfügung stehen, bevor das Modul aus dem Bundle geladen wird.

Der „Life Cycle“ regelt die Ausführung eines Moduls. Es gibt die Zustände *INSTALLED* für „bekannt im OSGi-Framework und ausreichende Ausführungsumgebung vorhanden“; *RESOLVED* für „Abhängigkeiten zu anderen Modulen aufgelöst, d.h. alle benötigten Services sind vorhanden und aktiv“; *STARTING* für „das Laden und Aktivieren des Programms“; *ACTIVE* während der Laufzeit; *STOPPING* für „das Anhalten und Beenden der Laufzeit“ und *UNINSTALLED* für „das Entladen des Bundles aus dem OSGi-Framework“. Jedes Modul muss entsprechend auf diese Zustände reagieren.

Über die Services wird die Funktionalität eines Moduls angeboten. Es ist also die öffentliche Schicht. Module können von anderen Modulen oder nur von bestimmten Services abhängig sein. Verschiedene Module können gleiche Services anbieten. Auch verschiedene Versionen eines Moduls können parallel in einem OSGi-Framework ausgeführt werden.

Damit nur die öffentliche Schicht eines Moduls sichtbar ist, muss ein OSGi-Framework sicher stellen, dass die jeweilige interne Implementierung sich in verschiedenen Klassenräumen befinden. Weiterhin müssen zyklische Abhängigkeiten richtig aufgelöst und die Reihenfolge der Aktivierung festgelegt werden.

2.4.1 OSGi und Java 7 bzw. 8

OSGi ist ein aktives Projekt, die letzte Spezifikation (R4.3) erschien im April 2011. Es gibt Bestrebungen OSGi direkt in die Java Plattform (JDK) zu integrieren, JSR 277: Java Module System[28]. Doch es gibt auch andere Projekte, wie *Jigsaw*[24], JSR 294: Improved Modularity Support in the Java Programming Language[30], welche eine Modularisierung von Java ermöglichen wollen und in Konkurrenz zu OSGi stehen. Daneben gibt es noch JSR 291: Dynamic Component Support for Java SE[29], welches zum Ziel hat ein Komponenten-/Modulkonzept in die JVM zu integrieren. Eigentlich war es geplant ein Modularisierungssystem schon in Java 7 einzuführen, doch es wurde auf Java 8 verschoben[25].

2.5 Dependency Injection

Dependency Injection ist ein Entwurfsmuster welches sowohl in Eclipse, als auch in Saros eine große Rolle spielt, deshalb möchte ich es vorstellen. Das Ziel dieses Musters ist die Abhängigkeiten zwischen Objekten in einem objektorientierten System zu minimieren. Wenn ein Objekt `Alice` ein anderes Objekt `Bob` verwenden muss, muss das zweite vorher irgendwie erzeugt werden. `Alice` kann es direkt machen, doch damit würde es die Implementierung von `Bob` auch direkt kennen. Sogar wenn `Bob` über eine Abstraktionsschicht `IBob` verfügt, muss trotzdem die Implementierung bekannt sein, um es zu erzeugen, wie in Abbildung 2.4 dargestellt.

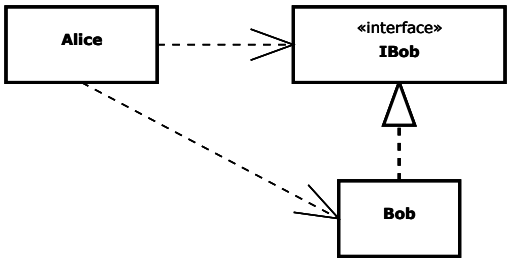

Abbildung 2.4 *Direkte Abhängigkeit zwischen Alice und Bob*

Mit *Dependency Injection* übernimmt die Erzeugung eine andere Klasse, z.B. `Assembler`, welche die Implementierung von `Bob` kennt, diese erzeugt und `Alice` zuweist bzw. rein injiziert, wie es bei *Dependency Injection* heißt. So kann `Alice` auf die Funktionalität von `Bob` über die Abstraktionsschicht `IBob` zugreifen, ohne die Implementierung zu kennen. Im `Assembler` kann konfiguriert werden, welche Implementierung über `IBob` an `Alice` zugewiesen wird. Das ist in Abbildung 2.5 dargestellt.

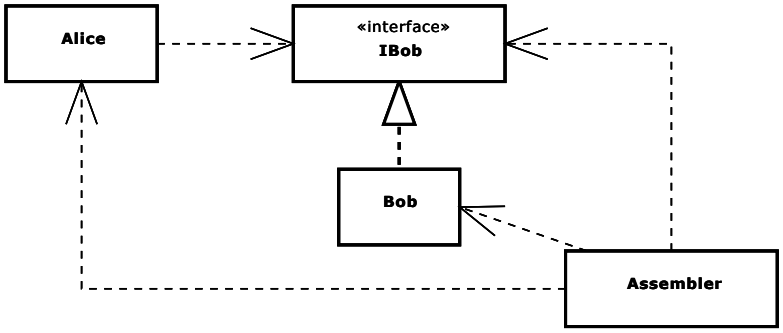

Abbildung 2.5 *Keine direkte Abhängigkeit zwischen Alice und Bob mehr*

Martin Fowler[22] unterscheidet 3 Arten von Dependency Injection: *Interface Injection* - wo die benötigten Referenzen über ein Interface injiziert werden, *Setter Injection* - über Setter-Methoden und *Constructor Injection* - über die Parameter des Konstruktors einer Klasse. Vor allem die letzte Variante wird ausgiebig in Saros verwendet. Weiterhin gibt es noch andere Arten der *Injection*, wie z.B. über öffentliche Felder einer Klasse oder Annotationen.

2.5.1 PicoContainer

Der *PicoContainer*[5] ist ein Java-Framework, welches *Dependency Injection* anbietet und von Saros verwendet wird. Nach der Abbildung 2.5 wäre es der `Assembler`. Im Vergleich zu anderen Frameworks für *Dependency Injection* ist der *PicoContainer* leicht, klein und einfach zu integrieren.

[5] http://picocontainer.org

2.6 XMPP

Die Abkürzung XMPP steht für „Extensible Messaging and Presence Protocol“ und ist ein Echtzeitübertragungsprotokoll, auf Basis von XML[34]. Es kann für Nachrichtenübermittlung, Konferenzen mit mehreren Teilnehmern, Anzeigen des Online-Status, Datei-, Sprach-, Videoübertragungen und andere Dienste verwendet werden. Die Netzwerk-Architektur von XMPP ist ähnlich zu Simple Mail Transfer Protocol (SMTP). Es gibt keinen zentralen XMPP-Server, es steht jedem frei seinen Server in das Netzwerk dazuzuschalten. Ein Teilnehmer im XMPP-Netzwerk bekommt eine eindeutige ID, den sogenannten „Jabber Identifier“ (JID) [6]. Dieser ist ähnlich zu einer E-Mail-Adresse aufgebaut, z.B. „name@myserver.de“, und setzt sich aus einem frei wählbaren Namen des Benutzers und des Domainnamen des Servers zusammen. Das XMPP-Protokoll wurde 2004 von der Internet Engineering Task Force (IETF) als offizieller Standard veröffentlicht.

Die Vorteile von XMPP liegen in seiner offenen Struktur und aktiver Weiterentwickelung. Weil es auf XML basiert, lässt es sich leicht durch eigene XML-Elemente erweitern und ist damit sehr flexibel. So entstand eine Reihe von Erweiterungen[35] für XMPP, welche von der XMPP Standards Foundation (XSF) verwaltet werden. Darunter ist Jingle, eine Erweiterung von XMPP für Peer-to-Peer-Sitzungen.

2.6.1 Smack

Smack[7] ist eine Open Source Java-Bibliothek für die Arbeit mit XMPP, welche von der Firma *Jive Software* entwickelt wurde. Am Anfang meiner Arbeit sah es so aus, dass Smack seit Jahren nicht mehr weiterentwickelt wurde, doch überraschenderweise wurde eine neue Version angekündigt. Smack implementiert die Jingle Erweiterung, welche von Saros benötigt wird.

[6] Das XMPP-Protokoll entstand aus dem Jabber-Projekt, welches 1998 von Jeremie Miller gegründet wurde. Deshalb taucht der Name Jabber noch immer in der XMPP-Spezifikation auf.

[7] http://www.igniterealtime.org/projects/smack/

3 Saros Architektur

In diesem Kapitel wird die aktuelle Architektur von Saros beschrieben. Als ich mit Saros angefangen habe zu arbeiten, gab es für Saros kein Dokument, welches sich ausschließlich mit der aktuellen Architektur des Systems beschäftigte. So musste ich zuerst aus verschiedenen Quellen die benötigten Informationen beziehen, um mir einen Überblick zu ermöglichen (Abschnitt 3.1). Als nächsten Schritt habe ich eine Sicht der aktuellen Saros-Architektur erstellt (Abschnitt 3.2). Danach werden die Probleme der aktuellen Saros Architektur beschrieben (Abschnitt 3.3) und Ansätze zu Verbesserung diskutiert (Abschnitt 3.4).

3.1 Architekturkonzepte im Saros-Projekt

Das für Entwickler benötigte Wissen im Saros-Projekt ist auf verschiedene Informationsspeicher aufgeteilt:

- **Webseite zum Projekt**
- **Mailinglisten** (spline.de und sourceforge.net)
- **Saros Quelltext**
- **Abschluss-/Studienarbeiten**
- **Personen, Team**
- **Prozesse** - Test- und Releasemanagement, Reviews
- **Werkzeuge** - Review-Board, Testlink und Bugtracker

3.1.1 Niedergeschriebenes Wissen

Die Webseite zum Projekt enthält einen groben Überblick über Saros und die Entwicklungsprozesse. Gut zum Einstieg oder dem ersten Kennenlernen geeignet, geht aber nicht ins Detail. Einige der Informationen sind auch inzwischen veraltet oder überholt. Leider ist es rein aus der Webseite nicht erkennbar, was noch aktuell ist. Es gibt zwar das Datum der letzten Veränderung einer Seite im Wiki-System, doch es ist nur eine relative Angabe über die Aktualität. Auf der Seite „Technical Information“ [21] gibt es eine Kontextsicht der Kommunikationsarchitektur von Saros, welche immer noch aktuell ist, Abbildung 3.1. Diese Kontextsicht zeigt das Konzept der Kommunikation der Saros-Clienten über XMPP-Nachrichten welche über Jabber-Server gesendet werden. Der Transfer von größeren Dateien erfolgt aus Performancegründen direkt, falls möglich.

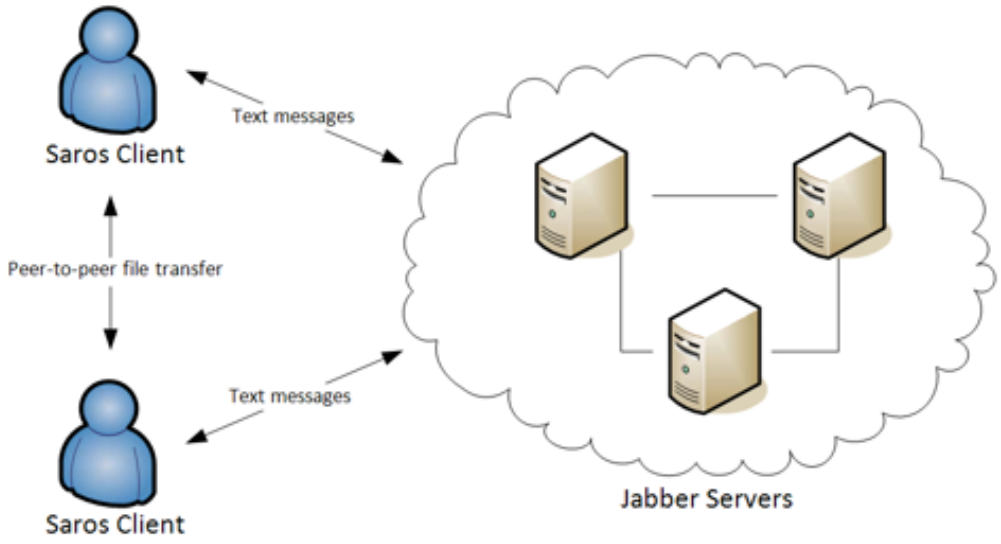

Abbildung 3.1 *Saros Kommunikationsarchitektur*

Auf der selben Seite ist auch beschrieben, dass Saros den *PicoContainer* verwendet, um das System zu managen. Mit dem *PicoContainer* lässt sich automatisch ein Diagramm über den Programmaufbau zu Laufzeit (Objektnetz mit den wichtigsten Klassen) der Software erstellen. Verlinkt ist zusätzlich ein Diagramm von Mai 2009, Revision 1200, welches aber inzwischen überholt ist. Mit einem aktuellen Diagramm (Revision 3375, Juni 2011), aus den Laufzeitdaten von Saros generiert, lässt sich ein visueller Vergleich über das Wachstum von Saros innerhalb eines Zeitraums von 2 Jahren erstellen, sehe Abbildung 3.2. Leider sind beide Diagramme nicht zum einfachen Verständnis von Saros oder der Architektur geeignet, da zu komplex, zu viel Information auf einmal und entweder auf dem Bildschirm, noch auf A4-Paper überblickbar.

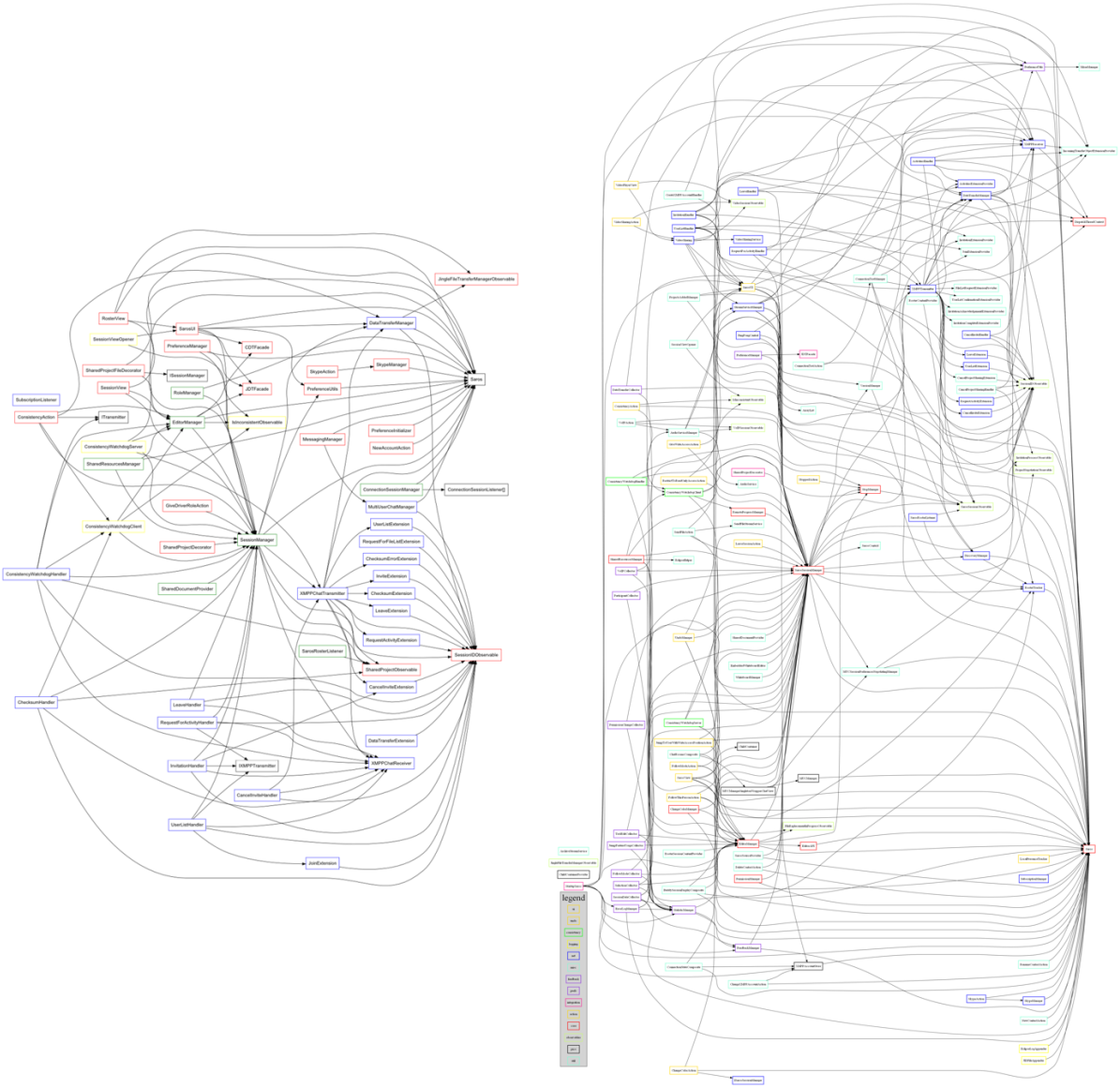

Abbildung 3.2 *Saros Objektnetz zur Laufzeit, links Mai 2009, rechts Juni 2011*

Aus der Mailingliste habe ich während meiner Arbeit eine wichtige strukturelle Änderung von Saros mitbekommen[8]. Dabei wurde das System von einer projektzentrierten auf eine sessionzentrierte Architektur umgestellt. So konnte Saros vorher nur jeweils ein Projekt während einer Session gemeinsam bearbeiten. Jetzt wurde es möglich, mehrere Projekte während einer Session zu bearbeiten und dynamisch hinzuzufügen, wie in Abbildung 3.3 dargestellt. So wurde die Session zu zentralen Stelle im System.

