Freie Universität Berlin

Bachelor am Institut für Informatik der Freien Universität Berlin,

Arbeitsgruppe Software Engineering

Refaktorisierung des Eclipse-Plugins Saros für die Portierung auf andere IDEs

Arndt Lasarzik
Matrikelnummer: 4366560
arndt.lasarzik@fu-berlin.de

Betreuer: Franz Zieris
Eingereicht bei: Prof. Dr. Lutz Prechelt
Zweitgutachterin: Prof. Dr. Elfriede Fehr

Berlin, 29. April 2015

Arndt Lasarzik

Zusammenfassung

Saros ist ein Eclipse-Plugin für verteilte Echtzeitprogrammierung für Gruppen bis zu fünf Personen. Letztes Jahr wurde auf Nutzerwunsch begonnen, Saros auch für andere IDEs zu portieren. Für die erste Portierung auf IntelliJ wurden diverse Klassen von Saros dupliziert, um einen Prototypen zu entwickeln. Es wurde beschlossen einen IDE-unabhängigen Kern zu entwickeln, der viele Klassen bereitstellt, die Saros in jeder IDE brauchen wird.

In meiner Bachelorarbeit erkläre ich, wie man die Abhängigkeiten zu einer IDE auflösen kann, um die Klassen in den Saros-Kern zu verschieben. Dies ist notwendig, da die Duplikate technische Schulden sind, die den Quellcode verschlechtern. Die Abhängigkeiten löse ich mit Refaktorisierungen und Designmuster auf und tilge damit der einige aufgenommenen Schulden. Das Resultat ist ein Kern, von dem aus die Saros-Plugins die notwendigsten Funktionen beziehen können, wodurch keine doppelten Implementierungen von Nöten sind.

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.

29. April 2015

Arndt Lasarzik

Inhaltsverzeichnis

1 Einleitung

1.1 Saros

Saros ist ein Eclipse-Plugin zur verteilten Paar/Gruppenprogrammierung. Es bietet einer Gruppe von Entwicklern die Möglichkeit, in nahezu Echtzeit, gemeinsam ortsunabhängig an demselben Quellcode zu arbeiten. Änderungen an dem Quellcode werden allen Beteiligten sofort farblich markiert und allen Programmierer zu Verfügung gestellt.
Eclipse ist in Deutschland mit einem Marktanteil von über 70% [Neu11] [1] die populärste IDE (integrierte Entwicklungsumgebung). Nichtsdestotrotz wird Saros auch für IntelliJ gewünscht. IntelliJ wird von Entwicklern geschätzt, und auch Google hat vor einiger Zeit die Standardprogrammierumgebung für Android von Eclipse auf IntelliJ geändert. IntelliJ ist bei vielen Programmierern beliebt, da es mehr Funktionen bereitstellt als Eclipse [Han13]. Daraufhin hat das Saros-Team beschlossen, die Entwicklung für IntelliJ zu starten. Einer weiteren Implementierung für NetBeans (siehe Kapitel 2.3.2) wird erprobt.

1.2 Funktionsweise von Saros

Saros ist ein Werkzeug für die verteilte Programmierung. Dazu werden Quellcodeänderungen von einem Teilnehmer zu den anderen Teilnehmern mittels XMPP[2] versendet. Nur während des Einladungsprozesses werden Daten auch mittels P2P-Verbindung versendet, da hier größere Datenmengen anfallen können. Um Saros nutzen zu können, müssen alle Teilnehmer sich mit einem XMPP-Account einloggen.
Saros führt für die Quellcodeänderungen auch Dateioperationen durch, die über Eclipse bereitgestellte Schnittstellen vorgenommen werden. Projekte, die hinzugefügt werden, werden von Eclipse nicht auf Veränderungen überwacht. Die IDE geht davon aus, dass Änderungen über die IDE selbst vorgenommen werden und daher ist eine Kontrolle, ob sich die Dateien verändert haben, nicht notwendig. Es ist daher wichtig, die Schnittstellen von Eclipse für Dateimanipulationen zu nutzen, um keine Inkonsistenzen zu erzeugen. Für eine ausführlichere Beschreibung von der Funktionsweise von Saros, ist die Masterarbeit von Patrick Schlott [Sch13] zu empfehlen.

1.3 Portierung von Saros auf IntelliJ

Saros wurde als Eclipse-Projekt gestartet und weist viele Kopplungen mit dieser IDE auf. Im letzten Jahr (2014) startete die Entwicklung von Saros-IntelliJ (kurz Saros/I[3]). Bei der Entwicklung von Saros/I sollte ein Prototyp

[1] Stand 2011
[2] XMPP ist ein Protokoll für Nachrichtenübertragung.
[3] Saros für Eclipse bekam intern die Bezeichnung Saros/E

entwickelt werden, der die notwendigsten Funktionen von Saros implementiert, um die Machbarkeit zu demonstrieren. Der Prototyp soll unabhängig von Saros/E funktionsfähig sein. Dazu muss er eine Accountverwaltung haben, mit der man neue XMPP-Accounts hinzufügen kann und muss sich auch mit einem gespeichertem Account verbinden können. Außerdem muss man eine Sitzung starten, um Quellcodeänderungen von anderen Sitzungsteilnehmern zu erhalten und eigene senden zu können.
Es war nicht möglich, den vorhandenen Quellcode von Eclipse für IntelliJ zu nutzen, da die Schnittstellen von Eclipse in IntelliJ nicht zur Verfügung stehen. Langfristig soll der Saros-Kern die notwendigsten Funktionen zum Betreiben von Saros zur Verfügung stellen. Es ist nicht trivial die Abhängigkeiten von Eclipse zu lösen, daher konnte dies kurzfristig nur teilweise umgesetzt werden. Um dennoch den Prototypen erstellen zu können, wurden Teile vom Saros Quellcode kopiert und für IntelliJ angepasst. Dieses Vorgehen ermöglichte den Prototypen, hatte aber den Nachteil, dass dadurch technische Schulden angehäuft wurden.

1.4 Motivation

Bei der Betrachtung von einer Vielzahl von Programmen, stellt man fest, dass die meisten immer größer und komplexer wurden. Nahezu jede Software bekommt mit der Zeit neue Funktionen, ein überarbeitetes Design und viele andere Veränderungen. Dabei werden die meisten Programme schon lange nicht mehr von einem Entwickler alleine entwickelt und gepflegt, sondern von mehreren. Das Spektrum reicht von einer handvoll bis zu mehreren tausend Entwickler und es ist klar, dass die Software festen Standards folgen muss, damit dies gelingt. Der Quellcode muss verständlich und wartbar sein. Ohne Qualitätsstandards erodiert [Fow99, S. 47] die Struktur vor sich hin, bis die Fehler und Defekte das Projekt zum Erliegen bringen und das Produkt von Grund auf neu entwickelt werden muss. Software-Erosion (software decay, Bit rot) beschreibt den Prozess des Verfalls der Strukturen einer Software.
Es ist utopisch anzunehmen, dass die ganze Zeit ein klarer und sauberer Stand gehalten werden kann. Es gibt verschiedenste Gründe, warum schlechter Code geschrieben wird. Oft ist es der recht banale Grund, dass eine Deadline sich nähert und die unsaubere Lösung die einfachste und schnellste Lösung ist. Ich bin fest davon überzeugt, dass dies nicht immer sinnvoll ist, da das Design verschlechtert wurde und der nächste Entwickler mit dem Problem zurechtkommen muss. Dieser hat die Möglichkeit, das Design zu korrigieren, oder damit weiterzuarbeiten. Das Problem ist, dass mit der Zeit „Workarounds“ bilden sehr viel mehr Zeit kosten, als eine korrekte Umsetzung zuvor. Diese „Workarounds“ addieren sich zeitlich, der Code wird unübersichtlicher und fehleranfälliger. Es ist eine Spirale, die unterbrochen werden muss.
Saros befindet sich in einer ähnlichen Position. Es gibt viele wechselnde

Entwickler mit unterschiedlichen Erfahrungen bei der Programmierung und hat einen Softwarestand der durch die Entwicklung von Saros/I gelitten hat. Ich möchte zeigen, wie man die Softwarequalität schrittweise wieder steigern kann, ohne Defekte einzubauen oder sich andere Nachteile einzuholen. Ich bin davon überzeugt, dass diese Technik zum Verbessern und Korrigieren von der Software sehr wichtig ist, nicht nur für Saros sondern auch im Generellen.

2 Problemstellung

Meine Analysen beginnen, als der Saros-Kern erstellt wurde und die Arbeiten an Saros/I begonnen haben. Der Kern bot zu Beginn der Saros/I-Entwicklung noch nicht sehr viele Funktionalitäten an, weshalb es zu einer großen Menge an Duplikaten kam. Durch diese kam es zu einigen sehr wichtigen Analysen. Es wurde ermittelt, welche Klassen überhaupt relevant für den Saros-Kern sind.

2.1 Herausforderungen bei der Entwicklung

Saros/I soll nach der initialen Erstellung die notwendigsten Funktionen für den Betrieb von Saros anbieten. Es soll sichergestellt werden, dass Saros/E mit Saros/I kompatibel ist. Die wichtigsten Komponenten dafür sind:

- die Dateioperationen
  Zur Programmierung gehört es Code zu schreiben. Die Änderung muss Saros bei den anderen Teilnehmern durchführen können.

- die Netzwerkkomponente
  Saros muss Daten versenden können, um Änderungen den anderen Teilnehmern mitzuteilen.

- die Accountverwaltung
  Saros benutzt zum Versenden von Informationen XMPP. Dazu sind Accounts notwendig, die jede Instanz verwalten können muss.

- die Sitzungsverwaltung
  In einer Saros-Sitzung besteht aus einer Vielzahl von Unterkomponenten(Concurrency Management, ActivitySequencer, Activity Handling). Dabei werden die Arbeitsschritte von der Benutzeroberfläche bis zur Netzwerkschicht koordiniert.

- der Einladungsprozess
  Der Einladungsprozess fügt einen neuen Nutzer zu einer Sitzung hinzu und sorgt dafür, dass die geteilten Projekte zu diesem gesendet und entpackt werden. Er stellt einen konsistenten Stand her.

Diese Komponenten müssen in den IDE-unabhängigen Kern verschoben werden. Durch die gemeinsame Nutzung des Kerns von allen Saros Plugins (Saros/E, Saros/I,..) ist es einfacher, die Kompatibilität zu gewährleisten und so für Nutzerakzeptanz zu sorgen (siehe Abbildung[4] 1).

[4] Die Abbildung ist von http://www.saros-project.org/saros-intellij-plan

Abbildung 1: Der Kern von Saros bietet eine gemeinsame Basis für die Saros-Implementierungen.

2.1.1 Steigerung der Codequalität

Durch die Verwendung von einer Klasse für mehrere Saros-Plugins werden Duplikate von bestehenden Klassen vermieden. Das Verändern von Quellcode um die Codequalität zu steigern, ohne das äußere Verhalten zu ändern heißt *Refaktorisierung*. Es dient dazu die technischen Schulden abzubauen. Das Standardwerk für Refaktorisierungen heißt „Refactoring“ und wurde von Martin Fowler [Fow99] geschrieben. Es beschreibt diverse Methodiken um Gerüche zu entfernen und den Quellcode so zu verbessern.
Die Refaktorisierungen sind die passenden Werkzeuge, mit denen man die Verdrahtungen des Saros-Quellcodes mit Eclipse lösen kann, ohne neue Schulden aufnehmen zu müssen. Dabei gilt es stets zu erkennen, wie die bestehenden Probleme beseitigt werden können.

2.2 Zielstellung dieser Arbeit

Ich möchte, dass nach Beendigen meiner Arbeit die wichtigsten Teile des Saros-Quellcodes im Saros-Kern liegen. Dabei meine ich die elementarsten Funktionen, die jedes Saros-Plugin benötigt, sollen im Kern verfügbar sein. Darunter verstehe ich

- die Dateioperationen
- die Netzwerkkomponente
- die Accountverwaltung
- die Sitzungsverwaltung
- der Einladungsprozess

Es ist kein Ziel Saros/I fertigzustellen, und alle Duplikationen aus dem Quellcode zu entfernen. Dies übersteigt den Umfang der Arbeit deutlich. Ich möchte zeigen, dass es möglich ist, die IDE-Abhängigkeiten mit Refaktorisierungen auszulösen. Dabei ist es auch durchaus sinnvoll andere technische Schulden damit zu tilgen, da die verbesserte Struktur die Arbeit erleichtert.

Außerdem möchte ich den Quellcode in einem besseren Zustand hinterlassen, als ich ihn vorgefunden habe und folgenden Entwicklern die Weiterentwicklung vereinfachen.

2.3 Ähnliche Abschlussarbeiten

Es gibt ähnliche Abschlussarbeiten, die während meiner Ausarbeitungen erstellt wurden oder noch werden. Im Folgenden möchte ich diese kurz Vorstellen und erklären, warum sie Auswirkungen auf meine Arbeit haben oder meine auf deren Arbeit hat.