[8] E-Mail „Specification document“ auf saros@lists.spline.de, 06.01.2011

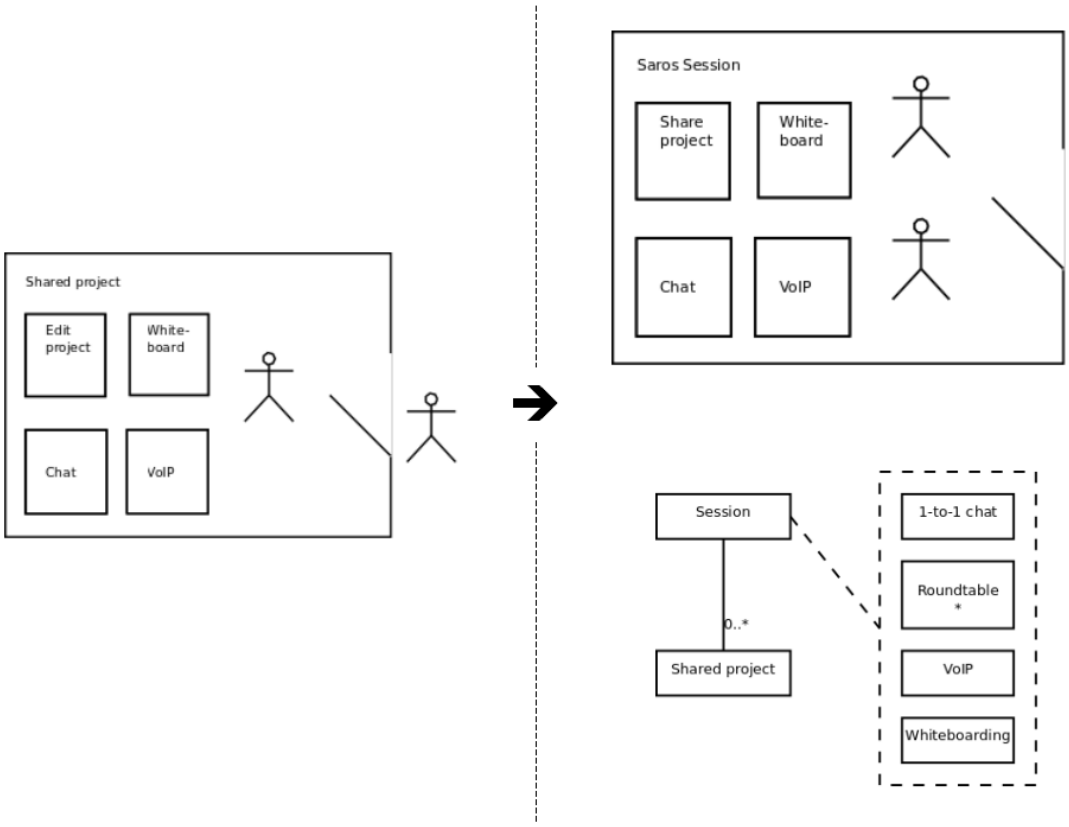

Abbildung 3.3 *Umstellung der projektzentrierten (links), auf eine sessionzentrierte Architektur*

3.1.2 Quelltext

Der Quelltext von Saros ist die zentralste und aktuellste Quelle im Projekt. Die Packages-Aufteilung bietet einen guten Überblick über das System und die Bezeichnungen der Klassen helfen beim Verstehen der enthaltenen Funktionalität. Doch eine Architektur oder der Aufbau des Systems lässt sich nicht ohne weitere Mittel erkennen. Beim Lesen des Quelltextes fällt sofort auf, dass viele Klassen, die nach der Bezeichnung eigentlich zu verschiedenen Teilen des Systems gehören, direkte Abhängigkeiten haben. Weiterhin ist Saros, die GUI und die Eclipse-Funktionalität ineinander verflochten, klare Grenzen sind nicht zu erkennen. Zusätzlich muss man das Eclipse Plugin-System kennen, um Saros zu verstehen.

Ich habe versucht den Quelltext mit automatischen Werkzeugen zu analysieren. Dabei kamen JDepend[9] und X-Radar[10] zum Einsatz. Diese lieferten zwar eine Menge Informationen zu der Quelltextqualität, z.B. dass es viele direkte Abhängigkeiten im System gibt, doch sie halfen mir nicht dabei die Architektur von Saros besser zu verstehen.

Eine für mich interessante Eigenschaft in Saros ist die Annotation `dpp.annotations.Component`. Damit markierte Klassen werden in dem *PicoContainer*

[9] http://www.clarkware.com/software/JDepend.html

[10] http://xradar.sourceforge.net

zu Modulen zusammengefasst. Klassen ohne diese Annotation gehören automatisch zu dem Modul *misc*. Damit wird eine Strukturierung von Saros in Module erkennbar. Folgende Module lassen sich in Saros finden:

- @Component(module = "accountManagement")
- @Component(module = "action")
- @Component(module = "communication")
- @Component(module = "consistency")
- @Component(module = "core")
- @Component(module = "feedback")
- @Component(module = "integration")
- @Component(module = "logging")
- @Component(module = "misc")
- @Component(module = "net")
- @Component(module = "observables")
- @Component(module = "pico")
- @Component(module = "prefs")
- @Component(module = "ui")
- @Component(module = "undo")
- @Component(module = "util")

3.1.3 Wissen im Team

Während meiner Arbeit an Saros war das Team eine der wichtigsten Quellen für Wissen über Saros. Ich habe mit 5 Entwicklern über das Projekt, über ihre Arbeit, über die Architektur von Saros, sowie über die Probleme und die Wünsche im Bezug darauf gesprochen. Vor allem war es mir wichtig zu erfahren, wie sie überhaupt Saros sehen und wie sie es gerne sehen würden. Folgende Liste an Stichpunkten entstand dabei:

- **Probleme**
    - Die Architektur von Saros ist nicht so einfach zu verstehen und es gibt zu wenig Abstraktion
    - Es gibt keine echten Schnittstellen zwischen GUI, Logik und Basis im System
    - Tests lassen zu wünschen übrig
    - Nicht alle Studenten, welche am Saros-Projekt arbeiten oder es gemacht haben, haben die selbe Erfahrung im Bereich Softwareentwicklung, entsprechenden Prozessen oder größeren Softwareprojekten. Das spiegelt sich auch im Quelltext wieder
    - Für Anfänger im Saros Projekt ist es oft unklar, was wie abläuft. Dieses Wissen muss man sich erst selber durch Nachfragen und Ausprobieren aneignen, was Zeit kostet
    - Neue Teilsysteme und teilweise auch Änderungen werden an Saros „drangeheftet“, ohne in den vorhandenen Code rein zu gehen, diesen zu verstehen, sinnvoll zu ändern und damit zu verbessern
    - Hat ein Entwickler ein Teilsystem, welches nicht gerade zu den Kernkomponenten gehört, fertig gestellt, seine Arbeit abgeschlossen und das

Projekt als aktiver Mitarbeiter verlassen, liegt dieses Teilsystem oft brach und keiner kümmert sich darum

- **Wünsche**
  - Die High-Level Konzepte sollten direkt auf den Code abgebildet sein
  - Die Logik sollte von der GUI unabhängig sein
  - Mehr Flexibilität für Saros, um das Testen von Teilsystemen (z.B. ohne GUI) zu ermöglichen
  - Es sollte möglich sein das System stellenweise zu ändern, ohne dass auch andere Stellen, zu welchen eigentlich kein Bezug besteht, angepasst werden müssen
  - Ein Plan für das Testen
  - Aktivere Entwickler, die Tests schneller erstellen
  - Es wurden auch andere Entwicklungswerkzeuge (im Bezug auf das Review-Board und Testlink) gewünscht, ohne aber konkret zu sagen, welche
  - Das Saros Projekt sollte sich auf die Kernkompetenzen konzentrieren und keine Teilsysteme entwickelt, welche auch durch externe Software leicht ersetzbar sind. Gemeint waren damit solche Komponenten, wie der Chat und VoIP

Von einem Entwickler habe ich eine Komponentensicht bekommen, wie aus seiner Sicht die vereinfachte Architektur des Systems aussehen sollte, Abbildung 3.4. Dabei gibt es 4 Hauptkomponenten (von unten nach oben): die Netzwerkschicht, die Topologie (Struktur der Verbindungen mehrerer Saros-Clienten), die Session und der Rest des Systems.

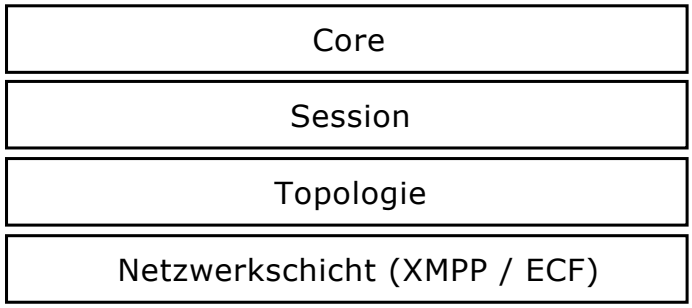

Abbildung 3.4 *Vorschlag einer Saros Architektur seitens eines Entwicklers*

Von einem anderen Entwickler, welcher für seine Arbeit das System auch analysierte, habe ich ein aktuelles Komponentendiagramm des Systems bekommen, Abbildung 3.5.

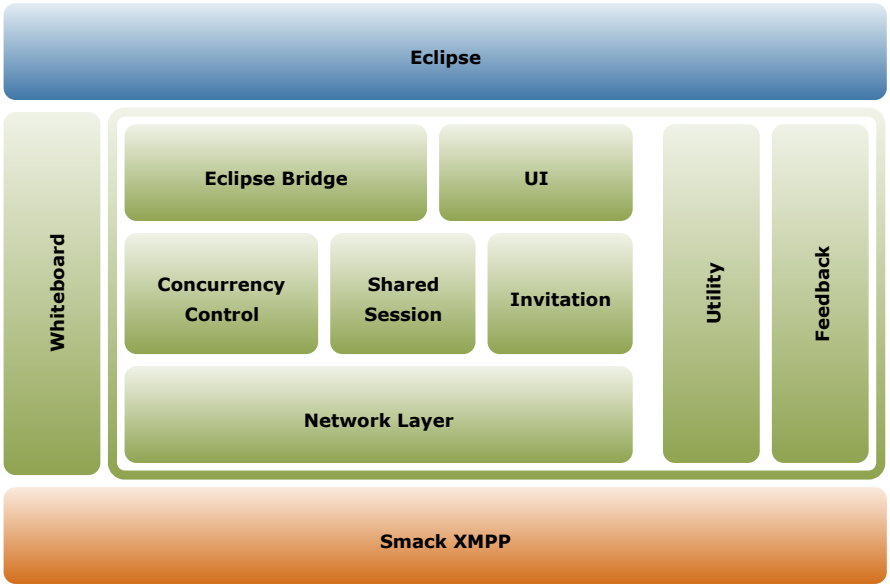

Abbildung 3.5 *Saros Komponentendiagramm (Stand Juni 2011)*

3.1.4 Wissen aus Prozessen und Werkzeuge

Während meiner Arbeit habe ich sowohl die Rollen des Release-, als auch Testmanagers übernommen. Während dieser Zeit konnte ich die aktuellen technischen Probleme von Saros und deren Behebung beobachten. Ich habe daran teilgenommen und so den Quelltext von Saros besser kennengelernt. Weiterhin konnte ich die Prozesse im Team nachvollziehen. Es war eine gute Gelegenheit Saros mit mehreren Personen ausführlich zu testen und darüber zu diskutieren.

Die im Projekt verwendeten Werkzeuge, das Review-Board, Testlink und Bugtracker, bieten eine punktierte Sicht auf bestimmte Probleme, Diskussionen darüber und deren Lösungen. Sie sind aber zu fein, um einen größeren Gesamtüberblick über das System zu bekommen.

3.2 Aktuelle Saros Architektur

Aus den Informationen des vorherigen Abschnitts habe ich die aktuelle Saros Architektur abgeleitet. Diese werde ich in zwei Abschnitten behandeln, einmal aus der Komponentensicht und einmal aus der Programmaufbausicht.

3.2.1 Komponentensicht

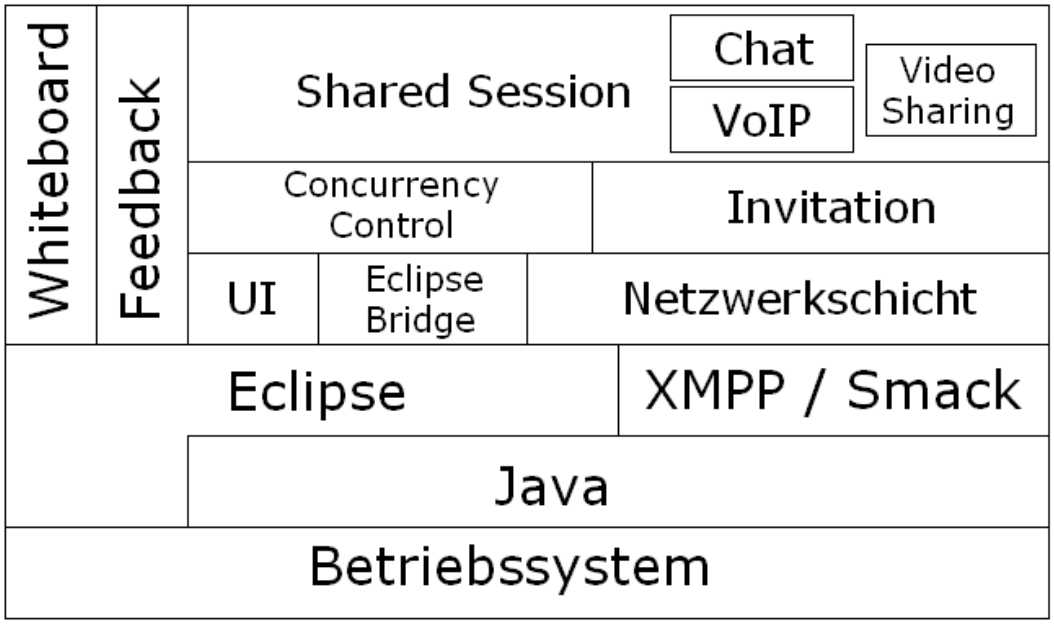

Abbildung 3.6 *Saros Architektur (Stand Juni 2011)*

Das Diagramm in Abbildung 3.6 zeigt die Komponenten von Saros und den Aufbau des Systems von unten nach oben. Saros setzt direkt auf *Eclipse* und der Open Source XMPP (Jabber) Bibliothek *Smack* auf. Die *Netzwerkschicht* übernimmt die Aufgabe die Ereignisse im System in XMPP und zurück zu transformieren. In der *UI* sind die Views, die Dialoge und Wizards für Saros enthalten. Die *Eclipse Bridge* stellt eine Brücke von *Eclipse* zum Kern von Saros, der *Concurrency Control* her, darunter die Text- oder allgemeine Eingabeverarbeitung. Die *Concurrency Control* ist für die Koordinierung der gemeinsamen Arbeit zwischen den Saros-Clienten zuständig. *Invitation* fasst den Einladungsprozess, welcher dem Aufbau der Session dient, zusammen. In *Shared Session* erfolgt die Koordination der darunterliegenden Komponenten während einer aktiven Session.