2.3.1 Qualitätssicherung durch statische Codeanalysen

Die Arbeit von Arsenij Solovjev beschäftigt sich mit Erosion von Software, die häufig durch ein Fehlen von Verständnis über das vorhandene Softwaredesign entsteht [Sol14]. Er beschreibt, wie die Softwarearchitektur ein Bestandteil der Analysen zur Qualitätssicherung werden kann. Die Arbeit setzt präventiv an, möchte verhindern, dass technische Schulden entstehen. Die Entwickler können damit frühzeitig gewarnt werden und solche Fehler vermeiden.
Durch diese Arbeit ist es mir leichter gefallen Probleme im Quellcode zu finden und zu beheben, ehe sie im Master-Branch Schäden verursacht haben.

2.3.2 Saros für Netbeans

Neben der Entwicklung von Saros für IntelliJ wird derzeit von Sabine Bender an Saros für Netbeans gearbeit. Sie profitiert von den Refaktorisierungen, die bereits für Saros/I getätigt wurden. Für die Entwicklung an Saros/I wurden bereits viele Analysen für die Entwicklung des Saros-Kerns vorgenommen, die bei Netbeans nicht mehr durchgeführt werden müssen.

2.3.3 Vereinheitlichung der grafischen Benutzeroberfläche von Saros

Eine Vereinheitlichung der grafischen Benutzeroberfläche für Saros/E und Saros/I entwickeln Matthias Bohnstedt [Boh15], Christian Cikryt [Cik15] und Bastian Sieker[Sie15]. Ziel ist es eine HTML- und JavaScript-Oberfläche bereitzustellen um auch hier den Wartungsaufwand durch aufwendige Doppelimplementationen zu vermeiden. Sie versuchen durch eine Neuentwicklung die alte Benutzeroberfläche zu ersetzen, dabei sind sie teilweise auf Funktionen angewiesen, die ich in den Kern verschoben habe.

2.3.4 Saros-Server

Nils Bussas hat in seiner Bachelorarbeit [Bus14] den Prototypen für einen Saros-Server präsentiert. Diese Arbeit kreiert den Saros-Server im Eclipse-

Plugin als zusätzliche Funktionalität, und nicht als eigenständigen Dienst. Ute Neise arbeitet derzeit in ihrer Studienarbeit [Nei15] an der Weiterentwicklung vom Saros-Server-Prototypen. Langfristig ist es gewünscht, den Saros-Server IDE-unabhängig laufen zu lassen. Diese Funktion würde einen ausgebauten Kern voraussetzen, an dem ich mitarbeite, der aber noch nicht ausreichend ist.

3 Grundlagen

Ich habe bereits im Kapitel über Herausforderungen bei der Entwicklung (S. 4) darüber geschrieben, was technische Schulden sind und warum diese getilgt werden sollten. Ich habe auch beschrieben, dass ich dies mit Refaktorisierungen erreichen möchte. In diesem Kapitel möchte ich das Thema vertiefen und erklären, wie man Gerüche in der Software wahrnimmt und an kurzen Beispielen zeigen, wie man die Gerüche durch Refaktorisierungen entfernt.

3.1 Refaktorisierung

3.1.1 Was ist eine Refaktorisierung?

In dem Standardwerk „Refactoring“ von Martin Fowler beschreibt er die Refaktorisierung als „den Prozess“, ein Softwaresystem so zu verändern, dass das externe Verhalten nicht geändert wird, der Code aber eine bessere interne Struktur erhält. [Fow99, S. 9].
Die Refaktorisierung ist kein Allheilmittel. Der Entwickler muss abwägen, wann eine Refaktorisierung sinnvoll ist und wann nicht. Es ist auch nicht immer objektiv beurteilbar, welche Änderung die Lesbarkeit tatsächlich steigert. Sollte die interne Struktur soweit erodiert sein, dass eine Refaktorisierung länger dauern würde als eine Neuentwicklung, dann sollte dies auch in Betracht gezogen werden. Außerdem setzt eine Refaktorisierung weitgehend funktionierenden Code voraus. Wenn man Probleme hat, ein Programm zum Laufen zu bekommen, dann sind Refaktorisierungen nicht sinnvoll.[5]
Das Hauptaugenmerk liegt auf der Verbesserung der internen Struktur. Explizit sind damit keine Leistungsverbesserungen (Performance) gemeint, da diese häufig schwer verständlich sind. Nach einer Refaktorisierung soll der Quellcode verständlicher und leichter anpassbar sein als vorher. Damit lässt sich auch die Gefahr von Defekten reduzieren, weil sie sichtbarer sind als vorher. Fowler unterscheidet dabei strikt zwischen Implementierung neuer Funktionen und einer Refaktorisierung. Er benutzt die Metapher der zwei Hüte:

Bei der Refaktorisierung werden kleine Änderungen vorgenommen, die das äußere Verhalten nicht beeinflussen. Jede kleine Änderung wird getestet, damit kein Defekt auftreten kann. Mit einer Reihe von kleinen Änderungen ist die Entwicklung häufig zügiger, da die Fehlersuche sich auf einen kleineren Bereich beschränkt.

Bei dem Hinzufügen von Funktionen wird neuer Code hinzugefügt. Außerdem ist es notwendig neue Tests zu schreiben und vorhandene anzu-

[5] Die meisten Funktionen von Saros laufen fehlerfrei.

passen. Die vorhandenen Tests können fehlschlagen, weil sich das Verhalten ändern kann.

Desweiteren stellt er fest:

> "You can only wear one hat at a time."[Fow14]

Die Metapher zeigt, dass ein Entwickler zwei verschiedene Rollen verkörpern muss, wenn er refaktorisiert und eine neue Funktion implementiert. Es ist möglich Rollen öfter zu wechseln, doch zu einem Zeitpunkt kann man jeweils nur eine Rolle einnehmen. Diese Erkenntnis ist wichtig, wenn man den Aufgabenbereich von der Rolle „Refaktorisierung“ definieren möchte.

> "Der Vorgang des Refactorings umfasst das Entfernen von Duplikaten, das Vereinfachen komplexer Logik und das Verdeutlichen von unklaren Quelltext."[Ker04, S. 33]

Bei solchen Handlungen besteht immer die Gefahr, selbst Defekte zu erzeugen. Sowohl Fowler als auch Kerievsky halten es für sehr wichtig, den Code zu testen, um die Gefahr für neue Defekte zu minimieren.

3.1.2 „Code Smell“

Refaktorisierungen bieten sich an, um Abhängigkeiten von Klassen aufzulösen, wie ich es bei meiner Arbeit getan habe. Gerüche zeigen dem Entwickler, wo der Quellcode ist, der verbessert werden sollte. Eine Liste mit Smells bieten Fowler [Fow99] und Kerievsky [Ker04] in ihren Büchern. Ich möchte deshalb nur drei Gerüche im folgenden beschreiben.

Komplexe Bedingungen sind potenzielle Fehlerquellen und damit ein Geruch. Es werden häufig Fehler bei der Implementierung gemacht, sodass oft viel Zeit beim Debugging benötigt wird, da es nicht sofort ersichtlich ist, dass ein Fehler in der Bedingung liegt. Ein Beispiel dafür bietet der Patch 1978[6]. Der eigentliche Fehler wurde durch eine übereilte Fehlerkorrektur erst eingebaut[7].

Listing 1: Diese Methode wird durch ein Defekt immer vorzeitig beendet

```
if (resource.getType() != IResource.FILE
 || resource.getType() != IResource.FOLDER)
     return;
```

Da eine Ressource[8] nicht gleichzeitig eine Datei als auch ein Ordner sein kann, wird die Methode immer vorzeitig beendet und nicht durchlaufen. Der

[6] http://saros-build.imp.fu-berlin.de/gerrit/#/c/1978/
[7] http://saros-build.imp.fu-berlin.de/gerrit/#/c/1977/
[8] Auf die Ressourcen gehe ich in Kapitel 5.1.1 auf Seite 21 ein.

Autor des Patches hat bereits selbst festgestellt, das er die De Morgansche Gesetze falsch angewendet hat. Die De Morgansche Gesetze sind Regeln zur booleschen Aussagenlogik und beschreiben die korrekte Umformung von boolschen Aussagen.

Listing 2: Die Logik wurde korrigiert.

```
if (!(resource.getType() == IResource.FILE
   || resource.getType() == IResource.FOLDER))
            return;
```

Lange Logikausdrücke sind schwer zu lesen und deshalb fehleranfällig. Dieses Beispiel zeigt, das ein Defekt leicht wegen eines Geruches übersehen werden kann. Wenn die Ressource keine Datei und kein Ordner ist, dann soll die Methode abgebrochen werden. Teilweise sind Refaktorisierungen auch ein Geschmacksurteil. Der Autor war zufrieden mit der Behebung des Fehlers, obwohl der Ausdruck noch einfacher zu formulieren gewesen wäre, indem man ihn der normalen Sprache weiter annähert.

Lange Methoden erledigen meistens mehrere Aufgaben und sind dadurch oft schwer zu lesen. In den meisten Fällen kann das Problem behoben werden, indem man Teile des Quellcodes in eine eigene Methode auslagert. In der Klasse CreateArchiveTask von Saros war die run()- Methode zu lang[9]. Die Methode errechnete die Größe von allen zu übertragenen Dateien und hat anschließend jede Datei gelesen und in ein Archiv geschrieben. Danach wurde der Monitor aktualisiert, der dem Nutzer den Fortschritt anzeigt. Es sind viele verschachtelte Aufrufe mit Schleifen, Try/Catch-Blöcken und Wenn/Dann Bedingungen. Durch das Herauslösen von der Berechnung der Dateigröße konnte die Methode ein wenig übersichtlicher gemacht werden, da Anstatt der Berechnung nur noch der Methodenaufruf „getTotalFileSize(files)“ in der run()-Methode vorhanden ist. Diese Änderung ist sinnvoll, da die Schritte auf demselben Abstraktionsniveau liegen. Ursprünglich wollte ich noch mehr Methoden herauslösen[10], aber die Entwickler sahen den Vorteil nicht, sodass es dabei blieb.

Kommentare sind auch ein Indiz für einen Geruch. Wenn der Quellcode nicht durch geschickt gewählte Variablen- und Methodennamen sprechend[11] genug ist und der Entwickler sich veranlasst fühlt einen Kommentar zu schreiben, um den Sachverhalt erklären zu müssen, ist dies ein Zeichen, das der Quellcode optimiert werden sollte. Entweder ist er zu komplex, fehleranfällig, unvollständig oder unverständlich. Alles sind Gründe, den Code zu ändern.

[9] http://saros-build.imp.fu-berlin.de/gerrit/#/c/1803/
[10] http://saros-build.imp.fu-berlin.de/gerrit/#/c/1796/
[11] Damit meine ich, dass die Methode genau das tut, wonach sie benannt ist.

3.1.3 Technische Schulden

Technische Schulden sind eine Metapher, die zeigen soll, dass hastige Quellcodeänderungen sich negativ auswirken können und so zu mehr Arbeit bei der Beseitigung des Codes oder der Entwicklung neuer Funktionalitäten führt. Die Metapher lehnt sich an die Finanzwelt an, bei dem Schulden, die genommen wurden, wieder zurückgezahlt werden müssen.[Fow03]
Bei der Entwicklung des Saros/I Prototypen durch die Duplizierung von diversen Klassen vom Saros-Quellcode sank die Softwarequalität und stiegen die technischen Schulden. Duplikationen im Code bedeuten immer, dass der Entwickler wissen muss, dass mehrere Codefragmente angepasst werden müssen, damit der Quellcode sich nicht auseinanderentwickelt. Sollten diese Duplikationen nicht im selben Maße angepasst werden, besteht die Gefahr, dass sich der Code auseinanderentwickelt, höheren Aufwand bei der Wartung verursacht und die Kompatibilität verliert.
Die Entwicklung von Saros/I erfolgte zunächst in einem Feature-Branch. In diesen Entwicklungszweig wurden die Kopien der vorhandenen Klassen angelegt und angepasst. Parallel dazu entwickelte sich der Master-Branch (der führende Entwicklungszweig) weiter, ohne das die Änderungen regelmäßig in den neuen Branch überführt wurden. Die Folge war ein hoher Aufwand den Feature-Branch mit den Master-Branch zusammenzuführen. Dies zeigt, wie wichtig es ist, technische Schulden zu tilgen.
Neben Duplikationen von Quellcode gibt es weitere technische Schulden, die im Quellcode von Saros gefunden werden können. Technische Schulden werden oft als „Bad Smell“ wahrgenommen [Sur14]. Der (übelriechende) Geruch ist ebenfalls eine Metapher für einen Bereich im Quellcode, dessen Qualität verbessert werden sollte.
Technische Schulden entstehen nicht nur bei der Umsetzung von größeren Projekten, wie bei Saros/I. Häufig ist es ein schleichender Prozess, der nach und nach einsetzt. Die Gründe sind genauso vielfältig wie deren Ausprägungen. Es kann zum Beispiel an der Unerfahrenheit der Entwickler oder am mangelnden Qualitätsmanagements liegen. Bei Saros trifft das auch zu, wobei das Qualitätsmanagement mittlerweile gut ausgebaut ist und auch weiter optimiert wird (siehe Abschnitt 3.3).