Neben den Kernkomponenten gibt es noch einige Teilsysteme, welche die Arbeit mit Saros erleichtern sollen. Dazu gehören der *Chat*, *VoIP* und *Video Sharing*, welche für die zwischenmenschliche Kommunikation zuständig sind. Diese Komponenten hängen von der *Shared Session* ab. Das bedeutet, dass sie nur dann funktionieren, wenn eine Session aktiviert wurde, obwohl sie auch eigenständig laufen könnten. Das *Feedback* gibt den Anwendern eine Möglichkeit den Saros Entwicklern Informationen über die Benutzung des Programms mitzuteilen. Das *Whiteboard* ermöglicht es gemeinsam Zeichnungen (Diagramme, Freihand) anzufertigen. Es ist nur sehr lose an Saros gekoppelt und ein eigenständiges Projekt in Eclipse (ein Bundel nach OSGi). Das lokale Zeichnen im *Whiteboard* funktioniert auch ohne einer aktiven Session. Die Daten werden bei der Herstellung einer Verbindung zwischen den Saros-Clienten synchronisiert.

Es muss beachtet werden, dass es keine scharfen Grenzen oder definierte Schnittstellen zwischen den einzelnen Komponenten im System gibt, auch wenn das Diagramm es suggerieren könnte. Zum Beispiel nimmt die *UI* Komponente auf dem Diagramm zwar nicht

viel Platz ein, doch ihre Funktionalität befindet sich auch in den anderen Komponenten und umgekehrt.

3.2.2 Programmaufbausicht

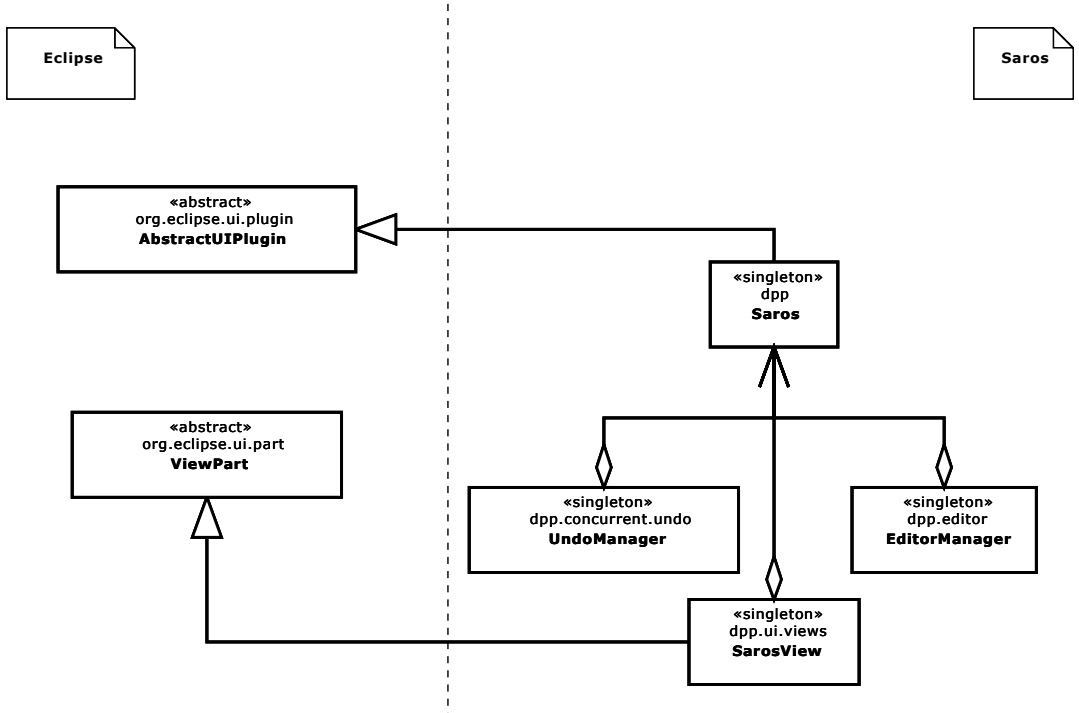

Abbildung 3.7 *Vereinfachter Aufbau des Saros-Systems*

Die zentrale Klasse `Saros` erbt vom `AbstractUIPlugin` des Plugin-Konzeptes von Eclipse. Wenn Eclipse das Saros-Plugin aktiviert, wird in der `Saros` Klasse der *PicoContainer* initialisiert, mit einer Liste von Klassennamen aus dem Saros-System. Der *PicoContainer* übernimmt jetzt die Aufgabe diese Klassen über *Dependency Injection* in der richtigen Reihenfolge zu initialisieren und einander zuzuweisen. Dieses Konzept des Programmaufbaus von Saros ist auf der rechten Seite in der Abbildung 3.7 dargestellt. Stellvertretend für diese Art von Klassen sind hier der `UndoManager` und `EditorManager` erwähnt. Die vollständige Liste (welche sich regelmäßig ändert) ist in der Klasse `dpp.SarosContext` zu finden.

Nicht alle Klassen werden direkt bei der Konstruktion der `Saros` Klasse vom *PicoContainer* erzeugt. Weil das Eclipse Plugin-Konzept die benötigten Teile eines Eclipse-Plugins nur dann aktiviert, wenn diese verwendet/benötigt werden, z.B. Views wie `SarosView`, werden diese von Eclipse erzeugt und erst zu Laufzeit dem *PicoContainer* hinzugefügt, wo diesen die benötigten Klassen aus dem Saros-System injiziert werden.

Das vollständige vom *PicoContainer* erstellte Objektnetz des laufenden Saros-Systems kann visualisiert werden und würde wie in Abbildung 3.2 aussehen. Die Anleitung zu Visualisierung ist in der Developer Guide[11] auf der Webseite des Saros-Projektes zu finden, unter dem Abschnitt „How to create an Architecture Diagram“.

[11] http://www.saros-project.org/HowToDevel

3.3 Probleme der aktuellen Saros Architektur

3.3.1 Historisch gewachsen

Die jetzige Saros Architektur lässt sich am besten als eine flache Ebene beschreiben, wo sich jedes Teilsystem direkt kennt oder kennen kann. Höhere Abstraktionsschichten gibt es nicht. So ein System lässt sich mit der Entwicklungsgeschichte von Saros erklären. Am Anfang war das System einfach und übersichtlich, doch mit der Zeit kamen immer mehr neue Features hinzu, welche einfach daneben gestellt bzw. in den *PicoContainer* geworfen wurden. Die Integration der neuen Features in das vorhandene System erfolgte direkt, was in einer hohen Anzahl der Abhängigkeiten resultierte. So stieg die Komplexität des gesamten Systems, was zum verlorenen Überblick, erhöhter Fehlerhäufigkeit und größerem Wartungsaufwand führte. In der Abbildung 3.2 ist das Objektnetz des aktuellen Systems (11.7.1.DEVEL) dargestellt, was auch zugleich die Komponentensicht darstellt. So ein Diagramm kann zwar dabei helfen Saros zu verstehen, doch es ist mühsam und kostet Zeit. Weil das Diagramm automatisch aus den Laufzeitdaten von Saros generiert wird, kommt hinzu, dass mit größeren Änderungen am System sich auch dieses Diagramm stark verändert, wobei der Wiedererkennungswert verloren geht.

3.3.2 Strukturelle Probleme

Die aktuelle Architektur wird über *Dependency Injection* und konkrete Klassen realisiert. Dabei kommt ein *PicoContainer* zum Einsatz, mit welchem Saros zusammengebaut wird. Das Prinzip des *Dependency Injection* wird dabei nicht sauber eingehalten. Nach Martin Fowler[22] sollte am besten nicht direkt auf die Implementierung zugegriffen werden, sondern nur auf eine abstrakte Schicht. Der *PicoContainer* unterstützt dieses Verhalten, doch es wird nicht verwendet. Die Referenzen, welche aus dem *PicoContainer* bezogen werden, sind direkte Verweise auf konkrete Klassen und somit die Implementierung. In Listing 3.1 ist z.B. eine *Constructor Injection* zu sehen, wobei die Parameter konkrete Klassen sind. Nach Gernot Starke([1], S. 162) sollte es am besten nur Abhängigkeiten von Abstraktionen und nicht von konkreten Implementierungen geben. Dies kann zu Problemen führen, wenn z.B. eine alternative Implementierung einer Klasse verwendet werden muss, ohne die aktuelle aus dem System zu entfernen. Auch strukturelle Änderungen solcher Klassen werden schwieriger.

Mit dem *PicoContainer* wird der Aufbau des Systems an dieses Framework delegiert und somit der Kontrolle der Entwickler entzogen. Das hat selbstverständlich Vorteile, vor allem beim Zusammenbau und Integration neuer Features, also dem Hinzufügen. Doch wenn ein Redesign des Systems erfolgen sollte, also Verändern oder Entfernen, wird die Intransparenz des *PicoContainer* zum Problem. Manche Komponenten werden dann nicht mehr richtig initialisiert, was erst zu Laufzeit auffallen kann. Der Einsatz des *PicoContainers* hat außerdem dazu geführt, dass an einigen Stellen der Quelltext stark aufgeblasen wurde, sehe Listing 3.1. Das Hinzufügen/Entfernen einer Klasse aus dem *PicoContainer* muss an allen solchen Stellen bekannt gemacht werden. Weiterhin wiederholen sich die Parameterlisten aus ähnlichen Klassen, was zu redundantem Quelltext führt. In einigen Klassen werden manche Parameter gar nicht mehr verwendet, doch noch immer im Konstruktor deklariert.

Weiterhin ist die aktuelle Architektur so gestaltet, dass immer nur genau eine Konfiguration des Systems erstellt werden kann. Änderungen an der Konfiguration oder der Zusammenstellung des Aufbaus haben immer Auswirkungen auf das ganze System. Das erschwert das Testen einzelner Teile unabhängig vom System, um Fehler besser zu lokalisieren oder weniger Aufwand/Ressourcen für einen Test zu verwenden. Wegen der starken Verflechtung von Eclipse, der GUI und Saros ist das System nur schwer auf andere Plattformen (z.B. Netbeans) zu portieren.

```
public class StreamServiceManager implements Startable {

    public StreamServiceManager(
        XMPPReceiver xmppReceiver,
        DataTransferManager dataTransferManager,
        SarosSessionObservable sarosSessionObservable,
        Saros saros,
        SarosSessionManager sessionManager,
        List<StreamService> streamServices,
        IncomingTransferObjectExtensionProvider
           incomingTransferObjectExtensionProvider) {
        /* ... */
    }
}
```

Listing 3.1 *Klasse wird über PicoContainer injiziert*

3.3.3 Organisatorische Probleme

Die Entwicklung von Saros mit den Mitteln einer Universität hat den Effekt, dass ein großer Teil der Entwickler für eine kurze Zeit am Projekt arbeitet (in der Regel 3 bzw. 6 Monate) und danach nicht mehr aktiv zu Verfügung steht. Somit besteht eine hohe Fluktuation an Entwicklern. Weiterhin gibt es bemerkbare Unterschiede an Erfahrung im Bereich Softwareentwicklung. Während der Arbeit an Saros ist ein Teil der Entwickler außerdem damit beschäftigt seine Abschluss-/Studienarbeit fertig zu stellen. Somit kann ein Entwickler die meiste Zeit darauf verwenden sich vor allem um seinen Teil des Programms zu kümmern und wenig zu dem gesamten System beitragen. Jeder sollte zwar regelmäßig an den Reviews der Patches teilnehmen, doch in der Praxis lief es eher schleppend, oft erst nach direkten Aufforderungen. Das hatte mehrere Gründe: Zeitmangel, fehlendes Wissen in dem Teil des Systems wo das Review stattfand, größere Patches oder der fehlende direkte Kontakt zwischen den Entwicklern, um Fragen schneller zu klären.

3.3.4 Probleme der Dokumentation

Das während der Arbeit an Saros entstandene strukturierte Wissen, welches für andere und neue Entwickler wichtig ist, wird meistens in der jeweiligen Abschluss-/Studienarbeit schriftlich festgehalten. Doch es braucht Zeit, sich durch diese Informationsquelle durchzuarbeiten, vor allem weil nicht alles direkten Bezug auf das System hat oder nicht mehr aktuell ist. Für Hintergründe ist es wichtig, aber um das aktuelle System schnell zu verstehen ist es eher ein Hindernis.

In den Archiven der Mailinglisten finden sich viele Informationen zum Projekt, leider ist es eher als großer Haufen zu betrachten, welcher durchsucht werden kann. Wichtige Entscheidungen, welche auf der Mailingliste besprochen wurden, werden anderswo nicht festgehalten, wie z.B. die Umstellung von einer projektzentrierten auf eine sessionzentrierte Architektur. Ähnlich sieht es bei Informationen in den Werkzeugen (Review-Board, Testlink) aus. Es ist da, es kann durchsucht werden, aber es kostet Zeit und es ist nicht immer klar, was noch aktuell ist.

Weiterhin sollte beachtet werden, dass Saros auch umfassendes Wissen über das Eclipse Plugin-System erfordert. Wer bisher damit nicht vertraut war, muss sich auch dieses Wissen aneignen.

Eine zusammenfassende und aktuelle Dokumentation von Saros, wo die wichtigsten Informationen der jeweiligen Arbeiten und die wichtigen Entscheidungen zum Projekt in kompakter Form zusammenfließen, gibt es zu Zeit nicht. Somit bleibt der Quelltext als Haupt-, und Nachfragen bei den anderen und früheren Entwicklern als Sekundärdokumentationsquelle für das aktuelle Wissen. Es ist eine Eigenschaft der agilen Softwareentwicklung, dass die Dokumentation keine große Priorität hat. Im Zentrum steht der Quelltext und das funktionierende Programm, sowie die Personen, welche es entwickeln. Damit in einem Softwareprojekt auf die detaillierte Dokumentation verzichtet werden kann, müssen die Beteiligten über längere Zeit am Projekt arbeiten und ihr Wissen immer wieder einsetzen/auffrischen bzw. an andere weitergeben. Doch die aktuelle Projektorganisation erlaubt es nicht, Wissen über einen längeren Zeitraum in den Beteiligten zu sammeln. Zu oft wechseln die Entwickler. Entwickler, welche das Projekt verlassen, nehmen ihr Wissen mit, wenn es nicht niedergeschrieben wurde. Neue Entwickler müssen sich am Anfang durch einen Haufen, welcher ständig wächst, an Informationen durcharbeiten, in einer begrenzten Zeit von 3 bzw. 6 Monaten. Ich habe es auch nicht geschafft jede Abschluss-/Studienarbeit zu Saros durchzulesen oder den ganzen Quelltext in jedem Detail zu verstehen. Auch kein anderer Saros-Entwickler, mit welchen ich persönlich gesprochen habe, konnte das für sich behaupten.

3.3.5 Zusammenfassung

- das System ist schwer zu überblicken, es fehlt eine höhere dokumentierte Sicht, welche immer das aktuelle System wiederspiegelt
- direkte Abhängigkeiten zwischen den Teilsystemen
- neue Features können „überall“ integriert werden, wo es eben gerade passt. Klare Grenzen der Teilsysteme gibt es nicht
- bei dem *PicoContainer* wird *Dependency Injection* nicht richtig verwendet
- der *PicoContainer* macht das System intransparent und erschwert das Verändern bzw. Entfernen der einzelnen Komponenten
- die Konfiguration des Systems ist nicht flexibel, um nur Teile davon laufen zu lassen oder auf andere Plattformen (außerhalb von Eclipse) zu portieren
- die Entwickler wechseln oft und arbeiten am System in Teilzeit
- es fehlt eine kompakte und aktuelle Dokumentation, mit einem Projektplan und den wichtigen Entscheidungen zum Projekt

3.4 Ansätze zu Verbesserung

Nach den bisherigen Erkenntnissen lassen sich die Probleme in der Saros Architektur in zwei Bereiche einteilen: Programmstruktur bzw. Abstraktion im System und die Dokumentation. Die Organisation des Projektes ist zum Teil auch ein Problem und ein Grund der mangelnden Qualität, doch diese ist nur schwer bei den z.Z. zur Verfügung stehenden Mitteln zu ändern. Zu Dokumentation im Projekt läuft parallel eine andere Arbeit deshalb konzentriere ich mich im weiteren Verlauf auf die Verbesserung der Programmstruktur.