3.2 Entwurfsmuster

Entwurfsmuster sind Vorlagen für wiederkehrende Entwurfsprobleme in der Architektur. Diese Muster lassen sich weiter in Erzeugungsmuster, Strukturmuster und Verhaltensmuster einteilen. Im folgenden möchte ich einige dieser Muster erklären, die für den Saros-Kern verwendet wurden.

3.2.1 Abstrakte Fabrik

Fabriken stellen ein gängiges Erzeugungsmuster dar. Die abstrakte Fabrik ist eine Schnittstelle, die Methoden für die konkreten Fabriken zu Verfügung stellt (siehe Abbildung 2).

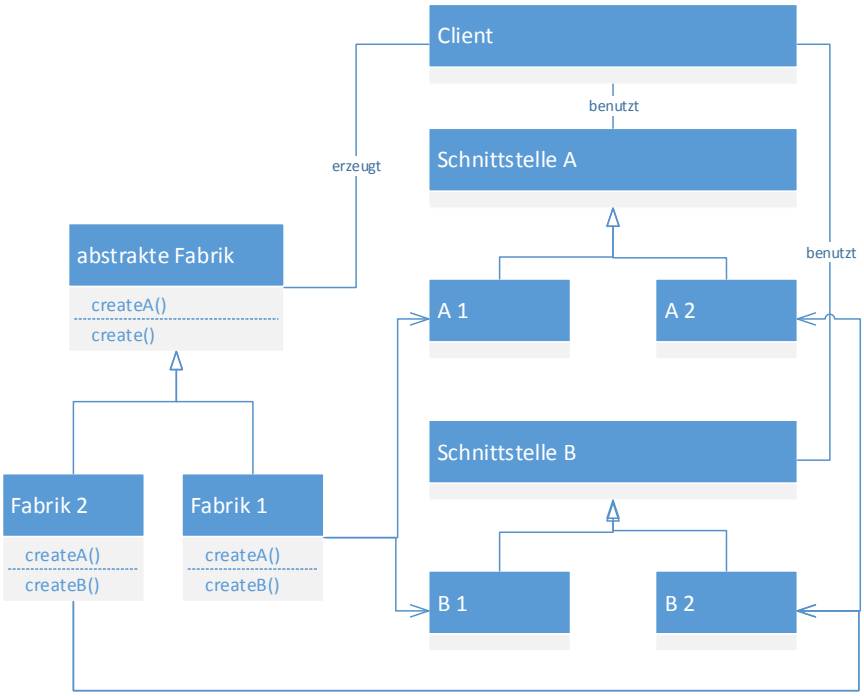

Abbildung 2: Der Client benutzt Schnittstelle A und B. Die Implementation erfolgt durch die Fabrik

Der Client arbeitet nur auf den Schnittstellen. Durch die Fabrik gelangt der Client an das entsprechende Objekt. Der Client ist immer noch verantwortlich, welche Fabrik er nutzt. Das Entwurfsmuster der abstrakten Fabrik ist sinnvoll, wenn mehrere Gruppen von Objekten benutzt werden, die leicht austauschbar sein müssen. [EG94, S. 99ff]
Bei Saros ist dies beim Dateisystem der Fall. Eclipse besitzt eigene Klassen, die den Zugriff auf Dateien, wie Pfade, Ordner oder Arbeitsbereiche steuern. Analog gilt dies bei IntelliJ. Ein Beispiel bietet die abstrakte Fabrik „IPathFactory“. Sie konnte das Problem mit den verschiedenen IPath-Objekten lösen (Siehe Kapitel 5.1.1).

3.2.2 Adapter

Ein Adapter gehört in die Kategorie der Strukturmuster. Er ermöglicht die Zusammenarbeit von inkompatiblen Klassen, die sonst nicht zusammenarbeiten könnten. Indem eine Hülle um das inkompatible Objekt gelegt wird, kann der Adapter die Signatur von der anderen Schnittstelle annehmen und somit die Kompatibilität herstellen [EG94, S.157f].

3.2.3 Proxy

Ein Proxy ist ein Stellvertreter für ein anderes Objekt. Es soll damit der Zugriff auf dieses Objekt kontrolliert werden und dient häufig als Platzhalter.

3.2.4 Dependency Injection

In der objektorientierten Programmierung ist ein Objekt verantwortlich, seine Abhängigkeiten selbst zu verwalten. Es muss daher Wissen über seine Umgebung mitbringen, um Objekte zu kreieren, die es selbst benötigt. In der Regel braucht das Objekt dieses Wissen nicht für die Erfüllung seiner Aufgaben, sondern nur für seine Abhängigkeiten.

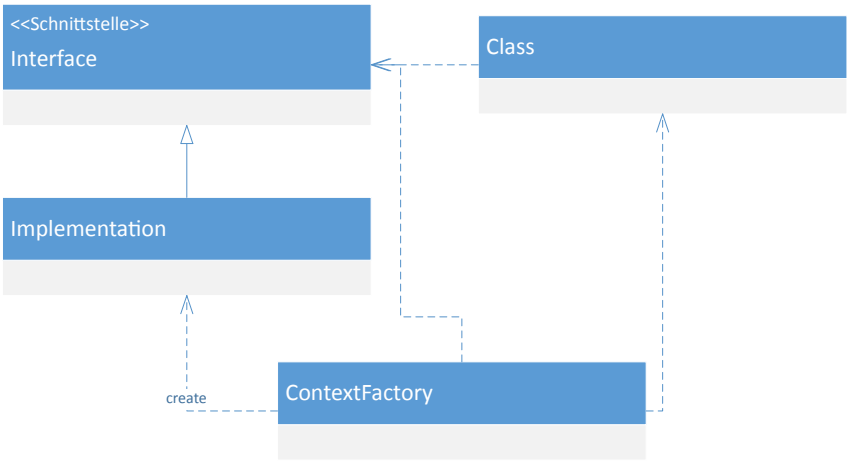

Abbildung 3: Die Klasse muss sich nicht mehr um die Erzeugung kümmern.

Mit der Dependency Injection (Abhängigkeit injizieren) kann man diesen Kontrollfluss umkehren (Inversion of Control) [Fow04]. Es wird eine neue Klasse erstellt, die sich um die Generierung der Abhängigkeiten kümmert und das Objekt bekommt diese überreicht (siehe Abbildung 3).
Der Unterschied zur abstrakten Fabrik besteht darin, dass das Objekt bei der Dependency Injection keine Verantwortung zum Erzeugen mehr hat. Bei der abstrakten Fabrik, muss das Objekt die entsprechende Implementation der Fabrik erzeugen. Dies fällt hier weg.

Bei Saros ermöglicht diese Technik nicht nur simpleren Quellcode[12] sondern führt auch dazu, dass die Klassen im Kern ihre Umgebung nicht mehr genau kennen müssen. Die Implementation wird zur Laufzeit überreicht. Dabei spielt es keine Rolle, ob die Klasse in Eclipse oder IntelliJ ausgeführt wird.

3.3 Qualitätssicherung

Wenn Quellcodeänderungen vorgenommen werden, besteht immer die Gefahr, dass Defekte entstehen. „Never change a running system“[13] oder „if it ain't broke, don't fix it“ sind „Weisheiten“ die jeder Entwickler gehört hat. Sie geben Ausreden notwendige Veränderungen hinauszuschieben und so das Problem nur noch größer werden zu lassen. Sie beruhen auf Angst [BF00], das System zu beschädigen und so Ausfälle zu produzieren. Fragile Systeme sind wartungsunfreundlich. Sie erhöhen den Aufwand Änderungen einzupflegen. Die Entwickler sind unsicher, benötigen mehr Zeit und dies Erhöht abermals die Kosten für die Software.

Doch woher kommt die Unsicherheit? Unsicherheiten entstehen, aus Angst, das Seiteneffekte auftreten, die einen Defekt an einer anderen Stelle auftreten lassen. Um dem entgegenzuwirken muss die Software getestet werden. Je mehr von der Software getestet wird, desto sicherer ist der Programmierer, keinen neuen Defekt einzubauen.
Refaktorisierungen sind Softwareänderungen, die genauso Defekte verursachen können, wie alle anderen Quellcodeänderungen auch. Deswegen möchte ich im folgenden Beschreiben, wie man Refaktorisierungen gegen Defekte absichern kann und dem Entwickler Vertrauen gibt, diese Änderungen auch durchzuführen.
Diese qualitätsfördernden Maßnahmen sind kein Garant für ein fehlerfreies Funktionieren der Software, aber sie bieten eine solide Grundlage und fangen die häufigsten und trivialsten Fehler ab, ehe sie Schaden anrichten können.

3.3.1 Versionsverwaltung Git

Ein Versionierungssystem koordiniert den gemeinsamen Zugriff auf Dateien und ermöglicht die gleichzeitige Entwicklung in mehreren Zweigen (Branches). Neben der Protokollierung der Änderungen werden die einzelnen Entwicklungsschritte archiviert. Dies ermöglicht es, jederzeit zu einem vorigen Stand zurückzugehen.

[12] Die Abhängigkeit kann durch einer Annotation (Anmerkungen für den Compiler) injiziert werden.

[13] Die genaue Herkunft lässt sich nicht bestimmen. Wahrscheinlich ist es eine Abwandlung von „Never change a Winning team.“ von Alf Ramsey.

3.3.2 Automatisierte Tests

Ein wichtiges Werkzeug für die Qualitätssicherung sind automatisierte Tests. Diese Tests können einzelne Komponenten oder die Software im Ganzen testen. Saros besitzt dafür JUnit-Tests, die jeder Entwickler beim Programmieren ausführen lassen kann. Sie sind einfach auszuführen und sehr schnell, weshalb sie gut bei Refaktorisierungen einsetzbar sind. Außerdem besitzt Saros zahlreiche Tests, die die Benutzeroberfläche testen, indem sie das Programm starten und typisches Benutzerverhalten simulieren. Diese Tests dauern wesentlich länger, erkennen aber ein Versagen der Software, die bei reinen Komponententests nicht auffallen würden.

3.3.3 Kontinuierliche Integration

Die kontinuierliche Integration beschreibt den Prozess, der mehrere Komponenten miteinander integriert, das heißt, sie werden zu einer Anwendung hinzugefügt. Dabei wird der Quellcode übersetzt (kompiliert) und die automatischen Tests (ohne die GUI-Tests) ausgeführt. Durch die häufige Ausführung (nach jedem Commit in Git) kann die Gefahr von fehlerhaften Softwarezuständen im Hauptentwicklungszweig verringert werden. Es werden Übersetzungsfehler und Fehler in den Tests sichtbar, die ohne kontinuierlicher Integration hätten übersehen werden können. Der Entwickler wird auch automatisiert über den Abschluss des Prozesses informiert.
Der Saros-Entwickler Stefan Rossbach hat sich diesem Thema in seiner Bachelorarbeit gewidmet und bietet ausführlichere Erklärungen zu diesem Thema. [Ros11]

3.3.4 Statische Codeanalyse

Statische Codeanalyse ist ein Verfahren, bei der der Quellcode einer Reihe von formalen Prüfungen unterzogen wird. Bei diesen Verfahren können bestimmte Fehlerarten und Gerüche entdeckt werden. Außerdem können dem Programmierer Hinweise gegeben werden, um diese Fehler/Anmerkungen zu beseitigen. Bei diesen Verfahren wird der Quellcode nicht ausgeführt.

3.3.5 Codereview Plattform Gerrit

Automatisiert durchgeführte Analysen und Tests decken viele Fehler, Defekte oder schlechte Implementierungen nicht auf. Es ist daher ratsam, andere Entwickler den Code beurteilen zu lassen, da diese mit ihrer Expertise viele nützliche Hinweise geben können. Die statischen Analysen können nur unterstützen, ersetzen aber eine Review nicht. Um den Reviewprozess einfach und produktiv zu halten, ist ein Werkzeug, wie Gerrit, sehr hilfreich.

Bei Saros wird jeder eingereichte Patch in Gerrit angezeigt und die Entwickler haben die Möglichkeit in diesen Patch Kommentare zu hinterlassen und ihn zu bewerten. In der Regel erfolgt dies, nachdem der Patch gebaut wurde und die Tests ausgeführt wurden. Eine Codereview ermöglicht den Quellcode in einer Art zu bewerten, die mit einer statischen Codeanalyse nur bedingt möglich ist, da der Reviewer seine Expertise und Wissen zu der Software nutzt. Dies ist sehr schwer automatisiert abzubilden. Durch die Beurteilung von anderen Entwicklern ist die Gefahr, Defekte übersehen zu haben, deutlich niedriger. Dank der Arbeit von Arsenij Solovjev [Sol14] können seit kurzem die Ergebnisse der statischen Codeanalyse in Gerrit eingesehen werden. Dadurch sind diese für den Entwickler sichtbarer.