Um die Softwarearchitektur zu verbessern schlage ich die Einführung einer **dokumentierten Modulsicht** in Saros vor. Diese Modulsicht werde ich in Kapitel 4 vorstellen und in Kapitel 5 zeigen, wie es in Saros integriert werden kann. Dabei verfolge ich folgende Ziele:

- eine einfache, verständliche und überschaubare Softwarearchitektur für Saros, welche einen schnellen und aktuellen Überblick über das ganze System bietet
- höhere Abstraktionsebene
- vereinfachte und leicht zu wartende Dokumentation der Modulsicht auf Basis von Formularen
- Aufteilen des Systems in kleinere unabhängige Module
- die einzelnen Module sollten sich nur über ihre abstrakte Schicht kennen
- den Einstieg für neue Entwickler erleichtern und ihnen einen schnellen Überblick über das System bieten
- ein Entwickler sollte nicht das ganze System von Innen kennen, um daran arbeiten zu können
- mehr Flexibilität das System zu verändern und neue Features einzuführen bzw. diese zu testen
- die Architektur sollte die aktuelle Organisation des Projektes unterstützen

4 Saros verbesserte Architektur

Aufbauend auf den Erkenntnissen und Zielen aus dem letzten Kapitel beschreibe ich hier eine **dokumentierte Modulsicht** für Saros. Dazu werden folgende Schritte gemacht:

- eine Abstraktion des Systems
- eine Modularisierung des Systems
- variable Konstruktion und Konfiguration des Systems
- ein formularbasiertes und leicht zu pflegendes Dokumentationsformat der Architektur

4.1 Abstraktion

Ein Problem in Saros ist der direkte Zugriff auf konkrete Klassen. Diese Vorgehensweise macht das System schwer zu warten. Konkrete Klassen können nicht so einfach durch eine alternative Implementierung ausgetauscht werden. Durch die Sichtbarkeit der öffentlichen Methoden und Attributen einer Implementierung, kann von überall auf die Funktionalität zugegriffen werden, was nicht immer so beabsichtig ist. Weiterhin enthalten die konkreten Klassen die Schnittstellen und die Implementierung in einer Klasse. So hängt oft die Schnittstelle von der Implementierung ab, was zu hohen Änderungsraten der Schnittstelle führt.

Aus diesen Gründen ist es notwendig in das System eine Abstraktionsschicht zwischen den Teilsystemen einzuführen. Anstatt auf konkrete Klassen direkt zuzugreifen, sollte es über eine abstrakte Schnittstelle passieren. Die Funktionalität einer Schnittstelle sollte im Normalfall vor der Implementierung festgelegt werden. Weiterhin sollte die Schnittstelle das Konzept eines Teilsystems beschreiben, nicht dessen Implementierung. Außerdem sollte die Definition der Schnittstelle eines Teilsystems auch als selbstdokumentierender Quelltext dieses Teilsystems dienen.

Das Ziel der Abstraktionsschicht ist Saros zu vereinfachen. Die Funktionalität eines Teilsystems sollte vollständig anhand der Schicht und der dazugehörigen Dokumentation verständlich sein, die Implementierung muss nicht bekannt sein. Die einzelnen Teilsysteme sollten möglichst klein und leicht überblickbar gehalten werden, damit es einfacher ist sich darin schnell einzuarbeiten. Weiterhin sollte die Implementierung einer Schnittstelle austauschbar sein (alternative Implementierung, externe Lösung, Mock-Implementierung, nächste Version), ohne dass andere Teilsysteme dazu verändert werden müssen.

4.2 Modularisierung

Teilsysteme, welche über abstrakte Schnittstellen miteinander verbunden sind, können als Module betrachtet werden. Es gibt eine Reihe von Konzepten für modulbasierte Systeme auf Basis von Java. Das zu Zeit verbreitetste ist der OSGi Standard, auf welchem auch die

Eclipse Plattform aufsetzt. Weitere Konzepte sind die JSR 294[30], JSR 277[28] und JSR 291[29]. Diese Java Specification Requests zielen darauf ab, Modularisierung direkt in Java zu unterstützen, doch das passiert frühestens in Java 8.

Für die Umsetzung der Module in Saros, habe ich mich entschieden ein eigenes Konzept zu verwenden, welches eine Vereinfachung des OSGi Standards darstellt. Saros gleich auf OSGi umzustellen würde viel Arbeit, Zeit und zusätzliches Wissen von den Entwicklern erfordern. Doch zuerst muss Saros sauber in Module mit abstrakten Zugriffsschichten aufgeteilt werden und diese müssen von der jeweiligen Implementierung unabhängig werden. Wenn Saros nur noch aus solchen Modulen und zustandslosen Hilfsklassen besteht, kann der nächste Schritt der echten Portierung auf OSGi oder auf das Modulkonzept von Java 8 gemacht werden.

4.2.1 Basis

OSGi verwendet als Basis *Bundles* und *Services*. Bundles stellen eine bestimmte Funktionalität dar, sie können auch als Module betrachtet werden. Der Zugriff auf die Funktionalität eines Bundles läuft über Services, welche eine abstrakte Zugriffsschicht darstellen. Jedes Bundle kann beliebig viele Services anbieten. OSGi selber ist nur eine Definition eines Systems für Module, konkrete Systeme werden mit *OSGi-Frameworks* realisiert.

Von OSGi habe ich drei abstrakte Konzepte abgeleitet: *ISaros* (OSGi-Framework), *ISarosModule* (Bundle) und *ISarosService* (Service).

- **ISaros** - repräsentiert ein Saros-System welches Module und ihre Services anbietet. Auf einer JVM können mehrere *ISaros*-Instanzen laufen, z.B. zu Testzwecken. Doch die jeweiligen Module sind an genau eine *ISaros*-Instanz gebunden.
- **ISarosModule** - repräsentiert eine konkrete Komponente über eine abstrakte Schicht. Ein *ISarosModule* kann beliebig viele Services anbieten. Weiterhin kann ein *ISarosModule* mehrere Implementierungen haben, wobei zur Laufzeit nur eine davon in einer *ISaros*-Instanz aktiv sein kann.
- **ISarosService** - repräsentiert eine konkrete Funktionalität eines Moduls über eine Schnittstelle nach dem Single-Responsibility Prinzip. Ein Modul hat nur einen *ISarosService* vom gleichen Typ. Es gibt zwei Arten von *ISarosService*: entweder gehört es nur zu einem Modul und wird in seiner abstrakten Schicht definiert oder es ist global, in der Systemschicht definiert und kann von verschiedenen Modulen angeboten werden. Ein Beispiel für einen globalen Service wäre z.B. das Sammeln von Informationen für das Feedback.

Zusätzlich zu den Konzepten aus OSGi führe ich noch zwei Abstraktionen hinzu: *IObject* und *IValue*. Diese Konzepte sollten helfen die Modulschicht noch feiner zu strukturieren.

- **IObject** - repräsentiert ein vom Modul erzeugtes Datenobjekt. So ein Datenobjekt kann nicht direkt oder von Außen erzeugt werden, sondern muss über einen Service bezogen werden. Der Zustand des Objektes kann verändert werden.
- **IValue** - repräsentiert einen bestimmten Wert und ist nach dem Erzeugen nicht mehr veränderbar. *IValues* können außerhalb der Module implementiert und zu

Parameterübergabe an die Services verwendet werden. Das *IValue* folgt den Value-Object Entwurfsmuster und soll das System robuster gegen Änderungen machen.

Das Konzept ist als UML-Diagramm in Abbildung 4.1 zu sehen.

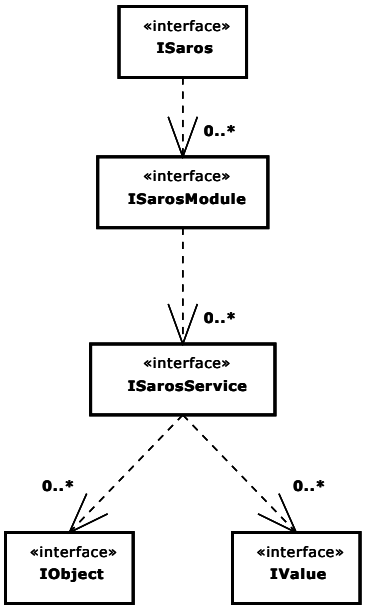

Abbildung 4.1 *Basis der verbesserten Architektur für Saros*

Diese Basis kann auch als eine Analogie zum Buch angesehen werden. Es gibt das Buch, es gibt Kapitel und Unterkapitel. Wenn eine bestimmte Funktionalität gesucht wird, kann über die Dokumentation (sehe Kapitel 4.4) nachgesehen werden, wo diese sich befindet. So können die Entwickler sich schneller im System zurechtfinden.

- **ISaros (Buch)**
  - **IModule1 (Kapitel)**
    - **IService1 (Unterkapitel)**
    - **IService2 (Unterkapitel)**
  - **IModule2**
    - **IService1 (Unterkapitel)**
    - **IService2 (Unterkapitel)**

4.2.2 Basis Implementierung

Die vorher vorgestellten Abstraktionen werden in der Implementierung jeweils in Form eines Interfaces dargestellt. Das `ISaros`-Interface (Listing 4.1) bietet Möglichkeiten die installierten Module und Services abzufragen.

```
/**
 * The representation of the Saros system
 */
public interface ISaros {

    /**
     * Lists all modules
     *
     * @return returns a unmodifiable List of all installed modules in
     *         this Saros system
     */
    public List<ISarosModule> listModules();

    /**
     * Module access
     *
     * @param cls
     *            module interface class
     * @return returns the module, that implements the specified module
     *         interface
     */
    public <T extends ISarosModule> T getModule(Class<T> cls);

    /**
     * Service access
     *
     * @param cls
     *            service interface class
     * @return returns the first service, that implements
     *         the specified service interface, or null
     *         if no such service available
     */
    public <T extends ISarosService> T getService(Class<T> cls);

    /**
     * Service access
     *
     * @param cls
     *            service interface class
     * @return returns all services, which implements the specified
     *         service interface, or a empty list if no such
     *         service available
     */
    public <T extends ISarosService> List<T> getServices(
        Class<T> cls);

    /**
     * Service access
     *
     * @param module_cls
     *            module interface class
     * @param service_cls
     *            service interface class
     * @return returns the service from the module, that implements the
     *         specified service interface, or null if no such service
     *         available
```

```
     */
    public <T1 extends ISarosModule,
            T2 extends ISarosService> T2 getService(
                                                Class<T1> module_cls,
                                                Class<T2> service_cls);
}
```

Listing 4.1 *ISaros*

Das `ISarosModule`-Interface (Listing 4.2) bietet Möglichkeiten die vorhandenen Services in einem Modul abzufragen.

```
/**
 * The representation of a module
 */
public interface ISarosModule {

    /**
     * Lists all services
     *
     * @return returns a unmodifiable List of all available services in
     *         this Saros system
     */
    public List<ISarosService> listServices();

    /**
     * Lists all services
     *
     * @param cls
     *            service interface class
     * @return returns true, if the module has a service, that
     *         implements the specified service interface
     */
    public <T extends ISarosService> boolean supportServices(
                                                        Class<T> cls);

    /**
     * Service access
     *
     * @param cls
     *            service interface class
     * @return returns the service, that implements the specified
     *         service interface, or null if no such service
     *         available
     */
    public <T extends ISarosService> T getService(Class<T> cls);
}
```

Listing 4.2 *ISarosModule*

Die `ISarosService`-, `IObject`- und `IValue`-Interfaces (Listings 4.3, 4.4 und 4.5) dienen als Basis für die Definition eigener Interfaces mit der benötigten Funktionalität für ein konkretes Modul.

```
/**
 * The representation of a module service
 */
public interface ISarosService {

}
```

Listing 4.3 *ISarosService*

```
/**
 * The representation of a module object
 */
public interface IObject {

}
```

Listing 4.4 *ISarosObject*

```
/**
 * The representation of a module value
 */
public interface IValue {

}
```

Listing 4.5 *ISarosValue*

4.2.3 Verwendung

Über diese Basis können jetzt ganz einfach die Module und die Services abgefragt und verwendet werden, wie im Listing 4.6. Das Beispiel zeigt wie die Unterstützung von Services durch ein Modul abgefragt werden kann und wie im negativen Fall darauf reagiert werden kann.

```
// Module 1
public interface IModule1 extends ISarosModule { /* ... */ }

// Service 1 of Module 1
public interface IM1Service1 extends ISarosService {
    public String getString();
}

/* ... */

ISaros saros;

// get IModule1 from Saros
IModule1 module1 = saros.getModule(IModule1.class);
```

```
// test if IModule1 has the IM1Service1
if (module1.supportServices(IM1Service1.class)) {

    // use service
    IM1Service1 service1 = module1
        .getService(IM1Service1.class);

    String string = service1.getString();
} else {
// do something other
}
```

Listing 4.6 *ISarosService*

4.3 Konstruktion und Konfiguration

Die Konstruktion eines *ISaros*-Systems basiert auf dem Builder-Pattern. Dazu wird ein Interface definiert, über welches ein neues *ISaros*-System erstellt werden kann, sehe Abbildung 4.2 und Listing 4.7.

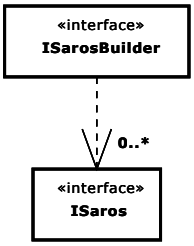

Abbildung 4.2 *Abstraktion eines Builders für ISaros*

```
/**
 * Builder for Saros
 */
public interface ISarosBuilder {

    /**
     * Query for the availability
     *
     * @return remaining count of possible new ISaros creations
     */
    public int getRemainingCount();

    /**
     * Create a new ISaros system
     *
     * @return a new ISaros System
     * @throws SarosBuilderException
     *             when it is not possible to create
     *             a new ISaros System
```

```
     */
    public ISaros newSaros() throws SarosBuilderException;
}
```

Listing 4.7 *ISarosBuilder*

Über dieses Interface sollte es möglich sein verschiedene *ISaros*-Instanzen mit verschiedenen Konfigurationen zu erstellen. Auf einer JVM sollte es so möglich werden, mit dem selben oder verschiedenen Buildern mehrere *ISaros*-Instanzen zu erstellen, z.B. für Testzwecke oder verschiedene Konfigurationen. Über eine konkrete Implementierung eines Builders sollte es auch möglich sein, den Buildprozess zu konfigurieren.

4.4 ModuleDoc (Dokumentation)

Für die Dokumentation des vorgestellten Systems habe ich ein eigenes Format auf Basis von XML erstellt. Die Überlegung dahinter war, ein sehr einfaches und an das Saros-Projekt anpassbares Formular zu erstellen. Ich habe mich dabei an dem Schablonendokumentation-Entwurfmuster([3], S. 159) orientiert. Das Formular sollte einfach sein, wenig Zeit zum Ausfühlen brauchen und von den Entwicklern akzeptiert werden. Weiterhin kann es durch das Saros-Projekt erweitert werden, um weiteren Anforderungen zu genügen. Das es auf XML basiert, lässt es sich später in andere Formate konvertieren, z.B. *xADL* oder *arc42*. So müssen die Entwickler diese Formate nicht kennen.

Jedes Modul wird in einer eigenen Datei beschrieben. Die XML-Dateien werden im Projektverzeichnis unter dem Pfad `docs/modules` und dem Namen des Moduls als Dateiname mit der Endung .xml abgelegt und sind damit jederzeit in der IDE verfügbar, sowie von der Versionsverwaltung erfasst. Die Beschreibungen der Module sind nicht an den Java-Quelltext, und damit die Implementierung, gebunden. Somit können die Module unabhängig und noch vor der Implementierung definiert werden. Mit XSL/T[12] können diese Beschreibungen in verschiedene Formate zu Betrachtung konvertiert werden, wie z.B. HTML, PDF oder Wiki. Als Namen für dieses System habe ich **ModuleDoc** gewählt, als Analogie zu JavaDoc. Im Listing 4.8 ist ein Beispiel für die Struktur des Formulars zu sehen.

```
<?xml version="1.0" encoding="UTF-8" ?>

<?xml-stylesheet type="text/xsl"
                 href="./style/saros-modules.xslt" ?>

<saros-module version="1.0"
              xml:lang="de">

  <name>ICoreModule</name>

  <description version="1.0.0">
    ICoreModule ist das Hauptmodul des Systems
    und muss immer in einem produktiven System verfügbar sein
  </description>
```

[12] Extensible Stylesheet Language - Transformationssprache für XML-Dokumente

```
  <description version="1.1.0">
    ICoreModule wurde um IUserService erweitert
  </description>

  <services>
    <service>
      <name>IUserService</name>
      <description version="1.0.0">
        Über den IUserService kann die Verwaltung der Anwenderkonten
        im System durchgeführt werden. Damit ist es möglich
        die vorhandenen Anwenderkonten abzufragen, neue anzulegen oder
        zu löschen.
      </description>
    </service>
  </services>

  <objects>
    <object>
      <name>IUserObject</name>
      <description version="1.0.0">
        IUserObject repräsentiert ein Anwenderkonto im System
      </description>
    </object>
  </services>

  <values>
    <!-- Analog zu objects -->
  </values>

</saros-module>
```

Listing 4.8 *XML-Formular für die Beschreibung der Module*

Mit dieser Vorgehensweise sollte einerseits der Aufwand der Erstellung und Pflege der Architekturdokumentation reduziert werden, andererseits sollte ein implementierungsunabhängiger Überblick über das System gegeben werden. Damit sollte für neue Entwickler ermöglicht werden nachzuvollziehen, was und wie vorher geplant war. Weiterhin sollte *ModuleDoc* zu Kommunikation anregen. Immer wenn bei der Entwicklung die bestehende Schicht nicht mehr ausreicht, sollten die betroffenen Entwickler das zusammen mit dem Projektleiter besprechen, die Schicht ändern und in ModuleDoc aufschreiben.

Mit der Angabe der `xml-stylesheet` Processing Instruction am Anfang der XML-Datei können die Modul-Definitionen im Webbrowser direkt betrachtet werden. Für eine dauerhafte Konvertierung in andere Formate wird ein Apache Ant-Skript verwendet. Im Anhang A befindet sich die Referenz zu ModuleDoc.

5 Integration in Saros

Dieses Kapitel beschreibt, wie die im vorherigen Kapitel definierte Architektur in das bestehende Saros integriert werden kann. Weil Saros ständig weiterentwickelt und verändert wird, muss diese Integration am lebenden System passieren, um die anderen Entwickler nicht an ihrer Arbeit zu behindern. Deshalb wird die Integration in 5 Phasen unterteilt:

1. Eine *ISaros*-Schicht über das System legen
2. Das bestehende Saros als ganzes zu einem großen *ICoreModule* Modul machen
3. Die Saros-Konstruktion auf das Saros-Builder System umstellen
4. Saros und den Eclipse Plugin-Mechanismus trennen
5. Einzelne Module aus dem *ICoreModule* rauslösen, bis das *ICoreModule* aufgelöst ist

Die ersten 4 Phasen sind schnell umzusetzen, dass wird in Kapitel 5.1 beschrieben. Danach ist eine neue Schicht um das bestehenden System gelegt worden und das Interface `ISaros` kann genutzt werden. Die Phase 5, ab Kapitel 5.2, ist dagegen aufwendig, weil hier die einzelnen Module definiert und gegen andere abgegrenzt werden müssen.

5.1 Eine neue Fassade für Saros

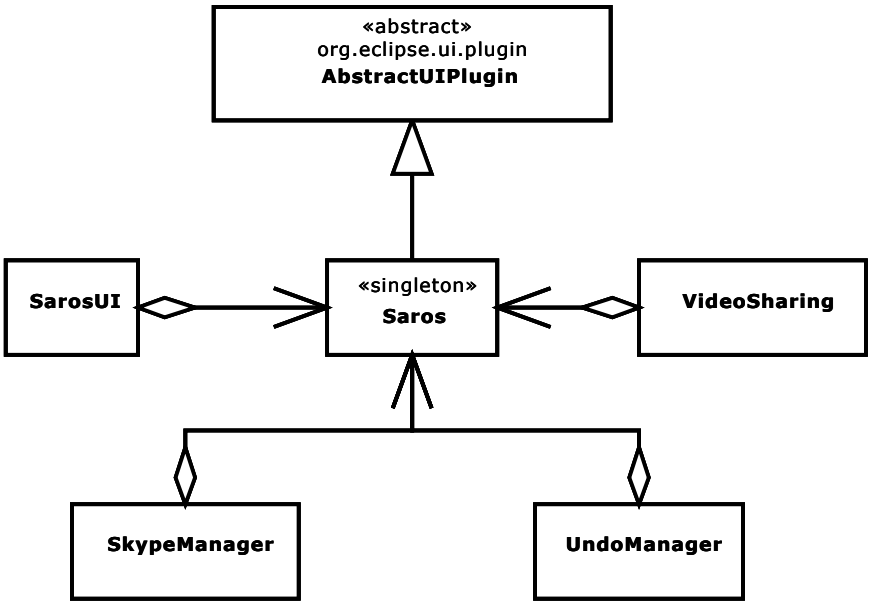

Abbildung 5.1 *Direkter Zugriff auf die zentrale Klasse Saros*

Zu Zeit sieht die Architektur so aus, dass verschiedene Teile von Saros direkt auf die Klasse `Saros` zugreifen, welche selber zugleich ein Singleton (innerhalb von Eclipse) ist und von AbstractUIPlugin erbt, sehe Abbildung 5.1.

5.1.1 Phase 1 - ISaros Schicht

Jetzt werden wir zwischen der Klasse `Saros` und den zukünftigen Modulen eine abstrakte Schicht einführen. Dazu werden zusätzliche Packages in Saros definiert:

- `dpp.modules` - hier befindet sich die Saros Architekturschicht, wie in Kapitel 4 definiert, unter anderem die `ISaros` und `ISarosModule` Interfaces
- `dpp.modules.internal` - interne Klassen für `dpp.modules`

Um die Arbeit mit den `ISaros` und `ISarosModule` Interfaces zu vereinfachen, erstellen wir dafür abstrakte Klassen, welche die Funktionalität der Interfaces implementieren, dargestellt in Abbildung 5.2.

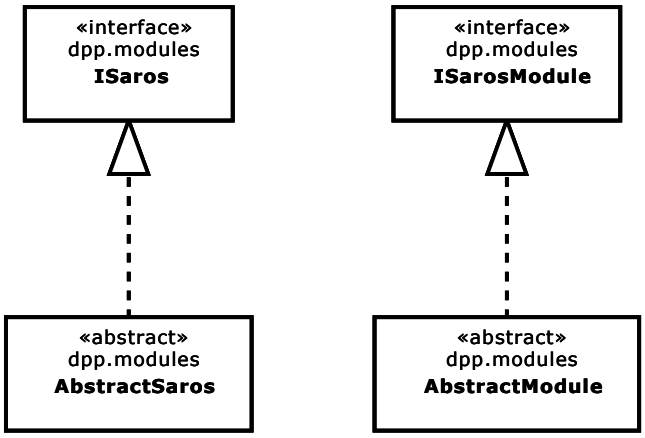

Abbildung 5.2 *Abstrakte Implementierung von ISaros und ISarosModule*

Zusätzlich führen wir in den beiden abstrakten Klassen jeweils eine interne Funktion ein, um die jeweiligen Listen von Modulen und Services zu verändern, Listing 5.1 und 5.2

```
/**
 * Lists all modules
 *
 * @return returns a modifiable List of all installed modules
 *         in this Saros system
 */
protected List<ISarosModule> listModulesModifiable() {
    return modules;
}
```

Listing 5.1 *AbstractSaros mit veränderbarer Modul-Liste*

```
/**
 * Lists all services
 *
 * @return returns a modifiable List of all available services
 *         in this Saros system
 */
protected List<ISarosService> listServicesModifiable() {
    return services;
}
```

Listing 5.2 *AbstractModule mit veränderbarer Service-Liste*

Als nächsten Schritt erstellen wir einen *ISaros*-Wrapper für das bisherige Saros. Dazu wird eine neue Klasse `SarosClassicImpl` definiert, welche das Interface `ISaros` implementiert und intern eine Referenz auf die Klasse `Saros` anlegt. `SarosClassicImpl` ist für das restliche System unsichtbar und kann nur über das Interface `ISaros` benutzt werden. Deshalb wird dieses Klasse in einem internen Package abgelegt, Abbildung 5.3.

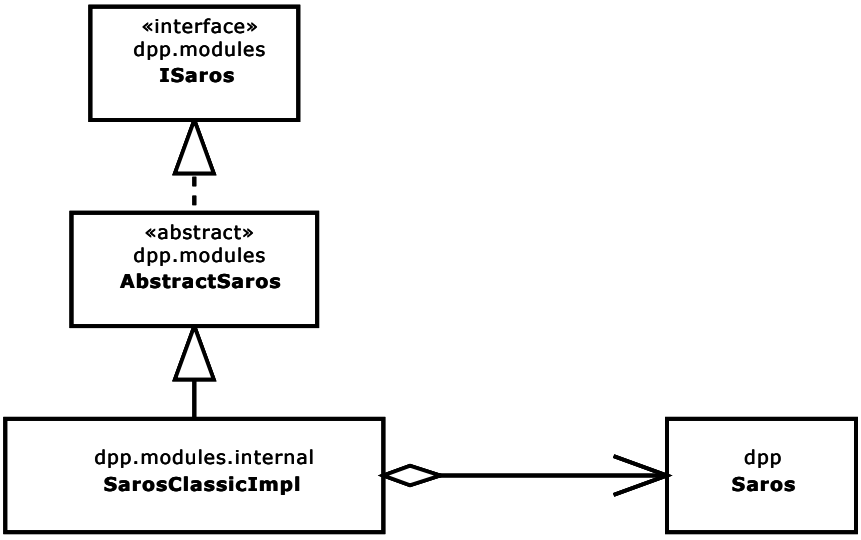

Abbildung 5.3 *SarosClassicImpl Wrapper für Saros*

Um den Zugriff auf `ISaros` im bisherigen System zu vereinfachen fügen wir eine Referenz auf `ISaros` in `Saros` ein, Listing 5.3. Das sollte keine dauerhafte Lösung sein, sondern nur als Übergang, bis das System vollständig modularisiert ist. Die Referenz wird von `SarosClassicImpl` gesetzt.

```
public class Saros /* ... */{

    protected ISaros isaros = null;

    public ISaros getISaros() {
        return isaros;
    }
```

```
    public void setISaros(ISaros saros) {
        this.isaros = saros;
    }
}
```

Listing 5.3 *ISaros in Saros verlinken, um den Zugriff zu erleichtern*

5.1.2 Phase 2 - ICoreModule

Jetzt folgt das erste Modul, *ICoreModule*, welches einfach eine Repräsentation des vorhandenen Systems darstellt und im Package `dpp.modules.core` abgelegt wird. Das *ICoreModule* ist als Schnittstelle/Übergangslösung zwischen dem aktuellem und einem modularisierten System gedacht. Um auf aktuelle System zuzugreifen, wird ein *ISarosClassicService* definiert, welcher nur dafür zuständig ist, den Zugriff auf die Klasse `Saros` zu ermöglichen, Abbildung 5.4 und Listing 5.4.

- `dpp.modules.core` - hier befindet sich die Definition des ICoreModules und seiner Services.
- `dpp.modules.internal` - interne Klassen für `dpp.modules.core`

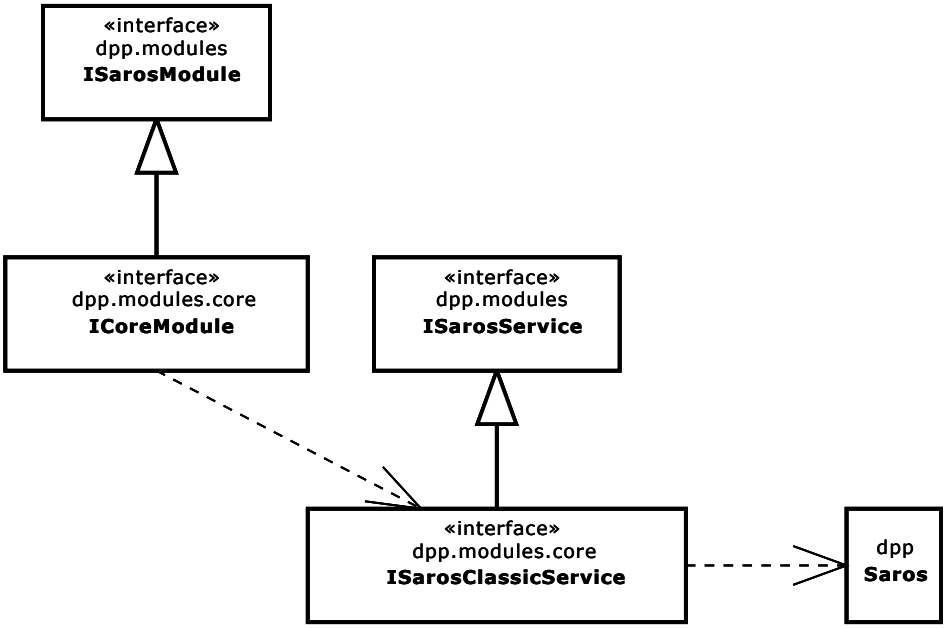

Abbildung 5.4 *Das ICoreModule mit ISarosClassicService*

```
public interface ISarosClassicService extends ISarosService {

    public Saros getSarosClassic();

}
```

Listing 5.4 *ISarosClassicService als Zugriff auf die vorhandene Saros Klasse*

Die Implementierung des *ICoreModule* wird im Package `dpp.modules.core.internal` als `SarosClassicCoreModuleImpl` abgelegt und zu `SarosClassicImpl`m> hinzugefügt. An dieser Stelle sind wir so weit Saros mit der neuen Schicht zu starten, wie in Listing 5.5.

```
ISaros isaros = new SarosClassicImpl();

ICoreModule saros_core = isaros.getModule(ICoreModule.class);

ISarosClassicService saros_classic = saros_core
    .getService(ISarosClassicService.class);

Saros saros = saros_classic.getSarosClassic();
```

Listing 5.5 *Starten von Saros über die neue Schicht*

Mit der Erstellung des *ICoreModule* müssen wir dieses auch dokumentieren. Dazu legen wir unter `docs/modules` eine neue Datei `ICoreModule.xml` an und beschreiben das Modul wie in Kapitel 4.4 definiert wurde. `ICoreModule.xml` ist im Anhang B zu sehen.

5.1.3 Phase 3 - Saros-Builder

Im vorherigen Beispiel haben wir `SarosClassicImpl` direkt erzeugt. Jetzt stellen wir diesen Prozess auf das `ISarosBuilder` Interface um.

- `dpp.builder` - hier befindet sich das Buildsystem der Saros-Architekturschicht, wie in Kapitel 4 definiert, unter anderem das `ISarosBuilder` Interface und die `SarosBuilderException` Klasse.