4 Analyse

4.1 Analyse der Klassen und deren Reihenfolge

Im Vorfeld habe ich mir Gedanken gemacht, welche Klassen von den Refaktorisierungen betroffen sein werden. Es war von Anfang an klar, dass nicht alle Funktionen von Saros/E in den anderen IDEs zwingend notwendig sein werden. Es stellte sich bald heraus, dass diese Gedanken eigentlich überflüssig waren, da die parallel gestartete Entwicklung von Saros/I von Raimondas Kvietkauskas und Holger Schmeisky die notwendigen Klassen identifiziert haben. Sie haben die Duplikate angelegt und eine weitergehende Betrachtung dessen war überflüssig, bis diese Klassen durch Kernklassen ersetzt wurden.
Als nächstes waren die Überlegungen nach der Reihenfolge, denen ich einen großen Stellenwert zuschreiben wollte. Die enorme Anzahl an Duplikaten lies mich zweifeln, alle Klassen in den Kern verschieben zu können, daher wollte ich zumindest die wichtigsten bearbeiten, sodass die übrigen nur noch Fleißarbeiten wären. Die Parameter, die ich erkannte waren aber schlichtweg nur die Anzahl der Nutzungen einer Klasse und die Bitten von Entwicklern, bestimmte Klassen zu refaktorisieren, damit diese vom Kern aus genutzt werden können. Letzterer Parameter empfand ich als wichtiger, da die Relevanz offensichtlich höher war.

4.1.1 Methoden oder Klassen

Am Anfang stellte sich die Frage, ob es sinnvoll ist, abstrakte Klassen im Kern bereitzustellen. Lediglich die IDE-abhängigen Teile des Codes sollten in der IDE-Implementiert werden während der Rest sich im Kern befindet. Dieser Vorstoß wurde verworfen, als klar wurde, dass sehr viele Klassen dieselben Abhängigkeiten haben und man so nur den Kern mit ähnlich aufgebauten abstrakten Klassen füllt. Häufig wären mehrere Refaktorisierungen nötig gewesen, um den IDE-abhängigen Teil so zu separieren. Der Code hätte funktioniert, wäre aber nicht so „sauber“ gewesen. Die tatsächlichen Umsetzung hat den Vorteil, dass mit den Adaptern die Abhängigkeiten aufgelöst wurden und damit allen Klassen in Saros zu Verfügung stehen.

4.2 Abhängigkeitsgraphen

Ich wusste bei meinen Ausarbeitungen, was mein Ziel war, kannte aber den Weg nicht. Deswegen habe ich Abhängigkeitsgraphen erstellt. Ich habe damit die Reihenfolge der zu refaktorisierenden Klassen bestimmmt, indem ich erst Klassen mit wenig Abhängigkeiten refaktorisiert habe und somit, Schritt für Schritt, an die komplexen Klassen mit vielen Abhängigkeiten herankam. Saros ist komplex und groß. Deswegen ist der Graph vom Einladungsprozess

am Anfang meiner Arbeit sehr undurchsichtig[14] (siehe Abbildung 13 auf Seite 37). Dieser Graph ist für dieses Dokument stark verkleinert und an die Seitengröße A4 angepasst. Jedes einzelne Rechteck entspricht dabei einer Klasse aus dem Einladungsprozess. Die Grafik soll zeigen, wie kompliziert und aufwendig diese Aufgabe war.
Durch die schrittweise Entfernung einzelner Abhängigkeiten war es möglich, den Graphen übersichtlicher werden zu lassen. Häufig war es durch einen Adapter möglich, mehrere Klassen in den Kern zu verschieben. Einen späteren Stand zeigt Abbildung 14 (Seite 38) ein sichtlich deutlicheres Bild. Hier war es für mich möglich gleich zwei zyklische Abhängigkeiten im Einladungsprozess zu erkennen.
Ich musste während der Arbeit diese Graphen mehrfach aktualisieren, da sich die Abhängigkeiten verändert haben. Der Graph wurde durch das stetige Lösen von Abhängigkeiten einfacher und dadurch lies sich besser mit ihm arbeiten.

5 Implementierung

Bisher habe ich beschrieben, warum Saros einen IDE-unabhängigen Kern braucht, wenn es mehr als eine IDE unterstützen will. Desweiteren habe ich erklärt, dass ich diese mit einer Reihe von Refaktorisierungen umsetzen möchte. Im folgenden möchte ich erläutern, wie ich meine Ziele umgesetzt habe.

Die größten Probleme waren die Abhängigkeiten von der Eclipse-API. Sie ließen sich nicht auflösen und es musste ein Weg gefunden werden, die Funktionalität trotzdem anzubieten. Es gab dabei Muster, die sich immer wieder finden ließen und wahrscheinlich auch in der Zukunft angewendet werden können. Im folgenden möchte ich diese Muster anhand von Beispielen erklären, da ich davon überzeugt bin, dass sie dadurch deutlicher und greifbarer werden.

5.1 Erstellung von Adaptern

Die häufigste Methode, um die Abhängigkeiten in den „Kernklassen“ zu entfernen, war die Erstellung von Adaptern. Diese können meistens ohne Probleme die abhängigen Klassen ersetzen. Es müssen im Quellcode Konvertierungen in beide Richtungen (Eclipse $\longleftrightarrow$ Saros) erfolgen. Dies wird in den Fabriken durchgeführt, die jede IDE-Implementation bekommen hat. Als Beispiel sollen die Dateiressourcen dienen, bei der ich die Adapter und

[14] Auf den vollständigen Graphen für Saros verzichte ich anzugeben, da dieser auf mehrere Seiten verteilt werden müsste.

die Fabrik erklären werde. Das Verfahren konnte auch bei der Versionskontrolle (VCS) und der Fortschrittsanzeige (progress monitor) angewandt werden.

5.1.1 Adapter für das Dateisystem

Dateisystemoperationen

Um eine verteilte Paar/Gruppen-Programmierung zu ermöglichen ist es unerlässlich, dass vorgenommene Quellcodeänderungen von einem Sitzungsteilnehmer zu den Anderen gesendet werden. Saros muss dazu die Änderungen vom Quellcode feststellen, versenden und bei den anderen Teilnehmern die Datei ändern. Es muss ein konsistenter Stand bei jedem Teilnehmer erhalten bleiben, da sonst Saros in seiner Funktion versagen würde. Es ist daher wichtig, das Saros über Dateioperationen benachrichtigt wird und auch Dateioperationen ausführen kann. IDEs wie Eclipse überwachen aber nicht die Dateien, die bei ihnen verwaltet werden. Damit Eclipse weiß, dass eine Datei geändert wurde, muss die Dateiänderung über die API von Eclipse vollzogen werden.

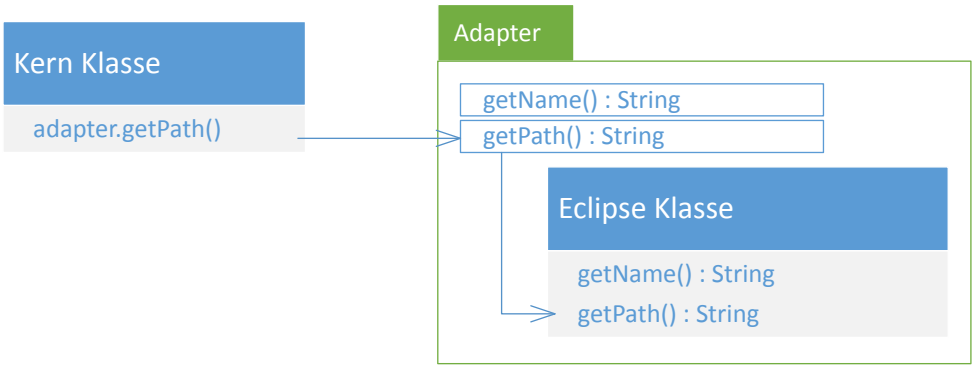

Abbildung 4: Der Aufruf von der Kern-Klasse wird delegiert.

Um Dateioperationen über die Eclipse-API im Kern von Saros durchführen zu können, müssen für Klassen im Kern Stellvertreterobjekte erzeugt werden.
Für die Umsetzung bietet sich der Adapter (S. 13) an. Die Eclipse-Abhängigkeiten sind offensichtlich nicht aufzulösen und die Dateioperationen müssen im Kern verfügbar sein. Daher kann die Eclipse-API nicht direkt verwendet werden.
Der Adapter bildet eine Hülle um das eigentliche Objekt, welches bei Saros/E weiterhin das Objekt von Eclipse ist. Es implementiert eine Schnittstelle, welche die Methoden bereitstellt, die Saros von diesem Objekt erwartet. Sämtliche Funktionsaufrufe werden zu dem umhüllten Objekt weitergeleitet (delegiert) (siehe Abbildung 4).

Listing 3: Auszug aus EclipsePathImpl

```
...
public class EclipsePathImpl implements IPath {

    private final org.eclipse.core.runtime.IPath
       delegate;

    EclipsePathImpl(org.eclipse.core.runtime.IPath
       delegate) {
        if (delegate == null)
            throw new NullPointerException("delegate
               is null");

        this.delegate = delegate;
    }
...
    @Override
    public boolean isAbsolute() {
        return delegate.isAbsolute();
    }
...
```

In Listing 3 kann man die Implementierung von einem Adapter sehen. Ein „Path“ ist eine Kollektion von Zeichenketten, die Informationen zu einer Ressource (Datei/Ordner) enthält. IPath ist eine Schnittstelle, die Methoden deklariert, die Path implementieren muss. EclipsePathImpl implementiert die Saros Schnittstelle für IPath[15]. Dem Konstruktor von EclipsePathImpl wird das zu umschließende Eclipse Objekt übergeben.
In den meisten Fällen benutzen die Saros-Klassen die Schnittstellen, anstatt deren Implementierung direkt. Damit folgt Saros dem Abhängigkeits-Inversionsprinzip [Kü09, S.73]. Die Abhängigkeit wird dadurch von der Implementierung losgelöst und ermöglicht dessen Austausch. Dies machen wir uns zunutze, indem jede IDE seinen eigenen Adapter bereitstellt, in der das IDE-Objekt umhüllt wird.
Für den Saros-Kern wurden neue Schnittstellen geschaffen, die die Dateioperationen bereitstellen. Ich habe daran gearbeitet, die Umsetzung wurde aber von Stefan Rossbach durchgeführt[16] Die Implementation von den Schnittstellen wird über Adapter vorgenommen, da jede IDE ihre eigene API für das Manipulieren der Dateien besitzt und daher mit einem Adapter versehen werden muss. Die Klassen im Kern benutzen diese Dateioperationsschnittstellen und die Adapter rufen diese vom IDE-Objekt auf.
Als zweites Entwurfsmuster wird die Abstrakte Fabrik (S. 12) verwendet. Die

[15] Eclipse hat eine gleichnamige Schnittstelle.
[16] http://saros-build.imp.fu-berlin.de/gerrit/#/c/1415/

Erzeugung von den Kern-Ressource-Objekten soll in Fabriken stattfinden, um unabhängig von deren Erzeugung zu sein. Für jede IDE wurden Adapter für die Ressourcen angelegt, damit die Kern-Klassen, die die Schnittstelle benutzen, nichts über deren Implementation wissen müssen.
Jede IDE hat seinen eigenen Adapter für Ressourcen und die Klassen, die die Schnittstelle benutzen, müssen nichts über deren Implementation wissen.

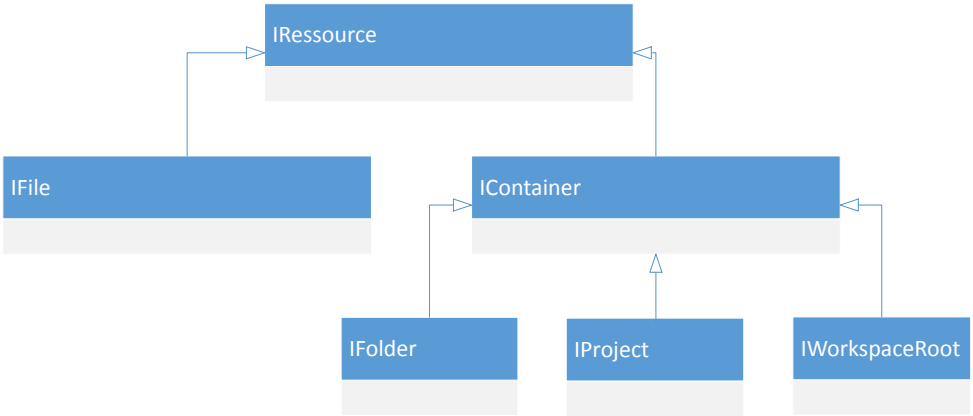

Abbildung 5: Hierarchie der Dateisystemschnittellen

Die Ressourcenhierarchie kann man in Abbildung 5 sehen. Es gibt allgemeine Ressourcen, die nicht näher spezifiziert sind. Dies können aber Ausprägungen wie File (Datei) oder Folder (Ordner) annehmen. Saros und Eclipse haben einen identischen Aufbau dieser Hierarchie. Dies hat zusammen mit den Adaptern den Vorteil, dass die Ressourcen von Eclipse mit den Ressourcen von Saros ausgetauscht werden können, ohne aufwendige Anpassungen in den einzelnen Klassen vornehmen zu müssen.