Im selben Package legen wir die öffentliche Builder-Implementierung für `SarosClassicImpl` ab, namens `SarosBuilderClassic`, Listing 5.6.

```
public class SarosBuilderClassic implements ISarosBuilder {

    protected SarosClassicImpl sarosClassic = null;

    public SarosBuilderClassic() {
        super();
    }

    public int getRemainingCount() {
        return sarosClassic != null ? 0 : 1;
    }

    public ISaros newSaros() throws SarosBuilderException {
        if (sarosClassic == null) {
            sarosClassic = new SarosClassicImpl();
            return sarosClassic;
        }

        throw new SarosBuilderException(
```

```
                "Only one new instance of ISaros can be created" +
                " with the SarosBuilderClassic!");
        }
    }
```

Listing 5.6 *Builder für SarosClassicImpl*

Jetzt sind wir in der Lage die Konstruktion von Saros über das Builder Interface zu bewerkstelligen, wie in Listing 5.7 dargestellt.

```
ISaros isaros = null;

try {
    ISarosBuilder saros_builder = new SarosBuilderClassic();
    saros = saros_builder.newSaros();
} catch (SarosBuilderException e) {
    log.error("Couldn't create ISaros!", e);
}

ICoreModule saros_core = isaros.getModule(ICoreModule.class);
```

Listing 5.7 *Starten von Saros über einen Builder*

5.1.4 Phase 4 - Eclipse Plugin-Mechanismus abtrennen

Ein letztes Problem bleibt noch. Die bestehende Klasse `Saros` erbt von der `AbstractUIPlugin` Klasse des Eclipse Plugin-Mechanismus. Das macht es schwierig alternative Implementierungen für `ISaros` zu entwickeln. Deshalb gehen wir nach dem Single-Responsibility Principle vor und trennen den Plugin-Mechanismus von Saros ab. Der erste Schritt wäre die Klasse `Saros` nicht mehr vom `AbstractUIPlugin` erben zu lassen. Selbstverständlich führt das zu einer Reihe von Fehlern, weil `Saros` und darauf zugreifende Klassen die Funktionalität aus `AbstractUIPlugin` verwendet haben. Um das weiterhin zu gewährleisten, wird Delegation verwendet. `AbstractUIPlugin` wird zu `Saros` extern hinzugefügt und die benötigten Funktionen werden nachgebildet, sowie alle `@Override` entfernt. Das ist in im Listing 5.8 dargestellt.

```
// 1. Vererbung entfernen
public class Saros /* extends AbstractUIPlugin */{

/* ... */

// 2. Alle @Override deaktivieren
    // @Override
    public void start(BundleContext context) throws Exception {

    // @Override
    public void stop(BundleContext context) throws Exception {

/* ... */

// 3. AbstractUIPlugin extern setzen
```

```
    protected AbstractUIPlugin eclipse_plugin = null;

    public AbstractUIPlugin getEclipsePlugin() {
        return eclipse_plugin;
    }

    public void setEclipsePlugin(AbstractUIPlugin plugin) {
        this.eclipse_plugin = plugin;
    }

// 4. Delegation der aus AbstractUIPlugin benötigten Funktionen

    public IPreferenceStore getPreferenceStore() {
        return eclipse_plugin.getPreferenceStore();
    }

    public Bundle getBundle() {
        return eclipse_plugin.getBundle();
    }

    public ILog getLog() {
        return eclipse_plugin.getLog();
    }

    public IPath getStateLocation() {
        return eclipse_plugin.getStateLocation();
    }
```

Listing 5.8 *Modifikationen an der Klasse Saros, um Plugin-Mechanismus abzutrennen*

Damit haben wir den Eclipse Plugin-Mechanismus von der Definition der Klasse `Saros` getrennt. Selbstverständlich wird noch der Zugriff auf eine Instanz von `AbstractUIPlugin` in der Klasse `Saros` benötigt, aber das Saros-Plugin muss nicht mehr über die `Saros` Klasse gestartet werden. Um Saros in Eclipse zu starten, legen wir eine neue Klasse `SarosEclipsePlugin` an.

- `dpp.eclipseplugin` - hier liegt die neue Eclipse Plugin-Klasse, `SarosEclipsePlugin`.

Um das Plugin an die Klasse `Saros` zu übergeben und Saros richtig zu initialisieren, führen wir einen neuen Service in das *ICoreModule* ein: *IEclipsePluginService*, Abbildung 5.5 und Listing 5.9. Die Beschreibung des Services wird `ICoreModule.xml` hinzugefügt.

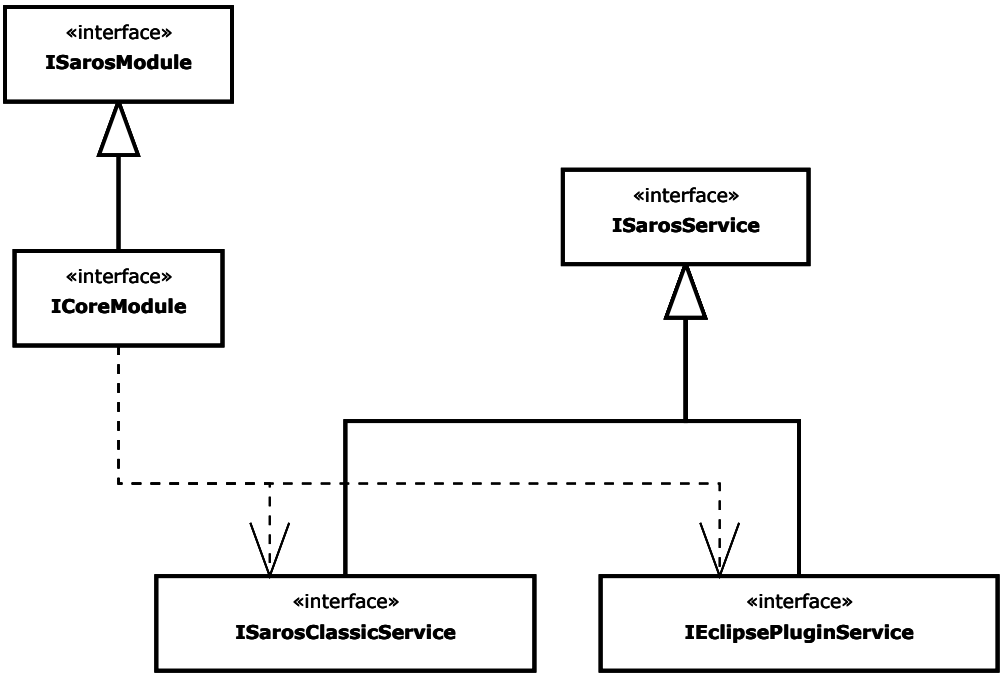

Abbildung 5.5 *ICoreModule mit dem IEclipsePluginService*

```
public interface IEclipsePluginService extends ISarosService {

    public void init(AbstractUIPlugin eclipse_plugin);

    public void start(BundleContext context) throws Exception;

    public void stop(BundleContext context) throws Exception;

}
```

Listing 5.9 *IEclipsePluginService*

Jetzt wird es möglich in `SarosEclipsePlugin` ein ISaros-System zu erzeugen und bei Bedarf das Plugin zu übergeben, sehe Listing 5.10. In Abbildung 5.6 ist der neue Plugin-Aufbau dargestellt.

```
public class SarosEclipsePlugin extends AbstractUIPlugin {

    protected static Logger log =
                Logger.getLogger(SarosEclipsePlugin.class.getName());

    protected ISaros saros = null;

    @Override
    public void start(BundleContext context) throws Exception {
```

```
        super.start(context);

        try {
            ISarosBuilder saros_builder = new SarosBuilderClassic();
            saros = saros_builder.newSaros();
        } catch (SarosBuilderException e) {
            log.error("Couldn't create ISaros!", e);
        }

        ICoreModule saros_core = saros.getModule(ICoreModule.class);

        if (saros_core.supportServices(IEclipsePluginService.class)) {

            IEclipsePluginService plugin_service = saros_core
                .getService(IEclipsePluginService.class);

            plugin_service.init(this);
            plugin_service.start(context);
        }
    }

    @Override
    public void stop(BundleContext context) throws Exception {

        ICoreModule saros_core = saros.getModule(ICoreModule.class);

        if (saros_core.supportServices(IEclipsePluginService.class)) {

            IEclipsePluginService plugin_service = saros_core
                .getService(IEclipsePluginService.class);

            plugin_service.stop(context);
        }
    }
}
```

Listing 5.10 *SarosEclipsePlugin*

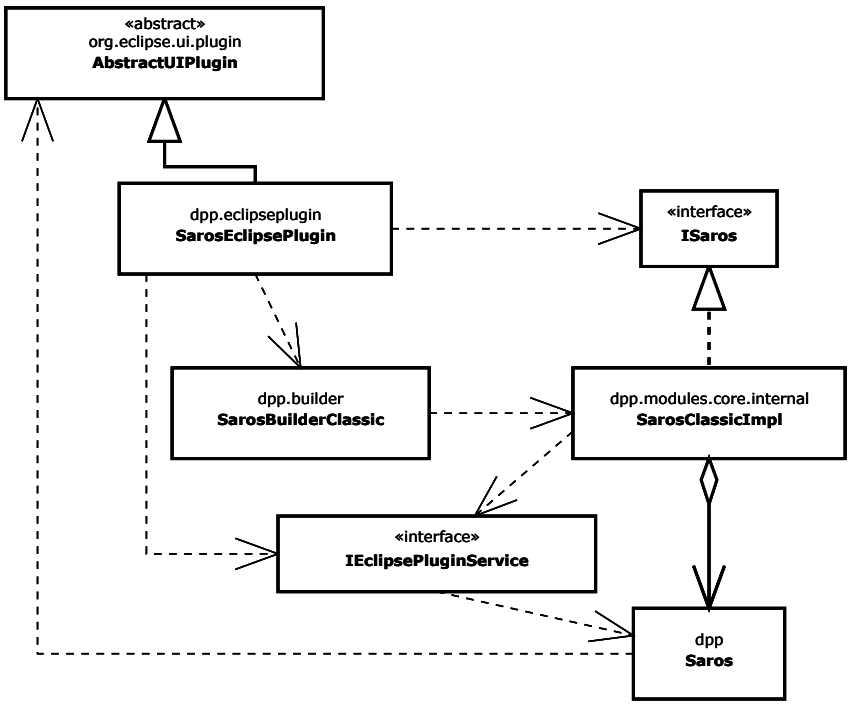

Abbildung 5.6 *Der neue Plugin-Aufbau mit SarosEclipsePlugin*

Zuletzt muss nur noch in `plugin.xml` der Activator von `dpp.Saros` auf `dpp.eclipseplugin.SarosEclipsePlugin` umgestellt werden und schon kann Saros über verschiedene Implementierung des Plugin-Mechanismus und eines ISaros-Systems gestartet werden.

5.1.5 Phase 1-4 - Zusammenfassung

Das Ziel der vorherigen Phasen war, eine abstrakte Schicht um Saros zu legen, mit einem großen Modul für das bestehende System, um alternative Implementierungen von Saros zu ermöglichen. Das bestehende System wird dadurch nicht beeinträchtigt und kann weiterentwickelt werden. Parallel dazu können einzelne Module aus ICoreModule rausgezogen werden. Im weiteren Verlauf dieses Kapitels wird auf Basis dieser Schicht gearbeitet.

5.2 Phase 5 - Inkrementelle Integration

Im vorherigen Abschnitt wurden die Grundlage der Modularisierung für Saros gelegt. Wir haben ein großes Modul, das *ICoreModule*, wo die bisherige Funktionalität zusammengefasst ist. Jetzt können wir einzelne Module definieren, abgrenzen und aus *ICoreModule* rausziehen. Das Ziel ist *ICoreModule* vollständig aufzulösen. Dazu gehen wir folgend vor:

1. Funktionalität identifizieren
2. Modulschicht definieren

3. Abhängigkeiten auflösen

Als Beispiel dafür wird folgend das *IAccountManagerModule* aus *ICoreModule* rausgelöst. Es wird für die Verwaltung der XMPP-Konten im Saros zuständig sein.

5.2.1 Funktionalität identifizieren

Die bisherige Verwaltung der XMPP-Konten wurde in Saros von zwei Klassen übernommen: `XMPPAccountStore` führt eine Liste der Konten und `XMPPAccount` repräsentiert ein XMPP-Konto, welche in `dpp.accountManagement` zu finden sind. Auf die beiden Klassen wird im System direkt zugegriffen. Für die Erstellung einer Schicht der beiden Klassen habe ich es zuerst mit der Refactoring-Funktion „Extract Interface...“ von Eclipse versucht, aber das neue Interface enthielt einfach alle öffentlichen Methoden einer Klasse. Doch nicht alle öffentlichen Methoden wurden auch außerhalb verwendet. Um herauszufinden, welche Funktionalität überhaupt vom System benötigt wird, habe ich die Funktionen in beiden Klassen kurz auskommentiert und eine Liste der benötigten Methoden erstellt, für `XMPPAccountStore` und `XMPPAccount`. Die erste Liste habe ich gleich in Kategorien eingeteilt:

- für `XMPPAccountStore`
    - Konto Verwaltung
        - contains(JID)
        - contains(String, String)
        - hasActiveAccount()
        - getActiveAccount()
        - setAccountActive(XMPPAccount)
        - getAllAccounts()
        - getAccount(int)
        - createNewAccount(String, String, String)
        - changeAccountData(int, String, String, String)
        - deleteAccount(XMPPAccount)
    - Sicherung der Daten
        - loadAccounts()
        - saveAccounts()
        - flush()
    - Abfrage aller Domains der Konten
        - getDomains()
- für `XMPPAccount`
    - getId()
    - getUsername()
    - getPassword()
    - getServer()

5.2.2 Modulschicht definieren

Nachdem die Funktionalität des *IAccountManagerModule* feststeht, wird das Modul, die Services und die Objekte definiert. Ich habe mich entschieden die Funktionalität von `XMPPAccountStore` in drei Services einzuteilen, nach dem „Interface Segregation“ Prinzip, um die Übersicht zu vereinfachen. Die Aufteilung erfolgt nach den Kategorien, wie im vorherigen Abschnitt definiert. Das hätte den Vorteil, dass z.B. ein Service für die Persistenz auf verschiedene Arten (Festplatte, Datenbank, Null) implementiert werden kann. Folgende Elemente gehören jetzt zu dem *IAccountManagerModule*:

- `dpp.modules.accountManagement` - Package für das *IAccountManagerModule*
- `IAccountManagerModule` - repräsentiert das *IAccountManagerModule*
- `IAccountService` - Service für die Verwaltung der Konten
- `IAccountStoreService` - Service für die Persistenz der Konto-Daten
- `IAccountDomainListService` - Service für eine Liste mit den Domainnamen der Konten
- `IAccountObject` - ein Objekt, welches ein Konto im System repräsentiert

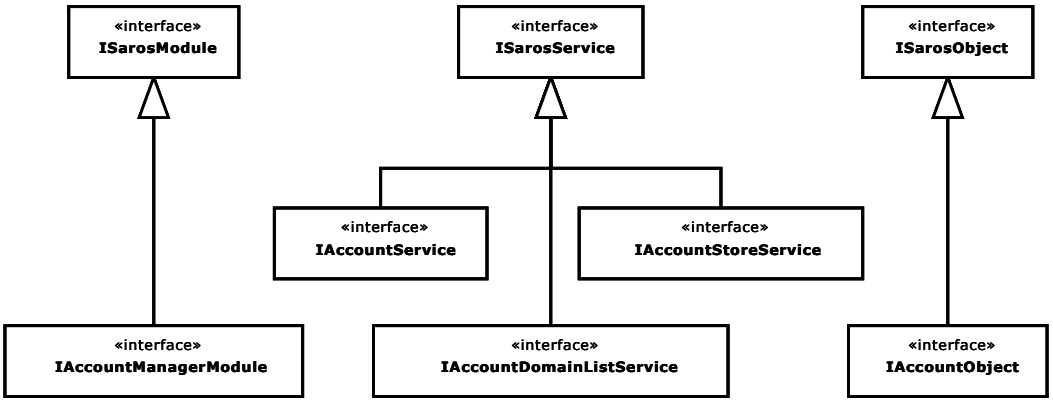

Abbildung 5.7 *Der neue Plugin-Aufbau mit SarosEclipsePlugin*

Zu Dokumentation des *IAccountManagerModule* wird die `IAccountManagerModule.xml` Datei angelegt, ist im Anhang B zu sehen.

5.2.3 Abhängigkeiten auflösen

Zu den Klassen `XMPPAccountStore` und `XMPPAccount` werden jetzt die Service- bzw. Objekt-Schnittstellen hinzugefügt. Damit bekommen wir eine erste Implementierung des *IAccountManagerModule* und können es in der Klasse `SarosClassicImpl` nutzen, wobei wir einfach auf die vorhandenen Instanzen von XMPPAccountStore im PicoContainer zugreifen.