5.1.2 Einführung eines Null-Objektes

Unter Umständen ist die Verbindung zu einem XMPP-Server gestört, weil eine Firewall die Verbindung verhindert. Für diesen Fall bietet Saros/E die Möglichkeit einen Proxy-Server für die SOCKS-Protokoll anzugeben. Saros leitet dann den Datenverkehr über diesen Server, der die Daten dann weiterleitet. Dies ist ein erweiterte Funktionalität, die Saros/I zu Beginn nicht braucht.
Diese Funktion ist aber in den Quellcode von der Netzwerkkomponente integriert, die in den Kern muss. Anstatt den Quellcode so zu modifizieren, dass das Objekt für den Proxy „null“ sein darf, kann man auch ein Null-Objekt einführen, der das geeignete Null-Verhalten bereitstellt. [Ker04, S. 327f]

Listing 4: Null-Objekt Implementierung

```
/**
 * A {@link IProxyResolver} that does nothing.
 */
public class NullProxyResolver implements
    IProxyResolver {

     @Override
     public ProxyInfo resolve(String host) {
         return null;
     }
}
```

Mit der Einführung des Null-Objektes gelang es mir die Abhängigkeit vom ConnectionHandler, der von mir bereits in den Kern verschoben wurde[17], zu erfüllen und somit die Klasse ConnectServerAction im IntelliJ Teil von Saros von Duplikationen zu befreien[18]. Der ConnectionHandler implementierte bereits einige Funkionen, die in die Action-Klasse kopiert wurden, um die Funktionalität zu haben

5.1.3 Probleme mit den Adaptern

Die Objekte können in der Regel einfach mit einem Adapter umhüllt oder aus der Hülle befreit werden, da sich die IDE-spezifischen Objekte in den Adaptern befinden und nur aus ihnen herausgeholt werden müssen. Es gibt allerdings Grenzen dafür.

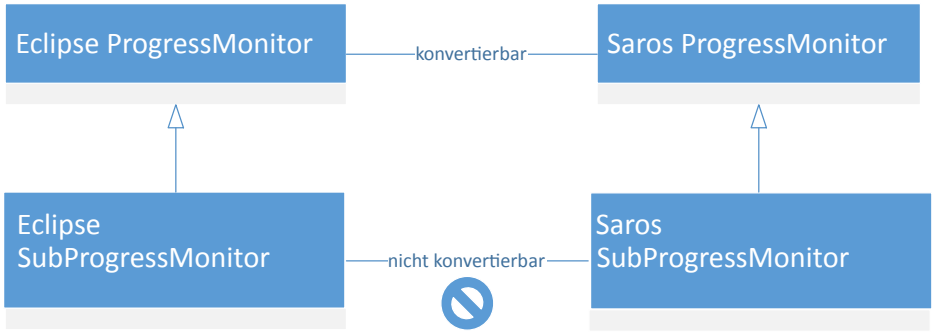

Abbildung 6: Ein SubProgressMonitor lässt sich nicht in den anderen überführen.

Eclipse bietet in seiner API einen ProgressMonitor und einen SubProgressmonitor an. Der ProgressMonitor zeigt den Fortschritt einer Aktion an. Der SubProgressMonitor erhält eine Anzahl von Millisekunden vom ProgressMonitor. Verkettete Aktionen lassen sich so besser überwachen. In der Saros-API sieht es ganz ähnlich aus. Der Unterschied liegt primär daran,

[17] http://saros-build.imp.fu-berlin.de/gerrit/#/c/2117/
[18] http://saros-build.imp.fu-berlin.de/gerrit/#/c/2128/4

dass der ProgressMonitor ein Adapter ist. Der SubProgressMonitor umhüllt wiederrum den ProgressMonitor. Bei dem Versuch der Konvertierung von einem Saros-SubProgressMonitor in einem Eclipse-ProgressMonitor wird lediglich die Hülle des SubProgressMonitor entfernt und der Versuch scheitert (siehe Abbildung 6). Der SubProgressMonitor hat die Signatur vom ProgressMonitor und daher gibt es keine Warnung für den Programmierer, dass der Versuch scheitern kann. Stefan Rossbach hat die Rückkonvertierungsmethode als deprecated (missbilligt) markiert. Dieses Beispiel zeigt, dass die Anwendung von Adaptern auch Probleme mit sich ziehen kann.

5.2 Verschieben von Abhängigkeiten

Die einfachste Art, die Abhängigkeiten zu entfernen ist, sie aus der Klasse zu verschieben. Durch die Nutzung einer Schnittstelle, anstatt des Objektes selbst, ist die Abhängigkeit indirekter und kann durch eine beliebige Implementation ersetzt werden. Es ist nicht immer davon auszugehen, dass dieses Prinzip direkt angegangen werden kann. Deswegen möchte ich nach dem ersten Beispiel eine aufwendigere Refaktorisierung beschreiben, die in mehreren Schritten stattgefunden hat (Abschnitt 5.2.2).
In Abschnitt 5.2.3 beschreibe ich einen Sonderfall. Diese Refaktorisierung nimmt zunächst technische Schulden auf, damit andere Ziele früher erreicht werden können.

Einladungsprozess

Der Einladungsprozess von Saros ermöglicht das Starten einer Sitzung mit anderen Teilnehmern (session invitation) und das Übertragen von einem Projekt (project invitation). Es ist zwingend notwendig, dass alle Teilnehmer in einer Sitzung sind, damit Quellcodeänderungen übertragen werden können. Ohne den Einladungsprozess könnte keine Sitzung gestartet werden. Neben dem Starten einer Sitzung muss auch der Quellcode zu Beginn der Sitzung übertragen werden, falls er nicht bereits vorhanden ist. Da der Einladungsprozess relevant für das Funktionieren von Saros ist, soll er in den Kern verschoben werden.
Der Einladungsprozess ist komplex und ich werde nur auf bestimmte Abschnitte eingehen. Für ein vollständiges Bild möchte ich auf die Arbeit von Patrick Schlott [Sch13, S. 75ff.] verweisen.
Beim Einladungsprozess tauscht der Client mit dem Host Informationen über die zu sendenen Dateien aus. Diese Verhandlungen werden vom OutgoingProjectNegotiation (Host) und vom IncomingProjectNegotiation (Client) durchgeführt. Bei der OutgoingProjectNegotiation (OPN) wird eine Hilfsklasse EditorManager verwendet, um vor dem Archivieren der zu sendenen Dateien, die ungesicherten Veränderungen im Editorfenster zu speichern. Dies ist notwendig, um den aktuellen Stand des Hosts senden zu

können.

5.2.1 Abhängigkeits-Inversionsprinzip

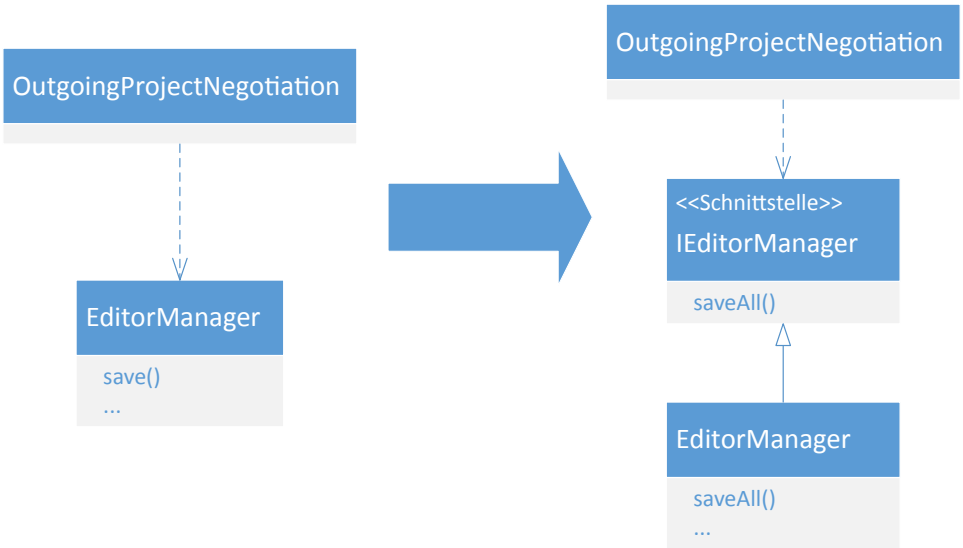

Abbildung 7: OutgoingProjectNegotiation hat nun Abhängigkeiten zur Schnittstelle IEditorManager.

Der EditorManager ist eine Klasse von Saros, besitzt aber IDE-Abhängigkeiten, da er Dateioperationen ausführt. Dies stellte kein Problem dar, bis die Klasse in den Kern verschoben werden sollte. Die Lösung wird durch die Umsetzung vom Abhängigkeits-Inversionsprinzip erreicht. Stefan Rossbach hat ein Interface für den EditorManager erzeugt, welches der EditorManager implementiert[19]. Nun konnte der OPN dieses verwenden und ist nicht mehr darauf angewiesen, die Implementierung vom Interface zu kennen (siehe Abbildung 7)[20]. Mittels Dependency Injection (S. 13) konnte die Kopplung aufgehoben werden. Ich habe die Änderungen auf IntelliJ übertragen[21].

5.2.2 VCS-Abhängigkeiten

Saros bietet Unterstützung für das Versionsverwaltungssystem Subversion (SVN)[22]. Dazu wird das Eclipse-Plugin Subclipse[23] benötigt. In den Einstellungen von Saros kann man die Unterstützung aktivieren. Dadurch kann Saros dem SVN Änderungen mitteilen.

[19] Der Methodenname ist sich geändert, da dieser nun passender ist.
[20] http://saros-build.imp.fu-berlin.de/gerrit/#/c/1936/3
[21] http://saros-build.imp.fu-berlin.de/gerrit/#/c/2088/
[22] https://subversion.apache.org/
[23] http://subclipse.tigris.org/

Die Untersützung von SVN benötigt eine API, die nur in Eclipse zur Verfügung steht. Deswegen muss sie ebenfalls abstrahiert werden, damit die IncomingProjectNegotiation frei von IDE Abhängigkeiten ist. In der Negotiation werden die Dateien mit dem SVN abgeglichen, falls aktiviert.
Die Struktur der Negotiation war ein wenig erodiert, deswegen musste im ersten Schritt der Code, der sich um das SVN kümmert, von der restlichen Logik getrennt werden. Mit der Refaktorisierungstechnik „extract method“ [Fow99, S.89f] wurde dies von Stefan Rossbach nach meinem ersten Vorschlag umgesetzt[24]. Dadurch ist der Quellcode besser gegliedert worden.
Die IncomingProjectNegotiation ermittelt eine Liste von fehlenden Dateien, damit diese beim Host angefordert werden kann. Bei der Analyse der Berechnung wurde festgestellt, dass die Dateien mit dem SVN abgeglichen wurden, jedoch ohne eine Konsequenz aus dem Ergebnis zu ziehen. Wenn das Projekt unter der Versionskontrolle steht, dann müsste die Liste immer leer sein. Es gibt keinen Anlass die Daten mit dem SVN bei der Berechnung der Dateiunterschiede abzugleichen, deswegen wurde der Aufruf in der Negotiation mit „null“ ersetzt. Andere Klassen könnten diese Informationen benutzen, deswegen wurde die Methodensignatur nicht geändert[25].
Saros unterstützt momentan SVN als Versionierungssystem. Prinzipiell ist es möglich, die Unterstützung für andere Versionierungssysteme, wie zum Beispiel Git, zu erweitern. Deswegen gibt es ein allgemeines Interface für Versionskontrollsysteme: VCSProvider. Die Negotiation benutzte die Kindklasse von VCSProvider, den VCSAdapter, für seine Ausführungen auf dem SVN. Mit dem Anpassen und Kopieren der benutzten Methoden aus dem VCSAdapter in den VCSProvider gelang es, die Abhängigkeiten für die Negotiation zu entfernen (siehe Abbildung 8). Die Methoden sind Überladen, das heißt, dass es mehrere Methoden mit denselben Namen existieren, die aber unterschiedliche Eingangsparameter haben. Wenn die Methode benutzt wird, die die Saros-Ressourcen bekommt, dann konvertiert sie diese mit einer Fabrik in Eclipse Ressourcen und ruft dann die gleichnamige Methode auf, die die Eclipse-Ressourcen annimmt und in welcher sich die eigentliche Logik befindet.