Der nächste Schritt ist jede Referenz oder Verwendung auf `XMPPAccountStore` bzw. `XMPPAccount` durch die neue Schicht zu ersetzen. Dazu können wir die Referenz auf `ISaros` aus `Saros` nutzen, wie in Abschnitt 5.1.1 beschrieben. Oder `ISaros` wird in die beteiligten Klassen injiziert.

```
// XMPPAccountStore accountService;
IAccountsService accountService;

/* ... */

Saros saros = ...;

accountService = saros.getISaros().getService(IAccountService.class);

/* ... */

// XMPPAccount account = accountService.getActiveAccount();
IAccountObject account = accountService.getActiveAccount();
```

Listing 5.11 *AbstractSaros mit veränderbarer Modul-Liste*

Wurden alle Referenzen auf die `XMPPAccountStore` und `XMPPAccount` Klassen aufgelöst, können diese umbenannt oder verschoben werden, um sicher zu stellen, dass sie nicht mehr direkt verwendet werden.

5.3 Weitere Module

Für die Modularisierung von Saros, schlage ich eine Modulstruktur vor, wie in Abbildung 5.8 dargestellt.

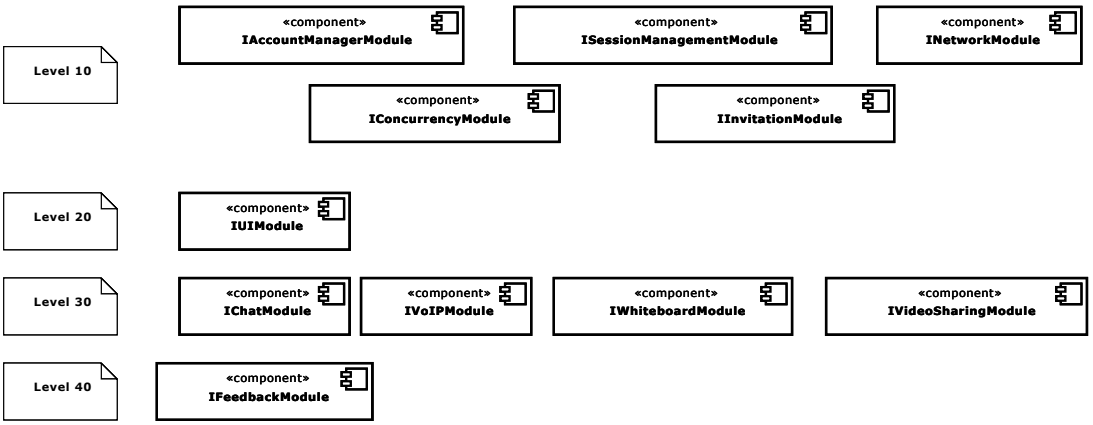

Abbildung 5.8 *Modularisierung von Saros*

Diese Struktur baut auf der Komponentensicht aus Abschnitt 3.2.1 auf. Die Levels stehen jeweils für die Abhängigkeit der Module zu einander. Module mit höherem Level dürfen die Module mit kleinerem Level kennen, umgekehrt nicht. So muss z.B. das *IChatModule* das *ISessionManagementModule* selber überwachen und einen Chat für eine Session starten. Auf der anderen Seite sollte *ISessionManagementModule* gar nicht wissen, dass *IChatModule* existiert.

5.4 Besonderheiten des Saros-Projektes

5.4.1 Dependency Injection von Eclipse

Nicht alle Teile von Saros stehen nach der Konstruktion zu Verfügung. Manche werden erst später von Eclipse erzeugt, wie z.B. Views. Um diese in ein *ISaros*-System zu integrieren, kann ein Injection-Service dafür definiert werden, Beispiel in Listing 5.12.

```
public interface IEclipseUIBridge extends ISarosService {

    public void insertView(String id, ViewPart view);

}
```

Listing 5.12 *AbstractSaros mit veränderbarer Modul-Liste*

5.4.2 Test

Es können Tests entwickelt werden, welche die Implementierung über die Modulschicht testen. So können die Tests noch vor der Umsetzung eines Moduls erstellt werden.

5.4.3 Portabilität

Um Saros auf andere IDE-Systeme einfacher portieren zu können, sollte bei der Modularisierung die Abhängigkeit von Eclipse in der Modulschicht dokumentiert werden. Dazu wäre es sinnvoll, alle Module, Services oder Objekte, welche Eclipse kennen sollen, mit z.B. „Eclipse“ im Namen zu markieren, wie schon bei `IEclipsePluginService` oder `IEclipseUIBridge`. Später wäre es einfacher zu identifizieren, welche Stellen angepasst oder erweitert werden müssen, um eine Portierung zu ermöglichen.

6 Bewertung

6.1 Vorgehen

In diesem Kapitel wird die in Kapitel 4 und 5 beschriebene Architektur bewertet. Dazu verwende ich das *SAAM* Verfahren, welches in Kapitel 2.3 beschrieben wurde. Als erstes habe ich versucht die beteiligten Personen und ihre Rollen im Projekt zu identifizieren:

- **Anwender** - benutzen das System, geben Feedback, melden Fehler
- **Entwickler**
    - für die Ausarbeitung einer Abschluss-/Studienarbeit
    - aus der Projektleitung
    - im Rahmen eines Werkvertrages
    - während eines Softwareprojektes (Veranstaltung)
- **Akzeptanztester** - regelmäßige Tests, umfangreiches Feedback

Wegen des technischen Themas und der Art des Projektes (kein direkter Kontakt zu Anwendern), habe ich vor allem in den Gesprächen mit anderen Entwicklern und der Projektleitung 5 Szenarios entwickelt welche nachfolgend beschrieben werden. Mit meinem Betreuer habe ich versucht eine Gewichtung für diese Szenarios zu finden, doch es war schwierig eine nachvollziehbare Bewertung zu erstellen. Eine Idee war mit leicht, mittel oder schwer zu bewerten (oder wie mein Betreuer es umschrieben hatte: Bachelor-, Master- oder Diplomarbeit). Eine andere Idee war die Größe des Quelltextes, die Komplexität oder die Manntage hinzu zu nehmen, nur ließ es sich kaum zuverlässig einschätzen. Aus diesem Grund habe ich entschieden die Schritte aufzuschreiben, welche notwendig sind, um ein Szenario zu erfüllen.

Als **direkt** betrachte ich Szenarios, welche keine Änderungen an der Architektur oder den Schnittstellendefinition (Module, Services) in gesamten System erfordern. Als **indirekt**, welche eine Änderung an den Schnittstellen und ModuleDoc-Beschreibungen erfordern.

6.2 Szenario 1 - Neuer Entwickler

Dieses Szenario beschreibt einen typischen Vorgang im Saros-Projekt. Ein neuer Entwickler, welcher eventuell parallel seine Abschluss-/Studienarbeit macht, kommt ins Projekt. Diese Person muss sich mit Saros vertraut machen, um erfolgreich daran arbeiten zu können.

Das ist ein **direktes** Szenario. Die Informationen aus ModuleDoc sollten dem neuen Entwickler einen Überblick über das System und seine Aufteilung geben. So kann diese Person die Stellen finden, an welchen sie arbeiten will/muss und sich dann in die jeweilige

Implementierung vertiefen. Von den anderen Teilen des Systems sollte es ausreichen nur die Modulschicht zu kennen.

6.3 Szenario 2 - Fehlerbehebung

In diesem Szenario wird eine Situation betrachtet, wo im System ein neuer Fehler entdeckt wurde und behoben werden muss.

Wenn der Fehler nicht einfach zu beseitigen ist und sich aus dem Zusammenspiel von mehreren Modulen ergibt, kann versucht werden die einzelnen Module durch alternative oder Mock-Implementierungen zu ersetzen, das System zu testen und den Fehler so einzugrenzen. Das kann sowohl ein **direktes**, wie auch ein **indirektes** Szenario sein, abhängig ob zu der Behebung des Fehlers die Schnittstellendefinition verändert werden muss. Bei einer Änderung sollten sich die betroffenen Entwickler und der Projektleiter zusammensetzen, die Änderungen abstimmen und in ModuleDoc reinschreiben.

6.4 Szenario 3 - neue Features

Bei diesem Szenario soll ein neues Teilsystem oder eine neue Komponente in das System integriert werden.

Das ist ein **indirektes** Szenario. Es müssen ein neues Modul und seine Schnittstellen definiert werden. Weiterhin muss es in ModuleDoc beschrieben werden. Danach erst kann es implementiert werden. Bei neuen Erkenntnissen bezüglich der Modulschicht während der Implementierung, sollten die Änderungen mit anderen Entwicklern besprochen und in ModuleDoc angepasst werden. Es können auch mehrere Implementierungen eines Moduls erfolgen, um z.B. eine eigene mit einer externen Lösung (in Form einer Bibliothek) zu vergleichen.

6.5 Szenario 4 - Tests

Dieses Szenario beschäftigt sich mit der Erstellung von automatischen Tests.

Das ist ein **direktes** Szenario. Mit der Aufteilung in eine abstrakte Modulschicht und einer von außen nicht sichtbaren Implementierung, können Tests entwickelt werden, welche die Implementierung nur über die Modulschicht testen. Diese Tests können schon vor der Implementierung erstellt werden.

6.6 Szenario 5 - Portierung

Das letzte Szenario betrachtet die Portierung von Saros auf andere Plattformen außerhalb von Eclipse.

Das ist ein **indirektes** Szenario. Dazu ist es notwendig in der Modulschicht die Eclipse- von der Saros-Funktionalität abzutrennen. Module und Services, welche Zugriff auf Eclipse

brauchen, sollten keine Kernfunktionalität von Saros anbieten. Weiterhin sollten diese Schichtelemente mit dem Schlüsselwort „Eclipse“ im Namen versehen werden. Bei einer Portierung wäre es so einfacher die Stellen im System, welche Eclipse voraussetzen, zu identifizieren und entsprechend darauf zu reagieren.

6.7 Gesamtbewertung

Die 5 Szenarios haben ergeben, dass es jeweils 2 direkte und 2 indirekte Szenarios gibt, sowie eines, welches beides sein kann. Die indirekten Szenarios zeigen, dass Saros inkrementell entwickelt wird. Um aus diesen Szenarios direkte zu machen, müsste ein konkreter Plan für die nächsten Jahre erstellt werden, welcher sich danach nicht mehr verändert. Doch das ist bei der aktuellen Art des Projektes nicht möglich. Weiterhin zeigen die indirekten Szenarios genauer wo die Probleme der Architektur aus Kapitel 4 und 5 liegen können, wenn diese umgesetzt wird.

7 Ausblick

7.1 Umsetzung

In meiner Arbeit habe ich ein Konzept einer neuen Architektur für Saros vorgestellt. Das tatsächliche und vollständige Umsetzen dieser Architektur ist zugegeben nicht einfach, weil dazu Saros stark umgeschrieben werden muss. Doch mit der inkrementellen Methode, ein Modul nach dem anderen abzugrenzen, dafür eine Schicht zu definieren und die Abhängigkeiten zu der Implementierung zu beseitigen, kann Saros Schritt für Schritt modularisiert werden, während der Weiterentwicklung.

7.2 Einschränkungen des Designs

Das Konzept sieht bisher vor, dass ein Modul von einem bestimmten Typ nur einmal im System vorhanden ist. Das gleiche gilt auch für die Services im Bezug zu einem Modul (verschiedene Module können schon einen Service vom selben Typ anbieten). Wenn mehrere Implementierungen des selben Moduls oder Services parallel laufen müssen, muss das Konzept angepasst werden. Auch wollte ich in den Entwurf des Modulsystems (aus Kapitel 4) den OSGi Lademechanismus integrieren. Damit könnte jedes Modul deklarieren, welche Module es benötigt, um lauffähig zu sein. Doch dieses Konzept würde in Konkurrenz zu dem PicoContainer treten und ist eigentlich nicht notwendig. Das wichtigste ist zuerst das System sauber in Module aufzuteilen. Danach wäre es einfach das System auf ein vorhandenes Modulsystem umzustellen.

7.3 Umstellung auf OSGi oder das Modularisierungskonzept von Java 8

In der Zukunft wird es in Java ein eigenes integriertes Modul-System geben (*Project Jigsaw*[24]), auch wenn noch nicht feststeht wie es konkret aussehen wird. Deshalb wäre es von Vorteil schon Heute Saros darauf auszurichten.

7.4 ModuleDoc

Eigentlich wollte ich ModuleDoc noch zusätzlich mit einem System versehen, mit welchen es möglich wäre UML-Diagramme direkt aus dem XML-Quelltext in Form von SVG-Dateien[13] zu erzeugen. Doch ich habe den Aufwand so einer Entwicklung überschätzt.
Mit diesem zusätzlichen System können Komponentensichten als UML-Diagramme aus der Beschreibung in ModuleDoc erstellt werden. Das würde zum besseren Verständnis

[13] Scalable Vector Graphics - XML-Format für Vektorgrafiken

des Systems beitragen. Dabei sollten die Diagramme nicht vollständig automatisch generiert werden. Es wäre möglich gewesen die Darstellung zu parametrisieren, um den Wiedererkennungswert zu sichern, wenn z.B. Elemente in das Diagramm hinzukommen oder entfernt werden. Zusätzlich wollte ich die Möglichkeit einbauen, eine Komponentensicht sinnvoll auf mehrere Diagramme zu verteilen, damit einerseits der Überblick über komplexere Strukturen einfacher wird und andererseits eine Art Buch entsteht, wo in einer sinnvollen Reihenfolge von Seite zu Seite gelesen werden kann.

A ModuleDoc

ModuleDoc ist ein auf XML basierendes Format für die Dokumentation der abstrakten Modulschicht und deren Implementierungen. Es folgt dem Schablonendokumentation-Entwurfmuster([3], S. 159) und soll als Formular die Arbeit mit der Dokumentation vereinfachen. ModuleDoc ist nicht an den Quelltext gebunden und damit können die Module schon vor der Implementierung definiert werden. Aufsetzend auf XML ist ModuleDoc leicht erweiterbar durch neue Elemente, diese müssen nur in der Beschreibung des Formats festgehalten werden. Es sind keine speziellen Werkzeuge erforderlich, außer einem Text- oder XML-Editor, um mit ModuleDoc zu arbeiten. Es eignet sich damit wunderbar, mit dem Quelltext zusammen abgelegt zu werden.

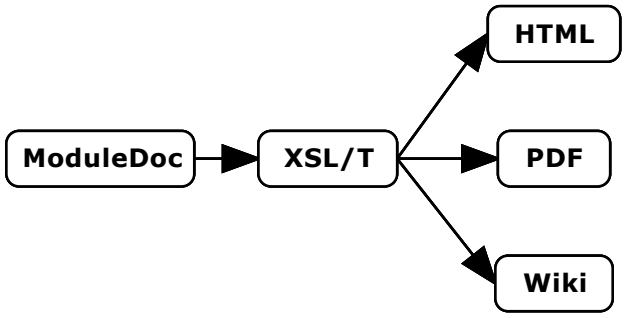

Abbildung A.1 *Konvertierung von ModuleDoc über XSL/T*

ModuleDoc kann über XSL/T in verschiedene Formate konvertiert werden, z.B. HTML, PDF oder Wiki, wie in Abbildung A.1 dargestellt. Durch die Angabe einer XML Processing Instruction können die ModuleDoc-Dateien direkt in einem Webbrowser betrachtet werden. Ein Beispiel wie ModuleDoc mit einem HTML-Stylesheet im Browser aussehen würde, ist am Ende dieses Anhangs zu sehen.

A.1 Struktur

Jedes Modul wird in einer eigenen XML-Datei beschrieben. Das Root-Element von ModuleDoc ist `saros-module`. Die Unterelemente beschreiben jeweils einen Teil des Moduls:

- **name** - Name des Moduls
- **description** - die Beschreibung eines Moduls als Text. Die Beschreibung kann jeweils versioniert werden, mit dem Attribut `version`. Es können mehrere `description` Tags hintereinander gesetzt werden, um die Entwicklung eines Moduls leichter verfolgbar zu machen. Die Sortierung erfolgt Anhand des `version` Attributes, welches entweder als Versionsnummern (1.0.0) oder als Zeichenkette interpretiert wird. Im

letzteren Fall passiert die Sortierung alphabetisch. Über das Attribut `author` kann eine ID des Entwicklers angegeben werden, welcher für diesen Teil der Beschreibung zuständig war. Der vollständige Name und weitere Angaben zu dem Entwickler werden in einer externen Datei gespeichert und erst bei der Betrachtung geladen.

- **services** - Gruppierung für die Beschreibungen der einzelner Services
- **objects** - Gruppierung für die Beschreibungen der Objekte eines Moduls
- **values** - Gruppierung für die Beschreibungen der Wertobjekte eines Moduls
- **resources** - Gruppierung für die Auflistung aller Ressourcen, welche zu einem Modul gehören. Darunter fallen alle Packages, Klassen, Konfigurationsdateien, Programmbibliotheken und andere binäre Dateien, wie Grafiken oder Klänge
- **implementations** - Gruppierung für die Beschreibung der einzelnen Implementierungen des Moduls

Im Listing A.1 ist ein Beispiel für die Hauptstruktur einer ModuleDoc-Datei.

```
<?xml version="1.0" encoding="UTF-8" ?>

<?xml-stylesheet type="text/xsl"
                 href="./style/saros-modules.xslt" ?>

<saros-module version="1.0"
              xml:lang="de">

  <name>Modul Name</name>

  <description version="1.0.0" author="slawa">
    Beschreibung des Moduls
  </description>

  <services>
  </services>

  <objects>
  </objects>

  <values>
  </values>

  <resources>
  </resources>

  <implementations>
  </implementations>

</saros-module>
```

Listing A.1 *Formular für die Beschreibung der Module*

Alle Elemente bis auf *name* und mindestens ein *description* sind optional. Listing A.2 zeigt ein Beispiel von einer minimalen Definition in ModuleDoc.

```
<?xml version="1.0" encoding="UTF-8" ?>

<?xml-stylesheet type="text/xsl"
                 href="./style/saros-modules.xslt" ?>

<saros-module version="1.0"
              xml:lang="de">

  <name>Modul Name</name>

  <description version="1.0.0">
    Beschreibung des Moduls
  </description>

</saros-module>
```

Listing A.2 *minimales ModuleDoc Formular*

A.2 Services, Objects und Values

Das `services` Element nimmt beliebig viele `service` Elemente auf, welche jeweils einen Service des Moduls beschreiben. Das `service` Element besteht aus einem `name` und mindestens einem `description` Element, die genauso definiert sind, wie im Root-Element.

```
<services>

  <service>
    <name>ServiceName</name>

    <description version="1.0.0">
      Beschreibung des Services
    </description>
  </service>

</services>
```

Listing A.3 *Beschreibung eines Services in ModuleDoc*

Die Elemente `objects` und `values` sind analog zu `services` definiert.

A.3 Ressource

Das `resources` Element nimmt beliebig viele `resource` Elemente auf, welche jeweils eine Ressource des Moduls beschreiben. Das können die Programmteile (Klassen, Packages) oder Dateien sein. Ein `resource` Elemente hat zwei Attribute: `type` und `res`. Mit dem ersten wird der Typ der Ressource beschrieben, mit dem zweiten die Ressource selber. Der Typ kann beliebig gewählt werden.

```
<resources>

  <resource type="package" res="de.fu_berlin.inf.dpp.*" />
  <resource type="class" res="de.fu_berlin.inf.dpp.Saros" />
  <resource type="library" res="lib/commons-lang-2.4.jar" />
  <resource type="config" res="config.xml" />
  <resource type="image" res="images/icon.png" />

</resources>
```

Listing A.4 *Beschreibung der Ressourcen in ModuleDoc*

A.4 Implementierungen

Das `implementations` Element nimmt beliebig viele `implementation` Elemente auf, welche jeweils eine Implementierung des Moduls beschreiben. Das `implementation` Element besteht aus einem `name`, mindestens einem `description` und einem `resources` Element. Diese sind genauso definiert, wie im Root-Element.

```
<implementations>

  <implementation>
    <name>ImplName</name>

    <description version="1.0.0">
      Beschreibung der Implementation
    </description>
  </implementation>

</implementations>
```

Listing A.5 *Beschreibung einer Implementierung in ModuleDoc*

A.5 Zusammenfassen der Beschreibung mehrerer Module

Mit dem `module` Element können mehrere Modulbeschreibungen zu einer zusammengefasst werden

```
<?xml version="1.0" encoding="UTF-8" ?>

<?xml-stylesheet type="text/xsl"
                 href="./style/saros-modules.xslt" ?>

<saros-module version="1.0"
              xml:lang="de">

  <module href="IModule1.xml" />
  <module href="IModule2.xml" />
```

```
  <module href="IModule3.xml" />

</saros-module>
```

Listing A.6 *Zusammenfassung bestimmter Module zu einem Dokument*

A.6 Textgestaltung

Zu besseren Hervorhebung von Text in einem `description` Element können die Tags `b` (hervorgehoben), `i` (kursiv), `u` (unterstrichen), `del` (durchgestrichen) und `ins` (eingefügt) verwendet werden. Für eine erweiterte Strukturierung können auch Listen und Tabellen eingeführt werden, doch man muss immer beachten, ModuleDoc ist ein Formular für Daten. Es sollte so einfach wie möglich gehalten werden und keine Gestaltungsinformationen enthalten.

A.7 Darstellung im Browser

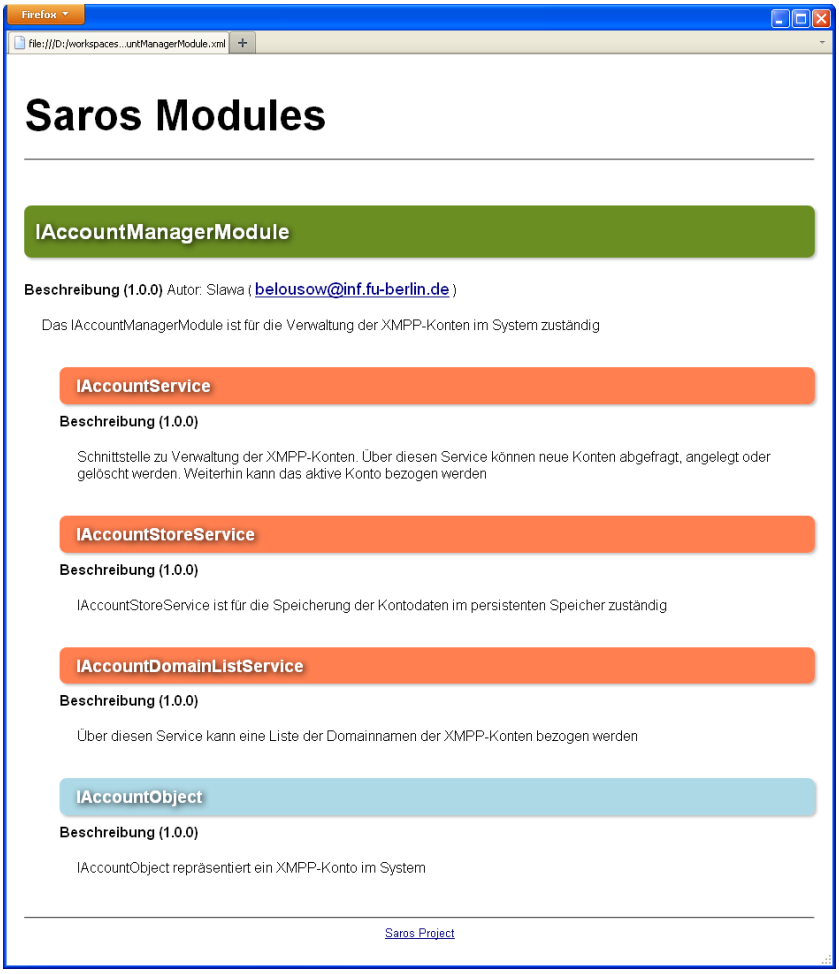

Abbildung A.1 *ModuleDoc im Firefox*

B Modul Beschreibungen

B.1 ICoreModule

```
<?xml version="1.0" encoding="UTF-8" ?>

<?xml-stylesheet type="text/xsl"
                 href="./style/saros-modules.xslt" ?>

<saros-module version="1.0"
              xml:lang="de"
              >

  <name>ICoreModule</name>

  <description version="1.0.0" author="slawa">
    ICoreModule ist die Repräsentation des Saros-Systems
    vor der Modularisierung
  </description>

  <services>

    <service>
      <name>ISarosClassicService</name>

      <description version="1.0.0">
        ISarosClassicService bietet einen
        Zugriff auf die Klasse Saros
      </description>
    </service>

    <service>
      <name>IEclipsePluginService</name>

      <description version="1.0.0">
        Der IEclipsePluginService ist eine Schnittstelle
        zu Übergabe des Eclipse Plugin-Mechanismus
        an das ICoreModule
      </description>
    </service>

  </services>

  <resources>
    <resource
      type="package"
      res="de.fu_berlin.inf.dpp.modules.core.*" />
  </resources>
```

```
</saros-module>
```

Listing B.1 *ICoreModule Beschreibung in ModuleDoc*

B.2 IAccountManagerModule

```
<?xml version="1.0" encoding="UTF-8" ?>

<?xml-stylesheet type="text/xsl"
                 href="./style/saros-modules.xslt" ?>

<saros-module version="1.0"
              xml:lang="de"
              >

  <name>IAccountManagerModule</name>

  <description version="1.0.0" author="slawa">
    Das IAccountManagerModule ist für die Verwaltung
    der XMPP-Konten im System zuständig
  </description>

  <services>

    <service>
      <name>IAccountService</name>

      <description version="1.0.0">
        Schnittstelle zu Verwaltung der XMPP-Konten.
        Über diesen Service können neue Konten abgefragt,
        angelegt oder gelöscht werden. Weiterhin
        kann das aktive Konto bezogen werden
      </description>
    </service>

    <service>
      <name>IAccountStoreService</name>

      <description version="1.0.0">
        IAccountStoreService ist für die Speicherung
        der Kontodaten im persistenten Speicher zuständig
      </description>
    </service>

    <service>
      <name>IAccountDomainListService</name>

      <description version="1.0.0">
        Über diesen Service kann eine Liste der Domainnamen
        der XMPP-Konten bezogen werden
      </description>
    </service>
```

```
  </services>

  <objects>

    <object>
      <name>IAccountObject</name>

      <description version="1.0.0">
        IAccountObject repräsentiert ein XMPP-Konto im System
      </description>
    </object>

  </objects>

  <resources>
    <resource
      type="package"
      res="de.fu_berlin.inf.dpp.modules.accountManagement.*" />
  </resources>

</saros-module>
```

Listing B.2 *IAccountManagerModule Beschreibung in ModuleDoc*

Quellenangaben

[1] Starke, G.: Effektive Software-Architekturen. Ein praktischer Leitfaden. 3. Auflage. München: Carl Hanser Verlag, 2008

[2] Starke, G./Hruschka, P.: Software-Architektur kompakt. - angemessen und zielorientiert. Heidelberg: Spektrum Akademischer Verlag, 2009

[3] Eilebrecht, K./Starke, G.: Patterns kompakt. Entwurfsmuster für effektive Software-Entwicklung. 3. Auflage. Heidelberg: Spektrum Akademischer Verlag, 2010

[4] Vogel, O./Arnold, I./Chughtai, A./Ihler, E./Kehrer, T./Mehlig, U./Zdun, U.: Software-Architektur. Grundlagen - Konzepte - Praxis. 2. Auflage. Heidelberg: Spektrum Akademischer Verlag, 2009

[5] Zöller-Greer, P.: Software Architekturen. Grundlagen und Anwendungen. 3. Auflage. Wächtersbach: Composia Verlag, 2010

[6] Wolf, H./Bleek, W.: Agile Softwareentwicklung. Werte, Konzepte und Methoden. Heidelberg: dpunkt.verlag, 2011

[7] Kübeck, S.: Software-Sanierung. Weiterentwicklung, Testen und Refactoring bestehender Software. mitp, 2009

[8] Feathers, M.: Effektives Arbeiten mit Legacy Code. Refactoring und Testen bestehender Software. mitp, 2011

[9] Martin, R.: Clean Code. Refactoring, Patterns, Testen und Techniken für sauberen Code. mitp, 2009

[10] Fowler, M.: Refactoring. oder: wie Sie das Design vorhandener Software verbessern. München: Addison-Wesley, 2005

[11] Schmidt-Casdorff, C./Vogel, T.: OSGi. Einstieg und Überblick. Frankfurt am Main: entwickler.press, 2009

[12] Daum, B.: Java-Entwicklung mit Eclipse 3.3. Anwendungen, Plugins und Rich Clients. Heidelberg: dpunkt.verlag, 2008

[13] Pilone, D.: UML 2.0 kurz & gut. 2. Auflage. Köln: O'Reilly, 2006

[14] Mangano, S.: XSLT Cookbook. 2. Auflage. O'Reilly, 2005

[15] Software Engineering Institute (2011): „Defining Software Architecture“. URL: http://www.sei.cmu.edu/architecture/start/definitions.cfm [Mai 2011]

[16] Software Engineering Institute (2011): „Architecture Analysis and Design Language“. URL: http://www.sei.cmu.edu/dependability/tools/aadl/index.cfm [Juni 2011]

[17] University of California (2005): „xADL“. URL: http://www.isr.uci.edu/projects/xarchuci/ [Mai 2011]

[18] Hruschka, P./Starke, G. (2011): „Das arc42 Template“. URL: http://www.arc42.de/template/template.html [April 2011]

[19] Zörner, S. (2011): „Historisch gewachsen? Java-Architekturen angemessen dokumentieren“. URL: https://bed-con.org/index.php/de/vortraege#vortrag11 [April 2011]

[20] Djemili, R. (2006): „Entwicklung einer Eclipse-Erweiterung zur Realisierung und Protokollierung verteilter Paarprogrammierung“. URL: http://www.inf.fu-berlin.de/inst/ag-se/theses/Djemili06-eclipse-erweiterung-PP.pdf [Mai 2011]

[21] Oezbek, C. (2009): „Technical Information“. URL: http://www.saros-project.org/Technicals [Juni 2011]

[22] Fowler, M. (2004): „Inversion of Control Containers and the Dependency Injection pattern“. URL: http://martinfowler.com/articles/injection.html [Mai 2011]

[23] OSGi Alliance (2011): „How to get Started with OSGi“. URL: http://www.osgi.org/About/HowOSGi [Mai 2011]

[24] Oracle Corporation (2011): „Project Jigsaw“. URL: http://openjdk.java.net/projects/jigsaw/ [Mai 2011]

[25] Oracle Corporation (2011): „JDK 7 Features“. URL: http://openjdk.java.net/projects/jdk7/features/ [Mai 2011]

[26] Reinhold, M. (2008): „The Modular Java Platform“. URL: http://mreinhold.org/blog/modular-java-platform [Mai 2011]

[27] Reinhold, M. (2008): „Project Jigsaw“. URL: http://mreinhold.org/blog/jigsaw [Mai 2011]

[28] Buckley, A. (2006): „JSR 277: Java Module System“. URL: http://jcp.org/en/jsr/detail?id=277 [Mai 2011]

[29] Watson, T. (2007): „JSR 291: Dynamic Component Support for Java SE“. URL: http://jcp.org/en/jsr/detail?id=291 [Mai 2011]

[30] Buckley, A. (2007): „JSR 294: Improved Modularity Support in the JavaTM Programming Language“. URL: http://www.jcp.org/en/jsr/detail?id=294 [Mai 2011]

[31] Kazman, R./Bass, L./Abowd, G./Webb, M. (1994): „SAAM: A Method for Analyzing the Properties of Software Architectures“. URL: http://www.sei.cmu.edu/library/abstracts/whitepapers/icse16.cfm [Juni 2011]

[32] Kazman, R./Abowd, G./Bass, L./Clements, P. (1996): „Scenario-Based Analysis of Software Architecture“. URL: http://www.sei.cmu.edu/library/abstracts/whitepapers/scenariobasedanalysisnov1996.cfm [Juni 2011]

[33] Lilley, C./Schepers, D. (2011): „Scalable Vector Graphics (SVG) 1.1“. URL: http://www.w3.org/TR/SVG/ [Mai 2011]

[34] XMPP Standards Foundation, . (2011): „About - The XMPP Standards Foundation“. URL: http://xmpp.org/about-xmpp/ [Juni 2011]

[35] XMPP Standards Foundation, . (2011): „XMPP Extensions“. URL: http://xmpp.org/xmpp-protocols/xmpp-extensions/ [Juni 2011]