Listing 5: Konvertierung von Saros Ressourcen

```
@Override
  public boolean isManaged(de.fu_berlin.inf.dpp.
      filesystem.IResource resource) {
       return isManaged(ResourceAdapterFactory.
          convertBack(resource));
  }
```

[24] http://saros-build.imp.fu-berlin.de/gerrit/#/c/2178/
[25] http://saros-build.imp.fu-berlin.de/gerrit/#/c/2197/

```
    @Override
    public String getRepositoryString(
        de.fu_berlin.inf.dpp.filesystem.IResource
            resource) {
        return getRepositoryString(
            ResourceAdapterFactory.convertBack(resource
            ));
    }
    ...
```

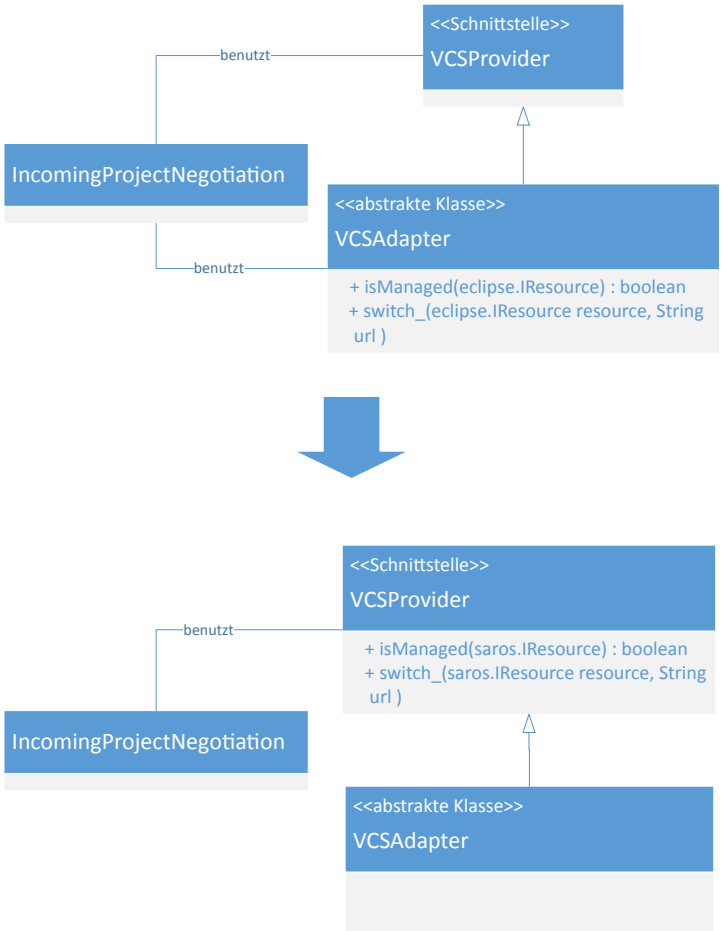

Abbildung 8: Die Refaktorisierung „move Method“ macht die direkte Benutzung vom VCS Adapter überflüssig

Es blieb ein Problem. Der VCSAdapter ist selbst nur eine abstrakte Klasse und wird vom SubclipseAdapter implementiert. Dieser braucht aber zwingend Eclipse-Dateien, während die Negotiation Saros-Dateien nutzt. Deswegen muss der VCSAdapter alle Ressourcen konvertieren (siehe Listing 5). Die Umsetzung erfolgte in Gemeinschaftsarbeit mit Stefan Rossbach.

5.2.3 Herunterziehen vom ISarosSessionManager

Es ist ein legitimes Verfahren, eine temporäre Verschlechterung des Designs, als Umweg für das Erreichen eines wichtigeren Ziels, in Kauf zu nehmen [uSR06]. Dies hab ich beim ISarosSessionManager-Interface so durchgeführt.

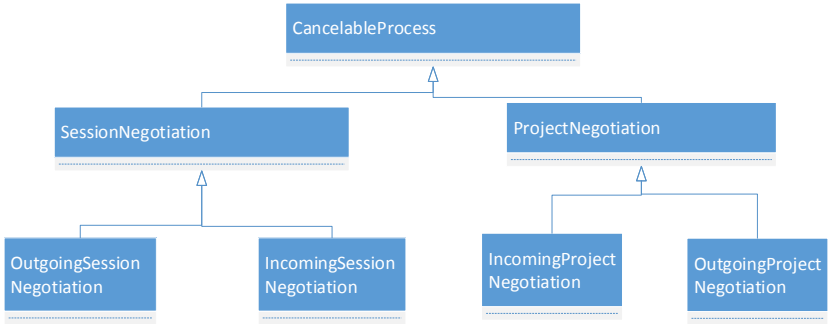

Abbildung 9: Die Hierachie von den Verhandlungsklassen

In der Abbildung 9 erkennen wir, dass während des Einladungsprozesses sowohl der Sitzungsaufbau, als auch der Projektaustausch von CancelableProcess erbt. Dies ist auch sinnvoll, da beide Aktionen jederzeit abgebrochen werden können. Des weiteren ist ersichtlich, dass es für den eingehenden Prozess als auch für den jeweiligen ausgehenden Prozess eine Oberklasse gibt, die Gemeinsamkeiten sammelt und beiden Ausprägungen zur Verfügung stellt. ProjectNegotiation hat dabei eine Abhängigkeit von ISarosSessionManager und ISarosSessionManager hatte eine Abhängigkeit zu Eclipse, welche zunächst nicht aufgelöst werden konnte[26]. Um das Problem zu lösen, habe ich das Verfahren „Push Down Field“ [Fow99, S. 266] verwendet[27]. Dabei wird das Feld vom Typ ISarosSessionManager in die Kindklassen verschoben (siehe Abbildung 10).
Aufgrund von weiteren Abhängigkeiten von Incoming/OutgoingProjectNegotiation war dies ein probater Weg einen Schritt vorwärts zu kommen. Durch diesen Schritt war es möglich, Beobachterklassen zu verschieben[28]. Es ist einfacher zu erkennen, welche Abhängigkeiten noch zu entfernen sind, wenn sich die Anzahl frühzeitig deutlich reduzieren lässt. Die entgegengesetzte Refaktorisierung „Pull Up Field“ um den ISarosSessionManager wieder in die Klasse ProjectNegotiation zu verschieben, wurde von mir umgesetzt, als der ISarosSessionManager im Kern war[29].

[26] Dies erfolgte im Patch 1950 (http://saros-build.imp.fu-berlin.de/gerrit/#/c/1950/)
[27] http://saros-build.imp.fu-berlin.de/gerrit/#/c/1633/
[28] http://saros-build.imp.fu-berlin.de/gerrit/#/c/1651/
[29] http://saros-build.imp.fu-berlin.de/gerrit/#/c/2152

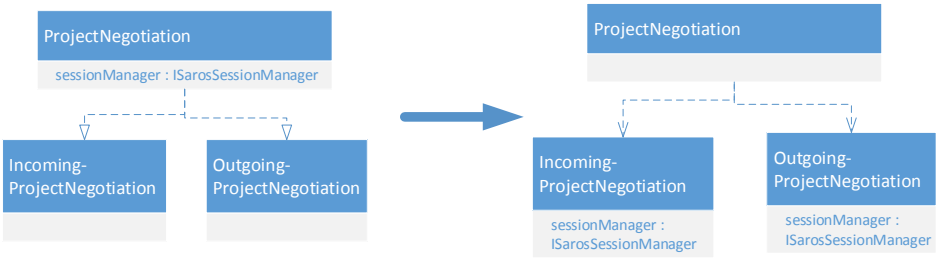

Abbildung 10: Die Abhängigkeit wird verschoben

5.2.4 Exception Proxy

Nachdem der Client eine Sitzung akzeptiert hat, wird eine Dateiliste erstellt, in der die fehlenden Dateien ermittelt werden. Diese wird dann an den Host gesendet, der die Dateien in ein Archiv verpackt und dann zum Client sendet. Dabei können Fehler entstehen, zum Beispiel, weil die Dateien nicht mehr existieren oder der Dateipfad nicht korrekt ist. Falls dies der Fall ist, kann das Programm nicht mehr normal weiterlaufen, weil die Ressource nicht vorhanden ist und so nicht für Operationen bereitsteht. In solchen Fällen werden Exceptions (Ausnahmen) erforderlich, damit das Programm sich nicht unerwartet verhält.
Solche Exceptions sind also notwendig, um auf unerwartete Probleme zu reagieren. Das Erstellen von einem Archiv kann sehr lange dauern, je nachdem, auf welchem System die Operation durchgeführt wird. Der Benutzer hat dabei die Möglichkeit den Vorgang abzubrechen. Dies geschieht über eine „OperationCanceledException“(OCE). Ich hatte bereits beschrieben, dass Eclipse erwartet, das Dateimanipulationen über deren API getätigt werden, da sonst Eclipse diese Manipulationen nicht registriert. Dies erwartet Eclipse auch bei einer OCE [Fou01].

Das Problem ist, dass dieser Vorgang zum Einladungsprozess gehört, welcher in den Kern soll. Mit Hilfe des Entwurfsmuster „Proxy“ konnte das Problem gelöst werden. Es gibt keine Möglichkeit im Kern ein IDE-spezifisches Objekt zu erzeugen. Deswegen wurde eine Saros-OperationCanceledException geschrieben, welche stattdessen geworfen wird[30]. Da die Dateioperationen über Adapter ausgeführt werden, wird zur Laufzeit auch der Eclipse-Workspace verwendet (bei Saros/E). In dem Adapter vom Eclipse-Workspace, kann die Saros-OCE gefangen werden und stattdessen eine Eclipse-OCE geworfen werden[31] (siehe Abbildung 11).

[30] Exceptions werden geworfen, falls der Fehlerfall eintritt.
[31] http://saros-build.imp.fu-berlin.de/gerrit/#/c/2125/

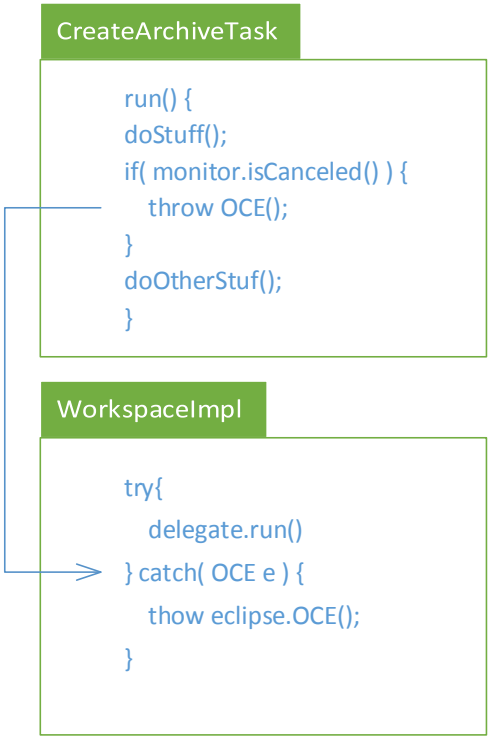

Abbildung 11: Die OperationCanceledException wird gefangen und mit der korrespondierenden Eclipse Exception ersetzt.

Durch meine Umsetzung konnte gewährleistet werden, dass Eclipse weiterhin über den Abbruch informiert wird.
Die Saros OCE ist allerdings nicht identisch mit der von Eclipse. Sie wurde als Checked Exception anstatt einer Runtime Exception entworfen. Der Unterschied liegt in der Handhabung. Während Java dem Programmierer zwingt, die Checked Exception zu fangen, kann der Entwickler bei der Runtime Exception dies auch unterlassen. Runtime Exception treten in der Regel zur Laufzeit auf, wenn eine Variable einen unerwarteten Wert für eine Operation hat oder nicht initialisiert wurde. In unserem Fall ist der Abbruch eine Aktion, die vom Nutzer ausgeführt wird und somit ein erwartbares Ereignis darstellt. Deswegen ist diese Umsetzung als CheckedException sinnvoll und abweichend von Eclipse.
Analog sind meine Ausführungen bei DecompressArchiveTask. Auf der Clientseite entpackt DecompressArchiveTask das Archiv, welches vom Host mit CreateArchiveTask erstellt wurde.

5.3 Designbruch entfernen

Einige Problem existieren nur, weil es zu Designbrüchen gekommen ist. Diese sind sehr individuell und daher lässt sich keine pauschale Antwort für das Beseitigen des Problems geben. Ich möchte trotzdem diese Refaktorisierungen besprechen, da die Werkzeuge, die uns die Autoren Fowler [Fow99] und Gamma [EG94] geben, trotzdem angewendet werden können.

Zyklische Abhängigkeit zwischen IncomingProjectNegotiation und AddProjectToSessionWizard entfernen

Beim Hinzufügen eines neuen Projektes in eine Sitzung, wird der Client dem Benutzer einen Dialog öffnen. Der Dialog dient dem Teilnehmer einen Speicherort und einen Namen für ein ankommendes Projekt zu geben. Dies passiert beim Sitzungsstart, kann aber auch während einer Sitzung auftreten, falls der Host ein neues Projekt hinzufügt[32].
Dieser Dialog wird vom AddProjectToSessionWizard erstellt. Der Wizard benötigt die IncomingProjectNegotiation, um die notwendigen Informationen für den Dialog zu erhalten.
Es gab aber eine Designverletzung. Der Wizard hat sich bei der Negotiation registriert, um im Falle eines Abbruchs des Dialoges durch den Host informiert zu werden. Dies sorgte für eine zyklische Abhängigkeit und zu einer „unangebrachter Intimität“[Fow99, S.70].
Die IncomingProjectNegotiation ist ein Teil vom Einladungsmanagements, während der AddProjectToSessionWizard ein Teil der Benutzeroberfläche ist [Sch13, S.79]. Die Abhängigkeit von IncomingProjectNegotiation zu dem Wizard ist nicht gewünscht, da die Negotiation in den Kern verschoben werden soll.

Listing 6: anonyme Implementierung vom CancelListener

```
private final CancelListener cancelListener = new
   CancelListener() {

        @Override
        public void canceled(final ProcessTools.
           CancelLocation location,
            final String message) {
            cancelWizard(peer, message, location);
        }
    };
```

Der Wizard hat sich noch im Konstruktor bei der Negotiation registriert, um über einen Abbruch informiert zu werden. Damit der Wizard weiterhin von der Negotiation benachrichtigt wird, aber den statischen Aufruf verliert,

[32] Ute Neise arbeitet daran, dass der Client ebenfalls Projekte hinzufügen kann. Dies dient als Vorbereitung für den Saros Server.

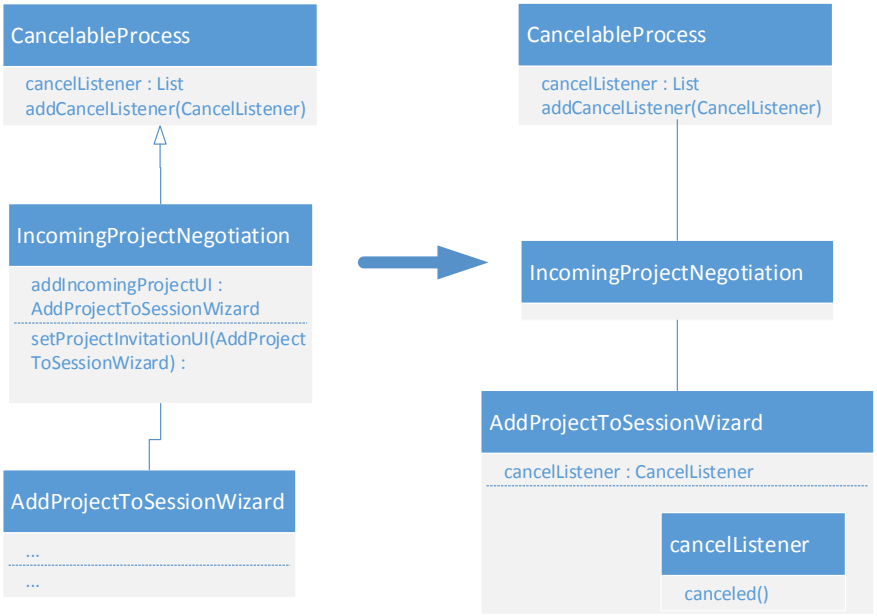

Abbildung 12: Durch den CancelListener ist die statische Abhängigkeit entfernt worden

wurde ein Listener (Zuhörer) eingesetzt. Dies ist möglich, da die Negotiation von CancelableProcess erbt (siehe Abbildung 9 auf Seite 27). Der CancelableProcess bietet die Möglichkeit, sich bei ihm als Zuhörer zu registrieren. Sollte ein Abbruch stattfinden, wird der Zuhörer aufgerufen. Der CancelListener ist ein Interface, der bei CancelableProcess als Zuhörer registriert werden kann. Da der Wizard einen speziellen CancelListener braucht, habe ich eine anonyme Implementierung vorgenommen. In Java kann die Implementation eines Objektes auch anonym (ohne Namen) in einem anderen Objekt erfolgen. Die anonyme Implementation kann man in Listing 6 sehen.

Ich konnte mit den CancelListener die statische Abhängigkeit komplett aus der IncomingProjectNegotiation entfernen. Die Veränderung kann man in der Abbildung 12 sehen. Das Problem der Abhängigkeit konnte durch das Beseitigen von Designverletzungen (technischen Schulden) gelöst werden.

6 Resumee

Zum Abschluss der Arbeit möchte ich meine Gedanken zu dem Erreichten und dem Unerreichten äußern. Außerdem möchte ich Probleme besprechen und Lösungsvorschläge präsentieren, damit die Fehler nicht noch einmal gemacht werden.

6.1 Angewendete Muster

Es gibt einen großen Katalog für Refaktorisierungen oder für Entwurfsmuster, die beschrieben wurden. Die Hauptaufgabe, die ich mir gesetzt hatte, sorgte aber dafür, dass ich nur wenige davon effektiv anwenden konnte. Diese Muster fanden sich immer wieder. Dies ist gut, da ich gehofft hatte solche zu finden. Es war aber ratsam nicht stur diese anzuwenden, da eine ausführliche Analyse der Gegebenheiten eine geeignetere Lösung zu Tage bringen kann. Es mussten manchmal keine Refaktorisierungen vorgenommen werden, weil der Quellcode die Abhängigkeit eigentlich gar nicht benötigte. Der Grund ließ sich oft nicht ermitteln, jedenfalls nicht in einem angemessenen Zeitraum. Vermutlich lag es an erodierten Strukturen oder unvollständigen Implementieren/Reparaturen, zumindest war das die Vermutung von anderen Entwicklern und mir.

6.2 Die Erweiterung des Saros Kerns

Es konnten sehr viele Klassen von den Eclipse-Abhängigkeiten befreit werden und in den Kern verschoben werden. Dadurch konnten einige Klassen von Saros/I entfernt werden, da diese Duplikate waren.
Diese Arbeit wurde keineswegs von mir alleine durchgeführt. Große Unterstützung hatte ich von Stefan Rossbach, der sehr viele von meinen Vorschlägen beurteilt und einige überarbeitet hat. Außerdem habe ich eng mit Holger Schmeisky an den IntelliJ-Duplikaten gearbeitet sowie Analysen mit ihm und Franz Zieris bezüglich des Saros-Kerns durchgeführt.
Während meiner Arbeit habe ich festgestellt, dass die verwendeten Entwurfsmuster und die Muster für die Refaktorisierungen sehr hilfreich waren. Sie erlaubten ein Problem einzuschätzen und zu lösen. In den meisten Fällen ging es um Abhängigkeiten zur API von Eclipse. Diese konnten nur in sehr wenigen Fällen ersetzt werden. In den meisten Fällen konnten die Abhängigkeiten durch Adapter gelöst werden. In einigen Fällen konnte ein Stellvertreter den Platz einnehmen, um dann mit dem IDE-Objekt ersetzt zu werden.
Meine Ziele für diese Arbeit waren, die elementarsten Bestandteile im Kern für andere IDEs verfügbar zu machen. Dies gelang auch für das Dateisystem und den Einladungsprozess.
Die Entwicklung war teilweise sehr schleppend und wurde von einer Reihe

von Fehlern begleitet, welches den Umfang der Arbeit reduzierte. Zusätzlich habe ich auch Patches vorgenommen, die nicht zu meiner Arbeit gehören, wie ein Patch für das kommende Release von Saros[33] oder eine Vielzahl von Reviews von anderen Patches.

6.3 Fehlschläge

6.3.1 Abhängigkeitsanalysen

Die Analysen über die Abhängigkeiten stellte sich teilweise sehr zeitaufwendig dar, da sich der Graph änderte und die IntelliJ und die Eclipse Quellcodeversionen auf einen unterschiedlichen Stand waren. Nicht immer waren die Klassenbezeichnungen identisch gewählt. Es kam durchaus vor, dass Eclipse Implementierungen von Klassen das Präfix „Eclipse“ trugen, während die IntelliJ-Klassen keinen derartigen Präfix hatten. Als Beispiel dazu soll „IPath“ dienen.
Desweiteren wurden einige Patches sehr schnell mit dem Hauptentwicklungszweig zusammengefügt. Ich hatte diese teilweise übersehen gehabt.
Es war leichtgläubig anzunehmen, dass Analyse von den IntelliJ Duplikaten einfacher wäre. Der Quellcode war schlicht nicht aktuell. Man hätte zuerst kontrollieren müssen, welche Duplikate nicht mehr gebraucht werden und stets überprüfen, dass dies noch aktuell ist.

6.3.2 Entscheidungsprozess

Einige Probleme gab es bei der Kommunikation und Absprache der Entwicklung. Als Beispiel soll mein Versuch der Überarbeitung von „IPreferences“ genommen werden. Diese Schnittstelle sollte den Aufruf des Einstellungsspeicher der jeweiligen IDE abstrahieren. Die einzelnen Werte aus den Einstellungen sollten nicht mehr über Zeichenketten abgerufen und gespeichert werden, sondern über Methoden, die entsprechend benannt wurden. So sollte ein Aufruf für die Adresse des Server nicht „store.getBoolean(„server_name“)“, sondern „getServerName()“ sein. Es musste nur noch an einer Stelle der Abruf über einen String stattfinden, anstatt bei jeden Aufruf mit einem gewissen Fehlerpotential bei unerfahrenen Programmierern. Der potenzielle Nachteil, dass Methoden für ungenutzte Einstellungen in der jeweiligen IDE implementiert werden müssen, war für einige Entwickler so enorm, dass dieses Interface durch ein anderes ausgetauscht werden musste. Der neu implementierte Entwurf war eigentlich mein ursprünglicher, den ich verworfen hatte.
Bei solchen Designentscheidungen sehe ich das Problem der räumlichen Distanz der Entwickler gegeben. Diskussionen auf der Saros Mailingliste sind

[33] http://saros-build.imp.fu-berlin.de/gerrit/#/c/2053/

sehr schwer zu führen und ich habe nicht das Gefühl, dass sich der bessere Vorschlag durchsetzt, sondern der, der am vehementen vertreten wird.
Glücklicherweise waren die Diskussionsgruppen sehr klein, weshalb man nach der initialen Nachricht, den Diskussionspartner direkt anschreiben und mit ihm diskutieren konnte.
Welches die beste Lösung für das Problem mit den Einstellungen ist, lässt sich momentan nicht mit Gewissheit sagen, da niemand in die Zukunft schauen kann, um zu sehen, wie sich Saros entwickelt.

6.4 Theorie und Praxis

Die Theorie und Praxis trennen bekanntlicherweise oft Welten, da die Theorie einen Idealzustand beschreibt, der in der Praxis nicht vorkommt. Man muss sich damit arrangieren und auf die anzutreffenden Zustände vorbereiten.

6.4.1 Respekt vor dem Code

Zu Beginn meiner Ausarbeitungen hatte ich sehr viel Respekt vor den Quellcode. Ich habe versucht, meine Refaktorisierungen so korrekt wie möglich zu machen. Es sollte keinerlei Veränderung im Verhalten sichtbar sein und keine Veränderungen im Quellcode stattfinden, die ich nicht genau vorher analysiert hatte. Ich war stets der Meinung, „der Programmierer, der das geschrieben hat, muss sich dabei etwas gedacht haben“. Dabei stellte sich heraus: nicht unbedingt. Dies scheint ein wenig komisch zu sein, lässt sich aber anhand der hohen Fluktuation der Saros-Entwickler erklären. Viele arbeiten während ihrer Abschlussarbeit an Saros mit und verlassen das Projekt, nachdem sie eingearbeitet sind. Die Qualitätssicherungsmaßnahmen existierten nicht die ganze Zeit und wurden nach und nach hinzugefügt. Dabei ist es durchaus möglich, dass Entwickler durch Unwissenheit riechenden Quellcode eingebaut haben.

6.4.2 Qualitätssicherungsmaßnahmen

Während meiner Ausarbeitungen hat Arsenij Solovjev für seine Masterarbeit Jenkins mit Sonar verbunden, sodass die Analyseergebnisse direkt in Gerrit als Kommentar hinzugefügt wurden [Sol14]. Ich empfand dies als sehr hilfreich, dass sie einen selbst als Reviewer entlastete andere Patches auf sehr simple Fehler zu untersuchen und man selbst diesen Hinweisen nachgehen konnte, noch ehe ein anderer Entwickler Feedback hinterlassen hat. Sie beschleunigte die Entwicklung leicht. Dies förderte auch ein Problem zu Tage, dass einige Bemerkungen, die von Sonar hinzugefügt wurden, fehlerhaft waren. Es wurden Besonderheiten von Saros nicht erkannt und daher wurde dem Entwickler etwas Falsches nahegelegt. Besonders Entwickler, die

noch nicht mit Saros vertraut waren, konnten nicht ohne weiteres die Sinnhaftigkeit dieser Aussagen bewerten. Es machte für mich deutlich, dass die beste Sicherungsmaßnahme die Code-Review selbst war, obwohl die Qualität sehr schwankend war. Je nachdem wie viele und welche Entwickler beteiligt waren.

6.4.3 You had one Job

Refactorisierungen sollten entkoppelt von anderen Anpassungen der Software stattfinden, da man nur so überprüfen kann, ob die auftretenden Fehler durch die Refaktorisierung oder der neuen/modifizierten Code entstanden sind. Während der Beurteilungsphase der eingereichten Patches wurde öfters von einigen Entwicklern verlangt, andere Anpassungen, als die durchgeführte Refaktorisierung durchzuführen. Die angeforderten Änderungen reichten von Javadoc hinzufügen über andere Refaktorisierungen bis zum Entfernen von bestimmten Methoden. Bei kleineren Änderungen (Hinzufügen von Dokumentation für Methoden) ist es verständlich, dass dafür kein weiterer Patch geschrieben werden muss. Es wird erst ein Problem, wenn zusätzlich die API geändert wird und so zahlreiche Klassen angepasst werden müssen, die sonst unberührt geblieben wären. Außerdem erhöht es unnötig den Aufwand für den einzelnen Patch. Es könnten durch diese Änderungen ebenfalls Defekte entstehen, die im besten Fall den Entwicklungsprozess nur verlangsamen.
Ich denke, dass es normal ist, dass die Reviewer Probleme bei der Review entdecken und den Entwickler um Behebung von diesen bittet. Es hat für mich ein bisschen gedauert, bis ich den richtigen Umgang mit diesen Anforderungen gefunden hatte. Wenn die Anforderungen mit den eigentlichen Patch nichts mehr zu tun haben und es potenzielle Fehlerquellen beinhaltet, sollte der Verweis auf ein neues Patchset erfolgen. Es ist nicht zwingend Notwendig, diesen Patch komplett zu bearbeiten, da die anderen Entwickler ihn auch übernehmen können.

6.5 Fazit und Ausblick

In meiner Arbeit habe ich den Entwicklungsprozess von Saros und im speziellen Saros/I begleitet. Dabei wurden sehr viele Veränderungen durchgeführt, die den Kern um eine Vielzahl von Klassen anwachsen lies. Ich wollte mit meiner Arbeit zeigen, wie man die Probleme bei der Umstrukturierung der Architektur beheben kann, ohne Defekte oder „Gerüche“ in die Software einzubauen. Dazu habe ich erklärt, was Refaktorisierungen sind und weshalb sie ein probates Mittel sind die Struktur zu verbessern. Desweiteren habe ich einige Design Pattern benutzt und erklärt, um die Refaktorisierungen sinnvoll anwenden zu können. Es sind Techniken, die gut ineinander greifen und somit mächtige Werkzeuge darstellen. Unerlässlich war es dabei, den Entwicklungsprozess zu skizzieren, da diesen die Qualitätssicherung stattfin-

det und dadurch wertvolle Hinweise für eine gute Umsetzung bereithält. Es musste sichergestellt werden, das meine Veränderungen keine (oder besser gesagt: möglichst wenige) Defekte verursacht, da mein primäres Interesse an einer Verbesserung der internen Struktur lag, anstatt einer neuen Funktion. Die Probleme, denen ich mich stellen musste, waren im überwiegenden Teil aufzulösende Abhängigkeiten und Behebung von „riechenden“ Code. Das erstere Problem lies sich gut mit den beschriebenen Mustern behandeln. Da die Anforderungen immer ähnlich waren, sind es nicht viele Muster, die zum Einsatz kamen. Das zweite Problem war stets spezifisch für die jeweilige Klasse und es lies sich kein wiederkehrendes Muster erkennen.
Ich bin mir sicher, dass die Entwicklung an Saros/I weitergehen wird und das es weitere Refaktorisierungen für den Kern geben wird. Es sind noch nicht alle essentiellen Bestandteile von Saros im Kern. Von den Entwicklern wurde die ganze Zeit eiserne Disziplin gefordert, um Klassen in den Kern verschieben zu dürfen. Es wurde darauf geachtet, dass die größten Gerüche vorher entfernt werden, damit der Kern weniger Altlasten hat. Dies ist ein durchaus vernünftiger Schritt, welcher unbedingt konsequent durchgehalten werden muss.

A Abhängigkeitsgraphen

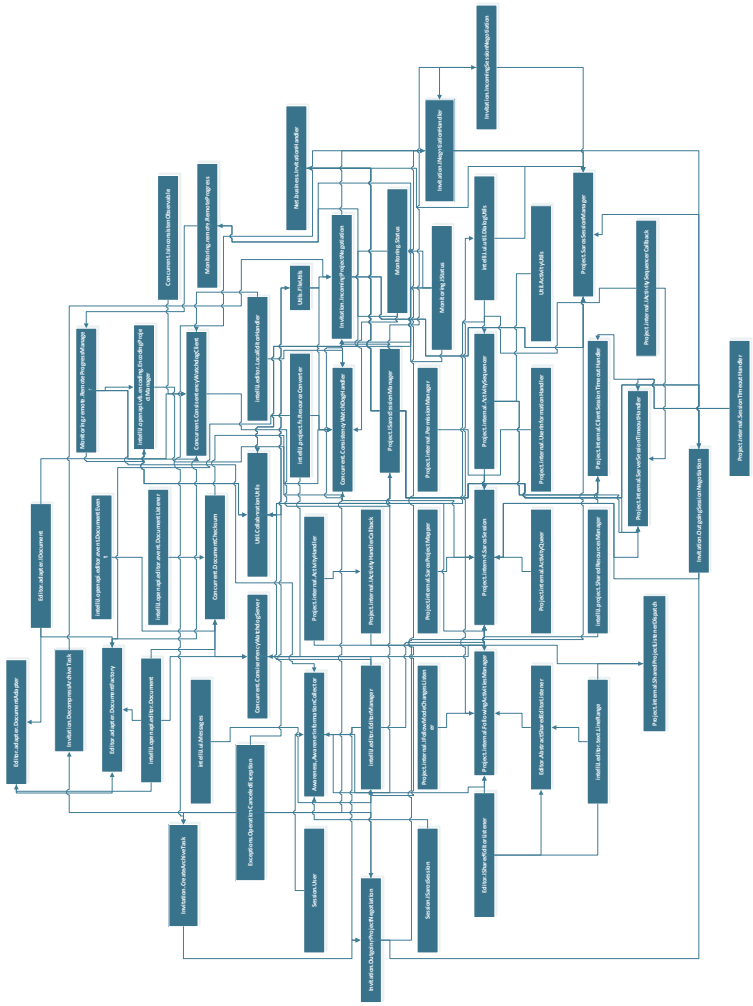

Abbildung 13: Abhängigkeiten vom Einladungsprozess zu Beginn meiner Analysen.

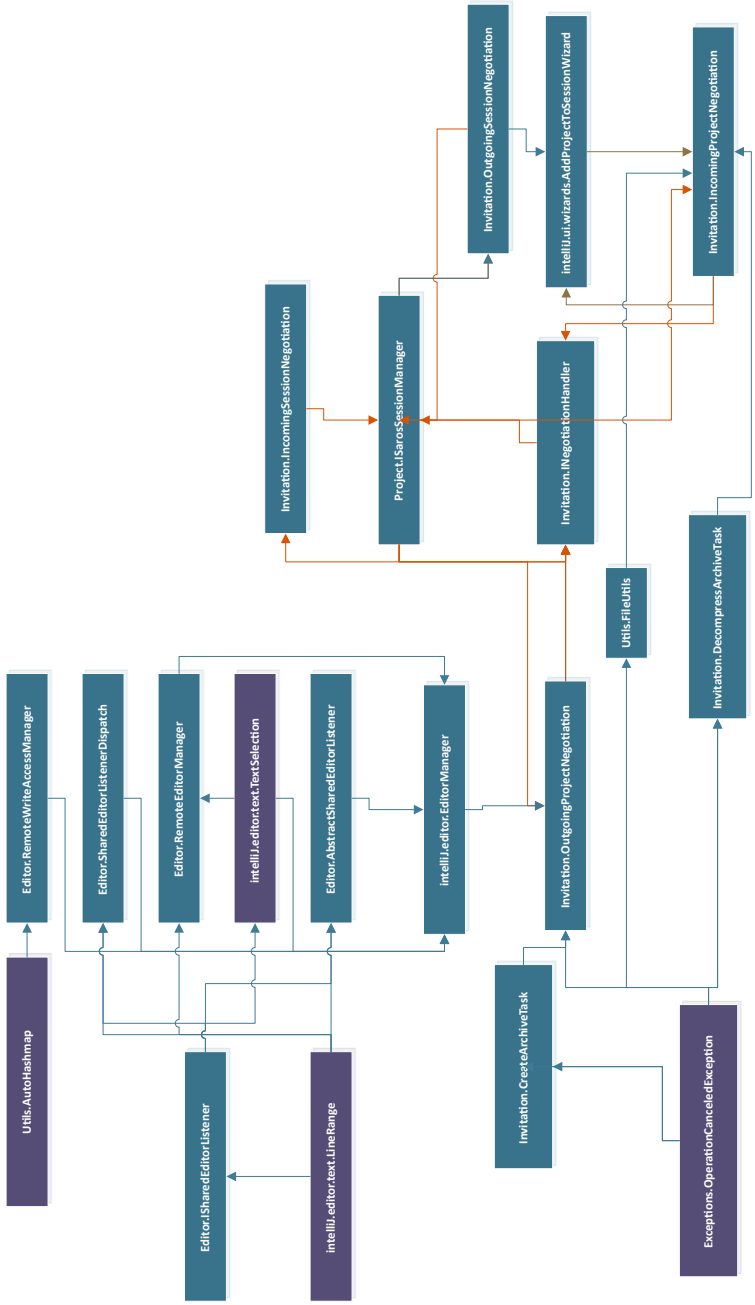

Abbildung 14: Abhängigkeiten zu einem späteren Zeitpunkt

Literatur

[BF00] Kent Beck and Martin Fowler. *Planning Extreme Programming*. Addision-Wesley, 2000.

[Boh15] Matthias Bohnstedt. Cross-Plattform-GUI Entwicklung in Saros. https://www.mi.fu-berlin.de/w/SE/CrossPlattformGUIEntwicklungInSaros, 2015. Eingesehen am 22.04.2015.

[Bus14] Nils Hauke Bussas. Entwicklung eines Server-Prototypen fur Saros. Bachelor's thesis, Freie Universität Berlin, 2014.

[Cik15] Christian Cikryt. Evaluating the Use of a Web Browser to Unify GUI Development for IDE Plug-ins. https://www.mi.fu-berlin.de/w/SE/ThesisTestingSaros, 2015. Eingesehen am 22.04.2015.

[EG94] Ralph E. Johnson und John Vlissides Erich Gamma, Richard Helm. *Design Patterns. Elements of Reusable Object-Oriented Software*. Prentice Hall, 1994.

[Fou01] Eclipse Foundation. Interface IWorkspaceRunnable. http://help.eclipse.org/indigo/topic/org.eclipse.platform.doc.isv/reference/api/org/eclipse/core/resources/IWorkspaceRunnable.html, 2001. Eingesehen am 23.04.2015.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[Fow03] Matrin Fowler. TechnicalDebt. http://martinfowler.com/bliki/TechnicalDebt.html, 2003. Eingesehen am 22.04.2015.

[Fow04] Martin Fowler. Inversion of control containers and the dependency injection pattern. http://martinfowler.com/articles/injection.html, 2004. Eingesehen am 22.04.2015.

[Fow14] Matrin Fowler. Workflows of refactoring. http://martinfowler.com/articles/workflowsOfRefactoring/#2hats, 2014. Eingesehen am 22.04.2015.

[Han13] Tam Hanna. Eclipse vs. intellij idea: Religionskrieg im mobilbereich. http://www.heise.de/developer/artikel/Eclipse-vs-IntelliJ-IDEA-Religionskrieg-im-Mobilbereich-1893303.html, 2013. Eingesehen am 22.04.2015.

[Kü09] Sebastian Kübeck. *Softwaresanierung*. mitp-Verlag, 2009.

[Ker04] Joshua Kerievsky. *Refactoring to Patterns*. Pearson Higher Education, 2004.

[Nei15] Ute Neise. Weiterentwicklung des Saros-Servers. https://www.mi.fu-berlin.de/w/SE/ThesisWeiterentwicklungSarosServer, 2015. Eingesehen am 22.04.2015.

[Neu11] Alexander Neumann. 10 Jahre Eclipse: Konsolidierung des Java-IDE-Markts. http://www.heise.de/developer/meldung/10-Jahre-Eclipse-Konsolidierung-des-Java-IDE-Markts-1370644.htm, 2011. Eingesehen am 22.04.2015.

[Ros11] Stefan Rossbach. Einführung einer kontinuierlichen Integrationsumgebung und Verbesserung des Test-Frameworks. Bachelor's thesis, Freie Universität Berlin, 2011.

[Sch13] Patrick Schlott. Analyse und Verbesserung der Architektur eines nebenläufigen und verteilten Softwaresystems. Master's thesis, Freie Universität Berlin, 2013.

[Sie15] Bastian Sieker. User-Centered Development of a JavaScript and HTML-based GUI for Saros. Ankündigung auf der Maillingliste am 21.04.2015, 2015.

[Sol14] Arsenij Solovjev. Operationalizing the Architecture of an Agile Software Project. Master's thesis, Freie Universität Berlin, 2014.

[Sur14] Girish Suryanarayana. *Refactoring for Software Design Smells: Managing Technical Debt*. Morgan Kaufmann, 2014.

[uSR06] Matrin Lippert und Stefan Roock. *Refactoring in Large Software Projects*. Wiley Computer Publishing, 2006.